Refining Authenticated Key Agreement with Strong Adversaries

Joseph Lallemand, Christoph Sprenger, and David Basin

March 17, 2025

Contents

1 Proving infrastructure **3**
1.1 Prover configuration . . . 3
1.2 Forward reasoning ("attributes") . . . 3
1.3 General results . . . 3
1.3.1 Maps . . . 3
1.3.2 Set . . . 3
1.3.3 Relations . . . 4
1.3.4 Lists . . . 4
1.3.5 Finite sets . . . 4

2 Models, Invariants and Refinements **5**
2.1 Specifications, reachability, and behaviours. . . . 5
2.1.1 Finite behaviours . . . 5
2.1.2 Specifications, observability, and implementation . . . 6
2.2 Invariants . . . 9
2.2.1 Hoare triples . . . 9
2.2.2 Characterization of reachability . . . 10
2.2.3 Invariant proof rules . . . 10
2.3 Refinement . . . 11
2.3.1 Relational Hoare tuples . . . 11
2.3.2 Refinement proof obligations . . . 13
2.3.3 Deriving invariants from refinements . . . 14
2.3.4 Refinement of specifications . . . 15

3 Message definitions **18**
3.1 Messages . . . 18

4 Message theory **27**
4.1 Message composition . . . 27
4.2 Message decomposition . . . 29
4.3 Lemmas about combined composition/decomposition . . . 31
4.4 Accessible message parts . . . 32
4.4.1 Lemmas about combinations with composition and decomposition . . . 35
4.5 More lemmas about combinations of closures . . . 35

5 Environment: Dolev-Yao Intruder **38**

6 Secrecy Model (L0) **39**
6.1 State and events . . . 39
6.2 Proof of secrecy invariant . . . 40

7 Non-injective Agreement (L0) **42**
7.1 Signals . . . 42
7.2 State and events . . . 42
7.3 Non injective agreement invariant . . . 43

8 Injective Agreement (L0) **45**
8.1 State and events . . . 45
8.2 Injective agreement invariant . . . 46
8.3 Refinement . . . 46
8.4 Derived invariant . . . 47

9 Runs **48**
9.1 Type definitions . . . 48

10 Channel Messages **49**
10.1 Channel messages . . . 49
10.2 Extract . . . 49
10.3 Fake . . . 50
10.4 Closure of Dolev-Yao, extract and fake . . . 52
10.4.1 *dy-fake-msg*: returns messages, closure of DY and extr is sufficient . . 52
10.4.2 *dy-fake-chan*: returns channel messages . . . 53

11 Payloads and Support for Channel Message Implementations **55**
11.1 Payload messages . . . 55
11.2 *isLtKey*: is a long term key . . . 58
11.3 *keys-of*: the long term keys of an agent . . . 58
11.4 *Keys-bad*: bounds on the attacker's knowledge of long-term keys. . . . 59
11.5 *broken K*: pairs of agents where at least one is compromised. . . . 61
11.6 *Enc-keys-clean S*: messages with "clean" symmetric encryptions. . . . 61
11.7 Sets of messages with particular constructors . . . 62
11.7.1 Lemmas for moving message sets out of *analz* . . . 63

12 Assumptions for Channel Message Implementation **65**
12.1 First step: basic implementation locale . . . 65
12.2 Second step: basic and analyze assumptions . . . 66
12.3 Third step: *valid-implem* . . . 68

13 Lemmas Following from Channel Message Implementation Assumptions **69**
13.1 Message implementations and abstractions . . . 69
13.2 Extractable messages . . . 70
13.2.1 Partition I to keep only the extractable messages . . . 71
13.2.2 Partition of *extractable* . . . 71

13.3 Lemmas for proving intruder refinement (L2-L3) 71
13.3.1 First: we only keep the extractable messages 72
13.3.2 Only keep the extracted messages (instead of extractable) 72
13.3.3 Keys and Tags can be moved out of the *analz* 72
13.3.4 Final lemmas, using all the previous ones 72
13.3.5 Partitioning *analz ik* 73

14 Symmetric Implementation of Channel Messages 74
14.1 Implementation of channel messages 74
14.2 Lemmas to pull implementation sets out of *analz* 75
14.2.1 Pull *implInsecSet* out of *analz* 75
14.3 Pull *implConfidSet* out of *analz* 75
14.4 Pull *implSecureSet* out of *analz* 76
14.5 Pull *implAuthSet* out of *analz* 77
14.6 Locale interpretations 78

15 Asymmetric Implementation of Channel Messages 79
15.1 Implementation of channel messages 79
15.2 Lemmas to pull implementation sets out of *analz* 80
15.2.1 Pull *PairAgentSet* out of *analz* 80
15.2.2 Pull *implInsecSet* out of *analz* 80
15.3 Pull *implConfidSet* out of *analz* 80
15.4 Pull *implAuthSet* out of *analz* 81
15.5 Pull *implSecureSet* out of *analz* 81
15.6 Locale interpretations 82

16 Key Transport Protocol with PFS (L1) 84
16.1 State and Events 84
16.2 Refinement: secrecy 88
16.3 Derived invariants: secrecy 90
16.4 Invariants 90
16.4.1 inv1 90
16.4.2 inv2 90
16.4.3 inv3 (derived) 91
16.5 Refinement: injective agreement 91
16.6 Derived invariants: injective agreement 93

17 Key Transport Protocol with PFS (L2) 94
17.1 State and Events 94
17.2 Invariants 99
17.2.1 inv1 99
17.2.2 inv2 (authentication guard) 99
17.2.3 inv3 (authentication guard) 100
17.2.4 inv4 101
17.2.5 inv5 101
17.2.6 inv6 102
17.2.7 inv7 102

17.2.8 inv8 . . . 104
17.3 Refinement . . . 104
17.4 Derived invariants . . . 106

18 Key Transport Protocol with PFS (L3 locale) 108
18.1 State and Events . . . 108
18.2 Invariants . . . 112
18.2.1 inv1: No long-term keys as message parts . . . 112
18.2.2 inv2: *l3-state.bad s* indeed contains "bad" keys . . . 112
18.2.3 inv3 . . . 113
18.2.4 inv4: the intruder knows the tags . . . 113
18.2.5 inv5 . . . 114
18.2.6 inv6 . . . 114
18.2.7 inv7 . . . 115
18.2.8 inv8 . . . 115
18.2.9 inv9 . . . 116
18.3 Refinement . . . 116
18.3.1 Protocol events . . . 117
18.3.2 Intruder events . . . 118
18.3.3 Compromise events . . . 118
18.4 Derived invariants . . . 119
18.4.1 inv10: secrets contain no implementation material . . . 119
18.4.2 Partial secrecy . . . 120
18.4.3 Secrecy . . . 120
18.4.4 Injective agreement . . . 120

19 Key Transport Protocol with PFS (L3, asymmetric implementation) 122

20 Key Transport Protocol with PFS (L3, symmetric implementation) 123

21 Authenticated Diffie Hellman Protocol (L1) 124
21.1 State and Events . . . 124
21.2 Refinement: secrecy . . . 129
21.3 Derived invariants: secrecy . . . 130
21.4 Invariants: *Init* authenticates *Resp* . . . 131
21.4.1 inv1 . . . 131
21.4.2 inv2 . . . 131
21.4.3 inv3 (derived) . . . 132
21.5 Invariants: *Resp* authenticates *Init* . . . 132
21.5.1 inv4 . . . 132
21.5.2 inv5 . . . 133
21.5.3 inv6 (derived) . . . 134
21.6 Refinement: injective agreement (*Init* authenticates *Resp*) . . . 134
21.7 Derived invariants: injective agreement (*Init* authenticates *Resp*) . . . 136
21.8 Refinement: injective agreement (*Resp* authenticates *Init*) . . . 136
21.9 Derived invariants: injective agreement (*Resp* authenticates *Init*) . . . 137

22 Authenticated Diffie-Hellman Protocol (L2) 139
22.1 State and Events . . . 139
22.2 Invariants . . . 145
22.2.1 inv1 . . . 145
22.2.2 inv2 (authentication guard) . . . 145
22.2.3 inv3 (authentication guard) . . . 146
22.2.4 inv4 . . . 146
22.2.5 inv4' . . . 147
22.2.6 inv5 . . . 147
22.2.7 inv6 . . . 148
22.2.8 inv7 . . . 148
22.2.9 inv8: form of the secrets . . . 150
22.3 Refinement . . . 151
22.4 Derived invariants . . . 153

23 Authenticated Diffie-Hellman Protocol (L3 locale) 155
23.1 State and Events . . . 155
23.2 Invariants . . . 159
23.2.1 inv1: No long-term keys as message parts . . . 159
23.2.2 inv2: *l3-state.bad s* indeed contains "bad" keys . . . 160
23.2.3 inv3 . . . 160
23.2.4 inv4: the intruder knows the tags . . . 161
23.2.5 inv5 . . . 162
23.2.6 inv6 . . . 162
23.2.7 inv7 . . . 163
23.2.8 inv8 . . . 163
23.2.9 inv9 . . . 164
23.3 Refinement . . . 164
23.3.1 Protocol events . . . 165
23.3.2 Intruder events . . . 166
23.3.3 Compromise events . . . 166
23.4 Derived invariants . . . 167
23.4.1 inv10: secrets contain no implementation material . . . 167
23.4.2 Partial secrecy . . . 168
23.4.3 Secrecy . . . 168
23.4.4 Injective agreement . . . 168

24 Authenticated Diffie-Hellman Protocol (L3, asymmetric) 170

25 Authenticated Diffie-Hellman Protocol (L3, symmetric) 171

26 SKEME Protocol (L1) 172
26.1 State and Events . . . 172
26.2 Refinement: secrecy . . . 177
26.3 Derived invariants: secrecy . . . 179
26.4 Invariants: *Init* authenticates *Resp* . . . 179
26.4.1 inv1 . . . 179

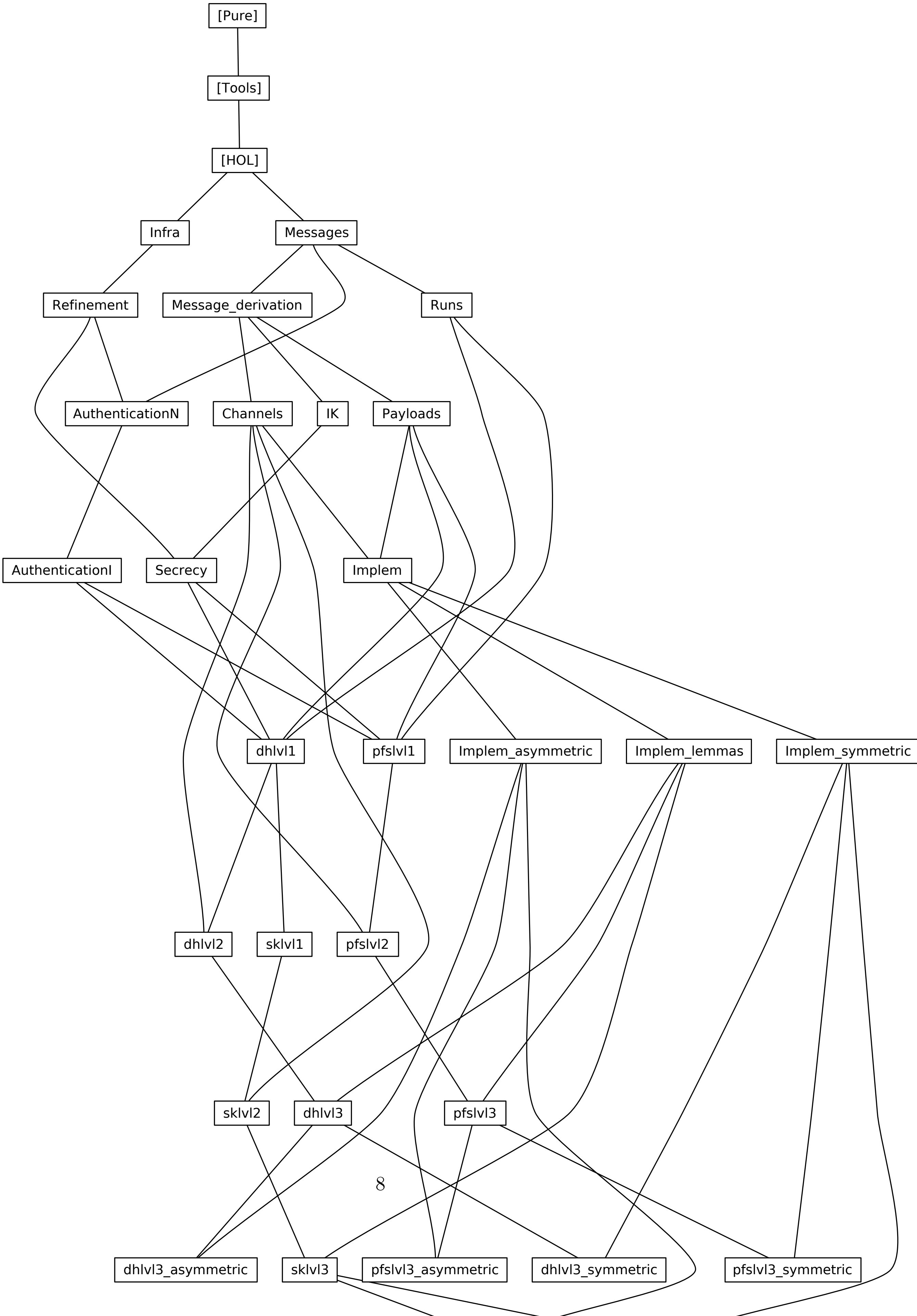
26.4.2 inv2 . . . 180
26.4.3 inv3 (derived) . . . 180
26.5 Invariants: Resp authenticates Init . . . 181
26.5.1 inv4 . . . 181
26.5.2 inv5 . . . 182
26.5.3 inv6 (derived) . . . 182
26.6 Refinement: injective agreement (Init authenticates Resp) . . . 183
26.7 Derived invariants: injective agreement (*Init* authenticates *Resp*) . . . 184
26.8 Refinement: injective agreement (*Resp* authenticates *Init*) . . . 184
26.9 Derived invariants: injective agreement (*Resp* authenticates *Init*) . . . 186

27 SKEME Protocol (L2) 187
27.1 State and Events . . . 187
27.2 Invariants . . . 194
27.2.1 inv1 . . . 194
27.2.2 inv2 . . . 194
27.2.3 inv3 . . . 195
27.2.4 hmac preservation lemmas . . . 197
27.2.5 inv4 (authentication guard) . . . 198
27.2.6 inv5 (authentication guard) . . . 199
27.2.7 inv6 . . . 200
27.2.8 inv6' . . . 201
27.2.9 inv7: form of the secrets . . . 201
27.3 Refinement . . . 202
27.4 Derived invariants . . . 204

28 SKEME Protocol (L3 locale) 206
28.1 State and Events . . . 206
28.2 Invariants . . . 211
28.2.1 inv1: No long-term keys as message parts . . . 211
28.2.2 inv2: *l3-state.bad s* indeed contains "bad" keys . . . 212
28.2.3 inv3 . . . 212
28.2.4 inv4: the intruder knows the tags . . . 213
28.2.5 inv5 . . . 213
28.2.6 inv6 . . . 214
28.2.7 inv7 . . . 214
28.2.8 inv8 . . . 215
28.2.9 inv9 . . . 216
28.3 Refinement . . . 216
28.3.1 Protocol events . . . 217
28.3.2 Intruder events . . . 218
28.3.3 Compromise events . . . 218
28.4 Derived invariants . . . 219
28.4.1 inv10: secrets contain no implementation material . . . 219
28.4.2 Partial secrecy . . . 220
28.4.3 Secrecy . . . 220
28.4.4 Injective agreement . . . 220

29 SKEME Protocol (L3 with asymmetric implementation) 222

30 SKEME Protocol (L3 with symmetric implementation) 223


[Pure]
[Tools]
[HOL]
Infra
Messages
Refinement
Message_derivation
Runs
AuthenticationN
Channels
IK
Payloads
AuthenticationI
Secrecy
Implem
dhlvl1
pfslvl1
Implem_asymmetric
Implem_lemmas
Implem_symmetric
dhlvl2
sklvl1
pfslvl2
sklvl2
dhlvl3
pfslvl3
dhlvl3_asymmetric
sklvl3
pfslvl3_asymmetric
dhlvl3_symmetric
pfslvl3_symmetric

1 Proving infrastructure

theory *Infra* **imports** *Main*
begin

1.1 Prover configuration

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

1.2 Forward reasoning ("attributes")

The following lemmas are used to produce intro/elim rules from set definitions and relation definitions.

lemmas *set-def-to-intro* = *meta-eq-to-obj-eq* [*THEN eqset-imp-iff*, *THEN iffD2*]
lemmas *set-def-to-dest* = *meta-eq-to-obj-eq* [*THEN eqset-imp-iff*, *THEN iffD1*]
lemmas *set-def-to-elim* = *set-def-to-dest* [*elim-format*]

lemmas *setc-def-to-intro* =
set-def-to-intro [**where** *B*={*x*. *P x*} **for** *P*, *to-pred*]

lemmas *setc-def-to-dest* =
set-def-to-dest [**where** *B*={*x*. *P x*} **for** *P*, *to-pred*]

lemmas *setc-def-to-elim* = *setc-def-to-dest* [*elim-format*]

lemmas *rel-def-to-intro* = *setc-def-to-intro* [**where** *x*=(*s*, *t*) **for** *s t*]
lemmas *rel-def-to-dest* = *setc-def-to-dest* [**where** *x*=(*s*, *t*) **for** *s t*]
lemmas *rel-def-to-elim* = *rel-def-to-dest* [*elim-format*]

1.3 General results

1.3.1 Maps

We usually remove *domIff* from the simpset and clasets due to annoying behavior. Sometimes the lemmas below are more well-behaved than *domIff*. Usually to be used as "dest: dom_lemmas". However, adding them as permanent dest rules slows down proofs too much, so we refrain from doing this.

lemma *map-definedness*:
$f\, x = Some\ y \Longrightarrow x \in dom\ f$
⟨*proof*⟩

lemma *map-definedness-contra*:
$[\![\ f\, x = Some\ y;\ z \notin dom\ f\ ]\!] \Longrightarrow x \neq z$
⟨*proof*⟩

lemmas *dom-lemmas* = *map-definedness map-definedness-contra*

1.3.2 Set

lemma *vimage-image-subset*: $A \subseteq f-`(f`A)$
⟨*proof*⟩

1.3.3 Relations

lemma *Image-compose* [*simp*]:
 (*R1 O R2*)‘‘*A* = *R2*‘‘(*R1*‘‘*A*)
⟨*proof*⟩

1.3.4 Lists

lemma *map-comp*: *map* (*g o f*) = *map g o map f*
⟨*proof*⟩

declare *map-comp-map* [*simp del*]

lemma *take-prefix*: ⟦ *take n l* = *xs* ⟧ ⟹ ∃ *xs'*. *l* = *xs* @ *xs'*
⟨*proof*⟩

1.3.5 Finite sets

Cardinality.

declare *arg-cong* [**where** *f*=*card*, *intro*]

lemma *finite-positive-cardI* [*intro*!]:
 ⟦ *A* ≠ {}; *finite A* ⟧ ⟹ *0* < *card A*
⟨*proof*⟩

lemma *finite-positive-cardD* [*dest*!]:
 ⟦ *0* < *card A*; *finite A* ⟧ ⟹ *A* ≠ {}
⟨*proof*⟩

lemma *finite-zero-cardI* [*intro*!]:
 ⟦ *A* = {}; *finite A* ⟧ ⟹ *card A* = *0*
⟨*proof*⟩

lemma *finite-zero-cardD* [*dest*!]:
 ⟦ *card A* = *0*; *finite A* ⟧ ⟹ *A* = {}
⟨*proof*⟩

end

2 Models, Invariants and Refinements

theory *Refinement* **imports** *Infra*
begin

2.1 Specifications, reachability, and behaviours.

Transition systems are multi-pointed graphs.

record $'s$ *TS* =
  init :: $'s$ *set*
  trans :: $('s \times 's)$ *set*

The inductive set of reachable states.

inductive-set
  reach :: $('s, 'a)$ *TS-scheme* $\Rightarrow$ $'s$ *set*
  for T :: $('s, 'a)$ *TS-scheme*
where
  r-init [*intro*]: $s \in init\ T \Longrightarrow s \in reach\ T$
| *r-trans* [*intro*]: $\llbracket (s, t) \in trans\ T;\ s \in reach\ T \rrbracket \Longrightarrow t \in reach\ T$

2.1.1 Finite behaviours

Note that behaviours grow at the head of the list, i.e., the initial state is at the end.

inductive-set
  beh :: $('s, 'a)$ *TS-scheme* $\Rightarrow$ $('s\ list)$ *set*
  for T :: $('s, 'a)$ *TS-scheme*
where
  b-empty [*iff*]: $[] \in beh\ T$
| *b-init* [*intro*]: $s \in init\ T \Longrightarrow [s] \in beh\ T$
| *b-trans* [*intro*]: $\llbracket s \# b \in beh\ T;\ (s, t) \in trans\ T \rrbracket \Longrightarrow t \# s \# b \in beh\ T$

inductive-cases *beh-non-empty*: $s \# b \in beh\ T$

Behaviours are prefix closed.

lemma *beh-immediate-prefix-closed*:
  $s \# b \in beh\ T \Longrightarrow b \in beh\ T$
⟨*proof*⟩

lemma *beh-prefix-closed*:
  $c @ b \in beh\ T \Longrightarrow b \in beh\ T$
⟨*proof*⟩

States in behaviours are exactly reachable.

lemma *beh-in-reach* [*rule-format*]:
  $b \in beh\ T \Longrightarrow (\forall s \in set\ b.\ s \in reach\ T)$
⟨*proof*⟩

lemma *reach-in-beh*:
  assumes $s \in reach\ T$ **shows** $\exists b \in beh\ T.\ s \in set\ b$
⟨*proof*⟩

lemma *reach-equiv-beh-states*: $reach\ T = \bigcup\ (set`(beh\ T))$
⟨*proof*⟩

2.1.2 Specifications, observability, and implementation

Specifications add an observer function to transition systems.

record $('s,\ 'o)\ spec = 's\ TS\ +$
  $obs :: 's \Rightarrow 'o$

lemma *beh-obs-upd* [*simp*]: $beh\ (S(|\ obs := x\ |)) = beh\ S$
⟨*proof*⟩

lemma *reach-obs-upd* [*simp*]: $reach\ (S(|\ obs := x\ |)) = reach\ S$
⟨*proof*⟩

Observable behaviour and reachability.

definition
  $obeh :: ('s,\ 'o)\ spec \Rightarrow ('o\ list)\ set$ **where**
  $obeh\ S \equiv (map\ (obs\ S))`(beh\ S)$

definition
  $oreach :: ('s,\ 'o)\ spec \Rightarrow 'o\ set$ **where**
  $oreach\ S \equiv (obs\ S)`(reach\ S)$

lemma *oreach-equiv-obeh-states*:
  $oreach\ S = \bigcup\ (set`(obeh\ S))$
⟨*proof*⟩

lemma *obeh-pi-translation*:
  $(map\ pi)`(obeh\ S) = obeh\ (S(|\ obs := pi\ o\ (obs\ S)\ |))$
⟨*proof*⟩

lemma *oreach-pi-translation*:
  $pi`(oreach\ S) = oreach\ (S(|\ obs := pi\ o\ (obs\ S)\ |))$
⟨*proof*⟩

A predicate P on the states of a specification is *observable* if it cannot distinguish between states yielding the same observation. Equivalently, P is observable if it is the inverse image under the observation function of a predicate on observations.

definition
  $observable :: ['s \Rightarrow 'o,\ 's\ set] \Rightarrow bool$
where
  $observable\ ob\ P \equiv \forall\ s\ s'.\ ob\ s = ob\ s' \longrightarrow s' \in P \longrightarrow s \in P$

definition
  $observable2 :: ['s \Rightarrow 'o,\ 's\ set] \Rightarrow bool$
where
  $observable2\ ob\ P \equiv \exists\ Q.\ P = ob-`Q$

definition

observable3 :: $['s \Rightarrow 'o, 's\ set] \Rightarrow bool$
where
observable3 ob P $\equiv$ *ob*$-$'*ob*'*P* $\subseteq$ *P* — other direction holds trivially

lemma *observableE* [*elim*]:
$[\![$*observable ob P*; *ob s* = *ob s'*; *s'* $\in$ *P*$]\!] \Longrightarrow$ *s* $\in$ *P*
⟨*proof*⟩

lemma *observable2-equiv-observable*: *observable2 ob P* = *observable ob P*
⟨*proof*⟩

lemma *observable3-equiv-observable2*: *observable3 ob P* = *observable2 ob P*
⟨*proof*⟩

lemma *observable-id* [*simp*]: *observable id P*
⟨*proof*⟩

The set extension of a function *ob* is the left adjoint of a Galois connection on the powerset lattices over domain and range of *ob* where the right adjoint is the inverse image function.

lemma *image-vimage-adjoints*: (*ob*'*P* $\subseteq$ *Q*) = (*P* $\subseteq$ *ob*$-$'*Q*)
⟨*proof*⟩

declare *image-vimage-subset* [*simp*, *intro*]
declare *vimage-image-subset* [*simp*, *intro*]

Similar but "reversed" (wrt to adjointness) relationships only hold under additional conditions.

lemma *image-r-vimage-l*: $[\![$ *Q* $\subseteq$ *ob*'*P*; *observable ob P* $]\!] \Longrightarrow$ *ob*$-$'*Q* $\subseteq$ *P*
⟨*proof*⟩

lemma *vimage-l-image-r*: $[\![$ *ob*$-$'*Q* $\subseteq$ *P*; *Q* $\subseteq$ *range ob* $]\!] \Longrightarrow$ *Q* $\subseteq$ *ob*'*P*
⟨*proof*⟩

Internal and external invariants

lemma *external-from-internal-invariant*:
$[\![$ *reach S* $\subseteq$ *P*; (*obs S*)'*P* $\subseteq$ *Q* $]\!]$
$\Longrightarrow$ *oreach S* $\subseteq$ *Q*
⟨*proof*⟩

lemma *external-from-internal-invariant-vimage*:
$[\![$ *reach S* $\subseteq$ *P*; *P* $\subseteq$ (*obs S*)$-$'*Q* $]\!]$
$\Longrightarrow$ *oreach S* $\subseteq$ *Q*
⟨*proof*⟩

lemma *external-to-internal-invariant-vimage*:
$[\![$ *oreach S* $\subseteq$ *Q*; (*obs S*)$-$'*Q* $\subseteq$ *P* $]\!]$
$\Longrightarrow$ *reach S* $\subseteq$ *P*
⟨*proof*⟩

lemma *external-to-internal-invariant*:
$[\![$ *oreach S* $\subseteq$ *Q*; *Q* $\subseteq$ (*obs S*)'*P*; *observable* (*obs S*) *P* $]\!]$
$\Longrightarrow$ *reach S* $\subseteq$ *P*

⟨*proof*⟩

lemma *external-equiv-internal-invariant-vimage*:
  $[\![P = (obs\ S)-`Q]\!]$
  $\Longrightarrow (oreach\ S \subseteq Q) = (reach\ S \subseteq P)$
⟨*proof*⟩

lemma *external-equiv-internal-invariant*:
  $[\![(obs\ S)`P = Q;\ observable\ (obs\ S)\ P]\!]$
  $\Longrightarrow (oreach\ S \subseteq Q) = (reach\ S \subseteq P)$
⟨*proof*⟩

Our notion of implementation is inclusion of observable behaviours.

definition
  $implements :: [{'p} \Rightarrow {'o},\ ({'s},\ {'o})\ spec,\ ({'t},\ {'p})\ spec] \Rightarrow bool$ **where**
  $implements\ pi\ Sa\ Sc \equiv (map\ pi)`(obeh\ Sc) \subseteq obeh\ Sa$

Reflexivity and transitivity

lemma *implements-refl*: *implements id S S*
⟨*proof*⟩

lemma *implements-trans*:
  $[\![implements\ pi1\ S1\ S2;\ implements\ pi2\ S2\ S3]\!]$
  $\Longrightarrow implements\ (pi1\ o\ pi2)\ S1\ S3$
⟨*proof*⟩

Preservation of external invariants

lemma *implements-oreach*:
  $implements\ pi\ Sa\ Sc \Longrightarrow pi`(oreach\ Sc) \subseteq oreach\ Sa$
⟨*proof*⟩

lemma *external-invariant-preservation*:
  $[\![oreach\ Sa \subseteq Q;\ implements\ pi\ Sa\ Sc]\!]$
  $\Longrightarrow pi`(oreach\ Sc) \subseteq Q$
⟨*proof*⟩

lemma *external-invariant-translation*:
  $[\![oreach\ Sa \subseteq Q;\ pi-`Q \subseteq P;\ implements\ pi\ Sa\ Sc]\!]$
  $\Longrightarrow oreach\ Sc \subseteq P$
⟨*proof*⟩

Preservation of internal invariants

lemma *internal-invariant-translation*:
  $[\![reach\ Sa \subseteq Pa;\ Pa \subseteq obs\ Sa\ -`\ Qa;\ pi\ -`\ Qa \subseteq Q;\ obs\ S\ -`\ Q \subseteq P;$
    $implements\ pi\ Sa\ S]\!]$
  $\Longrightarrow reach\ S \subseteq P$
⟨*proof*⟩

2.2 Invariants

First we define Hoare triples over transition relations and then we derive proof rules to establish invariants.

2.2.1 Hoare triples

definition
  PO-hoare :: $['s\ set, ('s \times 's)\ set, 's\ set] \Rightarrow bool$
    (‹(3{-} - {> -})› [0, 0, 0] 90)
where
  $\{pre\}\ R\ \{>\ post\} \equiv R``pre \subseteq post$

lemmas *PO-hoare-defs* = *PO-hoare-def Image-def*

lemma $\{P\}\ R\ \{>\ Q\} = (\forall\, s\ t.\ s \in P \longrightarrow (s,\ t) \in R \longrightarrow t \in Q)$
⟨*proof*⟩

Some essential facts about Hoare triples.

lemma *hoare-conseq-left* [*intro*]:
  $[\![\ \{P'\}\ R\ \{>\ Q\};\ P \subseteq P'\ ]\!]$
  $\Longrightarrow \{P\}\ R\ \{>\ Q\}$
⟨*proof*⟩

lemma *hoare-conseq-right*:
  $[\![\ \{P\}\ R\ \{>\ Q'\};\ Q' \subseteq Q\ ]\!]$
  $\Longrightarrow \{P\}\ R\ \{>\ Q\}$
⟨*proof*⟩

lemma *hoare-false-left* [*simp*]:
  $\{\{\}\}\ R\ \{>\ Q\}$
⟨*proof*⟩

lemma *hoare-true-right* [*simp*]:
  $\{P\}\ R\ \{>\ UNIV\}$
⟨*proof*⟩

lemma *hoare-conj-right* [*intro*!]:
  $[\![\ \{P\}\ R\ \{>\ Q1\};\ \{P\}\ R\ \{>\ Q2\}\ ]\!]$
  $\Longrightarrow \{P\}\ R\ \{>\ Q1 \cap Q2\}$
⟨*proof*⟩

Special transition relations.

lemma *hoare-stop* [*simp*, *intro*!]:
  $\{P\}\ \{\}\ \{>\ Q\}$
⟨*proof*⟩

lemma *hoare-skip* [*simp*, *intro*!]:
  $P \subseteq Q \Longrightarrow \{P\}\ Id\ \{>\ Q\}$
⟨*proof*⟩

lemma *hoare-trans-Un* [*iff*]:

$\{P\}$ *R1* $\cup$ *R2* $\{> Q\} = (\{P\}$ *R1* $\{> Q\} \wedge \{P\}$ *R2* $\{> Q\})$
⟨*proof*⟩

lemma *hoare-trans-UN* [*iff*]:
$\{P\} \bigcup x.\ R\ x\ \{> Q\} = (\forall x.\ \{P\}\ R\ x\ \{> Q\})$
⟨*proof*⟩

lemma *hoare-apply*:
$\{P\}\ R\ \{>Q\} \Longrightarrow x \in P \Longrightarrow (x,\ y) \in R \Longrightarrow y \in Q$
⟨*proof*⟩

2.2.2 Characterization of reachability

lemma *reach-init*: *reach* $T \subseteq I \Longrightarrow$ *init* $T \subseteq I$
⟨*proof*⟩

lemma *reach-trans*: *reach* $T \subseteq I \Longrightarrow \{$*reach* $T\}$ *trans* $T\ \{> I\}$
⟨*proof*⟩

Useful consequences.

corollary *init-reach* [*iff*]: *init* $T \subseteq$ *reach* T
⟨*proof*⟩

corollary *trans-reach* [*iff*]: $\{$*reach* $T\}$ *trans* $T\ \{>$ *reach* $T\}$
⟨*proof*⟩

2.2.3 Invariant proof rules

Basic proof rule for invariants.

lemma *inv-rule-basic*:
⟦ *init* $T \subseteq P$; $\{P\}$ (*trans* T) $\{> P\}$ ⟧
$\Longrightarrow$ *reach* $T \subseteq P$
⟨*proof*⟩

General invariant proof rule. This rule is complete (set $I =$ *reach* T).

lemma *inv-rule*:
⟦ *init* $T \subseteq I$; $I \subseteq P$; $\{I\}$ (*trans* T) $\{> I\}$ ⟧
$\Longrightarrow$ *reach* $T \subseteq P$
⟨*proof*⟩

The following rule is equivalent to the previous one.

lemma *INV-rule*:
⟦ *init* $T \subseteq I$; $\{I \cap$ *reach* $T\}$ (*trans* T) $\{> I\}$ ⟧
$\Longrightarrow$ *reach* $T \subseteq I$
⟨*proof*⟩

Proof of equivalence.

lemma *inv-rule-from-INV-rule*:
⟦ *init* $T \subseteq I$; $I \subseteq P$; $\{I\}$ (*trans* T) $\{> I\}$ ⟧
$\Longrightarrow$ *reach* $T \subseteq P$
⟨*proof*⟩

lemma *INV-rule-from-inv-rule*:
  $\llbracket$ *init* $T \subseteq I$; $\{I \cap$ *reach* $T\}$ (*trans* T) $\{> I\}$ $\rrbracket$
  $\Longrightarrow$ *reach* $T \subseteq I$
⟨*proof*⟩

Incremental proof rule for invariants using auxiliary invariant(s). This rule might have become obsolete by addition of *INV_rule*.

lemma *inv-rule-incr*:
  $\llbracket$ *init* $T \subseteq I$; $\{I \cap J\}$ (*trans* T) $\{> I\}$; *reach* $T \subseteq J$ $\rrbracket$
  $\Longrightarrow$ *reach* $T \subseteq I$
⟨*proof*⟩

2.3 Refinement

Our notion of refinement is simulation. We first define a general notion of relational Hoare tuple, which we then use to define the refinement proof obligation. Finally, we show that observation-consistent refinement of specifications implies the implementation relation between them.

2.3.1 Relational Hoare tuples

Relational Hoare tuples formalize the following generalized simulation diagram:

```
 o -- Ra --> o
 |           |
pre         post
 |           |
 v           V
 o -- Rc --> o
```

Here, *Ra* and *Rc* are the abstract and concrete transition relations, and *pre* and *post* are the pre- and post-relations. (In the definiton below, the operator (O) stands for relational composition, which is defined as follows: (O) $\equiv \lambda r\ s.\ \{(xa,\ x).\ ((\lambda x\ xa.\ (x,\ xa) \in r)\ OO\ (\lambda x\ xa.\ (x,\ xa) \in s))\ xa\ x\}$.)

definition
  PO-rhoare ::
    $[('s \times 't)\ set,\ ('s \times 's)\ set,\ ('t \times 't)\ set,\ ('s \times 't)\ set] \Rightarrow bool$
    (‹(*4*{-} -, - {> -})› [*0*, *0*, *0*] *90*)
where
  $\{pre\}$ *Ra*, *Rc* $\{>$ *post*$\}$ $\equiv$ *pre* O *Rc* $\subseteq$ *Ra* O *post*

lemmas *PO-rhoare-defs* = *PO-rhoare-def relcomp-unfold*

Facts about relational Hoare tuples.

lemma *relhoare-conseq-left* [*intro*]:
  $\llbracket$ $\{pre'\}$ *Ra*, *Rc* $\{>$ *post*$\}$; *pre* $\subseteq$ *pre'* $\rrbracket$
  $\Longrightarrow$ $\{pre\}$ *Ra*, *Rc* $\{>$ *post*$\}$
⟨*proof*⟩

lemma *relhoare-conseq-right*: — do NOT declare [intro]
$[\![$ $\{pre\}$ Ra, Rc $\{> post'\}$; $post' \subseteq post$ $]\!]$
$\Longrightarrow \{pre\}$ Ra, Rc $\{> post\}$
⟨*proof*⟩

lemma *relhoare-false-left* [*simp*]: — do NOT declare [intro]
$\{\ \{\}\ \}$ Ra, Rc $\{> post\}$
⟨*proof*⟩

lemma *relhoare-true-right* [*simp*]: — not true in general
$\{pre\}$ Ra, Rc $\{> UNIV\} = (Domain\ (pre\ O\ Rc) \subseteq Domain\ Ra)$
⟨*proof*⟩

lemma *Domain-rel-comp* [*intro*]:
$Domain\ pre \subseteq R \Longrightarrow Domain\ (pre\ O\ Rc) \subseteq R$
⟨*proof*⟩

lemma *rel-hoare-skip* [*iff*]: $\{R\}$ Id, Id $\{> R\}$
⟨*proof*⟩

Reflexivity and transitivity.

lemma *relhoare-refl* [*simp*]: $\{Id\}$ R, R $\{> Id\}$
⟨*proof*⟩

lemma *rhoare-trans*:
$[\![$ $\{R1\}$ $T1$, $T2$ $\{> R1\}$; $\{R2\}$ $T2$, $T3$ $\{> R2\}$ $]\!]$
$\Longrightarrow \{R1\ O\ R2\}$ $T1$, $T3$ $\{> R1\ O\ R2\}$
⟨*proof*⟩

Conjunction in the post-relation cannot be split in general. However, here are two useful special cases. In the first case the abstract transtition relation is deterministic and in the second case one conjunct is a cartesian product of two state predicates.

lemma *relhoare-conj-right-det*:
$[\![$ $\{pre\}$ Ra, Rc $\{> post1\}$; $\{pre\}$ Ra, Rc $\{> post2\}$;
single-valued Ra $]\!]$ — only for deterministic *Ra*!
$\Longrightarrow \{pre\}$ Ra, Rc $\{> post1 \cap post2\}$
⟨*proof*⟩

lemma *relhoare-conj-right-cartesian* [*intro*]:
$[\![$ $\{Domain\ pre\}$ Ra $\{> I\}$; $\{Range\ pre\}$ Rc $\{> J\}$;
$\{pre\}$ Ra, Rc $\{> post\}$ $]\!]$
$\Longrightarrow \{pre\}$ Ra, Rc $\{> post \cap I \times J\}$
⟨*proof*⟩

Separate rule for cartesian products.

corollary *relhoare-cartesian*:
$[\![$ $\{Domain\ pre\}$ Ra $\{> I\}$; $\{Range\ pre\}$ Rc $\{> J\}$;
$\{pre\}$ Ra, Rc $\{> post\}$ $]\!]$ — any *post*, including *UNIV*!
$\Longrightarrow \{pre\}$ Ra, Rc $\{> I \times J\}$
⟨*proof*⟩

Unions of transition relations.

lemma *relhoare-concrete-Un* [*simp*]:
  $\{pre\}$ Ra, $Rc1 \cup Rc2$ $\{> post\}$
  $= (\{pre\}\ Ra,\ Rc1\ \{> post\} \wedge \{pre\}\ Ra,\ Rc2\ \{> post\})$
⟨*proof*⟩

lemma *relhoare-concrete-UN* [*simp*]:
  $\{pre\}\ Ra,\ \bigcup x.\ Rc\ x\ \{> post\} = (\forall x.\ \{pre\}\ Ra,\ Rc\ x\ \{> post\})$
⟨*proof*⟩

lemma *relhoare-abstract-Un-left* [*intro*]:
  $[\![\ \{pre\}\ Ra1,\ Rc\ \{> post\}\ ]\!]$
  $\Longrightarrow \{pre\}\ Ra1 \cup Ra2,\ Rc\ \{> post\}$
⟨*proof*⟩

lemma *relhoare-abstract-Un-right* [*intro*]:
  $[\![\ \{pre\}\ Ra2,\ Rc\ \{> post\}\ ]\!]$
  $\Longrightarrow \{pre\}\ Ra1 \cup Ra2,\ Rc\ \{> post\}$
⟨*proof*⟩

lemma *relhoare-abstract-UN* [*intro*]: — ! might be too aggressive? INDEED.
  $[\![\ \{pre\}\ Ra\ x,\ Rc\ \{> post\}\ ]\!]$
  $\Longrightarrow \{pre\}\ \bigcup x.\ Ra\ x,\ Rc\ \{> post\}$
⟨*proof*⟩

Inclusion of abstract transition relations.

lemma *relhoare-abstract-trans-weak* [*intro*]:
  $[\![\ \{pre\}\ Ra',\ Rc\ \{> post\};\ Ra' \subseteq Ra\ ]\!]$
  $\Longrightarrow \{pre\}\ Ra,\ Rc\ \{> post\}$
⟨*proof*⟩

2.3.2 Refinement proof obligations

A transition system refines another one if the initial states and the transitions are refined. Initial state refinement means that for each concrete initial state there is a related abstract one. Transition refinement means that the simulation relation is preserved (as expressed by a relational Hoare tuple).

definition
  PO-refines ::
    $[('s \times 't)\ set,\ ('s,\ 'a)\ TS\text{-}scheme,\ ('t,\ 'b)\ TS\text{-}scheme] \Rightarrow bool$
where
  $PO\text{-}refines\ R\ Ta\ Tc \equiv ($
    $init\ Tc \subseteq R``(init\ Ta)$
    $\wedge\ \{R\}\ (trans\ Ta),\ (trans\ Tc)\ \{> R\}$
  $)$

lemma *PO-refinesI*:
  $[\![\ init\ Tc \subseteq R``(init\ Ta);\ \{R\}\ (trans\ Ta),\ (trans\ Tc)\ \{> R\}\ ]\!] \Longrightarrow PO\text{-}refines\ R\ Ta\ Tc$
⟨*proof*⟩

lemma *PO-refinesE* [*elim*]:
  $[\![\ PO\text{-}refines\ R\ Ta\ Tc;\ [\![\ init\ Tc \subseteq R``(init\ Ta);\ \{R\}\ (trans\ Ta),\ (trans\ Tc)\ \{> R\}\ ]\!] \Longrightarrow P\ ]\!]$
  $\Longrightarrow P$

$\langle proof \rangle$

Basic refinement rule. This is just an introduction rule for the definition.

lemma *refine-basic*:
$[\![$ *init Tc* $\subseteq$ *R*``(*init Ta*); {*R*} (*trans Ta*), (*trans Tc*) {> *R*} $]\!]$
$\Longrightarrow$ *PO-refines R Ta Tc*
$\langle proof \rangle$

The following proof rule uses individual invariants I and J of the concrete and abstract systems to strengthen the simulation relation R.

The hypotheses state that these state predicates are indeed invariants. Note that the precondition of the invariant preservation hypotheses for I and J are strengthened by adding the predicates *Domain* ($R \cap$ *UNIV* $\times J$) and *Range* ($R \cap I \times$ *UNIV*), respectively. In particular, the latter predicate may be essential, if a concrete invariant depends on the simulation relation and an abstract invariant, i.e. to "transport" abstract invariants to the concrete system.

lemma *refine-init-using-invariants*:
$[\![$ *init Tc* $\subseteq$ *R*``(*init Ta*); *init Ta* $\subseteq I$; *init Tc* $\subseteq J$ $]\!]$
$\Longrightarrow$ *init Tc* $\subseteq$ ($R \cap I \times J$)``(*init Ta*)
$\langle proof \rangle$

lemma *refine-trans-using-invariants*:
$[\![$ {$R \cap I \times J$} (*trans Ta*), (*trans Tc*) {> *R*};
{$I \cap$ *Domain* ($R \cap$ *UNIV* $\times J$)} (*trans Ta*) {> *I*};
{$J \cap$ *Range* ($R \cap I \times$ *UNIV*)} (*trans Tc*) {> *J*} $]\!]$
$\Longrightarrow$ {$R \cap I \times J$} (*trans Ta*), (*trans Tc*) {> $R \cap I \times J$}
$\langle proof \rangle$

This is our main rule for refinements.

lemma *refine-using-invariants*:
$[\![$ {$R \cap I \times J$} (*trans Ta*), (*trans Tc*) {> *R*};
{$I \cap$ *Domain* ($R \cap$ *UNIV* $\times J$)} (*trans Ta*) {> *I*};
{$J \cap$ *Range* ($R \cap I \times$ *UNIV*)} (*trans Tc*) {> *J*};
init Tc $\subseteq$ *R*``(*init Ta*);
init Ta $\subseteq I$; *init Tc* $\subseteq J$ $]\!]$
$\Longrightarrow$ *PO-refines* ($R \cap I \times J$) *Ta Tc*
$\langle proof \rangle$

2.3.3 Deriving invariants from refinements

Some invariants can only be proved after the simulation has been established, because they depend on the simulation relation and some abstract invariants. Here is a rule to derive invariant theorems from the refinement.

lemma *PO-refines-implies-Range-init*:
PO-refines R Ta Tc $\Longrightarrow$ *init Tc* $\subseteq$ *Range R*
$\langle proof \rangle$

lemma *PO-refines-implies-Range-trans*:
PO-refines R Ta Tc $\Longrightarrow$ {*Range R*} *trans Tc* {> *Range R*}
$\langle proof \rangle$

lemma *PO-refines-implies-Range-invariant*:
  $PO\text{-}refines\ R\ Ta\ Tc \Longrightarrow reach\ Tc \subseteq Range\ R$
⟨*proof*⟩

The following rules are more useful in proofs.

corollary *INV-init-from-refinement*:
  $[\![\ PO\text{-}refines\ R\ Ta\ Tc;\ Range\ R \subseteq I\ ]\!]$
  $\Longrightarrow init\ Tc \subseteq I$
⟨*proof*⟩

corollary *INV-trans-from-refinement*:
  $[\![\ PO\text{-}refines\ R\ Ta\ Tc;\ K \subseteq Range\ R;\ Range\ R \subseteq I\ ]\!]$
  $\Longrightarrow \{K\}\ trans\ Tc\ \{>\ I\}$
⟨*proof*⟩

corollary *INV-from-refinement*:
  $[\![\ PO\text{-}refines\ R\ Ta\ Tc;\ Range\ R \subseteq I\ ]\!]$
  $\Longrightarrow reach\ Tc \subseteq I$
⟨*proof*⟩

2.3.4 Refinement of specifications

Lift relation membership to finite sequences

inductive-set
  $seq\text{-}lift :: ('s \times 't)\ set \Rightarrow ('s\ list \times 't\ list)\ set$
  for $R :: ('s \times 't)\ set$
where
  sl-nil [*iff*]: $([],\ []) \in seq\text{-}lift\ R$
| *sl-cons* [*intro*]:
    $[\![\ (xs,\ ys) \in seq\text{-}lift\ R;\ (x,\ y) \in R\ ]\!] \Longrightarrow (x\#xs,\ y\#ys) \in seq\text{-}lift\ R$

inductive-cases *sl-cons-right-invert*: $(ba',\ t\ \#\ bc) \in seq\text{-}lift\ R$

For each concrete behaviour there is a related abstract one.

lemma *behaviour-refinement*:
  $[\![\ PO\text{-}refines\ R\ Ta\ Tc;\ bc \in beh\ Tc]\!]$
  $\Longrightarrow \exists\, ba \in beh\ Ta.\ (ba,\ bc) \in seq\text{-}lift\ R$
⟨*proof*⟩

Observation consistency of a relation is defined using a mediator function *pi* to abstract the concrete observation. This allows us to also refine the observables as we move down a refinement branch.

definition
  obs-consistent ::
    $[('s \times 't)\ set,\ 'p \Rightarrow 'o,\ ('s,\ 'o)\ spec,\ ('t,\ 'p)\ spec] \Rightarrow bool$
where
  $obs\text{-}consistent\ R\ pi\ Sa\ Sc \equiv (\forall\, s\ t.\ (s,\ t) \in R \longrightarrow pi\ (obs\ Sc\ t) = obs\ Sa\ s)$

lemma *obs-consistent-refl* [*iff*]: *obs-consistent Id id S S*
⟨*proof*⟩

lemma *obs-consistent-trans* [*intro*]:
$\llbracket$ *obs-consistent R1 pi1 S1 S2*; *obs-consistent R2 pi2 S2 S3* $\rrbracket$
$\Longrightarrow$ *obs-consistent* (*R1 O R2*) (*pi1 o pi2*) *S1 S3*
⟨*proof*⟩

lemma *obs-consistent-empty*: *obs-consistent* {} *pi Sa Sc*
⟨*proof*⟩

lemma *obs-consistent-conj1* [*intro*]:
obs-consistent R pi Sa Sc $\Longrightarrow$ *obs-consistent* ($R \cap R'$) *pi Sa Sc*
⟨*proof*⟩

lemma *obs-consistent-conj2* [*intro*]:
obs-consistent R pi Sa Sc $\Longrightarrow$ *obs-consistent* ($R' \cap R$) *pi Sa Sc*
⟨*proof*⟩

lemma *obs-consistent-behaviours*:
$\llbracket$ *obs-consistent R pi Sa Sc*; *bc* $\in$ *beh Sc*; *ba* $\in$ *beh Sa*; (*ba*, *bc*) $\in$ *seq-lift R*$\rrbracket$
$\Longrightarrow$ *map pi* (*map* (*obs Sc*) *bc*) = *map* (*obs Sa*) *ba*
⟨*proof*⟩

Definition of refinement proof obligations.

definition
refines ::
[($'s \times 't$) *set*, $'p \Rightarrow 'o$, ($'s$, $'o$) *spec*, ($'t$, $'p$) *spec*] $\Rightarrow$ *bool*
where
refines R pi Sa Sc $\equiv$ *obs-consistent R pi Sa Sc* $\wedge$ *PO-refines R Sa Sc*

lemmas *refines-defs* =
refines-def PO-refines-def

lemma *refinesI*:
$\llbracket$ *PO-refines R Sa Sc*; *obs-consistent R pi Sa Sc* $\rrbracket$
$\Longrightarrow$ *refines R pi Sa Sc*
⟨*proof*⟩

lemma *refinesE* [*elim*]:
$\llbracket$ *refines R pi Sa Sc*; $\llbracket$ *PO-refines R Sa Sc*; *obs-consistent R pi Sa Sc* $\rrbracket \Longrightarrow P$ $\rrbracket$
$\Longrightarrow P$
⟨*proof*⟩

Reflexivity and transitivity of refinement.

lemma *refinement-reflexive*: *refines Id id S S*
⟨*proof*⟩

lemma *refinement-transitive*:
$\llbracket$ *refines R1 pi1 S1 S2*; *refines R2 pi2 S2 S3* $\rrbracket$
$\Longrightarrow$ *refines* (*R1 O R2*) (*pi1 o pi2*) *S1 S3*
⟨*proof*⟩

Soundness of refinement for proving implementation

lemma *observable-behaviour-refinement*:

$\llbracket$ *refines R pi Sa Sc*; *bc* $\in$ *obeh Sc* $\rrbracket \Longrightarrow$ *map pi bc* $\in$ *obeh Sa*
⟨*proof*⟩

theorem *refinement-soundness*:
refines R pi Sa Sc $\Longrightarrow$ *implements pi Sa Sc*
⟨*proof*⟩

Extended versions of refinement proof rules including observations

lemmas *Refinement-basic* = *refine-basic* [*THEN refinesI*]
lemmas *Refinement-using-invariants* = *refine-using-invariants* [*THEN refinesI*]

lemma *refines-reachable-strengthening*:
refines R pi Sa Sc $\Longrightarrow$ *refines* ($R \cap$ *reach Sa* $\times$ *reach Sc*) *pi Sa Sc*
⟨*proof*⟩

Inhertitance of internal invariants through refinements

lemma *INV-init-from-Refinement*:
$\llbracket$*refines R pi Sa Sc*; *Range R* $\subseteq I\rrbracket \Longrightarrow$ *init Sc* $\subseteq I$
⟨*proof*⟩

lemma *INV-trans-from-Refinement*:
$\llbracket$*refines R pi Sa Sc*; $K \subseteq$ *Range R*; *Range R* $\subseteq I\rrbracket \Longrightarrow \{K\}$ *TS.trans Sc* $\{> I\}$
⟨*proof*⟩

lemma *INV-from-Refinement-basic*:
$\llbracket$ *refines R pi Sa Sc*; *Range R* $\subseteq I$ $\rrbracket \Longrightarrow$ *reach Sc* $\subseteq I$
⟨*proof*⟩

lemma *INV-from-Refinement-using-invariants*:
assumes *refines R pi Sa Sc Range* ($R \cap I \times J$) $\subseteq K$
reach Sa $\subseteq I$ *reach Sc* $\subseteq J$
shows *reach Sc* $\subseteq K$
⟨*proof*⟩

end

3 Message definitions

theory *Messages*
imports *Main*
begin

3.1 Messages

Agents

datatype
  agent = *Agent nat*

Nonces

typedecl *fid-t*

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

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

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

datatype
  nonce-t =
    nonce-fresh fresh-t
  | *nonce-atk nat*

Keys

datatype *ltkey* =
  sharK agent agent
| *publK agent*
| *privK agent*

datatype *ephkey* =
  epublK nonce-t
| *eprivK nonce-t*

datatype *tag* = *insec* | *auth* | *confid* | *secure*

Messages

datatype *cmsg* =
  cAgent agent
| *cNumber nat*
| *cNonce nonce-t*
| *cLtK ltkey*
| *cEphK ephkey*
| *cPair cmsg cmsg*
| *cEnc cmsg cmsg*
| *cAenc cmsg cmsg*
| *cSign cmsg cmsg*

| *cHash cmsg*
| *cTag tag*
| *cExp cmsg cmsg*

fun *catomic* :: *cmsg* ⇒ *bool*
where
catomic (*cAgent* -) = *True*
| *catomic* (*cNumber* -) = *True*
| *catomic* (*cNonce* -) = *True*
| *catomic* (*cLtK* -) = *True*
| *catomic* (*cEphK* -) = *True*
| *catomic* (*cTag* -) = *True*
| *catomic* - = *False*

inductive *eq* :: *cmsg* ⇒ *cmsg* ⇒ *bool*
where
— equations
Permute [*intro*]:*eq* (*cExp* (*cExp a b*) *c*) (*cExp* (*cExp a c*) *b*)
— closure by context
| *Tag*[*intro*]: *eq* (*cTag t*) (*cTag t*)
| *Agent*[*intro*]: *eq* (*cAgent A*) (*cAgent A*)
| *Nonce*[*intro*]:*eq* (*cNonce x*) (*cNonce x*)
| *Number*[*intro*]:*eq* (*cNumber x*) (*cNumber x*)
| *LtK*[*intro*]:*eq* (*cLtK x*) (*cLtK x*)
| *EphK*[*intro*]:*eq* (*cEphK x*) (*cEphK x*)
| *Pair*[*intro*]:*eq a b* ⟹ *eq c d* ⟹ *eq* (*cPair a c*) (*cPair b d*)
| *Enc*[*intro*]:*eq a b* ⟹ *eq c d* ⟹ *eq* (*cEnc a c*) (*cEnc b d*)
| *Aenc*[*intro*]:*eq a b* ⟹ *eq c d* ⟹ *eq* (*cAenc a c*) (*cAenc b d*)
| *Sign*[*intro*]:*eq a b* ⟹ *eq c d* ⟹ *eq* (*cSign a c*) (*cSign b d*)
| *Hash*[*intro*]:*eq a b* ⟹ *eq* (*cHash a*) (*cHash b*)
| *Exp*[*intro*]:*eq a b* ⟹ *eq c d* ⟹ *eq* (*cExp a c*) (*cExp b d*)
— reflexive closure is not needed here because the context closure implies it
— symmetric closure is not needed as it is easier to include equations in both directions
— transitive closure
| *Tr*[*intro*]: *eq a b* ⟹ *eq b c* ⟹ *eq a c*

lemma *eq-sym*: *eq a b* ⟷ *eq b a*
⟨*proof*⟩

lemma *eq-Sym* [*intro*]: *eq a b* ⟹ *eq b a*
⟨*proof*⟩

lemma *eq-refl* [*simp*, *intro*]: *eq a a*
⟨*proof*⟩

inductive cases; keep the transitivity case, so we prove the the right lemmas by hand.

lemma *eq-Number*: *eq* (*cNumber N*) *y* ⟹ *y* = *cNumber N*
⟨*proof*⟩
lemma *eq-Agent*: *eq* (*cAgent A*) *y* ⟹ *y* = *cAgent A*
⟨*proof*⟩
lemma *eq-Nonce*: *eq* (*cNonce N*) *y* ⟹ *y* = *cNonce N*
⟨*proof*⟩

lemma *eq-LtK*: *eq* (*cLtK N*) *y* $\Longrightarrow$ *y* = *cLtK N*
⟨*proof*⟩
lemma *eq-EphK*: *eq* (*cEphK N*) *y* $\Longrightarrow$ *y* = *cEphK N*
⟨*proof*⟩
lemma *eq-Tag*: *eq* (*cTag N*) *y* $\Longrightarrow$ *y* = *cTag N*
⟨*proof*⟩
lemma *eq-Hash*: *eq* (*cHash X*) *y* $\Longrightarrow \exists$ *Y*. *y* = *cHash Y* $\wedge$ *eq X Y*
⟨*proof*⟩
lemma *eq-Pair*: *eq* (*cPair X Y*) *y* $\Longrightarrow \exists$ *X' Y'*. *y* = *cPair X' Y'* $\wedge$ *eq X X'* $\wedge$ *eq Y Y'*
⟨*proof*⟩
lemma *eq-Enc*: *eq* (*cEnc X Y*) *y* $\Longrightarrow \exists$ *X' Y'*. *y* = *cEnc X' Y'* $\wedge$ *eq X X'* $\wedge$ *eq Y Y'*
⟨*proof*⟩
lemma *eq-Aenc*: *eq* (*cAenc X Y*) *y* $\Longrightarrow \exists$ *X' Y'*. *y* = *cAenc X' Y'* $\wedge$ *eq X X'* $\wedge$ *eq Y Y'*
⟨*proof*⟩
lemma *eq-Sign*: *eq* (*cSign X Y*) *y* $\Longrightarrow \exists$ *X' Y'*. *y* = *cSign X' Y'* $\wedge$ *eq X X'* $\wedge$ *eq Y Y'*
⟨*proof*⟩
lemma *eq-Exp*: *eq* (*cExp X Y*) *y* $\Longrightarrow \exists$ *X' Y'*. *y* = *cExp X' Y'*
⟨*proof*⟩

lemmas *eqD-aux* = *eq-Number eq-Agent eq-Nonce eq-LtK eq-EphK eq-Tag*
eq-Hash eq-Pair eq-Enc eq-Aenc eq-Sign eq-Exp
lemmas *eqD* [*dest*] = *eqD-aux eqD-aux* [*OF eq-Sym*]

Quotient construction

quotient-type *msg* = *cmsg* / *eq*
morphisms *Re Ab*
⟨*proof*⟩

lift-definition *Number* :: *nat* ⇒ *msg* **is** *cNumber* ⟨*proof*⟩
lift-definition *Nonce* :: *nonce-t* ⇒ *msg* **is** *cNonce* ⟨*proof*⟩
lift-definition *Agent* :: *agent* ⇒ *msg* **is** *cAgent* ⟨*proof*⟩
lift-definition *LtK* :: *ltkey* ⇒ *msg* **is** *cLtK* ⟨*proof*⟩
lift-definition *EphK* :: *ephkey* ⇒ *msg* **is** *cEphK* ⟨*proof*⟩
lift-definition *Pair* :: *msg* ⇒ *msg* ⇒ *msg* **is** *cPair* ⟨*proof*⟩
lift-definition *Enc* :: *msg* ⇒ *msg* ⇒ *msg* **is** *cEnc* ⟨*proof*⟩
lift-definition *Aenc* :: *msg* ⇒ *msg* ⇒ *msg* **is** *cAenc* ⟨*proof*⟩
lift-definition *Exp* :: *msg* ⇒ *msg* ⇒ *msg* **is** *cExp* ⟨*proof*⟩
lift-definition *Tag* :: *tag* ⇒ *msg* **is** *cTag* ⟨*proof*⟩
lift-definition *Hash* :: *msg* ⇒ *msg* **is** *cHash* ⟨*proof*⟩
lift-definition *Sign* :: *msg* ⇒ *msg* ⇒ *msg* **is** *cSign* ⟨*proof*⟩

lemmas *msg-defs* =
Agent-def Number-def Nonce-def LtK-def EphK-def Pair-def
Enc-def Aenc-def Exp-def Hash-def Tag-def Sign-def

Commutativity of exponents

lemma *permute-exp* [*simp*]: *Exp* (*Exp X Y*) *Z* = *Exp* (*Exp X Z*) *Y*
⟨*proof*⟩

lift-definition *atomic* :: *msg* ⇒ *bool* **is** *catomic* ⟨*proof*⟩

abbreviation
composed :: *msg* ⇒ *bool* **where**
composed $X \equiv \neg$*atomic* X

lemma *atomic-Agent* [*simp*, *intro*]: *atomic* (*Agent* X) ⟨*proof*⟩
lemma *atomic-Tag* [*simp*, *intro*]: *atomic* (*Tag* X) ⟨*proof*⟩
lemma *atomic-Nonce* [*simp*, *intro*]: *atomic* (*Nonce* X) ⟨*proof*⟩
lemma *atomic-Number* [*simp*, *intro*]: *atomic* (*Number* X) ⟨*proof*⟩
lemma *atomic-LtK* [*simp*, *intro*]: *atomic* (*LtK* X) ⟨*proof*⟩
lemma *atomic-EphK* [*simp*, *intro*]: *atomic* (*EphK* X) ⟨*proof*⟩

lemma *non-atomic-Pair* [*simp*]: ¬*atomic* (*Pair* x y) ⟨*proof*⟩
lemma *non-atomic-Enc* [*simp*]: ¬*atomic* (*Enc* x y) ⟨*proof*⟩
lemma *non-atomic-Aenc* [*simp*]: ¬*atomic* (*Aenc* x y) ⟨*proof*⟩
lemma *non-atomic-Sign* [*simp*]: ¬*atomic* (*Sign* x y) ⟨*proof*⟩
lemma *non-atomic-Exp* [*simp*]: ¬*atomic* (*Exp* x y) ⟨*proof*⟩
lemma *non-atomic-Hash* [*simp*]: ¬*atomic* (*Hash* x) ⟨*proof*⟩

lemma *Nonce-Nonce*: (*Nonce* X = *Nonce* X') = ($X = X'$) ⟨*proof*⟩
lemma *Nonce-Agent*: *Nonce* $X \neq$ *Agent* X' ⟨*proof*⟩
lemma *Nonce-Number*: *Nonce* $X \neq$ *Number* X' ⟨*proof*⟩
lemma *Nonce-Hash*: *Nonce* $X \neq$ *Hash* X' ⟨*proof*⟩
lemma *Nonce-Tag*: *Nonce* $X \neq$ *Tag* X' ⟨*proof*⟩
lemma *Nonce-EphK*: *Nonce* $X \neq$ *EphK* X' ⟨*proof*⟩
lemma *Nonce-LtK*: *Nonce* $X \neq$ *LtK* X' ⟨*proof*⟩
lemma *Nonce-Pair*: *Nonce* $X \neq$ *Pair* X' Y' ⟨*proof*⟩
lemma *Nonce-Enc*: *Nonce* $X \neq$ *Enc* X' Y' ⟨*proof*⟩
lemma *Nonce-Aenc*: *Nonce* $X \neq$ *Aenc* X' Y' ⟨*proof*⟩
lemma *Nonce-Sign*: *Nonce* $X \neq$ *Sign* X' Y' ⟨*proof*⟩
lemma *Nonce-Exp*: *Nonce* $X \neq$ *Exp* X' Y' ⟨*proof*⟩

lemma *Agent-Nonce*: *Agent* $X \neq$ *Nonce* X' ⟨*proof*⟩
lemma *Agent-Agent*: (*Agent* X = *Agent* X') = ($X = X'$) ⟨*proof*⟩
lemma *Agent-Number*: *Agent* $X \neq$ *Number* X' ⟨*proof*⟩
lemma *Agent-Hash*: *Agent* $X \neq$ *Hash* X' ⟨*proof*⟩
lemma *Agent-Tag*: *Agent* $X \neq$ *Tag* X' ⟨*proof*⟩
lemma *Agent-EphK*: *Agent* $X \neq$ *EphK* X' ⟨*proof*⟩
lemma *Agent-LtK*: *Agent* $X \neq$ *LtK* X' ⟨*proof*⟩
lemma *Agent-Pair*: *Agent* $X \neq$ *Pair* X' Y' ⟨*proof*⟩
lemma *Agent-Enc*: *Agent* $X \neq$ *Enc* X' Y' ⟨*proof*⟩
lemma *Agent-Aenc*: *Agent* $X \neq$ *Aenc* X' Y' ⟨*proof*⟩
lemma *Agent-Sign*: *Agent* $X \neq$ *Sign* X' Y' ⟨*proof*⟩
lemma *Agent-Exp*: *Agent* $X \neq$ *Exp* X' Y' ⟨*proof*⟩

lemma *Number-Nonce*: *Number* $X \neq$ *Nonce* X' ⟨*proof*⟩
lemma *Number-Agent*: *Number* $X \neq$ *Agent* X' ⟨*proof*⟩
lemma *Number-Number*: (*Number* X = *Number* X') = ($X = X'$) ⟨*proof*⟩
lemma *Number-Hash*: *Number* $X \neq$ *Hash* X' ⟨*proof*⟩
lemma *Number-Tag*: *Number* $X \neq$ *Tag* X' ⟨*proof*⟩
lemma *Number-EphK*: *Number* $X \neq$ *EphK* X' ⟨*proof*⟩
lemma *Number-LtK*: *Number* $X \neq$ *LtK* X' ⟨*proof*⟩
lemma *Number-Pair*: *Number* $X \neq$ *Pair* X' Y' ⟨*proof*⟩
lemma *Number-Enc*: *Number* $X \neq$ *Enc* X' Y' ⟨*proof*⟩

lemma *Number-Aenc*: $Number\ X \neq Aenc\ \ X'\ Y'$ ⟨*proof*⟩
lemma *Number-Sign*: $Number\ X \neq Sign\ \ X'\ Y'$ ⟨*proof*⟩
lemma *Number-Exp*: $Number\ X \neq Exp\ \ X'\ Y'$ ⟨*proof*⟩

lemma *Hash-Nonce*: $Hash\ X \neq Nonce\ \ X'$ ⟨*proof*⟩
lemma *Hash-Agent*: $Hash\ X \neq Agent\ \ X'$ ⟨*proof*⟩
lemma *Hash-Number*: $Hash\ X \neq Number\ \ X'$ ⟨*proof*⟩
lemma *Hash-Hash*: $(Hash\ X = Hash\ X') = (X = X')$ ⟨*proof*⟩
lemma *Hash-Tag*: $Hash\ X \neq Tag\ \ X'$ ⟨*proof*⟩
lemma *Hash-EphK*: $Hash\ X \neq EphK\ \ X'$ ⟨*proof*⟩
lemma *Hash-LtK*: $Hash\ X \neq LtK\ \ X'$ ⟨*proof*⟩
lemma *Hash-Pair*: $Hash\ X \neq Pair\ \ X'\ Y'$ ⟨*proof*⟩
lemma *Hash-Enc*: $Hash\ X \neq Enc\ \ X'\ Y'$ ⟨*proof*⟩
lemma *Hash-Aenc*: $Hash\ X \neq Aenc\ \ X'\ Y'$ ⟨*proof*⟩
lemma *Hash-Sign*: $Hash\ X \neq Sign\ \ X'\ Y'$ ⟨*proof*⟩
lemma *Hash-Exp*: $Hash\ X \neq Exp\ \ X'\ Y'$ ⟨*proof*⟩

lemma *Tag-Nonce*: $Tag\ X \neq Nonce\ \ X'$ ⟨*proof*⟩
lemma *Tag-Agent*: $Tag\ X \neq Agent\ \ X'$ ⟨*proof*⟩
lemma *Tag-Number*: $Tag\ X \neq Number\ \ X'$ ⟨*proof*⟩
lemma *Tag-Hash*: $Tag\ X \neq Hash\ \ X'$ ⟨*proof*⟩
lemma *Tag-Tag*: $(Tag\ X = Tag\ X') = (X = X')$ ⟨*proof*⟩
lemma *Tag-EphK*: $Tag\ X \neq EphK\ \ X'$ ⟨*proof*⟩
lemma *Tag-LtK*: $Tag\ X \neq LtK\ \ X'$ ⟨*proof*⟩
lemma *Tag-Pair*: $Tag\ X \neq Pair\ \ X'\ Y'$ ⟨*proof*⟩
lemma *Tag-Enc*: $Tag\ X \neq Enc\ \ X'\ Y'$ ⟨*proof*⟩
lemma *Tag-Aenc*: $Tag\ X \neq Aenc\ \ X'\ Y'$ ⟨*proof*⟩
lemma *Tag-Sign*: $Tag\ X \neq Sign\ \ X'\ Y'$ ⟨*proof*⟩
lemma *Tag-Exp*: $Tag\ X \neq Exp\ \ X'\ Y'$ ⟨*proof*⟩

lemma *EphK-Nonce*: $EphK\ X \neq Nonce\ \ X'$ ⟨*proof*⟩
lemma *EphK-Agent*: $EphK\ X \neq Agent\ \ X'$ ⟨*proof*⟩
lemma *EphK-Number*: $EphK\ X \neq Number\ \ X'$ ⟨*proof*⟩
lemma *EphK-Hash*: $EphK\ X \neq Hash\ \ X'$ ⟨*proof*⟩
lemma *EphK-Tag*: $EphK\ X \neq Tag\ \ X'$ ⟨*proof*⟩
lemma *EphK-EphK*: $(EphK\ X = EphK\ X') = (X = X')$ ⟨*proof*⟩
lemma *EphK-LtK*: $EphK\ X \neq LtK\ \ X'$ ⟨*proof*⟩
lemma *EphK-Pair*: $EphK\ X \neq Pair\ \ X'\ Y'$ ⟨*proof*⟩
lemma *EphK-Enc*: $EphK\ X \neq Enc\ \ X'\ Y'$ ⟨*proof*⟩
lemma *EphK-Aenc*: $EphK\ X \neq Aenc\ \ X'\ Y'$ ⟨*proof*⟩
lemma *EphK-Sign*: $EphK\ X \neq Sign\ \ X'\ Y'$ ⟨*proof*⟩
lemma *EphK-Exp*: $EphK\ X \neq Exp\ \ X'\ Y'$ ⟨*proof*⟩

lemma *LtK-Nonce*: $LtK\ X \neq Nonce\ \ X'$ ⟨*proof*⟩
lemma *LtK-Agent*: $LtK\ X \neq Agent\ \ X'$ ⟨*proof*⟩
lemma *LtK-Number*: $LtK\ X \neq Number\ \ X'$ ⟨*proof*⟩
lemma *LtK-Hash*: $LtK\ X \neq Hash\ \ X'$ ⟨*proof*⟩
lemma *LtK-Tag*: $LtK\ X \neq Tag\ \ X'$ ⟨*proof*⟩
lemma *LtK-EphK*: $LtK\ X \neq EphK\ \ X'$ ⟨*proof*⟩
lemma *LtK-LtK*: $(LtK\ X = LtK\ X') = (X = X')$ ⟨*proof*⟩
lemma *LtK-Pair*: $LtK\ X \neq Pair\ \ X'\ Y'$ ⟨*proof*⟩
lemma *LtK-Enc*: $LtK\ X \neq Enc\ \ X'\ Y'$ ⟨*proof*⟩
lemma *LtK-Aenc*: $LtK\ X \neq Aenc\ \ X'\ Y'$ ⟨*proof*⟩

lemma *LtK-Sign*: $LtK\ X \neq Sign\ \ X'\ Y'$ ⟨*proof*⟩
lemma *LtK-Exp*: $LtK\ X \neq Exp\ \ X'\ Y'$ ⟨*proof*⟩

lemma *Pair-Nonce*: $Pair\ X\ Y \neq Nonce\ \ X'$ ⟨*proof*⟩
lemma *Pair-Agent*: $Pair\ X\ Y \neq Agent\ \ X'$ ⟨*proof*⟩
lemma *Pair-Number*: $Pair\ X\ Y \neq Number\ \ X'$ ⟨*proof*⟩
lemma *Pair-Hash*: $Pair\ X\ Y \neq Hash\ \ X'$ ⟨*proof*⟩
lemma *Pair-Tag*: $Pair\ X\ Y \neq Tag\ \ X'$ ⟨*proof*⟩
lemma *Pair-EphK*: $Pair\ X\ Y \neq EphK\ \ X'$ ⟨*proof*⟩
lemma *Pair-LtK*: $Pair\ X\ Y \neq LtK\ \ X'$ ⟨*proof*⟩
lemma *Pair-Pair*: $(Pair\ X\ Y = Pair\ X'\ Y') = (X = X' \wedge\ Y = Y')$ ⟨*proof*⟩
lemma *Pair-Enc*: $Pair\ X\ Y \neq Enc\ \ X'\ Y'$ ⟨*proof*⟩
lemma *Pair-Aenc*: $Pair\ X\ Y \neq Aenc\ \ X'\ Y'$ ⟨*proof*⟩
lemma *Pair-Sign*: $Pair\ X\ Y \neq Sign\ \ X'\ Y'$ ⟨*proof*⟩
lemma *Pair-Exp*: $Pair\ X\ Y \neq Exp\ \ X'\ Y'$ ⟨*proof*⟩

lemma *Enc-Nonce*: $Enc\ X\ Y \neq Nonce\ \ X'$ ⟨*proof*⟩
lemma *Enc-Agent*: $Enc\ X\ Y \neq Agent\ \ X'$ ⟨*proof*⟩
lemma *Enc-Number*: $Enc\ X\ Y \neq Number\ \ X'$ ⟨*proof*⟩
lemma *Enc-Hash*: $Enc\ X\ Y \neq Hash\ \ X'$ ⟨*proof*⟩
lemma *Enc-Tag*: $Enc\ X\ Y \neq Tag\ \ X'$ ⟨*proof*⟩
lemma *Enc-EphK*: $Enc\ X\ Y \neq EphK\ \ X'$ ⟨*proof*⟩
lemma *Enc-LtK*: $Enc\ X\ Y \neq LtK\ \ X'$ ⟨*proof*⟩
lemma *Enc-Pair*: $Enc\ X\ Y \neq Pair\ \ X'\ Y'$ ⟨*proof*⟩
lemma *Enc-Enc*: $(Enc\ X\ Y = Enc\ X'\ Y') = (X = X' \wedge\ Y = Y')$ ⟨*proof*⟩
lemma *Enc-Aenc*: $Enc\ X\ Y \neq Aenc\ \ X'\ Y'$ ⟨*proof*⟩
lemma *Enc-Sign*: $Enc\ X\ Y \neq Sign\ \ X'\ Y'$ ⟨*proof*⟩
lemma *Enc-Exp*: $Enc\ X\ Y \neq Exp\ \ X'\ Y'$ ⟨*proof*⟩

lemma *Aenc-Nonce*: $Aenc\ X\ Y \neq Nonce\ \ X'$ ⟨*proof*⟩
lemma *Aenc-Agent*: $Aenc\ X\ Y \neq Agent\ \ X'$ ⟨*proof*⟩
lemma *Aenc-Number*: $Aenc\ X\ Y \neq Number\ \ X'$ ⟨*proof*⟩
lemma *Aenc-Hash*: $Aenc\ X\ Y \neq Hash\ \ X'$ ⟨*proof*⟩
lemma *Aenc-Tag*: $Aenc\ X\ Y \neq Tag\ \ X'$ ⟨*proof*⟩
lemma *Aenc-EphK*: $Aenc\ X\ Y \neq EphK\ \ X'$ ⟨*proof*⟩
lemma *Aenc-LtK*: $Aenc\ X\ Y \neq LtK\ \ X'$ ⟨*proof*⟩
lemma *Aenc-Pair*: $Aenc\ X\ Y \neq Pair\ \ X'\ Y'$ ⟨*proof*⟩
lemma *Aenc-Enc*: $Aenc\ X\ Y \neq Enc\ \ X'\ Y'$ ⟨*proof*⟩
lemma *Aenc-Aenc*: $(Aenc\ X\ Y = Aenc\ X'\ Y') = (X = X' \wedge\ Y = Y')$ ⟨*proof*⟩
lemma *Aenc-Sign*: $Aenc\ X\ Y \neq Sign\ \ X'\ Y'$ ⟨*proof*⟩
lemma *Aenc-Exp*: $Aenc\ X\ Y \neq Exp\ \ X'\ Y'$ ⟨*proof*⟩

lemma *Sign-Nonce*: $Sign\ X\ Y \neq Nonce\ \ X'$ ⟨*proof*⟩
lemma *Sign-Agent*: $Sign\ X\ Y \neq Agent\ \ X'$ ⟨*proof*⟩
lemma *Sign-Number*: $Sign\ X\ Y \neq Number\ \ X'$ ⟨*proof*⟩
lemma *Sign-Hash*: $Sign\ X\ Y \neq Hash\ \ X'$ ⟨*proof*⟩
lemma *Sign-Tag*: $Sign\ X\ Y \neq Tag\ \ X'$ ⟨*proof*⟩
lemma *Sign-EphK*: $Sign\ X\ Y \neq EphK\ \ X'$ ⟨*proof*⟩
lemma *Sign-LtK*: $Sign\ X\ Y \neq LtK\ \ X'$ ⟨*proof*⟩
lemma *Sign-Pair*: $Sign\ X\ Y \neq Pair\ \ X'\ Y'$ ⟨*proof*⟩
lemma *Sign-Enc*: $Sign\ X\ Y \neq Enc\ \ X'\ Y'$ ⟨*proof*⟩
lemma *Sign-Aenc*: $Sign\ X\ Y \neq Aenc\ \ X'\ Y'$ ⟨*proof*⟩
lemma *Sign-Sign*: $(Sign\ X\ Y = Sign\ X'\ Y') = (X = X' \wedge\ Y = Y')$ ⟨*proof*⟩

lemma *Sign-Exp*: *Sign X Y* $\neq$ *Exp X' Y'* ⟨*proof*⟩

lemma *Exp-Nonce*: *Exp X Y* $\neq$ *Nonce X'* ⟨*proof*⟩
lemma *Exp-Agent*: *Exp X Y* $\neq$ *Agent X'* ⟨*proof*⟩
lemma *Exp-Number*: *Exp X Y* $\neq$ *Number X'* ⟨*proof*⟩
lemma *Exp-Hash*: *Exp X Y* $\neq$ *Hash X'* ⟨*proof*⟩
lemma *Exp-Tag*: *Exp X Y* $\neq$ *Tag X'* ⟨*proof*⟩
lemma *Exp-EphK*: *Exp X Y* $\neq$ *EphK X'* ⟨*proof*⟩
lemma *Exp-LtK*: *Exp X Y* $\neq$ *LtK X'* ⟨*proof*⟩
lemma *Exp-Pair*: *Exp X Y* $\neq$ *Pair X' Y'* ⟨*proof*⟩
lemma *Exp-Enc*: *Exp X Y* $\neq$ *Enc X' Y'* ⟨*proof*⟩
lemma *Exp-Aenc*: *Exp X Y* $\neq$ *Aenc X' Y'* ⟨*proof*⟩
lemma *Exp-Sign*: *Exp X Y* $\neq$ *Sign X' Y'* ⟨*proof*⟩

lemmas *msg-inject* [*iff*, *induct-simp*] =
Nonce-Nonce Agent-Agent Number-Number Hash-Hash Tag-Tag EphK-EphK LtK-LtK
Pair-Pair Enc-Enc Aenc-Aenc Sign-Sign

lemmas *msg-distinct* [*simp*, *induct-simp*] =
Nonce-Agent Nonce-Number Nonce-Hash Nonce-Tag Nonce-EphK Nonce-LtK Nonce-Pair
Nonce-Enc Nonce-Aenc Nonce-Sign Nonce-Exp
Agent-Nonce Agent-Number Agent-Hash Agent-Tag Agent-EphK Agent-LtK Agent-Pair
Agent-Enc Agent-Aenc Agent-Sign Agent-Exp
Number-Nonce Number-Agent Number-Hash Number-Tag Number-EphK Number-LtK
Number-Pair Number-Enc Number-Aenc Number-Sign Number-Exp
Hash-Nonce Hash-Agent Hash-Number Hash-Tag Hash-EphK Hash-LtK Hash-Pair
Hash-Enc Hash-Aenc Hash-Sign Hash-Exp
Tag-Nonce Tag-Agent Tag-Number Tag-Hash Tag-EphK Tag-LtK Tag-Pair
Tag-Enc Tag-Aenc Tag-Sign Tag-Exp
EphK-Nonce EphK-Agent EphK-Number EphK-Hash EphK-Tag EphK-LtK EphK-Pair
EphK-Enc EphK-Aenc EphK-Sign EphK-Exp
LtK-Nonce LtK-Agent LtK-Number LtK-Hash LtK-Tag LtK-EphK LtK-Pair
LtK-Enc LtK-Aenc LtK-Sign LtK-Exp
Pair-Nonce Pair-Agent Pair-Number Pair-Hash Pair-Tag Pair-EphK Pair-LtK
Pair-Enc Pair-Aenc Pair-Sign Pair-Exp
Enc-Nonce Enc-Agent Enc-Number Enc-Hash Enc-Tag Enc-EphK Enc-LtK Enc-Pair
Enc-Aenc Enc-Sign Enc-Exp
Aenc-Nonce Aenc-Agent Aenc-Number Aenc-Hash Aenc-Tag Aenc-EphK Aenc-LtK
Aenc-Pair Aenc-Enc Aenc-Sign Aenc-Exp
Sign-Nonce Sign-Agent Sign-Number Sign-Hash Sign-Tag Sign-EphK Sign-LtK
Sign-Pair Sign-Enc Sign-Aenc Sign-Exp
Exp-Nonce Exp-Agent Exp-Number Exp-Hash Exp-Tag Exp-EphK Exp-LtK Exp-Pair
Exp-Enc Exp-Aenc Exp-Sign

consts *Ngen* :: *nat*
abbreviation *Gen* ≡ *Number Ngen*
abbreviation *cGen* ≡ *cNumber Ngen*

abbreviation
InsecTag ≡ *Tag insec*

abbreviation
AuthTag ≡ *Tag auth*

abbreviation
ConfidTag ≡ *Tag confid*

abbreviation
SecureTag ≡ *Tag secure*

abbreviation
Tags ≡ *range Tag*

abbreviation
NonceF :: *fresh-t* ⇒ *msg* **where**
NonceF N ≡ *Nonce* (*nonce-fresh N*)

abbreviation
NonceA :: *nat* ⇒ *msg* **where**
NonceA N ≡ *Nonce* (*nonce-atk N*)

abbreviation
shrK :: *agent* ⇒ *agent* ⇒ *msg* **where**
shrK A B ≡ *LtK* (*sharK A B*)

abbreviation
pubK :: *agent* ⇒ *msg* **where**
pubK A ≡ *LtK* (*publK A*)

abbreviation
priK :: *agent* ⇒ *msg* **where**
priK A ≡ *LtK* (*privK A*)

abbreviation
epubK :: *nonce-t* ⇒ *msg* **where**
epubK N ≡ *EphK* (*epublK N*)

abbreviation
epriK :: *nonce-t* ⇒ *msg* **where**
epriK N ≡ *EphK* (*eprivK N*)

abbreviation
epubKF :: *fresh-t* ⇒ *msg* **where**
epubKF N ≡ *EphK* (*epublK* (*nonce-fresh N*))

abbreviation
epriKF :: *fresh-t* ⇒ *msg* **where**
epriKF N ≡ *EphK* (*eprivK* (*nonce-fresh N*))

abbreviation
epubKA :: *nat* ⇒ *msg* **where**
epubKA N ≡ *EphK* (*epublK* (*nonce-atk N*))

abbreviation

epriKA :: *nat* ⇒ *msg* **where**
epriKA N ≡ *EphK* (*eprivK* (*nonce-atk N*))

Concrete syntax: messages appear as <A,B,NA>, etc...

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

hash macs

abbreviation
hmac :: *msg* ⇒ *msg* ⇒ *msg* **where**
hmac M K ≡ *Hash* ⟨*M*, *K*⟩

recover some kind of injectivity for Exp

lemma *eq-expgen*:
$eq\ X\ Y \Longrightarrow (\forall\ X'.\ X = cExp\ cGen\ X' \longrightarrow (\exists\ Z.\ Y = (cExp\ cGen\ Z) \wedge eq\ X'\ Z)) \wedge$
$(\forall\ Y'.\ Y = cExp\ cGen\ Y' \longrightarrow (\exists\ Z.\ X = (cExp\ cGen\ Z) \wedge eq\ Y'\ Z))$
⟨*proof*⟩

lemma *Exp-Gen-inj*: $Exp\ Gen\ X = Exp\ Gen\ Y \Longrightarrow X = Y$
⟨*proof*⟩

lemma *eq-expexpgen*:
$eq\ X\ Y \Longrightarrow (\forall\ X'\ X''.\ X = cExp\ (cExp\ cGen\ X')\ X'' \longrightarrow$
$(\exists\ Y'\ Y''.\ Y = cExp\ (cExp\ cGen\ Y')\ Y'' \wedge$
$((eq\ X'\ Y' \wedge eq\ X''\ Y'') \vee (eq\ X'\ Y'' \wedge eq\ X''\ Y'))))$
⟨*proof*⟩

lemma *Exp-Exp-Gen-inj*:
$Exp\ (Exp\ Gen\ X)\ X' = Z \Longrightarrow$
$(\exists\ Y\ Y'.\ Z = Exp\ (Exp\ Gen\ Y)\ Y' \wedge ((X = Y \wedge X' = Y') \vee (X = Y' \wedge X' = Y)))$
⟨*proof*⟩

lemma *Exp-Exp-Gen-inj2*:
$Exp\ (Exp\ Gen\ X)\ X' = Exp\ Z\ Y' \Longrightarrow$
$(Y' = X \wedge Z = Exp\ Gen\ X') \vee (Y' = X' \wedge Z = Exp\ Gen\ X)$
⟨*proof*⟩

end

4 Message theory

theory *Message-derivation*
imports *Messages*
begin

This theory is adapted from Larry Paulson's original Message theory.

4.1 Message composition

Dolev-Yao message synthesis.

inductive-set
  synth :: *msg set* ⇒ *msg set*
  for *H* :: *msg set*
  where
    Ax [*intro*]: $X \in H \Longrightarrow X \in synth\ H$
  | *Agent* [*simp*, *intro*]: $Agent\ A \in synth\ H$
  | *Number* [*simp*, *intro*]: $Number\ n \in synth\ H$
  | *NonceA* [*simp*, *intro*]: $NonceA\ n \in synth\ H$
  | *EpubKA* [*simp*, *intro*]: $epubKA\ n \in synth\ H$
  | *EpriKA* [*simp*, *intro*]: $epriKA\ n \in synth\ H$
  | *Hash* [*intro*]: $X \in synth\ H \Longrightarrow Hash\ X \in synth\ H$
  | *Pair* [*intro*]: $X \in synth\ H \Longrightarrow Y \in synth\ H \Longrightarrow (Pair\ X\ Y) \in synth\ H$
  | *Enc* [*intro*]: $X \in synth\ H \Longrightarrow Y \in synth\ H \Longrightarrow (Enc\ X\ Y) \in synth\ H$
  | *Aenc* [*intro*]: $X \in synth\ H \Longrightarrow Y \in synth\ H \Longrightarrow (Aenc\ X\ Y) \in synth\ H$
  | *Sign* [*intro*]: $X \in synth\ H \Longrightarrow Y \in synth\ H \Longrightarrow Sign\ X\ Y \in synth\ H$
  | *Exp* [*intro*]: $X \in synth\ H \Longrightarrow Y \in synth\ H \Longrightarrow (Exp\ X\ Y) \in synth\ H$

Lemmas about Dolev-Yao message synthesis.

lemma *synth-mono* [*mono-set*]: $G \subseteq H \Longrightarrow synth\ G \subseteq synth\ H$
  ⟨*proof*⟩

lemmas *synth-monotone* = *synth-mono* [*THEN* [*2*] *rev-subsetD*]

— [*elim*!] slows down certain proofs, e.g., $[\![\ synth\ H \cap B \subseteq \{\}\ ]\!] \Longrightarrow P$
inductive-cases *NonceF-synth*: $NonceF\ n \in synth\ H$
inductive-cases *LtK-synth*: $LtK\ K \in synth\ H$
inductive-cases *EpubKF-synth*: $epubKF\ K \in synth\ H$
inductive-cases *EpriKF-synth*: $epriKF\ K \in synth\ H$
inductive-cases *Hash-synth*: $Hash\ X \in synth\ H$
inductive-cases *Pair-synth*: $Pair\ X\ Y \in synth\ H$
inductive-cases *Enc-synth*: $Enc\ X\ K \in synth\ H$
inductive-cases *Aenc-synth*: $Aenc\ X\ K \in synth\ H$
inductive-cases *Sign-synth*: $Sign\ X\ K \in synth\ H$
inductive-cases *Tag-synth*: $Tag\ t \in synth\ H$

lemma *EpriK-synth* [*elim*]: $epriK\ K \in synth\ H \Longrightarrow$
    $epriK\ K \in H \vee (\exists\ N.\ epriK\ K = epriKA\ N)$
⟨*proof*⟩

lemma *EpubK-synth* [*elim*]: $epubK\ K \in synth\ H \Longrightarrow$
    $epubK\ K \in H \vee (\exists\ N.\ epubK\ K = epubKA\ N)$

⟨*proof*⟩

lemmas *synth-inversion* [*elim*] =
NonceF-synth LtK-synth EpubKF-synth EpriKF-synth Hash-synth Pair-synth Enc-synth Aenc-synth Sign-synth Tag-synth

lemma *synth-increasing*: $H \subseteq synth\ H$
⟨*proof*⟩

lemma *synth-Int1*: $x \in synth\ (A \cap B) \Longrightarrow x \in synth\ A$
⟨*proof*⟩

lemma *synth-Int2*: $x \in synth\ (A \cap B) \Longrightarrow x \in synth\ B$
⟨*proof*⟩

lemma *synth-Int*: $x \in synth\ (A \cap B) \Longrightarrow x \in synth\ A \cap synth\ B$
⟨*proof*⟩

lemma *synth-Un*: $synth\ G \cup synth\ H \subseteq synth\ (G \cup H)$
⟨*proof*⟩

lemma *synth-insert*: $insert\ X\ (synth\ H) \subseteq synth\ (insert\ X\ H)$
⟨*proof*⟩

lemma *synth-synthD* [*dest!*]: $X \in synth\ (synth\ H) \Longrightarrow X \in synth\ H$
⟨*proof*⟩

lemma *synth-idem* [*simp*]: $synth\ (synth\ H) = synth\ H$
⟨*proof*⟩

lemma *synth-subset-iff*: $synth\ G \subseteq synth\ H \longleftrightarrow G \subseteq synth\ H$
⟨*proof*⟩

lemma *synth-trans*: $[\![\ X \in synth\ G;\ G \subseteq synth\ H\ ]\!] \Longrightarrow X \in synth\ H$
⟨*proof*⟩

lemma *synth-cut*: $[\![\ Y \in synth\ (insert\ X\ H);\ X \in synth\ H\ ]\!] \Longrightarrow Y \in synth\ H$
⟨*proof*⟩

lemma *Nonce-synth-eq* [*simp*]: $(NonceF\ N \in synth\ H) = (NonceF\ N \in H)$
⟨*proof*⟩

lemma *LtK-synth-eq* [*simp*]: $(LtK\ K \in synth\ H) = (LtK\ K \in H)$
⟨*proof*⟩

lemma *EpubKF-synth-eq* [*simp*]: $(epubKF\ K \in synth\ H) = (epubKF\ K \in H)$
⟨*proof*⟩

lemma *EpriKF-synth-eq* [*simp*]: $(epriKF\ K \in synth\ H) = (epriKF\ K \in H)$
⟨*proof*⟩

lemma *Enc-synth-eq1* [*simp*]:
$K \notin synth\ H \Longrightarrow (Enc\ X\ K \in synth\ H) = (Enc\ X\ K \in H)$
⟨*proof*⟩

lemma *Enc-synth-eq2* [*simp*]:
$X \notin synth\ H \Longrightarrow (Enc\ X\ K \in synth\ H) = (Enc\ X\ K \in H)$
⟨*proof*⟩

lemma *Aenc-synth-eq1* [*simp*]:
$K \notin synth\ H \Longrightarrow (Aenc\ X\ K \in synth\ H) = (Aenc\ X\ K \in H)$
⟨*proof*⟩

lemma *Aenc-synth-eq2* [*simp*]:
$X \notin synth\ H \Longrightarrow (Aenc\ X\ K \in synth\ H) = (Aenc\ X\ K \in H)$
⟨*proof*⟩

lemma *Sign-synth-eq1* [*simp*]:
$K \notin synth\ H \Longrightarrow (Sign\ X\ K \in synth\ H) = (Sign\ X\ K \in H)$
⟨*proof*⟩

lemma *Sign-synth-eq2* [*simp*]:
$X \notin synth\ H \Longrightarrow (Sign\ X\ K \in synth\ H) = (Sign\ X\ K \in H)$
⟨*proof*⟩

4.2 Message decomposition

Dolev-Yao message decomposition using known keys.

inductive-set
analz :: *msg set* ⇒ *msg set*
for *H* :: *msg set*
where
Ax [*intro*]: $X \in H \Longrightarrow X \in analz\ H$
| *Fst*: $Pair\ X\ Y \in analz\ H \Longrightarrow X \in analz\ H$
| *Snd*: $Pair\ X\ Y \in analz\ H \Longrightarrow Y \in analz\ H$
| *Dec* [*dest*]:
$[\![Enc\ X\ Y \in analz\ H;\ Y \in synth\ (analz\ H)]\!] \Longrightarrow X \in analz\ H$
| *Adec-lt* [*dest*]:
$[\![Aenc\ X\ (LtK\ (publK\ Y)) \in analz\ H;\ priK\ Y \in analz\ H]\!] \Longrightarrow X \in analz\ H$
| *Adec-eph* [*dest*]:
$[\![Aenc\ X\ (EphK\ (epublK\ Y)) \in analz\ H;\ epriK\ Y \in synth\ (analz\ H)]\!] \Longrightarrow X \in analz\ H$
| *Sign-getmsg* [*dest*]:
$Sign\ X\ (priK\ Y) \in analz\ H \Longrightarrow pubK\ Y \in analz\ H \Longrightarrow X \in analz\ H$

Lemmas about Dolev-Yao message decomposition.

lemma *analz-mono*: $G \subseteq H \Longrightarrow analz(G) \subseteq analz(H)$
⟨*proof*⟩

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

lemma *Pair-analz* [*elim!*]:
$[\![Pair\ X\ Y \in analz\ H;\ [\![X \in analz\ H;\ Y \in analz\ H]\!] \Longrightarrow P]\!] \Longrightarrow P$

⟨*proof*⟩

lemma *analz-empty* [*simp*]: *analz* {} = {}
⟨*proof*⟩

lemma *analz-increasing*: $H \subseteq$ *analz*(H)
⟨*proof*⟩

lemma *analz-analzD* [*dest*!]: $X \in$ *analz* (*analz* H) $\Longrightarrow X \in$ *analz* H
⟨*proof*⟩

lemma *analz-idem* [*simp*]: *analz* (*analz* H) = *analz* H
⟨*proof*⟩

lemma *analz-Un*: *analz* $G \cup$ *analz* $H \subseteq$ *analz* ($G \cup H$)
⟨*proof*⟩

lemma *analz-insertI*: $X \in$ *analz* $H \Longrightarrow X \in$ *analz* (*insert* Y H)
⟨*proof*⟩

lemma *analz-insert*: *insert* X (*analz* H) $\subseteq$ *analz* (*insert* X H)
⟨*proof*⟩

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

lemma *analz-subset-iff* [*simp*]: *analz* $G \subseteq$ *analz* $H \longleftrightarrow G \subseteq$ *analz* H
⟨*proof*⟩

lemma *analz-trans*: $X \in$ *analz* $G \Longrightarrow$ $G \subseteq$ *analz* $H \Longrightarrow X \in$ *analz* H
⟨*proof*⟩

lemma *analz-cut*: $Y \in$ *analz* (*insert* X H) $\Longrightarrow$ $X \in$ *analz* $H \Longrightarrow Y \in$ *analz* H
⟨*proof*⟩

lemma *analz-insert-eq*: $X \in$ *analz* $H \Longrightarrow$ *analz* (*insert* X H) = *analz* H
⟨*proof*⟩

lemma *analz-subset-cong*:
analz $G \subseteq$ *analz* $G' \Longrightarrow$
analz $H \subseteq$ *analz* $H' \Longrightarrow$
analz ($G \cup H$) $\subseteq$ *analz* ($G' \cup H'$)
⟨*proof*⟩

lemma *analz-cong*:
analz G = *analz* $G' \Longrightarrow$
analz H = *analz* $H' \Longrightarrow$
analz ($G \cup H$) = *analz* ($G' \cup H'$)
⟨*proof*⟩

lemma *analz-insert-cong*:
analz H = *analz* $H' \Longrightarrow$
analz (*insert* X H) = *analz* (*insert* X H')
⟨*proof*⟩

lemma *analz-trivial*:
$\forall X\ Y$. *Pair* $X\ Y \notin H \Longrightarrow$
$\forall X\ Y$. *Enc* $X\ Y \notin H \Longrightarrow$
$\forall X\ Y$. *Aenc* $X\ Y \notin H \Longrightarrow$
$\forall X\ Y$. *Sign* $X\ Y \notin H \Longrightarrow$
analz $H = H$
⟨*proof*⟩

lemma *analz-analz-Un* [*simp*]: *analz* (*analz* $G \cup H$) = *analz* ($G \cup H$)
⟨*proof*⟩

lemma *analz-Un-analz* [*simp*]: *analz* ($G \cup$ *analz* H) = *analz* ($G \cup H$)
⟨*proof*⟩

Lemmas about analz and insert.

lemma *analz-insert-Agent* [*simp*]:
analz (*insert* (*Agent* A) H) = *insert* (*Agent* A) (*analz* H)
⟨*proof*⟩

4.3 Lemmas about combined composition/decomposition

lemma *synth-analz-incr*: $H \subseteq$ *synth* (*analz* H)
⟨*proof*⟩

lemmas *synth-analz-increasing* = *synth-analz-incr* [*THEN* [*2*] *rev-subsetD*]

lemma *synth-analz-mono*: $G \subseteq H \Longrightarrow$ *synth* (*analz* G) $\subseteq$ *synth* (*analz* H)
⟨*proof*⟩

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

lemma *lem1*:
$Y \in$ *synth* (*analz* (*synth* $G \cup H$) $\cap$ (*analz* ($G \cup H$) $\cup$ *synth* G))
$\Longrightarrow Y \in$ *synth* (*analz* ($G \cup H$))
⟨*proof*⟩

lemma *lem2*: $\{a.\ a \in$ *analz* ($G \cup H$) $\vee\ a \in$ *synth* $G\}$ = *analz* ($G \cup H$) $\cup$ *synth* G ⟨*proof*⟩

lemma *analz-synth-Un*: *analz* (*synth* $G \cup H$) = *analz* ($G \cup H$) $\cup$ *synth* G
⟨*proof*⟩

lemma *analz-synth*: *analz* (*synth* H) = *analz* $H \cup$ *synth* H
⟨*proof*⟩

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

⟨*proof*⟩

lemma *synth-analz-synth* [*simp*]: *synth* (*analz* (*synth* *H*)) = *synth* (*analz* *H*)
⟨*proof*⟩

lemma *analz-synth-analz* [*simp*]: *analz* (*synth* (*analz* *H*)) = *synth* (*analz* *H*)
⟨*proof*⟩

lemma *synth-analz-idem* [*simp*]: *synth* (*analz* (*synth* (*analz* *H*))) = *synth* (*analz* *H*)
⟨*proof*⟩

lemma *insert-subset-synth-analz* [*simp*]:
  $X \in$ *synth* (*analz* *H*) $\Longrightarrow$ *insert* *X* *H* $\subseteq$ *synth* (*analz* *H*)
⟨*proof*⟩

lemma *synth-analz-insert* [*simp*]:
  assumes $X \in$ *synth* (*analz* *H*)
  shows *synth* (*analz* (*insert* *X* *H*)) = *synth* (*analz* *H*)
⟨*proof*⟩

4.4 Accessible message parts

Accessible message parts: all subterms that are in principle extractable by the Dolev-Yao attacker, i.e., provided he knows all keys. Note that keys in key positions and messages under hashes are not message parts in this sense.

inductive-set
  parts :: *msg set* => *msg set*
  for *H* :: *msg set*
where
    Inj [*intro*]: $X \in H \Longrightarrow X \in$ *parts* *H*
  | *Fst* [*intro*]: *Pair* *X* *Y* $\in$ *parts* *H* $\Longrightarrow X \in$ *parts* *H*
  | *Snd* [*intro*]: *Pair* *X* *Y* $\in$ *parts* *H* $\Longrightarrow Y \in$ *parts* *H*
  | *Dec* [*intro*]: *Enc* *X* *Y* $\in$ *parts* *H* $\Longrightarrow X \in$ *parts* *H*
  | *Adec* [*intro*]: *Aenc* *X* *Y* $\in$ *parts* *H* $\Longrightarrow X \in$ *parts* *H*
  | *Sign-getmsg* [*intro*]: *Sign* *X* *Y* $\in$ *parts* *H* $\Longrightarrow X \in$ *parts* *H*

Lemmas about accessible message parts.

lemma *parts-mono* [*mono-set*]: $G \subseteq H \Longrightarrow$ *parts* *G* $\subseteq$ *parts* *H*
⟨*proof*⟩

lemmas *parts-monotone* = *parts-mono* [*THEN* [*2*] *rev-subsetD*]

lemma *Pair-parts* [*elim*]:
  ⟦ *Pair* *X* *Y* $\in$ *parts* *H*; ⟦ $X \in$ *parts* *H*; $Y \in$ *parts* *H* ⟧ $\Longrightarrow P$ ⟧ $\Longrightarrow P$
⟨*proof*⟩

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

⟨proof⟩

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

lemma *parts-empty* [*simp*]: *parts* {} = {}
⟨proof⟩

lemma *parts-atomic* [*simp*]: *atomic x* ⟹ *parts* {*x*} = {*x*}
⟨proof⟩

lemma *parts-InsecTag* [*simp*]: *parts* {*Tag t*} = {*Tag t*}
⟨proof⟩

lemma *parts-emptyE* [*elim*!]: *X* ∈ *parts* {} ⟹ *P*
⟨proof⟩

lemma *parts-Tags* [*simp*]:
  parts Tags = *Tags*
⟨proof⟩

lemma *parts-singleton*: *X* ∈ *parts H* ⟹ ∃ *Y*∈*H*. *X* ∈ *parts* {*Y*}
⟨proof⟩

lemma *parts-Agents* [*simp*]:
  parts (*Agent*‘ *G*) = *Agent*‘ *G*
⟨proof⟩

lemma *parts-Un* [*simp*]: *parts* (*G* ∪ *H*) = *parts G* ∪ *parts H*
⟨proof⟩

lemma *parts-insert-subset-Un*:
  assumes *X* ∈ *G*
  shows *parts* (*insert X H*) ⊆ *parts G* ∪ *parts H*
⟨proof⟩

lemma *parts-insert*: *parts* (*insert X H*) = *parts* {*X*} ∪ *parts H*
⟨proof⟩

lemma *parts-insert2*:
  parts (*insert X* (*insert Y H*)) = *parts* {*X*} ∪ *parts* {*Y*} ∪ *parts H*
⟨proof⟩

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

lemma *parts-insert-subset*: *insert X* (*parts H*) ⊆ *parts* (*insert X H*)
⟨proof⟩

lemma *parts-partsD* [*dest*!]: *X* ∈ *parts* (*parts H*) ⟹ *X* ∈ *parts H*
⟨proof⟩

lemma *parts-idem* [*simp*]: *parts* (*parts* *H*) = *parts* *H*
⟨*proof*⟩

lemma *parts-subset-iff* [*simp*]: (*parts* *G* ⊆ *parts* *H*) ⟷ (*G* ⊆ *parts* *H*)
⟨*proof*⟩

lemma *parts-trans*: *X* ∈ *parts* *G* ⟹ *G* ⊆ *parts* *H* ⟹ *X* ∈ *parts* *H*
⟨*proof*⟩

lemma *parts-cut*:
Y ∈ *parts* (*insert* *X* *G*) ⟹ *X* ∈ *parts* *H* ⟹ *Y* ∈ *parts* (*G* ∪ *H*)
⟨*proof*⟩

lemma *parts-cut-eq* [*simp*]: *X* ∈ *parts* *H* ⟹ *parts* (*insert* *X* *H*) = *parts* *H*
⟨*proof*⟩

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

lemma *parts-insert-Agent* [*simp*]:
parts (*insert* (*Agent* *agt*) *H*) = *insert* (*Agent* *agt*) (*parts* *H*)
⟨*proof*⟩

lemma *parts-insert-Nonce* [*simp*]:
parts (*insert* (*Nonce* *N*) *H*) = *insert* (*Nonce* *N*) (*parts* *H*)
⟨*proof*⟩

lemma *parts-insert-Number* [*simp*]:
parts (*insert* (*Number* *N*) *H*) = *insert* (*Number* *N*) (*parts* *H*)
⟨*proof*⟩

lemma *parts-insert-LtK* [*simp*]:
parts (*insert* (*LtK* *K*) *H*) = *insert* (*LtK* *K*) (*parts* *H*)
⟨*proof*⟩

lemma *parts-insert-Hash* [*simp*]:
parts (*insert* (*Hash* *X*) *H*) = *insert* (*Hash* *X*) (*parts* *H*)
⟨*proof*⟩

lemma *parts-insert-Enc* [*simp*]:
parts (*insert* (*Enc* *X* *Y*) *H*) = *insert* (*Enc* *X* *Y*) (*parts* {*X*} ∪ *parts* *H*)
⟨*proof*⟩

lemma *parts-insert-Aenc* [*simp*]:
parts (*insert* (*Aenc* *X* *Y*) *H*) = *insert* (*Aenc* *X* *Y*) (*parts* {*X*} ∪ *parts* *H*)
⟨*proof*⟩

lemma *parts-insert-Sign* [*simp*]:
parts (*insert* (*Sign* *X* *Y*) *H*) = *insert* (*Sign* *X* *Y*) (*parts* {*X*} ∪ *parts* *H*)

⟨*proof*⟩

lemma *parts-insert-Pair* [*simp*]:
$parts\ (insert\ (Pair\ X\ Y)\ H) = insert\ (Pair\ X\ Y)\ (parts\ \{X\} \cup parts\ \{Y\} \cup parts\ H)$
⟨*proof*⟩

4.4.1 Lemmas about combinations with composition and decomposition

lemma *analz-subset-parts*: $analz\ H \subseteq parts\ H$
⟨*proof*⟩

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

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

lemma *parts-analz* [*simp*]: $parts\ (analz\ H) = parts\ H$
⟨*proof*⟩

lemma *analz-parts* [*simp*]: $analz\ (parts\ H) = parts\ H$
⟨*proof*⟩

lemma *parts-synth* [*simp*]: $parts\ (synth\ H) = parts\ H \cup synth\ H$
⟨*proof*⟩

lemma *Fake-parts-insert*:
$X \in synth\ (analz\ H) \Longrightarrow parts\ (insert\ X\ H) \subseteq synth\ (analz\ H) \cup parts\ H$
⟨*proof*⟩

lemma *Fake-parts-insert-in-Un*:
$Z \in parts\ (insert\ X\ H) \Longrightarrow$
$X \in synth\ (analz\ H) \Longrightarrow$
$Z \in synth\ (analz\ H) \cup parts\ H$
⟨*proof*⟩

lemma *analz-conj-parts* [*simp*]:
$X \in analz\ H \wedge X \in parts\ H \longleftrightarrow X \in analz\ H$
⟨*proof*⟩

lemma *analz-disj-parts* [*simp*]:
$X \in analz\ H \vee X \in parts\ H \longleftrightarrow X \in parts\ H$
⟨*proof*⟩

4.5 More lemmas about combinations of closures

Combinations of *synth* and *analz*.

lemma *Pair-synth-analz* [*simp*]:
$Pair\ X\ Y \in synth\ (analz\ H) \longleftrightarrow X \in synth\ (analz\ H) \wedge Y \in synth\ (analz\ H)$
⟨*proof*⟩

lemma *Enc-synth-analz*:
$Y \in synth\ (analz\ H) \Longrightarrow$
$(Enc\ X\ Y \in synth\ (analz\ H)) \longleftrightarrow (X \in synth\ (analz\ H))$
⟨*proof*⟩

lemma *Hash-synth-analz* [*simp*]:
$X \notin synth\ (analz\ H) \Longrightarrow$
$(Hash\ (Pair\ X\ Y) \in synth\ (analz\ H)) \longleftrightarrow (Hash\ (Pair\ X\ Y) \in analz\ H)$
⟨*proof*⟩

lemma *gen-analz-insert-eq*:
$[\![\ X \in analz\ G;\ G \subseteq H\ ]\!] \Longrightarrow analz\ (insert\ X\ H) = analz\ H$
⟨*proof*⟩

lemma *synth-analz-insert-eq*:
$[\![\ X \in synth\ (analz\ G);\ G \subseteq H\ ]\!] \Longrightarrow synth\ (analz\ (insert\ X\ H)) = synth\ (analz\ H)$
⟨*proof*⟩

lemma *Fake-parts-sing*:
$X \in synth\ (analz\ H) \Longrightarrow parts\ \{X\} \subseteq synth\ (analz\ H) \cup parts\ H$
⟨*proof*⟩

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

lemma *analz-hash-nonce* [*simp*]:
$analz\ \{M.\ \exists N.\ M = Hash\ (Nonce\ N)\} = \{M.\ \exists N.\ M = Hash\ (Nonce\ N)\}$
⟨*proof*⟩

lemma *synth-analz-hash-nonce* [*simp*]:
$NonceF\ N \notin synth\ (analz\ \{M.\ \exists N.\ M = Hash\ (Nonce\ N)\})$
⟨*proof*⟩

lemma *synth-analz-idem-mono*:
$S \subseteq synth\ (analz\ S') \Longrightarrow synth\ (analz\ S) \subseteq synth\ (analz\ S')$
⟨*proof*⟩

lemmas *synth-analz-idem-monoI* =
synth-analz-idem-mono [*THEN* [*2*] *rev-subsetD*]

lemma *analz-synth-subset*:
$analz\ X \subseteq synth\ (analz\ X') \Longrightarrow$
$analz\ Y \subseteq synth\ (analz\ Y') \Longrightarrow$
$analz\ (X \cup Y) \subseteq synth\ (analz\ (X' \cup Y'))$
⟨*proof*⟩

lemma *analz-synth-subset-Un1* :
$analz\ X \subseteq synth\ (analz\ X') \Longrightarrow analz\ (X \cup Y) \subseteq synth\ (analz\ (X' \cup Y))$
⟨*proof*⟩

lemma *analz-synth-subset-Un2* :
  $analz\ X \subseteq synth\ (analz\ X') \Longrightarrow analz\ (Y \cup X) \subseteq synth\ (analz\ (Y \cup X'))$
⟨*proof*⟩

lemma *analz-synth-insert*:
  $analz\ X \subseteq synth\ (analz\ X') \Longrightarrow analz\ (insert\ Y\ X) \subseteq synth\ (analz\ (insert\ Y\ X'))$
⟨*proof*⟩

lemma *Fake-analz-insert-Un*:
  assumes $Y \in analz\ (insert\ X\ H)$ **and** $X \in synth\ (analz\ G)$
  shows $Y \in synth\ (analz\ G) \cup analz\ (G \cup H)$
⟨*proof*⟩

end

5 Environment: Dolev-Yao Intruder

theory *IK*
imports *Message-derivation*
begin

Basic state contains intruder knowledge. The secrecy model and concrete Level 1 states will be record extensions of this state.

record *ik-state* =
  ik :: *msg set*

Dolev-Yao intruder event adds a derived message.

definition
  ik-dy :: *msg* ⇒ ('*a ik-state-scheme* ∗ '*a ik-state-scheme*) *set*
where
  ik-dy m ≡ {(*s*, *s'*).
    — guard
    $m \in synth\ (analz\ (ik\ s)) \wedge$

    — action
    $s' = s\ (\!|ik := ik\ s \cup \{m\}|\!)$
  }

definition
  ik-trans :: ('*a ik-state-scheme* ∗ '*a ik-state-scheme*) *set*
where
  $ik\text{-}trans \equiv (\bigcup\ m.\ \ ik\text{-}dy\ m)$

lemmas *ik-trans-defs* = *ik-trans-def ik-dy-def*

lemma *ik-trans-ik-increasing*: $(s, s') \in ik\text{-}trans \Longrightarrow ik\ s \subseteq ik\ s'$
⟨*proof*⟩

lemma *ik-trans-synth-analz-ik-increasing*:
  $(s, s') \in ik\text{-}trans \Longrightarrow synth\ (analz\ (ik\ s)) \subseteq synth\ (analz\ (ik\ s'))$
⟨*proof*⟩

end

6 Secrecy Model (L0)

theory *Secrecy*
imports *Refinement IK*
begin

declare *domIff* [*simp*, *iff del*]

6.1 State and events

Level 0 secrecy state: extend intruder knowledge with set of secrets.

record *s0-state* = *ik-state* +
  secret :: *msg set*

Definition of the secrecy invariant: DY closure of intruder knowledge and set of secrets are disjoint.

definition
  s0-secrecy :: *'a s0-state-scheme set*
where
  s0-secrecy ≡ {*s*. *synth* (*analz* (*ik s*)) ∩ *secret s* = {}}

lemmas *s0-secrecyI* = *s0-secrecy-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *s0-secrecyE* [*elim*] = *s0-secrecy-def* [*THEN setc-def-to-elim*, *rule-format*]

Two events: add/declare a message as a secret and learn a (non-secret) message.

definition
  s0-add-secret :: *msg* ⇒ (*'a s0-state-scheme* ∗ *'a s0-state-scheme*) *set*
where
  s0-add-secret m ≡ {(*s*,*s'*).
    — guard
    m ∉ *synth* (*analz* (*ik s*)) ∧

    — action
    s' = *s*(|*secret* := *insert m* (*secret s*)|)
  }

definition
  s0-learn :: *msg* ⇒ (*'a s0-state-scheme* ∗ *'a s0-state-scheme*) *set*
where
  s0-learn m ≡ {(*s*,*s'*).
    — guard
    s(|*ik* := *insert m* (*ik s*)|) ∈ *s0-secrecy* ∧

    — action
    s' = *s*(|*ik* := *insert m* (*ik s*)|)
  }

definition
  s0-learn' :: *msg* ⇒ (*'a s0-state-scheme* ∗ *'a s0-state-scheme*) *set*
where
  s0-learn' m ≡ {(*s*,*s'*).
    — guard

$synth\ (analz\ (insert\ m\ (ik\ s))) \cap secret\ s = \{\} \wedge$

— action
$s' = s(\!|ik := insert\ m\ (ik\ s)|\!)$
}

definition
s0-trans :: ('a s0-state-scheme ∗ 'a s0-state-scheme) set
where
$s0\text{-}trans \equiv (\bigcup m.\ s0\text{-}add\text{-}secret\ m) \cup (\bigcup m.\ s0\text{-}learn\ m) \cup Id$

Initial state is any state satisfying the invariant. The whole state is observable. Put all together to define the L0 specification.

definition
s0-init :: 'a s0-state-scheme set
where
s0-init ≡ *s0-secrecy*

type-synonym
s0-obs = *s0-state*

definition
s0 :: (*s0-state*, *s0-obs*) *spec* **where**
s0 ≡ (|
init = *s0-init*,
trans = *s0-trans*,
obs = *id*
|)

lemmas *s0-defs* = *s0-def s0-init-def s0-trans-def s0-add-secret-def s0-learn-def*
lemmas *s0-all-defs* = *s0-defs ik-trans-defs*

lemma *s0-obs-id* [*simp*]: *obs s0* = *id*
⟨*proof*⟩

6.2 Proof of secrecy invariant

lemma *s0-secrecy-init* [*iff*]: *init s0* ⊆ *s0-secrecy*
⟨*proof*⟩

lemma *s0-secrecy-trans* [*simp*, *intro*]: {*s0-secrecy*} *trans s0* {> *s0-secrecy*}
⟨*proof*⟩

lemma *s0-secrecy* [*iff*]:*reach s0* ⊆ *s0-secrecy*
⟨*proof*⟩

lemma *s0-obs-secrecy* [*iff*]:*oreach s0* ⊆ *s0-secrecy*
⟨*proof*⟩

lemma *s0-anyP-observable* [*iff*]: *observable* (*obs s0*) *P*

⟨*proof*⟩

end

7 Non-injective Agreement (L0)

theory *AuthenticationN* **imports** *Refinement Messages*
begin

declare *domIff* [*simp*, *iff del*]

7.1 Signals

signals

datatype *signal* =
  Running agent agent msg
| *Commit agent agent msg*

fun
  addSignal :: (*signal* ⇒ *nat*) ⇒ *signal* ⇒ *signal* ⇒ *nat*
where
  addSignal sigs s = *sigs* (*s* := *sigs s* + *1*)

7.2 State and events

level 0 non-injective agreement

record *a0n-state* =
  signals :: *signal* ⇒ *nat*   — multi-set of signals

type-synonym
  a0n-obs = *a0n-state*

Events

definition
  a0n-running :: *agent* ⇒ *agent* ⇒ *msg* ⇒ (*a0n-state* × *a0n-state*) *set*
where
  a0n-running A B M ≡ {(*s*,*s'*).
    — action
    s' = *s*(|*signals* := *addSignal* (*signals s*) (*Running A B M*)|)
  }

definition
  a0n-commit :: *agent* ⇒ *agent* ⇒ *msg* ⇒ (*a0n-state* × *a0n-state*) *set*
where
  a0n-commit A B M ≡ {(*s*, *s'*).
  — guard
    signals s (*Running A B M*) > *0* ∧
  — action
    s' = *s*(|*signals* := *addSignal* (*signals s*) (*Commit A B M*)|)
  }

definition
  a0n-trans :: (*a0n-state* × *a0n-state*) *set* **where**
  a0n-trans ≡ (⋃*A B M*. *a0n-running A B M*) ∪ (⋃*A B M*. *a0n-commit A B M*) ∪ *Id*

Level 0 state

definition
a0n-init :: *a0n-state set*
where
a0n-init ≡ {(|*signals* = λ*s*. *0*|)}

definition
a0n :: (*a0n-state*, *a0n-obs*) *spec* **where**
a0n ≡ (|
init = *a0n-init*,
trans = *a0n-trans*,
obs = *id*
|)

lemmas *a0n-defs* =
a0n-def a0n-init-def a0n-trans-def
a0n-running-def a0n-commit-def

lemma *a0n-obs-id* [*simp*]: *obs a0n* = *id*
⟨*proof*⟩

lemma *a0n-anyP-observable* [*iff*]: *observable* (*obs a0n*) *P*
⟨*proof*⟩

7.3 Non injective agreement invariant

Invariant: non injective agreement

definition
a0n-agreement :: *a0n-state set*
where
a0n-agreement ≡ {*s*. ∀ *A B M*.
signals s (*Commit A B M*) > *0* ⟶ *signals s* (*Running A B M*) > *0*
}

lemmas *a0n-agreementI* = *a0n-agreement-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *a0n-agreementE* [*elim*] = *a0n-agreement-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *a0n-agreementD* = *a0n-agreement-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 2*]

lemma *a0n-agreement-init* [*iff*]:
init a0n ⊆ *a0n-agreement*
⟨*proof*⟩

lemma *a0n-agreement-trans* [*iff*]:
{*a0n-agreement*} *trans a0n* {> *a0n-agreement*}
⟨*proof*⟩

lemma *a0n-agreement* [*iff*]: *reach a0n* ⊆ *a0n-agreement*
⟨*proof*⟩

lemma *a0n-obs-agreement* [*iff*]:
  oreach a0n ⊆ *a0n-agreement*
⟨*proof*⟩

end

8 Injective Agreement (L0)

theory *AuthenticationI*
imports *AuthenticationN*
begin

8.1 State and events

type-synonym
  a0i-state = *a0n-state*

type-synonym
  a0i-obs = *a0n-obs*

abbreviation
  a0i-init :: *a0n-state set*
where
  a0i-init ≡ *a0n-init*

abbreviation
  a0i-running :: *agent* ⇒ *agent* ⇒ *msg* ⇒ (*a0i-state* × *a0i-state*) *set*
where
  a0i-running ≡ *a0n-running*

lemmas *a0i-running-def* = *a0n-running-def*

definition
  a0i-commit :: *agent* ⇒ *agent* ⇒ *msg* ⇒ (*a0i-state* × *a0i-state*) *set*
where
  a0i-commit A B M ≡ {(*s*, *s'*).
   — guard
    signals s (*Commit A B M*) < *signals s* (*Running A B M*) ∧
   — actions:
    s' = *s*(|*signals* := *addSignal* (*signals s*) (*Commit A B M*)|)
  }

definition
  a0i-trans :: (*a0i-state* × *a0i-state*) *set* **where**
  a0i-trans ≡ (⋃ *A B M*. *a0i-running A B M*) ∪ (⋃ *A B M*. *a0i-commit A B M*) ∪ *Id*

definition
  a0i :: (*a0i-state*,*a0i-obs*) *spec* **where**
  a0i ≡ (|
   init = *a0i-init*,
   trans = *a0i-trans*,
   obs = *id*
  |)

lemmas *a0i-defs* = *a0n-defs a0i-def a0i-trans-def a0i-commit-def*

lemma *a0i-obs* [*simp*]: *obs a0i* = *id*

⟨*proof*⟩

lemma *a0i-anyP-observable* [*iff*]: *observable* (*obs a0i*) *P*
⟨*proof*⟩

8.2 Injective agreement invariant

definition
a0i-agreement :: *a0i-state set*
where
a0i-agreement ≡ {*s*. ∀ *A B M*.
signals s (*Commit A B M*) ≤ *signals s* (*Running A B M*)
}

lemmas *a0i-agreementI* =
a0i-agreement-def [*THEN setc-def-to-intro*, *rule-format*]
lemmas *a0i-agreementE* [*elim*] =
a0i-agreement-def [*THEN setc-def-to-elim*, *rule-format*]
lemmas *a0i-agreementD* =
a0i-agreement-def [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

lemma *PO-a0i-agreement-init* [*iff*]:
init a0i ⊆ *a0i-agreement*
⟨*proof*⟩

lemma *PO-a0i-agreement-trans* [*iff*]:
{*a0i-agreement*} *trans a0i* {> *a0i-agreement*}
⟨*proof*⟩

lemma *PO-a0i-agreement* [*iff*]: *reach a0i* ⊆ *a0i-agreement*
⟨*proof*⟩

lemma *PO-a0i-obs-agreement* [*iff*]: *oreach a0i* ⊆ *a0i-agreement*
⟨*proof*⟩

8.3 Refinement

definition
med0n0i :: *a0i-obs* ⇒ *a0i-obs*
where
med0n0i ≡ *id*

definition
R0n0i :: (*a0n-state* × *a0i-state*) *set*
where
R0n0i ≡ *Id*

lemma *PO-a0i-running-refines-a0n-running*:
{*R0n0i*} *a0n-running A B M*, *a0i-running A B M* {> *R0n0i*}
⟨*proof*⟩

lemma *PO-a0i-commit-refines-a0n-commit*:
{*R0n0i*} *a0n-commit A B M*, *a0i-commit A B M* {> *R0n0i*}
⟨*proof*⟩

lemmas *PO-a0i-trans-refines-a0n-trans* =
PO-a0i-running-refines-a0n-running
PO-a0i-commit-refines-a0n-commit

lemma *PO-a0i-refines-init-a0n* [*iff*]:
init a0i ⊆ *R0n0i*``(*init a0n*)
⟨*proof*⟩

lemma *PO-a0i-refines-trans-a0n* [*iff*]:
{*R0n0i*} *trans a0n*, *trans a0i* {> *R0n0i*}
⟨*proof*⟩

lemma *PO-obs-consistent* [*iff*]:
obs-consistent R0n0i med0n0i a0n a0i
⟨*proof*⟩

lemma *PO-a0i-refines-a0n*:
refines R0n0i med0n0i a0n a0i
⟨*proof*⟩

8.4 Derived invariant

lemma *iagreement-implies-niagreement* [*iff*]: *a0i-agreement* ⊆ *a0n-agreement*
⟨*proof*⟩

lemma *PO-a0i-a0n-agreement* [*iff*]: *reach a0i* ⊆ *a0n-agreement*
⟨*proof*⟩

lemma *PO-a0i-obs-a0n-agreement* [*iff*]: *oreach a0i* ⊆ *a0n-agreement*
⟨*proof*⟩

end

9 Runs

theory *Runs* **imports** *Messages*
begin

9.1 Type definitions

datatype *role-t* = *Init* | *Resp*

datatype *var* = *Var nat*

type-synonym
rid-t = *fid-t*

type-synonym
frame = *var* ⇀ *msg*

record *run-t* =
role :: *role-t*
owner :: *agent*
partner :: *agent*

type-synonym
progress-t = *rid-t* ⇀ *var set*

fun
in-progress :: *var set option* ⇒ *var* ⇒ *bool*
where
in-progress (*Some S*) *x* = (*x* ∈ *S*)
|*in-progress None x* = *False*

fun
in-progressS :: *var set option* ⇒ *var set* ⇒ *bool*
where
in-progressS (*Some S*) *S'* = (*S'* ⊆ *S*)
|*in-progressS None S'* = *False*

lemma *in-progress-dom* [*elim*]: *in-progress* (*r R*) *x* ⟹ *R* ∈ *dom r*
⟨*proof*⟩

lemma *in-progress-Some* [*elim*]: *in-progress r x* ⟹ ∃ *x*. *r* = *Some x*
⟨*proof*⟩

lemma *in-progressS-elt* [*elim*]: *in-progressS r S* ⟹ *x* ∈ *S* ⟹ *in-progress r x*
⟨*proof*⟩

end

10 Channel Messages

theory *Channels*
imports *Message-derivation*
begin

10.1 Channel messages

datatype *chan* =
  Chan tag agent agent msg

abbreviation
  Insec :: [*agent*, *agent*, *msg*] ⇒ *chan* **where**
  Insec ≡ *Chan insec*

abbreviation
  Confid :: [*agent*, *agent*, *msg*] ⇒ *chan* **where**
  Confid ≡ *Chan confid*

abbreviation
  Auth :: [*agent*, *agent*, *msg*] ⇒ *chan* **where**
  Auth ≡ *Chan auth*

abbreviation
  Secure :: [*agent*, *agent*, *msg*] ⇒ *chan* **where**
  Secure ≡ *Chan secure*

10.2 Extract

The set of payload messages that can be extracted from a set of (crypto) messages and a set of channel messages, given a set of bad agents. The second rule states that the payload can be extracted from insecure and authentic channels as well as from channels with a compromised endpoint.

inductive-set
  extr :: *agent set* ⇒ *msg set* ⇒ *chan set* ⇒ *msg set*
  for *bad* :: *agent set*
  and *IK* :: *msg set*
  and *H* :: *chan set*
where
  extr-Inj: *M* ∈ *IK* ⟹ *M* ∈ *extr bad IK H*
| *extr-Chan*:
    ⟦ *Chan c A B M* ∈ *H*; *c* = *insec* ∨ *c* = *auth* ∨ *A* ∈ *bad* ∨ *B* ∈ *bad* ⟧ ⟹ *M* ∈ *extr bad IK H*

declare *extr.intros* [*intro*]
declare *extr.cases* [*elim*]

lemma *extr-empty-chan* [*simp*]: *extr bad IK* {} = *IK*
⟨*proof*⟩

lemma *IK-subset-extr*: *IK* ⊆ *extr bad IK chan*

⟨*proof*⟩

lemma *extr-mono-chan* [*dest*]: $G \subseteq H \Longrightarrow$ *extr bad IK G* $\subseteq$ *extr bad IK H*
⟨*proof*⟩

lemma *extr-mono-IK* [*dest*]: *IK1* $\subseteq$ *IK2* $\Longrightarrow$ *extr bad IK1 H* $\subseteq$ *extr bad IK2 H*
⟨*proof*⟩

lemma *extr-mono-bad* [*dest*]: *bad* $\subseteq$ *bad'* $\Longrightarrow$ *extr bad IK H* $\subseteq$ *extr bad' IK H*
⟨*proof*⟩

lemmas *extr-monotone-chan* [*elim*] = *extr-mono-chan* [*THEN* [*2*] *rev-subsetD*]
lemmas *extr-monotone-IK* [*elim*] = *extr-mono-IK* [*THEN* [*2*] *rev-subsetD*]
lemmas *extr-monotone-bad* [*elim*] = *extr-mono-bad* [*THEN* [*2*] *rev-subsetD*]

lemma *extr-mono* [*intro*]: $[\![b \subseteq b'; I \subseteq I'; C \subseteq C']\!] \Longrightarrow$ *extr b I C* $\subseteq$ *extr b' I' C'*
⟨*proof*⟩

lemmas *extr-monotone* [*elim*] = *extr-mono* [*THEN* [*2*] *rev-subsetD*]

lemma *extr-insert* [*intro*]: $M \in$ *extr bad IK H* $\Longrightarrow M \in$ *extr bad IK* (*insert C H*)
⟨*proof*⟩

lemma *extr-insert-Chan* [*simp*]:
  extr bad IK (*insert* (*Chan c A B M*) *H*)
   = (*if* c = *insec* $\lor$ c = *auth* $\lor$ $A \in$ *bad* $\lor$ $B \in$ *bad*
      then insert M (*extr bad IK H*) *else extr bad IK H*)
⟨*proof*⟩

lemma *extr-insert-chan-eq*: *extr bad IK* (*insert X CH*) = *extr bad IK* $\{X\}$ $\cup$ *extr bad IK CH*
⟨*proof*⟩

lemma *extr-insert-IK-eq* [*simp*]: *extr bad* (*insert X IK*) *CH* = *insert X* (*extr bad IK CH*)
⟨*proof*⟩

lemma *extr-insert-bad*:
  extr (*insert A bad*) *IK CH* $\subseteq$
   extr bad IK CH $\cup$ $\{M. \exists B.$ *Confid A B M* $\in$ *CH* $\lor$ *Confid B A M* $\in$ *CH* $\lor$
                Secure A B M $\in$ *CH* $\lor$ *Secure B A M* $\in$ *CH*$\}$
⟨*proof*⟩

lemma *extr-insert-Confid* [*simp*]:
  $A \notin$ *bad* $\Longrightarrow$
   $B \notin$ *bad* $\Longrightarrow$
   extr bad IK (*insert* (*Confid A B X*) *CH*) = *extr bad IK CH*
⟨*proof*⟩

10.3 Fake

The set of channel messages that an attacker can fake given a set of compromised agents, a set of crypto messages and a set of channel messages. The second rule states that an attacker can fake an insecure or confidential messages or a channel message with a compromised endpoint using a payload that he knows.

inductive-set
fake :: *agent set* ⇒ *msg set* ⇒ *chan set* ⇒ *chan set*
for *bad* :: *agent set*
and *IK* :: *msg set*
and *chan* :: *chan set*
where
fake-Inj: $M \in chan \Longrightarrow M \in fake\ bad\ IK\ chan$
| *fake-New*:
$[\![\ M \in IK;\ c = insec \lor c = confid \lor A \in bad \lor B \in bad\ ]\!]$
$\Longrightarrow Chan\ c\ A\ B\ M \in fake\ bad\ IK\ chan$

declare *fake.cases* [*elim*]
declare *fake.intros* [*intro*]

lemmas *fake-intros* = *fake-Inj fake-New*

lemma *fake-mono-bad* [*intro*]:
$bad \subseteq bad' \Longrightarrow fake\ bad\ IK\ chan \subseteq fake\ bad'\ IK\ chan$
⟨*proof*⟩

lemma *fake-mono-ik* [*intro*]:
$IK \subseteq IK' \Longrightarrow fake\ bad\ IK\ chan \subseteq fake\ bad\ IK'\ chan$
⟨*proof*⟩

lemma *fake-mono-chan* [*intro*]:
$chan \subseteq chan' \Longrightarrow fake\ bad\ IK\ chan \subseteq fake\ bad\ IK\ chan'$
⟨*proof*⟩

lemma *fake-mono* [*intro*]:
$[\![\ bad \subseteq bad';\ IK \subseteq IK';\ chan \subseteq chan']\!] \Longrightarrow fake\ bad\ IK\ chan \subseteq fake\ bad'\ IK'\ chan'$
⟨*proof*⟩

lemmas *fake-monotone-bad* [*elim*] = *fake-mono-bad* [*THEN* [*2*] *rev-subsetD*]
lemmas *fake-monotone-ik* [*elim*] = *fake-mono-ik* [*THEN* [*2*] *rev-subsetD*]
lemmas *fake-monotone-chan* [*elim*] = *fake-mono-chan* [*THEN* [*2*] *rev-subsetD*]
lemmas *fake-monotone* [*elim*] = *fake-mono* [*THEN* [*2*] *rev-subsetD*]

lemma *chan-subset-fake*: $chan \subseteq fake\ bad\ IK\ chan$
⟨*proof*⟩

lemma *extr-fake*:
$X \in fake\ bad\ IK\ chan \Longrightarrow extr\ bad\ IK'\ \{X\} \subseteq IK \cup extr\ bad\ IK'\ chan$
⟨*proof*⟩

lemmas *extr-fake-2* [*elim*] = *extr-fake* [*THEN* [*2*] *rev-subsetD*]

lemma *fake-parts-extr-singleton*:
$X \in fake\ bad\ IK\ chan \Longrightarrow parts\ (extr\ bad\ IK'\ \{X\}) \subseteq parts\ IK \cup parts\ (extr\ bad\ IK'\ chan)$
⟨*proof*⟩

lemmas *fake-parts-extr-singleton-2* [*elim*] = *fake-parts-extr-singleton* [*THEN* [*2*] *rev-subsetD*]

lemma *fake-parts-extr-insert*:
assumes $X \in$ *fake bad IK CH*
shows *parts* (*extr bad IK′* (*insert X CH*)) $\subseteq$ *parts* (*extr bad IK′ CH*) $\cup$ *parts IK*
⟨*proof*⟩

lemma *fake-synth-analz-extr*:
assumes $X \in$ *fake bad* (*synth* (*analz* (*extr bad IK CH*))) *CH*
shows *synth* (*analz* (*extr bad IK* (*insert X CH*))) = *synth* (*analz* (*extr bad IK CH*))
⟨*proof*⟩

10.4 Closure of Dolev-Yao, extract and fake

10.4.1 *dy-fake-msg*: returns messages, closure of DY and extr is sufficient

Close *extr* under Dolev-Yao closure using *synth* and *analz*. This will be used in Level 2 attacker events to fake crypto messages.

definition
dy-fake-msg :: *agent set* ⇒ *msg set* ⇒ *chan set* ⇒ *msg set*
where
dy-fake-msg b i c = *synth* (*analz* (*extr b i c*))

lemma *dy-fake-msg-empty* [*simp*]: *dy-fake-msg bad* {} {} = *synth* {}
⟨*proof*⟩

lemma *dy-fake-msg-mono-bad* [*dest*]: *bad* $\subseteq$ *bad′* $\Longrightarrow$ *dy-fake-msg bad I C* $\subseteq$ *dy-fake-msg bad′ I C*
⟨*proof*⟩

lemma *dy-fake-msg-mono-ik* [*dest*]: $G \subseteq H \Longrightarrow$ *dy-fake-msg bad G C* $\subseteq$ *dy-fake-msg bad H C*
⟨*proof*⟩

lemma *dy-fake-msg-mono-chan* [*dest*]: $G \subseteq H \Longrightarrow$ *dy-fake-msg bad I G* $\subseteq$ *dy-fake-msg bad I H*
⟨*proof*⟩

lemmas *dy-fake-msg-monotone-bad* [*elim*] = *dy-fake-msg-mono-bad* [*THEN* [*2*] *rev-subsetD*]
lemmas *dy-fake-msg-monotone-ik* [*elim*] = *dy-fake-msg-mono-ik* [*THEN* [*2*] *rev-subsetD*]
lemmas *dy-fake-msg-monotone-chan* [*elim*] = *dy-fake-msg-mono-chan* [*THEN* [*2*] *rev-subsetD*]

lemma *dy-fake-msg-insert* [*intro*]:
$M \in$ *dy-fake-msg bad I C* $\Longrightarrow M \in$ *dy-fake-msg bad I* (*insert X C*)
⟨*proof*⟩

lemma *dy-fake-msg-mono* [*intro*]:
⟦ $b \subseteq b'$; $I \subseteq I'$; $C \subseteq C'$ ⟧ $\Longrightarrow$ *dy-fake-msg b I C* $\subseteq$ *dy-fake-msg b′ I′ C′*
⟨*proof*⟩

lemmas *dy-fake-msg-monotone* [*elim*] = *dy-fake-msg-mono* [*THEN* [*2*] *rev-subsetD*]

lemma *dy-fake-msg-insert-chan*:
$x = insec \lor x = auth \Longrightarrow$
$M \in$ *dy-fake-msg bad IK* (*insert* (*Chan x A B M*) *CH*)
⟨*proof*⟩

10.4.2 *dy-fake-chan*: returns channel messages

The set of all channel messages that an attacker can fake is obtained using *fake* with the sets of possible payload messages derived with *dy-fake-msg* defined above. This will be used in Level 2 attacker events to fake channel messages.

definition
dy-fake-chan :: *agent set* ⇒ *msg set* ⇒ *chan set* ⇒ *chan set*
where
dy-fake-chan b i c = *fake b* (*dy-fake-msg b i c*) *c*

lemma *dy-fake-chan-mono-bad* [*intro*]:
bad ⊆ *bad′* ⟹ *dy-fake-chan bad I C* ⊆ *dy-fake-chan bad′ I C*
⟨*proof*⟩

lemma *dy-fake-chan-mono-ik* [*intro*]:
T ⊆ *T′* ⟹ *dy-fake-chan bad T C* ⊆ *dy-fake-chan bad T′ C*
⟨*proof*⟩

lemma *dy-fake-chan-mono-chan* [*intro*]:
C ⊆ *C′* ⟹ *dy-fake-chan bad T C* ⊆ *dy-fake-chan bad T C′*
⟨*proof*⟩

lemmas *dy-fake-chan-monotone-bad* [*elim*] = *dy-fake-chan-mono-bad* [*THEN* [*2*] *rev-subsetD*]
lemmas *dy-fake-chan-monotone-ik* [*elim*] = *dy-fake-chan-mono-ik* [*THEN* [*2*] *rev-subsetD*]
lemmas *dy-fake-chan-monotone-chan* [*elim*] = *dy-fake-chan-mono-chan* [*THEN* [*2*] *rev-subsetD*]

lemma *dy-fake-chan-mono* [*intro*]:
assumes *b* ⊆ *b′* **and** *I* ⊆ *I′* **and** *C* ⊆ *C′*
shows *dy-fake-chan b I C* ⊆ *dy-fake-chan b′ I′ C′*
⟨*proof*⟩

lemmas *dy-fake-chan-monotone* [*elim*] = *dy-fake-chan-mono* [*THEN* [*2*] *rev-subsetD*]

lemma *dy-fake-msg-subset-synth-analz*:
⟦*extr bad IK chan* ⊆ *T* ⟧ ⟹ *dy-fake-msg bad IK chan* ⊆ *synth* (*analz T*)
⟨*proof*⟩

lemma *dy-fake-chan-mono2*:
⟦ *extr bad IK chan* ⊆ *synth* (*analz y*); *chan* ⊆ *fake bad* (*synth* (*analz y*)) *z* ⟧
⟹ *dy-fake-chan bad IK chan* ⊆ *fake bad* (*synth* (*analz y*)) *z*
⟨*proof*⟩

lemma *extr-subset-dy-fake-msg*: *extr bad IK chan* ⊆ *dy-fake-msg bad IK chan*
⟨*proof*⟩

lemma *dy-fake-chan-extr-insert*:
$M \in$ *dy-fake-chan bad IK CH* $\Longrightarrow$ *extr bad IK* (*insert M CH*) $\subseteq$ *dy-fake-msg bad IK CH*
⟨*proof*⟩

lemma *dy-fake-chan-extr-insert-parts*:
$M \in$ *dy-fake-chan bad IK CH* $\Longrightarrow$
parts (*extr bad IK* (*insert M CH*)) $\subseteq$ *parts* (*extr bad IK CH*) $\cup$ *dy-fake-msg bad IK CH*
⟨*proof*⟩

lemma *dy-fake-msg-extr*:
extr bad ik chan $\subseteq$ *synth* (*analz X*) $\Longrightarrow$ *dy-fake-msg bad ik chan* $\subseteq$ *synth* (*analz X*)
⟨*proof*⟩

lemma *extr-insert-dy-fake-msg*:
$M \in$ *dy-fake-msg bad IK CH* $\Longrightarrow$ *extr bad* (*insert M IK*) *CH* $\subseteq$ *dy-fake-msg bad IK CH*
⟨*proof*⟩

lemma *dy-fake-msg-insert-dy-fake-msg*:
$M \in$ *dy-fake-msg bad IK CH* $\Longrightarrow$ *dy-fake-msg bad* (*insert M IK*) *CH* $\subseteq$ *dy-fake-msg bad IK CH*
⟨*proof*⟩

lemma *synth-analz-insert-dy-fake-msg*:
$M \in$ *dy-fake-msg bad IK CH* $\Longrightarrow$ *synth* (*analz* (*insert M IK*)) $\subseteq$ *dy-fake-msg bad IK CH*
⟨*proof*⟩

lemma *Fake-insert-dy-fake-msg*:
$M \in$ *dy-fake-msg bad IK CH* $\Longrightarrow$
extr bad IK CH $\subseteq$ *synth* (*analz X*) $\Longrightarrow$
synth (*analz* (*insert M IK*)) $\subseteq$ *synth* (*analz X*)
⟨*proof*⟩

lemma *dy-fake-chan-insert-chan*:
$x = insec \lor x = auth \Longrightarrow$
Chan x A B M $\in$ *dy-fake-chan bad IK* (*insert* (*Chan x A B M*) *CH*)
⟨*proof*⟩

lemma *dy-fake-chan-subset*:
CH $\subseteq$ *fake bad* (*dy-fake-msg bad IK CH*) *CH'* $\Longrightarrow$
dy-fake-chan bad IK CH $\subseteq$ *fake bad* (*dy-fake-msg bad IK CH*) *CH'*
⟨*proof*⟩

end

11 Payloads and Support for Channel Message Implementations

Definitions and lemmas that do not require the implementations.

theory *Payloads*
imports *Message-derivation*
begin

11.1 Payload messages

Payload messages contain no implementation material ie no long term keys or tags.

Define set of payloads for basic messages.

inductive-set *cpayload* :: *cmsg set* **where**
  cAgent A ∈ *cpayload*
| *cNumber T* ∈ *cpayload*
| *cNonce N* ∈ *cpayload*
| *cEphK K* ∈ *cpayload*
| *X* ∈ *cpayload* ⟹ *cHash X* ∈ *cpayload*
| ⟦ *X* ∈ *cpayload*; *Y* ∈ *cpayload* ⟧ ⟹ *cPair X Y* ∈ *cpayload*
| ⟦ *X* ∈ *cpayload*; *Y* ∈ *cpayload* ⟧ ⟹ *cEnc X Y* ∈ *cpayload*
| ⟦ *X* ∈ *cpayload*; *Y* ∈ *cpayload* ⟧ ⟹ *cAenc X Y* ∈ *cpayload*
| ⟦ *X* ∈ *cpayload*; *Y* ∈ *cpayload* ⟧ ⟹ *cSign X Y* ∈ *cpayload*
| ⟦ *X* ∈ *cpayload*; *Y* ∈ *cpayload* ⟧ ⟹ *cExp X Y* ∈ *cpayload*

Lift *cpayload* to the quotiented message type.

lift-definition *payload* :: *msg set* **is** *cpayload* ⟨*proof*⟩

Lemmas used to prove the intro and inversion rules for *payload*.

lemma *eq-rep-abs*: *eq x* (*Re* (*Ab x*))
⟨*proof*⟩

lemma *eq-cpayload*:
  assumes *eq x y* **and** *x* ∈ *cpayload*
  shows *y* ∈ *cpayload*
⟨*proof*⟩

lemma *abs-payload*: *Ab x* ∈ *payload* ⟷ *x* ∈ *cpayload*
⟨*proof*⟩

lemma *abs-cpayload-rep*: *x* ∈ *Ab‘ cpayload* ⟷ *Re x* ∈ *cpayload*
⟨*proof*⟩

lemma *payload-rep-cpayload*: *Re x* ∈ *cpayload* ⟷ *x* ∈ *payload*
⟨*proof*⟩

Manual proof of payload introduction rules. Transfer does not work for these

declare *cpayload.intros* [*intro*]
lemma *payload-AgentI*: *Agent A* ∈ *payload*

⟨*proof*⟩
lemma *payload-NonceI*: *Nonce N* ∈ *payload*
⟨*proof*⟩
lemma *payload-NumberI*: *Number N* ∈ *payload*
⟨*proof*⟩
lemma *payload-EphKI*: *EphK X* ∈ *payload*
⟨*proof*⟩
lemma *payload-HashI*: *x* ∈ *payload* ⟹ *Hash x* ∈ *payload*
⟨*proof*⟩
lemma *payload-PairI*: *x* ∈ *payload* ⟹ *y* ∈ *payload* ⟹ *Pair x y* ∈ *payload*
⟨*proof*⟩
lemma *payload-EncI*: *x* ∈ *payload* ⟹ *y* ∈ *payload* ⟹ *Enc x y* ∈ *payload*
⟨*proof*⟩
lemma *payload-AencI*: *x* ∈ *payload* ⟹ *y* ∈ *payload* ⟹ *Aenc x y* ∈ *payload*
⟨*proof*⟩
lemma *payload-SignI*: *x* ∈ *payload* ⟹ *y* ∈ *payload* ⟹ *Sign x y* ∈ *payload*
⟨*proof*⟩
lemma *payload-ExpI*: *x* ∈ *payload* ⟹ *y* ∈ *payload* ⟹ *Exp x y* ∈ *payload*
⟨*proof*⟩

lemmas *payload-intros* [*simp*, *intro*] =
payload-AgentI payload-NonceI payload-NumberI payload-EphKI payload-HashI
payload-PairI payload-EncI payload-AencI payload-SignI payload-ExpI

Manual proof of payload inversion rules, transfer does not work for these.

declare *cpayload.cases*[*elim*]
lemma *payload-Tag*: *Tag X* ∈ *payload* ⟹ *P*
⟨*proof*⟩

lemma *payload-LtK*: *LtK X* ∈ *payload* ⟹ *P*
⟨*proof*⟩
lemma *payload-Hash*: *Hash X* ∈ *payload* ⟹ (*X* ∈ *payload* ⟹ *P*) ⟹ *P*
⟨*proof*⟩
lemma *payload-Pair*: *Pair X Y* ∈ *payload* ⟹ (*X* ∈ *payload* ⟹ *Y* ∈ *payload* ⟹ *P*) ⟹ *P*
⟨*proof*⟩
lemma *payload-Enc*: *Enc X Y* ∈ *payload* ⟹ (*X* ∈ *payload* ⟹ *Y* ∈ *payload* ⟹ *P*) ⟹ *P*
⟨*proof*⟩
lemma *payload-Aenc*: *Aenc X Y* ∈ *payload* ⟹ (*X* ∈ *payload* ⟹ *Y* ∈ *payload* ⟹ *P*) ⟹ *P*
⟨*proof*⟩
lemma *payload-Sign*: *Sign X Y* ∈ *payload* ⟹ (*X* ∈ *payload* ⟹ *Y* ∈ *payload* ⟹ *P*) ⟹ *P*
⟨*proof*⟩
lemma *payload-Exp*: *Exp X Y* ∈ *payload* ⟹ (*X* ∈ *payload* ⟹ *Y* ∈ *payload* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

declare *cpayload.intros*[*rule del*]
declare *cpayload.cases*[*rule del*]

lemmas *payload-inductive-cases* =
payload-Tag payload-LtK payload-Hash
payload-Pair payload-Enc payload-Aenc payload-Sign payload-Exp

lemma *eq-exhaust*:
(⋀*x*. *eq y* (*cAgent x*) ⟹ *P*) ⟹

$(\bigwedge x.\ eq\ y\ (cNumber\ x) \Longrightarrow P) \Longrightarrow$
$(\bigwedge x.\ eq\ y\ (cNonce\ x) \Longrightarrow P) \Longrightarrow$
$(\bigwedge x.\ eq\ y\ (cLtK\ x) \Longrightarrow P) \Longrightarrow$
$(\bigwedge x.\ eq\ y\ (cEphK\ x) \Longrightarrow P) \Longrightarrow$
$(\bigwedge x\ x'.\ eq\ y\ (cPair\ x\ x') \Longrightarrow P) \Longrightarrow$
$(\bigwedge x\ x'.\ eq\ y\ (cEnc\ x\ x') \Longrightarrow P) \Longrightarrow$
$(\bigwedge x\ x'.\ eq\ y\ (cAenc\ x\ x') \Longrightarrow P) \Longrightarrow$
$(\bigwedge x\ x'.\ eq\ y\ (cSign\ x\ x') \Longrightarrow P) \Longrightarrow$
$(\bigwedge x.\ eq\ y\ (cHash\ x) \Longrightarrow P) \Longrightarrow$
$(\bigwedge x.\ eq\ y\ (cTag\ x) \Longrightarrow P) \Longrightarrow$
$(\bigwedge x\ x'.\ eq\ y\ (cExp\ x\ x') \Longrightarrow P) \Longrightarrow$
P
⟨*proof*⟩

lemma *msg-exhaust*:
$(\bigwedge x.\ y = Agent\ x \Longrightarrow P) \Longrightarrow$
$(\bigwedge x.\ y = Number\ x \Longrightarrow P) \Longrightarrow$
$(\bigwedge x.\ y = Nonce\ x \Longrightarrow P) \Longrightarrow$
$(\bigwedge x.\ y = LtK\ x \Longrightarrow P) \Longrightarrow$
$(\bigwedge x.\ y = EphK\ x \Longrightarrow P) \Longrightarrow$
$(\bigwedge x\ x'.\ y = Pair\ x\ x' \Longrightarrow P) \Longrightarrow$
$(\bigwedge x\ x'.\ y = Enc\ x\ x' \Longrightarrow P) \Longrightarrow$
$(\bigwedge x\ x'.\ y = Aenc\ x\ x' \Longrightarrow P) \Longrightarrow$
$(\bigwedge x\ x'.\ y = Sign\ x\ x' \Longrightarrow P) \Longrightarrow$
$(\bigwedge x.\ y = Hash\ x \Longrightarrow P) \Longrightarrow$
$(\bigwedge x.\ y = Tag\ x \Longrightarrow P) \Longrightarrow$
$(\bigwedge x\ x'.\ y = Exp\ x\ x' \Longrightarrow P) \Longrightarrow$
P
⟨*proof*⟩

lemma *payload-cases*:
$a \in payload \Longrightarrow$
$(\bigwedge A.\ a = Agent\ A \Longrightarrow P) \Longrightarrow$
$(\bigwedge T.\ a = Number\ T \Longrightarrow P) \Longrightarrow$
$(\bigwedge N.\ a = Nonce\ N \Longrightarrow P) \Longrightarrow$
$(\bigwedge K.\ a = EphK\ K \Longrightarrow P) \Longrightarrow$
$(\bigwedge X.\ a = Hash\ X \Longrightarrow X \in payload \Longrightarrow P) \Longrightarrow$
$(\bigwedge X\ Y.\ a = Pair\ X\ Y \Longrightarrow X \in payload \Longrightarrow Y \in payload \Longrightarrow P) \Longrightarrow$
$(\bigwedge X\ Y.\ a = Enc\ X\ Y \Longrightarrow X \in payload \Longrightarrow Y \in payload \Longrightarrow P) \Longrightarrow$
$(\bigwedge X\ Y.\ a = Aenc\ X\ Y \Longrightarrow X \in payload \Longrightarrow Y \in payload \Longrightarrow P) \Longrightarrow$
$(\bigwedge X\ Y.\ a = Sign\ X\ Y \Longrightarrow X \in payload \Longrightarrow Y \in payload \Longrightarrow P) \Longrightarrow$
$(\bigwedge X\ Y.\ a = Exp\ X\ Y \Longrightarrow X \in payload \Longrightarrow Y \in payload \Longrightarrow P) \Longrightarrow$
P
⟨*proof*⟩

declare *payload-cases* [*elim*]
declare *payload-inductive-cases* [*elim*]

Properties of payload; messages constructed from payload messages are also payloads.

lemma *payload-parts* [*simp*, *dest*]:
$[\![X \in parts\ S;\ S \subseteq payload]\!] \Longrightarrow X \in payload$
⟨*proof*⟩

lemma *payload-parts-singleton* [*simp*, *dest*]:
$[\![X \in parts \{Y\};\ Y \in payload]\!] \Longrightarrow X \in payload$
⟨*proof*⟩

lemma *payload-analz* [*simp*, *dest*]:
$[\![X \in analz\ S;\ S \subseteq payload]\!] \Longrightarrow X \in payload$
⟨*proof*⟩

lemma *payload-synth-analz*:
$[\![X \in synth\ (analz\ S);\ S \subseteq payload]\!] \Longrightarrow X \in payload$
⟨*proof*⟩

Important lemma: using messages with implementation material one can only synthesise more such messages.

lemma *synth-payload*:
$Y \cap payload = \{\} \Longrightarrow synth\ (X \cup Y) \subseteq synth\ X \cup -payload$
⟨*proof*⟩

lemma *synth-payload2*:
$Y \cap payload = \{\} \Longrightarrow synth\ (Y \cup X) \subseteq synth\ X \cup -payload$
⟨*proof*⟩

Lemma: in the case of the previous lemma, *synth* can be applied on the left with no consequence.

lemma *synth-idem-payload*:
$X \subseteq synth\ Y \cup -payload \Longrightarrow synth\ X \subseteq synth\ Y \cup -payload$
⟨*proof*⟩

11.2 *isLtKey*: is a long term key

lemma *LtKeys-payload* [*dest*]: $NI \subseteq payload \Longrightarrow NI \cap range\ LtK = \{\}$
⟨*proof*⟩

lemma *LtKeys-parts-payload* [*dest*]: $NI \subseteq payload \Longrightarrow parts\ NI \cap range\ LtK = \{\}$
⟨*proof*⟩

lemma *LtKeys-parts-payload-singleton* [*elim*]: $X \in payload \Longrightarrow LtK\ Y \in parts\ \{X\} \Longrightarrow False$
⟨*proof*⟩

lemma *parts-of-LtKeys* [*simp*]: $K \subseteq range\ LtK \Longrightarrow parts\ K = K$
⟨*proof*⟩

11.3 *keys-of*: the long term keys of an agent

definition
keys-of :: *agent* ⇒ *msg set*
where
$keys\text{-}of\ A \equiv insert\ (priK\ A)\ \{shrK\ B\ C \mid B\ C.\ B = A \vee C = A\}$

lemma *keys-of-Ltk* [*intro*!]: $keys\text{-}of\ A \subseteq range\ LtK$
⟨*proof*⟩

lemma *priK-keys-of* [*intro*!]:
  $priK\ A \in keys\text{-}of\ A$
⟨*proof*⟩

lemma *shrK-keys-of-1* [*intro*!]:
  $shrK\ A\ B \in keys\text{-}of\ A$
⟨*proof*⟩

lemma *shrK-keys-of-2* [*intro*!]:
  $shrK\ B\ A \in keys\text{-}of\ A$
⟨*proof*⟩
lemma *priK-keys-of-eq* [*dest*]:
  $priK\ B \in keys\text{-}of\ A \Longrightarrow A = B$
⟨*proof*⟩

lemma *shrK-keys-of-eq* [*dest*]:
  $shrK\ A\ B \in keys\text{-}of\ C \Longrightarrow A = C \lor B = C$
⟨*proof*⟩

lemma *def-keys-of* [*dest*]:
  $K \in keys\text{-}of\ A \Longrightarrow K = priK\ A \lor (\exists\ B.\ K = shrK\ A\ B \lor K = shrK\ B\ A)$
⟨*proof*⟩

lemma *parts-keys-of* [*simp*]: $parts\ (keys\text{-}of\ A) = keys\text{-}of\ A$
⟨*proof*⟩

lemma *analz-keys-of* [*simp*]: $analz\ (keys\text{-}of\ A) = keys\text{-}of\ A$
⟨*proof*⟩

11.4 *Keys-bad*: bounds on the attacker's knowledge of long-term keys.

A set of keys contains all public long term keys, and only the private/shared keys of bad agents.

definition
  $Keys\text{-}bad :: msg\ set \Rightarrow agent\ set \Rightarrow bool$
where
  $Keys\text{-}bad\ IK\ Bad \equiv$
    $IK \cap range\ LtK \subseteq range\ pubK \cup \bigcup\ (keys\text{-}of\ `\ Bad)$
    $\land\ range\ pubK \subseteq IK$

— basic lemmas

lemma *Keys-badI*:
  $[\![\ IK \cap range\ LtK \subseteq range\ pubK \cup priK`Bad \cup \{shrK\ A\ B \mid A\ B.\ A \in Bad \lor B \in Bad\};$
    $range\ pubK \subseteq IK\ ]\!]$
  $\Longrightarrow Keys\text{-}bad\ IK\ Bad$
⟨*proof*⟩

lemma *Keys-badE* [*elim*]:
  $[\![\ Keys\text{-}bad\ IK\ Bad;$
    $[\![\ range\ pubK \subseteq IK;$

$IK \cap range\ LtK \subseteq range\ pubK \cup \bigcup\ (keys\text{-}of\ `\ Bad)]\!]$
$\Longrightarrow P\ ]\!]$
$\Longrightarrow P$
⟨proof⟩

lemma *Keys-bad-Ltk* [*simp*]:
Keys-bad (*IK* ∩ *range LtK*) *Bad* ⟷ *Keys-bad IK Bad*
⟨proof⟩

lemma *Keys-bad-priK-D*: ⟦ *priK A* ∈ *IK*; *Keys-bad IK Bad* ⟧ ⟹ *A* ∈ *Bad*
⟨proof⟩

lemma *Keys-bad-shrK-D*: ⟦ *shrK A B* ∈ *IK*; *Keys-bad IK Bad* ⟧ ⟹ *A* ∈ *Bad* ∨ *B* ∈ *Bad*
⟨proof⟩

lemmas *Keys-bad-dests* [*dest*] = *Keys-bad-priK-D Keys-bad-shrK-D*

interaction with *insert*.

lemma *Keys-bad-insert-non-LtK*:
X ∉ *range LtK* ⟹ *Keys-bad* (*insert X IK*) *Bad* ⟷ *Keys-bad IK Bad*
⟨proof⟩

lemma *Keys-bad-insert-pubK*:
⟦ *Keys-bad IK Bad* ⟧ ⟹ *Keys-bad* (*insert* (*pubK A*) *IK*) *Bad*
⟨proof⟩

lemma *Keys-bad-insert-priK-bad*:
⟦ *Keys-bad IK Bad*; *A* ∈ *Bad* ⟧ ⟹ *Keys-bad* (*insert* (*priK A*) *IK*) *Bad*
⟨proof⟩

lemma *Keys-bad-insert-shrK-bad*:
⟦ *Keys-bad IK Bad*; *A* ∈ *Bad* ∨ *B* ∈ *Bad* ⟧ ⟹ *Keys-bad* (*insert* (*shrK A B*) *IK*) *Bad*
⟨proof⟩

lemmas *Keys-bad-insert-lemmas* [*simp*] =
Keys-bad-insert-non-LtK Keys-bad-insert-pubK
Keys-bad-insert-priK-bad Keys-bad-insert-shrK-bad

lemma *Keys-bad-insert-Fake*:
assumes *Keys-bad IK Bad*
and *parts IK* ∩ *range LtK* ⊆ *IK*
and *X* ∈ *synth* (*analz IK*)
shows *Keys-bad* (*insert X IK*) *Bad*
⟨proof⟩

lemma *Keys-bad-insert-keys-of*:
Keys-bad Ik Bad ⟹
Keys-bad (*keys-of A* ∪ *Ik*) (*insert A Bad*)
⟨proof⟩

lemma *Keys-bad-insert-payload*:
$Keys\text{-}bad\ Ik\ Bad \Longrightarrow$
$x \in payload \Longrightarrow$
$Keys\text{-}bad\ (insert\ x\ Ik)\ Bad$
⟨*proof*⟩

11.5 *broken K*: pairs of agents where at least one is compromised.

Set of pairs (A,B) such that the priK of A or B, or their shared key, is in K

definition
$broken :: msg\ set \Rightarrow (agent * agent)\ set$
where
$broken\ K \equiv \{(A,B) \mid A\ B.\ priK\ A \in K \vee priK\ B \in K \vee shrK\ A\ B \in K \vee shrK\ B\ A \in K\}$

lemma *brokenD* [*dest*!]:
$(A, B) \in broken\ K \Longrightarrow priK\ A \in K \vee priK\ B \in K \vee shrK\ A\ B \in K \vee shrK\ B\ A \in K$
⟨*proof*⟩

lemma *brokenI* [*intro*!]:
$priK\ A \in K \vee priK\ B \in K \vee shrK\ A\ B \in K \vee shrK\ B\ A \in K \Longrightarrow (A, B) \in broken\ K$
⟨*proof*⟩

11.6 *Enc-keys-clean S*: messages with "clean" symmetric encryptions.

All terms used as symmetric keys in S are either long term keys or messages without implementation material.

definition
$Enc\text{-}keys\text{-}clean :: msg\ set \Rightarrow bool$
where
$Enc\text{-}keys\text{-}clean\ S \equiv \forall X\ Y.\ Enc\ X\ Y \in parts\ S \longrightarrow Y \in range\ LtK \cup payload$

lemma *Enc-keys-cleanI*:
$\forall X\ Y.\ Enc\ X\ Y \in parts\ S \longrightarrow Y \in range\ LtK \cup payload \Longrightarrow Enc\text{-}keys\text{-}clean\ S$
⟨*proof*⟩
lemma *Enc-keys-clean-mono*:
$Enc\text{-}keys\text{-}clean\ H \Longrightarrow G \subseteq H \Longrightarrow Enc\text{-}keys\text{-}clean\ G$ — anti-tone
⟨*proof*⟩

lemma *Enc-keys-clean-Un* [*simp*]:
$Enc\text{-}keys\text{-}clean\ (G \cup H) \longleftrightarrow Enc\text{-}keys\text{-}clean\ G \wedge Enc\text{-}keys\text{-}clean\ H$
⟨*proof*⟩
lemma *Enc-keys-clean-analz*:
$Enc\ X\ K \in analz\ S \Longrightarrow Enc\text{-}keys\text{-}clean\ S \Longrightarrow K \in range\ LtK \cup payload$
⟨*proof*⟩
lemma *Enc-keys-clean-Tags* [*simp*,*intro*]: $Enc\text{-}keys\text{-}clean\ Tags$
⟨*proof*⟩

lemma *Enc-keys-clean-LtKeys* [*simp*,*intro*]: $K \subseteq range\ LtK \Longrightarrow Enc\text{-}keys\text{-}clean\ K$
⟨*proof*⟩

lemma *Enc-keys-clean-payload* [*simp*,*intro*]: $NI \subseteq payload \Longrightarrow Enc\text{-}keys\text{-}clean\ NI$
⟨*proof*⟩

11.7 Sets of messages with particular constructors

Sets of all pairs, ciphertexts, and signatures constructed from a set of messages.

abbreviation *AgentSet* :: *msg set*
where *AgentSet* ≡ *range Agent*

abbreviation *PairSet* :: *msg set* ⇒ *msg set* ⇒ *msg set*
where *PairSet G H* ≡ {*Pair X Y* | *X Y*. *X* ∈ *G* ∧ *Y* ∈ *H*}

abbreviation *EncSet* :: *msg set* ⇒ *msg set* ⇒ *msg set*
where *EncSet G K* ≡ {*Enc X Y* | *X Y*. *X* ∈ *G* ∧ *Y* ∈ *K*}

abbreviation *AencSet* :: *msg set* ⇒ *msg set* ⇒ *msg set*
where *AencSet G K* ≡ {*Aenc X Y* | *X Y*. *X* ∈ *G* ∧ *Y* ∈ *K*}

abbreviation *SignSet* :: *msg set* ⇒ *msg set* ⇒ *msg set*
where *SignSet G K* ≡ {*Sign X Y* | *X Y*. *X* ∈ *G* ∧ *Y* ∈ *K*}

abbreviation *HashSet* :: *msg set* ⇒ *msg set*
where *HashSet G* ≡ {*Hash X* | *X*. *X* ∈ *G*}

Move *Enc*, *Aenc*, *Sign*, and *Messages.Pair* sets out of *parts*.

lemma *parts-PairSet*:
parts (*PairSet G H*) ⊆ *PairSet G H* ∪ *parts G* ∪ *parts H*
⟨*proof*⟩

lemma *parts-EncSet*:
parts (*EncSet G K*) ⊆ *EncSet G K* ∪ *PairSet* (*range Agent*) *G* ∪ *range Agent* ∪ *parts G*
⟨*proof*⟩

lemma *parts-AencSet*:
parts (*AencSet G K*) ⊆ *AencSet G K* ∪ *PairSet* (*range Agent*) *G* ∪ *range Agent* ∪ *parts G*
⟨*proof*⟩

lemma *parts-SignSet*:
parts (*SignSet G K*) ⊆ *SignSet G K* ∪ *PairSet* (*range Agent*) *G* ∪ *range Agent* ∪ *parts G*
⟨*proof*⟩

lemma *parts-HashSet*:
parts (*HashSet G*) ⊆ *HashSet G*
⟨*proof*⟩

lemmas *parts-msgSet* = *parts-PairSet parts-EncSet parts-AencSet parts-SignSet parts-HashSet*
lemmas *parts-msgSetD* = *parts-msgSet* [*THEN* [*2*] *rev-subsetD*]

Remove the message sets from under the *Enc-keys-clean* predicate. Only when the first part is a set of agents or tags for *Messages.Pair*, this is sufficient.

lemma *Enc-keys-clean-PairSet-Agent-Un*:
Enc-keys-clean (*G* ∪ *H*) ⟹ *Enc-keys-clean* (*PairSet* (*Agent‘X*) *G* ∪ *H*)
⟨*proof*⟩

lemma *Enc-keys-clean-PairSet-Tag-Un*:

Enc-keys-clean $(G \cup H) \Longrightarrow$ *Enc-keys-clean* (*PairSet Tags G* $\cup$ *H*)
⟨*proof*⟩

lemma *Enc-keys-clean-AencSet-Un*:
Enc-keys-clean $(G \cup H) \Longrightarrow$ *Enc-keys-clean* (*AencSet G K* $\cup$ *H*)
⟨*proof*⟩

lemma *Enc-keys-clean-EncSet-Un*:
K $\subseteq$ *range LtK* $\Longrightarrow$ *Enc-keys-clean* $(G \cup H) \Longrightarrow$ *Enc-keys-clean* (*EncSet G K* $\cup$ *H*)
⟨*proof*⟩

lemma *Enc-keys-clean-SignSet-Un*:
Enc-keys-clean $(G \cup H) \Longrightarrow$ *Enc-keys-clean* (*SignSet G K* $\cup$ *H*)
⟨*proof*⟩

lemma *Enc-keys-clean-HashSet-Un*:
Enc-keys-clean $(G \cup H) \Longrightarrow$ *Enc-keys-clean* (*HashSet G* $\cup$ *H*)
⟨*proof*⟩

lemmas *Enc-keys-clean-msgSet-Un* =
Enc-keys-clean-PairSet-Tag-Un Enc-keys-clean-PairSet-Agent-Un
Enc-keys-clean-EncSet-Un Enc-keys-clean-AencSet-Un
Enc-keys-clean-SignSet-Un Enc-keys-clean-HashSet-Un

11.7.1 Lemmas for moving message sets out of *analz*

Pull *EncSet* out of *analz*.

lemma *analz-Un-EncSet*:
assumes *K* $\subseteq$ *range LtK* **and** *Enc-keys-clean* $(G \cup H)$
shows *analz* (*EncSet G K* $\cup$ *H*) $\subseteq$ *EncSet G K* $\cup$ *analz* $(G \cup H)$
⟨*proof*⟩

Pull *EncSet* out of *analz*, 2nd case: the keys are unknown.

lemma *analz-Un-EncSet2*:
assumes *Enc-keys-clean H* **and** *K* $\subseteq$ *range LtK* **and** *K* $\cap$ *synth* (*analz H*) = {}
shows *analz* (*EncSet G K* $\cup$ *H*) $\subseteq$ *EncSet G K* $\cup$ *analz H*
⟨*proof*⟩

Pull *AencSet* out of the *analz*.

lemma *analz-Un-AencSet*:
assumes *K* $\subseteq$ *range LtK* **and** *Enc-keys-clean* $(G \cup H)$
shows *analz* (*AencSet G K* $\cup$ *H*) $\subseteq$ *AencSet G K* $\cup$ *analz* $(G \cup H)$
⟨*proof*⟩

Pull *AencSet* out of *analz*, 2nd case: the keys are unknown.

lemma *analz-Un-AencSet2*:
assumes *Enc-keys-clean H* **and** *priK‘Ag* $\cap$ *synth* (*analz H*) = {}
shows *analz* (*AencSet G* (*pubK‘Ag*) $\cup$ *H*) $\subseteq$ *AencSet G* (*pubK‘Ag*) $\cup$ *analz H*
⟨*proof*⟩

Pull *PairSet* out of *analz*.

lemma *analz-Un-PairSet*:
analz (*PairSet G G′* ∪ *H*) ⊆ *PairSet G G′* ∪ *analz* (*G* ∪ *G′* ∪ *H*)
⟨*proof*⟩
lemma *analz-Un-SignSet*:
assumes *K* ⊆ *range LtK* **and** *Enc-keys-clean* (*G* ∪ *H*)
shows *analz* (*SignSet G K* ∪ *H*) ⊆ *SignSet G K* ∪ *analz* (*G* ∪ *H*)
⟨*proof*⟩

Pull *Tags* out of *analz*.

lemma *analz-Un-Tag*:
assumes *Enc-keys-clean H*
shows *analz* (*Tags* ∪ *H*) ⊆ *Tags* ∪ *analz H*
⟨*proof*⟩

Pull the *AgentSet* out of the *analz*.

lemma *analz-Un-AgentSet*:
shows *analz* (*AgentSet* ∪ *H*) ⊆ *AgentSet* ∪ *analz H*
⟨*proof*⟩

Pull *HashSet* out of *analz*.

lemma *analz-Un-HashSet*:
assumes *Enc-keys-clean H* **and** *G* ⊆ − *payload*
shows *analz* (*HashSet G* ∪ *H*) ⊆ *HashSet G* ∪ *analz H*
⟨*proof*⟩

end

12 Assumptions for Channel Message Implementation

We define a series of locales capturing our assumptions on channel message implementations.

theory *Implem*
imports *Channels Payloads*
begin

12.1 First step: basic implementation locale

This locale has no assumptions, it only fixes an implementation function and defines some useful abbreviations (impl*, impl*Set) and *valid*.

locale *basic-implem* =
  fixes *implem* :: *chan* ⇒ *msg*
begin

abbreviation *implInsec A B M* ≡ *implem* (*Insec A B M*)
abbreviation *implConfid A B M* ≡ *implem* (*Confid A B M*)
abbreviation *implAuth A B M* ≡ *implem* (*Auth A B M*)
abbreviation *implSecure A B M* ≡ *implem* (*Secure A B M*)

abbreviation *implInsecSet* :: *msg set* ⇒ *msg set*
where *implInsecSet G* ≡ {*implInsec A B M* | *A B M*. *M* ∈ *G*}

abbreviation *implConfidSet* :: (*agent* ∗ *agent*) *set* ⇒ *msg set* ⇒ *msg set*
where *implConfidSet Ag G* ≡ {*implConfid A B M* | *A B M*. (*A*, *B*) ∈ *Ag* ∧ *M* ∈ *G*}

abbreviation *implAuthSet* :: *msg set* ⇒ *msg set*
where *implAuthSet G* ≡ {*implAuth A B M* | *A B M*. *M* ∈ *G*}

abbreviation *implSecureSet* :: (*agent* ∗ *agent*) *set* ⇒ *msg set* ⇒ *msg set*
where *implSecureSet Ag G* ≡ {*implSecure A B M* | *A B M*. (*A*, *B*) ∈ *Ag* ∧ *M* ∈ *G*}

definition
  valid :: *msg set*
where
  valid ≡ {*implem* (*Chan x A B M*) | *x A B M*. *M* ∈ *payload*}

lemma *validI*:
  M ∈ *payload* ⟹ *implem* (*Chan x A B M*) ∈ *valid*
⟨*proof*⟩

lemma *validE*:
  X ∈ *valid* ⟹ (⋀ *x A B M*. *X* = *implem* (*Chan x A B M*) ⟹ *M* ∈ *payload* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *valid-cases*:
fixes *X P*
assumes *X* ∈ *valid*
    (⋀*A B M*. *X* = *implInsec A B M* ⟹ *M* ∈ *payload* ⟹ *P*)
    (⋀*A B M*. *X* = *implConfid A B M* ⟹ *M* ∈ *payload* ⟹ *P*)
    (⋀*A B M*. *X* = *implAuth A B M* ⟹ *M* ∈ *payload* ⟹ *P*)

$(\bigwedge A\ B\ M.\ X = implSecure\ A\ B\ M \Longrightarrow M \in payload \Longrightarrow P)$
shows P
⟨*proof*⟩

end

12.2 Second step: basic and analyze assumptions

This locale contains most of the assumptions on implem, i.e.:

- *impl-inj*: injectivity
- *parts-impl-inj*: injectivity through parts
- *Enc-parts-valid-impl*: if Enc X Y appears in parts of an implem, then it is in parts of the payload, or the key is either long term or payload
- *impl-composed*: the implementations are composed (not nonces, agents, tags etc.)
- *analz-Un-implXXXSet*: move the impl*Set out of the analz (only keep the payloads)
- *impl-Impl*: implementations contain implementation material
- *LtK-parts-impl*: no exposed long term keys in the implementations (i.e., they are only used as keys, or under hashes)

locale *semivalid-implem* = *basic-implem* +
— injectivity
assumes *impl-inj*:
$implem\ (Chan\ x\ A\ B\ M) = implem\ (Chan\ x'\ A'\ B'\ M')$
$\longleftrightarrow x = x' \wedge A = A' \wedge B = B' \wedge M = M'$
— implementations and parts
and *parts-impl-inj*:
$M' \in payload \Longrightarrow$
$implem\ (Chan\ x\ A\ B\ M) \in parts\ \{implem\ (Chan\ x'\ A'\ B'\ M')\} \Longrightarrow$
$x = x' \wedge A = A' \wedge B = B' \wedge M = M'$
and *Enc-keys-clean-valid*: $I \subseteq valid \Longrightarrow Enc\text{-}keys\text{-}clean\ I$
and *impl-composed*: $composed\ (implem\ Z)$
and *impl-Impl*: $implem\ (Chan\ x\ A\ B\ M) \notin payload$
— no ltk in the parts of an implementation
and *LtK-parts-impl*: $X \in valid \Longrightarrow LtK\ K \notin parts\ \{X\}$

— analyze assumptions:
and *analz-Un-implInsecSet*:
$[\![\ G \subseteq payload;\ Enc\text{-}keys\text{-}clean\ H\ ]\!]$
$\Longrightarrow analz\ (implInsecSet\ G \cup H) \subseteq synth\ (analz\ (G \cup H)) \cup -payload$
and *analz-Un-implConfidSet*:
$[\![\ G \subseteq payload;\ Enc\text{-}keys\text{-}clean\ H\ ]\!]$
$\Longrightarrow analz\ (implConfidSet\ Ag\ G \cup H) \subseteq synth\ (analz\ (G \cup H)) \cup -payload$
and *analz-Un-implConfidSet-2*:
$[\![\ G \subseteq payload;\ Enc\text{-}keys\text{-}clean\ H;\ Ag \cap broken\ (parts\ H \cap range\ LtK) = \{\}\ ]\!]$
$\Longrightarrow analz\ (implConfidSet\ Ag\ G \cup H) \subseteq synth\ (analz\ H) \cup -payload$
and *analz-Un-implAuthSet*:

$\llbracket$ $G \subseteq payload$; $Enc\text{-}keys\text{-}clean\ H$ $\rrbracket$
$\Longrightarrow analz\ (implAuthSet\ G \cup H) \subseteq synth\ (analz\ (G \cup H)) \cup -payload$
and *analz-Un-implSecureSet*:
$\llbracket$ $G \subseteq payload$; $Enc\text{-}keys\text{-}clean\ H$ $\rrbracket$
$\Longrightarrow analz\ (implSecureSet\ Ag\ G \cup H) \subseteq synth\ (analz\ (G \cup H)) \cup -payload$
and *analz-Un-implSecureSet-2*:
$\llbracket$ $G \subseteq payload$; $Enc\text{-}keys\text{-}clean\ H$; $Ag \cap broken\ (parts\ H \cap range\ LtK) = \{\}$ $\rrbracket$
$\Longrightarrow analz\ (implSecureSet\ Ag\ G \cup H) \subseteq synth\ (analz\ H) \cup -payload$

begin
— declare some attributes and abbreviations for the hypotheses
— and prove some simple consequences of the hypotheses
declare *impl-inj* [*simp*]

lemmas *parts-implE* [*elim*] = *parts-impl-inj* [*rotated 1*]

declare *impl-composed* [*simp*, *intro*]

lemma *composed-arg-cong*: $X = Y \Longrightarrow composed\ X \longleftrightarrow composed\ Y$
⟨*proof*⟩

lemma *implem-Tags-aux*: $implem\ (Chan\ x\ A\ B\ M) \notin Tags$ ⟨*proof*⟩
lemma *implem-Tags* [*simp*]: $implem\ x \notin Tags$ ⟨*proof*⟩
lemma *implem-LtK-aux*: $implem\ (Chan\ x\ A\ B\ M) \neq LtK\ K$ ⟨*proof*⟩
lemma *implem-LtK* [*simp*]: $implem\ x \neq LtK\ K$ ⟨*proof*⟩
lemma *implem-LtK2* [*simp*]: $implem\ x \notin range\ LtK$ ⟨*proof*⟩

declare *impl-Impl* [*simp*]

lemma *LtK-parts-impl-insert*:
$LtK\ K \in parts\ (insert\ (implem\ (Chan\ x\ A\ B\ M))\ S) \Longrightarrow M \in payload \Longrightarrow LtK\ K \in parts\ S$
⟨*proof*⟩

declare *LtK-parts-impl-insert* [*dest*]
declare *Enc-keys-clean-valid* [*simp*, *intro*]

lemma *valid-composed* [*simp*,*dest*]: $M \in valid \Longrightarrow composed\ M$
⟨*proof*⟩
lemma *valid-payload* [*dest*]: $\llbracket$ $X \in valid$; $X \in payload$ $\rrbracket \Longrightarrow P$
⟨*proof*⟩
lemma *valid-isLtKey* [*dest*]: $\llbracket$ $X \in valid$; $X \in range\ LtK$ $\rrbracket \Longrightarrow P$
⟨*proof*⟩

lemma *analz-valid*:
$H \subseteq payload \cup valid \cup range\ LtK \cup Tags \Longrightarrow$
$implem\ (Chan\ x\ A\ B\ M) \in analz\ H \Longrightarrow$
$implem\ (Chan\ x\ A\ B\ M) \in H$
⟨*proof*⟩

lemma *parts-valid-LtKeys-disjoint*:
$I \subseteq valid \Longrightarrow parts\ I \cap range\ LtK = \{\}$
⟨*proof*⟩

lemma *analz-LtKeys*:
$H \subseteq payload \cup valid \cup range\ LtK \cup Tags \Longrightarrow$
$analz\ H \cap range\ LtK \subseteq H$
⟨*proof*⟩

end

12.3 Third step: *valid-implem*

This extends *semivalid-implem* with four new assumptions, which under certain conditions give information on A, B, M when $implXXX\ A\ B\ M \in synth\ (analz\ Z)$. These assumptions are separated because interpretations are more easily proved, if the conclusions that follow from the *semivalid-implem* assumptions are already available.

locale *valid-implem* = *semivalid-implem* +

— Synthesize assumptions: conditions on payloads M implied by derivable
— channel messages with payload M.
assumes *implInsec-synth-analz*:
$H \subseteq payload \cup valid \cup range\ LtK \cup Tags \Longrightarrow$
$implInsec\ A\ B\ M \in synth\ (analz\ H) \Longrightarrow$
$implInsec\ A\ B\ M \in H \vee M \in synth\ (analz\ H)$
and *implConfid-synth-analz*:
$H \subseteq payload \cup valid \cup range\ LtK \cup Tags \Longrightarrow$
$implConfid\ A\ B\ M \in synth\ (analz\ H) \Longrightarrow$
$implConfid\ A\ B\ M \in H \vee M \in synth\ (analz\ H)$
and *implAuth-synth-analz*:
$H \subseteq payload \cup valid \cup range\ LtK \cup Tags \Longrightarrow$
$implAuth\ A\ B\ M \in synth\ (analz\ H) \Longrightarrow$
$implAuth\ A\ B\ M \in H \vee (M \in synth\ (analz\ H) \wedge (A, B) \in broken\ H)$
and *implSecure-synth-analz*:
$H \subseteq payload \cup valid \cup range\ LtK \cup Tags \Longrightarrow$
$implSecure\ A\ B\ M \in synth\ (analz\ H) \Longrightarrow$
$implSecure\ A\ B\ M \in H \vee (M \in synth\ (analz\ H) \wedge (A, B) \in broken\ H)$

end

13 Lemmas Following from Channel Message Implementation Assumptions

theory *Implem-lemmas*
imports *Implem*
begin

These lemmas require the assumptions added in the *valid-implem* locale.

context *semivalid-implem*
begin

13.1 Message implementations and abstractions

Abstracting a set of messages into channel messages.

definition
  $abs :: msg\ set \Rightarrow chan\ set$
where
  $abs\ S \equiv \{Chan\ x\ A\ B\ M \mid x\ A\ B\ M.\ M \in payload \wedge implem\ (Chan\ x\ A\ B\ M) \in S\}$

lemma *absE* [*elim*]:
  $[\![\ X \in abs\ H;$
    $\bigwedge x\ A\ B\ M.\ X = Chan\ x\ A\ B\ M \Longrightarrow M \in payload \Longrightarrow implem\ X \in H \Longrightarrow P\ ]\!]$
  $\Longrightarrow P$
⟨*proof*⟩

lemma *absI* [*intro*]: $M \in payload \Longrightarrow implem\ (Chan\ x\ A\ B\ M) \in H \Longrightarrow Chan\ x\ A\ B\ M \in abs\ H$
⟨*proof*⟩

lemma *abs-mono*: $G \subseteq H \Longrightarrow abs\ G \subseteq abs\ H$
⟨*proof*⟩

lemmas *abs-monotone* [*simp*] = *abs-mono* [*THEN* [*2*] *rev-subsetD*]

lemma *abs-empty* [*simp*]: $abs\ \{\} = \{\}$
⟨*proof*⟩

lemma *abs-Un-eq*: $abs\ (G \cup H) = abs\ G \cup abs\ H$
⟨*proof*⟩

General lemmas about implementations and *local.abs*.

lemma *abs-insert-payload* [*simp*]: $M \in payload \Longrightarrow abs\ (insert\ M\ S) = abs\ S$
⟨*proof*⟩

lemma *abs-insert-impl* [*simp*]:
  $M \in payload \Longrightarrow abs\ (insert\ (implem\ (Chan\ x\ A\ B\ M))\ S) = insert\ (Chan\ x\ A\ B\ M)\ (abs\ S)$
⟨*proof*⟩

lemma *extr-payload* [*simp*, *intro*]:
  $[\![\ X \in extr\ Bad\ NI\ (abs\ I);\ NI \subseteq payload\ ]\!] \Longrightarrow X \in payload$
⟨*proof*⟩

lemma *abs-Un-LtK*:
$K \subseteq range\ LtK \Longrightarrow abs\ (K \cup S) = abs\ S$
⟨*proof*⟩

lemma *abs-Un-keys-of* [*simp*]:
$abs\ (keys\text{-}of\ A \cup S) = abs\ S$
⟨*proof*⟩

Lemmas about *valid* and *local.abs*

lemma *abs-validSet*: $abs\ (S \cap valid) = abs\ S$
⟨*proof*⟩

lemma *valid-abs*: $M \in valid \Longrightarrow \exists\ M'.\ M' \in abs\ \{M\}$
⟨*proof*⟩

13.2 Extractable messages

extractable K I: subset of messages in *I* which are implementations (not necessarily valid since we do not require that they are payload) and can be extracted using the keys in K. It corresponds to L2 *extr*.

definition
$extractable :: msg\ set \Rightarrow msg\ set \Rightarrow msg\ set$
where
$extractable\ K\ I \equiv$
$\{implInsec\ A\ B\ M \mid A\ B\ M.\ implInsec\ A\ B\ M \in I\} \cup$
$\{implAuth\ A\ B\ M \mid A\ B\ M.\ implAuth\ A\ B\ M \in I\} \cup$
$\{implConfid\ A\ B\ M \mid A\ B\ M.\ implConfid\ A\ B\ M \in I \wedge (A, B) \in broken\ K\} \cup$
$\{implSecure\ A\ B\ M \mid A\ B\ M.\ implSecure\ A\ B\ M \in I \wedge (A, B) \in broken\ K\}$

lemma *extractable-red*: $extractable\ K\ I \subseteq I$
⟨*proof*⟩

lemma *extractableI*:
$implem\ (Chan\ x\ A\ B\ M) \in I \Longrightarrow$
$x = insec \vee x = auth \vee ((x = confid \vee x = secure) \wedge (A, B) \in broken\ K) \Longrightarrow$
$implem\ (Chan\ x\ A\ B\ M) \in extractable\ K\ I$
⟨*proof*⟩

lemma *extractableE*:
$X \in extractable\ K\ I \Longrightarrow$
$(\bigwedge A\ B\ M.\ X = implInsec\ A\ B\ M \Longrightarrow X \in I \Longrightarrow P) \Longrightarrow$
$(\bigwedge A\ B\ M.\ X = implAuth\ A\ B\ M \Longrightarrow X \in I \Longrightarrow P) \Longrightarrow$
$(\bigwedge A\ B\ M.\ X = implConfid\ A\ B\ M \Longrightarrow X \in I \Longrightarrow (A, B) \in broken\ K \Longrightarrow P) \Longrightarrow$
$(\bigwedge A\ B\ M.\ X = implSecure\ A\ B\ M \Longrightarrow X \in I \Longrightarrow (A, B) \in broken\ K \Longrightarrow P) \Longrightarrow$
P
⟨*proof*⟩

General lemmas about implementations and extractable.

lemma *implem-extractable* [*simp*]:
$\llbracket\ Keys\text{-}bad\ K\ Bad;\ implem\ (Chan\ x\ A\ B\ M) \in extractable\ K\ I;\ M \in payload\ \rrbracket$
$\Longrightarrow M \in extr\ Bad\ NI\ (abs\ I)$

$\langle proof \rangle$

Auxiliary lemmas about extractable messages: they are implementations.

lemma *valid-extractable* [*simp*]: $I \subseteq valid \Longrightarrow extractable\ K\ I \subseteq valid$
$\langle proof \rangle$

lemma *LtKeys-parts-extractable*:
$I \subseteq valid \Longrightarrow parts\ (extractable\ K\ I) \cap range\ LtK = \{\}$
$\langle proof \rangle$

lemma *LtKeys-parts-extractable-elt* [*simp*]:
$I \subseteq valid \Longrightarrow LtK\ ltk \notin parts\ (extractable\ K\ I)$
$\langle proof \rangle$

lemma *LtKeys-parts-implSecureSet*:
$parts\ (implSecureSet\ Ag\ payload) \cap range\ LtK = \{\}$
$\langle proof \rangle$

lemma *LtKeys-parts-implSecureSet-elt*:
$LtK\ K \notin parts\ (implSecureSet\ Ag\ payload)$
$\langle proof \rangle$

lemmas *LtKeys-parts* = *LtKeys-parts-payload parts-valid-LtKeys-disjoint*
LtKeys-parts-extractable LtKeys-parts-implSecureSet
LtKeys-parts-implSecureSet-elt

13.2.1 Partition I to keep only the extractable messages

Partition the implementation set.

lemma *impl-partition*:
$[\![\ NI \subseteq payload;\ I \subseteq valid\ ]\!] \Longrightarrow$
$I \subseteq extractable\ K\ I\ \cup$
$implConfidSet\ (UNIV - broken\ K)\ payload\ \cup$
$implSecureSet\ (UNIV - broken\ K)\ payload$
$\langle proof \rangle$

13.2.2 Partition of *extractable*

We partition the *extractable* set into insecure, confidential, authentic implementations.

lemma *extractable-partition*:
$[\![Keys\text{-}bad\ K\ Bad;\ NI \subseteq payload;\ I \subseteq valid]\!] \Longrightarrow$
$extractable\ K\ I \subseteq$
$implInsecSet\ (extr\ Bad\ NI\ (abs\ I))\ \cup$
$implConfidSet\ UNIV\ (extr\ Bad\ NI\ (abs\ I))\ \cup$
$implAuthSet\ (extr\ Bad\ NI\ (abs\ I))\ \cup$
$implSecureSet\ UNIV\ (extr\ Bad\ NI\ (abs\ I))$
$\langle proof \rangle$

13.3 Lemmas for proving intruder refinement (L2-L3)

Chain of lemmas used to prove the refinement for *l3-dy*. The ultimate goal is to show

synth (*analz* (*NI* ∪ *I* ∪ *K* ∪ *Tags*))
⊆ *synth* (*analz* (*extr Bad NI* (*local.abs I*))) ∪ − *payload*

13.3.1 First: we only keep the extractable messages

lemma *analz-NI-I-K-analz-NI-EI*:
assumes *HNI*: *NI* ⊆ *payload*
  and *HK*: *K* ⊆ *range LtK*
  and *HI*: *I* ⊆ *valid*
shows *analz* (*NI* ∪ *I* ∪ *K* ∪ *Tags*) ⊆
    synth (*analz* (*NI* ∪ *extractable K I* ∪ *K* ∪ *Tags*)) ∪ −*payload*
⟨*proof*⟩

13.3.2 Only keep the extracted messages (instead of extractable)

lemma *analz-NI-EI-K-synth-analz-NI-E-K*:
assumes *HNI*: *NI* ⊆ *payload*
  and *HK*: *K* ⊆ *range LtK*
  and *HI*: *I* ⊆ *valid*
  and *Hbad*: *Keys-bad K Bad*
shows *analz* (*NI* ∪ *extractable K I* ∪ *K* ∪ *Tags*)
  ⊆ *synth* (*analz* (*extr Bad NI* (*abs I*) ∪ *K* ∪ *Tags*)) ∪ −*payload*
⟨*proof*⟩

13.3.3 Keys and Tags can be moved out of the *analz*

lemma *analz-LtKeys-Tag*:
assumes *NI* ⊆ *payload* **and** *K* ⊆ *range LtK*
shows *analz* (*NI* ∪ *K* ∪ *Tags*) ⊆ *analz NI* ∪ *K* ∪ *Tags*
⟨*proof*⟩

lemma *analz-NI-E-K-analz-NI-E*:
 ⟦ *NI* ⊆ *payload*; *K* ⊆ *range LtK*; *I* ⊆ *valid* ⟧
 ⟹ *analz* (*extr Bad NI* (*abs I*) ∪ *K* ∪ *Tags*) ⊆ *analz* (*extr Bad NI* (*abs I*)) ∪ *K* ∪ *Tags*
⟨*proof*⟩

13.3.4 Final lemmas, using all the previous ones

lemma *analz-NI-I-K-synth-analz-NI-E*:
assumes
 Hbad: *Keys-bad K Bad* **and**
 HNI: *NI* ⊆ *payload* **and**
 HK: *K* ⊆ *range LtK* **and**
 HI: *I* ⊆ *valid*
shows
 analz (*NI* ∪ *I* ∪ *K* ∪ *Tags*) ⊆ *synth* (*analz* (*extr Bad NI* (*abs I*))) ∪ −*payload*
⟨*proof*⟩

Lemma actually used to prove the refinement.

lemma *synth-analz-NI-I-K-synth-analz-NI-E*:
 ⟦ *Keys-bad K Bad*; *NI* ⊆ *payload*; *K* ⊆ *range LtK*; *I* ⊆ *valid*⟧
 ⟹ *synth* (*analz* (*NI* ∪ *I* ∪ *K* ∪ *Tags*))

$\subseteq$ *synth* (*analz* (*extr Bad NI* (*abs I*))) $\cup$ $-$*payload*
⟨*proof*⟩

13.3.5 Partitioning *analz ik*

Two lemmas useful for proving the invariant

analz ik $\subseteq$ *synth* (*analz* (*ik* $\cap$ *payload* $\cup$ *ik* $\cap$ *valid* $\cup$ *ik* $\cap$ *range LtK* $\cup$ *Tags*))

lemma *analz-Un-partition*:
  analz S $\subseteq$ *synth* (*analz* ((*S* $\cap$ *payload*) $\cup$ (*S* $\cap$ *valid*) $\cup$ (*S* $\cap$ *range LtK*) $\cup$ *Tags*)) $\Longrightarrow$
  H $\subseteq$ *payload* $\cup$ *valid* $\cup$ *range LtK* $\Longrightarrow$
  analz (*H* $\cup$ *S*) $\subseteq$
    synth (*analz* (((*H* $\cup$ *S*) $\cap$ *payload*) $\cup$ ((*H* $\cup$ *S*) $\cap$ *valid*) $\cup$ ((*H* $\cup$ *S*) $\cap$ *range LtK*) $\cup$ *Tags*))
⟨*proof*⟩

lemma *analz-insert-partition*:
  analz S $\subseteq$ *synth* (*analz* ((*S* $\cap$ *payload*) $\cup$ (*S* $\cap$ *valid*) $\cup$ (*S* $\cap$ *range LtK*) $\cup$ *Tags*)) $\Longrightarrow$
  x $\in$ *payload* $\cup$ *valid* $\cup$ *range LtK* $\Longrightarrow$
  analz (*insert x S*) $\subseteq$
    synth (*analz* (((*insert x S*) $\cap$ *payload*) $\cup$ ((*insert x S*) $\cap$ *valid*) $\cup$
        ((*insert x S*) $\cap$ *range LtK*) $\cup$ *Tags*))
⟨*proof*⟩

end
end

14 Symmetric Implementation of Channel Messages

theory *Implem-symmetric*
imports *Implem*
begin

14.1 Implementation of channel messages

fun *implem-sym* :: *chan* ⇒ *msg* **where**
implem-sym (*Insec A B M*) = ⟨*InsecTag*, *Agent A*, *Agent B*, *M*⟩
|*implem-sym* (*Confid A B M*) = *Enc* ⟨*ConfidTag*, *M*⟩ (*shrK A B*)
|*implem-sym* (*Auth A B M*) = ⟨*M*, *hmac* ⟨*AuthTag*, *M*⟩ (*shrK A B*)⟩
|*implem-sym* (*Secure A B M*) = *Enc* ⟨*SecureTag*, *M*⟩ (*shrK A B*)

First step: *basic-implem*. Trivial as there are no assumption, this locale just defines some useful abbreviations and valid.

interpretation *sym*: *basic-implem implem-sym*
⟨*proof*⟩

Second step: *semivalid-implem*. Here we prove some basic properties such as injectivity and some properties about the interaction of sets of implementation messages with *analz*; these properties are proved as separate lemmas as the proofs are more complex.

Auxiliary: simpler definitions of the *implSets* for the proofs, using the *msgSet* definitions.

abbreviation *implInsecSet-aux* :: *msg set* ⇒ *msg set* **where**
implInsecSet-aux G ≡ *PairSet Tags* (*PairSet* (*range Agent*) (*PairSet* (*range Agent*) *G*))

abbreviation *implAuthSet-aux* :: *msg set* ⇒ *msg set* **where**
implAuthSet-aux G ≡ *PairSet G* (*HashSet* (*PairSet* (*PairSet Tags G*) (*range* (*case-prod shrK*))))

abbreviation *implConfidSet-aux* :: (*agent* ∗ *agent*) *set* ⇒ *msg set* ⇒ *msg set* **where**
implConfidSet-aux Ag G ≡ *EncSet* (*PairSet Tags G*) (*case-prod shrK'Ag*)

abbreviation *implSecureSet-aux* :: (*agent* ∗ *agent*) *set* ⇒ *msg set* ⇒ *msg set* **where**
implSecureSet-aux Ag G ≡ *EncSet* (*PairSet Tags G*) (*case-prod shrK'Ag*)

These auxiliary definitions are overapproximations.

lemma *implInsecSet-implInsecSet-aux*: *sym.implInsecSet G* ⊆ *implInsecSet-aux G*
⟨*proof*⟩

lemma *implAuthSet-implAuthSet-aux*: *sym.implAuthSet G* ⊆ *implAuthSet-aux G*
⟨*proof*⟩

lemma *implConfidSet-implConfidSet-aux*: *sym.implConfidSet Ag G* ⊆ *implConfidSet-aux Ag G*
⟨*proof*⟩

lemma *implSecureSet-implSecureSet-aux*: *sym.implSecureSet Ag G* ⊆ *implSecureSet-aux Ag G*
⟨*proof*⟩

lemmas *implSet-implSet-aux* =
implInsecSet-implInsecSet-aux implAuthSet-implAuthSet-aux
implConfidSet-implConfidSet-aux implSecureSet-implSecureSet-aux

declare *Enc-keys-clean-msgSet-Un* [*intro*]

14.2 Lemmas to pull implementation sets out of *analz*

All these proofs are similar:

1. prove the lemma for the *implSet-aux* and with the set added outside of *analz* given explicitly,
2. prove the lemma for the *implSet-aux* but with payload, and
3. prove the lemma for the *implSet*.

There are two cases for the confidential and secure messages: the general case (the payloads stay in *analz*) and the case where the key is unknown (the messages cannot be opened and are completely removed from the *analz*).

14.2.1 Pull *implInsecSet* out of *analz*

lemma *analz-Un-implInsecSet-aux-1*:
  Enc-keys-clean (*G* ∪ *H*) ⟹
   analz (*implInsecSet-aux G* ∪ *H*) ⊆
    implInsecSet-aux G ∪ *Tags* ∪
    PairSet (*range Agent*) (*PairSet* (*range Agent*) *G*) ∪
    PairSet (*range Agent*) *G* ∪
    analz (*range Agent* ∪ *G* ∪ (*range Agent* ∪ *H*))
⟨*proof*⟩

lemma *analz-Un-implInsecSet-aux-2*:
  Enc-keys-clean (*G* ∪ *H*) ⟹
   analz (*implInsecSet-aux G* ∪ *H*) ⊆
    implInsecSet-aux G ∪ *Tags* ∪
    synth (*analz* (*G* ∪ *H*))
⟨*proof*⟩

lemma *analz-Un-implInsecSet-aux-3*:
  Enc-keys-clean (*G* ∪ *H*) ⟹
   analz (*implInsecSet-aux G* ∪ *H*) ⊆ *synth* (*analz* (*G* ∪ *H*)) ∪ −*payload*
⟨*proof*⟩

lemma *analz-Un-implInsecSet*:
  Enc-keys-clean (*G* ∪ *H*) ⟹
   analz (*sym.implInsecSet G* ∪ *H*) ⊆ *synth* (*analz* (*G* ∪ *H*)) ∪ −*payload*
⟨*proof*⟩

14.3 Pull *implConfidSet* out of *analz*

lemma *analz-Un-implConfidSet-aux-1*:
  Enc-keys-clean (*G* ∪ *H*) ⟹
   analz (*implConfidSet-aux Ag G* ∪ *H*) ⊆
    implConfidSet-aux Ag G ∪ *PairSet Tags G* ∪ *Tags* ∪

analz $(G \cup H)$
⟨*proof*⟩

lemma *analz-Un-implConfidSet-aux-2*:
Enc-keys-clean $(G \cup H) \Longrightarrow$
analz (*implConfidSet-aux Ag G* $\cup$ *H*) $\subseteq$
implConfidSet-aux Ag G $\cup$ *PairSet Tags G* $\cup$ *Tags* $\cup$
synth (*analz* $(G \cup H)$)
⟨*proof*⟩

lemma *analz-Un-implConfidSet-aux-3*:
Enc-keys-clean $(G \cup H) \Longrightarrow$
analz (*implConfidSet-aux Ag G* $\cup$ *H*) $\subseteq$ *synth* (*analz* $(G \cup H)$) $\cup$ $-$*payload*
⟨*proof*⟩

lemma *analz-Un-implConfidSet*:
Enc-keys-clean $(G \cup H) \Longrightarrow$
analz (*sym.implConfidSet Ag G* $\cup$ *H*) $\subseteq$ *synth* (*analz* $(G \cup H)$) $\cup$ $-$*payload*
⟨*proof*⟩

Pull *implConfidSet* out of *analz*, 2nd case where the agents are honest.

lemma *analz-Un-implConfidSet-2-aux-1*:
Enc-keys-clean H $\Longrightarrow$
Ag $\cap$ *broken* (*parts H* $\cap$ *range LtK*) $= \{\} \Longrightarrow$
analz (*implConfidSet-aux Ag G* $\cup$ *H*) $\subseteq$ *implConfidSet-aux Ag G* $\cup$ *synth* (*analz H*)
⟨*proof*⟩

lemma *analz-Un-implConfidSet-2-aux-3*:
Enc-keys-clean H $\Longrightarrow$
Ag $\cap$ *broken* (*parts H* $\cap$ *range LtK*) $= \{\} \Longrightarrow$
analz (*implConfidSet-aux Ag G* $\cup$ *H*) $\subseteq$ *synth* (*analz H*) $\cup$ $-$*payload*
⟨*proof*⟩

lemma *analz-Un-implConfidSet-2*:
Enc-keys-clean H $\Longrightarrow$
Ag $\cap$ *broken* (*parts H* $\cap$ *range LtK*) $= \{\} \Longrightarrow$
analz (*sym.implConfidSet Ag G* $\cup$ *H*) $\subseteq$ *synth* (*analz H*) $\cup$ $-$*payload*
⟨*proof*⟩

14.4 Pull *implSecureSet* out of *analz*

lemma *analz-Un-implSecureSet-aux-1*:
Enc-keys-clean $(G \cup H) \Longrightarrow$
analz (*implSecureSet-aux Ag G* $\cup$ *H*) $\subseteq$
implSecureSet-aux Ag G $\cup$ *PairSet Tags G* $\cup$ *Tags* $\cup$
analz $(G \cup H)$
⟨*proof*⟩

lemma *analz-Un-implSecureSet-aux-2*:
Enc-keys-clean $(G \cup H) \Longrightarrow$
analz (*implSecureSet-aux Ag G* $\cup$ *H*) $\subseteq$
implSecureSet-aux Ag G $\cup$ *PairSet Tags G* $\cup$ *Tags* $\cup$
synth (*analz* $(G \cup H)$)

⟨*proof*⟩

lemma *analz-Un-implSecureSet-aux-3*:
Enc-keys-clean $(G \cup H) \Longrightarrow$
analz (*implSecureSet-aux Ag G* $\cup$ *H*) $\subseteq$ *synth* (*analz* $(G \cup H)$) $\cup$ $-$*payload*
⟨*proof*⟩

lemma *analz-Un-implSecureSet*:
Enc-keys-clean $(G \cup H) \Longrightarrow$
analz (*sym.implSecureSet Ag G* $\cup$ *H*) $\subseteq$ *synth* (*analz* $(G \cup H)$) $\cup$ $-$*payload*
⟨*proof*⟩

Pull *implSecureSet* out of *analz*, 2nd case, where the agents are honest.

lemma *analz-Un-implSecureSet-2-aux-1*:
Enc-keys-clean H $\Longrightarrow$
Ag $\cap$ *broken* (*parts H* $\cap$ *range LtK*) = {} $\Longrightarrow$
analz (*implSecureSet-aux Ag G* $\cup$ *H*) $\subseteq$ *implSecureSet-aux Ag G* $\cup$ *synth* (*analz H*)
⟨*proof*⟩

lemma *analz-Un-implSecureSet-2-aux-3*:
Enc-keys-clean H $\Longrightarrow$
Ag $\cap$ *broken* (*parts H* $\cap$ *range LtK*) = {} $\Longrightarrow$
analz (*implSecureSet-aux Ag G* $\cup$ *H*) $\subseteq$ *synth* (*analz H*) $\cup$ $-$*payload*
⟨*proof*⟩

lemma *analz-Un-implSecureSet-2*:
Enc-keys-clean H $\Longrightarrow$
Ag $\cap$ *broken* (*parts H* $\cap$ *range LtK*) = {} $\Longrightarrow$
analz (*sym.implSecureSet Ag G* $\cup$ *H*) $\subseteq$ *synth* (*analz H*) $\cup$ $-$*payload*
⟨*proof*⟩

14.5 Pull *implAuthSet* out of *analz*

lemma *analz-Un-implAuthSet-aux-1*:
Enc-keys-clean $(G \cup H) \Longrightarrow$
analz (*implAuthSet-aux G* $\cup$ *H*) $\subseteq$
implAuthSet-aux G $\cup$ *HashSet* (*PairSet* (*PairSet Tags G*) (*range* (*case-prod shrK*))) $\cup$
analz $(G \cup H)$
⟨*proof*⟩

lemma *analz-Un-implAuthSet-aux-2*:
Enc-keys-clean $(G \cup H) \Longrightarrow$
analz (*implAuthSet-aux G* $\cup$ *H*) $\subseteq$
implAuthSet-aux G $\cup$ *HashSet* (*PairSet* (*PairSet Tags G*) (*range* (*case-prod shrK*))) $\cup$
synth (*analz* $(G \cup H)$)
⟨*proof*⟩

lemma *analz-Un-implAuthSet-aux-3*:
Enc-keys-clean $(G \cup H) \Longrightarrow$
analz (*implAuthSet-aux G* $\cup$ *H*) $\subseteq$ *synth* (*analz* $(G \cup H)$) $\cup$ $-$*payload*
⟨*proof*⟩

lemma *analz-Un-implAuthSet*:

Enc-keys-clean $(G \cup H) \Longrightarrow$
analz $($*sym.implAuthSet* $G \cup H) \subseteq$ *synth* $($*analz* $(G \cup H)) \cup -$*payload*
⟨*proof*⟩

declare *Enc-keys-clean-msgSet-Un* [*rule del*]

14.6 Locale interpretations

interpretation *sym*: *semivalid-implem implem-sym*
⟨*proof*⟩

Third step: *valid-implem*. The lemmas giving conditions on M, A and B for *implXXX A B M* $\in$ *synth* $($*analz* $Z)$.

lemma *implInsec-synth-analz*:
$H \subseteq$ *payload* $\cup$ *sym.valid* $\cup$ *range LtK* $\cup$ *Tags* $\Longrightarrow$
sym.implInsec A B M $\in$ *synth* $($*analz* $H) \Longrightarrow$
sym.implInsec A B M $\in H \vee M \in$ *synth* $($*analz* $H)$
⟨*proof*⟩

lemma *implConfid-synth-analz*:
$H \subseteq$ *payload* $\cup$ *sym.valid* $\cup$ *range LtK* $\cup$ *Tags* $\Longrightarrow$
sym.implConfid A B M $\in$ *synth* $($*analz* $H) \Longrightarrow$
sym.implConfid A B M $\in H \vee M \in$ *synth* $($*analz* $H)$
⟨*proof*⟩

lemma *implAuth-synth-analz*:
$H \subseteq$ *payload* $\cup$ *sym.valid* $\cup$ *range LtK* $\cup$ *Tags* $\Longrightarrow$
sym.implAuth A B M $\in$ *synth* $($*analz* $H) \Longrightarrow$
sym.implAuth A B M $\in H \vee (M \in$ *synth* $($*analz* $H) \wedge (A, B) \in$ *broken* $H)$
⟨*proof*⟩

lemma *implSecure-synth-analz*:
$H \subseteq$ *payload* $\cup$ *sym.valid* $\cup$ *range LtK* $\cup$ *Tags* $\Longrightarrow$
sym.implSecure A B M $\in$ *synth* $($*analz* $H) \Longrightarrow$
sym.implSecure A B M $\in H \vee (M \in$ *synth* $($*analz* $H) \wedge (A, B) \in$ *broken* $H)$
⟨*proof*⟩

interpretation *sym*: *valid-implem implem-sym*
⟨*proof*⟩

end

15 Asymmetric Implementation of Channel Messages

theory *Implem-asymmetric*
imports *Implem*
begin

15.1 Implementation of channel messages

fun *implem-asym* :: *chan* ⇒ *msg* **where**
implem-asym (*Insec A B M*) = ⟨*InsecTag*, *Agent A*, *Agent B*, *M*⟩
|*implem-asym* (*Confid A B M*) = *Aenc* ⟨*Agent A*, *M*⟩ (*pubK B*)
|*implem-asym* (*Auth A B M*) = *Sign* ⟨*Agent B*, *M*⟩ (*priK A*)
|*implem-asym* (*Secure A B M*) = *Sign* (*Aenc* ⟨*SecureTag*, *Agent A*, *M*⟩ (*pubK B*)) (*priK A*)

First step: *basic-implem*. Trivial as there are no assumption, this locale just defines some useful abbreviations and valid.

interpretation *asym*: *basic-implem implem-asym*
⟨*proof*⟩

Second step: *semivalid-implem*. Here we prove some basic properties such as injectivity and some properties about the interaction of sets of implementation messages with *analz*; these properties are proved as separate lemmas as the proofs are more complex.

Auxiliary: simpler definitions of the *implSets* for the proofs, using the *msgSet* definitions.

abbreviation *implInsecSet-aux* :: *msg set* ⇒ *msg set*
where *implInsecSet-aux G* ≡ *PairSet Tags* (*PairSet AgentSet* (*PairSet AgentSet G*))

abbreviation *implAuthSet-aux* :: *msg set* ⇒ *msg set*
where *implAuthSet-aux G* ≡ *SignSet* (*PairSet AgentSet G*) (*range priK*)

abbreviation *implConfidSet-aux* :: (*agent* ∗ *agent*) *set* ⇒ *msg set* ⇒ *msg set*
where *implConfidSet-aux Ag G* ≡ *AencSet* (*PairSet AgentSet G*) (*pubK*‘ (*Ag* ‘‘ *UNIV*))

abbreviation *implSecureSet-aux* :: (*agent* ∗ *agent*) *set* ⇒ *msg set* ⇒ *msg set*
where *implSecureSet-aux Ag G* ≡
SignSet (*AencSet* (*PairSet Tags* (*PairSet AgentSet G*)) (*pubK*‘ (*Ag* ‘‘ *UNIV*))) (*range priK*)

These auxiliary definitions are overapproximations.

lemma *implInsecSet-implInsecSet-aux*: *asym.implInsecSet G* ⊆ *implInsecSet-aux G*
⟨*proof*⟩

lemma *implAuthSet-implAuthSet-aux*: *asym.implAuthSet G* ⊆ *implAuthSet-aux G*
⟨*proof*⟩

lemma *implConfidSet-implConfidSet-aux*: *asym.implConfidSet Ag G* ⊆ *implConfidSet-aux Ag G*
⟨*proof*⟩

lemma *implSecureSet-implSecureSet-aux*: *asym.implSecureSet Ag G* ⊆ *implSecureSet-aux Ag G*
⟨*proof*⟩

lemmas *implSet-implSet-aux* =
implInsecSet-implInsecSet-aux implAuthSet-implAuthSet-aux

implConfidSet-implConfidSet-aux implSecureSet-implSecureSet-aux

declare *Enc-keys-clean-msgSet-Un* [*intro*]

15.2 Lemmas to pull implementation sets out of *analz*

All these proofs are similar:

1. prove the lemma for the *implSet-aux* and with the set added outside of *analz* given explicitly,
2. prove the lemma for the *implSet-aux* but with payload, and
3. prove the lemma for the *implSet*.

There are two cases for the confidential and secure messages: the general case (the payloads stay in *analz*) and the case where the key is unknown (the messages cannot be opened and are completely removed from the *analz*).

15.2.1 Pull *PairAgentSet* out of *analz*

lemma *analz-Un-PairAgentSet*:
shows
$analz\ (PairSet\ AgentSet\ G \cup H) \subseteq PairSet\ AgentSet\ G \cup AgentSet \cup analz\ (G \cup H)$
⟨*proof*⟩

15.2.2 Pull *implInsecSet* out of *analz*

lemma *analz-Un-implInsecSet-aux-aux*:
assumes $Enc\text{-}keys\text{-}clean\ (G \cup H)$
shows $analz\ (implInsecSet\text{-}aux\ G \cup H) \subseteq implInsecSet\text{-}aux\ G \cup Tags \cup synth\ (analz\ (G \cup H))$
⟨*proof*⟩

lemma *analz-Un-implInsecSet-aux*:
$Enc\text{-}keys\text{-}clean\ (G \cup H) \Longrightarrow$
$analz\ (implInsecSet\text{-}aux\ G \cup H) \subseteq synth\ (analz\ (G \cup H)) \cup -payload$
⟨*proof*⟩

lemma *analz-Un-implInsecSet*:
$Enc\text{-}keys\text{-}clean\ (G \cup H) \Longrightarrow$
$analz\ (asym.implInsecSet\ G \cup H) \subseteq synth\ (analz\ (G \cup H)) \cup -payload$
⟨*proof*⟩

15.3 Pull *implConfidSet* out of *analz*

lemma *analz-Un-implConfidSet-aux-aux*:
$Enc\text{-}keys\text{-}clean\ (G \cup H) \Longrightarrow$
$analz\ (implConfidSet\text{-}aux\ Ag\ G \cup H) \subseteq$
$implConfidSet\text{-}aux\ Ag\ G \cup PairSet\ AgentSet\ G \cup$
$synth\ (analz\ (G \cup H))$
⟨*proof*⟩

lemma *analz-Un-implConfidSet-aux*:
Enc-keys-clean $(G \cup H) \Longrightarrow$
analz (*implConfidSet-aux Ag G* $\cup$ *H*) $\subseteq$ *synth* (*analz* $(G \cup H)$) $\cup$ $-$*payload*
⟨*proof*⟩

lemma *analz-Un-implConfidSet*:
Enc-keys-clean $(G \cup H) \Longrightarrow$
analz (*asym.implConfidSet Ag G* $\cup$ *H*) $\subseteq$ *synth* (*analz* $(G \cup H)$) $\cup$ $-$*payload*
⟨*proof*⟩

Pull *implConfidSet* out of *analz*, 2nd case where the agents are honest.

lemma *analz-Un-implConfidSet-aux-aux-2*:
Enc-keys-clean H $\Longrightarrow$
Ag $\cap$ *broken* (*parts H* $\cap$ *range LtK*) $= \{\} \Longrightarrow$
analz (*implConfidSet-aux Ag G* $\cup$ *H*) $\subseteq$ *implConfidSet-aux Ag G* $\cup$ *synth* (*analz H*)
⟨*proof*⟩

lemma *analz-Un-implConfidSet-aux-2*:
Enc-keys-clean H $\Longrightarrow$
Ag $\cap$ *broken* (*parts H* $\cap$ *range LtK*) $= \{\} \Longrightarrow$
analz (*implConfidSet-aux Ag G* $\cup$ *H*) $\subseteq$ *synth* (*analz H*) $\cup$ $-$*payload*
⟨*proof*⟩

lemma *analz-Un-implConfidSet-2*:
Enc-keys-clean H $\Longrightarrow$
Ag $\cap$ *broken* (*parts H* $\cap$ *range LtK*) $= \{\} \Longrightarrow$
analz (*asym.implConfidSet Ag G* $\cup$ *H*) $\subseteq$ *synth* (*analz H*) $\cup$ $-$*payload*
⟨*proof*⟩

15.4 Pull *implAuthSet* out of *analz*

lemma *analz-Un-implAuthSet-aux-aux*:
Enc-keys-clean $(G \cup H) \Longrightarrow$
analz (*implAuthSet-aux G* $\cup$ *H*) $\subseteq$ *implAuthSet-aux G* $\cup$ *synth* (*analz* $(G \cup H)$)
⟨*proof*⟩

lemma *analz-Un-implAuthSet-aux*:
Enc-keys-clean $(G \cup H) \Longrightarrow$
analz (*implAuthSet-aux G* $\cup$ *H*) $\subseteq$ *synth* (*analz* $(G \cup H)$) $\cup$ $-$*payload*
⟨*proof*⟩

lemma *analz-Un-implAuthSet*:
Enc-keys-clean $(G \cup H) \Longrightarrow$
analz (*asym.implAuthSet G* $\cup$ *H*) $\subseteq$ *synth* (*analz* $(G \cup H)$) $\cup$ $-$*payload*
⟨*proof*⟩

15.5 Pull *implSecureSet* out of *analz*

lemma *analz-Un-implSecureSet-aux-aux*:
Enc-keys-clean $(G \cup H) \Longrightarrow$
analz (*implSecureSet-aux Ag G* $\cup$ *H*) $\subseteq$
implSecureSet-aux Ag G $\cup$ *AencSet* (*PairSet Tags* (*PairSet AgentSet G*)) (*pubK*‘ (*Ag*“ *UNIV*)) $\cup$
PairSet Tags (*PairSet AgentSet G*) $\cup$ *Tags* $\cup$ *PairSet AgentSet G* $\cup$

synth (*analz* ($G \cup H$))
⟨*proof*⟩

lemma *analz-Un-implSecureSet-aux*:
Enc-keys-clean ($G \cup H$) $\Longrightarrow$
analz (*implSecureSet-aux Ag G* $\cup$ *H*) $\subseteq$ *synth* (*analz* ($G \cup H$)) $\cup$ $-$*payload*
⟨*proof*⟩

lemma *analz-Un-implSecureSet*:
Enc-keys-clean ($G \cup H$) $\Longrightarrow$
analz (*asym.implSecureSet Ag G* $\cup$ *H*) $\subseteq$ *synth* (*analz* ($G \cup H$)) $\cup$ $-$*payload*
⟨*proof*⟩

Pull *implSecureSet* out of *analz*, 2nd case, where the agents are honest.

lemma *analz-Un-implSecureSet-aux-aux-2*:
Enc-keys-clean ($G \cup H$) $\Longrightarrow$
Ag $\cap$ *broken* (*parts H* $\cap$ *range LtK*) = {} $\Longrightarrow$
analz (*implSecureSet-aux Ag G* $\cup$ *H*) $\subseteq$
implSecureSet-aux Ag G $\cup$ *AencSet* (*PairSet Tags* (*PairSet AgentSet G*)) (*pubK*‘ (*Ag*‘‘ *UNIV*)) $\cup$
synth (*analz H*)
⟨*proof*⟩

lemma *analz-Un-implSecureSet-aux-2*:
Enc-keys-clean ($G \cup H$) $\Longrightarrow$
Ag $\cap$ *broken* (*parts H* $\cap$ *range LtK*) = {} $\Longrightarrow$
analz (*implSecureSet-aux Ag G* $\cup$ *H*) $\subseteq$ *synth* (*analz H*) $\cup$ $-$*payload*
⟨*proof*⟩

lemma *analz-Un-implSecureSet-2*:
Enc-keys-clean ($G \cup H$) $\Longrightarrow$
Ag $\cap$ *broken* (*parts H* $\cap$ *range LtK*) = {} $\Longrightarrow$
analz (*asym.implSecureSet Ag G* $\cup$ *H*) $\subseteq$
synth (*analz H*) $\cup$ $-$*payload*
⟨*proof*⟩

declare *Enc-keys-clean-msgSet-Un* [*rule del*]

15.6 Locale interpretations

interpretation *asym*: *semivalid-implem implem-asym*
⟨*proof*⟩

Third step: *valid-implem*. The lemmas giving conditions on M, A and B for

implXXX A B M $\in$ *synth* (*analz Z*)

.

lemma *implInsec-synth-analz*:
H $\subseteq$ *payload* $\cup$ *asym.valid* $\cup$ *range LtK* $\cup$ *Tags* $\Longrightarrow$
asym.implInsec A B M $\in$ *synth* (*analz H*) $\Longrightarrow$
asym.implInsec A B M $\in$ *I* $\vee$ *M* $\in$ *synth* (*analz H*)
⟨*proof*⟩

lemma *implConfid-synth-analz*:
$H \subseteq payload \cup asym.valid \cup range\ LtK \cup Tags \Longrightarrow$
$asym.implConfid\ A\ B\ M \in synth\ (analz\ H) \Longrightarrow$
$asym.implConfid\ A\ B\ M \in H \vee M \in synth\ (analz\ H)$
⟨*proof*⟩

lemma *implAuth-synth-analz*:
$H \subseteq payload \cup asym.valid \cup range\ LtK \cup Tags \Longrightarrow$
$asym.implAuth\ A\ B\ M \in synth\ (analz\ H) \Longrightarrow$
$asym.implAuth\ A\ B\ M \in H \vee (M \in synth\ (analz\ H) \wedge (A, B) \in broken\ H)$
⟨*proof*⟩

lemma *implSecure-synth-analz*:
$H \subseteq payload \cup asym.valid \cup range\ LtK \cup Tags \Longrightarrow$
$asym.implSecure\ A\ B\ M \in synth\ (analz\ H) \Longrightarrow$
$asym.implSecure\ A\ B\ M \in H \vee (M \in synth\ (analz\ H) \wedge (A, B) \in broken\ H)$
⟨*proof*⟩

interpretation *asym*: *valid-implem implem-asym*
⟨*proof*⟩

end

16 Key Transport Protocol with PFS (L1)

theory *pfslvl1*
imports *Runs Secrecy AuthenticationI Payloads*
begin

declare *option.split-asm* [*split*]
declare *domIff* [*simp*, *iff del*]

16.1 State and Events

consts
sk :: *nat*
kE :: *nat*
Nend :: *nat*

Proofs break if *1* is used, because *simp* replaces it with *Suc 0*...

abbreviation
xpkE ≡ *Var 0*

abbreviation
xskE ≡ *Var 2*

abbreviation
xsk ≡ *Var 3*

abbreviation
xEnd ≡ *Var 4*

abbreviation
End ≡ *Number Nend*

domain of each role (protocol dependent)

fun *domain* :: *role-t* ⇒ *var set* **where**
domain Init = {*xpkE*, *xskE*, *xsk*}
| *domain Resp* = {*xpkE*, *xsk*}

consts
test :: *rid-t*

consts
guessed-runs :: *rid-t* ⇒ *run-t*
guessed-frame :: *rid-t* ⇒ *frame*

specification of the guessed frame

1. Domain

2. Well-typedness. The messages in the frame of a run never contain implementation material even if the agents of the run are dishonest. Therefore we consider only well-typed frames. This is notably required for the session key compromise; it also helps proving the partitionning of ik, since we know that the messages added by the protocol do not contain ltkeys in their payload and are therefore valid implementations.

3. We also ensure that the values generated by the frame owner are correctly guessed.

specification (*guessed-frame*)
guessed-frame-dom-spec [*simp*]:
$dom\ (guessed\text{-}frame\ R) = domain\ (role\ (guessed\text{-}runs\ R))$
guessed-frame-payload-spec [*simp*, *elim*]:
$guessed\text{-}frame\ R\ x = Some\ y \Longrightarrow y \in payload$
guessed-frame-Init-xpkE [*simp*]:
$role\ (guessed\text{-}runs\ R) = Init \Longrightarrow guessed\text{-}frame\ R\ xpkE = Some\ (epubKF\ (R\$kE))$
guessed-frame-Init-xskE [*simp*]:
$role\ (guessed\text{-}runs\ R) = Init \Longrightarrow guessed\text{-}frame\ R\ xskE = Some\ (epriKF\ (R\$kE))$
guessed-frame-Resp-xsk [*simp*]:
$role\ (guessed\text{-}runs\ R) = Resp \Longrightarrow guessed\text{-}frame\ R\ xsk = Some\ (NonceF\ (R\$sk))$
⟨*proof*⟩

abbreviation
$test\text{-}owner \equiv owner\ (guessed\text{-}runs\ test)$

abbreviation
$test\text{-}partner \equiv partner\ (guessed\text{-}runs\ test)$

level 1 state

record *l1-state* =
s0-state +
progress :: *progress-t*
signals :: *signal* ⇒ *nat*

type-synonym *l1-obs* = *l1-state*

abbreviation
run-ended :: *var set option* ⇒ *bool*
where
$run\text{-}ended\ r \equiv in\text{-}progress\ r\ xsk$

lemma *run-ended-not-None* [*elim*]:
$run\text{-}ended\ R \Longrightarrow R = None \Longrightarrow False$
⟨*proof*⟩

test-ended s ⟷ the test run has ended in *s*

abbreviation
test-ended :: *'a l1-state-scheme* ⇒ *bool*
where
$test\text{-}ended\ s \equiv run\text{-}ended\ (progress\ s\ test)$

a run can emit signals if it involves the same agents as the test run, and if the test run has not ended yet

definition
  can-signal :: *'a l1-state-scheme* ⇒ *agent* ⇒ *agent* ⇒ *bool*
where
  $can\text{-}signal\ s\ A\ B \equiv$
  $((A = test\text{-}owner \wedge B = test\text{-}partner) \vee (B = test\text{-}owner \wedge A = test\text{-}partner)) \wedge$
  $\neg\ test\text{-}ended\ s$

events

definition
  l1-learn :: *msg* ⇒ (*'a l1-state-scheme* ∗ *'a l1-state-scheme*) *set*
where
  $l1\text{-}learn\ m \equiv \{(s,s').$
    — guard
    $synth\ (analz\ (insert\ m\ (ik\ s))) \cap (secret\ s) = \{\} \ \wedge$
    — action
    $s' = s\ (\!|ik := ik\ s \cup \{m\}|\!)$
  }

protocol events

- step 1: create *Ra*, *A* generates *pkE*, *skE*
- step 2: create *Rb*, *B* reads *pkE* authentically, generates *K*, emits a running signal for *A*, *B*, (*pkE*, *K*)
- step 3: *A* reads *K* and *pkE* authentically, emits a commit signal for *A*, *B*, (*pkE*, *K*)

definition
  l1-step1 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ (*'a l1-state-scheme* ∗ *'a l1-state-scheme*) *set*
where
  $l1\text{-}step1\ Ra\ A\ B \equiv \{(s,\ s').$
    — guards:
    $Ra \notin dom\ (progress\ s) \wedge$
    $guessed\text{-}runs\ Ra = (\!|role{=}Init,\ owner{=}A,\ partner{=}B|\!) \wedge$
    — actions:
    $s' = s(\!|$
      $progress := (progress\ s)(Ra \mapsto \{xpkE,\ xskE\})$
      $|\!)$
  }

definition
  l1-step2 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *msg* ⇒ (*'a l1-state-scheme* ∗ *'a l1-state-scheme*) *set*
where
  $l1\text{-}step2\ Rb\ A\ B\ KE \equiv \{(s,\ s').$
    — guards:
    $guessed\text{-}runs\ Rb = (\!|role{=}Resp,\ owner{=}B,\ partner{=}A|\!) \wedge$
    $Rb \notin dom\ (progress\ s) \wedge$
    $guessed\text{-}frame\ Rb\ xpkE = Some\ KE \wedge$

$(can\text{-}signal\ s\ A\ B \longrightarrow$ — authentication guard
$(\exists\ Ra.\ guessed\text{-}runs\ Ra = (|role{=}Init,\ owner{=}A,\ partner{=}B|) \wedge$
$in\text{-}progress\ (progress\ s\ Ra)\ xpkE \wedge guessed\text{-}frame\ Ra\ xpkE = Some\ KE)) \wedge$
$(Rb = test \longrightarrow NonceF\ (Rb\$sk) \notin synth\ (analz\ (ik\ s))) \wedge$
— actions:
$s' = s(|\ progress := (progress\ s)(Rb \mapsto \{xpkE,\ xsk\}),$
$secret := \{x.\ x = NonceF\ (Rb\$sk) \wedge Rb = test\} \cup secret\ s,$
$signals := if\ can\text{-}signal\ s\ A\ B\ then$
$addSignal\ (signals\ s)\ (Running\ A\ B\ (\langle\ KE,\ NonceF\ (Rb\$sk)\ \rangle))$
$else$
$signals\ s$
$|)$
}

definition
$l1\text{-}step3 :: rid\text{-}t \Rightarrow agent \Rightarrow agent \Rightarrow msg \Rightarrow ('a\ l1\text{-}state\text{-}scheme * 'a\ l1\text{-}state\text{-}scheme)\ set$
where
$l1\text{-}step3\ Ra\ A\ B\ K \equiv \{(s,\ s').$
— guards:
$guessed\text{-}runs\ Ra = (|role{=}Init,\ owner{=}A,\ partner{=}B|) \wedge$
$progress\ s\ Ra = Some\ \{xpkE,\ xskE\} \wedge$
$guessed\text{-}frame\ Ra\ xsk = Some\ K \wedge$
$(can\text{-}signal\ s\ A\ B \longrightarrow$ — authentication guard
$(\exists\ Rb.\ guessed\text{-}runs\ Rb = (|role{=}Resp,\ owner{=}B,\ partner{=}A|) \wedge$
$progress\ s\ Rb = Some\ \{xpkE,\ xsk\} \wedge$
$guessed\text{-}frame\ Rb\ xpkE = Some\ (epubKF\ (Ra\$kE)) \wedge$
$guessed\text{-}frame\ Rb\ xsk = Some\ K)) \wedge$
$(Ra = test \longrightarrow K \notin synth\ (analz\ (ik\ s))) \wedge$

— actions:
$s' = s(|\ progress := (progress\ s)(Ra \mapsto \{xpkE,\ xskE,\ xsk\}),$
$secret := \{x.\ x = K \wedge Ra = test\} \cup secret\ s,$
$signals := if\ can\text{-}signal\ s\ A\ B\ then$
$addSignal\ (signals\ s)\ (Commit\ A\ B\ \langle epubKF\ (Ra\$kE),\ K\rangle)$
$else$
$signals\ s$
$|)$
}

specification

definition
$l1\text{-}init :: l1\text{-}state\ set$
where
$l1\text{-}init \equiv \{\ (|$
$ik = \{\},$
$secret = \{\},$
$progress = Map.empty,$
$signals = \lambda x.\ 0$
$|)\ \}$

definition
$l1\text{-}trans :: ('a\ l1\text{-}state\text{-}scheme * 'a\ l1\text{-}state\text{-}scheme)\ set$ **where**
$l1\text{-}trans \equiv (\bigcup m\ Ra\ Rb\ A\ B\ K\ KE.$

l1-step1 Ra A B ∪
l1-step2 Rb A B KE ∪
l1-step3 Ra A B K ∪
l1-learn m ∪
Id
)

definition
l1 :: (*l1-state*, *l1-obs*) *spec* **where**
l1 ≡ (|
init = *l1-init*,
trans = *l1-trans*,
obs = *id*
|)

lemmas *l1-defs* =
l1-def l1-init-def l1-trans-def
l1-learn-def
l1-step1-def l1-step2-def l1-step3-def

lemmas *l1-nostep-defs* =
l1-def l1-init-def l1-trans-def

lemma *l1-obs-id* [*simp*]: *obs l1* = *id*
⟨*proof*⟩

declare *domIff* [*iff*]

lemma *run-ended-trans*:
run-ended (*progress s R*) ⟹
(*s*, *s'*) ∈ *trans l1* ⟹
run-ended (*progress s' R*)
⟨*proof*⟩

declare *domIff* [*iff del*]

lemma *can-signal-trans*:
can-signal s' A B ⟹
(*s*, *s'*) ∈ *trans l1* ⟹
can-signal s A B
⟨*proof*⟩

16.2 Refinement: secrecy

mediator function

definition
med01s :: *l1-obs* ⇒ *s0-obs*
where
med01s t ≡ (| *ik* = *ik t*, *secret* = *secret t* |)

relation between states

definition
R01s :: (*s0-state* ∗ *l1-state*) *set*
where
R01s ≡ {(*s*,*s*′).
s = (|*ik* = *ik s*′, *secret* = *secret s*′|)
}

protocol independent events

lemma *l1-learn-refines-learn*:
{*R01s*} *s0-learn m*, *l1-learn m* {>*R01s*}
⟨*proof*⟩

protocol events

lemma *l1-step1-refines-skip*:
{*R01s*} *Id*, *l1-step1 Ra A B* {>*R01s*}
⟨*proof*⟩

lemma *l1-step2-refines-add-secret-skip*:
{*R01s*} *s0-add-secret* (*NonceF* (*Rb*$*sk*)) ∪ *Id*, *l1-step2 Rb A B KE* {>*R01s*}
⟨*proof*⟩

lemma *l1-step3-refines-add-secret-skip*:
{*R01s*} *s0-add-secret K* ∪ *Id*, *l1-step3 Ra A B K* {>*R01s*}
⟨*proof*⟩

refinement proof

lemmas *l1-trans-refines-s0-trans* =
l1-learn-refines-learn
l1-step1-refines-skip l1-step2-refines-add-secret-skip l1-step3-refines-add-secret-skip

lemma *l1-refines-init-s0* [*iff*]:
init l1 ⊆ *R01s* ‘‘ (*init s0*)
⟨*proof*⟩

lemma *l1-refines-trans-s0* [*iff*]:
{*R01s*} *trans s0*, *trans l1* {> *R01s*}
⟨*proof*⟩

lemma *obs-consistent-med01x* [*iff*]:
obs-consistent R01s med01s s0 l1
⟨*proof*⟩

refinement result

lemma *l1s-refines-s0* [*iff*]:
refines
R01s
med01s s0 l1
⟨*proof*⟩

lemma *l1-implements-s0* [*iff*]: *implements med01s s0 l1*
⟨*proof*⟩

16.3 Derived invariants: secrecy

abbreviation *l1-secrecy* ≡ *s0-secrecy*

lemma *l1-obs-secrecy* [*iff*]: *oreach l1* ⊆ *l1-secrecy*
⟨*proof*⟩

lemma *l1-secrecy* [*iff*]: *reach l1* ⊆ *l1-secrecy*
⟨*proof*⟩

16.4 Invariants

16.4.1 inv1

if a commit signal for a nonce has been emitted, then there is a finished initiator run with this nonce.

definition
  l1-inv1 :: *l1-state set*
where
  l1-inv1 ≡ {*s*. ∀ *Ra A B K*.
    signals s (*Commit A B* ⟨*epubKF* (*Ra$kE*), *K*⟩) > *0* ⟶
      guessed-runs Ra = (|*role=Init*, *owner=A*, *partner=B*|) ∧
      progress s Ra = *Some* {*xpkE*, *xskE*, *xsk*} ∧
      guessed-frame Ra xsk = *Some K*
   }

lemmas *l1-inv1I* = *l1-inv1-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l1-inv1E* [*elim*] = *l1-inv1-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l1-inv1D* = *l1-inv1-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l1-inv1-init* [*iff*]:
  init l1 ⊆ *l1-inv1*
⟨*proof*⟩

declare *domIff* [*iff*]

lemma *l1-inv1-trans* [*iff*]:
  {*l1-inv1*} *trans l1* {> *l1-inv1*}
⟨*proof*⟩

lemma *PO-l1-inv1* [*iff*]: *reach l1* ⊆ *l1-inv1*
⟨*proof*⟩

16.4.2 inv2

if a responder run knows a nonce, then a running signal for this nonce has been emitted

definition
  l1-inv2 :: *l1-state set*
where
  l1-inv2 ≡ {*s*. ∀ *KE A B Rb*.

guessed-runs Rb = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|) ⟶
progress s Rb = *Some* {*xpkE*, *xsk*} ⟶
guessed-frame Rb xpkE = *Some KE* ⟶
can-signal s A B ⟶
signals s (*Running A B* ⟨*KE*, *NonceF* (*Rb*$*sk*)⟩) > *0*
}

lemmas *l1-inv2I* = *l1-inv2-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l1-inv2E* [*elim*] = *l1-inv2-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l1-inv2D* = *l1-inv2-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l1-inv2-init* [*iff*]:
init l1 ⊆ *l1-inv2*
⟨*proof*⟩

lemma *l1-inv2-trans* [*iff*]:
{*l1-inv2*} *trans l1* {> *l1-inv2*}
⟨*proof*⟩

lemma *PO-l1-inv2* [*iff*]: *reach l1* ⊆ *l1-inv2*
⟨*proof*⟩

16.4.3 inv3 (derived)

if an unfinished initiator run and a finished responder run both know the same nonce, then the number of running signals for this nonce is strictly greater than the number of commit signals. (actually, there are 0 commit and 1 running)

definition
l1-inv3 :: *l1-state set*
where
l1-inv3 ≡ {*s*. ∀ *A B Rb Ra*.
guessed-runs Rb = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|) ⟶
progress s Rb = *Some* {*xpkE*, *xsk*} ⟶
guessed-frame Rb xpkE = *Some* (*epubKF* (*Ra*$*kE*)) ⟶
guessed-runs Ra = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ⟶
progress s Ra = *Some* {*xpkE*, *xskE*} ⟶
can-signal s A B ⟶
signals s (*Commit A B* (⟨*epubKF* (*Ra*$*kE*), *NonceF* (*Rb*$*sk*)⟩))
< *signals s* (*Running A B* (⟨*epubKF* (*Ra*$*kE*), *NonceF* (*Rb*$*sk*)⟩))
}

lemmas *l1-inv3I* = *l1-inv3-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l1-inv3E* [*elim*] = *l1-inv3-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l1-inv3D* = *l1-inv3-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l1-inv3-derived*: *l1-inv1* ∩ *l1-inv2* ⊆ *l1-inv3*
⟨*proof*⟩

16.5 Refinement: injective agreement

mediator function

definition
med01ia :: *l1-obs* ⇒ *a0i-obs*
where
med01ia t ≡ (|*a0n-state.signals* = *signals t*|)

relation between states

definition
R01ia :: (*a0i-state* ∗ *l1-state*) *set*
where
R01ia ≡ {(*s*,*s*′).
a0n-state.signals s = *signals s*′
}

protocol independent events

lemma *l1-learn-refines-a0-ia-skip*:
{*R01ia*} *Id*, *l1-learn m* {>*R01ia*}
⟨*proof*⟩

protocol events

lemma *l1-step1-refines-a0i-skip*:
{*R01ia*} *Id*, *l1-step1 Ra A B* {>*R01ia*}
⟨*proof*⟩

lemma *l1-step2-refines-a0i-running-skip*:
{*R01ia*} *a0i-running A B* ⟨*KE*, *NonceF* (*Rb*$*sk*)⟩ ∪ *Id*, *l1-step2 Rb A B KE* {>*R01ia*}
⟨*proof*⟩

lemma *l1-step3-refines-a0i-commit-skip*:
{*R01ia* ∩ (*UNIV* × *l1-inv3*)} *a0i-commit A B* ⟨*epubKF* (*Ra*$*kE*), *K*⟩ ∪ *Id*, *l1-step3 Ra A B K* {>*R01ia*}
⟨*proof*⟩

refinement proof

lemmas *l1-trans-refines-a0i-trans* =
l1-learn-refines-a0-ia-skip
l1-step1-refines-a0i-skip l1-step2-refines-a0i-running-skip l1-step3-refines-a0i-commit-skip

lemma *l1-refines-init-a0i* [*iff*]:
init l1 ⊆ *R01ia* `` (*init a0i*)
⟨*proof*⟩

lemma *l1-refines-trans-a0i* [*iff*]:
{*R01ia* ∩ (*UNIV* × (*l1-inv1* ∩ *l1-inv2*))} *trans a0i*, *trans l1* {> *R01ia*}
⟨*proof*⟩

lemma *obs-consistent-med01ia* [*iff*]:
obs-consistent R01ia med01ia a0i l1
⟨*proof*⟩

refinement result

lemma *l1-refines-a0i* [*iff*]:
refines
($R01ia \cap (reach\ a0i \times (l1\text{-}inv1 \cap l1\text{-}inv2))$)
med01ia a0i l1
⟨*proof*⟩

lemma *l1-implements-a0i* [*iff*]: *implements med01ia a0i l1*
⟨*proof*⟩

16.6 Derived invariants: injective agreement

definition
l1-iagreement :: (*'a l1-state-scheme*) *set*
where
$l1\text{-}iagreement \equiv \{s.\ \forall\ A\ B\ N.\ signals\ s\ (Commit\ A\ B\ N) \leq signals\ s\ (Running\ A\ B\ N)\}$

lemmas *l1-iagreementI* = *l1-iagreement-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l1-iagreementE* [*elim*] = *l1-iagreement-def* [*THEN setc-def-to-elim*, *rule-format*]

lemma *l1-obs-iagreement* [*iff*]: *oreach l1* ⊆ *l1-iagreement*
⟨*proof*⟩

lemma *l1-iagreement* [*iff*]: *reach l1* ⊆ *l1-iagreement*
⟨*proof*⟩

end

17 Key Transport Protocol with PFS (L2)

theory *pfslvl2*
imports *pfslvl1 Channels*
begin

declare *domIff* [*simp*, *iff del*]

17.1 State and Events

initial compromise

consts
  bad-init :: *agent set*

specification (*bad-init*)
  bad-init-spec: *test-owner* ∉ *bad-init* ∧ *test-partner* ∉ *bad-init*
⟨*proof*⟩

level 2 state

record *l2-state* =
  l1-state +
  chan :: *chan set*
  bad :: *agent set*

type-synonym *l2-obs* = *l2-state*

type-synonym
  l2-pred = *l2-state set*

type-synonym
  l2-trans = (*l2-state* × *l2-state*) *set*

attacker events

definition
  l2-dy-fake-msg :: *msg* ⇒ *l2-trans*
where
  l2-dy-fake-msg m ≡ {(*s*,*s'*).
    — guards
    m ∈ *dy-fake-msg* (*bad s*) (*ik s*) (*chan s*) ∧
    — actions
    s' = *s*(|*ik* := {*m*} ∪ *ik s*|)
  }

definition
  l2-dy-fake-chan :: *chan* ⇒ *l2-trans*
where
  l2-dy-fake-chan M ≡ {(*s*,*s'*).
    — guards
    M ∈ *dy-fake-chan* (*bad s*) (*ik s*) (*chan s*)∧
    — actions
    s' = *s*(|*chan* := {*M*} ∪ *chan s*|)

}

partnering

fun
  role-comp :: *role-t* ⇒ *role-t*
where
  role-comp Init = *Resp*
| *role-comp Resp* = *Init*

definition
  matching :: *frame* ⇒ *frame* ⇒ *bool*
where
  $matching\ sigma\ sigma' \equiv \forall\ x.\ x \in dom\ sigma \cap dom\ sigma' \longrightarrow sigma\ x = sigma'\ x$

definition
  partner-runs :: *rid-t* ⇒ *rid-t* ⇒ *bool*
where
  $partner\text{-}runs\ R\ R' \equiv$
    $role\ (guessed\text{-}runs\ R) = role\text{-}comp\ (role\ (guessed\text{-}runs\ R')) \wedge$
    $owner\ (guessed\text{-}runs\ R) = partner\ (guessed\text{-}runs\ R') \wedge$
    $owner\ (guessed\text{-}runs\ R') = partner\ (guessed\text{-}runs\ R) \wedge$
    $matching\ (guessed\text{-}frame\ R)\ (guessed\text{-}frame\ R')$

lemma *role-comp-inv* [*simp*]:
  $role\text{-}comp\ (role\text{-}comp\ x) = x$
⟨*proof*⟩

lemma *role-comp-inv-eq*:
  $y = role\text{-}comp\ x \longleftrightarrow x = role\text{-}comp\ y$
⟨*proof*⟩

definition
  partners :: *rid-t set*
where
  $partners \equiv \{R.\ partner\text{-}runs\ test\ R\}$

lemma *test-not-partner* [*simp*]:
  $test \notin partners$
⟨*proof*⟩

lemma *matching-symmetric*:
  $matching\ sigma\ sigma' \Longrightarrow matching\ sigma'\ sigma$
⟨*proof*⟩

lemma *partner-symmetric*:
  $partner\text{-}runs\ R\ R' \Longrightarrow partner\text{-}runs\ R'\ R$
⟨*proof*⟩

lemma *partner-unique*:
  $partner\text{-}runs\ R\ R'' \Longrightarrow partner\text{-}runs\ R\ R' \Longrightarrow R' = R''$
⟨*proof*⟩

lemma *partner-test*:
$R \in partners \Longrightarrow partner\text{-}runs\ R\ R' \Longrightarrow R' = test$
⟨*proof*⟩

compromising events

definition
l2-lkr-others :: *agent* ⇒ *l2-trans*
where
l2-lkr-others A ≡ {(*s*,*s'*).
— guards
$A \neq test\text{-}owner\ \wedge$
$A \neq test\text{-}partner\ \wedge$
— actions
$s' = s(\!|bad := \{A\} \cup bad\ s|\!)$
}

definition
l2-lkr-actor :: *agent* ⇒ *l2-trans*
where
l2-lkr-actor A ≡ {(*s*,*s'*).
— guards
$A = test\text{-}owner\ \wedge$
$A \neq test\text{-}partner\ \wedge$
— actions
$s' = s(\!|bad := \{A\} \cup bad\ s|\!)$
}

definition
l2-lkr-after :: *agent* ⇒ *l2-trans*
where
l2-lkr-after A ≡ {(*s*,*s'*).
— guards
test-ended s ∧
— actions
$s' = s(\!|bad := \{A\} \cup bad\ s|\!)$
}

definition
l2-skr :: *rid-t* ⇒ *msg* ⇒ *l2-trans*
where
l2-skr R K ≡ {(*s*,*s'*).
— guards
$R \neq test \wedge R \notin partners\ \wedge$
in-progress (*progress s R*) *xsk* ∧
guessed-frame R xsk = *Some K* ∧
— actions
$s' = s(\!|ik := \{K\} \cup ik\ s|\!)$
}

protocol events

definition

l2-step1 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *l2-trans*
where
l2-step1 Ra A B ≡ {(*s*, *s'*).
— guards:
Ra ∉ *dom* (*progress s*) ∧
guessed-runs Ra = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ∧
— actions:
s' = *s*(|
progress := (*progress s*)(*Ra* ↦ {*xpkE*, *xskE*}),
chan := {*Auth A B* (⟨*Number 0*, *epubKF* (*Ra*$*kE*)⟩)} ∪ (*chan s*)
|)
}

definition
l2-step2 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *msg* ⇒ *l2-trans*
where
l2-step2 Rb A B KE ≡ {(*s*, *s'*).
— guards:
guessed-runs Rb = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|) ∧
Rb ∉ *dom* (*progress s*) ∧
guessed-frame Rb xpkE = *Some KE* ∧
Auth A B ⟨*Number 0*, *KE*⟩ ∈ *chan s* ∧
— actions:
s' = *s*(|
progress := (*progress s*)(*Rb* ↦ {*xpkE*, *xsk*}),
chan := {*Auth B A* (*Aenc* (*NonceF* (*Rb*$*sk*)) *KE*)} ∪ (*chan s*),
signals := *if can-signal s A B then*
addSignal (*signals s*) (*Running A B* ⟨*KE*, *NonceF* (*Rb*$*sk*)⟩)
else
signals s,
secret := {*x*. *x* = *NonceF* (*Rb*$*sk*) ∧ *Rb* = *test*} ∪ *secret s*
|)
}

definition
l2-step3 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *msg* ⇒ *l2-trans*
where
l2-step3 Ra A B K ≡ {(*s*, *s'*).
— guards:
guessed-runs Ra = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ∧
progress s Ra = *Some* {*xpkE*, *xskE*} ∧
guessed-frame Ra xsk = *Some K* ∧
Auth B A (*Aenc K* (*epubKF* (*Ra*$*kE*))) ∈ *chan s* ∧
— actions:
s' = *s*(| *progress* := (*progress s*)(*Ra* ↦ {*xpkE*, *xskE*, *xsk*}),
signals := *if can-signal s A B then*
addSignal (*signals s*) (*Commit A B* ⟨*epubKF* (*Ra*$*kE*),*K*⟩)
else
signals s,
secret := {*x*. *x* = *K* ∧ *Ra* = *test*} ∪ *secret s*
|)
}

specification

definition
l2-init :: *l2-state set*
where
l2-init ≡ { ⦇
ik = {},
secret = {},
progress = *Map.empty*,
signals = $\lambda x.\ 0$,
chan = {},
bad = *bad-init*
⦈}

definition
l2-trans :: *l2-trans* **where**
l2-trans ≡ ($\bigcup$ *m M KE Rb Ra A B K*.
l2-step1 Ra A B ∪
l2-step2 Rb A B KE ∪
l2-step3 Ra A B m ∪
l2-dy-fake-chan M ∪
l2-dy-fake-msg m ∪
l2-lkr-others A ∪
l2-lkr-after A ∪
l2-skr Ra K ∪
Id
)

definition
l2 :: (*l2-state*, *l2-obs*) *spec* **where**
l2 ≡ ⦇
init = *l2-init*,
trans = *l2-trans*,
obs = *id*
⦈

lemmas *l2-loc-defs* =
l2-step1-def l2-step2-def l2-step3-def
l2-def l2-init-def l2-trans-def
l2-dy-fake-chan-def l2-dy-fake-msg-def
l2-lkr-after-def l2-lkr-others-def l2-skr-def

lemmas *l2-defs* = *l2-loc-defs ik-dy-def*

lemmas *l2-nostep-defs* = *l2-def l2-init-def l2-trans-def*

lemma *l2-obs-id* [*simp*]: *obs l2* = *id*
⟨*proof*⟩

Once a run is finished, it stays finished, therefore if the test is not finished at some point then it was not finished before either

declare *domIff* [*iff*]
lemma *l2-run-ended-trans*:
run-ended (*progress s R*) $\Longrightarrow$
(s, s') $\in$ *trans l2* $\Longrightarrow$
run-ended (*progress s' R*)
⟨*proof*⟩
declare *domIff* [*iff del*]

lemma *l2-can-signal-trans*:
can-signal s' A B $\Longrightarrow$
(s, s') $\in$ *trans l2* $\Longrightarrow$
can-signal s A B
⟨*proof*⟩

17.2 Invariants

17.2.1 inv1

If *can-signal s A B* (i.e., A, B are the test session agents and the test is not finished), then A, B are honest.

definition
l2-inv1 :: *l2-state set*
where
l2-inv1 $\equiv$ {*s*. $\forall$ *A B*.
can-signal s A B $\longrightarrow$
$A \notin$ *bad s* $\wedge$ $B \notin$ *bad s*
}

lemmas *l2-inv1I* = *l2-inv1-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l2-inv1E* [*elim*] = *l2-inv1-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l2-inv1D* = *l2-inv1-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l2-inv1-init* [*iff*]:
init l2 $\subseteq$ *l2-inv1*
⟨*proof*⟩

lemma *l2-inv1-trans* [*iff*]:
{*l2-inv1*} *trans l2* {> *l2-inv1*}
⟨*proof*⟩

lemma *PO-l2-inv1* [*iff*]: *reach l2* $\subseteq$ *l2-inv1*
⟨*proof*⟩

17.2.2 inv2 (authentication guard)

If *Auth A B* ⟨*Number 0*, *KE*⟩ $\in$ *chan s* and A, B are honest then the message has indeed been sent by an initiator run (with the right agents etc.)

definition
l2-inv2 :: *l2-state set*
where
l2-inv2 $\equiv$ {*s*. $\forall$ *A B KE*.
Auth A B ⟨*Number 0*, *KE*⟩ $\in$ *chan s* $\longrightarrow$

$A \notin bad\ s \wedge B \notin bad\ s \longrightarrow$
(∃ *Ra*.
guessed-runs Ra = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ∧
in-progress (*progress s Ra*) *xpkE* ∧
KE = *epubKF* (*Ra*$*kE*))
}

lemmas *l2-inv2I* = *l2-inv2-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l2-inv2E* [*elim*] = *l2-inv2-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l2-inv2D* = *l2-inv2-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l2-inv2-init* [*iff*]:
init l2 ⊆ *l2-inv2*
⟨*proof*⟩

lemma *l2-inv2-trans* [*iff*]:
{*l2-inv2*} *trans l2* {> *l2-inv2*}
⟨*proof*⟩

lemma *PO-l2-inv2* [*iff*]: *reach l2* ⊆ *l2-inv2*
⟨*proof*⟩

17.2.3 inv3 (authentication guard)

If *Auth B A* (*Aenc K* (*epubKF* (*Ra* $ *kE*))) ∈ *chan s* and *A*, *B* are honest then the message has indeed been sent by a responder run (etc).

definition
l2-inv3 :: *l2-state set*
where
l2-inv3 ≡ {*s*. ∀ *Ra A B K*.
Auth B A (*Aenc K* (*epubKF* (*Ra* $ *kE*))) ∈ *chan s* ⟶
A ∉ *bad s* ∧ *B* ∉ *bad s* ⟶
(∃ *Rb*.
guessed-runs Rb = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|) ∧
progress s Rb = *Some* {*xpkE*, *xsk*} ∧
guessed-frame Rb xpkE = *Some* (*epubKF* (*Ra*$*kE*))∧
K = *NonceF* (*Rb*$*sk*)
)
}

lemmas *l2-inv3I* = *l2-inv3-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l2-inv3E* [*elim*] = *l2-inv3-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l2-inv3D* = *l2-inv3-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l2-inv3-init* [*iff*]:
init l2 ⊆ *l2-inv3*
⟨*proof*⟩

lemma *l2-inv3-trans* [*iff*]:
{*l2-inv3*} *trans l2* {> *l2-inv3*}
⟨*proof*⟩

lemma *PO-l2-inv3* [*iff*]: *reach l2* ⊆ *l2-inv3*
⟨*proof*⟩

17.2.4 inv4

If the test run is finished and has the session key generated by a run, then this run is also finished.

definition
l2-inv4 :: *l2-state set*
where
l2-inv4 ≡ {*s*. ∀ *Rb*.
in-progress (*progress s test*) *xsk* ⟶
guessed-frame test xsk = *Some* (*NonceF* (*Rb$sk*)) ⟶
progress s Rb = *Some* {*xpkE*, *xsk*}
}

lemmas *l2-inv4I* = *l2-inv4-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l2-inv4E* [*elim*] = *l2-inv4-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l2-inv4D* = *l2-inv4-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l2-inv4-init* [*iff*]:
init l2 ⊆ *l2-inv4*
⟨*proof*⟩

lemma *l2-inv4-trans* [*iff*]:
{*l2-inv4* ∩ *l2-inv3* ∩ *l2-inv1*} *trans l2* {> *l2-inv4*}
⟨*proof*⟩

lemma *PO-l2-inv4* [*iff*]: *reach l2* ⊆ *l2-inv4*
⟨*proof*⟩

17.2.5 inv5

The only confidential or secure messages on the channel have been put there by the attacker.

definition
l2-inv5 :: *l2-state set*
where
l2-inv5 ≡ {*s*. ∀ *A B M*.
(*Confid A B M* ∈ *chan s* ∨ *Secure A B M* ∈ *chan s*) ⟶
M ∈ *dy-fake-msg* (*bad s*) (*ik s*) (*chan s*)
}

lemmas *l2-inv5I* = *l2-inv5-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l2-inv5E* [*elim*] = *l2-inv5-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l2-inv5D* = *l2-inv5-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l2-inv5-init* [*iff*]:
init l2 ⊆ *l2-inv5*
⟨*proof*⟩

lemma *l2-inv5-trans* [*iff*]:

{*l2-inv5*} *trans l2* {> *l2-inv5*}
⟨*proof*⟩

lemma *PO-l2-inv5* [*iff*]: *reach l2* ⊆ *l2-inv5*
⟨*proof*⟩

17.2.6 inv6

If an initiator *Ra* knows a session key *K*, then the attacker knows *Aenc K* (*epubKF* (*Ra* $ *kE*)).

definition
l2-inv6 :: *l2-state set*
where
l2-inv6 ≡ {*s*. ∀ *Ra K*.
role (*guessed-runs Ra*) = *Init* ⟶
in-progress (*progress s Ra*) *xsk* ⟶
guessed-frame Ra xsk = *Some K* ⟶
Aenc K (*epubKF* (*Ra*$*kE*)) ∈ *extr* (*bad s*) (*ik s*) (*chan s*)
}

lemmas *l2-inv6I* = *l2-inv6-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l2-inv6E* [*elim*] = *l2-inv6-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l2-inv6D* = *l2-inv6-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l2-inv6-init* [*iff*]:
init l2 ⊆ *l2-inv6*
⟨*proof*⟩

lemma *l2-inv6-trans* [*iff*]:
{*l2-inv6*} *trans l2* {> *l2-inv6*}
⟨*proof*⟩

lemma *PO-l2-inv6* [*iff*]: *reach l2* ⊆ *l2-inv6*
⟨*proof*⟩

17.2.7 inv7

Form of the messages in *extr* (*bad s*) (*ik s*) (*chan s*) = *synth* (*analz generators*).

abbreviation
generators ≡ *range epubK* ∪
{*Aenc* (*NonceF* (*Rb* $ *sk*)) (*epubKF* (*Ra*$*kE*)) |*Ra Rb*. ∃ *A B*.
guessed-runs Ra = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ∧
guessed-runs Rb = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|) ∧
guessed-frame Rb xpkE = *Some* (*epubKF* (*Ra*$*kE*))} ∪
{*NonceF* (*R* $ *sk*) |*R*. *R* ≠ *test* ∧ *R* ∉ *partners*}

lemma *analz-generators*: *analz generators* = *generators*
⟨*proof*⟩

definition
l2-inv7 :: *l2-state set*

where
  l2-inv7 ≡ {*s*.
    extr (*bad s*) (*ik s*) (*chan s*) ⊆
      synth (*analz* (*generators*))
  }

lemmas *l2-inv7I* = *l2-inv7-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l2-inv7E* [*elim*] = *l2-inv7-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l2-inv7D* = *l2-inv7-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l2-inv7-init* [*iff*]:
  init l2 ⊆ *l2-inv7*
⟨*proof*⟩

lemma *l2-inv7-step1*:
  {*l2-inv7*} *l2-step1 Ra A B* {> *l2-inv7*}
⟨*proof*⟩

lemma *l2-inv7-step2*:
  {*l2-inv1* ∩ *l2-inv2* ∩ *l2-inv4* ∩ *l2-inv7*} *l2-step2 Rb A B KE* {> *l2-inv7*}
⟨*proof*⟩

lemma *l2-inv7-step3*:
  {*l2-inv7*} *l2-step3 Rb A B K* {> *l2-inv7*}
⟨*proof*⟩

lemma *l2-inv7-dy-fake-msg*:
  {*l2-inv7*} *l2-dy-fake-msg M* {> *l2-inv7*}
⟨*proof*⟩

lemma *l2-inv7-dy-fake-chan*:
  {*l2-inv7*} *l2-dy-fake-chan M* {> *l2-inv7*}
⟨*proof*⟩

lemma *l2-inv7-lkr-others*:
  {*l2-inv7* ∩ *l2-inv5*} *l2-lkr-others A* {> *l2-inv7*}
⟨*proof*⟩

lemma *l2-inv7-lkr-after*:
  {*l2-inv7* ∩ *l2-inv5*} *l2-lkr-after A* {> *l2-inv7*}
⟨*proof*⟩

lemma *l2-inv7-skr*:
  {*l2-inv7* ∩ *l2-inv6*} *l2-skr R K* {> *l2-inv7*}
⟨*proof*⟩

lemmas *l2-inv7-trans-aux* =
  l2-inv7-step1 l2-inv7-step2 l2-inv7-step3
  l2-inv7-dy-fake-msg l2-inv7-dy-fake-chan
  l2-inv7-lkr-others l2-inv7-lkr-after l2-inv7-skr

lemma *l2-inv7-trans* [*iff*]:
  {*l2-inv7* ∩ *l2-inv1* ∩ *l2-inv2* ∩ *l2-inv4* ∩ *l2-inv5* ∩ *l2-inv6*} *trans l2* {> *l2-inv7*}

⟨*proof*⟩

lemma *PO-l2-inv7* [*iff*]: *reach l2* ⊆ *l2-inv7*
⟨*proof*⟩

lemma *l2-inv7-aux*:
  NonceF (*R*$*sk*) ∈ *analz* (*ik s*) ⟹ *s* ∈ *l2-inv7* ⟹ *R* ≠ *test* ∧ *R* ∉ *partners*
⟨*proof*⟩

17.2.8 inv8

Form of the secrets = nonces generated by test or partners

definition
  l2-inv8 :: *l2-state set*
where
  l2-inv8 ≡ {*s*.
    secret s ⊆ {*NonceF* (*R*$*sk*) | *R*. *R* = *test* ∨ *R* ∈ *partners*}
  }

lemmas *l2-inv8I* = *l2-inv8-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l2-inv8E* [*elim*] = *l2-inv8-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l2-inv8D* = *l2-inv8-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l2-inv8-init* [*iff*]:
  init l2 ⊆ *l2-inv8*
⟨*proof*⟩

lemma *l2-inv8-trans* [*iff*]:
  {*l2-inv8* ∩ *l2-inv1* ∩ *l2-inv3*} *trans l2* {> *l2-inv8*}
⟨*proof*⟩

lemma *PO-l2-inv8* [*iff*]: *reach l2* ⊆ *l2-inv8*
⟨*proof*⟩

17.3 Refinement

mediator function

definition
  med12s :: *l2-obs* ⇒ *l1-obs*
where
  med12s t ≡ (|
    ik = *ik t*,
    secret = *secret t*,
    progress = *progress t*,
    signals = *signals t*
    |)

relation between states

definition
  R12s :: (*l1-state* ∗ *l2-state*) *set*
where

R12s ≡ {(*s*,*s'*).
s = *med12s s'*
}

lemmas *R12s-defs* = *R12s-def med12s-def*

lemma *can-signal-R12* [*simp*]:
(*s1*, *s2*) ∈ *R12s* ⟹
can-signal s1 A B ⟷ *can-signal s2 A B*
⟨*proof*⟩

protocol events

lemma *l2-step1-refines-step1*:
{*R12s*} *l1-step1 Ra A B*, *l2-step1 Ra A B* {>*R12s*}
⟨*proof*⟩

lemma *l2-step2-refines-step2*:
{*R12s* ∩ *UNIV* × (*l2-inv1* ∩ *l2-inv2* ∩ *l2-inv7*)}
l1-step2 Rb A B KE, *l2-step2 Rb A B KE*
{>*R12s*}
⟨*proof*⟩

auxiliary lemma needed to prove that the nonce received by the test in step 3 comes from a partner

lemma *l2-step3-partners*:
guessed-runs test = (|*role* = *Init*, *owner* = *A*, *partner* = *B*|) ⟹
guessed-frame test xsk = *Some* (*NonceF* (*Rb*$*sk*)) ⟹
guessed-runs Rb = (|*role* = *Resp*, *owner* = *B*, *partner* = *A*|) ⟹
guessed-frame Rb xpkE = *Some* (*epubKF* (*test* $ *kE*)) ⟹
Rb ∈ *partners*
⟨*proof*⟩

lemma *l2-step3-refines-step3*:
{*R12s* ∩ *UNIV* × (*l2-inv1* ∩ *l2-inv3* ∩ *l2-inv7*)}
l1-step3 Ra A B K, *l2-step3 Ra A B K*
{>*R12s*}
⟨*proof*⟩

attacker events

lemma *l2-dy-fake-chan-refines-skip*:
{*R12s*} *Id*, *l2-dy-fake-chan M* {>*R12s*}
⟨*proof*⟩

lemma *l2-dy-fake-msg-refines-learn*:
{*R12s* ∩ *UNIV* × *l2-inv7* ∩ *UNIV* × *l2-inv8*} *l1-learn m*, *l2-dy-fake-msg m* {>*R12s*}
⟨*proof*⟩

compromising events

lemma *l2-lkr-others-refines-skip*:
{*R12s*} *Id*, *l2-lkr-others A* {>*R12s*}

⟨*proof*⟩

lemma *l2-lkr-after-refines-skip*:
$\{R12s\}$ *Id*, *l2-lkr-after A* $\{>R12s\}$
⟨*proof*⟩

lemma *l2-skr-refines-learn*:
$\{R12s \cap UNIV \times l2\text{-}inv7 \cap UNIV \times l2\text{-}inv6 \cap UNIV \times l2\text{-}inv8\}$ *l1-learn K*, *l2-skr R K* $\{>R12s\}$
⟨*proof*⟩

refinement proof

lemmas *l2-trans-refines-l1-trans* =
l2-dy-fake-msg-refines-learn l2-dy-fake-chan-refines-skip
l2-lkr-others-refines-skip l2-lkr-after-refines-skip l2-skr-refines-learn
l2-step1-refines-step1 l2-step2-refines-step2 l2-step3-refines-step3

lemma *l2-refines-init-l1* [*iff*]:
$init\ l2 \subseteq R12s$ ‘‘ $(init\ l1)$
⟨*proof*⟩

lemma *l2-refines-trans-l1* [*iff*]:
$\{R12s \cap (UNIV \times (l2\text{-}inv1 \cap l2\text{-}inv2 \cap l2\text{-}inv3 \cap l2\text{-}inv6 \cap l2\text{-}inv7 \cap l2\text{-}inv8))\}$ *trans l1*, *trans l2*
$\{> R12s\}$
⟨*proof*⟩

lemma *PO-obs-consistent-R12s* [*iff*]:
obs-consistent R12s med12s l1 l2
⟨*proof*⟩

lemma *l2-refines-l1* [*iff*]:
refines
$(R12s \cap$
$(reach\ l1 \times (l2\text{-}inv1 \cap l2\text{-}inv2 \cap l2\text{-}inv3 \cap l2\text{-}inv4 \cap l2\text{-}inv5 \cap l2\text{-}inv6 \cap l2\text{-}inv7 \cap l2\text{-}inv8)))$
med12s l1 l2
⟨*proof*⟩

lemma *l2-implements-l1* [*iff*]:
implements med12s l1 l2
⟨*proof*⟩

17.4 Derived invariants

We want to prove *l2-secrecy*: *dy-fake-msg* (*bad s*) (*ik s*) (*chan s*) $\cap$ *secret s* = {} but by refinement we only get *l2-partial-secrecy*: *synth* (*analz* (*ik s*)) $\cap$ *secret s* = {} This is fine, since a message in *dy-fake-msg* (*bad s*) (*ik s*) (*chan s*) could be added to *ik s*, and *l2-partial-secrecy* would still hold for this new state.

definition
l2-partial-secrecy :: (*'a l2-state-scheme*) *set*
where
l2-partial-secrecy $\equiv$ {*s*. *synth* (*analz* (*ik s*)) $\cap$ *secret s* = {}}

lemma *l2-obs-partial-secrecy* [*iff*]: *oreach l2* ⊆ *l2-partial-secrecy*
⟨*proof*⟩

lemma *l2-oreach-dy-fake-msg*:
s ∈ *oreach l2* ⟹ *x* ∈ *dy-fake-msg* (*bad s*) (*ik s*) (*chan s*) ⟹ *s* (|*ik* := *insert x* (*ik s*)|) ∈ *oreach l2*
⟨*proof*⟩

definition
l2-secrecy :: (*'a l2-state-scheme*) *set*
where
l2-secrecy ≡ {*s*. *dy-fake-msg* (*bad s*) (*ik s*) (*chan s*) ∩ *secret s* = {}}

lemma *l2-obs-secrecy* [*iff*]: *oreach l2* ⊆ *l2-secrecy*
⟨*proof*⟩

lemma *l2-secrecy* [*iff*]: *reach l2* ⊆ *l2-secrecy*
⟨*proof*⟩

abbreviation *l2-iagreement* ≡ *l1-iagreement*

lemma *l2-obs-iagreement* [*iff*]: *oreach l2* ⊆ *l2-iagreement*
⟨*proof*⟩

lemma *l2-iagreement* [*iff*]: *reach l2* ⊆ *l2-iagreement*
⟨*proof*⟩

end

18 Key Transport Protocol with PFS (L3 locale)

theory *pfslvl3*
imports *pfslvl2 Implem-lemmas*
begin

18.1 State and Events

Level 3 state

(The types have to be defined outside the locale.)

record *l3-state* = *l1-state* +
  bad :: *agent set*

type-synonym *l3-obs* = *l3-state*

type-synonym
  l3-pred = *l3-state set*

type-synonym
  l3-trans = (*l3-state* × *l3-state*) *set*

attacker event

definition
  l3-dy :: *msg* ⇒ *l3-trans*
where
  l3-dy ≡ *ik-dy*

compromise events

definition
  l3-lkr-others :: *agent* ⇒ *l3-trans*
where
  l3-lkr-others A ≡ {(*s*,*s′*).
    — guards
    A ≠ *test-owner* ∧
    A ≠ *test-partner* ∧
    — actions
    s′ = *s*⦇*bad* := {*A*} ∪ *bad s*,
         ik := *keys-of A* ∪ *ik s*⦈
  }

definition
  l3-lkr-actor :: *agent* ⇒ *l3-trans*
where
  l3-lkr-actor A ≡ {(*s*,*s′*).
    — guards
    A = *test-owner* ∧
    A ≠ *test-partner* ∧
    — actions
    s′ = *s*⦇*bad* := {*A*} ∪ *bad s*,
         ik := *keys-of A* ∪ *ik s*⦈
  }

definition
l3-lkr-after :: *agent* ⇒ *l3-trans*
where
l3-lkr-after A ≡ {(*s*,*s*′).
— guards
test-ended s ∧
— actions
s′ = *s*(|*bad* := {*A*} ∪ *bad s*,
ik := *keys-of A* ∪ *ik s*|)
}

definition
l3-skr :: *rid-t* ⇒ *msg* ⇒ *l3-trans*
where
l3-skr R K ≡ {(*s*,*s*′).
— guards
R ≠ *test* ∧ *R* ∉ *partners* ∧
in-progress (*progress s R*) *xsk* ∧
guessed-frame R xsk = *Some K* ∧
— actions
s′ = *s*(|*ik* := {*K*} ∪ *ik s*|)
}

New locale for the level 3 protocol

This locale does not add new assumptions, it is only used to separate the level 3 protocol from the implementation locale.

locale *pfslvl3* = *valid-implem*
begin

protocol events

definition
l3-step1 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *l3-trans*
where
l3-step1 Ra A B ≡ {(*s*, *s*′).
— guards:
Ra ∉ *dom* (*progress s*) ∧
guessed-runs Ra = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ∧
— actions:
s′ = *s*(|
progress := (*progress s*)(*Ra* ↦ {*xpkE*, *xskE*}),
ik := {*implAuth A B* ⟨*Number 0*, *epubKF* (*Ra*$*kE*)⟩} ∪ (*ik s*)
|)
}

definition
l3-step2 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *msg* ⇒ *l3-trans*
where
l3-step2 Rb A B KE ≡ {(*s*, *s*′).
— guards:
guessed-runs Rb = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|) ∧
Rb ∉ *dom* (*progress s*) ∧

$guessed\text{-}frame\ Rb\ xpkE = Some\ KE \wedge$
$implAuth\ A\ B\ \langle Number\ 0,\ KE\rangle \in ik\ s \wedge$
— actions:
$s' = s(\!|$
$progress := (progress\ s)(Rb \mapsto \{xpkE,\ xsk\}),$
$ik := \{implAuth\ B\ A\ (Aenc\ (NonceF\ (Rb\$sk))\ KE)\} \cup (ik\ s),$
$signals := if\ can\text{-}signal\ s\ A\ B\ then$
$addSignal\ (signals\ s)\ (Running\ A\ B\ \langle KE,\ NonceF\ (Rb\$sk)\rangle)$
$else$
$signals\ s,$
$secret := \{x.\ x = NonceF\ (Rb\$sk) \wedge Rb = test\} \cup secret\ s$
$|\!)$
}

definition
$l3\text{-}step3 :: rid\text{-}t \Rightarrow agent \Rightarrow agent \Rightarrow msg \Rightarrow l3\text{-}trans$
where
$l3\text{-}step3\ Ra\ A\ B\ K \equiv \{(s,\ s').$
— guards:
$guessed\text{-}runs\ Ra = (\!|role{=}Init,\ owner{=}A,\ partner{=}B|\!) \wedge$
$progress\ s\ Ra = Some\ \{xpkE,\ xskE\} \wedge$
$guessed\text{-}frame\ Ra\ xsk = Some\ K \wedge$
$implAuth\ B\ A\ (Aenc\ K\ (epubKF\ (Ra\$kE))) \in ik\ s \wedge$
— actions:
$s' = s(\!|\ progress := (progress\ s)(Ra \mapsto \{xpkE,\ xskE,\ xsk\}),$
$signals := if\ can\text{-}signal\ s\ A\ B\ then$
$addSignal\ (signals\ s)\ (Commit\ A\ B\ \langle epubKF\ (Ra\$kE),\ K\rangle)$
$else$
$signals\ s,$
$secret := \{x.\ x = K \wedge Ra = test\} \cup secret\ s$
$|\!)$
}

specification

initial compromise

definition
$ik\text{-}init :: msg\ set$
where
$ik\text{-}init \equiv \{priK\ C\ |C.\ C \in bad\text{-}init\} \cup \{pubK\ A\ |\ A.\ True\} \cup$
$\{shrK\ A\ B\ |A\ B.\ A \in bad\text{-}init \vee B \in bad\text{-}init\} \cup Tags$

lemmas about *ik-init*

lemma *parts-ik-init* [*simp*]: *parts ik-init = ik-init*
⟨*proof*⟩

lemma *analz-ik-init* [*simp*]: *analz ik-init = ik-init*
⟨*proof*⟩

lemma *abs-ik-init* [*iff*]: *abs ik-init* = {}
⟨*proof*⟩

lemma *payloadSet-ik-init* [*iff*]: *ik-init* ∩ *payload* = {}
⟨*proof*⟩

lemma *validSet-ik-init* [*iff*]: *ik-init* ∩ *valid* = {}
⟨*proof*⟩

definition
l3-init :: *l3-state set*
where
l3-init ≡ { (|
ik = *ik-init*,
secret = {},
progress = *Map.empty*,
signals = $\lambda x.\ 0$,
bad = *bad-init*
|)}

lemmas *l3-init-defs* = *l3-init-def ik-init-def*

definition
l3-trans :: *l3-trans*
where
l3-trans ≡ (⋃ *m M KE Rb Ra A B K*.
l3-step1 Ra A B ∪
l3-step2 Rb A B KE ∪
l3-step3 Ra A B m ∪
l3-dy M ∪
l3-lkr-others A ∪
l3-lkr-after A ∪
l3-skr Ra K ∪
Id
)

definition
l3 :: (*l3-state*, *l3-obs*) *spec* **where**
l3 ≡ (|
init = *l3-init*,
trans = *l3-trans*,
obs = *id*
|)

lemmas *l3-loc-defs* =
l3-step1-def l3-step2-def l3-step3-def
l3-def l3-init-defs l3-trans-def
l3-dy-def
l3-lkr-others-def l3-lkr-after-def l3-skr-def

lemmas *l3-defs* = *l3-loc-defs ik-dy-def*
lemmas *l3-nostep-defs* = *l3-def l3-init-def l3-trans-def*

lemma *l3-obs-id* [*simp*]: *obs l3* = *id*
⟨*proof*⟩

18.2 Invariants

18.2.1 inv1: No long-term keys as message parts

definition
l3-inv1 :: *l3-state set*
where
l3-inv1 ≡ {*s*.
parts (*ik s*) ∩ *range LtK* ⊆ *ik s*
}

lemmas *l3-inv1I* = *l3-inv1-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv1E* [*elim*] = *l3-inv1-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv1D* = *l3-inv1-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv1D′* [*dest*]: ⟦ *LtK K* ∈ *parts* (*ik s*); *s* ∈ *l3-inv1*⟧ ⟹ *LtK K* ∈ *ik s*
⟨*proof*⟩

lemma *l3-inv1-init* [*iff*]:
init l3 ⊆ *l3-inv1*
⟨*proof*⟩

lemma *l3-inv1-trans* [*iff*]:
{*l3-inv1*} *trans l3* {> *l3-inv1*}
⟨*proof*⟩

lemma *PO-l3-inv1* [*iff*]:
reach l3 ⊆ *l3-inv1*
⟨*proof*⟩

18.2.2 inv2: *l3-state.bad s* indeed contains "bad" keys

definition
l3-inv2 :: *l3-state set*
where
l3-inv2 ≡ {*s*.
Keys-bad (*ik s*) (*bad s*)
}

lemmas *l3-inv2I* = *l3-inv2-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv2E* [*elim*] = *l3-inv2-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv2D* = *l3-inv2-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv2-init* [*simp*,*intro*!]:
init l3 ⊆ *l3-inv2*
⟨*proof*⟩

lemma *l3-inv2-trans* [*simp*,*intro*!]:
{*l3-inv2* ∩ *l3-inv1*} *trans l3* {> *l3-inv2*}

⟨*proof*⟩

lemma *PO-l3-inv2* [*iff*]: *reach l3* ⊆ *l3-inv2*
⟨*proof*⟩

18.2.3 inv3

If a message can be analyzed from the intruder knowledge then it can be derived (using synth/analz) from the sets of implementation, non-implementation, and long-term key messages and the tags. That is, intermediate messages are not needed.

definition
  l3-inv3 :: *l3-state set*
where
  l3-inv3 ≡ {*s*.
    analz (*ik s*) ⊆
    synth (*analz* ((*ik s* ∩ *payload*) ∪ ((*ik s*) ∩ *valid*) ∪ (*ik s* ∩ *range LtK*) ∪ *Tags*))
  }

lemmas *l3-inv3I* = *l3-inv3-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv3E* = *l3-inv3-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv3D* = *l3-inv3-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv3-init* [*iff*]:
  init l3 ⊆ *l3-inv3*
⟨*proof*⟩

declare *domIff* [*iff del*]

Most of the cases in this proof are simple and very similar. The proof could probably be shortened.

lemma *l3-inv3-trans* [*simp*,*intro*!]:
  {*l3-inv3*} *trans l3* {> *l3-inv3*}
⟨*proof*⟩

lemma *PO-l3-inv3* [*iff*]: *reach l3* ⊆ *l3-inv3*
⟨*proof*⟩

18.2.4 inv4: the intruder knows the tags

definition
  l3-inv4 :: *l3-state set*
where
  l3-inv4 ≡ {*s*.
    Tags ⊆ *ik s*
  }

lemmas *l3-inv4I* = *l3-inv4-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv4E* [*elim*] = *l3-inv4-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv4D* = *l3-inv4-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv4-init* [*simp*,*intro*!]:
  init l3 ⊆ *l3-inv4*

⟨*proof*⟩

lemma *l3-inv4-trans* [*simp*,*intro*!]:
{*l3-inv4*} *trans l3* {> *l3-inv4*}
⟨*proof*⟩

lemma *PO-l3-inv4* [*simp*,*intro*!]: *reach l3* ⊆ *l3-inv4*
⟨*proof*⟩

The remaining invariants are derived from the others. They are not protocol dependent provided the previous invariants hold.

18.2.5 inv5

The messages that the L3 DY intruder can derive from the intruder knowledge (using *synth*/*analz*), are either implementations or intermediate messages or can also be derived by the L2 intruder from the set *extr* (*l3-state.bad s*) (*ik s* ∩ *payload*) (*local.abs* (*ik s*)), that is, given the non-implementation messages and the abstractions of (implementation) messages in the intruder knowledge.

definition
l3-inv5 :: *l3-state set*
where
l3-inv5 ≡ {*s*.
synth (*analz* (*ik s*)) ⊆
dy-fake-msg (*bad s*) (*ik s* ∩ *payload*) (*abs* (*ik s*)) ∪ −*payload*
}

lemmas *l3-inv5I* = *l3-inv5-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv5E* = *l3-inv5-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv5D* = *l3-inv5-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv5-derived*: *l3-inv2* ∩ *l3-inv3* ⊆ *l3-inv5*
⟨*proof*⟩

lemma *PO-l3-inv5* [*simp*,*intro*!]: *reach l3* ⊆ *l3-inv5*
⟨*proof*⟩

18.2.6 inv6

If the level 3 intruder can deduce a message implementing an insecure channel message, then:

- either the message is already in the intruder knowledge;
- or the message is constructed, and the payload can also be deduced by the intruder.

definition
l3-inv6 :: *l3-state set*
where
l3-inv6 ≡ {*s*. ∀ *A B M*.
(*implInsec A B M* ∈ *synth* (*analz* (*ik s*)) ∧ *M* ∈ *payload*) ⟶
(*implInsec A B M* ∈ *ik s* ∨ *M* ∈ *synth* (*analz* (*ik s*)))

}

lemmas *l3-inv6I* = *l3-inv6-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv6E* = *l3-inv6-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv6D* = *l3-inv6-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv6-derived* [*simp*,*intro*!]:
l3-inv3 ∩ *l3-inv4* ⊆ *l3-inv6*
⟨*proof*⟩

lemma *PO-l3-inv6* [*simp*,*intro*!]: *reach l3* ⊆ *l3-inv6*
⟨*proof*⟩

18.2.7 inv7

If the level 3 intruder can deduce a message implementing a confidential channel message, then:

- either the message is already in the intruder knowledge;
- or the message is constructed, and the payload can also be deduced by the intruder.

definition
l3-inv7 :: *l3-state set*
where
l3-inv7 ≡ {*s*. ∀ *A B M*.
(*implConfid A B M* ∈ *synth* (*analz* (*ik s*)) ∧ *M* ∈ *payload*) ⟶
(*implConfid A B M* ∈ *ik s* ∨ *M* ∈ *synth* (*analz* (*ik s*)))
}

lemmas *l3-inv7I* = *l3-inv7-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv7E* = *l3-inv7-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv7D* = *l3-inv7-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv7-derived* [*simp*,*intro*!]:
l3-inv3 ∩ *l3-inv4* ⊆ *l3-inv7*
⟨*proof*⟩

lemma *PO-l3-inv7* [*simp*,*intro*!]: *reach l3* ⊆ *l3-inv7*
⟨*proof*⟩

18.2.8 inv8

If the level 3 intruder can deduce a message implementing an authentic channel message then:

- either the message is already in the intruder knowledge;
- or the message is constructed, and in this case the payload can also be deduced by the intruder, and one of the agents is bad.

definition
l3-inv8 :: *l3-state set*

where
  l3-inv8 ≡ {*s*. ∀ *A B M*.
    (*implAuth A B M* ∈ *synth* (*analz* (*ik s*)) ∧ *M* ∈ *payload*) ⟶
    (*implAuth A B M* ∈ *ik s* ∨ (*M* ∈ *synth* (*analz* (*ik s*)) ∧ (*A* ∈ *bad s* ∨ *B* ∈ *bad s*)))
  }

lemmas *l3-inv8I* = *l3-inv8-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv8E* = *l3-inv8-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv8D* = *l3-inv8-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv8-derived* [*iff*]:
  l3-inv2 ∩ *l3-inv3* ∩ *l3-inv4* ⊆ *l3-inv8*
⟨*proof*⟩

lemma *PO-l3-inv8* [*iff*]: *reach l3* ⊆ *l3-inv8*
⟨*proof*⟩

18.2.9 inv9

If the level 3 intruder can deduce a message implementing a secure channel message then:

- either the message is already in the intruder knowledge;
- or the message is constructed, and in this case the payload can also be deduced by the intruder, and one of the agents is bad.

definition
  l3-inv9 :: *l3-state set*
where
  l3-inv9 ≡ {*s*. ∀ *A B M*.
    (*implSecure A B M* ∈ *synth* (*analz* (*ik s*)) ∧ *M* ∈ *payload*) ⟶
    (*implSecure A B M* ∈ *ik s* ∨ (*M* ∈ *synth* (*analz* (*ik s*)) ∧ (*A* ∈ *bad s* ∨ *B* ∈ *bad s*)))
  }

lemmas *l3-inv9I* = *l3-inv9-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv9E* = *l3-inv9-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv9D* = *l3-inv9-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv9-derived* [*iff*]:
  l3-inv2 ∩ *l3-inv3* ∩ *l3-inv4* ⊆ *l3-inv9*
⟨*proof*⟩

lemma *PO-l3-inv9* [*iff*]: *reach l3* ⊆ *l3-inv9*
⟨*proof*⟩

18.3 Refinement

mediator function

definition
  med23s :: *l3-obs* ⇒ *l2-obs*
where
  med23s t ≡ (|

ik = *ik t* ∩ *payload*,
secret = *secret t*,
progress = *progress t*,
signals = *signals t*,
chan = *abs* (*ik t*),
bad = *bad t*
⦈

relation between states

definition
R23s :: (*l2-state* ∗ *l3-state*) *set*
where
R23s ≡ {(*s*, *s'*).
s = *med23s s'*
}

lemmas *R23s-defs* = *R23s-def med23s-def*

lemma *R23sI*:
⟦ *ik s* = *ik t* ∩ *payload*; *secret s* = *secret t*; *progress s* = *progress t*; *signals s* = *signals t*;
chan s = *abs* (*ik t*); *l2-state.bad s* = *bad t* ⟧
⟹ (*s*, *t*) ∈ *R23s*
⟨*proof*⟩

lemma *R23sD*:
(*s*, *t*) ∈ *R23s* ⟹
ik s = *ik t* ∩ *payload* ∧ *secret s* = *secret t* ∧ *progress s* = *progress t* ∧ *signals s* = *signals t* ∧
chan s = *abs* (*ik t*) ∧ *l2-state.bad s* = *bad t*
⟨*proof*⟩

lemma *R23sE* [*elim*]:
⟦ (*s*, *t*) ∈ *R23s*;
⟦ *ik s* = *ik t* ∩ *payload*; *secret s* = *secret t*; *progress s* = *progress t*; *signals s* = *signals t*;
chan s = *abs* (*ik t*); *l2-state.bad s* = *bad t* ⟧ ⟹ *P* ⟧
⟹ *P*
⟨*proof*⟩

lemma *can-signal-R23* [*simp*]:
(*s2*, *s3*) ∈ *R23s* ⟹
can-signal s2 A B ⟷ *can-signal s3 A B*
⟨*proof*⟩

18.3.1 Protocol events

lemma *l3-step1-refines-step1*:
{*R23s*} *l2-step1 Ra A B*, *l3-step1 Ra A B* {>*R23s*}
⟨*proof*⟩

lemma *l3-step2-refines-step2*:
{*R23s*} *l2-step2 Rb A B KE*, *l3-step2 Rb A B KE*{>*R23s*}
⟨*proof*⟩

lemma *l3-step3-refines-step3*:
$\{R23s\}$ *l2-step3 Ra A B K*, *l3-step3 Ra A B K* $\{>R23s\}$
⟨*proof*⟩

18.3.2 Intruder events

lemma *l3-dy-payload-refines-dy-fake-msg*:
$M \in payload \Longrightarrow$
$\{R23s \cap UNIV \times l3\text{-}inv5\}$ *l2-dy-fake-msg M*, *l3-dy M* $\{>R23s\}$
⟨*proof*⟩

lemma *l3-dy-valid-refines-dy-fake-chan*:
$[\![M \in valid;\ M' \in abs\ \{M\}]\!] \Longrightarrow$
$\{R23s \cap UNIV \times (l3\text{-}inv5 \cap l3\text{-}inv6 \cap l3\text{-}inv7 \cap l3\text{-}inv8 \cap l3\text{-}inv9)\}$
l2-dy-fake-chan M′, *l3-dy M*
$\{>R23s\}$
⟨*proof*⟩

lemma *l3-dy-valid-refines-dy-fake-chan-Un*:
$M \in valid \Longrightarrow$
$\{R23s \cap UNIV \times (l3\text{-}inv5 \cap l3\text{-}inv6 \cap l3\text{-}inv7 \cap l3\text{-}inv8 \cap l3\text{-}inv9)\}$
$\bigcup M'.$ *l2-dy-fake-chan M′*, *l3-dy M*
$\{>R23s\}$
⟨*proof*⟩

lemma *l3-dy-isLtKey-refines-skip*:
$\{R23s\}$ *Id*, *l3-dy* (*LtK ltk*) $\{>R23s\}$
⟨*proof*⟩

lemma *l3-dy-others-refines-skip*:
$[\![M \notin range\ LtK;\ M \notin valid;\ M \notin payload]\!] \Longrightarrow$
$\{R23s\}$ *Id*, *l3-dy M* $\{>R23s\}$
⟨*proof*⟩

lemma *l3-dy-refines-dy-fake-msg-dy-fake-chan-skip*:
$\{R23s \cap UNIV \times (l3\text{-}inv5 \cap l3\text{-}inv6 \cap l3\text{-}inv7 \cap l3\text{-}inv8 \cap l3\text{-}inv9)\}$
l2-dy-fake-msg M $\cup$ ($\bigcup M'.$ *l2-dy-fake-chan M′*) $\cup$ *Id*, *l3-dy M*
$\{>R23s\}$
⟨*proof*⟩

18.3.3 Compromise events

lemma *l3-lkr-others-refines-lkr-others*:
$\{R23s\}$ *l2-lkr-others A*, *l3-lkr-others A* $\{>R23s\}$
⟨*proof*⟩

lemma *l3-lkr-after-refines-lkr-after*:
$\{R23s\}$ *l2-lkr-after A*, *l3-lkr-after A* $\{>R23s\}$
⟨*proof*⟩

lemma *l3-skr-refines-skr*:
  {*R23s*} *l2-skr R K*, *l3-skr R K* {>*R23s*}
⟨*proof*⟩

lemmas *l3-trans-refines-l2-trans* =
  l3-step1-refines-step1 l3-step2-refines-step2 l3-step3-refines-step3
  l3-dy-refines-dy-fake-msg-dy-fake-chan-skip
  l3-lkr-others-refines-lkr-others l3-lkr-after-refines-lkr-after l3-skr-refines-skr

lemma *l3-refines-init-l2* [*iff*]:
  init l3 ⊆ *R23s* `` (*init l2*)
⟨*proof*⟩

lemma *l3-refines-trans-l2* [*iff*]:
  {*R23s* ∩ (*UNIV* × (*l3-inv1* ∩ *l3-inv2* ∩ *l3-inv3* ∩ *l3-inv4*))} *trans l2*, *trans l3* {> *R23s*}
⟨*proof*⟩

lemma *PO-obs-consistent-R23s* [*iff*]:
  obs-consistent R23s med23s l2 l3
⟨*proof*⟩

lemma *l3-refines-l2* [*iff*]:
  refines
    (*R23s* ∩
     (*reach l2* × (*l3-inv1* ∩ *l3-inv2* ∩ *l3-inv3* ∩ *l3-inv4*)))
    med23s l2 l3
⟨*proof*⟩

lemma *l3-implements-l2* [*iff*]:
  implements med23s l2 l3
⟨*proof*⟩

18.4 Derived invariants

18.4.1 inv10: secrets contain no implementation material

definition
  l3-inv10 :: *l3-state set*
where
  l3-inv10 ≡ {*s*.
    secret s ⊆ *payload*
  }

lemmas *l3-inv10I* = *l3-inv10-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv10E* = *l3-inv10-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv10D* = *l3-inv10-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv10-init* [*iff*]:

init l3 ⊆ *l3-inv10*
⟨*proof*⟩

lemma *l3-inv10-trans* [*iff*]:
{*l3-inv10*} *trans l3* {> *l3-inv10*}
⟨*proof*⟩

lemma *PO-l3-inv10* [*iff*]: *reach l3* ⊆ *l3-inv10*
⟨*proof*⟩

lemma *l3-obs-inv10* [*iff*]: *oreach l3* ⊆ *l3-inv10*
⟨*proof*⟩

18.4.2 Partial secrecy

We want to prove *l3-secrecy*, ie *synth* (*analz* (*ik s*)) ∩ *secret s* = {}, but by refinement we only get *l3-partial-secrecy*: *dy-fake-msg* (*l3-state.bad s*) (*payloadSet* (*ik s*)) (*local.abs* (*ik s*)) ∩ *secret s* = {}. This is fine if secrets contain no implementation material. Then, by *inv5*, a message in *synth* (*analz* (*ik s*)) is in *dy-fake-msg* (*l3-state.bad s*) (*payloadSet* (*ik s*)) (*local.abs* (*ik s*)) ∪ − *payload*, and *l3-partial-secrecy* proves it is not a secret.

definition
l3-partial-secrecy :: (*'a l3-state-scheme*) *set*
where
l3-partial-secrecy ≡ {*s*.
dy-fake-msg (*bad s*) (*ik s* ∩ *payload*) (*abs* (*ik s*)) ∩ *secret s* = {}
}

lemma *l3-obs-partial-secrecy* [*iff*]: *oreach l3* ⊆ *l3-partial-secrecy*
⟨*proof*⟩

18.4.3 Secrecy

definition
l3-secrecy :: (*'a l3-state-scheme*) *set*
where
l3-secrecy ≡ *l1-secrecy*

lemma *l3-obs-inv5*: *oreach l3* ⊆ *l3-inv5*
⟨*proof*⟩

lemma *l3-obs-secrecy* [*iff*]: *oreach l3* ⊆ *l3-secrecy*
⟨*proof*⟩

lemma *l3-secrecy* [*iff*]: *reach l3* ⊆ *l3-secrecy*
⟨*proof*⟩

18.4.4 Injective agreement

abbreviation *l3-iagreement* ≡ *l1-iagreement*

lemma *l3-obs-iagreement* [*iff*]: *oreach l3* ⊆ *l3-iagreement*

⟨*proof*⟩

lemma *l3-iagreement* [*iff*]: *reach l3* ⊆ *l3-iagreement*
⟨*proof*⟩

end
end

19 Key Transport Protocol with PFS (L3, asymmetric implementation)

theory *pfslvl3-asymmetric*
imports *pfslvl3 Implem-asymmetric*
begin

interpretation *pfslvl3-asym*: *pfslvl3 implem-asym*
⟨*proof*⟩

end

20 Key Transport Protocol with PFS (L3, symmetric implementation)

theory *pfslvl3-symmetric*
imports *pfslvl3 Implem-symmetric*
begin

interpretation *pfslvl3-asym*: *pfslvl3 implem-sym*
⟨*proof*⟩

end

21 Authenticated Diffie Hellman Protocol (L1)

theory *dhlvl1*
imports *Runs Secrecy AuthenticationI Payloads*
begin

declare *option.split-asm* [*split*]

21.1 State and Events

consts
Nend :: *nat*

abbreviation *nx* :: *nat* **where** *nx* ≡ *2*
abbreviation *ny* :: *nat* **where** *ny* ≡ *3*

Proofs break if *1* is used, because *simp* replaces it with *Suc 0*. . . .

abbreviation
xEnd ≡ *Var 0*

abbreviation
xnx ≡ *Var 2*

abbreviation
xny ≡ *Var 3*

abbreviation
xsk ≡ *Var 4*

abbreviation
xgnx ≡ *Var 5*

abbreviation
xgny ≡ *Var 6*

abbreviation
End ≡ *Number Nend*

Domain of each role (protocol dependent).

fun *domain* :: *role-t* ⇒ *var set* **where**
domain Init = {*xnx*, *xgnx*, *xgny*, *xsk*, *xEnd*}
| *domain Resp* = {*xny*, *xgnx*, *xgny*, *xsk*, *xEnd*}

consts
test :: *rid-t*

consts
guessed-runs :: *rid-t* ⇒ *run-t*
guessed-frame :: *rid-t* ⇒ *frame*

Specification of the guessed frame:

1. Domain
2. Well-typedness. The messages in the frame of a run never contain implementation material even if the agents of the run are dishonest. Therefore we consider only well-typed frames. This is notably required for the session key compromise; it also helps proving the partitionning of ik, since we know that the messages added by the protocol do not contain ltkeys in their payload and are therefore valid implementations.
3. We also ensure that the values generated by the frame owner are correctly guessed.

specification (*guessed-frame*)
  guessed-frame-dom-spec [*simp*]:
    dom (*guessed-frame R*) = *domain* (*role* (*guessed-runs R*))
  guessed-frame-payload-spec [*simp*, *elim*]:
    guessed-frame R x = *Some y* $\Longrightarrow$ *y* ∈ *payload*
  guessed-frame-Init-xnx [*simp*]:
    role (*guessed-runs R*) = *Init* $\Longrightarrow$ *guessed-frame R xnx* = *Some* (*NonceF* (*R$nx*))
  guessed-frame-Init-xgnx [*simp*]:
    role (*guessed-runs R*) = *Init* $\Longrightarrow$ *guessed-frame R xgnx* = *Some* (*Exp Gen* (*NonceF* (*R$nx*)))
  guessed-frame-Resp-xny [*simp*]:
    role (*guessed-runs R*) = *Resp* $\Longrightarrow$ *guessed-frame R xny* = *Some* (*NonceF* (*R$ny*))
  guessed-frame-Resp-xgny [*simp*]:
    role (*guessed-runs R*) = *Resp* $\Longrightarrow$ *guessed-frame R xgny* = *Some* (*Exp Gen* (*NonceF* (*R$ny*)))
  guessed-frame-xEnd [*simp*]:
    guessed-frame R xEnd = *Some End*
⟨*proof*⟩

abbreviation
  test-owner ≡ *owner* (*guessed-runs test*)

abbreviation
  test-partner ≡ *partner* (*guessed-runs test*)

Level 1 state.

record *l1-state* =
  s0-state +
  progress :: *progress-t*
  signalsInit :: *signal* ⇒ *nat*
  signalsResp :: *signal* ⇒ *nat*

type-synonym *l1-obs* = *l1-state*

abbreviation
  run-ended :: *var set option* ⇒ *bool*
where
  run-ended r ≡ *in-progress r xEnd*

lemma *run-ended-not-None* [*elim*]:

run-ended R $\Longrightarrow$ *R* = *None* $\Longrightarrow$ *False*
⟨*proof*⟩

test-ended s $\longleftrightarrow$ the test run has ended in *s*.

abbreviation
test-ended :: *'a l1-state-scheme* ⇒ *bool*
where
test-ended s ≡ *run-ended* (*progress s test*)

A run can emit signals if it involves the same agents as the test run, and if the test run has not ended yet.

definition
can-signal :: *'a l1-state-scheme* ⇒ *agent* ⇒ *agent* ⇒ *bool*
where
can-signal s A B ≡
((*A* = *test-owner* ∧ *B* = *test-partner*) ∨ (*B* = *test-owner* ∧ *A* = *test-partner*)) ∧
¬ *test-ended s*

Events.

definition
l1-learn :: *msg* ⇒ (*'a l1-state-scheme* ∗ *'a l1-state-scheme*) *set*
where
l1-learn m ≡ {(*s*,*s'*).
— guard
synth (*analz* (*insert m* (*ik s*))) ∩ (*secret s*) = {} ∧
— action
s' = *s* ⦇*ik* := *ik s* ∪ {*m*}⦈
}

Potocol events.

- step 1: create *Ra*, *A* generates *nx*, computes g^{nx}
- step 2: create *Rb*, *B* reads g^{nx} insecurely, generates *ny*, computes g^{ny}, computes $g^{nx\ *\ ny}$, emits a running signal for *Init*, $g^{nx\ *\ ny}$
- step 3: *A* reads g^{ny} and g^{nx} authentically, computes $g^{ny\ *\ nx}$, emits a commit signal for *Init*, $g^{ny\ *\ nx}$, a running signal for *Resp*, $g^{ny\ *\ nx}$, declares the secret $g^{ny\ *\ nx}$
- step 4: *B* reads g^{nx} and g^{ny} authentically, emits a commit signal for *Resp*, $g^{nx\ *\ ny}$, declares the secret $g^{nx\ *\ ny}$

definition
l1-step1 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ (*'a l1-state-scheme* ∗ *'a l1-state-scheme*) *set*
where
l1-step1 Ra A B ≡ {(*s*, *s'*).
— guards:
Ra ∉ *dom* (*progress s*) ∧
guessed-runs Ra = ⦇*role*=*Init*, *owner*=*A*, *partner*=*B*⦈ ∧
— actions:
s' = *s*⦇

$progress := (progress\ s)(Ra \mapsto \{xnx,\ xgnx\})$
⦈
}

definition
$l1\text{-}step2$ :: $rid\text{-}t \Rightarrow agent \Rightarrow agent \Rightarrow msg \Rightarrow ('a\ l1\text{-}state\text{-}scheme * 'a\ l1\text{-}state\text{-}scheme)\ set$
where
$l1\text{-}step2\ Rb\ A\ B\ gnx \equiv \{(s,\ s').$
— guards:
$guessed\text{-}runs\ Rb = ⦇role{=}Resp,\ owner{=}B,\ partner{=}A⦈ \land$
$Rb \notin dom\ (progress\ s) \land$
$guessed\text{-}frame\ Rb\ xgnx = Some\ gnx \land$
$guessed\text{-}frame\ Rb\ xsk = Some\ (Exp\ gnx\ (NonceF\ (Rb\$ny))) \land$
— actions:
$s' = s⦇\ progress := (progress\ s)(Rb \mapsto \{xny,\ xgny,\ xgnx,\ xsk\}),$
$signalsInit := if\ can\text{-}signal\ s\ A\ B\ then$
$addSignal\ (signalsInit\ s)\ (Running\ A\ B\ (Exp\ gnx\ (NonceF\ (Rb\$ny))))$
$else$
$signalsInit\ s$
⦈
}

definition
$l1\text{-}step3$ :: $rid\text{-}t \Rightarrow agent \Rightarrow agent \Rightarrow msg \Rightarrow ('a\ l1\text{-}state\text{-}scheme * 'a\ l1\text{-}state\text{-}scheme)\ set$
where
$l1\text{-}step3\ Ra\ A\ B\ gny \equiv \{(s,\ s').$
— guards:
$guessed\text{-}runs\ Ra = ⦇role{=}Init,\ owner{=}A,\ partner{=}B⦈ \land$
$progress\ s\ Ra = Some\ \{xnx,\ xgnx\} \land$
$guessed\text{-}frame\ Ra\ xgny = Some\ gny \land$
$guessed\text{-}frame\ Ra\ xsk = Some\ (Exp\ gny\ (NonceF\ (Ra\$nx))) \land$
$(can\text{-}signal\ s\ A\ B \longrightarrow$ — authentication guard
$(\exists\ Rb.\ guessed\text{-}runs\ Rb = ⦇role{=}Resp,\ owner{=}B,\ partner{=}A⦈ \land$
$in\text{-}progressS\ (progress\ s\ Rb)\ \{xny,\ xgnx,\ xgny,\ xsk\} \land$
$guessed\text{-}frame\ Rb\ xgny = Some\ gny \land$
$guessed\text{-}frame\ Rb\ xgnx = Some\ (Exp\ Gen\ (NonceF\ (Ra\$nx))))) \land$
$(Ra = test \longrightarrow Exp\ gny\ (NonceF\ (Ra\$nx)) \notin synth\ (analz\ (ik\ s))) \land$

— actions:
$s' = s⦇\ progress := (progress\ s)(Ra \mapsto \{xnx,\ xgnx,\ xgny,\ xsk,\ xEnd\}),$
$secret := \{x.\ x = Exp\ gny\ (NonceF\ (Ra\$nx)) \land Ra = test\} \cup secret\ s,$
$signalsInit := if\ can\text{-}signal\ s\ A\ B\ then$
$addSignal\ (signalsInit\ s)\ (Commit\ A\ B\ (Exp\ gny\ (NonceF\ (Ra\$nx))))$
$else$
$signalsInit\ s,$
$signalsResp := if\ can\text{-}signal\ s\ A\ B\ then$
$addSignal\ (signalsResp\ s)\ (Running\ A\ B\ (Exp\ gny\ (NonceF\ (Ra\$nx))))$
$else$
$signalsResp\ s$

⦈
}

definition
l1-step4 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *msg* ⇒ (*'a l1-state-scheme* ∗ *'a l1-state-scheme*) *set*
where
l1-step4 Rb A B gnx ≡ {(*s*, *s'*).
— guards:
guessed-runs Rb = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|) ∧
progress s Rb = *Some* {*xny*, *xgnx*, *xgny*, *xsk*} ∧
guessed-frame Rb xgnx = *Some gnx* ∧
(*can-signal s A B* ⟶ — authentication guard
(∃ *Ra*. *guessed-runs Ra* = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ∧
in-progressS (*progress s Ra*) {*xnx*, *xgnx*, *xgny*, *xsk*, *xEnd*} ∧
guessed-frame Ra xgnx = *Some gnx* ∧
guessed-frame Ra xgny = *Some* (*Exp Gen* (*NonceF* (*Rb*$*ny*))))) ∧
(*Rb* = *test* ⟶ *Exp gnx* (*NonceF* (*Rb*$*ny*)) ∉ *synth* (*analz* (*ik s*))) ∧

— actions:
s' = *s*(| *progress* := (*progress s*)(*Rb* ↦ {*xny*, *xgnx*, *xgny*, *xsk*, *xEnd*}),
secret := {*x*. *x* = *Exp gnx* (*NonceF* (*Rb*$*ny*)) ∧ *Rb* = *test*} ∪ *secret s*,
signalsResp := *if can-signal s A B then*
addSignal (*signalsResp s*) (*Commit A B* (*Exp gnx* (*NonceF* (*Rb*$*ny*))))
else
signalsResp s
|)
}

Specification.

definition
l1-init :: *l1-state set*
where
l1-init ≡ { (|
ik = {},
secret = {},
progress = *Map.empty*,
signalsInit = λ*x*. *0*,
signalsResp = λ*x*. *0*
|) }

definition
l1-trans :: (*'a l1-state-scheme* ∗ *'a l1-state-scheme*) *set* **where**
l1-trans ≡ (⋃ *m Ra Rb A B x*.
l1-step1 Ra A B ∪
l1-step2 Rb A B x ∪
l1-step3 Ra A B x ∪
l1-step4 Rb A B x ∪
l1-learn m ∪
Id
)

definition
l1 :: (*l1-state*, *l1-obs*) *spec* **where**
l1 ≡ (|

init = *l1-init*,
trans = *l1-trans*,
obs = *id*
⦈

lemmas *l1-defs* =
l1-def l1-init-def l1-trans-def
l1-learn-def
l1-step1-def l1-step2-def l1-step3-def l1-step4-def

lemmas *l1-nostep-defs* =
l1-def l1-init-def l1-trans-def

lemma *l1-obs-id* [*simp*]: *obs l1* = *id*
⟨*proof*⟩

lemma *run-ended-trans*:
run-ended (*progress s R*) ⟹
(*s*, *s'*) ∈ *trans l1* ⟹
run-ended (*progress s' R*)
⟨*proof*⟩

lemma *can-signal-trans*:
can-signal s' A B ⟹
(*s*, *s'*) ∈ *trans l1* ⟹
can-signal s A B
⟨*proof*⟩

21.2 Refinement: secrecy

Mediator function.

definition
med01s :: *l1-obs* ⇒ *s0-obs*
where
med01s t ≡ ⦇ *ik* = *ik t*, *secret* = *secret t* ⦈

Relation between states.

definition
R01s :: (*s0-state* ∗ *l1-state*) *set*
where
R01s ≡ {(*s*,*s'*).
s = ⦇*ik* = *ik s'*, *secret* = *secret s'*⦈
}

Protocol independent events.

lemma *l1-learn-refines-learn*:
{*R01s*} *s0-learn m*, *l1-learn m* {>*R01s*}
⟨*proof*⟩

Protocol events.

lemma *l1-step1-refines-skip*:

{*R01s*} *Id*, *l1-step1 Ra A B* {>*R01s*}
⟨*proof*⟩

lemma *l1-step2-refines-skip*:
{*R01s*} *Id*, *l1-step2 Rb A B gnx* {>*R01s*}
⟨*proof*⟩

lemma *l1-step3-refines-add-secret-skip*:
{*R01s*} *s0-add-secret* (*Exp gny* (*NonceF* (*Ra*$*nx*))) ∪ *Id*, *l1-step3 Ra A B gny* {>*R01s*}
⟨*proof*⟩

lemma *l1-step4-refines-add-secret-skip*:
{*R01s*} *s0-add-secret* (*Exp gnx* (*NonceF* (*Rb*$*ny*))) ∪ *Id*, *l1-step4 Rb A B gnx* {>*R01s*}
⟨*proof*⟩

Refinement proof.

lemmas *l1-trans-refines-s0-trans* =
l1-learn-refines-learn
l1-step1-refines-skip l1-step2-refines-skip
l1-step3-refines-add-secret-skip l1-step4-refines-add-secret-skip

lemma *l1-refines-init-s0* [*iff*]:
init l1 ⊆ *R01s* ‘‘ (*init s0*)
⟨*proof*⟩

lemma *l1-refines-trans-s0* [*iff*]:
{*R01s*} *trans s0*, *trans l1* {> *R01s*}
⟨*proof*⟩

lemma *obs-consistent-med01x* [*iff*]:
obs-consistent R01s med01s s0 l1
⟨*proof*⟩

Refinement result.

lemma *l1s-refines-s0* [*iff*]:
refines
R01s
med01s s0 l1
⟨*proof*⟩

lemma *l1-implements-s0* [*iff*]: *implements med01s s0 l1*
⟨*proof*⟩

21.3 Derived invariants: secrecy

abbreviation *l1-secrecy* ≡ *s0-secrecy*

lemma *l1-obs-secrecy* [*iff*]: *oreach l1* ⊆ *l1-secrecy*
⟨*proof*⟩

lemma *l1-secrecy* [*iff*]: *reach l1* ⊆ *l1-secrecy*
⟨*proof*⟩

21.4 Invariants: *Init* authenticates *Resp*

21.4.1 inv1

If an initiator commit signal exists for $(g^{ny})^{Ra\ \$\ nx}$ then *Ra* is *Init*, has passed step 3, and has (*g*⁀*ny*)⁀(*Ra*$*nx*) as the key in its frame.

definition
  l1-inv1 :: *l1-state set*
where
  l1-inv1 ≡ {*s*. ∀ *Ra A B gny*.
    signalsInit s (*Commit A B* (*Exp gny* (*NonceF* (*Ra*$*nx*)))) > *0* ⟶
      guessed-runs Ra = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ∧
      progress s Ra = *Some* {*xnx*, *xgnx*, *xgny*, *xsk*, *xEnd*} ∧
      guessed-frame Ra xsk = *Some* (*Exp gny* (*NonceF* (*Ra*$*nx*)))
  }

lemmas *l1-inv1I* = *l1-inv1-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l1-inv1E* [*elim*] = *l1-inv1-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l1-inv1D* = *l1-inv1-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l1-inv1-init* [*iff*]:
  init l1 ⊆ *l1-inv1*
⟨*proof*⟩

lemma *l1-inv1-trans* [*iff*]:
  {*l1-inv1*} *trans l1* {> *l1-inv1*}
⟨*proof*⟩

lemma *PO-l1-inv1* [*iff*]: *reach l1* ⊆ *l1-inv1*
⟨*proof*⟩

21.4.2 inv2

If a *Resp* run *Rb* has passed step 2 then (if possible) an initiator running signal has been emitted.

definition
  l1-inv2 :: *l1-state set*
where
  l1-inv2 ≡ {*s*. ∀ *gnx A B Rb*.
    guessed-runs Rb = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|) ⟶
    in-progressS (*progress s Rb*) {*xny*, *xgnx*, *xgny*, *xsk*} ⟶
    guessed-frame Rb xgnx = *Some gnx* ⟶
    can-signal s A B ⟶
      signalsInit s (*Running A B* (*Exp gnx* (*NonceF* (*Rb*$*ny*)))) > *0*
  }

lemmas *l1-inv2I* = *l1-inv2-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l1-inv2E* [*elim*] = *l1-inv2-def* [*THEN setc-def-to-elim*, *rule-format*]

lemmas *l1-inv2D* = *l1-inv2-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l1-inv2-init* [*iff*]:
  init l1 ⊆ *l1-inv2*
⟨*proof*⟩

lemma *l1-inv2-trans* [*iff*]:
  {*l1-inv2*} *trans l1* {> *l1-inv2*}
⟨*proof*⟩

lemma *PO-l1-inv2* [*iff*]: *reach l1* ⊆ *l1-inv2*
⟨*proof*⟩

21.4.3 inv3 (derived)

If an *Init* run before step 3 and a *Resp* run after step 2 both know the same half-keys (more or less), then the number of *Init* running signals for the key is strictly greater than the number of *Init* commit signals. (actually, there are 0 commit and 1 running).

definition
  l1-inv3 :: *l1-state set*
where
  l1-inv3 ≡ {*s*. ∀ *A B Rb Ra gny*.
    guessed-runs Rb = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|) ⟶
    in-progressS (*progress s Rb*) {*xny*, *xgnx*, *xgny*, *xsk*} ⟶
    guessed-frame Rb xgny = *Some gny* ⟶
    guessed-frame Rb xgnx = *Some* (*Exp Gen* (*NonceF* (*Ra*$*nx*))) ⟶
    guessed-runs Ra = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ⟶
    progress s Ra = *Some* {*xnx*, *xgnx*} ⟶
    can-signal s A B ⟶
      signalsInit s (*Commit A B* (*Exp gny* (*NonceF* (*Ra*$*nx*))))
    < *signalsInit s* (*Running A B* (*Exp gny* (*NonceF* (*Ra*$*nx*))))
  }

lemmas *l1-inv3I* = *l1-inv3-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l1-inv3E* [*elim*] = *l1-inv3-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l1-inv3D* = *l1-inv3-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l1-inv3-derived*: *l1-inv1* ∩ *l1-inv2* ⊆ *l1-inv3*
⟨*proof*⟩

21.5 Invariants: *Resp* authenticates *Init*

21.5.1 inv4

If a *Resp* commit signal exists for $(g^{nx})^{Rb\ \$\ ny}$ then *Rb* is *Resp*, has finished its run, and has $(g^{nx})^{Rb\ \$\ ny}$ as the key in its frame.

definition
  l1-inv4 :: *l1-state set*
where
  l1-inv4 ≡ {*s*. ∀ *Rb A B gnx*.

signalsResp s (*Commit A B* (*Exp gnx* (*NonceF* (*Rb*\$*ny*)))) > *0* ⟶
guessed-runs Rb = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|) ∧
progress s Rb = *Some* {*xny*, *xgnx*, *xgny*, *xsk*, *xEnd*} ∧
guessed-frame Rb xgnx = *Some gnx*
}

lemmas *l1-inv4I* = *l1-inv4-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l1-inv4E* [*elim*] = *l1-inv4-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l1-inv4D* = *l1-inv4-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l1-inv4-init* [*iff*]:
init l1 ⊆ *l1-inv4*
⟨*proof*⟩

declare *domIff* [*iff*]

lemma *l1-inv4-trans* [*iff*]:
{*l1-inv4*} *trans l1* {> *l1-inv4*}
⟨*proof*⟩

declare *domIff* [*iff del*]

lemma *PO-l1-inv4* [*iff*]: *reach l1* ⊆ *l1-inv4*
⟨*proof*⟩

21.5.2 inv5

If an *Init* run *Ra* has passed step3 then (if possible) a *Resp* running signal has been emitted.

definition
l1-inv5 :: *l1-state set*
where
l1-inv5 ≡ {*s*. ∀ *gny A B Ra*.
guessed-runs Ra = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ⟶
in-progressS (*progress s Ra*) {*xnx*, *xgnx*, *xgny*, *xsk*, *xEnd*} ⟶
guessed-frame Ra xgny = *Some gny* ⟶
can-signal s A B ⟶
signalsResp s (*Running A B* (*Exp gny* (*NonceF* (*Ra*\$*nx*)))) > *0*
}

lemmas *l1-inv5I* = *l1-inv5-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l1-inv5E* [*elim*] = *l1-inv5-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l1-inv5D* = *l1-inv5-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l1-inv5-init* [*iff*]:
init l1 ⊆ *l1-inv5*
⟨*proof*⟩

lemma *l1-inv5-trans* [*iff*]:
{*l1-inv5*} *trans l1* {> *l1-inv5*}
⟨*proof*⟩

lemma *PO-l1-inv5* [*iff*]: *reach l1* ⊆ *l1-inv5*
⟨*proof*⟩

21.5.3 inv6 (derived)

If a *Resp* run before step 4 and an *Init* run after step 3 both know the same half-keys (more or less), then the number of *Resp* running signals for the key is strictly greater than the number of *Resp* commit signals. (actually, there are 0 commit and 1 running).

definition
  l1-inv6 :: *l1-state set*
where
  l1-inv6 ≡ {*s*. ∀ *A B Rb Ra gnx*.
    guessed-runs Ra = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ⟶
    in-progressS (*progress s Ra*) {*xnx*, *xgnx*, *xgny*, *xsk*, *xEnd*} ⟶
    guessed-frame Ra xgnx = *Some gnx* ⟶
    guessed-frame Ra xgny = *Some* (*Exp Gen* (*NonceF* (*Rb*$*ny*))) ⟶
    guessed-runs Rb = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|) ⟶
    progress s Rb = *Some* {*xny*, *xgnx*, *xgny*, *xsk*} ⟶
    can-signal s A B ⟶
      signalsResp s (*Commit A B* (*Exp gnx* (*NonceF* (*Rb*$*ny*))))
    < *signalsResp s* (*Running A B* (*Exp gnx* (*NonceF* (*Rb*$*ny*))))
  }

lemmas *l1-inv6I* = *l1-inv6-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l1-inv6E* [*elim*] = *l1-inv6-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l1-inv6D* = *l1-inv6-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l1-inv6-derived*:
  l1-inv4 ∩ *l1-inv5* ⊆ *l1-inv6*
⟨*proof*⟩

21.6 Refinement: injective agreement (*Init* authenticates *Resp*)

Mediator function.

definition
  med01iai :: *l1-obs* ⇒ *a0i-obs*
where
  med01iai t ≡ (|*a0n-state.signals* = *signalsInit t*|)

Relation between states.

definition
  R01iai :: (*a0i-state* ∗ *l1-state*) *set*
where
  R01iai ≡ {(*s*,*s'*).
    a0n-state.signals s = *signalsInit s'*
    }

Protocol-independent events.

lemma *l1-learn-refines-a0-ia-skip-i*:
  {*R01iai*} *Id*, *l1-learn m* {>*R01iai*}

⟨*proof*⟩

Protocol events.

lemma *l1-step1-refines-a0i-skip-i*:
{*R01iai*} *Id*, *l1-step1 Ra A B* {>*R01iai*}
⟨*proof*⟩

lemma *l1-step2-refines-a0i-running-skip-i*:
{*R01iai*} *a0i-running A B* (*Exp gnx* (*NonceF* (*Rb*$*ny*))) ∪ *Id*, *l1-step2 Rb A B gnx* {>*R01iai*}
⟨*proof*⟩

lemma *l1-step3-refines-a0i-commit-skip-i*:
{*R01iai* ∩ (*UNIV* × *l1-inv3*)}
a0i-commit A B (*Exp gny* (*NonceF* (*Ra*$*nx*))) ∪ *Id*,
l1-step3 Ra A B gny
{>*R01iai*}
⟨*proof*⟩

lemma *l1-step4-refines-a0i-skip-i*:
{*R01iai*} *Id*, *l1-step4 Rb A B gnx* {>*R01iai*}
⟨*proof*⟩

Refinement proof.

lemmas *l1-trans-refines-a0i-trans-i* =
l1-learn-refines-a0-ia-skip-i
l1-step1-refines-a0i-skip-i l1-step2-refines-a0i-running-skip-i
l1-step3-refines-a0i-commit-skip-i l1-step4-refines-a0i-skip-i

lemma *l1-refines-init-a0i-i* [*iff*]:
init l1 ⊆ *R01iai* ‘‘ (*init a0i*)
⟨*proof*⟩

lemma *l1-refines-trans-a0i-i* [*iff*]:
{*R01iai* ∩ (*UNIV* × (*l1-inv1* ∩ *l1-inv2*))} *trans a0i*, *trans l1* {> *R01iai*}
⟨*proof*⟩

lemma *obs-consistent-med01iai* [*iff*]:
obs-consistent R01iai med01iai a0i l1
⟨*proof*⟩

Refinement result.

lemma *l1-refines-a0i-i* [*iff*]:
refines
(*R01iai* ∩ (*reach a0i* × (*l1-inv1* ∩ *l1-inv2*)))
med01iai a0i l1
⟨*proof*⟩

lemma *l1-implements-a0i-i* [*iff*]: *implements med01iai a0i l1*
⟨*proof*⟩

21.7 Derived invariants: injective agreement (*Init* authenticates *Resp*)

definition
l1-iagreement-Init :: ('*a l1-state-scheme*) *set*
where
l1-iagreement-Init ≡ {*s*. ∀ *A B N*.
signalsInit s (*Commit A B N*) ≤ *signalsInit s* (*Running A B N*)
}

lemmas *l1-iagreement-InitI* = *l1-iagreement-Init-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l1-iagreement-InitE* [*elim*] = *l1-iagreement-Init-def* [*THEN setc-def-to-elim*, *rule-format*]

lemma *l1-obs-iagreement-Init* [*iff*]: *oreach l1* ⊆ *l1-iagreement-Init*
⟨*proof*⟩

lemma *l1-iagreement-Init* [*iff*]: *reach l1* ⊆ *l1-iagreement-Init*
⟨*proof*⟩

21.8 Refinement: injective agreement (*Resp* authenticates *Init*)

Mediator function.

definition
med01iar :: *l1-obs* ⇒ *a0i-obs*
where
med01iar t ≡ (|*a0n-state.signals* = *signalsResp t*|)

Relation between states.

definition
R01iar :: (*a0i-state* ∗ *l1-state*) *set*
where
R01iar ≡ {(*s*,*s*′).
a0n-state.signals s = *signalsResp s*′
}

Protocol-independent events.

lemma *l1-learn-refines-a0-ia-skip-r*:
{*R01iar*} *Id*, *l1-learn m* {>*R01iar*}
⟨*proof*⟩

Protocol events.

lemma *l1-step1-refines-a0i-skip-r*:
{*R01iar*} *Id*, *l1-step1 Ra A B* {>*R01iar*}
⟨*proof*⟩

lemma *l1-step2-refines-a0i-skip-r*:
{*R01iar*} *Id*, *l1-step2 Rb A B gnx* {>*R01iar*}
⟨*proof*⟩

lemma *l1-step3-refines-a0i-running-skip-r*:
{*R01iar*} *a0i-running A B* (*Exp gny* (*NonceF* (*Ra*$*nx*))) ∪ *Id*, *l1-step3 Ra A B gny* {>*R01iar*}

⟨*proof*⟩

lemma *l1-step4-refines-a0i-commit-skip-r*:
  {*R01iar* ∩ *UNIV*×*l1-inv6*}
    a0i-commit A B (*Exp gnx* (*NonceF* (*Rb$ny*))) ∪ *Id*,
    l1-step4 Rb A B gnx
  {>*R01iar*}
⟨*proof*⟩

Refinement proofs.

lemmas *l1-trans-refines-a0i-trans-r* =
  l1-learn-refines-a0-ia-skip-r
  l1-step1-refines-a0i-skip-r l1-step2-refines-a0i-skip-r
  l1-step3-refines-a0i-running-skip-r l1-step4-refines-a0i-commit-skip-r

lemma *l1-refines-init-a0i-r* [*iff*]:
  init l1 ⊆ *R01iar* `` (*init a0i*)
⟨*proof*⟩

lemma *l1-refines-trans-a0i-r* [*iff*]:
  {*R01iar* ∩ (*UNIV* × (*l1-inv4* ∩ *l1-inv5*))} *trans a0i*, *trans l1* {> *R01iar*}
⟨*proof*⟩

lemma *obs-consistent-med01iar* [*iff*]:
  obs-consistent R01iar med01iar a0i l1
⟨*proof*⟩

Refinement result.

lemma *l1-refines-a0i-r* [*iff*]:
  refines
    (*R01iar* ∩ (*reach a0i* × (*l1-inv4* ∩ *l1-inv5*)))
    med01iar a0i l1
⟨*proof*⟩

lemma *l1-implements-a0i-r* [*iff*]: *implements med01iar a0i l1*
⟨*proof*⟩

21.9 Derived invariants: injective agreement (*Resp* authenticates *Init*)

definition
  l1-iagreement-Resp :: (*'a l1-state-scheme*) *set*
where
  l1-iagreement-Resp ≡ {*s*. ∀ *A B N*.
    signalsResp s (*Commit A B N*) ≤ *signalsResp s* (*Running A B N*)
  }

lemmas *l1-iagreement-RespI* = *l1-iagreement-Resp-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l1-iagreement-RespE* [*elim*] = *l1-iagreement-Resp-def* [*THEN setc-def-to-elim*, *rule-format*]

lemma *l1-obs-iagreement-Resp* [*iff*]: *oreach l1* ⊆ *l1-iagreement-Resp*

⟨*proof*⟩

lemma *l1-iagreement-Resp* [*iff*]: *reach l1* ⊆ *l1-iagreement-Resp*
⟨*proof*⟩

end

22 Authenticated Diffie-Hellman Protocol (L2)

theory *dhlvl2*
imports *dhlvl1 Channels*
begin

declare *domIff* [*simp*, *iff del*]

22.1 State and Events

Initial compromise.

consts
  bad-init :: *agent set*

specification (*bad-init*)
  bad-init-spec: *test-owner* ∉ *bad-init* ∧ *test-partner* ∉ *bad-init*
⟨*proof*⟩

Level 2 state.

record *l2-state* =
  l1-state +
  chan :: *chan set*
  bad :: *agent set*

type-synonym *l2-obs* = *l2-state*

type-synonym
  l2-pred = *l2-state set*

type-synonym
  l2-trans = (*l2-state* × *l2-state*) *set*

Attacker events.

definition
  l2-dy-fake-msg :: *msg* ⇒ *l2-trans*
where
  l2-dy-fake-msg m ≡ {(*s*,*s'*).
    — guards
    m ∈ *dy-fake-msg* (*bad s*) (*ik s*) (*chan s*) ∧
    — actions
    s' = *s*(|*ik* := {*m*} ∪ *ik s*|)
  }

definition
  l2-dy-fake-chan :: *chan* ⇒ *l2-trans*
where
  l2-dy-fake-chan M ≡ {(*s*,*s'*).
    — guards
    M ∈ *dy-fake-chan* (*bad s*) (*ik s*) (*chan s*)∧
    — actions
    s' = *s*(|*chan* := {*M*} ∪ *chan s*|)

}

Partnering.

fun
  role-comp :: *role-t* ⇒ *role-t*
where
  role-comp Init = *Resp*
| *role-comp Resp* = *Init*

definition
  matching :: *frame* ⇒ *frame* ⇒ *bool*
where
  matching sigma sigma' ≡ ∀ *x*. *x* ∈ *dom sigma* ∩ *dom sigma'* ⟶ *sigma x* = *sigma' x*

definition
  partner-runs :: *rid-t* ⇒ *rid-t* ⇒ *bool*
where
  partner-runs R R' ≡
    role (*guessed-runs R*) = *role-comp* (*role* (*guessed-runs R'*)) ∧
    owner (*guessed-runs R*) = *partner* (*guessed-runs R'*) ∧
    owner (*guessed-runs R'*) = *partner* (*guessed-runs R*) ∧
    matching (*guessed-frame R*) (*guessed-frame R'*)

lemma *role-comp-inv* [*simp*]:
  role-comp (*role-comp x*) = *x*
⟨*proof*⟩

lemma *role-comp-inv-eq*:
  y = *role-comp x* ⟷ *x* = *role-comp y*
⟨*proof*⟩

definition
  partners :: *rid-t set*
where
  partners ≡ {*R*. *partner-runs test R*}

lemma *test-not-partner* [*simp*]:
  test ∉ *partners*
⟨*proof*⟩

lemma *matching-symmetric*:
  matching sigma sigma' ⟹ *matching sigma' sigma*
⟨*proof*⟩

lemma *partner-symmetric*:
  partner-runs R R' ⟹ *partner-runs R' R*
⟨*proof*⟩

The unicity of the parther is actually protocol dependent: it only holds if there are generated fresh nonces (which identify the runs) in the frames.

lemma *partner-unique*:

$partner\text{-}runs\ R\ R'' \Longrightarrow partner\text{-}runs\ R\ R' \Longrightarrow R' = R''$
⟨*proof*⟩

lemma *partner-test*:
$R \in partners \Longrightarrow partner\text{-}runs\ R\ R' \Longrightarrow R' = test$
⟨*proof*⟩

Compromising events.

definition
l2-lkr-others :: *agent* ⇒ *l2-trans*
where
l2-lkr-others $A \equiv \{(s,s')$.
— guards
$A \neq test\text{-}owner \wedge$
$A \neq test\text{-}partner \wedge$
— actions
$s' = s(\!|bad := \{A\} \cup bad\ s|\!)$
}

definition
l2-lkr-actor :: *agent* ⇒ *l2-trans*
where
l2-lkr-actor $A \equiv \{(s,s')$.
— guards
$A = test\text{-}owner \wedge$
$A \neq test\text{-}partner \wedge$
— actions
$s' = s(\!|bad := \{A\} \cup bad\ s|\!)$
}

definition
l2-lkr-after :: *agent* ⇒ *l2-trans*
where
l2-lkr-after $A \equiv \{(s,s')$.
— guards
$test\text{-}ended\ s \wedge$
— actions
$s' = s(\!|bad := \{A\} \cup bad\ s|\!)$
}

definition
l2-skr :: *rid-t* ⇒ *msg* ⇒ *l2-trans*
where
l2-skr $R\ K \equiv \{(s,s')$.
— guards
$R \neq test \wedge R \notin partners \wedge$
$in\text{-}progress\ (progress\ s\ R)\ xsk \wedge$
$guessed\text{-}frame\ R\ xsk = Some\ K \wedge$
— actions
$s' = s(\!|ik := \{K\} \cup ik\ s|\!)$
}

Protocol events:

- step 1: create *Ra*, *A* generates *nx*, computes and insecurely sends g^{nx}
- step 2: create *Rb*, *B* receives g^{nx} insecurely, generates *ny*, computes g^{ny}, authentically sends (g^{ny}, g^{nx}), computes $g^{nx\ *\ ny}$, emits a running signal for *Init*, $g^{nx\ *\ ny}$
- step 3: *A* receives g^{ny} and g^{nx} authentically, sends (g^{nx}, g^{ny}) authentically, computes $g^{ny\ *\ nx}$, emits a commit signal for *Init*, $g^{ny\ *\ nx}$, a running signal for *Resp*, $g^{ny\ *\ nx}$, declares the secret $g^{ny\ *\ nx}$
- step 4: *B* receives g^{nx} and g^{ny} authentically, emits a commit signal for *Resp*, $g^{nx\ *\ ny}$, declares the secret $g^{nx\ *\ ny}$

definition
  l2-step1 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *l2-trans*
where
  l2-step1 Ra A B ≡ {(*s*, *s'*).
    — guards:
    Ra ∉ *dom* (*progress s*) ∧
    guessed-runs Ra = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ∧
    — actions:
    s' = *s*(|
      progress := (*progress s*)(*Ra* ↦ {*xnx*, *xgnx*}),
      chan := {*Insec A B* (*Exp Gen* (*NonceF* (*Ra*$*nx*)))} ∪ (*chan s*)
    |)
  }

definition
  l2-step2 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *msg* ⇒ *l2-trans*
where
  l2-step2 Rb A B gnx ≡ {(*s*, *s'*).
    — guards:
    guessed-runs Rb = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|) ∧
    Rb ∉ *dom* (*progress s*) ∧
    guessed-frame Rb xgnx = *Some gnx* ∧
    guessed-frame Rb xsk = *Some* (*Exp gnx* (*NonceF* (*Rb*$*ny*))) ∧
    Insec A B gnx ∈ *chan s* ∧
    — actions:
    s' = *s*(| *progress* := (*progress s*)(*Rb* ↦ {*xny*, *xgny*, *xgnx*, *xsk*}),
        chan := {*Auth B A* ⟨*Number 0*, *Exp Gen* (*NonceF* (*Rb*$*ny*)), *gnx*⟩} ∪ (*chan s*),
        signalsInit := *if can-signal s A B then*
            addSignal (*signalsInit s*) (*Running A B* (*Exp gnx* (*NonceF* (*Rb*$*ny*))))
          else
            signalsInit s
      |)
  }

definition
  l2-step3 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *msg* ⇒ *l2-trans*
where
  l2-step3 Ra A B gny ≡ {(*s*, *s'*).
    — guards:

guessed-runs Ra = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ∧
progress s Ra = *Some* {*xnx*, *xgnx*} ∧
guessed-frame Ra xgny = *Some gny* ∧
guessed-frame Ra xsk = *Some* (*Exp gny* (*NonceF* (*Ra*$*nx*))) ∧
Auth B A ⟨*Number 0*, *gny*, *Exp Gen* (*NonceF* (*Ra*$*nx*))⟩ ∈ *chan s* ∧
— actions:
s' = *s*(| *progress* := (*progress s*)(*Ra* ↦ {*xnx*, *xgnx*, *xgny*, *xsk*, *xEnd*}),
chan := {*Auth A B* ⟨*Number 1*, *Exp Gen* (*NonceF* (*Ra*$*nx*)), *gny*⟩} ∪ *chan s*,
secret := {*x*. *x* = *Exp gny* (*NonceF* (*Ra*$*nx*)) ∧ *Ra* = *test*} ∪ *secret s*,
signalsInit := *if can-signal s A B then*
addSignal (*signalsInit s*) (*Commit A B* (*Exp gny* (*NonceF* (*Ra*$*nx*))))
else
signalsInit s,
signalsResp := *if can-signal s A B then*
addSignal (*signalsResp s*) (*Running A B* (*Exp gny* (*NonceF* (*Ra*$*nx*))))
else
signalsResp s
|)
}

definition
l2-step4 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *msg* ⇒ *l2-trans*
where
l2-step4 Rb A B gnx ≡ {(*s*, *s'*).
— guards:
guessed-runs Rb = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|) ∧
progress s Rb = *Some* {*xny*, *xgnx*, *xgny*, *xsk*} ∧
guessed-frame Rb xgnx = *Some gnx* ∧
Auth A B ⟨*Number 1*, *gnx*, *Exp Gen* (*NonceF* (*Rb*$*ny*))⟩ ∈ *chan s* ∧

— actions:
s' = *s*(| *progress* := (*progress s*)(*Rb* ↦ {*xny*, *xgnx*, *xgny*, *xsk*, *xEnd*}),
secret := {*x*. *x* = *Exp gnx* (*NonceF* (*Rb*$*ny*)) ∧ *Rb* = *test*} ∪ *secret s*,
signalsResp := *if can-signal s A B then*
addSignal (*signalsResp s*) (*Commit A B* (*Exp gnx* (*NonceF* (*Rb*$*ny*))))
else
signalsResp s
|)
}

Specification.

definition
l2-init :: *l2-state set*
where
l2-init ≡ { (|
ik = {},
secret = {},
progress = *Map.empty*,
signalsInit = λ*x*. *0*,
signalsResp = λ*x*. *0*,
chan = {},
bad = *bad-init*

|)}

definition
l2-trans :: *l2-trans* **where**
l2-trans ≡ (⋃ *m M X Rb Ra A B K*.
l2-step1 Ra A B ∪
l2-step2 Rb A B X ∪
l2-step3 Ra A B X ∪
l2-step4 Rb A B X ∪
l2-dy-fake-chan M ∪
l2-dy-fake-msg m ∪
l2-lkr-others A ∪
l2-lkr-after A ∪
l2-skr Ra K ∪
Id
)

definition
l2 :: (*l2-state*, *l2-obs*) *spec* **where**
l2 ≡ (|
init = *l2-init*,
trans = *l2-trans*,
obs = *id*
|)

lemmas *l2-loc-defs* =
l2-step1-def l2-step2-def l2-step3-def l2-step4-def
l2-def l2-init-def l2-trans-def
l2-dy-fake-chan-def l2-dy-fake-msg-def
l2-lkr-after-def l2-lkr-others-def l2-skr-def

lemmas *l2-defs* = *l2-loc-defs ik-dy-def*

lemmas *l2-nostep-defs* = *l2-def l2-init-def l2-trans-def*

lemma *l2-obs-id* [*simp*]: *obs l2* = *id*
⟨*proof*⟩

Once a run is finished, it stays finished, therefore if the test is not finished at some point then it was not finished before either.

declare *domIff* [*iff*]
lemma *l2-run-ended-trans*:
run-ended (*progress s R*) ⟹
(*s*, *s'*) ∈ *trans l2* ⟹
run-ended (*progress s' R*)
⟨*proof*⟩
declare *domIff* [*iff del*]

lemma *l2-can-signal-trans*:
can-signal s' A B ⟹
(*s*, *s'*) ∈ *trans l2* ⟹

can-signal s A B
⟨*proof*⟩

22.2 Invariants

22.2.1 inv1

If *can-signal s A B* (i.e., *A*, *B* are the test session agents and the test is not finished), then A and B are honest.

definition
l2-inv1 :: *l2-state set*
where
l2-inv1 ≡ {*s*. ∀ *A B*.
can-signal s A B ⟶
A ∉ *bad s* ∧ *B* ∉ *bad s*
}

lemmas *l2-inv1I* = *l2-inv1-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l2-inv1E* [*elim*] = *l2-inv1-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l2-inv1D* = *l2-inv1-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l2-inv1-init* [*iff*]:
init l2 ⊆ *l2-inv1*
⟨*proof*⟩

lemma *l2-inv1-trans* [*iff*]:
{*l2-inv1*} *trans l2* {> *l2-inv1*}
⟨*proof*⟩

lemma *PO-l2-inv1* [*iff*]: *reach l2* ⊆ *l2-inv1*
⟨*proof*⟩

22.2.2 inv2 (authentication guard)

If *Auth B A* ⟨*Number 0*, *gny*, *Exp Gen* (*NonceF* (*Ra* $ *nx*))⟩ ∈ *chan s* and *A*, *B* are honest then the message has indeed been sent by a responder run (etc).

definition
l2-inv2 :: *l2-state set*
where
l2-inv2 ≡ {*s*. ∀ *Ra A B gny*.
Auth B A ⟨*Number 0*, *gny*, *Exp Gen* (*NonceF* (*Ra*$*nx*))⟩ ∈ *chan s* ⟶
A ∉ *bad s* ∧ *B* ∉ *bad s* ⟶
(∃ *Rb*. *guessed-runs Rb* = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|) ∧
in-progressS (*progress s Rb*) {*xny*, *xgnx*, *xgny*, *xsk*} ∧
gny = *Exp Gen* (*NonceF* (*Rb*$*ny*)) ∧
guessed-frame Rb xgnx = *Some* (*Exp Gen* (*NonceF* (*Ra*$*nx*))))
}

lemmas *l2-inv2I* = *l2-inv2-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l2-inv2E* [*elim*] = *l2-inv2-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l2-inv2D* = *l2-inv2-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l2-inv2-init* [*iff*]:
  init l2 ⊆ *l2-inv2*
⟨*proof*⟩

lemma *l2-inv2-trans* [*iff*]:
  {*l2-inv2*} *trans l2* {> *l2-inv2*}
⟨*proof*⟩

lemma *PO-l2-inv2* [*iff*]: *reach l2* ⊆ *l2-inv2*
⟨*proof*⟩

22.2.3 inv3 (authentication guard)

If *Auth A B* ⟨*Number 1*, *gnx*, *Exp Gen* (*NonceF* (*Rb$ny*))⟩ ∈ *chan s* and *A*, *B* are honest then the message has indeed been sent by an initiator run (etc).

definition
  l2-inv3 :: *l2-state set*
where
  l2-inv3 ≡ {*s*. ∀ *Rb A B gnx*.
    Auth A B ⟨*Number 1*, *gnx*, *Exp Gen* (*NonceF* (*Rb$ny*))⟩ ∈ *chan s* ⟶
    A ∉ *bad s* ∧ *B* ∉ *bad s* ⟶
     (∃ *Ra*. *guessed-runs Ra* = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ∧
          in-progressS (*progress s Ra*) {*xnx*, *xgnx*, *xgny*, *xsk*, *xEnd*} ∧
          guessed-frame Ra xgnx = *Some gnx* ∧
          guessed-frame Ra xgny = *Some* (*Exp Gen* (*NonceF* (*Rb$ny*))))
  }

lemmas *l2-inv3I* = *l2-inv3-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l2-inv3E* [*elim*] = *l2-inv3-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l2-inv3D* = *l2-inv3-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l2-inv3-init* [*iff*]:
  init l2 ⊆ *l2-inv3*
⟨*proof*⟩

lemma *l2-inv3-trans* [*iff*]:
  {*l2-inv3*} *trans l2* {> *l2-inv3*}
⟨*proof*⟩

lemma *PO-l2-inv3* [*iff*]: *reach l2* ⊆ *l2-inv3*
⟨*proof*⟩

22.2.4 inv4

For an initiator, the session key is always *gny*^*nx*.

definition
  l2-inv4 :: *l2-state set*
where
  l2-inv4 ≡ {*s*. ∀ *Ra A B gny*.
   guessed-runs Ra = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ⟶
   in-progress (*progress s Ra*) *xsk* ⟶

guessed-frame Ra xgny = Some gny $\longrightarrow$
guessed-frame Ra xsk = Some (Exp gny (NonceF (Ra$nx)))
}

lemmas *l2-inv4I = l2-inv4-def [THEN setc-def-to-intro, rule-format]*
lemmas *l2-inv4E [elim] = l2-inv4-def [THEN setc-def-to-elim, rule-format]*
lemmas *l2-inv4D = l2-inv4-def [THEN setc-def-to-dest, rule-format, rotated 1, simplified]*

lemma *l2-inv4-init [iff]*:
init l2 $\subseteq$ *l2-inv4*
⟨*proof*⟩

lemma *l2-inv4-trans [iff]*:
{*l2-inv4*} *trans l2* {> *l2-inv4*}
⟨*proof*⟩

lemma *PO-l2-inv4 [iff]*: *reach l2* $\subseteq$ *l2-inv4*
⟨*proof*⟩

22.2.5 inv4'

For a responder, the session key is always *gnx^ny*.

definition
l2-inv4' :: *l2-state set*
where
l2-inv4' ≡ {*s*. ∀ *Rb A B gnx*.
guessed-runs Rb = (|*role=Resp, owner=B, partner=A*|) $\longrightarrow$
in-progress (progress s Rb) xsk $\longrightarrow$
guessed-frame Rb xgnx = Some gnx $\longrightarrow$
guessed-frame Rb xsk = Some (Exp gnx (NonceF (Rb$ny)))
}

lemmas *l2-inv4'I = l2-inv4'-def [THEN setc-def-to-intro, rule-format]*
lemmas *l2-inv4'E [elim] = l2-inv4'-def [THEN setc-def-to-elim, rule-format]*
lemmas *l2-inv4'D = l2-inv4'-def [THEN setc-def-to-dest, rule-format, rotated 1, simplified]*

lemma *l2-inv4'-init [iff]*:
init l2 $\subseteq$ *l2-inv4'*
⟨*proof*⟩

lemma *l2-inv4'-trans [iff]*:
{*l2-inv4'*} *trans l2* {> *l2-inv4'*}
⟨*proof*⟩

lemma *PO-l2-inv4' [iff]*: *reach l2* $\subseteq$ *l2-inv4'*
⟨*proof*⟩

22.2.6 inv5

The only confidential or secure messages on the channel have been put there by the attacker.

definition
l2-inv5 :: *l2-state set*

where
  l2-inv5 ≡ {*s*. ∀ *A B M*.
    (*Confid A B M* ∈ *chan s* ∨ *Secure A B M* ∈ *chan s*) ⟶
    M ∈ *dy-fake-msg* (*bad s*) (*ik s*) (*chan s*)
  }

lemmas *l2-inv5I* = *l2-inv5-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l2-inv5E* [*elim*] = *l2-inv5-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l2-inv5D* = *l2-inv5-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l2-inv5-init* [*iff*]:
  init l2 ⊆ *l2-inv5*
⟨*proof*⟩

lemma *l2-inv5-trans* [*iff*]:
  {*l2-inv5*} *trans l2* {> *l2-inv5*}
⟨*proof*⟩

lemma *PO-l2-inv5* [*iff*]: *reach l2* ⊆ *l2-inv5*
⟨*proof*⟩

22.2.7 inv6

For a run R (with any role), the session key always has the form $something^n$ where n is a nonce generated by R.

definition
  l2-inv6 :: *l2-state set*
where
  l2-inv6 ≡ {*s*. ∀ *R*.
    in-progress (*progress s R*) *xsk* ⟶
    (∃ *X N*.
      guessed-frame R xsk = *Some* (*Exp X* (*NonceF* (*R*$*N*))))
  }

lemmas *l2-inv6I* = *l2-inv6-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l2-inv6E* [*elim*] = *l2-inv6-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l2-inv6D* = *l2-inv6-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l2-inv6-init* [*iff*]:
  init l2 ⊆ *l2-inv6*
⟨*proof*⟩

lemma *l2-inv6-trans* [*iff*]:
  {*l2-inv6*} *trans l2* {> *l2-inv6*}
⟨*proof*⟩

lemma *PO-l2-inv6* [*iff*]: *reach l2* ⊆ *l2-inv6*
⟨*proof*⟩

22.2.8 inv7

Form of the messages in *extr* (*bad s*) (*ik s*) (*chan s*) = *synth* (*analz generators*).

abbreviation
generators ≡ {*x*. ∃ *N*. *x* = *Exp Gen* (*Nonce N*)} ∪
{*Exp y* (*NonceF* (*R*$*N*)) |*y N R*. *R* ≠ *test* ∧ *R* ∉ *partners*}

lemma *analz-generators*: *analz generators* = *generators*
⟨*proof*⟩

definition
l2-inv7 :: *l2-state set*
where
l2-inv7 ≡ {*s*.
extr (*bad s*) (*ik s*) (*chan s*) ⊆
synth (*analz* (*generators*))
}

lemmas *l2-inv7I* = *l2-inv7-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l2-inv7E* [*elim*] = *l2-inv7-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l2-inv7D* = *l2-inv7-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l2-inv7-init* [*iff*]:
init l2 ⊆ *l2-inv7*
⟨*proof*⟩

lemma *l2-inv7-step1*:
{*l2-inv7*} *l2-step1 Ra A B* {> *l2-inv7*}
⟨*proof*⟩

lemma *l2-inv7-step2*:
{*l2-inv7*} *l2-step2 Rb A B gnx* {> *l2-inv7*}
⟨*proof*⟩

lemma *l2-inv7-step3*:
{*l2-inv7*} *l2-step3 Ra A B gny* {> *l2-inv7*}
⟨*proof*⟩

lemma *l2-inv7-step4*:
{*l2-inv7*} *l2-step4 Rb A B gnx* {> *l2-inv7*}
⟨*proof*⟩

lemma *l2-inv7-dy-fake-msg*:
{*l2-inv7*} *l2-dy-fake-msg M* {> *l2-inv7*}
⟨*proof*⟩

lemma *l2-inv7-dy-fake-chan*:
{*l2-inv7*} *l2-dy-fake-chan M* {> *l2-inv7*}
⟨*proof*⟩

lemma *l2-inv7-lkr-others*:
{*l2-inv7* ∩ *l2-inv5*} *l2-lkr-others A* {> *l2-inv7*}
⟨*proof*⟩

lemma *l2-inv7-lkr-after*:
{*l2-inv7* ∩ *l2-inv5*} *l2-lkr-after A* {> *l2-inv7*}

⟨*proof*⟩

lemma *l2-inv7-skr*:
  {*l2-inv7* ∩ *l2-inv6*} *l2-skr R K* {> *l2-inv7*}

⟨*proof*⟩

lemmas *l2-inv7-trans-aux* =
  l2-inv7-step1 l2-inv7-step2 l2-inv7-step3 l2-inv7-step4
  l2-inv7-dy-fake-msg l2-inv7-dy-fake-chan
  l2-inv7-lkr-others l2-inv7-lkr-after l2-inv7-skr

lemma *l2-inv7-trans* [*iff*]:
  {*l2-inv7* ∩ *l2-inv5* ∩ *l2-inv6*} *trans l2* {> *l2-inv7*}
⟨*proof*⟩

lemma *PO-l2-inv7* [*iff*]: *reach l2* ⊆ *l2-inv7*
⟨*proof*⟩

Auxiliary dest rule for inv7.

lemmas *l2-inv7D-aux* =
  l2-inv7D [*THEN* [*2*] *subset-trans*, *THEN synth-analz-mono*, *simplified*,
    THEN [*2*] *rev-subsetD*, *rotated 1*, *OF IK-subset-extr*]

22.2.9 inv8: form of the secrets

definition
  l2-inv8 :: *l2-state set*
where
  l2-inv8 ≡ {*s*.
    secret s ⊆ {*Exp* (*Exp Gen* (*NonceF* (*R*\$*N*))) (*NonceF* (*R′*\$*N′*)) | *N N′ R R′*.
      R = *test* ∧ *R′* ∈ *partners*}
  }

lemmas *l2-inv8I* = *l2-inv8-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l2-inv8E* [*elim*] = *l2-inv8-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l2-inv8D* = *l2-inv8-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l2-inv8-init* [*iff*]:
  init l2 ⊆ *l2-inv8*
⟨*proof*⟩

Steps 3 and 4 are the hard part.

lemma *l2-inv8-step3*:
  {*l2-inv8* ∩ *l2-inv1* ∩ *l2-inv2* ∩ *l2-inv4′*} *l2-step3 Ra A B gny* {> *l2-inv8*}
⟨*proof*⟩

lemma *l2-inv8-step4*:
  {*l2-inv8* ∩ *l2-inv1* ∩ *l2-inv3* ∩ *l2-inv4* ∩ *l2-inv4′*} *l2-step4 Rb A B gnx* {> *l2-inv8*}
⟨*proof*⟩

lemma *l2-inv8-trans* [*iff*]:

{*l2-inv8* ∩ *l2-inv1* ∩ *l2-inv2* ∩ *l2-inv3* ∩ *l2-inv4* ∩ *l2-inv4'*} *trans l2* {> *l2-inv8*}
⟨*proof*⟩

lemma *PO-l2-inv8* [*iff*]: *reach l2* ⊆ *l2-inv8*
⟨*proof*⟩

Auxiliary destruction rule for inv8.

lemma *Exp-Exp-Gen-synth*:
Exp (*Exp Gen X*) *Y* ∈ *synth H* ⟹ *Exp* (*Exp Gen X*) *Y* ∈ *H* ∨ *X* ∈ *synth H* ∨ *Y* ∈ *synth H*
⟨*proof*⟩

lemma *l2-inv8-aux*:
s ∈ *l2-inv8* ⟹
x ∈ *secret s* ⟹
x ∉ *synth* (*analz generators*)
⟨*proof*⟩

22.3 Refinement

Mediator function.

definition
med12s :: *l2-obs* ⇒ *l1-obs*
where
med12s t ≡ ⦇
ik = *ik t*,
secret = *secret t*,
progress = *progress t*,
signalsInit = *signalsInit t*,
signalsResp = *signalsResp t*
⦈

Relation between states.

definition
R12s :: (*l1-state* ∗ *l2-state*) *set*
where
R12s ≡ {(*s*,*s'*).
s = *med12s s'*
}

lemmas *R12s-defs* = *R12s-def med12s-def*

lemma *can-signal-R12* [*simp*]:
(*s1*, *s2*) ∈ *R12s* ⟹
can-signal s1 A B ⟷ *can-signal s2 A B*
⟨*proof*⟩

Protocol events.

lemma *l2-step1-refines-step1*:
{*R12s*} *l1-step1 Ra A B*, *l2-step1 Ra A B* {>*R12s*}
⟨*proof*⟩

lemma *l2-step2-refines-step2*:
  {*R12s*} *l1-step2 Rb A B gnx*, *l2-step2 Rb A B gnx* {>*R12s*}
⟨*proof*⟩

For step3 and 4, we prove the level 1 guard, i.e., "the future session key is not in *synth* (*analz* (*ik s*))" using the fact that inv8 also holds for the future state in which the session key is already in *secret s*.

lemma *l2-step3-refines-step3*:
  {*R12s* ∩ *UNIV* × (*l2-inv1* ∩ *l2-inv2* ∩ *l2-inv4'* ∩ *l2-inv7* ∩ *l2-inv8*)}
    l1-step3 Ra A B gny, *l2-step3 Ra A B gny*
  {>*R12s*}
⟨*proof*⟩

lemma *l2-step4-refines-step4*:
  {*R12s* ∩ *UNIV* × (*l2-inv1* ∩ *l2-inv3* ∩ *l2-inv4* ∩ *l2-inv4'* ∩ *l2-inv7* ∩ *l2-inv8*)}
    l1-step4 Rb A B gnx, *l2-step4 Rb A B gnx*
  {>*R12s*}
⟨*proof*⟩

Attacker events.

lemma *l2-dy-fake-chan-refines-skip*:
  {*R12s*} *Id*, *l2-dy-fake-chan M* {>*R12s*}
⟨*proof*⟩

lemma *l2-dy-fake-msg-refines-learn*:
  {*R12s* ∩ *UNIV* × (*l2-inv7* ∩ *l2-inv8*)} *l1-learn m*, *l2-dy-fake-msg m* {>*R12s*}
⟨*proof*⟩

Compromising events.

lemma *l2-lkr-others-refines-skip*:
  {*R12s*} *Id*, *l2-lkr-others A* {>*R12s*}
⟨*proof*⟩

lemma *l2-lkr-after-refines-skip*:
  {*R12s*} *Id*, *l2-lkr-after A* {>*R12s*}
⟨*proof*⟩

lemma *l2-skr-refines-learn*:
  {*R12s* ∩ *UNIV* × *l2-inv7* ∩ *UNIV* × *l2-inv6* ∩ *UNIV* × *l2-inv8*} *l1-learn K*, *l2-skr R K* {>*R12s*}
⟨*proof*⟩

Refinement proof.

lemmas *l2-trans-refines-l1-trans* =
  l2-dy-fake-msg-refines-learn l2-dy-fake-chan-refines-skip
  l2-lkr-others-refines-skip l2-lkr-after-refines-skip l2-skr-refines-learn
  l2-step1-refines-step1 l2-step2-refines-step2 l2-step3-refines-step3 l2-step4-refines-step4

lemma *l2-refines-init-l1* [*iff*]:
  init l2 ⊆ *R12s* ‘‘ (*init l1*)
⟨*proof*⟩

lemma *l2-refines-trans-l1* [*iff*]:
{*R12s* ∩ (*UNIV* × (*l2-inv1* ∩ *l2-inv2* ∩ *l2-inv3* ∩ *l2-inv4* ∩ *l2-inv4'* ∩
l2-inv6 ∩ *l2-inv7* ∩ *l2-inv8*))}
trans l1, *trans l2*
{> *R12s*}
⟨*proof*⟩

lemma *PO-obs-consistent-R12s* [*iff*]:
obs-consistent R12s med12s l1 l2
⟨*proof*⟩

lemma *l2-refines-l1* [*iff*]:
refines
(*R12s* ∩
(*reach l1* × (*l2-inv1* ∩ *l2-inv2* ∩ *l2-inv3* ∩ *l2-inv4* ∩ *l2-inv4'* ∩ *l2-inv5* ∩
l2-inv6 ∩ *l2-inv7* ∩ *l2-inv8*)))
med12s l1 l2
⟨*proof*⟩

lemma *l2-implements-l1* [*iff*]:
implements med12s l1 l2
⟨*proof*⟩

22.4 Derived invariants

We want to prove *l2-secrecy*: *dy-fake-msg* (*bad s*) (*ik s*) (*chan s*) ∩ *secret s* = {} but by refinement we only get *l2-partial-secrecy*: *synth* (*analz* (*ik s*)) ∩ *secret s* = {} This is fine, since a message in *dy-fake-msg* (*bad s*) (*ik s*) (*chan s*) could be added to *ik s*, and *l2-partial-secrecy* would still hold for this new state.

definition
l2-partial-secrecy :: (*'a l2-state-scheme*) *set*
where
l2-partial-secrecy ≡ {*s*. *synth* (*analz* (*ik s*)) ∩ *secret s* = {}}

lemma *l2-obs-partial-secrecy* [*iff*]: *oreach l2* ⊆ *l2-partial-secrecy*
⟨*proof*⟩

lemma *l2-oreach-dy-fake-msg*:
⟦ *s* ∈ *oreach l2*; *x* ∈ *dy-fake-msg* (*bad s*) (*ik s*) (*chan s*) ⟧
⟹ *s* ⦇*ik* := *insert x* (*ik s*)⦈ ∈ *oreach l2*
⟨*proof*⟩

definition
l2-secrecy :: (*'a l2-state-scheme*) *set*
where
l2-secrecy ≡ {*s*. *dy-fake-msg* (*bad s*) (*ik s*) (*chan s*) ∩ *secret s* = {}}

lemma *l2-obs-secrecy* [*iff*]: *oreach l2* ⊆ *l2-secrecy*

⟨*proof*⟩

lemma *l2-secrecy* [*iff*]: *reach l2* ⊆ *l2-secrecy*
⟨*proof*⟩

abbreviation *l2-iagreement-Init* ≡ *l1-iagreement-Init*

lemma *l2-obs-iagreement-Init* [*iff*]: *oreach l2* ⊆ *l2-iagreement-Init*
⟨*proof*⟩

lemma *l2-iagreement-Init* [*iff*]: *reach l2* ⊆ *l2-iagreement-Init*
⟨*proof*⟩

abbreviation *l2-iagreement-Resp* ≡ *l1-iagreement-Resp*

lemma *l2-obs-iagreement-Resp* [*iff*]: *oreach l2* ⊆ *l2-iagreement-Resp*
⟨*proof*⟩

lemma *l2-iagreement-Resp* [*iff*]: *reach l2* ⊆ *l2-iagreement-Resp*
⟨*proof*⟩

end

23 Authenticated Diffie-Hellman Protocol (L3 locale)

theory *dhlvl3*
imports *dhlvl2 Implem-lemmas*
begin

23.1 State and Events

Level 3 state.

(The types have to be defined outside the locale.)

record *l3-state* = *l1-state* +
  bad :: *agent set*

type-synonym *l3-obs* = *l3-state*

type-synonym
  l3-pred = *l3-state set*

type-synonym
  l3-trans = (*l3-state* × *l3-state*) *set*

Attacker event.

definition
  l3-dy :: *msg* ⇒ *l3-trans*
where
  l3-dy ≡ *ik-dy*

Compromise events.

definition
  l3-lkr-others :: *agent* ⇒ *l3-trans*
where
  l3-lkr-others A ≡ {(*s*,*s'*).
    — guards
    A ≠ *test-owner* ∧
    A ≠ *test-partner* ∧
    — actions
    s' = *s*(|*bad* := {*A*} ∪ *bad s*,
          ik := *keys-of A* ∪ *ik s*|)
  }

definition
  l3-lkr-actor :: *agent* ⇒ *l3-trans*
where
  l3-lkr-actor A ≡ {(*s*,*s'*).
    — guards
    A = *test-owner* ∧
    A ≠ *test-partner* ∧
    — actions
    s' = *s*(|*bad* := {*A*} ∪ *bad s*,
          ik := *keys-of A* ∪ *ik s*|)
  }

definition
  l3-lkr-after :: *agent* ⇒ *l3-trans*
where
  l3-lkr-after A ≡ {(*s*,*s'*).
    — guards
    test-ended s ∧
    — actions
    s' = *s*(|*bad* := {*A*} ∪ *bad s*,
        ik := *keys-of A* ∪ *ik s*|)
  }

definition
  l3-skr :: *rid-t* ⇒ *msg* ⇒ *l3-trans*
where
  l3-skr R K ≡ {(*s*,*s'*).
    — guards
    R ≠ *test* ∧ *R* ∉ *partners* ∧
    in-progress (*progress s R*) *xsk* ∧
    guessed-frame R xsk = *Some K* ∧
    — actions
    s' = *s*(|*ik* := {*K*} ∪ *ik s*|)
  }

New locale for the level 3 protocol. This locale does not add new assumptions, it is only used to separate the level 3 protocol from the implementation locale.

locale *dhlvl3* = *valid-implem*
begin

Protocol events:

- step 1: create *Ra*, *A* generates *nx*, computes and insecurely sends g^{nx}
- step 2: create *Rb*, *B* receives g^{nx} insecurely, generates *ny*, computes g^{ny}, authentically sends (g^{ny}, g^{nx}), computes $g^{nx * ny}$, emits a running signal for *Init*, $g^{nx * ny}$
- step 3: *A* receives g^{ny} and g^{nx} authentically, sends (g^{nx}, g^{ny}) authentically, computes $g^{ny * nx}$, emits a commit signal for *Init*, $g^{ny * nx}$, a running signal for *Resp*, $g^{ny * nx}$, declares the secret $g^{ny * nx}$
- step 4: *B* receives g^{nx} and g^{ny} authentically, emits a commit signal for *Resp*, $g^{nx * ny}$, declares the secret $g^{nx * ny}$

definition
    l3-step1 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *l3-trans*
where
  l3-step1 Ra A B ≡ {(*s*, *s'*).
    — guards:
    Ra ∉ *dom* (*progress s*) ∧
    guessed-runs Ra = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ∧
    — actions:
    s' = *s*(|

progress := (*progress s*)(*Ra* ↦ {*xnx*, *xgnx*}),
ik := {*implInsec A B* (*Exp Gen* (*NonceF* (*Ra*\$*nx*)))} ∪ *ik s*
⦈
}

definition
l3-step2 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *msg* ⇒ *l3-trans*
where
l3-step2 Rb A B gnx ≡ {(*s*, *s*′).
— guards:
guessed-runs Rb = ⦇*role*=*Resp*, *owner*=*B*, *partner*=*A*⦈ ∧
Rb ∉ *dom* (*progress s*) ∧
guessed-frame Rb xgnx = *Some gnx* ∧
guessed-frame Rb xsk = *Some* (*Exp gnx* (*NonceF* (*Rb*\$*ny*))) ∧
implInsec A B gnx ∈ *ik s* ∧
— actions:
s′ = *s*⦇ *progress* := (*progress s*)(*Rb* ↦ {*xny*, *xgny*, *xgnx*, *xsk*}),
ik := {*implAuth B A* ⟨*Number 0*, *Exp Gen* (*NonceF* (*Rb*\$*ny*)), *gnx*⟩} ∪ *ik s*,
signalsInit := *if can-signal s A B then*
addSignal (*signalsInit s*) (*Running A B* (*Exp gnx* (*NonceF* (*Rb*\$*ny*))))
else
signalsInit s
⦈
}

definition
l3-step3 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *msg* ⇒ *l3-trans*
where
l3-step3 Ra A B gny ≡ {(*s*, *s*′).
— guards:
guessed-runs Ra = ⦇*role*=*Init*, *owner*=*A*, *partner*=*B*⦈ ∧
progress s Ra = *Some* {*xnx*, *xgnx*} ∧
guessed-frame Ra xgny = *Some gny* ∧
guessed-frame Ra xsk = *Some* (*Exp gny* (*NonceF* (*Ra*\$*nx*))) ∧
implAuth B A ⟨*Number 0*, *gny*, *Exp Gen* (*NonceF* (*Ra*\$*nx*))⟩ ∈ *ik s* ∧
— actions:
s′ = *s*⦇ *progress* := (*progress s*)(*Ra* ↦ {*xnx*, *xgnx*, *xgny*, *xsk*, *xEnd*}),
ik := {*implAuth A B* ⟨*Number 1*, *Exp Gen* (*NonceF* (*Ra*\$*nx*)), *gny*⟩} ∪ *ik s*,
secret := {*x*. *x* = *Exp gny* (*NonceF* (*Ra*\$*nx*)) ∧ *Ra* = *test*} ∪ *secret s*,
signalsInit := *if can-signal s A B then*
addSignal (*signalsInit s*) (*Commit A B* (*Exp gny* (*NonceF* (*Ra*\$*nx*))))
else
signalsInit s,
signalsResp := *if can-signal s A B then*
addSignal (*signalsResp s*) (*Running A B* (*Exp gny* (*NonceF* (*Ra*\$*nx*))))
else
signalsResp s
⦈
}

definition

l3-step4 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *msg* ⇒ *l3-trans*
where
l3-step4 Rb A B gnx ≡ {(*s*, *s*′).
— guards:
guessed-runs Rb = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|) ∧
progress s Rb = *Some* {*xny*, *xgnx*, *xgny*, *xsk*} ∧
guessed-frame Rb xgnx = *Some gnx* ∧
implAuth A B ⟨*Number 1*, *gnx*, *Exp Gen* (*NonceF* (*Rb*$*ny*))⟩ ∈ *ik s* ∧

— actions:
s′ = *s*(| *progress* := (*progress s*)(*Rb* ↦ {*xny*, *xgnx*, *xgny*, *xsk*, *xEnd*}),
secret := {*x*. *x* = *Exp gnx* (*NonceF* (*Rb*$*ny*)) ∧ *Rb* = *test*} ∪ *secret s*,
signalsResp := *if can-signal s A B then*
addSignal (*signalsResp s*) (*Commit A B* (*Exp gnx* (*NonceF* (*Rb*$*ny*))))
else
signalsResp s
|)
}

Specification.

Initial compromise.

definition
ik-init :: *msg set*
where
ik-init ≡ {*priK C* |*C*. *C* ∈ *bad-init*} ∪ {*pubK A* | *A*. *True*} ∪
{*shrK A B* |*A B*. *A* ∈ *bad-init* ∨ *B* ∈ *bad-init*} ∪ *Tags*

Lemmas about *ik-init*.

lemma *parts-ik-init* [*simp*]: *parts ik-init* = *ik-init*
⟨*proof*⟩

lemma *analz-ik-init* [*simp*]: *analz ik-init* = *ik-init*
⟨*proof*⟩

lemma *abs-ik-init* [*iff*]: *abs ik-init* = {}
⟨*proof*⟩

lemma *payloadSet-ik-init* [*iff*]: *ik-init* ∩ *payload* = {}
⟨*proof*⟩

lemma *validSet-ik-init* [*iff*]: *ik-init* ∩ *valid*= {}
⟨*proof*⟩

definition
l3-init :: *l3-state set*
where
l3-init ≡ { (|
ik = *ik-init*,
secret = {},
progress = *Map.empty*,
signalsInit = *λx*. *0*,

signalsResp = $\lambda x.\ 0$,
bad = *bad-init*
⦈}

lemmas *l3-init-defs* = *l3-init-def ik-init-def*

definition
l3-trans :: *l3-trans*
where
l3-trans ≡ (⋃ *M X Rb Ra A B K*.
l3-step1 Ra A B ∪
l3-step2 Rb A B X ∪
l3-step3 Ra A B X ∪
l3-step4 Rb A B X ∪
l3-dy M ∪
l3-lkr-others A ∪
l3-lkr-after A ∪
l3-skr Ra K ∪
Id
)

definition
l3 :: (*l3-state*, *l3-obs*) *spec* **where**
l3 ≡ ⦇
init = *l3-init*,
trans = *l3-trans*,
obs = *id*
⦈

lemmas *l3-loc-defs* =
l3-step1-def l3-step2-def l3-step3-def l3-step4-def
l3-def l3-init-defs l3-trans-def
l3-dy-def
l3-lkr-others-def l3-lkr-after-def l3-skr-def

lemmas *l3-defs* = *l3-loc-defs ik-dy-def*
lemmas *l3-nostep-defs* = *l3-def l3-init-def l3-trans-def*

lemma *l3-obs-id* [*simp*]: *obs l3* = *id*
⟨*proof*⟩

23.2 Invariants

23.2.1 inv1: No long-term keys as message parts

definition
l3-inv1 :: *l3-state set*
where
l3-inv1 ≡ {*s*.
parts (*ik s*) ∩ *range LtK* ⊆ *ik s*
}

lemmas *l3-inv1I* = *l3-inv1-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv1E* [*elim*] = *l3-inv1-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv1D* = *l3-inv1-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv1D′* [*dest*]: ⟦ *LtK K* ∈ *parts* (*ik s*); *s* ∈ *l3-inv1*⟧ ⟹ *LtK K* ∈ *ik s*
⟨*proof*⟩

lemma *l3-inv1-init* [*iff*]:
  init l3 ⊆ *l3-inv1*
⟨*proof*⟩

lemma *l3-inv1-trans* [*iff*]:
  {*l3-inv1*} *trans l3* {> *l3-inv1*}
⟨*proof*⟩

lemma *PO-l3-inv1* [*iff*]:
  reach l3 ⊆ *l3-inv1*
⟨*proof*⟩

23.2.2 inv2: *l3-state.bad s* indeed contains "bad" keys

definition
  l3-inv2 :: *l3-state set*
where
  l3-inv2 ≡ {*s*.
    Keys-bad (*ik s*) (*bad s*)
  }

lemmas *l3-inv2I* = *l3-inv2-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv2E* [*elim*] = *l3-inv2-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv2D* = *l3-inv2-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv2-init* [*simp*,*intro*!]:
  init l3 ⊆ *l3-inv2*
⟨*proof*⟩

lemma *l3-inv2-trans* [*simp*,*intro*!]:
  {*l3-inv2* ∩ *l3-inv1*} *trans l3* {> *l3-inv2*}
⟨*proof*⟩

lemma *PO-l3-inv2* [*iff*]: *reach l3* ⊆ *l3-inv2*
⟨*proof*⟩

23.2.3 inv3

If a message can be analyzed from the intruder knowledge then it can be derived (using *synth*/*analz*) from the sets of implementation, non-implementation, and long-term key messages and the tags. That is, intermediate messages are not needed.

definition
  l3-inv3 :: *l3-state set*

where
l3-inv3 ≡ {*s*.
analz (*ik s*) ⊆
synth (*analz* ((*ik s* ∩ *payload*) ∪ ((*ik s*) ∩ *valid*) ∪ (*ik s* ∩ *range LtK*) ∪ *Tags*))
}

lemmas *l3-inv3I* = *l3-inv3-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv3E* = *l3-inv3-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv3D* = *l3-inv3-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv3-init* [*iff*]:
init l3 ⊆ *l3-inv3*
⟨*proof*⟩

declare *domIff* [*iff del*]

Most of the cases in this proof are simple and very similar. The proof could probably be shortened.

lemma *l3-inv3-trans* [*simp*,*intro*!]:
{*l3-inv3*} *trans l3* {> *l3-inv3*}
⟨*proof*⟩

lemma *PO-l3-inv3* [*iff*]: *reach l3* ⊆ *l3-inv3*
⟨*proof*⟩

23.2.4 inv4: the intruder knows the tags

definition
l3-inv4 :: *l3-state set*
where
l3-inv4 ≡ {*s*.
Tags ⊆ *ik s*
}

lemmas *l3-inv4I* = *l3-inv4-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv4E* [*elim*] = *l3-inv4-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv4D* = *l3-inv4-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv4-init* [*simp*,*intro*!]:
init l3 ⊆ *l3-inv4*
⟨*proof*⟩

lemma *l3-inv4-trans* [*simp*,*intro*!]:
{*l3-inv4*} *trans l3* {> *l3-inv4*}
⟨*proof*⟩

lemma *PO-l3-inv4* [*simp*,*intro*!]: *reach l3* ⊆ *l3-inv4*
⟨*proof*⟩

The remaining invariants are derived from the others. They are not protocol dependent provided the previous invariants hold.

23.2.5 inv5

The messages that the L3 DY intruder can derive from the intruder knowledge (using *synth*/*analz*), are either implementations or intermediate messages or can also be derived by the L2 intruder from the set *extr* (*l3-state.bad s*) (*ik s* ∩ *payload*) (*local.abs* (*ik s*)), that is, given the non-implementation messages and the abstractions of (implementation) messages in the intruder knowledge.

definition
  l3-inv5 :: *l3-state set*
where
  l3-inv5 ≡ {*s*.
    synth (*analz* (*ik s*)) ⊆
    dy-fake-msg (*bad s*) (*ik s* ∩ *payload*) (*abs* (*ik s*)) ∪ −*payload*
  }

lemmas *l3-inv5I* = *l3-inv5-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv5E* = *l3-inv5-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv5D* = *l3-inv5-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv5-derived*: *l3-inv2* ∩ *l3-inv3* ⊆ *l3-inv5*
⟨*proof*⟩

lemma *PO-l3-inv5* [*simp*,*intro*!]: *reach l3* ⊆ *l3-inv5*
⟨*proof*⟩

23.2.6 inv6

If the level 3 intruder can deduce a message implementing an insecure channel message, then:

- either the message is already in the intruder knowledge;
- or the message is constructed, and the payload can also be deduced by the intruder.

definition
  l3-inv6 :: *l3-state set*
where
  l3-inv6 ≡ {*s*. ∀ *A B M*.
    (*implInsec A B M* ∈ *synth* (*analz* (*ik s*)) ∧ *M* ∈ *payload*) ⟶
    (*implInsec A B M* ∈ *ik s* ∨ *M* ∈ *synth* (*analz* (*ik s*)))
  }

lemmas *l3-inv6I* = *l3-inv6-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv6E* = *l3-inv6-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv6D* = *l3-inv6-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv6-derived* [*simp*,*intro*!]:
  l3-inv3 ∩ *l3-inv4* ⊆ *l3-inv6*
⟨*proof*⟩

lemma *PO-l3-inv6* [*simp*,*intro*!]: *reach l3* ⊆ *l3-inv6*
⟨*proof*⟩

23.2.7 inv7

If the level 3 intruder can deduce a message implementing a confidential channel message, then either

- the message is already in the intruder knowledge, or
- the message is constructed, and the payload can also be deduced by the intruder.

definition
l3-inv7 :: *l3-state set*
where
l3-inv7 ≡ {*s*. ∀ *A B M*.
(*implConfid A B M* ∈ *synth* (*analz* (*ik s*)) ∧ *M* ∈ *payload*) ⟶
(*implConfid A B M* ∈ *ik s* ∨ *M* ∈ *synth* (*analz* (*ik s*)))
}

lemmas *l3-inv7I* = *l3-inv7-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv7E* = *l3-inv7-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv7D* = *l3-inv7-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv7-derived* [*simp*,*intro*!]:
l3-inv3 ∩ *l3-inv4* ⊆ *l3-inv7*
⟨*proof*⟩

lemma *PO-l3-inv7* [*simp*,*intro*!]: *reach l3* ⊆ *l3-inv7*
⟨*proof*⟩

23.2.8 inv8

If the level 3 intruder can deduce a message implementing an authentic channel message then either

- the message is already in the intruder knowledge, or
- the message is constructed, and in this case the payload can also be deduced by the intruder, and one of the agents is bad.

definition
l3-inv8 :: *l3-state set*
where
l3-inv8 ≡ {*s*. ∀ *A B M*.
(*implAuth A B M* ∈ *synth* (*analz* (*ik s*)) ∧ *M* ∈ *payload*) ⟶
(*implAuth A B M* ∈ *ik s* ∨ (*M* ∈ *synth* (*analz* (*ik s*)) ∧ (*A* ∈ *bad s* ∨ *B* ∈ *bad s*)))
}

lemmas *l3-inv8I* = *l3-inv8-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv8E* = *l3-inv8-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv8D* = *l3-inv8-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv8-derived* [*iff*]:
l3-inv2 ∩ *l3-inv3* ∩ *l3-inv4* ⊆ *l3-inv8*

⟨*proof*⟩

lemma *PO-l3-inv8* [*iff*]: *reach l3* ⊆ *l3-inv8*
⟨*proof*⟩

23.2.9 inv9

If the level 3 intruder can deduce a message implementing a secure channel message then either:

- the message is already in the intruder knowledge, or
- the message is constructed, and in this case the payload can also be deduced by the intruder, and one of the agents is bad.

definition
  l3-inv9 :: *l3-state set*
where
  l3-inv9 ≡ {*s*. ∀ *A B M*.
    (*implSecure A B M* ∈ *synth* (*analz* (*ik s*)) ∧ *M* ∈ *payload*) ⟶
    (*implSecure A B M* ∈ *ik s* ∨ (*M* ∈ *synth* (*analz* (*ik s*)) ∧ (*A* ∈ *bad s* ∨ *B* ∈ *bad s*)))
  }

lemmas *l3-inv9I* = *l3-inv9-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv9E* = *l3-inv9-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv9D* = *l3-inv9-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv9-derived* [*iff*]:
  l3-inv2 ∩ *l3-inv3* ∩ *l3-inv4* ⊆ *l3-inv9*
⟨*proof*⟩

lemma *PO-l3-inv9* [*iff*]: *reach l3* ⊆ *l3-inv9*
⟨*proof*⟩

23.3 Refinement

Mediator function.

definition
  med23s :: *l3-obs* ⇒ *l2-obs*
where
  med23s t ≡ (|
    ik = *ik t* ∩ *payload*,
    secret = *secret t*,
    progress = *progress t*,
    signalsInit = *signalsInit t*,
    signalsResp = *signalsResp t*,
    chan = *abs* (*ik t*),
    bad = *bad t*
    |)

Relation between states.

definition

R23s :: (*l2-state* ∗ *l3-state*) *set*
where
R23s ≡ {(*s*, *s*′).
s = *med23s s*′
}

lemmas *R23s-defs* = *R23s-def med23s-def*

lemma *R23sI*:
⟦ *ik s* = *ik t* ∩ *payload*; *secret s* = *secret t*; *progress s* = *progress t*;
signalsInit s = *signalsInit t*; *signalsResp s* = *signalsResp t*;
chan s = *abs* (*ik t*); *l2-state.bad s* = *bad t* ⟧
⟹ (*s*, *t*) ∈ *R23s*
⟨*proof*⟩

lemma *R23sD*:
(*s*, *t*) ∈ *R23s* ⟹
ik s = *ik t* ∩ *payload* ∧ *secret s* = *secret t* ∧ *progress s* = *progress t* ∧
signalsInit s = *signalsInit t* ∧ *signalsResp s* = *signalsResp t* ∧
chan s = *abs* (*ik t*) ∧ *l2-state.bad s* = *bad t*
⟨*proof*⟩

lemma *R23sE* [*elim*]:
⟦ (*s*, *t*) ∈ *R23s*;
⟦ *ik s* = *ik t* ∩ *payload*; *secret s* = *secret t*; *progress s* = *progress t*;
signalsInit s = *signalsInit t*; *signalsResp s* = *signalsResp t*;
chan s = *abs* (*ik t*); *l2-state.bad s* = *bad t* ⟧ ⟹ *P* ⟧
⟹ *P*
⟨*proof*⟩

lemma *can-signal-R23* [*simp*]:
(*s2*, *s3*) ∈ *R23s* ⟹
can-signal s2 A B ⟷ *can-signal s3 A B*
⟨*proof*⟩

23.3.1 Protocol events

lemma *l3-step1-refines-step1*:
{*R23s*} *l2-step1 Ra A B*, *l3-step1 Ra A B* {>*R23s*}
⟨*proof*⟩

lemma *l3-step2-refines-step2*:
{*R23s*} *l2-step2 Rb A B gnx*, *l3-step2 Rb A B gnx*{>*R23s*}
⟨*proof*⟩

lemma *l3-step3-refines-step3*:
{*R23s*} *l2-step3 Ra A B gny*, *l3-step3 Ra A B gny* {>*R23s*}
⟨*proof*⟩

lemma *l3-step4-refines-step4*:
{*R23s*} *l2-step4 Rb A B gnx*, *l3-step4 Rb A B gnx* {>*R23s*}
⟨*proof*⟩

23.3.2 Intruder events

lemma *l3-dy-payload-refines-dy-fake-msg*:
$M \in \mathit{payload} \Longrightarrow$
$\{\mathit{R23s} \cap \mathit{UNIV} \times \mathit{l3\text{-}inv5}\}\ \mathit{l2\text{-}dy\text{-}fake\text{-}msg}\ M,\ \mathit{l3\text{-}dy}\ M\ \{>\mathit{R23s}\}$
⟨*proof*⟩

lemma *l3-dy-valid-refines-dy-fake-chan*:
$[\![\ M \in \mathit{valid};\ M' \in \mathit{abs}\ \{M\}\ ]\!] \Longrightarrow$
$\{\mathit{R23s} \cap \mathit{UNIV} \times (\mathit{l3\text{-}inv5} \cap \mathit{l3\text{-}inv6} \cap \mathit{l3\text{-}inv7} \cap \mathit{l3\text{-}inv8} \cap \mathit{l3\text{-}inv9})\}$
$\mathit{l2\text{-}dy\text{-}fake\text{-}chan}\ M',\ \mathit{l3\text{-}dy}\ M$
$\{>\mathit{R23s}\}$
⟨*proof*⟩

lemma *l3-dy-valid-refines-dy-fake-chan-Un*:
$M \in \mathit{valid} \Longrightarrow$
$\{\mathit{R23s} \cap \mathit{UNIV} \times (\mathit{l3\text{-}inv5} \cap \mathit{l3\text{-}inv6} \cap \mathit{l3\text{-}inv7} \cap \mathit{l3\text{-}inv8} \cap \mathit{l3\text{-}inv9})\}$
$\bigcup M'.\ \mathit{l2\text{-}dy\text{-}fake\text{-}chan}\ M',\ \mathit{l3\text{-}dy}\ M$
$\{>\mathit{R23s}\}$
⟨*proof*⟩

lemma *l3-dy-isLtKey-refines-skip*:
$\{\mathit{R23s}\}\ \mathit{Id},\ \mathit{l3\text{-}dy}\ (\mathit{LtK}\ \mathit{ltk})\ \{>\mathit{R23s}\}$
⟨*proof*⟩

lemma *l3-dy-others-refines-skip*:
$[\![\ M \notin \mathit{range}\ \mathit{LtK};\ M \notin \mathit{valid};\ M \notin \mathit{payload}\ ]\!] \Longrightarrow$
$\{\mathit{R23s}\}\ \mathit{Id},\ \mathit{l3\text{-}dy}\ M\ \{>\mathit{R23s}\}$
⟨*proof*⟩

lemma *l3-dy-refines-dy-fake-msg-dy-fake-chan-skip*:
$\{\mathit{R23s} \cap \mathit{UNIV} \times (\mathit{l3\text{-}inv5} \cap \mathit{l3\text{-}inv6} \cap \mathit{l3\text{-}inv7} \cap \mathit{l3\text{-}inv8} \cap \mathit{l3\text{-}inv9})\}$
$\mathit{l2\text{-}dy\text{-}fake\text{-}msg}\ M \cup (\bigcup M'.\ \mathit{l2\text{-}dy\text{-}fake\text{-}chan}\ M') \cup \mathit{Id},\ \mathit{l3\text{-}dy}\ M$
$\{>\mathit{R23s}\}$
⟨*proof*⟩

23.3.3 Compromise events

lemma *l3-lkr-others-refines-lkr-others*:
$\{\mathit{R23s}\}\ \mathit{l2\text{-}lkr\text{-}others}\ A,\ \mathit{l3\text{-}lkr\text{-}others}\ A\ \{>\mathit{R23s}\}$
⟨*proof*⟩

lemma *l3-lkr-after-refines-lkr-after*:
$\{\mathit{R23s}\}\ \mathit{l2\text{-}lkr\text{-}after}\ A,\ \mathit{l3\text{-}lkr\text{-}after}\ A\ \{>\mathit{R23s}\}$
⟨*proof*⟩

lemma *l3-skr-refines-skr*:
$\{\mathit{R23s}\}\ \mathit{l2\text{-}skr}\ R\ K,\ \mathit{l3\text{-}skr}\ R\ K\ \{>\mathit{R23s}\}$
⟨*proof*⟩

lemmas *l3-trans-refines-l2-trans* =
l3-step1-refines-step1 l3-step2-refines-step2 l3-step3-refines-step3 l3-step4-refines-step4
l3-dy-refines-dy-fake-msg-dy-fake-chan-skip
l3-lkr-others-refines-lkr-others l3-lkr-after-refines-lkr-after l3-skr-refines-skr

lemma *l3-refines-init-l2* [*iff*]:
init l3 ⊆ *R23s* `` (*init l2*)
⟨*proof*⟩

lemma *l3-refines-trans-l2* [*iff*]:
{*R23s* ∩ (*UNIV* × (*l3-inv1* ∩ *l3-inv2* ∩ *l3-inv3* ∩ *l3-inv4*))} *trans l2*, *trans l3* {> *R23s*}
⟨*proof*⟩

lemma *PO-obs-consistent-R23s* [*iff*]:
obs-consistent R23s med23s l2 l3
⟨*proof*⟩

lemma *l3-refines-l2* [*iff*]:
refines
(*R23s* ∩
(*reach l2* × (*l3-inv1* ∩ *l3-inv2* ∩ *l3-inv3* ∩ *l3-inv4*)))
med23s l2 l3
⟨*proof*⟩

lemma *l3-implements-l2* [*iff*]:
implements med23s l2 l3
⟨*proof*⟩

23.4 Derived invariants

23.4.1 inv10: secrets contain no implementation material

definition
l3-inv10 :: *l3-state set*
where
l3-inv10 ≡ {*s*.
secret s ⊆ *payload*
}

lemmas *l3-inv10I* = *l3-inv10-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv10E* = *l3-inv10-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv10D* = *l3-inv10-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv10-init* [*iff*]:
init l3 ⊆ *l3-inv10*
⟨*proof*⟩

lemma *l3-inv10-trans* [*iff*]:

{*l3-inv10*} *trans l3* {> *l3-inv10*}
⟨*proof*⟩

lemma *PO-l3-inv10* [*iff*]: *reach l3* ⊆ *l3-inv10*
⟨*proof*⟩

lemma *l3-obs-inv10* [*iff*]: *oreach l3* ⊆ *l3-inv10*
⟨*proof*⟩

23.4.2 Partial secrecy

We want to prove *l3-secrecy*, i.e., *synth* (*analz* (*ik s*)) ∩ *secret s* = {}, but by refinement we only get *l3-partial-secrecy*: *dy-fake-msg* (*l3-state.bad s*) (*payloadSet* (*ik s*)) (*local.abs* (*ik s*)) ∩ *secret s* = {}. This is fine if secrets contain no implementation material. Then, by *inv5*, a message in *synth* (*analz* (*ik s*)) is in *dy-fake-msg* (*l3-state.bad s*) (*payloadSet* (*ik s*)) (*local.abs* (*ik s*)) ∪ − *payload*, and *l3-partial-secrecy* proves it is not a secret.

definition
l3-partial-secrecy :: (′*a l3-state-scheme*) *set*
where
l3-partial-secrecy ≡ {*s*.
dy-fake-msg (*bad s*) (*ik s* ∩ *payload*) (*abs* (*ik s*)) ∩ *secret s* = {}
}

lemma *l3-obs-partial-secrecy* [*iff*]: *oreach l3* ⊆ *l3-partial-secrecy*
⟨*proof*⟩

23.4.3 Secrecy

definition
l3-secrecy :: (′*a l3-state-scheme*) *set*
where
l3-secrecy ≡ *l1-secrecy*

lemma *l3-obs-inv5*: *oreach l3* ⊆ *l3-inv5*
⟨*proof*⟩

lemma *l3-obs-secrecy* [*iff*]: *oreach l3* ⊆ *l3-secrecy*
⟨*proof*⟩

lemma *l3-secrecy* [*iff*]: *reach l3* ⊆ *l3-secrecy*
⟨*proof*⟩

23.4.4 Injective agreement

abbreviation *l3-iagreement-Init* ≡ *l1-iagreement-Init*

lemma *l3-obs-iagreement-Init* [*iff*]: *oreach l3* ⊆ *l3-iagreement-Init*
⟨*proof*⟩

lemma *l3-iagreement-Init* [*iff*]: *reach l3* ⊆ *l3-iagreement-Init*
⟨*proof*⟩

abbreviation *l3-iagreement-Resp* ≡ *l1-iagreement-Resp*

lemma *l3-obs-iagreement-Resp* [*iff*]: *oreach l3* ⊆ *l3-iagreement-Resp*
⟨*proof*⟩

lemma *l3-iagreement-Resp* [*iff*]: *reach l3* ⊆ *l3-iagreement-Resp*
⟨*proof*⟩

end
end

24 Authenticated Diffie-Hellman Protocol (L3, asymmetric)

theory *dhlvl3-asymmetric*
imports *dhlvl3 Implem-asymmetric*
begin

interpretation *dhlvl3-asym*: *dhlvl3 implem-asym*
⟨*proof*⟩

end

25 Authenticated Diffie-Hellman Protocol (L3, symmetric)

theory *dhlvl3-symmetric*
imports *dhlvl3 Implem-symmetric*
begin

interpretation *dhlvl3-sym*: *dhlvl3 implem-sym*
⟨*proof*⟩

end

26 SKEME Protocol (L1)

theory *sklvl1*
imports *dhlvl1*
begin

declare *option.split-asm* [*split*]

26.1 State and Events

abbreviation *ni* :: *nat* **where** *ni* ≡ *4*
abbreviation *nr* :: *nat* **where** *nr* ≡ *5*

Proofs break if *1* is used, because *simp* replaces it with *Suc 0*. . . .

abbreviation
  xni ≡ *Var 7*

abbreviation
  xnr ≡ *Var 8*

Domain of each role (protocol-dependent).

fun *domain* :: *role-t* ⇒ *var set* **where**
  domain Init = {*xnx*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*, *xEnd*}
| *domain Resp* = {*xny*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*, *xEnd*}

consts
  guessed-frame :: *rid-t* ⇒ *frame*

Specification of the guessed frame:

1. Domain.
2. Well-typedness. The messages in the frame of a run never contain implementation material even if the agents of the run are dishonest. Therefore we consider only well-typed frames. This is notably required for the session key compromise; it also helps proving the partitionning of ik, since we know that the messages added by the protocol do not contain ltkeys in their payload and are therefore valid implementations.
3. We also ensure that the values generated by the frame owner are correctly guessed.
4. The new frame extends the previous one (from *Key-Agreement-Strong-Adversaries.dhlvl1*)

specification (*guessed-frame*)
  guessed-frame-dom-spec [*simp*]:
    dom (*guessed-frame R*) = *domain* (*role* (*guessed-runs R*))
  guessed-frame-payload-spec [*simp*, *elim*]:
    guessed-frame R x = *Some y* ⟹ *y* ∈ *payload*
  guessed-frame-Init-xnx [*simp*]:
    role (*guessed-runs R*) = *Init* ⟹ *guessed-frame R xnx* = *Some* (*NonceF* (*R$nx*))
  guessed-frame-Init-xgnx [*simp*]:
    role (*guessed-runs R*) = *Init* ⟹ *guessed-frame R xgnx* = *Some* (*Exp Gen* (*NonceF* (*R$nx*)))
  guessed-frame-Init-xni [*simp*]:

role (*guessed-runs R*) = *Init* $\Longrightarrow$ *guessed-frame R xni* = *Some* (*NonceF* (*R$ni*))
guessed-frame-Resp-xny [*simp*]:
role (*guessed-runs R*) = *Resp* $\Longrightarrow$ *guessed-frame R xny* = *Some* (*NonceF* (*R$ny*))
guessed-frame-Resp-xgny [*simp*]:
role (*guessed-runs R*) = *Resp* $\Longrightarrow$ *guessed-frame R xgny* = *Some* (*Exp Gen* (*NonceF* (*R$ny*)))
guessed-frame-Resp-xnr [*simp*]:
role (*guessed-runs R*) = *Resp* $\Longrightarrow$ *guessed-frame R xnr* = *Some* (*NonceF* (*R$nr*))
guessed-frame-xEnd [*simp*]:
guessed-frame R xEnd = *Some End*
guessed-frame-eq [*simp*]:
x ∈ {*xnx*, *xny*, *xgnx*, *xgny*, *xsk*, *xEnd*} $\Longrightarrow$ *dhlvl1.guessed-frame R x* = *guessed-frame R x*
⟨*proof*⟩

record *skl1-state* =
l1-state +
signalsInit2 :: *signal* ⇒ *nat*
signalsResp2 :: *signal* ⇒ *nat*

type-synonym *skl1-obs* = *skl1-state*

Protocol events:

- step 1: create *Ra*, *A* generates *nx* and *ni*, computes g^{nx}
- step 2: create *Rb*, *B* reads *ni* and g^{nx} insecurely, generates *ny* and *nr*, computes g^{ny}, computes $g^{nx * ny}$, emits a running signal for *Init*, *ni*, *nr*, $g^{nx * ny}$
- step 3: *A* reads g^{ny} and g^{nx} authentically, computes $g^{ny * nx}$, emits a commit signal for *Init*, *ni*, *nr*, $g^{ny * nx}$, a running signal for *Resp*, *ni*, *nr*, $g^{ny * nx}$, declares the secret $g^{ny * nx}$
- step 4: *B* reads *nr*, *ni*, g^{nx} and g^{ny} authentically, emits a commit signal for *Resp*, *ni*, *nr*, $g^{nx * ny}$, declares the secret $g^{nx * ny}$

definition
skl1-step1 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ (*'a skl1-state-scheme* ∗ *'a skl1-state-scheme*) *set*
where
skl1-step1 Ra A B ≡ {(*s*, *s'*).
— guards:
Ra ∉ *dom* (*progress s*) ∧
guessed-runs Ra = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ∧
— actions:
s' = *s*(|
progress := (*progress s*)(*Ra* ↦ {*xnx*, *xni*, *xgnx*})
|)
}

definition
skl1-step2 ::
rid-t ⇒ *agent* ⇒ *agent* ⇒ *msg* ⇒ *msg* ⇒ (*'a skl1-state-scheme* ∗ *'a skl1-state-scheme*) *set*
where

skl1-step2 Rb A B Ni gnx ≡ {(*s*, *s'*).
— guards:
guessed-runs Rb = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|) ∧
Rb ∉ *dom* (*progress s*) ∧
guessed-frame Rb xgnx = *Some gnx* ∧
guessed-frame Rb xni = *Some Ni* ∧
guessed-frame Rb xsk = *Some* (*Exp gnx* (*NonceF* (*Rb$ny*))) ∧
— actions:
s' = *s*(| *progress* := (*progress s*)(*Rb* ↦ {*xny*, *xni*, *xnr*, *xgny*, *xgnx*, *xsk*}),
signalsInit :=
if can-signal s A B then
addSignal (*signalsInit s*)
(*Running A B* ⟨*Ni*, *NonceF* (*Rb$nr*), *Exp gnx* (*NonceF* (*Rb$ny*))⟩)
else
signalsInit s,
signalsInit2 :=
if can-signal s A B then
addSignal (*signalsInit2 s*) (*Running A B* (*Exp gnx* (*NonceF* (*Rb$ny*))))
else
signalsInit2 s
|)
}

definition
skl1-step3 ::
rid-t ⇒ *agent* ⇒ *agent* ⇒ *msg* ⇒ *msg* ⇒ (*'a skl1-state-scheme* ∗ *'a skl1-state-scheme*) *set*
where
skl1-step3 Ra A B Nr gny ≡ {(*s*, *s'*).
— guards:
guessed-runs Ra = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ∧
progress s Ra = *Some* {*xnx*, *xni*, *xgnx*} ∧
guessed-frame Ra xgny = *Some gny* ∧
guessed-frame Ra xnr = *Some Nr* ∧
guessed-frame Ra xsk = *Some* (*Exp gny* (*NonceF* (*Ra$nx*))) ∧
(*can-signal s A B* ⟶ — authentication guard
(∃ *Rb*. *guessed-runs Rb* = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|) ∧
in-progressS (*progress s Rb*) {*xny*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*} ∧
guessed-frame Rb xgny = *Some gny* ∧
guessed-frame Rb xnr = *Some Nr* ∧
guessed-frame Rb xni = *Some* (*NonceF* (*Ra$ni*)) ∧
guessed-frame Rb xgnx = *Some* (*Exp Gen* (*NonceF* (*Ra$nx*))))) ∧
(*Ra* = *test* ⟶ *Exp gny* (*NonceF* (*Ra$nx*)) ∉ *synth* (*analz* (*ik s*))) ∧

— actions:
s' = *s*(| *progress* := (*progress s*)(*Ra* ↦ {*xnx*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*, *xEnd*}),
secret := {*x*. *x* = *Exp gny* (*NonceF* (*Ra$nx*)) ∧ *Ra* = *test*} ∪ *secret s*,
signalsInit :=
if can-signal s A B then
addSignal (*signalsInit s*)
(*Commit A B* ⟨*NonceF* (*Ra$ni*), *Nr*, *Exp gny* (*NonceF* (*Ra$nx*))⟩)
else
signalsInit s,
signalsInit2 :=

```
        if can-signal s A B then
          addSignal (signalsInit2 s) (Commit A B (Exp gny (NonceF (Ra$nx))))
        else
          signalsInit2 s,
      signalsResp :=
        if can-signal s A B then
          addSignal (signalsResp s)
                (Running A B ⟨NonceF (Ra$ni), Nr, Exp gny (NonceF (Ra$nx))⟩)
        else
          signalsResp s,
      signalsResp2 :=
        if can-signal s A B then
          addSignal (signalsResp2 s) (Running A B (Exp gny (NonceF (Ra$nx))))
        else
          signalsResp2 s
    ⦈
  }
```

definition

```
  skl1-step4 ::
    rid-t ⇒ agent ⇒ agent ⇒ msg ⇒ msg ⇒ ('a skl1-state-scheme ∗ 'a skl1-state-scheme) set
```

where

```
  skl1-step4 Rb A B Ni gnx ≡ {(s, s').
    — guards:
    guessed-runs Rb = ⦇role=Resp, owner=B, partner=A⦈ ∧
    progress s Rb = Some {xny, xni, xnr, xgnx, xgny, xsk} ∧
    guessed-frame Rb xgnx = Some gnx ∧
    guessed-frame Rb xni = Some Ni ∧
    (can-signal s A B ⟶ — authentication guard
      (∃ Ra. guessed-runs Ra = ⦇role=Init, owner=A, partner=B⦈ ∧
           in-progressS (progress s Ra) {xnx, xni, xnr, xgnx, xgny, xsk, xEnd} ∧
           guessed-frame Ra xgnx = Some gnx ∧
           guessed-frame Ra xni = Some Ni ∧
           guessed-frame Ra xnr = Some (NonceF (Rb$nr)) ∧
           guessed-frame Ra xgny = Some (Exp Gen (NonceF (Rb$ny))))) ∧
    (Rb = test ⟶ Exp gnx (NonceF (Rb$ny)) ∉ synth (analz (ik s))) ∧

    — actions:
    s' = s⦇ progress := (progress s)(Rb ↦ {xny, xni, xnr, xgnx, xgny, xsk, xEnd}),
          secret := {x. x = Exp gnx (NonceF (Rb$ny)) ∧ Rb = test} ∪ secret s,
          signalsResp :=
            if can-signal s A B then
              addSignal (signalsResp s)
                    (Commit A B ⟨Ni, NonceF (Rb$nr), Exp gnx (NonceF (Rb$ny))⟩)
            else
              signalsResp s,
          signalsResp2 :=
            if can-signal s A B then
              addSignal (signalsResp2 s) (Commit A B (Exp gnx (NonceF (Rb$ny))))
            else
              signalsResp2 s
        ⦈
```

}

Specification.

definition
skl1-trans :: (*'a skl1-state-scheme* ∗ *'a skl1-state-scheme*) *set* **where**
skl1-trans ≡ (⋃ *m Ra Rb A B x y*.
skl1-step1 Ra A B ∪
skl1-step2 Rb A B x y ∪
skl1-step3 Ra A B x y ∪
skl1-step4 Rb A B x y ∪
l1-learn m ∪
Id
)

definition
skl1-init :: *skl1-state set*
where
skl1-init ≡ { (|
ik = {},
secret = {},
progress = *Map.empty*,
signalsInit = λ*x*. *0*,
signalsResp = λ*x*. *0*,
signalsInit2 = λ*x*. *0*,
signalsResp2 = λ*x*. *0*
|) }

definition
skl1 :: (*skl1-state*, *skl1-obs*) *spec* **where**
skl1 ≡ (|
init = *skl1-init*,
trans = *skl1-trans*,
obs = *id*
|)

lemmas *skl1-defs* =
skl1-def skl1-init-def skl1-trans-def
l1-learn-def
skl1-step1-def skl1-step2-def skl1-step3-def skl1-step4-def

lemmas *skl1-nostep-defs* =
skl1-def skl1-init-def skl1-trans-def

lemma *skl1-obs-id* [*simp*]: *obs skl1* = *id*
⟨*proof*⟩

lemma *run-ended-trans*:
run-ended (*progress s R*) ⟹
(*s*, *s'*) ∈ *trans skl1* ⟹
run-ended (*progress s' R*)
⟨*proof*⟩

lemma *can-signal-trans*:
  can-signal s' *A B* $\Longrightarrow$
  $(s,\ s') \in$ *trans skl1* $\Longrightarrow$
  can-signal s A B
⟨*proof*⟩

26.2 Refinement: secrecy

fun *option-inter* :: *var set* ⇒ *var set option* ⇒ *var set option*
where
  option-inter S (*Some x*) = *Some* ($x \cap S$)
 |*option-inter S None* = *None*

definition *med-progress* :: *progress-t* ⇒ *progress-t*
where
  med-progress r ≡ λ *R*. *option-inter* {*xnx*, *xny*, *xgnx*, *xgny*, *xsk*, *xEnd*} (*r R*)

lemma *med-progress-upd* [*simp*]:
  med-progress (*r*(*R* ↦ *S*)) = (*med-progress r*) (*R* ↦ *S* ∩ {*xnx*, *xny*, *xgnx*, *xgny*, *xsk*, *xEnd*})
⟨*proof*⟩

lemma *med-progress-Some*:
  r x = *Some s* ⟹ *med-progress r x* = *Some* (*s* ∩ {*xnx*, *xny*, *xgnx*, *xgny*, *xsk*, *xEnd*})
⟨*proof*⟩

lemma *med-progress-None* [*simp*]: *med-progress r x* = *None* ⟷ *r x* = *None*
⟨*proof*⟩

lemma *med-progress-Some2* [*dest*]:
  med-progress r x = *Some y* ⟹ ∃ *z*. *r x* = *Some z* ∧ *y* = *z* ∩ {*xnx*, *xny*, *xgnx*, *xgny*, *xsk*, *xEnd*}
⟨*proof*⟩

lemma *med-progress-dom* [*simp*]: *dom* (*med-progress r*) = *dom r*
⟨*proof*⟩

lemma *med-progress-empty* [*simp*]: *med-progress Map.empty* = *Map.empty*
⟨*proof*⟩

Mediator function.

definition
  med11 :: *skl1-obs* ⇒ *l1-obs*
where
  med11 t ≡ (|*ik* = *ik t*,
      secret=*secret t*,
      progress = *med-progress* (*progress t*),
      signalsInit = *signalsInit2 t*,
      signalsResp = *signalsResp2 t*|)

relation between states

definition
  R11 :: (*l1-state* ∗ *skl1-state*) *set*
where

$R11 \equiv \{(s,s').$
$s = med11\ s'$
$\}$

lemmas *R11-defs* = *R11-def med11-def*

lemma *in-progress-med-progress*:
$x \in \{xnx, xny, xgnx, xgny, xsk, xEnd\}$
$\Longrightarrow$ *in-progress* (*med-progress r R*) *x* $\longleftrightarrow$ *in-progress* (*r R*) *x*
⟨*proof*⟩

lemma *in-progressS-eq*: *in-progressS S S'* $\longleftrightarrow$ ($S \neq None \wedge (\forall\ x \in S'.$ *in-progress S x*))
⟨*proof*⟩

lemma *in-progressS-med-progress*:
in-progressS (*r R*) *S*
$\Longrightarrow$ *in-progressS* (*med-progress r R*) ($S \cap \{xnx, xny, xgnx, xgny, xsk, xEnd\}$)
⟨*proof*⟩

lemma *can-signal-R11* [*simp*]:
$(s1, s2) \in R11 \Longrightarrow$
can-signal s1 A B $\longleftrightarrow$ *can-signal s2 A B*
⟨*proof*⟩

Protocol-independent events.

lemma *skl1-learn-refines-learn*:
{*R11*} *l1-learn m*, *l1-learn m* {>*R11*}
⟨*proof*⟩

Protocol events.

lemma *skl1-step1-refines-step1*:
{*R11*} *l1-step1 Ra A B*, *skl1-step1 Ra A B* {>*R11*}
⟨*proof*⟩

lemma *skl1-step2-refines-step2*:
{*R11*} *l1-step2 Rb A B gnx*, *skl1-step2 Rb A B Ni gnx* {>*R11*}
⟨*proof*⟩

lemma *skl1-step3-refines-step3*:
{*R11*} *l1-step3 Ra A B gny*, *skl1-step3 Ra A B Nr gny* {>*R11*}
⟨*proof*⟩

lemma *skl1-step4-refines-step4*:
{*R11*} *l1-step4 Rb A B gnx*, *skl1-step4 Rb A B Ni gnx* {>*R11*}
⟨*proof*⟩

Refinement proof.

lemmas *skl1-trans-refines-l1-trans* =
skl1-learn-refines-learn
skl1-step1-refines-step1 skl1-step2-refines-step2
skl1-step3-refines-step3 skl1-step4-refines-step4

lemma *skl1-refines-init-l1* [*iff*]:
init skl1 $\subseteq$ *R11* `` (*init l1*)
⟨*proof*⟩

lemma *skl1-refines-trans-l1* [*iff*]:
{*R11*} *trans l1*, *trans skl1* {> *R11*}
⟨*proof*⟩

lemma *obs-consistent-med11* [*iff*]:
obs-consistent R11 med11 l1 skl1
⟨*proof*⟩

Refinement result.

lemma *skl1-refines-l1* [*iff*]:
refines
R11
med11 l1 skl1
⟨*proof*⟩

lemma *skl1-implements-l1* [*iff*]: *implements med11 l1 skl1*
⟨*proof*⟩

26.3 Derived invariants: secrecy

lemma *skl1-obs-secrecy* [*iff*]: *oreach skl1* $\subseteq$ *s0-secrecy*
⟨*proof*⟩

lemma *skl1-secrecy* [*iff*]: *reach skl1* $\subseteq$ *s0-secrecy*
⟨*proof*⟩

26.4 Invariants: *Init* authenticates *Resp*

26.4.1 inv1

If an initiator commit signal exists for *Ra* \$ *ni*, *Nr*, $(g^{ny})^{Ra\ \$\ nx}$, then *Ra* is *Init*, has passed step 3, and has the nonce *Nr*, and (*g*^*ny*)^(*Ra*\$*nx*) as the key in its frame.

definition
skl1-inv1 :: *skl1-state set*
where
skl1-inv1 ≡ {*s*. ∀ *Ra A B gny Nr*.
signalsInit s (*Commit A B* ⟨*NonceF* (*Ra*\$*ni*), *Nr*, *Exp gny* (*NonceF* (*Ra*\$*nx*))⟩) > *0* ⟶
guessed-runs Ra = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ∧
progress s Ra = *Some* {*xnx*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*, *xEnd*} ∧
guessed-frame Ra xnr = *Some Nr* ∧
guessed-frame Ra xsk = *Some* (*Exp gny* (*NonceF* (*Ra*\$*nx*)))
}

lemmas *skl1-inv1I* = *skl1-inv1-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *skl1-inv1E* [*elim*] = *skl1-inv1-def* [*THEN setc-def-to-elim*, *rule-format*]

lemmas *skl1-inv1D* = *skl1-inv1-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *skl1-inv1-init* [*iff*]:
  init skl1 ⊆ *skl1-inv1*
⟨*proof*⟩

lemma *skl1-inv1-trans* [*iff*]:
  {*skl1-inv1*} *trans skl1* {> *skl1-inv1*}
⟨*proof*⟩

lemma *PO-skl1-inv1* [*iff*]: *reach skl1* ⊆ *skl1-inv1*
⟨*proof*⟩

26.4.2 inv2

If a *Resp* run *Rb* has passed step 2 then (if possible) an initiator running signal has been emitted.

definition
  skl1-inv2 :: *skl1-state set*
where
  skl1-inv2 ≡ {*s*. ∀ *gnx A B Rb Ni*.
    guessed-runs Rb = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|) ⟶
    in-progressS (*progress s Rb*) {*xny*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*} ⟶
    guessed-frame Rb xgnx = *Some gnx* ⟶
    guessed-frame Rb xni = *Some Ni* ⟶
    can-signal s A B ⟶
      signalsInit s (*Running A B* ⟨*Ni*, *NonceF* (*Rb*$*nr*), *Exp gnx* (*NonceF* (*Rb*$*ny*))⟩) > *0*
  }

lemmas *skl1-inv2I* = *skl1-inv2-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *skl1-inv2E* [*elim*] = *skl1-inv2-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *skl1-inv2D* = *skl1-inv2-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *skl1-inv2-init* [*iff*]:
  init skl1 ⊆ *skl1-inv2*
⟨*proof*⟩

lemma *skl1-inv2-trans* [*iff*]:
  {*skl1-inv2*} *trans skl1* {> *skl1-inv2*}
⟨*proof*⟩

lemma *PO-skl1-inv2* [*iff*]: *reach skl1* ⊆ *skl1-inv2*
⟨*proof*⟩

26.4.3 inv3 (derived)

If an *Init* run before step 3 and a *Resp* run after step 2 both know the same half-keys and nonces (more or less), then the number of *Init* running signals for the key is strictly greater than the number of *Init* commit signals. (actually, there are 0 commit and 1 running).

definition
skl1-inv3 :: *skl1-state set*
where
skl1-inv3 ≡ {*s*. ∀ *A B Rb Ra gny Nr*.
guessed-runs Rb = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|) ⟶
in-progressS (*progress s Rb*) {*xny*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*} ⟶
guessed-frame Rb xgny = *Some gny* ⟶
guessed-frame Rb xnr = *Some Nr* ⟶
guessed-frame Rb xni = *Some* (*NonceF* (*Ra*$*ni*)) ⟶
guessed-frame Rb xgnx = *Some* (*Exp Gen* (*NonceF* (*Ra*$*nx*))) ⟶
guessed-runs Ra = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ⟶
progress s Ra = *Some* {*xnx*, *xgnx*, *xni*} ⟶
can-signal s A B ⟶
signalsInit s (*Commit A B* ⟨*NonceF* (*Ra*$*ni*), *Nr*, *Exp gny* (*NonceF* (*Ra*$*nx*))⟩)
< *signalsInit s* (*Running A B* ⟨*NonceF* (*Ra*$*ni*), *Nr*, *Exp gny* (*NonceF* (*Ra*$*nx*))⟩)
}

lemmas *skl1-inv3I* = *skl1-inv3-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *skl1-inv3E* [*elim*] = *skl1-inv3-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *skl1-inv3D* = *skl1-inv3-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *skl1-inv3-derived*: *skl1-inv1* ∩ *skl1-inv2* ⊆ *skl1-inv3*
⟨*proof*⟩

26.5 Invariants: Resp authenticates Init

26.5.1 inv4

If a *Resp* commit signal exists for *Ni*, *Rb* $ *nr*, $(g^{nx})^{Rb \ \$ \ ny}$ then *Rb* is *Resp*, has finished its run, and has the nonce *Ni* and $(g^{nx})^{Rb \ \$ \ ny}$ as the key in its frame.

definition
skl1-inv4 :: *skl1-state set*
where
skl1-inv4 ≡ {*s*. ∀ *Rb A B gnx Ni*.
signalsResp s (*Commit A B* ⟨*Ni*, *NonceF* (*Rb*$*nr*), *Exp gnx* (*NonceF* (*Rb*$*ny*))⟩) > *0* ⟶
guessed-runs Rb = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|) ∧
progress s Rb = *Some* {*xny*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*, *xEnd*} ∧
guessed-frame Rb xgnx = *Some gnx* ∧
guessed-frame Rb xni = *Some Ni*
}

lemmas *skl1-inv4I* = *skl1-inv4-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *skl1-inv4E* [*elim*] = *skl1-inv4-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *skl1-inv4D* = *skl1-inv4-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *skl1-inv4-init* [*iff*]:
init skl1 ⊆ *skl1-inv4*
⟨*proof*⟩

lemma *skl1-inv4-trans* [*iff*]:
{*skl1-inv4*} *trans skl1* {> *skl1-inv4*}

⟨*proof*⟩

lemma *PO-skl1-inv4* [*iff*]: *reach skl1* ⊆ *skl1-inv4*
⟨*proof*⟩

26.5.2 inv5

If an *Init* run *Ra* has passed step3 then (if possible) a *Resp* running signal has been emitted.

definition
  skl1-inv5 :: *skl1-state set*
where
  skl1-inv5 ≡ {*s*. ∀ *gny A B Ra Nr*.
    guessed-runs Ra = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ⟶
    in-progressS (*progress s Ra*) {*xnx*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*, *xEnd*} ⟶
    guessed-frame Ra xgny = *Some gny* ⟶
    guessed-frame Ra xnr = *Some Nr* ⟶
    can-signal s A B ⟶
      signalsResp s (*Running A B* ⟨*NonceF* (*Ra*$*ni*), *Nr*, *Exp gny* (*NonceF* (*Ra*$*nx*))⟩) > *0*
  }

lemmas *skl1-inv5I* = *skl1-inv5-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *skl1-inv5E* [*elim*] = *skl1-inv5-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *skl1-inv5D* = *skl1-inv5-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *skl1-inv5-init* [*iff*]:
  init skl1 ⊆ *skl1-inv5*
⟨*proof*⟩

lemma *skl1-inv5-trans* [*iff*]:
  {*skl1-inv5*} *trans skl1* {> *skl1-inv5*}
⟨*proof*⟩

lemma *PO-skl1-inv5* [*iff*]: *reach skl1* ⊆ *skl1-inv5*
⟨*proof*⟩

26.5.3 inv6 (derived)

If a *Resp* run before step 4 and an *Init* run after step 3 both know the same half-keys (more or less), then the number of *Resp* running signals for the key is strictly greater than the number of *Resp* commit signals. (actually, there are 0 commit and 1 running).

definition
  skl1-inv6 :: *skl1-state set*
where
  skl1-inv6 ≡ {*s*. ∀ *A B Rb Ra gnx Ni*.
    guessed-runs Ra = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ⟶
    in-progressS (*progress s Ra*) {*xnx*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*, *xEnd*} ⟶
    guessed-frame Ra xgnx = *Some gnx* ⟶
    guessed-frame Ra xni = *Some Ni* ⟶
    guessed-frame Ra xgny = *Some* (*Exp Gen* (*NonceF* (*Rb*$*ny*))) ⟶
    guessed-frame Ra xnr = *Some* (*NonceF* (*Rb*$*nr*)) ⟶

guessed-runs Rb = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|) ⟶
progress s Rb = *Some* {*xny*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*} ⟶
can-signal s A B ⟶
signalsResp s (*Commit A B* ⟨*Ni*, *NonceF* (*Rb*$*nr*), *Exp gnx* (*NonceF* (*Rb*$*ny*))⟩)
< *signalsResp s* (*Running A B* ⟨*Ni*, *NonceF* (*Rb*$*nr*), *Exp gnx* (*NonceF* (*Rb*$*ny*))⟩)
}

lemmas *skl1-inv6I* = *skl1-inv6-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *skl1-inv6E* [*elim*] = *skl1-inv6-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *skl1-inv6D* = *skl1-inv6-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *skl1-inv6-derived*:
skl1-inv4 ∩ *skl1-inv5* ⊆ *skl1-inv6*
⟨*proof*⟩

26.6 Refinement: injective agreement (Init authenticates Resp)

Mediator function.

definition
med0sk1iai :: *skl1-obs* ⇒ *a0i-obs*
where
med0sk1iai t ≡ (|*a0n-state.signals* = *signalsInit t*|)

Relation between states.

definition
R0sk1iai :: (*a0i-state* ∗ *skl1-state*) *set*
where
R0sk1iai ≡ {(*s*,*s*′).
a0n-state.signals s = *signalsInit s*′
}

Protocol-independent events.

lemma *skl1-learn-refines-a0-ia-skip-i*:
{*R0sk1iai*} *Id*, *l1-learn m* {>*R0sk1iai*}
⟨*proof*⟩

Protocol events.

lemma *skl1-step1-refines-a0i-skip-i*:
{*R0sk1iai*} *Id*, *skl1-step1 Ra A B* {>*R0sk1iai*}
⟨*proof*⟩

lemma *skl1-step2-refines-a0i-running-skip-i*:
{*R0sk1iai*} *a0i-running A B* ⟨*Ni*, *NonceF* (*Rb*$*nr*),*Exp gnx* (*NonceF* (*Rb*$*ny*))⟩ ∪ *Id*,
skl1-step2 Rb A B Ni gnx {>*R0sk1iai*}
⟨*proof*⟩

lemma *skl1-step3-refines-a0i-commit-skip-i*:
{*R0sk1iai* ∩ (*UNIV* × *skl1-inv3*)}
a0i-commit A B ⟨*NonceF* (*Ra*$*ni*), *Nr*, *Exp gny* (*NonceF* (*Ra*$*nx*))⟩ ∪ *Id*,
skl1-step3 Ra A B Nr gny

{>*R0sk1iai*}
⟨*proof*⟩

lemma *skl1-step4-refines-a0i-skip-i*:
{*R0sk1iai*} *Id*, *skl1-step4 Rb A B Ni gnx* {>*R0sk1iai*}
⟨*proof*⟩

refinement proof

lemmas *skl1-trans-refines-a0i-trans-i* =
skl1-learn-refines-a0-ia-skip-i
skl1-step1-refines-a0i-skip-i skl1-step2-refines-a0i-running-skip-i
skl1-step3-refines-a0i-commit-skip-i skl1-step4-refines-a0i-skip-i

lemma *skl1-refines-init-a0i-i* [*iff*]:
init skl1 ⊆ *R0sk1iai* ‘‘ (*init a0i*)
⟨*proof*⟩

lemma *skl1-refines-trans-a0i-i* [*iff*]:
{*R0sk1iai* ∩ (*UNIV* × (*skl1-inv1* ∩ *skl1-inv2*))} *trans a0i*, *trans skl1* {> *R0sk1iai*}
⟨*proof*⟩

lemma *obs-consistent-med01iai* [*iff*]:
obs-consistent R0sk1iai med0sk1iai a0i skl1
⟨*proof*⟩

refinement result

lemma *skl1-refines-a0i-i* [*iff*]:
refines
(*R0sk1iai* ∩ (*reach a0i* × (*skl1-inv1* ∩ *skl1-inv2*)))
med0sk1iai a0i skl1
⟨*proof*⟩

lemma *skl1-implements-a0i-i* [*iff*]: *implements med0sk1iai a0i skl1*
⟨*proof*⟩

26.7 Derived invariants: injective agreement (*Init* authenticates *Resp*)

lemma *skl1-obs-iagreement-Init* [*iff*]: *oreach skl1* ⊆ *l1-iagreement-Init*
⟨*proof*⟩

lemma *skl1-iagreement-Init* [*iff*]: *reach skl1* ⊆ *l1-iagreement-Init*
⟨*proof*⟩

26.8 Refinement: injective agreement (*Resp* authenticates *Init*)

Mediator function.

definition
med0sk1iar :: *skl1-obs* ⇒ *a0i-obs*
where
med0sk1iar t ≡ ⦇*a0n-state.signals* = *signalsResp t*⦈

Relation between states.

definition
  R0sk1iar :: (*a0i-state* ∗ *skl1-state*) *set*
where
  R0sk1iar ≡ {(*s*,*s′*).
    a0n-state.signals s = *signalsResp s′*
    }

Protocol independent events.

lemma *skl1-learn-refines-a0-ia-skip-r*:
  {*R0sk1iar*} *Id*, *l1-learn m* {>*R0sk1iar*}
⟨*proof*⟩

Protocol events.

lemma *skl1-step1-refines-a0i-skip-r*:
  {*R0sk1iar*} *Id*, *skl1-step1 Ra A B* {>*R0sk1iar*}
⟨*proof*⟩

lemma *skl1-step2-refines-a0i-skip-r*:
  {*R0sk1iar*} *Id*, *skl1-step2 Rb A B Ni gnx* {>*R0sk1iar*}
⟨*proof*⟩

lemma *skl1-step3-refines-a0i-running-skip-r*:
  {*R0sk1iar*}
     a0i-running A B ⟨*NonceF* (*Ra*$*ni*), *Nr*, *Exp gny* (*NonceF* (*Ra*$*nx*))⟩ ∪ *Id*,
     skl1-step3 Ra A B Nr gny
   {>*R0sk1iar*}
⟨*proof*⟩

lemma *skl1-step4-refines-a0i-commit-skip-r*:
  {*R0sk1iar* ∩ *UNIV*×*skl1-inv6*}
     a0i-commit A B ⟨*Ni*, *NonceF* (*Rb*$*nr*), *Exp gnx* (*NonceF* (*Rb*$*ny*))⟩ ∪ *Id*,
     skl1-step4 Rb A B Ni gnx
   {>*R0sk1iar*}
⟨*proof*⟩

Refinement proof.

lemmas *skl1-trans-refines-a0i-trans-r* =
  skl1-learn-refines-a0-ia-skip-r
  skl1-step1-refines-a0i-skip-r skl1-step2-refines-a0i-skip-r
  skl1-step3-refines-a0i-running-skip-r skl1-step4-refines-a0i-commit-skip-r

lemma *skl1-refines-init-a0i-r* [*iff*]:
  init skl1 ⊆ *R0sk1iar* ‘‘ (*init a0i*)
⟨*proof*⟩

lemma *skl1-refines-trans-a0i-r* [*iff*]:
  {*R0sk1iar* ∩ (*UNIV* × (*skl1-inv4* ∩ *skl1-inv5*))} *trans a0i*, *trans skl1* {> *R0sk1iar*}
⟨*proof*⟩

lemma *obs-consistent-med0sk1iar* [*iff*]:
obs-consistent R0sk1iar med0sk1iar a0i skl1
⟨*proof*⟩

Refinement result.

lemma *skl1-refines-a0i-r* [*iff*]:
refines
(*R0sk1iar* ∩ (*reach a0i* × (*skl1-inv4* ∩ *skl1-inv5*)))
med0sk1iar a0i skl1
⟨*proof*⟩

lemma *skl1-implements-a0i-r* [*iff*]: *implements med0sk1iar a0i skl1*
⟨*proof*⟩

26.9 Derived invariants: injective agreement (*Resp* authenticates *Init*)

lemma *skl1-obs-iagreement-Resp* [*iff*]: *oreach skl1* ⊆ *l1-iagreement-Resp*
⟨*proof*⟩

lemma *skl1-iagreement-Resp* [*iff*]: *reach skl1* ⊆ *l1-iagreement-Resp*
⟨*proof*⟩

end

27 SKEME Protocol (L2)

theory *sklvl2*
imports *sklvl1 Channels*
begin

declare *domIff* [*simp*, *iff del*]

27.1 State and Events

Initial compromise.

consts
  bad-init :: *agent set*

specification (*bad-init*)
  bad-init-spec: *test-owner* ∉ *bad-init* ∧ *test-partner* ∉ *bad-init*
⟨*proof*⟩

Level 2 state.

record *l2-state* =
  skl1-state +
  chan :: *chan set*
  bad :: *agent set*

type-synonym *l2-obs* = *l2-state*

type-synonym
  l2-pred = *l2-state set*

type-synonym
  l2-trans = (*l2-state* × *l2-state*) *set*

Attacker events.

definition
  l2-dy-fake-msg :: *msg* ⇒ *l2-trans*
where
  l2-dy-fake-msg m ≡ {(*s*,*s′*).
    — guards
    m ∈ *dy-fake-msg* (*bad s*) (*ik s*) (*chan s*) ∧
    — actions
    s′ = *s*⦇*ik* := {*m*} ∪ *ik s*⦈
  }

definition
  l2-dy-fake-chan :: *chan* ⇒ *l2-trans*
where
  l2-dy-fake-chan M ≡ {(*s*,*s′*).
    — guards
    M ∈ *dy-fake-chan* (*bad s*) (*ik s*) (*chan s*)∧
    — actions
    s′ = *s*⦇*chan* := {*M*} ∪ *chan s*⦈

}

Partnering.

fun
  role-comp :: *role-t* ⇒ *role-t*
where
  role-comp Init = *Resp*
| *role-comp Resp* = *Init*

definition
  matching :: *frame* ⇒ *frame* ⇒ *bool*
where
  $matching\ sigma\ sigma' \equiv \forall\ x.\ x \in dom\ sigma \cap dom\ sigma' \longrightarrow sigma\ x = sigma'\ x$

definition
  partner-runs :: *rid-t* ⇒ *rid-t* ⇒ *bool*
where
  partner-runs R R' ≡
    role (*guessed-runs R*) = *role-comp* (*role* (*guessed-runs R'*)) ∧
    owner (*guessed-runs R*) = *partner* (*guessed-runs R'*) ∧
    owner (*guessed-runs R'*) = *partner* (*guessed-runs R*) ∧
    matching (*guessed-frame R*) (*guessed-frame R'*)

lemma *role-comp-inv* [*simp*]:
  role-comp (*role-comp x*) = *x*
⟨*proof*⟩

lemma *role-comp-inv-eq*:
  y = *role-comp x* ⟷ *x* = *role-comp y*
⟨*proof*⟩

definition
  partners :: *rid-t set*
where
  partners ≡ {*R*. *partner-runs test R*}

lemma *test-not-partner* [*simp*]:
  test ∉ *partners*
⟨*proof*⟩

lemma *matching-symmetric*:
  matching sigma sigma' ⟹ *matching sigma' sigma*
⟨*proof*⟩

lemma *partner-symmetric*:
  partner-runs R R' ⟹ *partner-runs R' R*
⟨*proof*⟩

The unicity of the parther is actually protocol dependent: it only holds if there are generated fresh nonces (which identify the runs) in the frames

lemma *partner-unique*:

$partner\text{-}runs\ R\ R'' \Longrightarrow partner\text{-}runs\ R\ R' \Longrightarrow R' = R''$
⟨*proof*⟩

lemma *partner-test*:
$R \in partners \Longrightarrow partner\text{-}runs\ R\ R' \Longrightarrow R' = test$
⟨*proof*⟩

compromising events

definition
l2-lkr-others :: *agent* ⇒ *l2-trans*
where
l2-lkr-others $A \equiv \{(s,s')$.
— guards
$A \neq test\text{-}owner \wedge$
$A \neq test\text{-}partner \wedge$
— actions
$s' = s(\!|bad := \{A\} \cup bad\ s|\!)$
}

definition
l2-lkr-actor :: *agent* ⇒ *l2-trans*
where
l2-lkr-actor $A \equiv \{(s,s')$.
— guards
$A = test\text{-}owner \wedge$
$A \neq test\text{-}partner \wedge$
— actions
$s' = s(\!|bad := \{A\} \cup bad\ s|\!)$
}

definition
l2-lkr-after :: *agent* ⇒ *l2-trans*
where
l2-lkr-after $A \equiv \{(s,s')$.
— guards
$test\text{-}ended\ s \wedge$
— actions
$s' = s(\!|bad := \{A\} \cup bad\ s|\!)$
}

definition
l2-skr :: *rid-t* ⇒ *msg* ⇒ *l2-trans*
where
l2-skr $R\ K \equiv \{(s,s')$.
— guards
$R \neq test \wedge R \notin partners \wedge$
$in\text{-}progress\ (progress\ s\ R)\ xsk \wedge$
$guessed\text{-}frame\ R\ xsk = Some\ K \wedge$
— actions
$s' = s(\!|ik := \{K\} \cup ik\ s|\!)$
}

Protocol events (with $K = H(ni, nr)$):

- step 1: create *Ra*, *A* generates *nx* and *ni*, confidentially sends *ni*, computes and insecurely sends g^{nx}
- step 2: create *Rb*, *B* receives *ni* (confidentially) and g^{nx} (insecurely), generates *ny* and *nr*, confidentially sends *nr*, insecurely sends g^{ny} and $MAC_K(g^{nx}, g^{ny}, B, A)$ computes $g^{nx\ *\ ny}$, emits a running signal for *Init*, *ni*, *nr*, $g^{nx\ *\ ny}$
- step 3: *A* receives *nr* confidentially, and g^{ny} and the MAC insecurely, sends $MAC_K(g^{ny}, g^{nx}, A, B)$ insecurely, computes $g^{ny\ *\ nx}$, emits a commit signal for *Init*, *ni*, *nr*, $g^{ny\ *\ nx}$, a running signal for *Resp*, *ni*, *nr*, $g^{ny\ *\ nx}$, declares the secret $g^{ny\ *\ nx}$
- step 4: *B* receives the MAC insecurely, emits a commit signal for *Resp*, *ni*, *nr*, $g^{nx\ *\ ny}$, declares the secret $g^{nx\ *\ ny}$

```
definition
  l2-step1 :: rid-t ⇒ agent ⇒ agent ⇒ l2-trans
where
 l2-step1 Ra A B ≡ {(s, s').
   — guards:
   Ra ∉ dom (progress s) ∧
   guessed-runs Ra = (|role=Init, owner=A, partner=B|) ∧
   — actions:
   s' = s(|
     progress := (progress s)(Ra ↦ {xnx, xni, xgnx}),
     chan := {Confid A B (NonceF (Ra$ni))} ∪
            ({Insec A B (Exp Gen (NonceF (Ra$nx)))} ∪
             (chan s))
     |)
 }
```

```
definition
 l2-step2 :: rid-t ⇒ agent ⇒ agent ⇒ msg ⇒ msg ⇒ l2-trans
where
 l2-step2 Rb A B Ni gnx ≡ {(s, s').
   — guards:
   guessed-runs Rb = (|role=Resp, owner=B, partner=A|) ∧
   Rb ∉ dom (progress s) ∧
   guessed-frame Rb xgnx = Some gnx ∧
   guessed-frame Rb xni = Some Ni ∧
   guessed-frame Rb xsk = Some (Exp gnx (NonceF (Rb$ny))) ∧
   Confid A B Ni ∈ chan s ∧
   Insec A B gnx ∈ chan s ∧
   — actions:
   s' = s(| progress := (progress s)(Rb ↦ {xny, xni, xnr, xgny, xgnx, xsk}),
          chan := {Confid B A (NonceF (Rb$nr))} ∪
                ({Insec B A
                    ⟨Exp Gen (NonceF (Rb$ny)),
                     hmac ⟨Number 0, gnx, Exp Gen (NonceF (Rb$ny)), Agent B, Agent A⟩
                          (Hash ⟨Ni, NonceF (Rb$nr)⟩)⟩ } ∪
                 (chan s)),
          signalsInit :=
```

if can-signal s A B then
addSignal (signalsInit s)
(Running A B ⟨Ni, NonceF (Rb$nr), Exp gnx (NonceF (Rb$ny))⟩)
else
signalsInit s,
signalsInit2 :=
if can-signal s A B then
addSignal (signalsInit2 s) (Running A B (Exp gnx (NonceF (Rb$ny))))
else
signalsInit2 s
⦈
}

definition
l2-step3 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *msg* ⇒ *msg* ⇒ *l2-trans*
where
l2-step3 Ra A B Nr gny ≡ {(*s*, *s'*).
— guards:
guessed-runs Ra = ⦇*role*=*Init*, *owner*=*A*, *partner*=*B*⦈ ∧
progress s Ra = *Some* {*xnx*, *xni*, *xgnx*} ∧
guessed-frame Ra xgny = *Some gny* ∧
guessed-frame Ra xnr = *Some Nr* ∧
guessed-frame Ra xsk = *Some* (*Exp gny* (*NonceF* (*Ra$nx*))) ∧
Confid B A Nr ∈ *chan s* ∧
Insec B A ⟨*gny*, *hmac* ⟨*Number 0*, *Exp Gen* (*NonceF* (*Ra$nx*)), *gny*, *Agent B*, *Agent A*⟩
(*Hash* ⟨*NonceF* (*Ra$ni*), *Nr*⟩)⟩ ∈ *chan s* ∧
— actions:
s' = *s*⦇ *progress* := (*progress s*)(*Ra* ↦ {*xnx*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*, *xEnd*}),
chan := {*Insec A B*
(*hmac* ⟨*Number 1*, *gny*, *Exp Gen* (*NonceF* (*Ra$nx*)), *Agent A*, *Agent B*⟩
(*Hash* ⟨*NonceF* (*Ra$ni*), *Nr*⟩))}
∪ *chan s*,
secret := {*x*. *x* = *Exp gny* (*NonceF* (*Ra$nx*)) ∧ *Ra* = *test*} ∪ *secret s*,
signalsInit :=
if can-signal s A B then
addSignal (signalsInit s)
(Commit A B ⟨NonceF (Ra$ni), Nr, Exp gny (NonceF (Ra$nx))⟩)
else
signalsInit s,
signalsInit2 :=
if can-signal s A B then
addSignal (signalsInit2 s) (Commit A B (Exp gny (NonceF (Ra$nx))))
else
signalsInit2 s,
signalsResp :=
if can-signal s A B then
addSignal (signalsResp s)
(Running A B ⟨NonceF (Ra$ni), Nr, Exp gny (NonceF (Ra$nx))⟩)
else
signalsResp s,
signalsResp2 :=
if can-signal s A B then

addSignal (*signalsResp2 s*) (*Running A B* (*Exp gny* (*NonceF* (*Ra*$*nx*))))
else
signalsResp2 s
⦈
}

definition
l2-step4 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *msg* ⇒ *msg* ⇒ *l2-trans*
where
l2-step4 Rb A B Ni gnx ≡ {(*s*, *s*′).
— guards:
guessed-runs Rb = ⦇*role*=*Resp*, *owner*=*B*, *partner*=*A*⦈ ∧
progress s Rb = *Some* {*xny*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*} ∧
guessed-frame Rb xgnx = *Some gnx* ∧
guessed-frame Rb xni = *Some Ni* ∧
Insec A B (*hmac* ⟨*Number 1*, *Exp Gen* (*NonceF* (*Rb*$*ny*)), *gnx*, *Agent A*, *Agent B*⟩
(*Hash* ⟨*Ni*, *NonceF* (*Rb*$*nr*)⟩)) ∈ *chan s* ∧

— actions:
s′ = *s*⦇ *progress* := (*progress s*)(*Rb* ↦ {*xny*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*, *xEnd*}),
secret := {*x*. *x* = *Exp gnx* (*NonceF* (*Rb*$*ny*)) ∧ *Rb* = *test*} ∪ *secret s*,
signalsResp :=
if can-signal s A B then
addSignal (*signalsResp s*)
(*Commit A B* ⟨*Ni*, *NonceF* (*Rb*$*nr*), *Exp gnx* (*NonceF* (*Rb*$*ny*))⟩)
else
signalsResp s,
signalsResp2 :=
if can-signal s A B then
addSignal (*signalsResp2 s*) (*Commit A B* (*Exp gnx* (*NonceF* (*Rb*$*ny*))))
else
signalsResp2 s
⦈
}

specification

definition
l2-init :: *l2-state set*
where
l2-init ≡ { ⦇
ik = {},
secret = {},
progress = *Map.empty*,
signalsInit = λ*x*. *0*,
signalsResp = λ*x*. *0*,
signalsInit2 = λ*x*. *0*,
signalsResp2 = λ*x*. *0*,
chan = {},
bad = *bad-init*
⦈}

definition

l2-trans :: *l2-trans* **where**
l2-trans ≡ (⋃ *m M X Rb Ra A B K Y*.
l2-step1 Ra A B ∪
l2-step2 Rb A B X Y ∪
l2-step3 Ra A B X Y ∪
l2-step4 Rb A B X Y ∪
l2-dy-fake-chan M ∪
l2-dy-fake-msg m ∪
l2-lkr-others A ∪
l2-lkr-after A ∪
l2-skr Ra K ∪
Id
)

definition
l2 :: (*l2-state*, *l2-obs*) *spec* **where**
l2 ≡ (|
init = *l2-init*,
trans = *l2-trans*,
obs = *id*
|)

lemmas *l2-loc-defs* =
l2-step1-def l2-step2-def l2-step3-def l2-step4-def
l2-def l2-init-def l2-trans-def
l2-dy-fake-chan-def l2-dy-fake-msg-def
l2-lkr-after-def l2-lkr-others-def l2-skr-def

lemmas *l2-defs* = *l2-loc-defs ik-dy-def*

lemmas *l2-nostep-defs* = *l2-def l2-init-def l2-trans-def*
lemmas *l2-step-defs* =
l2-step1-def l2-step2-def l2-step3-def l2-step4-def
l2-dy-fake-chan-def l2-dy-fake-msg-def l2-lkr-after-def l2-lkr-others-def l2-skr-def

lemma *l2-obs-id* [*simp*]: *obs l2* = *id*
⟨*proof*⟩

Once a run is finished, it stays finished, therefore if the test is not finished at some point then it was not finished before either.

declare *domIff* [*iff*]
lemma *l2-run-ended-trans*:
run-ended (*progress s R*) ⟹
(*s*, *s'*) ∈ *trans l2* ⟹
run-ended (*progress s' R*)
⟨*proof*⟩
declare *domIff* [*iff del*]

lemma *l2-can-signal-trans*:
can-signal s' A B ⟹
(*s*, *s'*) ∈ *trans l2* ⟹
can-signal s A B

⟨*proof*⟩

lemma *in-progressS-trans*:
  in-progressS (*progress s R*) $S \Longrightarrow (s, s') \in$ *trans l2* $\Longrightarrow$ *in-progressS* (*progress s' R*) *S*
⟨*proof*⟩

27.2 Invariants

27.2.1 inv1

If *can-signal s A B* (i.e., *A*, *B* are the test session agents and the test is not finished), then *A*, *B* are honest.

definition
  l2-inv1 :: *l2-state set*
where
  l2-inv1 ≡ {*s*. ∀ *A B*.
    can-signal s A B ⟶
    A ∉ *bad s* ∧ *B* ∉ *bad s*
  }

lemmas *l2-inv1I* = *l2-inv1-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l2-inv1E* [*elim*] = *l2-inv1-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l2-inv1D* = *l2-inv1-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l2-inv1-init* [*iff*]:
  init l2 ⊆ *l2-inv1*
⟨*proof*⟩

lemma *l2-inv1-trans* [*iff*]:
  {*l2-inv1*} *trans l2* {> *l2-inv1*}
⟨*proof*⟩

lemma *PO-l2-inv1* [*iff*]: *reach l2* ⊆ *l2-inv1*
⟨*proof*⟩

27.2.2 inv2

For a run *R* (with any role), the session key is always $something^n$ where *n* is a nonce generated by *R*.

definition
  l2-inv2 :: *l2-state set*
where
  l2-inv2 ≡ {*s*. ∀ *R*.
    in-progress (*progress s R*) *xsk* ⟶
    (∃ *X N*.
      guessed-frame R xsk = *Some* (*Exp X* (*NonceF* (*R$N*))))
  }

lemmas *l2-inv2I* = *l2-inv2-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l2-inv2E* [*elim*] = *l2-inv2-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l2-inv2D* = *l2-inv2-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l2-inv2-init* [*iff*]:
  init l2 ⊆ *l2-inv2*
⟨*proof*⟩

lemma *l2-inv2-trans* [*iff*]:
  {*l2-inv2*} *trans l2* {> *l2-inv2*}
⟨*proof*⟩

lemma *PO-l2-inv2* [*iff*]: *reach l2* ⊆ *l2-inv2*
⟨*proof*⟩

27.2.3 inv3

definition
  bad-runs s = {*R*. *owner* (*guessed-runs R*) ∈ *bad s* ∨ *partner* (*guessed-runs R*) ∈ *bad s*}

abbreviation
  generators :: *l2-state* ⇒ *msg set*
where
  generators s ≡
    — from the *insec* messages in steps 1 2
    {*x*. ∃ *N*. *x* = *Exp Gen* (*Nonce N*)} ∪
    — from the opened *confid* messages in steps 1 2
    {*x*. ∃ *R* ∈ *bad-runs s*. *x* = *NonceF* (*R*$*ni*) ∨ *x* = *NonceF* (*R*$*nr*)} ∪
    — from the *insec* messages in steps 2 3
    {*x*. ∃ *y y′ z*. *x* = *hmac* ⟨*y*, *y′*⟩ (*Hash z*)} ∪
    — from the *skr*
    {*Exp y* (*NonceF* (*R*$*N*)) | *y N R*. *R* ≠ *test* ∧ *R* ∉ *partners*}

lemma *analz-generators*: *analz* (*generators s*) = *generators s*
⟨*proof*⟩

definition
  faked-chan-msgs :: *l2-state* ⇒ *chan set*
where
  faked-chan-msgs s =
    {*Chan x A B M* | *x A B M*. *M* ∈ *synth* (*analz* (*extr* (*bad s*) (*ik s*) (*chan s*)))}

definition
  chan-generators :: *chan set*
where
  chan-generators = {*x*. ∃ *n R*. — the messages that can't be opened
    x = *Confid* (*owner* (*guessed-runs R*)) (*partner* (*guessed-runs R*)) (*NonceF* (*R*$*n*)) ∧
    (*n* = *ni* ∨ *n* = *nr*)
  }

definition
  l2-inv3 :: *l2-state set*
where
  l2-inv3 ≡ {*s*.
    extr (*bad s*) (*ik s*) (*chan s*) ⊆ *synth* (*analz* (*generators s*)) ∧

chan s ⊆ *faked-chan-msgs s* ∪ *chan-generators*
}

lemmas *l2-inv3-aux-defs* = *faked-chan-msgs-def chan-generators-def*

lemmas *l2-inv3I* = *l2-inv3-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l2-inv3E* = *l2-inv3-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l2-inv3D* = *l2-inv3-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l2-inv3-init* [*iff*]:
init l2 ⊆ *l2-inv3*
⟨*proof*⟩

lemma *l2-inv3-step1*:
{*l2-inv3*} *l2-step1 Ra A B* {> *l2-inv3*}
⟨*proof*⟩

lemma *l2-inv3-step2*:
{*l2-inv3*} *l2-step2 Rb A B Ni gnx* {> *l2-inv3*}
⟨*proof*⟩

lemma *l2-inv3-step3*:
{*l2-inv3*} *l2-step3 Ra A B Nr gny* {> *l2-inv3*}
⟨*proof*⟩

lemma *l2-inv3-step4*:
{*l2-inv3*} *l2-step4 Rb A B Ni gnx* {> *l2-inv3*}
⟨*proof*⟩

lemma *l2-inv3-dy-fake-msg*:
{*l2-inv3*} *l2-dy-fake-msg M* {> *l2-inv3*}
⟨*proof*⟩

lemma *l2-inv3-dy-fake-chan*:
{*l2-inv3*} *l2-dy-fake-chan M* {> *l2-inv3*}
⟨*proof*⟩

lemma *l2-inv3-lkr-others*:
{*l2-inv3*} *l2-lkr-others A* {> *l2-inv3*}
⟨*proof*⟩

lemma *l2-inv3-lkr-after*:
{*l2-inv3*} *l2-lkr-after A* {> *l2-inv3*}
⟨*proof*⟩

lemma *l2-inv3-skr*:
{*l2-inv3* ∩ *l2-inv2*} *l2-skr R K* {> *l2-inv3*}
⟨*proof*⟩

lemmas *l2-inv3-trans-aux* =
l2-inv3-step1 l2-inv3-step2 l2-inv3-step3 l2-inv3-step4

l2-inv3-dy-fake-msg l2-inv3-dy-fake-chan
l2-inv3-lkr-others l2-inv3-lkr-after l2-inv3-skr

lemma *l2-inv3-trans* [*iff*]:
{*l2-inv3* ∩ *l2-inv2*} *trans l2* {> *l2-inv3*}
⟨*proof*⟩

lemma *PO-l2-inv3* [*iff*]: *reach l2* ⊆ *l2-inv3*
⟨*proof*⟩

Auxiliary dest rule for inv3.

lemmas *l2-inv3D-aux* =
l2-inv3D [*THEN conjunct1*,
THEN [*2*] *subset-trans*,
THEN synth-analz-mono, *simplified*,
THEN [*2*] *rev-subsetD*, *rotated 1*, *OF IK-subset-extr*]

lemma *l2-inv3D-HashNonce1*:
s ∈ *l2-inv3* ⟹
Hash ⟨*NonceF* (*R$N*), *X*⟩ ∈ *synth* (*analz* (*extr* (*bad s*) (*ik s*) (*chan s*))) ⟹
R ∈ *bad-runs s*
⟨*proof*⟩

lemma *l2-inv3D-HashNonce2*:
s ∈ *l2-inv3* ⟹
Hash ⟨*X*, *NonceF* (*R$N*)⟩ ∈ *synth* (*analz* (*extr* (*bad s*) (*ik s*) (*chan s*))) ⟹
R ∈ *bad-runs s*
⟨*proof*⟩

27.2.4 hmac preservation lemmas

If (*s*, *s*′) ∈ *TS.trans l2* then the MACs (with secret keys) that the attacker knows in *s*′ (overapproximated by those in *parts* (*extr* (*bad s*′) (*ik s*′) (*chan s*′))) are already known in *s*, except in the case of the steps 2 and 3 of the protocol.

lemma *hmac-key-unknown*:
hmac X K ∈ *synth* (*analz H*) ⟹ *K* ∉ *synth* (*analz H*) ⟹ *hmac X K* ∈ *analz H*
⟨*proof*⟩

lemma *parts-exp* [*simp*]:*parts* {*Exp X Y*} = {*Exp X Y*}
⟨*proof*⟩

lemma *hmac-trans-1-4-skr-extr-fake*:
hmac X K ∈ *parts* (*extr* (*bad s*′) (*ik s*′) (*chan s*′)) ⟹
K ∉ *synth* (*analz* (*extr* (*bad s*) (*ik s*) (*chan s*))) ⟹ — necessary for the *dy-fake-msg* case
s ∈ *l2-inv2* ⟹ — necessary for the *skr* case
(*s*, *s*′) ∈ *l2-step1 Ra A B* ∪ *l2-step4 Rb A B Ni gnx* ∪ *l2-skr R KK* ∪
l2-dy-fake-msg M ∪ *l2-dy-fake-chan MM* ⟹
hmac X K ∈ *parts* (*extr* (*bad s*) (*ik s*) (*chan s*))
⟨*proof*⟩

lemma *hmac-trans-2*:
hmac X K ∈ *parts* (*extr* (*bad s*′) (*ik s*′) (*chan s*′)) ⟹

(s, s′) ∈ l2-step2 Rb A B Ni gnx ⟹
hmac X K ∈ parts (extr (bad s) (ik s) (chan s)) ∨
(X = ⟨Number 0, gnx, Exp Gen (NonceF (Rb$ny)), Agent B, Agent A⟩ ∧
K = Hash ⟨Ni, NonceF (Rb$nr)⟩ ∧
guessed-runs Rb = (|role=Resp, owner=B, partner=A|) ∧
progress s′ Rb = Some {xny, xni, xnr, xgnx, xgny, xsk} ∧
guessed-frame Rb xgnx = Some gnx ∧
guessed-frame Rb xni = Some Ni)
⟨proof⟩

lemma *hmac-trans-3*:
hmac X K ∈ parts (extr (bad s′) (ik s′) (chan s′)) ⟹
(s, s′) ∈ l2-step3 Ra A B Nr gny ⟹
hmac X K ∈ parts (extr (bad s) (ik s) (chan s)) ∨
(X = ⟨Number 1, gny, Exp Gen (NonceF (Ra$nx)), Agent A, Agent B⟩ ∧
K = Hash ⟨NonceF (Ra$ni), Nr⟩ ∧
guessed-runs Ra = (|role=Init, owner=A, partner=B|) ∧
progress s′ Ra = Some {xnx, xni, xnr, xgnx, xgny, xsk, xEnd} ∧
guessed-frame Ra xgny = Some gny ∧
guessed-frame Ra xnr = Some Nr)
⟨proof⟩

lemma *hmac-trans-lkr-aux*:
hmac X K ∈ parts {M. ∃ x A B. Chan x A B M ∈ chan s} ⟹
K ∉ synth (analz (extr (bad s) (ik s) (chan s))) ⟹
s ∈ l2-inv3 ⟹
hmac X K ∈ parts (extr (bad s) (ik s) (chan s))
⟨proof⟩

lemma *hmac-trans-lkr*:
hmac X K ∈ parts (extr (bad s′) (ik s′) (chan s′)) ⟹
K ∉ synth (analz (extr (bad s) (ik s) (chan s))) ⟹
s ∈ l2-inv3 ⟹
(s, s′) ∈ l2-lkr-others A ∪ l2-lkr-after A ⟹
hmac X K ∈ parts (extr (bad s) (ik s) (chan s))
⟨proof⟩

lemmas *hmac-trans* = *hmac-trans-1-4-skr-extr-fake hmac-trans-lkr hmac-trans-2 hmac-trans-3*

27.2.5 inv4 (authentication guard)

If HMAC is *parts (extr (bad s) (ik s) (chan s))* and *A*, *B* are honest then the message has indeed been sent by a responder run (etc).

definition
l2-inv4 :: l2-state set
where
l2-inv4 ≡ {s. ∀ Ra A B gny Nr.
hmac ⟨Number 0, Exp Gen (NonceF (Ra$nx)), gny, Agent B, Agent A⟩
(Hash ⟨NonceF (Ra$ni), Nr⟩) ∈ parts (extr (bad s) (ik s) (chan s)) ⟶
guessed-runs Ra = (|role=Init, owner=A, partner=B|) ⟶
A ∉ bad s ∧ B ∉ bad s ⟶

(∃ *Rb. guessed-runs Rb* = (|*role=Resp*, *owner=B*, *partner=A*|) ∧
in-progressS (*progress s Rb*) {*xny*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*} ∧
guessed-frame Rb xgny = *Some gny* ∧
guessed-frame Rb xnr = *Some Nr* ∧
guessed-frame Rb xni = *Some* (*NonceF* (*Ra$ni*)) ∧
guessed-frame Rb xgnx = *Some* (*Exp Gen* (*NonceF* (*Ra$nx*))))
}

lemmas *l2-inv4I* = *l2-inv4-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l2-inv4E* [*elim*] = *l2-inv4-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l2-inv4D* = *l2-inv4-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l2-inv4-init* [*iff*]:
init l2 ⊆ *l2-inv4*
⟨*proof*⟩

lemma *l2-inv4-trans* [*iff*]:
{*l2-inv4* ∩ *l2-inv2* ∩ *l2-inv3*} *trans l2* {> *l2-inv4*}
⟨*proof*⟩

lemma *PO-l2-inv4* [*iff*]: *reach l2* ⊆ *l2-inv4*
⟨*proof*⟩

lemma *auth-guard-step3*:
s ∈ *l2-inv4* ⟹
s ∈ *l2-inv1* ⟹
Insec B A ⟨*gny*, *hmac* ⟨*Number 0*, *Exp Gen* (*NonceF* (*Ra$nx*)), *gny*, *Agent B*, *Agent A*⟩
(*Hash* ⟨*NonceF* (*Ra$ni*), *Nr*⟩)⟩
∈ *chan s* ⟹
guessed-runs Ra = (|*role=Init*, *owner=A*, *partner=B*|) ⟹
can-signal s A B ⟹
(∃ *Rb. guessed-runs Rb* = (|*role=Resp*, *owner=B*, *partner=A*|) ∧
in-progressS (*progress s Rb*) {*xny*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*} ∧
guessed-frame Rb xgny = *Some gny* ∧
guessed-frame Rb xnr = *Some Nr* ∧
guessed-frame Rb xni = *Some* (*NonceF* (*Ra$ni*)) ∧
guessed-frame Rb xgnx = *Some* (*Exp Gen* (*NonceF* (*Ra$nx*))))
⟨*proof*⟩

27.2.6 inv5 (authentication guard)

If MAC is in *parts* (*extr* (*bad s*) (*ik s*) (*chan s*)) and *A*, *B* are honest then the message has indeed been sent by an initiator run (etc).

definition
l2-inv5 :: *l2-state set*
where
l2-inv5 ≡ {*s*. ∀ *Rb A B gnx Ni*.
hmac ⟨*Number 1*, *Exp Gen* (*NonceF* (*Rb$ny*)), *gnx*, *Agent A*, *Agent B*⟩
(*Hash* ⟨*Ni*, *NonceF* (*Rb$nr*)⟩) ∈ *parts* (*extr* (*bad s*) (*ik s*) (*chan s*)) ⟶
guessed-runs Rb = (|*role=Resp*, *owner=B*, *partner=A*|) ⟶
A ∉ *bad s* ∧ *B* ∉ *bad s* ⟶

(∃ *Ra*. *guessed-runs Ra* = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ∧
in-progressS (*progress s Ra*) {*xnx*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*, *xEnd*} ∧
guessed-frame Ra xgnx = *Some gnx* ∧
guessed-frame Ra xni = *Some Ni* ∧
guessed-frame Ra xnr = *Some* (*NonceF* (*Rb*$*nr*)) ∧
guessed-frame Ra xgny = *Some* (*Exp Gen* (*NonceF* (*Rb*$*ny*))))
}

lemmas *l2-inv5I* = *l2-inv5-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l2-inv5E* [*elim*] = *l2-inv5-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l2-inv5D* = *l2-inv5-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l2-inv5-init* [*iff*]:
init l2 ⊆ *l2-inv5*
⟨*proof*⟩

lemma *l2-inv5-trans* [*iff*]:
{*l2-inv5* ∩ *l2-inv2* ∩ *l2-inv3*} *trans l2* {> *l2-inv5*}
⟨*proof*⟩

lemma *PO-l2-inv5* [*iff*]: *reach l2* ⊆ *l2-inv5*
⟨*proof*⟩

lemma *auth-guard-step4*:
s ∈ *l2-inv5* ⟹
s ∈ *l2-inv1* ⟹
Insec A B (*hmac* ⟨*Number 1*, *Exp Gen* (*NonceF* (*Rb*$*ny*)), *gnx*, *Agent A*, *Agent B*⟩
(*Hash* ⟨*Ni*, *NonceF* (*Rb*$*nr*)⟩))
∈ *chan s* ⟹
guessed-runs Rb = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|) ⟹
can-signal s A B ⟹
(∃ *Ra*. *guessed-runs Ra* = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ∧
in-progressS (*progress s Ra*) {*xnx*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*, *xEnd*} ∧
guessed-frame Ra xgnx = *Some gnx* ∧
guessed-frame Ra xni = *Some Ni* ∧
guessed-frame Ra xnr = *Some* (*NonceF* (*Rb*$*nr*)) ∧
guessed-frame Ra xgny = *Some* (*Exp Gen* (*NonceF* (*Rb*$*ny*))))
⟨*proof*⟩

27.2.7 inv6

For an initiator, the session key is always gny^{nx}.

definition
l2-inv6 :: *l2-state set*
where
l2-inv6 ≡ {*s*. ∀ *Ra A B gny*.
guessed-runs Ra = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ⟶
in-progress (*progress s Ra*) *xsk* ⟶
guessed-frame Ra xgny = *Some gny* ⟶
guessed-frame Ra xsk = *Some* (*Exp gny* (*NonceF* (*Ra*$*nx*)))
}

lemmas *l2-inv6I* = *l2-inv6-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l2-inv6E* [*elim*] = *l2-inv6-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l2-inv6D* = *l2-inv6-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l2-inv6-init* [*iff*]:
init l2 ⊆ *l2-inv6*
⟨*proof*⟩

lemma *l2-inv6-trans* [*iff*]:
{*l2-inv6*} *trans l2* {> *l2-inv6*}
⟨*proof*⟩

lemma *PO-l2-inv6* [*iff*]: *reach l2* ⊆ *l2-inv6*
⟨*proof*⟩

27.2.8 inv6'

For a responder, the session key is always gnx^{ny}.

definition
l2-inv6' :: *l2-state set*
where
l2-inv6' ≡ {*s*. ∀ *Rb A B gnx*.
guessed-runs Rb = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|) ⟶
in-progress (*progress s Rb*) *xsk* ⟶
guessed-frame Rb xgnx = *Some gnx* ⟶
guessed-frame Rb xsk = *Some* (*Exp gnx* (*NonceF* (*Rb*$*ny*)))
}

lemmas *l2-inv6'I* = *l2-inv6'-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l2-inv6'E* [*elim*] = *l2-inv6'-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l2-inv6'D* = *l2-inv6'-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l2-inv6'-init* [*iff*]:
init l2 ⊆ *l2-inv6'*
⟨*proof*⟩

lemma *l2-inv6'-trans* [*iff*]:
{*l2-inv6'*} *trans l2* {> *l2-inv6'*}
⟨*proof*⟩

lemma *PO-l2-inv6'* [*iff*]: *reach l2* ⊆ *l2-inv6'*
⟨*proof*⟩

27.2.9 inv7: form of the secrets

definition
l2-inv7 :: *l2-state set*
where
l2-inv7 ≡ {*s*.
secret s ⊆ {*Exp* (*Exp Gen* (*NonceF* (*R*$*N*))) (*NonceF* (*R'*$*N'*)) | *N N' R R'*.
R = *test* ∧ *R'* ∈ *partners* ∧ (*N*=*nx* ∨ *N*=*ny*) ∧ (*N'*=*nx* ∨ *N'*=*ny*)}

}

lemmas *l2-inv7I* = *l2-inv7-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l2-inv7E* [*elim*] = *l2-inv7-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l2-inv7D* = *l2-inv7-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

lemma *l2-inv7-init* [*iff*]:
  init l2 ⊆ *l2-inv7*
⟨*proof*⟩

Steps 3 and 4 are the hard part.

lemma *l2-inv7-step3*:
  {*l2-inv7* ∩ *l2-inv1* ∩ *l2-inv4* ∩ *l2-inv6'*} *l2-step3 Ra A B Nr gny* {> *l2-inv7*}
⟨*proof*⟩

lemma *l2-inv7-step4*:
  {*l2-inv7* ∩ *l2-inv1* ∩ *l2-inv5* ∩ *l2-inv6* ∩ *l2-inv6'*} *l2-step4 Rb A B Ni gnx* {> *l2-inv7*}
⟨*proof*⟩

lemma *l2-inv7-trans* [*iff*]:
  {*l2-inv7* ∩ *l2-inv1* ∩ *l2-inv4* ∩ *l2-inv5* ∩ *l2-inv6* ∩ *l2-inv6'*} *trans l2* {> *l2-inv7*}
⟨*proof*⟩

lemma *PO-l2-inv7* [*iff*]: *reach l2* ⊆ *l2-inv7*
⟨*proof*⟩

auxiliary dest rule for inv7

lemma *Exp-Exp-Gen-synth*:
Exp (*Exp Gen X*) *Y* ∈ *synth H* ⟹ *Exp* (*Exp Gen X*) *Y* ∈ *H* ∨ *X* ∈ *synth H* ∨ *Y* ∈ *synth H*
⟨*proof*⟩

lemma *l2-inv7-aux*:
  s ∈ *l2-inv7* ⟹
   x ∈ *secret s* ⟹
   x ∉ *synth* (*analz* (*generators s*))
⟨*proof*⟩

27.3 Refinement

Mediator function.

definition
  med12s :: *l2-obs* ⇒ *skl1-obs*
where
  med12s t ≡ (|
    ik = *ik t*,
    secret = *secret t*,
    progress = *progress t*,
    signalsInit = *signalsInit t*,
    signalsResp = *signalsResp t*,
    signalsInit2 = *signalsInit2 t*,
    signalsResp2 = *signalsResp2 t*

⟯

Relation between states.

definition
R12s :: (*skl1-state* ∗ *l2-state*) *set*
where
R12s ≡ {(*s*,*s*′).
s = *med12s s*′
}

lemmas *R12s-defs* = *R12s-def med12s-def*

lemma *can-signal-R12* [*simp*]:
(*s1*, *s2*) ∈ *R12s* ⟹
can-signal s1 A B ⟷ *can-signal s2 A B*
⟨*proof*⟩

Protocol events.

lemma *l2-step1-refines-step1*:
{*R12s*} *skl1-step1 Ra A B*, *l2-step1 Ra A B* {>*R12s*}
⟨*proof*⟩

lemma *l2-step2-refines-step2*:
{*R12s*} *skl1-step2 Rb A B Ni gnx*, *l2-step2 Rb A B Ni gnx* {>*R12s*}
⟨*proof*⟩

for step3 and 4, we prove the level 1 guard, i.e., "the future session key is not in *synth* (*analz* (*ik s*))", using the fact that inv8 also holds for the future state in which the session key is already in *secret s*

lemma *l2-step3-refines-step3*:
{*R12s* ∩ *UNIV* × (*l2-inv1* ∩ *l2-inv3* ∩ *l2-inv4* ∩ *l2-inv6*′ ∩ *l2-inv7*)}
skl1-step3 Ra A B Nr gny, *l2-step3 Ra A B Nr gny*
{>*R12s*}
⟨*proof*⟩

lemma *l2-step4-refines-step4*:
{*R12s* ∩ *UNIV* × (*l2-inv1* ∩ *l2-inv3* ∩ *l2-inv5* ∩ *l2-inv6* ∩ *l2-inv6*′ ∩ *l2-inv7*)}
skl1-step4 Rb A B Ni gnx, *l2-step4 Rb A B Ni gnx*
{>*R12s*}
⟨*proof*⟩

attacker events

lemma *l2-dy-fake-chan-refines-skip*:
{*R12s*} *Id*, *l2-dy-fake-chan M* {>*R12s*}
⟨*proof*⟩

lemma *l2-dy-fake-msg-refines-learn*:
{*R12s* ∩ *UNIV* × (*l2-inv3* ∩ *l2-inv7*)} *l1-learn m*, *l2-dy-fake-msg m* {>*R12s*}
⟨*proof*⟩

compromising events

lemma *l2-lkr-others-refines-skip*:
{*R12s*} *Id*, *l2-lkr-others A* {>*R12s*}
⟨*proof*⟩

lemma *l2-lkr-after-refines-skip*:
{*R12s*} *Id*, *l2-lkr-after A* {>*R12s*}
⟨*proof*⟩

lemma *l2-skr-refines-learn*:
{*R12s* ∩ *UNIV* × (*l2-inv2* ∩ *l2-inv3* ∩ *l2-inv7*)} *l1-learn K*, *l2-skr R K* {>*R12s*}
⟨*proof*⟩

Refinement proof.

lemmas *l2-trans-refines-l1-trans* =
l2-dy-fake-msg-refines-learn l2-dy-fake-chan-refines-skip
l2-lkr-others-refines-skip l2-lkr-after-refines-skip l2-skr-refines-learn
l2-step1-refines-step1 l2-step2-refines-step2 l2-step3-refines-step3 l2-step4-refines-step4

lemma *l2-refines-init-l1* [*iff*]:
init l2 ⊆ *R12s* `` (*init skl1*)
⟨*proof*⟩

lemma *l2-refines-trans-l1* [*iff*]:
{*R12s* ∩ (*UNIV* × (*l2-inv1* ∩ *l2-inv2* ∩ *l2-inv3* ∩ *l2-inv4* ∩ *l2-inv5* ∩
l2-inv6 ∩ *l2-inv6'* ∩ *l2-inv7*))}
trans skl1, *trans l2*
{> *R12s*}
⟨*proof*⟩

lemma *PO-obs-consistent-R12s* [*iff*]:
obs-consistent R12s med12s skl1 l2
⟨*proof*⟩

lemma *l2-refines-l1* [*iff*]:
refines
(*R12s* ∩
(*reach skl1* × (*l2-inv1* ∩ *l2-inv2* ∩ *l2-inv3* ∩ *l2-inv4* ∩ *l2-inv5* ∩
l2-inv6 ∩ *l2-inv6'* ∩ *l2-inv7*)))
med12s skl1 l2
⟨*proof*⟩

lemma *l2-implements-l1* [*iff*]:
implements med12s skl1 l2
⟨*proof*⟩

27.4 Derived invariants

We want to prove *l2-secrecy*: *dy-fake-msg* (*bad s*) (*ik s*) (*chan s*) ∩ *secret s* = {} but by refinement we only get *l2-partial-secrecy*: *synth* (*analz* (*ik s*)) ∩ *secret s* = {} This is fine, since a message in *dy-fake-msg* (*bad s*) (*ik s*) (*chan s*) could be added to *ik s*, and *l2-partial-secrecy* would still hold for this new state.

definition
l2-partial-secrecy :: ('*a l2-state-scheme*) *set*
where
l2-partial-secrecy ≡ {*s*. *synth* (*analz* (*ik s*)) ∩ *secret s* = {}}

lemma *l2-obs-partial-secrecy* [*iff*]: *oreach l2* ⊆ *l2-partial-secrecy*
⟨*proof*⟩

lemma *l2-oreach-dy-fake-msg*:
⟦ *s* ∈ *oreach l2*; *x* ∈ *dy-fake-msg* (*bad s*) (*ik s*) (*chan s*) ⟧
⟹ *s* ⦇*ik* := *insert x* (*ik s*)⦈ ∈ *oreach l2*
⟨*proof*⟩

definition
l2-secrecy :: ('*a l2-state-scheme*) *set*
where
l2-secrecy ≡ {*s*. *dy-fake-msg* (*bad s*) (*ik s*) (*chan s*) ∩ *secret s* = {}}

lemma *l2-obs-secrecy* [*iff*]: *oreach l2* ⊆ *l2-secrecy*
⟨*proof*⟩

lemma *l2-secrecy* [*iff*]: *reach l2* ⊆ *l2-secrecy*
⟨*proof*⟩

abbreviation *l2-iagreement-Init* ≡ *l1-iagreement-Init*

lemma *l2-obs-iagreement-Init* [*iff*]: *oreach l2* ⊆ *l2-iagreement-Init*
⟨*proof*⟩

lemma *l2-iagreement-Init* [*iff*]: *reach l2* ⊆ *l2-iagreement-Init*
⟨*proof*⟩

abbreviation *l2-iagreement-Resp* ≡ *l1-iagreement-Resp*

lemma *l2-obs-iagreement-Resp* [*iff*]: *oreach l2* ⊆ *l2-iagreement-Resp*
⟨*proof*⟩

lemma *l2-iagreement-Resp* [*iff*]: *reach l2* ⊆ *l2-iagreement-Resp*
⟨*proof*⟩

end

28 SKEME Protocol (L3 locale)

theory *sklvl3*
imports *sklvl2 Implem-lemmas*
begin

28.1 State and Events

Level 3 state.

(The types have to be defined outside the locale.)

record *l3-state* = *skl1-state* +
  bad :: *agent set*

type-synonym *l3-obs* = *l3-state*

type-synonym
  l3-pred = *l3-state set*

type-synonym
  l3-trans = (*l3-state* × *l3-state*) *set*

attacker event

definition
  l3-dy :: *msg* ⇒ *l3-trans*
where
  l3-dy ≡ *ik-dy*

Compromise events.

definition
  l3-lkr-others :: *agent* ⇒ *l3-trans*
where
  l3-lkr-others A ≡ {(*s*,*s'*).
    — guards
    A ≠ *test-owner* ∧
    A ≠ *test-partner* ∧
    — actions
    s' = *s*(|*bad* := {*A*} ∪ *bad s*,
          ik := *keys-of A* ∪ *ik s*|)
  }

definition
  l3-lkr-actor :: *agent* ⇒ *l3-trans*
where
  l3-lkr-actor A ≡ {(*s*,*s'*).
    — guards
    A = *test-owner* ∧
    A ≠ *test-partner* ∧
    — actions
    s' = *s*(|*bad* := {*A*} ∪ *bad s*,
          ik := *keys-of A* ∪ *ik s*|)
  }

definition
l3-lkr-after :: *agent* ⇒ *l3-trans*
where
l3-lkr-after A ≡ {(*s*,*s*′).
— guards
test-ended s ∧
— actions
s′ = *s*(|*bad* := {*A*} ∪ *bad s*,
ik := *keys-of A* ∪ *ik s*|)
}

definition
l3-skr :: *rid-t* ⇒ *msg* ⇒ *l3-trans*
where
l3-skr R K ≡ {(*s*,*s*′).
— guards
R ≠ *test* ∧ *R* ∉ *partners* ∧
in-progress (*progress s R*) *xsk* ∧
guessed-frame R xsk = *Some K* ∧
— actions
s′ = *s*(|*ik* := {*K*} ∪ *ik s*|)
}

New locale for the level 3 protocol. This locale does not add new assumptions, it is only used to separate the level 3 protocol from the implementation locale.

locale *sklvl3* = *valid-implem*
begin

Protocol events (with $K = H(ni, nr)$):

- step 1: create *Ra*, *A* generates *nx* and *ni*, confidentially sends *ni*, computes and insecurely sends g^{nx}
- step 2: create *Rb*, *B* receives *ni* (confidentially) and g^{nx} (insecurely), generates *ny* and *nr*, confidentially sends *nr*, insecurely sends g^{ny} and $MAC_K(g^{nx}, g^{ny}, B, A)$ computes $g^{nx\ *\ ny}$, emits a running signal for *Init*, *ni*, *nr*, $g^{nx\ *\ ny}$
- step 3: *A* receives *nr* confidentially, and g^{ny} and the MAC insecurely, sends $MAC_K(g^{ny}, g^{nx}, A, B)$ insecurely, computes $g^{ny\ *\ nx}$, emits a commit signal for *Init*, *ni*, *nr*, $g^{ny\ *\ nx}$, a running signal for *Resp*, *ni*, *nr*, $g^{ny\ *\ nx}$, declares the secret $g^{ny\ *\ nx}$
- step 4: *B* receives the MAC insecurely, emits a commit signal for *Resp*, *ni*, *nr*, $g^{nx\ *\ ny}$, declares the secret $g^{nx\ *\ ny}$

definition
l3-step1 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *l3-trans*
where
l3-step1 Ra A B ≡ {(*s*, *s*′).
— guards:
Ra ∉ *dom* (*progress s*) ∧
guessed-runs Ra = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ∧

```
    — actions:
    s′ = s(|
      progress := (progress s)(Ra ↦ {xnx, xni, xgnx}),
      ik := {implConfid A B (NonceF (Ra$ni))} ∪
              ({implInsec A B (Exp Gen (NonceF (Ra$nx)))} ∪
               (ik s))
      |)
  }
```

definition

```
  l3-step2 :: rid-t ⇒ agent ⇒ agent ⇒ msg ⇒ msg ⇒ l3-trans
```

where

```
  l3-step2 Rb A B Ni gnx ≡ {(s, s′).
    — guards:
    guessed-runs Rb = (|role=Resp, owner=B, partner=A|) ∧
    Rb ∉ dom (progress s) ∧
    guessed-frame Rb xgnx = Some gnx ∧
    guessed-frame Rb xni = Some Ni ∧
    guessed-frame Rb xsk = Some (Exp gnx (NonceF (Rb$ny))) ∧
    implConfid A B Ni ∈ ik s ∧
    implInsec A B gnx ∈ ik s ∧
    — actions:
    s′ = s(| progress := (progress s)(Rb ↦ {xny, xni, xnr, xgny, xgnx, xsk}),
            ik := {implConfid B A (NonceF (Rb$nr))} ∪
                   ({implInsec B A ⟨Exp Gen (NonceF (Rb$ny)),
                                  hmac ⟨Number 0, gnx, Exp Gen (NonceF (Rb$ny)), Agent B, Agent A⟩
                                       (Hash ⟨Ni, NonceF (Rb$nr)⟩)⟩ } ∪
                    (ik s)),
            signalsInit :=
              if can-signal s A B then
                addSignal (signalsInit s)
                          (Running A B ⟨Ni, NonceF (Rb$nr), Exp gnx (NonceF (Rb$ny))⟩)
              else
                signalsInit s,
            signalsInit2 :=
              if can-signal s A B then
                addSignal (signalsInit2 s) (Running A B (Exp gnx (NonceF (Rb$ny))))
              else
                signalsInit2 s
         |)
  }
```

definition

```
  l3-step3 :: rid-t ⇒ agent ⇒ agent ⇒ msg ⇒ msg ⇒ l3-trans
```

where

```
  l3-step3 Ra A B Nr gny ≡ {(s, s′).
    — guards:
    guessed-runs Ra = (|role=Init, owner=A, partner=B|) ∧
    progress s Ra = Some {xnx, xni, xgnx} ∧
    guessed-frame Ra xgny = Some gny ∧
    guessed-frame Ra xnr = Some Nr ∧
```

guessed-frame Ra xsk = *Some* (*Exp gny* (*NonceF* (*Ra*\$*nx*))) ∧
implConfid B A Nr ∈ *ik s* ∧
implInsec B A ⟨*gny*, *hmac* ⟨*Number 0*, *Exp Gen* (*NonceF* (*Ra*\$*nx*)), *gny*, *Agent B*, *Agent A*⟩
(*Hash* ⟨*NonceF* (*Ra*\$*ni*), *Nr*⟩)⟩ ∈ *ik s* ∧
— actions:
s′ = *s*⦇ *progress* := (*progress s*)(*Ra* ↦ {*xnx*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*, *xEnd*}),
ik := {*implInsec A B* (*hmac* ⟨*Number 1*, *gny*, *Exp Gen* (*NonceF* (*Ra*\$*nx*)), *Agent A*, *Agent B*⟩
(*Hash* ⟨*NonceF* (*Ra*\$*ni*), *Nr*⟩))} ∪ *ik s*,
secret := {*x*. *x* = *Exp gny* (*NonceF* (*Ra*\$*nx*)) ∧ *Ra* = *test*} ∪ *secret s*,
signalsInit :=
if can-signal s A B then
addSignal (*signalsInit s*)
(*Commit A B* ⟨*NonceF* (*Ra*\$*ni*), *Nr*, *Exp gny* (*NonceF* (*Ra*\$*nx*))⟩)
else
signalsInit s,
signalsInit2 :=
if can-signal s A B then
addSignal (*signalsInit2 s*) (*Commit A B* (*Exp gny* (*NonceF* (*Ra*\$*nx*))))
else
signalsInit2 s,
signalsResp :=
if can-signal s A B then
addSignal (*signalsResp s*)
(*Running A B* ⟨*NonceF* (*Ra*\$*ni*), *Nr*, *Exp gny* (*NonceF* (*Ra*\$*nx*))⟩)
else
signalsResp s,
signalsResp2 :=
if can-signal s A B then
addSignal (*signalsResp2 s*) (*Running A B* (*Exp gny* (*NonceF* (*Ra*\$*nx*))))
else
signalsResp2 s
⦈
}

definition
l3-step4 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *msg* ⇒ *msg* ⇒ *l3-trans*
where
l3-step4 Rb A B Ni gnx ≡ {(*s*, *s′*).
— guards:
guessed-runs Rb = ⦇*role*=*Resp*, *owner*=*B*, *partner*=*A*⦈ ∧
progress s Rb = *Some* {*xny*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*} ∧
guessed-frame Rb xgnx = *Some gnx* ∧
guessed-frame Rb xni = *Some Ni* ∧
implInsec A B (*hmac* ⟨*Number 1*, *Exp Gen* (*NonceF* (*Rb*\$*ny*)), *gnx*, *Agent A*, *Agent B*⟩
(*Hash* ⟨*Ni*, *NonceF* (*Rb*\$*nr*)⟩)) ∈ *ik s* ∧

— actions:
s′ = *s*⦇ *progress* := (*progress s*)(*Rb* ↦ {*xny*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*, *xEnd*}),
secret := {*x*. *x* = *Exp gnx* (*NonceF* (*Rb*\$*ny*)) ∧ *Rb* = *test*} ∪ *secret s*,
signalsResp :=
if can-signal s A B then

addSignal (*signalsResp s*)
(*Commit A B* ⟨*Ni*, *NonceF* (*Rb*$*nr*), *Exp gnx* (*NonceF* (*Rb*$*ny*))⟩)
else
signalsResp s,
signalsResp2 :=
if can-signal s A B then
addSignal (*signalsResp2 s*) (*Commit A B* (*Exp gnx* (*NonceF* (*Rb*$*ny*))))
else
signalsResp2 s
⦈
}

Specification.

Initial compromise.

definition
ik-init :: *msg set*
where
ik-init ≡ {*priK C* |*C*. *C* ∈ *bad-init*} ∪ {*pubK A* | *A*. *True*} ∪
{*shrK A B* |*A B*. *A* ∈ *bad-init* ∨ *B* ∈ *bad-init*} ∪ *Tags*

lemmas about *ik-init*

lemma *parts-ik-init* [*simp*]: *parts ik-init* = *ik-init*
⟨*proof*⟩

lemma *analz-ik-init* [*simp*]: *analz ik-init* = *ik-init*
⟨*proof*⟩

lemma *abs-ik-init* [*iff*]: *abs ik-init* = {}
⟨*proof*⟩

lemma *payloadSet-ik-init* [*iff*]: *ik-init* ∩ *payload* = {}
⟨*proof*⟩

lemma *validSet-ik-init* [*iff*]: *ik-init* ∩ *valid*= {}
⟨*proof*⟩

definition
l3-init :: *l3-state set*
where
l3-init ≡ { ⦇
ik = *ik-init*,
secret = {},
progress = *Map.empty*,
signalsInit = λ*x*. *0*,
signalsResp = λ*x*. *0*,
signalsInit2 = λ*x*. *0*,
signalsResp2 = λ*x*. *0*,
bad = *bad-init*
⦈}

lemmas *l3-init-defs* = *l3-init-def ik-init-def*

definition
l3-trans :: *l3-trans*
where
  l3-trans ≡ (⋃ *M N X Rb Ra A B K*.
    l3-step1 Ra A B ∪
    l3-step2 Rb A B N X ∪
    l3-step3 Ra A B N X ∪
    l3-step4 Rb A B N X ∪
    l3-dy M ∪
    l3-lkr-others A ∪
    l3-lkr-after A ∪
    l3-skr Ra K ∪
    Id
)

definition
  l3 :: (*l3-state*, *l3-obs*) *spec* **where**
  l3 ≡ (|
    init = *l3-init*,
    trans = *l3-trans*,
    obs = *id*
  |)

lemmas *l3-loc-defs* =
  l3-step1-def l3-step2-def l3-step3-def l3-step4-def
  l3-def l3-init-defs l3-trans-def
  l3-dy-def
  l3-lkr-others-def l3-lkr-after-def l3-skr-def

lemmas *l3-defs* = *l3-loc-defs ik-dy-def*
lemmas *l3-nostep-defs* = *l3-def l3-init-def l3-trans-def*

lemma *l3-obs-id* [*simp*]: *obs l3* = *id*
⟨*proof*⟩

28.2 Invariants

28.2.1 inv1: No long-term keys as message parts

definition
  l3-inv1 :: *l3-state set*
where
  l3-inv1 ≡ {*s*.
    parts (*ik s*) ∩ *range LtK* ⊆ *ik s*
  }

lemmas *l3-inv1I* = *l3-inv1-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv1E* [*elim*] = *l3-inv1-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv1D* = *l3-inv1-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv1D′* [*dest*]: ⟦ *LtK K* ∈ *parts* (*ik s*); *s* ∈ *l3-inv1*⟧ ⟹ *LtK K* ∈ *ik s*
⟨*proof*⟩

lemma *l3-inv1-init* [*iff*]:
  init l3 ⊆ *l3-inv1*
⟨*proof*⟩

lemma *l3-inv1-trans* [*iff*]:
  {*l3-inv1*} *trans l3* {> *l3-inv1*}
⟨*proof*⟩

lemma *PO-l3-inv1* [*iff*]:
  reach l3 ⊆ *l3-inv1*
⟨*proof*⟩

28.2.2 inv2: *l3-state.bad s* indeed contains "bad" keys

definition
  l3-inv2 :: *l3-state set*
where
  l3-inv2 ≡ {*s*.
    Keys-bad (*ik s*) (*bad s*)
  }

lemmas *l3-inv2I* = *l3-inv2-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv2E* [*elim*] = *l3-inv2-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv2D* = *l3-inv2-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv2-init* [*simp*,*intro*!]:
  init l3 ⊆ *l3-inv2*
⟨*proof*⟩

lemma *l3-inv2-trans* [*simp*,*intro*!]:
  {*l3-inv2* ∩ *l3-inv1*} *trans l3* {> *l3-inv2*}
⟨*proof*⟩

lemma *PO-l3-inv2* [*iff*]: *reach l3* ⊆ *l3-inv2*
⟨*proof*⟩

28.2.3 inv3

If a message can be analyzed from the intruder knowledge then it can be derived (using *synth*/*analz*) from the sets of implementation, non-implementation, and long-term key messages and the tags. That is, intermediate messages are not needed.

definition
  l3-inv3 :: *l3-state set*
where
  l3-inv3 ≡ {*s*.
    analz (*ik s*) ⊆
    synth (*analz* ((*ik s* ∩ *payload*) ∪ ((*ik s*) ∩ *valid*) ∪ (*ik s* ∩ *range LtK*) ∪ *Tags*))
  }

lemmas *l3-inv3I* = *l3-inv3-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv3E* = *l3-inv3-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv3D* = *l3-inv3-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv3-init* [*iff*]:
  init l3 ⊆ *l3-inv3*
⟨*proof*⟩

declare *domIff* [*iff del*]

Most of the cases in this proof are simple and very similar. The proof could probably be shortened.

lemma *l3-inv3-trans* [*simp*,*intro*!]:
  {*l3-inv3*} *trans l3* {> *l3-inv3*}
⟨*proof*⟩

lemma *PO-l3-inv3* [*iff*]: *reach l3* ⊆ *l3-inv3*
⟨*proof*⟩

28.2.4 inv4: the intruder knows the tags

definition
  l3-inv4 :: *l3-state set*
where
  l3-inv4 ≡ {*s*.
    Tags ⊆ *ik s*
  }

lemmas *l3-inv4I* = *l3-inv4-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv4E* [*elim*] = *l3-inv4-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv4D* = *l3-inv4-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv4-init* [*simp*,*intro*!]:
  init l3 ⊆ *l3-inv4*
⟨*proof*⟩

lemma *l3-inv4-trans* [*simp*,*intro*!]:
  {*l3-inv4*} *trans l3* {> *l3-inv4*}
⟨*proof*⟩

lemma *PO-l3-inv4* [*simp*,*intro*!]: *reach l3* ⊆ *l3-inv4*
⟨*proof*⟩

The remaining invariants are derived from the others. They are not protocol dependent provided the previous invariants hold.

28.2.5 inv5

The messages that the L3 DY intruder can derive from the intruder knowledge (using *synth*/*analz*), are either implementations or intermediate messages or can also be derived by the L2 intruder from the set *extr* (*l3-state.bad s*) (*ik s* ∩ *payload*) (*local.abs* (*ik s*)), that is, given the non-implementation messages and the abstractions of (implementation) messages in the intruder knowledge.

definition
l3-inv5 :: *l3-state set*
where
l3-inv5 ≡ {*s*.
synth (*analz* (*ik s*)) ⊆
dy-fake-msg (*bad s*) (*ik s* ∩ *payload*) (*abs* (*ik s*)) ∪ −*payload*
}

lemmas *l3-inv5I* = *l3-inv5-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv5E* = *l3-inv5-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv5D* = *l3-inv5-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv5-derived*: *l3-inv2* ∩ *l3-inv3* ⊆ *l3-inv5*
⟨*proof*⟩

lemma *PO-l3-inv5* [*simp*,*intro*!]: *reach l3* ⊆ *l3-inv5*
⟨*proof*⟩

28.2.6 inv6

If the level 3 intruder can deduce a message implementing an insecure channel message, then either:

- the message is already in the intruder knowledge, or
- the message is constructed, and the payload can also be deduced by the intruder.

definition
l3-inv6 :: *l3-state set*
where
l3-inv6 ≡ {*s*. ∀ *A B M*.
(*implInsec A B M* ∈ *synth* (*analz* (*ik s*)) ∧ *M* ∈ *payload*) ⟶
(*implInsec A B M* ∈ *ik s* ∨ *M* ∈ *synth* (*analz* (*ik s*)))
}

lemmas *l3-inv6I* = *l3-inv6-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv6E* = *l3-inv6-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv6D* = *l3-inv6-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv6-derived* [*simp*,*intro*!]:
l3-inv3 ∩ *l3-inv4* ⊆ *l3-inv6*
⟨*proof*⟩

lemma *PO-l3-inv6* [*simp*,*intro*!]: *reach l3* ⊆ *l3-inv6*
⟨*proof*⟩

28.2.7 inv7

If the level 3 intruder can deduce a message implementing a confidential channel message, then either:

- the message is already in the intruder knowledge, or
- the message is constructed, and the payload can also be deduced by the intruder.

definition
l3-inv7 :: *l3-state set*
where
l3-inv7 ≡ {*s*. ∀ *A B M*.
(*implConfid A B M* ∈ *synth* (*analz* (*ik s*)) ∧ *M* ∈ *payload*) ⟶
(*implConfid A B M* ∈ *ik s* ∨ *M* ∈ *synth* (*analz* (*ik s*)))
}

lemmas *l3-inv7I* = *l3-inv7-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv7E* = *l3-inv7-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv7D* = *l3-inv7-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv7-derived* [*simp*,*intro*!]:
l3-inv3 ∩ *l3-inv4* ⊆ *l3-inv7*
⟨*proof*⟩

lemma *PO-l3-inv7* [*simp*,*intro*!]: *reach l3* ⊆ *l3-inv7*
⟨*proof*⟩

28.2.8 inv8

If the level 3 intruder can deduce a message implementing an authentic channel message then either:

- the message is already in the intruder knowledge, or
- the message is constructed, and in this case the payload can also be deduced by the intruder, and one of the agents is bad.

definition
l3-inv8 :: *l3-state set*
where
l3-inv8 ≡ {*s*. ∀ *A B M*.
(*implAuth A B M* ∈ *synth* (*analz* (*ik s*)) ∧ *M* ∈ *payload*) ⟶
(*implAuth A B M* ∈ *ik s* ∨ (*M* ∈ *synth* (*analz* (*ik s*)) ∧ (*A* ∈ *bad s* ∨ *B* ∈ *bad s*)))
}

lemmas *l3-inv8I* = *l3-inv8-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv8E* = *l3-inv8-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv8D* = *l3-inv8-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv8-derived* [*iff*]:
l3-inv2 ∩ *l3-inv3* ∩ *l3-inv4* ⊆ *l3-inv8*
⟨*proof*⟩

lemma *PO-l3-inv8* [*iff*]: *reach l3* ⊆ *l3-inv8*
⟨*proof*⟩

28.2.9 inv9

If the level 3 intruder can deduce a message implementing a secure channel message then either:

- the message is already in the intruder knowledge, or
- the message is constructed, and in this case the payload can also be deduced by the intruder, and one of the agents is bad.

definition
  l3-inv9 :: *l3-state set*
where
  l3-inv9 ≡ {*s*. ∀ *A B M*.
    (*implSecure A B M* ∈ *synth* (*analz* (*ik s*)) ∧ *M* ∈ *payload*) ⟶
    (*implSecure A B M* ∈ *ik s* ∨ (*M* ∈ *synth* (*analz* (*ik s*)) ∧ (*A* ∈ *bad s* ∨ *B* ∈ *bad s*)))
  }

lemmas *l3-inv9I* = *l3-inv9-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv9E* = *l3-inv9-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv9D* = *l3-inv9-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv9-derived* [*iff*]:
  l3-inv2 ∩ *l3-inv3* ∩ *l3-inv4* ⊆ *l3-inv9*
⟨*proof*⟩

lemma *PO-l3-inv9* [*iff*]: *reach l3* ⊆ *l3-inv9*
⟨*proof*⟩

28.3 Refinement

Mediator function.

definition
  med23s :: *l3-obs* ⇒ *l2-obs*
where
  med23s t ≡ ⦇
    ik = *ik t* ∩ *payload*,
    secret = *secret t*,
    progress = *progress t*,
    signalsInit = *signalsInit t*,
    signalsResp = *signalsResp t*,
    signalsInit2 = *signalsInit2 t*,
    signalsResp2 = *signalsResp2 t*,
    chan = *abs* (*ik t*),
    bad = *bad t*
    ⦈

Relation between states.

definition
  R23s :: (*l2-state* ∗ *l3-state*) *set*
where
  R23s ≡ {(*s*, *s'*).

$s = med23s\ s'$
}

lemmas *R23s-defs* = *R23s-def med23s-def*

lemma *R23sI*:
⟦ *ik s* = *ik t* ∩ *payload*; *secret s* = *secret t*; *progress s* = *progress t*;
signalsInit s = *signalsInit t*; *signalsResp s* = *signalsResp t*;
signalsInit2 s = *signalsInit2 t*; *signalsResp2 s* = *signalsResp2 t*;
chan s = *abs* (*ik t*); *l2-state.bad s* = *bad t* ⟧
⟹ (*s*, *t*) ∈ *R23s*
⟨*proof*⟩

lemma *R23sD*:
(*s*, *t*) ∈ *R23s* ⟹
ik s = *ik t* ∩ *payload* ∧ *secret s* = *secret t* ∧ *progress s* = *progress t* ∧
signalsInit s = *signalsInit t* ∧ *signalsResp s* = *signalsResp t* ∧
signalsInit2 s = *signalsInit2 t* ∧ *signalsResp2 s* = *signalsResp2 t* ∧
chan s = *abs* (*ik t*) ∧ *l2-state.bad s* = *bad t*
⟨*proof*⟩

lemma *R23sE* [*elim*]:
⟦ (*s*, *t*) ∈ *R23s*;
⟦ *ik s* = *ik t* ∩ *payload*; *secret s* = *secret t*; *progress s* = *progress t*;
signalsInit s = *signalsInit t*; *signalsResp s* = *signalsResp t*;
signalsInit2 s = *signalsInit2 t*; *signalsResp2 s* = *signalsResp2 t*;
chan s = *abs* (*ik t*); *l2-state.bad s* = *bad t* ⟧ ⟹ *P* ⟧
⟹ *P*
⟨*proof*⟩

lemma *can-signal-R23* [*simp*]:
(*s2*, *s3*) ∈ *R23s* ⟹
can-signal s2 A B ⟷ *can-signal s3 A B*
⟨*proof*⟩

28.3.1 Protocol events

lemma *l3-step1-refines-step1*:
{*R23s*} *l2-step1 Ra A B*, *l3-step1 Ra A B* {>*R23s*}
⟨*proof*⟩

lemma *l3-step2-refines-step2*:
{*R23s*} *l2-step2 Rb A B Ni gnx*, *l3-step2 Rb A B Ni gnx*{>*R23s*}
⟨*proof*⟩

lemma *l3-step3-refines-step3*:
{*R23s*} *l2-step3 Ra A B Nr gny*, *l3-step3 Ra A B Nr gny* {>*R23s*}
⟨*proof*⟩

lemma *l3-step4-refines-step4*:
{*R23s*} *l2-step4 Rb A B Ni gnx*, *l3-step4 Rb A B Ni gnx* {>*R23s*}
⟨*proof*⟩

28.3.2 Intruder events

lemma *l3-dy-payload-refines-dy-fake-msg*:
$M \in payload \Longrightarrow$
$\{R23s \cap UNIV \times l3\text{-}inv5\}$ $l2\text{-}dy\text{-}fake\text{-}msg\ M$, $l3\text{-}dy\ M$ $\{>R23s\}$
⟨*proof*⟩

lemma *l3-dy-valid-refines-dy-fake-chan*:
$[\![\ M \in valid;\ M' \in abs\ \{M\}\ ]\!] \Longrightarrow$
$\{R23s \cap UNIV \times (l3\text{-}inv5 \cap l3\text{-}inv6 \cap l3\text{-}inv7 \cap l3\text{-}inv8 \cap l3\text{-}inv9)\}$
$l2\text{-}dy\text{-}fake\text{-}chan\ M'$, $l3\text{-}dy\ M$
$\{>R23s\}$
⟨*proof*⟩

lemma *l3-dy-valid-refines-dy-fake-chan-Un*:
$M \in valid \Longrightarrow$
$\{R23s \cap UNIV \times (l3\text{-}inv5 \cap l3\text{-}inv6 \cap l3\text{-}inv7 \cap l3\text{-}inv8 \cap l3\text{-}inv9)\}$
$\bigcup M'.\ l2\text{-}dy\text{-}fake\text{-}chan\ M'$, $l3\text{-}dy\ M$
$\{>R23s\}$
⟨*proof*⟩

lemma *l3-dy-isLtKey-refines-skip*:
$\{R23s\}$ Id, $l3\text{-}dy\ (LtK\ ltk)$ $\{>R23s\}$
⟨*proof*⟩

lemma *l3-dy-others-refines-skip*:
$[\![\ M \notin range\ LtK;\ M \notin valid;\ M \notin payload\ ]\!] \Longrightarrow$
$\{R23s\}$ Id, $l3\text{-}dy\ M$ $\{>R23s\}$
⟨*proof*⟩

lemma *l3-dy-refines-dy-fake-msg-dy-fake-chan-skip*:
$\{R23s \cap UNIV \times (l3\text{-}inv5 \cap l3\text{-}inv6 \cap l3\text{-}inv7 \cap l3\text{-}inv8 \cap l3\text{-}inv9)\}$
$l2\text{-}dy\text{-}fake\text{-}msg\ M \cup (\bigcup M'.\ l2\text{-}dy\text{-}fake\text{-}chan\ M') \cup Id$, $l3\text{-}dy\ M$
$\{>R23s\}$
⟨*proof*⟩

28.3.3 Compromise events

lemma *l3-lkr-others-refines-lkr-others*:
$\{R23s\}$ $l2\text{-}lkr\text{-}others\ A$, $l3\text{-}lkr\text{-}others\ A$ $\{>R23s\}$
⟨*proof*⟩

lemma *l3-lkr-after-refines-lkr-after*:
$\{R23s\}$ $l2\text{-}lkr\text{-}after\ A$, $l3\text{-}lkr\text{-}after\ A$ $\{>R23s\}$
⟨*proof*⟩

lemma *l3-skr-refines-skr*:
$\{R23s\}$ $l2\text{-}skr\ R\ K$, $l3\text{-}skr\ R\ K$ $\{>R23s\}$
⟨*proof*⟩

lemmas *l3-trans-refines-l2-trans* =
l3-step1-refines-step1 l3-step2-refines-step2 l3-step3-refines-step3 l3-step4-refines-step4
l3-dy-refines-dy-fake-msg-dy-fake-chan-skip
l3-lkr-others-refines-lkr-others l3-lkr-after-refines-lkr-after l3-skr-refines-skr

lemma *l3-refines-init-l2* [*iff*]:
init l3 ⊆ *R23s* ‘‘ (*init l2*)
⟨*proof*⟩

lemma *l3-refines-trans-l2* [*iff*]:
{*R23s* ∩ (*UNIV* × (*l3-inv1* ∩ *l3-inv2* ∩ *l3-inv3* ∩ *l3-inv4*))} *trans l2*, *trans l3* {> *R23s*}
⟨*proof*⟩

lemma *PO-obs-consistent-R23s* [*iff*]:
obs-consistent R23s med23s l2 l3
⟨*proof*⟩

lemma *l3-refines-l2* [*iff*]:
refines
(*R23s* ∩
(*reach l2* × (*l3-inv1* ∩ *l3-inv2* ∩ *l3-inv3* ∩ *l3-inv4*)))
med23s l2 l3
⟨*proof*⟩

lemma *l3-implements-l2* [*iff*]:
implements med23s l2 l3
⟨*proof*⟩

28.4 Derived invariants

28.4.1 inv10: secrets contain no implementation material

definition
l3-inv10 :: *l3-state set*
where
l3-inv10 ≡ {*s*.
secret s ⊆ *payload*
}

lemmas *l3-inv10I* = *l3-inv10-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *l3-inv10E* = *l3-inv10-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *l3-inv10D* = *l3-inv10-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *l3-inv10-init* [*iff*]:
init l3 ⊆ *l3-inv10*
⟨*proof*⟩

lemma *l3-inv10-trans* [*iff*]:

{*l3-inv10*} *trans l3* {> *l3-inv10*}
⟨*proof*⟩

lemma *PO-l3-inv10* [*iff*]: *reach l3* ⊆ *l3-inv10*
⟨*proof*⟩

lemma *l3-obs-inv10* [*iff*]: *oreach l3* ⊆ *l3-inv10*
⟨*proof*⟩

28.4.2 Partial secrecy

We want to prove *l3-secrecy*, ie *synth* (*analz* (*ik s*)) ∩ *secret s* = {}, but by refinement we only get *l3-partial-secrecy*: *dy-fake-msg* (*l3-state.bad s*) (*payloadSet* (*ik s*)) (*local.abs* (*ik s*)) ∩ *secret s* = {}. This is fine if secrets contain no implementation material. Then, by *inv5*, a message in *synth* (*analz* (*ik s*)) is in *dy-fake-msg* (*l3-state.bad s*) (*payloadSet* (*ik s*)) (*local.abs* (*ik s*)) ∪ − *payload*, and *l3-partial-secrecy* proves it is not a secret.

definition
l3-partial-secrecy :: (*'a l3-state-scheme*) *set*
where
l3-partial-secrecy ≡ {*s*.
dy-fake-msg (*bad s*) (*ik s* ∩ *payload*) (*abs* (*ik s*)) ∩ *secret s* = {}
}

lemma *l3-obs-partial-secrecy* [*iff*]: *oreach l3* ⊆ *l3-partial-secrecy*
⟨*proof*⟩

28.4.3 Secrecy

definition
l3-secrecy :: (*'a l3-state-scheme*) *set*
where
l3-secrecy ≡ *l1-secrecy*

lemma *l3-obs-inv5*: *oreach l3* ⊆ *l3-inv5*
⟨*proof*⟩

lemma *l3-obs-secrecy* [*iff*]: *oreach l3* ⊆ *l3-secrecy*
⟨*proof*⟩

lemma *l3-secrecy* [*iff*]: *reach l3* ⊆ *l3-secrecy*
⟨*proof*⟩

28.4.4 Injective agreement

abbreviation *l3-iagreement-Init* ≡ *l1-iagreement-Init*

lemma *l3-obs-iagreement-Init* [*iff*]: *oreach l3* ⊆ *l3-iagreement-Init*
⟨*proof*⟩

lemma *l3-iagreement-Init* [*iff*]: *reach l3* ⊆ *l3-iagreement-Init*
⟨*proof*⟩

abbreviation *l3-iagreement-Resp* ≡ *l1-iagreement-Resp*

lemma *l3-obs-iagreement-Resp* [*iff*]: *oreach l3* ⊆ *l3-iagreement-Resp*
⟨*proof*⟩

lemma *l3-iagreement-Resp* [*iff*]: *reach l3* ⊆ *l3-iagreement-Resp*
⟨*proof*⟩

end
end

29 SKEME Protocol (L3 with asymmetric implementation)

theory *sklvl3-asymmetric*
imports *sklvl3 Implem-asymmetric*
begin

interpretation *sklvl3-asym*: *sklvl3 implem-asym*
⟨*proof*⟩

end

30 SKEME Protocol (L3 with symmetric implementation)

theory *sklvl3-symmetric*
imports *sklvl3 Implem-symmetric*
begin

interpretation *sklvl3-sym*: *sklvl3 implem-sym*
⟨*proof*⟩

end