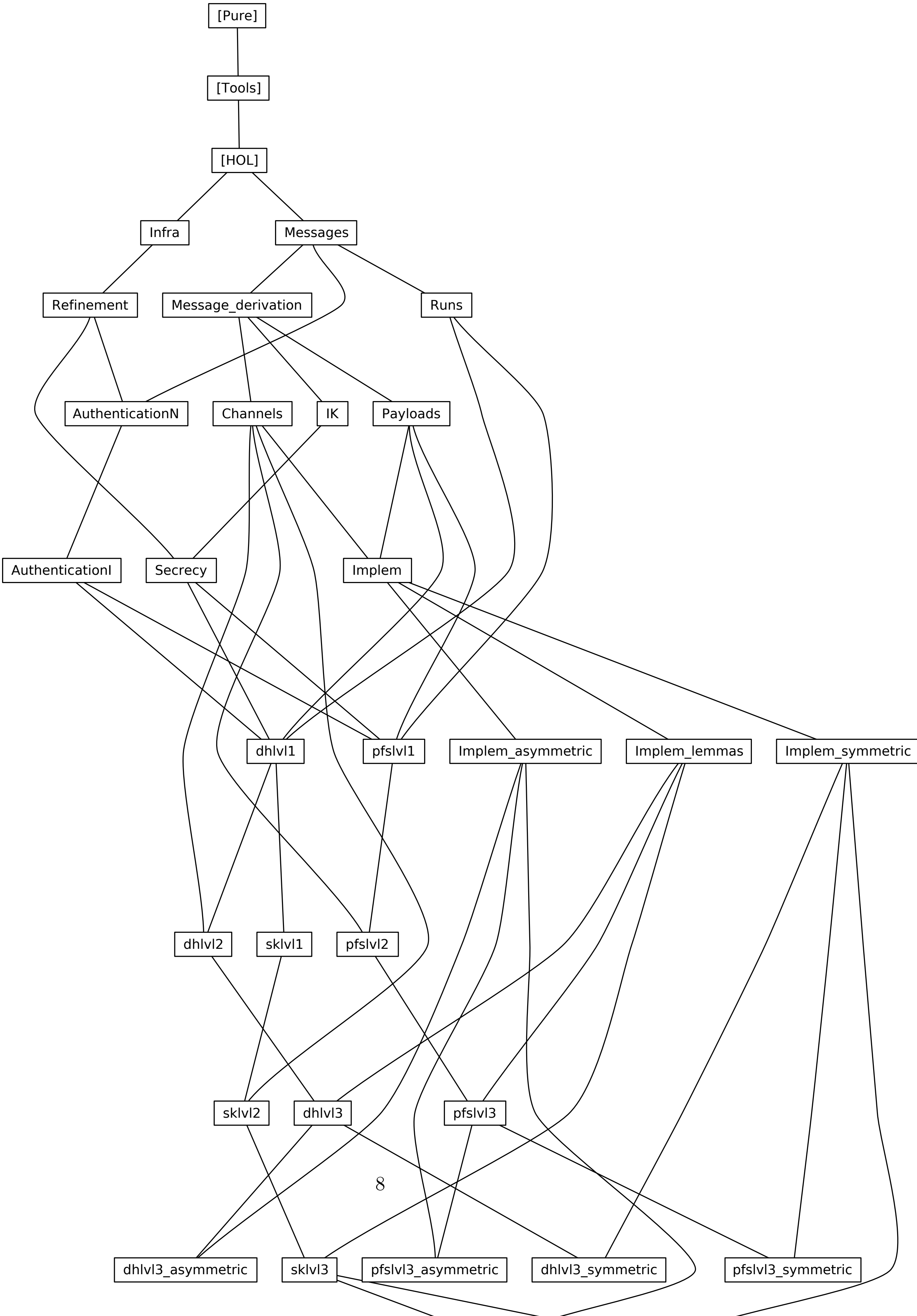
Refining Authenticated Key Agreement with Strong Adversaries

Joseph Lallemand, Christoph Sprenger, and David Basin

March 17, 2025

Contents


[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*:
  $f\ x = Some\ y \Longrightarrow x \in dom\ f$
by (*simp add*: *domIff*)

lemma *map-definedness-contra*:
  $[\![\ f\ x = Some\ y;\ z \notin dom\ f\ ]\!] \Longrightarrow x \neq z$
by (*auto simp add*: *domIff*)

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

1.3.2 Set

lemma *vimage-image-subset*: $A \subseteq f-`(f`A)$
by (*auto simp add*: *image-def vimage-def*)

1.3.3 Relations

lemma *Image-compose* [*simp*]:
 (*R1 O R2*)``*A* = *R2*``(*R1*``*A*)
by (*auto*)

1.3.4 Lists

lemma *map-comp*: *map* (*g o f*) = *map g o map f*
by (*simp*)

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

lemma *take-prefix*: ⟦ *take n l* = *xs* ⟧ ⟹ ∃ *xs'*. *l* = *xs* @ *xs'*
by (*induct l arbitrary*: *n xs*, *auto*)
 (*rename-tac n*, *case-tac n*, *auto*)

1.3.5 Finite sets

Cardinality.

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

lemma *finite-positive-cardI* [*intro*!]:
 ⟦ *A* ≠ {}; *finite A* ⟧ ⟹ *0* < *card A*
by (*auto*)

lemma *finite-positive-cardD* [*dest*!]:
 ⟦ *0* < *card A*; *finite A* ⟧ ⟹ *A* ≠ {}
by (*auto*)

lemma *finite-zero-cardI* [*intro*!]:
 ⟦ *A* = {}; *finite A* ⟧ ⟹ *card A* = *0*
by (*auto*)

lemma *finite-zero-cardD* [*dest*!]:
 ⟦ *card A* = *0*; *finite A* ⟧ ⟹ *A* = {}
by (*auto*)

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* × *'s*) *set*

The inductive set of reachable states.

inductive-set
  reach :: (*'s*, *'a*) *TS-scheme* ⇒ *'s set*
  for *T* :: (*'s*, *'a*) *TS-scheme*
where
  r-init [*intro*]: *s* ∈ *init T* ⟹ *s* ∈ *reach T*
| *r-trans* [*intro*]: ⟦ (*s*, *t*) ∈ *trans T*; *s* ∈ *reach T* ⟧ ⟹ *t* ∈ *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* ⇒ (*'s list*) *set*
  for *T* :: (*'s*, *'a*) *TS-scheme*
where
  b-empty [*iff*]: [] ∈ *beh T*
| *b-init* [*intro*]: *s* ∈ *init T* ⟹ [*s*] ∈ *beh T*
| *b-trans* [*intro*]: ⟦ *s* # *b* ∈ *beh T*; (*s*, *t*) ∈ *trans T* ⟧ ⟹ *t* # *s* # *b* ∈ *beh T*

inductive-cases *beh-non-empty*: *s* # *b* ∈ *beh T*

Behaviours are prefix closed.

lemma *beh-immediate-prefix-closed*:
  s # *b* ∈ *beh T* ⟹ *b* ∈ *beh T*
by (*erule beh-non-empty*, *auto*)

lemma *beh-prefix-closed*:
  c @ *b* ∈ *beh T* ⟹ *b* ∈ *beh T*
by (*induct c*, *auto dest*!: *beh-immediate-prefix-closed*)

States in behaviours are exactly reachable.

lemma *beh-in-reach* [*rule-format*]:
  b ∈ *beh T* ⟹ (∀ *s* ∈ *set b*. *s* ∈ *reach T*)
by (*erule beh.induct*) (*auto*)

lemma *reach-in-beh*:
  assumes *s* ∈ *reach T* **shows** ∃ *b* ∈ *beh T*. *s* ∈ *set b*
using *assms*
proof (*induction s rule*: *reach.induct*)

case (*r-init s*)
hence *s* ∈ *set* [*s*] **and** [*s*] ∈ *beh T* **by** *auto*
thus *?case* **by** *fastforce*
next
case (*r-trans s t*)
then obtain *b* **where** *b* ∈ *beh T* **and** *s* ∈ *set b* **by** *blast*
from ‹*s* ∈ *set b*› **obtain** *b1 b2* **where** *b* = *b2* @ *s* # *b1* **by** (*blast dest*: *split-list*)
with ‹*b* ∈ *beh T*› **have** *s* # *b1* ∈ *beh T* **by** (*blast intro*: *beh-prefix-closed*)
with ‹(*s*, *t*) ∈ *trans T*› **have** *t* # *s* # *b1* ∈ *beh T* **by** *blast*
thus *?case* **by** *force*
qed

lemma *reach-equiv-beh-states*: *reach T* = ⋃ (*set*‘(*beh T*))
by (*auto intro*!: *reach-in-beh beh-in-reach*)

2.1.2 Specifications, observability, and implementation

Specifications add an observer function to transition systems.

record (′*s*, ′*o*) *spec* = ′*s TS* +
obs :: ′*s* ⇒ ′*o*

lemma *beh-obs-upd* [*simp*]: *beh* (*S*(| *obs* := *x* |)) = *beh S*
by (*safe*) (*erule beh.induct*, *auto*)+

lemma *reach-obs-upd* [*simp*]: *reach* (*S*(| *obs* := *x* |)) = *reach S*
by (*safe*) (*erule reach.induct*, *auto*)+

Observable behaviour and reachability.

definition
obeh :: (′*s*, ′*o*) *spec* ⇒ (′*o list*) *set* **where**
obeh S ≡ (*map* (*obs S*))‘(*beh S*)

definition
oreach :: (′*s*, ′*o*) *spec* ⇒ ′*o set* **where**
oreach S ≡ (*obs S*)‘(*reach S*)

lemma *oreach-equiv-obeh-states*:
oreach S = ⋃ (*set*‘(*obeh S*))
by (*auto simp add*: *reach-equiv-beh-states oreach-def obeh-def*)

lemma *obeh-pi-translation*:
(*map pi*)‘(*obeh S*) = *obeh* (*S*(| *obs* := *pi o* (*obs S*) |))
by (*auto simp add*: *obeh-def image-comp*)

lemma *oreach-pi-translation*:
pi‘(*oreach S*) = *oreach* (*S*(| *obs* := *pi o* (*obs S*) |))
by (*auto simp add*: *oreach-def*)

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*]:
  $[\![observable\ ob\ P;\ ob\ s = ob\ s';\ s' \in P]\!] \Longrightarrow s \in P$
by (*unfold observable-def*) (*fast*)

lemma *observable2-equiv-observable*: $observable2\ ob\ P = observable\ ob\ P$
by (*unfold observable-def observable2-def*) (*auto*)

lemma *observable3-equiv-observable2*: $observable3\ ob\ P = observable2\ ob\ P$
by (*unfold observable3-def observable2-def*) (*auto*)

lemma *observable-id* [*simp*]: *observable id P*
by (*simp add*: *observable-def*)

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*: $(ob`P \subseteq Q) = (P \subseteq ob-`Q)$
by *auto*

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*: $[\![\ Q \subseteq ob`P;\ observable\ ob\ P\ ]\!] \Longrightarrow ob-`Q \subseteq P$
by (*auto*)

lemma *vimage-l-image-r*: $[\![\ ob-`Q \subseteq P;\ Q \subseteq range\ ob\ ]\!] \Longrightarrow Q \subseteq ob`P$
by (*drule image-mono* [**where** *f=ob*], *auto*)

Internal and external invariants

lemma *external-from-internal-invariant*:
  $[\![\ reach\ S \subseteq P;\ (obs\ S)`P \subseteq Q\ ]\!]$
  $\Longrightarrow oreach\ S \subseteq Q$
by (*auto simp add*: *oreach-def*)

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

by (*erule external-from-internal-invariant*) (*auto*)

lemma *external-to-internal-invariant-vimage*:
  $[\![$ *oreach* $S \subseteq Q$; (*obs* S)$-$‘$Q \subseteq P$ $]\!]$
  $\Longrightarrow$ *reach* $S \subseteq P$
by (*auto simp add*: *oreach-def*)

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

lemma *external-equiv-internal-invariant-vimage*:
  $[\![$ $P =$ (*obs* S)$-$‘Q $]\!]$
  $\Longrightarrow$ (*oreach* $S \subseteq Q$) $=$ (*reach* $S \subseteq P$)
by (*fast intro*: *external-from-internal-invariant-vimage*
            external-to-internal-invariant-vimage
      del: *subsetI*)

lemma *external-equiv-internal-invariant*:
  $[\![$ (*obs* S)‘$P = Q$; *observable* (*obs* S) P $]\!]$
  $\Longrightarrow$ (*oreach* $S \subseteq Q$) $=$ (*reach* $S \subseteq P$)
by (*rule external-equiv-internal-invariant-vimage*) (*auto*)

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

lemma *implements-trans*:
  $[\![$ *implements pi1 S1 S2*; *implements pi2 S2 S3* $]\!]$
  $\Longrightarrow$ *implements* (*pi1 o pi2*) *S1 S3*
by (*fastforce simp add*: *implements-def image-subset-iff*)

Preservation of external invariants

lemma *implements-oreach*:
  implements pi Sa Sc $\Longrightarrow$ *pi*‘(*oreach Sc*) $\subseteq$ *oreach Sa*
by (*auto simp add*: *implements-def oreach-equiv-obeh-states dest*!: *subsetD*)

lemma *external-invariant-preservation*:
  $[\![$ *oreach Sa* $\subseteq Q$; *implements pi Sa Sc* $]\!]$
  $\Longrightarrow$ *pi*‘(*oreach Sc*) $\subseteq Q$
by (*rule subset-trans* [*OF implements-oreach*]) (*auto*)

lemma *external-invariant-translation*:
  $[\![$ *oreach Sa* $\subseteq Q$; *pi*$-$‘$Q \subseteq P$; *implements pi Sa Sc* $]\!]$

$\Longrightarrow$ *oreach Sc* $\subseteq$ *P*
apply (*rule subset-trans* [*OF vimage-image-subset*, *of pi*])
apply (*rule subset-trans* [**where** *B*=*pi*−‘*Q*])
apply (*intro vimage-mono external-invariant-preservation*, *auto*)
done

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*
by (*rule external-from-internal-invariant-vimage* [
THEN external-invariant-translation,
THEN external-to-internal-invariant-vimage])

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* × '*s*) *set*, '*s set*] ⇒ *bool*
(‹(*3*{-} - {> -})› [*0*, *0*, *0*] *90*)
where
{*pre*} *R* {> *post*} ≡ *R*‘‘*pre* $\subseteq$ *post*

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

lemma {*P*} *R* {> *Q*} = (∀ *s t*. *s* ∈ *P* ⟶ (*s*, *t*) ∈ *R* ⟶ *t* ∈ *Q*)
by (*auto simp add*: *PO-hoare-defs*)

Some essential facts about Hoare triples.

lemma *hoare-conseq-left* [*intro*]:
⟦ {*P'*} *R* {> *Q*}; *P* $\subseteq$ *P'* ⟧
$\Longrightarrow$ {*P*} *R* {> *Q*}
by (*auto simp add*: *PO-hoare-defs*)

lemma *hoare-conseq-right*:
⟦ {*P*} *R* {> *Q'*}; *Q'* $\subseteq$ *Q* ⟧
$\Longrightarrow$ {*P*} *R* {> *Q*}
by (*auto simp add*: *PO-hoare-defs*)

lemma *hoare-false-left* [*simp*]:
{{}} *R* {> *Q*}
by (*auto simp add*: *PO-hoare-defs*)

lemma *hoare-true-right* [*simp*]:
{*P*} *R* {> *UNIV*}
by (*auto simp add*: *PO-hoare-defs*)

lemma *hoare-conj-right* [*intro*!]:
$[\![\{P\}\ R\ \{>\ Q1\};\ \{P\}\ R\ \{>\ Q2\}]\!]$
$\Longrightarrow \{P\}\ R\ \{>\ Q1 \cap Q2\}$
by (*auto simp add*: *PO-hoare-defs*)

Special transition relations.

lemma *hoare-stop* [*simp*, *intro*!]:
$\{P\}\ \{\}\ \{>\ Q\}$
by (*auto simp add*: *PO-hoare-defs*)

lemma *hoare-skip* [*simp*, *intro*!]:
$P \subseteq Q \Longrightarrow \{P\}\ Id\ \{>\ Q\}$
by (*auto simp add*: *PO-hoare-defs*)

lemma *hoare-trans-Un* [*iff*]:
$\{P\}\ R1 \cup R2\ \{>\ Q\} = (\{P\}\ R1\ \{>\ Q\} \wedge \{P\}\ R2\ \{>\ Q\})$
by (*auto simp add*: *PO-hoare-defs*)

lemma *hoare-trans-UN* [*iff*]:
$\{P\} \bigcup\ x.\ R\ x\ \{>\ Q\} = (\forall x.\ \{P\}\ R\ x\ \{>\ Q\})$
by (*auto simp add*: *PO-hoare-defs*)

lemma *hoare-apply*:
$\{P\}\ R\ \{>Q\} \Longrightarrow x \in P \Longrightarrow (x,\ y) \in R \Longrightarrow y \in Q$
by (*auto simp add*: *PO-hoare-defs*)

2.2.2 Characterization of reachability

lemma *reach-init*: $reach\ T \subseteq I \Longrightarrow init\ T \subseteq I$
by (*auto dest*: *subsetD*)

lemma *reach-trans*: $reach\ T \subseteq I \Longrightarrow \{reach\ T\}\ trans\ T\ \{>\ I\}$
by (*auto simp add*: *PO-hoare-defs*)

Useful consequences.

corollary *init-reach* [*iff*]: $init\ T \subseteq reach\ T$
by (*rule reach-init*, *simp*)

corollary *trans-reach* [*iff*]: $\{reach\ T\}\ trans\ T\ \{>\ reach\ T\}$
by (*rule reach-trans*, *simp*)

2.2.3 Invariant proof rules

Basic proof rule for invariants.

lemma *inv-rule-basic*:
$[\![init\ T \subseteq P;\ \{P\}\ (trans\ T)\ \{>\ P\}]\!]$
$\Longrightarrow reach\ T \subseteq P$
by (*safe*, *erule reach.induct*, *auto simp add*: *PO-hoare-def*)

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* $T \subseteq P$
apply (*rule subset-trans*, *auto*) — strengthen goal
apply (*erule reach.induct*, *auto simp add*: *PO-hoare-def*)
done

The following rule is equivalent to the previous one.

lemma *INV-rule*:
  $\llbracket$ *init* $T \subseteq I$; $\{I \cap$ *reach* $T\}$ (*trans* T) $\{> I\}$ $\rrbracket$
  $\Longrightarrow$ *reach* $T \subseteq I$
by (*safe*, *erule reach.induct*, *auto simp add*: *PO-hoare-defs*)

Proof of equivalence.

lemma *inv-rule-from-INV-rule*:
  $\llbracket$ *init* $T \subseteq I$; $I \subseteq P$; $\{I\}$ (*trans* T) $\{> I\}$ $\rrbracket$
  $\Longrightarrow$ *reach* $T \subseteq P$
apply (*rule subset-trans*, *auto del*: *subsetI*)
apply (*rule INV-rule*, *auto*)
done

lemma *INV-rule-from-inv-rule*:
  $\llbracket$ *init* $T \subseteq I$; $\{I \cap$ *reach* $T\}$ (*trans* T) $\{> I\}$ $\rrbracket$
  $\Longrightarrow$ *reach* $T \subseteq I$
by (*rule-tac* I=$I \cap$ *reach* T **in** *inv-rule*, *auto*)

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* $T \subseteq I$
by (*rule INV-rule*, *auto*)

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*]:
  $[\![\ \{pre'\}\ Ra, Rc\ \{>\ post\};\ pre \subseteq pre'\ ]\!]$
  $\Longrightarrow \{pre\}\ Ra, Rc\ \{>\ post\}$
by (*auto simp add*: *PO-rhoare-defs dest*!: *subsetD*)

lemma *relhoare-conseq-right*: — do NOT declare [intro]
  $[\![\ \{pre\}\ Ra, Rc\ \{>\ post'\};\ post' \subseteq post\ ]\!]$
  $\Longrightarrow \{pre\}\ Ra, Rc\ \{>\ post\}$
by (*auto simp add*: *PO-rhoare-defs*)

lemma *relhoare-false-left* [*simp*]: — do NOT declare [intro]
  $\{\ \{\}\ \}\ Ra, Rc\ \{>\ post\}$
by (*auto simp add*: *PO-rhoare-defs*)

lemma *relhoare-true-right* [*simp*]: — not true in general
  $\{pre\}\ Ra, Rc\ \{>\ UNIV\} = (Domain\ (pre\ O\ Rc) \subseteq Domain\ Ra)$
by (*auto simp add*: *PO-rhoare-defs*)

lemma *Domain-rel-comp* [*intro*]:
  $Domain\ pre \subseteq R \Longrightarrow Domain\ (pre\ O\ Rc) \subseteq R$
by (*auto simp add*: *Domain-def*)

lemma *rel-hoare-skip* [*iff*]: $\{R\}\ Id, Id\ \{>\ R\}$
by (*auto simp add*: *PO-rhoare-def*)

Reflexivity and transitivity.

lemma *relhoare-refl* [*simp*]: $\{Id\}\ R, R\ \{>\ Id\}$
by (*auto simp add*: *PO-rhoare-defs*)

lemma *rhoare-trans*:
  $[\![\ \{R1\}\ T1, T2\ \{>\ R1\};\ \{R2\}\ T2, T3\ \{>\ R2\}\ ]\!]$
  $\Longrightarrow \{R1\ O\ R2\}\ T1, T3\ \{>\ R1\ O\ R2\}$
apply (*auto simp add*: *PO-rhoare-def del*: *subsetI*)
apply (*drule subset-refl* [*THEN relcomp-mono*, **where** *r*=*R1*])
apply (*drule subset-refl* [*THEN* [*2*] *relcomp-mono*, **where** *s*=*R2*])
apply (*auto simp add*: *O-assoc del*: *subsetI*)
done

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* ∩ *post2*}
by (*auto simp add*: *PO-rhoare-defs dest*: *single-valuedD dest*!: *subsetD*)

lemma *relhoare-conj-right-cartesian* [*intro*]:
$[\![$ {*Domain pre*} *Ra* {> *I*}; {*Range pre*} *Rc* {> *J*};
{*pre*} *Ra*, *Rc* {> *post*} $]\!]$
$\Longrightarrow$ {*pre*} *Ra*, *Rc* {> *post* ∩ *I* × *J*}
by (*force simp add*: *PO-rhoare-defs PO-hoare-defs Domain-def Range-def*)

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* × *J*}
by (*auto intro*: *relhoare-conseq-right*)

Unions of transition relations.

lemma *relhoare-concrete-Un* [*simp*]:
{*pre*} *Ra*, *Rc1* ∪ *Rc2* {> *post*}
= ({*pre*} *Ra*, *Rc1* {> *post*} ∧ {*pre*} *Ra*, *Rc2* {> *post*})
apply (*auto simp add*: *PO-rhoare-defs*)
apply (*auto dest*!: *subsetD*)
done

lemma *relhoare-concrete-UN* [*simp*]:
{*pre*} *Ra*, ⋃*x*. *Rc x* {> *post*} = (∀*x*. {*pre*} *Ra*, *Rc x* {> *post*})
apply (*auto simp add*: *PO-rhoare-defs*)
apply (*auto dest*!: *subsetD*)
done

lemma *relhoare-abstract-Un-left* [*intro*]:
$[\![$ {*pre*} *Ra1*, *Rc* {> *post*} $]\!]$
$\Longrightarrow$ {*pre*} *Ra1* ∪ *Ra2*, *Rc* {> *post*}
by (*auto simp add*: *PO-rhoare-defs*)

lemma *relhoare-abstract-Un-right* [*intro*]:
$[\![$ {*pre*} *Ra2*, *Rc* {> *post*} $]\!]$
$\Longrightarrow$ {*pre*} *Ra1* ∪ *Ra2*, *Rc* {> *post*}
by (*auto simp add*: *PO-rhoare-defs*)

lemma *relhoare-abstract-UN* [*intro*]: — ! might be too aggressive? INDEED.
$[\![$ {*pre*} *Ra x*, *Rc* {> *post*} $]\!]$
$\Longrightarrow$ {*pre*} ⋃*x*. *Ra x*, *Rc* {> *post*}
apply (*auto simp add*: *PO-rhoare-defs*)
apply (*auto dest*!: *subsetD*)
done

Inclusion of abstract transition relations.

lemma *relhoare-abstract-trans-weak* [*intro*]:
$\llbracket$ {*pre*} *Ra'*, *Rc* {> *post*}; *Ra'* $\subseteq$ *Ra* $\rrbracket$
$\Longrightarrow$ {*pre*} *Ra*, *Rc* {> *post*}
by (*auto simp add*:*PO-rhoare-defs*)

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-scheme*, ('*t*, '*b*) *TS-scheme*] $\Rightarrow$ *bool*
where
PO-refines R Ta Tc $\equiv$ (
init Tc $\subseteq$ *R*``(*init Ta*)
$\wedge$ {*R*} (*trans Ta*), (*trans Tc*) {> *R*}
)

lemma *PO-refinesI*:
$\llbracket$ *init Tc* $\subseteq$ *R*``(*init Ta*); {*R*} (*trans Ta*), (*trans Tc*) {> *R*} $\rrbracket$ $\Longrightarrow$ *PO-refines R Ta Tc*
by (*simp add*: *PO-refines-def*)

lemma *PO-refinesE* [*elim*]:
$\llbracket$ *PO-refines R Ta Tc*; $\llbracket$ *init Tc* $\subseteq$ *R*``(*init Ta*); {*R*} (*trans Ta*), (*trans Tc*) {> *R*} $\rrbracket$ $\Longrightarrow$ *P* $\rrbracket$
$\Longrightarrow$ *P*
by (*simp add*: *PO-refines-def*)

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

lemma *refine-basic*:
$\llbracket$ *init Tc* $\subseteq$ *R*``(*init Ta*); {*R*} (*trans Ta*), (*trans Tc*) {> *R*} $\rrbracket$
$\Longrightarrow$ *PO-refines R Ta Tc*
by (*simp add*: *PO-refines-def*)

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*:
$\llbracket$ *init Tc* $\subseteq$ *R*``(*init Ta*); *init Ta* $\subseteq$ *I*; *init Tc* $\subseteq$ *J* $\rrbracket$
$\Longrightarrow$ *init Tc* $\subseteq$ ($R \cap I \times J$)``(*init Ta*)
by (*auto simp add*: *Image-def dest*!: *bspec subsetD*)

lemma *refine-trans-using-invariants*:
$\llbracket$ {$R \cap I \times J$} (*trans Ta*), (*trans Tc*) {> *R*};

$\{I \cap Domain\ (R \cap UNIV \times J)\}\ (trans\ Ta)\ \{> I\}$;
$\{J \cap Range\ (R \cap I \times UNIV)\}\ (trans\ Tc)\ \{> J\}\ ]\!]$
$\Longrightarrow \{R \cap I \times J\}\ (trans\ Ta), (trans\ Tc)\ \{> R \cap I \times J\}$
by (*rule relhoare-conj-right-cartesian*) (*auto*)

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* $(R \cap I \times J)\ Ta\ Tc$
by (*unfold PO-refines-def*)
(*intro refine-init-using-invariants refine-trans-using-invariants conjI*)

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 $\Longrightarrow$ $init\ Tc \subseteq Range\ R$
by (*auto simp add*: *PO-refines-def*)

lemma *PO-refines-implies-Range-trans*:
PO-refines R Ta Tc $\Longrightarrow$ $\{Range\ R\}\ trans\ Tc\ \{> Range\ R\}$
by (*auto simp add*: *PO-refines-def PO-rhoare-def PO-hoare-def*)

lemma *PO-refines-implies-Range-invariant*:
PO-refines R Ta Tc $\Longrightarrow$ $reach\ Tc \subseteq Range\ R$
by (*rule INV-rule*)
(*auto intro*!: *PO-refines-implies-Range-init*
PO-refines-implies-Range-trans)

The following rules are more useful in proofs.

corollary *INV-init-from-refinement*:
$[\![$ *PO-refines R Ta Tc*; $Range\ R \subseteq I\ ]\!]$
$\Longrightarrow init\ Tc \subseteq I$
by (*drule PO-refines-implies-Range-init*, *auto*)

corollary *INV-trans-from-refinement*:
$[\![$ *PO-refines R Ta Tc*; $K \subseteq Range\ R$; $Range\ R \subseteq I\ ]\!]$
$\Longrightarrow \{K\}\ trans\ Tc\ \{> I\}$
apply (*drule PO-refines-implies-Range-trans*)
apply (*auto intro*: *hoare-conseq-right*)
done

corollary *INV-from-refinement*:
$[\![$ *PO-refines R Ta Tc*; $Range\ R \subseteq I\ ]\!]$
$\Longrightarrow reach\ Tc \subseteq I$
by (*drule PO-refines-implies-Range-invariant*, *fast*)

2.3.4 Refinement of specifications

Lift relation membership to finite sequences

inductive-set
  seq-lift :: (*'s* × *'t*) *set* ⇒ (*'s list* × *'t list*) *set*
  for *R* :: (*'s* × *'t*) *set*
where
  sl-nil [*iff*]: ([], []) ∈ *seq-lift R*
| *sl-cons* [*intro*]:
    ⟦ (*xs*, *ys*) ∈ *seq-lift R*; (*x*, *y*) ∈ *R* ⟧ ⟹ (*x#xs*, *y#ys*) ∈ *seq-lift R*

inductive-cases *sl-cons-right-invert*: (*ba'*, *t* # *bc*) ∈ *seq-lift R*

For each concrete behaviour there is a related abstract one.

lemma *behaviour-refinement*:
  ⟦ *PO-refines R Ta Tc*; *bc* ∈ *beh Tc*⟧
  ⟹ ∃ *ba* ∈ *beh Ta*. (*ba*, *bc*) ∈ *seq-lift R*
apply (*erule beh.induct*, *auto*)
  — case: singleton
  apply (*clarsimp simp add*: *PO-refines-def Image-def*)
  apply (*drule subsetD*, *auto*)

  — case: cons; first construct related abstract state
  apply (*erule sl-cons-right-invert*, *clarsimp*)
  apply (*rename-tac s bc s' ba t*)
  — now construct abstract transition
  apply (*auto simp add*: *PO-refines-def PO-rhoare-def*)
  apply (*thin-tac X* ⊆ *Y* **for** *X Y*)
  apply (*drule subsetD*, *auto*)
done

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* × *'t*) *set*, *'p* ⇒ *'o*, (*'s*, *'o*) *spec*, (*'t*, *'p*) *spec*] ⇒ *bool*
where
  obs-consistent R pi Sa Sc ≡ (∀ *s t*. (*s*, *t*) ∈ *R* ⟶ *pi* (*obs Sc t*) = *obs Sa s*)

lemma *obs-consistent-refl* [*iff*]: *obs-consistent Id id S S*
by (*simp add*: *obs-consistent-def*)

lemma *obs-consistent-trans* [*intro*]:
  ⟦ *obs-consistent R1 pi1 S1 S2*; *obs-consistent R2 pi2 S2 S3* ⟧
  ⟹ *obs-consistent* (*R1 O R2*) (*pi1 o pi2*) *S1 S3*
by (*auto simp add*: *obs-consistent-def*)

lemma *obs-consistent-empty*: *obs-consistent* {} *pi Sa Sc*
by (*auto simp add*: *obs-consistent-def*)

lemma *obs-consistent-conj1* [*intro*]:

obs-consistent R pi Sa Sc ⟹ *obs-consistent* (*R* ∩ *R'*) *pi Sa Sc*
by (*auto simp add*: *obs-consistent-def*)

lemma *obs-consistent-conj2* [*intro*]:
obs-consistent R pi Sa Sc ⟹ *obs-consistent* (*R'* ∩ *R*) *pi Sa Sc*
by (*auto simp add*: *obs-consistent-def*)

lemma *obs-consistent-behaviours*:
⟦ *obs-consistent R pi Sa Sc*; *bc* ∈ *beh Sc*; *ba* ∈ *beh Sa*; (*ba*, *bc*) ∈ *seq-lift R*⟧
⟹ *map pi* (*map* (*obs Sc*) *bc*) = *map* (*obs Sa*) *ba*
by (*erule seq-lift.induct*) (*auto simp add*: *obs-consistent-def*)

Definition of refinement proof obligations.

definition
refines ::
[('s × 't) set, 'p ⇒ 'o, ('s, 'o) spec, ('t, 'p) spec] ⇒ bool
where
refines R pi Sa Sc ≡ *obs-consistent R pi Sa Sc* ∧ *PO-refines R Sa Sc*

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

lemma *refinesI*:
⟦ *PO-refines R Sa Sc*; *obs-consistent R pi Sa Sc* ⟧
⟹ *refines R pi Sa Sc*
by (*simp add*: *refines-def*)

lemma *refinesE* [*elim*]:
⟦ *refines R pi Sa Sc*; ⟦ *PO-refines R Sa Sc*; *obs-consistent R pi Sa Sc* ⟧ ⟹ *P* ⟧
⟹ *P*
by (*simp add*: *refines-def*)

Reflexivity and transitivity of refinement.

lemma *refinement-reflexive*: *refines Id id S S*
by (*auto simp add*: *refines-defs*)

lemma *refinement-transitive*:
⟦ *refines R1 pi1 S1 S2*; *refines R2 pi2 S2 S3* ⟧
⟹ *refines* (*R1 O R2*) (*pi1 o pi2*) *S1 S3*
apply (*auto simp add*: *refines-defs del*: *subsetI*
intro: *rhoare-trans*)
apply (*fastforce dest*: *Image-mono*)
done

Soundness of refinement for proving implementation

lemma *observable-behaviour-refinement*:
⟦ *refines R pi Sa Sc*; *bc* ∈ *obeh Sc* ⟧ ⟹ *map pi bc* ∈ *obeh Sa*
by (*auto simp add*: *refines-def obeh-def image-def*
dest!: *behaviour-refinement obs-consistent-behaviours*)

theorem *refinement-soundness*:
refines R pi Sa Sc ⟹ *implements pi Sa Sc*

by (*auto simp add*: *implements-def*
elim!: *observable-behaviour-refinement*)

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*
by (*auto intro*!: *Refinement-using-invariants*)

Inhertitance of internal invariants through refinements

lemma *INV-init-from-Refinement*:
⟦*refines R pi Sa Sc*; *Range R* $\subseteq$ *I*⟧ $\Longrightarrow$ *init Sc* $\subseteq$ *I*
by (*blast intro*: *INV-init-from-refinement*)

lemma *INV-trans-from-Refinement*:
⟦*refines R pi Sa Sc*; *K* $\subseteq$ *Range R*; *Range R* $\subseteq$ *I*⟧ $\Longrightarrow$ {*K*} *TS.trans Sc* {> *I*}
by (*blast intro*: *INV-trans-from-refinement*)

lemma *INV-from-Refinement-basic*:
⟦ *refines R pi Sa Sc*; *Range R* $\subseteq$ *I* ⟧ $\Longrightarrow$ *reach Sc* $\subseteq$ *I*
by (*rule INV-from-refinement*) *blast*

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 (*rule INV-from-Refinement-basic*)
show *refines* (*R* $\cap$ *reach Sa* $\times$ *reach Sc*) *pi Sa Sc* **using** *assms*(*1*)
by (*rule refines-reachable-strengthening*)
next
show *Range* (*R* $\cap$ *reach Sa* $\times$ *reach Sc*) $\subseteq$ *K* **using** *assms*(*2*–*4*) **by** *blast*
qed

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*
by (*auto elim*: *eq.induct*)

lemma *eq-Sym* [*intro*]: *eq a b* ⟹ *eq b a*
by (*auto elim*: *eq.induct*)

lemma *eq-refl* [*simp*, *intro*]: *eq a a*
by (*induction a*, *auto*)

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*
by (*induction cNumber N y rule*: *eq.induct*, *auto*)
lemma *eq-Agent*: *eq* (*cAgent A*) *y* ⟹ *y* = *cAgent A*
by (*induction cAgent A y rule*: *eq.induct*, *auto*)
lemma *eq-Nonce*: *eq* (*cNonce N*) *y* ⟹ *y* = *cNonce N*
by (*induction cNonce N y rule*: *eq.induct*, *auto*)

lemma *eq-LtK*: *eq* (*cLtK N*) *y* $\Longrightarrow$ *y* = *cLtK N*
  by (*induction cLtK N y rule*: *eq.induct*, *auto*)
lemma *eq-EphK*: *eq* (*cEphK N*) *y* $\Longrightarrow$ *y* = *cEphK N*
  by (*induction cEphK N y rule*: *eq.induct*, *auto*)
lemma *eq-Tag*: *eq* (*cTag N*) *y* $\Longrightarrow$ *y* = *cTag N*
  by (*induction cTag N y rule*: *eq.induct*, *auto*)
lemma *eq-Hash*: *eq* (*cHash X*) *y* $\Longrightarrow \exists Y$. *y* = *cHash Y* $\wedge$ *eq X Y*
  by (*drule eq.induct* [**where** *P*=λx. λy. $\forall$ *X*. *x* = *cHash X* $\longrightarrow$ ($\exists Y$. *y* = *cHash Y* $\wedge$ *eq X Y*)],
    auto elim!:*Tr*)
lemma *eq-Pair*: *eq* (*cPair X Y*) *y* $\Longrightarrow$ $\exists$ X' Y'. *y* = *cPair* X' Y' $\wedge$ *eq X* X' $\wedge$ *eq Y* Y'
  apply (*drule eq.induct* [**where**
   P=λx. λy. $\forall$ *X Y*. *x* = *cPair X Y* $\longrightarrow$ ($\exists$ X' Y'. *y* = *cPair* X' Y' $\wedge$ *eq X* X' $\wedge$ *eq Y* Y')])
  apply (*auto elim*!: *Tr*)
done
lemma *eq-Enc*: *eq* (*cEnc X Y*) *y* $\Longrightarrow$ $\exists$ X' Y'. *y* = *cEnc* X' Y' $\wedge$ *eq X* X' $\wedge$ *eq Y* Y'
  apply (*drule eq.induct* [**where**
   P=λx. λy. $\forall$ *X Y*. *x* = *cEnc X Y* $\longrightarrow$ ($\exists$ X' Y'. *y* = *cEnc* X' Y' $\wedge$ *eq X* X' $\wedge$ *eq Y* Y')])
  apply (*auto elim*!: *Tr*)
done
lemma *eq-Aenc*: *eq* (*cAenc X Y*) *y* $\Longrightarrow$ $\exists$ X' Y'. *y* = *cAenc* X' Y' $\wedge$ *eq X* X' $\wedge$ *eq Y* Y'
  apply (*drule eq.induct* [**where**
   P=λx. λy. $\forall$ *X Y*. *x* = *cAenc X Y* $\longrightarrow$ ($\exists$ X' Y'. *y* = *cAenc* X' Y' $\wedge$ *eq X* X' $\wedge$ *eq Y* Y')])
  apply (*auto elim*!: *Tr*)
done
lemma *eq-Sign*: *eq* (*cSign X Y*) *y* $\Longrightarrow$ $\exists$ X' Y'. *y* = *cSign* X' Y' $\wedge$ *eq X* X' $\wedge$ *eq Y* Y'
  apply (*drule eq.induct* [**where**
   P=λx. λy. $\forall$ *X Y*. *x* = *cSign X Y* $\longrightarrow$ ($\exists$ X' Y'. *y* = *cSign* X' Y' $\wedge$ *eq X* X' $\wedge$ *eq Y* Y')])
  apply (*auto elim*!: *Tr*)
done
lemma *eq-Exp*: *eq* (*cExp X Y*) *y* $\Longrightarrow$ $\exists$ X' Y'. *y* = *cExp* X' Y'
  apply (*drule eq.induct* [**where**
   P=λx. λy. $\forall$ *X Y*. *x* = *cExp X Y* $\longrightarrow$ ($\exists$ X' Y'. *y* = *cExp* X' Y')])
  apply (*auto elim*!: *Tr*)
done

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

lift-definition *Number* :: *nat* $\Rightarrow$ *msg* **is** *cNumber* **by** −
lift-definition *Nonce* :: *nonce-t* $\Rightarrow$ *msg* **is** *cNonce* **by** −
lift-definition *Agent* :: *agent* $\Rightarrow$ *msg* **is** *cAgent* **by** −
lift-definition *LtK* :: *ltkey* $\Rightarrow$ *msg* **is** *cLtK* **by** −
lift-definition *EphK* :: *ephkey* $\Rightarrow$ *msg* **is** *cEphK* **by** −
lift-definition *Pair* :: *msg* $\Rightarrow$ *msg* $\Rightarrow$ *msg* **is** *cPair* **by** *auto*
lift-definition *Enc* :: *msg* $\Rightarrow$ *msg* $\Rightarrow$ *msg* **is** *cEnc* **by** *auto*
lift-definition *Aenc* :: *msg* $\Rightarrow$ *msg* $\Rightarrow$ *msg* **is** *cAenc* **by** *auto*

lift-definition *Exp* :: *msg* ⇒ *msg* ⇒ *msg* **is** *cExp* **by** *auto*
lift-definition *Tag* :: *tag* ⇒ *msg* **is** *cTag* **by** −
lift-definition *Hash* :: *msg* ⇒ *msg* **is** *cHash* **by** *auto*
lift-definition *Sign* :: *msg* ⇒ *msg* ⇒ *msg* **is** *cSign* **by** *auto*

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*
by (*transfer*, *auto*)

lift-definition *atomic* :: *msg* ⇒ *bool* **is** *catomic* **by** (*erule eq.induct*, *auto*)

abbreviation
composed :: *msg* ⇒ *bool* **where**
composed X ≡ ¬*atomic X*

lemma *atomic-Agent* [*simp*, *intro*]: *atomic* (*Agent X*) **by** (*transfer*, *auto*)
lemma *atomic-Tag* [*simp*, *intro*]: *atomic* (*Tag X*) **by** (*transfer*, *auto*)
lemma *atomic-Nonce* [*simp*, *intro*]: *atomic* (*Nonce X*) **by** (*transfer*, *auto*)
lemma *atomic-Number* [*simp*, *intro*]: *atomic* (*Number X*) **by** (*transfer*, *auto*)
lemma *atomic-LtK* [*simp*, *intro*]: *atomic* (*LtK X*) **by** (*transfer*, *auto*)
lemma *atomic-EphK* [*simp*, *intro*]: *atomic* (*EphK X*) **by** (*transfer*, *auto*)

lemma *non-atomic-Pair* [*simp*]: ¬*atomic* (*Pair x y*) **by** (*transfer*, *auto*)
lemma *non-atomic-Enc* [*simp*]: ¬*atomic* (*Enc x y*) **by** (*transfer*, *auto*)
lemma *non-atomic-Aenc* [*simp*]: ¬*atomic* (*Aenc x y*) **by** (*transfer*, *auto*)
lemma *non-atomic-Sign* [*simp*]: ¬*atomic* (*Sign x y*) **by** (*transfer*, *auto*)
lemma *non-atomic-Exp* [*simp*]: ¬*atomic* (*Exp x y*) **by** (*transfer*, *auto*)
lemma *non-atomic-Hash* [*simp*]: ¬*atomic* (*Hash x*) **by** (*transfer*, *auto*)

lemma *Nonce-Nonce*: (*Nonce X* = *Nonce X′*) = (*X* = *X′*) **by** *transfer auto*
lemma *Nonce-Agent*: *Nonce X* ≠ *Agent X′* **by** *transfer auto*
lemma *Nonce-Number*: *Nonce X* ≠ *Number X′* **by** *transfer auto*
lemma *Nonce-Hash*: *Nonce X* ≠ *Hash X′* **by** *transfer auto*
lemma *Nonce-Tag*: *Nonce X* ≠ *Tag X′* **by** *transfer auto*
lemma *Nonce-EphK*: *Nonce X* ≠ *EphK X′* **by** *transfer auto*
lemma *Nonce-LtK*: *Nonce X* ≠ *LtK X′* **by** *transfer auto*
lemma *Nonce-Pair*: *Nonce X* ≠ *Pair X′ Y′* **by** *transfer auto*
lemma *Nonce-Enc*: *Nonce X* ≠ *Enc X′ Y′* **by** *transfer auto*
lemma *Nonce-Aenc*: *Nonce X* ≠ *Aenc X′ Y′* **by** *transfer auto*
lemma *Nonce-Sign*: *Nonce X* ≠ *Sign X′ Y′* **by** *transfer auto*
lemma *Nonce-Exp*: *Nonce X* ≠ *Exp X′ Y′* **by** *transfer auto*

lemma *Agent-Nonce*: *Agent X* ≠ *Nonce X′* **by** *transfer auto*
lemma *Agent-Agent*: (*Agent X* = *Agent X′*) = (*X* = *X′*) **by** *transfer auto*
lemma *Agent-Number*: *Agent X* ≠ *Number X′* **by** *transfer auto*
lemma *Agent-Hash*: *Agent X* ≠ *Hash X′* **by** *transfer auto*
lemma *Agent-Tag*: *Agent X* ≠ *Tag X′* **by** *transfer auto*
lemma *Agent-EphK*: *Agent X* ≠ *EphK X′* **by** *transfer auto*
lemma *Agent-LtK*: *Agent X* ≠ *LtK X′* **by** *transfer auto*

lemma *Agent-Pair*: *Agent* $X \neq$ *Pair* X' Y' **by** *transfer auto*
lemma *Agent-Enc*: *Agent* $X \neq$ *Enc* X' Y' **by** *transfer auto*
lemma *Agent-Aenc*: *Agent* $X \neq$ *Aenc* X' Y' **by** *transfer auto*
lemma *Agent-Sign*: *Agent* $X \neq$ *Sign* X' Y' **by** *transfer auto*
lemma *Agent-Exp*: *Agent* $X \neq$ *Exp* X' Y' **by** *transfer auto*

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

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

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

lemma *EphK-Nonce*: *EphK* $X \neq$ *Nonce* X' **by** *transfer auto*
lemma *EphK-Agent*: *EphK* $X \neq$ *Agent* X' **by** *transfer auto*
lemma *EphK-Number*: *EphK* $X \neq$ *Number* X' **by** *transfer auto*
lemma *EphK-Hash*: *EphK* $X \neq$ *Hash* X' **by** *transfer auto*
lemma *EphK-Tag*: *EphK* $X \neq$ *Tag* X' **by** *transfer auto*
lemma *EphK-EphK*: (*EphK* $X =$ *EphK* X') $= (X = X')$ **by** *transfer auto*
lemma *EphK-LtK*: *EphK* $X \neq$ *LtK* X' **by** *transfer auto*
lemma *EphK-Pair*: *EphK* $X \neq$ *Pair* X' Y' **by** *transfer auto*

lemma *EphK-Enc*: $EphK\ X \neq Enc\ \ X'\ Y'$ **by** *transfer auto*
lemma *EphK-Aenc*: $EphK\ X \neq Aenc\ \ X'\ Y'$ **by** *transfer auto*
lemma *EphK-Sign*: $EphK\ X \neq Sign\ \ X'\ Y'$ **by** *transfer auto*
lemma *EphK-Exp*: $EphK\ X \neq Exp\ \ X'\ Y'$ **by** *transfer auto*

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

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

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

lemma *Aenc-Nonce*: $Aenc\ X\ Y \neq Nonce\ \ X'$ **by** *transfer auto*
lemma *Aenc-Agent*: $Aenc\ X\ Y \neq Agent\ \ X'$ **by** *transfer auto*
lemma *Aenc-Number*: $Aenc\ X\ Y \neq Number\ \ X'$ **by** *transfer auto*
lemma *Aenc-Hash*: $Aenc\ X\ Y \neq Hash\ \ X'$ **by** *transfer auto*
lemma *Aenc-Tag*: $Aenc\ X\ Y \neq Tag\ \ X'$ **by** *transfer auto*
lemma *Aenc-EphK*: $Aenc\ X\ Y \neq EphK\ \ X'$ **by** *transfer auto*
lemma *Aenc-LtK*: $Aenc\ X\ Y \neq LtK\ \ X'$ **by** *transfer auto*
lemma *Aenc-Pair*: $Aenc\ X\ Y \neq Pair\ \ X'\ Y'$ **by** *transfer auto*
lemma *Aenc-Enc*: $Aenc\ X\ Y \neq Enc\ \ X'\ Y'$ **by** *transfer auto*

lemma *Aenc-Aenc*: $(Aenc\ X\ Y = Aenc\ X'\ Y') = (X = X' \wedge Y = Y')$ **by** *transfer auto*
lemma *Aenc-Sign*: $Aenc\ X\ Y \neq Sign\ \ X'\ Y'$ **by** *transfer auto*
lemma *Aenc-Exp*: $Aenc\ X\ Y \neq Exp\ \ X'\ Y'$ **by** *transfer auto*

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

lemma *Exp-Nonce*: $Exp\ X\ Y \neq Nonce\ \ X'$ **by** *transfer auto*
lemma *Exp-Agent*: $Exp\ X\ Y \neq Agent\ \ X'$ **by** *transfer auto*
lemma *Exp-Number*: $Exp\ X\ Y \neq Number\ \ X'$ **by** *transfer auto*
lemma *Exp-Hash*: $Exp\ X\ Y \neq Hash\ \ X'$ **by** *transfer auto*
lemma *Exp-Tag*: $Exp\ X\ Y \neq Tag\ \ X'$ **by** *transfer auto*
lemma *Exp-EphK*: $Exp\ X\ Y \neq EphK\ \ X'$ **by** *transfer auto*
lemma *Exp-LtK*: $Exp\ X\ Y \neq LtK\ \ X'$ **by** *transfer auto*
lemma *Exp-Pair*: $Exp\ X\ Y \neq Pair\ \ X'\ Y'$ **by** *transfer auto*
lemma *Exp-Enc*: $Exp\ X\ Y \neq Enc\ \ X'\ Y'$ **by** *transfer auto*
lemma *Exp-Aenc*: $Exp\ X\ Y \neq Aenc\ \ X'\ Y'$ **by** *transfer auto*
lemma *Exp-Sign*: $Exp\ X\ Y \neq Sign\ \ X'\ Y'$ **by** *transfer auto*

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 ⟹ (∀ *X'*. *X* = *cExp cGen X'* ⟶ (∃ *Z*. *Y* = (*cExp cGen Z*) ∧ *eq X' Z*)) ∧
(∀ *Y'*. *Y* = *cExp cGen Y'* ⟶ (∃ *Z*. *X* = (*cExp cGen Z*) ∧ *eq Y' Z*))
by (*erule eq.induct*, *auto elim*!: *Tr*)

lemma *Exp-Gen-inj*: *Exp Gen X* = *Exp Gen Y* ⟹ *X* = *Y*
by (*transfer*, *auto dest*: *eq-expgen*)

lemma *eq-expexpgen*:
eq X Y ⟹ (∀ *X' X''*. *X* = *cExp* (*cExp cGen X'*) *X''* ⟶
(∃ *Y' Y''*. *Y* = *cExp* (*cExp cGen Y'*) *Y''* ∧
((*eq X' Y'* ∧ *eq X'' Y''*) ∨ (*eq X' Y''* ∧ *eq X'' Y'*))))
apply (*erule eq.induct*, *simp-all*)
apply ((*drule eq-expgen*)+, *force*)
apply (*auto*, *blast*+)
done

lemma *Exp-Exp-Gen-inj*:

$Exp\ (Exp\ Gen\ X)\ X' = Z \Longrightarrow$
$(\exists\ Y\ Y'.\ Z = Exp\ (Exp\ Gen\ Y)\ Y' \wedge ((X = Y \wedge X' = Y') \vee (X = Y' \wedge X' = Y)))$
by (*transfer*, *auto dest*: *eq-expexpgen*)

lemma *Exp-Exp-Gen-inj2*:
$Exp\ (Exp\ Gen\ X)\ X' = Exp\ Z\ Y' \Longrightarrow$
$(Y' = X \wedge Z = Exp\ Gen\ X') \vee (Y' = X' \wedge Z = Exp\ Gen\ X)$
apply (*transfer*, *auto*)
apply (*drule eq-expexpgen*, *auto*)+
done

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* ∈ *H* ⟹ *X* ∈ *synth H*
  | *Agent* [*simp*, *intro*]: *Agent A* ∈ *synth H*
  | *Number* [*simp*, *intro*]: *Number n* ∈ *synth H*
  | *NonceA* [*simp*, *intro*]: *NonceA n* ∈ *synth H*
  | *EpubKA* [*simp*, *intro*]: *epubKA n* ∈ *synth H*
  | *EpriKA* [*simp*, *intro*]: *epriKA n* ∈ *synth H*
  | *Hash* [*intro*]: *X* ∈ *synth H* ⟹ *Hash X* ∈ *synth H*
  | *Pair* [*intro*]: *X* ∈ *synth H* ⟹ *Y* ∈ *synth H* ⟹ (*Pair X Y*) ∈ *synth H*
  | *Enc* [*intro*]: *X* ∈ *synth H* ⟹ *Y* ∈ *synth H* ⟹ (*Enc X Y*) ∈ *synth H*
  | *Aenc* [*intro*]: *X* ∈ *synth H* ⟹ *Y* ∈ *synth H* ⟹ (*Aenc X Y*) ∈ *synth H*
  | *Sign* [*intro*]: *X* ∈ *synth H* ⟹ *Y* ∈ *synth H* ⟹ *Sign X Y* ∈ *synth H*
  | *Exp* [*intro*]: *X* ∈ *synth H* ⟹ *Y* ∈ *synth H* ⟹ (*Exp X Y*) ∈ *synth H*

Lemmas about Dolev-Yao message synthesis.

lemma *synth-mono* [*mono-set*]: *G* ⊆ *H* ⟹ *synth G* ⊆ *synth H*
  by (*auto*, *erule synth.induct*, *auto*)

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

— [*elim*!] slows down certain proofs, e.g., ⟦ *synth H* ∩ *B* ⊆ {} ⟧ ⟹ *P*
inductive-cases *NonceF-synth*: *NonceF n* ∈ *synth H*
inductive-cases *LtK-synth*: *LtK K* ∈ *synth H*
inductive-cases *EpubKF-synth*: *epubKF K* ∈ *synth H*
inductive-cases *EpriKF-synth*: *epriKF K* ∈ *synth H*
inductive-cases *Hash-synth*: *Hash X* ∈ *synth H*
inductive-cases *Pair-synth*: *Pair X Y* ∈ *synth H*
inductive-cases *Enc-synth*: *Enc X K* ∈ *synth H*
inductive-cases *Aenc-synth*: *Aenc X K* ∈ *synth H*
inductive-cases *Sign-synth*: *Sign X K* ∈ *synth H*
inductive-cases *Tag-synth*: *Tag t* ∈ *synth H*

lemma *EpriK-synth* [*elim*]: *epriK K* ∈ *synth H* ⟹
    epriK K ∈ *H* ∨ (∃ *N*. *epriK K* = *epriKA N*)
by (*cases K*, *auto elim*: *EpriKF-synth*)

lemma *EpubK-synth* [*elim*]: *epubK K* ∈ *synth H* ⟹
    epubK K ∈ *H* ∨ (∃ *N*. *epubK K* = *epubKA N*)

by (*cases K*, *auto elim*: *EpubKF-synth*)

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*: $H \subseteq synth\ H$
by *blast*

lemma *synth-Int1*: $x \in synth\ (A \cap B) \Longrightarrow x \in synth\ A$
by (*erule synth.induct*) (*auto*)

lemma *synth-Int2*: $x \in synth\ (A \cap B) \Longrightarrow x \in synth\ B$
by (*erule synth.induct*) (*auto*)

lemma *synth-Int*: $x \in synth\ (A \cap B) \Longrightarrow x \in synth\ A \cap synth\ B$
by (*blast intro*: *synth-Int1 synth-Int2*)

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

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

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

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

lemma *synth-subset-iff*: $synth\ G \subseteq synth\ H \longleftrightarrow G \subseteq synth\ H$
by (*blast dest*: *synth-mono*)

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

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

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

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

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

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

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

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

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

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

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

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

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*: $G \subseteq H \Longrightarrow analz(G) \subseteq analz(H)$
by (*safe*, *erule analz.induct*) (*auto dest*: *Fst Snd synth-Int2*)

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

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

by (*blast dest*: *analz.Fst analz.Snd*)

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

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

lemma *analz-analzD* [*dest*!]: $X \in$ *analz* (*analz* H) $\Longrightarrow X \in$ *analz* H
by (*induction X rule*: *analz.induct*) (*auto dest*: *synth-monotone*)

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

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

lemma *analz-insertI*: $X \in$ *analz* $H \Longrightarrow X \in$ *analz* (*insert* Y H)
by (*blast intro*: *analz-monotone*)

lemma *analz-insert*: *insert* X (*analz* H) $\subseteq$ *analz* (*insert* X H)
by (*blast intro*: *analz-monotone*)

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
by (*blast dest*: *analz-mono*)

lemma *analz-trans*: $X \in$ *analz* $G \Longrightarrow G \subseteq$ *analz* $H \Longrightarrow X \in$ *analz* H
by (*drule analz-mono*) *blast*

lemma *analz-cut*: $Y \in$ *analz* (*insert* X H) $\Longrightarrow X \in$ *analz* $H \Longrightarrow Y \in$ *analz* H
by (*erule analz-trans*) *blast*

lemma *analz-insert-eq*: $X \in$ *analz* $H \Longrightarrow$ *analz* (*insert* X H) = *analz* H
by (*blast intro*: *analz-cut analz-insertI*)

lemma *analz-subset-cong*:
analz $G \subseteq$ *analz* $G' \Longrightarrow$
analz $H \subseteq$ *analz* $H' \Longrightarrow$
analz ($G \cup H$) $\subseteq$ *analz* ($G' \cup H'$)
apply *simp*
apply (*iprover intro*: *conjI subset-trans analz-mono Un-upper1 Un-upper2*)
done

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

by (*intro equalityI analz-subset-cong*, *simp-all*)

lemma *analz-insert-cong*:
  analz H = *analz H′* ⟹
   analz (*insert X H*) = *analz* (*insert X H′*)
by (*force simp only*: *insert-def intro*!: *analz-cong*)

lemma *analz-trivial*:
  ∀ *X Y*. *Pair X Y* ∉ *H* ⟹
   ∀ *X Y*. *Enc X Y* ∉ *H* ⟹
   ∀ *X Y*. *Aenc X Y* ∉ *H* ⟹
   ∀ *X Y*. *Sign X Y* ∉ *H* ⟹
   analz H = *H*
apply *safe*
apply (*erule analz.induct*, *blast*+)
done

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

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

Lemmas about analz and insert.

lemma *analz-insert-Agent* [*simp*]:
  analz (*insert* (*Agent A*) *H*) = *insert* (*Agent A*) (*analz H*)
apply (*rule analz-insert-eq-I*)
apply (*erule analz.induct*)
thm *analz.induct*
apply *fastforce*
apply *fastforce*
apply *fastforce*
defer *1*
apply *fastforce*
defer *1*
apply *fastforce*

apply (*rename-tac x X Y*)
apply (*subgoal-tac Y* ∈ *synth* (*analz H*), *auto*)
apply (*thin-tac Enc X Y* ∈ *Z* **for** *Z*)+
apply (*drule synth-Int2*, *auto*)
apply (*erule synth.induct*, *auto*)

apply (*rename-tac X Y*)
apply (*subgoal-tac epriK Y* ∈ *synth* (*analz H*), *auto*)
apply (*thin-tac Aenc X* (*epubK Y*) ∈ *Z* **for** *Z*)+
apply (*erule synth.induct*, *auto*)
done

4.3 Lemmas about combined composition/decomposition

lemma *synth-analz-incr*: $H \subseteq synth\ (analz\ H)$
by *auto*

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

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

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

lemma *lem1*:
  $Y \in synth\ (analz\ (synth\ G \cup H) \cap (analz\ (G \cup H) \cup synth\ G))$
$\Longrightarrow Y \in synth\ (analz\ (G \cup H))$
apply (*rule subsetD*, *auto simp add*: *synth-subset-iff intro*: *analz-increasing synth-monotone*)
done

lemma *lem2*: $\{a.\ a \in analz\ (G \cup H) \lor a \in synth\ G\} = analz\ (G \cup H) \cup synth\ G$ **by** *auto*

lemma *analz-synth-Un*: $analz\ (synth\ G \cup H) = analz\ (G \cup H) \cup synth\ G$
proof (*intro equalityI subsetI*)
  fix x
  assume $x \in analz\ (synth\ G \cup H)$
  thus $x \in analz\ (G \cup H) \cup synth\ G$
  by (*induction x rule*: *analz.induct*)
    (*auto simp add*: *lem2 intro*: *analz-monotone Fst Snd Dec Adec-eph Adec-lt Sign-getmsg*
      dest: *lem1*)
next
  fix x
  assume $x \in analz\ (G \cup H) \cup synth\ G$
  thus $x \in analz\ (synth\ G \cup H)$
    by (*blast intro*: *analz-monotone*)
qed

lemma *analz-synth*: $analz\ (synth\ H) = analz\ H \cup synth\ H$
by (*rule analz-synth-Un* [**where** *H*={}, *simplified*])

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

lemma *synth-analz-synth* [*simp*]: $synth\ (analz\ (synth\ H)) = synth\ (analz\ H)$
by (*auto del*:*subsetI*) (*auto simp add*: *synth-subset-iff analz-synth*)

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

lemma *synth-analz-idem* [*simp*]: $synth\ (analz\ (synth\ (analz\ H))) = synth\ (analz\ H)$
by (*simp only*: *analz-synth-analz*) *simp*

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

lemma *synth-analz-insert* [*simp*]:
  assumes $X \in synth\ (analz\ H)$
  shows $synth\ (analz\ (insert\ X\ H)) = synth\ (analz\ H)$
using *assms*
proof (*intro equalityI subsetI*)
  fix Z
  assume $Z \in synth\ (analz\ (insert\ X\ H))$
  hence $Z \in synth\ (analz\ (synth\ (analz\ H)))$ **using** *assms*
    by − (*erule synth-analz-monotone*, *rule insert-subset-synth-analz*)
  thus $Z \in synth\ (analz\ H)$ **using** *assms* **by** *simp*
qed (*auto intro*: *synth-analz-monotone*)

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*]: $G \subseteq H \Longrightarrow parts\ G \subseteq parts\ H$
by (*auto*, *erule parts.induct*, *auto*)

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

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

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

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

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

by (*safe*, *erule parts.induct*, *auto*)

lemma *parts-atomic* [*simp*]: *atomic x* $\Longrightarrow$ *parts* {*x*} = {*x*}
by (*auto*, *erule parts.induct*, *auto*)

lemma *parts-InsecTag* [*simp*]: *parts* {*Tag t*} = {*Tag t*}
by (*safe*, *erule parts.induct*) (*auto*)

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

lemma *parts-Tags* [*simp*]:
parts Tags = *Tags*
by (*rule*, *rule*, *erule parts.induct*, *auto*)

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

lemma *parts-Agents* [*simp*]:
parts (*Agent*' *G*) = *Agent*' *G*
by (*auto elim*: *parts.induct*)

lemma *parts-Un* [*simp*]: *parts* (*G* $\cup$ *H*) = *parts G* $\cup$ *parts H*
proof
show *parts* (*G* $\cup$ *H*) $\subseteq$ *parts G* $\cup$ *parts H*
by (*rule*, *erule parts.induct*) (*auto*)
next
show *parts G* $\cup$ *parts H* $\subseteq$ *parts* (*G* $\cup$ *H*)
by (*intro Un-least parts-mono Un-upper1 Un-upper2*)
qed

lemma *parts-insert-subset-Un*:
assumes *X* $\in$ *G*
shows *parts* (*insert X H*) $\subseteq$ *parts G* $\cup$ *parts H*
proof −
from *assms* **have** *parts* (*insert X H*) $\subseteq$ *parts* (*G* $\cup$ *H*) **by** (*blast intro*!: *parts-mono*)
thus *?thesis* **by** *simp*
qed

lemma *parts-insert*: *parts* (*insert X H*) = *parts* {*X*} $\cup$ *parts H*
by (*blast intro*!: *parts-insert-subset-Un intro*: *parts-monotone*)

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

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

lemma *parts-insert-subset*: *insert X* (*parts H*) $\subseteq$ *parts* (*insert X H*)

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

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

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

lemma *parts-subset-iff* [*simp*]: $(parts\ G \subseteq parts\ H) \longleftrightarrow (G \subseteq parts\ H)$
by (*blast dest*: *parts-mono*)

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

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

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

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

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

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

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

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

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

lemma *parts-insert-Enc* [*simp*]:
$parts\ (insert\ (Enc\ X\ Y)\ H) = insert\ (Enc\ X\ Y)\ (parts\ \{X\} \cup parts\ H)$
apply (*rule equalityI*)
apply (*rule subsetI*)

apply (*erule parts.induct*, *auto*)
done

lemma *parts-insert-Aenc* [*simp*]:
parts (*insert* (*Aenc X Y*) *H*) = *insert* (*Aenc X Y*) (*parts* {*X*} ∪ *parts H*)
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*erule parts.induct*, *auto*)
done

lemma *parts-insert-Sign* [*simp*]:
parts (*insert* (*Sign X Y*) *H*) = *insert* (*Sign X Y*) (*parts* {*X*} ∪ *parts H*)
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*erule parts.induct*, *auto*)
done

lemma *parts-insert-Pair* [*simp*]:
parts (*insert* (*Pair X Y*) *H*) = *insert* (*Pair X Y*) (*parts* {*X*} ∪ *parts* {*Y*} ∪ *parts H*)
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*erule parts.induct*, *auto*)
done

4.4.1 Lemmas about combinations with composition and decomposition

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

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

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

lemma *parts-synth* [*simp*]: *parts* (*synth H*) = *parts H* ∪ *synth H*
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*erule parts.induct*)
apply (*blast intro*: *synth-increasing* [*THEN parts-mono*, *THEN subsetD*])+

done

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

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

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

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

4.5 More lemmas about combinations of closures

Combinations of *synth* and *analz*.

lemma *Pair-synth-analz* [*simp*]:
  $Pair\ X\ Y \in synth\ (analz\ H) \longleftrightarrow X \in synth\ (analz\ H) \land Y \in synth\ (analz\ H)$
by *blast*

lemma *Enc-synth-analz*:
  $Y \in synth\ (analz\ H) \Longrightarrow$
   $(Enc\ X\ Y \in synth\ (analz\ H)) \longleftrightarrow (X \in synth\ (analz\ H))$
by *blast*

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

lemma *gen-analz-insert-eq*:
  $[\![\ X \in analz\ G;\ G \subseteq H\ ]\!] \Longrightarrow analz\ (insert\ X\ H) = analz\ H$
by (*blast intro*: *analz-cut analz-insertI analz-monotone*)

lemma *synth-analz-insert-eq*:
  $[\![\ X \in synth\ (analz\ G);\ G \subseteq H\ ]\!] \Longrightarrow synth\ (analz\ (insert\ X\ H)) = synth\ (analz\ H)$
by (*blast intro*!: *synth-analz-insert synth-analz-monotone*)

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

apply (*rule subset-trans*)
apply (*erule-tac* [*2*] *Fake-parts-insert*)
apply (*rule parts-mono*, *blast*)
done

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

lemma *analz-hash-nonce* [*simp*]:
  analz $\{M.\ \exists N.\ M = Hash\ (Nonce\ N)\} = \{M.\ \exists N.\ M = Hash\ (Nonce\ N)\}$
by (*auto*, *rule analz.induct*, *auto*)

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

lemma *synth-analz-idem-mono*:
  $S \subseteq synth\ (analz\ S') \Longrightarrow synth\ (analz\ S) \subseteq synth\ (analz\ S')$
by (*frule synth-analz-mono*, *auto*)

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 −
  assume $analz\ X \subseteq synth\ (analz\ X')$
  then have $HX{:}analz\ X \subseteq analz\ (synth\ (analz\ X'))$ **by** *blast*
  assume $analz\ Y \subseteq synth\ (analz\ Y')$
  then have $HY{:}analz\ Y \subseteq analz\ (synth\ (analz\ Y'))$ **by** *blast*
  from *analz-subset-cong* [*OF HX HY*]
  have $analz\ (X \cup Y) \subseteq analz\ (synth\ (analz\ X') \cup synth\ (analz\ Y'))$
    by *blast*
  also have ... $\subseteq analz\ (synth\ (analz\ X' \cup analz\ Y'))$
    by (*intro analz-mono synth-Un*)
  also have ... $\subseteq analz\ (synth\ (analz\ (X' \cup Y')))$
    by (*intro synth-mono analz-mono analz-Un*)
  also have ... $\subseteq synth\ (analz\ (X' \cup Y'))$
    by *auto*
  finally show $analz\ (X \cup Y) \subseteq synth\ (analz\ (X' \cup Y'))$
    by *auto*
qed

lemma *analz-synth-subset-Un1* :
  $analz\ X \subseteq synth\ (analz\ X') \Longrightarrow analz\ (X \cup Y) \subseteq synth\ (analz\ (X' \cup Y))$
using *analz-synth-subset* **by** *blast*

lemma *analz-synth-subset-Un2* :

analz X ⊆ *synth* (*analz X'*) ⟹ *analz* (*Y* ∪ *X*) ⊆ *synth* (*analz* (*Y* ∪ *X'*))
using *analz-synth-subset* **by** *blast*

lemma *analz-synth-insert*:
analz X ⊆ *synth* (*analz X'*) ⟹ *analz* (*insert Y X*) ⊆ *synth* (*analz* (*insert Y X'*))
by (*metis analz-synth-subset-Un2 insert-def*)

lemma *Fake-analz-insert-Un*:
assumes *Y* ∈ *analz* (*insert X H*) **and** *X* ∈ *synth* (*analz G*)
shows *Y* ∈ *synth* (*analz G*) ∪ *analz* (*G* ∪ *H*)
proof −
from *assms* **have** *Y* ∈ *analz* (*synth* (*analz G*) ∪ *H*)
by (*blast intro*: *analz-mono* [*THEN* [*2*] *rev-subsetD*]
analz-mono [*THEN synth-mono*, *THEN* [*2*] *rev-subsetD*])
thus *?thesis* **by** (*simp add*: *sup.commute*)
qed

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 ∈ *synth* (*analz* (*ik s*)) ∧

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

definition
  ik-trans :: (*'a ik-state-scheme* ∗ *'a ik-state-scheme*) *set*
where
  ik-trans ≡ (⋃ *m*. *ik-dy m*)

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

lemma *ik-trans-ik-increasing*: (*s*, *s'*) ∈ *ik-trans* ⟹ *ik s* ⊆ *ik s'*
by (*auto simp add*: *ik-trans-defs*)

lemma *ik-trans-synth-analz-ik-increasing*:
  (*s*, *s'*) ∈ *ik-trans* ⟹ *synth* (*analz* (*ik s*)) ⊆ *synth* (*analz* (*ik s'*))
by (*simp only*: *synth-analz-mono ik-trans-ik-increasing*)

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*))) ∩ *secret s* = {} ∧

— action
s′ = *s*⦇*ik* := *insert m* (*ik s*)⦈
}

definition
s0-trans :: (′*a s0-state-scheme* ∗ ′*a s0-state-scheme*) *set*
where
s0-trans ≡ (⋃*m*. *s0-add-secret m*) ∪ (⋃*m*. *s0-learn m*) ∪ *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*
by (*simp add*: *s0-def*)

6.2 Proof of secrecy invariant

lemma *s0-secrecy-init* [*iff*]: *init s0* ⊆ *s0-secrecy*
by (*simp add*: *s0-defs*)

lemma *s0-secrecy-trans* [*simp*, *intro*]: {*s0-secrecy*} *trans s0* {> *s0-secrecy*}
apply (*auto simp add*: *s0-all-defs PO-hoare-defs intro*!: *s0-secrecyI*)
apply (*auto*)
done

lemma *s0-secrecy* [*iff*]:*reach s0* ⊆ *s0-secrecy*
by (*rule inv-rule-basic*, *auto*)

lemma *s0-obs-secrecy* [*iff*]:*oreach s0* ⊆ *s0-secrecy*
by (*rule external-from-internal-invariant*) (*auto del*: *subsetI*)

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

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*
by (*simp add*: *a0n-def*)

lemma *a0n-anyP-observable* [*iff*]: *observable* (*obs a0n*) *P*
by (*auto*)

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*
by (*auto simp add*: *a0n-defs intro*!: *a0n-agreementI*)

lemma *a0n-agreement-trans* [*iff*]:
{*a0n-agreement*} *trans a0n* {> *a0n-agreement*}
by (*auto simp add*: *PO-hoare-defs a0n-defs intro*!: *a0n-agreementI*)

lemma *a0n-agreement* [*iff*]: *reach a0n* ⊆ *a0n-agreement*
by (*rule inv-rule-basic*) (*auto*)

lemma *a0n-obs-agreement* [*iff*]:
  oreach a0n ⊆ *a0n-agreement*
apply (*rule external-from-internal-invariant*, *fast*)
apply (*subst a0n-def*, *auto*)
done

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*

by (*simp add*: *a0i-def*)

lemma *a0i-anyP-observable* [*iff*]: *observable* (*obs a0i*) *P*
by (*auto*)

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*
by (*auto simp add*: *a0i-defs intro*!: *a0i-agreementI*)

lemma *PO-a0i-agreement-trans* [*iff*]:
{*a0i-agreement*} *trans a0i* {> *a0i-agreement*}
apply (*auto simp add*: *PO-hoare-defs a0i-defs intro*!: *a0i-agreementI*)
apply (*auto dest*: *a0i-agreementD intro*: *le-SucI*)
done

lemma *PO-a0i-agreement* [*iff*]: *reach a0i* ⊆ *a0i-agreement*
by (*rule inv-rule-basic*) (*auto*)

lemma *PO-a0i-obs-agreement* [*iff*]: *oreach a0i* ⊆ *a0i-agreement*
apply (*rule external-from-internal-invariant*, *fast*)
apply (*subst a0i-def*, *auto*)
done

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*}
by (*unfold R0n0i-def*) (*rule relhoare-refl*)

lemma *PO-a0i-commit-refines-a0n-commit*:
  {*R0n0i*} *a0n-commit A B M*, *a0i-commit A B M* {> *R0n0i*}
by (*auto simp add*: *PO-rhoare-defs R0n0i-def a0i-defs*)

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*)
by (*auto simp add*: *R0n0i-def a0i-defs*)

lemma *PO-a0i-refines-trans-a0n* [*iff*]:
  {*R0n0i*} *trans a0n*, *trans a0i* {> *R0n0i*}
by (*auto simp add*: *a0n-def a0n-trans-def a0i-def a0i-trans-def*
        intro!: *PO-a0i-trans-refines-a0n-trans relhoare-abstract-UN*)

lemma *PO-obs-consistent* [*iff*]:
  obs-consistent R0n0i med0n0i a0n a0i
by (*auto simp add*: *obs-consistent-def R0n0i-def med0n0i-def a0i-def a0n-def*)

lemma *PO-a0i-refines-a0n*:
  refines R0n0i med0n0i a0n a0i
by (*rule Refinement-basic*) (*auto*)

8.4 Derived invariant

lemma *iagreement-implies-niagreement* [*iff*]: *a0i-agreement* ⊆ *a0n-agreement*
apply (*auto intro*!: *a0n-agreementI*)
apply (*drule a0i-agreementD*, *drule order.strict-trans2*, *auto*)
done

lemma *PO-a0i-a0n-agreement* [*iff*]: *reach a0i* ⊆ *a0n-agreement*
by (*rule subset-trans*, *rule*, *rule*)

lemma *PO-a0i-obs-a0n-agreement* [*iff*]: *oreach a0i* ⊆ *a0n-agreement*
by (*rule subset-trans*, *rule*, *rule*)

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*
by (*cases r R*, *auto*)

lemma *in-progress-Some* [*elim*]: *in-progress r x* ⟹ ∃ *x*. *r* = *Some x*
by (*cases r*, *auto*)

lemma *in-progressS-elt* [*elim*]: *in-progressS r S* ⟹ *x* ∈ *S* ⟹ *in-progress r x*
by (*cases r*, *auto*)

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*
by (*auto*)

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

by (*auto*)

lemma *extr-mono-chan* [*dest*]: $G \subseteq H \Longrightarrow$ *extr bad IK G* $\subseteq$ *extr bad IK H*
by (*safe*, *erule extr.induct*, *auto*)

lemma *extr-mono-IK* [*dest*]: *IK1* $\subseteq$ *IK2* $\Longrightarrow$ *extr bad IK1 H* $\subseteq$ *extr bad IK2 H*
by (*safe*) (*erule extr.induct*, *auto*)

lemma *extr-mono-bad* [*dest*]: *bad* $\subseteq$ *bad'* $\Longrightarrow$ *extr bad IK H* $\subseteq$ *extr bad' IK H*
by (*safe*, *erule extr.induct*, *auto*)

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'*
by (*force*)

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*)
by (*auto*)

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

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

lemma *extr-insert-IK-eq* [*simp*]: *extr bad* (*insert X IK*) *CH* = *insert X* (*extr bad IK CH*)
by (*auto*)

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* $\vee$ *Confid B A M* $\in$ *CH* $\vee$
            Secure A B M $\in$ *CH* $\vee$ *Secure B A M* $\in$ *CH*$\}$
by (*rule*, *erule extr.induct*, *auto intro*: *tag.exhaust*)

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*
by *auto*

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*:
$[\![M \in IK; c = insec \vee c = confid \vee A \in bad \vee B \in bad]\!]$
$\Longrightarrow Chan\ c\ A\ B\ M \in fake\ bad\ IK\ chan$

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

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

lemma *fake-mono-bad* [*intro*]:
$bad \subseteq bad' \Longrightarrow fake\ bad\ IK\ chan \subseteq fake\ bad'\ IK\ chan$
by (*auto*)

lemma *fake-mono-ik* [*intro*]:
$IK \subseteq IK' \Longrightarrow fake\ bad\ IK\ chan \subseteq fake\ bad\ IK'\ chan$
by (*auto*)

lemma *fake-mono-chan* [*intro*]:
$chan \subseteq chan' \Longrightarrow fake\ bad\ IK\ chan \subseteq fake\ bad\ IK\ chan'$
by (*auto*)

lemma *fake-mono* [*intro*]:
$[\![bad \subseteq bad'; IK \subseteq IK'; chan \subseteq chan']\!] \Longrightarrow fake\ bad\ IK\ chan \subseteq fake\ bad'\ IK'\ chan'$
by (*auto*, *erule fake.cases*, *auto*)

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*: $chan \subseteq fake\ bad\ IK\ chan$
by *auto*

lemma *extr-fake*:
$X \in fake\ bad\ IK\ chan \Longrightarrow extr\ bad\ IK'\ \{X\} \subseteq IK \cup extr\ bad\ IK'\ chan$
by *auto*

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

lemma *fake-parts-extr-singleton*:
$X \in fake\ bad\ IK\ chan \Longrightarrow parts\ (extr\ bad\ IK'\ \{X\}) \subseteq parts\ IK \cup parts\ (extr\ bad\ IK'\ chan)$
by (*rule extr-fake* [*THEN parts-mono*, *simplified*])

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

lemma *fake-parts-extr-insert*:
assumes *X* ∈ *fake bad IK CH*
shows *parts* (*extr bad IK′* (*insert X CH*)) ⊆ *parts* (*extr bad IK′ CH*) ∪ *parts IK*
proof −
  have *parts* (*extr bad IK′* (*insert X CH*)) ⊆ *parts* (*extr bad IK′* {*X*}) ∪ *parts* (*extr bad IK′ CH*)
    by (*auto simp*: *extr-insert-chan-eq* [**where** *CH*=*CH*])
  also have ... ⊆ *parts* (*extr bad IK′ CH*) ∪ *parts IK* **using** *assms*
    by (*auto dest*!: *fake-parts-extr-singleton*)
  finally show *?thesis* **.**
qed

lemma *fake-synth-analz-extr*:
assumes *X* ∈ *fake bad* (*synth* (*analz* (*extr bad IK CH*))) *CH*
shows *synth* (*analz* (*extr bad IK* (*insert X CH*))) = *synth* (*analz* (*extr bad IK CH*))
using *assms*
proof (*intro equalityI*)
  have *synth* (*analz* (*extr bad IK* (*insert X CH*)))
     ⊆ *synth* (*analz* (*extr bad IK* {*X*} ∪ *extr bad IK CH*))
    by − (*rule synth-analz-mono*, *auto*)
  also have ... ⊆ *synth* (*analz* (*synth* (*analz* (*extr bad IK CH*)) ∪ *extr bad IK CH*)) **using** *assms*
    by − (*rule synth-analz-mono*, *auto*)
  also have ... ⊆ *synth* (*analz* (*synth* (*analz* (*extr bad IK CH*))))
    by − (*rule synth-analz-mono*, *auto*)
  also have ... ⊆ *synth* (*analz* (*extr bad IK CH*)) **by** *simp*
  finally show *synth* (*analz* (*extr bad IK* (*insert X CH*))) ⊆ *synth* (*analz* (*extr bad IK CH*)) **.**
next
  have *extr bad IK CH* ⊆ *extr bad IK* (*insert X CH*)
    by *auto*
  then show *synth* (*analz* (*extr bad IK CH*)) ⊆ *synth* (*analz* (*extr bad IK* (*insert X CH*)))
    by − (*rule synth-analz-mono*, *auto*)
qed

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* {}
by (*auto simp add*: *dy-fake-msg-def*)

lemma *dy-fake-msg-mono-bad* [*dest*]: $bad \subseteq bad' \Longrightarrow$ *dy-fake-msg bad I C* $\subseteq$ *dy-fake-msg bad' I C*
by (*auto simp add*: *dy-fake-msg-def intro*!: *synth-analz-mono*)

lemma *dy-fake-msg-mono-ik* [*dest*]: $G \subseteq H \Longrightarrow$ *dy-fake-msg bad G C* $\subseteq$ *dy-fake-msg bad H C*
by (*auto simp add*: *dy-fake-msg-def intro*!: *synth-analz-mono*)

lemma *dy-fake-msg-mono-chan* [*dest*]: $G \subseteq H \Longrightarrow$ *dy-fake-msg bad I G* $\subseteq$ *dy-fake-msg bad I H*
by (*auto simp add*: *dy-fake-msg-def intro*!: *synth-analz-mono*)

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*)
by (*auto*)

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'*
by (*force simp add*: *dy-fake-msg-def intro*!: *synth-analz-mono*)

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*)
by (*auto simp add*: *dy-fake-msg-def*)

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*]:
  $bad \subseteq bad' \Longrightarrow$ *dy-fake-chan bad I C* $\subseteq$ *dy-fake-chan bad' I C*
by (*auto simp add*: *dy-fake-chan-def*)

lemma *dy-fake-chan-mono-ik* [*intro*]:
  $T \subseteq T' \Longrightarrow$ *dy-fake-chan bad T C* $\subseteq$ *dy-fake-chan bad T' C*
by (*auto simp add*: *dy-fake-chan-def*)

lemma *dy-fake-chan-mono-chan* [*intro*]:
  $C \subseteq C' \Longrightarrow$ *dy-fake-chan bad T C* $\subseteq$ *dy-fake-chan bad T C'*
by (*auto simp add*: *dy-fake-chan-def*)

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 $b \subseteq b'$ **and** $I \subseteq I'$ **and** $C \subseteq C'$
shows *dy-fake-chan b I C* ⊆ *dy-fake-chan b′ I′ C′*
proof −
have *dy-fake-chan b I C* ⊆ *dy-fake-chan b′ I C* **using** ‹$b \subseteq b'$› **by** *auto*
also have ... ⊆ *dy-fake-chan b′ I′ C* **using** ‹$I \subseteq I'$› **by** *auto*
also have ... ⊆ *dy-fake-chan b′ I′ C′* **using** ‹$C \subseteq C'$› **by** *auto*
finally show *?thesis* **.**
qed

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*)
by (*auto simp add*: *dy-fake-msg-def synth-analz-mono*)

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*
apply (*auto simp add*: *dy-fake-chan-def*, *erule fake.cases*, *auto*)
apply (*auto intro*!: *fake-New dest*!: *dy-fake-msg-subset-synth-analz*)
done

lemma *extr-subset-dy-fake-msg*: *extr bad IK chan* ⊆ *dy-fake-msg bad IK chan*
by (*auto simp add*: *dy-fake-msg-def*)

lemma *dy-fake-chan-extr-insert*:
M ∈ *dy-fake-chan bad IK CH* ⟹ *extr bad IK* (*insert M CH*) ⊆ *dy-fake-msg bad IK CH*
by (*auto simp add*: *dy-fake-chan-def dy-fake-msg-def dest*: *fake-synth-analz-extr*)

lemma *dy-fake-chan-extr-insert-parts*:
M ∈ *dy-fake-chan bad IK CH* ⟹
parts (*extr bad IK* (*insert M CH*)) ⊆ *parts* (*extr bad IK CH*) ∪ *dy-fake-msg bad IK CH*
by (*drule dy-fake-chan-extr-insert* [*THEN parts-mono*], *auto simp add*: *dy-fake-msg-def*)

lemma *dy-fake-msg-extr*:
extr bad ik chan ⊆ *synth* (*analz X*) ⟹ *dy-fake-msg bad ik chan* ⊆ *synth* (*analz X*)
by (*drule synth-analz-mono*) (*auto simp add*: *dy-fake-msg-def*)

lemma *extr-insert-dy-fake-msg*:
M ∈ *dy-fake-msg bad IK CH* ⟹ *extr bad* (*insert M IK*) *CH* ⊆ *dy-fake-msg bad IK CH*
by (*auto simp add*: *dy-fake-msg-def*)

lemma *dy-fake-msg-insert-dy-fake-msg*:
M ∈ *dy-fake-msg bad IK CH* ⟹ *dy-fake-msg bad* (*insert M IK*) *CH* ⊆ *dy-fake-msg bad IK CH*
by (*drule synth-analz-mono* [*OF extr-insert-dy-fake-msg*], *auto simp add*: *dy-fake-msg-def*)

lemma *synth-analz-insert-dy-fake-msg*:

$M \in \textit{dy-fake-msg bad IK CH} \Longrightarrow \textit{synth} (\textit{analz} (\textit{insert M IK})) \subseteq \textit{dy-fake-msg bad IK CH}$
by (*auto dest*!: *dy-fake-msg-insert-dy-fake-msg*, *erule subsetD*,
auto simp add: *dy-fake-msg-def elim*: *synth-analz-monotone*)

lemma *Fake-insert-dy-fake-msg*:
$M \in \textit{dy-fake-msg bad IK CH} \Longrightarrow$
$\textit{extr bad IK CH} \subseteq \textit{synth} (\textit{analz } X) \Longrightarrow$
$\textit{synth} (\textit{analz} (\textit{insert M IK})) \subseteq \textit{synth} (\textit{analz } X)$
by (*auto dest*!: *synth-analz-insert-dy-fake-msg dy-fake-msg-extr*)

lemma *dy-fake-chan-insert-chan*:
$x = \textit{insec} \lor x = \textit{auth} \Longrightarrow$
$\textit{Chan } x\ A\ B\ M \in \textit{dy-fake-chan bad IK} (\textit{insert} (\textit{Chan } x\ A\ B\ M)\ CH)$
by (*auto simp add*: *dy-fake-chan-def*)

lemma *dy-fake-chan-subset*:
$CH \subseteq \textit{fake bad} (\textit{dy-fake-msg bad IK CH})\ CH' \Longrightarrow$
$\textit{dy-fake-chan bad IK CH} \subseteq \textit{fake bad} (\textit{dy-fake-msg bad IK CH})\ CH'$
by (*auto simp add*: *dy-fake-chan-def*)

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* **by** −

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

lemma *eq-rep-abs*: *eq x* (*Re* (*Ab x*))
by (*simp add*: *Quotient3-msg rep-abs-rsp*)

lemma *eq-cpayload*:
  assumes *eq x y* **and** *x* ∈ *cpayload*
  shows *y* ∈ *cpayload*
using *assms* **by** (*induction x y rule*: *eq.induct*, *auto intro*: *cpayload.intros elim*: *cpayload.cases*)

lemma *abs-payload*: *Ab x* ∈ *payload* ⟷ *x* ∈ *cpayload*
by (*auto simp add*: *payload-def msg.abs-eq-iff eq-cpayload eq-sym cpayload.intros*
    elim: *cpayload.cases*)

lemma *abs-cpayload-rep*: *x* ∈ *Ab‘ cpayload* ⟷ *Re x* ∈ *cpayload*
apply (*auto elim*: *eq-cpayload* [*OF eq-rep-abs*])
apply (*subgoal-tac x* = *Ab* (*Re x*), *auto*)
using *Quotient3-abs-rep Quotient3-msg* **by** *fastforce*

lemma *payload-rep-cpayload*: *Re x* ∈ *cpayload* ⟷ *x* ∈ *payload*
by (*auto simp add*: *payload-def abs-cpayload-rep*)

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

declare *cpayload.intros* [*intro*]
lemma *payload-AgentI*: *Agent A* ∈ *payload*
 by (*auto simp add*: *msg-defs abs-payload*)
lemma *payload-NonceI*: *Nonce N* ∈ *payload*
 by (*auto simp add*: *msg-defs abs-payload*)
lemma *payload-NumberI*: *Number N* ∈ *payload*
 by (*auto simp add*: *msg-defs abs-payload*)
lemma *payload-EphKI*: *EphK X* ∈ *payload*
 by (*auto simp add*: *msg-defs abs-payload*)
lemma *payload-HashI*: *x* ∈ *payload* ⟹ *Hash x* ∈ *payload*
 by (*auto simp add*: *msg-defs payload-rep-cpayload abs-payload*)
lemma *payload-PairI*: *x* ∈ *payload* ⟹ *y* ∈ *payload* ⟹ *Pair x y* ∈ *payload*
 by (*auto simp add*: *msg-defs payload-rep-cpayload abs-payload*)
lemma *payload-EncI*: *x* ∈ *payload* ⟹ *y* ∈ *payload* ⟹ *Enc x y* ∈ *payload*
 by (*auto simp add*: *msg-defs payload-rep-cpayload abs-payload*)
lemma *payload-AencI*: *x* ∈ *payload* ⟹ *y* ∈ *payload* ⟹ *Aenc x y* ∈ *payload*
 by (*auto simp add*: *msg-defs payload-rep-cpayload abs-payload*)
lemma *payload-SignI*: *x* ∈ *payload* ⟹ *y* ∈ *payload* ⟹ *Sign x y* ∈ *payload*
 by (*auto simp add*: *msg-defs payload-rep-cpayload abs-payload*)
lemma *payload-ExpI*: *x* ∈ *payload* ⟹ *y* ∈ *payload* ⟹ *Exp x y* ∈ *payload*
by (*auto simp add*: *msg-defs payload-rep-cpayload abs-payload*)

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*
apply (*auto simp add*: *payload-def msg-defs msg.abs-eq-iff eq-sym*)
apply (*auto dest*!: *eq-cpayload simp add*: *abs-cpayload-rep*)
done

lemma *payload-LtK*: *LtK X* ∈ *payload* ⟹ *P*
apply (*auto simp add*: *payload-def msg-defs msg.abs-eq-iff eq-sym*)
apply (*auto dest*!: *eq-cpayload simp add*: *abs-cpayload-rep*)
done
lemma *payload-Hash*: *Hash X* ∈ *payload* ⟹ (*X* ∈ *payload* ⟹ *P*) ⟹ *P*
apply (*auto simp add*: *payload-def msg-defs msg.abs-eq-iff eq-sym*)
apply (*auto dest*!: *eq-cpayload simp add*: *abs-cpayload-rep*)
done
lemma *payload-Pair*: *Pair X Y* ∈ *payload* ⟹ (*X* ∈ *payload* ⟹ *Y* ∈ *payload* ⟹ *P*) ⟹ *P*
apply (*auto simp add*: *payload-def msg-defs msg.abs-eq-iff eq-sym*)
apply (*auto dest*!: *eq-cpayload simp add*: *abs-cpayload-rep*)
done
lemma *payload-Enc*: *Enc X Y* ∈ *payload* ⟹ (*X* ∈ *payload* ⟹ *Y* ∈ *payload* ⟹ *P*) ⟹ *P*
apply (*auto simp add*: *payload-def msg-defs msg.abs-eq-iff eq-sym*)
apply (*auto dest*!: *eq-cpayload simp add*: *abs-cpayload-rep*)
done
lemma *payload-Aenc*: *Aenc X Y* ∈ *payload* ⟹ (*X* ∈ *payload* ⟹ *Y* ∈ *payload* ⟹ *P*) ⟹ *P*
apply (*auto simp add*: *payload-def msg-defs msg.abs-eq-iff eq-sym*)
apply (*auto dest*!: *eq-cpayload simp add*: *abs-cpayload-rep*)
done

lemma *payload-Sign*: *Sign X Y* ∈ *payload* ⟹ (*X* ∈ *payload* ⟹ *Y* ∈ *payload* ⟹ *P*) ⟹ *P*
apply (*auto simp add*: *payload-def msg-defs msg.abs-eq-iff eq-sym*)
apply (*auto dest*!: *eq-cpayload simp add*: *abs-cpayload-rep*)
done
lemma *payload-Exp*: *Exp X Y* ∈ *payload* ⟹ (*X* ∈ *payload* ⟹ *Y* ∈ *payload* ⟹ *P*) ⟹ *P*
apply (*auto simp add*: *payload-def Exp-def msg.abs-eq-iff eq-sym*)
apply (*auto dest*!: *eq-cpayload simp add*: *abs-cpayload-rep*)
done

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*) ⟹
(⋀*x*. *eq y* (*cNumber x*) ⟹ *P*) ⟹
(⋀*x*. *eq y* (*cNonce x*) ⟹ *P*) ⟹
(⋀*x*. *eq y* (*cLtK x*) ⟹ *P*) ⟹
(⋀*x*. *eq y* (*cEphK x*) ⟹ *P*) ⟹
(⋀*x x*′. *eq y* (*cPair x x*′) ⟹ *P*) ⟹
(⋀*x x*′. *eq y* (*cEnc x x*′) ⟹ *P*) ⟹
(⋀*x x*′. *eq y* (*cAenc x x*′) ⟹ *P*) ⟹
(⋀*x x*′. *eq y* (*cSign x x*′) ⟹ *P*) ⟹
(⋀*x*. *eq y* (*cHash x*) ⟹ *P*) ⟹
(⋀*x*. *eq y* (*cTag x*) ⟹ *P*) ⟹
(⋀*x x*′. *eq y* (*cExp x x*′) ⟹ *P*) ⟹
P
apply (*cases y*)
apply (*meson Messages.eq-refl*)+
done

lemma *msg-exhaust*:
(⋀*x*. *y* = *Agent x* ⟹ *P*) ⟹
(⋀*x*. *y* = *Number x* ⟹ *P*) ⟹
(⋀*x*. *y* = *Nonce x* ⟹ *P*) ⟹
(⋀*x*. *y* = *LtK x* ⟹ *P*) ⟹
(⋀*x*. *y* = *EphK x* ⟹ *P*) ⟹
(⋀*x x*′. *y* = *Pair x x*′ ⟹ *P*) ⟹
(⋀*x x*′. *y* = *Enc x x*′ ⟹ *P*) ⟹
(⋀*x x*′. *y* = *Aenc x x*′ ⟹ *P*) ⟹
(⋀*x x*′. *y* = *Sign x x*′ ⟹ *P*) ⟹
(⋀*x*. *y* = *Hash x* ⟹ *P*) ⟹
(⋀*x*. *y* = *Tag x* ⟹ *P*) ⟹
(⋀*x x*′. *y* = *Exp x x*′ ⟹ *P*) ⟹
P
apply *transfer*
apply (*erule eq-exhaust*, *auto*)
done

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
by (*erule msg-exhaust* [*of a*], *auto elim*: *payload-inductive-cases*)

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*]:
$[\![X \in parts\ S;\ S \subseteq payload]\!] \Longrightarrow X \in payload$
by (*erule parts.induct*) (*auto*)

lemma *payload-parts-singleton* [*simp*, *dest*]:
$[\![X \in parts\ \{Y\};\ Y \in payload]\!] \Longrightarrow X \in payload$
by (*erule parts.induct*) (*auto*)

lemma *payload-analz* [*simp*, *dest*]:
$[\![X \in analz\ S;\ S \subseteq payload]\!] \Longrightarrow X \in payload$
by (*auto dest*: *analz-into-parts*)

lemma *payload-synth-analz*:
$[\![X \in synth\ (analz\ S);\ S \subseteq payload]\!] \Longrightarrow X \in payload$
by (*erule synth.induct*) (*auto intro*: *payload-analz*)

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

lemma *synth-payload*:
$Y \cap payload = \{\} \Longrightarrow synth\ (X \cup Y) \subseteq synth\ X \cup -payload$
by (*rule*, *erule synth.induct*) (*auto*)

lemma *synth-payload2*:
$Y \cap payload = \{\} \Longrightarrow synth\ (Y \cup X) \subseteq synth\ X \cup -payload$
by (*rule*, *erule synth.induct*) (*auto*)

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

lemma *synth-idem-payload*:
$X \subseteq synth\ Y \cup -payload \Longrightarrow synth\ X \subseteq synth\ Y \cup -payload$
by (*auto dest*: *synth-mono subset-trans* [*OF - synth-payload*])

11.2 *isLtKey*: is a long term key

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

by (*auto*)

lemma *LtKeys-parts-payload* [*dest*]: $NI \subseteq payload \Longrightarrow parts\ NI \cap range\ LtK = \{\}$
by (*auto*)

lemma *LtKeys-parts-payload-singleton* [*elim*]: $X \in payload \Longrightarrow LtK\ Y \in parts\ \{X\} \Longrightarrow False$
by (*auto*)

lemma *parts-of-LtKeys* [*simp*]: $K \subseteq range\ LtK \Longrightarrow parts\ K = K$
by (*rule*, *rule*, *erule parts.induct*, *auto*)

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

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

lemma *keys-of-Ltk* [*intro*!]: $keys\text{-}of\ A \subseteq range\ LtK$
by (*auto simp add*: *keys-of-def*)

lemma *priK-keys-of* [*intro*!]:
$priK\ A \in keys\text{-}of\ A$
by (*simp add*: *keys-of-def*)

lemma *shrK-keys-of-1* [*intro*!]:
$shrK\ A\ B \in keys\text{-}of\ A$
by (*simp add*: *keys-of-def*)

lemma *shrK-keys-of-2* [*intro*!]:
$shrK\ B\ A \in keys\text{-}of\ A$
by (*simp add*: *keys-of-def*)
lemma *priK-keys-of-eq* [*dest*]:
$priK\ B \in keys\text{-}of\ A \Longrightarrow A = B$
by (*simp add*: *keys-of-def*)

lemma *shrK-keys-of-eq* [*dest*]:
$shrK\ A\ B \in keys\text{-}of\ C \Longrightarrow A = C \lor B = C$
by (*simp add*: *keys-of-def*)

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

lemma *parts-keys-of* [*simp*]: $parts\ (keys\text{-}of\ A) = keys\text{-}of\ A$
by (*auto intro*!: *parts-of-LtKeys*)

lemma *analz-keys-of* [*simp*]: $analz\ (keys\text{-}of\ A) = keys\text{-}of\ A$
by (*rule*, *rule*, *erule analz.induct*, *auto*)

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
$Keys\text{-}bad\ IK\ Bad \equiv$
$IK \cap range\ LtK \subseteq range\ pubK \cup \bigcup\ (keys\text{-}of\ `\ Bad)$
$\wedge\ range\ pubK \subseteq IK$

— basic lemmas

lemma *Keys-badI*:
$[\![\ IK \cap range\ LtK \subseteq range\ pubK \cup priK`Bad \cup \{shrK\ A\ B \mid A\ B.\ A \in Bad \vee B \in Bad\};$
$range\ pubK \subseteq IK\ ]\!]$
$\Longrightarrow Keys\text{-}bad\ IK\ Bad$
by (*auto simp add*: *Keys-bad-def*)

lemma *Keys-badE* [*elim*]:
$[\![\ Keys\text{-}bad\ IK\ Bad;$
$[\![\ range\ pubK \subseteq IK;$
$IK \cap range\ LtK \subseteq range\ pubK \cup \bigcup\ (keys\text{-}of\ `\ Bad)]\!]$
$\Longrightarrow P\ ]\!]$
$\Longrightarrow P$
by (*auto simp add*: *Keys-bad-def*)

lemma *Keys-bad-Ltk* [*simp*]:
$Keys\text{-}bad\ (IK \cap range\ LtK)\ Bad \longleftrightarrow Keys\text{-}bad\ IK\ Bad$
by (*auto simp add*: *Keys-bad-def*)

lemma *Keys-bad-priK-D*: $[\![\ priK\ A \in IK;\ Keys\text{-}bad\ IK\ Bad\ ]\!] \Longrightarrow A \in Bad$
by (*auto simp add*: *Keys-bad-def*)

lemma *Keys-bad-shrK-D*: $[\![\ shrK\ A\ B \in IK;\ Keys\text{-}bad\ IK\ Bad\ ]\!] \Longrightarrow A \in Bad \vee B \in Bad$
by (*auto simp add*: *Keys-bad-def*)

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

interaction with *insert*.

lemma *Keys-bad-insert-non-LtK*:
$X \notin range\ LtK \Longrightarrow Keys\text{-}bad\ (insert\ X\ IK)\ Bad \longleftrightarrow Keys\text{-}bad\ IK\ Bad$
by (*auto simp add*: *Keys-bad-def*)

lemma *Keys-bad-insert-pubK*:
$[\![\ Keys\text{-}bad\ IK\ Bad\ ]\!] \Longrightarrow Keys\text{-}bad\ (insert\ (pubK\ A)\ IK)\ Bad$
by (*auto simp add*: *Keys-bad-def*)

lemma *Keys-bad-insert-priK-bad*:
$[\![\ Keys\text{-}bad\ IK\ Bad;\ A \in Bad\ ]\!] \Longrightarrow Keys\text{-}bad\ (insert\ (priK\ A)\ IK)\ Bad$
by (*auto simp add*: *Keys-bad-def*)

lemma *Keys-bad-insert-shrK-bad*:
  ⟦ *Keys-bad IK Bad*; $A \in Bad \lor B \in Bad$ ⟧ $\Longrightarrow$ *Keys-bad* (*insert* (*shrK A B*) *IK*) *Bad*
by (*auto simp add*: *Keys-bad-def*)

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 $X \in$ *synth* (*analz IK*)
shows *Keys-bad* (*insert X IK*) *Bad*
proof *cases*
  assume $X \in$ *range LtK*
  then obtain *ltk* **where** X = *LtK ltk* **by** *blast*
  thus *?thesis* **using** *assms*
    by (*auto simp add*: *insert-absorb dest*: *analz-into-parts*)
next
  assume $X \notin$ *range LtK*
  thus *?thesis* **using** *assms*(*1*) **by** *simp*
qed

lemma *Keys-bad-insert-keys-of*:
  Keys-bad Ik Bad $\Longrightarrow$
   Keys-bad (*keys-of A* ∪ *Ik*) (*insert A Bad*)
by (*auto simp add*: *Keys-bad-def*)

lemma *Keys-bad-insert-payload*:
  Keys-bad Ik Bad $\Longrightarrow$
   $x \in$ *payload* $\Longrightarrow$
   Keys-bad (*insert x Ik*) *Bad*
by (*auto simp add*: *Keys-bad-def*)

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* ⇒ (*agent* ∗ *agent*) *set*
where
  broken K ≡ {(*A*,*B*) | *A B*. *priK A* ∈ *K* ∨ *priK B* ∈ *K* ∨ *shrK A B* ∈ *K* ∨ *shrK B A* ∈ *K*}

lemma *brokenD* [*dest*!]:
  (*A*, *B*) ∈ *broken K* $\Longrightarrow$ *priK A* ∈ *K* ∨ *priK B* ∈ *K* ∨ *shrK A B* ∈ *K* ∨ *shrK B A* ∈ *K*
by (*simp add*: *broken-def*)

lemma *brokenI* [*intro*!]:
  priK A ∈ *K* ∨ *priK B* ∈ *K* ∨ *shrK A B* ∈ *K* ∨ *shrK B A* ∈ *K* $\Longrightarrow$ (*A*, *B*) ∈ *broken K*
by (*auto simp add*: *broken-def*)

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-keys-clean :: *msg set* ⇒ *bool*
where
  Enc-keys-clean S ≡ ∀ *X Y*. *Enc X Y* ∈ *parts S* ⟶ *Y* ∈ *range LtK* ∪ *payload*

lemma *Enc-keys-cleanI*:
  ∀ *X Y*. *Enc X Y* ∈ *parts S* ⟶ *Y* ∈ *range LtK* ∪ *payload* ⟹ *Enc-keys-clean S*
by (*simp add*: *Enc-keys-clean-def*)

— general lemmas about *Enc-keys-clean*
lemma *Enc-keys-clean-mono*:
  Enc-keys-clean H ⟹ *G* ⊆ *H* ⟹ *Enc-keys-clean G* — anti-tone
by (*auto simp add*: *Enc-keys-clean-def dest*!: *parts-monotone* [**where** *G*=*G*])

lemma *Enc-keys-clean-Un* [*simp*]:
  Enc-keys-clean (*G* ∪ *H*) ⟷ *Enc-keys-clean G* ∧ *Enc-keys-clean H*
by (*auto simp add*: *Enc-keys-clean-def*)

— from *Enc-keys-clean S*, the property on *parts S* also holds for *analz S*
lemma *Enc-keys-clean-analz*:
  Enc X K ∈ *analz S* ⟹ *Enc-keys-clean S* ⟹ *K* ∈ *range LtK* ∪ *payload*
by (*auto simp add*: *Enc-keys-clean-def dest*: *analz-into-parts*)

— *Enc-keys-clean* and different types of messages
lemma *Enc-keys-clean-Tags* [*simp*,*intro*]: *Enc-keys-clean Tags*
by (*auto simp add*: *Enc-keys-clean-def*)

lemma *Enc-keys-clean-LtKeys* [*simp*,*intro*]: *K* ⊆ *range LtK* ⟹ *Enc-keys-clean K*
by (*auto simp add*: *Enc-keys-clean-def*)

lemma *Enc-keys-clean-payload* [*simp*,*intro*]: *NI* ⊆ *payload* ⟹ *Enc-keys-clean NI*
by (*auto simp add*: *Enc-keys-clean-def*)

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 $AencSet\ G\ K \equiv \{Aenc\ X\ Y \mid X\ Y.\ X \in G \wedge Y \in K\}$

abbreviation $SignSet :: msg\ set \Rightarrow msg\ set \Rightarrow msg\ set$
where $SignSet\ G\ K \equiv \{Sign\ X\ Y \mid X\ Y.\ X \in G \wedge Y \in K\}$

abbreviation $HashSet :: msg\ set \Rightarrow msg\ set$
where $HashSet\ G \equiv \{Hash\ X \mid X.\ X \in G\}$

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

lemma *parts-PairSet*:
  $parts\ (PairSet\ G\ H) \subseteq PairSet\ G\ H \cup parts\ G \cup parts\ H$
by (*rule*, *erule parts.induct*, *auto*)

lemma *parts-EncSet*:
  $parts\ (EncSet\ G\ K) \subseteq EncSet\ G\ K \cup PairSet\ (range\ Agent)\ G \cup range\ Agent \cup parts\ G$
by (*rule*, *erule parts.induct*, *auto*)

lemma *parts-AencSet*:
  $parts\ (AencSet\ G\ K) \subseteq AencSet\ G\ K \cup PairSet\ (range\ Agent)\ G \cup range\ Agent \cup parts\ G$
by (*rule*, *erule parts.induct*, *auto*)

lemma *parts-SignSet*:
  $parts\ (SignSet\ G\ K) \subseteq SignSet\ G\ K \cup PairSet\ (range\ Agent)\ G \cup range\ Agent \cup parts\ G$
by (*rule*, *erule parts.induct*, *auto*)

lemma *parts-HashSet*:
  $parts\ (HashSet\ G) \subseteq HashSet\ G$
by (*rule*, *erule parts.induct*, *auto*)

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*:
  $Enc\text{-}keys\text{-}clean\ (G \cup H) \Longrightarrow Enc\text{-}keys\text{-}clean\ (PairSet\ (Agent`X)\ G \cup H)$
by (*auto simp add*: *Enc-keys-clean-def dest*!: *parts-msgSetD*)

lemma *Enc-keys-clean-PairSet-Tag-Un*:
  $Enc\text{-}keys\text{-}clean\ (G \cup H) \Longrightarrow Enc\text{-}keys\text{-}clean\ (PairSet\ Tags\ G \cup H)$
by (*auto simp add*: *Enc-keys-clean-def dest*!: *parts-msgSetD*)

lemma *Enc-keys-clean-AencSet-Un*:
  $Enc\text{-}keys\text{-}clean\ (G \cup H) \Longrightarrow Enc\text{-}keys\text{-}clean\ (AencSet\ G\ K \cup H)$
by (*auto simp add*: *Enc-keys-clean-def dest*!: *parts-msgSetD*)

lemma *Enc-keys-clean-EncSet-Un*:
  $K \subseteq range\ LtK \Longrightarrow Enc\text{-}keys\text{-}clean\ (G \cup H) \Longrightarrow Enc\text{-}keys\text{-}clean\ (EncSet\ G\ K \cup H)$
by (*auto simp add*: *Enc-keys-clean-def dest*!: *parts-msgSetD*)

lemma *Enc-keys-clean-SignSet-Un*:
  $Enc\text{-}keys\text{-}clean\ (G \cup H) \Longrightarrow Enc\text{-}keys\text{-}clean\ (SignSet\ G\ K \cup H)$

by (*auto simp add*: *Enc-keys-clean-def dest*!: *parts-msgSetD*)

lemma *Enc-keys-clean-HashSet-Un*:
  Enc-keys-clean (*G* ∪ *H*) ⟹ *Enc-keys-clean* (*HashSet G* ∪ *H*)
by (*auto simp add*: *Enc-keys-clean-def dest*!: *parts-msgSetD*)

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* ⊆ *range LtK* **and** *Enc-keys-clean* (*G* ∪ *H*)
shows *analz* (*EncSet G K* ∪ *H*) ⊆ *EncSet G K* ∪ *analz* (*G* ∪ *H*)
proof
  fix *X*
  assume *X* ∈ *analz* (*EncSet G K* ∪ *H*)
  thus *X* ∈ *EncSet G K* ∪ *analz* (*G* ∪ *H*)
  proof (*induction X rule*: *analz.induct*)
    case (*Dec Y K'*)
    from *Dec.IH*(*1*) **show** *?case*
    proof
      assume *Enc Y K'* ∈ *analz* (*G* ∪ *H*)
      have *K'* ∈ *synth* (*analz* (*G* ∪ *H*))
      proof −
        have *K'* ∈ *range LtK* ∪ *payload* **using** ‹*Enc Y K'* ∈ *analz* (*G* ∪ *H*)› *assms*(*2*)
          by (*blast dest*: *Enc-keys-clean-analz*)
        moreover
        have *K'* ∈ *synth* (*EncSet G K* ∪ *analz* (*G* ∪ *H*)) **using** *Dec.IH*(*2*)
          by (*auto simp add*: *Collect-disj-eq dest*: *synth-Int2*)
        moreover
        hence *K'* ∈ *synth* (*analz* (*G* ∪ *H*)) ∪ −*payload* **using** *assms*(*1*)
          by (*blast dest*!: *synth-payload2* [*THEN* [*2*] *rev-subsetD*])
        ultimately show *?thesis* **by** *auto*
      qed
      thus *?case* **using** ‹*Enc Y K'* ∈ *analz* (*G* ∪ *H*)› **by** *auto*
    qed *auto*
  next
    case (*Adec-eph Y K'*)
    thus *?case* **by** (*auto dest*!: *EpriK-synth*)
  qed (*auto*)
qed

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

lemma *analz-Un-EncSet2*:
assumes *Enc-keys-clean H* **and** *K* ⊆ *range LtK* **and** *K* ∩ *synth* (*analz H*) = {}
shows *analz* (*EncSet G K* ∪ *H*) ⊆ *EncSet G K* ∪ *analz H*
proof
  fix *X*

assume *X* ∈ *analz* (*EncSet G K* ∪ *H*)
thus *X* ∈ *EncSet G K* ∪ *analz H*
proof (*induction X rule*: *analz.induct*)
case (*Dec Y K′*)
from *Dec.IH*(*1*) **show** *?case*
proof
assume *Enc Y K′* ∈ *analz H*
moreover have *K′* ∈ *synth* (*analz H*)
proof −
have *K′* ∈ *range LtK* ∪ *payload* **using** ‹*Enc Y K′* ∈ *analz H*› *assms*(*1*)
by (*auto dest*: *Enc-keys-clean-analz*)
moreover
from *Dec.IH*(*2*) **have** *H*: *K′* ∈ *synth* (*EncSet G K* ∪ *analz H*)
by (*auto simp add*: *Collect-disj-eq dest*: *synth-Int2*)
moreover
hence *K′* ∈ *synth* (*analz H*) ∪ −*payload*
proof (*rule synth-payload2* [*THEN* [*2*] *rev-subsetD*], *auto elim*!: *payload-Enc*)
fix *X Y*
assume *Y* ∈ *K Y* ∈ *payload*
with ‹*K* ⊆ *range LtK*› **obtain** *KK* **where** *Y* = *LtK KK* **by** *auto*
with ‹*Y* ∈ *payload*› **show** *False* **by** *auto*
qed
ultimately
show *?thesis* **by** *auto*
qed
ultimately show *?case* **by** *auto*
next
assume *Enc Y K′* ∈ *EncSet G K*
moreover hence *K′* ∈ *K* **by** *auto*
moreover with ‹*K* ⊆ *range LtK*› **obtain** *KK* **where** *K′* = *LtK KK* **by** *auto*
moreover with *Dec.IH*(*2*) **have** *K′* ∈ *analz H*
by (*auto simp add*: *Collect-disj-eq dest*: *synth-Int2*)
ultimately show *?case* **using** ‹*K* ∩ *synth* (*analz H*) = {}› **by** *auto*
qed
next
case (*Adec-eph Y K′*)
thus *?case* **by** (*auto dest*!: *EpriK-synth*)
qed (*insert assms*(*2*), *auto*)
qed

Pull *AencSet* out of the *analz*.

lemma *analz-Un-AencSet*:
assumes *K* ⊆ *range LtK* **and** *Enc-keys-clean* (*G* ∪ *H*)
shows *analz* (*AencSet G K* ∪ *H*) ⊆ *AencSet G K* ∪ *analz* (*G* ∪ *H*)
proof
fix *X*
assume *X* ∈ *analz* (*AencSet G K* ∪ *H*)
thus *X* ∈ *AencSet G K* ∪ *analz* (*G* ∪ *H*)
proof (*induction X rule*: *analz.induct*)
case (*Dec Y K′*)
from *Dec.IH*(*1*) **have** *Enc Y K′* ∈ *analz* (*G* ∪ *H*) **by** *auto*
moreover have *K′* ∈ *synth* (*analz* (*G* ∪ *H*))
proof −

have *K′* ∈ *range LtK* ∪ *payload* **using** ‹*Enc Y K′* ∈ *analz* (*G* ∪ *H*)› *assms*(*2*)
by (*blast dest*: *Enc-keys-clean-analz*)
moreover
have *K′* ∈ *synth* (*AencSet G K* ∪ *analz* (*G* ∪ *H*)) **using** *Dec.IH*(*2*)
by (*auto simp add*: *Collect-disj-eq dest*: *synth-Int2*)
moreover
hence *K′* ∈ *synth* (*analz* (*G* ∪ *H*)) ∪ −*payload* **using** *assms*(*1*)
by (*blast dest*!: *synth-payload2* [*THEN* [*2*] *rev-subsetD*])
ultimately
show *?thesis* **by** *auto*
qed
ultimately show *?case* **by** *auto*
next
case (*Adec-eph Y K′*)
thus *?case* **by** (*auto dest*!: *EpriK-synth*)
qed *auto*
qed

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

lemma *analz-Un-AencSet2*:
assumes *Enc-keys-clean H* **and** *priK‘Ag* ∩ *synth* (*analz H*) = {}
shows *analz* (*AencSet G* (*pubK‘Ag*) ∪ *H*) ⊆ *AencSet G* (*pubK‘Ag*) ∪ *analz H*
proof
fix *X*
assume *X* ∈ *analz* (*AencSet G* (*pubK‘ Ag*) ∪ *H*)
thus *X* ∈ *AencSet G* (*pubK‘ Ag*) ∪ *analz H*
proof (*induction X rule*: *analz.induct*)
case (*Dec Y K′*)
from *Dec.IH*(*1*) **have** *Enc Y K′* ∈ *analz H* **by** *auto*
moreover have *K′* ∈ *synth* (*analz H*)
proof −
have *K′* ∈ *range LtK* ∪ *payload* **using** ‹*Enc Y K′* ∈ *analz H*› *assms*(*1*)
by (*auto dest*: *Enc-keys-clean-analz*)
moreover
from *Dec.IH*(*2*) **have** *H*: *K′* ∈ *synth* (*AencSet G* (*pubK‘Ag*) ∪ *analz H*)
by (*auto simp add*: *Collect-disj-eq dest*: *synth-Int2*)
moreover
hence *K′* ∈ *synth* (*analz H*) ∪ −*payload*
by (*auto dest*: *synth-payload2* [*THEN* [*2*] *rev-subsetD*])
ultimately
show *?thesis* **by** *auto*
qed
ultimately show *?case* **by** *auto*
next
case (*Adec-eph Y K′*)
thus *?case* **by** (*auto dest*!: *EpriK-synth*)
qed (*insert assms*(*2*), *auto*)
qed

Pull *PairSet* out of *analz*.

lemma *analz-Un-PairSet*:
analz (*PairSet G G′* ∪ *H*) ⊆ *PairSet G G′* ∪ *analz* (*G* ∪ *G′* ∪ *H*)

proof
  fix *X*
  assume $X \in$ *analz* (*PairSet G G'* $\cup$ *H*)
  thus $X \in$ *PairSet G G'* $\cup$ *analz* ($G \cup G' \cup H$)
  proof (*induct X rule*: *analz.induct*)
    case (*Dec Y K*)
    from *Dec.hyps*(*2*) **have** *Enc Y K* $\in$ *analz* ($G \cup G' \cup H$) **by** *auto*
    moreover
    from *Dec.hyps*(*3*) **have** $K \in$ *synth* (*PairSet G G'* $\cup$ *analz* ($G \cup G' \cup H$))
      by (*auto simp add*: *Collect-disj-eq dest*: *synth-Int2*)
    then have $K \in$ *synth* (*analz* ($G \cup G' \cup H$))
      by (*elim synth-trans*) *auto*
    ultimately
    show *?case* **by** *auto*
  next
    case (*Adec-eph Y K*)
    thus *?case* **by** (*auto dest*!: *EpriK-synth*)
  qed *auto*
qed

— move the *SignSet* out of the *analz*
lemma *analz-Un-SignSet*:
assumes $K \subseteq$ *range LtK* **and** *Enc-keys-clean* ($G \cup H$)
shows *analz* (*SignSet G K* $\cup$ *H*) $\subseteq$ *SignSet G K* $\cup$ *analz* ($G \cup H$)
proof
  fix *X*
  assume $X \in$ *analz* (*SignSet G K* $\cup$ *H*)
  thus $X \in$ *SignSet G K* $\cup$ *analz* ($G \cup H$)
  proof (*induct X rule*: *analz.induct*)
    case (*Dec Y K'*)
    from *Dec.hyps*(*2*) **have** *Enc Y K'* $\in$ *analz* ($G \cup H$) **by** *auto*
    moreover have $K' \in$ *synth* (*analz* ($G \cup H$))
    proof $-$
      have $K' \in$ *range LtK* $\cup$ *payload* **using** ‹*Enc Y K'* $\in$ *analz* ($G \cup H$)› *assms*(*2*)
        by (*blast dest*: *Enc-keys-clean-analz*)
      moreover
      from *Dec.hyps*(*3*) **have** $K' \in$ *synth* (*SignSet G K* $\cup$ *analz* ($G \cup H$))
        by (*auto simp add*: *Collect-disj-eq dest*: *synth-Int2*)
      moreover
      hence $K' \in$ *synth* (*analz* ($G \cup H$)) $\cup$ $-$*payload* **using** *assms*(*1*)
        by (*blast dest*!: *synth-payload2* [*THEN* [*2*] *rev-subsetD*])
      ultimately
      show *?thesis* **by** *auto*
    qed
    ultimately show *?case* **by** *auto*
  next
    case (*Adec-eph Y K*)
    thus *?case* **by** (*auto dest*!: *EpriK-synth*)
  qed *auto*
qed

Pull *Tags* out of *analz*.

lemma *analz-Un-Tag*:

```
assumes Enc-keys-clean H
shows analz (Tags ∪ H) ⊆ Tags ∪ analz H
proof
  fix X
  assume X ∈ analz (Tags ∪ H)
  thus X ∈ Tags ∪ analz H
  proof (induction X rule: analz.induct)
    case (Dec Y K')
    have Enc Y K' ∈ analz H using Dec.IH(1) by auto
    moreover have K' ∈ synth (analz H)
    proof -
      have K' ∈ range LtK ∪ payload using ‹Enc Y K' ∈ analz H› assms
        by (auto dest: Enc-keys-clean-analz)
      moreover
      from Dec.IH(2) have K' ∈ synth (Tags ∪ analz H)
        by (auto simp add: Collect-disj-eq dest: synth-Int2)
      moreover
      hence K' ∈ synth (analz H) ∪ −payload
        by (auto dest: synth-payload2 [THEN [2] rev-subsetD])
      ultimately show ?thesis by auto
    qed
    ultimately show ?case by (auto)
  next
    case (Adec-eph Y K')
    thus ?case by (auto dest!: EpriK-synth)
  qed auto
qed
```

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

```
lemma analz-Un-AgentSet:
shows analz (AgentSet ∪ H) ⊆ AgentSet ∪ analz H
proof
  fix X
  assume X ∈ analz (AgentSet ∪ H)
  thus X ∈ AgentSet ∪ analz H
  proof (induction X rule: analz.induct)
    case (Dec Y K')
    from Dec.IH(1) have Enc Y K' ∈ analz H by auto
    moreover have K' ∈ synth (analz H)
      proof -
        from Dec.IH(2) have K' ∈ synth (AgentSet ∪ analz H)
          by (auto simp add: Collect-disj-eq dest: synth-Int2)
        moreover have synth (AgentSet ∪ analz H) ⊆ synth (analz H)
          by (auto simp add: synth-subset-iff)
        ultimately show ?thesis by auto
    qed
    ultimately show ?case by (auto)
  next
    case (Adec-eph Y K')
    thus ?case by (auto dest!: EpriK-synth)
  qed auto
qed
```

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
  fix X
  assume X ∈ analz (HashSet G ∪ H)
  thus X ∈ HashSet G ∪ analz H
  proof (induction X rule: analz.induct)
    case (Dec Y K')
    from Dec.IH(1) have Enc Y K' ∈ analz H by auto
    moreover have K' ∈ synth (analz H)
    proof −
      have K' ∈ range LtK ∪ payload using ‹Enc Y K' ∈ analz H› assms(1)
      by (auto dest: Enc-keys-clean-analz)
      thus ?thesis
      proof
        assume K' ∈ range LtK
        then obtain KK where K' = LtK KK by auto
        moreover
        with Dec.IH(2) show ?thesis
          by (auto simp add: Collect-disj-eq dest: synth-Int2)
      next
        assume K' ∈ payload
        moreover
        from assms have HashSet G ∩ payload = {} by auto
        moreover from Dec.IH(2) have K' ∈ synth (HashSet G ∪ analz H)
          by (auto simp add: Collect-disj-eq dest: synth-Int2)
        ultimately
        have K' ∈ synth (analz H) ∪ −payload
          by (auto dest: synth-payload2 [THEN [2] rev-subsetD])
        with ‹K' ∈ payload› show ?thesis by auto
      qed
    qed
    ultimately show ?case by auto
  next
    case (Adec-eph Y K')
    thus ?case by (auto dest!: EpriK-synth)
  qed (insert assms(2), auto)
qed

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

lemma *validE*:
  X ∈ *valid* ⟹ (⋀ *x A B M*. *X* = *implem* (*Chan x A B M*) ⟹ *M* ∈ *payload* ⟹ *P*) ⟹ *P*
by (*auto simp add*: *valid-def*)

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 −
from *assms* **have** $(\bigwedge x\ A\ B\ M.\ X = implem\ (Chan\ x\ A\ B\ M) \Longrightarrow M \in payload \Longrightarrow P) \Longrightarrow P$
by (*auto elim*: *validE*)
moreover from *assms* **have** $\bigwedge x\ A\ B\ M.\ X = implem\ (Chan\ x\ A\ B\ M) \Longrightarrow M \in payload \Longrightarrow P$
proof −
fix $x\ A\ B\ M$
assume $X = implem\ (Chan\ x\ A\ B\ M)\ M \in payload$
with *assms* **show** P **by** (*cases* x, *auto*)
qed
ultimately show *?thesis* **.**
qed

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*:
$implem\ (Chan\ x\ A\ B\ M) = implem\ (Chan\ x'\ A'\ B'\ M')$
$\longleftrightarrow x = x' \land A = A' \land B = B' \land M = M'$
— implementations and parts
and *parts-impl-inj*:
$M' \in payload \Longrightarrow$
$implem\ (Chan\ x\ A\ B\ M) \in parts\ \{implem\ (Chan\ x'\ A'\ B'\ M')\} \Longrightarrow$
$x = x' \land A = A' \land B = B' \land M = M'$
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*:
$[\![G \subseteq payload; Enc\text{-}keys\text{-}clean\ H]\!]$
$\Longrightarrow analz\ (implAuthSet\ G \cup H) \subseteq synth\ (analz\ (G \cup H)) \cup -payload$
and *analz-Un-implSecureSet*:
$[\![G \subseteq payload; Enc\text{-}keys\text{-}clean\ H]\!]$
$\Longrightarrow analz\ (implSecureSet\ Ag\ G \cup H) \subseteq synth\ (analz\ (G \cup H)) \cup -payload$
and *analz-Un-implSecureSet-2*:
$[\![G \subseteq payload; Enc\text{-}keys\text{-}clean\ H; Ag \cap broken\ (parts\ H \cap range\ LtK) = \{\}]\!]$
$\Longrightarrow analz\ (implSecureSet\ Ag\ G \cup H) \subseteq synth\ (analz\ H) \cup -payload$

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*: $X = Y \Longrightarrow composed\ X \longleftrightarrow composed\ Y$
by (*rule arg-cong*)

lemma *implem-Tags-aux*: $implem\ (Chan\ x\ A\ B\ M) \notin Tags$ **by** (*cases x*, *auto dest*: *composed-arg-cong*)
lemma *implem-Tags* [*simp*]: $implem\ x \notin Tags$ **by** (*cases x*, *auto simp add*: *implem-Tags-aux*)
lemma *implem-LtK-aux*: $implem\ (Chan\ x\ A\ B\ M) \neq LtK\ K$ **by** (*cases x*, *auto dest*: *composed-arg-cong*)
lemma *implem-LtK* [*simp*]: $implem\ x \neq LtK\ K$ **by** (*cases x*, *auto simp add*: *implem-LtK-aux*)
lemma *implem-LtK2* [*simp*]: $implem\ x \notin range\ LtK$ **by** (*cases x*, *auto simp add*: *implem-LtK-aux*)

declare *impl-Impl* [*simp*]

lemma *LtK-parts-impl-insert*:
$LtK\ K \in parts\ (insert\ (implem\ (Chan\ x\ A\ B\ M))\ S) \Longrightarrow M \in payload \Longrightarrow LtK\ K \in parts\ S$
apply (*simp add*: *parts-insert* [*of - S*], *clarify*)
apply (*auto dest*: *validI LtK-parts-impl*)
done

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

lemma *valid-composed* [*simp*,*dest*]: $M \in valid \Longrightarrow composed\ M$
by (*auto elim*: *validE*)

— lemmas: valid/payload are mutually exclusive

lemma *valid-payload* [*dest*]: ⟦ $X \in$ *valid*; $X \in$ *payload* ⟧ $\Longrightarrow P$
by (*auto elim*!: *validE*)

— valid/LtK are mutually exclusive
lemma *valid-isLtKey* [*dest*]: ⟦ $X \in$ *valid*; $X \in$ *range LtK* ⟧ $\Longrightarrow P$
by (*auto*)

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*) $\in H$
apply (*drule analz-into-parts*,
drule parts-monotone [*of - H payload* $\cup$ $H \cap$ *valid* $\cup$ *range LtK* $\cup$ *Tags*], *auto*)
apply (*drule parts-singleton*, *auto elim*!:*validE dest*: *parts-impl-inj*)
done

lemma *parts-valid-LtKeys-disjoint*:
$I \subseteq$ *valid* $\Longrightarrow$ *parts I* $\cap$ *range LtK* = {}
apply (*safe*, *drule parts-singleton*, *clarsimp*)
apply (*auto dest*: *subsetD LtK-parts-impl*)
done

lemma *analz-LtKeys*:
$H \subseteq$ *payload* $\cup$ *valid* $\cup$ *range LtK* $\cup$ *Tags* $\Longrightarrow$
analz H $\cap$ *range LtK* $\subseteq H$
apply *auto*
apply (*drule analz-into-parts*, *drule parts-monotone* [*of - H payload* $\cup$ *valid* $\cup$ $H \cap$ *range LtK* $\cup$ *Tags*], *auto*)
apply (*drule parts-singleton*, *auto elim*!:*validE dest*: *parts-impl-inj*)
done

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 $\in H \lor M \in$ *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 $\in H \lor M \in$ *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*]:
  $[\![X \in abs\ H;$
    $\bigwedge x\ A\ B\ M.\ X = Chan\ x\ A\ B\ M \Longrightarrow M \in payload \Longrightarrow implem\ X \in H \Longrightarrow P]\!]$
  $\Longrightarrow P$
by (*auto simp add*: *abs-def*)

lemma *absI* [*intro*]: $M \in payload \Longrightarrow implem\ (Chan\ x\ A\ B\ M) \in H \Longrightarrow Chan\ x\ A\ B\ M \in abs\ H$
by (*auto simp add*: *abs-def*)

lemma *abs-mono*: $G \subseteq H \Longrightarrow abs\ G \subseteq abs\ H$
by (*auto simp add*: *abs-def*)

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

lemma *abs-empty* [*simp*]: $abs\ \{\} = \{\}$
by (*auto simp add*: *abs-def*)

lemma *abs-Un-eq*: $abs\ (G \cup H) = abs\ G \cup abs\ H$
by (*auto simp add*: *abs-def*)

General lemmas about implementations and *local.abs*.

lemma *abs-insert-payload* [*simp*]: $M \in payload \Longrightarrow abs\ (insert\ M\ S) = abs\ S$
by (*auto simp add*: *abs-def*)

lemma *abs-insert-impl* [*simp*]:
  $M \in payload \Longrightarrow abs\ (insert\ (implem\ (Chan\ x\ A\ B\ M))\ S) = insert\ (Chan\ x\ A\ B\ M)\ (abs\ S)$
by (*auto simp add*: *abs-def*)

lemma *extr-payload* [*simp*, *intro*]:
  $[\![X \in extr\ Bad\ NI\ (abs\ I);\ NI \subseteq payload]\!] \Longrightarrow X \in payload$
by (*erule extr.induct*, *blast*, *auto*)

lemma *abs-Un-LtK*:
  $K \subseteq range\ LtK \Longrightarrow abs\ (K \cup S) = abs\ S$
by (*auto simp add*: *abs-Un-eq*)

lemma *abs-Un-keys-of* [*simp*]:
  $abs\ (keys\text{-}of\ A \cup S) = abs\ S$
by (*auto intro*!: *abs-Un-LtK*)

Lemmas about *valid* and *local.abs*

lemma *abs-validSet*: $abs\ (S \cap valid) = abs\ S$
by (*auto elim*: *absE intro*: *validI*)

lemma *valid-abs*: $M \in valid \Longrightarrow \exists\ M'.\ M' \in abs\ \{M\}$
by (*auto elim*: *validE*)

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*: $extractable\ K\ I \subseteq I$
by (*auto simp add*: *extractable-def*)

lemma *extractableI*:
  $implem\ (Chan\ x\ A\ B\ M) \in I \Longrightarrow$
   $x = insec \vee x = auth \vee ((x = confid \vee x = secure) \wedge (A,\ B) \in broken\ K) \Longrightarrow$
   $implem\ (Chan\ x\ A\ B\ M) \in extractable\ K\ I$
by (*auto simp add*: *extractable-def*)

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
by (*auto simp add*: *extractable-def brokenI*)

General lemmas about implementations and extractable.

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

by (*erule extractableE*, *auto*)

Auxiliary lemmas about extractable messages: they are implementations.

lemma *valid-extractable* [*simp*]: $I \subseteq valid \Longrightarrow extractable\ K\ I \subseteq valid$
by (*auto intro*: *subset-trans extractable-red del*: *subsetI*)

lemma *LtKeys-parts-extractable*:
$I \subseteq valid \Longrightarrow parts\ (extractable\ K\ I) \cap range\ LtK = \{\}$
by (*auto dest*: *valid-extractable intro*!: *parts-valid-LtKeys-disjoint*)

lemma *LtKeys-parts-extractable-elt* [*simp*]:
$I \subseteq valid \Longrightarrow LtK\ ltk \notin parts\ (extractable\ K\ I)$
by (*blast dest*: *LtKeys-parts-extractable*)

lemma *LtKeys-parts-implSecureSet*:
$parts\ (implSecureSet\ Ag\ payload) \cap range\ LtK = \{\}$
by (*auto intro*!: *parts-valid-LtKeys-disjoint intro*: *validI*)

lemma *LtKeys-parts-implSecureSet-elt*:
$LtK\ K \notin parts\ (implSecureSet\ Ag\ payload)$
using *LtKeys-parts-implSecureSet*
by *auto*

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*:
$[\![\ NI \subseteq payload;\ I \subseteq valid\ ]\!] \Longrightarrow$
$I \subseteq extractable\ K\ I\ \cup$
$implConfidSet\ (UNIV - broken\ K)\ payload\ \cup$
$implSecureSet\ (UNIV - broken\ K)\ payload$
by (*auto dest*!: *subsetD* [**where** *A*=*I*] *elim*!: *valid-cases intro*: *extractableI*)

13.2.2 Partition of *extractable*

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

lemma *extractable-partition*:
$[\![Keys\text{-}bad\ K\ Bad;\ NI \subseteq payload;\ I \subseteq valid]\!] \Longrightarrow$
$extractable\ K\ I \subseteq$
$implInsecSet\ (extr\ Bad\ NI\ (abs\ I))\ \cup$
$implConfidSet\ UNIV\ (extr\ Bad\ NI\ (abs\ I))\ \cup$
$implAuthSet\ (extr\ Bad\ NI\ (abs\ I))\ \cup$
$implSecureSet\ UNIV\ (extr\ Bad\ NI\ (abs\ I))$
apply (*rule*, *frule valid-extractable*, *drule subsetD* [**where** *A*=*extractable K I*], *fast*)
apply (*erule valid-cases*, *auto*)
done

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 −
 from HNI HI
 have analz (NI ∪ I ∪ K ∪ Tags) ⊆
       analz (NI ∪ (extractable K I ∪
                    implConfidSet (UNIV − broken K) payload ∪
                    implSecureSet (UNIV − broken K) payload)
              ∪ K ∪ Tags)
   by (intro analz-mono Un-mono impl-partition, simp-all)
 also have ... ⊆ analz (implConfidSet (UNIV − broken K) payload ∪
                 (implSecureSet (UNIV − broken K) payload ∪
                 (extractable K I ∪ NI ∪ K ∪ Tags)))
   by (auto)
 also have ... ⊆ synth (analz (implSecureSet (UNIV − broken K) payload ∪
               (extractable K I ∪ NI ∪ K ∪ Tags))) ∪ −payload
   proof (rule analz-Un-implConfidSet-2)
     show Enc-keys-clean (implSecureSet (UNIV − broken K) payload
                          ∪ (extractable K I ∪ NI ∪ K ∪ Tags))
       by (auto simp add: HNI HI HK intro: validI)
   next
     from HK HI HNI
     show (UNIV − broken K) ∩
          broken (parts (
            implSecureSet (UNIV − broken K) payload ∪
            (extractable K I ∪ NI ∪ K ∪ Tags)) ∩ range LtK) = {}
       by (auto simp add: LtKeys-parts
            LtKeys-parts-implSecureSet-elt [where Ag=− broken K, simplified])
   qed (auto)
 also have ... ⊆ synth (analz (extractable K I ∪ NI ∪ K ∪ Tags)) ∪ −payload
   proof (rule Un-least, rule synth-idem-payload)
     show analz (implSecureSet (UNIV − broken K) payload ∪
               (extractable K I ∪ NI ∪ K ∪ Tags))
           ⊆ synth (analz (extractable K I ∪ NI ∪ K ∪ Tags)) ∪ −payload
       proof (rule analz-Un-implSecureSet-2)
         show Enc-keys-clean (extractable K I ∪ NI ∪ K ∪ Tags)
           using HNI HK HI by auto
       next
         from HI HK HNI
```

show (*UNIV* − *broken K*) ∩
broken (*parts* (*extractable K I* ∪ *NI* ∪ *K* ∪ *Tags*) ∩ *range LtK*) = {}
by (*auto simp add*: *LtKeys-parts*)
qed (*auto*)
next
show −*payload* ⊆ *synth* (*analz* (*extractable K I* ∪ *NI* ∪ *K* ∪ *Tags*)) ∪ −*payload*
by *auto*
qed
also have ... ⊆ *synth* (*analz* (*NI* ∪ *extractable K I* ∪ *K* ∪ *Tags*)) ∪ −*payload*
by (*simp add*: *sup.left-commute sup-commute*)
finally show *?thesis* **.**
qed

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 −
from *HNI HI Hbad*
have *analz* (*NI* ∪ *extractable K I* ∪ *K* ∪ *Tags*) ⊆
analz (*NI* ∪ (*implInsecSet* (*extr Bad NI* (*abs I*)) ∪
implConfidSet UNIV (*extr Bad NI* (*abs I*)) ∪
implAuthSet (*extr Bad NI* (*abs I*)) ∪
implSecureSet UNIV (*extr Bad NI* (*abs I*))) ∪
K ∪ *Tags*)
by (*intro analz-mono Un-mono extractable-partition*) (*auto*)

also have ... ⊆ *analz* (*implInsecSet* (*extr Bad NI* (*abs I*)) ∪
(*implConfidSet UNIV* (*extr Bad NI* (*abs I*)) ∪
(*implAuthSet* (*extr Bad NI* (*abs I*)) ∪
(*implSecureSet UNIV* (*extr Bad NI* (*abs I*)) ∪
(*NI* ∪ *K* ∪ *Tags*)))))
by (*auto*)
also have ... ⊆ *synth* (*analz* (*extr Bad NI* (*abs I*) ∪
(*implConfidSet UNIV* (*extr Bad NI* (*abs I*)) ∪
(*implAuthSet* (*extr Bad NI* (*abs I*)) ∪
(*implSecureSet UNIV* (*extr Bad NI* (*abs I*)) ∪ (*NI* ∪ *K* ∪ *Tags*))))))
∪ −*payload*
by (*rule analz-Un-implInsecSet*)
(*auto simp only*: *Un-commute* [*of extr - - - -*] *Un-assoc Un-absorb*,
auto simp add: *HNI HK HI intro*!: *validI*)
also have ... ⊆ *synth* (*analz* (*extr Bad NI* (*abs I*) ∪
(*implAuthSet* (*extr Bad NI* (*abs I*)) ∪
(*implSecureSet UNIV* (*extr Bad NI* (*abs I*)) ∪ (*NI* ∪ *K* ∪ *Tags*)))))
∪ −*payload*
proof (*rule Un-least*, *rule synth-idem-payload*)
have *analz* (*implConfidSet UNIV* (*extr Bad NI* (*abs I*)) ∪
(*implAuthSet* (*extr Bad NI* (*abs I*)) ∪

(*implSecureSet UNIV* (*extr Bad NI* (*abs I*)) ∪
(*NI* ∪ (*K* ∪ *extr Bad NI* (*abs I*) ∪ *Tags*)))))
⊆ *synth* (*analz* (*extr Bad NI* (*abs I*) ∪
(*implAuthSet* (*extr Bad NI* (*abs I*)) ∪
(*implSecureSet UNIV* (*extr Bad NI* (*abs I*)) ∪
(*NI* ∪ (*K* ∪ *extr Bad NI* (*abs I*) ∪ *Tags*))))))
∪ −*payload*
by (*rule analz-Un-implConfidSet*)
(*auto simp only*: *Un-commute* [*of extr* - - - -] *Un-assoc Un-absorb*,
auto simp add: *HK HI HNI intro*!: *validI*)
then show *analz* (*extr Bad NI* (*abs I*) ∪
(*implConfidSet UNIV* (*extr Bad NI* (*abs I*)) ∪
(*implAuthSet* (*extr Bad NI* (*abs I*)) ∪
(*implSecureSet UNIV* (*extr Bad NI* (*abs I*)) ∪ (*NI* ∪ *K* ∪ *Tags*)))))
⊆ *synth* (*analz* (*extr Bad NI* (*abs I*) ∪
(*implAuthSet* (*extr Bad NI* (*abs I*)) ∪
(*implSecureSet UNIV* (*extr Bad NI* (*abs I*)) ∪
(*NI* ∪ *K* ∪ *Tags*)))))
∪ −*payload*
by (*simp add*: *inf-sup-aci*(*6*) *inf-sup-aci*(*7*))
next
show −*payload*
⊆ *synth* (*analz* (*extr Bad NI* (*abs I*) ∪
(*implAuthSet* (*extr Bad NI* (*abs I*)) ∪
(*implSecureSet UNIV* (*extr Bad NI* (*abs I*)) ∪ (*NI* ∪ *K* ∪ *Tags*)))))
∪ −*payload*
by *blast*
qed
also have ... ⊆ *synth* (*analz* (*extr Bad NI* (*abs I*) ∪
(*implSecureSet UNIV* (*extr Bad NI* (*abs I*)) ∪ (*NI* ∪ *K* ∪ *Tags*))))
∪ −*payload*
proof (*rule Un-least*, *rule synth-idem-payload*)
have *analz* (*implAuthSet* (*extr Bad NI* (*abs I*)) ∪
(*implSecureSet UNIV* (*extr Bad NI* (*abs I*)) ∪
(*NI* ∪ (*K* ∪ (*extr Bad NI* (*abs I*) ∪ *Tags*)))))
⊆ *synth* (*analz* (*extr Bad NI* (*abs I*) ∪
(*implSecureSet UNIV* (*extr Bad NI* (*abs I*)) ∪
(*NI* ∪ (*K* ∪ (*extr Bad NI* (*abs I*) ∪ *Tags*))))))
∪ −*payload*
by (*rule analz-Un-implAuthSet*)
(*auto simp only*: *Un-commute* [*of extr* - - - -] *Un-assoc Un-absorb*,
auto simp add: *HI HNI HK intro*!: *validI*)
then show *analz* (*extr Bad NI* (*abs I*) ∪
(*implAuthSet* (*extr Bad NI* (*abs I*)) ∪
(*implSecureSet UNIV* (*extr Bad NI* (*abs I*)) ∪ (*NI* ∪ *K* ∪ *Tags*))))
⊆ *synth* (*analz* (*extr Bad NI* (*abs I*) ∪
(*implSecureSet UNIV* (*extr Bad NI* (*abs I*)) ∪
(*NI* ∪ *K* ∪ *Tags*))))
∪ −*payload*
by (*simp add*: *inf-sup-aci*(*6*) *inf-sup-aci*(*7*))
next
show −*payload*
⊆ *synth* (*analz* (*extr Bad NI* (*abs I*) ∪

(*implSecureSet UNIV* (*extr Bad NI* (*abs I*))
∪ (*NI* ∪ *K* ∪ *Tags*))))
∪ −*payload*
by *blast*
qed
also have ... ⊆ *synth* (*analz* (*extr Bad NI* (*abs I*) ∪ (*NI* ∪ *K* ∪ *Tags*)))
∪ −*payload*
proof (*rule Un-least*, *rule synth-idem-payload*)
have *analz* (*implSecureSet UNIV* (*extr Bad NI* (*abs I*)) ∪
(*NI* ∪ (*K* ∪ (*extr Bad NI* (*abs I*) ∪ *Tags*))))
⊆ *synth* (*analz* (*extr Bad NI* (*abs I*) ∪
(*NI* ∪ (*K* ∪ (*extr Bad NI* (*abs I*) ∪ *Tags*)))))
∪ −*payload*
by (*rule analz-Un-implSecureSet*)
(*auto simp only*: *Un-commute* [*of extr - - - -*] *Un-assoc Un-absorb*,
auto simp add: *HI HNI HK intro*!: *validI*)
then show *analz* (*extr Bad NI* (*abs I*) ∪
(*implSecureSet UNIV* (*extr Bad NI* (*abs I*)) ∪ (*NI* ∪ *K* ∪ *Tags*)))
⊆ *synth* (*analz* (*extr Bad NI* (*abs I*) ∪ (*NI* ∪ *K* ∪ *Tags*)))
∪ −*payload*
by (*simp add*: *inf-sup-aci*(*6*) *inf-sup-aci*(*7*))
next
show −*payload*
⊆ *synth* (*analz* (*extr Bad NI* (*abs I*)∪ (*NI* ∪ *K* ∪ *Tags*)))
∪ −*payload*
by *blast*
qed
also have ... ⊆ *synth* (*analz* (*extr Bad NI* (*abs I*) ∪ *K* ∪ *Tags*)) ∪ −*payload*
by (*metis IK-subset-extr inf-sup-aci*(*6*) *set-eq-subset sup.absorb1*)
finally show *?thesis* **.**
qed

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
fix *X*
assume *H*: *X* ∈ *analz* (*NI* ∪ *K* ∪ *Tags*)
thus *X* ∈ *analz NI* ∪ *K* ∪ *Tags*
proof (*induction X rule*: *analz.induct*)
case (*Dec X Y*)
hence *Enc X Y* ∈ *payload* **using** *assms* **by** *auto*
moreover
from *Dec.IH*(*2*) **have** *Y* ∈ *synth* (*analz NI* ∪ (*K* ∪ *Tags*))
by (*auto simp add*: *Collect-disj-eq dest*!: *synth-Int2*)
ultimately show *?case* **using** *Dec.IH*(*1*) *assms*(*2*)
by (*auto dest*!: *synth-payload* [*THEN* [*2*] *rev-subsetD*])
next
case (*Adec-lt X Y*)
hence *Aenc X* (*pubK Y*) ∈ *payload* ∪ *range LtK* ∪ *Tags* **using** *assms*
by *auto*

then show *?case* **by** *auto*
next
case (*Sign-getmsg X Y*)
hence *Sign X* (*priK Y*) ∈ *payload* ∪ *range LtK* ∪ *Tags* **using** *assms* **by** *auto*
then show *?case* **by** *auto*
next
case (*Adec-eph X Y*)
then show *?case* **using** *assms* **by** (*auto dest*!: *EpriK-synth*)
qed (*insert assms*, *auto*)
qed

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*
by (*rule analz-LtKeys-Tag*) *auto*

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 −
from *HNI HK HI* **have** *analz* (*NI* ∪ *I* ∪ *K* ∪ *Tags*) ⊆
synth (*analz* (*NI* ∪ *extractable K I* ∪ *K* ∪ *Tags*)) ∪ −*payload*
by (*rule analz-NI-I-K-analz-NI-EI*)
also have ... ⊆ *synth* (*analz* (*extr Bad NI* (*abs I*) ∪ *K* ∪ *Tags*)) ∪ −*payload*
proof (*rule Un-least*, *simp-all*)
from *Hbad HNI HK HI* **have** *analz* (*NI* ∪ *extractable K I* ∪ *K* ∪ *Tags*) ⊆
synth (*analz* (*extr Bad NI* (*abs I*) ∪ *K* ∪ *Tags*)) ∪ −*payload*
by (*intro analz-NI-EI-K-synth-analz-NI-E-K*)
then show *synth* (*analz* (*NI* ∪ *extractable K I* ∪ *K* ∪ *Tags*)) ⊆
synth (*analz* (*extr Bad NI* (*abs I*) ∪ *K* ∪ *Tags*)) ∪ −*payload*
by (*rule synth-idem-payload*)
qed
also have ... ⊆ *synth* (*analz* (*extr Bad NI* (*abs I*))) ∪ −*payload*
proof (*rule Un-least*, *simp-all*)
from *HNI HK HI* **have** *analz* (*extr Bad NI* (*abs I*) ∪ *K* ∪ *Tags*) ⊆
analz (*extr Bad NI* (*abs I*)) ∪ *K* ∪ *Tags*
by (*rule analz-NI-E-K-analz-NI-E*)
also from *HK* **have** ... ⊆ *analz* (*extr Bad NI* (*abs I*)) ∪ −*payload*
by *auto*
also have ... ⊆ *synth* (*analz* (*extr Bad NI* (*abs I*))) ∪ −*payload*
by *auto*
finally show *synth* (*analz* (*extr Bad NI* (*abs I*) ∪ *K* ∪ *Tags*)) ⊆
synth (*analz* (*extr Bad NI* (*abs I*))) ∪ −*payload*
by (*rule synth-idem-payload*)
qed
finally show *?thesis* **.**

qed

Lemma actually used to prove the refinement.

lemma *synth-analz-NI-I-K-synth-analz-NI-E*:
$[\![$ *Keys-bad K Bad*; *NI* $\subseteq$ *payload*; *K* $\subseteq$ *range LtK*; *I* $\subseteq$ *valid*$]\!]$
$\Longrightarrow$ *synth* (*analz* (*NI* $\cup$ *I* $\cup$ *K* $\cup$ *Tags*))
$\subseteq$ *synth* (*analz* (*extr Bad NI* (*abs I*))) $\cup$ $-$*payload*
by (*intro synth-idem-payload analz-NI-I-K-synth-analz-NI-E*) (*assumption+*)

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 $-$
assume *H* $\subseteq$ *payload* $\cup$ *valid* $\cup$ *range LtK*
then have *HH*:*H* = (*H* $\cap$ *payload*) $\cup$ (*H* $\cap$ *valid*) $\cup$ (*H* $\cap$ *range LtK*)
by *auto*
assume *HA*:
analz S $\subseteq$ *synth* (*analz* ((*S* $\cap$ *payload*) $\cup$ (*S* $\cap$ *valid*) $\cup$ (*S* $\cap$ *range LtK*) $\cup$ *Tags*))
then have
analz (*H* $\cup$ *S*) $\subseteq$
synth (*analz* (*H* $\cup$ ((*S* $\cap$ *payload*) $\cup$ (*S* $\cap$ *valid*) $\cup$ (*S* $\cap$ *range LtK*) $\cup$ *Tags*)))
by (*rule analz-synth-subset-Un2*)
also with *HH* **have**
... $\subseteq$ *synth* (*analz* (((*H* $\cap$ *payload*) $\cup$ (*H* $\cap$ *valid*) $\cup$ (*H* $\cap$ *range LtK*)) $\cup$
((*S* $\cap$ *payload*) $\cup$ (*S* $\cap$ *valid*) $\cup$ (*S* $\cap$ *range LtK*) $\cup$ *Tags*)))
by *auto*
also have ... = *synth* (*analz* (((*H* $\cup$ *S*) $\cap$ *payload*) $\cup$ ((*H* $\cup$ *S*) $\cap$ *valid*) $\cup$
((*H* $\cup$ *S*) $\cap$ *range LtK*) $\cup$ *Tags*))
by (*simp add*: *Un-left-commute sup.commute Int-Un-distrib2*)
finally show *?thesis* **.**
qed

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*))
by (*simp only*: *insert-is-Un* [*of x S*], *erule analz-Un-partition*, *auto*)

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*
done

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*
by *auto*

lemma *implAuthSet-implAuthSet-aux*: *sym.implAuthSet G* ⊆ *implAuthSet-aux G*
by (*auto*, *auto*)

lemma *implConfidSet-implConfidSet-aux*: *sym.implConfidSet Ag G* ⊆ *implConfidSet-aux Ag G*
by (*auto*)

lemma *implSecureSet-implSecureSet-aux*: *sym.implSecureSet Ag G* ⊆ *implSecureSet-aux Ag G*
by (*auto*)

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 −
  assume H:Enc-keys-clean (G ∪ H)
  have analz (implInsecSet-aux G ∪ H) ⊆ implInsecSet-aux G ∪
        analz (Tags ∪ PairSet (range Agent) (PairSet (range Agent) G) ∪ H)
    by (rule analz-Un-PairSet)
  also have ... = implInsecSet-aux G ∪
          analz (Tags ∪ (PairSet (range Agent) (PairSet (range Agent) G) ∪ H))
    by (simp only: Un-assoc)
  also have ... ⊆ implInsecSet-aux G ∪
        (Tags ∪ analz (PairSet (range Agent) (PairSet (range Agent) G) ∪ H))
    by (rule Un-mono, blast, rule analz-Un-Tag, blast intro: H)
  also have ... = implInsecSet-aux G ∪ Tags ∪
                analz (PairSet (range Agent) (PairSet (range Agent) G) ∪ H)
    by auto
  also have ... ⊆ implInsecSet-aux G ∪ Tags ∪ (PairSet (range Agent) (PairSet (range Agent) G) ∪
                analz (range Agent ∪ PairSet (range Agent) G ∪ H))
    by (rule Un-mono, blast, rule analz-Un-PairSet)
  also have ... = implInsecSet-aux G ∪ Tags ∪ PairSet (range Agent) (PairSet (range Agent) G) ∪
                analz (PairSet (range Agent) G ∪ (range Agent ∪ H))
    by (auto simp add: Un-assoc Un-commute)
  also have ... ⊆ implInsecSet-aux G ∪ Tags ∪ PairSet (range Agent) (PairSet (range Agent) G) ∪
                (PairSet (range Agent) G ∪ analz (range Agent ∪ G ∪ (range Agent ∪ H)))
    by (rule Un-mono, blast, rule analz-Un-PairSet)
  also have ... = implInsecSet-aux G ∪ Tags ∪ (PairSet (range Agent) (PairSet (range Agent) G) ∪
                PairSet (range Agent) G) ∪ analz (range Agent ∪ G ∪ (range Agent ∪ H))
```

```
    by (simp only: Un-assoc Un-commute)
   finally show ?thesis by auto
qed

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 −
  assume H:Enc-keys-clean (G ∪ H)
  have HH:PairSet (range Agent) (PairSet (range Agent) G) ∪
                PairSet (range Agent) G ⊆ synth (analz (G ∪ H))
    by auto
  have HHH:analz (range Agent ∪ G ∪ (range Agent ∪ H)) ⊆ synth (analz (G ∪ H))
    proof −
      have analz (range Agent ∪ G ∪ (range Agent ∪ H)) ⊆
            synth (analz (range Agent ∪ G ∪ (range Agent ∪ H)))
        by auto
      also have ... = synth (analz (synth (range Agent ∪ G ∪ (range Agent ∪ H)))) by auto
      also have ... ⊆ synth (analz (synth (G ∪ H)))
        proof (rule synth-analz-mono)
          have range Agent ∪ G ∪ (range Agent ∪ H) ⊆ synth (G ∪ H) by auto
          then have synth (range Agent ∪ G ∪ (range Agent ∪ H)) ⊆ synth (synth (G ∪ H))
            by (rule synth-mono)
          then show synth (range Agent ∪ G ∪ (range Agent ∪ H)) ⊆ synth (G ∪ H) by auto
        qed
      also have ... = synth (analz (G ∪ H)) by auto
      finally show ?thesis .
    qed
  from H have
   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))
    by (rule analz-Un-implInsecSet-aux-1)
  also have ... = implInsecSet-aux G ∪ Tags ∪
                 (PairSet (range Agent) (PairSet (range Agent) G) ∪
                 PairSet (range Agent) G) ∪ analz (range Agent ∪ G ∪ (range Agent ∪ H))
    by (simp only: Un-assoc Un-commute)
  also have ... ⊆ implInsecSet-aux G ∪ Tags ∪ synth (analz (G ∪ H)) ∪
                 synth (analz (G ∪ H))
     by ((rule Un-mono)+, auto simp add: HH HHH)
  finally show ?thesis by auto
qed

lemma analz-Un-implInsecSet-aux-3:
  Enc-keys-clean (G ∪ H) ⟹
   analz (implInsecSet-aux G ∪ H) ⊆ synth (analz (G ∪ H)) ∪ −payload
by (rule subset-trans [OF analz-Un-implInsecSet-aux-2], auto)

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

apply (*rule subset-trans* [*of* - *analz* (*implInsecSet-aux G* ∪ *H*) -])
apply (*rule analz-mono*, *rule Un-mono*, *blast intro*!: *implSet-implSet-aux*, *simp*)
using *analz-Un-implInsecSet-aux-3* **apply** *blast*
done

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* ∪ *H*)
proof −
  assume *H*:*Enc-keys-clean* (*G* ∪ *H*)
  have *analz* (*implConfidSet-aux Ag G* ∪ *H*) ⊆
        implConfidSet-aux Ag G ∪ *analz* (*PairSet Tags G* ∪ *H*)
    by (*rule analz-Un-EncSet*, *fast*, *blast intro*: *H*)
  also have ... ⊆ *implConfidSet-aux Ag G* ∪ (*PairSet Tags G* ∪ *analz* (*Tags* ∪ *G* ∪ *H*))
    by (*rule Un-mono*, *blast*, *rule analz-Un-PairSet*)
  also have ... = *implConfidSet-aux Ag G* ∪ *PairSet Tags G* ∪ *analz* (*Tags* ∪ (*G* ∪ *H*))
    by (*simp only*: *Un-assoc*)
  also have ... ⊆ *implConfidSet-aux Ag G* ∪ *PairSet Tags G* ∪ (*Tags* ∪ *analz* (*G* ∪ *H*))
    by (*rule Un-mono*, *blast*, *rule analz-Un-Tag*, *blast intro*: *H*)
   finally show *?thesis* **by** *auto*
qed

lemma *analz-Un-implConfidSet-aux-2*:
  Enc-keys-clean (*G* ∪ *H*) ⟹
   analz (*implConfidSet-aux Ag G* ∪ *H*) ⊆
     implConfidSet-aux Ag G ∪ *PairSet Tags G* ∪ *Tags* ∪
     synth (*analz* (*G* ∪ *H*))
proof −
  assume *H*:*Enc-keys-clean* (*G* ∪ *H*)
  from *H* **have** *analz* (*implConfidSet-aux Ag G* ∪ *H*) ⊆
              implConfidSet-aux Ag G ∪ *PairSet Tags G* ∪ *Tags* ∪ *analz* (*G* ∪ *H*)
    by (*rule analz-Un-implConfidSet-aux-1*)
  also have ... ⊆ *implConfidSet-aux Ag G* ∪ *PairSet Tags G* ∪ *Tags* ∪ *synth* (*analz* (*G* ∪ *H*))
     by *auto*
  finally show *?thesis* **by** *auto*
qed

lemma *analz-Un-implConfidSet-aux-3*:
  Enc-keys-clean (*G* ∪ *H*) ⟹
  analz (*implConfidSet-aux Ag G* ∪ *H*) ⊆ *synth* (*analz* (*G* ∪ *H*)) ∪ −*payload*
by (*rule subset-trans* [*OF analz-Un-implConfidSet-aux-2*], *auto*)

lemma *analz-Un-implConfidSet*:
  Enc-keys-clean (*G* ∪ *H*) ⟹
   analz (*sym.implConfidSet Ag G* ∪ *H*) ⊆ *synth* (*analz* (*G* ∪ *H*)) ∪ −*payload*
apply (*rule subset-trans* [*of* - *analz* (*implConfidSet-aux Ag G* ∪ *H*) -])
apply (*rule analz-mono*, *rule Un-mono*, *blast intro*!: *implSet-implSet-aux*, *simp*)
using *analz-Un-implConfidSet-aux-3* **apply** *blast*
done

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

lemma *analz-Un-implConfidSet-2-aux-1*:
  Enc-keys-clean H ⟹
   Ag ∩ *broken* (*parts H* ∩ *range LtK*) = {} ⟹
   analz (*implConfidSet-aux Ag G* ∪ *H*) ⊆ *implConfidSet-aux Ag G* ∪ *synth* (*analz H*)
apply (*rule subset-trans* [*OF analz-Un-EncSet2*], *simp*)
apply (*auto dest*:*analz-into-parts*)
done

lemma *analz-Un-implConfidSet-2-aux-3*:
  Enc-keys-clean H ⟹
   Ag ∩ *broken* (*parts H* ∩ *range LtK*) = {} ⟹
   analz (*implConfidSet-aux Ag G* ∪ *H*) ⊆ *synth* (*analz H*) ∪ −*payload*
by (*rule subset-trans* [*OF analz-Un-implConfidSet-2-aux-1*], *auto*)

lemma *analz-Un-implConfidSet-2*:
  Enc-keys-clean H ⟹
   Ag ∩ *broken* (*parts H* ∩ *range LtK*) = {} ⟹
   analz (*sym.implConfidSet Ag G* ∪ *H*) ⊆ *synth* (*analz H*) ∪ −*payload*
apply (*rule subset-trans* [*of - analz* (*implConfidSet-aux Ag G* ∪ *H*) -])
apply (*rule analz-mono*, *rule Un-mono*, *blast intro*!: *implSet-implSet-aux*, *simp*)
using *analz-Un-implConfidSet-2-aux-3* **apply** *auto*
done

14.4 Pull *implSecureSet* out of *analz*

lemma *analz-Un-implSecureSet-aux-1*:
  Enc-keys-clean (*G* ∪ *H*) ⟹
   analz (*implSecureSet-aux Ag G* ∪ *H*) ⊆
     implSecureSet-aux Ag G ∪ *PairSet Tags G* ∪ *Tags* ∪
     analz (*G* ∪ *H*)
proof −
  assume *H*:*Enc-keys-clean* (*G* ∪ *H*)
  have *analz* (*implSecureSet-aux Ag G* ∪ *H*) ⊆
       implSecureSet-aux Ag G ∪ *analz* (*PairSet Tags G* ∪ *H*)
    by (*rule analz-Un-EncSet*, *fast*, *blast intro*: *H*)
  also have ... ⊆ *implSecureSet-aux Ag G* ∪ (*PairSet Tags G* ∪ *analz* (*Tags* ∪ *G* ∪ *H*))
    by (*rule Un-mono*, *blast*, *rule analz-Un-PairSet*)
  also have ... = *implSecureSet-aux Ag G* ∪ *PairSet Tags G* ∪ *analz* (*Tags* ∪ (*G* ∪ *H*))
    by (*simp only*: *Un-assoc*)
  also have ... ⊆ *implSecureSet-aux Ag G* ∪ *PairSet Tags G* ∪ (*Tags* ∪ *analz* (*G* ∪ *H*))
    by (*rule Un-mono*, *blast*, *rule analz-Un-Tag*, *blast intro*: *H*)
   finally show *?thesis* **by** *auto*
qed

lemma *analz-Un-implSecureSet-aux-2*:
  Enc-keys-clean (*G* ∪ *H*) ⟹
   analz (*implSecureSet-aux Ag G* ∪ *H*) ⊆
     implSecureSet-aux Ag G ∪ *PairSet Tags G* ∪ *Tags* ∪
     synth (*analz* (*G* ∪ *H*))
proof −
  assume *H*:*Enc-keys-clean* (*G* ∪ *H*)
  from *H* **have** *analz* (*implSecureSet-aux Ag G* ∪ *H*) ⊆

implSecureSet-aux Ag G ∪ *PairSet Tags G* ∪ *Tags* ∪ *analz* (*G* ∪ *H*)
by (*rule analz-Un-implSecureSet-aux-1*)
also have ... ⊆ *implSecureSet-aux Ag G* ∪ *PairSet Tags G* ∪ *Tags* ∪ *synth* (*analz* (*G* ∪ *H*))
by *auto*
finally show *?thesis* **by** *auto*
qed

lemma *analz-Un-implSecureSet-aux-3*:
Enc-keys-clean (*G* ∪ *H*) ⟹
analz (*implSecureSet-aux Ag G* ∪ *H*) ⊆ *synth* (*analz* (*G* ∪ *H*)) ∪ −*payload*
by (*rule subset-trans* [*OF analz-Un-implSecureSet-aux-2*], *auto*)

lemma *analz-Un-implSecureSet*:
Enc-keys-clean (*G* ∪ *H*) ⟹
analz (*sym.implSecureSet Ag G* ∪ *H*) ⊆ *synth* (*analz* (*G* ∪ *H*)) ∪ −*payload*
apply (*rule subset-trans* [*of - analz* (*implSecureSet-aux Ag G* ∪ *H*) -])
apply (*rule analz-mono*, *rule Un-mono*, *blast intro*!: *implSet-implSet-aux*, *simp*)
using *analz-Un-implSecureSet-aux-3* **apply** *blast*
done

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

lemma *analz-Un-implSecureSet-2-aux-1*:
Enc-keys-clean H ⟹
Ag ∩ *broken* (*parts H* ∩ *range LtK*) = {} ⟹
analz (*implSecureSet-aux Ag G* ∪ *H*) ⊆ *implSecureSet-aux Ag G* ∪ *synth* (*analz H*)
apply (*rule subset-trans* [*OF analz-Un-EncSet2*], *simp*)
apply (*auto dest*:*analz-into-parts*)
done

lemma *analz-Un-implSecureSet-2-aux-3*:
Enc-keys-clean H ⟹
Ag ∩ *broken* (*parts H* ∩ *range LtK*) = {} ⟹
analz (*implSecureSet-aux Ag G* ∪ *H*) ⊆ *synth* (*analz H*) ∪ −*payload*
by (*rule subset-trans* [*OF analz-Un-implSecureSet-2-aux-1*], *auto*)

lemma *analz-Un-implSecureSet-2*:
Enc-keys-clean H ⟹
Ag ∩ *broken* (*parts H* ∩ *range LtK*) = {} ⟹
analz (*sym.implSecureSet Ag G* ∪ *H*) ⊆ *synth* (*analz H*) ∪ −*payload*
apply (*rule subset-trans* [*of - analz* (*implSecureSet-aux Ag G* ∪ *H*) -])
apply (*rule analz-mono*, *rule Un-mono*, *blast intro*!: *implSet-implSet-aux*, *simp*)
using *analz-Un-implSecureSet-2-aux-3* **apply** *auto*
done

14.5 Pull *implAuthSet* out of *analz*

lemma *analz-Un-implAuthSet-aux-1*:
Enc-keys-clean (*G* ∪ *H*) ⟹
analz (*implAuthSet-aux G* ∪ *H*) ⊆
implAuthSet-aux G ∪ *HashSet* (*PairSet* (*PairSet Tags G*) (*range* (*case-prod shrK*))) ∪
analz (*G* ∪ *H*)
proof −
assume *H*:*Enc-keys-clean* (*G* ∪ *H*)

have *analz* (*implAuthSet-aux G* ∪ *H*) ⊆ *implAuthSet-aux G* ∪
analz (*G* ∪ *HashSet* (*PairSet* (*PairSet Tags G*) (*range* (*case-prod shrK*))) ∪ *H*)
by (*rule analz-Un-PairSet*)
also have ... = *implAuthSet-aux G* ∪
analz (*HashSet* (*PairSet* (*PairSet Tags G*) (*range* (*case-prod shrK*))) ∪ *G* ∪ *H*)
by (*simp only*: *Un-assoc Un-commute*)
also have ... = *implAuthSet-aux G* ∪
analz (*HashSet* (*PairSet* (*PairSet Tags G*) (*range* (*case-prod shrK*))) ∪ (*G* ∪ *H*))
by (*simp only*: *Un-assoc*)
also have
... ⊆ *implAuthSet-aux G* ∪
(*HashSet* (*PairSet* (*PairSet Tags G*) (*range* (*case-prod shrK*))) ∪
analz (*G* ∪ *H*))
by (*rule Un-mono*, *blast*, *rule analz-Un-HashSet*, *blast intro*: *H*, *auto*)
also have ... = *implAuthSet-aux G* ∪
HashSet (*PairSet* (*PairSet Tags G*) (*range* (*case-prod shrK*))) ∪
analz (*G* ∪ *H*)
by *auto*
finally show *?thesis* **by** *auto*
qed

lemma *analz-Un-implAuthSet-aux-2*:
Enc-keys-clean (*G* ∪ *H*) ⟹
analz (*implAuthSet-aux G* ∪ *H*) ⊆
implAuthSet-aux G ∪ *HashSet* (*PairSet* (*PairSet Tags G*) (*range* (*case-prod shrK*))) ∪
synth (*analz* (*G* ∪ *H*))
proof −
assume *H*:*Enc-keys-clean* (*G* ∪ *H*)
from *H* **have**
analz (*implAuthSet-aux G* ∪ *H*) ⊆
implAuthSet-aux G ∪
HashSet (*PairSet* (*PairSet Tags G*) (*range* (*case-prod shrK*))) ∪
analz (*G* ∪ *H*)
by (*rule analz-Un-implAuthSet-aux-1*)
also have
... ⊆ *implAuthSet-aux G* ∪
HashSet (*PairSet* (*PairSet Tags G*) (*range* (*case-prod shrK*))) ∪
synth (*analz* (*G* ∪ *H*))
by *auto*
finally show *?thesis* **by** *auto*
qed

lemma *analz-Un-implAuthSet-aux-3*:
Enc-keys-clean (*G* ∪ *H*) ⟹
analz (*implAuthSet-aux G* ∪ *H*) ⊆ *synth* (*analz* (*G* ∪ *H*)) ∪ −*payload*
by (*rule subset-trans* [*OF analz-Un-implAuthSet-aux-2*], *auto*)

lemma *analz-Un-implAuthSet*:
Enc-keys-clean (*G* ∪ *H*) ⟹
analz (*sym.implAuthSet G* ∪ *H*) ⊆ *synth* (*analz* (*G* ∪ *H*)) ∪ −*payload*
apply (*rule subset-trans* [*of* - *analz* (*implAuthSet-aux G* ∪ *H*) -])
apply (*rule analz-mono*, *rule Un-mono*, *blast intro*!: *implSet-implSet-aux*, *simp*)
using *analz-Un-implAuthSet-aux-3* **apply** *blast*

done

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

14.6 Locale interpretations

```
interpretation sym: semivalid-implem implem-sym
proof (unfold-locales)
  fix x A B M x′ A′ B′ M′
  show implem-sym (Chan x A B M) = implem-sym (Chan x′ A′ B′ M′) ⟷
       x = x′ ∧ A = A′ ∧ B = B′ ∧ M = M′
    by (cases x, (cases x′, auto)+)
next
  fix M′ M x x′ A A′ B B′
  assume H: M′ ∈ payload
  then have A1: ⋀y. y ∈ parts {M′} ⟹ y ∈ payload
       and A2: ⋀y. M′ = y ⟹ y ∈ payload by auto
  assume implem-sym (Chan x A B M) ∈ parts {implem-sym (Chan x′ A′ B′ M′)}
  then show x = x′ ∧ A = A′ ∧ B = B′ ∧ M = M′
    by (cases x, (cases x′, auto dest!: A1 A2)+)
next
  fix I
  assume I ⊆ sym.valid
  then show Enc-keys-clean I
    proof (simp add: Enc-keys-clean-def, intro allI impI)
      fix X Y
      assume Enc X Y ∈ parts I
      obtain x A B M where M ∈ payload and Enc X Y ∈ parts {implem-sym (Chan x A B M)}
      using parts-singleton [OF ‹Enc X Y ∈ parts I›] ‹I ⊆ sym.valid›
        by (auto elim!: sym.validE)
      then show Y ∈ range LtK ∨ Y ∈ payload by (cases x, auto)
    qed
next
  fix Z
  show composed (implem-sym Z)
    proof (cases Z, simp)
      fix x A B M
      show composed (implem-sym (Chan x A B M)) by (cases x, auto)
    qed
next
  fix x A B M
  show implem-sym (Chan x A B M) ∉ payload
    by (cases x, auto)
next
  fix X K
  assume X ∈ sym.valid
  then obtain x A B M where M ∈ payload X = implem-sym (Chan x A B M)
    by (auto elim: sym.validE)
  then show LtK K ∉ parts {X}
  by (cases x, auto)

next
  fix G H
```

assume $G \subseteq$ *payload Enc-keys-clean H*
hence *Enc-keys-clean* $(G \cup H)$ **by** (*auto intro*: *Enc-keys-clean-Un*)
then show *analz* ($\{$*implem-sym* (*Insec A B M*) $|A\ B\ M.\ M \in G\} \cup H) \subseteq$
synth (*analz* $(G \cup H)) \cup -$ *payload*
by (*rule analz-Un-implInsecSet*)
next
fix *G H*
assume $G \subseteq$ *payload Enc-keys-clean H*
hence *Enc-keys-clean* $(G \cup H)$ **by** (*auto intro*: *Enc-keys-clean-Un*)
then show *analz* ($\{$*implem-sym* (*Auth A B M*) $|A\ B\ M.\ M \in G\} \cup H) \subseteq$
synth (*analz* $(G \cup H)) \cup -$ *payload*
by (*rule analz-Un-implAuthSet*)
next
fix *G H Ag*
assume $G \subseteq$ *payload Enc-keys-clean H*
hence *Enc-keys-clean* $(G \cup H)$ **by** (*auto intro*: *Enc-keys-clean-Un*)
then show *analz* ($\{$*implem-sym* (*Confid A B M*) $|A\ B\ M.\ (A, B) \in Ag \land M \in G\} \cup H) \subseteq$
synth (*analz* $(G \cup H)) \cup -$ *payload*
by (*rule analz-Un-implConfidSet*)
next
fix *G H Ag*
assume $G \subseteq$ *payload Enc-keys-clean H*
hence *Enc-keys-clean* $(G \cup H)$ **by** (*auto intro*: *Enc-keys-clean-Un*)
then show *analz* ($\{$*implem-sym* (*Secure A B M*) $|A\ B\ M.\ (A, B) \in Ag \land M \in G\} \cup H) \subseteq$
synth (*analz* $(G \cup H)) \cup -$ *payload*
by (*rule analz-Un-implSecureSet*)
next
fix *G H Ag*
assume *Enc-keys-clean H*
hence *Enc-keys-clean H* **by** *auto*
moreover assume $Ag \cap$ *broken* (*parts* $H \cap$ *range LtK*) $= \{\}$
ultimately show *analz* ($\{$*implem-sym* (*Confid A B M*) $|A\ B\ M.\ (A, B) \in Ag \land M \in G\} \cup H) \subseteq$
synth (*analz H*) $\cup -$ *payload*
by (*rule analz-Un-implConfidSet-2*)
next
fix *G H Ag*
assume *Enc-keys-clean H*
moreover assume $Ag \cap$ *broken* (*parts* $H \cap$ *range LtK*) $= \{\}$
ultimately show *analz* ($\{$*implem-sym* (*Secure A B M*) $|A\ B\ M.\ (A, B) \in Ag \land M \in G\} \cup H) \subseteq$
synth (*analz H*) $\cup -$ *payload*
by (*rule analz-Un-implSecureSet-2*)
qed

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 $\in H \lor M \in$ *synth* (*analz H*)
apply (*erule synth.cases*, *auto*)
done

lemma *implConfid-synth-analz*:

H ⊆ *payload* ∪ *sym.valid* ∪ *range LtK* ∪ *Tags* ⟹
sym.implConfid A B M ∈ *synth* (*analz H*) ⟹
sym.implConfid A B M ∈ *H* ∨ *M* ∈ *synth* (*analz H*)
apply (*erule synth.cases*, *auto*)
— 1 subgoal
apply (*frule sym.analz-valid* [**where** *x*=*confid*], *auto*)
done

lemma *implAuth-synth-analz*:
H ⊆ *payload* ∪ *sym.valid* ∪ *range LtK* ∪ *Tags* ⟹
sym.implAuth A B M ∈ *synth* (*analz H*) ⟹
sym.implAuth A B M ∈ *H* ∨ (*M* ∈ *synth* (*analz H*) ∧ (*A*, *B*) ∈ *broken H*)
using [[*simproc del*: *defined-all*]] **proof** (*erule synth.cases*, *simp-all*)
fix *X*
assume *H*: *H* ⊆ *payload* ∪ *sym.valid* ∪ *range LtK* ∪ *Tags*
assume *H'*:⟨*M*, *hmac* ⟨*AuthTag*, *M*⟩ (*shrK A B*)⟩ = *X* *X* ∈ *analz H*
hence *sym.implAuth A B M* ∈ *analz H* **by** *auto*
with *H* **have** *sym.implAuth A B M* ∈ *H* **by** (*rule sym.analz-valid*)
with *H'* **show** *X* ∈ *H* ∨ *M* ∈ *synth* (*analz H*) ∧ (*A*, *B*) ∈ *broken H*
by *auto*
next
fix *X Y*
assume *H*:*H* ⊆ *payload* ∪ *sym.valid* ∪ *range LtK* ∪ *Tags*
assume *H'*:*M* = *X* ∧ *hmac* ⟨*AuthTag*, *M*⟩ (*shrK A B*) = *Y* *Y* ∈ *synth* (*analz H*)
hence *hmac* ⟨*AuthTag*, *M*⟩ (*shrK A B*) ∈ *synth* (*analz H*) **by** *auto*
then have *hmac* ⟨*AuthTag*, *M*⟩ (*shrK A B*) ∈ *analz H* ∨
shrK A B ∈ *synth* (*analz H*)
by (*rule synth.cases*, *auto*)
then show ⟨*X*, *hmac* ⟨*AuthTag*, *X*⟩ (*shrK A B*)⟩ ∈ *H* ∨ (*A*, *B*) ∈ *broken H*
proof
assume *shrK A B* ∈ *synth* (*analz H*)
with *H* **have** (*A*, *B*) ∈ *broken H* **by** (*auto dest*:*sym.analz-LtKeys*)
then show *?thesis* **by** *auto*
next
assume *hmac* ⟨*AuthTag*, *M*⟩ (*shrK A B*) ∈ *analz H*
hence *hmac* ⟨*AuthTag*, *M*⟩ (*shrK A B*) ∈ *parts H* **by** (*rule analz-into-parts*)
with *H* **have** *hmac* ⟨*AuthTag*, *M*⟩ (*shrK A B*) ∈ *parts H*
by (*auto dest*!:*payload-parts elim*!:*payload-Hash*)
from *H* **obtain** *Z* **where** *Z* ∈ *H* **and** *H''*:*hmac* ⟨*AuthTag*, *M*⟩ (*shrK A B*) ∈ *parts* {*Z*}
using *parts-singleton* [*OF* ‹*hmac* ⟨*AuthTag*, *M*⟩ (*shrK A B*) ∈ *parts H*›] **by** *blast*
moreover with *H* **have** *Z* ∈ *sym.valid* **by** (*auto dest*!: *subsetD*)
moreover with *H''* **have** *Z* = *sym.implAuth A B M*
by (*auto*) (*erule sym.valid-cases*, *auto*)
ultimately have *sym.implAuth A B M* ∈ *H* **by** *auto*
with *H'* **show** *?thesis* **by** *auto*
qed
qed

lemma *implSecure-synth-analz*:
H ⊆ *payload* ∪ *sym.valid* ∪ *range LtK* ∪ *Tags* ⟹
sym.implSecure A B M ∈ *synth* (*analz H*) ⟹
sym.implSecure A B M ∈ *H* ∨ (*M* ∈ *synth* (*analz H*) ∧ (*A*, *B*) ∈ *broken H*)
apply (*erule synth.cases*, *auto*)

apply (*frule sym.analz-valid* [**where** *x=secure*], *auto*)
apply (*frule sym.analz-valid* [**where** *x=secure*], *auto*)
apply (*auto dest*:*sym.analz-LtKeys*)
done

interpretation *sym*: *valid-implem implem-sym*
proof (*unfold-locales*)
  fix *H A B M*
  assume $H \subseteq$ *payload* $\cup$ *sym.valid* $\cup$ *range LtK* $\cup$ *Tags*
        implem-sym (*Insec A B M*) $\in$ *synth* (*analz H*)
  then show *implem-sym* (*Insec A B M*) $\in H \vee M \in$ *synth* (*analz H*)
    by (*rule implInsec-synth-analz*)
next
  fix *H A B M*
  assume $H \subseteq$ *payload* $\cup$ *sym.valid* $\cup$ *range LtK* $\cup$ *Tags*
        implem-sym (*Confid A B M*) $\in$ *synth* (*analz H*)
  then show *implem-sym* (*Confid A B M*) $\in H \vee M \in$ *synth* (*analz H*)
    by (*rule implConfid-synth-analz*)
next
  fix *H A B M*
  assume $H \subseteq$ *payload* $\cup$ *sym.valid* $\cup$ *range LtK* $\cup$ *Tags*
        implem-sym (*Auth A B M*) $\in$ *synth* (*analz H*)
  then show *implem-sym* (*Auth A B M*) $\in H \vee$
            $M \in$ *synth* (*analz H*) $\wedge$ (*A*, *B*) $\in$ *broken H*
    by (*rule implAuth-synth-analz*)
next
  fix *H A B M*
  assume $H \subseteq$ *payload* $\cup$ *sym.valid* $\cup$ *range LtK* $\cup$ *Tags*
        implem-sym (*Secure A B M*) $\in$ *synth* (*analz H*)
  then show *implem-sym* (*Secure A B M*) $\in H \vee$
            $M \in$ *synth* (*analz H*) $\wedge$ (*A*, *B*) $\in$ *broken H*
    by (*rule implSecure-synth-analz*)
qed

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*
done

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*
by *auto*

lemma *implAuthSet-implAuthSet-aux*: *asym.implAuthSet G* ⊆ *implAuthSet-aux G*
by *auto*

lemma *implConfidSet-implConfidSet-aux*: *asym.implConfidSet Ag G* ⊆ *implConfidSet-aux Ag G*
by (*auto*, *blast*)

lemma *implSecureSet-implSecureSet-aux*: *asym.implSecureSet Ag G* ⊆ *implSecureSet-aux Ag G*
by (*auto*, *blast*)

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* ∪ *H*) ⊆ *PairSet AgentSet G* ∪ *AgentSet* ∪ *analz* (*G* ∪ *H*)
proof −
  have *analz* (*PairSet AgentSet G* ∪ *H*) ⊆ *PairSet AgentSet G* ∪ *analz* (*AgentSet* ∪ *G* ∪ *H*)
    by (*rule analz-Un-PairSet*)
  also have ... ⊆ *PairSet AgentSet G* ∪ *AgentSet* ∪ *analz* (*G* ∪ *H*)
    apply (*simp only*: *Un-assoc*)
    apply (*intro Un-mono analz-Un-AgentSet*, *fast*)
    done
  finally show *?thesis* **.**
qed

15.2.2 Pull *implInsecSet* out of *analz*

lemma *analz-Un-implInsecSet-aux-aux*:
assumes *Enc-keys-clean* (*G* ∪ *H*)
shows *analz* (*implInsecSet-aux G* ∪ *H*) ⊆ *implInsecSet-aux G* ∪ *Tags* ∪ *synth* (*analz* (*G* ∪ *H*))
proof −
  have *analz* (*implInsecSet-aux G* ∪ *H*) ⊆
      implInsecSet-aux G ∪ *analz* (*Tags* ∪ *PairSet AgentSet* (*PairSet AgentSet G*) ∪ *H*)
    by (*rule analz-Un-PairSet*)
  also have ... ⊆ *implInsecSet-aux G* ∪ *Tags* ∪ *analz* (*PairSet AgentSet* (*PairSet AgentSet G*) ∪ *H*)
    using *assms*
    apply −
    apply (*simp only*: *Un-assoc*, *rule Un-mono*, *fast*)
    apply (*rule analz-Un-Tag*, *blast*)
    done
  also have ... ⊆ *implInsecSet-aux G* ∪ *Tags* ∪ *PairSet AgentSet* (*PairSet AgentSet G*) ∪ *AgentSet*
      ∪ *analz* (*PairSet AgentSet G* ∪ *H*)
    apply −

apply (*simp only*: *Un-assoc*, (*rule Un-mono*, *fast*)+)
apply (*simp only*: *Un-assoc* [*symmetric*], *rule analz-Un-PairAgentSet*)
done
also have
... ⊆ *implInsecSet-aux G* ∪ *Tags* ∪ *PairSet AgentSet* (*PairSet AgentSet G*) ∪ *AgentSet* ∪ *PairSet AgentSet G* ∪ *AgentSet* ∪ *analz* (*G* ∪ *H*)
apply −
apply (*simp only*: *Un-assoc*, (*rule Un-mono*, *fast*)+)
apply (*simp only*: *Un-assoc* [*symmetric*], *rule analz-Un-PairAgentSet*)
done
also have ... ⊆ *implInsecSet-aux G* ∪ *Tags* ∪ *synth* (*analz* (*G* ∪ *H*))
apply −
apply (*simp only*: *Un-assoc*, (*rule Un-mono*, *fast*)+, *auto*)
done
finally show *?thesis* **.**
qed

lemma *analz-Un-implInsecSet-aux*:
Enc-keys-clean (*G* ∪ *H*) ⟹
analz (*implInsecSet-aux G* ∪ *H*) ⊆ *synth* (*analz* (*G* ∪ *H*)) ∪ −*payload*
by (*rule subset-trans* [*OF analz-Un-implInsecSet-aux-aux*], *auto*)

lemma *analz-Un-implInsecSet*:
Enc-keys-clean (*G* ∪ *H*) ⟹
analz (*asym.implInsecSet G* ∪ *H*) ⊆ *synth* (*analz* (*G* ∪ *H*)) ∪ −*payload*
apply (*rule subset-trans* [*of* - *analz* (*implInsecSet-aux G* ∪ *H*) -])
apply (*rule analz-mono*, *rule Un-mono*, *blast intro*!: *implSet-implSet-aux*, *simp*)
apply (*blast dest*: *analz-Un-implInsecSet-aux*)
done

15.3 Pull *implConfidSet* out of *analz*

lemma *analz-Un-implConfidSet-aux-aux*:
Enc-keys-clean (*G* ∪ *H*) ⟹
analz (*implConfidSet-aux Ag G* ∪ *H*) ⊆
implConfidSet-aux Ag G ∪ *PairSet AgentSet G* ∪
synth (*analz* (*G* ∪ *H*))
apply (*rule subset-trans* [*OF analz-Un-AencSet*], *blast*, *blast*)
apply (*simp only*: *Un-assoc*, *rule Un-mono*, *simp*)
apply (*rule subset-trans* [*OF analz-Un-PairAgentSet*], *blast*)
done

lemma *analz-Un-implConfidSet-aux*:
Enc-keys-clean (*G* ∪ *H*) ⟹
analz (*implConfidSet-aux Ag G* ∪ *H*) ⊆ *synth* (*analz* (*G* ∪ *H*)) ∪ −*payload*
by (*rule subset-trans* [*OF analz-Un-implConfidSet-aux-aux*], *auto*)

lemma *analz-Un-implConfidSet*:
Enc-keys-clean (*G* ∪ *H*) ⟹
analz (*asym.implConfidSet Ag G* ∪ *H*) ⊆ *synth* (*analz* (*G* ∪ *H*)) ∪ −*payload*
apply (*rule subset-trans* [*of* - *analz* (*implConfidSet-aux Ag G* ∪ *H*) -])
apply (*rule analz-mono*, *rule Un-mono*, *blast intro*!: *implSet-implSet-aux*, *simp*)
using *analz-Un-implConfidSet-aux* **apply** *blast*

done

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*)
apply (*rule subset-trans* [*OF analz-Un-AencSet2*], *simp*)
apply (*auto dest*:*analz-into-parts*)
done

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*
by (*rule subset-trans* [*OF analz-Un-implConfidSet-aux-aux-2*], *auto*)

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*
apply (*rule subset-trans* [*of - analz* (*implConfidSet-aux Ag G* $\cup$ *H*) -])
apply (*rule analz-mono*, *rule Un-mono*, *blast intro*!: *implSet-implSet-aux*, *simp*)
using *analz-Un-implConfidSet-aux-2* **apply** *auto*
done

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*))
apply (*rule subset-trans* [*OF analz-Un-SignSet*], *blast*, *blast*)
apply (*rule Un-mono*, *blast*)
apply (*rule subset-trans* [*OF analz-Un-PairAgentSet*], *blast*)
done

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*
by (*rule subset-trans* [*OF analz-Un-implAuthSet-aux-aux*], *auto*)

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*
apply (*rule subset-trans* [*of - analz* (*implAuthSet-aux G* $\cup$ *H*) -])
apply (*rule analz-mono*, *rule Un-mono*, *blast intro*!: *implSet-implSet-aux*, *simp*)
using *analz-Un-implAuthSet-aux* **apply** *blast*
done

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* ∪ *H*) ⊆
implSecureSet-aux Ag G ∪ *AencSet* (*PairSet Tags* (*PairSet AgentSet G*)) (*pubK*‘ (*Ag*‘‘ *UNIV*)) ∪
PairSet Tags (*PairSet AgentSet G*) ∪ *Tags* ∪ *PairSet AgentSet G* ∪
synth (*analz* (*G* ∪ *H*))
apply (*rule subset-trans* [*OF analz-Un-SignSet*], *blast*, *blast*)
apply (*simp only*: *Un-assoc*, *rule Un-mono*, *simp*)
apply (*rule subset-trans* [*OF analz-Un-AencSet*], *blast*, *blast*)
apply (*rule Un-mono*, *simp*)
apply (*rule subset-trans* [*OF analz-Un-PairSet*], *rule Un-mono*, *simp*, *simp only*: *Un-assoc*)
apply (*rule subset-trans* [*OF analz-Un-Tag*], *blast*)
apply (*rule Un-mono*, *simp*)
apply (*rule subset-trans* [*OF analz-Un-PairAgentSet*], *blast*)
done

lemma *analz-Un-implSecureSet-aux*:
Enc-keys-clean (*G* ∪ *H*) ⟹
analz (*implSecureSet-aux Ag G* ∪ *H*) ⊆ *synth* (*analz* (*G* ∪ *H*)) ∪ −*payload*
by (*rule subset-trans* [*OF analz-Un-implSecureSet-aux-aux*], *auto*)

lemma *analz-Un-implSecureSet*:
Enc-keys-clean (*G* ∪ *H*) ⟹
analz (*asym.implSecureSet Ag G* ∪ *H*) ⊆ *synth* (*analz* (*G* ∪ *H*)) ∪ −*payload*
apply (*rule subset-trans* [*of - analz* (*implSecureSet-aux Ag G* ∪ *H*) -])
apply (*rule analz-mono*, *rule Un-mono*, *blast intro*!: *implSet-implSet-aux*, *simp*)
using *analz-Un-implSecureSet-aux* **apply** *blast*
done

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

lemma *analz-Un-implSecureSet-aux-aux-2*:
Enc-keys-clean (*G* ∪ *H*) ⟹
Ag ∩ *broken* (*parts H* ∩ *range LtK*) = {} ⟹
analz (*implSecureSet-aux Ag G* ∪ *H*) ⊆
implSecureSet-aux Ag G ∪ *AencSet* (*PairSet Tags* (*PairSet AgentSet G*)) (*pubK*‘ (*Ag*‘‘ *UNIV*)) ∪
synth (*analz H*)
apply (*rule subset-trans* [*OF analz-Un-SignSet*], *blast*, *blast*)
apply (*simp only*: *Un-assoc*, *rule Un-mono*, *simp*)
apply (*rule subset-trans* [*OF analz-Un-AencSet2*], *simp*)
apply (*auto dest*: *analz-into-parts*)
done

lemma *analz-Un-implSecureSet-aux-2*:
Enc-keys-clean (*G* ∪ *H*) ⟹
Ag ∩ *broken* (*parts H* ∩ *range LtK*) = {} ⟹
analz (*implSecureSet-aux Ag G* ∪ *H*) ⊆ *synth* (*analz H*) ∪ −*payload*
by (*rule subset-trans* [*OF analz-Un-implSecureSet-aux-aux-2*], *auto*)

lemma *analz-Un-implSecureSet-2*:
Enc-keys-clean (*G* ∪ *H*) ⟹
Ag ∩ *broken* (*parts H* ∩ *range LtK*) = {} ⟹
analz (*asym.implSecureSet Ag G* ∪ *H*) ⊆
synth (*analz H*) ∪ −*payload*
apply (*rule subset-trans* [*of - analz* (*implSecureSet-aux Ag G* ∪ *H*) -])
apply (*rule analz-mono*, *rule Un-mono*, *blast intro*!: *implSet-implSet-aux*, *simp*)

```
using analz-Un-implSecureSet-aux-2 apply auto
done

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

15.6 Locale interpretations

```
interpretation asym: semivalid-implem implem-asym
proof (unfold-locales)
  fix x A B M x′ A′ B′ M′
  show implem-asym (Chan x A B M) = implem-asym (Chan x′ A′ B′ M′) ⟷
        x = x′ ∧ A = A′ ∧ B = B′ ∧ M = M′
    by (cases x, (cases x′, auto)+)
next
  fix M′ M x x′ A A′ B B′
  assume M′ ∈ payload implem-asym (Chan x A B M) ∈ parts {implem-asym (Chan x′ A′ B′ M′)}
  then show x = x′ ∧ A = A′ ∧ B = B′ ∧ M = M′
    by (cases x, auto,(cases x′, auto)+)
next
  fix I
  assume I ⊆ asym.valid
  then show Enc-keys-clean I
    proof (simp add: Enc-keys-clean-def, intro allI impI)
      fix X Y
      assume Enc X Y ∈ parts I
      obtain x A B M where M ∈ payload and Enc X Y ∈ parts {implem-asym (Chan x A B M)}
      using parts-singleton [OF ‹Enc X Y ∈ parts I›] ‹I ⊆ asym.valid›
        by (auto elim!: asym.validE)
      then show Y ∈ range LtK ∨ Y ∈ payload by (cases x, auto)
    qed
next
  fix Z
  show composed (implem-asym Z)
    proof (cases Z, simp)
      fix x A B M
      show composed (implem-asym (Chan x A B M)) by (cases x, auto)
    qed
next
  fix x A B M
  show implem-asym (Chan x A B M) ∉ payload
    by (cases x, auto)
next
  fix X K
  assume X ∈ asym.valid
  then obtain x A B M where M ∈ payload X = implem-asym (Chan x A B M)
    by (auto elim: asym.validE)
  then show LtK K ∉ parts {X}
  by (cases x, auto)
next
  fix G H
  assume G ⊆ payload Enc-keys-clean H
  hence Enc-keys-clean (G ∪ H) by (auto intro: Enc-keys-clean-Un)
  then show analz ({implem-asym (Insec A B M) |A B M. M ∈ G} ∪ H) ⊆
```

synth (*analz* $(G \cup H)) \cup -$ *payload*
by (*rule analz-Un-implInsecSet*)
next
fix *G H*
assume $G \subseteq$ *payload Enc-keys-clean H*
hence *Enc-keys-clean* $(G \cup H)$ **by** (*auto intro*: *Enc-keys-clean-Un*)
then show *analz* ({*implem-asym* (*Auth A B M*) |*A B M*. $M \in G\} \cup H) \subseteq$
synth (*analz* $(G \cup H)) \cup -$ *payload*
by (*rule analz-Un-implAuthSet*)
next
fix *G H Ag*
assume $G \subseteq$ *payload Enc-keys-clean H*
hence *Enc-keys-clean* $(G \cup H)$ **by** (*auto intro*: *Enc-keys-clean-Un*)
then show *analz* ({*implem-asym* (*Confid A B M*) |*A B M*. $(A, B) \in Ag \wedge M \in G\} \cup H) \subseteq$
synth (*analz* $(G \cup H)) \cup -$ *payload*
by (*rule analz-Un-implConfidSet*)
next
fix *G H Ag*
assume $G \subseteq$ *payload Enc-keys-clean H*
hence *Enc-keys-clean* $(G \cup H)$ **by** (*auto intro*: *Enc-keys-clean-Un*)
then show *analz* ({*implem-asym* (*Secure A B M*) |*A B M*. $(A, B) \in Ag \wedge M \in G\} \cup H) \subseteq$
synth (*analz* $(G \cup H)) \cup -$ *payload*
by (*rule analz-Un-implSecureSet*)
next
fix *G H Ag*
assume $G \subseteq$ *payload*
assume *Enc-keys-clean H*
moreover assume $Ag \cap$ *broken* (*parts* $H \cap$ *range LtK*) = {}
ultimately show *analz* ({*implem-asym* (*Confid A B M*) |*A B M*. $(A, B) \in Ag \wedge M \in G\} \cup H)$
$\subseteq$
synth (*analz* $H) \cup -$ *payload*
by (*rule analz-Un-implConfidSet-2*)
next
fix *G H Ag*
assume $G \subseteq$ *payload Enc-keys-clean H*
hence *Enc-keys-clean* $(G \cup H)$ **by** (*auto intro*: *Enc-keys-clean-Un*)
moreover assume $Ag \cap$ *broken* (*parts* $H \cap$ *range LtK*) = {}
ultimately
show *analz* ({*implem-asym* (*Secure A B M*) |*A B M*. $(A, B) \in Ag \wedge M \in G\} \cup H) \subseteq$
synth (*analz* $H) \cup -$ *payload*
by (*rule analz-Un-implSecureSet-2*)
qed

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*)
apply (*erule synth.cases*, *auto*)

done

lemma *implConfid-synth-analz*:
H ⊆ *payload* ∪ *asym.valid* ∪ *range LtK* ∪ *Tags* ⟹
asym.implConfid A B M ∈ *synth* (*analz H*) ⟹
asym.implConfid A B M ∈ *H* ∨ *M* ∈ *synth* (*analz H*)
apply (*erule synth.cases*, *auto*)
apply (*frule asym.analz-valid* [**where** *x*=*confid*], *auto*)
done

lemma *implAuth-synth-analz*:
H ⊆ *payload* ∪ *asym.valid* ∪ *range LtK* ∪ *Tags* ⟹
asym.implAuth A B M ∈ *synth* (*analz H*) ⟹
asym.implAuth A B M ∈ *H* ∨ (*M* ∈ *synth* (*analz H*) ∧ (*A*, *B*) ∈ *broken H*)
apply (*erule synth.cases*, *auto*)
apply (*frule asym.analz-valid* [**where** *x*=*auth*], *auto*)
apply (*frule asym.analz-valid* [**where** *x*=*auth*], *auto*)
apply (*blast dest*: *asym.analz-LtKeys*)
done

lemma *implSecure-synth-analz*:
H ⊆ *payload* ∪ *asym.valid* ∪ *range LtK* ∪ *Tags* ⟹
asym.implSecure A B M ∈ *synth* (*analz H*) ⟹
asym.implSecure A B M ∈ *H* ∨ (*M* ∈ *synth* (*analz H*) ∧ (*A*, *B*) ∈ *broken H*)
using [[*simproc del*: *defined-all*]] **proof** (*erule synth.cases*, *simp-all*)
fix *X*
assume *H*:*H* ⊆ *payload* ∪ *asym.valid* ∪ *range LtK* ∪ *Tags*
assume *H'*:*Sign* (*Aenc* ⟨*SecureTag*, *Agent A*, *M*⟩ (*pubK B*)) (*priK A*) = *X*
X ∈ *analz H*
hence *asym.implSecure A B M* ∈ *analz H* **by** *auto*
with *H* **have** *asym.implSecure A B M* ∈ *H* **by** (*rule asym.analz-valid*)
with *H'* **show** *X* ∈ *H* ∨ *M* ∈ *synth* (*analz H*) ∧ (*A*, *B*) ∈ *broken H*
by *auto*
next
fix *X Y*
assume *H*:*H* ⊆ *payload* ∪ *asym.valid* ∪ *range LtK* ∪ *Tags*
assume *H'*:*Aenc* ⟨*SecureTag*, *Agent A*, *M*⟩ (*pubK B*) = *X* ∧ *priK A* = *Y*
X ∈ *synth* (*analz H*) *Y* ∈ *synth* (*analz H*)
hence *priK A* ∈ *analz H* **by** *auto*
with *H* **have** *HAgents*:(*A*, *B*) ∈ *broken H* **by** (*auto dest*: *asym.analz-LtKeys*)
from *H'* **have** *Aenc* ⟨*SecureTag*, *Agent A*, *M*⟩ (*pubK B*) ∈ *synth* (*analz H*) **by** *auto*
then have *Aenc* ⟨*SecureTag*, *Agent A*, *M*⟩ (*pubK B*) ∈ *analz H* ∨
M ∈ *synth* (*analz H*)
by (*rule synth.cases*, *auto*)
then show *Sign X Y* ∈ *H* ∨ *M* ∈ *synth* (*analz H*) ∧ (*A*, *B*) ∈ *broken H*
proof
assume *M* ∈ *synth* (*analz H*)
with *HAgents* **show** *?thesis* **by** *auto*
next
assume *Aenc* ⟨*SecureTag*, *Agent A*, *M*⟩ (*pubK B*) ∈ *analz H*
hence *Aenc* ⟨*SecureTag*, *Agent A*, *M*⟩ (*pubK B*) ∈ *parts H* **by** (*rule analz-into-parts*)
from *H* **obtain** *Z* **where**
Z ∈ *H* **and** *H''*:*Aenc* ⟨*SecureTag*, *Agent A*, *M*⟩ (*pubK B*) ∈ *parts* {*Z*}

using *parts-singleton* [*OF* ‹*Aenc* ⟨*SecureTag*, *Agent A*, *M*⟩ (*pubK B*) ∈ *parts H*›]
by *blast*
moreover with *H* **have** *Z* ∈ *asym.valid* **by** (*auto dest*!: *subsetD*)
moreover with *H''* **have** *Z* = *asym.implSecure A B M*
by (*auto*) (*erule asym.valid-cases*, *auto*)
ultimately have *asym.implSecure A B M* ∈ *H* **by** *auto*
with *H'* **show** *?thesis* **by** *auto*
qed
qed

interpretation *asym*: *valid-implem implem-asym*
proof (*unfold-locales*)
fix *H A B M*
assume *H* ⊆ *payload* ∪ *asym.valid* ∪ *range LtK* ∪ *Tags*
implem-asym (*Insec A B M*) ∈ *synth* (*analz H*)
then show *implem-asym* (*Insec A B M*) ∈ *H* ∨ *M* ∈ *synth* (*analz H*)
by (*rule implInsec-synth-analz*)
next
fix *H A B M*
assume *H* ⊆ *payload* ∪ *asym.valid* ∪ *range LtK* ∪ *Tags*
implem-asym (*Confid A B M*) ∈ *synth* (*analz H*)
then show *implem-asym* (*Confid A B M*) ∈ *H* ∨ *M* ∈ *synth* (*analz H*)
by (*rule implConfid-synth-analz*)
next
fix *H A B M*
assume *H* ⊆ *payload* ∪ *asym.valid* ∪ *range LtK* ∪ *Tags*
implem-asym (*Auth A B M*) ∈ *synth* (*analz H*)
then show *implem-asym* (*Auth A B M*) ∈ *H* ∨
M ∈ *synth* (*analz H*) ∧ (*A*, *B*) ∈ *broken H*
by (*rule implAuth-synth-analz*)
next
fix *H A B M*
assume *H* ⊆ *payload* ∪ *asym.valid* ∪ *range LtK* ∪ *Tags*
implem-asym (*Secure A B M*) ∈ *synth* (*analz H*)
then show *implem-asym* (*Secure A B M*) ∈ *H* ∨
M ∈ *synth* (*analz H*) ∧ (*A*, *B*) ∈ *broken H*
by (*rule implSecure-synth-analz*)
qed

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-frame R*) = *domain* (*role* (*guessed-runs R*))
  guessed-frame-payload-spec [*simp*, *elim*]:
    guessed-frame R x = *Some y* ⟹ *y* ∈ *payload*
  guessed-frame-Init-xpkE [*simp*]:
    role (*guessed-runs R*) = *Init* ⟹ *guessed-frame R xpkE* = *Some* (*epubKF* (*R$kE*))
  guessed-frame-Init-xskE [*simp*]:
    role (*guessed-runs R*) = *Init* ⟹ *guessed-frame R xskE* = *Some* (*epriKF* (*R$kE*))
  guessed-frame-Resp-xsk [*simp*]:
    role (*guessed-runs R*) = *Resp* ⟹ *guessed-frame R xsk* = *Some* (*NonceF* (*R$sk*))
apply (*rule exI* [*of* -
    λ*R*.
      if role (*guessed-runs R*) = *Init*
      then [*xpkE* ↦ *epubKF* (*R$kE*), *xskE* ↦ *epriKF* (*R$kE*), *xsk* ↦ *End*]
      else [*xpkE* ↦ *End*, *xsk* ↦ *NonceF* (*R$sk*)]],
  auto simp add: *domIff intro*: *role-t.exhaust*)
done

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*
  signals :: *signal* ⇒ *nat*

type-synonym *l1-obs* = *l1-state*

abbreviation
  run-ended :: *var set option* ⇒ *bool*
where
  run-ended r ≡ *in-progress r xsk*

lemma *run-ended-not-None* [*elim*]:
  run-ended R ⟹ *R* = *None* ⟹ *False*
by (*fast dest*: *in-progress-Some*)

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*}|)
  }

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-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*})
      |)
  }

definition
  l1-step2 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *msg* ⇒ (*'a l1-state-scheme* ∗ *'a l1-state-scheme*) *set*

where
l1-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* ∧
(*can-signal s A B* ⟶ — authentication guard
(∃ *Ra*. *guessed-runs Ra* = (|*role=Init*, *owner=A*, *partner=B*|) ∧
in-progress (*progress s Ra*) *xpkE* ∧ *guessed-frame Ra xpkE* = *Some KE*)) ∧
(*Rb* = *test* ⟶ *NonceF* (*Rb$sk*) ∉ *synth* (*analz* (*ik s*))) ∧
— actions:
s′ = s(| *progress* := (*progress s*)(*Rb* ↦ {*xpkE*, *xsk*}),
secret := {*x*. *x* = *NonceF* (*Rb$sk*) ∧ *Rb* = *test*} ∪ *secret s*,
signals := *if can-signal s A B then*
addSignal (*signals s*) (*Running A B* (⟨ *KE*, *NonceF* (*Rb$sk*) ⟩))
else
signals s
|)
}

definition
l1-step3 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *msg* ⇒ (′*a l1-state-scheme* ∗ ′*a l1-state-scheme*) *set*
where
l1-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* ∧
(*can-signal s A B* ⟶ — authentication guard
(∃ *Rb*. *guessed-runs Rb* = (|*role=Resp*, *owner=B*, *partner=A*|) ∧
progress s Rb = *Some* {*xpkE*, *xsk*} ∧
guessed-frame Rb xpkE = *Some* (*epubKF* (*Ra$kE*)) ∧
guessed-frame Rb xsk = *Some K*)) ∧
(*Ra* = *test* ⟶ *K* ∉ *synth* (*analz* (*ik s*))) ∧

— actions:
s′ = s(| *progress* := (*progress s*)(*Ra* ↦ {*xpkE*, *xskE*, *xsk*}),
secret := {*x*. *x* = *K* ∧ *Ra* = *test*} ∪ *secret s*,
signals := *if can-signal s A B then*
addSignal (*signals s*) (*Commit A B* ⟨*epubKF* (*Ra$kE*), *K*⟩)
else
signals s
|)
}

specification

definition
l1-init :: *l1-state set*
where
l1-init ≡ { (|
ik = {},
secret = {},
progress = *Map.empty*,

signals = λ*x*. *0*
⦈ }

definition
l1-trans :: ('*a l1-state-scheme* ∗ '*a l1-state-scheme*) *set* **where**
l1-trans ≡ (⋃ *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*
by (*simp add*: *l1-def*)

declare *domIff* [*iff*]

lemma *run-ended-trans*:
run-ended (*progress s R*) ⟹
(*s*, *s'*) ∈ *trans l1* ⟹
run-ended (*progress s' R*)
apply (*auto simp add*: *l1-nostep-defs*)
apply (*simp add*: *l1-defs ik-dy-def*, *fast* ?)+
done

declare *domIff* [*iff del*]

lemma *can-signal-trans*:
can-signal s' A B ⟹
(*s*, *s'*) ∈ *trans l1* ⟹
can-signal s A B
by (*auto simp add*: *can-signal-def run-ended-trans*)

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*}
apply (*simp add*: *PO-rhoare-defs R01s-def*)
apply *auto*
apply (*simp add*: *l1-defs s0-defs s0-secrecy-def*)
done

protocol events

lemma *l1-step1-refines-skip*:
{*R01s*} *Id*, *l1-step1 Ra A B* {>*R01s*}
by (*auto simp add*: *PO-rhoare-defs R01s-def l1-step1-def*)

lemma *l1-step2-refines-add-secret-skip*:
{*R01s*} *s0-add-secret* (*NonceF* (*Rb*$*sk*)) ∪ *Id*, *l1-step2 Rb A B KE* {>*R01s*}
apply (*auto simp add*: *PO-rhoare-defs R01s-def s0-add-secret-def*)
apply (*auto simp add*: *l1-step2-def*)
done

lemma *l1-step3-refines-add-secret-skip*:
{*R01s*} *s0-add-secret K* ∪ *Id*, *l1-step3 Ra A B K* {>*R01s*}
apply (*auto simp add*: *PO-rhoare-defs R01s-def s0-add-secret-def*)
apply (*auto simp add*: *l1-step3-def*)
done

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*)
by (*auto simp add*: *R01s-def s0-defs l1-defs s0-secrecy-def*)

lemma *l1-refines-trans-s0* [*iff*]:
{*R01s*} *trans s0*, *trans l1* {> *R01s*}
by (*auto simp add*: *s0-def l1-def s0-trans-def l1-trans-def*
intro: *l1-trans-refines-s0-trans*)

lemma *obs-consistent-med01x* [*iff*]:
obs-consistent R01s med01s s0 l1
by (*auto simp add*: *obs-consistent-def R01s-def med01s-def*)

refinement result

lemma *l1s-refines-s0* [*iff*]:
refines
R01s
med01s s0 l1
by (*auto simp add*:*refines-def PO-refines-def*)

lemma *l1-implements-s0* [*iff*]: *implements med01s s0 l1*
by (*rule refinement-soundness*) (*fast*)

16.3 Derived invariants: secrecy

abbreviation *l1-secrecy* ≡ *s0-secrecy*

lemma *l1-obs-secrecy* [*iff*]: *oreach l1* ⊆ *l1-secrecy*
apply (*rule external-invariant-translation*
[*OF s0-obs-secrecy - l1-implements-s0*])
apply (*auto simp add*: *med01s-def s0-secrecy-def*)
done

lemma *l1-secrecy* [*iff*]: *reach l1* ⊆ *l1-secrecy*
by (*rule external-to-internal-invariant* [*OF l1-obs-secrecy*], *auto*)

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*
by (*auto simp add*: *l1-def l1-init-def l1-inv1-def*)

declare *domIff* [*iff*]

lemma *l1-inv1-trans* [*iff*]:
  {*l1-inv1*} *trans l1* {> *l1-inv1*}
apply (*auto simp add*: *PO-hoare-defs l1-nostep-defs intro*!: *l1-inv1I*)
apply (*auto simp add*: *l1-defs ik-dy-def l1-inv1-def*)
done

lemma *PO-l1-inv1* [*iff*]: *reach l1* ⊆ *l1-inv1*
by (*rule inv-rule-basic*) (*auto*)

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*
by (*auto simp add*: *l1-def l1-init-def l1-inv2-def*)

lemma *l1-inv2-trans* [*iff*]:
  {*l1-inv2*} *trans l1* {> *l1-inv2*}
apply (*auto simp add*: *PO-hoare-defs intro*!: *l1-inv2I*)
apply (*drule can-signal-trans*, *assumption*)
apply (*auto simp add*: *l1-nostep-defs*)
apply (*auto simp add*: *l1-defs ik-dy-def l1-inv2-def*)
done

lemma *PO-l1-inv2* [*iff*]: *reach l1* ⊆ *l1-inv2*
by (*rule inv-rule-basic*) (*auto*)

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*
apply (*auto intro*!: *l1-inv3I*)
apply (*auto dest*!: *l1-inv2D*)
apply (*rename-tac x A B Rb Ra*)
apply (*case-tac signals x* (*Commit A B* ⟨*epubKF* (*Ra* $ *kE*), *NonceF* (*Rb* $ *sk*)⟩) > *0*, *auto*)
apply (*fastforce dest*: *l1-inv1D elim*: *equalityE*)
done

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*}
apply (*auto simp add*: *PO-rhoare-defs R01ia-def*)

apply (*simp add*: *l1-learn-def*)
done

protocol events

lemma *l1-step1-refines-a0i-skip*:
  {*R01ia*} *Id*, *l1-step1 Ra A B* {>*R01ia*}
by (*auto simp add*: *PO-rhoare-defs R01ia-def l1-step1-def*)

lemma *l1-step2-refines-a0i-running-skip*:
  {*R01ia*} *a0i-running A B* ⟨*KE*, *NonceF* (*Rb*$*sk*)⟩ ∪ *Id*, *l1-step2 Rb A B KE* {>*R01ia*}
by (*auto simp add*: *PO-rhoare-defs R01ia-def*, *simp-all add*: *l1-step2-def a0i-running-def*, *auto*)

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*}
apply (*auto simp add*: *PO-rhoare-defs R01ia-def*)
apply (*auto simp add*: *l1-step3-def a0i-commit-def*)
apply (*force elim*!: *l1-inv3E*)+
done

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*)
by (*auto simp add*: *R01ia-def a0i-defs l1-defs*)

lemma *l1-refines-trans-a0i* [*iff*]:
  {*R01ia* ∩ (*UNIV* × (*l1-inv1* ∩ *l1-inv2*))} *trans a0i*, *trans l1* {> *R01ia*}
proof −
  let *?pre′* = *R01ia* ∩ (*UNIV* × *l1-inv3*)
  show *?thesis* (**is** {*?pre*} *?t1*, *?t2* {>*?post*})
  proof (*rule relhoare-conseq-left*)
    show *?pre* ⊆ *?pre′*
      using *l1-inv3-derived* **by** *blast*
  next
    show {*?pre′*} *?t1*, *?t2* {> *?post*}
      apply (*auto simp add*: *a0i-def l1-def a0i-trans-def l1-trans-def*)
      prefer *2* **using** *l1-step2-refines-a0i-running-skip* **apply** (*simp add*: *PO-rhoare-defs*, *blast*)
      prefer *2* **using** *l1-step3-refines-a0i-commit-skip* **apply** (*simp add*: *PO-rhoare-defs*, *blast*)
      apply (*blast intro*!:*l1-trans-refines-a0i-trans*)+
      done
  qed
qed

lemma *obs-consistent-med01ia* [*iff*]:
  obs-consistent R01ia med01ia a0i l1

by (*auto simp add*: *obs-consistent-def R01ia-def med01ia-def*)

refinement result

lemma *l1-refines-a0i* [*iff*]:
refines
($R01ia \cap (reach\ a0i \times (l1\text{-}inv1 \cap l1\text{-}inv2)))$
med01ia a0i l1
by (*rule Refinement-using-invariants*, *auto*)

lemma *l1-implements-a0i* [*iff*]: *implements med01ia a0i l1*
by (*rule refinement-soundness*) (*fast*)

16.6 Derived invariants: injective agreement

definition
l1-iagreement :: (*'a l1-state-scheme*) *set*
where
l1-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* $\subseteq$ *l1-iagreement*
apply (*rule external-invariant-translation*
[*OF PO-a0i-obs-agreement - l1-implements-a0i*])
apply (*auto simp add*: *med01ia-def l1-iagreement-def a0i-agreement-def*)
done

lemma *l1-iagreement* [*iff*]: *reach l1* $\subseteq$ *l1-iagreement*
by (*rule external-to-internal-invariant* [*OF l1-obs-iagreement*], *auto*)

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* $\notin$ *bad-init* $\wedge$ *test-partner* $\notin$ *bad-init*
by *auto*

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* $\times$ *l2-state*) *set*

attacker events

definition
  l2-dy-fake-msg :: *msg* $\Rightarrow$ *l2-trans*
where
  l2-dy-fake-msg m $\equiv$ $\{(s,s')$.
    — guards
    $m \in$ *dy-fake-msg* (*bad s*) (*ik s*) (*chan s*) $\wedge$
    — actions
    $s' = s(\!|$*ik* := $\{m\} \cup$ *ik s*$|\!)$
  }

definition
  l2-dy-fake-chan :: *chan* $\Rightarrow$ *l2-trans*
where
  l2-dy-fake-chan M $\equiv$ $\{(s,s')$.
    — guards
    $M \in$ *dy-fake-chan* (*bad s*) (*ik s*) (*chan s*)$\wedge$
    — actions
    $s' = s(\!|$*chan* := $\{M\} \cup$ *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*
by (*cases x*, *auto*)

lemma *role-comp-inv-eq*:
y = *role-comp x* ⟷ *x* = *role-comp y*
by (*auto elim*!: *role-comp.elims* [*OF sym*])

definition
partners :: *rid-t set*
where
partners ≡ {*R*. *partner-runs test R*}

lemma *test-not-partner* [*simp*]:
test ∉ *partners*
by (*auto simp add*: *partners-def partner-runs-def*, *cases role* (*guessed-runs test*), *auto*)

lemma *matching-symmetric*:
matching sigma sigma′ ⟹ *matching sigma′ sigma*
by (*auto simp add*: *matching-def*)

lemma *partner-symmetric*:
partner-runs R R′ ⟹ *partner-runs R′ R*
by (*auto simp add*: *partner-runs-def matching-symmetric*)

lemma *partner-unique*:
partner-runs R R′′ ⟹ *partner-runs R R′* ⟹ *R′* = *R′′*
proof −

assume H':*partner-runs R R'*
then have Hm': *matching* (*guessed-frame R*) (*guessed-frame R'*)
by (*auto simp add*: *partner-runs-def*)
assume H'':*partner-runs R R''*
then have Hm'': *matching* (*guessed-frame R*) (*guessed-frame R''*)
by (*auto simp add*: *partner-runs-def*)
show *?thesis*
proof (*cases role* (*guessed-runs R'*))
case *Init*
with H' *partner-symmetric* [*OF* H''] **have** *Hrole*:*role* (*guessed-runs R*) = *Resp*
role (*guessed-runs R''*) = *Init*
by (*auto simp add*: *partner-runs-def*)
with *Init Hm'* **have** *guessed-frame R xpkE* = *Some* (*epubKF* (*R'*\$*kE*))
by (*simp add*: *matching-def*)
moreover from *Hrole Hm''* **have** *guessed-frame R xpkE* = *Some* (*epubKF* (*R''*\$*kE*))
by (*simp add*: *matching-def*)
ultimately show *?thesis* **by** *simp*
next
case *Resp*
with H' *partner-symmetric* [*OF* H''] **have** *Hrole*:*role* (*guessed-runs R*) = *Init*
role (*guessed-runs R''*) = *Resp*
by (*auto simp add*: *partner-runs-def*)
with *Resp Hm'* **have** *guessed-frame R xsk* = *Some* (*NonceF* (*R'*\$*sk*))
by (*simp add*: *matching-def*)
moreover from *Hrole Hm''* **have** *guessed-frame R xsk* = *Some* (*NonceF* (*R''*\$*sk*))
by (*simp add*: *matching-def*)
ultimately show *?thesis* **by** *simp*
qed
qed

lemma *partner-test*:
$R \in$ *partners* $\Longrightarrow$ *partner-runs R R'* $\Longrightarrow$ R' = *test*
by (*auto intro*!:*partner-unique simp add*:*partners-def partner-symmetric*)

compromising events

definition
l2-lkr-others :: *agent* ⇒ *l2-trans*
where
l2-lkr-others A ≡ {(*s*,*s'*).
— guards
$A \neq$ *test-owner* ∧
$A \neq$ *test-partner* ∧
— actions
s' = *s*(|*bad* := {*A*} ∪ *bad s*|)
}

definition
l2-lkr-actor :: *agent* ⇒ *l2-trans*
where
l2-lkr-actor A ≡ {(*s*,*s'*).
— guards
A = *test-owner* ∧
$A \neq$ *test-partner* ∧

— actions
$s' = s(\!|bad := \{A\} \cup bad\ s|\!)$
}

definition
l2-lkr-after :: *agent* ⇒ *l2-trans*
where
$l2\text{-}lkr\text{-}after\ A \equiv \{(s,s').$
— guards
$test\text{-}ended\ s\ \wedge$
— actions
$s' = s(\!|bad := \{A\} \cup bad\ s|\!)$
}

definition
l2-skr :: *rid-t* ⇒ *msg* ⇒ *l2-trans*
where
$l2\text{-}skr\ R\ K \equiv \{(s,s').$
— 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

definition
l2-step1 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *l2-trans*
where
$l2\text{-}step1\ Ra\ A\ B \equiv \{(s,\ s').$
— guards:
$Ra \notin dom\ (progress\ s)\ \wedge$
$guessed\text{-}runs\ Ra = (\!|role{=}Init,\ owner{=}A,\ partner{=}B|\!)\ \wedge$
— actions:
$s' = s(\!|$
$progress := (progress\ s)(Ra \mapsto \{xpkE,\ xskE\}),$
$chan := \{Auth\ A\ B\ (\langle Number\ 0,\ epubKF\ (Ra\$kE)\rangle)\} \cup (chan\ s)$
$|\!)$
}

definition
l2-step2 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *msg* ⇒ *l2-trans*
where
$l2\text{-}step2\ Rb\ A\ B\ KE \equiv \{(s,\ s').$
— guards:
$guessed\text{-}runs\ Rb = (\!|role{=}Resp,\ owner{=}B,\ partner{=}A|\!)\ \wedge$
$Rb \notin dom\ (progress\ s)\ \wedge$
$guessed\text{-}frame\ Rb\ xpkE = Some\ KE\ \wedge$
$Auth\ A\ B\ \langle Number\ 0,\ KE\rangle \in chan\ s\ \wedge$
— actions:
$s' = s(\!|$
$progress := (progress\ s)(Rb \mapsto \{xpkE,\ xsk\}),$

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 = *λx*. *0*,
chan = {},
bad = *bad-init*
⦈}

definition
l2-trans :: *l2-trans* **where**
l2-trans ≡ (⋃ *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*
by (*simp add*: *l2-def*)

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*)
apply (*auto simp add*: *l2-nostep-defs*)
apply (*simp add*: *l2-defs*, *fast ?*)+
done
declare *domIff* [*iff del*]

lemma *l2-can-signal-trans*:
can-signal s' A B ⟹
(*s*, *s'*) ∈ *trans l2* ⟹
can-signal s A B
by (*auto simp add*: *can-signal-def l2-run-ended-trans*)

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 ≡ {*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*
by (*auto simp add*: *l2-def l2-init-def l2-inv1-def can-signal-def bad-init-spec*)

lemma *l2-inv1-trans* [*iff*]:
  {*l2-inv1*} *trans l2* {> *l2-inv1*}
proof (*auto simp add*: *PO-hoare-defs intro*!: *l2-inv1I del*: *conjI*)
  fix *s′ s* :: *l2-state*
  fix *A B*
  assume *HI*:*s* ∈ *l2-inv1*
  assume *HT*:(*s*, *s′*) ∈ *trans l2*
  assume *can-signal s′ A B*
  with *HT* **have** *HS*:*can-signal s A B*
    by (*auto simp add*: *l2-can-signal-trans*)
  with *HI* **have** *A* ∉ *bad s* ∧ *B* ∉ *bad s*
    by *fast*
  with *HS HT* **show** *A* ∉ *bad s′* ∧ *B* ∉ *bad s′*
    by (*auto simp add*: *l2-nostep-defs can-signal-def*)
      (*simp-all add*: *l2-defs*)
qed

lemma *PO-l2-inv1* [*iff*]: *reach l2* ⊆ *l2-inv1*
by (*rule inv-rule-basic*) (*auto*)

17.2.2 inv2 (authentication guard)

If *Auth A B* ⟨*Number 0*, *KE*⟩ ∈ *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 ≡ {*s*. ∀ *A B KE*.
    Auth A B ⟨*Number 0*, *KE*⟩ ∈ *chan s* ⟶
    A ∉ *bad s* ∧ *B* ∉ *bad s* ⟶
    (∃ *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
by (auto simp add: l2-def l2-init-def l2-inv2-def)

lemma l2-inv2-trans [iff]:
  {l2-inv2} trans l2 {> l2-inv2}
apply (auto simp add: PO-hoare-defs l2-nostep-defs intro!: l2-inv2I)
apply (auto simp add: l2-defs dy-fake-chan-def)
apply force+
done

lemma PO-l2-inv2 [iff]: reach l2 ⊆ l2-inv2
by (rule inv-rule-basic) (auto)
```

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
by (auto simp add: l2-def l2-init-def l2-inv3-def)

lemma l2-inv3-trans [iff]:
  {l2-inv3} trans l2 {> l2-inv3}
apply (auto simp add: PO-hoare-defs l2-nostep-defs intro!: l2-inv3I)
apply (auto simp add: l2-defs dy-fake-chan-def)
apply (simp-all add: domIff)
apply force+
done

lemma PO-l2-inv3 [iff]: reach l2 ⊆ l2-inv3
by (rule inv-rule-basic) (auto)
```

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*
by (*auto simp add*: *l2-def l2-init-def l2-inv4-def*)

lemma *l2-inv4-trans* [*iff*]:
{*l2-inv4* ∩ *l2-inv3* ∩ *l2-inv1*} *trans l2* {> *l2-inv4*}
apply (*auto simp add*: *PO-hoare-defs l2-nostep-defs intro*!: *l2-inv4I*)
apply (*auto simp add*: *l2-defs dy-fake-chan-def*)
apply (*auto dest*!: *l2-inv4D simp add*: *domIff can-signal-def*)
apply (*auto dest*!: *l2-inv3D intro*!:*l2-inv1D simp add*: *can-signal-def*)
done

lemma *PO-l2-inv4* [*iff*]: *reach l2* ⊆ *l2-inv4*
by (*rule-tac J*=*l2-inv3* ∩ *l2-inv1* **in** *inv-rule-incr*) (*auto*)

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*
by (*auto simp add*: *l2-def l2-init-def l2-inv5-def*)

lemma *l2-inv5-trans* [*iff*]:

{*l2-inv5*} *trans l2* {> *l2-inv5*}
apply (*auto simp add*: *PO-hoare-defs l2-nostep-defs intro*!: *l2-inv5I*)
apply (*auto simp add*: *l2-defs dy-fake-chan-def intro*: *l2-inv5D dy-fake-msg-monotone*)
done

lemma *PO-l2-inv5* [*iff*]: *reach l2* ⊆ *l2-inv5*
by (*rule inv-rule-basic*) (*auto*)

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*
by (*auto simp add*: *l2-def l2-init-def l2-inv6-def*)

lemma *l2-inv6-trans* [*iff*]:
  {*l2-inv6*} *trans l2* {> *l2-inv6*}
apply (*auto simp add*: *PO-hoare-defs l2-nostep-defs intro*!: *l2-inv6I*)
apply (*auto simp add*: *l2-defs dy-fake-chan-def dest*: *l2-inv6D*)
done

lemma *PO-l2-inv6* [*iff*]: *reach l2* ⊆ *l2-inv6*
by (*rule inv-rule-basic*) (*auto*)

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*

by (*rule*, *rule*, *erule analz.induct*, *auto*)

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*
by (*auto simp add*: *l2-def l2-init-def l2-inv7-def*)

lemma *l2-inv7-step1*:
  {*l2-inv7*} *l2-step1 Ra A B* {> *l2-inv7*}
apply (*auto simp add*: *PO-hoare-defs l2-defs intro*!: *l2-inv7I*)
apply (*auto dest*: *l2-inv7D* [*THEN* [*2*] *rev-subsetD*])+
done

lemma *l2-inv7-step2*:
  {*l2-inv1* ∩ *l2-inv2* ∩ *l2-inv4* ∩ *l2-inv7*} *l2-step2 Rb A B KE* {> *l2-inv7*}
proof (*auto simp add*: *PO-hoare-defs l2-nostep-defs intro*!: *l2-inv7I*)
  fix *s′ s* :: *l2-state*
  fix *x*
  assume *Hx*:*x* ∈ *extr* (*bad s′*) (*ik s′*) (*chan s′*)
  assume *Hi*:*s* ∈ *l2-inv7*
  assume *Hi′*:*s* ∈ *l2-inv1*
  assume *Hi″*:*s* ∈ *l2-inv2*
  assume *Hi‴*:*s* ∈ *l2-inv4*
  assume *Hs*:(*s*, *s′*) ∈ *l2-step2 Rb A B KE*
  from *Hx Hi Hs* **show** *x* ∈ *synth* (*analz* (*generators*))
    proof (*auto simp add*: *l2-defs dest*: *l2-inv7D* [*THEN* [*2*] *rev-subsetD*])

first case: *can-signal s A B*, which implies that *A*, *B* are honest, and therefore the public key received by *B* is not from the attacker, which proves that the message added to the channel is in {*z*. ∃ *x k*. *z* = *Aenc x* (*epubKF k*)}

      assume *Hc*:*Auth A B* ⟨*Number 0*, *KE*⟩ ∈ *chan s*
      assume *HRb*:*guessed-runs Rb* = (|*role* = *Resp*, *owner* = *B*, *partner* = *A*|)
              guessed-frame Rb xpkE = *Some KE*
      assume *Hcs*: *can-signal s A B*
      from *Hcs Hi′* **have** *A* ∉ *bad s* ∧ *B* ∉ *bad s*
        by *auto*
      with *Hc Hi″* **obtain** *Ra* **where** *KE* = *epubKF* (*Ra*$*kE*)
                          and *guessed-runs Ra* = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|)
        by (*auto dest*: *l2-inv2D*)
      with *HRb* **show** *Aenc* (*NonceF* (*Rb* $ *sk*)) *KE* ∈ *synth* (*analz generators*)
        by *blast*
    next

second case: ¬ *can-signal s A B*. We show that *Rb* is not test and not a partner: - *Rb* is not test because in that case test is not finished and *A*, *B* are the test agents, thus *can-signal s A B* - *Rb* is not a partner for the same reason therefore the message added to the channel can be constructed from {*NonceF* (*R* $ *sk*) |*R*. *R* ≠ *test* ∧ *R* ∉ *partners*}

assume *Hc*:*Auth A B* ⟨*Number 0*, *KE*⟩ ∈ *chan s*
assume *Hcs*:¬ *can-signal s A B*
assume *HRb*:*Rb* ∉ *dom* (*progress s*)
guessed-runs Rb = (|*role* = *Resp*, *owner* = *B*, *partner* = *A*|)
from *Hcs HRb* **have** *Rb* ≠ *test*
by (*auto simp add*: *can-signal-def domIff*)
moreover from *HRb Hi'''* *Hcs* **have** *Rb* ∉ *partners*
by (*clarify*, *auto simp add*: *partners-def partner-runs-def can-signal-def matching-def domIff*)
moreover from *Hc Hi* **have** *KE* ∈ *synth* (*analz* (*generators*))
by *auto*
ultimately show *Aenc* (*NonceF* (*Rb* $ *sk*)) *KE* ∈ *synth* (*analz* (*generators*))
by *blast*
qed
qed

lemma *l2-inv7-step3*:
{*l2-inv7*} *l2-step3 Rb A B K* {> *l2-inv7*}
by (*auto simp add*: *PO-hoare-defs l2-defs intro*!: *l2-inv7I dest*: *l2-inv7D* [*THEN* [*2*] *rev-subsetD*])

lemma *l2-inv7-dy-fake-msg*:
{*l2-inv7*} *l2-dy-fake-msg M* {> *l2-inv7*}
by (*auto simp add*: *PO-hoare-defs l2-defs extr-insert-IK-eq*
intro!: *l2-inv7I*
elim!: *l2-inv7E dy-fake-msg-extr* [*THEN* [*2*] *rev-subsetD*])

lemma *l2-inv7-dy-fake-chan*:
{*l2-inv7*} *l2-dy-fake-chan M* {> *l2-inv7*}
by (*auto simp add*: *PO-hoare-defs l2-defs*
intro!: *l2-inv7I*
dest: *dy-fake-chan-extr-insert* [*THEN* [*2*] *rev-subsetD*]
elim!: *l2-inv7E dy-fake-msg-extr* [*THEN* [*2*] *rev-subsetD*])

lemma *l2-inv7-lkr-others*:
{*l2-inv7* ∩ *l2-inv5*} *l2-lkr-others A* {> *l2-inv7*}
apply (*auto simp add*: *PO-hoare-defs l2-defs*
intro!: *l2-inv7I*
dest!: *extr-insert-bad* [*THEN* [*2*] *rev-subsetD*]
elim!: *l2-inv7E l2-inv5E*)
apply (*auto dest*: *dy-fake-msg-extr* [*THEN* [*2*] *rev-subsetD*])
done

lemma *l2-inv7-lkr-after*:
{*l2-inv7* ∩ *l2-inv5*} *l2-lkr-after A* {> *l2-inv7*}
apply (*auto simp add*: *PO-hoare-defs l2-defs*
intro!: *l2-inv7I*
dest!: *extr-insert-bad* [*THEN* [*2*] *rev-subsetD*]
elim!: *l2-inv7E l2-inv5E*)
apply (*auto dest*: *dy-fake-msg-extr* [*THEN* [*2*] *rev-subsetD*])

done

lemma *l2-inv7-skr*:
  {*l2-inv7* ∩ *l2-inv6*} *l2-skr R K* {> *l2-inv7*}
proof (*auto simp add*: *PO-hoare-defs l2-defs extr-insert-IK-eq intro*!: *l2-inv7I*,
      auto elim: *l2-inv7D* [*THEN subsetD*])
  fix *s*
  assume *HRtest*: *R* ≠ *test R* ∉ *partners*
  assume *Hi*: *s* ∈ *l2-inv7*
  assume *Hi'*: *s* ∈ *l2-inv6*
  assume *HRsk*: *in-progress* (*progress s R*) *xsk guessed-frame R xsk* = *Some K*
  show *K* ∈ *synth* (*analz generators*)
    proof (*cases role* (*guessed-runs R*))

first case: *R* is the initiator, then *Aenc K epk* is in *extr* (*bad s*) (*ik s*) (*chan s*) (by invariant) therefore either *K* ∈ *synth* (*analz generators*) which proves the goal or *Aenc K epk* ∈ *generators*, which means that *K* = *NonceF* (*Rb* $ *sk*) where *R* and *Rb* are matching and since *R* is not partner or test, neither is *Rb*, and therefore *K* ∈ *synth* (*analz generators*)

      assume *HRI*: *role* (*guessed-runs R*) = *Init*
      with *HRsk Hi Hi'* **have** *Aenc K* (*epubKF* (*R*$*kE*)) ∈ *synth* (*analz generators*)
        by (*auto dest*!: *l2-inv7D*)
      then have *K* ∈ *synth* (*analz generators*) ∨ *Aenc K* (*epubKF* (*R*$*kE*)) ∈ *generators*
        by (*rule synth.cases*, *simp-all*, *simp add*: *analz-generators*)
      with *HRsk* **show** *?thesis*
        proof *auto*
          fix *Rb A B*
          assume *HR*: *guessed-runs R* = (|*role* = *Init*, *owner* = *A*, *partner* = *B*|)
                    guessed-frame R xsk = *Some* (*NonceF* (*Rb* $ *sk*))
          assume *HRb*: *guessed-runs Rb* = (|*role* = *Resp*, *owner* = *B*, *partner* = *A*|)
                    guessed-frame Rb xpkE = *Some* (*epubKF* (*R* $ *kE*))
          from *HR HRb* **have** *partner-runs Rb R*
            by (*auto simp add*: *partner-runs-def matching-def*)
          with *HRtest* **have** *Rb* ≠ *test* ∧ *Rb* ∉ *partners*
            by (*auto dest*: *partner-test*, *simp add*:*partners-def*)
          then show *NonceF* (*Rb* $ *sk*) ∈ *analz generators*
            by *blast*
        qed
    next

second case: *R* is the Responder, then *K* is *R* $ *sk* which is in *synth* (*analz generators*) since *R* is not test or partner

      assume *HRI*: *role* (*guessed-runs R*) = *Resp*
      with *HRsk HRtest* **show** *?thesis*
        by *auto*
    qed
qed

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*}
by (*auto simp add*: *l2-nostep-defs intro*:*l2-inv7-trans-aux*)

lemma *PO-l2-inv7* [*iff*]: *reach l2* ⊆ *l2-inv7*
by (*rule-tac J*=*l2-inv1* ∩ *l2-inv2* ∩ *l2-inv4* ∩ *l2-inv5* ∩ *l2-inv6* **in** *inv-rule-incr*) (*auto*)

lemma *l2-inv7-aux*:
NonceF (*R*$*sk*) ∈ *analz* (*ik s*) ⟹ *s* ∈ *l2-inv7* ⟹ *R* ≠ *test* ∧ *R* ∉ *partners*
proof −
assume *H*:*s* ∈ *l2-inv7* **and** *H'*:*NonceF* (*R*$*sk*) ∈ *analz* (*ik s*)
then have *H''*:*NonceF* (*R*$*sk*) ∈ *analz* (*extr* (*bad s*) (*ik s*) (*chan s*))
by (*auto elim*: *analz-monotone*)
from *H* **have** *analz* (*extr* (*bad s*) (*ik s*) (*chan s*)) ⊆ *analz* (*synth* (*analz generators*))
by (*blast dest*: *analz-mono intro*: *l2-inv7D*)
with *H''* **have** *NonceF* (*R*$*sk*) ∈ *analz generators*
by *auto*
then have *NonceF* (*R*$*sk*) ∈ *generators*
by (*simp add*: *analz-generators*)
then show *?thesis*
by *auto*
qed

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*
by (*auto simp add*: *l2-def l2-init-def l2-inv8-def*)

lemma *l2-inv8-trans* [*iff*]:
{*l2-inv8* ∩ *l2-inv1* ∩ *l2-inv3*} *trans l2* {> *l2-inv8*}
supply [[*simproc del*: *defined-all*]]
apply (*auto simp add*: *PO-hoare-defs l2-nostep-defs intro*!: *l2-inv8I*)
apply (*auto simp add*: *l2-defs dy-fake-chan-def*)
apply (*auto simp add*: *partners-def partner-runs-def matching-def dest*!: *l2-inv3D*)
apply (*simp-all add*: *can-signal-def*)
done

lemma *PO-l2-inv8* [*iff*]: *reach l2* ⊆ *l2-inv8*
by (*rule-tac J*=*l2-inv1* ∩ *l2-inv3* **in** *inv-rule-incr*) (*auto*)

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
by (auto simp add: can-signal-def R12s-defs)
```

protocol events

```
lemma l2-step1-refines-step1:
  {R12s} l1-step1 Ra A B, l2-step1 Ra A B {>R12s}
by (auto simp add: PO-rhoare-defs R12s-defs l1-step1-def l2-step1-def)

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}
apply (auto simp add: PO-rhoare-defs R12s-defs l1-step2-def, simp-all add: l2-step2-def)
apply (auto dest!: l2-inv7-aux l2-inv2D)
done
```

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
by (auto simp add: partners-def partner-runs-def matching-def)
```

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*}
supply [[*simproc del*: *defined-all*]]
apply (*auto simp add*: *PO-rhoare-defs R12s-defs l1-step3-def*, *simp-all add*: *l2-step3-def*)
apply (*auto dest*!: *l2-inv3D l2-inv7-aux intro*:*l2-step3-partners*)
apply (*auto simp add*: *can-signal-def*)
done

attacker events

lemma *l2-dy-fake-chan-refines-skip*:
{*R12s*} *Id*, *l2-dy-fake-chan M* {>*R12s*}
by (*auto simp add*: *PO-rhoare-defs R12s-defs l2-defs*)

lemma *l2-dy-fake-msg-refines-learn*:
{*R12s* ∩ *UNIV* × *l2-inv7* ∩ *UNIV* × *l2-inv8*} *l1-learn m*, *l2-dy-fake-msg m* {>*R12s*}
apply (*auto simp add*: *PO-rhoare-defs R12s-defs l2-loc-defs l1-defs*)
apply (*drule Fake-insert-dy-fake-msg*, *erule l2-inv7D*)
apply (*auto simp add*: *analz-generators dest*!: *l2-inv8D*)
apply (*drule subsetD*, *simp*, *drule subsetD*, *simp*, *auto*)
done

compromising events

lemma *l2-lkr-others-refines-skip*:
{*R12s*} *Id*, *l2-lkr-others A* {>*R12s*}
by (*auto simp add*: *PO-rhoare-defs R12s-defs l2-loc-defs l1-defs*)

lemma *l2-lkr-after-refines-skip*:
{*R12s*} *Id*, *l2-lkr-after A* {>*R12s*}
by (*auto simp add*: *PO-rhoare-defs R12s-defs l2-loc-defs l1-defs*)

lemma *l2-skr-refines-learn*:
{*R12s* ∩ *UNIV* × *l2-inv7* ∩ *UNIV* × *l2-inv6* ∩ *UNIV* × *l2-inv8*} *l1-learn K*, *l2-skr R K* {>*R12s*}
proof (*auto simp add*: *PO-rhoare-defs R12s-defs l2-loc-defs l1-defs*)
fix *s* :: *l2-state* **fix** *x*
assume *H*:*s* ∈ *l2-inv7* *s* ∈ *l2-inv6*
R ∉ *partners R* ≠ *test run-ended* (*progress s R*) *guessed-frame R xsk* = *Some K*
assume *Hx*:*x* ∈ *synth* (*analz* (*insert K* (*ik s*)))
assume *x* ∈ *secret s s* ∈ *l2-inv8*
then obtain *R* **where** *x* = *NonceF* (*R*$*sk*) **and** *R* = *test* ∨ *R* ∈ *partners*
by *auto*
moreover from *H* **have** *s* (|*ik* := *insert K* (*ik s*)|) ∈ *l2-inv7*
by (*auto intro*: *hoare-apply* [*OF l2-inv7-skr*] *simp add*: *l2-defs*)
ultimately show *False* **using** *Hx*
by (*auto dest*: *l2-inv7-aux* [*rotated 1*])
qed

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 ⊆ *R12s* '' (*init l1*)
by (*auto simp add*: *R12s-defs l1-defs l2-loc-defs*)

lemma *l2-refines-trans-l1* [*iff*]:
{*R12s* ∩ (*UNIV* × (*l2-inv1* ∩ *l2-inv2* ∩ *l2-inv3* ∩ *l2-inv6* ∩ *l2-inv7* ∩ *l2-inv8*))} *trans l1*, *trans l2*
{> *R12s*}
by (*auto 0 3 simp add*: *l1-def l2-def l1-trans-def l2-trans-def*
intro!: *l2-trans-refines-l1-trans*)

lemma *PO-obs-consistent-R12s* [*iff*]:
obs-consistent R12s med12s l1 l2
by (*auto simp add*: *obs-consistent-def R12s-def med12s-def l2-defs*)

lemma *l2-refines-l1* [*iff*]:
refines
(*R12s* ∩
(*reach l1* × (*l2-inv1* ∩ *l2-inv2* ∩ *l2-inv3* ∩ *l2-inv4* ∩ *l2-inv5* ∩ *l2-inv6* ∩ *l2-inv7* ∩ *l2-inv8*)))
med12s l1 l2
by (*rule Refinement-using-invariants*, *auto*)

lemma *l2-implements-l1* [*iff*]:
implements med12s l1 l2
by (*rule refinement-soundness*) (*auto*)

17.4 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*
apply (*rule external-invariant-translation*
[*OF l1-obs-secrecy - l2-implements-l1*])
apply (*auto simp add*: *med12s-def s0-secrecy-def l2-partial-secrecy-def*)
done

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*
apply (*auto simp add*: *oreach-def*, *rule*, *simp-all*, *simp add*: *l2-def l2-trans-def l2-dy-fake-msg-def*)
apply *blast*
done

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*
apply (*auto simp add*:*l2-secrecy-def*)
apply (*drule l2-oreach-dy-fake-msg*, *simp-all*)
apply (*drule l2-obs-partial-secrecy* [*THEN* [*2*] *rev-subsetD*], *simp add*: *l2-partial-secrecy-def*)
apply *blast*
done

lemma *l2-secrecy* [*iff*]: *reach l2* ⊆ *l2-secrecy*
by (*rule external-to-internal-invariant* [*OF l2-obs-secrecy*], *auto*)

abbreviation *l2-iagreement* ≡ *l1-iagreement*

lemma *l2-obs-iagreement* [*iff*]: *oreach l2* ⊆ *l2-iagreement*
apply (*rule external-invariant-translation*
[*OF l1-obs-iagreement - l2-implements-l1*])
apply (*auto simp add*: *med12s-def l1-iagreement-def*)
done

lemma *l2-iagreement* [*iff*]: *reach l2* ⊆ *l2-iagreement*
by (*rule external-to-internal-invariant* [*OF l2-obs-iagreement*], *auto*)

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-frame Rb xpkE = Some KE ∧
implAuth A B ⟨*Number 0*, *KE*⟩ ∈ *ik s* ∧
— actions:
s′ = *s*⦇
progress := (*progress s*)(*Rb* ↦ {*xpkE*, *xsk*}),
ik := {*implAuth B A* (*Aenc* (*NonceF* (*Rb*$*sk*)) *KE*)} ∪ (*ik s*),
signals := *if can-signal s A B then*
addSignal (*signals s*) (*Running A B* ⟨*KE*, *NonceF* (*Rb*$*sk*)⟩)
else
signals s,
secret := {*x*. *x* = *NonceF* (*Rb*$*sk*) ∧ *Rb* = *test*} ∪ *secret s*
⦈
}

definition
l3-step3 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *msg* ⇒ *l3-trans*
where
l3-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* ∧
implAuth B A (*Aenc K* (*epubKF* (*Ra*$*kE*))) ∈ *ik s* ∧
— actions:
s′ = *s*⦇ *progress* := (*progress s*)(*Ra* ↦ {*xpkE*, *xskE*, *xsk*}),
signals := *if can-signal s A B then*
addSignal (*signals s*) (*Commit A B* ⟨*epubKF* (*Ra*$*kE*), *K*⟩)
else
signals s,
secret := {*x*. *x* = *K* ∧ *Ra* = *test*} ∪ *secret s*
⦈
}

specification

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*
by (*auto elim*!: *parts.induct*, *auto simp add*: *ik-init-def*)

lemma *analz-ik-init* [*simp*]: *analz ik-init* = *ik-init*
by (*auto dest*: *analz-into-parts*)

lemma *abs-ik-init* [*iff*]: *abs ik-init* = {}
apply (*auto elim*!: *absE*)
apply (*auto simp add*: *ik-init-def*)

done

lemma *payloadSet-ik-init* [*iff*]: *ik-init* ∩ *payload* = {}
by (*auto simp add*: *ik-init-def*)

lemma *validSet-ik-init* [*iff*]: *ik-init* ∩ *valid* = {}
by (*auto simp add*: *ik-init-def*)

definition
  l3-init :: *l3-state set*
where
  l3-init ≡ { (|
    ik = *ik-init*,
    secret = {},
    progress = *Map.empty*,
    signals = 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 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*
by (*simp add*: *l3-def*)

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*
by (*auto simp add*: *l3-inv1-def*)

lemma *l3-inv1-init* [*iff*]:
  init l3 ⊆ *l3-inv1*
by (*auto simp add*: *l3-def l3-init-def intro*!:*l3-inv1I*)

lemma *l3-inv1-trans* [*iff*]:
  {*l3-inv1*} *trans l3* {> *l3-inv1*}
apply (*auto simp add*: *PO-hoare-defs l3-nostep-defs intro*!: *l3-inv1I*)
apply (*auto simp add*: *l3-defs dy-fake-msg-def dy-fake-chan-def*
        parts-insert [**where** *H*=*ik* -] *parts-insert* [**where** *H*=*insert* - (*ik* -)]
        dest!: *Fake-parts-insert*)
apply (*auto dest*:*analz-into-parts*)
done

lemma *PO-l3-inv1* [*iff*]:
  reach l3 ⊆ *l3-inv1*
by (*rule inv-rule-basic*) (*auto*)

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*
by (*auto simp add*: *l3-def l3-init-defs intro*!:*l3-inv2I Keys-badI*)

lemma *l3-inv2-trans* [*simp*,*intro*!]:
  {*l3-inv2* ∩ *l3-inv1*} *trans l3* {> *l3-inv2*}
apply (*auto simp add*: *PO-hoare-defs l3-nostep-defs intro*!: *l3-inv2I*)
apply (*auto simp add*: *l3-defs dy-fake-msg-def dy-fake-chan-def*)

4 subgoals: dy, lkr*, skr

apply (*auto intro*: *Keys-bad-insert-Fake Keys-bad-insert-keys-of*)
apply (*auto intro*!: *Keys-bad-insert-payload*)
done

lemma *PO-l3-inv2* [*iff*]: *reach l3* ⊆ *l3-inv2*
by (*rule-tac J*=*l3-inv1* **in** *inv-rule-incr*) (*auto*)

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*
apply (*auto simp add*: *l3-def l3-init-def intro*!: *l3-inv3I*)
apply (*auto simp add*: *ik-init-def intro*!: *synth-increasing* [*THEN* [*2*] *rev-subsetD*])
done

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 (*simp add*: *l3-nostep-defs*, *safe*)
  fix *Ra A B*
  show {*l3-inv3*} *l3-step1 Ra A B* {> *l3-inv3*}
    apply (*auto simp add*: *PO-hoare-def l3-defs intro*!: *l3-inv3I dest*!: *l3-inv3D*)

```
    apply (auto intro!: validI dest!: analz-insert-partition [THEN [2] rev-subsetD])
    done
next
  fix Rb A B KE
  show {l3-inv3} l3-step2 Rb A B KE {> l3-inv3}
    apply (auto simp add: PO-hoare-def l3-defs intro!: l3-inv3I dest!: l3-inv3D)
    apply (auto intro!: validI dest!: analz-insert-partition [THEN [2] rev-subsetD])
    done
next
  fix Ra A B K
  show {l3-inv3} l3-step3 Ra A B K {> l3-inv3}
    apply (auto simp add: PO-hoare-def l3-defs intro!: l3-inv3I dest!: l3-inv3D)
    done
next
  fix m
  show {l3-inv3} l3-dy m {> l3-inv3}
    apply (auto simp add: PO-hoare-def l3-defs dy-fake-chan-def dy-fake-msg-def
                intro!: l3-inv3I dest!: l3-inv3D)
    apply (drule synth-analz-insert)
    apply (blast intro: synth-analz-monotone dest: synth-monotone)
    done
next
  fix A
  show {l3-inv3} l3-lkr-others A {> l3-inv3}
    apply (auto simp add: PO-hoare-def l3-defs intro!: l3-inv3I dest!: l3-inv3D)
    apply (drule analz-Un-partition [of - keys-of A], auto)
    done
next
  fix A
  show {l3-inv3} l3-lkr-after A {> l3-inv3}
    apply (auto simp add: PO-hoare-def l3-defs intro!: l3-inv3I dest!: l3-inv3D)
    apply (drule analz-Un-partition [of - keys-of A], auto)
    done
next
  fix R K
  show {l3-inv3} l3-skr R K {> l3-inv3}
    apply (auto simp add: PO-hoare-def l3-defs intro!: l3-inv3I dest!: l3-inv3D)
    apply (auto dest!: analz-insert-partition [THEN [2] rev-subsetD])
    done
qed

lemma PO-l3-inv3 [iff]: reach l3 ⊆ l3-inv3
by (rule inv-rule-basic) (auto)
```

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*
by (*auto simp add*: *l3-def l3-init-def ik-init-def intro*!:*l3-inv4I*)

lemma *l3-inv4-trans* [*simp*,*intro*!]:
  {*l3-inv4*} *trans l3* {> *l3-inv4*}
apply (*auto simp add*: *PO-hoare-defs l3-nostep-defs intro*!: *l3-inv4I*)
apply (*auto simp add*: *l3-defs dy-fake-chan-def dy-fake-msg-def*)
done

lemma *PO-l3-inv4* [*simp*,*intro*!]: *reach l3* ⊆ *l3-inv4*
by (*rule inv-rule-basic*) (*auto*)

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*
by (*auto simp add*: *abs-validSet dy-fake-msg-def intro*!: *l3-inv5I*
          dest!: *l3-inv3D* [*THEN synth-mono*, *THEN* [*2*] *rev-subsetD*]
          dest!: *synth-analz-NI-I-K-synth-analz-NI-E* [*THEN* [*2*] *rev-subsetD*])

lemma *PO-l3-inv5* [*simp*,*intro*!]: *reach l3* ⊆ *l3-inv5*
using *l3-inv5-derived PO-l3-inv2 PO-l3-inv3*
by *blast*

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*
apply (*auto intro*!: *l3-inv6I dest*!: *l3-inv3D*)

1 subgoal

apply (*drule synth-mono*, *simp*, *drule subsetD*, *assumption*)
apply (*auto dest*!: *implInsec-synth-analz* [*rotated 1*, **where** *H*=- ∪ -])
apply (*auto dest*!: *synth-analz-monotone* [*of* - - ∪ - *ik* -])
done

lemma *PO-l3-inv6* [*simp*,*intro*!]: *reach l3* ⊆ *l3-inv6*
using *l3-inv6-derived PO-l3-inv3 PO-l3-inv4*
by (*blast*)

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*
apply (*auto intro*!: *l3-inv7I dest*!: *l3-inv3D*)

1 subgoal

apply (*drule synth-mono*, *simp*, *drule subsetD*, *assumption*)
apply (*auto dest*!: *implConfid-synth-analz* [*rotated 1*, **where** *H*=- ∪ -])
apply (*auto dest*!: *synth-analz-monotone* [*of* - - ∪ - *ik* -])
done

lemma *PO-l3-inv7* [*simp*,*intro*!]: *reach l3* ⊆ *l3-inv7*
using *l3-inv7-derived PO-l3-inv3 PO-l3-inv4*
by (*blast*)

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*
apply (*auto intro*!: *l3-inv8I dest*!: *l3-inv3D l3-inv2D*)

2 subgoals: M is deducible and the agents are bad

apply (*drule synth-mono*, *simp*, *drule subsetD*, *assumption*)
apply (*auto dest*!: *implAuth-synth-analz* [*rotated 1*, **where** *H*=- ∪ -] *elim*!: *synth-analz-monotone*)

apply (*drule synth-mono*, *simp*, *drule subsetD*, *assumption*)
apply (*auto dest*!: *implAuth-synth-analz* [*rotated 1*, **where** *H*=- ∪ -])
done

lemma *PO-l3-inv8* [*iff*]: *reach l3* ⊆ *l3-inv8*
using *l3-inv8-derived*
PO-l3-inv3 PO-l3-inv2 PO-l3-inv4
by *blast*

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*
apply (*auto intro*!: *l3-inv9I dest*!:*l3-inv3D l3-inv2D*)

2 subgoals: M is deducible and the agents are bad

apply (*drule synth-mono*, *simp*, *drule subsetD*, *assumption*)
apply (*auto dest*!: *implSecure-synth-analz* [*rotated 1*, **where** *H*=- ∪ -] *elim*!: *synth-analz-monotone*)

apply (*drule synth-mono*, *simp*, *drule subsetD*, *assumption*)
apply (*auto dest*!: *implSecure-synth-analz* [*rotated 1*, **where** *H*=- ∪ -])
done

lemma *PO-l3-inv9* [*iff*]: *reach l3* ⊆ *l3-inv9*
using *l3-inv9-derived*
  PO-l3-inv3 PO-l3-inv2 PO-l3-inv4
by *blast*

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*
by (*auto simp add*: *R23s-def med23s-def*)

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*
by (*auto simp add*: *R23s-def med23s-def*)

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*
by (*auto simp add*: *R23s-def med23s-def*)

lemma *can-signal-R23* [*simp*]:
(*s2*, *s3*) ∈ *R23s* ⟹
can-signal s2 A B ⟷ *can-signal s3 A B*
by (*auto simp add*: *can-signal-def*)

18.3.1 Protocol events

lemma *l3-step1-refines-step1*:
{*R23s*} *l2-step1 Ra A B*, *l3-step1 Ra A B* {>*R23s*}
apply (*auto simp add*: *PO-rhoare-defs R23s-defs*)
apply (*auto simp add*: *l3-defs l2-step1-def*)
done

lemma *l3-step2-refines-step2*:
{*R23s*} *l2-step2 Rb A B KE*, *l3-step2 Rb A B KE*{>*R23s*}
apply (*auto simp add*: *PO-rhoare-defs R23s-defs l2-step2-def*)
apply (*auto simp add*: *l3-step2-def*)
done

lemma *l3-step3-refines-step3*:
{*R23s*} *l2-step3 Ra A B K*, *l3-step3 Ra A B K* {>*R23s*}
apply (*auto simp add*: *PO-rhoare-defs R23s-defs l2-step3-def*)
apply (*auto simp add*: *l3-step3-def*)

done

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* $\{>R23s\}$
apply (*auto simp add*: *PO-rhoare-defs R23s-defs*)
apply (*auto simp add*: *l3-defs l2-dy-fake-msg-def dest*: *l3-inv5D*)
done

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*
   $\{>R23s\}$
apply (*auto simp add*: *PO-rhoare-defs R23s-defs*, *simp add*: *l2-dy-fake-chan-def*)
apply (*auto simp add*: *l3-defs*)

1 subgoal

apply (*erule valid-cases*, *simp-all add*: *dy-fake-chan-def*)

Insec

  apply (*blast dest*: *l3-inv6D l3-inv5D*)

Confid

  apply (*blast dest*: *l3-inv7D l3-inv5D*)

Auth

  apply (*blast dest*: *l3-inv8D l3-inv5D*)

Secure

  apply (*blast dest*: *l3-inv9D l3-inv5D*)
done

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*
   $\{>R23s\}$
by (*auto dest*: *valid-abs intro*: *l3-dy-valid-refines-dy-fake-chan*)

lemma *l3-dy-isLtKey-refines-skip*:
  $\{R23s\}$ *Id*, *l3-dy* (*LtK ltk*) $\{>R23s\}$
apply (*auto simp add*: *PO-rhoare-defs R23s-defs l3-defs*)
apply (*auto elim*!: *absE*)
done

lemma *l3-dy-others-refines-skip*:
  $[\![M \notin range\ LtK; M \notin valid; M \notin payload]\!] \Longrightarrow$

{*R23s*} *Id*, *l3-dy M* {>*R23s*}
apply (*auto simp add*: *PO-rhoare-defs R23s-defs*)
apply (*auto simp add*: *l3-defs*)
apply (*auto elim*!: *absE intro*: *validI*)
done

lemma *l3-dy-refines-dy-fake-msg-dy-fake-chan-skip*:
{*R23s* ∩ *UNIV* × (*l3-inv5* ∩ *l3-inv6* ∩ *l3-inv7* ∩ *l3-inv8* ∩ *l3-inv9*)}
l2-dy-fake-msg M ∪ (⋃*M′*. *l2-dy-fake-chan M′*) ∪ *Id*, *l3-dy M*
{>*R23s*}
by (*cases M* ∈ *payload* ∪ *valid* ∪ *range LtK*)
(*auto dest*: *l3-dy-payload-refines-dy-fake-msg l3-dy-valid-refines-dy-fake-chan-Un*
intro: *l3-dy-isLtKey-refines-skip dest*!: *l3-dy-others-refines-skip*)

18.3.3 Compromise events

lemma *l3-lkr-others-refines-lkr-others*:
{*R23s*} *l2-lkr-others A*, *l3-lkr-others A* {>*R23s*}
apply (*auto simp add*: *PO-rhoare-defs R23s-defs*)
apply (*auto simp add*: *l3-defs l2-lkr-others-def*)
done

lemma *l3-lkr-after-refines-lkr-after*:
{*R23s*} *l2-lkr-after A*, *l3-lkr-after A* {>*R23s*}
apply (*auto simp add*: *PO-rhoare-defs R23s-defs*)
apply (*auto simp add*: *l3-defs l2-lkr-after-def*)
done

lemma *l3-skr-refines-skr*:
{*R23s*} *l2-skr R K*, *l3-skr R K* {>*R23s*}
apply (*auto simp add*: *PO-rhoare-defs R23s-defs*)
apply (*auto simp add*: *l3-defs l2-skr-def*)
done

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*)
by (*auto simp add*: *R23s-defs l2-defs l3-def l3-init-def*)

lemma *l3-refines-trans-l2* [*iff*]:
{*R23s* ∩ (*UNIV* × (*l3-inv1* ∩ *l3-inv2* ∩ *l3-inv3* ∩ *l3-inv4*))} *trans l2*, *trans l3* {> *R23s*}
proof −
let *?pre′* = *R23s* ∩ (*UNIV* × (*l3-inv5* ∩ *l3-inv6* ∩ *l3-inv7* ∩ *l3-inv8* ∩ *l3-inv9*))
show *?thesis* (**is** {*?pre*} *?t1*, *?t2* {>*?post*})

```
  proof (rule relhoare-conseq-left)
    show ?pre ⊆ ?pre′
      using l3-inv5-derived l3-inv6-derived l3-inv7-derived l3-inv8-derived l3-inv9-derived by blast
  next
    show {?pre′} ?t1, ?t2 {> ?post}
      by (auto simp add: l2-def l3-def l2-trans-def l3-trans-def
               intro!: l3-trans-refines-l2-trans)
  qed
qed

lemma PO-obs-consistent-R23s [iff]:
  obs-consistent R23s med23s l2 l3
by (auto simp add: obs-consistent-def R23s-def med23s-def l2-defs)


lemma l3-refines-l2 [iff]:
  refines
     (R23s ∩
      (reach l2 × (l3-inv1 ∩ l3-inv2 ∩ l3-inv3 ∩ l3-inv4)))
     med23s l2 l3
by (rule Refinement-using-invariants, auto)

lemma l3-implements-l2 [iff]:
  implements med23s l2 l3
by (rule refinement-soundness) (auto)
```

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
by (auto simp add: l3-def l3-init-def ik-init-def intro!:l3-inv10I)

lemma l3-inv10-trans [iff]:
  {l3-inv10} trans l3 {> l3-inv10}
apply (auto simp add: PO-hoare-defs l3-nostep-defs)
apply (auto simp add: l3-defs l3-inv10-def)
done

lemma PO-l3-inv10 [iff]: reach l3 ⊆ l3-inv10
by (rule inv-rule-basic) (auto)
```

lemma *l3-obs-inv10* [*iff*]: *oreach l3* ⊆ *l3-inv10*
by (*auto simp add*: *oreach-def*)

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*
apply (*rule external-invariant-translation* [*OF l2-obs-secrecy - l3-implements-l2*])
apply (*auto simp add*: *med23s-def l2-secrecy-def l3-partial-secrecy-def*)
done

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

lemma *l3-obs-secrecy* [*iff*]: *oreach l3* ⊆ *l3-secrecy*
apply (*rule*, *frule l3-obs-inv5* [*THEN* [*2*] *rev-subsetD*], *frule l3-obs-inv10* [*THEN* [*2*] *rev-subsetD*])
apply (*auto simp add*: *med23s-def l2-secrecy-def l3-secrecy-def s0-secrecy-def l3-inv10-def*)
using *l3-partial-secrecy-def* **apply** (*blast dest*!: *l3-inv5D subsetD* [*OF l3-obs-partial-secrecy*])
done

lemma *l3-secrecy* [*iff*]: *reach l3* ⊆ *l3-secrecy*
by (*rule external-to-internal-invariant* [*OF l3-obs-secrecy*], *auto*)

18.4.4 Injective agreement

abbreviation *l3-iagreement* ≡ *l1-iagreement*

lemma *l3-obs-iagreement* [*iff*]: *oreach l3* ⊆ *l3-iagreement*
apply (*rule external-invariant-translation* [*OF l2-obs-iagreement - l3-implements-l2*])
apply (*auto simp add*: *med23s-def l1-iagreement-def*)
done

lemma *l3-iagreement* [*iff*]: *reach l3* $\subseteq$ *l3-iagreement*
by (*rule external-to-internal-invariant* [*OF l3-obs-iagreement*], *auto*)

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*
by (*unfold-locales*)

end

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

theory *pfslvl3-symmetric*
imports *pfslvl3 Implem-symmetric*
begin

interpretation *pfslvl3-asym*: *pfslvl3 implem-sym*
by (*unfold-locales*)

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* ⟹ *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-Resp-xny [*simp*]:
    role (*guessed-runs R*) = *Resp* ⟹ *guessed-frame R xny* = *Some* (*NonceF* (*R$ny*))
  guessed-frame-Resp-xgny [*simp*]:
    role (*guessed-runs R*) = *Resp* ⟹ *guessed-frame R xgny* = *Some* (*Exp Gen* (*NonceF* (*R$ny*)))
  guessed-frame-xEnd [*simp*]:
    guessed-frame R xEnd = *Some End*
apply (*rule exI* [*of* -
    λ*R*.
      if role (*guessed-runs R*) = *Init then*
        [*xnx* ↦ *NonceF* (*R$nx*), *xgnx* ↦ *Exp Gen* (*NonceF* (*R$nx*)), *xgny* ↦ *End*,
         xsk ↦ *End*, *xEnd* ↦ *End*]
      else
        [*xny* ↦ *NonceF* (*R$ny*), *xgnx* ↦ *End*, *xgny* ↦ *Exp Gen* (*NonceF* (*R$ny*)),
         xsk ↦ *End*, *xEnd* ↦ *End*]],
  auto simp add: *domIff intro*: *role-t.exhaust*)
done

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 ⟹ *R* = *None* ⟹ *False*
by (*fast dest*: *in-progress-Some*)

test-ended s ⟷ 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* ↦ {*xnx*, *xgnx*})
      |)
  }

definition
  l1-step2 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *msg* ⇒ (*'a l1-state-scheme* ∗ *'a l1-state-scheme*) *set*
where
  l1-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$ny*))) ∧
    — actions:
    s' = *s*(| *progress* := (*progress s*)(*Rb* ↦ {*xny*, *xgny*, *xgnx*, *xsk*}),
          signalsInit := *if can-signal s A B then*
                addSignal (*signalsInit s*) (*Running A B* (*Exp gnx* (*NonceF* (*Rb$ny*))))
              else
                signalsInit s
        |)
  }

definition
  l1-step3 :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *msg* ⇒ (*'a l1-state-scheme* ∗ *'a l1-state-scheme*) *set*
where
  l1-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$nx*))) ∧
    (*can-signal s A B* ⟶ — authentication guard
      (∃ *Rb*. *guessed-runs Rb* = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|) ∧
          in-progressS (*progress s Rb*) {*xny*, *xgnx*, *xgny*, *xsk*} ∧
          guessed-frame Rb xgny = *Some gny* ∧
          guessed-frame Rb xgnx = *Some* (*Exp Gen* (*NonceF* (*Ra$nx*))))) ∧
    (*Ra* = *test* ⟶ *Exp gny* (*NonceF* (*Ra$nx*)) ∉ *synth* (*analz* (*ik s*))) ∧

    — actions:
    s' = *s*(| *progress* := (*progress s*)(*Ra* ↦ {*xnx*, *xgnx*, *xgny*, *xsk*, *xEnd*}),
          secret := {*x*. *x* = *Exp gny* (*NonceF* (*Ra$nx*)) ∧ *Ra* = *test*} ∪ *secret s*,
          signalsInit := *if can-signal s A B then*
              addSignal (*signalsInit s*) (*Commit A B* (*Exp gny* (*NonceF* (*Ra$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
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 = *λx*. *0*,
signalsResp = *λx*. *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*
by (*simp add*: *l1-def*)

lemma *run-ended-trans*:
run-ended (*progress s R*) ⟹
(*s*, *s'*) ∈ *trans l1* ⟹
run-ended (*progress s' R*)
apply (*auto simp add*: *l1-nostep-defs*)
apply (*auto simp add*: *l1-defs ik-dy-def*)
done

lemma *can-signal-trans*:
can-signal s' A B ⟹
(*s*, *s'*) ∈ *trans l1* ⟹
can-signal s A B
by (*auto simp add*: *can-signal-def run-ended-trans*)

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*}
apply (*simp add*: *PO-rhoare-defs R01s-def*)
apply *auto*
apply (*simp add*: *l1-defs s0-defs s0-secrecy-def*)
done

Protocol events.

lemma *l1-step1-refines-skip*:
  {*R01s*} *Id*, *l1-step1 Ra A B* {>*R01s*}
by (*auto simp add*: *PO-rhoare-defs R01s-def l1-step1-def*)

lemma *l1-step2-refines-skip*:
  {*R01s*} *Id*, *l1-step2 Rb A B gnx* {>*R01s*}
apply (*auto simp add*: *PO-rhoare-defs R01s-def*)
apply (*auto simp add*: *l1-step2-def*)
done

lemma *l1-step3-refines-add-secret-skip*:
  {*R01s*} *s0-add-secret* (*Exp gny* (*NonceF* (*Ra$nx*))) ∪ *Id*, *l1-step3 Ra A B gny* {>*R01s*}
apply (*auto simp add*: *PO-rhoare-defs R01s-def s0-add-secret-def*)
apply (*auto simp add*: *l1-step3-def*)
done

lemma *l1-step4-refines-add-secret-skip*:
  {*R01s*} *s0-add-secret* (*Exp gnx* (*NonceF* (*Rb$ny*))) ∪ *Id*, *l1-step4 Rb A B gnx* {>*R01s*}
apply (*auto simp add*: *PO-rhoare-defs R01s-def s0-add-secret-def*)
apply (*auto simp add*: *l1-step4-def*)
done

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*)
by (*auto simp add*: *R01s-def s0-defs l1-defs s0-secrecy-def*)

lemma *l1-refines-trans-s0* [*iff*]:
  {*R01s*} *trans s0*, *trans l1* {> *R01s*}
by (*auto simp add*: *s0-def l1-def s0-trans-def l1-trans-def*
        intro: *l1-trans-refines-s0-trans*)

lemma *obs-consistent-med01x* [*iff*]:
  obs-consistent R01s med01s s0 l1
by (*auto simp add*: *obs-consistent-def R01s-def med01s-def*)

Refinement result.

lemma *l1s-refines-s0* [*iff*]:
  refines
    R01s
    med01s s0 l1
by (*auto simp add*:*refines-def PO-refines-def*)

lemma *l1-implements-s0* [*iff*]: *implements med01s s0 l1*
by (*rule refinement-soundness*) (*fast*)

21.3 Derived invariants: secrecy

abbreviation *l1-secrecy* ≡ *s0-secrecy*

lemma *l1-obs-secrecy* [*iff*]: *oreach l1* ⊆ *l1-secrecy*
apply (*rule external-invariant-translation*
        [*OF s0-obs-secrecy - l1-implements-s0*])
apply (*auto simp add*: *med01s-def s0-secrecy-def*)
done

lemma *l1-secrecy* [*iff*]: *reach l1* ⊆ *l1-secrecy*
by (*rule external-to-internal-invariant* [*OF l1-obs-secrecy*], *auto*)

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*
by (*auto simp add*: *l1-def l1-init-def l1-inv1-def*)

lemma *l1-inv1-trans* [*iff*]:
{*l1-inv1*} *trans l1* {> *l1-inv1*}
apply (*auto simp add*: *PO-hoare-defs l1-nostep-defs intro*!: *l1-inv1I*)
apply (*auto simp add*: *l1-defs ik-dy-def l1-inv1-def dest*: *Exp-Exp-Gen-inj2* [*OF sym*])
done

lemma *PO-l1-inv1* [*iff*]: *reach l1* ⊆ *l1-inv1*
by (*rule inv-rule-basic*) (*auto*)

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*
by (*auto simp add*: *l1-def l1-init-def l1-inv2-def*)

lemma *l1-inv2-trans* [*iff*]:
{*l1-inv2*} *trans l1* {> *l1-inv2*}
apply (*auto simp add*: *PO-hoare-defs intro*!: *l1-inv2I*)
apply (*drule can-signal-trans*, *assumption*)
apply (*auto simp add*: *l1-nostep-defs*)
apply (*auto simp add*: *l1-defs ik-dy-def l1-inv2-def*)
done

lemma *PO-l1-inv2* [*iff*]: *reach l1* ⊆ *l1-inv2*
by (*rule inv-rule-basic*) (*auto*)

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*
apply (*auto intro*!: *l1-inv3I*)
apply (*auto dest*!: *l1-inv2D*)
apply (*rename-tac x A B Rb Ra*)
apply (*case-tac*
signalsInit x (*Commit A B* (*Exp* (*Exp Gen* (*NonceF* (*Rb* $ *ny*))) (*NonceF* (*Ra* $ *nx*)))) > *0*, *auto*)
apply (*fastforce dest*: *l1-inv1D elim*: *equalityE*)
done

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*
by (*auto simp add*: *l1-def l1-init-def l1-inv4-def*)

declare *domIff* [*iff*]

lemma *l1-inv4-trans* [*iff*]:
  {*l1-inv4*} *trans l1* {> *l1-inv4*}
apply (*auto simp add*: *PO-hoare-defs l1-nostep-defs intro*!: *l1-inv4I*)
apply (*auto simp add*: *l1-inv4-def l1-defs ik-dy-def dest*: *Exp-Exp-Gen-inj2* [*OF sym*])
done

declare *domIff* [*iff del*]

lemma *PO-l1-inv4* [*iff*]: *reach l1* ⊆ *l1-inv4*
by (*rule inv-rule-basic*) (*auto*)

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*
by (*auto simp add*: *l1-def l1-init-def l1-inv5-def*)

lemma *l1-inv5-trans* [*iff*]:
  {*l1-inv5*} *trans l1* {> *l1-inv5*}
apply (*auto simp add*: *PO-hoare-defs intro*!: *l1-inv5I*)
apply (*drule can-signal-trans*, *assumption*)
apply (*auto simp add*: *l1-nostep-defs*)
apply (*auto simp add*: *l1-defs ik-dy-def l1-inv5-def*)
done

lemma *PO-l1-inv5* [*iff*]: *reach l1* ⊆ *l1-inv5*
by (*rule inv-rule-basic*) (*auto*)

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 (auto intro!: l1-inv6I)
  fix s::l1-state fix A B Rb Ra
  assume HRun:guessed-runs Ra = (|role = Init, owner = A, partner = B|)
              in-progressS (progress s Ra) {xnx, xgnx, xgny, xsk, xEnd}
              guessed-frame Ra xgny = Some (Exp Gen (NonceF (Rb $ ny)))
              can-signal s A B
  assume HRb: progress s Rb = Some {xny, xgnx, xgny, xsk}
  assume I4:s ∈ l1-inv4
  assume I5:s ∈ l1-inv5
  from I4 HRb
  have signalsResp s (Commit A B (Exp (Exp Gen (NonceF (Rb $ ny))) (NonceF (Ra $ nx)))) > 0
       =⇒ False
    proof (auto dest!: l1-inv4D)
      assume {xny, xgnx, xgny, xsk, xEnd} = {xny, xgnx, xgny, xsk}
      thus ?thesis by force
    qed
  then have
    HC: signalsResp s (Commit A B (Exp (Exp Gen (NonceF (Rb $ ny))) (NonceF (Ra $ nx)))) = 0
    by auto
  from I5 HRun
  have signalsResp s (Running A B (Exp (Exp Gen (NonceF (Rb $ ny))) (NonceF (Ra $ nx)))) > 0
    by (auto dest!: l1-inv5D)
  with HC show
     signalsResp s (Commit A B (Exp (Exp Gen (NonceF (Rb $ ny))) (NonceF (Ra $ nx))))
    < signalsResp s (Running A B (Exp (Exp Gen (NonceF (Rb $ ny))) (NonceF (Ra $ nx))))
    by auto
qed
```

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*}
apply (*auto simp add*: *PO-rhoare-defs R01iai-def*)
apply (*simp add*: *l1-learn-def*)
done

Protocol events.

lemma *l1-step1-refines-a0i-skip-i*:
{*R01iai*} *Id*, *l1-step1 Ra A B* {>*R01iai*}
by (*auto simp add*: *PO-rhoare-defs R01iai-def l1-step1-def*)

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*}
by (*auto simp add*: *PO-rhoare-defs R01iai-def*, *simp-all add*: *l1-step2-def a0i-running-def*, *auto*)

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*}
apply (*auto simp add*: *PO-rhoare-defs R01iai-def*)
apply (*auto simp add*: *l1-step3-def a0i-commit-def*)
apply (*force elim*!: *l1-inv3E*)+
done

lemma *l1-step4-refines-a0i-skip-i*:
{*R01iai*} *Id*, *l1-step4 Rb A B gnx* {>*R01iai*}
by (*auto simp add*: *PO-rhoare-defs R01iai-def*, *auto simp add*: *l1-step4-def*)

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*)

by (*auto simp add*: *R01iai-def a0i-defs l1-defs*)

lemma *l1-refines-trans-a0i-i* [*iff*]:
{*R01iai* ∩ (*UNIV* × (*l1-inv1* ∩ *l1-inv2*))} *trans a0i*, *trans l1* {> *R01iai*}
proof −
let *?pre′* = *R01iai* ∩ (*UNIV* × *l1-inv3*)
show *?thesis* (**is** {*?pre*} *?t1*, *?t2* {>*?post*})
proof (*rule relhoare-conseq-left*)
show *?pre* ⊆ *?pre′*
using *l1-inv3-derived* **by** *blast*
next
show {*?pre′*} *?t1*, *?t2* {> *?post*}
apply (*auto simp add*: *a0i-def l1-def a0i-trans-def l1-trans-def*)
prefer *2* **using** *l1-step2-refines-a0i-running-skip-i* **apply** (*simp add*: *PO-rhoare-defs*, *blast*)
prefer *2* **using** *l1-step3-refines-a0i-commit-skip-i* **apply** (*simp add*: *PO-rhoare-defs*, *blast*)
apply (*blast intro*!:*l1-trans-refines-a0i-trans-i*)+
done
qed
qed

lemma *obs-consistent-med01iai* [*iff*]:
obs-consistent R01iai med01iai a0i l1
by (*auto simp add*: *obs-consistent-def R01iai-def med01iai-def*)

Refinement result.

lemma *l1-refines-a0i-i* [*iff*]:
refines
(*R01iai* ∩ (*reach a0i* × (*l1-inv1* ∩ *l1-inv2*)))
med01iai a0i l1
by (*rule Refinement-using-invariants*, *auto*)

lemma *l1-implements-a0i-i* [*iff*]: *implements med01iai a0i l1*
by (*rule refinement-soundness*) (*fast*)

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*
apply (*rule external-invariant-translation*
[*OF PO-a0i-obs-agreement - l1-implements-a0i-i*])
apply (*auto simp add*: *med01iai-def l1-iagreement-Init-def a0i-agreement-def*)

done

lemma *l1-iagreement-Init* [*iff*]: *reach l1* ⊆ *l1-iagreement-Init*
by (*rule external-to-internal-invariant* [*OF l1-obs-iagreement-Init*], *auto*)

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*}
apply (*auto simp add*: *PO-rhoare-defs R01iar-def*)
apply (*simp add*: *l1-learn-def*)
done

Protocol events.

lemma *l1-step1-refines-a0i-skip-r*:
  {*R01iar*} *Id*, *l1-step1 Ra A B* {>*R01iar*}
by (*auto simp add*: *PO-rhoare-defs R01iar-def l1-step1-def*)

lemma *l1-step2-refines-a0i-skip-r*:
  {*R01iar*} *Id*, *l1-step2 Rb A B gnx* {>*R01iar*}
by (*auto simp add*: *PO-rhoare-defs R01iar-def*, *auto simp add*:*l1-step2-def*)

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*}
by (*auto simp add*: *PO-rhoare-defs R01iar-def*, *simp-all add*: *l1-step3-def a0i-running-def*, *auto*)

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*}
apply (*auto simp add*: *PO-rhoare-defs R01iar-def*)
apply (*auto simp add*: *l1-step4-def a0i-commit-def*)
apply (*auto dest*!: *l1-inv6D* [*rotated 1*])
done

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*)
by (*auto simp add*: *R01iar-def a0i-defs l1-defs*)

lemma *l1-refines-trans-a0i-r* [*iff*]:
  {*R01iar* ∩ (*UNIV* × (*l1-inv4* ∩ *l1-inv5*))} *trans a0i*, *trans l1* {> *R01iar*}
proof −
  let *?pre′* = *R01iar* ∩ (*UNIV* × *l1-inv6*)
  show *?thesis* (**is** {*?pre*} *?t1*, *?t2* {>*?post*})
  proof (*rule relhoare-conseq-left*)
    show *?pre* ⊆ *?pre′*
      using *l1-inv6-derived* **by** *blast*
  next
    show {*?pre′*} *?t1*, *?t2* {> *?post*}
      apply (*auto simp add*: *a0i-def l1-def a0i-trans-def l1-trans-def*)
      prefer *3* **using** *l1-step3-refines-a0i-running-skip-r* **apply** (*simp add*: *PO-rhoare-defs*, *blast*)
      prefer *3* **using** *l1-step4-refines-a0i-commit-skip-r* **apply** (*simp add*: *PO-rhoare-defs*, *blast*)
      apply (*blast intro*!:*l1-trans-refines-a0i-trans-r*)+
      done
  qed
qed

lemma *obs-consistent-med01iar* [*iff*]:
  obs-consistent R01iar med01iar a0i l1
by (*auto simp add*: *obs-consistent-def R01iar-def med01iar-def*)

Refinement result.

lemma *l1-refines-a0i-r* [*iff*]:
  refines
    (*R01iar* ∩ (*reach a0i* × (*l1-inv4* ∩ *l1-inv5*)))
    med01iar a0i l1
by (*rule Refinement-using-invariants*, *auto*)

lemma *l1-implements-a0i-r* [*iff*]: *implements med01iar a0i l1*
by (*rule refinement-soundness*) (*fast*)

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*
apply (*rule external-invariant-translation*
[*OF PO-a0i-obs-agreement - l1-implements-a0i-r*])
apply (*auto simp add*: *med01iar-def l1-iagreement-Resp-def a0i-agreement-def*)
done

lemma *l1-iagreement-Resp* [*iff*]: *reach l1* ⊆ *l1-iagreement-Resp*
by (*rule external-to-internal-invariant* [*OF l1-obs-iagreement-Resp*], *auto*)

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: $test\text{-}owner \notin bad\text{-}init \wedge test\text{-}partner \notin bad\text{-}init$
by *auto*

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
$m \in dy\text{-}fake\text{-}msg\ (bad\ s)\ (ik\ s)\ (chan\ s) \wedge$
— actions
$s' = s(\!|ik := \{m\} \cup ik\ s|\!)$
}

definition
l2-dy-fake-chan :: *chan* ⇒ *l2-trans*
where
l2-dy-fake-chan M ≡ {(*s*,*s'*).
— guards
$M \in dy\text{-}fake\text{-}chan\ (bad\ s)\ (ik\ s)\ (chan\ s) \wedge$
— actions
$s' = s(\!|chan := \{M\} \cup chan\ s|\!)$

}

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*
by (*cases x*, *auto*)

lemma *role-comp-inv-eq*:
y = *role-comp x* ⟷ *x* = *role-comp y*
by (*auto elim*!: *role-comp.elims* [*OF sym*])

definition
partners :: *rid-t set*
where
partners ≡ {*R*. *partner-runs test R*}

lemma *test-not-partner* [*simp*]:
test ∉ *partners*
by (*auto simp add*: *partners-def partner-runs-def*, *cases role* (*guessed-runs test*), *auto*)

lemma *matching-symmetric*:
matching sigma sigma′ ⟹ *matching sigma′ sigma*
by (*auto simp add*: *matching-def*)

lemma *partner-symmetric*:
partner-runs R R′ ⟹ *partner-runs R′ R*
by (*auto simp add*: *partner-runs-def matching-symmetric*)

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-runs R R'' ⟹ partner-runs R R' ⟹ R' = R''
proof –
  assume H':partner-runs R R'
  then have Hm': matching (guessed-frame R) (guessed-frame R')
    by (auto simp add: partner-runs-def)
  assume H'':partner-runs R R''
  then have Hm'': matching (guessed-frame R) (guessed-frame R'')
    by (auto simp add: partner-runs-def)
  show ?thesis
    proof (cases role (guessed-runs R'))
      case Init
      with H' partner-symmetric [OF H''] have Hrole:role (guessed-runs R) = Resp
                                                role (guessed-runs R'') = Init
        by (auto simp add: partner-runs-def)
      with Init Hm' have guessed-frame R xgnx = Some (Exp Gen (NonceF (R'$nx)))
        by (simp add: matching-def)
      moreover from Hrole Hm'' have guessed-frame R xgnx = Some (Exp Gen (NonceF (R''$nx)))
        by (simp add: matching-def)
      ultimately show ?thesis by (auto dest: Exp-Gen-inj)
    next
      case Resp
      with H' partner-symmetric [OF H''] have Hrole:role (guessed-runs R) = Init
                                                role (guessed-runs R'') = Resp
        by (auto simp add: partner-runs-def)
      with Resp Hm' have guessed-frame R xgny = Some (Exp Gen (NonceF (R'$ny)))
        by (simp add: matching-def)
      moreover from Hrole Hm'' have guessed-frame R xgny = Some (Exp Gen (NonceF (R''$ny)))
        by (simp add: matching-def)
      ultimately show ?thesis by (auto dest: Exp-Gen-inj)
    qed
qed

lemma partner-test:
  R ∈ partners ⟹ partner-runs R R' ⟹ R' = test
by (auto intro!:partner-unique simp add:partners-def partner-symmetric)
```

Compromising events.

```
definition
  l2-lkr-others :: agent ⇒ l2-trans
where
  l2-lkr-others A ≡ {(s,s').
    — guards
    A ≠ test-owner ∧
    A ≠ test-partner ∧
    — actions
    s' = s(|bad := {A} ∪ bad s|)
  }

definition
  l2-lkr-actor :: agent ⇒ l2-trans
where
  l2-lkr-actor A ≡ {(s,s').
    — guards
```

$A = \textit{test-owner} \wedge$
$A \neq \textit{test-partner} \wedge$
— actions
$s' = s(\!| \textit{bad} := \{A\} \cup \textit{bad } s |\!)$
}

definition
l2-lkr-after :: *agent* ⇒ *l2-trans*
where
l2-lkr-after $A \equiv \{(s,s')$.
— guards
test-ended $s \wedge$
— actions
$s' = s(\!| \textit{bad} := \{A\} \cup \textit{bad } s |\!)$
}

definition
l2-skr :: *rid-t* ⇒ *msg* ⇒ *l2-trans*
where
l2-skr $R\ K \equiv \{(s,s')$.
— guards
$R \neq \textit{test} \wedge R \notin \textit{partners} \wedge$
in-progress (*progress* $s\ R$) *xsk* $\wedge$
guessed-frame R *xsk* = *Some* $K \wedge$
— actions
$s' = s(\!| \textit{ik} := \{K\} \cup \textit{ik } s |\!)$
}

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* $A\ B \equiv \{(s,\ s')$.
— guards:
$Ra \notin \textit{dom}\ (\textit{progress}\ s) \wedge$
guessed-runs *Ra* = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) $\wedge$
— 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*
by (*simp add*: *l2-def*)

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*)
apply (*auto simp add*: *l2-nostep-defs*)
apply (*auto simp add*: *l2-defs*)
done
declare *domIff* [*iff del*]

lemma *l2-can-signal-trans*:
can-signal s' A B ⟹
(*s*, *s'*) ∈ *trans l2* ⟹
can-signal s A B
by (*auto simp add*: *can-signal-def l2-run-ended-trans*)

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*
by (*auto simp add*: *l2-def l2-init-def l2-inv1-def can-signal-def bad-init-spec*)

lemma *l2-inv1-trans* [*iff*]:
{*l2-inv1*} *trans l2* {> *l2-inv1*}
proof (*auto simp add*: *PO-hoare-defs intro*!: *l2-inv1I del*: *conjI*)
fix *s′ s* :: *l2-state*
fix *A B*
assume *HI*:*s* ∈ *l2-inv1*
assume *HT*:(*s*, *s′*) ∈ *trans l2*
assume *can-signal s′ A B*
with *HT* **have** *HS*:*can-signal s A B*
by (*auto simp add*: *l2-can-signal-trans*)
with *HI* **have** *A* ∉ *bad s* ∧ *B* ∉ *bad s*
by *fast*
with *HS HT* **show** *A* ∉ *bad s′* ∧ *B* ∉ *bad s′*
by (*auto simp add*: *l2-nostep-defs can-signal-def*)
(*simp-all add*: *l2-defs*)
qed

lemma *PO-l2-inv1* [*iff*]: *reach l2* ⊆ *l2-inv1*
by (*rule inv-rule-basic*) (*auto*)

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*
by (*auto simp add*: *l2-def l2-init-def l2-inv2-def*)

lemma *l2-inv2-trans* [*iff*]:
{*l2-inv2*} *trans l2* {> *l2-inv2*}
apply (*auto simp add*: *PO-hoare-defs l2-nostep-defs intro*!: *l2-inv2I*)

apply (*auto simp add*: *l2-defs dy-fake-chan-def*)
apply *force*+
done

lemma *PO-l2-inv2* [*iff*]: *reach l2* ⊆ *l2-inv2*
by (*rule inv-rule-basic*) (*auto*)

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*
by (*auto simp add*: *l2-def l2-init-def l2-inv3-def*)

lemma *l2-inv3-trans* [*iff*]:
{*l2-inv3*} *trans l2* {> *l2-inv3*}
apply (*auto simp add*: *PO-hoare-defs l2-nostep-defs intro*!: *l2-inv3I*)
apply (*auto simp add*: *l2-defs dy-fake-chan-def*)
apply (*simp-all add*: *domIff insert-ident*)
apply *force*+
done

lemma *PO-l2-inv3* [*iff*]: *reach l2* ⊆ *l2-inv3*
by (*rule inv-rule-basic*) (*auto*)

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 ⟶
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 ⊆ *l2-inv4*
by (*auto simp add*: *l2-def l2-init-def l2-inv4-def*)

lemma *l2-inv4-trans* [*iff*]:
{*l2-inv4*} *trans l2* {> *l2-inv4*}
apply (*auto simp add*: *PO-hoare-defs l2-nostep-defs intro*!: *l2-inv4I*)
apply (*auto simp add*: *l2-defs dy-fake-chan-def*)
done

lemma *PO-l2-inv4* [*iff*]: *reach l2* ⊆ *l2-inv4*
by (*rule inv-rule-basic*) (*auto*)

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*|) ⟶
in-progress (*progress s Rb*) *xsk* ⟶
guessed-frame Rb xgnx = Some gnx ⟶
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 ⊆ *l2-inv4'*
by (*auto simp add*: *l2-def l2-init-def l2-inv4'-def*)

lemma *l2-inv4'-trans* [*iff*]:
{*l2-inv4'*} *trans l2* {> *l2-inv4'*}
apply (*auto simp add*: *PO-hoare-defs l2-nostep-defs intro*!: *l2-inv4'I*)
apply (*auto simp add*: *l2-defs dy-fake-chan-def*)
done

lemma *PO-l2-inv4'* [*iff*]: *reach l2* ⊆ *l2-inv4'*
by (*rule inv-rule-basic*) (*auto*)

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*
by (*auto simp add*: *l2-def l2-init-def l2-inv5-def*)

lemma *l2-inv5-trans* [*iff*]:
{*l2-inv5*} *trans l2* {> *l2-inv5*}
apply (*auto simp add*: *PO-hoare-defs l2-nostep-defs intro*!: *l2-inv5I*)
apply (*auto simp add*: *l2-defs dy-fake-chan-def intro*: *l2-inv5D dy-fake-msg-monotone*)
done

lemma *PO-l2-inv5* [*iff*]: *reach l2* ⊆ *l2-inv5*
by (*rule inv-rule-basic*) (*auto*)

22.2.7 inv6

For a run R (with any role), the session key always has the form $\mathit{something}^n$ 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*
by (*auto simp add*: *l2-def l2-init-def l2-inv6-def*)

lemma *l2-inv6-trans* [*iff*]:
{*l2-inv6*} *trans l2* {> *l2-inv6*}
apply (*auto simp add*: *PO-hoare-defs l2-nostep-defs intro*!: *l2-inv6I*)

apply (*auto simp add*: *l2-defs dy-fake-chan-def dest*: *l2-inv6D*)
done

lemma *PO-l2-inv6* [*iff*]: *reach l2* ⊆ *l2-inv6*
by (*rule inv-rule-basic*) (*auto*)

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*
by (*rule*, *rule*, *erule analz.induct*, *auto*)

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*
by (*auto simp add*: *l2-def l2-init-def l2-inv7-def*)

lemma *l2-inv7-step1*:
{*l2-inv7*} *l2-step1 Ra A B* {> *l2-inv7*}
apply (*auto simp add*: *PO-hoare-defs l2-defs intro*!: *l2-inv7I*)
apply (*auto intro*: *synth-analz-increasing*)
done

lemma *l2-inv7-step2*:
{*l2-inv7*} *l2-step2 Rb A B gnx* {> *l2-inv7*}
apply (*auto simp add*: *PO-hoare-defs l2-nostep-defs intro*!: *l2-inv7I*, *auto simp add*: *l2-defs*)
apply (*auto intro*: *synth-analz-increasing*)
done

lemma *l2-inv7-step3*:
{*l2-inv7*} *l2-step3 Ra A B gny* {> *l2-inv7*}
apply (*auto simp add*: *PO-hoare-defs l2-nostep-defs intro*!: *l2-inv7I*, *auto simp add*: *l2-defs*)
apply (*auto intro*: *synth-analz-increasing*)
apply (*auto dest*:*l2-inv7D* [*THEN* [*2*] *rev-subsetD*] *dest*!:*extr-Chan*)
done

lemma *l2-inv7-step4*:

{l2-inv7} l2-step4 Rb A B gnx {> l2-inv7}
by (*auto simp add*: *PO-hoare-defs l2-nostep-defs intro*!: *l2-inv7I*, *auto simp add*: *l2-defs*)

lemma *l2-inv7-dy-fake-msg*:
{l2-inv7} l2-dy-fake-msg M {> l2-inv7}
by (*auto simp add*: *PO-hoare-defs l2-defs extr-insert-IK-eq*
intro!: *l2-inv7I*
elim!: *l2-inv7E dy-fake-msg-extr* [*THEN* [*2*] *rev-subsetD*])

lemma *l2-inv7-dy-fake-chan*:
{l2-inv7} l2-dy-fake-chan M {> l2-inv7}
by (*auto simp add*: *PO-hoare-defs l2-defs*
intro!: *l2-inv7I*
dest: *dy-fake-chan-extr-insert* [*THEN* [*2*] *rev-subsetD*]
elim!: *l2-inv7E dy-fake-msg-extr* [*THEN* [*2*] *rev-subsetD*])

lemma *l2-inv7-lkr-others*:
{l2-inv7 ∩ l2-inv5} l2-lkr-others A {> l2-inv7}
apply (*auto simp add*: *PO-hoare-defs l2-defs*
intro!: *l2-inv7I*
dest!: *extr-insert-bad* [*THEN* [*2*] *rev-subsetD*]
elim!: *l2-inv7E l2-inv5E*)
apply (*auto dest*: *dy-fake-msg-extr* [*THEN* [*2*] *rev-subsetD*])
done

lemma *l2-inv7-lkr-after*:
{l2-inv7 ∩ l2-inv5} l2-lkr-after A {> l2-inv7}
apply (*auto simp add*: *PO-hoare-defs l2-defs*
intro!: *l2-inv7I*
dest!: *extr-insert-bad* [*THEN* [*2*] *rev-subsetD*]
elim!: *l2-inv7E l2-inv5E*)
apply (*auto dest*: *dy-fake-msg-extr* [*THEN* [*2*] *rev-subsetD*])
done

lemma *l2-inv7-skr*:
{l2-inv7 ∩ l2-inv6} l2-skr R K {> l2-inv7}

apply (*auto simp add*: *PO-hoare-defs l2-defs intro*!: *l2-inv7I*)
apply (*auto simp add*: *extr-insert-IK-eq dest*!: *l2-inv6D*)
apply (*auto intro*: *synth-analz-increasing*)
done

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}
by (*auto simp add*: *l2-nostep-defs intro*:*l2-inv7-trans-aux*)

lemma *PO-l2-inv7* [*iff*]: *reach l2* ⊆ *l2-inv7*
by (*rule-tac J*=*l2-inv5* ∩ *l2-inv6* **in** *inv-rule-incr*) (*auto*)

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*
by (*auto simp add*: *l2-def l2-init-def l2-inv8-def*)

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 (*auto simp add*: *PO-hoare-defs intro*!: *l2-inv8I*)
  fix *s s'* :: *l2-state* **fix** *x*
  assume *Hi*:*s* ∈ *l2-inv1 s* ∈ *l2-inv8 s* ∈ *l2-inv2 s* ∈ *l2-inv4'*
  assume *Ht*:(*s*, *s'*) ∈ *l2-step3 Ra A B gny*
  assume *Hs*:*x* ∈ *secret s'*
  from *Hs Ht* **have** *x* ∈ *secret s* ∨ (*Ra* = *test* ∧ *x* = *Exp gny* (*NonceF* (*Ra*$*nx*)))
    by (*auto simp add*: *l2-defs*)
  with *Hi Ht*
  show ∃ *N N' R'*. *x* = *Exp* (*Exp Gen* (*NonceF* (*R'* $ *N'*))) (*NonceF* (*test* $ *N*)) ∧ *R'* ∈ *partners*
    proof (*auto dest*: *l2-inv8D simp add*: *l2-defs*)
      assume *Hx*: *x* = *Exp gny* (*NonceF* (*test* $ *nx*))
      assume *can-signal s A B*
      with *Hi* **have** *HA*: *A* ∉ *bad s* ∧ *B* ∉ *bad s*
        by *auto*
      assume *Htest*: *guessed-runs test* = (|*role* = *Init*, *owner* = *A*, *partner* = *B*|)
        guessed-frame test xgny = *Some gny*
        guessed-frame test xsk = *Some* (*Exp gny* (*NonceF* (*test* $ *nx*)))
      assume *Auth B A* ⟨*Number 0*, *gny*, *Exp Gen* (*NonceF* (*test* $ *nx*))⟩ ∈ *chan s*
      with *Hi HA* **obtain** *Rb* **where** *HRb*:
        guessed-runs Rb = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|)
        in-progressS (*progress s Rb*) {*xny*, *xgnx*, *xgny*, *xsk*}
        gny = *Exp Gen* (*NonceF* (*Rb*$*ny*))
        guessed-frame Rb xgnx = *Some* (*Exp Gen* (*NonceF* (*test*$*nx*)))
        by (*auto dest*!: *l2-inv2D*)
      with *Hi*
      have *guessed-frame Rb xsk* = *Some* (*Exp* (*Exp Gen* (*NonceF* (*Rb*$*ny*))) (*NonceF* (*test*$*nx*)))

by (*auto dest*: *l2-inv4′D*)
with *HRb Htest* **have** *Rb* ∈ *partners*
by (*auto simp add*: *partners-def partner-runs-def matching-def*)
with *HRb* **have** *Exp gny* (*NonceF* (*test* \$ *nx*)) =
Exp (*Exp Gen* (*NonceF* (*Rb* \$ *ny*))) (*NonceF* (*test* \$ *nx*)) ∧ *Rb* ∈ *partners*
by *auto*
then show ∃ *N N′ R′*.
Exp gny (*NonceF* (*test* \$ *nx*)) = *Exp* (*Exp Gen* (*NonceF* (*R′* \$ *N′*))) (*NonceF* (*test* \$ *N*)) ∧
R′ ∈ *partners*
by *blast*
qed (*auto simp add*: *can-signal-def*)
qed

lemma *l2-inv8-step4*:
{*l2-inv8* ∩ *l2-inv1* ∩ *l2-inv3* ∩ *l2-inv4* ∩ *l2-inv4′*} *l2-step4 Rb A B gnx* {> *l2-inv8*}
proof (*auto simp add*: *PO-hoare-defs intro*!: *l2-inv8I*)
fix *s s′* :: *l2-state* **fix** *x*
assume *Hi*:*s* ∈ *l2-inv1 s* ∈ *l2-inv8 s* ∈ *l2-inv3 s* ∈ *l2-inv4 s* ∈ *l2-inv4′*
assume *Ht*:(*s*, *s′*) ∈ *l2-step4 Rb A B gnx*
assume *Hs*:*x* ∈ *secret s′*
from *Hs Ht* **have** *x* ∈ *secret s* ∨ (*Rb* = *test* ∧ *x* = *Exp gnx* (*NonceF* (*Rb*\$*ny*)))
by (*auto simp add*: *l2-defs*)
with *Hi Ht*
show ∃ *N N′ R′*. *x* = *Exp* (*Exp Gen* (*NonceF* (*R′* \$ *N′*))) (*NonceF* (*test* \$ *N*)) ∧ *R′* ∈ *partners*
proof (*auto dest*: *l2-inv8D simp add*: *l2-defs*)
assume *Hx*: *x* = *Exp gnx* (*NonceF* (*test* \$ *ny*))
assume *can-signal s A B*
with *Hi* **have** *HA*: *A* ∉ *bad s* ∧ *B* ∉ *bad s*
by *auto*
assume *Htest*: *guessed-runs test* = (|*role* = *Resp*, *owner* = *B*, *partner* = *A*|)
guessed-frame test xgnx = *Some gnx*
assume *progress s test* = *Some* {*xny*, *xgnx*, *xgny*, *xsk*}
with *Htest Hi* **have** *Htest′*: *guessed-frame test xsk* = *Some* (*Exp gnx* (*NonceF* (*test* \$ *ny*)))
by (*auto dest*: *l2-inv4′D*)
assume *Auth A B* ⟨*Number* (*Suc 0*), *gnx*, *Exp Gen* (*NonceF* (*test* \$ *ny*))⟩ ∈ *chan s*
with *Hi HA* **obtain** *Ra* **where** *HRa*:
guessed-runs Ra = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|)
in-progressS (*progress s Ra*) {*xnx*, *xgnx*, *xgny*, *xsk*, *xEnd*}
gnx = *Exp Gen* (*NonceF* (*Ra*\$*nx*))
guessed-frame Ra xgny = *Some* (*Exp Gen* (*NonceF* (*test*\$*ny*)))
by (*auto dest*!: *l2-inv3D*)
with *Hi*
have *guessed-frame Ra xsk* = *Some* (*Exp* (*Exp Gen* (*NonceF* (*Ra*\$*nx*))) (*NonceF* (*test*\$*ny*)))
by (*auto dest*: *l2-inv4D*)
with *HRa Htest Htest′* **have** *Ra* ∈ *partners*
by (*auto simp add*: *partners-def partner-runs-def matching-def*)
with *HRa* **have** *Exp gnx* (*NonceF* (*test* \$ *ny*)) =
Exp (*Exp Gen* (*NonceF* (*Ra* \$ *nx*))) (*NonceF* (*test* \$ *ny*)) ∧ *Ra* ∈ *partners*
by *auto*
then show ∃ *N N′ R′*.
Exp gnx (*NonceF* (*test* \$ *ny*)) = *Exp* (*Exp Gen* (*NonceF* (*R′* \$ *N′*))) (*NonceF* (*test* \$ *N*)) ∧
R′ ∈ *partners*
by *auto*

qed (*auto simp add*: *can-signal-def*)
qed

lemma *l2-inv8-trans* [*iff*]:
{*l2-inv8* ∩ *l2-inv1* ∩ *l2-inv2* ∩ *l2-inv3* ∩ *l2-inv4* ∩ *l2-inv4'*} *trans l2* {> *l2-inv8*}
apply (*auto simp add*: *l2-nostep-defs intro*!: *l2-inv8-step3 l2-inv8-step4*)
apply (*auto simp add*: *PO-hoare-defs intro*!: *l2-inv8I*)
apply (*auto simp add*: *l2-defs dy-fake-chan-def dest*: *l2-inv8D*)
done

lemma *PO-l2-inv8* [*iff*]: *reach l2* ⊆ *l2-inv8*
by (*rule-tac J=l2-inv1* ∩ *l2-inv2* ∩ *l2-inv3* ∩ *l2-inv4* ∩ *l2-inv4'* **in** *inv-rule-incr*) (*auto*)

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*
by (*erule synth.cases*, *auto dest*: *Exp-Exp-Gen-inj2*)

lemma *l2-inv8-aux*:
s ∈ *l2-inv8* ⟹
x ∈ *secret s* ⟹
x ∉ *synth* (*analz generators*)
apply (*auto simp add*: *analz-generators dest*!: *l2-inv8D* [*THEN* [*2*] *rev-subsetD*])
apply (*auto dest*!: *Exp-Exp-Gen-synth Exp-Exp-Gen-inj2*)
done

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
by (auto simp add: can-signal-def R12s-defs)
```

Protocol events.

```
lemma l2-step1-refines-step1:
  {R12s} l1-step1 Ra A B, l2-step1 Ra A B {>R12s}
by (auto simp add: PO-rhoare-defs R12s-defs l1-step1-def l2-step1-def)

lemma l2-step2-refines-step2:
  {R12s} l1-step2 Rb A B gnx, l2-step2 Rb A B gnx {>R12s}
by (auto simp add: PO-rhoare-defs R12s-defs l1-step2-def, simp-all add: l2-step2-def)
```

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 (auto simp add: PO-rhoare-defs R12s-defs)
  fix s s'
  assume Hi:s ∈ l2-inv1 s ∈ l2-inv2 s ∈ l2-inv4' s ∈ l2-inv7
  assume Ht: (s, s') ∈ l2-step3 Ra A B gny
  assume s ∈ l2-inv8
  with Hi Ht l2-inv8-step3 have Hi':s' ∈ l2-inv8
    by (auto simp add: PO-hoare-defs, blast)
  from Ht have Ra = test ⟶ Exp gny (NonceF (Ra$nx)) ∈ secret s'
    by (auto simp add: l2-defs)
  with Hi' have Ra = test ⟶ Exp gny (NonceF (Ra$nx)) ∉ synth (analz generators)
    by (auto dest: l2-inv8-aux)
  with Hi have G2:Ra = test ⟶ Exp gny (NonceF (Ra$nx)) ∉ synth (analz (ik s))
    by (auto dest!: l2-inv7D-aux)
  from Ht Hi have G1:
    can-signal s A B ⟶ (∃ Rb. guessed-runs Rb = ⦇role=Resp, owner=B, partner=A⦈ ∧
           in-progressS (progress s Rb) {xny, xgnx, xgny, xsk} ∧
           gny = Exp Gen (NonceF (Rb$ny)) ∧
           guessed-frame Rb xgnx = Some (Exp Gen (NonceF (Ra$nx))))
   by (auto dest!: l2-inv2D [rotated 2] simp add: l2-defs)

  with Ht G1 G2 show
    ((⦇ ik = ik s, secret = secret s, progress = progress s,
        signalsInit = signalsInit s, signalsResp = signalsResp s ⦈,
      ⦇ ik = ik s', secret = secret s', progress = progress s',
        signalsInit = signalsInit s', signalsResp = signalsResp s' ⦈)
          ∈ l1-step3 Ra A B gny
    apply (auto simp add: l2-step3-def, auto simp add: l1-step3-def)
    apply (auto simp add: can-signal-def)
    done
qed
```

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 (*auto simp add*: *PO-rhoare-defs R12s-defs*)
fix *s s'*
assume *Hi*:*s* ∈ *l2-inv1 s* ∈ *l2-inv3 s* ∈ *l2-inv4 s* ∈ *l2-inv4' s* ∈ *l2-inv7*
assume *Ht*: (*s*, *s'*) ∈ *l2-step4 Rb A B gnx*
assume *s* ∈ *l2-inv8*
with *Hi Ht l2-inv8-step4* **have** *Hi'*:*s'* ∈ *l2-inv8*
by (*auto simp add*: *PO-hoare-defs*, *blast*)
from *Ht* **have** *Rb* = *test* ⟶ *Exp gnx* (*NonceF* (*Rb*$*ny*)) ∈ *secret s'*
by (*auto simp add*: *l2-defs*)
with *Hi'* **have** *Rb* = *test* ⟶ *Exp gnx* (*NonceF* (*Rb*$*ny*)) ∉ *synth* (*analz generators*)
by (*auto dest*: *l2-inv8-aux*)
with *Hi* **have** *G2*:*Rb* = *test* ⟶ *Exp gnx* (*NonceF* (*Rb*$*ny*)) ∉ *synth* (*analz* (*ik s*))
by (*auto dest*!: *l2-inv7D-aux*)
from *Ht Hi* **have** *G1*:
can-signal s A B ⟶ (∃ *Ra*. *guessed-runs Ra* = ⦇*role*=*Init*, *owner*=*A*, *partner*=*B*⦈ ∧
in-progressS (*progress s Ra*) {*xnx*, *xgnx*, *xgny*, *xsk*, *xEnd*} ∧
guessed-frame Ra xgnx = *Some gnx* ∧
guessed-frame Ra xgny = *Some* (*Exp Gen* (*NonceF* (*Rb*$*ny*))))
by (*auto dest*!: *l2-inv3D* [*rotated 2*] *simp add*: *l2-defs*)

with *Ht G1 G2* **show**
((⦇ *ik* = *ik s*, *secret* = *secret s*, *progress* = *progress s*,
signalsInit = *signalsInit s*, *signalsResp* = *signalsResp s* ⦈,
⦇ *ik* = *ik s'*, *secret* = *secret s'*, *progress* = *progress s'*,
signalsInit = *signalsInit s'*, *signalsResp* = *signalsResp s'* ⦈)
∈ *l1-step4 Rb A B gnx*
apply (*auto simp add*: *l2-step4-def*, *auto simp add*: *l1-step4-def*)
apply (*auto simp add*: *can-signal-def*)
done
qed

Attacker events.

lemma *l2-dy-fake-chan-refines-skip*:
{*R12s*} *Id*, *l2-dy-fake-chan M* {>*R12s*}
by (*auto simp add*: *PO-rhoare-defs R12s-defs l2-defs*)

lemma *l2-dy-fake-msg-refines-learn*:
{*R12s* ∩ *UNIV* × (*l2-inv7* ∩ *l2-inv8*)} *l1-learn m*, *l2-dy-fake-msg m* {>*R12s*}
apply (*auto simp add*: *PO-rhoare-defs R12s-defs l2-loc-defs l1-defs*)
apply (*drule Fake-insert-dy-fake-msg*, *erule l2-inv7D*)
apply (*auto dest*!: *l2-inv8-aux*)
done

Compromising events.

lemma *l2-lkr-others-refines-skip*:
{*R12s*} *Id*, *l2-lkr-others A* {>*R12s*}
by (*auto simp add*: *PO-rhoare-defs R12s-defs l2-loc-defs l1-defs*)

```
lemma l2-lkr-after-refines-skip:
  {R12s} Id, l2-lkr-after A {>R12s}
by (auto simp add: PO-rhoare-defs R12s-defs l2-loc-defs l1-defs)

lemma l2-skr-refines-learn:
  {R12s ∩ UNIV × l2-inv7 ∩ UNIV × l2-inv6 ∩ UNIV × l2-inv8} l1-learn K, l2-skr R K {>R12s}
proof (auto simp add: PO-rhoare-defs R12s-defs l2-loc-defs l1-defs)
  fix s :: l2-state fix x
  assume H:s ∈ l2-inv7 s ∈ l2-inv6
        R ∉ partners R ≠ test in-progress (progress s R) xsk guessed-frame R xsk = Some K
  assume Hx:x ∈ synth (analz (insert K (ik s)))
  assume x ∈ secret s s ∈ l2-inv8
  then obtain R R' N N' where Hx':x = Exp (Exp Gen (NonceF (R$N))) (NonceF (R'$N'))
                             R = test ∧ R' ∈ partners
    by (auto dest!: l2-inv8D subsetD)
  from H have s (|ik := insert K (ik s)|) ∈ l2-inv7
    by (auto intro: hoare-apply [OF l2-inv7-skr] simp add: l2-defs)
  with Hx have x ∈ synth (analz (generators))
    by (auto dest: l2-inv7D-aux)
  with Hx' show False
    by (auto dest!: Exp-Exp-Gen-synth dest: Exp-Exp-Gen-inj2 simp add: analz-generators)
qed
```

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)
by (auto simp add: R12s-defs l1-defs l2-loc-defs)

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}
by (auto 0 3 simp add: l1-def l2-def l1-trans-def l2-trans-def
            intro: l2-trans-refines-l1-trans)


lemma PO-obs-consistent-R12s [iff]:
  obs-consistent R12s med12s l1 l2
by (auto simp add: obs-consistent-def R12s-def med12s-def l2-defs)

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
```

by (*rule Refinement-using-invariants*, *auto*)

lemma *l2-implements-l1* [*iff*]:
implements med12s l1 l2
by (*rule refinement-soundness*) (*auto*)

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*
apply (*rule external-invariant-translation*
[*OF l1-obs-secrecy - l2-implements-l1*])
apply (*auto simp add*: *med12s-def s0-secrecy-def l2-partial-secrecy-def*)
done

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*
apply (*auto simp add*: *oreach-def*, *rule*, *simp-all*,
simp add: *l2-def l2-trans-def l2-dy-fake-msg-def*)
apply *blast*
done

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*
apply (*auto simp add*:*l2-secrecy-def*)
apply (*drule l2-oreach-dy-fake-msg*, *simp-all*)
apply (*drule l2-obs-partial-secrecy* [*THEN* [*2*] *rev-subsetD*], *simp add*: *l2-partial-secrecy-def*)
apply *blast*
done

lemma *l2-secrecy* [*iff*]: *reach l2* ⊆ *l2-secrecy*
by (*rule external-to-internal-invariant* [*OF l2-obs-secrecy*], *auto*)

abbreviation *l2-iagreement-Init* ≡ *l1-iagreement-Init*

lemma *l2-obs-iagreement-Init* [*iff*]: *oreach l2* ⊆ *l2-iagreement-Init*
apply (*rule external-invariant-translation*
[*OF l1-obs-iagreement-Init - l2-implements-l1*])
apply (*auto simp add*: *med12s-def l1-iagreement-Init-def*)
done

lemma *l2-iagreement-Init* [*iff*]: *reach l2* ⊆ *l2-iagreement-Init*
by (*rule external-to-internal-invariant* [*OF l2-obs-iagreement-Init*], *auto*)

abbreviation *l2-iagreement-Resp* ≡ *l1-iagreement-Resp*

lemma *l2-obs-iagreement-Resp* [*iff*]: *oreach l2* ⊆ *l2-iagreement-Resp*
apply (*rule external-invariant-translation*
[*OF l1-obs-iagreement-Resp - l2-implements-l1*])
apply (*auto simp add*: *med12s-def l1-iagreement-Resp-def*)
done

lemma *l2-iagreement-Resp* [*iff*]: *reach l2* ⊆ *l2-iagreement-Resp*
by (*rule external-to-internal-invariant* [*OF l2-obs-iagreement-Resp*], *auto*)

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 $A \equiv \{(s,s')$.
    — guards
    $A \neq$ *test-owner* ∧
    $A \neq$ *test-partner* ∧
    — actions
    $s' = s$⦇*bad* := $\{A\} \cup$ *bad* s,
        ik := *keys-of* $A \cup$ *ik* s⦈
  }

definition
  l3-lkr-actor :: *agent* ⇒ *l3-trans*
where
  l3-lkr-actor $A \equiv \{(s,s')$.
    — guards
    $A =$ *test-owner* ∧
    $A \neq$ *test-partner* ∧
    — actions
    $s' = s$⦇*bad* := $\{A\} \cup$ *bad* s,
        ik := *keys-of* $A \cup$ *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

$l3\text{-}step4 :: rid\text{-}t \Rightarrow agent \Rightarrow agent \Rightarrow msg \Rightarrow l3\text{-}trans$
where
$l3\text{-}step4\ Rb\ A\ B\ gnx \equiv \{(s, s').$
— guards:
$guessed\text{-}runs\ Rb = (\!|role=Resp, owner=B, partner=A|\!) \wedge$
$progress\ s\ Rb = Some\ \{xny, xgnx, xgny, xsk\} \wedge$
$guessed\text{-}frame\ Rb\ xgnx = Some\ gnx \wedge$
$implAuth\ A\ B\ \langle Number\ 1, gnx, Exp\ Gen\ (NonceF\ (Rb\$ny))\rangle \in ik\ s \wedge$

— actions:
$s' = s(\!|\ progress := (progress\ s)(Rb \mapsto \{xny, xgnx, xgny, xsk, xEnd\}),$
$secret := \{x.\ x = Exp\ gnx\ (NonceF\ (Rb\$ny)) \wedge Rb = test\} \cup secret\ s,$
$signalsResp := if\ can\text{-}signal\ s\ A\ B\ then$
$addSignal\ (signalsResp\ s)\ (Commit\ A\ B\ (Exp\ gnx\ (NonceF\ (Rb\$ny))))$
$else$
$signalsResp\ s$
$|\!)$
$\}$

Specification.

Initial compromise.

definition
$ik\text{-}init :: msg\ set$
where
$ik\text{-}init \equiv \{priK\ C\ |C.\ C \in bad\text{-}init\} \cup \{pubK\ A \mid A.\ True\} \cup$
$\{shrK\ A\ B\ |A\ B.\ A \in bad\text{-}init \vee B \in bad\text{-}init\} \cup Tags$

Lemmas about *ik-init*.

lemma *parts-ik-init* [*simp*]: *parts ik-init* = *ik-init*
by (*auto elim*!: *parts.induct*, *auto simp add*: *ik-init-def*)

lemma *analz-ik-init* [*simp*]: *analz ik-init* = *ik-init*
by (*auto dest*: *analz-into-parts*)

lemma *abs-ik-init* [*iff*]: *abs ik-init* = {}
apply (*auto elim*!: *absE*)
apply (*auto simp add*: *ik-init-def*)
done

lemma *payloadSet-ik-init* [*iff*]: *ik-init* ∩ *payload* = {}
by (*auto simp add*: *ik-init-def*)

lemma *validSet-ik-init* [*iff*]: *ik-init* ∩ *valid*= {}
by (*auto simp add*: *ik-init-def*)

definition
$l3\text{-}init :: l3\text{-}state\ set$
where
$l3\text{-}init \equiv \{\ (\!|$
$ik = ik\text{-}init,$
$secret = \{\},$

progress = *Map.empty*,
signalsInit = $\lambda x.$ *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*
by (*simp add*: *l3-def*)

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*
by (*auto simp add*: *l3-inv1-def*)

lemma *l3-inv1-init* [*iff*]:
init l3 ⊆ *l3-inv1*
by (*auto simp add*: *l3-def l3-init-def intro*!:*l3-inv1I*)

lemma *l3-inv1-trans* [*iff*]:
{*l3-inv1*} *trans l3* {> *l3-inv1*}
apply (*auto simp add*: *PO-hoare-defs l3-nostep-defs intro*!: *l3-inv1I*)
apply (*auto simp add*: *l3-defs dy-fake-msg-def dy-fake-chan-def*
parts-insert [**where** *H*=*ik* -] *parts-insert* [**where** *H*=*insert* - (*ik* -)]
dest!: *Fake-parts-insert*)
apply (*auto dest*:*analz-into-parts*)
done

lemma *PO-l3-inv1* [*iff*]:
reach l3 ⊆ *l3-inv1*
by (*rule inv-rule-basic*) (*auto*)

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*
by (*auto simp add*: *l3-def l3-init-defs intro*!:*l3-inv2I Keys-badI*)

lemma *l3-inv2-trans* [*simp*,*intro*!]:
{*l3-inv2* ∩ *l3-inv1*} *trans l3* {> *l3-inv2*}
apply (*auto simp add*: *PO-hoare-defs l3-nostep-defs intro*!: *l3-inv2I*)
apply (*auto simp add*: *l3-defs dy-fake-msg-def dy-fake-chan-def*)

4 subgoals: dy, lkr*, skr

apply (*auto intro*: *Keys-bad-insert-Fake Keys-bad-insert-keys-of*)
apply (*auto intro*!: *Keys-bad-insert-payload*)

done

lemma *PO-l3-inv2* [*iff*]: *reach l3* ⊆ *l3-inv2*
by (*rule-tac J*=*l3-inv1* **in** *inv-rule-incr*) (*auto*)

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*
apply (*auto simp add*: *l3-def l3-init-def intro*!: *l3-inv3I*)
apply (*auto simp add*: *ik-init-def intro*!: *synth-increasing* [*THEN* [*2*] *rev-subsetD*])
done

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 (*simp add*: *l3-nostep-defs*, *safe*)
  fix *Ra A B*
  show {*l3-inv3*} *l3-step1 Ra A B* {> *l3-inv3*}
    apply (*auto simp add*: *PO-hoare-def l3-defs intro*!: *l3-inv3I dest*!: *l3-inv3D*)
    apply (*auto intro*!: *validI dest*!: *analz-insert-partition* [*THEN* [*2*] *rev-subsetD*])
    done
next
  fix *Rb A B gnx*
  show {*l3-inv3*} *l3-step2 Rb A B gnx* {> *l3-inv3*}
    apply (*auto simp add*: *PO-hoare-def l3-defs intro*!: *l3-inv3I dest*!: *l3-inv3D*)
    apply (*auto intro*!: *validI dest*!: *analz-insert-partition* [*THEN* [*2*] *rev-subsetD*])
    done
next
  fix *Ra A B gny*
  show {*l3-inv3*} *l3-step3 Ra A B gny* {> *l3-inv3*}
    apply (*auto simp add*: *PO-hoare-def l3-defs intro*!: *l3-inv3I dest*!: *l3-inv3D*)
    apply (*auto intro*!: *validI dest*!: *analz-insert-partition* [*THEN* [*2*] *rev-subsetD*])
    done

next
  fix *Rb A B gnx*
  show {*l3-inv3*} *l3-step4 Rb A B gnx* {> *l3-inv3*}
    apply (*auto simp add*: *PO-hoare-def l3-defs intro*!: *l3-inv3I dest*!: *l3-inv3D*)
    done
next
  fix *m*
  show {*l3-inv3*} *l3-dy m* {> *l3-inv3*}
    apply (*auto simp add*: *PO-hoare-def l3-defs dy-fake-chan-def dy-fake-msg-def*
      intro!: *l3-inv3I dest*!: *l3-inv3D*)
    apply (*drule synth-analz-insert*)
    apply (*blast intro*: *synth-analz-monotone dest*: *synth-monotone*)
    done
next
  fix *A*
  show {*l3-inv3*} *l3-lkr-others A* {> *l3-inv3*}
    apply (*auto simp add*: *PO-hoare-def l3-defs intro*!: *l3-inv3I dest*!: *l3-inv3D*)
    apply (*drule analz-Un-partition* [*of - keys-of A*], *auto*)
    done
next
  fix *A*
  show {*l3-inv3*} *l3-lkr-after A* {> *l3-inv3*}
    apply (*auto simp add*: *PO-hoare-def l3-defs intro*!: *l3-inv3I dest*!: *l3-inv3D*)
    apply (*drule analz-Un-partition* [*of - keys-of A*], *auto*)
    done
next
  fix *R K*
  show {*l3-inv3*} *l3-skr R K* {> *l3-inv3*}
    apply (*auto simp add*: *PO-hoare-def l3-defs intro*!: *l3-inv3I dest*!: *l3-inv3D*)
    apply (*auto dest*!: *analz-insert-partition* [*THEN* [*2*] *rev-subsetD*])
    done
qed

lemma *PO-l3-inv3* [*iff*]: *reach l3* ⊆ *l3-inv3*
by (*rule inv-rule-basic*) (*auto*)

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*
by (*auto simp add*: *l3-def l3-init-def ik-init-def intro*!:*l3-inv4I*)

lemma *l3-inv4-trans* [*simp*,*intro*!]:
  {*l3-inv4*} *trans l3* {> *l3-inv4*}
apply (*auto simp add*: *PO-hoare-defs l3-nostep-defs intro*!: *l3-inv4I*)
apply (*auto simp add*: *l3-defs dy-fake-chan-def dy-fake-msg-def*)
done

lemma *PO-l3-inv4* [*simp*,*intro*!]: *reach l3* ⊆ *l3-inv4*
by (*rule inv-rule-basic*) (*auto*)

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*
by (*auto simp add*: *abs-validSet dy-fake-msg-def intro*!: *l3-inv5I*
            dest!: *l3-inv3D* [*THEN synth-mono*, *THEN* [*2*] *rev-subsetD*]
            dest!: *synth-analz-NI-I-K-synth-analz-NI-E* [*THEN* [*2*] *rev-subsetD*])

lemma *PO-l3-inv5* [*simp*,*intro*!]: *reach l3* ⊆ *l3-inv5*
using *l3-inv5-derived PO-l3-inv2 PO-l3-inv3*
by *blast*

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*
apply (*auto intro*!: *l3-inv6I dest*!: *l3-inv3D*)

1 subgoal

apply (*drule synth-mono*, *simp*, *drule subsetD*, *assumption*)
apply (*auto dest*!: *implInsec-synth-analz* [*rotated 1*, **where** *H*=- ∪ -])
apply (*auto dest*!: *synth-analz-monotone* [*of* - - ∪ - *ik* -])
done

lemma *PO-l3-inv6* [*simp*,*intro*!]: *reach l3* ⊆ *l3-inv6*
using *l3-inv6-derived PO-l3-inv3 PO-l3-inv4*
by (*blast*)

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*
apply (*auto intro*!: *l3-inv7I dest*!: *l3-inv3D*)

1 subgoal

apply (*drule synth-mono*, *simp*, *drule subsetD*, *assumption*)
apply (*auto dest*!: *implConfid-synth-analz* [*rotated 1*, **where** *H*=- ∪ -])
apply (*auto dest*!: *synth-analz-monotone* [*of* - - ∪ - *ik* -])

done

lemma *PO-l3-inv7* [*simp*,*intro*!]: *reach l3* ⊆ *l3-inv7*
using *l3-inv7-derived PO-l3-inv3 PO-l3-inv4*
by (*blast*)

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*
apply (*auto intro*!: *l3-inv8I dest*!: *l3-inv3D l3-inv2D*)

2 subgoals: M is deducible and the agents are bad

apply (*drule synth-mono*, *simp*, *drule subsetD*, *assumption*)
apply (*auto dest*!: *implAuth-synth-analz* [*rotated 1*, **where** *H*=- ∪ -] *elim*!: *synth-analz-monotone*)

apply (*drule synth-mono*, *simp*, *drule subsetD*, *assumption*)
apply (*auto dest*!: *implAuth-synth-analz* [*rotated 1*, **where** *H*=- ∪ -] *elim*!: *synth-analz-monotone*)
done

lemma *PO-l3-inv8* [*iff*]: *reach l3* ⊆ *l3-inv8*
using *l3-inv8-derived*
 PO-l3-inv3 PO-l3-inv2 PO-l3-inv4
by *blast*

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*
apply (*auto intro*!: *l3-inv9I dest*!:*l3-inv3D l3-inv2D*)

2 subgoals: M is deducible and the agents are bad

apply (*drule synth-mono*, *simp*, *drule subsetD*, *assumption*)
apply (*auto dest*!: *implSecure-synth-analz* [*rotated 1*, **where** *H*=- ∪ -]
elim!: *synth-analz-monotone*)

apply (*drule synth-mono*, *simp*, *drule subsetD*, *assumption*)
apply (*auto dest*!: *implSecure-synth-analz* [*rotated 1*, **where** *H*=- ∪ -])
done

lemma *PO-l3-inv9* [*iff*]: *reach l3* ⊆ *l3-inv9*
using *l3-inv9-derived*
PO-l3-inv3 PO-l3-inv2 PO-l3-inv4
by *blast*

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
by (auto simp add: R23s-def med23s-def)

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
by (auto simp add: R23s-def med23s-def)

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
by (auto simp add: R23s-def med23s-def)

lemma can-signal-R23 [simp]:
  (s2, s3) ∈ R23s ⟹
   can-signal s2 A B ⟷ can-signal s3 A B
by (auto simp add: can-signal-def)
```

23.3.1 Protocol events

```
lemma l3-step1-refines-step1:
  {R23s} l2-step1 Ra A B, l3-step1 Ra A B {>R23s}
apply (auto simp add: PO-rhoare-defs R23s-defs)
apply (auto simp add: l3-defs l2-step1-def)
done

lemma l3-step2-refines-step2:
  {R23s} l2-step2 Rb A B gnx, l3-step2 Rb A B gnx{>R23s}
apply (auto simp add: PO-rhoare-defs R23s-defs l2-step2-def)
apply (auto simp add: l3-step2-def)
done

lemma l3-step3-refines-step3:
  {R23s} l2-step3 Ra A B gny, l3-step3 Ra A B gny {>R23s}
apply (auto simp add: PO-rhoare-defs R23s-defs l2-step3-def)
apply (auto simp add: l3-step3-def)
```

done

lemma *l3-step4-refines-step4*:
  {*R23s*} *l2-step4 Rb A B gnx*, *l3-step4 Rb A B gnx* {>*R23s*}
apply (*auto simp add*: *PO-rhoare-defs R23s-defs l2-step4-def*)
apply (*auto simp add*: *l3-step4-def*)
done

23.3.2 Intruder events

lemma *l3-dy-payload-refines-dy-fake-msg*:
  $M \in payload \Longrightarrow$
   {*R23s* ∩ *UNIV* × *l3-inv5*} *l2-dy-fake-msg M*, *l3-dy M* {>*R23s*}
apply (*auto simp add*: *PO-rhoare-defs R23s-defs*)
apply (*auto simp add*: *l3-defs l2-dy-fake-msg-def dest*: *l3-inv5D*)
done

lemma *l3-dy-valid-refines-dy-fake-chan*:
  ⟦ $M \in valid$; $M' \in abs\ \{M\}$ ⟧ $\Longrightarrow$
   {*R23s* ∩ *UNIV* × (*l3-inv5* ∩ *l3-inv6* ∩ *l3-inv7* ∩ *l3-inv8* ∩ *l3-inv9*)}
      l2-dy-fake-chan M′, *l3-dy M*
   {>*R23s*}
apply (*auto simp add*: *PO-rhoare-defs R23s-defs*, *simp add*: *l2-dy-fake-chan-def*)
apply (*auto simp add*: *l3-defs*)

1 subgoal

apply (*erule valid-cases*, *simp-all add*: *dy-fake-chan-def*)

Insec

  apply (*blast dest*: *l3-inv6D l3-inv5D*)

Confid

  apply (*blast dest*: *l3-inv7D l3-inv5D*)

Auth

  apply (*blast dest*: *l3-inv8D l3-inv5D*)

Secure

  apply (*blast dest*: *l3-inv9D l3-inv5D*)
done

lemma *l3-dy-valid-refines-dy-fake-chan-Un*:
  $M \in valid \Longrightarrow$
   {*R23s* ∩ *UNIV* × (*l3-inv5* ∩ *l3-inv6* ∩ *l3-inv7* ∩ *l3-inv8* ∩ *l3-inv9*)}
      ⋃*M′*. *l2-dy-fake-chan M′*, *l3-dy M*
   {>*R23s*}
by (*auto dest*: *valid-abs intro*: *l3-dy-valid-refines-dy-fake-chan*)

lemma *l3-dy-isLtKey-refines-skip*:
  {*R23s*} *Id*, *l3-dy* (*LtK ltk*) {>*R23s*}

apply (*auto simp add*: *PO-rhoare-defs R23s-defs l3-defs*)
apply (*auto elim*!: *absE*)
done

lemma *l3-dy-others-refines-skip*:
  ⟦ $M \notin$ *range LtK*; $M \notin$ *valid*; $M \notin$ *payload* ⟧ $\Longrightarrow$
   {*R23s*} *Id*, *l3-dy M* {>*R23s*}
apply (*auto simp add*: *PO-rhoare-defs R23s-defs*)
apply (*auto simp add*: *l3-defs*)
apply (*auto elim*!: *absE intro*: *validI*)
done

lemma *l3-dy-refines-dy-fake-msg-dy-fake-chan-skip*:
   {*R23s* ∩ *UNIV* × (*l3-inv5* ∩ *l3-inv6* ∩ *l3-inv7* ∩ *l3-inv8* ∩ *l3-inv9*)}
      l2-dy-fake-msg M ∪ ($\bigcup M'$. *l2-dy-fake-chan M'*) ∪ *Id*, *l3-dy M*
    {>*R23s*}
by (*cases M* ∈ *payload* ∪ *valid* ∪ *range LtK*)
   (*auto dest*: *l3-dy-payload-refines-dy-fake-msg l3-dy-valid-refines-dy-fake-chan-Un*
      intro: *l3-dy-isLtKey-refines-skip dest*!: *l3-dy-others-refines-skip*)

23.3.3 Compromise events

lemma *l3-lkr-others-refines-lkr-others*:
  {*R23s*} *l2-lkr-others A*, *l3-lkr-others A* {>*R23s*}
apply (*auto simp add*: *PO-rhoare-defs R23s-defs*)
apply (*auto simp add*: *l3-defs l2-lkr-others-def*)
done

lemma *l3-lkr-after-refines-lkr-after*:
  {*R23s*} *l2-lkr-after A*, *l3-lkr-after A* {>*R23s*}
apply (*auto simp add*: *PO-rhoare-defs R23s-defs*)
apply (*auto simp add*: *l3-defs l2-lkr-after-def*)
done

lemma *l3-skr-refines-skr*:
  {*R23s*} *l2-skr R K*, *l3-skr R K* {>*R23s*}
apply (*auto simp add*: *PO-rhoare-defs R23s-defs*)
apply (*auto simp add*: *l3-defs l2-skr-def*)
done

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*)
by (*auto simp add*: *R23s-defs l2-defs l3-def l3-init-def*)

```
lemma l3-refines-trans-l2 [iff]:
  {R23s ∩ (UNIV × (l3-inv1 ∩ l3-inv2 ∩ l3-inv3 ∩ l3-inv4))} trans l2, trans l3 {> R23s}
proof −
  let ?pre' = R23s ∩ (UNIV × (l3-inv5 ∩ l3-inv6 ∩ l3-inv7 ∩ l3-inv8 ∩ l3-inv9))
  show ?thesis (is {?pre} ?t1, ?t2 {>?post})
  proof (rule relhoare-conseq-left)
    show ?pre ⊆ ?pre'
      using l3-inv5-derived l3-inv6-derived l3-inv7-derived l3-inv8-derived l3-inv9-derived
      by blast
  next
    show {?pre'} ?t1, ?t2 {> ?post}
      by (auto simp add: l2-def l3-def l2-trans-def l3-trans-def
               intro!: l3-trans-refines-l2-trans)
  qed
qed

lemma PO-obs-consistent-R23s [iff]:
  obs-consistent R23s med23s l2 l3
by (auto simp add: obs-consistent-def R23s-def med23s-def l2-defs)


lemma l3-refines-l2 [iff]:
  refines
     (R23s ∩
      (reach l2 × (l3-inv1 ∩ l3-inv2 ∩ l3-inv3 ∩ l3-inv4)))
     med23s l2 l3
by (rule Refinement-using-invariants, auto)

lemma l3-implements-l2 [iff]:
  implements med23s l2 l3
by (rule refinement-soundness) (auto)
```

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
by (auto simp add: l3-def l3-init-def ik-init-def intro!:l3-inv10I)

lemma l3-inv10-trans [iff]:
```

{*l3-inv10*} *trans l3* {> *l3-inv10*}
apply (*auto simp add*: *PO-hoare-defs l3-nostep-defs*)
apply (*auto simp add*: *l3-defs l3-inv10-def*)
done

lemma *PO-l3-inv10* [*iff*]: *reach l3* ⊆ *l3-inv10*
by (*rule inv-rule-basic*) (*auto*)

lemma *l3-obs-inv10* [*iff*]: *oreach l3* ⊆ *l3-inv10*
by (*auto simp add*: *oreach-def*)

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*
apply (*rule external-invariant-translation* [*OF l2-obs-secrecy - l3-implements-l2*])
apply (*auto simp add*: *med23s-def l2-secrecy-def l3-partial-secrecy-def*)
done

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

lemma *l3-obs-secrecy* [*iff*]: *oreach l3* ⊆ *l3-secrecy*
apply (*rule*, *frule l3-obs-inv5* [*THEN* [*2*] *rev-subsetD*], *frule l3-obs-inv10* [*THEN* [*2*] *rev-subsetD*])
apply (*auto simp add*: *med23s-def l2-secrecy-def l3-secrecy-def s0-secrecy-def l3-inv10-def*)
using *l3-partial-secrecy-def* **apply** (*blast dest*!: *l3-inv5D subsetD* [*OF l3-obs-partial-secrecy*])
done

lemma *l3-secrecy* [*iff*]: *reach l3* ⊆ *l3-secrecy*
by (*rule external-to-internal-invariant* [*OF l3-obs-secrecy*], *auto*)

23.4.4 Injective agreement

abbreviation *l3-iagreement-Init* ≡ *l1-iagreement-Init*

lemma *l3-obs-iagreement-Init* [*iff*]: *oreach l3* ⊆ *l3-iagreement-Init*
apply (*rule external-invariant-translation*
[*OF l2-obs-iagreement-Init - l3-implements-l2*])
apply (*auto simp add*: *med23s-def l1-iagreement-Init-def*)
done

lemma *l3-iagreement-Init* [*iff*]: *reach l3* ⊆ *l3-iagreement-Init*
by (*rule external-to-internal-invariant* [*OF l3-obs-iagreement-Init*], *auto*)

abbreviation *l3-iagreement-Resp* ≡ *l1-iagreement-Resp*

lemma *l3-obs-iagreement-Resp* [*iff*]: *oreach l3* ⊆ *l3-iagreement-Resp*
apply (*rule external-invariant-translation*
[*OF l2-obs-iagreement-Resp - l3-implements-l2*])
apply (*auto simp add*: *med23s-def l1-iagreement-Resp-def*)
done

lemma *l3-iagreement-Resp* [*iff*]: *reach l3* ⊆ *l3-iagreement-Resp*
by (*rule external-to-internal-invariant* [*OF l3-obs-iagreement-Resp*], *auto*)

end
end

24 Authenticated Diffie-Hellman Protocol (L3, asymmetric)

theory *dhlvl3-asymmetric*
imports *dhlvl3 Implem-asymmetric*
begin

interpretation *dhlvl3-asym*: *dhlvl3 implem-asym*
by (*unfold-locales*)

end

25 Authenticated Diffie-Hellman Protocol (L3, symmetric)

theory *dhlvl3-symmetric*
imports *dhlvl3 Implem-symmetric*
begin

interpretation *dhlvl3-sym*: *dhlvl3 implem-sym*
by (*unfold-locales*)

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* ⟹ *guessed-frame R xni* = *Some* (*NonceF* (*R$ni*))
guessed-frame-Resp-xny [*simp*]:
role (*guessed-runs R*) = *Resp* ⟹ *guessed-frame R xny* = *Some* (*NonceF* (*R$ny*))
guessed-frame-Resp-xgny [*simp*]:
role (*guessed-runs R*) = *Resp* ⟹ *guessed-frame R xgny* = *Some* (*Exp Gen* (*NonceF* (*R$ny*)))
guessed-frame-Resp-xnr [*simp*]:
role (*guessed-runs R*) = *Resp* ⟹ *guessed-frame R xnr* = *Some* (*NonceF* (*R$nr*))
guessed-frame-xEnd [*simp*]:
guessed-frame R xEnd = *Some End*
guessed-frame-eq [*simp*]:
x ∈ {*xnx*, *xny*, *xgnx*, *xgny*, *xsk*, *xEnd*} ⟹ *dhlvl1.guessed-frame R x* = *guessed-frame R x*
apply (*rule exI* [*of* -
λ*R*.
if role (*guessed-runs R*) = *Init then*
(*dhlvl1.guessed-frame R*) (*xni* ↦ *NonceF* (*R$ni*), *xnr* ↦ *End*)
else
(*dhlvl1.guessed-frame R*) (*xnr* ↦ *NonceF* (*R$nr*), *xni* ↦ *End*)],
auto simp add: *domIff intro*: *role-t.exhaust*)
done

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 = λx. 0,
    signalsResp = λx. 0,
    signalsInit2 = λx. 0,
    signalsResp2 = λx. 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
by (simp add: skl1-def)
```

lemma *run-ended-trans*:
run-ended (*progress s R*) $\Longrightarrow$
(*s*, *s'*) $\in$ *trans skl1* $\Longrightarrow$
run-ended (*progress s' R*)
by (*auto simp add*: *skl1-nostep-defs*)
(*auto simp add*: *skl1-defs ik-dy-def domIff*)

lemma *can-signal-trans*:
can-signal s' A B $\Longrightarrow$
(*s*, *s'*) $\in$ *trans skl1* $\Longrightarrow$
can-signal s A B
by (*auto simp add*: *can-signal-def run-ended-trans*)

26.2 Refinement: secrecy

fun *option-inter* :: *var set* $\Rightarrow$ *var set option* $\Rightarrow$ *var set option*
where
option-inter S (*Some x*) = *Some* ($x \cap S$)
|*option-inter S None* = *None*

definition *med-progress* :: *progress-t* $\Rightarrow$ *progress-t*
where
med-progress r $\equiv \lambda$ *R*. *option-inter* {*xnx*, *xny*, *xgnx*, *xgny*, *xsk*, *xEnd*} (*r R*)

lemma *med-progress-upd* [*simp*]:
med-progress (*r*(*R* $\mapsto$ *S*)) = (*med-progress r*) (*R* $\mapsto$ *S* $\cap$ {*xnx*, *xny*, *xgnx*, *xgny*, *xsk*, *xEnd*})
by (*auto simp add*: *med-progress-def*)

lemma *med-progress-Some*:
r x = *Some s* $\Longrightarrow$ *med-progress r x* = *Some* (*s* $\cap$ {*xnx*, *xny*, *xgnx*, *xgny*, *xsk*, *xEnd*})
by (*auto simp add*: *med-progress-def*)

lemma *med-progress-None* [*simp*]: *med-progress r x* = *None* $\longleftrightarrow$ *r x* = *None*
by (*cases r x*, *auto simp add*: *med-progress-def*)

lemma *med-progress-Some2* [*dest*]:
med-progress r x = *Some y* $\Longrightarrow \exists$ *z*. *r x* = *Some z* $\wedge$ *y* = *z* $\cap$ {*xnx*, *xny*, *xgnx*, *xgny*, *xsk*, *xEnd*}
by (*cases r x*, *auto simp add*: *med-progress-def*)

lemma *med-progress-dom* [*simp*]: *dom* (*med-progress r*) = *dom r*
apply (*auto simp add*: *domIff med-progress-def*)
apply (*rename-tac x y*, *case-tac r x*, *auto*)
done

lemma *med-progress-empty* [*simp*]: *med-progress Map.empty* = *Map.empty*
by (*rule ext*, *auto*)

Mediator function.

definition
med11 :: *skl1-obs* $\Rightarrow$ *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 ≡ {(*s*,*s'*).
s = *med11 s'*
}

lemmas *R11-defs* = *R11-def med11-def*

lemma *in-progress-med-progress*:
x ∈ {*xnx*, *xny*, *xgnx*, *xgny*, *xsk*, *xEnd*}
⟹ *in-progress* (*med-progress r R*) *x* ⟷ *in-progress* (*r R*) *x*
by (*cases r R*, *auto*)
(*cases med-progress r R*, *auto*)+

lemma *in-progressS-eq*: *in-progressS S S'* ⟷ (*S* ≠ *None* ∧ (∀ *x* ∈ *S'*. *in-progress S x*))
by (*cases S*, *auto*)

lemma *in-progressS-med-progress*:
in-progressS (*r R*) *S*
⟹ *in-progressS* (*med-progress r R*) (*S* ∩ {*xnx*, *xny*, *xgnx*, *xgny*, *xsk*, *xEnd*})
by (*auto simp add*: *in-progressS-eq in-progress-med-progress*)

lemma *can-signal-R11* [*simp*]:
(*s1*, *s2*) ∈ *R11* ⟹
can-signal s1 A B ⟷ *can-signal s2 A B*
by (*auto simp add*: *can-signal-def R11-defs in-progress-med-progress*)

Protocol-independent events.

lemma *skl1-learn-refines-learn*:
{*R11*} *l1-learn m*, *l1-learn m* {>*R11*}
by (*auto simp add*: *PO-rhoare-defs R11-defs*)
(*simp add*: *l1-defs*)

Protocol events.

lemma *skl1-step1-refines-step1*:
{*R11*} *l1-step1 Ra A B*, *skl1-step1 Ra A B* {>*R11*}
by (*auto simp add*: *PO-rhoare-defs R11-defs l1-step1-def skl1-step1-def*)

lemma *skl1-step2-refines-step2*:
{*R11*} *l1-step2 Rb A B gnx*, *skl1-step2 Rb A B Ni gnx* {>*R11*}
by (*auto simp add*: *PO-rhoare-defs R11-defs l1-step2-def*)
(*auto simp add*: *skl1-step2-def*)

lemma *skl1-step3-refines-step3*:
 {*R11*} *l1-step3 Ra A B gny*, *skl1-step3 Ra A B Nr gny* {>*R11*}
apply (*auto simp add*: *PO-rhoare-defs R11-defs l1-step3-def*)
apply (*auto simp add*: *skl1-step3-def*, *auto dest*: *med-progress-Some*)
apply (*drule in-progressS-med-progress*, *auto*)+
done

lemma *skl1-step4-refines-step4*:
 {*R11*} *l1-step4 Rb A B gnx*, *skl1-step4 Rb A B Ni gnx* {>*R11*}
apply (*auto simp add*: *PO-rhoare-defs R11-defs l1-step4-def*)
apply (*auto simp add*: *skl1-step4-def*, *auto dest*: *med-progress-Some*)
apply (*drule in-progressS-med-progress*, *auto*)+
done

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 ⊆ *R11* ‘‘ (*init l1*)
by (*auto simp add*: *R11-defs l1-defs skl1-defs*)

lemma *skl1-refines-trans-l1* [*iff*]:
 {*R11*} *trans l1*, *trans skl1* {> *R11*}
by (*auto 0 3 simp add*: *l1-def skl1-def l1-trans-def skl1-trans-def*
  intro: *skl1-trans-refines-l1-trans*)

lemma *obs-consistent-med11* [*iff*]:
 obs-consistent R11 med11 l1 skl1
by (*auto simp add*: *obs-consistent-def R11-defs*)

Refinement result.

lemma *skl1-refines-l1* [*iff*]:
 refines
  R11
  med11 l1 skl1
by (*auto simp add:refines-def PO-refines-def*)

lemma *skl1-implements-l1* [*iff*]: *implements med11 l1 skl1*
by (*rule refinement-soundness*) (*fast*)

26.3 Derived invariants: secrecy

lemma *skl1-obs-secrecy* [*iff*]: *oreach skl1* ⊆ *s0-secrecy*
apply (*rule external-invariant-translation* [*OF l1-obs-secrecy - skl1-implements-l1*])
apply (*auto simp add*: *med11-def s0-secrecy-def*)
done

lemma *skl1-secrecy* [*iff*]: *reach skl1* ⊆ *s0-secrecy*

by (*rule external-to-internal-invariant* [*OF skl1-obs-secrecy*], *auto*)

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*
by (*auto simp add*: *skl1-def skl1-init-def skl1-inv1-def*)

lemma *skl1-inv1-trans* [*iff*]:
  {*skl1-inv1*} *trans skl1* {> *skl1-inv1*}
apply (*auto simp add*: *PO-hoare-defs skl1-nostep-defs intro*!: *skl1-inv1I*)
apply (*auto simp add*: *skl1-defs ik-dy-def skl1-inv1-def domIff dest*: *Exp-Exp-Gen-inj2* [*OF sym*])
done

lemma *PO-skl1-inv1* [*iff*]: *reach skl1* ⊆ *skl1-inv1*
by (*rule inv-rule-basic*) (*auto*)

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*
by (*auto simp add*: *skl1-def skl1-init-def skl1-inv2-def*)

lemma *skl1-inv2-trans* [*iff*]:
{*skl1-inv2*} *trans skl1* {> *skl1-inv2*}
apply (*auto simp add*: *PO-hoare-defs intro*!: *skl1-inv2I*)
apply (*drule can-signal-trans*, *assumption*)
apply (*auto simp add*: *skl1-nostep-defs*)
apply (*auto simp add*: *skl1-defs ik-dy-def skl1-inv2-def*)
done

lemma *PO-skl1-inv2* [*iff*]: *reach skl1* ⊆ *skl1-inv2*
by (*rule inv-rule-basic*) (*auto*)

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*
apply (*auto intro*!:*skl1-inv3I*)
apply (*auto dest*!: *skl1-inv2D*)
apply (*rename-tac x A B Rb Ra*)
apply (*case-tac*

signalsInit x (*Commit A B*
⟨*NonceF* (*Ra* $ *ni*), *NonceF* (*Rb* $ *nr*),
Exp (*Exp Gen* (*NonceF* (*Rb* $ *ny*))) (*NonceF* (*Ra* $ *nx*))⟩) > *0*, *auto*)
apply (*fastforce dest*: *skl1-inv1D elim*: *equalityE*)
done

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*
by (*auto simp add*: *skl1-def skl1-init-def skl1-inv4-def*)

lemma *skl1-inv4-trans* [*iff*]:
{*skl1-inv4*} *trans skl1* {> *skl1-inv4*}
apply (*auto simp add*: *PO-hoare-defs skl1-nostep-defs intro*!: *skl1-inv4I*)
apply (*auto simp add*: *skl1-inv4-def skl1-defs ik-dy-def domIff dest*: *Exp-Exp-Gen-inj2* [*OF sym*])
done

lemma *PO-skl1-inv4* [*iff*]: *reach skl1* ⊆ *skl1-inv4*
by (*rule inv-rule-basic*) (*auto*)

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*
by (*auto simp add*: *skl1-def skl1-init-def skl1-inv5-def*)

lemma *skl1-inv5-trans* [*iff*]:
{*skl1-inv5*} *trans skl1* {> *skl1-inv5*}
apply (*auto simp add*: *PO-hoare-defs intro*!: *skl1-inv5I*)
apply (*drule can-signal-trans*, *assumption*)
apply (*auto simp add*: *skl1-nostep-defs*)
apply (*auto simp add*: *skl1-defs ik-dy-def dest*: *skl1-inv5D*)
done

lemma *PO-skl1-inv5* [*iff*]: *reach skl1* ⊆ *skl1-inv5*
by (*rule inv-rule-basic*) (*auto*)

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 (*auto intro*!: *skl1-inv6I*)
fix *s*::*skl1-state* **fix** *A B Rb Ra*
assume *HRun*:*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* (*Exp Gen* (*NonceF* (*Rb* $ *ny*)))
guessed-frame Ra xnr = *Some* (*NonceF* (*Rb* $ *nr*))
can-signal s A B
assume *HRb*: *progress s Rb* = *Some* {*xny*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*}
assume *I4*:*s* ∈ *skl1-inv4*
assume *I5*:*s* ∈ *skl1-inv5*
from *I4 HRb* **have** *signalsResp s* (*Commit A B* ⟨*NonceF* (*Ra*$*ni*), *NonceF* (*Rb*$*nr*),
Exp (*Exp Gen* (*NonceF* (*Rb* $ *ny*))) (*NonceF* (*Ra* $ *nx*))⟩) > *0* ⟹ *False*
proof (*auto dest*!: *skl1-inv4D*)
assume {*xny*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*, *xEnd*} = {*xny*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*}
thus *?thesis* **by** *force*
qed
then have *HC*:*signalsResp s* (*Commit A B* ⟨*NonceF* (*Ra*$*ni*), *NonceF* (*Rb*$*nr*),
Exp (*Exp Gen* (*NonceF* (*Rb* $ *ny*))) (*NonceF* (*Ra* $ *nx*))⟩) = *0*
by *auto*
from *I5 HRun* **have** *signalsResp s* (*Running A B* ⟨*NonceF* (*Ra*$*ni*), *NonceF* (*Rb*$*nr*),
Exp (*Exp Gen* (*NonceF* (*Rb* $ *ny*))) (*NonceF* (*Ra* $ *nx*))⟩) > *0*
by (*auto dest*!: *skl1-inv5D*)
with *HC* **show** *signalsResp s* (*Commit A B* ⟨*NonceF* (*Ra*$*ni*), *NonceF* (*Rb*$*nr*),
Exp (*Exp Gen* (*NonceF* (*Rb* $ *ny*))) (*NonceF* (*Ra* $ *nx*))⟩)
< *signalsResp s* (*Running A B* ⟨*NonceF* (*Ra*$*ni*), *NonceF* (*Rb*$*nr*),
Exp (*Exp Gen* (*NonceF* (*Rb* $ *ny*))) (*NonceF* (*Ra* $ *nx*))⟩)
by *auto*
qed

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*}
apply (*auto simp add*: *PO-rhoare-defs R0sk1iai-def*)
apply (*simp add*: *l1-learn-def*)
done

Protocol events.

lemma *skl1-step1-refines-a0i-skip-i*:
{*R0sk1iai*} *Id*, *skl1-step1 Ra A B* {>*R0sk1iai*}
by (*auto simp add*: *PO-rhoare-defs R0sk1iai-def skl1-step1-def*)

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*}
by (*auto simp add*: *PO-rhoare-defs R0sk1iai-def*,
simp-all add: *skl1-step2-def a0i-running-def*, *auto*)

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*}
apply (*auto simp add*: *PO-rhoare-defs R0sk1iai-def*)
apply (*auto simp add*: *skl1-step3-def a0i-commit-def*)
apply (*frule skl1-inv3D*, *auto*)+
done

lemma *skl1-step4-refines-a0i-skip-i*:
{*R0sk1iai*} *Id*, *skl1-step4 Rb A B Ni gnx* {>*R0sk1iai*}
by (*auto simp add*: *PO-rhoare-defs R0sk1iai-def*, *auto simp add*: *skl1-step4-def*)

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*)
by (*auto simp add*: *R0sk1iai-def a0i-defs skl1-defs*)

lemma *skl1-refines-trans-a0i-i* [*iff*]:
{*R0sk1iai* ∩ (*UNIV* × (*skl1-inv1* ∩ *skl1-inv2*))} *trans a0i*, *trans skl1* {> *R0sk1iai*}
proof –
let *?pre′* = *R0sk1iai* ∩ (*UNIV* × *skl1-inv3*)
show *?thesis* (**is** {*?pre*} *?t1*, *?t2* {>*?post*})
proof (*rule relhoare-conseq-left*)
show *?pre* ⊆ *?pre′*
using *skl1-inv3-derived* **by** *blast*
next
show {*?pre′*} *?t1*, *?t2* {> *?post*}
apply (*auto simp add*: *a0i-def skl1-def a0i-trans-def skl1-trans-def*)
prefer *2* **using** *skl1-step2-refines-a0i-running-skip-i* **apply** (*simp add*: *PO-rhoare-defs*, *blast*)
prefer *2* **using** *skl1-step3-refines-a0i-commit-skip-i* **apply** (*simp add*: *PO-rhoare-defs*, *blast*)
apply (*blast intro*!:*skl1-trans-refines-a0i-trans-i*)+
done

qed
qed

lemma *obs-consistent-med01iai* [*iff*]:
obs-consistent R0sk1iai med0sk1iai a0i skl1
by (*auto simp add*: *obs-consistent-def R0sk1iai-def med0sk1iai-def*)

refinement result

lemma *skl1-refines-a0i-i* [*iff*]:
refines
(*R0sk1iai* ∩ (*reach a0i* × (*skl1-inv1* ∩ *skl1-inv2*)))
med0sk1iai a0i skl1
by (*rule Refinement-using-invariants*, *auto*)

lemma *skl1-implements-a0i-i* [*iff*]: *implements med0sk1iai a0i skl1*
by (*rule refinement-soundness*) (*fast*)

26.7 Derived invariants: injective agreement (*Init* authenticates *Resp*)

lemma *skl1-obs-iagreement-Init* [*iff*]: *oreach skl1* ⊆ *l1-iagreement-Init*
apply (*rule external-invariant-translation*
[*OF PO-a0i-obs-agreement - skl1-implements-a0i-i*])
apply (*auto simp add*: *med0sk1iai-def l1-iagreement-Init-def a0i-agreement-def*)
done

lemma *skl1-iagreement-Init* [*iff*]: *reach skl1* ⊆ *l1-iagreement-Init*
by (*rule external-to-internal-invariant* [*OF skl1-obs-iagreement-Init*], *auto*)

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*}
apply (*auto simp add*: *PO-rhoare-defs R0sk1iar-def*)
apply (*simp add*: *l1-learn-def*)
done

Protocol events.

lemma *skl1-step1-refines-a0i-skip-r*:
{*R0sk1iar*} *Id*, *skl1-step1 Ra A B* {>*R0sk1iar*}
by (*auto simp add*: *PO-rhoare-defs R0sk1iar-def skl1-step1-def*)

lemma *skl1-step2-refines-a0i-skip-r*:
{*R0sk1iar*} *Id*, *skl1-step2 Rb A B Ni gnx* {>*R0sk1iar*}
by (*auto simp add*: *PO-rhoare-defs R0sk1iar-def*, *auto simp add*:*skl1-step2-def*)

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*}
by (*auto simp add*: *PO-rhoare-defs R0sk1iar-def*,
simp-all add: *skl1-step3-def a0i-running-def*, *auto*)

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*}
apply (*auto simp add*: *PO-rhoare-defs R0sk1iar-def*)
apply (*auto simp add*: *skl1-step4-def a0i-commit-def*)
apply (*auto dest*!: *skl1-inv6D* [*rotated 1*])
done

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*)
by (*auto simp add*: *R0sk1iar-def a0i-defs skl1-defs*)

lemma *skl1-refines-trans-a0i-r* [*iff*]:
{*R0sk1iar* ∩ (*UNIV* × (*skl1-inv4* ∩ *skl1-inv5*))} *trans a0i*, *trans skl1* {> *R0sk1iar*}
proof −
let *?pre′* = *R0sk1iar* ∩ (*UNIV* × *skl1-inv6*)
show *?thesis* (**is** {*?pre*} *?t1*, *?t2* {>*?post*})
proof (*rule relhoare-conseq-left*)
show *?pre* ⊆ *?pre′*
using *skl1-inv6-derived* **by** *blast*
next
show {*?pre′*} *?t1*, *?t2* {> *?post*}
apply (*auto simp add*: *a0i-def skl1-def a0i-trans-def skl1-trans-def*)
prefer *3* **using** *skl1-step3-refines-a0i-running-skip-r* **apply** (*simp add*: *PO-rhoare-defs*, *blast*)
prefer *3* **using** *skl1-step4-refines-a0i-commit-skip-r* **apply** (*simp add*: *PO-rhoare-defs*, *blast*)

apply (*blast intro*!:*skl1-trans-refines-a0i-trans-r*)+
done
qed
qed

lemma *obs-consistent-med0sk1iar* [*iff*]:
obs-consistent R0sk1iar med0sk1iar a0i skl1
by (*auto simp add*: *obs-consistent-def R0sk1iar-def med0sk1iar-def*)

Refinement result.

lemma *skl1-refines-a0i-r* [*iff*]:
refines
(*R0sk1iar* ∩ (*reach a0i* × (*skl1-inv4* ∩ *skl1-inv5*)))
med0sk1iar a0i skl1
by (*rule Refinement-using-invariants*, *auto*)

lemma *skl1-implements-a0i-r* [*iff*]: *implements med0sk1iar a0i skl1*
by (*rule refinement-soundness*) (*fast*)

26.9 Derived invariants: injective agreement (*Resp* authenticates *Init*)

lemma *skl1-obs-iagreement-Resp* [*iff*]: *oreach skl1* ⊆ *l1-iagreement-Resp*
apply (*rule external-invariant-translation*
[*OF PO-a0i-obs-agreement - skl1-implements-a0i-r*])
apply (*auto simp add*: *med0sk1iar-def l1-iagreement-Resp-def a0i-agreement-def*)
done

lemma *skl1-iagreement-Resp* [*iff*]: *reach skl1* ⊆ *l1-iagreement-Resp*
by (*rule external-to-internal-invariant* [*OF skl1-obs-iagreement-Resp*], *auto*)

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* $\notin$ *bad-init* $\wedge$ *test-partner* $\notin$ *bad-init*
by *auto*

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' ≡ ∀ *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*
by (*cases x*, *auto*)

lemma *role-comp-inv-eq*:
  y = *role-comp x* ⟷ *x* = *role-comp y*
by (*auto elim*!: *role-comp.elims* [*OF sym*])

definition
  partners :: *rid-t set*
where
  partners ≡ {*R*. *partner-runs test R*}

lemma *test-not-partner* [*simp*]:
  test ∉ *partners*
by (*auto simp add*: *partners-def partner-runs-def*, *cases role* (*guessed-runs test*), *auto*)

lemma *matching-symmetric*:
  matching sigma sigma' ⟹ *matching sigma' sigma*
by (*auto simp add*: *matching-def*)

lemma *partner-symmetric*:
  partner-runs R R' ⟹ *partner-runs R' R*
by (*auto simp add*: *partner-runs-def matching-symmetric*)

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-runs R R″ $\Longrightarrow$ *partner-runs R R′* $\Longrightarrow$ *R′ = R″*
proof –
assume *H′:partner-runs R R′*
then have *Hm′: matching (guessed-frame R) (guessed-frame R′)*
by (*auto simp add: partner-runs-def*)
assume *H″:partner-runs R R″*
then have *Hm″: matching (guessed-frame R) (guessed-frame R″)*
by (*auto simp add: partner-runs-def*)
show *?thesis*
proof (*cases role (guessed-runs R′)*)
case *Init*
with *H′ partner-symmetric [OF H″]* **have** *Hrole:role (guessed-runs R) = Resp*
role (guessed-runs R″) = Init
by (*auto simp add: partner-runs-def*)
with *Init Hm′* **have** *guessed-frame R xgnx = Some (Exp Gen (NonceF (R′$nx)))*
by (*simp add: matching-def*)
moreover from *Hrole Hm″* **have** *guessed-frame R xgnx = Some (Exp Gen (NonceF (R″$nx)))*
by (*simp add: matching-def*)
ultimately show *?thesis* **by** (*auto dest: Exp-Gen-inj*)
next
case *Resp*
with *H′ partner-symmetric [OF H″]* **have** *Hrole:role (guessed-runs R) = Init*
role (guessed-runs R″) = Resp
by (*auto simp add: partner-runs-def*)
with *Resp Hm′* **have** *guessed-frame R xgny = Some (Exp Gen (NonceF (R′$ny)))*
by (*simp add: matching-def*)
moreover from *Hrole Hm″* **have** *guessed-frame R xgny = Some (Exp Gen (NonceF (R″$ny)))*
by (*simp add: matching-def*)
ultimately show *?thesis* **by** (*auto dest: Exp-Gen-inj*)
qed
qed

lemma *partner-test*:
R $\in$ *partners* $\Longrightarrow$ *partner-runs R R′* $\Longrightarrow$ *R′ = test*
by (*auto intro!:partner-unique simp add:partners-def partner-symmetric*)

compromising events

definition
l2-lkr-others :: *agent* $\Rightarrow$ *l2-trans*
where
l2-lkr-others A ≡ {(*s*,*s′*).
— guards
A ≠ *test-owner* ∧
A ≠ *test-partner* ∧
— actions
s′ = *s*(|*bad* := {*A*} ∪ *bad s*|)
}

definition
l2-lkr-actor :: *agent* $\Rightarrow$ *l2-trans*
where
l2-lkr-actor A ≡ {(*s*,*s′*).
— guards

A = *test-owner* ∧
$A \neq$ *test-partner* ∧
— actions
$s' = s$⦇*bad* := {A} ∪ *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} ∪ *bad s*⦈
}

definition
l2-skr :: *rid-t* ⇒ *msg* ⇒ *l2-trans*
where
l2-skr R K ≡ {(s,s').
— guards
$R \neq$ *test* ∧ $R \notin$ *partners* ∧
in-progress (*progress s R*) *xsk* ∧
guessed-frame R xsk = *Some K* ∧
— actions
$s' = s$⦇*ik* := {K} ∪ *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 = λx. 0,
    signalsResp = λx. 0,
    signalsInit2 = λx. 0,
    signalsResp2 = λx. 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*
by (*simp add*: *l2-def*)

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*)
apply (*auto simp add*: *l2-nostep-defs*)
apply (*auto simp add*: *l2-defs*)
done
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
by (*auto simp add*: *can-signal-def l2-run-ended-trans*)

lemma *in-progressS-trans*:
in-progressS (*progress s R*) *S* $\Longrightarrow$ (s, s') $\in$ *trans l2* $\Longrightarrow$ *in-progressS* (*progress* s' *R*) *S*
apply (*auto simp add*: *l2-nostep-defs*)
apply (*auto simp add*: *l2-defs domIff*)
done

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 $\equiv$ {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 ⊆ *l2-inv1*
by (*auto simp add*: *l2-def l2-init-def l2-inv1-def can-signal-def bad-init-spec*)

lemma *l2-inv1-trans* [*iff*]:
  {*l2-inv1*} *trans l2* {> *l2-inv1*}
proof (*auto simp add*: *PO-hoare-defs intro*!: *l2-inv1I del*: *conjI*)
  fix *s′ s* :: *l2-state*
  fix *A B*
  assume *HI*:*s* ∈ *l2-inv1*
  assume *HT*:(*s*, *s′*) ∈ *trans l2*
  assume *can-signal s′ A B*
  with *HT* **have** *HS*:*can-signal s A B*
    by (*auto simp add*: *l2-can-signal-trans*)
  with *HI* **have** *A* ∉ *bad s* ∧ *B* ∉ *bad s*
    by *fast*
  with *HS HT* **show** *A* ∉ *bad s′* ∧ *B* ∉ *bad s′*
    by (*auto simp add*: *l2-nostep-defs can-signal-def*)
      (*simp-all add*: *l2-defs*)
qed

lemma *PO-l2-inv1* [*iff*]: *reach l2* ⊆ *l2-inv1*
by (*rule inv-rule-basic*) (*auto*)

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*
by (*auto simp add*: *l2-def l2-init-def l2-inv2-def*)

lemma *l2-inv2-trans* [*iff*]:
  {*l2-inv2*} *trans l2* {> *l2-inv2*}

apply (*auto simp add*: *PO-hoare-defs l2-nostep-defs intro*!: *l2-inv2I*)
apply (*auto simp add*: *l2-defs dy-fake-chan-def dest*: *l2-inv2D*)
done

lemma *PO-l2-inv2* [*iff*]: *reach l2* ⊆ *l2-inv2*
by (*rule inv-rule-basic*) (*auto*)

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*
by (*rule*, *rule*, *erule analz.induct*) (*auto*)

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*
by (*auto simp add*: *l2-def l2-init-def l2-inv3-def*)

lemma *l2-inv3-step1*:
  {*l2-inv3*} *l2-step1 Ra A B* {> *l2-inv3*}
apply (*auto simp add*: *PO-hoare-defs l2-nostep-defs intro*!: *l2-inv3I*)
apply (*auto simp add*: *l2-defs bad-runs-def intro*: *synth-analz-increasing dest*!: *l2-inv3D*)
apply (*auto simp add*: *l2-inv3-aux-defs intro*: *synth-analz-monotone*
    dest!: *subsetD* [**where** *A*=*chan* -])
done

lemma *l2-inv3-step2*:
  {*l2-inv3*} *l2-step2 Rb A B Ni gnx* {> *l2-inv3*}
apply (*auto simp add*: *PO-hoare-defs l2-nostep-defs intro*!: *l2-inv3I*)
apply (*auto simp add*: *l2-defs*)
apply (*auto simp add*: *bad-runs-def intro*: *synth-analz-increasing dest*!: *l2-inv3D*)
apply (*auto simp add*: *l2-inv3-aux-defs*)   — SLOW, ca. 30s
apply (*blast intro*: *synth-analz-monotone analz.intros insert-iff synth-analz-increasing*
    dest!: *subsetD* [**where** *A*=*chan* -])+
done

lemma *l2-inv3-step3*:
  {*l2-inv3*} *l2-step3 Ra A B Nr gny* {> *l2-inv3*}
apply (*auto simp add*: *PO-hoare-defs l2-nostep-defs intro*!: *l2-inv3I*)
apply (*auto simp add*: *l2-defs bad-runs-def intro*: *synth-analz-increasing dest*!: *l2-inv3D*)
apply (*auto simp add*: *l2-inv3-aux-defs*)
apply (*blast intro*: *synth-analz-monotone dest*!: *subsetD* [**where** *A*=*chan* -])+
done

lemma *l2-inv3-step4*:
  {*l2-inv3*} *l2-step4 Rb A B Ni gnx* {> *l2-inv3*}
apply (*auto simp add*: *PO-hoare-defs l2-nostep-defs intro*!: *l2-inv3I*, *auto simp add*: *l2-defs*)
apply (*auto simp add*: *bad-runs-def intro*:*synth-analz-increasing dest*!: *l2-inv3D*)
apply (*auto simp add*: *l2-inv3-aux-defs dest*!: *subsetD* [**where** *A*=*chan* -])
done

lemma *l2-inv3-dy-fake-msg*:
  {*l2-inv3*} *l2-dy-fake-msg M* {> *l2-inv3*}
apply (*auto simp add*: *PO-hoare-defs l2-defs extr-insert-IK-eq*
    intro!: *l2-inv3I*
    elim!: *l2-inv3E dy-fake-msg-extr* [*THEN* [*2*] *rev-subsetD*])
apply (*auto intro*!: *fake-New*
    intro: *synth-analz-increasing fake-monotone dy-fake-msg-monotone*
      dy-fake-msg-insert-chan
    simp add: *bad-runs-def elim*!: *l2-inv3E*)
apply (*auto simp add*: *l2-inv3-aux-defs intro*: *synth-analz-monotone*
    dest!: *subsetD* [**where** *A*=*chan* -])

done

lemma *l2-inv3-dy-fake-chan*:
{*l2-inv3*} *l2-dy-fake-chan M* {> *l2-inv3*}
apply (*auto simp add*: *PO-hoare-defs l2-defs*
intro!: *l2-inv3I*
elim!: *l2-inv3E*)
apply (*auto intro*: *synth-analz-increasing simp add*: *bad-runs-def elim*!: *l2-inv3E*
dest:*dy-fake-msg-extr* [*THEN* [*2*] *rev-subsetD*]
dy-fake-chan-extr-insert [*THEN* [*2*] *rev-subsetD*]
dy-fake-chan-mono2)
apply (*simp add*: *l2-inv3-aux-defs dy-fake-chan-def dy-fake-msg-def*,
erule fake.cases, *simp-all*)
apply (*auto simp add*: *l2-inv3-aux-defs elim*!: *synth-analz-monotone*
dest!: *subsetD* [**where** *A*=*chan* -])
done

lemma *l2-inv3-lkr-others*:
{*l2-inv3*} *l2-lkr-others A* {> *l2-inv3*}
apply (*auto simp add*: *PO-hoare-defs l2-defs*
intro!: *l2-inv3I*
dest!: *extr-insert-bad* [*THEN* [*2*] *rev-subsetD*]
elim!: *l2-inv3E*)
apply (*auto simp add*: *l2-inv3-aux-defs bad-runs-def*
intro: *synth-analz-increasing synth-analz-monotone*)
apply (*drule synth-analz-mono* [**where** *G*=*extr* - - -], *auto*,
(*drule rev-subsetD* [**where** *A*=*chan* -], *simp*)+, *auto intro*: *synth-analz-increasing*,
drule rev-subsetD [**where** *A*=*synth* (*analz* (*extr* - - -))],
auto intro: *synth-analz-monotone*)+
done

lemma *l2-inv3-lkr-after*:
{*l2-inv3*} *l2-lkr-after A* {> *l2-inv3*}
apply (*auto simp add*: *PO-hoare-defs l2-defs intro*!: *l2-inv3I*
dest!: *extr-insert-bad* [*THEN* [*2*] *rev-subsetD*]
elim!: *l2-inv3E*)
apply (*auto simp add*: *l2-inv3-aux-defs bad-runs-def*
intro: *synth-analz-increasing synth-analz-monotone*)
apply (*drule synth-analz-mono* [**where** *G*=*extr* - - -], *auto*,
(*drule rev-subsetD* [**where** *A*=*chan* -], *simp*)+, *auto intro*: *synth-analz-increasing*,
drule rev-subsetD [**where** *A*=*synth* (*analz* (*extr* - - -))],
auto intro: *synth-analz-monotone*)+
done

lemma *l2-inv3-skr*:
{*l2-inv3* ∩ *l2-inv2*} *l2-skr R K* {> *l2-inv3*}
apply (*auto simp add*: *PO-hoare-defs l2-defs intro*!: *l2-inv3I dest*!: *l2-inv2D*)
apply (*auto simp add*: *l2-inv3-aux-defs bad-runs-def intro*: *synth-analz-increasing*
elim!: *l2-inv3E*)
apply (*blast intro*: *synth-analz-monotone dest*!: *subsetD* [**where** *A*=*chan* -])+
done

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*}
by (*auto simp add*: *l2-nostep-defs intro*:*l2-inv3-trans-aux*)

lemma *PO-l2-inv3* [*iff*]: *reach l2* ⊆ *l2-inv3*
by (*rule-tac J*=*l2-inv2* **in** *inv-rule-incr*) (*auto*)

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*
apply (*drule l2-inv3D*, *auto*, *drule synth-analz-monotone*, *auto simp add*: *analz-generators*)
apply (*erule synth.cases*, *auto*)
done

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*
apply (*drule l2-inv3D*, *auto*, *drule synth-analz-monotone*, *auto simp add*: *analz-generators*)
apply (*erule synth.cases*, *auto*)
done

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*
by (*erule synth.cases*, *auto*)

lemma *parts-exp* [*simp*]:*parts* {*Exp X Y*} = {*Exp X Y*}
by (*auto*,*erule parts.induct*, *auto*)

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))
apply (auto simp add: l2-defs parts-insert [where H=extr - - -]
                      parts-insert [where H=insert - (extr - - -)])
apply (auto dest!:l2-inv2D)
apply (auto dest!:dy-fake-chan-extr-insert-parts [THEN [2] rev-subsetD]
                parts-monotone [of - {M} synth (analz (extr (bad s) (ik s) (chan s)))],
      auto simp add: dy-fake-msg-def)
done

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 )
apply (auto simp add: l2-defs parts-insert [where H=extr - - -]
                      parts-insert [where H=insert - (extr - - -)])
done

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)
apply (auto simp add: l2-defs parts-insert [where H=extr - - -]
                      parts-insert [where H=insert - (extr - - -)])
done

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 −
  assume A:K ∉ synth (analz (extr (bad s) (ik s) (chan s))) s ∈ l2-inv3
  assume hmac X K ∈ parts {M. ∃ x A B. Chan x A B M ∈ chan s}
  then obtain x A B M where H:hmac X K ∈ parts {M} and H':Chan x A B M ∈ chan s
    by (auto dest: parts-singleton)
  assume s ∈ l2-inv3
  with H' have M ∈ range Nonce  ∨ M ∈ synth (analz (extr (bad s) (ik s) (chan s)))
    by (auto simp add: l2-inv3-aux-defs dest!: l2-inv3D, auto)
  with H show ?thesis
```

proof (*auto*)
assume $M \in$ *synth* (*analz* (*extr* (*bad s*) (*ik s*) (*chan s*)))
then have $\{M\} \subseteq$ *synth* (*analz* (*extr* (*bad s*) (*ik s*) (*chan s*))) **by** (*auto*)
then have *parts* $\{M\} \subseteq$ *parts* (*synth* (*analz* (*extr* (*bad s*) (*ik s*) (*chan s*))))
by (*rule parts-mono*)
with *H* **have** *hmac X K* $\in$ *parts* (*synth* (*analz* (*extr* (*bad s*) (*ik s*) (*chan s*)))) **by** *auto*
with *A* **show** *?thesis* **by** *auto*
qed
qed

lemma *hmac-trans-lkr*:
hmac X K $\in$ *parts* (*extr* (*bad s'*) (*ik s'*) (*chan s'*)) $\Longrightarrow$
K $\notin$ *synth* (*analz* (*extr* (*bad s*) (*ik s*) (*chan s*))) $\Longrightarrow$
s $\in$ *l2-inv3* $\Longrightarrow$
(*s*, *s'*) $\in$ *l2-lkr-others A* $\cup$ *l2-lkr-after A* $\Longrightarrow$
hmac X K $\in$ *parts* (*extr* (*bad s*) (*ik s*) (*chan s*))
apply (*auto simp add*: *l2-defs*
dest!: *parts-monotone* [*OF - extr-insert-bad*])
apply (*auto intro*: *parts-monotone intro*!:*hmac-trans-lkr-aux*)
done

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 $\equiv$ {*s*. $\forall$ *Ra A B gny Nr*.
hmac $\langle$*Number 0*, *Exp Gen* (*NonceF* (*Ra$nx*)), *gny*, *Agent B*, *Agent A*$\rangle$
(*Hash* $\langle$*NonceF* (*Ra$ni*), *Nr*$\rangle$) $\in$ *parts* (*extr* (*bad s*) (*ik s*) (*chan s*)) $\longrightarrow$
guessed-runs Ra = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) $\longrightarrow$
A $\notin$ *bad s* $\land$ *B* $\notin$ *bad s* $\longrightarrow$
($\exists$ *Rb*. *guessed-runs Rb* = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|) $\land$
in-progressS (*progress s Rb*) {*xny*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*} $\land$
guessed-frame Rb xgny = *Some gny* $\land$
guessed-frame Rb xnr = *Some Nr* $\land$
guessed-frame Rb xni = *Some* (*NonceF* (*Ra$ni*)) $\land$
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 $\subseteq$ *l2-inv4*
by (*auto simp add*: *l2-def l2-init-def l2-inv4-def*)

lemma *l2-inv4-trans* [*iff*]:
  {*l2-inv4* ∩ *l2-inv2* ∩ *l2-inv3*} *trans l2* {> *l2-inv4*}
proof (*auto simp add*: *PO-hoare-defs intro*!: *l2-inv4I*)
  fix *s′ s* :: *l2-state*
  fix *Ra A B gny Nr*
  assume *HHparts*:*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′*))
  assume *HRa*: *guessed-runs Ra* = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|)
  assume *Hi*:*s* ∈ *l2-inv4 s* ∈ *l2-inv2 s* ∈ *l2-inv3*
  assume *Ht*:(*s*, *s′*) ∈ *trans l2*
  assume *A* ∉ *bad s′ B* ∉ *bad s′*
  with *Ht* **have** *Hb*:*A* ∉ *bad s B* ∉ *bad s*
    by (*auto simp add*: *l2-nostep-defs*) (*simp-all add*: *l2-defs*)
  with *HRa Hi*
  have *HH*:*Hash* ⟨*NonceF* (*Ra*$ *ni*), *Nr*⟩ ∉ *synth* (*analz* (*extr* (*bad s*) (*ik s*) (*chan s*)))
    by (*auto dest*!: *l2-inv3D-HashNonce1 simp add*: *bad-runs-def*)
  from *Ht Hi HHparts HH*
  have *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*)) ∨
       (∃ *Rb*. (*s*, *s′*) ∈ *l2-step2 Rb A B* (*NonceF* (*Ra*$*ni*)) (*Exp Gen* (*NonceF* (*Ra* $ *nx*))) ∧
              gny = *Exp Gen* (*NonceF* (*Rb*$*ny*)) ∧
              Nr = *NonceF* (*Rb*$*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* (*Exp Gen* (*NonceF* (*Ra*$*nx*))) ∧
              guessed-frame Rb xni = *Some* (*NonceF* (*Ra*$*ni*)))
    apply (*auto simp add*: *l2-nostep-defs*)

    apply (*drule hmac-trans-1-4-skr-extr-fake*, *auto*)
    apply (*drule hmac-trans-2*, *auto*)
    apply (*drule hmac-trans-3*, *auto*)
    apply (*drule hmac-trans-1-4-skr-extr-fake*, *auto*)
    apply (*drule hmac-trans-1-4-skr-extr-fake*, *auto*)
    apply (*drule hmac-trans-1-4-skr-extr-fake*, *auto*)
    apply (*drule hmac-trans-lkr*, *auto*)
    apply (*drule hmac-trans-lkr*, *auto*)
    apply (*drule hmac-trans-1-4-skr-extr-fake*, *auto*)
    done
  then show ∃ *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 (*auto*)
      assume
        hmac ⟨*Number 0*, *Exp Gen* (*NonceF* (*Ra* $ *nx*)), *gny*, *Agent B*, *Agent A*⟩
             (*hmac* (*NonceF* (*Ra* $ *ni*)) *Nr*)
          ∈ *parts* (*extr* (*bad s*) (*ik s*) (*chan s*))
      with *Hi Hb HRa* **obtain** *Rb* **where**
        HRb: *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*)))
by (*auto dest*!: *l2-inv4D*)
with *Ht* **have** *in-progressS* (*progress s′ Rb*) {*xny*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*}
by (*auto elim*: *in-progressS-trans*)
with *HRb* **show** *?thesis* **by** *auto*
qed
qed

lemma *PO-l2-inv4* [*iff*]: *reach l2* ⊆ *l2-inv4*
by (*rule-tac J*=*l2-inv2* ∩ *l2-inv3* **in** *inv-rule-incr*) (*auto*)

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 −
assume *s* ∈ *l2-inv1 can-signal s A B*
hence *Hb*:*A* ∉ *bad s B* ∉ *bad s* **by** *auto*
assume *Insec B A* ⟨*gny*, *hmac* ⟨*Number 0*, *Exp Gen* (*NonceF* (*Ra*$*nx*)), *gny*, *Agent B*, *Agent A*⟩
(*Hash* ⟨*NonceF* (*Ra*$*ni*), *Nr*⟩)⟩ ∈ *chan s*
hence *HH*:
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*)) **by** *auto*
assume *s* ∈ *l2-inv4 guessed-runs Ra* = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|)
with *Hb HH* **show** *?thesis* **by** *auto*
qed

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*
by (*auto simp add*: *l2-def l2-init-def l2-inv5-def*)

lemma *l2-inv5-trans* [*iff*]:
{*l2-inv5* ∩ *l2-inv2* ∩ *l2-inv3*} *trans l2* {> *l2-inv5*}
proof (*auto simp add*: *PO-hoare-defs intro*!: *l2-inv5I*)
fix *s′ s* :: *l2-state*
fix *Rb A B gnx Ni*
assume *HHparts*:*hmac* ⟨*Number* (*Suc 0*), *Exp Gen* (*NonceF* (*Rb*$*ny*)), *gnx*, *Agent A*, *Agent B*⟩
(*Hash* ⟨*Ni*, *NonceF* (*Rb*$*nr*)⟩)
∈ *parts* (*extr* (*bad s′*) (*ik s′*) (*chan s′*))
assume *HRb*: *guessed-runs Rb* = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|)
assume *Hi*:*s* ∈ *l2-inv5 s* ∈ *l2-inv2 s* ∈ *l2-inv3*
assume *Ht*:(*s*, *s′*) ∈ *trans l2*
assume *A* ∉ *bad s′ B* ∉ *bad s′*
with *Ht* **have** *Hb*:*A* ∉ *bad s B* ∉ *bad s*
by (*auto simp add*: *l2-nostep-defs*) (*simp-all add*: *l2-defs*)
with *HRb Hi* **have** *HH*:*Hash* ⟨*Ni*, *NonceF* (*Rb*$*nr*)⟩ ∉ *synth* (*analz* (*extr* (*bad s*) (*ik s*) (*chan s*)))
by (*auto dest*!: *l2-inv3D-HashNonce2 simp add*: *bad-runs-def*)
from *Ht Hi HHparts HH* **have** *hmac* ⟨*Number 1*, *Exp Gen* (*NonceF* (*Rb*$*ny*)), *gnx*, *Agent A*, *Agent B*⟩
(*Hash* ⟨*Ni*, *NonceF* (*Rb*$*nr*)⟩) ∈ *parts* (*extr* (*bad s*) (*ik s*) (*chan s*)) ∨
(∃ *Ra*. (*s*, *s′*) ∈ *l2-step3 Ra A B* (*NonceF* (*Rb*$*nr*)) (*Exp Gen* (*NonceF* (*Rb*$*ny*))) ∧
gnx = *Exp Gen* (*NonceF* (*Ra*$*nx*)) ∧
Ni = *NonceF* (*Ra*$*ni*) ∧
guessed-runs Ra = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ∧
progress s′ Ra = *Some* {*xnx*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*, *xEnd*} ∧
guessed-frame Ra xgny = *Some* (*Exp Gen* (*NonceF* (*Rb*$*ny*))) ∧
guessed-frame Ra xnr = *Some* (*NonceF* (*Rb*$*nr*)))
apply (*auto simp add*: *l2-nostep-defs*)

apply (*drule hmac-trans-1-4-skr-extr-fake*, *auto*)
apply (*drule hmac-trans-2*, *auto*)
apply (*drule hmac-trans-3*, *auto*)
apply (*drule hmac-trans-1-4-skr-extr-fake*, *auto*)
apply (*drule hmac-trans-1-4-skr-extr-fake*, *auto*)
apply (*drule hmac-trans-1-4-skr-extr-fake*, *auto*)
apply (*drule hmac-trans-lkr*, *auto*)

apply (*drule hmac-trans-lkr*, *auto*)
apply (*drule hmac-trans-1-4-skr-extr-fake*, *auto*)
done
then show ∃ *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 (*auto*)
assume
hmac ⟨*Number* (*Suc 0*), *Exp Gen* (*NonceF* (*Rb*$*ny*)), *gnx*, *Agent A*, *Agent B*⟩
(*Hash* ⟨*Ni*, *NonceF* (*Rb*$*nr*)⟩)
∈ *parts* (*extr* (*bad s*) (*ik s*) (*chan s*))
with *Hi Hb HRb* **obtain** *Ra* **where** *HRa*:*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*)))
by (*auto dest*!: *l2-inv5D*)
with *Ht* **have** *in-progressS* (*progress s′ Ra*) {*xnx*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*, *xEnd*}
by (*auto elim*: *in-progressS-trans*)
with *HRa* **show** *?thesis* **by** *auto*
qed
qed

lemma *PO-l2-inv5* [*iff*]: *reach l2* ⊆ *l2-inv5*
by (*rule-tac J*=*l2-inv2* ∩ *l2-inv3* **in** *inv-rule-incr*) (*auto*)

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 −
assume *s* ∈ *l2-inv1 can-signal s A B*
hence *Hb*:*A* ∉ *bad s B* ∉ *bad s* **by** *auto*
assume *Insec A B* (*hmac* ⟨*Number 1*, *Exp Gen* (*NonceF* (*Rb*$*ny*)), *gnx*, *Agent A*, *Agent B*⟩
(*Hash* ⟨*Ni*, *NonceF* (*Rb*$*nr*)⟩)) ∈ *chan s*
hence *HH*:
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*)) **by** *auto*
assume *s* ∈ *l2-inv5 guessed-runs Rb* = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|)
with *Hb HH* **show** *?thesis* **by** *auto*
qed

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*
by (*auto simp add*: *l2-def l2-init-def l2-inv6-def*)

lemma *l2-inv6-trans* [*iff*]:
{*l2-inv6*} *trans l2* {> *l2-inv6*}
apply (*auto simp add*: *PO-hoare-defs l2-nostep-defs intro*!: *l2-inv6I*)
apply (*auto simp add*: *l2-defs dy-fake-chan-def*)
done

lemma *PO-l2-inv6* [*iff*]: *reach l2* ⊆ *l2-inv6*
by (*rule inv-rule-basic*) (*auto*)

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'*
by (*auto simp add*: *l2-def l2-init-def l2-inv6'-def*)

lemma *l2-inv6'-trans* [*iff*]:
  {*l2-inv6'*} *trans l2* {> *l2-inv6'*}
apply (*auto simp add*: *PO-hoare-defs l2-nostep-defs intro*!: *l2-inv6'I*)
apply (*auto simp add*: *l2-defs dy-fake-chan-def*)
done

lemma *PO-l2-inv6'* [*iff*]: *reach l2* ⊆ *l2-inv6'*
by (*rule inv-rule-basic*) (*auto*)

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*
by (*auto simp add*: *l2-def l2-init-def l2-inv7-def*)

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 (*auto simp add*: *PO-hoare-defs intro*!: *l2-inv7I*)
  fix *s s'* :: *l2-state* **fix** *x*
  assume *Hi*:*s* ∈ *l2-inv1 s* ∈ *l2-inv7 s* ∈ *l2-inv4 s* ∈ *l2-inv6'*
  assume *Ht*:(*s*, *s'*) ∈ *l2-step3 Ra A B Nr gny*
  assume *Hs*:*x* ∈ *secret s'*
  from *Hs Ht* **have** *x* ∈ *secret s* ∨ (*Ra* = *test* ∧ *x* = *Exp gny* (*NonceF* (*Ra*$*nx*)))
    by (*auto simp add*: *l2-defs*)
  with *Hi Ht*
  show ∃ *N N' R'*. *x* = *Exp* (*Exp Gen* (*NonceF* (*R'* $ *N'*))) (*NonceF* (*test* $ *N*)) ∧
          R' ∈ *partners* ∧ (*N* = *nx* ∨ *N* = *ny*) ∧ (*N'* = *nx* ∨ *N'* = *ny*)
    proof (*auto dest*: *l2-inv7D simp add*: *l2-defs*)
      assume *Htest*: *guessed-runs test* = ⦇*role* = *Init*, *owner* = *A*, *partner* = *B*⦈
            guessed-frame test xgny = *Some gny*
            guessed-frame test xnr = *Some Nr*
            guessed-frame test xsk = *Some* (*Exp gny* (*NonceF* (*test* $ *nx*)))
      assume
        Insec B A ⟨*gny*, *hmac* ⟨*Number 0*, *Exp Gen* (*NonceF* (*test*$*nx*)), *gny*, *Agent B*, *Agent A*⟩
              (*Hash* ⟨*NonceF* (*test*$*ni*), *Nr*⟩)⟩
          ∈ *chan s*

can-signal s A B
with *Htest Hi* **obtain** *Rb* **where** *HRb*:
guessed-runs Rb = (|*role*=*Resp*, *owner*=*B*, *partner*=*A*|)
in-progressS (*progress s Rb*) {*xny*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*}
gny = *Exp Gen* (*NonceF* (*Rb*$*ny*))
Nr = *NonceF* (*Rb*$*nr*)
guessed-frame Rb xni = *Some* (*NonceF* (*test*$*ni*))
guessed-frame Rb xgnx = *Some* (*Exp Gen* (*NonceF* (*test*$*nx*)))
by (*auto dest*!: *auth-guard-step3*)
with *Hi*
have *guessed-frame Rb xsk* = *Some* (*Exp* (*Exp Gen* (*NonceF* (*Rb*$*ny*))) (*NonceF* (*test*$*nx*)))
by (*auto dest*: *l2-inv6'D*)
with *HRb Htest* **have** *Rb* ∈ *partners*
by (*auto simp add*: *partners-def partner-runs-def*, *simp add*: *matching-def*)
with *HRb* **have** *Exp gny* (*NonceF* (*test* $ *nx*)) =
Exp (*Exp Gen* (*NonceF* (*Rb* $ *ny*))) (*NonceF* (*test* $ *nx*)) ∧ *Rb* ∈ *partners*
by *auto*
then show ∃ *N N' R'*.
Exp gny (*NonceF* (*test* $ *nx*)) = *Exp* (*Exp Gen* (*NonceF* (*R'* $ *N'*))) (*NonceF* (*test* $ *N*)) ∧
R' ∈ *partners* ∧ (*N* = *nx* ∨ *N* = *ny*) ∧ (*N'* = *nx* ∨ *N'* = *ny*)
by *blast*
qed (*auto simp add*: *can-signal-def*)
qed

lemma *l2-inv7-step4*:
{*l2-inv7* ∩ *l2-inv1* ∩ *l2-inv5* ∩ *l2-inv6* ∩ *l2-inv6'*} *l2-step4 Rb A B Ni gnx* {> *l2-inv7*}
proof (*auto simp add*: *PO-hoare-defs intro*!: *l2-inv7I*)
fix *s s'* :: *l2-state* **fix** *x*
assume *Hi*:*s* ∈ *l2-inv1 s* ∈ *l2-inv7 s* ∈ *l2-inv5 s* ∈ *l2-inv6 s* ∈ *l2-inv6'*
assume *Ht*:(*s*, *s'*) ∈ *l2-step4 Rb A B Ni gnx*
assume *Hs*:*x* ∈ *secret s'*
from *Hs Ht* **have** *x* ∈ *secret s* ∨ (*Rb* = *test* ∧ *x* = *Exp gnx* (*NonceF* (*Rb*$*ny*)))
by (*auto simp add*: *l2-defs*)
with *Hi Ht*
show ∃ *N N' R'*. *x* = *Exp* (*Exp Gen* (*NonceF* (*R'* $ *N'*))) (*NonceF* (*test* $ *N*)) ∧ *R'* ∈ *partners*
∧ (*N* = *nx* ∨ *N* = *ny*) ∧ (*N'* = *nx* ∨ *N'* = *ny*)
proof (*auto dest*: *l2-inv7D simp add*: *l2-defs*)
assume *Htest*: *guessed-runs test* = (|*role* = *Resp*, *owner* = *B*, *partner* = *A*|)
guessed-frame test xgnx = *Some gnx*
guessed-frame test xni = *Some Ni*
assume *progress s test* = *Some* {*xny*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*}
with *Htest Hi* **have** *Htest'*: *guessed-frame test xsk* = *Some* (*Exp gnx* (*NonceF* (*test* $ *ny*)))
by (*auto dest*: *l2-inv6'D*)
assume
Insec A B (*hmac* ⟨*Number* (*Suc 0*), *Exp Gen* (*NonceF* (*test*$*ny*)), *gnx*, *Agent A*, *Agent B*⟩
(*Hash* ⟨*Ni*, *NonceF* (*test* $ *nr*)⟩))
∈ *chan s*
can-signal s A B
with *Hi Htest* **obtain** *Ra* **where** *HRa*:
guessed-runs Ra = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|)
in-progressS (*progress s Ra*) {*xnx*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*, *xEnd*}
gnx = *Exp Gen* (*NonceF* (*Ra*$*nx*))
Ni = *NonceF* (*Ra*$*ni*)

guessed-frame Ra xgny = Some (Exp Gen (NonceF (test$ny)))
guessed-frame Ra xnr = Some (NonceF (test$nr))
by (*auto dest*!: *auth-guard-step4*)
with *Hi*
have *guessed-frame Ra xsk = Some (Exp (Exp Gen (NonceF (Ra$nx))) (NonceF (test$ny)))*
by (*auto dest*: *l2-inv6D*)
with *HRa Htest Htest′* **have** *Ra* ∈ *partners*
by (*auto simp add*: *partners-def partner-runs-def*, *simp add*: *matching-def*)
with *HRa* **have** *Exp gnx (NonceF (test $ ny))* =
Exp (Exp Gen (NonceF (Ra $ nx))) (NonceF (test $ ny)) ∧ *Ra* ∈ *partners*
by *auto*
then show ∃ *N N′ R′*.
Exp gnx (NonceF (test $ ny))
= *Exp (Exp Gen (NonceF (R′ $ N′))) (NonceF (test $ N))* ∧
R′ ∈ *partners* ∧ (*N* = *nx* ∨ *N* = *ny*) ∧ (*N′* = *nx* ∨ *N′* = *ny*)
by *auto*
qed (*auto simp add*: *can-signal-def*)
qed

lemma *l2-inv7-trans* [*iff*]:
{*l2-inv7* ∩ *l2-inv1* ∩ *l2-inv4* ∩ *l2-inv5* ∩ *l2-inv6* ∩ *l2-inv6′*} *trans l2* {> *l2-inv7*}
apply (*auto simp add*: *l2-nostep-defs intro*!: *l2-inv7-step3 l2-inv7-step4*)
apply (*auto simp add*: *PO-hoare-defs intro*!: *l2-inv7I*)
apply (*auto simp add*: *l2-defs dy-fake-chan-def dest*: *l2-inv7D*)
done

lemma *PO-l2-inv7* [*iff*]: *reach l2* ⊆ *l2-inv7*
by (*rule-tac J*=*l2-inv1* ∩ *l2-inv4* ∩ *l2-inv5* ∩ *l2-inv6* ∩ *l2-inv6′* **in** *inv-rule-incr*) (*auto*)

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*
by (*erule synth.cases*, *auto dest*: *Exp-Exp-Gen-inj2*)

lemma *l2-inv7-aux*:
s ∈ *l2-inv7* ⟹
x ∈ *secret s* ⟹
x ∉ *synth (analz (generators s))*
apply (*auto simp add*: *analz-generators dest*!: *l2-inv7D* [*THEN* [*2*] *rev-subsetD*])
apply (*auto dest*!: *Exp-Exp-Gen-synth Exp-Exp-Gen-inj2*)
done

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*
by (*auto simp add*: *can-signal-def R12s-defs*)

Protocol events.

lemma *l2-step1-refines-step1*:
{*R12s*} *skl1-step1 Ra A B*, *l2-step1 Ra A B* {>*R12s*}
by (*auto simp add*: *PO-rhoare-defs R12s-defs skl1-step1-def l2-step1-def*)

lemma *l2-step2-refines-step2*:
{*R12s*} *skl1-step2 Rb A B Ni gnx*, *l2-step2 Rb A B Ni gnx* {>*R12s*}
by (*auto simp add*: *PO-rhoare-defs R12s-defs skl1-step2-def*, *simp-all add*: *l2-step2-def*)

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 (*auto simp add*: *PO-rhoare-defs R12s-defs*)
fix *s s*′
assume *Hi*:*s* ∈ *l2-inv1 s* ∈ *l2-inv4 s*∈ *l2-inv6*′
assume *Ht*: (*s*, *s*′) ∈ *l2-step3 Ra A B Nr gny*
assume *s* ∈ *l2-inv7 s* ∈ *l2-inv3*
with *Hi Ht l2-inv7-step3 l2-inv3-step3* **have** *Hi*′:*s*′ ∈ *l2-inv7 s*′∈ *l2-inv3*
by (*auto simp add*: *PO-hoare-defs*, *blast*, *blast*)
from *Ht* **have** *Ra* = *test* ⟶ *Exp gny* (*NonceF* (*Ra*$*nx*)) ∈ *secret s*′
by (*auto simp add*: *l2-defs*)
with *Hi*′ **have** *Ra* = *test* ⟶ *Exp gny* (*NonceF* (*Ra*$*nx*)) ∉ *synth* (*analz* (*generators s*′))
by (*auto dest*: *l2-inv7-aux*)
with *Hi*′ **have** *G2*:*Ra* = *test* ⟶ *Exp gny* (*NonceF* (*Ra*$*nx*)) ∉ *synth* (*analz* (*ik s*′))
by (*auto dest*!: *l2-inv3D-aux*)

from *Ht Hi* **have** *G1*:
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*} ∧
gny = *Exp Gen* (*NonceF* (*Rb*$*ny*)) ∧
Nr = *NonceF* (*Rb*$*nr*) ∧
guessed-frame Rb xgnx = *Some* (*Exp Gen* (*NonceF* (*Ra*$*nx*))) ∧
guessed-frame Rb xni = *Some* (*NonceF* (*Ra*$*ni*)))
by (*auto dest*!: *auth-guard-step3 simp add*: *l2-defs*)
with *Ht G1 G2* **show**
((|*ik* = *ik s*, *secret* = *secret s*, *progress* = *progress s*,
signalsInit = *signalsInit s*, *signalsResp* = *signalsResp s*,
signalsInit2 = *signalsInit2 s*, *signalsResp2* = *signalsResp2 s*|),
(|*ik* = *ik s*′, *secret* = *secret s*′, *progress* = *progress s*′,
signalsInit = *signalsInit s*′, *signalsResp* = *signalsResp s*′,
signalsInit2 = *signalsInit2 s*′, *signalsResp2* = *signalsResp2 s*′|))
∈ *skl1-step3 Ra A B Nr gny*
apply (*auto simp add*: *l2-step3-def*, *auto simp add*: *skl1-step3-def*)
apply (*auto simp add*: *can-signal-def*)
done
qed

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 (*auto simp add*: *PO-rhoare-defs R12s-defs*)
fix *s s*′
assume *Hi*:*s* ∈ *l2-inv1 s* ∈ *l2-inv5 s* ∈ *l2-inv6 s* ∈ *l2-inv6*′
assume *Ht*: (*s*, *s*′) ∈ *l2-step4 Rb A B Ni gnx*
assume *s* ∈ *l2-inv7 s* ∈ *l2-inv3*
with *Hi Ht l2-inv7-step4 l2-inv3-step4* **have** *Hi*′:*s*′ ∈ *l2-inv7 s*′ ∈ *l2-inv3*
by (*auto simp add*: *PO-hoare-defs*, *blast*, *blast*)
from *Ht* **have** *Rb* = *test* ⟶ *Exp gnx* (*NonceF* (*Rb*$*ny*)) ∈ *secret s*′
by (*auto simp add*: *l2-defs*)
with *Hi*′ **have** *Rb* = *test* ⟶ *Exp gnx* (*NonceF* (*Rb*$*ny*)) ∉ *synth* (*analz* (*generators s*′))
by (*auto dest*: *l2-inv7-aux*)
with *Hi*′ **have** *G2*:*Rb* = *test* ⟶ *Exp gnx* (*NonceF* (*Rb*$*ny*)) ∉ *synth* (*analz* (*ik s*′))
by (*auto dest*!: *l2-inv3D-aux*)
from *Ht Hi* **have** *G1*:
can-signal s A B ⟶ (∃ *Ra*. *guessed-runs Ra* = (|*role*=*Init*, *owner*=*A*, *partner*=*B*|) ∧
in-progressS (*progress s Ra*) {*xnx*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*, *xEnd*} ∧
guessed-frame Ra xgnx = *Some gnx* ∧
guessed-frame Ra xni = *Some Ni* ∧
guessed-frame Ra xgny = *Some* (*Exp Gen* (*NonceF* (*Rb*$*ny*))) ∧
guessed-frame Ra xnr = *Some* (*NonceF* (*Rb*$*nr*)))
by (*auto dest*!: *auth-guard-step4 simp add*: *l2-defs*)
with *Ht G1 G2* **show**
((|*ik* = *ik s*, *secret* = *secret s*, *progress* = *progress s*,
signalsInit = *signalsInit s*, *signalsResp* = *signalsResp s*,
signalsInit2 = *signalsInit2 s*, *signalsResp2* = *signalsResp2 s*|),
(|*ik* = *ik s*′, *secret* = *secret s*′, *progress* = *progress s*′,
signalsInit = *signalsInit s*′, *signalsResp* = *signalsResp s*′,
signalsInit2 = *signalsInit2 s*′, *signalsResp2* = *signalsResp2 s*′|))

$\in$ *skl1-step4 Rb A B Ni gnx*
apply (*auto simp add*: *l2-step4-def*, *auto simp add*: *skl1-step4-def*)
apply (*auto simp add*: *can-signal-def*)
done
qed

attacker events

lemma *l2-dy-fake-chan-refines-skip*:
{*R12s*} *Id*, *l2-dy-fake-chan M* {>*R12s*}
by (*auto simp add*: *PO-rhoare-defs R12s-defs l2-defs*)

lemma *l2-dy-fake-msg-refines-learn*:
{*R12s* ∩ *UNIV* × (*l2-inv3* ∩ *l2-inv7*)} *l1-learn m*, *l2-dy-fake-msg m* {>*R12s*}
apply (*auto simp add*: *PO-rhoare-defs R12s-defs l2-loc-defs l1-defs*)
apply (*drule Fake-insert-dy-fake-msg*, *erule l2-inv3D* [*THEN conjunct1*])
apply (*auto dest*!: *l2-inv7-aux*)
done

compromising events

lemma *l2-lkr-others-refines-skip*:
{*R12s*} *Id*, *l2-lkr-others A* {>*R12s*}
by (*auto simp add*: *PO-rhoare-defs R12s-defs l2-loc-defs l1-defs*)

lemma *l2-lkr-after-refines-skip*:
{*R12s*} *Id*, *l2-lkr-after A* {>*R12s*}
by (*auto simp add*: *PO-rhoare-defs R12s-defs l2-loc-defs l1-defs*)

lemma *l2-skr-refines-learn*:
{*R12s* ∩ *UNIV* × (*l2-inv2* ∩ *l2-inv3* ∩ *l2-inv7*)} *l1-learn K*, *l2-skr R K* {>*R12s*}
proof (*auto simp add*: *PO-rhoare-defs R12s-defs l2-loc-defs l1-defs*)
fix *s* :: *l2-state* **fix** *x*
assume *H*:
s ∈ *l2-inv2 s* ∈ *l2-inv3*
R ∉ *partners R* ≠ *test in-progress* (*progress s R*) *xsk guessed-frame R xsk* = *Some K*
assume *Hx*:*x* ∈ *synth* (*analz* (*insert K* (*ik s*)))
assume *x* ∈ *secret s s* ∈ *l2-inv7*
then obtain *R R′ N N′* **where** *Hx′*:*x* = *Exp* (*Exp Gen* (*NonceF* (*R*$*N*))) (*NonceF* (*R′*$*N′*))
R = *test* ∧ *R′* ∈ *partners* ∧ (*N*=*nx* ∨ *N*=*ny*) ∧ (*N′*=*nx* ∨ *N′*=*ny*)
by (*auto dest*!: *l2-inv7D subsetD*)
from *H* **have** *s* (|*ik* := *insert K* (*ik s*)|) ∈ *l2-inv3*
by (*auto intro*: *hoare-apply* [*OF l2-inv3-skr*] *simp add*: *l2-defs*)
with *Hx* **have** *x* ∈ *synth* (*analz* (*generators* (*s* (|*ik* := *insert K* (*ik s*)|))))
by (*auto dest*: *l2-inv3D-aux*)
with *Hx′* **show** *False*
by (*auto dest*!: *Exp-Exp-Gen-synth dest*: *Exp-Exp-Gen-inj2 simp add*: *analz-generators*)
qed

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*)
by (*auto simp add*: *R12s-defs skl1-defs l2-loc-defs*)

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*}
by (*auto 0 3 simp add*: *skl1-def l2-def skl1-trans-def l2-trans-def*
intro!: *l2-trans-refines-l1-trans*)

lemma *PO-obs-consistent-R12s* [*iff*]:
obs-consistent R12s med12s skl1 l2
by (*auto simp add*: *obs-consistent-def R12s-def med12s-def l2-defs*)

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
by (*rule Refinement-using-invariants*, *auto*)

lemma *l2-implements-l1* [*iff*]:
implements med12s skl1 l2
by (*rule refinement-soundness*) (*auto*)

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*
apply (*rule external-invariant-translation*
[*OF skl1-obs-secrecy - l2-implements-l1*])
apply (*auto simp add*: *med12s-def s0-secrecy-def l2-partial-secrecy-def*)
done

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*
apply (*auto simp add*: *oreach-def*, *rule*, *simp-all*,

simp add: *l2-def l2-trans-def l2-dy-fake-msg-def*)
apply *blast*
done

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*
apply (*auto simp add*:*l2-secrecy-def*)
apply (*drule l2-oreach-dy-fake-msg*, *simp-all*)
apply (*drule l2-obs-partial-secrecy* [*THEN* [*2*] *rev-subsetD*], *simp add*: *l2-partial-secrecy-def*)
apply *blast*
done

lemma *l2-secrecy* [*iff*]: *reach l2* ⊆ *l2-secrecy*
by (*rule external-to-internal-invariant* [*OF l2-obs-secrecy*], *auto*)

abbreviation *l2-iagreement-Init* ≡ *l1-iagreement-Init*

lemma *l2-obs-iagreement-Init* [*iff*]: *oreach l2* ⊆ *l2-iagreement-Init*
apply (*rule external-invariant-translation*
[*OF skl1-obs-iagreement-Init - l2-implements-l1*])
apply (*auto simp add*: *med12s-def l1-iagreement-Init-def*)
done

lemma *l2-iagreement-Init* [*iff*]: *reach l2* ⊆ *l2-iagreement-Init*
by (*rule external-to-internal-invariant* [*OF l2-obs-iagreement-Init*], *auto*)

abbreviation *l2-iagreement-Resp* ≡ *l1-iagreement-Resp*

lemma *l2-obs-iagreement-Resp* [*iff*]: *oreach l2* ⊆ *l2-iagreement-Resp*
apply (*rule external-invariant-translation*
[*OF skl1-obs-iagreement-Resp - l2-implements-l1*])
apply (*auto simp add*: *med12s-def l1-iagreement-Resp-def*)
done

lemma *l2-iagreement-Resp* [*iff*]: *reach l2* ⊆ *l2-iagreement-Resp*
by (*rule external-to-internal-invariant* [*OF l2-obs-iagreement-Resp*], *auto*)

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:
$s' = s(\!|$
  $progress := (progress\ s)(Ra \mapsto \{xnx,\ xni,\ xgnx\}),$
  $ik := \{implConfid\ A\ B\ (NonceF\ (Ra\$ni))\} \cup$
  $(\{implInsec\ A\ B\ (Exp\ Gen\ (NonceF\ (Ra\$nx)))\} \cup$
  $(ik\ s))$
$|\!)$
}

definition
$l3\text{-}step2 :: rid\text{-}t \Rightarrow agent \Rightarrow agent \Rightarrow msg \Rightarrow msg \Rightarrow l3\text{-}trans$
where
$l3\text{-}step2\ Rb\ A\ B\ Ni\ gnx \equiv \{(s,\ s').$
— guards:
$guessed\text{-}runs\ Rb = (\!|role{=}Resp,\ owner{=}B,\ partner{=}A|\!) \wedge$
$Rb \notin dom\ (progress\ s) \wedge$
$guessed\text{-}frame\ Rb\ xgnx = Some\ gnx \wedge$
$guessed\text{-}frame\ Rb\ xni = Some\ Ni \wedge$
$guessed\text{-}frame\ Rb\ xsk = Some\ (Exp\ gnx\ (NonceF\ (Rb\$ny))) \wedge$
$implConfid\ A\ B\ Ni \in ik\ s \wedge$
$implInsec\ A\ B\ gnx \in ik\ s \wedge$
— actions:
$s' = s(\!|\ progress := (progress\ s)(Rb \mapsto \{xny,\ xni,\ xnr,\ xgny,\ xgnx,\ xsk\}),$
$ik := \{implConfid\ B\ A\ (NonceF\ (Rb\$nr))\} \cup$
$(\{implInsec\ B\ A\ \langle Exp\ Gen\ (NonceF\ (Rb\$ny)),$
$hmac\ \langle Number\ 0,\ gnx,\ Exp\ Gen\ (NonceF\ (Rb\$ny)),\ Agent\ B,\ Agent\ A\rangle$
$(Hash\ \langle Ni,\ NonceF\ (Rb\$nr)\rangle)\rangle\ \} \cup$
$(ik\ s)),$
$signalsInit :=$
$if\ can\text{-}signal\ s\ A\ B\ then$
$addSignal\ (signalsInit\ s)$
$(Running\ A\ B\ \langle Ni,\ NonceF\ (Rb\$nr),\ Exp\ gnx\ (NonceF\ (Rb\$ny))\rangle)$
$else$
$signalsInit\ s,$
$signalsInit2 :=$
$if\ can\text{-}signal\ s\ A\ B\ then$
$addSignal\ (signalsInit2\ s)\ (Running\ A\ B\ (Exp\ gnx\ (NonceF\ (Rb\$ny))))$
$else$
$signalsInit2\ s$
$|\!)$
}

definition
$l3\text{-}step3 :: rid\text{-}t \Rightarrow agent \Rightarrow agent \Rightarrow msg \Rightarrow msg \Rightarrow l3\text{-}trans$
where
$l3\text{-}step3\ Ra\ A\ B\ Nr\ gny \equiv \{(s,\ s').$
— guards:
$guessed\text{-}runs\ Ra = (\!|role{=}Init,\ owner{=}A,\ partner{=}B|\!) \wedge$
$progress\ s\ Ra = Some\ \{xnx,\ xni,\ xgnx\} \wedge$
$guessed\text{-}frame\ Ra\ xgny = Some\ gny \wedge$
$guessed\text{-}frame\ Ra\ xnr = Some\ Nr \wedge$

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*
by (*auto elim*!: *parts.induct*, *auto simp add*: *ik-init-def*)

lemma *analz-ik-init* [*simp*]: *analz ik-init* = *ik-init*
by (*auto dest*: *analz-into-parts*)

lemma *abs-ik-init* [*iff*]: *abs ik-init* = {}
apply (*auto elim*!: *absE*)
apply (*auto simp add*: *ik-init-def*)
done

lemma *payloadSet-ik-init* [*iff*]: *ik-init* ∩ *payload* = {}
by (*auto simp add*: *ik-init-def*)

lemma *validSet-ik-init* [*iff*]: *ik-init* ∩ *valid*= {}
by (*auto simp add*: *ik-init-def*)

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*
by (*simp add*: *l3-def*)

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*
by (*auto simp add*: *l3-inv1-def*)

lemma *l3-inv1-init* [*iff*]:
  init l3 ⊆ *l3-inv1*
by (*auto simp add*: *l3-def l3-init-def intro*!:*l3-inv1I*)

lemma *l3-inv1-trans* [*iff*]:
  {*l3-inv1*} *trans l3* {> *l3-inv1*}
apply (*auto simp add*: *PO-hoare-defs l3-nostep-defs intro*!: *l3-inv1I*)
apply (*auto simp add*: *l3-defs dy-fake-msg-def dy-fake-chan-def*
        parts-insert [**where** *H*=*ik* -] *parts-insert* [**where** *H*=*insert* - (*ik* -)]
        dest!: *Fake-parts-insert*)
apply (*auto dest*:*analz-into-parts*)
done

lemma *PO-l3-inv1* [*iff*]:
  reach l3 ⊆ *l3-inv1*
by (*rule inv-rule-basic*) (*auto*)

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*
by (*auto simp add*: *l3-def l3-init-defs intro*!:*l3-inv2I Keys-badI*)

lemma *l3-inv2-trans* [*simp*,*intro*!]:
  {*l3-inv2* ∩ *l3-inv1*} *trans l3* {> *l3-inv2*}
apply (*auto simp add*: *PO-hoare-defs l3-nostep-defs intro*!: *l3-inv2I*)
apply (*auto simp add*: *l3-defs dy-fake-msg-def dy-fake-chan-def*)

4 subgoals: dy, lkr*, skr.

apply (*auto intro*: *Keys-bad-insert-Fake Keys-bad-insert-keys-of*)
apply (*auto intro*!: *Keys-bad-insert-payload*)
done

lemma *PO-l3-inv2* [*iff*]: *reach l3* ⊆ *l3-inv2*
by (*rule-tac J*=*l3-inv1* **in** *inv-rule-incr*) (*auto*)

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*
apply (*auto simp add*: *l3-def l3-init-def intro*!: *l3-inv3I*)
apply (*auto simp add*: *ik-init-def intro*!: *synth-increasing* [*THEN* [*2*] *rev-subsetD*])
done

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 (*simp add*: *l3-nostep-defs*, *safe*)
fix *Ra A B*
show {*l3-inv3*} *l3-step1 Ra A B* {> *l3-inv3*}
apply (*auto simp add*: *PO-hoare-def l3-defs intro*!: *l3-inv3I dest*!: *l3-inv3D*)
apply (*auto intro*!: *validI dest*!: *analz-insert-partition* [*THEN* [*2*] *rev-subsetD*])
done
next
fix *Rb A B Ni gnx*
show {*l3-inv3*} *l3-step2 Rb A B Ni gnx* {> *l3-inv3*}
apply (*auto simp add*: *PO-hoare-def l3-defs intro*!: *l3-inv3I dest*!: *l3-inv3D*)
apply (*auto intro*!: *validI dest*!: *analz-insert-partition* [*THEN* [*2*] *rev-subsetD*])
done
next
fix *Ra A B Nr gny*
show {*l3-inv3*} *l3-step3 Ra A B Nr gny* {> *l3-inv3*}
apply (*auto simp add*: *PO-hoare-def l3-defs intro*!: *l3-inv3I dest*!: *l3-inv3D*)
apply (*auto intro*!: *validI dest*!: *analz-insert-partition* [*THEN* [*2*] *rev-subsetD*])
done
next
fix *Rb A B Ni gnx*
show {*l3-inv3*} *l3-step4 Rb A B Ni gnx* {> *l3-inv3*}
apply (*auto simp add*: *PO-hoare-def l3-defs intro*!: *l3-inv3I dest*!: *l3-inv3D*)
done

next
  fix *m*
  show {*l3-inv3*} *l3-dy m* {> *l3-inv3*}
    apply (*auto simp add*: *PO-hoare-def l3-defs dy-fake-chan-def dy-fake-msg-def*
            intro!: *l3-inv3I dest*!: *l3-inv3D*)
    apply (*drule synth-analz-insert*)
    apply (*blast intro*: *synth-analz-monotone dest*: *synth-monotone*)
    done
next
  fix *A*
  show {*l3-inv3*} *l3-lkr-others A* {> *l3-inv3*}
    apply (*auto simp add*: *PO-hoare-def l3-defs intro*!: *l3-inv3I dest*!: *l3-inv3D*)
    apply (*drule analz-Un-partition* [*of - keys-of A*], *auto*)
    done
next
  fix *A*
  show {*l3-inv3*} *l3-lkr-after A* {> *l3-inv3*}
    apply (*auto simp add*: *PO-hoare-def l3-defs intro*!: *l3-inv3I dest*!: *l3-inv3D*)
    apply (*drule analz-Un-partition* [*of - keys-of A*], *auto*)
    done
next
  fix *R K*
  show {*l3-inv3*} *l3-skr R K* {> *l3-inv3*}
    apply (*auto simp add*: *PO-hoare-def l3-defs intro*!: *l3-inv3I dest*!: *l3-inv3D*)
    apply (*auto dest*!: *analz-insert-partition* [*THEN* [*2*] *rev-subsetD*])
    done
qed

lemma *PO-l3-inv3* [*iff*]: *reach l3* ⊆ *l3-inv3*
by (*rule inv-rule-basic*) (*auto*)

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*
by (*auto simp add*: *l3-def l3-init-def ik-init-def intro*!:*l3-inv4I*)

lemma *l3-inv4-trans* [*simp*,*intro*!]:
  {*l3-inv4*} *trans l3* {> *l3-inv4*}
apply (*auto simp add*: *PO-hoare-defs l3-nostep-defs intro*!: *l3-inv4I*)
apply (*auto simp add*: *l3-defs dy-fake-chan-def dy-fake-msg-def*)
done

lemma *PO-l3-inv4* [*simp*,*intro*!]: *reach l3* ⊆ *l3-inv4*
by (*rule inv-rule-basic*) (*auto*)

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*
by (*auto simp add*: *abs-validSet dy-fake-msg-def intro*!: *l3-inv5I*
    dest!: *l3-inv3D* [*THEN synth-mono*, *THEN* [*2*] *rev-subsetD*]
    dest!: *synth-analz-NI-I-K-synth-analz-NI-E* [*THEN* [*2*] *rev-subsetD*])

lemma *PO-l3-inv5* [*simp*,*intro*!]: *reach l3* ⊆ *l3-inv5*
using *l3-inv5-derived PO-l3-inv2 PO-l3-inv3*
by *blast*

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*
apply (*auto intro*!: *l3-inv6I dest*!: *l3-inv3D*)

1 subgoal

apply (*drule synth-mono*, *simp*, *drule subsetD*, *assumption*)
apply (*auto dest*!: *implInsec-synth-analz* [*rotated 1*, **where** *H*=- ∪ -])
apply (*auto dest*!: *synth-analz-monotone* [*of* - - ∪ - *ik* -])
done

lemma *PO-l3-inv6* [*simp*,*intro*!]: *reach l3* ⊆ *l3-inv6*
using *l3-inv6-derived PO-l3-inv3 PO-l3-inv4*
by (*blast*)

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*
apply (*auto intro*!: *l3-inv7I dest*!: *l3-inv3D*)

1 subgoal

apply (*drule synth-mono*, *simp*, *drule subsetD*, *assumption*)
apply (*auto dest*!: *implConfid-synth-analz* [*rotated 1*, **where** *H*=- ∪ -])
apply (*auto dest*!: *synth-analz-monotone* [*of* - - ∪ - *ik* -])
done

lemma *PO-l3-inv7* [*simp*,*intro*!]: *reach l3* ⊆ *l3-inv7*
using *l3-inv7-derived PO-l3-inv3 PO-l3-inv4*
by (*blast*)

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*
apply (*auto intro*!: *l3-inv8I dest*!: *l3-inv3D l3-inv2D*)

2 subgoals: M is deducible and the agents are bad

apply (*drule synth-mono*, *simp*, *drule subsetD*, *assumption*)
apply (*auto dest*!: *implAuth-synth-analz* [*rotated 1*, **where** *H*=- ∪ -] *elim*!: *synth-analz-monotone*)

apply (*drule synth-mono*, *simp*, *drule subsetD*, *assumption*)
apply (*auto dest*!: *implAuth-synth-analz* [*rotated 1*, **where** *H*=- ∪ -] *elim*!: *synth-analz-monotone*)
done

lemma *PO-l3-inv8* [*iff*]: *reach l3* ⊆ *l3-inv8*
using *l3-inv8-derived*
PO-l3-inv3 PO-l3-inv2 PO-l3-inv4
by *blast*

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*
apply (*auto intro*!: *l3-inv9I dest*!:*l3-inv3D l3-inv2D*)

2 subgoals: M is deducible and the agents are bad.

apply (*drule synth-mono*, *simp*, *drule subsetD*, *assumption*)
apply (*auto dest*!: *implSecure-synth-analz* [*rotated 1*, **where** *H*=- ∪ -]
elim!: *synth-analz-monotone*)

apply (*drule synth-mono*, *simp*, *drule subsetD*, *assumption*)
apply (*auto dest*!: *implSecure-synth-analz* [*rotated 1*, **where** *H*=- ∪ -])
done

lemma *PO-l3-inv9* [*iff*]: *reach l3* ⊆ *l3-inv9*
using *l3-inv9-derived*
PO-l3-inv3 PO-l3-inv2 PO-l3-inv4
by *blast*

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*
by (*auto simp add*: *R23s-def med23s-def*)

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*
by (*auto simp add*: *R23s-def med23s-def*)

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*
by (*auto simp add*: *R23s-def med23s-def*)

lemma *can-signal-R23* [*simp*]:
(*s2*, *s3*) ∈ *R23s* ⟹
can-signal s2 A B ⟷ *can-signal s3 A B*
by (*auto simp add*: *can-signal-def*)

28.3.1 Protocol events

lemma *l3-step1-refines-step1*:
{*R23s*} *l2-step1 Ra A B*, *l3-step1 Ra A B* {>*R23s*}
apply (*auto simp add*: *PO-rhoare-defs R23s-defs*)
apply (*auto simp add*: *l3-defs l2-step1-def*)
done

lemma *l3-step2-refines-step2*:
{*R23s*} *l2-step2 Rb A B Ni gnx*, *l3-step2 Rb A B Ni gnx*{>*R23s*}
apply (*auto simp add*: *PO-rhoare-defs R23s-defs l2-step2-def*)
apply (*auto simp add*: *l3-step2-def*)
done

lemma *l3-step3-refines-step3*:
{*R23s*} *l2-step3 Ra A B Nr gny*, *l3-step3 Ra A B Nr gny* {>*R23s*}
apply (*auto simp add*: *PO-rhoare-defs R23s-defs l2-step3-def*)
apply (*auto simp add*: *l3-step3-def*)
done

lemma *l3-step4-refines-step4*:
{*R23s*} *l2-step4 Rb A B Ni gnx*, *l3-step4 Rb A B Ni gnx* {>*R23s*}
apply (*auto simp add*: *PO-rhoare-defs R23s-defs l2-step4-def*)
apply (*auto simp add*: *l3-step4-def*)
done

28.3.2 Intruder events

lemma *l3-dy-payload-refines-dy-fake-msg*:
M ∈ *payload* ⟹
{*R23s* ∩ *UNIV* × *l3-inv5*} *l2-dy-fake-msg M*, *l3-dy M* {>*R23s*}
apply (*auto simp add*: *PO-rhoare-defs R23s-defs*)
apply (*auto simp add*: *l3-defs l2-dy-fake-msg-def dest*: *l3-inv5D*)
done

lemma *l3-dy-valid-refines-dy-fake-chan*:
⟦ *M* ∈ *valid*; *M'* ∈ *abs* {*M*} ⟧ ⟹
{*R23s* ∩ *UNIV* × (*l3-inv5* ∩ *l3-inv6* ∩ *l3-inv7* ∩ *l3-inv8* ∩ *l3-inv9*)}
l2-dy-fake-chan M', *l3-dy M*
{>*R23s*}
apply (*auto simp add*: *PO-rhoare-defs R23s-defs*, *simp add*: *l2-dy-fake-chan-def*)
apply (*auto simp add*: *l3-defs*)

1 subgoal

apply (*erule valid-cases*, *simp-all add*: *dy-fake-chan-def*)

Insec

apply (*blast dest*: *l3-inv6D l3-inv5D*)

Confid

apply (*blast dest*: *l3-inv7D l3-inv5D*)

Auth

apply (*blast dest*: *l3-inv8D l3-inv5D*)

Secure

apply (*blast dest*: *l3-inv9D l3-inv5D*)
done

lemma *l3-dy-valid-refines-dy-fake-chan-Un*:
M ∈ *valid* ⟹
{*R23s* ∩ *UNIV* × (*l3-inv5* ∩ *l3-inv6* ∩ *l3-inv7* ∩ *l3-inv8* ∩ *l3-inv9*)}
⋃*M'*. *l2-dy-fake-chan M'*, *l3-dy M*
{>*R23s*}
by (*auto dest*: *valid-abs intro*: *l3-dy-valid-refines-dy-fake-chan*)

lemma *l3-dy-isLtKey-refines-skip*:
{*R23s*} *Id*, *l3-dy* (*LtK ltk*) {>*R23s*}
apply (*auto simp add*: *PO-rhoare-defs R23s-defs l3-defs*)

apply (*auto elim*!: *absE*)
done

lemma *l3-dy-others-refines-skip*:
$\llbracket$ *M* ∉ *range LtK*; *M* ∉ *valid*; *M* ∉ *payload* $\rrbracket \Longrightarrow$
{*R23s*} *Id*, *l3-dy M* {>*R23s*}
apply (*auto simp add*: *PO-rhoare-defs R23s-defs*)
apply (*auto simp add*: *l3-defs*)
apply (*auto elim*!: *absE intro*: *validI*)
done

lemma *l3-dy-refines-dy-fake-msg-dy-fake-chan-skip*:
{*R23s* ∩ *UNIV* × (*l3-inv5* ∩ *l3-inv6* ∩ *l3-inv7* ∩ *l3-inv8* ∩ *l3-inv9*)}
l2-dy-fake-msg M ∪ ($\bigcup$*M'*. *l2-dy-fake-chan M'*) ∪ *Id*, *l3-dy M*
{>*R23s*}
by (*cases M* ∈ *payload* ∪ *valid* ∪ *range LtK*)
(*auto dest*: *l3-dy-payload-refines-dy-fake-msg l3-dy-valid-refines-dy-fake-chan-Un*
intro: *l3-dy-isLtKey-refines-skip dest*!: *l3-dy-others-refines-skip*)

28.3.3 Compromise events

lemma *l3-lkr-others-refines-lkr-others*:
{*R23s*} *l2-lkr-others A*, *l3-lkr-others A* {>*R23s*}
apply (*auto simp add*: *PO-rhoare-defs R23s-defs*)
apply (*auto simp add*: *l3-defs l2-lkr-others-def*)
done

lemma *l3-lkr-after-refines-lkr-after*:
{*R23s*} *l2-lkr-after A*, *l3-lkr-after A* {>*R23s*}
apply (*auto simp add*: *PO-rhoare-defs R23s-defs*)
apply (*auto simp add*: *l3-defs l2-lkr-after-def*)
done

lemma *l3-skr-refines-skr*:
{*R23s*} *l2-skr R K*, *l3-skr R K* {>*R23s*}
apply (*auto simp add*: *PO-rhoare-defs R23s-defs*)
apply (*auto simp add*: *l3-defs l2-skr-def*)
done

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*)
by (*auto simp add*: *R23s-defs l2-defs l3-def l3-init-def*)

lemma *l3-refines-trans-l2* [*iff*]:
{*R23s* ∩ (*UNIV* × (*l3-inv1* ∩ *l3-inv2* ∩ *l3-inv3* ∩ *l3-inv4*))} *trans l2*, *trans l3* {> *R23s*}
proof −
let *?pre′* = *R23s* ∩ (*UNIV* × (*l3-inv5* ∩ *l3-inv6* ∩ *l3-inv7* ∩ *l3-inv8* ∩ *l3-inv9*))
show *?thesis* (**is** {*?pre*} *?t1*, *?t2* {>*?post*})
proof (*rule relhoare-conseq-left*)
show *?pre* ⊆ *?pre′*
using *l3-inv5-derived l3-inv6-derived l3-inv7-derived l3-inv8-derived l3-inv9-derived*
by *blast*
next
show {*?pre′*} *?t1*, *?t2* {> *?post*}
by (*auto simp add*: *l2-def l3-def l2-trans-def l3-trans-def*
intro!: *l3-trans-refines-l2-trans*)
qed
qed

lemma *PO-obs-consistent-R23s* [*iff*]:
obs-consistent R23s med23s l2 l3
by (*auto simp add*: *obs-consistent-def R23s-def med23s-def l2-defs*)

lemma *l3-refines-l2* [*iff*]:
refines
(*R23s* ∩
(*reach l2* × (*l3-inv1* ∩ *l3-inv2* ∩ *l3-inv3* ∩ *l3-inv4*)))
med23s l2 l3
by (*rule Refinement-using-invariants*, *auto*)

lemma *l3-implements-l2* [*iff*]:
implements med23s l2 l3
by (*rule refinement-soundness*) (*auto*)

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*
by (*auto simp add*: *l3-def l3-init-def ik-init-def intro*!:*l3-inv10I*)

lemma *l3-inv10-trans* [*iff*]:
{*l3-inv10*} *trans l3* {> *l3-inv10*}

apply (*auto simp add*: *PO-hoare-defs l3-nostep-defs*)
apply (*auto simp add*: *l3-defs l3-inv10-def*)
done

lemma *PO-l3-inv10* [*iff*]: *reach l3* ⊆ *l3-inv10*
by (*rule inv-rule-basic*) (*auto*)

lemma *l3-obs-inv10* [*iff*]: *oreach l3* ⊆ *l3-inv10*
by (*auto simp add*: *oreach-def*)

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*
apply (*rule external-invariant-translation* [*OF l2-obs-secrecy - l3-implements-l2*])
apply (*auto simp add*: *med23s-def l2-secrecy-def l3-partial-secrecy-def*)
done

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

lemma *l3-obs-secrecy* [*iff*]: *oreach l3* ⊆ *l3-secrecy*
apply (*rule*, *frule l3-obs-inv5* [*THEN* [*2*] *rev-subsetD*], *frule l3-obs-inv10* [*THEN* [*2*] *rev-subsetD*])
apply (*auto simp add*: *med23s-def l2-secrecy-def l3-secrecy-def s0-secrecy-def l3-inv10-def*)
using *l3-partial-secrecy-def* **apply** (*blast dest*!: *l3-inv5D subsetD* [*OF l3-obs-partial-secrecy*])
done

lemma *l3-secrecy* [*iff*]: *reach l3* ⊆ *l3-secrecy*
by (*rule external-to-internal-invariant* [*OF l3-obs-secrecy*], *auto*)

28.4.4 Injective agreement

abbreviation *l3-iagreement-Init* ≡ *l1-iagreement-Init*

lemma *l3-obs-iagreement-Init* [*iff*]: *oreach l3* ⊆ *l3-iagreement-Init*
apply (*rule external-invariant-translation*
  [*OF l2-obs-iagreement-Init - l3-implements-l2*])
apply (*auto simp add*: *med23s-def l1-iagreement-Init-def*)
done

lemma *l3-iagreement-Init* [*iff*]: *reach l3* ⊆ *l3-iagreement-Init*
by (*rule external-to-internal-invariant* [*OF l3-obs-iagreement-Init*], *auto*)

abbreviation *l3-iagreement-Resp* ≡ *l1-iagreement-Resp*

lemma *l3-obs-iagreement-Resp* [*iff*]: *oreach l3* ⊆ *l3-iagreement-Resp*
apply (*rule external-invariant-translation*
  [*OF l2-obs-iagreement-Resp - l3-implements-l2*])
apply (*auto simp add*: *med23s-def l1-iagreement-Resp-def*)
done

lemma *l3-iagreement-Resp* [*iff*]: *reach l3* ⊆ *l3-iagreement-Resp*
by (*rule external-to-internal-invariant* [*OF l3-obs-iagreement-Resp*], *auto*)

end
end

29 SKEME Protocol (L3 with asymmetric implementation)

theory *sklvl3-asymmetric*
imports *sklvl3 Implem-asymmetric*
begin

interpretation *sklvl3-asym*: *sklvl3 implem-asym*
by (*unfold-locales*)

end

30 SKEME Protocol (L3 with symmetric implementation)

theory *sklvl3-symmetric*
imports *sklvl3 Implem-symmetric*
begin

interpretation *sklvl3-sym*: *sklvl3 implem-sym*
by (*unfold-locales*)

end