Development of Security Protocols by Refinement

Christoph Sprenger and Ivano Somaini
ETH Zurich, Switzerland

March 19, 2025

Abstract

We propose a development method for security protocols based on stepwise refinement. Our refinement strategy transforms abstract security goals into protocols that are secure when operating over an insecure channel controlled by a Dolev-Yao-style intruder. As intermediate levels of abstraction, we employ messageless guard protocols and channel protocols communicating over channels with security properties. These abstractions provide insights on why protocols are secure and foster the development of families of protocols sharing common structure and properties. We have implemented our method in Isabelle/HOL and used it to develop different entity authentication and key establishment protocols, including realistic features such as key confirmation, replay caches, and encrypted tickets. Our development highlights that guard protocols and channel protocols provide fundamental abstractions for bridging the gap between security properties and standard protocol descriptions based on cryptographic messages. It also shows that our refinement approach scales to protocols of nontrivial size and complexity.

Contents

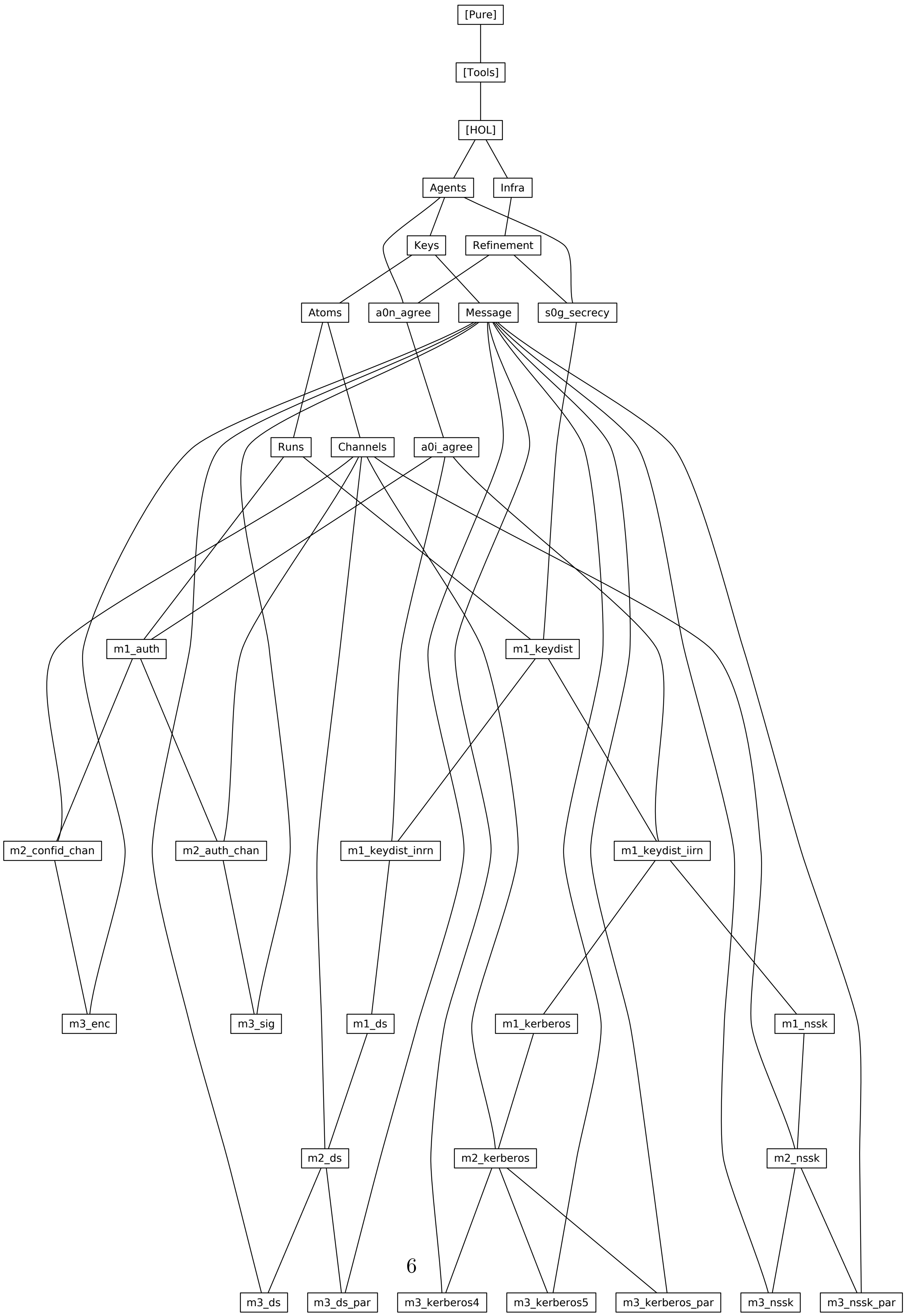

Figure 1: Theory dependencies

Preamble

Related Publications

The following papers describe our results in more detail:

- Christoph Sprenger and David Basin, *Developing Security Protocols by Refinement*, CCS 2010.
- Christoph Sprenger and David Basin, *Refining Key Establishment*, CSF 2012.
- Christoph Sprenger and David Basin, *Refining Security Protocols*, Journal of Computer Security (in submission), 2017.

Note: The Isabelle/HOL sources in this distribution also include the treatment of session key compromise. This is described in our journal paper (see above), which subsumes the CCS 2010 and CSF 2012 papers.

Mapping the model names in our papers to the Isabelle/HOL theories

For the sake of the presentation, the papers use shorter names for the models than the Isabelle theories. Here is a mapping of the names. On the left you find the model name used in the papers and on the right the corresponding Isabelle/HOL theory name. Note that the Isabelle theories contain a separate lemma or theorem for each invariant and refinement result.

Level 0

```
          Refinement/
s0           s0g_secrecy
a0n          a0n_agree
a0i          a0i_agree
```

Level 1

```
          Auth_simple/
a1           m1_auth

          Key_establish/
kt1          m1_keydist
kt1in        m1_keydist_iirn
kt1nn        m1_keydist_inrn
nssk1        m1_nssk
krb1         m1_kerberos
ds1          m1_ds
```

Level 2

```
        Auth_simple/
a2         m2_auth_chan
c2         m2_confid_chan

        Key_establish/
nssk2      m2_nssk
krb2       m2_kerberos
ds2        m2_ds
```

Level 3

```
        Auth_simple/
iso3       m3_sig
nsl3       m3_enc

        Key_establish/
nssk3d     m3_nssk_par
nssk3      m3_nssk
krb3d      m3_kerberos_par
krb3v      m3_kerberos5
krb3iv     m3_kerberos4
ds3d       m3_ds_par
ds3        m3_ds
```

Chapter 1

Protocol Modeling and Refinement Infrastructure

This chapter sets up our theory of refinement and the protocol modeling infrastructure.

1.1 Proving infrastructure

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

1.1.1 Prover configuration

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

1.1.2 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*]

1.1.3 General results

Maps

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

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

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

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

Set

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

Relations

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

Lists

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

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

lemma *take-prefix*: $[\![take\ n\ l = xs]\!] \Longrightarrow \exists xs'.\ l = xs\ @\ xs'$
⟨*proof*⟩

Finite sets

Cardinality.

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

lemma *finite-positive-cardI* [*intro*!]:
$[\![A \neq \{\};\ finite\ A]\!] \Longrightarrow 0 < card\ A$
⟨*proof*⟩

lemma *finite-positive-cardD* [*dest*!]:
$[\![0 < card\ A;\ finite\ A]\!] \Longrightarrow A \neq \{\}$

⟨*proof*⟩

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

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

end

1.2 Models, Invariants and Refinements

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

1.2.1 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*

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*
⟨*proof*⟩

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

States in behaviours are exactly reachable.

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

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

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

Specifications, observability, and implementation

Specifications add an observer function to transition systems.

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

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

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

Observable behaviour and reachability.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Internal and external invariants

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

lemma *external-from-internal-invariant-vimage*:
  $\llbracket$ *reach* $S \subseteq P$; $P \subseteq (obs\ S)-`Q$ $\rrbracket$
  $\Longrightarrow$ *oreach* $S \subseteq Q$
⟨*proof*⟩

lemma *external-to-internal-invariant-vimage*:
  $\llbracket$ *oreach* $S \subseteq Q$; $(obs\ S)-`Q \subseteq P$ $\rrbracket$
  $\Longrightarrow$ *reach* $S \subseteq P$
⟨*proof*⟩

lemma *external-to-internal-invariant*:
  $\llbracket$ *oreach* $S \subseteq Q$; $Q \subseteq (obs\ S)`P$; *observable* $(obs\ S)\ P$ $\rrbracket$
  $\Longrightarrow$ *reach* $S \subseteq P$
⟨*proof*⟩

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

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

Our notion of implementation is inclusion of observable behaviours.

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

Reflexivity and transitivity

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

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

Preservation of external invariants

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

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

lemma *external-invariant-translation*:

⟦ *oreach Sa* ⊆ *Q*; *pi*−'*Q* ⊆ *P*; *implements pi Sa Sc* ⟧
⟹ *oreach Sc* ⊆ *P*
⟨*proof*⟩

Preservation of internal invariants

lemma *internal-invariant-translation*:
⟦ *reach Sa* ⊆ *Pa*; *Pa* ⊆ *obs Sa* −' *Qa*; *pi* −' *Qa* ⊆ *Q*; *obs S* −' *Q* ⊆ *P*;
implements pi Sa S ⟧
⟹ *reach S* ⊆ *P*
⟨*proof*⟩

1.2.2 Invariants

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

Hoare triples

definition
PO-hoare :: [′*s set*, (′*s* × ′*s*) *set*, ′*s set*] ⇒ *bool*
(‹(*3*{-} - {> -})› [*0*, *0*, *0*] *90*)
where
{*pre*} *R* {> *post*} ≡ *R*''*pre* ⊆ *post*

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

lemma {*P*} *R* {> *Q*} = (∀ *s t*. *s* ∈ *P* ⟶ (*s*, *t*) ∈ *R* ⟶ *t* ∈ *Q*)
⟨*proof*⟩

Some essential facts about Hoare triples.

lemma *hoare-conseq-left* [*intro*]:
⟦ {*P*′} *R* {> *Q*}; *P* ⊆ *P*′ ⟧
⟹ {*P*} *R* {> *Q*}
⟨*proof*⟩

lemma *hoare-conseq-right*:
⟦ {*P*} *R* {> *Q*′}; *Q*′ ⊆ *Q* ⟧
⟹ {*P*} *R* {> *Q*}
⟨*proof*⟩

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

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

lemma *hoare-conj-right* [*intro*!]:
⟦ {*P*} *R* {> *Q1*}; {*P*} *R* {> *Q2*} ⟧
⟹ {*P*} *R* {> *Q1* ∩ *Q2*}
⟨*proof*⟩

Special transition relations.

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

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

lemma *hoare-trans-Un* [*iff*]:
$\{P\}\ R1 \cup R2\ \{>\ Q\} = (\{P\}\ R1\ \{>\ Q\} \wedge \{P\}\ R2\ \{>\ Q\})$
⟨*proof*⟩

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

Characterization of reachability

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

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

Useful consequences.

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

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

Invariant proof rules

Basic proof rule for invariants.

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

General invariant proof rule. This rule is complete (set $I = \mathit{reach}\ T$).

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

The following rule is equivalent to the previous one.

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

Proof of equivalence.

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

lemma *INV-rule-from-inv-rule*:
  ⟦ *init T* ⊆ *I*; {*I* ∩ *reach T*} (*trans T*) {> *I*} ⟧
  ⟹ *reach T* ⊆ *I*
⟨*proof*⟩

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

lemma *inv-rule-incr*:
  ⟦ *init T* ⊆ *I*; {*I* ∩ *J*} (*trans T*) {> *I*}; *reach T* ⊆ *J* ⟧
  ⟹ *reach T* ⊆ *I*
⟨*proof*⟩

1.2.3 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.

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*) ≡ λ*r s*. {(*xa*, *x*). ((λ*x xa*. (*x*, *xa*) ∈ *r*) *OO* (λ*x xa*. (*x*, *xa*) ∈ *s*)) *xa x*}.)

definition
  PO-rhoare ::
    [('*s* × '*t*) *set*, ('*s* × '*s*) *set*, ('*t* × '*t*) *set*, ('*s* × '*t*) *set*] ⇒ *bool*
    (‹(*4*{-} -, - {> -})› [*0*, *0*, *0*] *90*)
where
  {*pre*} *Ra*, *Rc* {> *post*} ≡ *pre O Rc* ⊆ *Ra O post*

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

Facts about relational Hoare tuples.

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

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

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

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

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

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

Reflexivity and transitivity.

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

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

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

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

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

Separate rule for cartesian products.

corollary *relhoare-cartesian*:

⟦ {*Domain pre*} *Ra* {> *I*}; {*Range pre*} *Rc* {> *J*};
  {*pre*} *Ra*, *Rc* {> *post*} ⟧ — any post, including *UNIV*!
⟹ {*pre*} *Ra*, *Rc* {> *I* × *J*}
⟨*proof*⟩

Unions of transition relations.

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

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

lemma *relhoare-abstract-Un-left* [*intro*]:
⟦ {*pre*} *Ra1*, *Rc* {> *post*} ⟧
⟹ {*pre*} *Ra1* ∪ *Ra2*, *Rc* {> *post*}
⟨*proof*⟩

lemma *relhoare-abstract-Un-right* [*intro*]:
⟦ {*pre*} *Ra2*, *Rc* {> *post*} ⟧
⟹ {*pre*} *Ra1* ∪ *Ra2*, *Rc* {> *post*}
⟨*proof*⟩

lemma *relhoare-abstract-UN* [*intro*!]: — might be too aggressive?
⟦ {*pre*} *Ra x*, *Rc* {> *post*} ⟧
⟹ {*pre*} ⋃*x*. *Ra x*, *Rc* {> *post*}
⟨*proof*⟩

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* × *'t*) *set*, (*'s*, *'a*) *TS-scheme*, (*'t*, *'b*) *TS-scheme*] ⇒ *bool*
where
PO-refines R Ta Tc ≡ (
  init Tc ⊆ *R*``(*init Ta*)
  ∧ {*R*} (*trans Ta*), (*trans Tc*) {> *R*}
)

lemma *PO-refinesI*:
⟦ *init Tc* ⊆ *R*``(*init Ta*); {*R*} (*trans Ta*), (*trans Tc*) {> *R*} ⟧ ⟹ *PO-refines R Ta Tc*
⟨*proof*⟩

lemma *PO-refinesE* [*elim*]:
⟦ *PO-refines R Ta Tc*; ⟦ *init Tc* ⊆ *R*``(*init Ta*); {*R*} (*trans Ta*), (*trans Tc*) {> *R*} ⟧ ⟹ *P* ⟧
⟹ *P*

⟨*proof*⟩

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

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

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

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

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

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

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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

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

The following rules are more useful in proofs.

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

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

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

Refinement of specifications

Lift relation membership to finite sequences

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

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

For each concrete behaviour there is a related abstract one.

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

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

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

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

lemma *obs-consistent-trans* [*intro*]:
  ⟦ *obs-consistent R1 pi1 S1 S2*; *obs-consistent R2 pi2 S2 S3* ⟧
  ⟹ *obs-consistent* (*R1 O R2*) (*pi1 o pi2*) *S1 S3*
⟨*proof*⟩

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

lemma *obs-consistent-conj1* [*intro*]:
  obs-consistent R pi Sa Sc ⟹ *obs-consistent* (*R* ∩ *R'*) *pi Sa Sc*
⟨*proof*⟩

lemma *obs-consistent-conj2* [*intro*]:
  obs-consistent R pi Sa Sc ⟹ *obs-consistent* (*R'* ∩ *R*) *pi Sa Sc*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *refinesE* [*elim*]:
  ⟦ *refines R pi Sa Sc*; ⟦ *PO-refines R Sa Sc*; *obs-consistent R pi Sa Sc* ⟧ ⟹ *P* ⟧
  ⟹ *P*
⟨*proof*⟩

Reflexivity and transitivity of refinement.

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

lemma *refinement-transitive*:
  ⟦ *refines R1 pi1 S1 S2*; *refines R2 pi2 S2 S3* ⟧
  ⟹ *refines* (*R1 O R2*) (*pi1 o pi2*) *S1 S3*
⟨*proof*⟩

Soundness of refinement for proving implementation

lemma *observable-behaviour-refinement*:

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

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

Extended versions of refinement proof rules including observations

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

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

Inhertitance of internal invariants through refinements

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

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

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

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

end

1.3 Atomic messages

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

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

1.3.1 Agents

datatype — We allow any number of agents plus an honest server.
agent = *Server* | *Agent nat*

consts
bad :: *agent set* — compromised agents

specification (*bad*)
Server-not-bad [*iff*]: *Server* $\notin$ *bad*
⟨*proof*⟩

abbreviation
good :: *agent set*
where
good ≡ −*bad*

abbreviation
Sv :: *agent*
where
Sv ≡ *Server*

1.3.2 Nonces

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

typedecl *fid-t*

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

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

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

Nonces

type-synonym
nonce = *fresh-t*

end

1.4 Symmetric and Assymetric Keys

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

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

datatype *ltkey* = — long-term keys
sharK agent — key shared with server
| *publK agent* — agent's public key
| *privK agent* — agent's private key

datatype *key* =

  sesK fresh-t — session key
| *ltK ltkey* — long-term key

abbreviation
  shrK :: *agent* ⇒ *key* **where**
  shrK A ≡ *ltK* (*sharK A*)

abbreviation
  pubK :: *agent* ⇒ *key* **where**
  pubK A ≡ *ltK* (*publK A*)

abbreviation
  priK :: *agent* ⇒ *key* **where**
  priK A ≡ *ltK* (*privK A*)

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

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

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

lemma *invKey-K*: *K* ∈ *symKeys* ⟹ *invKey K* = *K*
⟨*proof*⟩

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

lemma *invKey-invKey-id* [*simp*]: *invKey* (*invKey K*) = *K*
⟨*proof*⟩

lemma *invKey-eq* [*simp*]: (*invKey K* = *invKey K'*) = (*K*=*K'*)
⟨*proof*⟩

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

1.4.1 Asymmetric Keys

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

lemma *privateKey-neq-publicKey*: *priK A* ≠ *pubK A'*
⟨*proof*⟩

lemma *publicKey-neq-privateKey*: *pubK A* ≠ *priK A'*
⟨*proof*⟩

1.4.2 Basic properties of *pubK* and *priK*

lemma *publicKey-inject* [*iff*]: (*pubK A* = *pubK A'*) = (*A* = *A'*)

⟨*proof*⟩

lemma *not-symKeys-pubK* [*iff*]: $pubK\ A \notin symKeys$
⟨*proof*⟩

lemma *not-symKeys-priK* [*iff*]: $priK\ A \notin symKeys$
⟨*proof*⟩

lemma *symKey-neq-priK*: $K \in symKeys \Longrightarrow K \neq priK\ A$
⟨*proof*⟩

lemma *symKeys-neq-imp-neq*: $(K \in symKeys) \neq (K' \in symKeys) \Longrightarrow K \neq K'$
⟨*proof*⟩

lemma *symKeys-invKey-iff* [*iff*]: $(invKey\ K \in symKeys) = (K \in symKeys)$
⟨*proof*⟩

1.4.3 "Image" equations that hold for injective functions

lemma *invKey-image-eq* [*simp*]: $(invKey\ x \in invKey`A) = (x \in A)$
⟨*proof*⟩

lemma *invKey-pubK-image-priK-image* [*simp*]: $invKey\ `\ pubK\ `\ AS = priK\ `\ AS$
⟨*proof*⟩

lemma *publicKey-notin-image-privateKey*: $pubK\ A \notin priK\ `\ AS$
⟨*proof*⟩

lemma *privateKey-notin-image-publicKey*: $priK\ x \notin pubK\ `\ AA$
⟨*proof*⟩

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

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

1.4.4 Symmetric Keys

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

lemma *sym-sesK*: $sesK\ f \in symKeys$ — All session keys are symmetric
⟨*proof*⟩

lemma *sym-shrK*: $shrK\ X \in symKeys$ — All shared keys are symmetric
⟨*proof*⟩

Symmetric keys and inversion

lemma *symK-eq-invKey*: $[\![\ SK = invKey\ K;\ SK \in symKeys\ ]\!] \Longrightarrow K = SK$
⟨*proof*⟩

Image-related lemmas.

lemma *publicKey-notin-image-shrK*: $pubK\ x \notin shrK\ `\ AA$
⟨*proof*⟩

lemma *privateKey-notin-image-shrK*: $priK\ x \notin shrK\ `\ AA$
⟨*proof*⟩

lemma *shrK-notin-image-publicKey*: $shrK\ x \notin pubK\ `\ AA$
⟨*proof*⟩

lemma *shrK-notin-image-privateKey*: $shrK\ x \notin priK\ `\ AA$
⟨*proof*⟩

lemma *sesK-notin-image-shrK* [*simp*]: $sesK\ K \notin shrK`AA$
⟨*proof*⟩

lemma *shrK-notin-image-sesK* [*simp*]: $shrK\ K \notin sesK`AA$
⟨*proof*⟩

lemma *sesK-image-eq* [*simp*]: $(sesK\ x \in sesK\ `\ AA) = (x \in AA)$
⟨*proof*⟩

lemma *shrK-image-eq* [*simp*]: $(shrK\ x \in shrK\ `\ AA) = (x \in AA)$
⟨*proof*⟩

end

1.5 Atomic messages

theory *Atoms* **imports** *Keys*
begin

1.5.1 Atoms datatype

datatype *atom* =
aAgt agent
| *aNon nonce*
| *aKey key*
| *aNum nat*

1.5.2 Long-term key setup (abstractly)

Suppose an initial long-term key setup without looking into the structure of long-term keys.

Remark: This setup is agnostic with respect to the structure of the type *ltkey*. Ideally, the type *ltkey* should be a parameter of the type *key*, which is instantiated only at Level 3.

consts
ltkeySetup :: (*ltkey* × *agent*) *set* — LT key setup, for now unspecified

The initial key setup contains static, long-term keys.

definition
keySetup :: (*key* × *agent*) *set* **where**

keySetup ≡ {(*ltK K*, *A*) | *K A*. (*K*, *A*) ∈ *ltkeySetup*}

Corrupted keys are the long-term keys known by bad agents.

definition
corrKey :: *key set* **where**
corrKey ≡ *keySetup*$^{-1}$ ‘‘ *bad*

lemma *corrKey-Dom-keySetup* [*simp*, *intro*]: *K* ∈ *corrKey* ⟹ *K* ∈ *Domain keySetup*
⟨*proof*⟩

lemma *keySetup-noSessionKeys* [*simp*]: (*sesK K*, *A*) ∉ *keySetup*
⟨*proof*⟩

lemma *corrKey-noSessionKeys* [*simp*]: *sesK K* ∉ *corrKey*
⟨*proof*⟩

end

1.6 Protocol runs

theory *Runs* **imports** *Atoms*
begin

1.6.1 Runs

Define some typical roles.

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

fun
roleIdx :: *role-t* ⇒ *nat*
where
roleIdx Init = *0*
| *roleIdx Resp* = *1*
| *roleIdx Serv* = *2*

The type of runs is a partial function from run identifiers to a triple consisting of a role, a list of agents, and a list of atomic messages recorded during the run's execution.

The type of roles could be made a parameter for more flexibility.

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

type-synonym
runs-t = *rid-t* ⇀ *role-t* × *agent list* × *atom list*

1.6.2 Run abstraction

Define a function that lifts a function on roles and atom lists to a function on runs.

definition
map-runs :: ([*role-t*, *atom list*] ⇒ *atom list*) ⇒ *runs-t* ⇒ *runs-t*

where
  map-runs h runz rid ≡ *case runz rid of*
    None ⇒ *None*
  | *Some* (*rol*, *agts*, *al*) ⇒ *Some* (*rol*, *agts*, *h rol al*)

lemma *map-runs-empty* [*simp*]: *map-runs h Map.empty* = *Map.empty*
⟨*proof*⟩

lemma *map-runs-dom* [*simp*]: *dom* (*map-runs h runz*) = *dom runz*
⟨*proof*⟩

lemma *map-runs-update* [*simp*]:
  map-runs h (*runz*(*R* ↦ (*rol*, *agts*, *al*)))
  = (*map-runs h runz*)(*R* ↦ (*rol*, *agts*, *h rol al*))
⟨*proof*⟩

end

1.7 Channel Messages

theory *Channels* **imports** *Atoms*
begin

1.7.1 Channel messages

datatype *secprop* = *auth* | *confid*

type-synonym
  chtyp = *secprop set*

abbreviation
  secure :: *chtyp* **where**
  secure ≡ {*auth*, *confid*}

datatype *payload* = *Msg atom list*

datatype *chmsg* =
  StatCh chtyp agent agent payload
| *DynCh chtyp key payload*

Abbreviations for use in protocol defs

abbreviation
  Insec :: [*agent*, *agent*, *payload*] ⇒ *chmsg* **where**
  Insec ≡ *StatCh* {}

abbreviation
  Confid :: [*agent*, *agent*, *payload*] ⇒ *chmsg* **where**
  Confid ≡ *StatCh* {*confid*}

abbreviation

Auth :: [*agent*, *agent*, *payload*] ⇒ *chmsg* **where**
Auth ≡ *StatCh* {*auth*}

abbreviation
Secure :: [*agent*, *agent*, *payload*] ⇒ *chmsg* **where**
Secure ≡ *StatCh* {*auth*, *confid*}

abbreviation
dConfid :: [*key*, *payload*] ⇒ *chmsg* **where**
dConfid ≡ *DynCh* {*confid*}

abbreviation
dAuth :: [*key*, *payload*] ⇒ *chmsg* **where**
dAuth ≡ *DynCh* {*auth*}

abbreviation
dSecure :: [*key*, *payload*] ⇒ *chmsg* **where**
dSecure ≡ *DynCh* {*auth*, *confid*}

1.7.2 Keys used in dynamic channel messages

definition
keys-for :: *chmsg set* ⇒ *key set* **where**
keys-for H ≡ {*K*. ∃ *c M*. *DynCh c K M* ∈ *H*}

lemma *keys-forI* [*dest*]: *DynCh c K M* ∈ *H* ⟹ *K* ∈ *keys-for H*
⟨*proof*⟩

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

lemma *keys-for-monotone*: *G* ⊆ *H* ⟹ *keys-for G* ⊆ *keys-for H*
⟨*proof*⟩

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

lemma *keys-for-insert-StatCh* [*simp*]:
keys-for (*insert* (*StatCh c A B M*) *H*) = *keys-for H*
⟨*proof*⟩

lemma *keys-for-insert-DynCh* [*simp*]:
keys-for (*insert* (*DynCh c K M*) *H*) = *insert K* (*keys-for H*)
⟨*proof*⟩

1.7.3 Atoms in a set of channel messages

The set of atoms contained in a set of channel messages. We also include the public atoms, i.e., the agent names, numbers, and corrupted keys.

inductive-set
atoms :: *chmsg set* ⇒ *atom set*

for *H* :: *chmsg set*
where
at-StatCh: ⟦ *StatCh c A B* (*Msg M*) ∈ *H*; *At* ∈ *set M* ⟧ ⟹ *At* ∈ *atoms H*
| *at-DynCh*: ⟦ *DynCh c K* (*Msg M*) ∈ *H*; *At* ∈ *set M* ⟧ ⟹ *At* ∈ *atoms H*

declare *atoms.intros* [*intro*]

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

lemma *atoms-monotone*: *G* ⊆ *H* ⟹ *atoms G* ⊆ *atoms H*
⟨*proof*⟩

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

lemma *atoms-insert-StatCh* [*simp*]:
atoms (*insert* (*StatCh c A B* (*Msg M*)) *H*) = *set M* ∪ *atoms H*
⟨*proof*⟩

lemma *atoms-insert-DynCh* [*simp*]:
atoms (*insert* (*DynCh c K* (*Msg M*)) *H*) = *set M* ∪ *atoms H*
⟨*proof*⟩

1.7.4 Intruder knowledge (atoms)

Atoms that the intruder can extract from a set of channel messages.

inductive-set
extr :: *atom set* ⇒ *chmsg set* ⇒ *atom set*
for *T* :: *atom set*
and *H* :: *chmsg set*
where
extr-Inj: *At* ∈ *T* ⟹ *At* ∈ *extr T H*
| *extr-StatCh*:
⟦ *StatCh c A B* (*Msg M*) ∈ *H*; *At* ∈ *set M*; *confid* ∉ *c* ∨ *A* ∈ *bad* ∨ *B* ∈ *bad* ⟧
⟹ *At* ∈ *extr T H*
| *extr-DynCh*:
⟦ *DynCh c K* (*Msg M*) ∈ *H*; *At* ∈ *set M*; *confid* ∉ *c* ∨ *aKey K* ∈ *extr T H* ⟧
⟹ *At* ∈ *extr T H*

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

Typical parameter describing initial intruder knowledge.

definition
ik0 :: *atom set* **where**
ik0 ≡ *range aAgt* ∪ *range aNum* ∪ *aKey*‘*corrKey*

lemma *ik0-aAgt* [*iff*]: *aAgt A* ∈ *ik0*
⟨*proof*⟩

lemma *ik0-aNum* [*iff*]: *aNum T* ∈ *ik0*

⟨*proof*⟩

lemma *ik0-aNon* [*iff*]: *aNon N* $\notin$ *ik0*
⟨*proof*⟩

lemma *ik0-aKey-corr* [*simp*]: (*aKey K* $\in$ *ik0*) = (*K* $\in$ *corrKey*)
⟨*proof*⟩

Basic lemmas

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

lemma *extr-monotone* [*dest*]: *G* $\subseteq$ *H* $\Longrightarrow$ *extr T G* $\subseteq$ *extr T H*
⟨*proof*⟩

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

lemma *extr-monotone-param* [*dest*]: *T* $\subseteq$ *U* $\Longrightarrow$ *extr T H* $\subseteq$ *extr U H*
⟨*proof*⟩

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

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

lemma *extr-into-atoms* [*dest*]: *At* $\in$ *extr T H* $\Longrightarrow$ *At* $\in$ *T* $\cup$ *atoms H*
⟨*proof*⟩

Insertion lemmas for atom parameters

lemma *extr-insert-non-key-param* [*simp*]:
  assumes *At* $\in$ *range aNon* $\cup$ *range aAgt* $\cup$ *range aNum*
  shows *extr* (*insert At T*) *H* = *insert At* (*extr T H*)
⟨*proof*⟩

lemma *extr-insert-unused-key-param* [*simp*]:
  assumes *K* $\notin$ *keys-for H*
  shows *extr* (*insert* (*aKey K*) *T*) *H* = *insert* (*aKey K*) (*extr T H*)
⟨*proof*⟩

Insertion lemmas for each type of channel message

Note that the parameter accumulates the extracted atoms. In particular, these may include keys that may open further dynamically confidential messages.

lemma *extr-insert-StatCh* [*simp*]:
  extr T (*insert* (*StatCh c A B* (*Msg M*)) *H*)
   = (*if confid* $\notin$ *c* $\vee$ *A* $\in$ *bad* $\vee$ *B* $\in$ *bad then extr* (*set M* $\cup$ *T*) *H else extr T H*)
⟨*proof*⟩

lemma *extr-insert-DynCh* [*simp*]:
  extr T (*insert* (*DynCh c K* (*Msg M*)) *H*)

$= (if\ confid \notin c \vee aKey\ K \in extr\ T\ H\ then\ extr\ (set\ M \cup T)\ H\ else\ extr\ T\ H)$
⟨*proof*⟩

declare *extr.cases* [*rule del*, *elim*]

1.7.5 Faking messages

Channel messages that are fakeable from a given set of channel messages. Parameters are a set of atoms and a set of freshness identifiers.

For faking messages on dynamic non-authentic channels, we cannot allow the intruder to use arbitrary keys. Otherwise, we would lose the possibility to generate fresh values in our model. Therefore, the chosen keys must correspond to session keys associated with existing runs (i.e., from set *rkeys U*).

abbreviation
  rkeys :: *fid-t set* ⇒ *key set* **where**
  $rkeys\ U \equiv sesK`(\lambda(x, y).\ x\ \$\ y)`(U \times (UNIV::nat\ set))$

lemma *rkeys-sesK* [*simp*, *dest*]: $sesK\ (R\$i) \in rkeys\ U \Longrightarrow R \in U$
⟨*proof*⟩

inductive-set
  fake :: *atom set* ⇒ *fid-t set* ⇒ *chmsg set* ⇒ *chmsg set*
  for *T* :: *atom set*
  and *U* :: *fid-t set*
  and *H* :: *chmsg set*
where
  fake-Inj:
    $M \in H \Longrightarrow M \in fake\ T\ U\ H$
| *fake-StatCh*:
    $[\![\ set\ M \subseteq extr\ T\ H;\ auth \notin c \vee A \in bad \vee B \in bad\ ]\!]$
  $\Longrightarrow StatCh\ c\ A\ B\ (Msg\ M) \in fake\ T\ U\ H$
| *fake-DynCh*:
    $[\![\ set\ M \subseteq extr\ T\ H;\ auth \notin c \wedge K \in rkeys\ U \vee aKey\ K \in extr\ T\ H\ ]\!]$
  $\Longrightarrow DynCh\ c\ K\ (Msg\ M) \in fake\ T\ U\ H$

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

lemmas *fake-intros* = *fake-StatCh fake-DynCh*

lemma *fake-expanding* [*intro*]: $H \subseteq fake\ T\ U\ H$
⟨*proof*⟩

lemma *fake-monotone* [*intro*]: $G \subseteq H \Longrightarrow fake\ T\ U\ G \subseteq fake\ T\ U\ H$
⟨*proof*⟩

lemma *fake-monotone-param1* [*intro*]:
  $T \subseteq T' \Longrightarrow fake\ T\ U\ H \subseteq fake\ T'\ U\ H$
⟨*proof*⟩

lemmas *fake-mono* [*elim*] = *fake-monotone* [*THEN* [*2*] *rev-subsetD*]
lemmas *fake-mono-param1* [*elim*] = *fake-monotone-param1* [*THEN* [*2*] *rev-subsetD*]

Atoms and extr together with fake

lemma *atoms-fake* [*simp*]: *atoms* (*fake* *T* *U* *H*) = *T* ∪ *atoms* *H*
⟨*proof*⟩

lemma *extr-fake* [*simp*]:
  assumes *T′* ⊆ *T* **shows** *extr* *T* (*fake* *T′* *U* *H*) = *extr* *T* *H*
⟨*proof*⟩

end

1.8 Theory of Agents and Messages for Security Protocols

theory *Message* **imports** *Keys* **begin**

lemma *Un-idem-collapse* [*simp*]: *A* ∪ (*B* ∪ *A*) = *B* ∪ *A*
⟨*proof*⟩

datatype
  msg = *Agent* *agent* — Agent names
    | *Number nat* — Ordinary integers, timestamps, ...
    | *Nonce* *nonce* — Unguessable nonces
    | *Key* *key* — Crypto keys
    | *Hash* *msg* — Hashing
    | *MPair* *msg msg* — Compound messages
    | *Crypt* *key msg* — Encryption, public- or shared-key

Concrete syntax: messages appear as ⦃*A*,*B*,*NA*⦄, etc...

syntax
  -MTuple :: [*′a*, *args*] => *′a* ∗ *′b* (‹(‹*indent=2 notation=*‹*mixfix message tuple*››⦃-,/ -⦄)›)
syntax-consts
  -MTuple == *MPair*
translations
  ⦃*x*, *y*, *z*⦄ == ⦃*x*, ⦃*y*, *z*⦄⦄
  ⦃*x*, *y*⦄ == *CONST MPair x y*

definition
  HPair :: [*msg*,*msg*] ⇒ *msg* (‹(*4Hash*[-] /-)› [*0*, *1000*])
where
  — Message Y paired with a MAC computed with the help of X
  Hash[*X*] *Y* ≡ ⦃*Hash*⦃*X*,*Y*⦄, *Y*⦄

definition

keysFor :: *msg set* ⇒ *key set*
where
— Keys useful to decrypt elements of a message set
keysFor H ≡ *invKey* ‘ {*K*. ∃ *X*. *Crypt K X* ∈ *H*}

Inductive Definition of All Parts" of a Message

inductive-set
parts :: *msg set* => *msg set*
for *H* :: *msg set*
where
Inj [*intro*]: *X* ∈ *H* ==> *X* ∈ *parts H*
| *Fst*: {|*X*, *Y*|} ∈ *parts H* ==> *X* ∈ *parts H*
| *Snd*: {|*X*, *Y*|} ∈ *parts H* ==> *Y* ∈ *parts H*
| *Body*: *Crypt K X* ∈ *parts H* ==> *X* ∈ *parts H*

Monotonicity

lemma *parts-mono*: *G* ⊆ *H* ==> *parts*(*G*) ⊆ *parts*(*H*)
⟨*proof*⟩

Equations hold because constructors are injective.

lemma *Other-image-eq* [*simp*]: (*Agent x* ∈ *Agent*‘*A*) = (*x*:*A*)
⟨*proof*⟩

lemma *Key-image-eq* [*simp*]: (*Key x* ∈ *Key*‘*A*) = (*x*∈*A*)
⟨*proof*⟩

lemma *Nonce-Key-image-eq* [*simp*]: (*Nonce x* ∉ *Key*‘*A*)
⟨*proof*⟩

1.8.1 keysFor operator

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

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

lemma *keysFor-UN* [*simp*]: *keysFor* (⋃ *i*∈*A*. *H i*) = (⋃ *i*∈*A*. *keysFor* (*H i*))
⟨*proof*⟩

Monotonicity

lemma *keysFor-mono*: *G* ⊆ *H* ==> *keysFor*(*G*) ⊆ *keysFor*(*H*)
⟨*proof*⟩

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

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

lemma *keysFor-insert-Number* [*simp*]: *keysFor* (*insert* (*Number N*) *H*) = *keysFor H*

⟨*proof*⟩

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

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

lemma *keysFor-insert-MPair* [*simp*]: *keysFor* (*insert* $\{|X, Y|\}$ *H*) = *keysFor H*
⟨*proof*⟩

lemma *keysFor-insert-Crypt* [*simp*]:
  keysFor (*insert* (*Crypt K X*) *H*) = *insert* (*invKey K*) (*keysFor H*)
⟨*proof*⟩

lemma *keysFor-image-Key* [*simp*]: *keysFor* (*Key‘E*) = {}
⟨*proof*⟩

lemma *Crypt-imp-invKey-keysFor*: *Crypt K X* ∈ *H* ==> *invKey K* ∈ *keysFor H*
⟨*proof*⟩

1.8.2 Inductive relation "parts"

lemma *MPair-parts*:
  [| $\{|X, Y|\}$ ∈ *parts H*;
    [| *X* ∈ *parts H*; *Y* ∈ *parts H* |] ==> *P* |] ==> *P*
⟨*proof*⟩

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

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

lemma *parts-increasing*: *H* ⊆ *parts*(*H*)
⟨*proof*⟩

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

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

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

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

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

Unions

lemma *parts-Un-subset1*: *parts*(*G*) ∪ *parts*(*H*) ⊆ *parts*(*G* ∪ *H*)
⟨*proof*⟩

lemma *parts-Un-subset2*: *parts*$(G \cup H) \subseteq$ *parts*$(G) \cup$ *parts*(H)
⟨*proof*⟩

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

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

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

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

Added to simplify arguments to parts, analz and synth.

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

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

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

Idempotence and transitivity

lemma *parts-partsD* [*dest*!]: $X \in$ *parts* (*parts* H) ==> $X \in$ *parts* H
⟨*proof*⟩

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

lemma *parts-subset-iff* [*simp*]: (*parts* $G \subseteq$ *parts* H) = ($G \subseteq$ *parts* H)
⟨*proof*⟩

lemma *parts-trans*: [| $X \in$ *parts* G; $G \subseteq$ *parts* H |] ==> $X \in$ *parts* H
⟨*proof*⟩

Cut

lemma *parts-cut*:
  [| $Y \in$ *parts* (*insert* X G); $X \in$ *parts* H |] ==> $Y \in$ *parts* $(G \cup H)$
⟨*proof*⟩

lemma *parts-cut-eq* [*simp*]: $X \in$ *parts* H ==> *parts* (*insert* X H) = *parts* H
⟨*proof*⟩

Rewrite rules for pulling out atomic messages

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

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

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

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

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

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

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

lemma *parts-insert-MPair* [*simp*]:
$parts\ (insert\ \{\!|X,Y|\!\}\ H) =$
$insert\ \{\!|X,Y|\!\}\ (parts\ (insert\ X\ (insert\ Y\ H)))$
⟨*proof*⟩

lemma *parts-image-Key* [*simp*]: $parts\ (Key`N) = Key`N$
⟨*proof*⟩

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

1.8.3 Inductive relation "analz"

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

inductive-set
analz :: *msg set* => *msg set*
for *H* :: *msg set*
where
Inj [*intro*,*simp*] : $X \in H$ ==> $X \in analz\ H$
| *Fst*: $\{\!|X,Y|\!\} \in analz\ H$ ==> $X \in analz\ H$
| *Snd*: $\{\!|X,Y|\!\} \in analz\ H$ ==> $Y \in analz\ H$
| *Decrypt* [*dest*]:
[|*Crypt K X* ∈ *analz H*; *Key*(*invKey K*): *analz H*|] ==> *X* ∈ *analz H*

Monotonicity; Lemma 1 of Lowe's paper

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

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

Making it safe speeds up proofs

lemma *MPair-analz* [*elim*!]:
  [| $\{\!|X, Y|\!\} \in analz\ H$;
    [| $X \in analz\ H$; $Y \in analz\ H$ |] ==> P
  |] ==> P
⟨*proof*⟩

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

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

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

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

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

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

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

General equational properties

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

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

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

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

Rewrite rules for pulling out atomic messages

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

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

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

analz (*insert* (*Nonce N*) *H*) = *insert* (*Nonce N*) (*analz H*)
⟨*proof*⟩

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

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

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

lemma *analz-insert-Key* [*simp*]:
K ∉ *keysFor* (*analz H*) ==>
analz (*insert* (*Key K*) *H*) = *insert* (*Key K*) (*analz H*)
⟨*proof*⟩

lemma *analz-insert-MPair* [*simp*]:
analz (*insert* ⦃*X*, *Y*⦄ *H*) =
insert ⦃*X*, *Y*⦄ (*analz* (*insert X* (*insert Y H*)))
⟨*proof*⟩

Can pull out enCrypted message if the Key is not known

lemma *analz-insert-Crypt*:
Key (*invKey K*) ∉ *analz H*
==> *analz* (*insert* (*Crypt K X*) *H*) = *insert* (*Crypt K X*) (*analz H*)
⟨*proof*⟩

lemma *lemma1*: *Key* (*invKey K*) ∈ *analz H* ==>
analz (*insert* (*Crypt K X*) *H*) ⊆
insert (*Crypt K X*) (*analz* (*insert X H*))
⟨*proof*⟩

lemma *lemma2*: *Key* (*invKey K*) ∈ *analz H* ==>
insert (*Crypt K X*) (*analz* (*insert X H*)) ⊆
analz (*insert* (*Crypt K X*) *H*)
⟨*proof*⟩

lemma *analz-insert-Decrypt*:
Key (*invKey K*) ∈ *analz H* ==>
analz (*insert* (*Crypt K X*) *H*) =
insert (*Crypt K X*) (*analz* (*insert X H*))
⟨*proof*⟩

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

lemma *analz-Crypt-if* [*simp*]:
analz (*insert* (*Crypt K X*) *H*) =
(*if* (*Key* (*invKey K*) ∈ *analz H*)
then insert (*Crypt K X*) (*analz* (*insert X H*))
else insert (*Crypt K X*) (*analz H*))

⟨*proof*⟩

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

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

lemma *analz-image-Key* [*simp*]: $analz\ (Key`N) = Key`N$
⟨*proof*⟩

Idempotence and transitivity

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

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

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

lemma *analz-trans*: $[|\ X \in analz\ G;\ \ G \subseteq analz\ H\ |] ==> X \in analz\ H$
⟨*proof*⟩

Cut; Lemma 2 of Lowe

lemma *analz-cut*: $[|\ Y \in analz\ (insert\ X\ H);\ \ X \in analz\ H\ |] ==> Y \in analz\ H$
⟨*proof*⟩

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

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

A congruence rule for "analz"

lemma *analz-subset-cong*:
$[|\ analz\ G \subseteq analz\ G';\ analz\ H \subseteq analz\ H'\ |]$
$==> analz\ (G \cup H) \subseteq analz\ (G' \cup H')$
⟨*proof*⟩

lemma *analz-cong*:
$[|\ analz\ G = analz\ G';\ analz\ H = analz\ H'\ |]$
$==> analz\ (G \cup H) = analz\ (G' \cup H')$
⟨*proof*⟩

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

If there are no pairs or encryptions then analz does nothing

lemma *analz-trivial*:

$[| \forall X\ Y.\ \{|X,Y|\} \notin H;\ \forall X\ K.\ Crypt\ K\ X \notin H\ |] ==> analz\ H = H$
⟨*proof*⟩

1.8.4 Inductive relation "synth"

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

inductive-set
synth :: *msg set* => *msg set*
for *H* :: *msg set*
where
Inj [*intro*]: $X \in H ==> X \in synth\ H$
| *Agent* [*intro*]: $Agent\ agt \in synth\ H$
| *Number* [*intro*]: $Number\ n \in synth\ H$
| *Hash* [*intro*]: $X \in synth\ H ==> Hash\ X \in synth\ H$
| *MPair* [*intro*]: $[|X \in synth\ H;\ Y \in synth\ H|] ==> \{|X,Y|\} \in synth\ H$
| *Crypt* [*intro*]: $[|X \in synth\ H;\ Key(K) \in H|] ==> Crypt\ K\ X \in synth\ H$

Monotonicity

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

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

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

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

Unions

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

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

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

Idempotence and transitivity

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

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

lemma *synth-subset-iff* [*simp*]: (*synth* $G \subseteq$ *synth* H) = ($G \subseteq$ *synth* H)
⟨*proof*⟩

lemma *synth-trans*: [| $X \in$ *synth* G; $G \subseteq$ *synth* H |] ==> $X \in$ *synth* H
⟨*proof*⟩

Cut; Lemma 2 of Lowe

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

lemma *Agent-synth* [*simp*]: *Agent* $A \in$ *synth* H
⟨*proof*⟩

lemma *Number-synth* [*simp*]: *Number* $n \in$ *synth* H
⟨*proof*⟩

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

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

lemma *Crypt-synth-eq* [*simp*]:
Key $K \notin H$ ==> (*Crypt* K $X \in$ *synth* H) = (*Crypt* K $X \in H$)
⟨*proof*⟩

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

Combinations of parts, analz and synth

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

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

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

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

chsp: added

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

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

For reasoning about the Fake rule in traces

lemma *parts-insert-subset-Un*: $X \in G ==> parts(insert\ X\ H) \subseteq parts\ G \cup parts\ H$
⟨*proof*⟩

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

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

lemma *Fake-parts-insert-in-Un*:
$[|Z \in parts\ (insert\ X\ H);\ X \in synth\ (analz\ H)|]$
$==> Z \in synth\ (analz\ H) \cup parts\ H$
⟨*proof*⟩

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

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

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

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

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

lemma *MPair-synth-analz* [*iff*]:
$(\{\!|X, Y|\!\} \in synth\ (analz\ H)) =$
$(X \in synth\ (analz\ H)\ \&\ Y \in synth\ (analz\ H))$
⟨*proof*⟩

lemma *Crypt-synth-analz*:
$[|\ Key\ K \in analz\ H;\ Key\ (invKey\ K) \in analz\ H\ |]$
$==> (Crypt\ K\ X \in synth\ (analz\ H)) = (X \in synth\ (analz\ H))$
⟨*proof*⟩

lemma *Hash-synth-analz* [*simp*]:
$X \notin synth\ (analz\ H)$
$==> (Hash\{\!|X, Y|\!\} \in synth\ (analz\ H)) = (Hash\{\!|X, Y|\!\} \in analz\ H)$
⟨*proof*⟩

1.8.5 HPair: a combination of Hash and MPair

Freeness

lemma *Agent-neq-HPair*: $Agent\ A\ \sim= Hash[X]\ Y$

⟨*proof*⟩

lemma *Nonce-neq-HPair*: *Nonce N* ~= *Hash*[*X*] *Y*
⟨*proof*⟩

lemma *Number-neq-HPair*: *Number N* ~= *Hash*[*X*] *Y*
⟨*proof*⟩

lemma *Key-neq-HPair*: *Key K* ~= *Hash*[*X*] *Y*
⟨*proof*⟩

lemma *Hash-neq-HPair*: *Hash Z* ~= *Hash*[*X*] *Y*
⟨*proof*⟩

lemma *Crypt-neq-HPair*: *Crypt K X'* ~= *Hash*[*X*] *Y*
⟨*proof*⟩

lemmas *HPair-neqs* = *Agent-neq-HPair Nonce-neq-HPair Number-neq-HPair*
Key-neq-HPair Hash-neq-HPair Crypt-neq-HPair

declare *HPair-neqs* [*iff*]
declare *HPair-neqs* [*symmetric*, *iff*]

lemma *HPair-eq* [*iff*]: (*Hash*[*X'*] *Y'* = *Hash*[*X*] *Y*) = (*X'* = *X* & *Y'*=*Y*)
⟨*proof*⟩

lemma *MPair-eq-HPair* [*iff*]:
({|*X'*, *Y'*|} = *Hash*[*X*] *Y*) = (*X'* = *Hash*{|*X*, *Y*|} & *Y'*=*Y*)
⟨*proof*⟩

lemma *HPair-eq-MPair* [*iff*]:
(*Hash*[*X*] *Y* = {|*X'*, *Y'*|}) = (*X'* = *Hash*{|*X*, *Y*|} & *Y'*=*Y*)
⟨*proof*⟩

Specialized laws, proved in terms of those for Hash and MPair

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

lemma *parts-insert-HPair* [*simp*]:
parts (*insert* (*Hash*[*X*] *Y*) *H*) =
insert (*Hash*[*X*] *Y*) (*insert* (*Hash*{|*X*, *Y*|}) (*parts* (*insert Y H*)))
⟨*proof*⟩

lemma *analz-insert-HPair* [*simp*]:
analz (*insert* (*Hash*[*X*] *Y*) *H*) =
insert (*Hash*[*X*] *Y*) (*insert* (*Hash*{|*X*, *Y*|}) (*analz* (*insert Y H*)))
⟨*proof*⟩

lemma *HPair-synth-analz* [*simp*]:
X ∉ *synth* (*analz H*)
==> (*Hash*[*X*] *Y* ∈ *synth* (*analz H*)) =
(*Hash*{|*X*, *Y*|} ∈ *analz H* & *Y* ∈ *synth* (*analz H*))

⟨*proof*⟩

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

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

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

lemmas *pushKeys* =
  insert-commute [*of Key K Agent C* **for** *K C*]
  insert-commute [*of Key K Nonce N* **for** *K N*]
  insert-commute [*of Key K Number N* **for** *K N*]
  insert-commute [*of Key K Hash X* **for** *K X*]
  insert-commute [*of Key K MPair X Y* **for** *K X Y*]
  insert-commute [*of Key K Crypt X K′* **for** *K K′ X*]

lemmas *pushCrypts* =
  insert-commute [*of Crypt X K Agent C* **for** *X K C*]
  insert-commute [*of Crypt X K Agent C* **for** *X K C*]
  insert-commute [*of Crypt X K Nonce N* **for** *X K N*]
  insert-commute [*of Crypt X K Number N* **for** *X K N*]
  insert-commute [*of Crypt X K Hash X′* **for** *X K X′*]
  insert-commute [*of Crypt X K MPair X′ Y* **for** *X K X′ Y*]

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

lemmas *pushes* = *pushKeys pushCrypts*

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

declare *o-def* [*simp*]

lemma *Crypt-notin-image-Key* [*simp*]: *Crypt K X* ∉ *Key* ‘ *A*
⟨*proof*⟩

lemma *Hash-notin-image-Key* [*simp*] :*Hash X* ∉ *Key* ‘ *A*
⟨*proof*⟩

lemma *synth-analz-mono*: *G*⊆*H* ==> *synth* (*analz*(*G*)) ⊆ *synth* (*analz*(*H*))
⟨*proof*⟩

lemma *Fake-analz-eq* [*simp*]:
    X ∈ *synth*(*analz H*) ==> *synth* (*analz* (*insert X H*)) = *synth* (*analz H*)
⟨*proof*⟩

Two generalizations of *analz-insert-eq*

lemma *gen-analz-insert-eq* [*rule-format*]:
    X ∈ *analz H* ==> *ALL G*. *H* ⊆ *G* −−> *analz* (*insert X G*) = *analz G*
⟨*proof*⟩

lemma *synth-analz-insert-eq* [*rule-format*]:
    X ∈ *synth* (*analz H*)

==> *ALL G. H* ⊆ *G* --> (*Key K* ∈ *analz* (*insert X G*)) = (*Key K* ∈ *analz G*)
⟨*proof*⟩

lemma *Fake-parts-sing*:
X ∈ *synth* (*analz H*) ==> *parts*{*X*} ⊆ *synth* (*analz H*) ∪ *parts H*
⟨*proof*⟩

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

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

declare *invKey-K* [*simp*]

end

1.9 Secrecy with Leaking (global version)

theory *s0g-secrecy* **imports** *Refinement Agents*
begin

This model extends the global secrecy model by adding a *leak* event, which models that the adversary can learn messages through leaks of some (unspecified) kind.

Proof tool configuration. Avoid annoying automatic unfolding of *dom*.

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

1.9.1 State

The only state variable is a knowledge relation, an authorization relation, and a leakage relation.

(*d*, *A*) ∈ *kn s* means that the agent *A* knows data *d*. (*d*, *A*) ∈ *az s* means that the agent *A* is authorized to know data *d*. (*d*, *A*) ∈ *lk s* means that data *d* has leaked to agent *A*. Leakage models potential unauthorized knowledge.

record *'d s0g-state* =
kn :: (*'d* × *agent*) *set*
az :: (*'d* × *agent*) *set*
lk :: *'d set* — leaked data

type-synonym
'd s0g-obs = *'d s0g-state*

abbreviation
lkr s ≡ *lk s* × *UNIV*

1.9.2 Invariant definitions

Global secrecy is stated as an invariant.

definition
s0g-secrecy :: *'d s0g-state set*
where

$s0g\text{-}secrecy \equiv \{s.\ kn\ s \subseteq az\ s \cup lkr\ s\}$

lemmas *s0g-secrecyI* = *s0g-secrecy-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *s0g-secrecyE* [*elim*] =
s0g-secrecy-def [*THEN setc-def-to-elim*, *rule-format*]

Data that someone is authorized to know and leaked data is known by someone.

definition
s0g-dom :: *'d s0g-state set*
where
$s0g\text{-}dom \equiv \{s.\ Domain\ (az\ s \cup lkr\ s) \subseteq Domain\ (kn\ s)\}$

lemmas *s0g-domI* = *s0g-dom-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *s0g-domE* [*elim*] = *s0g-dom-def* [*THEN setc-def-to-elim*, *rule-format*]

1.9.3 Events

New secrets may be generated anytime.

definition
$s0g\text{-}gen :: ['d,\ agent,\ agent\ set] \Rightarrow ('d\ s0g\text{-}state \times 'd\ s0g\text{-}state)\ set$
where
$s0g\text{-}gen\ d\ A\ G \equiv \{(s,\ s1).$
— guards:
$A \in G \wedge$
$d \notin Domain\ (kn\ s) \wedge$ — fresh item

— actions:
$s1 = s(\!|$
$kn := insert\ (d,\ A)\ (kn\ s),$
$az := az\ s \cup \{d\} \times (if\ G \cap bad = \{\}\ then\ G\ else\ UNIV)$
$|\!)$
$\}$

Learning secrets.

definition
s0g-learn ::
$['d,\ agent] \Rightarrow ('d\ s0g\text{-}state \times 'd\ s0g\text{-}state)\ set$
where
$s0g\text{-}learn\ d\ B \equiv \{(s,\ s1).$
— guards:
— $d \in Domain\ (kn\ s) \wedge$ someone knows d (follows from authorization)

— check authorization or leakage to preserve secrecy
$(d,\ B) \in az\ s \cup lkr\ s \wedge$

— actions:
$s1 = s(\!|\ kn := insert\ (d,\ B)\ (kn\ s)\ |\!)$
$\}$

Leaking secrets.

definition

$s0g\text{-}leak$ ::
$'d \Rightarrow ('d\ s0g\text{-}state \times 'd\ s0g\text{-}state)\ set$
where
$s0g\text{-}leak\ d \equiv \{(s,\ s1).$
— guards:
$d \in Domain\ (kn\ s) \wedge$ — someone knows d

— actions:
$s1 = s(\!|\ lk := insert\ d\ (lk\ s)\ |\!)$
}

1.9.4 Specification

definition
$s0g\text{-}init$ :: $'d\ s0g\text{-}state\ set$
where
$s0g\text{-}init \equiv s0g\text{-}secrecy \cap s0g\text{-}dom$ — any state satisfying invariants

definition
$s0g\text{-}trans$ :: $('d\ s0g\text{-}state \times 'd\ s0g\text{-}state)\ set$ **where**
$s0g\text{-}trans \equiv (\bigcup d\ A\ B\ G.$
$s0g\text{-}gen\ d\ A\ G \cup$
$s0g\text{-}learn\ d\ B \cup$
$s0g\text{-}leak\ d \cup$
Id
)

definition
$s0g$ :: $('d\ s0g\text{-}state,\ 'd\ s0g\text{-}obs)\ spec$ **where**
$s0g \equiv (\!|$
$init = s0g\text{-}init,$
$trans = s0g\text{-}trans,$
$obs = id$
$|\!)$

lemmas *s0g-defs* =
s0g-def s0g-init-def s0g-trans-def
s0g-gen-def s0g-learn-def s0g-leak-def

lemma *s0g-obs-id* [*simp*]: *obs s0g* = *id*
⟨*proof*⟩

All state predicates are trivially observable.

lemma *s0g-anyP-observable* [*iff*]: *observable* (*obs s0g*) *P*
⟨*proof*⟩

1.9.5 Invariant proofs

1.9.6 inv1: Secrecy

lemma *PO-s0g-secrecy-init* [*iff*]:
$init\ s0g \subseteq s0g\text{-}secrecy$
⟨*proof*⟩

lemma *PO-s0g-secrecy-trans* [*iff*]:
{*s0g-secrecy*} *trans s0g* {> *s0g-secrecy*}
⟨*proof*⟩

lemma *PO-s0g-secrecy* [*iff*]:*reach s0g* ⊆ *s0g-secrecy*
⟨*proof*⟩

As en external invariant.

lemma *PO-s0g-obs-secrecy* [*iff*]:*oreach s0g* ⊆ *s0g-secrecy*
⟨*proof*⟩

1.9.7 inv2: Authorized and leaked data is known to someone

lemma *PO-s0g-dom-init* [*iff*]:
init s0g ⊆ *s0g-dom*
⟨*proof*⟩

lemma *PO-s0g-dom-trans* [*iff*]:
{*s0g-dom*} *trans s0g* {> *s0g-dom*}
⟨*proof*⟩

lemma *PO-s0g-dom* [*iff*]: *reach s0g* ⊆ *s0g-dom*
⟨*proof*⟩

As en external invariant.

lemma *PO-s0g-obs-dom* [*iff*]: *oreach s0g* ⊆ *s0g-dom*
⟨*proof*⟩

end

1.10 Non-injective Agreement

theory *a0n-agree* **imports** *Refinement Agents*
begin

The initial model abstractly specifies entity authentication, where one agent/role authenticates another. More precisely, this property corresponds to non-injective agreement on a data set *ds*. We use Running and Commit signals to obtain a protocol-independent extensional specification.

Proof tool configuration. Avoid annoying automatic unfolding of *dom*.

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

1.10.1 State

Signals. At this stage there are no protocol runs yet. All we model are the signals that indicate a certain progress of a protocol run by one agent/role (Commit signal) and the other role (Running signal). The signals contain a list of agents that are assumed to be honest and a polymorphic data set to be agreed upon, which is instantiated later.

Usually, the agent list will contain the names of the two agents who want to agree on the data, but sometimes one of the agents is honest by assumption (e.g., the server) or the honesty of additional agents needs to be assumed for the agreement to hold.

datatype *'ds signal* =
  Running agent list 'ds
| *Commit agent list 'ds*

record *'ds a0n-state* =
  signals :: *'ds signal* ⇒ *nat* — multi-set of signals
  corrupted :: *'ds set* — set of corrupted data

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

1.10.2 Events

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

Running signal, indicating end of responder run.

definition
  a0n-running :: [*agent list*, *'ds*] ⇒ (*'ds a0n-state* × *'ds a0n-state*) *set*
where
  a0n-running h d ≡ {(*s*, *s'*).
    — actions:
    s' = *s*(|
      signals := (*signals s*)(*Running h d* := *signals s* (*Running h d*) + *1*)
    |)
  }

Commit signal, marking end of initiator run.

definition
  a0n-commit :: [*agent list*, *'ds*] ⇒ (*'ds a0n-state* × *'ds a0n-state*) *set*
where
  a0n-commit h d ≡ {(*s*, *s'*).
    — guards:
    (*set h* ⊆ *good* ⟶ *d* ∉ *corrupted s* ⟶ *signals s* (*Running h d*) > *0*) ∧

    — actions:
    s' = *s*(|
      signals := (*signals s*)(*Commit h d* := *signals s* (*Commit h d*) + *1*)
    |)
  }

Data corruption.

definition

$a0n\text{-}corrupt :: 'ds\ set \Rightarrow ('ds\ a0n\text{-}state \times 'ds\ a0n\text{-}state)\ set$
where
$a0n\text{-}corrupt\ ds \equiv \{(s, s').$
— actions:
$s' = s(\!|$
$corrupted := corrupted\ s \cup ds$
$|\!)$
$\}$

Transition system.

definition
$a0n\text{-}trans :: ('ds\ a0n\text{-}state \times 'ds\ a0n\text{-}state)\ set$ **where**
$a0n\text{-}trans \equiv (\bigcup h\ d\ ds.$
$a0n\text{-}running\ h\ d \cup$
$a0n\text{-}commit\ h\ d \cup$
$a0n\text{-}corrupt\ ds \cup$
Id
$)$

definition
$a0n :: ('ds\ a0n\text{-}state, 'ds\ a0n\text{-}obs)\ spec$ **where**
$a0n \equiv (\!|$
$init = a0n\text{-}init,$
$trans = a0n\text{-}trans,$
$obs = id$
$|\!)$

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

Any property is trivially observable.

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

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

1.10.3 Invariants

1.10.4 inv1: non-injective agreement

This is an extensional variant of Lowe's *non-injective agreement* of the first with the second agent (by convention) in h on data d [Lowe97].

definition
$a0n\text{-}inv1\text{-}niagree :: 'ds\ a0n\text{-}state\ set$
where
$a0n\text{-}inv1\text{-}niagree \equiv \{s.\ \forall h\ d.$
$set\ h \subseteq good \longrightarrow d \notin corrupted\ s \longrightarrow$
$signals\ s\ (Commit\ h\ d) > 0 \longrightarrow signals\ s\ (Running\ h\ d) > 0$
$\}$

lemmas *a0n-inv1-niagreeI* =
a0n-inv1-niagree-def [*THEN setc-def-to-intro*, *rule-format*]
lemmas *a0n-inv1-niagreeE* [*elim*] =
a0n-inv1-niagree-def [*THEN setc-def-to-elim*, *rule-format*]
lemmas *a0n-inv1-niagreeD* =
a0n-inv1-niagree-def [*THEN setc-def-to-dest*, *rule-format*, *rotated 2*]

Invariance proof.

lemma *PO-a0n-inv1-niagree-init* [*iff*]:
init a0n ⊆ *a0n-inv1-niagree*
⟨*proof*⟩

lemma *PO-a0n-inv1-niagree-trans* [*iff*]:
{*a0n-inv1-niagree*} *trans a0n* {> *a0n-inv1-niagree*}
⟨*proof*⟩

lemma *PO-a0n-inv1-niagree* [*iff*]: *reach a0n* ⊆ *a0n-inv1-niagree*
⟨*proof*⟩

This is also an external invariant.

lemma *a0n-obs-inv1-niagree* [*iff*]:
oreach a0n ⊆ *a0n-inv1-niagree*
⟨*proof*⟩

end

1.11 Injective Agreement

theory *a0i-agree* **imports** *a0n-agree*
begin

This refinement adds injectiveness to the agreement property.

1.11.1 State

The state and observations are the same as in the previous model.

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

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

1.11.2 Events

We just refine the commit event. Everything else remains the same.

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

abbreviation
a0i-running :: [*agent list*, *'ds*] ⇒ (*'ds a0i-state* × *'ds a0i-state*) *set*
where
a0i-running ≡ *a0n-running*

definition
a0i-commit ::
[*agent list*, *'ds*] ⇒ (*'ds a0i-state* × *'ds a0i-state*) *set*
where
a0i-commit h d ≡ {(*s*, *s'*).
— guards:
(*set h* ⊆ *good* ⟶ *d* ∉ *corrupted s* ⟶
signals s (*Commit h d*) < *signals s* (*Running h d*)) ∧

— actions:
s' = *s*⦇
signals := (*signals s*)(*Commit h d* := *signals s* (*Commit h d*) + *1*)
⦈
}

abbreviation
a0i-corrupt :: *'ds set* ⇒ (*'ds a0i-state* × *'ds a0i-state*) *set*
where
a0i-corrupt ≡ *a0n-corrupt*

Transition system.

definition
a0i-trans :: (*'ds a0i-state* × *'ds a0i-state*) *set* **where**
a0i-trans ≡ (⋃ *h d ds*.
a0i-running h d ∪
a0i-commit h d ∪
a0i-corrupt ds ∪
Id
)

definition
a0i :: (*'ds a0i-state*, *'ds 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

Any property is trivially observable.

lemma *a0i-obs* [*simp*]: *obs a0i* = *id*
⟨*proof*⟩

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

1.11.3 Invariants

Injective agreement.

definition
a0i-inv1-iagree :: *'ds a0i-state set*
where
a0i-inv1-iagree ≡ {*s*. ∀ *h d*.
set h ⊆ *good* ⟶ *d* ∉ *corrupted s* ⟶
signals s (*Commit h d*) ≤ *signals s* (*Running h d*)
}

lemmas *a0i-inv1-iagreeI* =
a0i-inv1-iagree-def [*THEN setc-def-to-intro*, *rule-format*]
lemmas *a0i-inv1-iagreeE* [*elim*] =
a0i-inv1-iagree-def [*THEN setc-def-to-elim*, *rule-format*]
lemmas *a0i-inv1-iagreeD* =
a0i-inv1-iagree-def [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

lemma *PO-a0i-inv1-iagree-init* [*iff*]:
init a0i ⊆ *a0i-inv1-iagree*
⟨*proof*⟩

lemma *PO-a0i-inv1-iagree-trans* [*iff*]:
{*a0i-inv1-iagree*} *trans a0i* {> *a0i-inv1-iagree*}
⟨*proof*⟩

lemma *PO-a0i-inv1-iagree* [*iff*]: *reach a0i* ⊆ *a0i-inv1-iagree*
⟨*proof*⟩

As an external invariant.

lemma *PO-a0i-obs-inv1-iagree* [*iff*]: *oreach a0i* ⊆ *a0i-inv1-iagree*
⟨*proof*⟩

1.11.4 Refinement

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

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

lemma *PO-a0i-running-refines-a0n-running*:
{*R0n0i*}
(*a0n-running h d*), (*a0i-running h d*)
{> *R0n0i*}
⟨*proof*⟩

lemma *PO-a0i-commit-refines-a0n-commit*:

{*R0n0i*}
  (*a0n-commit h d*), (*a0i-commit h d*)
 {> *R0n0i*}
⟨*proof*⟩

lemma *PO-a0i-corrupt-refines-a0n-corrupt*:
{*R0n0i*}
  (*a0n-corrupt d*), (*a0i-corrupt d*)
 {> *R0n0i*}
⟨*proof*⟩

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

All together now...

lemma *PO-m1-refines-init-a0n* [*iff*]:
init a0i ⊆ *R0n0i*``(*init a0n*)
⟨*proof*⟩

lemma *PO-m1-refines-trans-a0n* [*iff*]:
{*R0n0i*}
  (*trans a0n*), (*trans a0i*)
 {> *R0n0i*}
⟨*proof*⟩

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

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

1.11.5 Derived invariants

lemma *iagree-implies-niagree* [*iff*]: *a0i-inv1-iagree* ⊆ *a0n-inv1-niagree*
⟨*proof*⟩

Non-injective agreeement as internal and external invariants.

lemma *PO-a0i-a0n-inv1-niagree* [*iff*]: *reach a0i* ⊆ *a0n-inv1-niagree*
⟨*proof*⟩

lemma *PO-a0i-obs-a0n-inv1-niagree* [*iff*]: *oreach a0i* ⊆ *a0n-inv1-niagree*
⟨*proof*⟩

end

Chapter 2

Unidirectional Authentication Protocols

In this chapter, we derive some simple unilateral authentication protocols. We have a single abstract model at Level 1. We then refine this model into two channel protocols (Level 2), one using authentic channels and one using confidential channels. We then refine these in turn into cryptographic protocols (Level 3) respectively using signatures and public-key encryption.

2.1 Refinement 1: Abstract Protocol

theory *m1-auth* **imports** *../Refinement/Runs ../Refinement/a0i-agree*
begin

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

2.1.1 State

We introduce protocol runs.

record *m1-state* =
  runs :: *runs-t*

type-synonym
  m1-obs = *m1-state*

definition
  m1-init :: *m1-state set* **where**
  m1-init ≡ { (|
    runs = *Map.empty*
  |) }

2.1.2 Events

definition — refines *skip*
  m1-step1 :: [*rid-t*, *agent*, *agent*, *nonce*] ⇒ (*m1-state* × *m1-state*) *set*
where

m1-step1 Ra A B Na ≡ {(*s*, *s1*).

— guards
Ra ∉ *dom* (*runs s*) ∧ — new initiator run
Na = *Ra*$*0* ∧ — generated nonce

— actions
s1 = *s*⦇
runs := (*runs s*)(
Ra ↦ (*Init*, [*A*, *B*], [])
)
⦈
}

definition — refines *a0i-running*
m1-step2 :: [*rid-t*, *agent*, *agent*, *nonce*, *nonce*] ⇒ (*m1-state* × *m1-state*) *set*
where
m1-step2 Rb A B Na Nb ≡ {(*s*, *s1*). — *Ni* is completely arbitrary

— guards
Rb ∉ *dom* (*runs s*) ∧ — new responder run
Nb = *Rb*$*0* ∧ — generated nonce

— actions
s1 = *s*⦇
runs := (*runs s*)(*Rb* ↦ (*Resp*, [*A*, *B*], [*aNon Na*]))
⦈
}

definition — refines *a0i-commit*
m1-step3 ::
[*rid-t*, *agent*, *agent*, *nonce*, *nonce*] ⇒ (*m1-state* × *m1-state*) *set*
where
m1-step3 Ra A B Na Nb ≡ {(*s*, *s1*).

— guards
runs s Ra = *Some* (*Init*, [*A*, *B*], []) ∧
Na = *Ra*$*0* ∧

— authentication guard:
(*A* ∉ *bad* ∧ *B* ∉ *bad* ⟶ (∃ *Rb*.
Nb = *Rb*$*0* ∧ *runs s Rb* = *Some* (*Resp*, [*A*, *B*], [*aNon Na*]))) ∧

— actions
s1 = *s*⦇
runs := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [*aNon Nb*]))
⦈
}

Transition system.

definition
m1-trans :: (*m1-state* × *m1-state*) *set* **where**
m1-trans ≡ (⋃ *A B Ra Rb Na Nb*.

m1-step1 Ra A B Na ∪
m1-step2 Rb A B Na Nb ∪
m1-step3 Ra A B Na Nb ∪
Id
)

definition
m1 :: (*m1-state*, *m1-obs*) *spec* **where**
m1 ≡ (|
init = *m1-init*,
trans = *m1-trans*,
obs = *id*
|)

lemmas *m1-defs* =
m1-def m1-init-def m1-trans-def
m1-step1-def m1-step2-def m1-step3-def

2.1.3 Simulation relation

We define two auxiliary functions to reconstruct the signals of the initial model from completed initiator and responder runs of the current one.

type-synonym
irsig = *nonce* × *nonce*

fun
runs2sigs :: *runs-t* ⇒ *irsig signal* ⇒ *nat*
where
runs2sigs runz (*Commit* [*A*, *B*] (*Ra$0*, *Nb*)) =
(*if runz Ra* = *Some* (*Init*, [*A*, *B*], [*aNon Nb*]) *then 1 else 0*)

| *runs2sigs runz* (*Running* [*A*, *B*] (*Na*, *Rb$0*)) =
(*if runz Rb* = *Some* (*Resp*, [*A*, *B*], [*aNon Na*]) *then 1 else 0*)

| *runs2sigs runz* - = *0*

Simulation relation and mediator function. We map completed initiator and responder runs to commit and running signals, respectively.

definition
med10 :: *m1-obs* ⇒ *irsig a0i-obs* **where**
med10 o1 ≡ (| *signals* = *runs2sigs* (*runs o1*), *corrupted* = {} |)

definition
R01 :: (*irsig a0i-state* × *m1-state*) *set* **where**
R01 ≡ {(*s*, *t*). *signals s* = *runs2sigs* (*runs t*) ∧ *corrupted s* = {} }

lemmas *R01-defs* = *R01-def med10-def*

Lemmas about the auxiliary functions

Basic lemmas

lemma *runs2sigs-empty* [*simp*]:
  $runz = Map.empty \Longrightarrow runs2sigs\ runz = (\lambda x.\ 0)$
⟨*proof*⟩

Update lemmas

lemma *runs2sigs-upd-init-none* [*simp*]:
  $[\![\ Ra \notin dom\ runz\ ]\!]$
  $\Longrightarrow runs2sigs\ (runz(Ra \mapsto (Init,\ [A,\ B],\ []))) = runs2sigs\ runz$
⟨*proof*⟩

lemma *runs2sigs-upd-init-some* [*simp*]:
  $[\![\ runz\ Ra = Some\ (Init,\ [A,\ B],\ [])\ ]\!]$
  $\Longrightarrow runs2sigs\ (runz(Ra \mapsto (Init,\ [A,\ B],\ [aNon\ Nb]))) =$
    $(runs2sigs\ runz)(Commit\ [A,\ B]\ (Ra\$0,\ Nb) := 1)$
⟨*proof*⟩

lemma *runs2sigs-upd-resp* [*simp*]:
  $[\![\ Rb \notin dom\ runz\ ]\!]$
  $\Longrightarrow runs2sigs\ (runz(Rb \mapsto (Resp,\ [A,\ B],\ [aNon\ Na]))) =$
    $(runs2sigs\ runz)(Running\ [A,\ B]\ (Na,\ Rb\$0) := 1)$
⟨*proof*⟩

2.1.4 Refinement

lemma *PO-m1-step1-refines-skip*:
  {*R01*}
    Id, (*m1-step1 Ra A B Na*)
  {> *R01*}
⟨*proof*⟩

lemma *PO-m1-step2-refines-a0i-running*:
  {*R01*}
    (*a0i-running* [*A*, *B*] (*Na*, *Nb*)), (*m1-step2 Rb A B Na Nb*)
  {> *R01*}
⟨*proof*⟩

lemma *PO-m1-step3-refines-a0i-commit*:
  {*R01*}
    (*a0i-commit* [*A*, *B*] (*Na*, *Nb*)), (*m1-step3 Ra A B Na Nb*)
  {> *R01*}
⟨*proof*⟩

lemmas *PO-m1-trans-refines-a0i-trans* =
  PO-m1-step1-refines-skip PO-m1-step2-refines-a0i-running
  PO-m1-step3-refines-a0i-commit

All together now...

lemma *PO-m1-refines-init-a0i* [*iff*]:
  $init\ m1 \subseteq R01``(init\ a0i)$
⟨*proof*⟩

lemma *PO-m1-refines-trans-a0i* [*iff*]:
  {*R01*}

(*trans a0i*), (*trans m1*)
{> *R01*}
⟨*proof*⟩

lemma *PO-obs-consistent* [*iff*]:
obs-consistent R01 med10 a0i m1
⟨*proof*⟩

lemma *PO-m1-refines-a0i*:
refines R01 med10 a0i m1
⟨*proof*⟩

end

2.2 Refinement 2a: Authentic Channel Protocol

theory *m2-auth-chan* **imports** *m1-auth ../Refinement/Channels*
begin

We refine the abstract authentication protocol to a version of the ISO/IEC 9798-3 protocol using abstract channels. In standard protocol notation, the original protocol is specified as follows.

$$\begin{array}{lrcl} \text{M1.} & A \rightarrow B & : & A, B, N_A \\ \text{M2.} & B \rightarrow A & : & \{N_B, N_A, A\}_{K^{-1}(B)} \end{array}$$

We introduce insecure channels between pairs of agents for the first message and authentic channels for the second.

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

2.2.1 State

State: we extend the state with insecure and authentic channels defined above.

record *m2-state* = *m1-state* +
chan :: *chmsg set*

Observations.

type-synonym
m2-obs = *m1-state*

definition
m2-obs :: *m2-state* ⇒ *m2-obs* **where**
m2-obs s ≡ (|
runs = *runs s*
|)

2.2.2 Events

definition
m2-step1 :: [*rid-t*, *agent*, *agent*, *nonce*] ⇒ (*m2-state* × *m2-state*) *set*
where

```
m2-step1 Ra A B Na ≡ {(s, s1).
   — guards
   Ra ∉ dom (runs s) ∧
   Na = Ra$0 ∧

   — actions
   s1 = s(|
     runs := (runs s)(Ra ↦ (Init, [A, B], [])),
     chan := insert (Insec A B (Msg [aNon Na])) (chan s)
   |)
 }
```

definition
```
 m2-step2 :: [rid-t, agent, agent, nonce, nonce] ⇒ (m2-state × m2-state) set
```
where
```
 m2-step2 Rb A B Na Nb ≡ {(s, s1).
   — guards
   Rb ∉ dom (runs s) ∧
   Nb = Rb$0 ∧

   Insec A B (Msg [aNon Na]) ∈ chan s ∧                    — rcv M1

   — actions
   s1 = s(|
     runs := (runs s)(Rb ↦ (Resp, [A, B], [aNon Na])),
     chan := insert (Auth B A (Msg [aNon Nb, aNon Na])) (chan s)  — snd M2
   |)
 }
```

definition
```
 m2-step3 :: [rid-t, agent, agent, nonce, nonce] ⇒ (m2-state × m2-state) set
```
where
```
 m2-step3 Ra A B Na Nb ≡ {(s, s1).
   — guards
   runs s Ra = Some (Init, [A, B], []) ∧
   Na = Ra$0 ∧

   Auth B A (Msg [aNon Nb, aNon Na]) ∈ chan s ∧            — recv M2

   — actions
   s1 = s(|
     runs := (runs s)(Ra ↦ (Init, [A, B], [aNon Nb]))
   |)
 }
```

Intruder fake event.

definition — refines *Id*
```
 m2-fake :: (m2-state × m2-state) set
```
where
```
 m2-fake ≡ {(s, s1).

   — actions:
   s1 = s(| chan := fake ik0 (dom (runs s)) (chan s) |)
```

}

Transition system.

definition
m2-init :: *m2-state set*
where
m2-init ≡ { (|
runs = *Map.empty*,
chan = {}
|) }

definition
m2-trans :: (*m2-state* × *m2-state*) *set* **where**
m2-trans ≡ (⋃ *A B Ra Rb Na Nb*.
(*m2-step1 Ra A B Na*) ∪
(*m2-step2 Rb A B Na Nb*) ∪
(*m2-step3 Ra A B Na Nb*) ∪
m2-fake ∪
Id
)

definition
m2 :: (*m2-state*, *m2-obs*) *spec* **where**
m2 ≡ (|
init = *m2-init*,
trans = *m2-trans*,
obs = *m2-obs*
|)

lemmas *m2-defs* =
m2-def m2-init-def m2-trans-def m2-obs-def
m2-step1-def m2-step2-def m2-step3-def m2-fake-def

2.2.3 Invariants

Authentic channel and responder

This property relates the messages in the authentic channel to the responder run frame.

definition
m2-inv1-auth :: *m2-state set* **where**
m2-inv1-auth ≡ {*s*. ∀ *A B Na Nb*.
Auth B A (*Msg* [*aNon Nb*, *aNon Na*]) ∈ *chan s* ⟶ *B* ∉ *bad* ⟶ *A* ∉ *bad* ⟶
(∃ *Rb*. *runs s Rb* = *Some* (*Resp*, [*A*, *B*], [*aNon Na*]) ∧ *Nb* = *Rb*$*0*)
}

lemmas *m2-inv1-authI* =
m2-inv1-auth-def [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv1-authE* [*elim*] =
m2-inv1-auth-def [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv1-authD* [*dest*] =
m2-inv1-auth-def [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

Invariance proof.

lemma *PO-m2-inv2-init* [*iff*]:
  init m2 ⊆ *m2-inv1-auth*
⟨*proof*⟩

lemma *PO-m2-inv2-trans* [*iff*]:
  {*m2-inv1-auth*} *trans m2* {> *m2-inv1-auth*}
⟨*proof*⟩

lemma *PO-m2-inv2* [*iff*]: *reach m2* ⊆ *m2-inv1-auth*
⟨*proof*⟩

2.2.4 Refinement

Simulation relation and mediator function. This is a pure superposition refinement.

definition
  R12 :: (*m1-state* × *m2-state*) *set* **where**
  R12 ≡ {(*s*, *t*). *runs s* = *runs t*}   — That's it!

definition
  med21 :: *m2-obs* ⇒ *m1-obs* **where**
  med21 ≡ *id*

Refinement proof

lemma *PO-m2-step1-refines-m1-step1*:
  {*R12*}
    (*m1-step1 Ra A B Na*), (*m2-step1 Ra A B Na*)
  {> *R12*}
⟨*proof*⟩

lemma *PO-m2-step2-refines-m1-step2*:
  {*R12*}
    (*m1-step2 Ra A B Na Nb*), (*m2-step2 Ra A B Na Nb*)
  {> *R12*}
⟨*proof*⟩

lemma *PO-m2-step3-refines-m1-step3*:
  {*R12* ∩ *UNIV* × *m2-inv1-auth*}
    (*m1-step3 Ra A B Na Nb*), (*m2-step3 Ra A B Na Nb*)
  {> *R12*}
⟨*proof*⟩

New fake event refines skip.

lemma *PO-m2-fake-refines-m1-skip*:
  {*R12*} *Id*, *m2-fake* {> *R12*}
⟨*proof*⟩

lemmas *PO-m2-trans-refines-m1-trans* =
  PO-m2-step1-refines-m1-step1 PO-m2-step2-refines-m1-step2
  PO-m2-step3-refines-m1-step3 PO-m2-fake-refines-m1-skip

All together now...

lemma *PO-m2-refines-init-m1* [*iff*]:

init m2 $\subseteq$ *R12*“(*init m1*)
⟨*proof*⟩

lemma *PO-m2-refines-trans-m1* [*iff*]:
{*R12* $\cap$ *UNIV* $\times$ *m2-inv1-auth*}
(*trans m1*), (*trans m2*)
{> *R12*}
⟨*proof*⟩

lemma *PO-obs-consistent* [*iff*]:
obs-consistent R12 med21 m1 m2
⟨*proof*⟩

lemma *m2-refines-m1*:
refines
(*R12* $\cap$ *UNIV* $\times$ *m2-inv1-auth*)
med21 m1 m2
⟨*proof*⟩

end

2.3 Refinement 2b: Confidential Channel Protocol

theory *m2-confid-chan* **imports** *m1-auth ../Refinement/Channels*
begin

We refine the abstract authentication protocol to the first two steps of the Needham-Schroeder-Lowe protocol, which we call NSL/2. In standard protocol notation, the original protocol is specified as follows.

$$\begin{array}{lll} \text{M1.} & A \to B & : & \{N_A, A\}_{K(B)} \\ \text{M2.} & B \to A & : & \{N_A, N_B, B\}_{K(A)} \end{array}$$

At this refinement level, we abstract the encrypted messages to non-cryptographic messages transmitted on confidential channels.

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

2.3.1 State and observations

record *m2-state* = *m1-state* +
chan :: *chmsg set* — channels

type-synonym
m2-obs = *m1-state*

definition
m2-obs :: *m2-state* ⇒ *m2-obs* **where**
m2-obs s ≡ (|
runs = *runs s*
|)

2.3.2 Events

definition
  m2-init :: *m2-state set*
where
  m2-init ≡ { (|
    runs = *Map.empty*,
    chan = {}
  |) }

definition
  m2-step1 :: [*rid-t*, *agent*, *agent*, *nonce*] ⇒ (*m2-state* × *m2-state*) *set*
where
  m2-step1 Ra A B Na ≡ {(*s*, *s1*).

    — guards:
    Ra ∉ *dom* (*runs s*) ∧
    Na = *Ra$0* ∧

    — actions:
    s1 = *s*(|
      runs := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [])),
      — send *Na* on confidential channel 1
      chan := *insert* (*Confid A B* (*Msg* [*aNon Na*])) (*chan s*)
    |)
  }

definition
  m2-step2 :: [*rid-t*, *agent*, *agent*, *nonce*, *nonce*] ⇒ (*m2-state* × *m2-state*) *set*
where
  m2-step2 Rb A B Na Nb ≡ {(*s*, *s1*).

    — guards
    Rb ∉ *dom* (*runs s*) ∧
    Nb = *Rb$0* ∧

    Confid A B (*Msg* [*aNon Na*]) ∈ *chan s* ∧ — receive M1

    — actions
    s1 = *s*(|
      runs := (*runs s*)(*Rb* ↦ (*Resp*, [*A*, *B*], [*aNon Na*])),
      chan := *insert* (*Confid B A* (*Msg* [*aNon Na*, *aNon Nb*])) (*chan s*)
    |)
  }

definition
  m2-step3 :: [*rid-t*, *agent*, *agent*, *nonce*, *nonce*] ⇒ (*m2-state* × *m2-state*) *set*
where
  m2-step3 Ra A B Na Nb ≡ {(*s*, *s1*).

— guards
runs s Ra = Some (Init, [A, B], []) ∧
Na = Ra$0 ∧

Confid B A (Msg [aNon Na, aNon Nb]) ∈ chan s ∧ — receive M2

— actions
s1 = s⦇
runs := (runs s)(Ra ↦ (Init, [A, B], [aNon Nb]))
⦈
}

Intruder fake event.

definition — refines *Id*
m2-fake :: (m2-state × m2-state) set
where
m2-fake ≡ {(s, s1).
— actions:
s1 = s⦇ chan := fake ik0 (dom (runs s)) (chan s) ⦈
}

Transition system.

definition
m2-trans :: (m2-state × m2-state) set **where**
m2-trans ≡ (⋃ A B Ra Rb Na Nb.
m2-step1 Ra A B Na ∪
m2-step2 Rb A B Na Nb ∪
m2-step3 Ra A B Na Nb ∪
m2-fake ∪
Id
)

definition
m2 :: (m2-state, m2-obs) spec **where**
m2 ≡ ⦇
init = m2-init,
trans = m2-trans,
obs = m2-obs
⦈

lemmas *m2-defs =*
m2-def m2-init-def m2-trans-def m2-obs-def
m2-step1-def m2-step2-def m2-step3-def m2-fake-def

2.3.3 Invariants

Invariant 1: Messages only contains generated nonces.

definition
m2-inv1-nonces :: m2-state set **where**
m2-inv1-nonces ≡ {s. ∀ R.
aNon (R$0) ∈ atoms (chan s) ⟶ R ∈ dom (runs s)
}

lemmas *m2-inv1-noncesI* =
m2-inv1-nonces-def [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv1-noncesE* [*elim*] =
m2-inv1-nonces-def [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv1-noncesD* =
m2-inv1-nonces-def [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

lemma *PO-m2-inv1-init* [*iff*]: *init m2* ⊆ *m2-inv1-nonces*
⟨*proof*⟩

lemma *PO-m2-inv1-trans* [*iff*]:
{*m2-inv1-nonces*} *trans m2* {> *m2-inv1-nonces*}
⟨*proof*⟩

lemma *PO-m2-inv012* [*iff*]:
reach m2 ⊆ *m2-inv1-nonces*
⟨*proof*⟩

Invariant 3: relates message 2 with the responder run

It is needed, together with initiator nonce secrecy, in proof obligation REF/*m2-step2*.

definition
m2-inv3-msg2 :: *m2-state set* **where**
m2-inv3-msg2 ≡ {*s*. ∀ *A B Na Nb*.
Confid B A (*Msg* [*aNon Na*, *aNon Nb*]) ∈ *chan s* ⟶
aNon Na ∉ *extr ik0* (*chan s*) ⟶
(∃ *Rb*. *Nb* = *Rb*$*0* ∧ *runs s Rb* = *Some* (*Resp*, [*A*, *B*], [*aNon Na*]))
}

lemmas *m2-inv3-msg2I* = *m2-inv3-msg2-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv3-msg2E* [*elim*] = *m2-inv3-msg2-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv3-msg2D* = *m2-inv3-msg2-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

lemma *PO-m2-inv4-init* [*iff*]:
init m2 ⊆ *m2-inv3-msg2*
⟨*proof*⟩

lemma *PO-m2-inv4-trans* [*iff*]:
{*m2-inv3-msg2*} *trans m2* {> *m2-inv3-msg2*}
⟨*proof*⟩

lemma *PO-m2-inv4* [*iff*]: *reach m2* ⊆ *m2-inv3-msg2*
⟨*proof*⟩

Invariant 4: Initiator nonce secrecy.

It is needed in the proof obligation REF/*m2-step2*. It would be sufficient to prove the invariant for the case x = *None*, but we have generalized it here.

definition

m2-inv4-inon-secret :: *m2-state set* **where**
m2-inv4-inon-secret ≡ {*s*. ∀ *A B Ra al*.
runs s Ra = *Some* (*Init*, [*A*, *B*], *al*) ⟶
A ∉ *bad* ⟶ *B* ∉ *bad* ⟶
aNon (*Ra*$*0*) ∉ *extr ik0* (*chan s*)
}

lemmas *m2-inv4-inon-secretI* =
m2-inv4-inon-secret-def [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv4-inon-secretE* [*elim*] =
m2-inv4-inon-secret-def [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv4-inon-secretD* =
m2-inv4-inon-secret-def [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

lemma *PO-m2-inv3-init* [*iff*]:
init m2 ⊆ *m2-inv4-inon-secret*
⟨*proof*⟩

lemma *PO-m2-inv3-trans* [*iff*]:
{*m2-inv4-inon-secret* ∩ *m2-inv1-nonces*}
trans m2
{> *m2-inv4-inon-secret*}
⟨*proof*⟩

lemma *PO-m2-inv3* [*iff*]: *reach m2* ⊆ *m2-inv4-inon-secret*
⟨*proof*⟩

2.3.4 Refinement

definition
R12 :: (*m1-state* × *m2-state*) *set* **where**
R12 ≡ {(*s*, *t*). *runs s* = *runs t*}

abbreviation
med21 :: *m2-obs* ⇒ *m1-obs* **where**
med21 ≡ *id*

Proof obligations.

lemma *PO-m2-step1-refines-m1-step1*:
{*R12*}
(*m1-step1 Ra A B Na*), (*m2-step1 Ra A B Na*)
{> *R12*}
⟨*proof*⟩

lemma *PO-m2-step2-refines-m1-step2*:
{*R12*}
(*m1-step2 Rb A B Na Nb*), (*m2-step2 Rb A B Na Nb*)
{> *R12*}
⟨*proof*⟩

lemma *PO-m2-step3-refines-m1-step3*:
{*R12* ∩ *UNIV* × (*m2-inv4-inon-secret* ∩ *m2-inv3-msg2*)}

(*m1-step3 Ra A B Na Nb*), (*m2-step3 Ra A B Na Nb*)
{> *R12*}
⟨*proof*⟩

New fake events refine skip.

lemma *PO-m2-fake-refines-skip*:
{*R12*} *Id*, *m2-fake* {> *R12*}
⟨*proof*⟩

lemmas *PO-m2-trans-refines-m1-trans* =
PO-m2-step1-refines-m1-step1 PO-m2-step2-refines-m1-step2
PO-m2-step3-refines-m1-step3 PO-m2-fake-refines-skip

All together now...

lemma *PO-m2-refines-init-m1* [*iff*]:
init m2 ⊆ *R12*``(*init m1*)
⟨*proof*⟩

lemma *PO-m2-refines-trans-m1* [*iff*]:
{*R12* ∩
UNIV × (*m2-inv4-inon-secret* ∩ *m2-inv3-msg2*)}
(*trans m1*), (*trans m2*)
{> *R12*}
⟨*proof*⟩

lemma *PO-R12-obs-consistent* [*iff*]:
obs-consistent R12 med21 m1 m2
⟨*proof*⟩

lemma *PO-m3-refines-m2*:
refines
(*R12* ∩
UNIV × (*m2-inv4-inon-secret* ∩ *m2-inv3-msg2* ∩ *m2-inv1-nonces*))
med21 m1 m2
⟨*proof*⟩

end

2.4 Refinement 3a: Signature-based Dolev-Yao Protocol (Variant A)

theory *m3-sig* **imports** *m2-auth-chan ../Refinement/Message*
begin

We implement the channel protocol of the previous refinement with signatures and add a full-fledged Dolev-Yao adversary. In this variant, the adversary is realized using Paulson's closure operators for message derivation (as opposed to a collection of one-step derivation events a la Strand spaces).

Proof tool configuration. Avoid annoying automatic unfolding of *dom* (again).

declare *domIff* [*simp*, *iff del*]
declare *analz-into-parts* [*dest*]

2.4.1 State

We extend the state of *m1* with insecure and authentic channels between each pair of agents.

record *m3-state* = *m1-state* +
  IK :: *msg set*   — intruder knowledge

type-synonym
  m3-obs = *m1-state*

definition
  m3-obs :: *m3-state* ⇒ *m3-obs* **where**
  m3-obs s ≡ (|
    runs = *runs s*
  |)

2.4.2 Events

definition
  m3-step1 :: [*rid-t*, *agent*, *agent*, *nonce*] ⇒ (*m3-state* × *m3-state*) *set*
where
  m3-step1 Ra A B Na ≡ {(*s*, *s1*).

    — guards
    Ra ∉ *dom* (*runs s*) ∧
    Na = *Ra*$*0* ∧

    — actions
    s1 = *s*(|
      runs := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [])),
      IK := *insert* {|*Agent A*, *Agent B*, *Nonce Na*|} (*IK s*)   — send msg 1
    |)
  }

definition
  m3-step2 :: [*rid-t*, *agent*, *agent*, *nonce*, *nonce*] ⇒ (*m3-state* × *m3-state*) *set*
where
  m3-step2 Rb A B Na Nb ≡ {(*s*, *s1*).

    — guards
    Rb ∉ *dom* (*runs s*) ∧
    Nb = *Rb*$*0* ∧

    {|*Agent A*, *Agent B*, *Nonce Na*|} ∈ *IK s* ∧   — receive msg 1

    — actions
    s1 = *s*(|
      runs := (*runs s*)(*Rb* ↦ (*Resp*, [*A*, *B*], [*aNon Na*])),
                                                — send msg 2
      IK := *insert* (*Crypt* (*priK B*) {|*Nonce Nb*, *Nonce Na*, *Agent A*|}) (*IK s*)

⦈
}

definition
m3-step3 :: [*rid-t*, *agent*, *agent*, *nonce*, *nonce*] ⇒ (*m3-state* × *m3-state*) *set*
where
m3-step3 Ra A B Na Nb ≡ {(*s*, *s1*).

— guards
runs s Ra = *Some* (*Init*, [*A*, *B*], []) ∧
Na = *Ra*$*0* ∧

Crypt (*priK B*) {|*Nonce Nb*, *Nonce Na*, *Agent A*|} ∈ *IK s* ∧ — recv msg 2

— actions
s1 = *s*⦇
runs := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [*aNon Nb*]))
⦈
}

The intruder messages are now generated by a full-fledged Dolev-Yao intruder.

definition
m3-DY-fake :: (*m3-state* × *m3-state*) *set*
where
m3-DY-fake ≡ {(*s*, *s1*).

— actions:
s1 = *s*⦇
IK := *synth* (*analz* (*IK s*))
⦈
}

Transition system.

definition
m3-init :: *m3-state set*
where
m3-init ≡ { ⦇
runs = *Map.empty*,
IK = (*Key*‘*priK*‘*bad*) ∪ (*Key*‘*range pubK*) ∪ (*Key*‘*shrK*‘*bad*)
⦈ }

definition
m3-trans :: (*m3-state* × *m3-state*) *set* **where**
m3-trans ≡ (⋃ *A B Ra Rb Na Nb*.
m3-step1 Ra A B Na ∪
m3-step2 Rb A B Na Nb ∪
m3-step3 Ra A B Na Nb ∪
m3-DY-fake ∪
Id
)

definition

m3 :: (*m3-state*, *m3-obs*) *spec* **where**
m3 ≡ ⦇
init = *m3-init*,
trans = *m3-trans*,
obs = *m3-obs*
⦈

lemmas *m3-defs* =
m3-def m3-init-def m3-trans-def m3-obs-def
m3-step1-def m3-step2-def m3-step3-def
m3-DY-fake-def

2.4.3 Invariants

Specialize injectiveness of parts to enable aggressive application.

lemmas *parts-Inj-IK* = *parts.Inj* [**where** *H*=*IK s* **for** *s*]
lemmas *analz-Inj-IK* = *analz.Inj* [**where** *H*=*IK s* **for** *s*]

The following invariants do not depend on the protocol messages. We want to keep this compilation refinement from channel protocols to full-fledged Dolev-Yao protocols as generic as possible.

inv1: Long-term key secrecy

Private signing keys are secret, that is, the intruder only knows private keys of corrupted agents.

The invariant uses the weaker *parts* operator instead of the perhaps more intuitive *analz* in its premise. This strengthens the invariant and potentially simplifies its proof.

definition
m3-inv1-lkeysec :: *m3-state set* **where**
m3-inv1-lkeysec ≡ {*s*. ∀ *A*.
Key (*priK A*) ∈ *analz* (*IK s*) ⟶ *A* ∈ *bad*
}

lemmas *m3-inv1-lkeysecI* =
m3-inv1-lkeysec-def [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv1-lkeysecE* [*elim*] =
m3-inv1-lkeysec-def [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv1-lkeysecD* =
m3-inv1-lkeysec-def [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

lemma *PO-m3-inv1-lkeysec-init* [*iff*]:
init m3 ⊆ *m3-inv1-lkeysec*
⟨*proof*⟩

lemma *PO-m3-inv1-lkeysec-trans* [*iff*]:
{*m3-inv1-lkeysec*} *trans m3* {> *m3-inv1-lkeysec*}
⟨*proof*⟩

lemma *PO-m3-inv1-lkeysec* [*iff*]: *reach m3* ⊆ *m3-inv1-lkeysec*
⟨*proof*⟩

inv2: Intruder knows long-term keys of bad guys

definition
m3-inv2-badkeys :: *m3-state set*
where
m3-inv2-badkeys ≡ {*s*. ∀ *C*.
C ∈ *bad* ⟶ *Key* (*priK C*) ∈ *analz* (*IK s*)
}

lemmas *m3-inv2-badkeysI* =
m3-inv2-badkeys-def [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv2-badkeysE* [*elim*] =
m3-inv2-badkeys-def [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv2-badkeysD* [*dest*] =
m3-inv2-badkeys-def [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

Invariance proof.

lemma *PO-m3-inv2-badkeys-init* [*iff*]:
init m3 ⊆ *m3-inv2-badkeys*
⟨*proof*⟩

lemma *PO-m3-inv2-badkeys-trans* [*iff*]:
{*m3-inv2-badkeys*} *trans m3* {> *m3-inv2-badkeys*}
⟨*proof*⟩

lemma *PO-m3-inv2-badkeys* [*iff*]: *reach m3* ⊆ *m3-inv2-badkeys*
⟨*proof*⟩

inv3: Intruder knows all public keys (NOT USED)

This invariant is only needed with equality in *R23-msgs*.

definition
m3-inv3-pubkeys :: *m3-state set*
where
m3-inv3-pubkeys ≡ {*s*. ∀ *C*.
Key (*pubK C*) ∈ *analz* (*IK s*)
}

lemmas *m3-inv3-pubkeysI* =
m3-inv3-pubkeys-def [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv3-pubkeysE* [*elim*] =
m3-inv3-pubkeys-def [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv3-pubkeysD* [*dest*] =
m3-inv3-pubkeys-def [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

Invariance proof.

lemma *PO-m3-inv3-pubkeys-init* [*iff*]:
init m3 ⊆ *m3-inv3-pubkeys*
⟨*proof*⟩

lemma *PO-m3-inv3-pubkeys-trans* [*iff*]:
{*m3-inv3-pubkeys*} *trans m3* {> *m3-inv3-pubkeys*}
⟨*proof*⟩

lemma *PO-m3-inv3-pubkeys* [*iff*]: *reach m3* ⊆ *m3-inv3-pubkeys*
⟨*proof*⟩

2.4.4 Refinement

Automatic tool tuning. Tame too-agressive pair decomposition, which is declared as a safe elim rule ([elim!]).

lemmas *MPair-parts* [*rule del*, *elim*]
lemmas *MPair-analz* [*rule del*, *elim*]

Simulation relation

abbreviation
nonces :: *msg set* ⇒ *nonce set*
where
nonces H ≡ {*N*. *Nonce N* ∈ *analz H*}

abbreviation
ink :: *chmsg set* ⇒ *nonce set*
where
ink H ≡ {*N*. *aNon N* ∈ *extr ik0 H*}

Abstraction function on sets of messages.

inductive-set
abs-msg :: *msg set* ⇒ *chmsg set*
for *H* :: *msg set*
where
am-M1:
⦃*Agent A*, *Agent B*, *Nonce Na*⦄ ∈ *H*
⟹ *Insec A B* (*Msg* [*aNon Na*]) ∈ *abs-msg H*

| *am-M2*:
Crypt (*priK B*) ⦃*Nonce Nb*, *Nonce Na*, *Agent A*⦄ ∈ *H*
⟹ *Auth B A* (*Msg* [*aNon Nb*, *aNon Na*]) ∈ *abs-msg H*

The simulation relation is canonical. It states that the protocol messages in the intruder knowledge refine the abstract messages appearing in the channels *Insec* and *Auth*.

definition
R23-msgs :: (*m2-state* × *m3-state*) *set* **where**
R23-msgs ≡ {(*s*, *t*). *abs-msg* (*parts* (*IK t*)) ⊆ *chan s*} — with *parts*!

definition
R23-ink :: (*m2-state* × *m3-state*) *set* **where**
R23-ink ≡ {(*s*, *t*). *nonces* (*IK t*) ⊆ *ink* (*chan s*)}

definition
R23-preserved :: (*m2-state* × *m3-state*) *set* **where**

R23-preserved ≡ {(*s*, *t*). *runs s* = *runs t*}

definition
R23 :: (*m2-state* × *m3-state*) *set* **where**
R23 ≡ *R23-msgs* ∩ *R23-ink* ∩ *R23-preserved*

lemmas *R23-defs* = *R23-def R23-msgs-def R23-ink-def R23-preserved-def*

Mediator function: nothing new.

definition
med32 :: *m3-obs* ⇒ *m2-obs* **where**
med32 ≡ *id*

lemmas *R23-msgsI* =
R23-msgs-def [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-msgsE* [*elim*] =
R23-msgs-def [*THEN rel-def-to-elim*, *simplified*, *rule-format*]
lemmas *R23-msgsE′* [*elim*] =
R23-msgs-def [*THEN rel-def-to-dest*, *simplified*, *rule-format*, *THEN subsetD*]

lemmas *R23-inkI* =
R23-ink-def [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-inkE* [*elim*] =
R23-ink-def [*THEN rel-def-to-elim*, *simplified*, *rule-format*]

lemmas *R23-preservedI* =
R23-preserved-def [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-preservedE* [*elim*] =
R23-preserved-def [*THEN rel-def-to-elim*, *simplified*, *rule-format*]

lemmas *R23-intros* = *R23-msgsI R23-inkI R23-preservedI*

Facts about the abstraction function

declare *abs-msg.intros* [*intro*!]
declare *abs-msg.cases* [*elim*!]

lemma *abs-msg-empty*: *abs-msg* {} = {}
⟨*proof*⟩

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

lemma *abs-msg-mono* [*elim*]:
⟦ *m* ∈ *abs-msg G*; *G* ⊆ *H* ⟧ ⟹ *m* ∈ *abs-msg H*
⟨*proof*⟩

lemma *abs-msg-insert-mono* [*intro*]:
⟦ *m* ∈ *abs-msg H* ⟧ ⟹ *m* ∈ *abs-msg* (*insert m′ H*)
⟨*proof*⟩

Abstraction of concretely fakeable message yields abstractly fakeable messages. This is the key lemma for the refinement of the intruder.

lemma *abs-msg-DY-subset-fakeable*:
  ⟦ (*s*, *t*) ∈ *R23-msgs*; (*s*, *t*) ∈ *R23-ink*; *t* ∈ *m3-inv1-lkeysec* ⟧
  ⟹ *abs-msg* (*synth* (*analz* (*IK t*))) ⊆ *fake ik0* (*dom* (*runs s*)) (*chan s*)
⟨*proof*⟩

lemma *absmsg-parts-subset-fakeable*:
  ⟦ (*s*, *t*) ∈ *R23-msgs* ⟧
  ⟹ *abs-msg* (*parts* (*IK t*)) ⊆ *fake ik0* (−*dom* (*runs s*)) (*chan s*)
⟨*proof*⟩

declare *abs-msg-DY-subset-fakeable* [*simp*, *intro*!]
declare *absmsg-parts-subset-fakeable* [*simp*, *intro*!]

Refinement proof

lemma *PO-m3-step1-refines-m2-step1*:
  {*R23*}
    (*m2-step1 Ra A B Na*), (*m3-step1 Ra A B Na*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-step2-refines-m2-step2*:
  {*R23* ~~∩ *UNIV* × *m3-inv3-pubkeys*~~}
    (*m2-step2 Rb A B Na Nb*), (*m3-step2 Rb A B Na Nb*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-step3-refines-m2-step3*:
  {*R23*}
    (*m2-step3 Ra A B Na Nb*), (*m3-step3 Ra A B Na Nb*)
  {> *R23*}
⟨*proof*⟩

The Dolev-Yao fake event refines the abstract fake event.

lemma *PO-m3-DY-fake-refines-m2-fake*:
  {*R23* ∩ *UNIV* × (*m3-inv1-lkeysec* ∩ *m3-inv2-badkeys*)}
    m2-fake, *m3-DY-fake*
  {> *R23*}
⟨*proof*⟩

All together now...

lemmas *PO-m3-trans-refines-m2-trans* =
  PO-m3-step1-refines-m2-step1 PO-m3-step2-refines-m2-step2
  PO-m3-step3-refines-m2-step3 PO-m3-DY-fake-refines-m2-fake

lemma *PO-m3-refines-init-m2* [*iff*]:
  init m3 ⊆ *R23*‘‘(*init m2*)
⟨*proof*⟩

lemma *PO-m3-refines-trans-m2* [*iff*]:

{*R23* ∩ *UNIV* × (*m3-inv2-badkeys* ∩ *m3-inv1-lkeysec*)}
(*trans m2*), (*trans m3*)
{> *R23*}
⟨*proof*⟩

lemma *PO-obs-consistent* [*iff*]:
obs-consistent R23 med32 m2 m3
⟨*proof*⟩

lemma *PO-m3-refines-m2*:
refines
(*R23* ∩ *UNIV* × (*m3-inv2-badkeys* ∩ *m3-inv1-lkeysec*))
med32 m2 m3
⟨*proof*⟩

end

2.5 Refinement 3b: Encryption-based Dolev-Yao Protocol (Variant A)

theory *m3-enc* **imports** *m2-confid-chan ../Refinement/Message*
begin

This refines the channel protocol using public-key encryption and adds a full-fledged Dolev-Yao adversary. In this variant, the adversary is realized using Paulson's message derivation closure operators (as opposed to a collection of one-step message construction and decomposition events a la Strand spaces).

Proof tool configuration. Avoid annoying automatic unfolding of *dom* (again).

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

A general lemma about *parts* (move?!).

lemmas *parts-insertD* = *parts-insert* [*THEN equalityD1*, *THEN subsetD*]

2.5.1 State and observations

We extend the state of *m1* with two confidential channels between each pair of agents, one channel for each protocol message.

record *m3-state* = *m1-state* +
IK :: *msg set* — intruder knowledge

Observations: local agent states.

type-synonym
m3-obs = *m1-obs*

definition
m3-obs :: *m3-state* ⇒ *m3-obs* **where**
m3-obs s ≡ (|
runs = *runs s*
|)

2.5.2 Events

definition
  $m3\text{-}step1 :: [rid\text{-}t, agent, agent, nonce] \Rightarrow (m3\text{-}state \times m3\text{-}state)\ set$
where
  $m3\text{-}step1\ Ra\ A\ B\ Na \equiv \{(s, s1).$

— guards:
$Ra \notin dom\ (runs\ s) \wedge$
$Na = Ra\$0 \wedge$

— actions:
$s1 = s(\!|$
$runs := (runs\ s)(Ra \mapsto (Init, [A, B], [])),$
$IK := insert\ (Crypt\ (pubK\ B)\ \{\!|Nonce\ Na, Agent\ A|\!\})\ (IK\ s)$
$|\!)$
$\}$

definition
  $m3\text{-}step2 ::$
  $[rid\text{-}t, agent, agent, nonce, nonce] \Rightarrow (m3\text{-}state \times m3\text{-}state)\ set$
where
  $m3\text{-}step2\ Rb\ A\ B\ Na\ Nb \equiv \{(s, s1).$

— guards
$Rb \notin dom\ (runs\ s) \wedge$
$Nb = Rb\$0 \wedge$

$Crypt\ (pubK\ B)\ \{\!|Nonce\ Na, Agent\ A|\!\} \in IK\ s \wedge$ — receive msg 1

— actions
$s1 = s(\!|$
$runs := (runs\ s)(Rb \mapsto (Resp, [A, B], [aNon\ Na])),$
$IK := insert\ (Crypt\ (pubK\ A)\ \{\!|Nonce\ Na, Nonce\ Nb, Agent\ B|\!\})\ (IK\ s)$
$|\!)$
$\}$

definition
  $m3\text{-}step3 :: [rid\text{-}t, agent, agent, nonce, nonce] \Rightarrow (m3\text{-}state \times m3\text{-}state)\ set$
where
  $m3\text{-}step3\ Ra\ A\ B\ Na\ Nb \equiv \{(s, s1).$

— guards
$runs\ s\ Ra = Some\ (Init, [A, B], []) \wedge$
$Na = Ra\$0 \wedge$

$Crypt\ (pubK\ A)\ \{\!|Nonce\ Na, Nonce\ Nb, Agent\ B|\!\} \in IK\ s \wedge$ — recv msg2

— actions
$s1 = s(\!|$
$runs := (runs\ s)(Ra \mapsto (Init, [A, B], [aNon\ Nb]))$
$|\!)$
$\}$

Standard Dolev-Yao intruder.

```
definition
  m3-DY-fake :: (m3-state × m3-state) set
where
  m3-DY-fake ≡ {(s, s1).

    — actions:
    s1 = s⦇ IK := synth (analz (IK s)) ⦈
  }
```

Transition system.

```
definition
  m3-init :: m3-state set
where
  m3-init ≡ { ⦇
    runs = Map.empty,
    IK = (Key`priK`bad) ∪ (Key`range pubK) ∪ (Key`shrK`bad)
  ⦈ }

definition
  m3-trans :: (m3-state × m3-state) set where
  m3-trans ≡ (⋃ A B Ra Rb Na Nb.
    m3-step1 Ra A B Na    ∪
    m3-step2 Rb A B Na Nb ∪
    m3-step3 Ra A B Na Nb ∪
    m3-DY-fake ∪
    Id
  )

definition
  m3 :: (m3-state, m3-obs) spec where
  m3 ≡ ⦇
   init = m3-init,
   trans = m3-trans,
   obs = m3-obs
  ⦈

lemmas m3-defs =
  m3-def m3-init-def m3-trans-def m3-obs-def
  m3-step1-def m3-step2-def m3-step3-def
  m3-DY-fake-def
```

2.5.3 Invariants

Automatic tool tuning. Tame too-agressive pair decomposition, which is declared as a safe elim rule ([elim!]).

```
lemmas MPair-parts [rule del, elim]
lemmas MPair-analz [rule del, elim]
```

Specialize injectiveness of *parts* and *analz* to enable aggressive application.

```
lemmas parts-Inj-IK = parts.Inj [where H=IK s for s]
```

lemmas *analz-Inj-IK* = *analz.Inj* [**where** *H*=*IK s* **for** *s*]

declare *analz-into-parts* [*dest*]

inv1: Key secrecy

Decryption keys are secret, that is, the intruder only knows private keys of corrupted agents.

definition
  m3-inv1-keys :: *m3-state set* **where**
  m3-inv1-keys ≡ {*s*. ∀ *A*.
    Key (*priK A*) ∈ *parts* (*IK s*) ⟶ *A* ∈ *bad*
  }

lemmas *m3-inv1-keysI* = *m3-inv1-keys-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv1-keysE* [*elim*] =
  m3-inv1-keys-def [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv1-keysD* [*dest*] =
  m3-inv1-keys-def [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

lemma *PO-m3-inv1-keys-init* [*iff*]:
  init m3 ⊆ *m3-inv1-keys*
⟨*proof*⟩

lemma *PO-m3-inv1-keys-trans* [*iff*]:
  {*m3-inv1-keys*} *trans m3* {> *m3-inv1-keys*}
⟨*proof*⟩

lemma *PO-m3-inv1-keys* [*iff*]: *reach m3* ⊆ *m3-inv1-keys*
⟨*proof*⟩

2.5.4 Simulation relation

Simulation relation is canonical. It states that the protocol messages appearing in the intruder knowledge refine those occurring on the abstract confidential channels. Moreover, if the concrete intruder knows a nonce then so does the abstract one (as defined by *ink*).

Abstraction function on sets of messages.

inductive-set
  abs-msg :: *msg set* ⇒ *chmsg set*
  for *H* :: *msg set*
where
  am-msg1:
    Crypt (*pubK B*) ⦃*Nonce Na*, *Agent A*⦄ ∈ *H*
  ⟹ *Confid A B* (*Msg* [*aNon Na*]) ∈ *abs-msg H*

| *am-msg2*:
    Crypt (*pubK A*) ⦃*Nonce Na*, *Nonce Nb*, *Agent B*⦄ ∈ *H*
  ⟹ *Confid B A* (*Msg* [*aNon Na*, *aNon Nb*]) ∈ *abs-msg H*

declare *abs-msg.intros* [*intro*!]

declare *abs-msg.cases* [*elim*!]

The simulation relation is canonical. It states that the protocol messages in the intruder knowledge refine the abstract messages appearing on the confidential channels.

definition
R23-msgs :: (*m2-state* × *m3-state*) *set* **where**
R23-msgs ≡ {(*s*, *t*). *abs-msg* (*parts* (*IK t*)) ⊆ *chan s*} — with *parts*!

definition
R23-non :: (*m2-state* × *m3-state*) *set* **where**
R23-non ≡ {(*s*, *t*). ∀ *N*. *Nonce N* ∈ *analz* (*IK t*) ⟶ *aNon N* ∈ *extr ik0* (*chan s*)}

definition
R23-pres :: (*m2-state* × *m3-state*) *set* **where**
R23-pres ≡ {(*s*, *t*). *runs s* = *runs t*}

definition
R23 :: (*m2-state* × *m3-state*) *set* **where**
R23 ≡ *R23-msgs* ∩ *R23-non* ∩ *R23-pres*

lemmas *R23-defs* =
R23-def R23-msgs-def R23-non-def R23-pres-def

lemmas *R23-msgsI* =
R23-msgs-def [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-msgsE* [*elim*] =
R23-msgs-def [*THEN rel-def-to-elim*, *simplified*, *rule-format*]
lemmas *R23-msgsE′* [*elim*] =
R23-msgs-def [*THEN rel-def-to-dest*, *simplified*, *rule-format*, *THEN subsetD*]

lemmas *R23-nonI* =
R23-non-def [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-nonE* [*elim*] =
R23-non-def [*THEN rel-def-to-elim*, *simplified*, *rule-format*]

lemmas *R23-presI* =
R23-pres-def [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-presE* [*elim*] =
R23-pres-def [*THEN rel-def-to-elim*, *simplified*, *rule-format*]

lemmas *R23-intros* = *R23-msgsI R23-nonI R23-presI*

Mediator function.

abbreviation
med32 :: *m3-obs* ⇒ *m2-obs* **where**
med32 ≡ *id*

2.5.5 Misc lemmas

General facts about *abs-msg*

lemma *abs-msg-empty*: *abs-msg* {} = {}
⟨*proof*⟩

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

lemma *abs-msg-mono* [*elim*]:
  ⟦ $m \in$ *abs-msg* G; $G \subseteq H$ ⟧ ⟹ $m \in$ *abs-msg* H
⟨*proof*⟩

lemma *abs-msg-insert-mono* [*intro*]:
  ⟦ $m \in$ *abs-msg* H ⟧ ⟹ $m \in$ *abs-msg* (*insert* m' H)
⟨*proof*⟩

Abstraction of concretely fakeable message yields abstractly fakeable messages. This is the key lemma for the refinement of the intruder.

lemma *abs-msg-DY-subset-fake*:
  ⟦ $(s, t) \in$ *R23-msgs*; $(s, t) \in$ *R23-non*; $t \in$ *m3-inv1-keys* ⟧
  ⟹ *abs-msg* (*synth* (*analz* (*IK* t))) ⊆ *fake ik0* (*dom* (*runs* s)) (*chan* s)
⟨*proof*⟩

lemma *abs-msg-parts-subset-fake*:
  ⟦ $(s, t) \in$ *R23-msgs* ⟧
  ⟹ *abs-msg* (*parts* (*IK* t)) ⊆ *fake ik0* (−*dom* (*runs* s)) (*chan* s)
⟨*proof*⟩

declare *abs-msg-DY-subset-fake* [*simp*, *intro*!]
declare *abs-msg-parts-subset-fake* [*simp*, *intro*!]

2.5.6 Refinement proof

Proofs obligations.

lemma *PO-m3-step1-refines-m2-step1*:
  {*R23* ∩ *UNIV* × *m3-inv1-keys*}
    (*m2-step1 Ra A B Na*), (*m3-step1 Ra A B Na*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-step2-refines-m2-step2*:
  {*R23* ∩ *UNIV* × *m3-inv1-keys*}
    (*m2-step2 Rb A B Na Nb*), (*m3-step2 Rb A B Na Nb*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-step3-refines-m2-step3*:
  {*R23*}
    (*m2-step3 Ra A B Na Nb*), (*m3-step3 Ra A B Na Nb*)
  {> *R23*}
⟨*proof*⟩

Dolev-Yao fake event refines abstract fake event.

lemma *PO-m3-DY-fake-refines-m2-fake*:
  {*R23* ∩ *UNIV* × *m3-inv1-keys*}

(*m2-fake*), (*m3-DY-fake*)
{> *R23*}
⟨*proof*⟩

All together now...

lemmas *PO-m3-trans-refines-m2-trans* =
PO-m3-step1-refines-m2-step1 PO-m3-step2-refines-m2-step2
PO-m3-step3-refines-m2-step3 PO-m3-DY-fake-refines-m2-fake

lemma *PO-m3-refines-init-m2* [*iff*]:
init m3 ⊆ *R23*''(*init m2*)
⟨*proof*⟩

lemma *PO-m3-refines-trans-m2* [*iff*]:
{*R23* ∩ *UNIV* × *m3-inv1-keys*}
(*trans m2*), (*trans m3*)
{> *R23*}
⟨*proof*⟩

lemma *PO-R23-obs-consistent* [*iff*]:
obs-consistent R23 med32 m2 m3
⟨*proof*⟩

lemma *PO-m3-refines-m2* [*iff*]:
refines
(*R23* ∩ *UNIV* × *m3-inv1-keys*)
med32 m2 m3
⟨*proof*⟩

end

Chapter 3

Key Establishment Protocols

In this chapter, we develop several key establishment protocols:

- Needham-Schroeder Shared Key (NSSK)
- core Kerberos IV and V, and
- Denning-Sacco.

3.1 Basic abstract key distribution (L1)

theory *m1-keydist* **imports** *../Refinement/Runs ../Refinement/s0g-secrecy*
begin

The first refinement introduces the protocol roles, local memory of the agents and the communication structure of the protocol. For actual communication, the "receiver" directly reads the memory of the "sender".

It captures the core of essentials of server-based key distribution protocols: The server generates a key that the clients read from his memory. At this stage we are only interested in secrecy preservation, not in authentication.

declare *option.split-asm* [*split*]
declare *domIff* [*simp*, *iff del*]

consts
  sk :: *nat* — identifier used for session keys

3.1.1 State

Runs record the protocol participants (initiator, responder) and the keys learned during the execution. In later refinements, we will also add nonces and timestamps to the run record.

The variables *kn* and *az* from *s0g-secrecy-leak* are replaced by runs using a data refinement. Variable *lk* is concretized into variable *leak*.

We define the state in two separate record definitions. The first one has just a runs field and the second extends this with a leak field. Later refinements may define different state for leaks (e.g. to record more context).

record *m1r-state* =
  runs :: *runs-t*

record *m1x-state* = *m1r-state* +
  leak :: *key set* — keys leaked to attacker

type-synonym *m1x-obs* = *m1x-state*

Predicate types for invariants and transition relation types. Use the r-version for invariants and transitions if there is no reference to the leak variable. This improves reusability in later refinements.

type-synonym *'x m1r-pred* = *'x m1r-state-scheme set*
type-synonym *'x m1x-pred* = *'x m1x-state-scheme set*

type-synonym *'x m1r-trans* = (*'x m1r-state-scheme* × *'x m1r-state-scheme*) *set*
type-synonym *'x m1x-trans* = (*'x m1x-state-scheme* × *'x m1x-state-scheme*) *set*

Key knowledge and authorization (reconstruction)

Key knowledge and authorization relations, reconstructed from the runs and an unspecified initial key setup. These auxiliary definitions are used in some event guards and in the simulation relation (see below).

Knowledge relation (reconstructed)

inductive-set
  knC :: *runs-t* ⇒ (*key* × *agent*) *set* **for** *runz* :: *runs-t*
where
  knC-init:
    runz Ra = *Some* (*Init*, [*A*, *B*], *aKey K* # *al*) ⟹ (*K*, *A*) ∈ *knC runz*
| *knC-resp*:
    runz Rb = *Some* (*Resp*, [*A*, *B*], *aKey K* # *al*) ⟹ (*K*, *B*) ∈ *knC runz*
| *knC-serv*:
    ⟦ *Rs* ∈ *dom runz*; *fst* (*the* (*runz Rs*)) = *Serv* ⟧ ⟹ (*sesK* (*Rs$sk*), *Sv*) ∈ *knC runz*
| *knC-0*:
    (*K*, *A*) ∈ *keySetup* ⟹ (*K*, *A*) ∈ *knC runz*

Authorization relation (reconstructed)

inductive-set
  azC :: *runs-t* ⇒ (*key* × *agent*) *set* **for** *runz* :: *runs-t*
where
  azC-good:
    ⟦ *runz Rs* = *Some* (*Serv*, [*A*, *B*], *al*); *C* ∈ {*A*, *B*, *Sv*} ⟧
   ⟹ (*sesK* (*Rs$sk*), *C*) ∈ *azC runz*
| *azC-bad*:
    ⟦ *runz Rs* = *Some* (*Serv*, [*A*, *B*], *al*); *A* ∈ *bad* ∨ *B* ∈ *bad* ⟧
   ⟹ (*sesK* (*Rs$sk*), *C*) ∈ *azC runz*
| *azC-0*:
    ⟦ (*K*, *C*) ∈ *keySetup* ⟧ ⟹ (*K*, *C*) ∈ *azC runz*

declare *knC.intros* [*intro*]

declare *azC.intros* [*intro*]

Misc lemmas: empty state, projections, ...

lemma *knC-empty* [*simp*]: *knC Map.empty* = *keySetup*
⟨*proof*⟩

lemma *azC-empty* [*simp*]: *azC Map.empty* = *keySetup*
⟨*proof*⟩

azC and run abstraction

lemma *azC-map-runs* [*simp*]: *azC* (*map-runs h runz*) = *azC runz*
⟨*proof*⟩

Update lemmas for *knC*

lemma *knC-upd-Init-Resp-None*:
  ⟦ $R \notin dom\ runz$; $rol \in \{Init, Resp\}$ ⟧
  $\Longrightarrow knC\ (runz(R \mapsto (rol, [A, B], []))) = knC\ runz$
⟨*proof*⟩

lemma *knC-upd-Init-Some*:
  ⟦ $runz\ Ra = Some\ (Init, [A, B], [])$ ⟧
  $\Longrightarrow knC\ (runz(Ra \mapsto (Init, [A, B], [aKey\ Kab]))) = insert\ (Kab, A)\ (knC\ runz)$
⟨*proof*⟩

lemma *knC-upd-Resp-Some*:
  ⟦ $runz\ Ra = Some\ (Resp, [A, B], [])$ ⟧
  $\Longrightarrow knC\ (runz(Ra \mapsto (Resp, [A, B], [aKey\ Kab]))) = insert\ (Kab, B)\ (knC\ runz)$
⟨*proof*⟩

lemma *knC-upd-Server*:
  ⟦ $Rs \notin dom\ runz$ ⟧
  $\Longrightarrow knC\ (runz(Rs \mapsto (Serv, [A, B], []))) = insert\ (sesK\ (Rs\$sk), Sv)\ (knC\ runz)$
⟨*proof*⟩

lemmas *knC-upd-lemmas* [*simp*] =
  knC-upd-Init-Resp-None knC-upd-Init-Some knC-upd-Resp-Some
  knC-upd-Server

Update lemmas for *azC*

lemma *azC-upd-Init-None*:
  ⟦ $Ra \notin dom\ runz$ ⟧
  $\Longrightarrow azC\ (runz(Ra \mapsto (Init, [A, B], []))) = azC\ runz$
⟨*proof*⟩

lemma *azC-upd-Resp-None*:
  ⟦ $Rb \notin dom\ runz$ ⟧
  $\Longrightarrow azC\ (runz(Rb \mapsto (Resp, [A, B], []))) = azC\ runz$
⟨*proof*⟩

lemma *azC-upd-Init-Some*:
  ⟦ $runz\ Ra = Some\ (Init, [A, B], [])$ ⟧
  $\Longrightarrow azC\ (runz(Ra \mapsto (Init, [A, B], al))) = azC\ runz$

⟨proof⟩

lemma *azC-upd-Resp-Some*:
  $[\![$ *runz Rb* = *Some* (*Resp*, [*A*, *B*], []) $]\!]$
  $\Longrightarrow$ *azC* (*runz*(*Rb* $\mapsto$ (*Resp*, [*A*, *B*], *al*))) = *azC runz*
⟨proof⟩

lemma *azC-upd-Serv-bad*:
  $[\![$ *Rs* $\notin$ *dom runz*; *A* $\in$ *bad* $\vee$ *B* $\in$ *bad* $]\!]$
  $\Longrightarrow$ *azC* (*runz*(*Rs* $\mapsto$ (*Serv*, [*A*, *B*], *al*))) = *azC runz* $\cup$ {*sesK* (*Rs*\$*sk*)} $\times$ *UNIV*
⟨proof⟩

lemma *azC-upd-Serv-good*:
  $[\![$ *Rs* $\notin$ *dom runz*; *K* = *sesK* (*Rs*\$*sk*); *A* $\notin$ *bad*; *B* $\notin$ *bad* $]\!]$
  $\Longrightarrow$ *azC* (*runz*(*Rs* $\mapsto$ (*Serv*, [*A*, *B*], *al*)))
      = *azC runz* $\cup$ {(*K*, *A*), (*K*, *B*), (*K*, *Sv*)}
⟨proof⟩

lemma *azC-upd-Serv*:
  $[\![$ *Rs* $\notin$ *dom runz*; *K* = *sesK* (*Rs*\$*sk*) $]\!]$
  $\Longrightarrow$ *azC* (*runz*(*Rs* $\mapsto$ (*Serv*, [*A*, *B*], *al*))) =
     azC runz $\cup$ {*K*} $\times$ (*if A* $\notin$ *bad* $\wedge$ *B* $\notin$ *bad then* {*A*, *B*, *Sv*} *else UNIV*)
⟨proof⟩

lemmas *azC-upd-lemmas* [*simp*] =
  azC-upd-Init-None azC-upd-Resp-None
  azC-upd-Init-Some azC-upd-Resp-Some azC-upd-Serv

3.1.2 Events

definition — by *A*, refines skip
  m1x-step1 :: [*rid-t*, *agent*, *agent*] $\Rightarrow$ *'x m1r-trans*
where
  m1x-step1 Ra A B $\equiv$ {(*s*, *s1*).

    — guards:
    Ra $\notin$ *dom* (*runs s*) $\wedge$ — *Ra* is fresh

    — actions:
    — create initiator thread
    s1 = *s*(| *runs* := (*runs s*)(*Ra* $\mapsto$ (*Init*, [*A*, *B*], [])) |)
  }

definition — by *B*, refines skip
  m1x-step2 :: [*rid-t*, *agent*, *agent*] $\Rightarrow$ *'x m1r-trans*
where
  m1x-step2 Rb A B $\equiv$ {(*s*, *s1*).

    — guards:
    Rb $\notin$ *dom* (*runs s*) $\wedge$ — *Rb* is fresh

    — actions:
    — create responder thread

$s1 = s(\!|\ runs := (runs\ s)(Rb \mapsto (Resp,\ [A,\ B],\ []))\ |\!)$
}

definition — by *Sv*, refines *s0g-gen*
m1x-step3 :: [*rid-t*, *agent*, *agent*, *key*] ⇒ *'x m1r-trans*
where
m1x-step3 Rs A B Kab ≡ {(*s*, *s1*).

— guards:
$Rs \notin dom\ (runs\ s) \wedge$ — *Rs* is fresh
$Kab = sesK\ (Rs\$sk) \wedge$ — generate session key

— actions:
$s1 = s(\!|\ runs := (runs\ s)(Rs \mapsto (Serv,\ [A,\ B],\ []))\ |\!)$
}

definition — by *A*, refines *s0g-learn*
m1x-step4 :: [*rid-t*, *agent*, *agent*, *key*] ⇒ *'x m1x-trans*
where
m1x-step4 Ra A B Kab ≡ {(*s*, *s1*).
— guards:
$runs\ s\ Ra = Some\ (Init,\ [A,\ B],\ []) \wedge$
$(Kab \notin leak\ s \longrightarrow (Kab,\ A) \in azC\ (runs\ s)) \wedge$ — authorization guard

— actions:
$s1 = s(\!|\ runs := (runs\ s)(Ra \mapsto (Init,\ [A,\ B],\ [aKey\ Kab]))\ |\!)$
}

definition — by *B*, refines *s0g-learn*
m1x-step5 :: [*rid-t*, *agent*, *agent*, *key*] ⇒ *'x m1x-trans*
where
m1x-step5 Rb A B Kab ≡ {(*s*, *s1*).
— guards:
$runs\ s\ Rb = Some\ (Resp,\ [A,\ B],\ []) \wedge$
$(Kab \notin leak\ s \longrightarrow (Kab,\ B) \in azC\ (runs\ s)) \wedge$ — authorization guard

— actions:
$s1 = s(\!|\ runs := (runs\ s)(Rb \mapsto (Resp,\ [A,\ B],\ [aKey\ Kab]))\ |\!)$
}

definition — by attacker, refines *s0g-leak*
m1x-leak :: *rid-t* ⇒ *'x m1x-trans*
where
m1x-leak Rs ≡ {(*s*, *s1*).
— guards:
$Rs \in dom\ (runs\ s) \wedge$
$fst\ (the\ (runs\ s\ Rs)) = Serv \wedge$ — compromise server run *Rs*

— actions:
$s1 = s(\!|\ leak := insert\ (sesK\ (Rs\$sk))\ (leak\ s)\ |\!)$
}

3.1.3 Specification

definition
  $m1x\text{-}init :: m1x\text{-}state\ set$
where
  $m1x\text{-}init \equiv \{ (\!|$
    $runs = Map.empty,$
    $leak = corrKey$ — statically corrupted keys initially leaked
  $|\!) \}$

definition
  $m1x\text{-}trans :: 'x\ m1x\text{-}trans$ **where**
  $m1x\text{-}trans \equiv (\bigcup A\ B\ Ra\ Rb\ Rs\ Kab.$
    $m1x\text{-}step1\ Ra\ A\ B \cup$
    $m1x\text{-}step2\ Rb\ A\ B \cup$
    $m1x\text{-}step3\ Rs\ A\ B\ Kab \cup$
    $m1x\text{-}step4\ Ra\ A\ B\ Kab \cup$
    $m1x\text{-}step5\ Rb\ A\ B\ Kab \cup$
    $m1x\text{-}leak\ Rs \cup$
    Id
)

definition
  $m1x :: (m1x\text{-}state, m1x\text{-}obs)\ spec$ **where**
  $m1x \equiv (\!|$
   $init = m1x\text{-}init,$
   $trans = m1x\text{-}trans,$
   $obs = id$
  $|\!)$

lemmas *m1x-defs* =
  m1x-def m1x-init-def m1x-trans-def
  m1x-step1-def m1x-step2-def m1x-step3-def m1x-step4-def m1x-step5-def
  m1x-leak-def

lemma *m1x-obs-id* [*simp*]: $obs\ m1x = id$
⟨*proof*⟩

3.1.4 Invariants

inv1: Key definedness

Only run identifiers or static keys can be (concretely) known or authorized keys. (This reading corresponds to the contraposition of the property expressed below.)

definition
  $m1x\text{-}inv1\text{-}key :: m1x\text{-}state\ set$
where
  $m1x\text{-}inv1\text{-}key \equiv \{s.\ \forall Rs\ A.$
    $Rs \notin dom\ (runs\ s) \longrightarrow$
      $(sesK\ (Rs\$sk), A) \notin knC\ (runs\ s) \wedge$
      $(sesK\ (Rs\$sk), A) \notin azC\ (runs\ s) \wedge$
      $sesK\ (Rs\$sk) \notin leak\ s$
  $\}$

lemmas *m1x-inv1-keyI* = *m1x-inv1-key-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m1x-inv1-keyE* [*elim*] =
  m1x-inv1-key-def [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m1x-inv1-keyD* [*dest*] =
  m1x-inv1-key-def [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

Invariance proof.

lemma *PO-m1x-inv1-key-init* [*iff*]:
  init m1x ⊆ *m1x-inv1-key*
⟨*proof*⟩

lemma *PO-m1x-inv1-key-trans* [*iff*]:
  {*m1x-inv1-key*} *trans m1x* {> *m1x-inv1-key*}
⟨*proof*⟩

lemma *PO-m1x-inv1-key* [*iff*]: *reach m1x* ⊆ *m1x-inv1-key*
⟨*proof*⟩

3.1.5 Refinement of s0g

med10: The mediator function maps a concrete observation to an abstract one.

definition
  med01x :: *m1x-obs* ⇒ *key s0g-obs*
where
  med01x t ≡ (| *kn* = *knC* (*runs t*), *az* = *azC* (*runs t*), *lk* = *leak t* |)

R01: The simulation relation expreses key knowledge and authorization in terms of the client and server run information.

definition
  R01x :: (*key s0g-state* × *m1x-state*) *set* **where**
  R01x ≡ {(*s*, *t*). *s* = *med01x t*}

lemmas *R01x-defs* = *R01x-def med01x-def*

Refinement proof.

lemma *PO-m1x-step1-refines-skip*:
  {*R01x*}
    Id, (*m1x-step1 Ra A B*)
  {> *R01x*}
⟨*proof*⟩

lemma *PO-m1x-step2-refines-skip*:
  {*R01x*}
    Id, (*m1x-step2 Rb A B*)
  {> *R01x*}
⟨*proof*⟩

lemma *PO-m1x-step3-refines-s0g-gen*:
  {*R01x* ∩ *UNIV* × *m1x-inv1-key*}
    (*s0g-gen Kab Sv* {*Sv*, *A*, *B*}), (*m1x-step3 Rs A B Kab*)

{> *R01x*}
⟨*proof*⟩

lemma *PO-m1x-step4-refines-s0g-learn*:
{*R01x*}
(*s0g-learn Kab A*), (*m1x-step4 Ra A B Kab*)
{> *R01x*}
⟨*proof*⟩

lemma *PO-m1x-step5-refines-s0g-learn*:
{*R01x*}
(*s0g-learn Kab B*), (*m1x-step5 Rb A B Kab*)
{> *R01x*}
⟨*proof*⟩

lemma *PO-m1x-leak-refines-s0g-leak*:
{*R01x*}
(*s0g-leak* (*sesK* (*Rs$sk*))), (*m1x-leak Rs*)
{> *R01x*}
⟨*proof*⟩

All together now...

lemmas *PO-m1x-trans-refines-s0g-trans* =
PO-m1x-step1-refines-skip PO-m1x-step2-refines-skip
PO-m1x-step3-refines-s0g-gen PO-m1x-step4-refines-s0g-learn
PO-m1x-step5-refines-s0g-learn PO-m1x-leak-refines-s0g-leak

lemma *PO-m1x-refines-init-s0g* [*iff*]:
init m1x ⊆ *R01x*‘‘(*init s0g*)
⟨*proof*⟩

lemma *PO-m1x-refines-trans-s0g* [*iff*]:
{*R01x* ∩ *UNIV* × *m1x-inv1-key*}
(*trans s0g*), (*trans m1x*)
{> *R01x*}
⟨*proof*⟩

Observation consistency.

lemma *obs-consistent-med01x* [*iff*]:
obs-consistent R01x med01x s0g m1x
⟨*proof*⟩

Refinement result.

lemma *PO-m1x-refines-s0g* [*iff*]:
refines
(*R01x* ∩ *UNIV* × *m1x-inv1-key*)
med01x s0g m1x
⟨*proof*⟩

lemma *m1x-implements-s0g* [*iff*]: *implements med01x s0g m1x*
⟨*proof*⟩

3.1.6 Derived invariants

inv2: Secrecy

Secrecy, expressed in terms of runs.

definition
m1x-secrecy :: *'x m1x-pred*
where
$m1x\text{-}secrecy \equiv \{s.\ knC\ (runs\ s) \subseteq azC\ (runs\ s) \cup leak\ s \times UNIV\}$

lemmas *m1x-secrecyI* = *m1x-secrecy-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m1x-secrecyE* [*elim*] = *m1x-secrecy-def* [*THEN setc-def-to-elim*, *rule-format*]

Invariance proof.

lemma *PO-m1x-obs-secrecy* [*iff*]: *oreach m1x* ⊆ *m1x-secrecy*
⟨*proof*⟩

lemma *PO-m1x-secrecy* [*iff*]: *reach m1x* ⊆ *m1x-secrecy*
⟨*proof*⟩

end

3.2 Abstract (i/n)-authenticated key transport (L1)

theory *m1-keydist-iirn* **imports** *m1-keydist ../Refinement/a0i-agree*
begin

We add authentication for the initiator and responder to the basic server-based key transport protocol:

1. the initiator injectively agrees with the server on the key and some additional data
2. the responder non-injectively agrees with the server on the key and some additional data.

The "additional data" is a parameter of this model.

declare *option.split* [*split*]

consts
na :: *nat*

3.2.1 State

The state type remains the same, but in this model we will record nonces and timestamps in the run frame.

type-synonym *m1a-state* = *m1x-state*
type-synonym *m1a-obs* = *m1x-obs*

type-synonym *'x m1a-pred* = *'x m1x-pred*

type-synonym *'x m1a-trans* = *'x m1x-trans*

We need some parameters regarding the list of freshness values stored by the server. These should be defined in further refinements.

consts
is-len :: *nat* — num of agreeing list elements for initiator-server
rs-len :: *nat* — num of agreeing list elements for responder-server

3.2.2 Events

definition — by *A*, refines *m1x-step1*
m1a-step1 :: [*rid-t*, *agent*, *agent*, *nonce*] ⇒ *'x m1r-trans*
where
m1a-step1 Ra A B Na ≡ {(*s*, *s1*).
— guards:
Ra ∉ *dom* (*runs s*) ∧ — *Ra* is fresh
Na = *Ra*$*na* ∧ — NEW: generate a nonce

— actions:
— create initiator thread
s1 = *s*⦇ *runs* := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [])) ⦈
}

definition — by *B*, refines *m1x-step2*
m1a-step2 :: [*rid-t*, *agent*, *agent*] ⇒ *'x m1r-trans*
where
m1a-step2 ≡ *m1x-step2*

definition — by *Sv*, refines *m1x-step3*
m1a-step3 :: [*rid-t*, *agent*, *agent*, *key*, *nonce*, *atom list*] ⇒ *'x m1r-trans*
where
m1a-step3 Rs A B Kab Na al ≡ {(*s*, *s1*).
— guards:
Rs ∉ *dom* (*runs s*) ∧ — fresh run id
Kab = *sesK* (*Rs*$*sk*) ∧ — generate session key

— actions:
s1 = *s*⦇ *runs* := (*runs s*)(*Rs* ↦ (*Serv*, [*A*, *B*], *aNon Na* # *al*)) ⦈
}

definition — by *A*, refines *m1x-step4*
m1a-step4 :: [*rid-t*, *agent*, *agent*, *nonce*, *key*, *atom list*] ⇒ *'x m1a-trans*
where
m1a-step4 Ra A B Na Kab nla ≡ {(*s*, *s'*).
— guards:
runs s Ra = *Some* (*Init*, [*A*, *B*], []) ∧
(*Kab* ∉ *leak s* ⟶ (*Kab*, *A*) ∈ *azC* (*runs s*)) ∧ — authorization guard
Na = *Ra*$*na* ∧ — fix parameter

— new guard for agreement with server on (*Kab*, *B*, *Na*, *isl*),
— where *isl* = *take is-len nla*; injectiveness by including *Na*
(*A* ∉ *bad* ⟶ (∃ *Rs*. *Kab* = *sesK* (*Rs*$*sk*) ∧
runs s Rs = *Some* (*Serv*, [*A*, *B*], *aNon Na* # *take is-len nla*))) ∧

— actions:
$s' = s(\| runs := (runs\ s)(Ra \mapsto (Init, [A, B], aKey\ Kab\ \#\ nla)) \|)$
}

definition — by B, refines *m1x-step5*
m1a-step5 :: [*rid-t*, *agent*, *agent*, *key*, *atom list*] ⇒ *'x m1a-trans*
where
m1a-step5 Rb A B Kab nlb ≡ {(*s*, *s1*).
— guards:
$runs\ s\ Rb = Some\ (Resp, [A, B], []) \wedge$
$(Kab \notin leak\ s \longrightarrow (Kab, B) \in azC\ (runs\ s)) \wedge$ — authorization guard

— guard for showing agreement with server on (*Kab*, *A*, *rsl*),
— where *rsl* = *take rs-len nlb*; this agreement is non-injective

$(B \notin bad \longrightarrow (\exists Rs\ Na.\ Kab = sesK\ (Rs\$sk) \wedge$
$runs\ s\ Rs = Some\ (Serv, [A, B], aNon\ Na\ \#\ take\ rs\text{-}len\ nlb))) \wedge$

— actions:
$s1 = s(\| runs := (runs\ s)(Rb \mapsto (Resp, [A, B], aKey\ Kab\ \#\ nlb)) \|)$
}

definition — by attacker, refines *m1x-leak*
m1a-leak :: *rid-t* ⇒ *'x m1x-trans*
where
m1a-leak = *m1x-leak*

3.2.3 Specification

definition
m1a-init :: *m1a-state set*
where
m1a-init ≡ *m1x-init*

definition
m1a-trans :: *'x m1a-trans* **where**
m1a-trans ≡ (⋃ *A B Ra Rb Rs Na Kab nls nla nlb*.
m1a-step1 Ra A B Na ∪
m1a-step2 Rb A B ∪
m1a-step3 Rs A B Kab Na nls ∪
m1a-step4 Ra A B Na Kab nla ∪
m1a-step5 Rb A B Kab nlb ∪
m1a-leak Rs ∪
Id
)

definition
m1a :: (*m1a-state*, *m1a-obs*) *spec* **where**
m1a ≡ (|
init = *m1a-init*,
trans = *m1a-trans*,
obs = *id*

⦈

lemma *init-m1a*: *init m1a* = *m1a-init*
⟨*proof*⟩

lemma *trans-m1a*: *trans m1a* = *m1a-trans*
⟨*proof*⟩

lemma *obs-m1a* [*simp*]: *obs m1a* = *id*
⟨*proof*⟩

lemmas *m1a-loc-defs* =
m1a-def m1a-init-def m1a-trans-def
m1a-step1-def m1a-step2-def m1a-step3-def m1a-step4-def m1a-step5-def
m1a-leak-def

lemmas *m1a-defs* = *m1a-loc-defs m1x-defs*

3.2.4 Invariants

inv0: Finite domain

There are only finitely many runs. This is needed to establish the responder/initiator agreement.

definition
m1a-inv0-fin :: *'x m1r-pred*
where
m1a-inv0-fin ≡ {*s. finite* (*dom* (*runs s*))}

lemmas *m1a-inv0-finI* = *m1a-inv0-fin-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m1a-inv0-finE* [*elim*] = *m1a-inv0-fin-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m1a-inv0-finD* = *m1a-inv0-fin-def* [*THEN setc-def-to-dest*, *rule-format*]

Invariance proof.

lemma *PO-m1a-inv0-fin-init* [*iff*]:
init m1a ⊆ *m1a-inv0-fin*
⟨*proof*⟩

lemma *PO-m1a-inv0-fin-trans* [*iff*]:
{*m1a-inv0-fin*} *trans m1a* {> *m1a-inv0-fin*}
⟨*proof*⟩

lemma *PO-m1a-inv0-fin* [*iff*]: *reach m1a* ⊆ *m1a-inv0-fin*
⟨*proof*⟩

3.2.5 Refinement of *m1x*

Simulation relation

Define run abstraction.

fun
rm1x1a :: *role-t* ⇒ *atom list* ⇒ *atom list*

where
  rm1x1a Init = take 1 — take *Kab* from *Kab* # *nla*
| *rm1x1a Resp = take 1* — take *Kab* from *Kab* # *nlb*
| *rm1x1a Serv = take 0* — drop all from [*Na*]

abbreviation
  runs1x1a :: *runs-t* ⇒ *runs-t* **where**
  runs1x1a ≡ *map-runs rm1x1a*

med1x1: The mediator function maps a concrete observation to an abstract one.

definition
  med1x1a :: *m1a-obs* ⇒ *m1x-obs* **where**
  med1x1a t ≡ (| *runs* = *runs1x1a* (*runs t*), *leak* = *leak t* |)

R1x1a: The simulation relation is defined in terms of the mediator function.

definition
  R1x1a :: (*m1x-state* × *m1a-state*) *set* **where**
  R1x1a ≡ {(*s*, *t*). *s* = *med1x1a t*}

lemmas *R1x1a-defs* =
  R1x1a-def med1x1a-def

Refinement proof

lemma *PO-m1a-step1-refines-m1x-step1*:
  {*R1x1a*}
    (*m1x-step1 Ra A B*), (*m1a-step1 Ra A B Na*)
  {> *R1x1a*}
⟨*proof*⟩

lemma *PO-m1a-step2-refines-m1x-step2*:
  {*R1x1a*}
    (*m1x-step2 Rb A B*), (*m1a-step2 Rb A B*)
  {> *R1x1a*}
⟨*proof*⟩

lemma *PO-m1a-step3-refines-m1x-step3*:
  {*R1x1a*}
    (*m1x-step3 Rs A B Kab*), (*m1a-step3 Rs A B Kab Na nls*)
  {> *R1x1a*}
⟨*proof*⟩

lemma *PO-m1a-step4-refines-m1x-step4*:
  {*R1x1a*}
    (*m1x-step4 Ra A B Kab*), (*m1a-step4 Ra A B Na Kab nla*)
  {> *R1x1a*}
⟨*proof*⟩

lemma *PO-m1a-step5-refines-m1x-step5*:
  {*R1x1a*}
    (*m1x-step5 A B Rb Kab*), (*m1a-step5 A B Rb Kab nlb*)
  {> *R1x1a*}
⟨*proof*⟩

lemma *PO-m1a-leak-refines-m1x-leak*:
{*R1x1a*}
(*m1x-leak Rs*), (*m1a-leak Rs*)
{> *R1x1a*}
⟨*proof*⟩

All together now...

lemmas *PO-m1a-trans-refines-m1x-trans* =
PO-m1a-step1-refines-m1x-step1 PO-m1a-step2-refines-m1x-step2
PO-m1a-step3-refines-m1x-step3 PO-m1a-step4-refines-m1x-step4
PO-m1a-step5-refines-m1x-step5 PO-m1a-leak-refines-m1x-leak

lemma *PO-m1a-refines-init-m1x* [*iff*]:
init m1a ⊆ *R1x1a*``(*init m1x*)
⟨*proof*⟩

lemma *PO-m1a-refines-trans-m1x* [*iff*]:
{*R1x1a*}
(*trans m1x*), (*trans m1a*)
{> *R1x1a*}
⟨*proof*⟩

Observation consistency.

lemma *obs-consistent-med1x1a* [*iff*]:
obs-consistent R1x1a med1x1a m1x m1a
⟨*proof*⟩

Refinement result.

lemma *PO-m1a-refines-m1x* [*iff*]:
refines R1x1a med1x1a m1x m1a
⟨*proof*⟩

lemma *m1a-implements-m1x* [*iff*]: *implements med1x1a m1x m1a*
⟨*proof*⟩

By transitivity:

lemma *m1a-implements-s0g* [*iff*]: *implements* (*med01x o med1x1a*) *s0g m1a*
⟨*proof*⟩

inv (inherited): Secrecy

Secrecy preserved from *m1x*.

lemma *knC-runs1x1a* [*simp*]: *knC* (*runs1x1a runz*) = *knC runz*
⟨*proof*⟩

lemma *PO-m1a-obs-secrecy* [*iff*]: *oreach m1a* ⊆ *m1x-secrecy*
⟨*proof*⟩

lemma *PO-m1a-secrecy* [*iff*]: *reach m1a* ⊆ *m1x-secrecy*
⟨*proof*⟩

3.2.6 Refinement of *a0i* for initiator/server

For the initiator, we get an injective agreement with the server on the session key, the responder name, the initiator's nonce and the list of freshness values *isl*.

Simulation relation

We define two auxiliary functions to reconstruct the signals of the initial model from completed initiator and server runs.

type-synonym
issig = *key* × *agent* × *nonce* × *atom list*

fun
is-runs2sigs :: *runs-t* ⇒ *issig signal* ⇒ *nat*
where
is-runs2sigs runz (*Running* [*A*, *Sv*] (*Kab*, *B*, *Na*, *nl*)) =
  (*if* ∃ *Rs*. *Kab* = *sesK* (*Rs*$*sk*) ∧
    runz Rs = *Some* (*Serv*, [*A*, *B*], *aNon Na* # *nl*)
   then 1 else 0)

| *is-runs2sigs runz* (*Commit* [*A*, *Sv*] (*Kab*, *B*, *Na*, *nl*)) =
  (*if* ∃ *Ra nla*. *Na* = *Ra*$*na* ∧
    runz Ra = *Some* (*Init*, [*A*, *B*], *aKey Kab* # *nla*) ∧
    take is-len nla = *nl*
   then 1 else 0)

| *is-runs2sigs runz* - = *0*

Simulation relation and mediator function. We map completed initiator and responder runs to commit and running signals, respectively.

definition
med-a0m1a-is :: *m1a-obs* ⇒ *issig a0i-obs* **where**
med-a0m1a-is o1 ≡ (| *signals* = *is-runs2sigs* (*runs o1*), *corrupted* = {} |)

definition
R-a0m1a-is :: (*issig a0i-state* × *m1a-state*) *set* **where**
R-a0m1a-is ≡ {(*s*, *t*). *signals s* = *is-runs2sigs* (*runs t*) ∧ *corrupted s* = {} }

lemmas *R-a0m1a-is-defs* = *R-a0m1a-is-def med-a0m1a-is-def*

Lemmas about the auxiliary functions

lemma *is-runs2sigs-empty* [*simp*]:
runz = *Map.empty* ⟹ *is-runs2sigs runz* = (λ*s*. *0*)
⟨*proof*⟩

Update lemmas

lemma *is-runs2sigs-upd-init-none* [*simp*]:
⟦ *Ra* ∉ *dom runz* ⟧
⟹ *is-runs2sigs* (*runz*(*Ra* ↦ (*Init*, [*A*, *B*], []))) = *is-runs2sigs runz*
⟨*proof*⟩

lemma *is-runs2sigs-upd-resp-none* [*simp*]:
  ⟦ *Rb* ∉ *dom runz* ⟧
  ⟹ *is-runs2sigs* (*runz*(*Rb* ↦ (*Resp*, [*A*, *B*], []))) = *is-runs2sigs runz*
⟨*proof*⟩

lemma *is-runs2sigs-upd-serv* [*simp*]:
  ⟦ *Rs* ∉ *dom runz* ⟧
  ⟹ *is-runs2sigs* (*runz*(*Rs* ↦ (*Serv*, [*A*, *B*], *aNon Na* # *ils*))) =
    (*is-runs2sigs runz*)(*Running* [*A*, *Sv*] (*sesK* (*Rs*$*sk*), *B*, *Na*, *ils*) := *1*)
⟨*proof*⟩

lemma *is-runs2sigs-upd-init-some* [*simp*]:
  ⟦ *runz Ra* = *Some* (*Init*, [*A*, *B*], []); *ils* = *take is-len nla* ⟧
  ⟹ *is-runs2sigs* (*runz*(*Ra* ↦ (*Init*, [*A*, *B*], *aKey Kab* # *nla*))) =
    (*is-runs2sigs runz*)(*Commit* [*A*, *Sv*] (*Kab*, *B*, *Ra*$*na*, *ils*) := *1*)
⟨*proof*⟩

lemma *is-runs2sigs-upd-resp-some* [*simp*]:
  ⟦ *runz Rb* = *Some* (*Resp*, [*A*, *B*], []) ⟧
  ⟹ *is-runs2sigs* (*runz*(*Rb* ↦ (*Resp*, [*A*, *B*], *aKey Kab* # *nlb*))) =
    is-runs2sigs runz
⟨*proof*⟩

Refinement proof

lemma *PO-m1a-step1-refines-a0-is-skip*:
  {*R-a0m1a-is*}
    Id, (*m1a-step1 Ra A B Na*)
  {> *R-a0m1a-is*}
⟨*proof*⟩

lemma *PO-m1a-step2-refines-a0-is-skip*:
  {*R-a0m1a-is*}
    Id, (*m1a-step2 Rb A B*)
  {> *R-a0m1a-is*}
⟨*proof*⟩

lemma *PO-m1a-step3-refines-a0-is-running*:
  {*R-a0m1a-is*}
    (*a0i-running* [*A*, *Sv*] (*Kab*, *B*, *Na*, *nls*)),
    (*m1a-step3 Rs A B Kab Na nls*)
  {> *R-a0m1a-is*}
⟨*proof*⟩

lemma *PO-m1a-step4-refines-a0-is-commit*:
  {*R-a0m1a-is* ∩ *UNIV* × *m1a-inv0-fin*}
    (*a0i-commit* [*A*, *Sv*] (*Kab*, *B*, *Na*, *take is-len nla*)),
    (*m1a-step4 Ra A B Na Kab nla*)
  {> *R-a0m1a-is*}
⟨*proof*⟩

lemma *PO-m1a-step5-refines-a0-is-skip*:

{*R-a0m1a-is*}
  Id, (*m1a-step5 A B Rb Kab nlb*)
 {> *R-a0m1a-is*}
⟨*proof*⟩

lemma *PO-m1a-leak-refines-a0-is-skip*:
 {*R-a0m1a-is*}
  Id, (*m1a-leak Rs*)
 {> *R-a0m1a-is*}
⟨*proof*⟩

All together now...

lemmas *PO-m1a-trans-refines-a0-is-trans* =
 PO-m1a-step1-refines-a0-is-skip PO-m1a-step2-refines-a0-is-skip
 PO-m1a-step3-refines-a0-is-running PO-m1a-step4-refines-a0-is-commit
 PO-m1a-step5-refines-a0-is-skip PO-m1a-leak-refines-a0-is-skip

lemma *PO-m1a-refines-init-a0-is* [*iff*]:
 init m1a ⊆ *R-a0m1a-is*``(*init a0i*)
⟨*proof*⟩

lemma *PO-m1a-refines-trans-a0-is* [*iff*]:
 {*R-a0m1a-is* ∩ *a0i-inv1-iagree* × *m1a-inv0-fin*}
  (*trans a0i*), (*trans m1a*)
 {> *R-a0m1a-is*}
⟨*proof*⟩

lemma *obs-consistent-med-a0m1a-is* [*iff*]:
 obs-consistent R-a0m1a-is med-a0m1a-is a0i m1a
⟨*proof*⟩

Refinement result.

lemma *PO-m1a-refines-a0-is* [*iff*]:
 refines (*R-a0m1a-is* ∩ *a0i-inv1-iagree* × *m1a-inv0-fin*) *med-a0m1a-is a0i m1a*
⟨*proof*⟩

lemma *m1a-implements-a0-is*: *implements med-a0m1a-is a0i m1a*
⟨*proof*⟩

inv2i (inherited): Initiator and server

This is a translation of the agreement property to Level 1. It follows from the refinement and is needed to prove inv1.

definition
 m1a-inv2i-serv :: *'x m1x-state-scheme set*
where
 m1a-inv2i-serv ≡ {*s*. ∀ *A B Ra Kab nla*.
  A ∉ *bad* ⟶
  runs s Ra = *Some* (*Init*, [*A*, *B*], *aKey Kab* # *nla*) ⟶
   (∃ *Rs*. *Kab* = *sesK* (*Rs*$*sk*) ∧
    runs s Rs = *Some* (*Serv*, [*A*, *B*], *aNon* (*Ra*$*na*) # *take is-len nla*))

}

lemmas *m1a-inv2i-servI* =
m1a-inv2i-serv-def [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m1a-inv2i-servE* =
m1a-inv2i-serv-def [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m1a-inv2i-servD* =
m1a-inv2i-serv-def [*THEN setc-def-to-dest*, *rule-format*, *rotated* −*1*]

Invariance proof, see below after init/serv authentication proof.

lemma *PO-m1a-inv2i-serv* [*iff*]:
reach m1a ⊆ *m1a-inv2i-serv*
⟨*proof*⟩

inv1: Key freshness for initiator

The initiator obtains key freshness from the injective agreement with the server AND the fact that there is only one server run with a given key.

definition
m1a-inv1-ifresh :: *'x m1a-pred*
where
m1a-inv1-ifresh ≡ {*s*. ∀ *A A' B B' Ra Ra' Kab nl nl'*.
runs s Ra = *Some* (*Init*, [*A*, *B*], *aKey Kab* # *nl*) ⟶
runs s Ra' = *Some* (*Init*, [*A'*, *B'*], *aKey Kab* # *nl'*) ⟶
A ∉ *bad* ⟶ *B* ∉ *bad* ⟶ *Kab* ∉ *leak s* ⟶
Ra = *Ra'*
}

lemmas *m1a-inv1-ifreshI* = *m1a-inv1-ifresh-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m1a-inv1-ifreshE* [*elim*] = *m1a-inv1-ifresh-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m1a-inv1-ifreshD* = *m1a-inv1-ifresh-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

Invariance proof

lemma *PO-m1a-inv1-ifresh-init* [*iff*]:
init m1a ⊆ *m1a-inv1-ifresh*
⟨*proof*⟩

lemma *PO-m1a-inv1-ifresh-step4*:
{*m1a-inv1-ifresh* ∩ *m1a-inv2i-serv* ∩ *m1x-secrecy*}
m1a-step4 Ra A B Na Kab nla
{> *m1a-inv1-ifresh*}
⟨*proof*⟩

lemma *PO-m1a-inv1-ifresh-trans* [*iff*]:
{*m1a-inv1-ifresh* ∩ *m1a-inv2i-serv* ∩ *m1x-secrecy*} *trans m1a* {> *m1a-inv1-ifresh*}
⟨*proof*⟩

lemma *PO-m1a-inv1-ifresh* [*iff*]: *reach m1a* ⊆ *m1a-inv1-ifresh*
⟨*proof*⟩

3.2.7 Refinement of *a0n* for responder/server

For the responder, we get a non-injective agreement with the server on the session key, the initiator's name, and additional data.

Simulation relation

We define two auxiliary functions to reconstruct the signals of the initial model from completed responder and server runs.

type-synonym
rssig = key × agent × atom list

abbreviation
rs-commit :: [*runs-t*, *agent*, *agent*, *key*, *atom list*] ⇒ *rid-t set*
where
rs-commit runz A B Kab rsl ≡ {*Rb*. ∃ *nlb*.
runz Rb = *Some* (*Resp*, [*A*, *B*], *aKey Kab* # *nlb*) ∧ *take rs-len nlb* = *rsl*
}

fun
rs-runs2sigs :: *runs-t* ⇒ *rssig signal* ⇒ *nat*
where
rs-runs2sigs runz (*Running* [*B*, *Sv*] (*Kab*, *A*, *rsl*)) =
(*if* (∃ *Rs Na*. *Kab* = *sesK* (*Rs$sk*) ∧
runz Rs = *Some* (*Serv*, [*A*, *B*], *aNon Na* # *rsl*))
then 1 else 0)

| *rs-runs2sigs runz* (*Commit* [*B*, *Sv*] (*Kab*, *A*, *rsl*)) =
card (*rs-commit runz A B Kab rsl*)

| *rs-runs2sigs runz - = 0*

Simulation relation and mediator function. We map completed initiator and responder runs to commit and running signals, respectively.

definition
med-a0m1a-rs :: *m1a-obs* ⇒ *rssig a0n-obs* **where**
med-a0m1a-rs o1 ≡ ⦇ *signals* = *rs-runs2sigs* (*runs o1*), *corrupted* = {} ⦈

definition
R-a0m1a-rs :: (*rssig a0n-state* × *m1a-state*) *set* **where**
R-a0m1a-rs ≡ {(*s*, *t*). *signals s* = *rs-runs2sigs* (*runs t*) ∧ *corrupted s* = {} }

lemmas *R-a0m1a-rs-defs* = *R-a0m1a-rs-def med-a0m1a-rs-def*

Lemmas about the auxiliary functions

Other lemmas

lemma *rs-runs2sigs-empty* [*simp*]:
runz = *Map.empty* ⟹ *rs-runs2sigs runz* = (λ*s*. *0*)
⟨*proof*⟩

lemma *rs-commit-finite* [*simp*, *intro*]:
finite (*dom runz*) $\Longrightarrow$ *finite* (*rs-commit runz A B Kab nls*)
⟨*proof*⟩

Update lemmas

lemma *rs-runs2sigs-upd-init-none* [*simp*]:
⟦ *Ra* $\notin$ *dom runz* ⟧
$\Longrightarrow$ *rs-runs2sigs* (*runz*(*Ra* $\mapsto$ (*Init*, [*A*, *B*], []))) = *rs-runs2sigs runz*
⟨*proof*⟩

lemma *rs-runs2sigs-upd-resp-none* [*simp*]:
⟦ *Rb* $\notin$ *dom runz* ⟧
$\Longrightarrow$ *rs-runs2sigs* (*runz*(*Rb* $\mapsto$ (*Resp*, [*A*, *B*], []))) = *rs-runs2sigs runz*
⟨*proof*⟩

lemma *rs-runs2sigs-upd-serv* [*simp*]:
⟦ *Rs* $\notin$ *dom runz* ⟧
$\Longrightarrow$ *rs-runs2sigs* (*runz*(*Rs* $\mapsto$ (*Serv*, [*A*, *B*], *aNon Na* # *nls*))) =
(*rs-runs2sigs runz*)(*Running* [*B*, *Sv*] (*sesK* (*Rs*$*sk*), *A*, *nls*) := *1*)
⟨*proof*⟩

lemma *rs-runs2sigs-upd-init-some* [*simp*]:
⟦ *runz Ra* = *Some* (*Init*, [*A*, *B*], []) ⟧
$\Longrightarrow$ *rs-runs2sigs* (*runz*(*Ra* $\mapsto$ (*Init*, [*A*, *B*], *aKey Kab* # *nl*))) =
rs-runs2sigs runz
⟨*proof*⟩

lemma *rs-runs2sigs-upd-resp-some* [*simp*]:
⟦ *runz Rb* = *Some* (*Resp*, [*A*, *B*], []); *finite* (*dom runz*);
rsl = *take rs-len nlb* ⟧
$\Longrightarrow$ *rs-runs2sigs* (*runz*(*Rb* $\mapsto$ (*Resp*, [*A*, *B*], *aKey Kab* # *nlb*))) =
(*rs-runs2sigs runz*)(
Commit [*B*, *Sv*] (*Kab*, *A*, *rsl*) := *Suc* (*card* (*rs-commit runz A B Kab rsl*)))
⟨*proof*⟩

Refinement proof

lemma *PO-m1a-step1-refines-a0-rs-skip*:
{*R-a0m1a-rs*}
Id, (*m1a-step1 Ra A B Na*)
{> *R-a0m1a-rs*}
⟨*proof*⟩

lemma *PO-m1a-step2-refines-a0-rs-skip*:
{*R-a0m1a-rs*}
Id, (*m1a-step2 Rb A B*)
{> *R-a0m1a-rs*}
⟨*proof*⟩

lemma *PO-m1a-step3-refines-a0-rs-running*:
{*R-a0m1a-rs*}
(*a0n-running* [*B*, *Sv*] (*Kab*, *A*, *nls*)),
(*m1a-step3 Rs A B Kab Na nls*)

{> *R-a0m1a-rs*}
⟨*proof*⟩

lemma *PO-m1a-step4-refines-a0-rs-skip*:
{*R-a0m1a-rs*}
Id, (*m1a-step4 Ra A B Na Kab nla*)
{> *R-a0m1a-rs*}
⟨*proof*⟩

lemma *PO-m1a-step5-refines-a0-rs-commit*:
{*R-a0m1a-rs* ∩ *UNIV* × *m1a-inv0-fin*}
(*a0n-commit* [*B*, *Sv*] (*Kab*, *A*, *take rs-len nlb*)),
(*m1a-step5 Rb A B Kab nlb*)
{> *R-a0m1a-rs*}
⟨*proof*⟩

lemma *PO-m1a-leak-refines-a0-rs-skip*:
{*R-a0m1a-rs*}
Id, (*m1a-leak Rs*)
{> *R-a0m1a-rs*}
⟨*proof*⟩

All together now...

lemmas *PO-m1a-trans-refines-a0-rs-trans* =
PO-m1a-step1-refines-a0-rs-skip PO-m1a-step2-refines-a0-rs-skip
PO-m1a-step3-refines-a0-rs-running PO-m1a-step4-refines-a0-rs-skip
PO-m1a-step5-refines-a0-rs-commit PO-m1a-leak-refines-a0-rs-skip

lemma *PO-m1a-refines-init-ra0n* [*iff*]:
init m1a ⊆ *R-a0m1a-rs*‘‘(*init a0n*)
⟨*proof*⟩

lemma *PO-m1a-refines-trans-ra0n* [*iff*]:
{*R-a0m1a-rs* ∩ *a0n-inv1-niagree* × *m1a-inv0-fin*}
(*trans a0n*), (*trans m1a*)
{> *R-a0m1a-rs*}
⟨*proof*⟩

lemma *obs-consistent-med-a0m1a-rs* [*iff*]:
obs-consistent
(*R-a0m1a-rs* ∩ *a0n-inv1-niagree* × *m1a-inv0-fin*)
med-a0m1a-rs a0n m1a
⟨*proof*⟩

Refinement result.

lemma *PO-m1a-refines-a0-rs* [*iff*]:
refines (*R-a0m1a-rs* ∩ *a0n-inv1-niagree* × *m1a-inv0-fin*) *med-a0m1a-rs a0n m1a*
⟨*proof*⟩

lemma *m1a-implements-ra0n*: *implements med-a0m1a-rs a0n m1a*
⟨*proof*⟩

inv2r (inherited): Responder and server

This is a translation of the agreement property to Level 1. It follows from the refinement and not needed here but later.

definition
m1a-inv2r-serv :: *'x m1x-state-scheme set*
where
m1a-inv2r-serv ≡ {*s*. ∀ *A B Rb Kab nlb*.
B ∉ *bad* ⟶
runs s Rb = *Some* (*Resp*, [*A*, *B*], *aKey Kab* # *nlb*) ⟶
(∃ *Rs Na*. *Kab* = *sesK* (*Rs$sk*) ∧
runs s Rs = *Some* (*Serv*, [*A*, *B*], *aNon Na* # *take rs-len nlb*))
}

lemmas *m1a-inv2r-servI* =
m1a-inv2r-serv-def [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m1a-inv2r-servE* [*elim*] =
m1a-inv2r-serv-def [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m1a-inv2r-servD* =
m1a-inv2r-serv-def [*THEN setc-def-to-dest*, *rule-format*, *rotated* −*1*]

Invariance proof

lemma *PO-m1a-inv2r-serv* [*iff*]:
reach m1a ⊆ *m1a-inv2r-serv*
⟨*proof*⟩

end

3.3 Abstract (n/n)-authenticated key transport (L1)

theory *m1-keydist-inrn* **imports** *m1-keydist ../Refinement/a0i-agree*
begin

We add authentication for the initiator and responder to the basic server-based key transport protocol:

1. the initiator injectively agrees with the server on the key and some additional data
2. the responder non-injectively agrees with the server on the key and some additional data.

The "additional data" is a parameter of this model.

declare *option.split* [*split*]

3.3.1 State

The state type remains the same, but in this model we will record nonces and timestamps in the run frame.

type-synonym *m1a-state* = *m1x-state*

type-synonym *m1a-obs* = *m1x-obs*

type-synonym *'x m1a-pred* = *'x m1x-pred*
type-synonym *'x m1a-trans* = *'x m1x-trans*

We need some parameters regarding the list of freshness values stored by the server. These should be defined in further refinements.

consts
  is-len :: *nat* — num of agreeing list elements for initiator-server
  rs-len :: *nat* — num of agreeing list elements for responder-server

3.3.2 Events

definition — by A, refines *m1x-step1*
  m1a-step1 :: [*rid-t*, *agent*, *agent*] ⇒ *'x m1r-trans*
where
  m1a-step1 ≡ *m1x-step1*

definition — by B, refines *m1x-step2*
  m1a-step2 :: [*rid-t*, *agent*, *agent*] ⇒ *'x m1r-trans*
where
  m1a-step2 ≡ *m1x-step2*

definition — by *Sv*, refines *m1x-step3*
  m1a-step3 :: [*rid-t*, *agent*, *agent*, *key*, *atom list*] ⇒ *'x m1r-trans*
where
  m1a-step3 Rs A B Kab al ≡ {(*s*, *s1*).
    — guards:
    Rs ∉ *dom* (*runs s*) ∧ — fresh run id
    Kab = *sesK* (*Rs*$*sk*) ∧ — generate session key

    — actions:
    s1 = *s*⦇ *runs* := (*runs s*)(*Rs* ↦ (*Serv*, [*A*, *B*], *al*)) ⦈
  }

definition — by A, refines *m1x-step4*
  m1a-step4 :: [*rid-t*, *agent*, *agent*, *key*, *atom list*] ⇒ *'x m1a-trans*
where
  m1a-step4 Ra A B Kab nla ≡ {(*s*, *s'*).
    — guards:
    runs s Ra = *Some* (*Init*, [*A*, *B*], []) ∧
    (*Kab* ∉ *leak s* ⟶ (*Kab*, *A*) ∈ *azC* (*runs s*)) ∧ — authorization guard

    — new guard for non-injective agreement with server on (*Kab*, *B*, *isl*),
    — where *isl* = *take is-len nla*
    (*A* ∉ *bad* ⟶ (∃ *Rs*. *Kab* = *sesK* (*Rs*$*sk*) ∧
      runs s Rs = *Some* (*Serv*, [*A*, *B*], *take is-len nla*))) ∧

    — actions:
    s' = *s*⦇ *runs* := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], *aKey Kab* # *nla*)) ⦈
  }

definition — by B, refines *m1x-step5*

m1a-step5 :: [*rid-t*, *agent*, *agent*, *key*, *atom list*] ⇒ *'x m1a-trans*
where
m1a-step5 Rb A B Kab nlb ≡ {(*s*, *s1*).
— guards:
runs s Rb = *Some* (*Resp*, [*A*, *B*], []) ∧
(*Kab* ∉ *leak s* ⟶ (*Kab*, *B*) ∈ *azC* (*runs s*)) ∧ — authorization guard

— guard for non-injective agreement with server on (*Kab*, *A*, *rsl*)
— where *rsl* = *take rs-len nlb*
(*B* ∉ *bad* ⟶ (∃ *Rs*. *Kab* = *sesK* (*Rs*$*sk*) ∧
runs s Rs = *Some* (*Serv*, [*A*, *B*], *take rs-len nlb*))) ∧

— actions:
s1 = *s*(| *runs* := (*runs s*)(*Rb* ↦ (*Resp*, [*A*, *B*], *aKey Kab* # *nlb*)) |)
}

definition — by attacker, refines *m1x-leak*
m1a-leak :: *rid-t* ⇒ *'x m1x-trans*
where
m1a-leak = *m1x-leak*

3.3.3 Specification

definition
m1a-init :: *m1a-state set*
where
m1a-init ≡ *m1x-init*

definition
m1a-trans :: *'x m1a-trans* **where**
m1a-trans ≡ (⋃ *A B Ra Rb Rs Kab nls nla nlb*.
m1a-step1 Ra A B ∪
m1a-step2 Rb A B ∪
m1a-step3 Rs A B Kab nls ∪
m1a-step4 Ra A B Kab nla ∪
m1a-step5 Rb A B Kab nlb ∪
m1a-leak Rs ∪
Id
)

definition
m1a :: (*m1a-state*, *m1a-obs*) *spec* **where**
m1a ≡ (|
init = *m1a-init*,
trans = *m1a-trans*,
obs = *id*
|)

lemma *init-m1a*: *init m1a* = *m1a-init*
⟨*proof*⟩

lemma *trans-m1a*: *trans m1a* = *m1a-trans*
⟨*proof*⟩

lemma *obs-m1a* [*simp*]: *obs m1a* = *id*
⟨*proof*⟩

lemmas *m1a-loc-defs* =
m1a-def m1a-init-def m1a-trans-def
m1a-step1-def m1a-step2-def m1a-step3-def m1a-step4-def m1a-step5-def
m1a-leak-def

lemmas *m1a-defs* = *m1a-loc-defs m1x-defs*

3.3.4 Invariants

inv0: Finite domain

There are only finitely many runs. This is needed to establish the responder/initiator agreement.

definition
m1a-inv0-fin :: *'x m1r-pred*
where
m1a-inv0-fin ≡ {*s. finite* (*dom* (*runs s*))}

lemmas *m1a-inv0-finI* = *m1a-inv0-fin-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m1a-inv0-finE* [*elim*] = *m1a-inv0-fin-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m1a-inv0-finD* = *m1a-inv0-fin-def* [*THEN setc-def-to-dest*, *rule-format*]

Invariance proof.

lemma *PO-m1a-inv0-fin-init* [*iff*]:
init m1a ⊆ *m1a-inv0-fin*
⟨*proof*⟩

lemma *PO-m1a-inv0-fin-trans* [*iff*]:
{*m1a-inv0-fin*} *trans m1a* {> *m1a-inv0-fin*}
⟨*proof*⟩

lemma *PO-m1a-inv0-fin* [*iff*]: *reach m1a* ⊆ *m1a-inv0-fin*
⟨*proof*⟩

3.3.5 Refinement of *m1x*

Simulation relation

Define run abstraction.

fun
rm1x1a :: *role-t* ⇒ *atom list* ⇒ *atom list*
where
rm1x1a Init = *take 1* — take *Kab* from *Kab* # *nla*
| *rm1x1a Resp* = *take 1* — take *Kab* from *Kab* # *nlb*
| *rm1x1a Serv* = *take 0* — drop all from *nls*

abbreviation
runs1x1a :: *runs-t* ⇒ *runs-t* **where**

runs1x1a ≡ *map-runs rm1x1a*

med1x1: The mediator function maps a concrete observation to an abstract one.

definition
med1x1a :: *m1a-obs* ⇒ *m1x-obs* **where**
med1x1a t ≡ (| *runs* = *runs1x1a* (*runs t*), *leak* = *leak t* |)

R1x1a: The simulation relation is defined in terms of the mediator function.

definition
R1x1a :: (*m1x-state* × *m1a-state*) *set* **where**
R1x1a ≡ {(*s*, *t*). *s* = *med1x1a t*}

lemmas *R1x1a-defs* =
R1x1a-def med1x1a-def

Refinement proof

lemma *PO-m1a-step1-refines-m1x-step1*:
{*R1x1a*}
(*m1x-step1 Ra A B*), (*m1a-step1 Ra A B*)
{> *R1x1a*}
⟨*proof*⟩

lemma *PO-m1a-step2-refines-m1x-step2*:
{*R1x1a*}
(*m1x-step2 Rb A B*), (*m1a-step2 Rb A B*)
{> *R1x1a*}
⟨*proof*⟩

lemma *PO-m1a-step3-refines-m1x-step3*:
{*R1x1a*}
(*m1x-step3 Rs A B Kab*), (*m1a-step3 Rs A B Kab nls*)
{> *R1x1a*}
⟨*proof*⟩

lemma *PO-m1a-step4-refines-m1x-step4*:
{*R1x1a*}
(*m1x-step4 Ra A B Kab*), (*m1a-step4 Ra A B Kab nla*)
{> *R1x1a*}
⟨*proof*⟩

lemma *PO-m1a-step5-refines-m1x-step5*:
{*R1x1a*}
(*m1x-step5 Rb A B Kab*), (*m1a-step5 Rb A B Kab nlb*)
{> *R1x1a*}
⟨*proof*⟩

lemma *PO-m1a-leak-refines-m1x-leak*:
{*R1x1a*}
(*m1x-leak Rs*), (*m1a-leak Rs*)
{> *R1x1a*}
⟨*proof*⟩

All together now...

lemmas *PO-m1a-trans-refines-m1x-trans* =
PO-m1a-step1-refines-m1x-step1 PO-m1a-step2-refines-m1x-step2
PO-m1a-step3-refines-m1x-step3 PO-m1a-step4-refines-m1x-step4
PO-m1a-step5-refines-m1x-step5 PO-m1a-leak-refines-m1x-leak

lemma *PO-m1a-refines-init-m1x* [*iff*]:
init m1a ⊆ *R1x1a*“(*init m1x*)
⟨*proof*⟩

lemma *PO-m1a-refines-trans-m1x* [*iff*]:
{*R1x1a*}
(*trans m1x*), (*trans m1a*)
{> *R1x1a*}
⟨*proof*⟩

Observation consistency.

lemma *obs-consistent-med1x1a* [*iff*]:
obs-consistent R1x1a med1x1a m1x m1a
⟨*proof*⟩

Refinement result.

lemma *PO-m1a-refines-m1x* [*iff*]:
refines R1x1a med1x1a m1x m1a
⟨*proof*⟩

lemma *m1a-implements-m1x* [*iff*]: *implements med1x1a m1x m1a*
⟨*proof*⟩

3.3.6 Refinement of *a0n* for initiator/server

For the initiator, we get an non-injective agreement with the server on the session key, the responder name, and the atom list *isl*.

Simulation relation

We define two auxiliary functions to reconstruct the signals of the initial model from completed initiator and server runs.

type-synonym
issig = *key* × *agent* × *atom list*

abbreviation
is-commit :: [*runs-t*, *agent*, *agent*, *key*, *atom list*] ⇒ *rid-t set*
where
is-commit runz A B Kab sl ≡ {*Ra*. ∃ *nla*.
runz Ra = *Some* (*Init*, [*A*, *B*], *aKey Kab* # *nla*) ∧ *take is-len nla* = *sl*
}

fun

is-runs2sigs :: *runs-t* ⇒ *issig signal* ⇒ *nat*
where
is-runs2sigs runz (*Running* [*A*, *Sv*] (*Kab*, *B*, *sl*)) =
(*if* ∃ *Rs nls*. *Kab* = *sesK* (*Rs*$*sk*) ∧
runz Rs = *Some* (*Serv*, [*A*, *B*], *nls*) ∧ *take is-len nls* = *sl*
then 1 else 0)

| *is-runs2sigs runz* (*Commit* [*A*, *Sv*] (*Kab*, *B*, *sl*)) =
card (*is-commit runz A B Kab sl*)

| *is-runs2sigs runz* - = *0*

Simulation relation and mediator function. We map completed initiator and responder runs to commit and running signals, respectively.

definition
med-a0m1a-is :: *m1a-obs* ⇒ *issig a0i-obs* **where**
med-a0m1a-is o1 ≡ (| *signals* = *is-runs2sigs* (*runs o1*), *corrupted* = {} |)

definition
R-a0m1a-is :: (*issig a0i-state* × *m1a-state*) *set* **where**
R-a0m1a-is ≡ {(*s*, *t*). *signals s* = *is-runs2sigs* (*runs t*) ∧ *corrupted s* = {} }

lemmas *R-a0m1a-is-defs* = *R-a0m1a-is-def med-a0m1a-is-def*

Lemmas about the auxiliary functions

lemma *is-runs2sigs-empty* [*simp*]:
runz = *Map.empty* ⟹ *is-runs2sigs runz* = (λ*s*. *0*)
⟨*proof*⟩

lemma *is-commit-finite* [*simp*, *intro*]:
finite (*dom runz*) ⟹ *finite* (*is-commit runz A B Kab nls*)
⟨*proof*⟩

Update lemmas

lemma *is-runs2sigs-upd-init-none* [*simp*]:
⟦ *Ra* ∉ *dom runz* ⟧
⟹ *is-runs2sigs* (*runz*(*Ra* ↦ (*Init*, [*A*, *B*], []))) = *is-runs2sigs runz*
⟨*proof*⟩

lemma *is-runs2sigs-upd-resp-none* [*simp*]:
⟦ *Rb* ∉ *dom runz* ⟧
⟹ *is-runs2sigs* (*runz*(*Rb* ↦ (*Resp*, [*A*, *B*], []))) = *is-runs2sigs runz*
⟨*proof*⟩

lemma *is-runs2sigs-upd-serv* [*simp*]:
⟦ *Rs* ∉ *dom runz* ⟧
⟹ *is-runs2sigs* (*runz*(*Rs* ↦ (*Serv*, [*A*, *B*], *nls*))) =
(*is-runs2sigs runz*)(*Running* [*A*, *Sv*] (*sesK* (*Rs*$*sk*), *B*, *take is-len nls*) := *1*)
⟨*proof*⟩

lemma *is-runs2sigs-upd-init-some* [*simp*]:

⟦ *runz Ra* = *Some* (*Init*, [*A*, *B*], []); *finite* (*dom runz*);
ils = *take is-len nla* ⟧
⟹ *is-runs2sigs* (*runz*(*Ra* ↦ (*Init*, [*A*, *B*], *aKey Kab* # *nla*))) =
(*is-runs2sigs runz*)(
Commit [*A*, *Sv*] (*Kab*, *B*, *ils*) :=
Suc (*card* (*is-commit runz A B Kab ils*)))
⟨*proof*⟩

lemma *is-runs2sigs-upd-resp-some* [*simp*]:
⟦ *runz Rb* = *Some* (*Resp*, [*A*, *B*], []) ⟧
⟹ *is-runs2sigs* (*runz*(*Rb* ↦ (*Resp*, [*A*, *B*], *aKey Kab* # *nlb*))) =
is-runs2sigs runz
⟨*proof*⟩

Refinement proof

lemma *PO-m1a-step1-refines-a0-is-skip*:
{*R-a0m1a-is*}
Id, (*m1a-step1 Ra A B*)
{> *R-a0m1a-is*}
⟨*proof*⟩

lemma *PO-m1a-step2-refines-a0-is-skip*:
{*R-a0m1a-is*}
Id, (*m1a-step2 Rb A B*)
{> *R-a0m1a-is*}
⟨*proof*⟩

lemma *PO-m1a-step3-refines-a0-is-running*:
{*R-a0m1a-is*}
(*a0n-running* [*A*, *Sv*] (*Kab*, *B*, *take is-len nls*)),
(*m1a-step3 Rs A B Kab nls*)
{> *R-a0m1a-is*}
⟨*proof*⟩

lemma *PO-m1a-step4-refines-a0-is-commit*:
{*R-a0m1a-is* ∩ *UNIV* × *m1a-inv0-fin*}
(*a0n-commit* [*A*, *Sv*] (*Kab*, *B*, *take is-len nla*)),
(*m1a-step4 Ra A B Kab nla*)
{> *R-a0m1a-is*}
⟨*proof*⟩

lemma *PO-m1a-step5-refines-a0-is-skip*:
{*R-a0m1a-is*}
Id, (*m1a-step5 Rb A B Kab nlb*)
{> *R-a0m1a-is*}
⟨*proof*⟩

lemma *PO-m1a-leak-refines-a0-is-skip*:
{*R-a0m1a-is*}
Id, (*m1a-leak Rs*)
{> *R-a0m1a-is*}
⟨*proof*⟩

All together now...

lemmas *PO-m1a-trans-refines-a0-is-trans* =
PO-m1a-step1-refines-a0-is-skip PO-m1a-step2-refines-a0-is-skip
PO-m1a-step3-refines-a0-is-running PO-m1a-step4-refines-a0-is-commit
PO-m1a-step5-refines-a0-is-skip PO-m1a-leak-refines-a0-is-skip

lemma *PO-m1a-refines-init-a0-is* [*iff*]:
init m1a ⊆ *R-a0m1a-is*``(*init a0n*)
⟨*proof*⟩

lemma *PO-m1a-refines-trans-a0-is* [*iff*]:
{*R-a0m1a-is* ∩ *UNIV* × *m1a-inv0-fin*}
(*trans a0n*), (*trans m1a*)
{> *R-a0m1a-is*}
⟨*proof*⟩

lemma *obs-consistent-med-a0m1a-is* [*iff*]:
obs-consistent R-a0m1a-is med-a0m1a-is a0n m1a
⟨*proof*⟩

Refinement result.

lemma *PO-m1a-refines-a0-is* [*iff*]:
refines (*R-a0m1a-is* ∩ *UNIV* × *m1a-inv0-fin*) *med-a0m1a-is a0n m1a*
⟨*proof*⟩

lemma *m1a-implements-a0-is*: *implements med-a0m1a-is a0n m1a*
⟨*proof*⟩

3.3.7 Refinement of *a0n* for responder/server

For the responder, we get a non-injective agreement with the server on the session key, the initiator's name, and additional data.

Simulation relation

We define two auxiliary functions to reconstruct the signals of the initial model from completed responder and server runs.

type-synonym
rssig = *key* × *agent* × *atom list*

abbreviation
rs-commit :: [*runs-t*, *agent*, *agent*, *key*, *atom list*] ⇒ *rid-t set*
where
rs-commit runz A B Kab rsl ≡ {*Rb*. ∃ *nlb*.
runz Rb = *Some* (*Resp*, [*A*, *B*], *aKey Kab* # *nlb*) ∧ *take rs-len nlb* = *rsl*
}

fun
rs-runs2sigs :: *runs-t* ⇒ *rssig signal* ⇒ *nat*
where

rs-runs2sigs runz (*Running* [*B*, *Sv*] (*Kab*, *A*, *rsl*)) =
(*if* ∃ *Rs nls*. *Kab* = *sesK* (*Rs*$*sk*) ∧
runz Rs = *Some* (*Serv*, [*A*, *B*], *nls*) ∧ *take rs-len nls* = *rsl*
then 1 else 0)

| *rs-runs2sigs runz* (*Commit* [*B*, *Sv*] (*Kab*, *A*, *rsl*)) =
card (*rs-commit runz A B Kab rsl*)

| *rs-runs2sigs runz* - = *0*

Simulation relation and mediator function. We map completed initiator and responder runs to commit and running signals, respectively.

definition
med-a0m1a-rs :: *m1a-obs* ⇒ *rssig a0n-obs* **where**
med-a0m1a-rs o1 ≡ (| *signals* = *rs-runs2sigs* (*runs o1*), *corrupted* = {} |)

definition
R-a0m1a-rs :: (*rssig a0n-state* × *m1a-state*) *set* **where**
R-a0m1a-rs ≡ {(*s*, *t*). *signals s* = *rs-runs2sigs* (*runs t*) ∧ *corrupted s* = {} }

lemmas *R-a0m1a-rs-defs* = *R-a0m1a-rs-def med-a0m1a-rs-def*

Lemmas about the auxiliary functions

Other lemmas

lemma *rs-runs2sigs-empty* [*simp*]:
runz = *Map.empty* ⟹ *rs-runs2sigs runz* = (λ*s*. *0*)
⟨*proof*⟩

lemma *rs-commit-finite* [*simp*, *intro*]:
finite (*dom runz*) ⟹ *finite* (*rs-commit runz A B Kab nls*)
⟨*proof*⟩

Update lemmas

lemma *rs-runs2sigs-upd-init-none* [*simp*]:
⟦ *Ra* ∉ *dom runz* ⟧
⟹ *rs-runs2sigs* (*runz*(*Ra* ↦ (*Init*, [*A*, *B*], []))) = *rs-runs2sigs runz*
⟨*proof*⟩

lemma *rs-runs2sigs-upd-resp-none* [*simp*]:
⟦ *Rb* ∉ *dom runz* ⟧
⟹ *rs-runs2sigs* (*runz*(*Rb* ↦ (*Resp*, [*A*, *B*], []))) = *rs-runs2sigs runz*
⟨*proof*⟩

lemma *rs-runs2sigs-upd-serv* [*simp*]:
⟦ *Rs* ∉ *dom runz* ⟧
⟹ *rs-runs2sigs* (*runz*(*Rs* ↦ (*Serv*, [*A*, *B*], *nls*))) =
(*rs-runs2sigs runz*)(*Running* [*B*, *Sv*] (*sesK* (*Rs*$*sk*), *A*, *take rs-len nls*) := *1*)
⟨*proof*⟩

lemma *rs-runs2sigs-upd-init-some* [*simp*]:
⟦ *runz Ra* = *Some* (*Init*, [*A*, *B*], []) ⟧

⟹ *rs-runs2sigs* (*runz*(*Ra* ↦ (*Init*, [*A*, *B*], *aKey Kab* # *nl*))) =
rs-runs2sigs runz
⟨*proof*⟩

lemma *rs-runs2sigs-upd-resp-some* [*simp*]:
⟦ *runz Rb* = *Some* (*Resp*, [*A*, *B*], []); *finite* (*dom runz*);
rsl = *take rs-len nlb* ⟧
⟹ *rs-runs2sigs* (*runz*(*Rb* ↦ (*Resp*, [*A*, *B*], *aKey Kab* # *nlb*))) =
(*rs-runs2sigs runz*)(
Commit [*B*, *Sv*] (*Kab*, *A*, *rsl*) := *Suc* (*card* (*rs-commit runz A B Kab rsl*)))
⟨*proof*⟩

Refinement proof

lemma *PO-m1a-step1-refines-a0-rs-skip*:
{*R-a0m1a-rs*}
Id, (*m1a-step1 Ra A B*)
{> *R-a0m1a-rs*}
⟨*proof*⟩

lemma *PO-m1a-step2-refines-a0-rs-skip*:
{*R-a0m1a-rs*}
Id, (*m1a-step2 Rb A B*)
{> *R-a0m1a-rs*}
⟨*proof*⟩

lemma *PO-m1a-step3-refines-a0-rs-running*:
{*R-a0m1a-rs*}
(*a0n-running* [*B*, *Sv*] (*Kab*, *A*, *take rs-len nls*)),
(*m1a-step3 Rs A B Kab nls*)
{> *R-a0m1a-rs*}
⟨*proof*⟩

lemma *PO-m1a-step4-refines-a0-rs-skip*:
{*R-a0m1a-rs*}
Id, (*m1a-step4 Ra A B Kab nla*)
{> *R-a0m1a-rs*}
⟨*proof*⟩

lemma *PO-m1a-step5-refines-a0-rs-commit*:
{*R-a0m1a-rs* ∩ *UNIV* × *m1a-inv0-fin*}
(*a0n-commit* [*B*, *Sv*] (*Kab*, *A*, *take rs-len nlb*)),
(*m1a-step5 Rb A B Kab nlb*)
{> *R-a0m1a-rs*}
⟨*proof*⟩

lemma *PO-m1a-leak-refines-a0-rs-skip*:
{*R-a0m1a-rs*}
Id, (*m1a-leak Rs*)
{> *R-a0m1a-rs*}
⟨*proof*⟩

All together now...

lemmas *PO-m1a-trans-refines-a0-rs-trans =*
PO-m1a-step1-refines-a0-rs-skip PO-m1a-step2-refines-a0-rs-skip
PO-m1a-step3-refines-a0-rs-running PO-m1a-step4-refines-a0-rs-skip
PO-m1a-step5-refines-a0-rs-commit PO-m1a-leak-refines-a0-rs-skip

lemma *PO-m1a-refines-init-ra0n* [*iff*]:
init m1a ⊆ *R-a0m1a-rs*``(*init a0n*)
⟨*proof*⟩

lemma *PO-m1a-refines-trans-ra0n* [*iff*]:
{*R-a0m1a-rs* ∩ *UNIV* × *m1a-inv0-fin*}
(*trans a0n*), (*trans m1a*)
{> *R-a0m1a-rs*}
⟨*proof*⟩

lemma *obs-consistent-med-a0m1a-rs* [*iff*]:
obs-consistent (*R-a0m1a-rs* ∩ *UNIV* × *m1a-inv0-fin*) *med-a0m1a-rs a0n m1a*
⟨*proof*⟩

Refinement result.

lemma *PO-m1a-refines-a0-rs* [*iff*]:
refines (*R-a0m1a-rs* ∩ *UNIV* × *m1a-inv0-fin*) *med-a0m1a-rs a0n m1a*
⟨*proof*⟩

lemma *m1a-implements-ra0n*: *implements med-a0m1a-rs a0n m1a*
⟨*proof*⟩

end

3.4 Abstract Kerberos core protocol (L1)

theory *m1-kerberos* **imports** *m1-keydist-iirn*
begin

We augment the basic abstract key distribution model such that the server sends a timestamp along with the session key. We use a cache to guard against replay attacks and timestamp validity checks to ensure recentness of the session key.

We establish three refinements for this model, namely that this model refines

1. the authenticated key distribution model *m1-keydist-iirn*,
2. the injective agreement model *a0i*, instantiated such that the responder agrees with the initiator on the session key, its timestamp and the initiator's authenticator timestamp.
3. the injective agreement model *a0i*, instantiated such that the initiator agrees with the responder on the session key, its timestamp and the initiator's authenticator timestamp.

3.4.1 State

We extend the basic key distribution by adding timestamps. We add a clock variable modeling the current time and an authenticator replay cache recording triples (*A*, *Kab*, *Ta*) of agents, session keys, and authenticator timestamps. The inclusion of the session key avoids false replay rejections for different keys with identical authenticator timestamps.

The frames, runs, and observations remain the same as in the previous model, but we will use the *nat list*'s to store timestamps.

type-synonym
  time = *nat* — for clock and timestamps

consts
  Ls :: *time* — life time for session keys
  La :: *time* — life time for authenticators

State and observations

record
  m1-state = *m1r-state* +
    leak :: (*key* × *agent* × *agent* × *nonce* × *time*) *set* — key leaked plus context
    clk :: *time*
    cache :: (*agent* × *key* × *time*) *set*

type-synonym *m1-obs* = *m1-state*

type-synonym *'x m1-pred* = *'x m1-state-scheme set*
type-synonym *'x m1-trans* = (*'x m1-state-scheme* × *'x m1-state-scheme*) *set*

consts
  END :: *atom* — run end marker (for initiator)

3.4.2 Events

definition — by *A*, refines *m1x-step1*
  m1-step1 :: [*rid-t*, *agent*, *agent*, *nonce*] ⇒ *'x m1-trans*
where
  m1-step1 ≡ *m1a-step1*

definition — by *B*, refines *m1x-step2*
  m1-step2 :: [*rid-t*, *agent*, *agent*] ⇒ *'x m1-trans*
where
  m1-step2 ≡ *m1a-step2*

definition — by *Sv*, refines *m1x-step3*
  m1-step3 :: [*rid-t*, *agent*, *agent*, *key*, *nonce*, *time*] ⇒ *'x m1-trans*
where
  m1-step3 Rs A B Kab Na Ts ≡ {(*s*, *s'*).
    — new guards:
    Ts = *clk s* ∧ — fresh timestamp

    — rest as before:
    (*s*, *s'*) ∈ *m1a-step3 Rs A B Kab Na* [*aNum Ts*]
  }

definition — by A, refines *m1x-step5*
  m1-step4 :: [*rid-t*, *agent*, *agent*, *nonce*, *key*, *time*, *time*] ⇒ *'x m1-trans*
where
  m1-step4 Ra A B Na Kab Ts Ta ≡ {(*s*, *s'*).
    — previous guards:
    runs s Ra = *Some* (*Init*, [*A*, *B*], []) ∧
    (*Kab* ∉ *Domain* (*leak s*) ⟶ (*Kab*, *A*) ∈ *azC* (*runs s*)) ∧ — authorization guard
    Na = *Ra*$*na* ∧ — fix parameter

    — guard for agreement with server on (*Kab*, *B*, *Na*, *isl*),
    — where *isl* = *take is-len nla*; injectiveness by including *Na*
    (*A* ∉ *bad* ⟶ (∃ *Rs*. *Kab* = *sesK* (*Rs*$*sk*) ∧
      runs s Rs = *Some* (*Serv*, [*A*, *B*], [*aNon Na*, *aNum Ts*]))) ∧

    — new guards:
    Ta = *clk s* ∧ — fresh timestamp
    clk s < *Ts* + *Ls* ∧ — ensure session key recentness

    — actions:
    s' = *s*⦇ *runs* := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [*aKey Kab*, *aNum Ts*, *aNum Ta*])) ⦈
  }

definition — by B, refines *m1x-step4*
  m1-step5 :: [*rid-t*, *agent*, *agent*, *key*, *time*, *time*] ⇒ *'x m1-trans*
where
  m1-step5 Rb A B Kab Ts Ta ≡ {(*s*, *s'*).
    — previous guards:
    runs s Rb = *Some* (*Resp*, [*A*, *B*], []) ∧
    (*Kab* ∉ *Domain* (*leak s*) ⟶ (*Kab*, *B*) ∈ *azC* (*runs s*)) ∧ — authorization guard

    — guard for showing agreement with server on (*Kab*, *A*, *rsl*),
    — where *rsl* = *take rs-len nlb*; this agreement is non-injective
    (*B* ∉ *bad* ⟶ (∃ *Rs Na*. *Kab* = *sesK* (*Rs*$*sk*) ∧
      runs s Rs = *Some* (*Serv*, [*A*, *B*], [*aNon Na*, *aNum Ts*]))) ∧

    — new guards:
    — guard for showing agreement with initiator *A* on (*Kab*, *Ts*, *Ta*)
    (*A* ∉ *bad* ⟶ *B* ∉ *bad* ⟶
     (∃ *Ra nl*. *runs s Ra* = *Some* (*Init*, [*A*, *B*], *aKey Kab* # *aNum Ts* # *aNum Ta* # *nl*))) ∧

    — ensure recentness of session key
    clk s < *Ts* + *Ls* ∧

    — check validity of authenticator and prevent its replay
    — 'replays' with fresh authenticator ok!
    clk s < *Ta* + *La* ∧
    (*B*, *Kab*, *Ta*) ∉ *cache s* ∧

    — actions:
    s' = *s*⦇
      runs := (*runs s*)(*Rb* ↦ (*Resp*, [*A*, *B*], [*aKey Kab*, *aNum Ts*, *aNum Ta*])),
      cache := *insert* (*B*, *Kab*, *Ta*) (*cache s*)

  ⦈
}

definition — by A, refines *skip*
  m1-step6 :: [*rid-t*, *agent*, *agent*, *nonce*, *key*, *time*, *time*] ⇒ *'x m1-trans*
where
  m1-step6 Ra A B Na Kab Ts Ta ≡ {(*s*, *s'*).
    runs s Ra = *Some* (*Init*, [*A*, *B*], [*aKey Kab*, *aNum Ts*, *aNum Ta*]) ∧ — key recv'd before
    Na = *Ra*$*na* ∧ — fix parameter

    — check key's freshness [NEW]
    — *clk s* < *Ts* + *Ls* ∧

    — guard for showing agreement with B on *Kab*, *Ts*, and *Ta*
    (*A* ∉ *bad* ⟶ *B* ∉ *bad* ⟶
      (∃ *Rb*. *runs s Rb* = *Some* (*Resp*, [*A*, *B*], [*aKey Kab*, *aNum Ts*, *aNum Ta*]))) ∧

    — actions: (redundant) update local state marks successful termination
    s' = *s*⦇
      runs := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [*aKey Kab*, *aNum Ts*, *aNum Ta*, *END*]))
    ⦈
  }

definition — by attacker, refines *m1a-leak*
  m1-leak :: [*rid-t*, *agent*, *agent*, *nonce*, *time*] ⇒ *'x m1-trans*
where
  m1-leak Rs A B Na Ts ≡ {(*s*, *s1*).
    — guards:
    runs s Rs = *Some* (*Serv*, [*A*, *B*], [*aNon Na*, *aNum Ts*]) ∧
    (*clk s* ≥ *Ts* + *Ls*) ∧ — only compromise 'old' session keys

    — actions:
    — record session key as leaked;
    s1 = *s*⦇ *leak* := *insert* (*sesK* (*Rs*$*sk*), *A*, *B*, *Na*, *Ts*) (*leak s*) ⦈
  }

Clock tick event

definition — refines *skip*
  m1-tick :: *time* ⇒ *'x m1-trans*
where
  m1-tick T ≡ {(*s*, *s'*).
    s' = *s*⦇ *clk* := *clk s* + *T* ⦈
  }

Purge event: purge cache of expired timestamps

definition — refines *skip*
  m1-purge :: *agent* ⇒ *'x m1-trans*
where
  m1-purge A ≡ {(*s*, *s'*).
    s' = *s*⦇
      cache := *cache s* − {(*A*, *K*, *T*) | *A K T*.
        (*A*, *K*, *T*) ∈ *cache s* ∧ *T* + *La* ≤ *clk s*

}
⦈
}

3.4.3 Specification

definition
m1-init :: *m1-state set*
where
$m1\text{-}init \equiv \{ ⦇ runs = Map.empty, leak = corrKey \times \{undefined\}, clk = 0, cache = \{\} ⦈ \}$

definition
m1-trans :: *'x m1-trans* **where**
m1-trans ≡ (⋃ *A B Ra Rb Rs Na Kab Ts Ta T*.
m1-step1 Ra A B Na ∪
m1-step2 Rb A B ∪
m1-step3 Rs A B Kab Na Ts ∪
m1-step4 Ra A B Na Kab Ts Ta ∪
m1-step5 Rb A B Kab Ts Ta ∪
m1-step6 Ra A B Na Kab Ts Ta ∪
m1-leak Rs A B Na Ts ∪
m1-tick T ∪
m1-purge A ∪
Id
)

definition
m1 :: (*m1-state*, *m1-obs*) *spec* **where**
m1 ≡ ⦇
init = *m1-init*,
trans = *m1-trans*,
obs = *id*
⦈

lemmas *m1-loc-defs* =
m1-def m1-init-def m1-trans-def
m1-step1-def m1-step2-def m1-step3-def m1-step4-def m1-step5-def
m1-step6-def m1-leak-def m1-purge-def m1-tick-def

lemmas *m1-defs* = *m1-loc-defs m1a-defs*

lemma *m1-obs-id* [*simp*]: *obs m1* = *id*
⟨*proof*⟩

3.4.4 Invariants

inv0: Finite domain

There are only finitely many runs. This is needed to establish the responder/initiator agreement.

definition
m1-inv0-fin :: *'x m1-pred*

where
  m1-inv0-fin ≡ {*s. finite* (*dom* (*runs s*))}

lemmas *m1-inv0-finI* = *m1-inv0-fin-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m1-inv0-finE* [*elim*] = *m1-inv0-fin-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m1-inv0-finD* = *m1-inv0-fin-def* [*THEN setc-def-to-dest*, *rule-format*]

Invariance proofs.

lemma *PO-m1-inv0-fin-init* [*iff*]:
  init m1 ⊆ *m1-inv0-fin*
⟨*proof*⟩

lemma *PO-m1-inv0-fin-trans* [*iff*]:
  {*m1-inv0-fin*} *trans m1* {> *m1-inv0-fin*}
⟨*proof*⟩

lemma *PO-m1-inv0-fin* [*iff*]: *reach m1* ⊆ *m1-inv0-fin*
⟨*proof*⟩

inv1: Caching invariant for responder

definition
  m1-inv1r-cache :: *'x m1-pred*
where
  m1-inv1r-cache ≡ {*s.* ∀ *Rb A B Kab Ts Ta nl.*
    runs s Rb = *Some* (*Resp*, [*A*, *B*], *aKey Kab* # *aNum Ts* # *aNum Ta* # *nl*) ⟶
    clk s < *Ta* + *La* ⟶
      (*B*, *Kab*, *Ta*) ∈ *cache s*
  }

lemmas *m1-inv1r-cacheI* = *m1-inv1r-cache-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m1-inv1r-cacheE* [*elim*] = *m1-inv1r-cache-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m1-inv1r-cacheD* = *m1-inv1r-cache-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

Invariance proof

lemma *PO-m1-inv1r-cache-init* [*iff*]:
  init m1 ⊆ *m1-inv1r-cache*
⟨*proof*⟩

lemma *PO-m1-inv1r-cache-trans* [*iff*]:
  {*m1-inv1r-cache*} *trans m1* {> *m1-inv1r-cache*}
⟨*proof*⟩

lemma *PO-m1-inv1r-cache* [*iff*]: *reach m1* ⊆ *m1-inv1r-cache*
⟨*proof*⟩

3.4.5 Refinement of *m1a*

Simulation relation

The abstraction removes all but the first freshness identifiers (corresponding to *Kab* and *Ts*) from the initiator and responder frames and leaves the server's freshness ids untouched.

overloading *is-len′* ≡ *is-len* *rs-len′* ≡ *rs-len* **begin**
definition *is-len-def* [*simp*]: *is-len′* ≡ *1*::*nat*
definition *rs-len-def* [*simp*]: *rs-len′* ≡ *1*::*nat*
end

fun
  rm1a1 :: *role-t* ⇒ *atom list* ⇒ *atom list*
where
  rm1a1 Init = *take* (*Suc is-len*) — take *Kab*, *Ts*; drop *Ta*
| *rm1a1 Resp* = *take* (*Suc rs-len*) — take *Kab*, *Ts*; drop *Ta*
| *rm1a1 Serv* = *id* — take *Na*, *Ts*

abbreviation
  runs1a1 :: *runs-t* ⇒ *runs-t* **where**
  runs1a1 ≡ *map-runs rm1a1*

lemma *knC-runs1a1* [*simp*]:
  knC (*runs1a1 runz*) = *knC runz*
⟨*proof*⟩

med1a1: The mediator function maps a concrete observation (i.e., run) to an abstract one.

R1a1: The simulation relation is defined in terms of the mediator function.

definition
  med1a1 :: *m1-obs* ⇒ *m1a-obs* **where**
  med1a1 s ≡ (| *runs* = *runs1a1* (*runs s*), *m1x-state.leak* = *Domain* (*leak s*) |)

definition
  R1a1 :: (*m1a-state* × *m1-state*) *set* **where**
  R1a1 ≡ {(*s*, *t*). *s* = *med1a1 t*}

lemmas *R1a1-defs* = *R1a1-def med1a1-def*

Refinement proof

lemma *PO-m1-step1-refines-m1a-step1*:
  {*R1a1*}
    (*m1a-step1 Ra A B Na*), (*m1-step1 Ra A B Na*)
  {> *R1a1*}
⟨*proof*⟩

lemma *PO-m1-step2-refines-m1a-step2*:
  {*R1a1*}
    (*m1a-step2 Rb A B*), (*m1-step2 Rb A B*)
  {> *R1a1*}
⟨*proof*⟩

lemma *PO-m1-step3-refines-m1a-step3*:
  {*R1a1*}
    (*m1a-step3 Rs A B Kab Na* [*aNum Ts*]), (*m1-step3 Rs A B Kab Na Ts*)
  {> *R1a1*}
⟨*proof*⟩

lemma *PO-m1-step4-refines-m1a-step4*:
{*R1a1*}
(*m1a-step4 Ra A B Na Kab* [*aNum Ts*]), (*m1-step4 Ra A B Na Kab Ts Ta*)
{> *R1a1*}
⟨*proof*⟩

lemma *PO-m1-step5-refines-m1a-step5*:
{*R1a1*}
(*m1a-step5 Rb A B Kab* [*aNum Ts*]), (*m1-step5 Rb A B Kab Ts Ta*)
{> *R1a1*}
⟨*proof*⟩

lemma *PO-m1-step6-refines-m1a-skip*:
{*R1a1*}
Id, (*m1-step6 Ra A B Na Kab Ts Ta*)
{> *R1a1*}
⟨*proof*⟩

lemma *PO-m1-leak-refines-m1a-leak*:
{*R1a1*}
(*m1a-leak Rs*), (*m1-leak Rs A B Na Ts*)
{> *R1a1*}
⟨*proof*⟩

lemma *PO-m1-tick-refines-m1a-skip*:
{*R1a1*}
Id, (*m1-tick T*)
{> *R1a1*}
⟨*proof*⟩

lemma *PO-m1-purge-refines-m1a-skip*:
{*R1a1*}
Id, (*m1-purge A*)
{> *R1a1*}
⟨*proof*⟩

All together now...

lemmas *PO-m1-trans-refines-m1a-trans* =
PO-m1-step1-refines-m1a-step1 PO-m1-step2-refines-m1a-step2
PO-m1-step3-refines-m1a-step3 PO-m1-step4-refines-m1a-step4
PO-m1-step5-refines-m1a-step5 PO-m1-step6-refines-m1a-skip
PO-m1-leak-refines-m1a-leak PO-m1-tick-refines-m1a-skip
PO-m1-purge-refines-m1a-skip

lemma *PO-m1-refines-init-m1a* [*iff*]:
init m1 ⊆ *R1a1*``(*init m1a*)
⟨*proof*⟩

lemma *PO-m1-refines-trans-m1a* [*iff*]:
{*R1a1*}
(*trans m1a*), (*trans m1*)
{> *R1a1*}
⟨*proof*⟩

Observation consistency.

lemma *obs-consistent-med1a1* [*iff*]:
  obs-consistent R1a1 med1a1 m1a m1
⟨*proof*⟩

Refinement result.

lemma *PO-m1-refines-m1a* [*iff*]:
  refines R1a1 med1a1 m1a m1
⟨*proof*⟩

lemma *m1-implements-m1a* [*iff*]: *implements med1a1 m1a m1*
⟨*proof*⟩

inv (inherited): Secrecy

Secrecy, as external and internal invariant

definition
  m1-secrecy :: *'x m1-pred* **where**
  m1-secrecy ≡ {*s. knC* (*runs s*) ⊆ *azC* (*runs s*) ∪ *Domain* (*leak s*) × *UNIV*}

lemmas *m1-secrecyI* = *m1-secrecy-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m1-secrecyE* [*elim*] = *m1-secrecy-def* [*THEN setc-def-to-elim*, *rule-format*]

lemma *PO-m1-obs-secrecy* [*iff*]: *oreach m1* ⊆ *m1-secrecy*
⟨*proof*⟩

lemma *PO-m1-secrecy* [*iff*]: *reach m1* ⊆ *m1-secrecy*
⟨*proof*⟩

inv (inherited): Responder auth server.

definition
  m1-inv2r-serv :: *'x m1r-pred*
where
  m1-inv2r-serv ≡ {*s*. ∀ *A B Rb Kab Ts nlb*.
    B ∉ *bad* ⟶
    runs s Rb = *Some* (*Resp*, [*A*, *B*], *aKey Kab* # *aNum Ts* # *nlb*) ⟶
      (∃ *Rs Na*. *Kab* = *sesK* (*Rs*$*sk*) ∧
        runs s Rs = *Some* (*Serv*, [*A*, *B*], [*aNon Na*, *aNum Ts*]))
  }

lemmas *m1-inv2r-servI* = *m1-inv2r-serv-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m1-inv2r-servE* [*elim*] = *m1-inv2r-serv-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m1-inv2r-servD* = *m1-inv2r-serv-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated* −*1*]

Proof of invariance.

lemma *PO-m1-inv2r-serv* [*iff*]: *reach m1* ⊆ *m1-inv2r-serv*
⟨*proof*⟩

inv (inherited): Initiator auth server.

Simplified version of invariant *m1a-inv2i-serv*.

definition
  m1-inv2i-serv :: *'x m1r-pred*
where
  m1-inv2i-serv ≡ {*s*. ∀ *A B Ra Kab Ts nla*.
    A ∉ *bad* ⟶
    runs s Ra = *Some* (*Init*, [*A*, *B*], *aKey Kab* # *aNum Ts* # *nla*) ⟶
      (∃ *Rs*. *Kab* = *sesK* (*Rs*$*sk*) ∧
        runs s Rs = *Some* (*Serv*, [*A*, *B*], [*aNon* (*Ra*$*na*), *aNum Ts*]))
  }

lemmas *m1-inv2i-servI* = *m1-inv2i-serv-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m1-inv2i-servE* [*elim*] = *m1-inv2i-serv-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m1-inv2i-servD* = *m1-inv2i-serv-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated* −*1*]

Proof of invariance.

lemma *PO-m1-inv2i-serv* [*iff*]: *reach m1* ⊆ *m1-inv2i-serv*
⟨*proof*⟩

declare *PO-m1-inv2i-serv* [*THEN subsetD*, *intro*]

inv (inherited): Initiator key freshness

definition
  m1-inv1-ifresh :: *'x m1-pred*
where
  m1-inv1-ifresh ≡ {*s*. ∀ *A A' B B' Ra Ra' Kab nl nl'*.
    runs s Ra = *Some* (*Init*, [*A*, *B*], *aKey Kab* # *nl*) ⟶
    runs s Ra' = *Some* (*Init*, [*A'*, *B'*], *aKey Kab* # *nl'*) ⟶
    A ∉ *bad* ⟶ *B* ∉ *bad* ⟶ *Kab* ∉ *Domain* (*leak s*) ⟶
      Ra = *Ra'*
  }

lemmas *m1-inv1-ifreshI* = *m1-inv1-ifresh-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m1-inv1-ifreshE* [*elim*] = *m1-inv1-ifresh-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m1-inv1-ifreshD* = *m1-inv1-ifresh-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

lemma *PO-m1-ifresh* [*iff*]: *reach m1* ⊆ *m1-inv1-ifresh*
⟨*proof*⟩

3.4.6 Refinement of *a0i* for responder/initiator

The responder injectively agrees with the initiator on *Kab*, *Ts*, and *Ta*.

Simulation relation

We define two auxiliary functions to reconstruct the signals of the initial model from completed initiator and responder runs.

type-synonym
  risig = *key* × *time* × *time*

abbreviation

ri-running :: [*runs-t*, *agent*, *agent*, *key*, *time*, *time*] ⇒ *rid-t set*
where
ri-running runz A B Kab Ts Ta ≡ {*Ra*. ∃ *nl*.
runz Ra = *Some* (*Init*, [*A*, *B*], *aKey Kab* # *aNum Ts* # *aNum Ta* # *nl*)
}

abbreviation
ri-commit :: [*runs-t*, *agent*, *agent*, *key*, *time*, *time*] ⇒ *rid-t set*
where
ri-commit runz A B Kab Ts Ta ≡ {*Rb*. ∃ *nl*.
runz Rb = *Some* (*Resp*, [*A*, *B*], *aKey Kab* # *aNum Ts* # *aNum Ta* # *nl*)
}

fun
ri-runs2sigs :: *runs-t* ⇒ *risig signal* ⇒ *nat*
where
ri-runs2sigs runz (*Running* [*B*, *A*] (*Kab*, *Ts*, *Ta*)) =
card (*ri-running runz A B Kab Ts Ta*)

| *ri-runs2sigs runz* (*Commit* [*B*, *A*] (*Kab*, *Ts*, *Ta*)) =
card (*ri-commit runz A B Kab Ts Ta*)

| *ri-runs2sigs runz* - = *0*

Simulation relation and mediator function. We map completed initiator and responder runs to commit and running signals, respectively.

definition
med-a0iim1-ri :: *m1-obs* ⇒ *risig a0i-obs* **where**
med-a0iim1-ri o1 ≡ (| *signals* = *ri-runs2sigs* (*runs o1*), *corrupted* = {} |)

definition
R-a0iim1-ri :: (*risig a0i-state* × *m1-state*) *set* **where**
R-a0iim1-ri ≡ {(*s*, *t*). *signals s* = *ri-runs2sigs* (*runs t*) ∧ *corrupted s* = {} }

lemmas *R-a0iim1-ri-defs* = *R-a0iim1-ri-def med-a0iim1-ri-def*

Lemmas about the auxiliary functions

Other lemmas

lemma *ri-runs2sigs-empty* [*simp*]:
runz = *Map.empty* ⟹ *ri-runs2sigs runz* = (λ*s*. *0*)
⟨*proof*⟩

lemma *finite-ri-running* [*simp*, *intro*]:
finite (*dom runz*) ⟹ *finite* (*ri-running runz A B Kab Ts Ta*)
⟨*proof*⟩

lemma *finite-ri-commit* [*simp*, *intro*]:
finite (*dom runz*) ⟹ *finite* (*ri-commit runz A B Kab Ts Ta*)
⟨*proof*⟩

Update lemmas

lemma *ri-runs2sigs-upd-init-none* [*simp*]:
  ⟦ *Na* ∉ *dom runz* ⟧
  ⟹ *ri-runs2sigs* (*runz*(*Na* ↦ (*Init*, [*A*, *B*], []))) = *ri-runs2sigs runz*
⟨*proof*⟩

lemma *ri-runs2sigs-upd-resp-none* [*simp*]:
  ⟦ *Rb* ∉ *dom runz* ⟧
  ⟹ *ri-runs2sigs* (*runz*(*Rb* ↦ (*Resp*, [*A*, *B*], []))) = *ri-runs2sigs runz*
⟨*proof*⟩

lemma *ri-runs2sigs-upd-serv* [*simp*]:
  ⟦ *Rs* ∉ *dom runz* ⟧
  ⟹ *ri-runs2sigs* (*runz*(*Rs* ↦ (*Serv*, [*A*, *B*], [*aNon Na*, *aNum Ts*])))
      = *ri-runs2sigs runz*
⟨*proof*⟩

lemma *ri-runs2sigs-upd-init-some* [*simp*]:
  ⟦ *runz Ra* = *Some* (*Init*, [*A*, *B*], []); *finite* (*dom runz*) ⟧
  ⟹ *ri-runs2sigs* (*runz*(*Ra* ↦ (*Init*, [*A*, *B*], [*aKey Kab*, *aNum Ts*, *aNum Ta*]))) =
    (*ri-runs2sigs runz*)(
      Running [*B*, *A*] (*Kab*, *Ts*, *Ta*) :=
        Suc (*card* (*ri-running runz A B Kab Ts Ta*)))
⟨*proof*⟩

lemma *ri-runs2sigs-upd-resp-some* [*simp*]:
  ⟦ *runz Rb* = *Some* (*Resp*, [*A*, *B*], []); *finite* (*dom runz*) ⟧
  ⟹ *ri-runs2sigs* (*runz*(*Rb* ↦ (*Resp*, [*A*, *B*], [*aKey Kab*, *aNum Ts*, *aNum Ta*]))) =
    (*ri-runs2sigs runz*)(
      Commit [*B*, *A*] (*Kab*, *Ts*, *Ta*) :=
        Suc (*card* (*ri-commit runz A B Kab Ts Ta*)))
⟨*proof*⟩

lemma *ri-runs2sigs-upd-init-some2* [*simp*]:
  ⟦ *runz Ra* = *Some* (*Init*, [*A*, *B*], [*aKey Kab*, *aNum Ts*, *aNum Ta*]) ⟧
  ⟹ *ri-runs2sigs* (*runz*(*Ra* ↦ (*Init*, [*A*, *B*], [*aKey Kab*, *aNum Ts*, *aNum Ta*, *END*]))) =
    ri-runs2sigs runz
⟨*proof*⟩

Refinement proof

lemma *PO-m1-step1-refines-a0-ri-skip*:
  {*R-a0iim1-ri*}
    Id, (*m1-step1 Ra A B Na*)
  {> *R-a0iim1-ri*}
⟨*proof*⟩

lemma *PO-m1-step2-refines-a0-ri-skip*:
  {*R-a0iim1-ri*}
    Id, (*m1-step2 Rb A B*)
  {> *R-a0iim1-ri*}
⟨*proof*⟩

lemma *PO-m1-step3-refines-a0-ri-skip*:

{*R-a0iim1-ri*}
Id, (*m1-step3 Rs A B Kab Na Ts*)
{> *R-a0iim1-ri*}
⟨*proof*⟩

lemma *PO-m1-step4-refines-a0-ri-running*:
{*R-a0iim1-ri* ∩ *UNIV* × *m1-inv0-fin*}
(*a0i-running* [*B*, *A*] (*Kab*, *Ts*, *Ta*)), (*m1-step4 Ra A B Na Kab Ts Ta*)
{> *R-a0iim1-ri*}
⟨*proof*⟩

lemma *PO-m1-step5-refines-a0-ri-commit*:
{*R-a0iim1-ri* ∩ *UNIV* × (*m1-inv1r-cache* ∩ *m1-inv0-fin*)}
(*a0i-commit* [*B*, *A*] (*Kab*, *Ts*, *Ta*)), (*m1-step5 Rb A B Kab Ts Ta*)
{> *R-a0iim1-ri*}
⟨*proof*⟩

lemma *PO-m1-step6-refines-a0-ri-skip*:
{*R-a0iim1-ri*}
Id, (*m1-step6 Ra A B Na Kab Ts Ta*)
{> *R-a0iim1-ri*}
⟨*proof*⟩

lemma *PO-m1-leak-refines-a0-ri-skip*:
{*R-a0iim1-ri*}
Id, (*m1-leak Rs A B Na Ts*)
{> *R-a0iim1-ri*}
⟨*proof*⟩

lemma *PO-m1-tick-refines-a0-ri-skip*:
{*R-a0iim1-ri*}
Id, (*m1-tick T*)
{> *R-a0iim1-ri*}
⟨*proof*⟩

lemma *PO-m1-purge-refines-a0-ri-skip*:
{*R-a0iim1-ri*}
Id, (*m1-purge A*)
{> *R-a0iim1-ri*}
⟨*proof*⟩

All together now...

lemmas *PO-m1-trans-refines-a0-ri-trans* =
PO-m1-step1-refines-a0-ri-skip *PO-m1-step2-refines-a0-ri-skip*
PO-m1-step3-refines-a0-ri-skip *PO-m1-step4-refines-a0-ri-running*
PO-m1-step5-refines-a0-ri-commit *PO-m1-step6-refines-a0-ri-skip*
PO-m1-leak-refines-a0-ri-skip *PO-m1-tick-refines-a0-ri-skip*
PO-m1-purge-refines-a0-ri-skip

lemma *PO-m1-refines-init-a0-ri* [*iff*]:
init m1 ⊆ *R-a0iim1-ri*``(*init a0i*)
⟨*proof*⟩

lemma *PO-m1-refines-trans-a0-ri* [*iff*]:
{*R-a0iim1-ri* ∩ *a0i-inv1-iagree* × (*m1-inv1r-cache* ∩ *m1-inv0-fin*)}
(*trans a0i*), (*trans m1*)
{> *R-a0iim1-ri*}
⟨*proof*⟩

lemma *obs-consistent-med-a0iim1-ri* [*iff*]:
obs-consistent
(*R-a0iim1-ri* ∩ *a0i-inv1-iagree* × (*m1-inv1r-cache* ∩ *m1-inv0-fin*))
med-a0iim1-ri a0i m1
⟨*proof*⟩

Refinement result.

lemma *PO-m1-refines-a0ii-ri* [*iff*]:
refines
(*R-a0iim1-ri* ∩ *a0i-inv1-iagree* × (*m1-inv1r-cache* ∩ *m1-inv0-fin*))
med-a0iim1-ri a0i m1
⟨*proof*⟩

lemma *m1-implements-a0ii-ri*: *implements med-a0iim1-ri a0i m1*
⟨*proof*⟩

inv3 (inherited): Responder and initiator

This is a translation of the agreement property to Level 1. It follows from the refinement and is needed to prove inv4 below.

definition
m1-inv3r-init :: *'x m1-pred*
where
m1-inv3r-init ≡ {*s*. ∀ *A B Rb Kab Ts Ta nlb*.
B ∉ *bad* ⟶ *A* ∉ *bad* ⟶ *Kab* ∉ *Domain* (*leak s*) ⟶
runs s Rb = *Some* (*Resp*, [*A*, *B*], *aKey Kab* # *aNum Ts* # *aNum Ta* # *nlb*) ⟶
(∃ *Ra nla*.
runs s Ra = *Some* (*Init*, [*A*, *B*], *aKey Kab* # *aNum Ts* # *aNum Ta* # *nla*))
}

lemmas *m1-inv3r-initI* = *m1-inv3r-init-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m1-inv3r-initE* [*elim*] = *m1-inv3r-init-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m1-inv3r-initD* = *m1-inv3r-init-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated* −*1*]

Invariance proof.

lemma *PO-m1-inv3r-init* [*iff*]: *reach m1* ⊆ *m1-inv3r-init*
⟨*proof*⟩

inv4: Key freshness for responder

definition
m1-inv4-rfresh :: *'x m1-pred*
where
m1-inv4-rfresh ≡ {*s*. ∀ *Rb1 Rb2 A1 A2 B1 B2 Kab Ts1 Ts2 Ta1 Ta2*.
runs s Rb1 = *Some* (*Resp*, [*A1*, *B1*], [*aKey Kab*, *aNum Ts1*, *aNum Ta1*]) ⟶

runs s Rb2 = Some (Resp, [A2, B2], [aKey Kab, aNum Ts2, aNum Ta2]) ⟶
B1 ∉ *bad* ⟶ *A1* ∉ *bad* ⟶ *Kab* ∉ *Domain (leak s)* ⟶
Rb1 = Rb2
}

lemmas *m1-inv4-rfreshI = m1-inv4-rfresh-def [THEN setc-def-to-intro, rule-format]*
lemmas *m1-inv4-rfreshE [elim] = m1-inv4-rfresh-def [THEN setc-def-to-elim, rule-format]*
lemmas *m1-inv4-rfreshD = m1-inv4-rfresh-def [THEN setc-def-to-dest, rule-format, rotated 1]*

Proof of key freshness for responder. All cases except step5 are straightforward.

lemma *PO-m1-inv4-rfresh-step5*:
{*m1-inv4-rfresh* ∩ *m1-inv3r-init* ∩ *m1-inv2r-serv* ∩ *m1-inv1r-cache* ∩
m1-secrecy ∩ *m1-inv1-ifresh*}
(*m1-step5 Rb A B Kab Ts Ta*)
{> *m1-inv4-rfresh*}
⟨*proof*⟩

lemmas *PO-m1-inv4-rfresh-step5-lemmas =*
PO-m1-inv4-rfresh-step5

lemma *PO-m1-inv4-rfresh-init [iff]*:
init m1 ⊆ *m1-inv4-rfresh*
⟨*proof*⟩

lemma *PO-m1-inv4-rfresh-trans [iff]*:
{*m1-inv4-rfresh* ∩ *m1-inv3r-init* ∩ *m1-inv2r-serv* ∩ *m1-inv1r-cache* ∩
m1-secrecy ∩ *m1-inv1-ifresh*}
trans m1
{> *m1-inv4-rfresh*}
⟨*proof*⟩

lemma *PO-m1-inv4-rfresh [iff]*: *reach m1* ⊆ *m1-inv4-rfresh*
⟨*proof*⟩

lemma *PO-m1-obs-inv4-rfresh [iff]*: *oreach m1* ⊆ *m1-inv4-rfresh*
⟨*proof*⟩

3.4.7 Refinement of *a0i* for initiator/responder

The initiator injectively agrees with the responder on *Kab*, *Ts*, and *Ta*.

Simulation relation

We define two auxiliary functions to reconstruct the signals of the initial model from completed initiator and responder runs.

type-synonym
irsig = key × time × time

abbreviation
ir-running :: [*runs-t, agent, agent, key, time, time*] ⇒ *rid-t set*

where
ir-running runz A B Kab Ts Ta ≡ {*Rb*. ∃ *nl*.
runz Rb = *Some* (*Resp*, [*A*, *B*], *aKey Kab* # *aNum Ts* # *aNum Ta* # *nl*)
}

abbreviation
ir-commit :: [*runs-t*, *agent*, *agent*, *key*, *time*, *time*] ⇒ *rid-t set*
where
ir-commit runz A B Kab Ts Ta ≡ {*Ra*. ∃ *nl*.
runz Ra = *Some* (*Init*, [*A*, *B*], *aKey Kab* # *aNum Ts* # *aNum Ta* # *END* # *nl*)
}

fun
ir-runs2sigs :: *runs-t* ⇒ *risig signal* ⇒ *nat*
where
ir-runs2sigs runz (*Running* [*A*, *B*] (*Kab*, *Ts*, *Ta*)) =
card (*ir-running runz A B Kab Ts Ta*)

| *ir-runs2sigs runz* (*Commit* [*A*, *B*] (*Kab*, *Ts*, *Ta*)) =
card (*ir-commit runz A B Kab Ts Ta*)

| *ir-runs2sigs runz* - = *0*

Simulation relation and mediator function. We map completed initiator and responder runs to commit and running signals, respectively.

definition
med-a0iim1-ir :: *m1-obs* ⇒ *irsig a0i-obs* **where**
med-a0iim1-ir o1 ≡ (| *signals* = *ir-runs2sigs* (*runs o1*), *corrupted* = {} |)

definition
R-a0iim1-ir :: (*irsig a0i-state* × *m1-state*) *set* **where**
R-a0iim1-ir ≡ {(*s*, *t*). *signals s* = *ir-runs2sigs* (*runs t*) ∧ *corrupted s* = {} }

lemmas *R-a0iim1-ir-defs* = *R-a0iim1-ir-def med-a0iim1-ir-def*

Lemmas about the auxiliary functions

lemma *ir-runs2sigs-empty* [*simp*]:
runz = *Map.empty* ⟹ *ir-runs2sigs runz* = (λ*s*. *0*)
⟨*proof*⟩

lemma *ir-commit-finite* [*simp*, *intro*]:
finite (*dom runz*) ⟹ *finite* (*ir-commit runz A B Kab Ts Ta*)
⟨*proof*⟩

Update lemmas

lemma *ir-runs2sigs-upd-init-none* [*simp*]:
⟦ *Ra* ∉ *dom runz* ⟧
⟹ *ir-runs2sigs* (*runz*(*Ra* ↦ (*Init*, [*A*, *B*], []))) = *ir-runs2sigs runz*
⟨*proof*⟩

lemma *ir-runs2sigs-upd-resp-none* [*simp*]:
  ⟦ *Rb* ∉ *dom runz* ⟧
  ⟹ *ir-runs2sigs* (*runz*(*Rb* ↦ (*Resp*, [*A*, *B*], []))) = *ir-runs2sigs runz*
⟨*proof*⟩

lemma *ir-runs2sigs-upd-serv* [*simp*]:
  ⟦ *Rs* ∉ *dom* (*runs y*) ⟧
  ⟹ *ir-runs2sigs* ((*runs y*)(*Rs* ↦ (*Serv*, [*A*, *B*], [*aNon Na*, *aNum Ts*])))
      = *ir-runs2sigs* (*runs y*)
⟨*proof*⟩

lemma *ir-runs2sigs-upd-init-some* [*simp*]:
  ⟦ *runz Ra* = *Some* (*Init*, [*A*, *B*], []) ⟧
  ⟹ *ir-runs2sigs* (*runz*(*Ra* ↦ (*Init*, [*A*, *B*], [*aKey Kab*, *aNum Ts*, *aNum Ta*]))) =
     ir-runs2sigs runz
⟨*proof*⟩

lemma *ir-runs2sigs-upd-resp-some-raw*:
  assumes
    runz Rb = *Some* (*Resp*, [*A*, *B*], [])
    finite (*dom runz*)
  shows
    ir-runs2sigs (*runz*(*Rb* ↦ (*Resp*, [*A*, *B*], [*aKey Kab*, *aNum Ts*, *aNum Ta*]))) *s* =
     ((*ir-runs2sigs runz*)(
       Running [*A*, *B*] (*Kab*, *Ts*, *Ta*) :=
         Suc (*card* (*ir-running runz A B Kab Ts Ta*)))) *s*
  ⟨*proof*⟩

lemma *ir-runs2sigs-upd-resp-some* [*simp*]:
  ⟦ *runz Rb* = *Some* (*Resp*, [*A*, *B*], []); *finite* (*dom runz*) ⟧
  ⟹ *ir-runs2sigs* (*runz*(*Rb* ↦ (*Resp*, [*A*, *B*], [*aKey Kab*, *aNum Ts*, *aNum Ta*]))) =
     (*ir-runs2sigs runz*)(
       Running [*A*, *B*] (*Kab*, *Ts*, *Ta*) :=
         Suc (*card* (*ir-running runz A B Kab Ts Ta*)))
⟨*proof*⟩

lemma *ir-runs2sigs-upd-init-some2-raw*:
  assumes
    runz Ra = *Some* (*Init*, [*A*, *B*], [*aKey Kab*, *aNum Ts*, *aNum Ta*])
    finite (*dom runz*)
  shows
    ir-runs2sigs (*runz*(*Ra* ↦ (*Init*, [*A*, *B*], [*aKey Kab*, *aNum Ts*, *aNum Ta*, *END*]))) *s* =
     ((*ir-runs2sigs runz*)(
        Commit [*A*, *B*] (*Kab*, *Ts*, *Ta*) :=
          Suc (*card* (*ir-commit runz A B Kab Ts Ta*)))) *s*
  ⟨*proof*⟩

lemma *ir-runs2sigs-upd-init-some2* [*simp*]:
  ⟦ *runz Na* = *Some* (*Init*, [*A*, *B*], [*aKey Kab*, *aNum Ts*, *aNum Ta*]); *finite* (*dom runz*) ⟧
  ⟹ *ir-runs2sigs* (*runz*(*Na* ↦ (*Init*, [*A*, *B*], [*aKey Kab*, *aNum Ts*, *aNum Ta*, *END*]))) =
     (*ir-runs2sigs runz*)(
        Commit [*A*, *B*] (*Kab*, *Ts*, *Ta*) :=

Suc (*card* (*ir-commit runz A B Kab Ts Ta*)))
⟨*proof*⟩

Refinement proof

lemma *PO-m1-step1-refines-ir-a0ii-skip*:
{*R-a0iim1-ir*}
Id, (*m1-step1 Ra A B Na*)
{> *R-a0iim1-ir*}
⟨*proof*⟩

lemma *PO-m1-step2-refines-ir-a0ii-skip*:
{*R-a0iim1-ir*}
Id, (*m1-step2 Rb A B*)
{> *R-a0iim1-ir*}
⟨*proof*⟩

lemma *PO-m1-step3-refines-ir-a0ii-skip*:
{*R-a0iim1-ir*}
Id, (*m1-step3 Rs A B Kab Na Ts*)
{> *R-a0iim1-ir*}
⟨*proof*⟩

lemma *PO-m1-step4-refines-ir-a0ii-skip*:
{*R-a0iim1-ir*}
Id, (*m1-step4 Ra A B Na Kab Ts Ta*)
{> *R-a0iim1-ir*}
⟨*proof*⟩

lemma *PO-m1-step5-refines-ir-a0ii-running*:
{*R-a0iim1-ir* ∩ *UNIV* × *m1-inv0-fin*}
(*a0i-running* [*A*, *B*] (*Kab*, *Ts*, *Ta*)), (*m1-step5 Rb A B Kab Ts Ta*)
{> *R-a0iim1-ir*}
⟨*proof*⟩

lemma *PO-m1-step6-refines-ir-a0ii-commit*:
{*R-a0iim1-ir* ∩ *UNIV* × *m1-inv0-fin*}
(*a0n-commit* [*A*, *B*] (*Kab*, *Ts*, *Ta*)), (*m1-step6 Ra A B Na Kab Ts Ta*)
{> *R-a0iim1-ir*}
⟨*proof*⟩

lemma *PO-m1-leak-refines-ir-a0ii-skip*:
{*R-a0iim1-ir*}
Id, (*m1-leak Rs A B Na Ts*)
{> *R-a0iim1-ir*}
⟨*proof*⟩

lemma *PO-m1-tick-refines-ir-a0ii-skip*:
{*R-a0iim1-ir*}
Id, (*m1-tick T*)
{> *R-a0iim1-ir*}
⟨*proof*⟩

lemma *PO-m1-purge-refines-ir-a0ii-skip*:
{*R-a0iim1-ir*}
Id, (*m1-purge A*)
{> *R-a0iim1-ir*}
⟨*proof*⟩

All together now...

lemmas *PO-m1-trans-refines-ir-a0ii-trans* =
PO-m1-step1-refines-ir-a0ii-skip PO-m1-step2-refines-ir-a0ii-skip
PO-m1-step3-refines-ir-a0ii-skip PO-m1-step4-refines-ir-a0ii-skip
PO-m1-step5-refines-ir-a0ii-running PO-m1-step6-refines-ir-a0ii-commit
PO-m1-leak-refines-ir-a0ii-skip PO-m1-tick-refines-ir-a0ii-skip
PO-m1-purge-refines-ir-a0ii-skip

lemma *PO-m1-refines-init-ir-a0ii* [*iff*]:
init m1 ⊆ *R-a0iim1-ir*``(*init a0n*)
⟨*proof*⟩

lemma *PO-m1-refines-trans-ir-a0ii* [*iff*]:
{*R-a0iim1-ir* ∩ *UNIV* × *m1-inv0-fin*}
(*trans a0n*), (*trans m1*)
{> *R-a0iim1-ir*}
⟨*proof*⟩

Observation consistency.

lemma *obs-consistent-med-a0iim1-ir* [*iff*]:
obs-consistent
(*R-a0iim1-ir* ∩ *UNIV* × *m1-inv0-fin*)
med-a0iim1-ir a0n m1
⟨*proof*⟩

Refinement result.

lemma *PO-m1-refines-a0ii-ir* [*iff*]:
refines (*R-a0iim1-ir* ∩ *UNIV* × *m1-inv0-fin*)
med-a0iim1-ir a0n m1
⟨*proof*⟩

lemma *m1-implements-a0ii-ir*: *implements med-a0iim1-ir a0n m1*
⟨*proof*⟩

end

3.5 Abstract Kerberos core protocol (L2)

theory *m2-kerberos* **imports** *m1-kerberos ../Refinement/Channels*
begin

We model an abstract version of the core Kerberos protocol:

$$\begin{array}{lll} \text{M1.} & A \rightarrow S: & A, B, Na \\ \text{M2a.} & S \rightarrow A: & \{Kab, Ts, B, Na\}_{Kas} \\ \text{M2b.} & S \rightarrow B: & \{Kab, Ts, A\}_{Kbs} \\ \text{M3.} & A \rightarrow B: & \{A, Ta\}_{Kab} \\ \text{M4.} & B \rightarrow A: & \{Ta\}_{Kab} \end{array}$$

Message 1 is sent over an insecure channel, the other four (cleartext) messages over secure channels.

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

3.5.1 State

State and observations

record *m2-state* = *m1-state* +
  chan :: *chmsg set* — channel messages

type-synonym
  m2-obs = *m1-state*

definition
  m2-obs :: *m2-state* ⇒ *m2-obs* **where**
  m2-obs s ≡ (|
    runs = *runs s*,
    leak = *leak s*,
    clk = *clk s*,
    cache = *cache s*
  |)

type-synonym
  m2-pred = *m2-state set*

type-synonym
  m2-trans = (*m2-state* × *m2-state*) *set*

3.5.2 Events

Protocol events.

definition — by *A*, refines *m1a-step1*
  m2-step1 :: [*rid-t*, *agent*, *agent*, *nonce*] ⇒ *m2-trans*
where
  m2-step1 Ra A B Na ≡ {(*s*, *s1*).
    — guards:
    Ra ∉ *dom* (*runs s*) ∧ — *Ra* is fresh
    Na = *Ra*$*na* ∧ — generate nonce

    — actions:
    — create initiator thread and send message 1
    s1 = *s*(|
      runs := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [])),

chan := *insert* (*Insec A B* (*Msg* [*aNon Na*])) (*chan s*) — send *M1*
⦈
}

definition — by *B*, refines *m1e-step2*
m2-step2 :: [*rid-t*, *agent*, *agent*] ⇒ *m2-trans*
where
m2-step2 ≡ *m1-step2*

definition — by *Server*, refines *m1e-step3*
m2-step3 ::
[*rid-t*, *agent*, *agent*, *key*, *nonce*, *time*] ⇒ *m2-trans*
where
m2-step3 Rs A B Kab Na Ts ≡ {(*s*, *s1*).

— guards:
Rs ∉ *dom* (*runs s*) ∧ — fresh server run
Kab = *sesK* (*Rs*$*sk*) ∧ — fresh session key
Ts = *clk s* ∧ — fresh timestamp

Insec A B (*Msg* [*aNon Na*]) ∈ *chan s* ∧ — recv *M1*

— actions:
— record key and send messages 2 and 3
s1 = *s*⦇
runs := (*runs s*)(*Rs* ↦ (*Serv*, [*A*, *B*], [*aNon Na*, *aNum Ts*])),
chan := {*Secure Sv A* (*Msg* [*aKey Kab*, *aAgt B*, *aNum Ts*, *aNon Na*]), — send *M2a/b*
Secure Sv B (*Msg* [*aKey Kab*, *aAgt A*, *aNum Ts*])} ∪ *chan s*
⦈
}

definition — by *A*, refines *m1e-step4*
m2-step4 :: [*rid-t*, *agent*, *agent*, *nonce*, *key*, *time*, *time*] ⇒ *m2-trans*
where
m2-step4 Ra A B Na Kab Ts Ta ≡ {(*s*, *s1*).
— guards:
runs s Ra = *Some* (*Init*, [*A*, *B*], []) ∧ — session key not yet recv'd
Na = *Ra*$*na* ∧ — fix nonce
Ta = *clk s* ∧ — fresh timestamp
clk s < *Ts* + *Ls* ∧ — ensure key recentness

Secure Sv A (*Msg* [*aKey Kab*, *aAgt B*, *aNum Ts*, *aNon Na*]) ∈ *chan s* ∧ — recv *M2a*

— actions:
— record session key
s1 = *s*⦇
runs := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [*aKey Kab*, *aNum Ts*, *aNum Ta*])),
chan := *insert* (*dAuth Kab* (*Msg* [*aAgt A*, *aNum Ta*])) (*chan s*) — send *M3*
⦈
}

definition — by *B*, refines *m1e-step5*
m2-step5 :: [*rid-t*, *agent*, *agent*, *key*, *time*, *time*] ⇒ *m2-trans*

where
m2-step5 Rb A B Kab Ts Ta ≡ {(*s*, *s1*).
— guards:
runs s Rb = *Some* (*Resp*, [*A*, *B*], []) ∧ — *Kab* not yet received
Secure Sv B (*Msg* [*aKey Kab*, *aAgt A*, *aNum Ts*]) ∈ *chan s* ∧ — recv *M2b*
dAuth Kab (*Msg* [*aAgt A*, *aNum Ta*]) ∈ *chan s* ∧ — recv *M3*

— ensure freshness of session key
clk s < *Ts* + *Ls* ∧

— check authenticator's validity and replay; 'replays' with fresh authenticator ok!
clk s < *Ta* + *La* ∧
(*B*, *Kab*, *Ta*) ∉ *cache s* ∧

— actions:
— record session key, send message *M4*
s1 = *s*⦇
runs := (*runs s*)(*Rb* ↦ (*Resp*, [*A*, *B*], [*aKey Kab*, *aNum Ts*, *aNum Ta*])),
cache := *insert* (*B*, *Kab*, *Ta*) (*cache s*),
chan := *insert* (*dAuth Kab* (*Msg* [*aNum Ta*])) (*chan s*) — send *M4*
⦈
}

definition — by *A*, refines *m1e-step6*
m2-step6 :: [*rid-t*, *agent*, *agent*, *nonce*, *key*, *time*, *time*] ⇒ *m2-trans*
where
m2-step6 Ra A B Na Kab Ts Ta ≡ {(*s*, *s*′).
runs s Ra = *Some* (*Init*, [*A*, *B*], [*aKey Kab*, *aNum Ts*, *aNum Ta*]) ∧ — key recv'd before
Na = *Ra$na* ∧ — generated nonce

clk s < *Ts* + *Ls* ∧ — check session key's recentness

dAuth Kab (*Msg* [*aNum Ta*]) ∈ *chan s* ∧ — recv *M4*

— actions:
s′ = *s*⦇
runs := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [*aKey Kab*, *aNum Ts*, *aNum Ta*, *END*]))
⦈
}

Clock tick event

definition — refines *m1-tick*
m2-tick :: *time* ⇒ *m2-trans*
where
m2-tick ≡ *m1-tick*

Purge event: purge cache of expired timestamps

definition — refines *m1-purge*
m2-purge :: *agent* ⇒ *m2-trans*
where
m2-purge ≡ *m1-purge*

Intruder events.

definition — refines *m1-leak*
m2-leak :: [*rid-t*, *agent*, *agent*, *nonce*, *time*] ⇒ *m2-trans*
where
m2-leak Rs A B Na Ts ≡ {(*s*, *s1*).
— guards:
runs s Rs = *Some* (*Serv*, [*A*, *B*], [*aNon Na*, *aNum Ts*]) ∧
(*clk s* ≥ *Ts* + *Ls*) ∧ — only compromise 'old' session keys

— actions:
— record session key as leaked;
— intruder sends himself an insecure channel message containing the key
s1 = *s*⦇ *leak* := *insert* (*sesK* (*Rs*$*sk*), *A*, *B*, *Na*, *Ts*) (*leak s*),
chan := *insert* (*Insec undefined undefined* (*Msg* [*aKey* (*sesK* (*Rs*$*sk*))])) (*chan s*) ⦈
}

definition — refines *Id*
m2-fake :: *m2-trans*
where
m2-fake ≡ {(*s*, *s1*).

— actions:
s1 = *s*⦇
— close under fakeable messages
chan := *fake ik0* (*dom* (*runs s*)) (*chan s*)
⦈
}

3.5.3 Transition system

definition
m2-init :: *m2-pred*
where
m2-init ≡ { ⦇
runs = *Map.empty*,
leak = *corrKey* × {*undefined*},
clk = *0*,
cache = {},
chan = {}
⦈ }

definition
m2-trans :: *m2-trans* **where**
m2-trans ≡ (⋃ *A B Ra Rb Rs Na Kab Ts Ta T*.
m2-step1 Ra A B Na ∪
m2-step2 Rb A B ∪
m2-step3 Rs A B Kab Na Ts ∪
m2-step4 Ra A B Na Kab Ts Ta ∪
m2-step5 Rb A B Kab Ts Ta ∪
m2-step6 Ra A B Na Kab Ts Ta ∪
m2-tick T ∪
m2-purge A ∪
m2-leak Rs A B Na Ts ∪
m2-fake ∪

Id
)

definition
m2 :: (*m2-state*, *m2-obs*) *spec* **where**
m2 ≡ (|
init = *m2-init*,
trans = *m2-trans*,
obs = *m2-obs*
|)

lemmas *m2-loc-defs* =
m2-def m2-init-def m2-trans-def m2-obs-def
m2-step1-def m2-step2-def m2-step3-def m2-step4-def m2-step5-def
m2-step6-def m2-tick-def m2-purge-def m2-leak-def m2-fake-def

lemmas *m2-defs* = *m2-loc-defs m1-defs*

3.5.4 Invariants and simulation relation

inv1: Key definedness

All session keys in channel messages stem from existing runs.

definition
m2-inv1-keys :: *m2-state set*
where
m2-inv1-keys ≡ {*s*. ∀ *R*.
aKey (*sesK* (*R*\$*sk*)) ∈ *atoms* (*chan s*) ∨ *sesK* (*R*\$*sk*) ∈ *Domain* (*leak s*) ⟶
R ∈ *dom* (*runs s*)
}

lemmas *m2-inv1-keysI* = *m2-inv1-keys-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv1-keysE* [*elim*] = *m2-inv1-keys-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv1-keysD* = *m2-inv1-keys-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

Invariance proof.

lemma *PO-m2-inv1-keys-init* [*iff*]:
init m2 ⊆ *m2-inv1-keys*
⟨*proof*⟩

lemma *PO-m2-inv1-keys-trans* [*iff*]:
{*m2-inv1-keys*} *trans m2* {> *m2-inv1-keys*}
⟨*proof*⟩

lemma *PO-m2-inv1-keys* [*iff*]: *reach m2* ⊆ *m2-inv1-keys*
⟨*proof*⟩

inv2: Definedness of used keys

definition
m2-inv2-keys-for :: *m2-state set*
where

m2-inv2-keys-for ≡ {*s*. ∀ *R*.
  sesK (*R*$*sk*) ∈ *keys-for* (*chan s*) ⟶ *R* ∈ *dom* (*runs s*)
}

lemmas *m2-inv2-keys-forI* = *m2-inv2-keys-for-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv2-keys-forE* [*elim*] = *m2-inv2-keys-for-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv2-keys-forD* = *m2-inv2-keys-for-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

Invariance proof.

lemma *PO-m2-inv2-keys-for-init* [*iff*]:
  init m2 ⊆ *m2-inv2-keys-for*
⟨*proof*⟩

lemma *PO-m2-inv2-keys-for-trans* [*iff*]:
  {*m2-inv2-keys-for* ∩ *m2-inv1-keys*} *trans m2* {> *m2-inv2-keys-for*}
⟨*proof*⟩

lemma *PO-m2-inv2-keys-for* [*iff*]: *reach m2* ⊆ *m2-inv2-keys-for*
⟨*proof*⟩

inv3a: Session key compromise

A L2 version of a session key comprise invariant. Roughly, it states that adding a set of keys *KK* to the parameter *T* of *extr* does not help the intruder to extract keys other than those in *KK* or extractable without adding *KK*.

definition
  m2-inv3a-sesK-compr :: *m2-state set*
where
  m2-inv3a-sesK-compr ≡ {*s*. ∀ *K KK*.
    ~~*KK* ⊆ *range sesK* ⟶~~
    aKey K ∈ *extr* (*aKey*‘*KK* ∪ *ik0*) (*chan s*) ⟷ (*K* ∈ *KK* ∨ *aKey K* ∈ *extr ik0* (*chan s*))
}

lemmas *m2-inv3a-sesK-comprI* = *m2-inv3a-sesK-compr-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv3a-sesK-comprE* [*elim*] = *m2-inv3a-sesK-compr-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv3a-sesK-comprD* = *m2-inv3a-sesK-compr-def* [*THEN setc-def-to-dest*, *rule-format*]

Additional lemma to get the keys in front

lemmas *insert-commute-aKey* = *insert-commute* [**where** *x*=*aKey K* **for** *K*]

lemmas *m2-inv3a-sesK-compr-simps* =
  m2-inv3a-sesK-comprD
  m2-inv3a-sesK-comprD [**where** *KK*=*insert Kab KK* **for** *Kab KK*, *simplified*]
  m2-inv3a-sesK-comprD [**where** *KK*={*Kab*} **for** *Kab*, *simplified*]
  insert-commute-aKey

lemma *PO-m2-inv3a-sesK-compr-init* [*iff*]:
  init m2 ⊆ *m2-inv3a-sesK-compr*
⟨*proof*⟩

lemma *PO-m2-inv3a-sesK-compr-trans* [*iff*]:

{*m2-inv3a-sesK-compr*} *trans m2* {> *m2-inv3a-sesK-compr*}
⟨*proof*⟩

lemma *PO-m2-inv3a-sesK-compr* [*iff*]: *reach m2* ⊆ *m2-inv3a-sesK-compr*
⟨*proof*⟩

inv3b: Leakage of old session keys

Only old session keys are leaked to the intruder.

definition
m2-inv3b-leak :: *m2-state set*
where
m2-inv3b-leak ≡ {*s*. ∀ *Rs A B Na Ts*.
(*sesK* (*Rs$sk*), *A*, *B*, *Na*, *Ts*) ∈ *leak s* ⟶ *clk s* ≥ *Ts* + *Ls*
}

lemmas *m2-inv3b-leakI* = *m2-inv3b-leak-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv3b-leakE* [*elim*] = *m2-inv3b-leak-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv3b-leakD* = *m2-inv3b-leak-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

Invariance proof.

lemma *PO-m2-inv3b-leak-init* [*iff*]:
init m2 ⊆ *m2-inv3b-leak*
⟨*proof*⟩

lemma *PO-m2-inv3b-leak-trans* [*iff*]:
{*m2-inv3b-leak* ∩ *m2-inv1-keys*} *trans m2* {> *m2-inv3b-leak*}
⟨*proof*⟩

lemma *PO-m2-inv3b-leak* [*iff*]: *reach m2* ⊆ *m2-inv3b-leak*
⟨*proof*⟩

inv3: Lost session keys

inv3: Lost but not leaked session keys generated by the server for at least one bad agent. This invariant is needed in the proof of the strengthening of the authorization guards in steps 4 and 5 (e.g., *Kab* ∉ *Domain* (*leaks s*) ⟶ (*Kab*, *A*) ∈ *azC* (*runs s*) for the initiator's step4).

definition
m2-inv3-extrKey :: *m2-state set*
where
m2-inv3-extrKey ≡ {*s*. ∀ *K*.
aKey K ∈ *extr ik0* (*chan s*) ⟶
(*K* ∈ *corrKey* ∧ *K* ∈ *Domain* (*leak s*)) ∨
(∃ *R A′ B′ Na′ Ts′*. *K* = *sesK* (*R$sk*) ∧
runs s R = *Some* (*Serv*, [*A′*, *B′*], [*aNon Na′*, *aNum Ts′*]) ∧
(*A′* ∈ *bad* ∨ *B′* ∈ *bad* ∨ (*K*, *A′*, *B′*, *Na′*, *Ts′*) ∈ *leak s*))
}

lemmas *m2-inv3-extrKeyI* = *m2-inv3-extrKey-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv3-extrKeyE* [*elim*] = *m2-inv3-extrKey-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv3-extrKeyD* = *m2-inv3-extrKey-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

lemma *PO-m2-inv3-extrKey-init* [*iff*]:
  init m2 ⊆ *m2-inv3-extrKey*
⟨*proof*⟩

lemma *PO-m2-inv3-extrKey-trans* [*iff*]:
  {*m2-inv3-extrKey* ∩ *m2-inv3a-sesK-compr*}
      trans m2
   {> *m2-inv3-extrKey*}
⟨*proof*⟩

lemma *PO-m2-inv3-extrKey* [*iff*]: *reach m2* ⊆ *m2-inv3-extrKey*
⟨*proof*⟩

inv4: Messages M2a/M2b for good agents and server state

inv4: Secure messages to honest agents and server state; one variant for each of M2a and M2b. These invariants establish guard strengthening for server authentication by the initiator and the responder.

definition
  m2-inv4-M2a :: *m2-state set*
where
  m2-inv4-M2a ≡ {*s*. ∀ *A B Kab Ts Na*.
    Secure Sv A (*Msg* [*aKey Kab*, *aAgt B*, *aNum Ts*, *aNon Na*]) ∈ *chan s* ⟶ *A* ∈ *good* ⟶
      (∃ *Rs*. *Kab* = *sesK* (*Rs*$*sk*) ∧
         runs s Rs = *Some* (*Serv*, [*A*, *B*], [*aNon Na*, *aNum Ts*]))
  }

definition
  m2-inv4-M2b :: *m2-state set*
where
  m2-inv4-M2b ≡ {*s*. ∀ *A B Kab Ts*.
    Secure Sv B (*Msg* [*aKey Kab*, *aAgt A*, *aNum Ts*]) ∈ *chan s* ⟶ *B* ∈ *good* ⟶
      (∃ *Rs Na*. *Kab* = *sesK* (*Rs*$*sk*) ∧
         runs s Rs = *Some* (*Serv*, [*A*, *B*], [*aNon Na*, *aNum Ts*]))
  }

lemmas *m2-inv4-M2aI* = *m2-inv4-M2a-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv4-M2aE* [*elim*] = *m2-inv4-M2a-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv4-M2aD* = *m2-inv4-M2a-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

lemmas *m2-inv4-M2bI* = *m2-inv4-M2b-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv4-M2bE* [*elim*] = *m2-inv4-M2b-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv4-M2bD* = *m2-inv4-M2b-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

Invariance proofs.

lemma *PO-m2-inv4-M2a-init* [*iff*]:
  init m2 ⊆ *m2-inv4-M2a*
⟨*proof*⟩

lemma *PO-m2-inv4-M2a-trans* [*iff*]:
  {*m2-inv4-M2a*} *trans m2* {> *m2-inv4-M2a*}

⟨*proof*⟩

lemma *PO-m2-inv4-M2a* [*iff*]: *reach m2* ⊆ *m2-inv4-M2a*
⟨*proof*⟩

lemma *PO-m2-inv4-M2b-init* [*iff*]:
  init m2 ⊆ *m2-inv4-M2b*
⟨*proof*⟩

lemma *PO-m2-inv4-M2b-trans* [*iff*]:
  {*m2-inv4-M2b*} *trans m2* {> *m2-inv4-M2b*}
⟨*proof*⟩

lemma *PO-m2-inv4-M2b* [*iff*]: *reach m2* ⊆ *m2-inv4-M2b*
⟨*proof*⟩

Consequence needed in proof of inv8/step5 and inv9/step4: The session key uniquely identifies other fields in M2a and M2b, provided it is secret.

lemma *m2-inv4-M2a-M2b-match*:
  ⟦ *Secure Sv A′* (*Msg* [*aKey Kab*, *aAgt B′*, *aNum Ts′*, *aNon N*]) ∈ *chan s*;
    Secure Sv B (*Msg* [*aKey Kab*, *aAgt A*, *aNum Ts*]) ∈ *chan s*;
    aKey Kab ∉ *extr ik0* (*chan s*); *s* ∈ *m2-inv4-M2a*; *s* ∈ *m2-inv4-M2b* ⟧
  ⟹ *A* = *A′* ∧ *B* = *B′* ∧ *Ts* = *Ts′*
⟨*proof*⟩

More consequences of invariants. Needed in ref/step4 and ref/step5 respectively to show the strengthening of the authorization guards.

lemma *m2-inv34-M2a-authorized*:
  assumes *Secure Sv A* (*Msg* [*aKey K*, *aAgt B*, *aNum T*, *aNon N*]) ∈ *chan s*
      s ∈ *m2-inv4-M2a s* ∈ *m2-inv3-extrKey*
      K ∉ *Domain* (*leak s*)
  shows (*K*, *A*) ∈ *azC* (*runs s*)
⟨*proof*⟩

lemma *m2-inv34-M2b-authorized*:
  assumes *Secure Sv B* (*Msg* [*aKey K*, *aAgt A*, *aNum T*]) ∈ *chan s*
      s ∈ *m2-inv4-M2b s* ∈ *m2-inv3-extrKey*
      K ∉ *Domain* (*leak s*)
  shows (*K*, *B*) ∈ *azC* (*runs s*)
  ⟨*proof*⟩

inv5 (derived): Key secrecy for server

inv5: Key secrecy from server perspective. This invariant links the abstract notion of key secrecy to the intruder key knowledge.

definition
  m2-inv5-ikk-sv :: *m2-state set*
where
  m2-inv5-ikk-sv ≡ {*s*. ∀ *R A B Na Ts*.
    runs s R = *Some* (*Serv*, [*A*, *B*], [*aNon Na*, *aNum Ts*]) ⟶ *A* ∈ *good* ⟶ *B* ∈ *good* ⟶

aKey (*sesK* (*R$sk*)) ∈ *extr ik0* (*chan s*) ⟶
    (*sesK* (*R$sk*), *A*, *B*, *Na*, *Ts*) ∈ *leak s*
}

lemmas *m2-inv5-ikk-svI* = *m2-inv5-ikk-sv-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv5-ikk-svE* [*elim*] = *m2-inv5-ikk-sv-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv5-ikk-svD* = *m2-inv5-ikk-sv-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

Invariance proof. This invariant follows from *m2-inv3-extrKey*.

lemma *m2-inv5-ikk-sv-derived*:
  s ∈ *m2-inv3-extrKey* ⟹ *s* ∈ *m2-inv5-ikk-sv*
⟨*proof*⟩

lemma *PO-m2-inv5-ikk-sv* [*iff*]: *reach m2* ⊆ *m2-inv5-ikk-sv*
⟨*proof*⟩

inv6 (derived): Key secrecy for initiator

This invariant is derivable (see below).

definition
  m2-inv6-ikk-init :: *m2-state set*
where
  m2-inv6-ikk-init ≡ {*s*. ∀ *A B Ra K Ts nl*.
    runs s Ra = *Some* (*Init*, [*A*, *B*], *aKey K* # *aNum Ts* # *nl*) ⟶ *A* ∈ *good* ⟶ *B* ∈ *good* ⟶
    aKey K ∈ *extr ik0* (*chan s*) ⟶
      (*K*, *A*, *B*, *Ra$na*, *Ts*) ∈ *leak s*
  }

lemmas *m2-inv6-ikk-initI* = *m2-inv6-ikk-init-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv6-ikk-initE* [*elim*] = *m2-inv6-ikk-init-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv6-ikk-initD* = *m2-inv6-ikk-init-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

inv7 (derived): Key secrecy for responder

This invariant is derivable (see below).

definition
  m2-inv7-ikk-resp :: *m2-state set*
where
  m2-inv7-ikk-resp ≡ {*s*. ∀ *A B Rb K Ts nl*.
    runs s Rb = *Some* (*Resp*, [*A*, *B*], *aKey K* # *aNum Ts* # *nl*) ⟶ *A* ∈ *good* ⟶ *B* ∈ *good* ⟶
    aKey K ∈ *extr ik0* (*chan s*) ⟶
      (∃ *Na*. (*K*, *A*, *B*, *Na*, *Ts*) ∈ *leak s*)
  }

lemmas *m2-inv7-ikk-respI* = *m2-inv7-ikk-resp-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv7-ikk-respE* [*elim*] = *m2-inv7-ikk-resp-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv7-ikk-respD* = *m2-inv7-ikk-resp-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

inv8: Relating M4 to the responder state

This invariant relates message M4 from the responder to the responder's state. It is required in the refinement of step 6 to prove that the initiator agrees with the responder on (A, B, Ta, Kab).

definition
  m2-inv8-M4 :: *m2-state set*
where
  $m2\text{-}inv8\text{-}M4 \equiv \{s.\ \forall\, Kab\ A\ B\ Ts\ Ta\ N.$
    $Secure\ Sv\ A\ (Msg\ [aKey\ Kab,\ aAgt\ B,\ aNum\ Ts,\ aNon\ N]) \in chan\ s \longrightarrow$
    $dAuth\ Kab\ (Msg\ [aNum\ Ta]) \in chan\ s \longrightarrow$
    $aKey\ Kab \notin extr\ ik0\ (chan\ s) \longrightarrow$
      $(\exists\, Rb.\ runs\ s\ Rb = Some\ (Resp,\ [A,\ B],\ [aKey\ Kab,\ aNum\ Ts,\ aNum\ Ta]))$
  $\}$

lemmas *m2-inv8-M4I* = *m2-inv8-M4-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv8-M4E* [*elim*] = *m2-inv8-M4-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv8-M4D* = *m2-inv8-M4-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

Invariance proof.

lemma *PO-m2-inv8-M4-init* [*iff*]:
  $init\ m2 \subseteq m2\text{-}inv8\text{-}M4$
⟨*proof*⟩

lemma *PO-m2-inv8-M4-trans* [*iff*]:
  $\{m2\text{-}inv8\text{-}M4 \cap m2\text{-}inv4\text{-}M2a \cap m2\text{-}inv4\text{-}M2b \cap m2\text{-}inv3a\text{-}sesK\text{-}compr \cap m2\text{-}inv2\text{-}keys\text{-}for\}$
    trans m2
  $\{> m2\text{-}inv8\text{-}M4\}$
⟨*proof*⟩

lemma *PO-m2-inv8-M4* [*iff*]: $reach\ m2 \subseteq m2\text{-}inv8\text{-}M4$
⟨*proof*⟩

inv9a: Relating the initiator state to M2a

definition
  m2-inv9a-init-M2a :: *m2-state set*
where
  $m2\text{-}inv9a\text{-}init\text{-}M2a \equiv \{s.\ \forall\, A\ B\ Ra\ Kab\ Ts\ z.$
    $runs\ s\ Ra = Some\ (Init,\ [A,\ B],\ aKey\ Kab\ \#\ aNum\ Ts\ \#\ z) \longrightarrow$
      $Secure\ Sv\ A\ (Msg\ [aKey\ Kab,\ aAgt\ B,\ aNum\ Ts,\ aNon\ (Ra\$na)]) \in chan\ s$
  $\}$

lemmas *m2-inv9a-init-M2aI* = *m2-inv9a-init-M2a-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv9a-init-M2aE* [*elim*] = *m2-inv9a-init-M2a-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv9a-init-M2aD* = *m2-inv9a-init-M2a-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

Invariance proof.

lemma *PO-m2-inv9a-init-M2a-init* [*iff*]:
  $init\ m2 \subseteq m2\text{-}inv9a\text{-}init\text{-}M2a$
⟨*proof*⟩

lemma *PO-m2-inv9a-init-M2a-trans* [*iff*]:
  $\{m2\text{-}inv9a\text{-}init\text{-}M2a\}\ trans\ m2\ \{> m2\text{-}inv9a\text{-}init\text{-}M2a\}$
⟨*proof*⟩

lemma *PO-m2-inv9a-init-M2a* [*iff*]: *reach m2* ⊆ *m2-inv9a-init-M2a*
⟨*proof*⟩

inv9: Relating M3 to the initiator state

This invariant relates message M3 to the initiator's state. It is required in step 5 of the refinement to prove that the initiator agrees with the responder on (A, B, Ta, Kab).

definition
  m2-inv9-M3 :: *m2-state set*
where
  m2-inv9-M3 ≡ {*s*. ∀ *Kab A B Ts Ta*.
    Secure Sv B (*Msg* [*aKey Kab*, *aAgt A*, *aNum Ts*]) ∈ *chan s* ⟶
    dAuth Kab (*Msg* [*aAgt A*, *aNum Ta*]) ∈ *chan s* ⟶
    aKey Kab ∉ *extr ik0* (*chan s*) ⟶ ~~*A* ∉ *bad* ∧ *B* ∉ *bad* ⟶~~
      (∃ *Ra nl*. *runs s Ra* = *Some* (*Init*, [*A*, *B*], *aKey Kab* # *aNum Ts* # *aNum Ta* # *nl*))
  }

lemmas *m2-inv9-M3I* = *m2-inv9-M3-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv9-M3E* [*elim*] = *m2-inv9-M3-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv9-M3D* = *m2-inv9-M3-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

Invariance proof.

lemma *PO-m2-inv9-M3-init* [*iff*]:
  init m2 ⊆ *m2-inv9-M3*
⟨*proof*⟩

lemma *PO-m2-inv9-M3-trans* [*iff*]:
  {*m2-inv9-M3* ∩ *m2-inv4-M2a* ∩ *m2-inv4-M2b* ∩ *m2-inv3a-sesK-compr* ∩ *m2-inv2-keys-for*}
    trans m2
  {> *m2-inv9-M3*}
⟨*proof*⟩

lemma *PO-m2-inv9-M3* [*iff*]: *reach m2* ⊆ *m2-inv9-M3*
⟨*proof*⟩

3.5.5 Refinement

The simulation relation. This is a pure superposition refinement.

definition
  R12 :: (*m1-state* × *m2-state*) *set* **where**
  R12 ≡ {(*s*, *t*). *runs s* = *runs t* ∧ *leak s* = *leak t* ∧ *clk s* = *clk t* ∧ *cache s* = *cache t*}

The mediator function is the identity.

definition
  med21 :: *m2-obs* ⇒ *m1-obs* **where**
  med21 = *id*

Refinement proof.

lemma *PO-m2-step1-refines-m1-step1*:
  {*R12*}

(*m1-step1 Ra A B Na*), (*m2-step1 Ra A B Na*)
{> *R12*}
⟨*proof*⟩

lemma *PO-m2-step2-refines-m1-step2*:
{*R12*}
(*m1-step2 Rb A B*), (*m2-step2 Rb A B*)
{> *R12*}
⟨*proof*⟩

lemma *PO-m2-step3-refines-m1-step3*:
{*R12*}
(*m1-step3 Rs A B Kab Na Ts*), (*m2-step3 Rs A B Kab Na Ts*)
{> *R12*}
⟨*proof*⟩

lemma *PO-m2-step4-refines-m1-step4*:
{*R12* ∩ *UNIV* × (*m2-inv4-M2a* ∩ *m2-inv3-extrKey*)}
(*m1-step4 Ra A B Na Kab Ts Ta*), (*m2-step4 Ra A B Na Kab Ts Ta*)
{> *R12*}
⟨*proof*⟩

lemma *PO-m2-step5-refines-m1-step5*:
{*R12* ∩ *UNIV*
× (*m2-inv9-M3* ∩ *m2-inv5-ikk-sv* ∩ *m2-inv4-M2b* ∩ *m2-inv3-extrKey* ∩ *m2-inv3b-leak*)}
(*m1-step5 Rb A B Kab Ts Ta*), (*m2-step5 Rb A B Kab Ts Ta*)
{> *R12*}
⟨*proof*⟩

lemma *PO-m2-step6-refines-m1-step6*:
{*R12* ∩ *UNIV*
× (*m2-inv9a-init-M2a* ∩ *m2-inv8-M4* ∩ *m2-inv5-ikk-sv* ∩ *m2-inv4-M2a* ∩ *m2-inv3b-leak*)}
(*m1-step6 Ra A B Na Kab Ts Ta*), (*m2-step6 Ra A B Na Kab Ts Ta*)
{> *R12*}
⟨*proof*⟩

lemma *PO-m2-tick-refines-m1-tick*:
{*R12*}
(*m1-tick T*), (*m2-tick T*)
{> *R12*}
⟨*proof*⟩

lemma *PO-m2-purge-refines-m1-purge*:
{*R12*}
(*m1-purge A*), (*m2-purge A*)
{> *R12*}
⟨*proof*⟩

lemma *PO-m2-leak-refines-leak*:
{*R12*}
m1-leak Rs A B Na Ts, *m2-leak Rs A B Na Ts*
{> *R12*}
⟨*proof*⟩

lemma *PO-m2-fake-refines-skip*:
{*R12*}
Id, *m2-fake*
{> *R12*}
⟨*proof*⟩

All together now...

lemmas *PO-m2-trans-refines-m1-trans* =
PO-m2-step1-refines-m1-step1 PO-m2-step2-refines-m1-step2
PO-m2-step3-refines-m1-step3 PO-m2-step4-refines-m1-step4
PO-m2-step5-refines-m1-step5 PO-m2-step6-refines-m1-step6
PO-m2-tick-refines-m1-tick PO-m2-purge-refines-m1-purge
PO-m2-leak-refines-leak PO-m2-fake-refines-skip

lemma *PO-m2-refines-init-m1* [*iff*]:
init m2 ⊆ *R12*``(*init m1*)
⟨*proof*⟩

lemma *PO-m2-refines-trans-m1* [*iff*]:
{*R12* ∩
UNIV × (*m2-inv9-M3* ∩ *m2-inv9a-init-M2a* ∩ *m2-inv8-M4* ∩
m2-inv4-M2b ∩ *m2-inv4-M2a* ∩ *m2-inv3-extrKey* ∩ *m2-inv3b-leak*)}
(*trans m1*), (*trans m2*)
{> *R12*}
⟨*proof*⟩

lemma *PO-obs-consistent-R12* [*iff*]:
obs-consistent R12 med21 m1 m2
⟨*proof*⟩

Refinement result.

lemma *m2-refines-m1* [*iff*]:
refines
(*R12* ∩
(*UNIV* ×
(*m2-inv9-M3* ∩ *m2-inv9a-init-M2a* ∩ *m2-inv8-M4* ∩
m2-inv4-M2b ∩ *m2-inv4-M2a* ∩ *m2-inv3-extrKey* ∩ *m2-inv3b-leak* ∩
m2-inv3a-sesK-compr ∩ *m2-inv2-keys-for* ∩ *m2-inv1-keys*)))
med21 m1 m2
⟨*proof*⟩

lemma *m2-implements-m1* [*iff*]:
implements med21 m1 m2
⟨*proof*⟩

3.5.6 Inherited and derived invariants

Show preservation of invariants *m1-inv2i-serv* and *m1-inv2r-serv* from *m1*.

lemma *PO-m2-sat-m1-inv2i-serv* [*iff*]: *reach m2* ⊆ *m1-inv2i-serv*
⟨*proof*⟩

lemma *PO-m2-sat-m1-inv2r-serv* [*iff*]: *reach m2* ⊆ *m1-inv2r-serv*
⟨*proof*⟩

Now we derive the L2 key secrecy invariants for the initiator and the responder (see above for the definitions).

lemma *PO-m2-inv6-init-ikk* [*iff*]: *reach m2* ⊆ *m2-inv6-ikk-init*
⟨*proof*⟩

lemma *PO-m2-inv6-resp-ikk* [*iff*]: *reach m2* ⊆ *m2-inv7-ikk-resp*
⟨*proof*⟩

end

3.6 Core Kerberos, "parallel" variant (L3)

theory *m3-kerberos-par* **imports** *m2-kerberos ../Refinement/Message*
begin

We model a direct implementation of the channel-based core Kerberos protocol at Level 2 without ticket forwarding:

M1.	$A \rightarrow S:$	A, B, Na
M2a.	$S \rightarrow A:$	$\{Kab, B, Ts, Na\}_{Kas}$
M2b.	$S \rightarrow B:$	$\{Kab, A, Ts\}_{Kbs}$
M3.	$A \rightarrow B:$	$\{A, Ta\}_{Kab}$
M4.	$B \rightarrow A:$	$\{Ta\}_{Kab}$

Proof tool configuration. Avoid annoying automatic unfolding of *dom*.

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

3.6.1 Setup

Now we can define the initial key knowledge.

overloading *ltkeySetup′* ≡ *ltkeySetup* **begin**
definition *ltkeySetup-def*: *ltkeySetup′* ≡ {(*sharK C*, *A*) | *C A*. *A* = *C* ∨ *A* = *Sv*}
end

lemma *corrKey-shrK-bad* [*simp*]: *corrKey* = *shrK‘bad*
⟨*proof*⟩

3.6.2 State

The secure channels are star-shaped to/from the server. Therefore, we have only one agent in the relation.

record *m3-state* = *m1-state* +

IK :: *msg set* — intruder knowledge

Observable state: *runs*, *m1-state.leak*, *clk*, and *cache*.

type-synonym
m3-obs = *m2-obs*

definition
m3-obs :: *m3-state* ⇒ *m3-obs* **where**
m3-obs s ≡ ⦇ *runs* = *runs s*, *leak* = *leak s*, *clk* = *clk s*, *cache* = *cache s* ⦈

type-synonym
m3-pred = *m3-state set*

type-synonym
m3-trans = (*m3-state* × *m3-state*) *set*

3.6.3 Events

Protocol events.

definition — by *A*, refines *m2-step1*
m3-step1 :: [*rid-t*, *agent*, *agent*, *nonce*] ⇒ *m3-trans*
where
m3-step1 Ra A B Na ≡ {(*s*, *s1*).
— guards:
Ra ∉ *dom* (*runs s*) ∧ — *Ra* is fresh
Na = *Ra*$*na* ∧ — generated nonce

— actions:
s1 = *s*⦇
runs := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [])),
IK := *insert* ⦃*Agent A*, *Agent B*, *Nonce Na*⦄ (*IK s*) — send *M1*
⦈
}

definition — by *B*, refines *m2-step2*
m3-step2 :: [*rid-t*, *agent*, *agent*] ⇒ *m3-trans*
where
m3-step2 ≡ *m1-step2*

definition — by *Server*, refines *m2-step3*
m3-step3 :: [*rid-t*, *agent*, *agent*, *key*, *nonce*, *time*] ⇒ *m3-trans*
where
m3-step3 Rs A B Kab Na Ts ≡ {(*s*, *s1*).
— guards:
Rs ∉ *dom* (*runs s*) ∧ — fresh server run
Kab = *sesK* (*Rs*$*sk*) ∧ — fresh session key

⦃*Agent A*, *Agent B*, *Nonce Na*⦄ ∈ *IK s* ∧ — recv *M1*
Ts = *clk s* ∧ — fresh timestamp

— actions:
— record session key and send *M2*

$s1 = s⦇$
$runs := (runs\ s)(Rs \mapsto (Serv, [A, B], [aNon\ Na, aNum\ Ts])),$
$IK := insert\ (Crypt\ (shrK\ A)\ \{|Key\ Kab, Agent\ B, Number\ Ts, Nonce\ Na|\})$
$(insert\ (Crypt\ (shrK\ B)\ \{|Key\ Kab, Agent\ A, Number\ Ts|\})\ (IK\ s))$
$⦈$
}

definition — by A, refines *m2-step4*
$m3\text{-}step4 :: [rid\text{-}t, agent, agent, nonce, key, time, time] \Rightarrow m3\text{-}trans$
where
$m3\text{-}step4\ Ra\ A\ B\ Na\ Kab\ Ts\ Ta \equiv \{(s, s1).$
— guards:
$runs\ s\ Ra = Some\ (Init, [A, B], []) \wedge$ — key not yet recv'd
$Na = Ra\$na \wedge$ — generated nonce

$Crypt\ (shrK\ A)$ — recv *M2a*
$\{|Key\ Kab, Agent\ B, Number\ Ts, Nonce\ Na|\} \in IK\ s \wedge$

— read current time
$Ta = clk\ s \wedge$

— check freshness of session key
$clk\ s < Ts + Ls \wedge$

— actions:
— record session key and send *M3*
$s1 = s⦇$
$runs := (runs\ s)(Ra \mapsto (Init, [A, B], [aKey\ Kab, aNum\ Ts, aNum\ Ta])),$
$IK := insert\ (Crypt\ Kab\ \{|Agent\ A, Number\ Ta|\})\ (IK\ s)$ — *M3*
$⦈$
}

definition — by B, refines *m2-step5*
$m3\text{-}step5 :: [rid\text{-}t, agent, agent, key, time, time] \Rightarrow m3\text{-}trans$
where
$m3\text{-}step5\ Rb\ A\ B\ Kab\ Ts\ Ta \equiv \{(s, s1).$
— guards:
$runs\ s\ Rb = Some\ (Resp, [A, B], []) \wedge$ — key not yet recv'd

$Crypt\ (shrK\ B)\ \{|Key\ Kab, Agent\ A, Number\ Ts|\} \in IK\ s \wedge$ — recv *M2b*
$Crypt\ Kab\ \{|Agent\ A, Number\ Ta|\} \in IK\ s \wedge$ — recv *M3*

— ensure freshness of session key
$clk\ s < Ts + Ls \wedge$

— check authenticator's validity and replay; 'replays' with fresh authenticator ok!
$clk\ s < Ta + La \wedge$
$(B, Kab, Ta) \notin cache\ s \wedge$

— actions:
— record session key
$s1 = s⦇$
$runs := (runs\ s)(Rb \mapsto (Resp, [A, B], [aKey\ Kab, aNum\ Ts, aNum\ Ta])),$

cache := *insert* (*B*, *Kab*, *Ta*) (*cache s*),
IK := *insert* (*Crypt Kab* (*Number Ta*)) (*IK s*) — send *M4*
⦈
}

definition — by *A*, refines *m2-step6*
m3-step6 :: [*rid-t*, *agent*, *agent*, *nonce*, *key*, *time*, *time*] ⇒ *m3-trans*
where
m3-step6 Ra A B Na Kab Ts Ta ≡ {(*s*, *s'*).
— guards:
runs s Ra = *Some* (*Init*, [*A*, *B*], [*aKey Kab*, *aNum Ts*, *aNum Ta*]) ∧ — knows key
Na = *Ra*\$*na* ∧ — generated nonce
clk s < *Ts* + *Ls* ∧ — check session key's recentness

Crypt Kab (*Number Ta*) ∈ *IK s* ∧ — recv *M4*

— actions:
s' = *s*⦇
runs := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [*aKey Kab*, *aNum Ts*, *aNum Ta*, *END*]))
⦈
}

Clock tick event

definition — refines *m2-tick*
m3-tick :: *time* ⇒ *m3-trans*
where
m3-tick ≡ *m1-tick*

Purge event: purge cache of expired timestamps

definition — refines *m2-purge*
m3-purge :: *agent* ⇒ *m3-trans*
where
m3-purge ≡ *m1-purge*

Session key compromise.

definition — refines *m2-leak*
m3-leak :: [*rid-t*, *agent*, *agent*, *nonce*, *time*] ⇒ *m3-trans*
where
m3-leak Rs A B Na Ts ≡ {(*s*, *s1*).
— guards:
runs s Rs = *Some* (*Serv*, [*A*, *B*], [*aNon Na*, *aNum Ts*]) ∧
(*clk s* ≥ *Ts* + *Ls*) ∧ — only compromise 'old' session keys!

— actions:
— record session key as leaked and add it to intruder knowledge
s1 = *s*⦇ *leak* := *insert* (*sesK* (*Rs*\$*sk*), *A*, *B*, *Na*, *Ts*) (*leak s*),
IK := *insert* (*Key* (*sesK* (*Rs*\$*sk*))) (*IK s*) ⦈
}

Intruder fake event. The following "Dolev-Yao" event generates all intruder-derivable messages.

definition — refines *m2-fake*

m3-DY-fake :: *m3-trans*
where
m3-DY-fake ≡ {(*s*, *s1*).

— actions:
s1 = *s*(| *IK* := *synth* (*analz* (*IK* *s*)) |) — take DY closure
}

3.6.4 Transition system

definition
m3-init :: *m3-pred*
where
m3-init ≡ { (|
runs = *Map.empty*,
leak = *shrK*'*bad* × {*undefined*},
clk = *0*,
cache = {},
IK = *Key*'*shrK*'*bad*
|) }

definition
m3-trans :: *m3-trans* **where**
m3-trans ≡ ($\bigcup$ *A B Ra Rb Rs Na Kab Ts Ta T*.
m3-step1 Ra A B Na ∪
m3-step2 Rb A B ∪
m3-step3 Rs A B Kab Na Ts ∪
m3-step4 Ra A B Na Kab Ts Ta ∪
m3-step5 Rb A B Kab Ts Ta ∪
m3-step6 Ra A B Na Kab Ts Ta ∪
m3-tick T ∪
m3-purge A ∪
m3-leak Rs A B Na Ts ∪
m3-DY-fake ∪
Id
)

definition
m3 :: (*m3-state*, *m3-obs*) *spec* **where**
m3 ≡ (|
init = *m3-init*,
trans = *m3-trans*,
obs = *m3-obs*
|)

lemmas *m3-loc-defs* =
m3-def m3-init-def m3-trans-def m3-obs-def
m3-step1-def m3-step2-def m3-step3-def m3-step4-def m3-step5-def
m3-step6-def m3-tick-def m3-purge-def m3-leak-def m3-DY-fake-def

lemmas *m3-defs* = *m3-loc-defs m2-defs*

3.6.5 Invariants

Specialized injection that we can apply more aggressively.

lemmas *analz-Inj-IK* = *analz.Inj* [**where** *H*=*IK s* **for** *s*]
lemmas *parts-Inj-IK* = *parts.Inj* [**where** *H*=*IK s* **for** *s*]

declare *parts-Inj-IK* [*dest*!]

declare *analz-into-parts* [*dest*]

inv1: Secrecy of pre-distributed shared keys

inv1: Secrecy of long-term keys

definition
  m3-inv1-lkeysec :: *m3-pred*
where
  m3-inv1-lkeysec ≡ {*s*. ∀ *C*.
    (*Key* (*shrK C*) ∈ *parts* (*IK s*) ⟶ *C* ∈ *bad*) ∧
    (*C* ∈ *bad* ⟶ *Key* (*shrK C*) ∈ *IK s*)
  }

lemmas *m3-inv1-lkeysecI* = *m3-inv1-lkeysec-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv1-lkeysecE* [*elim*] = *m3-inv1-lkeysec-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv1-lkeysec-dest* = *m3-inv1-lkeysec-def* [*THEN setc-def-to-dest*, *rule-format*]

Invariance proof.

lemma *PO-m3-inv1-lkeysec-init* [*iff*]:
  init m3 ⊆ *m3-inv1-lkeysec*
⟨*proof*⟩

lemma *PO-m3-inv1-lkeysec-trans* [*iff*]:
  {*m3-inv1-lkeysec*} *trans m3* {> *m3-inv1-lkeysec*}
⟨*proof*⟩

lemma *PO-m3-inv1-lkeysec* [*iff*]: *reach m3* ⊆ *m3-inv1-lkeysec*
⟨*proof*⟩

Useful simplifier lemmas

lemma *m3-inv1-lkeysec-for-parts* [*simp*]:
  ⟦ *s* ∈ *m3-inv1-lkeysec* ⟧ ⟹ *Key* (*shrK C*) ∈ *parts* (*IK s*) ⟷ *C* ∈ *bad*
⟨*proof*⟩

lemma *m3-inv1-lkeysec-for-analz* [*simp*]:
  ⟦ *s* ∈ *m3-inv1-lkeysec* ⟧ ⟹ *Key* (*shrK C*) ∈ *analz* (*IK s*) ⟷ *C* ∈ *bad*
⟨*proof*⟩

inv7a: Session keys not used to encrypt other session keys

Session keys are not used to encrypt other keys. Proof requires generalization to sets of session keys.

NOTE: This invariant will be derived from the corresponding L2 invariant using the simulation relation.

definition
  m3-inv7a-sesK-compr :: *m3-pred*
where
  $m3\text{-}inv7a\text{-}sesK\text{-}compr \equiv \{s.\ \forall K\ KK.$
    $KK \subseteq range\ sesK \longrightarrow$
    $(Key\ K \in analz\ (Key`KK \cup (IK\ s))) = (K \in KK \vee Key\ K \in analz\ (IK\ s))$
  $\}$

lemmas *m3-inv7a-sesK-comprI* = *m3-inv7a-sesK-compr-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv7a-sesK-comprE* = *m3-inv7a-sesK-compr-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv7a-sesK-comprD* = *m3-inv7a-sesK-compr-def* [*THEN setc-def-to-dest*, *rule-format*]

Additional lemma

lemmas *insert-commute-Key* = *insert-commute* [**where** *x*=*Key K* **for** *K*]

lemmas *m3-inv7a-sesK-compr-simps* =
  m3-inv7a-sesK-comprD
  m3-inv7a-sesK-comprD [**where** *KK*={*Kab*} **for** *Kab*, *simplified*]
  m3-inv7a-sesK-comprD [**where** *KK*=*insert Kab KK* **for** *Kab KK*, *simplified*]
  insert-commute-Key

3.6.6 Refinement

Message abstraction and simulation relation

Abstraction function on sets of messages.

inductive-set
  abs-msg :: *msg set* ⇒ *chmsg set*
  for *H* :: *msg set*
where
  am-M1:
   $\{\!|Agent\ A,\ Agent\ B,\ Nonce\ N|\!\} \in H$
  $\Longrightarrow Insec\ A\ B\ (Msg\ [aNon\ N]) \in abs\text{-}msg\ H$
| *am-M2a*:
    $Crypt\ (shrK\ C)\ \{\!|Key\ K,\ Agent\ B,\ Number\ T,\ Nonce\ N|\!\} \in H$
  $\Longrightarrow Secure\ Sv\ C\ (Msg\ [aKey\ K,\ aAgt\ B,\ aNum\ T,\ aNon\ N]) \in abs\text{-}msg\ H$
| *am-M2b*:
    $Crypt\ (shrK\ C)\ \{\!|Key\ K,\ Agent\ A,\ Number\ T|\!\} \in H$
  $\Longrightarrow Secure\ Sv\ C\ (Msg\ [aKey\ K,\ aAgt\ A,\ aNum\ T]) \in abs\text{-}msg\ H$
| *am-M3*:
    $Crypt\ K\ \{\!|Agent\ A,\ Number\ T|\!\} \in H$
  $\Longrightarrow dAuth\ K\ (Msg\ [aAgt\ A,\ aNum\ T]) \in abs\text{-}msg\ H$
| *am-M4*:
    $Crypt\ K\ (Number\ T) \in H$
  $\Longrightarrow dAuth\ K\ (Msg\ [aNum\ T]) \in abs\text{-}msg\ H$

R23: The simulation relation. This is a data refinement of the insecure and secure channels of refinement 2.

definition

R23-msgs :: (*m2-state* × *m3-state*) *set* **where**
R23-msgs ≡ {(*s*, *t*). *abs-msg* (*parts* (*IK t*)) ⊆ *chan s* }

definition
R23-keys :: (*m2-state* × *m3-state*) *set* **where**
R23-keys ≡ {(*s*, *t*). ∀ *KK K*. *KK* ⊆ *range sesK* ⟶
Key K ∈ *analz* (*Key'KK* ∪ (*IK t*)) ⟷ *aKey K* ∈ *extr* (*aKey'KK* ∪ *ik0*) (*chan s*)
}

definition
R23-non :: (*m2-state* × *m3-state*) *set* **where**
R23-non ≡ {(*s*, *t*). ∀ *KK N*. *KK* ⊆ *range sesK* ⟶
Nonce N ∈ *analz* (*Key'KK* ∪ (*IK t*)) ⟷ *aNon N* ∈ *extr* (*aKey'KK* ∪ *ik0*) (*chan s*)
}

definition
R23-pres :: (*m2-state* × *m3-state*) *set* **where**
R23-pres ≡ {(*s*, *t*). *runs s* = *runs t* ∧ *leak s* = *leak t* ∧ *clk s* = *clk t* ∧ *cache s* = *cache t*}

definition
R23 :: (*m2-state* × *m3-state*) *set* **where**
R23 ≡ *R23-msgs* ∩ *R23-keys* ∩ *R23-non* ∩ *R23-pres*

lemmas *R23-defs* =
R23-def R23-msgs-def R23-keys-def R23-non-def R23-pres-def

The mediator function is the identity here.

definition
med32 :: *m3-obs* ⇒ *m2-obs* **where**
med32 ≡ *id*

lemmas *R23-msgsI* = *R23-msgs-def* [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-msgsE* [*elim*] = *R23-msgs-def* [*THEN rel-def-to-elim*, *simplified*, *rule-format*]
lemmas *R23-msgsE'* [*elim*] = *R23-msgs-def* [*THEN rel-def-to-dest*, *simplified*, *rule-format*, *THEN subsetD*]

lemmas *R23-keysI* = *R23-keys-def* [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-keysE* [*elim*] = *R23-keys-def* [*THEN rel-def-to-elim*, *simplified*, *rule-format*]

lemmas *R23-nonI* = *R23-non-def* [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-nonE* [*elim*] = *R23-non-def* [*THEN rel-def-to-elim*, *simplified*, *rule-format*]

lemmas *R23-presI* = *R23-pres-def* [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-presE* [*elim*] = *R23-pres-def* [*THEN rel-def-to-elim*, *simplified*, *rule-format*]

lemmas *R23-intros* = *R23-msgsI R23-keysI R23-nonI R23-presI*

Simplifier lemmas for various instantiations (keys and nonces).

lemmas *R23-keys-simp* = *R23-keys-def* [*THEN rel-def-to-dest*, *simplified*, *rule-format*]
lemmas *R23-keys-simps* =
R23-keys-simp

R23-keys-simp [**where** *KK*={}, *simplified*]
R23-keys-simp [**where** *KK*={*K'*} **for** *K'*, *simplified*]
R23-keys-simp [**where** *KK*=*insert K' KK* **for** *K' KK*, *simplified*, *OF* - *conjI*]

lemmas *R23-non-simp* = *R23-non-def* [*THEN rel-def-to-dest*, *simplified*, *rule-format*]
lemmas *R23-non-simps* =
R23-non-simp
R23-non-simp [**where** *KK*={}, *simplified*]
R23-non-simp [**where** *KK*={*K*} **for** *K*, *simplified*]
R23-non-simp [**where** *KK*=*insert K KK* **for** *K KK*, *simplified*, *OF* - *conjI*]

lemmas *R23-simps* = *R23-keys-simps R23-non-simps*

General lemmas

General facts about *abs-msg*

declare *abs-msg.intros* [*intro*!]
declare *abs-msg.cases* [*elim*!]

lemma *abs-msg-empty*: *abs-msg* {} = {}
⟨*proof*⟩

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

lemma *abs-msg-mono* [*elim*]:
⟦ *m* ∈ *abs-msg G*; *G* ⊆ *H* ⟧ ⟹ *m* ∈ *abs-msg H*
⟨*proof*⟩

lemma *abs-msg-insert-mono* [*intro*]:
⟦ *m* ∈ *abs-msg H* ⟧ ⟹ *m* ∈ *abs-msg* (*insert m' H*)
⟨*proof*⟩

Facts about *abs-msg* concerning abstraction of fakeable messages. This is crucial for proving the refinement of the intruder event.

lemma *abs-msg-DY-subset-fakeable*:
⟦ (*s*, *t*) ∈ *R23-msgs*; (*s*, *t*) ∈ *R23-keys*; (*s*, *t*) ∈ *R23-non*; *t* ∈ *m3-inv1-lkeysec* ⟧
⟹ *abs-msg* (*synth* (*analz* (*IK t*))) ⊆ *fake ik0* (*dom* (*runs s*)) (*chan s*)
⟨*proof*⟩

Refinement proof

Pair decomposition. These were set to `elim!`, which is too agressive here.

declare *MPair-analz* [*rule del*, *elim*]
declare *MPair-parts* [*rule del*, *elim*]

Protocol events.

lemma *PO-m3-step1-refines-m2-step1*:
{*R23*}
(*m2-step1 Ra A B Na*), (*m3-step1 Ra A B Na*)

{> *R23*}
⟨*proof*⟩

lemma *PO-m3-step2-refines-m2-step2*:
{*R23*}
(*m2-step2 Rb A B*), (*m3-step2 Rb A B*)
{> *R23*}
⟨*proof*⟩

lemma *PO-m3-step3-refines-m2-step3*:
{*R23* ∩ (*m2-inv3a-sesK-compr*) × (*m3-inv7a-sesK-compr* ∩ *m3-inv1-lkeysec*)}
(*m2-step3 Rs A B Kab Na Ts*), (*m3-step3 Rs A B Kab Na Ts*)
{> *R23*}
⟨*proof*⟩

lemma *PO-m3-step4-refines-m2-step4*:
{*R23* ∩ *UNIV* × (*m3-inv1-lkeysec*) }
(*m2-step4 Ra A B Na Kab Ts Ta*), (*m3-step4 Ra A B Na Kab Ts Ta*)
{> *R23*}
⟨*proof*⟩

lemma *PO-m3-step5-refines-m2-step5*:
{*R23*}
(*m2-step5 Rb A B Kab Ts Ta*), (*m3-step5 Rb A B Kab Ts Ta*)
{> *R23*}
⟨*proof*⟩

lemma *PO-m3-step6-refines-m2-step6*:
{*R23*}
(*m2-step6 Ra A B Na Kab Ts Ta*), (*m3-step6 Ra A B Na Kab Ts Ta*)
{> *R23*}
⟨*proof*⟩

lemma *PO-m3-tick-refines-m2-tick*:
{*R23*}
(*m2-tick T*), (*m3-tick T*)
{>*R23*}
⟨*proof*⟩

lemma *PO-m3-purge-refines-m2-purge*:
{*R23*}
(*m2-purge A*), (*m3-purge A*)
{>*R23*}
⟨*proof*⟩

Intruder events.

lemma *PO-m3-leak-refines-m2-leak*:
{*R23*}
(*m2-leak Rs A B Na Ts*), (*m3-leak Rs A B Na Ts*)
{>*R23*}
⟨*proof*⟩

lemma *PO-m3-DY-fake-refines-m2-fake*:
{*R23* ∩ *UNIV* × *m3-inv1-lkeysec*}
m2-fake, *m3-DY-fake*
{> *R23*}
⟨*proof*⟩

All together now...

lemmas *PO-m3-trans-refines-m2-trans* =
PO-m3-step1-refines-m2-step1 PO-m3-step2-refines-m2-step2
PO-m3-step3-refines-m2-step3 PO-m3-step4-refines-m2-step4
PO-m3-step5-refines-m2-step5 PO-m3-step6-refines-m2-step6
PO-m3-tick-refines-m2-tick PO-m3-purge-refines-m2-purge
PO-m3-leak-refines-m2-leak PO-m3-DY-fake-refines-m2-fake

lemma *PO-m3-refines-init-m2* [*iff*]:
init m3 ⊆ *R23*``(*init m2*)
⟨*proof*⟩

lemma *PO-m3-refines-trans-m2* [*iff*]:
{*R23* ∩ (*m2-inv3a-sesK-compr*) × (*m3-inv7a-sesK-compr* ∩ *m3-inv1-lkeysec*)}
(*trans m2*), (*trans m3*)
{> *R23*}
⟨*proof*⟩

lemma *PO-m3-observation-consistent* [*iff*]:
obs-consistent R23 med32 m2 m3
⟨*proof*⟩

Refinement result.

lemma *m3-refines-m2* [*iff*]:
refines
(*R23* ∩ (*m2-inv3a-sesK-compr*) × (*m3-inv1-lkeysec*))
med32 m2 m3
⟨*proof*⟩

lemma *m3-implements-m2* [*iff*]:
implements med32 m2 m3
⟨*proof*⟩

3.6.7 Inherited invariants

inv3 (derived): Key secrecy for initiator

definition
m3-inv3-ikk-init :: *m3-state set*
where
m3-inv3-ikk-init ≡ {*s*. ∀ *A B Ra K Ts nl*.
runs s Ra = *Some* (*Init*, [*A*, *B*], *aKey K* # *aNum Ts* # *nl*) ⟶ *A* ∈ *good* ⟶ *B* ∈ *good* ⟶
Key K ∈ *analz* (*IK s*) ⟶
(*K*, *A*, *B*, *Ra*$*na*, *Ts*) ∈ *leak s*
}

lemmas *m3-inv3-ikk-initI* = *m3-inv3-ikk-init-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv3-ikk-initE* [*elim*] = *m3-inv3-ikk-init-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv3-ikk-initD* = *m3-inv3-ikk-init-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

lemma *PO-m3-inv3-ikk-init*: *reach m3* ⊆ *m3-inv3-ikk-init*
⟨*proof*⟩

inv4 (derived): Key secrecy for responder

definition
  m3-inv4-ikk-resp :: *m3-state set*
where
  m3-inv4-ikk-resp ≡ {*s*. ∀ *A B Rb K Ts nl*.
    runs s Rb = *Some* (*Resp*, [*A*, *B*], *aKey K* # *aNum Ts* # *nl*) ⟶ *A* ∈ *good* ⟶ *B* ∈ *good* ⟶
    Key K ∈ *analz* (*IK s*) ⟶
      (∃ *Na*. (*K*, *A*, *B*, *Na*, *Ts*) ∈ *leak s*)
  }

lemmas *m3-inv4-ikk-respI* = *m3-inv4-ikk-resp-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv4-ikk-respE* [*elim*] = *m3-inv4-ikk-resp-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv4-ikk-respD* = *m3-inv4-ikk-resp-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

lemma *PO-m3-inv4-ikk-resp*: *reach m3* ⊆ *m3-inv4-ikk-resp*
⟨*proof*⟩

end

3.7 Core Kerberos 5 (L3)

theory *m3-kerberos5* **imports** *m2-kerberos ../Refinement/Message*
begin

We model the core Kerberos 5 protocol:

$$\begin{array}{lll} \text{M1.} & A \rightarrow S: & A, B, Na \\ \text{M2.} & S \rightarrow A: & \{Kab, B, Ts, Na\}_{Kas}, \{Kab, A, Ts\}_{Kbs} \\ \text{M3.} & A \rightarrow B: & \{A, Ta\}_{Kab}, \{Kab, A, Ts\}_{Kbs} \\ \text{M4.} & B \rightarrow A: & \{Ta\}_{Kab} \end{array}$$

Proof tool configuration. Avoid annoying automatic unfolding of *dom*.

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

3.7.1 Setup

Now we can define the initial key knowledge.

overloading *ltkeySetup′* ≡ *ltkeySetup* **begin**
definition *ltkeySetup-def*: *ltkeySetup′* ≡ {(*sharK C*, *A*) | *C A*. *A* = *C* ∨ *A* = *Sv*}
end

lemma *corrKey-shrK-bad* [*simp*]: *corrKey* = *shrK*'*bad*
⟨*proof*⟩

3.7.2 State

The secure channels are star-shaped to/from the server. Therefore, we have only one agent in the relation.

record *m3-state* = *m1-state* +
  IK :: *msg set* — intruder knowledge

Observable state: *runs*, *m1-state.leak*, *clk*, and *cache*.

type-synonym
  m3-obs = *m2-obs*

definition
  m3-obs :: *m3-state* ⇒ *m3-obs* **where**
  m3-obs s ≡ (| *runs* = *runs s*, *leak* = *leak s*, *clk* = *clk s*, *cache* = *cache s* |)

type-synonym
  m3-pred = *m3-state set*

type-synonym
  m3-trans = (*m3-state* × *m3-state*) *set*

3.7.3 Events

Protocol events.

definition — by *A*, refines *m2-step1*
  m3-step1 :: [*rid-t*, *agent*, *agent*, *nonce*] ⇒ *m3-trans*
where
  m3-step1 Ra A B Na ≡ {(*s*, *s1*).
    — guards:
    Ra ∉ *dom* (*runs s*) ∧ — *Ra* is fresh
    Na = *Ra*$*na* ∧ — generated nonce

    — actions:
    s1 = *s*(|
      runs := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [])),
      IK := *insert* {|*Agent A*, *Agent B*, *Nonce Na*|} (*IK s*) — send M1
    |)
  }

definition — by *B*, refines *m2-step2*
  m3-step2 :: [*rid-t*, *agent*, *agent*] ⇒ *m3-trans*
where
  m3-step2 ≡ *m1-step2*

definition — by *Server*, refines *m2-step3*
  m3-step3 :: [*rid-t*, *agent*, *agent*, *key*, *nonce*, *time*] ⇒ *m3-trans*
where
  m3-step3 Rs A B Kab Na Ts ≡ {(*s*, *s1*).

— guards:
Rs ∉ *dom* (*runs s*) ∧ — fresh server run
Kab = *sesK* (*Rs$sk*) ∧ — fresh session key

{|*Agent A*, *Agent B*, *Nonce Na*|} ∈ *IK s* ∧ — recv *M1*
Ts = *clk s* ∧ — fresh timestamp

— actions:
— record session key and send *M2*
s1 = *s*(|
runs := (*runs s*)(*Rs* ↦ (*Serv*, [*A*, *B*], [*aNon Na*, *aNum Ts*])),
IK := *insert* {|*Crypt* (*shrK A*) {|*Key Kab*, *Agent B*, *Number Ts*, *Nonce Na*|},
Crypt (*shrK B*) {|*Key Kab*, *Agent A*, *Number Ts*|}|}
(*IK s*)
|)
}

definition — by *A*, refines *m2-step4*
m3-step4 :: [*rid-t*, *agent*, *agent*, *nonce*, *key*, *time*, *time*, *msg*] ⇒ *m3-trans*
where
m3-step4 Ra A B Na Kab Ts Ta X ≡ {(*s*, *s1*).

— guards:
runs s Ra = *Some* (*Init*, [*A*, *B*], []) ∧ — key not yet recv'd
Na = *Ra$na* ∧ — generated nonce

{|*Crypt* (*shrK A*) — recv *M2*
{|*Key Kab*, *Agent B*, *Number Ts*, *Nonce Na*|}, *X*|} ∈ *IK s* ∧

— read current time
Ta = *clk s* ∧

— check freshness of session key
clk s < *Ts* + *Ls* ∧

— actions:
— record session key and send *M3*
s1 = *s*(|
runs := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [*aKey Kab*, *aNum Ts*, *aNum Ta*])),
IK := *insert* {|*Crypt Kab* {|*Agent A*, *Number Ta*|}, *X*|} (*IK s*) — *M3*
|)
}

definition — by *B*, refines *m2-step5*
m3-step5 :: [*rid-t*, *agent*, *agent*, *key*, *time*, *time*] ⇒ *m3-trans*
where
m3-step5 Rb A B Kab Ts Ta ≡ {(*s*, *s1*).
— guards:
runs s Rb = *Some* (*Resp*, [*A*, *B*], []) ∧ — key not yet recv'd

{|*Crypt Kab* {|*Agent A*, *Number Ta*|}, — recv *M3*
Crypt (*shrK B*) {|*Key Kab*, *Agent A*, *Number Ts*|}|} ∈ *IK s* ∧

— ensure freshness of session key
$clk\ s < Ts + Ls\ \wedge$

— check authenticator's validity and replay; 'replays' with fresh authenticator ok!
$clk\ s < Ta + La\ \wedge$
$(B, Kab, Ta) \notin cache\ s\ \wedge$

— actions:
— record session key
$s1 = s($|
  $runs := (runs\ s)(Rb \mapsto (Resp, [A, B], [aKey\ Kab, aNum\ Ts, aNum\ Ta])),$
  $cache := insert\ (B, Kab, Ta)\ (cache\ s),$
  $IK := insert\ (Crypt\ Kab\ (Number\ Ta))\ (IK\ s)$ — send *M4*
|$)$
}

definition — by A, refines *m2-step6*
m3-step6 :: [*rid-t*, *agent*, *agent*, *nonce*, *key*, *time*, *time*] ⇒ *m3-trans*
where
m3-step6 Ra A B Na Kab Ts Ta ≡ {(*s*, *s'*).
— guards:
$runs\ s\ Ra = Some\ (Init, [A, B], [aKey\ Kab, aNum\ Ts, aNum\ Ta])\ \wedge$ — knows key
$Na = Ra\$na\ \wedge$ — generated nonce
$clk\ s < Ts + Ls\ \wedge$ — check session key's recentness

$Crypt\ Kab\ (Number\ Ta) \in IK\ s\ \wedge$ — recv *M4*

— actions:
$s' = s($|
  $runs := (runs\ s)(Ra \mapsto (Init, [A, B], [aKey\ Kab, aNum\ Ts, aNum\ Ta, END]))$
|$)$
}

Clock tick event

definition — refines *m2-tick*
m3-tick :: *time* ⇒ *m3-trans*
where
m3-tick ≡ *m1-tick*

Purge event: purge cache of expired timestamps

definition — refines *m2-purge*
m3-purge :: *agent* ⇒ *m3-trans*
where
m3-purge ≡ *m1-purge*

Session key compromise.

definition — refines *m2-leak*
m3-leak :: [*rid-t*, *agent*, *agent*, *nonce*, *time*] ⇒ *m3-trans*
where
m3-leak Rs A B Na Ts ≡ {(*s*, *s1*).
— guards:
$runs\ s\ Rs = Some\ (Serv, [A, B], [aNon\ Na, aNum\ Ts])\ \wedge$

$(clk\ s \geq Ts + Ls) \wedge$ — only compromise 'old' session keys

— actions:
— record session key as leaked and add it to intruder knowledge
$s1 = s(\!| \ leak := insert\ (sesK\ (Rs\$sk),\ A,\ B,\ Na,\ Ts)\ (leak\ s),$
$IK := insert\ (Key\ (sesK\ (Rs\$sk)))\ (IK\ s)\ |\!)$
}

Intruder fake event. The following "Dolev-Yao" event generates all intruder-derivable messages.

definition — refines *m2-fake*
m3-DY-fake :: *m3-trans*
where
$m3\text{-}DY\text{-}fake \equiv \{(s,\ s1).$

— actions:
$s1 = s(|\ IK := synth\ (analz\ (IK\ s))\ |)$ — take DY closure
}

3.7.4 Transition system

definition
m3-init :: *m3-pred*
where
$m3\text{-}init \equiv \{\ (\!|$
$runs = Map.empty,$
$leak = shrK`bad \times \{undefined\},$
$clk = 0,$
$cache = \{\},$
$IK = Key`shrK`bad$
$|\!)\ \}$

definition
m3-trans :: *m3-trans* **where**
$m3\text{-}trans \equiv (\bigcup A\ B\ Ra\ Rb\ Rs\ Na\ Kab\ Ts\ Ta\ T\ X.$
$m3\text{-}step1\ Ra\ A\ B\ Na \cup$
$m3\text{-}step2\ Rb\ A\ B \cup$
$m3\text{-}step3\ Rs\ A\ B\ Kab\ Na\ Ts \cup$
$m3\text{-}step4\ Ra\ A\ B\ Na\ Kab\ Ts\ Ta\ X \cup$
$m3\text{-}step5\ Rb\ A\ B\ Kab\ Ts\ Ta \cup$
$m3\text{-}step6\ Ra\ A\ B\ Na\ Kab\ Ts\ Ta \cup$
$m3\text{-}tick\ T \cup$
$m3\text{-}purge\ A \cup$
$m3\text{-}leak\ Rs\ A\ B\ Na\ Ts \cup$
$m3\text{-}DY\text{-}fake \cup$
Id
)

definition
m3 :: (*m3-state*, *m3-obs*) *spec* **where**
$m3 \equiv (\!|$
$init = m3\text{-}init,$
$trans = m3\text{-}trans,$

obs = *m3-obs*
⦈

lemmas *m3-loc-defs* =
m3-def m3-init-def m3-trans-def m3-obs-def
m3-step1-def m3-step2-def m3-step3-def m3-step4-def m3-step5-def
m3-step6-def m3-tick-def m3-purge-def m3-leak-def m3-DY-fake-def

lemmas *m3-defs* = *m3-loc-defs m2-defs*

3.7.5 Invariants

Specialized injection that we can apply more aggressively.

lemmas *analz-Inj-IK* = *analz.Inj* [**where** *H=IK s* **for** *s*]
lemmas *parts-Inj-IK* = *parts.Inj* [**where** *H=IK s* **for** *s*]

declare *parts-Inj-IK* [*dest*!]

declare *analz-into-parts* [*dest*]

inv1: Secrecy of pre-distributed shared keys

definition
m3-inv1-lkeysec :: *m3-pred*
where
m3-inv1-lkeysec ≡ {*s*. ∀ *C*.
(*Key* (*shrK C*) ∈ *parts* (*IK s*) ⟶ *C* ∈ *bad*) ∧
(*C* ∈ *bad* ⟶ *Key* (*shrK C*) ∈ *IK s*)
}

lemmas *m3-inv1-lkeysecI* = *m3-inv1-lkeysec-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv1-lkeysecE* [*elim*] = *m3-inv1-lkeysec-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv1-lkeysecD* = *m3-inv1-lkeysec-def* [*THEN setc-def-to-dest*, *rule-format*]

Invariance proof.

lemma *PO-m3-inv1-lkeysec-init* [*iff*]:
init m3 ⊆ *m3-inv1-lkeysec*
⟨*proof*⟩

lemma *PO-m3-inv1-lkeysec-trans* [*iff*]:
{*m3-inv1-lkeysec*} *trans m3* {> *m3-inv1-lkeysec*}
⟨*proof*⟩

lemma *PO-m3-inv1-lkeysec* [*iff*]: *reach m3* ⊆ *m3-inv1-lkeysec*
⟨*proof*⟩

Useful simplifier lemmas

lemma *m3-inv1-lkeysec-for-parts* [*simp*]:
⟦ *s* ∈ *m3-inv1-lkeysec* ⟧ ⟹ *Key* (*shrK C*) ∈ *parts* (*IK s*) ⟷ *C* ∈ *bad*
⟨*proof*⟩

lemma *m3-inv1-lkeysec-for-analz* [*simp*]:

$[\![s \in \textit{m3-inv1-lkeysec}]\!] \Longrightarrow \textit{Key} (\textit{shrK } C) \in \textit{analz} (\textit{IK } s) \longleftrightarrow C \in \textit{bad}$
⟨*proof*⟩

inv2: Session keys not used to encrypt other session keys

Session keys are not used to encrypt other keys. Proof requires generalization to sets of session keys.

NOTE: This invariant will be inherted from the corresponding L2 invariant using the simulation relation.

definition
  m3-inv2-sesK-compr :: *m3-pred*
where
  m3-inv2-sesK-compr ≡ {*s*. ∀ *K KK*.
    KK ⊆ *range sesK* ⟶
    (*Key K* ∈ *analz* (*Key‘KK* ∪ (*IK s*))) = (*K* ∈ *KK* ∨ *Key K* ∈ *analz* (*IK s*))
  }

lemmas *m3-inv2-sesK-comprI* = *m3-inv2-sesK-compr-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv2-sesK-comprE* = *m3-inv2-sesK-compr-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv2-sesK-comprD* = *m3-inv2-sesK-compr-def* [*THEN setc-def-to-dest*, *rule-format*]

Additional lemma

lemmas *insert-commute-Key* = *insert-commute* [**where** *x*=*Key K* **for** *K*]

lemmas *m3-inv2-sesK-compr-simps* =
  m3-inv2-sesK-comprD
  m3-inv2-sesK-comprD [**where** *KK*=*insert Kab KK* **for** *Kab KK*, *simplified*]
  m3-inv2-sesK-comprD [**where** *KK*={*Kab*} **for** *Kab*, *simplified*]
  insert-commute-Key

3.7.6 Refinement

Message abstraction and simulation relation

Abstraction function on sets of messages.

inductive-set
  abs-msg :: *msg set* ⇒ *chmsg set*
  for *H* :: *msg set*
where
  am-M1:
   ⦃*Agent A*, *Agent B*, *Nonce N*⦄ ∈ *H*
  ⟹ *Insec A B* (*Msg* [*aNon N*]) ∈ *abs-msg H*
| *am-M2a*:
    Crypt (*shrK C*) ⦃*Key K*, *Agent B*, *Number T*, *Nonce N*⦄ ∈ *H*
  ⟹ *Secure Sv C* (*Msg* [*aKey K*, *aAgt B*, *aNum T*, *aNon N*]) ∈ *abs-msg H*
| *am-M2b*:
    Crypt (*shrK C*) ⦃*Key K*, *Agent A*, *Number T*⦄ ∈ *H*
  ⟹ *Secure Sv C* (*Msg* [*aKey K*, *aAgt A*, *aNum T*]) ∈ *abs-msg H*
| *am-M3*:
    Crypt K ⦃*Agent A*, *Number T*⦄ ∈ *H*
  ⟹ *dAuth K* (*Msg* [*aAgt A*, *aNum T*]) ∈ *abs-msg H*

| *am-M4*:
  Crypt K (*Number T*) ∈ *H*
  ⟹ *dAuth K* (*Msg* [*aNum T*]) ∈ *abs-msg H*

R23: The simulation relation. This is a data refinement of the insecure and secure channels of refinement 2.

definition
  R23-msgs :: (*m2-state* × *m3-state*) *set* **where**
  R23-msgs ≡ {(*s*, *t*). *abs-msg* (*parts* (*IK t*)) ⊆ *chan s* }

definition
  R23-keys :: (*m2-state* × *m3-state*) *set* **where**
  R23-keys ≡ {(*s*, *t*). ∀ *KK K*. *KK* ⊆ *range sesK* ⟶
    Key K ∈ *analz* (*Key*‘*KK* ∪ (*IK t*)) ⟷ *aKey K* ∈ *extr* (*aKey*‘*KK* ∪ *ik0*) (*chan s*)
  }

definition
  R23-non :: (*m2-state* × *m3-state*) *set* **where**
  R23-non ≡ {(*s*, *t*). ∀ *KK N*. *KK* ⊆ *range sesK* ⟶
    Nonce N ∈ *analz* (*Key*‘*KK* ∪ (*IK t*)) ⟷ *aNon N* ∈ *extr* (*aKey*‘*KK* ∪ *ik0*) (*chan s*)
  }

definition
  R23-pres :: (*m2-state* × *m3-state*) *set* **where**
  R23-pres ≡ {(*s*, *t*). *runs s* = *runs t* ∧ *leak s* = *leak t* ∧ *clk s* = *clk t* ∧ *cache s* = *cache t*}

definition
  R23 :: (*m2-state* × *m3-state*) *set* **where**
  R23 ≡ *R23-msgs* ∩ *R23-keys* ∩ *R23-non* ∩ *R23-pres*

lemmas *R23-defs* =
  R23-def R23-msgs-def R23-keys-def R23-non-def R23-pres-def

The mediator function is the identity here.

definition
  med32 :: *m3-obs* ⇒ *m2-obs* **where**
  med32 ≡ *id*

lemmas *R23-msgsI* = *R23-msgs-def* [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-msgsE* [*elim*] = *R23-msgs-def* [*THEN rel-def-to-elim*, *simplified*, *rule-format*]
lemmas *R23-msgsE′* [*elim*] = *R23-msgs-def* [*THEN rel-def-to-dest*, *simplified*, *rule-format*, *THEN subsetD*]

lemmas *R23-keysI* = *R23-keys-def* [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-keysE* [*elim*] = *R23-keys-def* [*THEN rel-def-to-elim*, *simplified*, *rule-format*]

lemmas *R23-nonI* = *R23-non-def* [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-nonE* [*elim*] = *R23-non-def* [*THEN rel-def-to-elim*, *simplified*, *rule-format*]

lemmas *R23-presI* = *R23-pres-def* [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-presE* [*elim*] = *R23-pres-def* [*THEN rel-def-to-elim*, *simplified*, *rule-format*]

lemmas *R23-intros* = *R23-msgsI R23-keysI R23-nonI R23-presI*

Simplifier lemmas for various instantiations (keys and nonces).

lemmas *R23-keys-simp* = *R23-keys-def* [*THEN rel-def-to-dest*, *simplified*, *rule-format*]
lemmas *R23-keys-simps* =
  R23-keys-simp
  R23-keys-simp [**where** *KK*={}, *simplified*]
  R23-keys-simp [**where** *KK*={*K'*} **for** *K'*, *simplified*]
  R23-keys-simp [**where** *KK*=*insert K' KK* **for** *K' KK*, *simplified*, *OF - conjI*]

lemmas *R23-non-simp* = *R23-non-def* [*THEN rel-def-to-dest*, *simplified*, *rule-format*]
lemmas *R23-non-simps* =
  R23-non-simp
  R23-non-simp [**where** *KK*={}, *simplified*]
  R23-non-simp [**where** *KK*={*K*} **for** *K*, *simplified*]
  R23-non-simp [**where** *KK*=*insert K KK* **for** *K KK*, *simplified*, *OF - conjI*]

lemmas *R23-simps* = *R23-keys-simps R23-non-simps*

General lemmas

General facts about *abs-msg*

declare *abs-msg.intros* [*intro*!]
declare *abs-msg.cases* [*elim*!]

lemma *abs-msg-empty*: *abs-msg* {} = {}
⟨*proof*⟩

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

lemma *abs-msg-mono* [*elim*]:
  ⟦ *m* ∈ *abs-msg G*; *G* ⊆ *H* ⟧ ⟹ *m* ∈ *abs-msg H*
⟨*proof*⟩

lemma *abs-msg-insert-mono* [*intro*]:
  ⟦ *m* ∈ *abs-msg H* ⟧ ⟹ *m* ∈ *abs-msg* (*insert m' H*)
⟨*proof*⟩

Facts about *abs-msg* concerning abstraction of fakeable messages. This is crucial for proving the refinement of the intruder event.

lemma *abs-msg-DY-subset-fakeable*:
  ⟦ (*s*, *t*) ∈ *R23-msgs*; (*s*, *t*) ∈ *R23-keys*; (*s*, *t*) ∈ *R23-non*; *t* ∈ *m3-inv1-lkeysec* ⟧
  ⟹ *abs-msg* (*synth* (*analz* (*IK t*))) ⊆ *fake ik0* (*dom* (*runs s*)) (*chan s*)
⟨*proof*⟩

Refinement proof

Pair decomposition. These were set to `elim!`, which is too agressive here.

declare *MPair-analz* [*rule del*, *elim*]
declare *MPair-parts* [*rule del*, *elim*]

Protocol events.

lemma *PO-m3-step1-refines-m2-step1*:
  {*R23*}
    (*m2-step1 Ra A B Na*), (*m3-step1 Ra A B Na*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-step2-refines-m2-step2*:
  {*R23*}
    (*m2-step2 Rb A B*), (*m3-step2 Rb A B*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-step3-refines-m2-step3*:
  {*R23* ∩ (*m2-inv3a-sesK-compr*) × (*m3-inv2-sesK-compr* ∩ *m3-inv1-lkeysec*)}
    (*m2-step3 Rs A B Kab Na Ts*), (*m3-step3 Rs A B Kab Na Ts*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-step4-refines-m2-step4*:
  {*R23* ∩ *UNIV* × *m3-inv1-lkeysec*}
    (*m2-step4 Ra A B Na Kab Ts Ta*), (*m3-step4 Ra A B Na Kab Ts Ta X*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-step5-refines-m2-step5*:
  {*R23*}
    (*m2-step5 Rb A B Kab Ts Ta*), (*m3-step5 Rb A B Kab Ts Ta*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-step6-refines-m2-step6*:
  {*R23*}
    (*m2-step6 Ra A B Na Kab Ts Ta*), (*m3-step6 Ra A B Na Kab Ts Ta*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-tick-refines-m2-tick*:
  {*R23*}
    (*m2-tick T*), (*m3-tick T*)
  {>*R23*}
⟨*proof*⟩

lemma *PO-m3-purge-refines-m2-purge*:
  {*R23*}
    (*m2-purge A*), (*m3-purge A*)
  {>*R23*}
⟨*proof*⟩

Intruder events.

lemma *PO-m3-leak-refines-m2-leak*:
{*R23*}
(*m2-leak Rs A B Na Ts*), (*m3-leak Rs A B Na Ts*)
{>*R23*}
⟨*proof*⟩

lemma *PO-m3-DY-fake-refines-m2-fake*:
{*R23* ∩ *m2-inv3a-sesK-compr* × (*m3-inv2-sesK-compr* ∩ *m3-inv1-lkeysec*)}
m2-fake, *m3-DY-fake*
{> *R23*}
⟨*proof*⟩

All together now...

lemmas *PO-m3-trans-refines-m2-trans* =
PO-m3-step1-refines-m2-step1 PO-m3-step2-refines-m2-step2
PO-m3-step3-refines-m2-step3 PO-m3-step4-refines-m2-step4
PO-m3-step5-refines-m2-step5 PO-m3-step6-refines-m2-step6
PO-m3-tick-refines-m2-tick PO-m3-purge-refines-m2-purge
PO-m3-leak-refines-m2-leak PO-m3-DY-fake-refines-m2-fake

lemma *PO-m3-refines-init-m2* [*iff*]:
init m3 ⊆ *R23*‘‘(*init m2*)
⟨*proof*⟩

lemma *PO-m3-refines-trans-m2* [*iff*]:
{*R23* ∩ (*m2-inv3a-sesK-compr*) × (*m3-inv2-sesK-compr* ∩ *m3-inv1-lkeysec*)}
(*trans m2*), (*trans m3*)
{> *R23*}
⟨*proof*⟩

lemma *PO-m3-observation-consistent* [*iff*]:
obs-consistent R23 med32 m2 m3
⟨*proof*⟩

Refinement result.

lemma *m3-refines-m2* [*iff*]:
refines
(*R23* ∩ (*m2-inv3a-sesK-compr*) × (*m3-inv1-lkeysec*))
med32 m2 m3
⟨*proof*⟩

lemma *m3-implements-m2* [*iff*]:
implements med32 m2 m3
⟨*proof*⟩

3.7.7 Inherited invariants

inv3 (derived): Key secrecy for initiator

definition
m3-inv3-ikk-init :: *m3-state set*
where

m3-inv3-ikk-init ≡ {*s*. ∀ *A B Ra K Ts nl*.
runs s Ra = *Some* (*Init*, [*A*, *B*], *aKey K* # *aNum Ts* # *nl*) ⟶ *A* ∈ *good* ⟶ *B* ∈ *good* ⟶
Key K ∈ *analz* (*IK s*) ⟶
(*K*, *A*, *B*, *Ra$na*, *Ts*) ∈ *leak s*
}

lemmas *m3-inv3-ikk-initI* = *m3-inv3-ikk-init-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv3-ikk-initE* [*elim*] = *m3-inv3-ikk-init-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv3-ikk-initD* = *m3-inv3-ikk-init-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

lemma *PO-m3-inv3-ikk-init*: *reach m3* ⊆ *m3-inv3-ikk-init*
⟨*proof*⟩

inv4 (derived): Key secrecy for responder

definition
m3-inv4-ikk-resp :: *m3-state set*
where
m3-inv4-ikk-resp ≡ {*s*. ∀ *A B Rb K Ts nl*.
runs s Rb = *Some* (*Resp*, [*A*, *B*], *aKey K* # *aNum Ts* # *nl*) ⟶ *A* ∈ *good* ⟶ *B* ∈ *good* ⟶
Key K ∈ *analz* (*IK s*) ⟶
(∃ *Na*. (*K*, *A*, *B*, *Na*, *Ts*) ∈ *leak s*)
}

lemmas *m3-inv4-ikk-respI* = *m3-inv4-ikk-resp-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv4-ikk-respE* [*elim*] = *m3-inv4-ikk-resp-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv4-ikk-respD* = *m3-inv4-ikk-resp-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

lemma *PO-m3-inv4-ikk-resp*: *reach m3* ⊆ *m3-inv4-ikk-resp*
⟨*proof*⟩

end

3.8 Core Kerberos 4 (L3)

theory *m3-kerberos4* **imports** *m2-kerberos ../Refinement/Message*
begin

We model the core Kerberos 4 protocol:

$$\begin{array}{lll}
\text{M1.} & A \to S: & A, B \\
\text{M2.} & S \to A: & \{Kab, B, Ts, Na, \{Kab, A, Ts\}_{Kbs}\}_{Kas} \\
\text{M3.} & A \to B: & \{A, Ta\}_{Kab}, \{Kab, A, Ts\}_{Kbs} \\
\text{M4.} & B \to A: & \{Ta\}_{Kab}
\end{array}$$

Proof tool configuration. Avoid annoying automatic unfolding of *dom*.

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

3.8.1 Setup

Now we can define the initial key knowledge.

overloading *ltkeySetup'* ≡ *ltkeySetup* **begin**
definition *ltkeySetup-def*: *ltkeySetup'* ≡ {(*sharK C*, *A*) | *C A*. *A* = *C* ∨ *A* = *Sv*}
end

lemma *corrKey-shrK-bad* [*simp*]: *corrKey* = *shrK*‘*bad*
⟨*proof*⟩

3.8.2 State

The secure channels are star-shaped to/from the server. Therefore, we have only one agent in the relation.

record *m3-state* = *m1-state* +
  IK :: *msg set* — intruder knowledge

Observable state: *runs*, *clk*, and *cache*.

type-synonym
  m3-obs = *m2-obs*

definition
  m3-obs :: *m3-state* ⇒ *m3-obs* **where**
  m3-obs s ≡ (| *runs* = *runs s*, *leak* = *leak s*, *clk* = *clk s*, *cache* = *cache s* |)

type-synonym
  m3-pred = *m3-state set*

type-synonym
  m3-trans = (*m3-state* × *m3-state*) *set*

3.8.3 Events

Protocol events.

definition — by *A*, refines *m2-step1*
  m3-step1 :: [*rid-t*, *agent*, *agent*, *nonce*] ⇒ *m3-trans*
where
  m3-step1 Ra A B Na ≡ {(*s*, *s1*).
    — guards:
    Ra ∉ *dom* (*runs s*) ∧ — *Ra* is fresh
    Na = *Ra*$*na* ∧ — generated nonce

    — actions:
    s1 = *s*(|
      runs := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [])),
      IK := *insert* {|*Agent A*, *Agent B*, *Nonce Na*|} (*IK s*) — send *M1*
    |)
  }

definition — by *B*, refines *m2-step2*
  m3-step2 :: [*rid-t*, *agent*, *agent*] ⇒ *m3-trans*
where
  m3-step2 ≡ *m1-step2*

definition — by *Server*, refines *m2-step3*
  m3-step3 :: [*rid-t*, *agent*, *agent*, *key*, *nonce*, *time*] ⇒ *m3-trans*
where
  m3-step3 Rs A B Kab Na Ts ≡ {(*s*, *s1*).
    — guards:
    Rs ∉ *dom* (*runs s*) ∧ — fresh server run
    Kab = *sesK* (*Rs$sk*) ∧ — fresh session key

    ⦃*Agent A*, *Agent B*, *Nonce Na*⦄ ∈ *IK s* ∧ — recv *M1*
    Ts = *clk s* ∧ — fresh timestamp

    — actions:
    — record session key and send *M2*
    s1 = *s*⦇
      runs := (*runs s*)(*Rs* ↦ (*Serv*, [*A*, *B*], [*aNon Na*, *aNum Ts*])),
      IK := *insert* (*Crypt* (*shrK A*) — send *M2*
        ⦃*Key Kab*, *Agent B*, *Number Ts*, *Nonce Na*,
          Crypt (*shrK B*) ⦃*Key Kab*, *Agent A*, *Number Ts*⦄⦄)
        (*IK s*)
    ⦈
  }

definition — by *A*, refines *m2-step4*
  m3-step4 :: [*rid-t*, *agent*, *agent*, *nonce*, *key*, *time*, *time*, *msg*] ⇒ *m3-trans*
where
  m3-step4 Ra A B Na Kab Ts Ta X ≡ {(*s*, *s1*).

    — guards:
    runs s Ra = *Some* (*Init*, [*A*, *B*], []) ∧ — key not yet recv'd
    Na = *Ra$na* ∧ — generated nonce

    Crypt (*shrK A*) — recv *M2*
      ⦃*Key Kab*, *Agent B*, *Number Ts*, *Nonce Na*, *X*⦄ ∈ *IK s* ∧

    — read current time
    Ta = *clk s* ∧

    — check freshness of session key
    clk s < *Ts* + *Ls* ∧

    — actions:
    — record session key and send *M3*
    s1 = *s*⦇
      runs := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [*aKey Kab*, *aNum Ts*, *aNum Ta*])),
      IK := *insert* ⦃*Crypt Kab* ⦃*Agent A*, *Number Ta*⦄, *X*⦄ (*IK s*) — *M3*
    ⦈
  }

definition — by *B*, refines *m2-step5*
  m3-step5 :: [*rid-t*, *agent*, *agent*, *key*, *time*, *time*] ⇒ *m3-trans*
where
  m3-step5 Rb A B Kab Ts Ta ≡ {(*s*, *s1*).
    — guards:

$runs\ s\ Rb = Some\ (Resp,\ [A,\ B],\ [])\ \wedge$ — key not yet recv'd

$\{\!|Crypt\ Kab\ \{\!|Agent\ A,\ Number\ Ta|\!\},$ — recv *M3*
$Crypt\ (shrK\ B)\ \{\!|Key\ Kab,\ Agent\ A,\ Number\ Ts|\!\}|\!\} \in IK\ s\ \wedge$

— ensure freshness of session key
$clk\ s < Ts + Ls\ \wedge$

— check authenticator's validity and replay; 'replays' with fresh authenticator ok!
$clk\ s < Ta + La\ \wedge$
$(B,\ Kab,\ Ta) \notin cache\ s\ \wedge$

— actions:
— record session key
$s1 = s(\!|$
$runs := (runs\ s)(Rb \mapsto (Resp,\ [A,\ B],\ [aKey\ Kab,\ aNum\ Ts,\ aNum\ Ta])),$
$cache := insert\ (B,\ Kab,\ Ta)\ (cache\ s),$
$IK := insert\ (Crypt\ Kab\ (Number\ Ta))\ (IK\ s)$ — send *M4*
$|\!)$
}

definition — by *A*, refines *m2-step6*
$m3\text{-}step6 :: [rid\text{-}t,\ agent,\ agent,\ nonce,\ key,\ time,\ time] \Rightarrow m3\text{-}trans$
where
$m3\text{-}step6\ Ra\ A\ B\ Na\ Kab\ Ts\ Ta \equiv \{(s,\ s').$
— guards:
$runs\ s\ Ra = Some\ (Init,\ [A,\ B],\ [aKey\ Kab,\ aNum\ Ts,\ aNum\ Ta])\ \wedge$ — knows key
$Na = Ra\$na\ \wedge$ — generated nonce
$clk\ s < Ts + Ls\ \wedge$ — check session key's recentness

$Crypt\ Kab\ (Number\ Ta) \in IK\ s\ \wedge$ — recv *M4*

— actions:
$s' = s(\!|$
$runs := (runs\ s)(Ra \mapsto (Init,\ [A,\ B],\ [aKey\ Kab,\ aNum\ Ts,\ aNum\ Ta,\ END]))$
$|\!)$
}

Clock tick event

definition — refines *m2-tick*
$m3\text{-}tick :: time \Rightarrow m3\text{-}trans$
where
$m3\text{-}tick \equiv m1\text{-}tick$

Purge event: purge cache of expired timestamps

definition — refines *m2-purge*
$m3\text{-}purge :: agent \Rightarrow m3\text{-}trans$
where
$m3\text{-}purge \equiv m1\text{-}purge$

Session key compromise.

definition — refines *m2-leak*

m3-leak :: [*rid-t*, *agent*, *agent*, *nonce*, *time*] ⇒ *m3-trans*
where
m3-leak Rs A B Na Ts ≡ {(*s*, *s1*).
— guards:
runs s Rs = *Some* (*Serv*, [*A*, *B*], [*aNon Na*, *aNum Ts*]) ∧
(*clk s* ≥ *Ts* + *Ls*) ∧ — only compromise 'old' session keys!

— actions:
— record session key as leaked and add it to intruder knowledge
s1 = *s*⦇ *leak* := *insert* (*sesK* (*Rs*$*sk*), *A*, *B*, *Na*, *Ts*) (*leak s*),
IK := *insert* (*Key* (*sesK* (*Rs*$*sk*))) (*IK s*) ⦈
}

Intruder fake event. The following "Dolev-Yao" event generates all intruder-derivable messages.

definition — refines *m2-fake*
m3-DY-fake :: *m3-trans*
where
m3-DY-fake ≡ {(*s*, *s1*).

— actions:
s1 = *s*⦇ *IK* := *synth* (*analz* (*IK s*)) ⦈ — take DY closure
}

3.8.4 Transition system

definition
m3-init :: *m3-pred*
where
m3-init ≡ { ⦇
runs = *Map.empty*,
leak = *shrK*'*bad* × {*undefined*},
clk = *0*,
cache = {},
IK = *Key*'*shrK*'*bad*
⦈ }

definition
m3-trans :: *m3-trans* **where**
m3-trans ≡ (⋃*A B Ra Rb Rs Na Kab Ts Ta T X*.
m3-step1 Ra A B Na ∪
m3-step2 Rb A B ∪
m3-step3 Rs A B Kab Na Ts ∪
m3-step4 Ra A B Na Kab Ts Ta X ∪
m3-step5 Rb A B Kab Ts Ta ∪
m3-step6 Ra A B Na Kab Ts Ta ∪
m3-tick T ∪
m3-purge A ∪
m3-leak Rs A B Na Ts ∪
m3-DY-fake ∪
Id
)

definition
m3 :: (*m3-state*, *m3-obs*) *spec* **where**
m3 ≡ (|
init = *m3-init*,
trans = *m3-trans*,
obs = *m3-obs*
|)

lemmas *m3-loc-defs* =
m3-def m3-init-def m3-trans-def m3-obs-def
m3-step1-def m3-step2-def m3-step3-def m3-step4-def m3-step5-def
m3-step6-def m3-tick-def m3-purge-def m3-leak-def m3-DY-fake-def

lemmas *m3-defs* = *m3-loc-defs m2-defs*

3.8.5 Invariants

Specialized injection that we can apply more aggressively.

lemmas *analz-Inj-IK* = *analz.Inj* [**where** *H*=*IK s* **for** *s*]
lemmas *parts-Inj-IK* = *parts.Inj* [**where** *H*=*IK s* **for** *s*]

declare *parts-Inj-IK* [*dest*!]

declare *analz-into-parts* [*dest*]

inv4: Secrecy of pre-distributed shared keys

definition
m3-inv4-lkeysec :: *m3-pred*
where
m3-inv4-lkeysec ≡ {*s*. ∀ *C*.
(*Key* (*shrK C*) ∈ *parts* (*IK s*) ⟶ *C* ∈ *bad*) ∧
(*C* ∈ *bad* ⟶ *Key* (*shrK C*) ∈ *IK s*)
}

lemmas *m3-inv4-lkeysecI* = *m3-inv4-lkeysec-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv4-lkeysecE* [*elim*] = *m3-inv4-lkeysec-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv4-lkeysecD* = *m3-inv4-lkeysec-def* [*THEN setc-def-to-dest*, *rule-format*]

Invariance proof.

lemma *PO-m3-inv4-lkeysec-init* [*iff*]:
init m3 ⊆ *m3-inv4-lkeysec*
⟨*proof*⟩

lemma *PO-m3-inv4-lkeysec-trans* [*iff*]:
{*m3-inv4-lkeysec*} *trans m3* {> *m3-inv4-lkeysec*}
⟨*proof*⟩

lemma *PO-m3-inv4-lkeysec* [*iff*]: *reach m3* ⊆ *m3-inv4-lkeysec*
⟨*proof*⟩

Useful simplifier lemmas

lemma *m3-inv4-lkeysec-for-parts* [*simp*]:
$[\![s \in m3\text{-}inv4\text{-}lkeysec]\!] \Longrightarrow Key\ (shrK\ C) \in parts\ (IK\ s) \longleftrightarrow C \in bad$
⟨*proof*⟩

lemma *m3-inv4-lkeysec-for-analz* [*simp*]:
$[\![s \in m3\text{-}inv4\text{-}lkeysec]\!] \Longrightarrow Key\ (shrK\ C) \in analz\ (IK\ s) \longleftrightarrow C \in bad$
⟨*proof*⟩

inv6: Ticket shape for honestly encrypted M2

definition
m3-inv6-ticket :: *m3-pred*
where
$m3\text{-}inv6\text{-}ticket \equiv \{s.\ \forall A\ B\ T\ K\ N\ X.$
$A \notin bad \longrightarrow$
$Crypt\ (shrK\ A)\ \{\!|Key\ K,\ Agent\ B,\ Number\ T,\ Nonce\ N,\ X|\!\} \in parts\ (IK\ s) \longrightarrow$
$X = Crypt\ (shrK\ B)\ \{\!|Key\ K,\ Agent\ A,\ Number\ T|\!\} \wedge K \in range\ sesK$
$\}$

lemmas *m3-inv6-ticketI* = *m3-inv6-ticket-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv6-ticketE* [*elim*] = *m3-inv6-ticket-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv6-ticketD* = *m3-inv6-ticket-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated* −*1*]

Invariance proof.

lemma *PO-m3-inv6-ticket-init* [*iff*]:
$init\ m3 \subseteq m3\text{-}inv6\text{-}ticket$
⟨*proof*⟩

lemma *PO-m3-inv6-ticket-trans* [*iff*]:
$\{m3\text{-}inv6\text{-}ticket \cap m3\text{-}inv4\text{-}lkeysec\}\ trans\ m3\ \{> m3\text{-}inv6\text{-}ticket\}$
⟨*proof*⟩

lemma *PO-m3-inv6-ticket* [*iff*]: $reach\ m3 \subseteq m3\text{-}inv6\text{-}ticket$
⟨*proof*⟩

inv7: Session keys not used to encrypt other session keys

Session keys are not used to encrypt other keys. Proof requires generalization to sets of session keys.

NOTE: For Kerberos 4, this invariant cannot be inherited from the corresponding L2 invariant. The simulation relation is only an implication not an equivalence.

definition
m3-inv7a-sesK-compr :: *m3-pred*
where
$m3\text{-}inv7a\text{-}sesK\text{-}compr \equiv \{s.\ \forall K\ KK.$
$KK \subseteq range\ sesK \longrightarrow$
$(Key\ K \in analz\ (Key`KK \cup (IK\ s))) = (K \in KK \vee Key\ K \in analz\ (IK\ s))$
$\}$

lemmas *m3-inv7a-sesK-comprI* = *m3-inv7a-sesK-compr-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv7a-sesK-comprE* = *m3-inv7a-sesK-compr-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv7a-sesK-comprD* = *m3-inv7a-sesK-compr-def* [*THEN setc-def-to-dest*, *rule-format*]

Additional lemma

lemmas *insert-commute-Key* = *insert-commute* [**where** *x*=*Key K* **for** *K*]

lemmas *m3-inv7a-sesK-compr-simps* =
  m3-inv7a-sesK-comprD
  m3-inv7a-sesK-comprD [**where** *KK*=*insert Kab KK* **for** *Kab KK*, *simplified*]
  m3-inv7a-sesK-comprD [**where** *KK*={*Kab*} **for** *Kab*, *simplified*]
  insert-commute-Key

Invariance proof.

lemma *PO-m3-inv7a-sesK-compr-step4*:
  {*m3-inv7a-sesK-compr* ∩ *m3-inv6-ticket* ∩ *m3-inv4-lkeysec*}
     m3-step4 Ra A B Na Kab Ts Ta X
   {> *m3-inv7a-sesK-compr*}
⟨*proof*⟩

All together now.

lemmas *PO-m3-inv7a-sesK-compr-trans-lemmas* =
  PO-m3-inv7a-sesK-compr-step4

lemma *PO-m3-inv7a-sesK-compr-init* [*iff*]:
  init m3 ⊆ *m3-inv7a-sesK-compr*
⟨*proof*⟩

lemma *PO-m3-inv7a-sesK-compr-trans* [*iff*]:
  {*m3-inv7a-sesK-compr* ∩ *m3-inv6-ticket* ∩ *m3-inv4-lkeysec*}
    trans m3
   {> *m3-inv7a-sesK-compr*}
⟨*proof*⟩

lemma *PO-m3-inv7a-sesK-compr* [*iff*]: *reach m3* ⊆ *m3-inv7a-sesK-compr*
⟨*proof*⟩

inv7b: Session keys not used to encrypt nonces

Session keys are not used to encrypt nonces. The proof requires a generalization to sets of session keys.

definition
  m3-inv7b-sesK-compr-non :: *m3-pred*
where
  m3-inv7b-sesK-compr-non ≡ {*s*. ∀ *N KK*.
    KK ⊆ *range sesK* ⟶ (*Nonce N* ∈ *analz* (*Key*‘*KK* ∪ (*IK s*))) = (*Nonce N* ∈ *analz* (*IK s*))
  }

lemmas *m3-inv7b-sesK-compr-nonI* = *m3-inv7b-sesK-compr-non-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv7b-sesK-compr-nonE* = *m3-inv7b-sesK-compr-non-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv7b-sesK-compr-nonD* = *m3-inv7b-sesK-compr-non-def* [*THEN setc-def-to-dest*, *rule-format*]

lemmas *m3-inv7b-sesK-compr-non-simps* =
  m3-inv7b-sesK-compr-nonD
  m3-inv7b-sesK-compr-nonD [**where** *KK*=*insert Kab KK* **for** *Kab KK*, *simplified*]

m3-inv7b-sesK-compr-nonD [**where** *KK*={*Kab*} **for** *Kab*, *simplified*]
insert-commute-Key

Invariance proof.

lemma *PO-m3-inv7b-sesK-compr-non-step3*:
{*m3-inv7b-sesK-compr-non*} *m3-step3 Rs A B Kab Na Ts* {> *m3-inv7b-sesK-compr-non*}
⟨*proof*⟩

lemma *PO-m3-inv7b-sesK-compr-non-step4*:
{*m3-inv7b-sesK-compr-non* ∩ *m3-inv6-ticket* ∩ *m3-inv4-lkeysec*}
m3-step4 Ra A B Na Kab Ts Ta X
{> *m3-inv7b-sesK-compr-non*}
⟨*proof*⟩

All together now.

lemmas *PO-m3-inv7b-sesK-compr-non-trans-lemmas* =
PO-m3-inv7b-sesK-compr-non-step3 PO-m3-inv7b-sesK-compr-non-step4

lemma *PO-m3-inv7b-sesK-compr-non-init* [*iff*]:
init m3 ⊆ *m3-inv7b-sesK-compr-non*
⟨*proof*⟩

lemma *PO-m3-inv7b-sesK-compr-non-trans* [*iff*]:
{*m3-inv7b-sesK-compr-non* ∩ *m3-inv6-ticket* ∩ *m3-inv4-lkeysec*}
trans m3
{> *m3-inv7b-sesK-compr-non*}
⟨*proof*⟩

lemma *PO-m3-inv7b-sesK-compr-non* [*iff*]: *reach m3* ⊆ *m3-inv7b-sesK-compr-non*
⟨*proof*⟩

3.8.6 Refinement

Message abstraction and simulation relation

Abstraction function on sets of messages.

inductive-set
abs-msg :: *msg set* ⇒ *chmsg set*
for *H* :: *msg set*
where
am-M1:
⦃*Agent A*, *Agent B*, *Nonce N*⦄ ∈ *H*
⟹ *Insec A B* (*Msg* [*aNon N*]) ∈ *abs-msg H*
| *am-M2a*:
Crypt (*shrK C*) ⦃*Key K*, *Agent B*, *Number T*, *Nonce N*, *X*⦄ ∈ *H*
⟹ *Secure Sv C* (*Msg* [*aKey K*, *aAgt B*, *aNum T*, *aNon N*]) ∈ *abs-msg H*
| *am-M2b*:
Crypt (*shrK C*) ⦃*Key K*, *Agent A*, *Number T*⦄ ∈ *H*
⟹ *Secure Sv C* (*Msg* [*aKey K*, *aAgt A*, *aNum T*]) ∈ *abs-msg H*
| *am-M3*:
Crypt K ⦃*Agent A*, *Number T*⦄ ∈ *H*
⟹ *dAuth K* (*Msg* [*aAgt A*, *aNum T*]) ∈ *abs-msg H*

| *am-M4*:
  $Crypt\ K\ (Number\ T) \in H$
  $\Longrightarrow dAuth\ K\ (Msg\ [aNum\ T]) \in abs\text{-}msg\ H$

R23: The simulation relation. This is a data refinement of the insecure and secure channels of refinement 2.

definition
  $R23\text{-}msgs :: (m2\text{-}state \times m3\text{-}state)\ set$ **where**
  $R23\text{-}msgs \equiv \{(s,\ t).\ abs\text{-}msg\ (parts\ (IK\ t)) \subseteq chan\ s\ \}$

definition
  $R23\text{-}keys :: (m2\text{-}state \times m3\text{-}state)\ set$ **where**
  $R23\text{-}keys \equiv \{(s,\ t).\ \forall KK\ K.\ KK \subseteq range\ sesK \longrightarrow$
    $Key\ K \in analz\ (Key`KK \cup (IK\ t)) \longrightarrow aKey\ K \in extr\ (aKey`KK \cup ik0)\ (chan\ s)$
  $\}$

definition
  $R23\text{-}non :: (m2\text{-}state \times m3\text{-}state)\ set$ **where**
  $R23\text{-}non \equiv \{(s,\ t).\ \forall KK\ N.\ KK \subseteq range\ sesK \longrightarrow$
    $Nonce\ N \in analz\ (Key`KK \cup (IK\ t)) \longrightarrow aNon\ N \in extr\ (aKey`KK \cup ik0)\ (chan\ s)$
  $\}$

definition
  $R23\text{-}pres :: (m2\text{-}state \times m3\text{-}state)\ set$ **where**
  $R23\text{-}pres \equiv \{(s,\ t).\ runs\ s = runs\ t \wedge leak\ s = leak\ t \wedge clk\ s = clk\ t \wedge cache\ s = cache\ t\}$

definition
  $R23 :: (m2\text{-}state \times m3\text{-}state)\ set$ **where**
  $R23 \equiv R23\text{-}msgs \cap R23\text{-}keys \cap R23\text{-}non \cap R23\text{-}pres$

lemmas *R23-defs* =
  R23-def R23-msgs-def R23-keys-def R23-non-def R23-pres-def

The mediator function is the identity here.

definition
  $med32 :: m3\text{-}obs \Rightarrow m2\text{-}obs$ **where**
  $med32 \equiv id$

lemmas *R23-msgsI* = *R23-msgs-def* [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-msgsE* [*elim*] = *R23-msgs-def* [*THEN rel-def-to-elim*, *simplified*, *rule-format*]
lemmas *R23-msgsE′* [*elim*] =
  R23-msgs-def [*THEN rel-def-to-dest*, *simplified*, *rule-format*, *THEN subsetD*]

lemmas *R23-keysI* = *R23-keys-def* [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-keysE* [*elim*] = *R23-keys-def* [*THEN rel-def-to-elim*, *simplified*, *rule-format*]
lemmas *R23-keysD* = *R23-keys-def* [*THEN rel-def-to-dest*, *simplified*, *rule-format*, *rotated 2*]

lemmas *R23-nonI* = *R23-non-def* [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-nonE* [*elim*] = *R23-non-def* [*THEN rel-def-to-elim*, *simplified*, *rule-format*]
lemmas *R23-nonD* = *R23-non-def* [*THEN rel-def-to-dest*, *simplified*, *rule-format*, *rotated 2*]

lemmas *R23-presI* = *R23-pres-def* [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-presE* [*elim*] = *R23-pres-def* [*THEN rel-def-to-elim*, *simplified*, *rule-format*]

lemmas *R23-intros* = *R23-msgsI R23-keysI R23-nonI R23-presI*

Lemmas for various instantiations (keys and nonces).

lemmas *R23-keys-dests* =
  R23-keysD
  R23-keysD [**where** *KK*={}, *simplified*]
  R23-keysD [**where** *KK*={*K*} **for** *K*, *simplified*]
  R23-keysD [**where** *KK*=*insert K KK* **for** *K KK*, *simplified*, *OF - - conjI*]

lemmas *R23-non-dests* =
  R23-nonD
  R23-nonD [**where** *KK*={}, *simplified*]
  R23-nonD [**where** *KK*={*K*} **for** *K*, *simplified*]
  R23-nonD [**where** *KK*=*insert K KK* **for** *K KK*, *simplified*, *OF - - conjI*]

lemmas *R23-dests* = *R23-keys-dests R23-non-dests*

General lemmas

General facts about *abs-msg*

declare *abs-msg.intros* [*intro*!]
declare *abs-msg.cases* [*elim*!]

lemma *abs-msg-empty*: *abs-msg* {} = {}
⟨*proof*⟩

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

lemma *abs-msg-mono* [*elim*]:
  ⟦ *m* ∈ *abs-msg G*; *G* ⊆ *H* ⟧ ⟹ *m* ∈ *abs-msg H*
⟨*proof*⟩

lemma *abs-msg-insert-mono* [*intro*]:
  ⟦ *m* ∈ *abs-msg H* ⟧ ⟹ *m* ∈ *abs-msg* (*insert m′ H*)
⟨*proof*⟩

Facts about *abs-msg* concerning abstraction of fakeable messages. This is crucial for proving the refinement of the intruder event.

lemma *abs-msg-DY-subset-fakeable*:
  ⟦ (*s*, *t*) ∈ *R23-msgs*; (*s*, *t*) ∈ *R23-keys*; (*s*, *t*) ∈ *R23-non*; *t* ∈ *m3-inv4-lkeysec* ⟧
  ⟹ *abs-msg* (*synth* (*analz* (*IK t*))) ⊆ *fake ik0* (*dom* (*runs s*)) (*chan s*)
⟨*proof*⟩

Refinement proof

Pair decomposition. These were set to `elim!`, which is too agressive here.

declare *MPair-analz* [*rule del*, *elim*]
declare *MPair-parts* [*rule del*, *elim*]

Protocol events.

lemma *PO-m3-step1-refines-m2-step1*:
{*R23*}
(*m2-step1 Ra A B Na*), (*m3-step1 Ra A B Na*)
{> *R23*}
⟨*proof*⟩

lemma *PO-m3-step2-refines-m2-step2*:
{*R23*}
(*m2-step2 Rb A B*), (*m3-step2 Rb A B*)
{> *R23*}
⟨*proof*⟩

lemma *PO-m3-step3-refines-m2-step3*:
{*R23* ∩ (*m2-inv3a-sesK-compr*) × (*m3-inv7a-sesK-compr* ∩ *m3-inv4-lkeysec*)}
(*m2-step3 Rs A B Kab Na Ts*), (*m3-step3 Rs A B Kab Na Ts*)
{> *R23*}
⟨*proof*⟩

lemma *PO-m3-step4-refines-m2-step4*:
{*R23* ∩ (*UNIV*)
× (*m3-inv7a-sesK-compr* ∩ *m3-inv7b-sesK-compr-non* ∩ *m3-inv6-ticket* ∩ *m3-inv4-lkeysec*)}
(*m2-step4 Ra A B Na Kab Ts Ta*), (*m3-step4 Ra A B Na Kab Ts Ta X*)
{> *R23*}
⟨*proof*⟩

lemma *PO-m3-step5-refines-m2-step5*:
{*R23*}
(*m2-step5 Rb A B Kab Ts Ta*), (*m3-step5 Rb A B Kab Ts Ta*)
{> *R23*}
⟨*proof*⟩

lemma *PO-m3-step6-refines-m2-step6*:
{*R23*}
(*m2-step6 Ra A B Na Kab Ts Ta*), (*m3-step6 Ra A B Na Kab Ts Ta*)
{> *R23*}
⟨*proof*⟩

lemma *PO-m3-tick-refines-m2-tick*:
{*R23*}
(*m2-tick T*), (*m3-tick T*)
{>*R23*}
⟨*proof*⟩

lemma *PO-m3-purge-refines-m2-purge*:
{*R23*}
(*m2-purge A*), (*m3-purge A*)
{>*R23*}
⟨*proof*⟩

Intruder events.

lemma *PO-m3-leak-refines-m2-leak*:
  {*R23*}
    (*m2-leak Rs A B Na Ts*), (*m3-leak Rs A B Na Ts*)
  {>*R23*}
⟨*proof*⟩

lemma *PO-m3-DY-fake-refines-m2-fake*:
  {*R23* ∩ *UNIV* × (*m3-inv4-lkeysec*)}
    m2-fake, *m3-DY-fake*
  {> *R23*}
⟨*proof*⟩

All together now...

lemmas *PO-m3-trans-refines-m2-trans* =
  PO-m3-step1-refines-m2-step1 PO-m3-step2-refines-m2-step2
  PO-m3-step3-refines-m2-step3 PO-m3-step4-refines-m2-step4
  PO-m3-step5-refines-m2-step5 PO-m3-step6-refines-m2-step6
  PO-m3-tick-refines-m2-tick PO-m3-purge-refines-m2-purge
  PO-m3-leak-refines-m2-leak PO-m3-DY-fake-refines-m2-fake

lemma *PO-m3-refines-init-m2* [*iff*]:
  init m3 ⊆ *R23*``(*init m2*)
⟨*proof*⟩

lemma *PO-m3-refines-trans-m2* [*iff*]:
  {*R23* ∩ (*m2-inv3a-sesK-compr*)
      × (*m3-inv7a-sesK-compr* ∩ *m3-inv7b-sesK-compr-non* ∩ *m3-inv6-ticket* ∩ *m3-inv4-lkeysec*)}
    (*trans m2*), (*trans m3*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-observation-consistent* [*iff*]:
  obs-consistent R23 med32 m2 m3
⟨*proof*⟩

Refinement result.

lemma *m3-refines-m2* [*iff*]:
  refines
    (*R23* ∩
     (*m2-inv3a-sesK-compr*) ×
     (*m3-inv7a-sesK-compr* ∩ *m3-inv7b-sesK-compr-non* ∩ *m3-inv6-ticket* ∩ *m3-inv4-lkeysec*))
    med32 m2 m3
⟨*proof*⟩

lemma *m3-implements-m2* [*iff*]:
  implements med32 m2 m3
⟨*proof*⟩

3.8.7 Inherited invariants

inv3 (derived): Key secrecy for initiator

definition
  m3-inv3-ikk-init :: *m3-state set*
where
  m3-inv3-ikk-init ≡ {*s*. ∀ *A B Ra K Ts nl*.
    runs s Ra = *Some* (*Init*, [*A*, *B*], *aKey K* # *aNum Ts* # *nl*) ⟶ *A* ∈ *good* ⟶ *B* ∈ *good* ⟶
    Key K ∈ *analz* (*IK s*) ⟶
      (*K*, *A*, *B*, *Ra*$*na*, *Ts*) ∈ *leak s*
  }

lemmas *m3-inv3-ikk-initI* = *m3-inv3-ikk-init-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv3-ikk-initE* [*elim*] = *m3-inv3-ikk-init-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv3-ikk-initD* = *m3-inv3-ikk-init-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

lemma *PO-m3-inv3-ikk-init*: *reach m3* ⊆ *m3-inv3-ikk-init*
⟨*proof*⟩

inv4 (derived): Key secrecy for responder

definition
  m3-inv4-ikk-resp :: *m3-state set*
where
  m3-inv4-ikk-resp ≡ {*s*. ∀ *A B Rb K Ts nl*.
    runs s Rb = *Some* (*Resp*, [*A*, *B*], *aKey K* # *aNum Ts* # *nl*) ⟶ *A* ∈ *good* ⟶ *B* ∈ *good* ⟶
    Key K ∈ *analz* (*IK s*) ⟶
      (∃ *Na*. (*K*, *A*, *B*, *Na*, *Ts*) ∈ *leak s*)
  }

lemmas *m3-inv4-ikk-respI* = *m3-inv4-ikk-resp-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv4-ikk-respE* [*elim*] = *m3-inv4-ikk-resp-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv4-ikk-respD* = *m3-inv4-ikk-resp-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

lemma *PO-m3-inv4-ikk-resp*: *reach m3* ⊆ *m3-inv4-ikk-resp*
⟨*proof*⟩

end

3.9 Abstract Needham-Schroeder Shared Key (L1)

theory *m1-nssk* **imports** *m1-keydist-iirn*
begin

We add augment the basic abstract key distribution model such that the server reads and stores the initiator's nonce. We show three refinements, namley that this model refines

1. the basic key distribution model *m1a*, and
2. the injective agreement model *a0i*, instantiated such that the initiator agrees with the server on the session key and its nonce.

3. the non-injective agreement model *a0n*, instantiated such that the responder agrees with the server on the session key.

consts
nb :: *nat* — responder nonce constant
END :: *atom* — run end marker for responder

3.9.1 State

We extend the basic key distribution by adding nonces. The frames, the state, and the observations remain the same as in the previous model, but we will use the *nat list*'s to store nonces.

record *m1-state* = *m1r-state* +
leak :: (*key* × *fresh-t* × *fresh-t*) *set* — keys leaked plus session context

type-synonym *m1-obs* = *m1-state*

type-synonym *'x m1-pred* = *'x m1-state-scheme set*
type-synonym *'x m1-trans* = (*'x m1-state-scheme* × *'x m1-state-scheme*) *set*

3.9.2 Events

definition — by *A*, refines *m1a-step1*
m1-step1 :: [*rid-t*, *agent*, *agent*, *nonce*] ⇒ *'x m1r-trans*
where
m1-step1 Ra A B Na ≡ *m1a-step1 Ra A B Na*

definition — by *B*, refines *m1a-step2*
m1-step2 :: [*rid-t*, *agent*, *agent*] ⇒ *'x m1r-trans*
where
m1-step2 Rb A B ≡ *m1a-step2 Rb A B*

definition — by *Sv*, refines *m1a-step3*
m1-step3 :: [*rid-t*, *agent*, *agent*, *nonce*, *key*] ⇒ *'x m1r-trans*
where
m1-step3 Rs A B Na Kab ≡ *m1a-step3 Rs A B Kab Na* []

definition — by *A*, refines *m1a-step4*
m1-step4 :: [*rid-t*, *agent*, *agent*, *nonce*, *key*] ⇒ *'x m1-trans*
where
m1-step4 Ra A B Na Kab ≡ {(*s*, *s'*).
— guards:
runs s Ra = *Some* (*Init*, [*A*, *B*], []) ∧
Na = *Ra$na* ∧ — fix parameter
(*Kab* ∉ *Domain* (*leak s*) ⟶ (*Kab*, *A*) ∈ *azC* (*runs s*)) ∧ — authorization guard

— new guard for agreement with server on (*Kab*, *B*, *Na*),
— injectiveness by including *Na*
(*A* ∉ *bad* ⟶ (∃ *Rs*. *Kab* = *sesK* (*Rs$sk*) ∧
runs s Rs = *Some* (*Serv*, [*A*, *B*], [*aNon Na*]))) ∧

— actions:

$s' = s(\!|\ runs := (runs\ s)(Ra \mapsto (Init,\ [A,\ B],\ [aKey\ Kab]))\ |\!)$
}

definition — by B, refines *m1a-step5*
m1-step5 :: [*rid-t*, *agent*, *agent*, *nonce*, *key*] ⇒ *'x m1-trans*
where
$m1\text{-}step5\ Rb\ A\ B\ Nb\ Kab \equiv \{(s,\ s').$
— new guards:
$Nb = Rb\$nb\ \wedge$ — generate Nb

— prev guards:
$runs\ s\ Rb = Some\ (Resp,\ [A,\ B],\ [])\ \wedge$
$(Kab \notin Domain\ (leak\ s) \longrightarrow (Kab,\ B) \in azC\ (runs\ s))\ \wedge$ — authorization guard

— guard for showing agreement with server on $(Kab,\ A)$,
— this agreement is non-injective
$(B \notin bad \longrightarrow (\exists\, Rs\ Na.\ Kab = sesK\ (Rs\$sk)\ \wedge$
$runs\ s\ Rs = Some\ (Serv,\ [A,\ B],\ [aNon\ Na])))\ \wedge$

— actions:
$s' = s(\!|\ runs := (runs\ s)(Rb \mapsto (Resp,\ [A,\ B],\ [aKey\ Kab]))\ |\!)$
}

definition — by A, refines *skip*
m1-step6 :: [*rid-t*, *agent*, *agent*, *nonce*, *nonce*, *key*] ⇒ *'x m1-trans*
where
$m1\text{-}step6\ Ra\ A\ B\ Na\ Nb\ Kab \equiv \{(s,\ s').$
$runs\ s\ Ra = Some\ (Init,\ [A,\ B],\ [aKey\ Kab])\ \wedge$ — key recv'd before
$Na = Ra\$na\ \wedge$

— guard for showing agreement with B on Kab and Nb
$(A \notin bad \longrightarrow B \notin bad \longrightarrow$
$(\forall\, Nb'.\ (Kab,\ Na,\ Nb') \notin leak\ s) \longrightarrow$ — NEW: weaker condition
$(\exists\, Rb\ nl.\ Nb = Rb\$nb \wedge runs\ s\ Rb = Some\ (Resp,\ [A,\ B],\ aKey\ Kab\ \#\ nl)))\ \wedge$

— actions:
$s' = s(\!|$
$runs := (runs\ s)(Ra \mapsto (Init,\ [A,\ B],\ [aKey\ Kab,\ aNon\ Nb]))$
$|\!)$
}

definition — by B, refines *skip*
m1-step7 :: [*rid-t*, *agent*, *agent*, *nonce*, *key*] ⇒ *'x m1-trans*
where
$m1\text{-}step7\ Rb\ A\ B\ Nb\ Kab \equiv \{(s,\ s').$
$runs\ s\ Rb = Some\ (Resp,\ [A,\ B],\ [aKey\ Kab])\ \wedge$ — key recv'd before
$Nb = Rb\$nb\ \wedge$

— guard for showing agreement with A on Kab and Nb
$(A \notin bad \longrightarrow B \notin bad \longrightarrow Kab \notin Domain\ (leak\ s) \longrightarrow$
— $(\forall\, Na'.\ (Kab,\ Na',\ Nb) \notin leak\ s) \longrightarrow$ too strong, does not work
$(\exists\, Ra.\ runs\ s\ Ra = Some\ (Init,\ [A,\ B],\ [aKey\ Kab,\ aNon\ Nb])))\ \wedge$

— actions: (redundant) update local state marks successful termination
$s' = s$⦇
runs := (*runs s*)(*Rb* ↦ (*Resp*, [*A*, *B*], [*aKey Kab*, *END*]))
⦈
}

definition — by attacker, refines *s0g-leak*
m1-leak :: [*rid-t*, *rid-t*, *rid-t*, *agent*, *agent*] ⇒ ′*x m1-trans*
where
m1-leak Rs Ra Rb A B ≡ {(*s*, *s1*).
— guards:
runs s Rs = *Some* (*Serv*, [*A*, *B*], [*aNon* (*Ra*$*na*)]) ∧
runs s Ra = *Some* (*Init*, [*A*, *B*], [*aKey* (*sesK* (*Rs*$*sk*)), *aNon* (*Rb*$*nb*)]) ∧
runs s Rb = *Some* (*Resp*, [*A*, *B*], [*aKey* (*sesK* (*Rs*$*sk*)), *END*]) ∧

— actions:
s1 = *s*⦇ *leak* := *insert* (*sesK* (*Rs*$*sk*), *Ra*$*na*, *Rb*$*nb*) (*leak s*) ⦈
}

3.9.3 Specification

abbreviation
m1-init :: *m1-state set*
where
m1-init ≡ { ⦇
runs = *Map.empty*,
leak = *corrKey* × {*undefined*} × {*undefined*} — initial leakage
⦈ }

definition
m1-trans :: ′*x m1-trans* **where**
m1-trans ≡ (⋃ *A B Ra Rb Rs Na Nb Kab*.
m1-step1 Ra A B Na ∪
m1-step2 Rb A B ∪
m1-step3 Rs A B Na Kab ∪
m1-step4 Ra A B Na Kab ∪
m1-step5 Rb A B Nb Kab ∪
m1-step6 Ra A B Na Nb Kab ∪
m1-step7 Rb A B Nb Kab ∪
m1-leak Rs Ra Rb A B ∪
Id
)

definition
m1 :: (*m1-state*, *m1-obs*) *spec* **where**
m1 ≡ ⦇
init = *m1-init*,
trans = *m1-trans*,
obs = *id*
⦈

lemmas *m1-loc-defs* =
m1-def m1-trans-def

m1-step1-def m1-step2-def m1-step3-def m1-step4-def m1-step5-def
m1-step6-def m1-step7-def m1-leak-def

lemmas *m1-defs* = *m1-loc-defs m1a-defs*

lemma *m1-obs-id* [*simp*]: *obs m1* = *id*
⟨*proof*⟩

3.9.4 Invariants

inv0: Finite domain

There are only finitely many runs. This is needed to establish the responder/initiator agreements. This is already defined in the previous model, we just need to show that it still holds in the current model.

abbreviation
m1-inv0-fin :: *'x m1-pred* **where**
m1-inv0-fin ≡ *m1a-inv0-fin*

lemmas *m1-inv0-finI* = *m1a-inv0-finI*
lemmas *m1-inv0-finE* = *m1a-inv0-finE*
lemmas *m1-inv0-finD* = *m1a-inv0-finD*

Invariance proofs.

lemma *PO-m1-inv0-fin-init* [*iff*]:
init m1 ⊆ *m1-inv0-fin*
⟨*proof*⟩

lemma *PO-m1-inv0-fin-trans* [*iff*]:
{*m1-inv0-fin*} *trans m1* {> *m1-inv0-fin*}
⟨*proof*⟩

lemma *PO-m1-inv0-fin* [*iff*]: *reach m1* ⊆ *m1-inv0-fin*
⟨*proof*⟩

declare *PO-m1-inv0-fin* [*THEN subsetD*, *intro*]

3.9.5 Refinement of *m1a*

Simulation relation

med1a1: The mediator function maps a concrete observation (i.e., run) to an abstract one.

Instantiate parameters regarding list of freshness identifiers stored at server.

overloading *is-len'* ≡ *is-len rs-len'* ≡ *rs-len* **begin**
definition *is-len-def* [*simp*]: *is-len'* ≡ *0::nat*
definition *rs-len-def* [*simp*]: *rs-len'* ≡ *0::nat*
end

fun
rm1a1 :: *role-t* ⇒ *atom list* ⇒ *atom list*
where

rm1a1 Init = *take* (*Suc is-len*) — take *Kab*
| *rm1a1 Resp* = *take* (*Suc rs-len*) — take *Kab*
| *rm1a1 Serv* = *id* — take all

abbreviation
runs1a1 :: *runs-t* ⇒ *runs-t* **where**
runs1a1 ≡ *map-runs rm1a1*

lemmas *runs1a1-def* = *map-runs-def*

lemma *knC-runs1a1* [*simp*]:
knC (*runs1a1 runz*) = *knC runz*
⟨*proof*⟩

R1a1: The simulation relation is defined in terms of the mediator function.

definition
med1a1 :: *m1-obs* ⇒ *m1a-obs* **where**
med1a1 s ≡ (| *runs* = *runs1a1* (*runs s*), *m1x-state.leak* = *Domain* (*leak s*) |)

definition
R1a1 :: (*m1a-state* × *m1-state*) *set* **where**
R1a1 ≡ {(*s*, *t*). *s* = *med1a1 t*}

lemmas *R1a1-defs* = *R1a1-def med1a1-def*

Refinement proof

lemma *PO-m1-step1-refines-m1a-step1*:
{*R1a1*}
(*m1a-step1 Ra A B Na*), (*m1-step1 Ra A B Na*)
{> *R1a1*}
⟨*proof*⟩

lemma *PO-m1-step2-refines-m1a-step2*:
{*R1a1*}
(*m1a-step2 Rb A B*), (*m1-step2 Rb A B*)
{> *R1a1*}
⟨*proof*⟩

lemma *PO-m1-step3-refines-m1a-step3*:
{*R1a1*}
(*m1a-step3 Rs A B Kab Na* []), (*m1-step3 Rs A B Na Kab*)
{> *R1a1*}
⟨*proof*⟩

lemma *PO-m1-step4-refines-m1a-step4*:
{*R1a1*}
(*m1a-step4 Ra A B Na Kab* []), (*m1-step4 Ra A B Na Kab*)
{> *R1a1*}
⟨*proof*⟩

lemma *PO-m1-step5-refines-m1a-step5*:
{*R1a1*}

(*m1a-step5 Rb A B Kab* []), (*m1-step5 Rb A B Nb Kab*)
{> *R1a1*}
⟨*proof*⟩

lemma *PO-m1-step6-refines-m1a-skip*:
{*R1a1*}
Id, (*m1-step6 Ra A B Na Nb Kab*)
{> *R1a1*}
⟨*proof*⟩

lemma *PO-m1-step7-refines-m1a-skip*:
{*R1a1*}
Id, (*m1-step7 Rb A B Nb Kab*)
{> *R1a1*}
⟨*proof*⟩

lemma *PO-m1-leak-refines-m1a-leak*:
{*R1a1*}
(*m1a-leak Rs*), (*m1-leak Rs Ra Rb A B*)
{> *R1a1*}
⟨*proof*⟩

All together now...

lemmas *PO-m1-trans-refines-m1a-trans* =
PO-m1-step1-refines-m1a-step1 PO-m1-step2-refines-m1a-step2
PO-m1-step3-refines-m1a-step3 PO-m1-step4-refines-m1a-step4
PO-m1-step5-refines-m1a-step5 PO-m1-step6-refines-m1a-skip
PO-m1-step7-refines-m1a-skip PO-m1-leak-refines-m1a-leak

lemma *PO-m1-refines-init-m1a* [*iff*]:
init m1 ⊆ *R1a1*``(*init m1a*)
⟨*proof*⟩

lemma *PO-m1-refines-trans-m1a* [*iff*]:
{*R1a1*}
(*trans m1a*), (*trans m1*)
{> *R1a1*}
⟨*proof*⟩

Observation consistency.

lemma *obs-consistent-med1a1* [*iff*]:
obs-consistent R1a1 med1a1 m1a m1
⟨*proof*⟩

Refinement result.

lemma *PO-m1-refines-m1a* [*iff*]:
refines R1a1 med1a1 m1a m1
⟨*proof*⟩

lemma *m1-implements-m1a* [*iff*]: *implements med1a1 m1a m1*
⟨*proof*⟩

inv (inherited): Key secrecy

Secrecy, as external and internal invariant

definition
  m1-secrecy :: *'x m1-pred* **where**
  m1-secrecy ≡ {*s. knC* (*runs s*) ⊆ *azC* (*runs s*) ∪ *Domain* (*leak s*) × *UNIV*}

lemmas *m1-secrecyI* = *m1-secrecy-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m1-secrecyE* [*elim*] = *m1-secrecy-def* [*THEN setc-def-to-elim*, *rule-format*]

lemma *PO-m1-obs-secrecy* [*iff*]: *oreach m1* ⊆ *m1-secrecy*
⟨*proof*⟩

lemma *PO-m1-secrecy* [*iff*]: *reach m1* ⊆ *m1-secrecy*
⟨*proof*⟩

inv (inherited): Initiator auth server.

Simplified version of invariant *m1a-inv2i-serv*.

definition
  m1-inv2i-serv :: *'x m1r-pred*
where
  m1-inv2i-serv ≡ {*s*. ∀ *A B Ra Na Kab nla*.
    A ∉ *bad* ⟶
    runs s Ra = *Some* (*Init*, [*A*, *B*], *aKey Kab* # *nla*) ⟶
    Na = *Ra*$*na* ⟶
      (∃ *Rs*. *Kab* = *sesK* (*Rs*$*sk*) ∧ *runs s Rs* = *Some* (*Serv*, [*A*, *B*], [*aNon Na*]))
  }

lemmas *m1-inv2i-servI* = *m1-inv2i-serv-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m1-inv2i-servE* [*elim*] = *m1-inv2i-serv-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m1-inv2i-servD* = *m1-inv2i-serv-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 2*]

Proof of invariance.

lemma *PO-m1-inv2i-serv* [*iff*]: *reach m1* ⊆ *m1-inv2i-serv*
⟨*proof*⟩

declare *PO-m1-inv2i-serv* [*THEN subsetD*, *intro*]

inv (inherited): Responder auth server.

Simplified version of invarant *m1a-inv2r-serv*.

definition
  m1-inv2r-serv :: *'x m1r-pred*
where
  m1-inv2r-serv ≡ {*s*. ∀ *A B Rb Kab nlb*.
    B ∉ *bad* ⟶
    runs s Rb = *Some* (*Resp*, [*A*, *B*], *aKey Kab* # *nlb*) ⟶
      (∃ *Rs Na*. *Kab* = *sesK* (*Rs*$*sk*) ∧ *runs s Rs* = *Some* (*Serv*, [*A*, *B*], [*aNon Na*]))
  }

lemmas *m1-inv2r-servI* = *m1-inv2r-serv-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m1-inv2r-servE* [*elim*] = *m1-inv2r-serv-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m1-inv2r-servD* = *m1-inv2r-serv-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated* −*1*]

Proof of invariance.

lemma *PO-m1-inv2r-serv* [*iff*]: *reach m1* ⊆ *m1-inv2r-serv*
⟨*proof*⟩

declare *PO-m1-inv2r-serv* [*THEN subsetD*, *intro*]

inv (inherited): Initiator key freshness

definition
  m1-inv3-ifresh :: *'x m1-pred*
where
  m1-inv3-ifresh ≡ {*s*. ∀ *A A' B B' Ra Ra' Kab nl nl'*.
    runs s Ra = *Some* (*Init*, [*A*, *B*], *aKey Kab* # *nl*) ⟶
    runs s Ra' = *Some* (*Init*, [*A'*, *B'*], *aKey Kab* # *nl'*) ⟶
    A ∉ *bad* ⟶ *B* ∉ *bad* ⟶ *Kab* ∉ *Domain* (*leak s*) ⟶
      Ra = *Ra'*
  }

lemmas *m1-inv3-ifreshI* = *m1-inv3-ifresh-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m1-inv3-ifreshE* [*elim*] = *m1-inv3-ifresh-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m1-inv3-ifreshD* = *m1-inv3-ifresh-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

lemma *PO-m1-inv3-ifresh* [*iff*]: *reach m1* ⊆ *m1-inv3-ifresh*
⟨*proof*⟩

3.9.6 Refinement of *a0i* for initiator/responder

Simulation relation

We define two auxiliary functions to reconstruct the signals of the initial model from completed initiator and responder runs. For the initiator, we get an injective agreement with the responder on Kab and Nb.

type-synonym
  irsig = *key* × *nonce*

abbreviation
  ir-commit :: [*runs-t*, *agent*, *agent*, *key*, *nonce*] ⇒ *rid-t set*
where
  ir-commit runz A B Kab Nb ≡ {*Ra*.
    runz Ra = *Some* (*Init*, [*A*, *B*], [*aKey Kab*, *aNon Nb*])
  }

fun
  ir-runs2sigs :: *runs-t* ⇒ *irsig signal* ⇒ *nat*
where
  ir-runs2sigs runz (*Commit* [*A*, *B*] (*Kab*, *Nb*)) =
    card (*ir-commit runz A B Kab Nb*)

| *ir-runs2sigs runz* (*Running* [*A*, *B*] (*Kab*, *Nb*)) =
  (*if* ∃ *Rb nl*. *Nb* = *Rb*$*nb* ∧ *runz Rb* = *Some* (*Resp*, [*A*, *B*], *aKey Kab* # *nl*)
  then 1 else 0)

| *ir-runs2sigs runz* - = *0*

Simulation relation and mediator function. We map completed initiator and responder runs to commit and running signals, respectively.

definition
med-a0im1-ir :: *m1-obs* ⇒ *irsig a0i-obs* **where**
med-a0im1-ir o1 ≡ (| *signals* = *ir-runs2sigs* (*runs o1*), *corrupted* = *Domain* (*leak o1*) × *UNIV* |)

definition
R-a0im1-ir :: (*irsig a0i-state* × *m1-state*) *set* **where**
R-a0im1-ir ≡ {(*s*, *t*). *signals s* = *ir-runs2sigs* (*runs t*) ∧ *corrupted s* = *Domain* (*leak t*) × *UNIV*}

lemmas *R-a0im1-ir-defs* = *R-a0im1-ir-def med-a0im1-ir-def*

Lemmas about the abstraction function

lemma *ir-runs2sigs-empty* [*simp*]:
runz = *Map.empty* ⟹ *ir-runs2sigs runz* = (λ*s*. *0*)
⟨*proof*⟩

lemma *finite-ir-commit* [*simp*, *intro*!]:
finite (*dom runz*) ⟹ *finite* (*ir-commit runz A B Kab Nb*)
⟨*proof*⟩

Update lemmas

lemma *ir-runs2sigs-upd-init-none* [*simp*]:
⟦ *Ra* ∉ *dom runz* ⟧
⟹ *ir-runs2sigs* (*runz*(*Ra* ↦ (*Init*, [*A*, *B*], []))) = *ir-runs2sigs runz*
⟨*proof*⟩

lemma *ir-runs2sigs-upd-resp-none* [*simp*]:
⟦ *Rb* ∉ *dom runz* ⟧
⟹ *ir-runs2sigs* (*runz*(*Rb* ↦ (*Resp*, [*A*, *B*], []))) = *ir-runs2sigs runz*
⟨*proof*⟩

lemma *ir-runs2sigs-upd-serv-none* [*simp*]:
⟦ *Rs* ∉ *dom runz* ⟧
⟹ *ir-runs2sigs* (*runz*(*Rs* ↦ (*Serv*, [*A*, *B*], *nl*))) = *ir-runs2sigs runz*
⟨*proof*⟩

lemma *ir-runs2sigs-upd-init-some* [*simp*]:
⟦ *runz Ra* = *Some* (*Init*, [*A*, *B*], []) ⟧
⟹ *ir-runs2sigs* (*runz*(*Ra* ↦ (*Init*, [*A*, *B*], [*aKey Kab*]))) = *ir-runs2sigs runz*
⟨*proof*⟩

lemma *ir-runs2sigs-upd-resp* [*simp*]:
⟦ *runz Rb* = *Some* (*Resp*, [*A*, *B*], []) ⟧

$\Longrightarrow$ *ir-runs2sigs* (*runz*(*Rb* ↦ (*Resp*, [*A*, *B*], [*aKey Kab*]))) =
  (*ir-runs2sigs runz*)(*Running* [*A*, *B*] (*Kab*, *Rb*$*nb*) := *1*)
⟨*proof*⟩

lemma *ir-runs2sigs-upd-init* [*simp*]:
⟦ *runz Ra* = *Some* (*Init*, [*A*, *B*], [*aKey Kab*]); *finite* (*dom runz*) ⟧
$\Longrightarrow$ *ir-runs2sigs* (*runz*(*Ra* ↦ (*Init*, [*A*, *B*], [*aKey Kab*, *aNon Nb*]))) =
  (*ir-runs2sigs runz*)
    (*Commit* [*A*, *B*] (*Kab*, *Nb*) := *Suc* (*card* (*ir-commit runz A B Kab Nb*)))
⟨*proof*⟩

lemma *ir-runs2sigs-upd-resp-some* [*simp*]:
⟦ *runz Rb* = *Some* (*Resp*, [*A*, *B*], [*aKey K*]) ⟧
$\Longrightarrow$ *ir-runs2sigs* (*runz*(*Rb* ↦ (*Resp*, [*A*, *B*], [*aKey K*, *END*]))) = *ir-runs2sigs runz*
⟨*proof*⟩

Needed for injectiveness of agreement.

lemma *m1-inv2i-serv-lemma*:
⟦ *runs t Ra* = *Some* (*Init*, [*A*, *B*], [*aKey Kab*, *aNon Nb*]);
  runs t Ra' = *Some* (*Init*, [*A*, *B*], [*aKey Kab*]);
  A ∉ *bad*; *t* ∈ *m1-inv2i-serv* ⟧
$\Longrightarrow$ *P*
⟨*proof*⟩

Refinement proof

lemma *PO-m1-step1-refines-ir-a0i-skip*:
{*R-a0im1-ir*}
  Id, (*m1-step1 Ra A B Na*)
 {> *R-a0im1-ir*}
⟨*proof*⟩

lemma *PO-m1-step2-refines-ir-a0i-skip*:
{*R-a0im1-ir*}
  Id, (*m1-step2 Rb A B*)
 {> *R-a0im1-ir*}
⟨*proof*⟩

lemma *PO-m1-step3-refines-ir-a0i-skip*:
{*R-a0im1-ir*}
  Id, (*m1-step3 Rs A B Na Kab*)
 {> *R-a0im1-ir*}
⟨*proof*⟩

lemma *PO-m1-step4-refines-ir-a0i-skip*:
{*R-a0im1-ir*}
  Id, (*m1-step4 Ra A B Na Kab*)
 {> *R-a0im1-ir*}
⟨*proof*⟩

lemma *PO-m1-step5-refines-ir-a0i-running*:
{*R-a0im1-ir*}
  (*a0i-running* [*A*, *B*] (*Kab*, *Nb*)), (*m1-step5 Rb A B Nb Kab*)

{> *R-a0im1-ir*}
⟨*proof*⟩

lemma *PO-m1-step6-refines-ir-a0i-commit*:
{*R-a0im1-ir* ∩ *UNIV* × (*m1-inv2i-serv* ∩ *m1-inv0-fin*)}
(*a0i-commit* [*A*, *B*] (*Kab*, *Nb*)), (*m1-step6 Ra A B Na Nb Kab*)
{> *R-a0im1-ir*}
⟨*proof*⟩

lemma *PO-m1-step7-refines-ir-a0i-skip*:
{*R-a0im1-ir*}
Id, (*m1-step7 Rb A B Nb Kab*)
{> *R-a0im1-ir*}
⟨*proof*⟩

lemma *PO-m1-leak-refines-ir-a0i-corrupt*:
{*R-a0im1-ir*}
(*a0i-corrupt* ({*sesK* (*Rs*$*sk*)} × *UNIV*)), (*m1-leak Rs Ra Rb A B*)
{> *R-a0im1-ir*}
⟨*proof*⟩

All together now...

lemmas *PO-m1-trans-refines-ir-a0i-trans* =
PO-m1-step1-refines-ir-a0i-skip PO-m1-step2-refines-ir-a0i-skip
PO-m1-step3-refines-ir-a0i-skip PO-m1-step4-refines-ir-a0i-skip
PO-m1-step5-refines-ir-a0i-running PO-m1-step6-refines-ir-a0i-commit
PO-m1-step7-refines-ir-a0i-skip PO-m1-leak-refines-ir-a0i-corrupt

lemma *PO-m1-refines-ir-init-a0i* [*iff*]:
init m1 ⊆ *R-a0im1-ir*‘‘(*init a0i*)
⟨*proof*⟩

lemma *PO-m1-refines-ir-trans-a0i* [*iff*]:
{*R-a0im1-ir* ∩ *reach a0i* × *reach m1*}
(*trans a0i*), (*trans m1*)
{> *R-a0im1-ir*}
⟨*proof*⟩

Observation consistency.

lemma *obs-consistent-med-a0im1-ir* [*iff*]:
obs-consistent R-a0im1-ir med-a0im1-ir a0i m1
⟨*proof*⟩

Refinement result.

lemma *PO-m1-refines-ir-a0i* [*iff*]:
refines
(*R-a0im1-ir* ∩ *reach a0i* × *reach m1*)
med-a0im1-ir a0i m1
⟨*proof*⟩

lemma *m1-implements-ir-a0i*: *implements med-a0im1-ir a0i m1*
⟨*proof*⟩

3.9.7 Refinement of *a0i* for responder/initiator

Simulation relation

We define two auxiliary functions to reconstruct the signals of the initial model from initiator and responder runs. For the responder, we get an injective agreement with the initiator on Kab and Nb.

type-synonym
  risig = *key* × *nonce*

abbreviation
  ri-running :: [*runs-t*, *agent*, *agent*, *key*, *nonce*] ⇒ *rid-t set*
where
  ri-running runz A B Kab Nb ≡ {*Ra*.
    runz Ra = *Some* (*Init*, [*A*, *B*], [*aKey Kab*, *aNon Nb*])
  }

fun
  ri-runs2sigs :: *runs-t* ⇒ *risig signal* ⇒ *nat*
where
  ri-runs2sigs runz (*Commit* [*B*, *A*] (*Kab*, *Nb*)) =
    (*if* ∃ *Rb*. *Nb* = *Rb*$*nb* ∧ *runz Rb* = *Some* (*Resp*, [*A*, *B*], [*aKey Kab*, *END*])
     then 1 else 0)

| *ri-runs2sigs runz* (*Running* [*B*, *A*] (*Kab*, *Nb*)) =
    card (*ri-running runz A B Kab Nb*)

| *ri-runs2sigs runz* - = *0*

Simulation relation and mediator function. We map completed initiator and responder runs to commit and running signals, respectively.

definition
  med-a0im1-ri :: *m1-obs* ⇒ *risig a0i-obs* **where**
  med-a0im1-ri o1 ≡ (| *signals* = *ri-runs2sigs* (*runs o1*), *corrupted* = *Domain* (*leak o1*) × *UNIV* |)

definition
  R-a0im1-ri :: (*risig a0i-state* × *m1-state*) *set* **where**
  R-a0im1-ri ≡ {(*s*, *t*). *signals s* = *ri-runs2sigs* (*runs t*) ∧ *corrupted s* = *Domain* (*leak t*) × *UNIV*}

lemmas *R-a0im1-ri-defs* = *R-a0im1-ri-def med-a0im1-ri-def*

Lemmas about the auxiliary functions

lemma *ri-runs2sigs-empty* [*simp*]:
  runz = *Map.empty* ⟹ *ri-runs2sigs runz* = (λ*s*. *0*)
⟨*proof*⟩

lemma *finite-inv-ri-running* [*simp*, *intro*!]:
  finite (*dom runz*) ⟹ *finite* (*ri-running runz A B Kab Nb*)
⟨*proof*⟩

Update lemmas

lemma *ri-runs2sigs-upd-init-none* [*simp*]:
  ⟦ *Na* ∉ *dom runz* ⟧
  ⟹ *ri-runs2sigs* (*runz*(*Na* ↦ (*Init*, [*A*, *B*], []))) = *ri-runs2sigs runz*
⟨*proof*⟩

lemma *ri-runs2sigs-upd-resp-none* [*simp*]:
  ⟦ *Rb* ∉ *dom runz* ⟧
  ⟹ *ri-runs2sigs* (*runz*(*Rb* ↦ (*Resp*, [*A*, *B*], []))) = *ri-runs2sigs runz*
⟨*proof*⟩

lemma *ri-runs2sigs-upd-serv-none* [*simp*]:
  ⟦ *Rs* ∉ *dom runz* ⟧
  ⟹ *ri-runs2sigs* (*runz*(*Rs* ↦ (*Serv*, [*A*, *B*], *nl*))) = *ri-runs2sigs runz*
⟨*proof*⟩

lemma *ri-runs2sigs-upd-init* [*simp*]:
  ⟦ *runz Ra* = *Some* (*Init*, [*A*, *B*], [*aKey Kab*]); *finite* (*dom runz*) ⟧
  ⟹ *ri-runs2sigs* (*runz*(*Ra* ↦ (*Init*, [*A*, *B*], [*aKey Kab*, *aNon Nb*]))) =
    (*ri-runs2sigs runz*)
      (*Running* [*B*, *A*] (*Kab*, *Nb*) := *Suc* (*card* (*ri-running runz A B Kab Nb*)))
⟨*proof*⟩

lemma *ri-runs2sigs-upd-init-some* [*simp*]:
  ⟦ *runz Ra* = *Some* (*Init*, [*A*, *B*], []) ⟧
  ⟹ *ri-runs2sigs* (*runz*(*Ra* ↦ (*Init*, [*A*, *B*], [*aKey Kab*]))) = *ri-runs2sigs runz*
⟨*proof*⟩

lemma *ri-runs2sigs-upd-resp-some* [*simp*]:
  ⟦ *runz Rb* = *Some* (*Resp*, [*A*, *B*], [])⟧
  ⟹ *ri-runs2sigs* (*runz*(*Rb* ↦ (*Resp*, [*A*, *B*], [*aKey K*]))) = *ri-runs2sigs runz*
⟨*proof*⟩

lemma *ri-runs2sigs-upd-resp-some2* [*simp*]:
  ⟦ *runz Rb* = *Some* (*Resp*, [*A*, *B*], [*aKey Kab*]) ⟧
  ⟹ *ri-runs2sigs* (*runz*(*Rb* ↦ (*Resp*, [*A*, *B*], [*aKey Kab*, *END*]))) =
    (*ri-runs2sigs runz*)(*Commit* [*B*, *A*] (*Kab*, *Rb*$*nb*) := *1*)
⟨*proof*⟩

Refinement proof

lemma *PO-m1-step1-refines-ri-a0i-skip*:
  {*R-a0im1-ri*}
    Id, (*m1-step1 Ra A B Na*)
  {> *R-a0im1-ri*}
⟨*proof*⟩

lemma *PO-m1-step2-refines-ri-a0i-skip*:
  {*R-a0im1-ri*}
    Id, (*m1-step2 Rb A B*)
  {> *R-a0im1-ri*}
⟨*proof*⟩

lemma *PO-m1-step3-refines-ri-a0i-skip*:

{*R-a0im1-ri*}
  Id, (*m1-step3 Rs A B Na Kab*)
  {> *R-a0im1-ri*}
⟨*proof*⟩

lemma *PO-m1-step4-refines-ri-a0i-skip*:
{*R-a0im1-ri*}
  Id, (*m1-step4 Ra A B Nb Kab*)
  {> *R-a0im1-ri*}
⟨*proof*⟩

lemma *PO-m1-step5-refines-ri-a0i-skip*:
{*R-a0im1-ri*}
  Id, (*m1-step5 Rb A B Nb Kab*)
  {> *R-a0im1-ri*}
⟨*proof*⟩

lemma *PO-m1-step6-refines-ri-a0i-running*:
{*R-a0im1-ri* ∩ *UNIV* × *m1-inv0-fin*}
  (*a0i-running* [*B*, *A*] (*Kab*, *Nb*)), (*m1-step6 Ra A B Na Nb Kab*)
  {> *R-a0im1-ri*}
⟨*proof*⟩

lemma *PO-m1-step7-refines-ri-a0i-commit*:
{*R-a0im1-ri* ∩ *UNIV* × *m1-inv0-fin*}
  (*a0i-commit* [*B*, *A*] (*Kab*, *Nb*)), (*m1-step7 Rb A B Nb Kab*)
  {> *R-a0im1-ri*}
⟨*proof*⟩

lemma *PO-m1-leak-refines-ri-a0i-corrupt*:
{*R-a0im1-ri*}
  (*a0i-corrupt* ({*sesK* (*Rs$sk*)} × *UNIV*)), (*m1-leak Rs Ra Rb A B*)
  {> *R-a0im1-ri*}
⟨*proof*⟩

All together now...

lemmas *PO-m1-trans-refines-ri-a0i-trans* =
PO-m1-step1-refines-ri-a0i-skip PO-m1-step2-refines-ri-a0i-skip
PO-m1-step3-refines-ri-a0i-skip PO-m1-step4-refines-ri-a0i-skip
PO-m1-step5-refines-ri-a0i-skip PO-m1-step6-refines-ri-a0i-running
PO-m1-step7-refines-ri-a0i-commit PO-m1-leak-refines-ri-a0i-corrupt

lemma *PO-m1-refines-ri-init-a0i* [*iff*]:
init m1 ⊆ *R-a0im1-ri*``(*init a0i*)
⟨*proof*⟩

lemma *PO-m1-refines-ri-trans-a0i* [*iff*]:
{*R-a0im1-ri* ∩ *a0i-inv1-iagree* × *m1-inv0-fin*}
  (*trans a0i*), (*trans m1*)
  {> *R-a0im1-ri*}
⟨*proof*⟩

Observation consistency.

lemma *obs-consistent-med-a0im1-ri* [*iff*]:
obs-consistent R-a0im1-ri med-a0im1-ri a0i m1
⟨*proof*⟩

Refinement result.

lemma *PO-m1-refines-ri-a0i* [*iff*]:
refines (*R-a0im1-ri* ∩ *a0i-inv1-iagree* × *m1-inv0-fin*) *med-a0im1-ri a0i m1*
⟨*proof*⟩

lemma *m1-implements-ri-a0i*: *implements med-a0im1-ri a0i m1*
⟨*proof*⟩

inv3 (inherited): Responder and initiator

This is a translation of the agreement property to Level 1. It follows from the refinement and is needed to prove inv4.

definition
m1-inv3r-init :: *'x m1-pred*
where
m1-inv3r-init ≡ {*s*. ∀ *A B Rb Kab*.
B ∉ *bad* ⟶ *A* ∉ *bad* ⟶ *Kab* ∉ *Domain* (*leak s*) ⟶
runs s Rb = *Some* (*Resp*, [*A*, *B*], [*aKey Kab*, *END*]) ⟶
(∃ *Ra nla*. *runs s Ra* = *Some* (*Init*, [*A*, *B*], *aKey Kab* # *aNon* (*Rb$nb*) # *nla*))
}

lemmas *m1-inv3r-initI* =
m1-inv3r-init-def [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m1-inv3r-initE* [*elim*] =
m1-inv3r-init-def [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m1-inv3r-initD* =
m1-inv3r-init-def [*THEN setc-def-to-dest*, *rule-format*, *rotated* −*1*]

Invariance proof.

lemma *PO-m1-inv3r-init* [*iff*]: *reach m1* ⊆ *m1-inv3r-init*
⟨*proof*⟩

inv4: Key freshness for responder

definition
m1-inv4-rfresh :: *'x m1-pred*
where
m1-inv4-rfresh ≡ {*s*. ∀ *Rb Rb' A A' B B' Kab*.
runs s Rb = *Some* (*Resp*, [*A*, *B*], [*aKey Kab*, *END*]) ⟶
runs s Rb' = *Some* (*Resp*, [*A'*, *B'*], [*aKey Kab*, *END*]) ⟶
B ∉ *bad* ⟶ *A* ∉ *bad* ⟶ *Kab* ∉ *Domain* (*leak s*) ⟶
Rb = *Rb'*
}

lemmas *m1-inv4-rfreshI* = *m1-inv4-rfresh-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m1-inv4-rfreshE* [*elim*] = *m1-inv4-rfresh-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m1-inv4-rfreshD* = *m1-inv4-rfresh-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

Proof of key freshness for responder

lemma *PO-m1-inv4-rfresh-init* [*iff*]:
init m1 ⊆ *m1-inv4-rfresh*
⟨*proof*⟩

lemma *PO-m1-inv4-rfresh-trans* [*iff*]:
{*m1-inv4-rfresh* ∩ *m1-inv3r-init* ∩ *m1-inv2r-serv* ∩ *m1-inv3-ifresh* ∩ *m1-secrecy*}
trans m1
{> *m1-inv4-rfresh*}
⟨*proof*⟩

lemma *PO-m1-inv4-rfresh* [*iff*]: *reach m1* ⊆ *m1-inv4-rfresh*
⟨*proof*⟩

lemma *PO-m1-obs-inv4-rfresh* [*iff*]: *oreach m1* ⊆ *m1-inv4-rfresh*
⟨*proof*⟩

end

3.10 Abstract Needham-Schroeder Shared Key (L2)

theory *m2-nssk* **imports** *m1-nssk ../Refinement/Channels*
begin

We model an abstract version of the Needham-Schroeder Shared Key protocol:

$$\begin{array}{lll} \text{M1.} & A \to S: & A, B, Na \\ \text{M2.} & S \to A: & \{Na, B, Kab, \{Kab, A\}_{Kbs}\}_{Kas} \\ \text{M3.} & A \to B: & \{A, Kab\}_{Kbs} \\ \text{M4.} & B \to A: & \{Nb\}_{Kab} \\ \text{M5.} & A \to B: & \{Nb - 1\}_{Kab} \end{array}$$

The last two message are supposed to authenticate A to B, but this fails as shown by Dening and Sacco. Therefore and since we are mainly interested in secrecy at this point, we drop the last two message from this development.

This refinement introduces channels with security properties. We model a parallel/"channel-pure" version of the first three messages of the NSSK protocol:

$$\begin{array}{lll} \text{M1.} & A \to S: & A, B, Na \\ \text{M2.} & S \to A: & \{Na, B, Kab\}_{Kas} \\ \text{M3.} & S \to B: & \{Kab, A\}_{Kbs} \end{array}$$

Message 1 is sent over an insecure channel, the other two message over secure channels to/from the server.

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

3.10.1 State

record *m2-state* = *m1-state* +

chan :: *chmsg set* — channel messages

type-synonym
m2-obs = *m1-state*

definition
m2-obs :: *m2-state* ⇒ *m2-obs* **where**
m2-obs s ≡ (| *runs* = *runs s*, *leak* = *leak s* |)

type-synonym
m2-pred = *m2-state set*

type-synonym
m2-trans = (*m2-state* × *m2-state*) *set*

3.10.2 Events

Protocol events.

definition — by *A*, refines *m1a-step1*
m2-step1 :: [*rid-t*, *agent*, *agent*, *nonce*] ⇒ *m2-trans*
where
m2-step1 Ra A B Na ≡ {(*s*, *s1*).

— guards:
Ra ∉ *dom* (*runs s*) ∧ — fresh run identifier
Na = *Ra*$*na* ∧ — generate nonce *Na*

— actions:
— create initiator thread and send message 1
s1 = *s*(|
runs := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [])),
chan := *insert* (*Insec A B* (*Msg* [*aNon Na*])) (*chan s*) — msg 1
|)
}

definition — by *B*, refines *m1a-step2*
m2-step2 :: [*rid-t*, *agent*, *agent*] ⇒ *m2-trans*
where
m2-step2 ≡ *m1-step2*

definition — by *Server*, refines *m1a-step3*
m2-step3 :: [*rid-t*, *agent*, *agent*, *nonce*, *key*] ⇒ *m2-trans*
where
m2-step3 Rs A B Na Kab ≡ {(*s*, *s1*).
— guards:
Rs ∉ *dom* (*runs s*) ∧ — new server run
Kab = *sesK* (*Rs*$*sk*) ∧ — fresh session key

Insec A B (*Msg* [*aNon Na*]) ∈ *chan s* ∧ — recv msg 1

— actions:
— record key and send messages 2 and 3

— note that last field in server record is for responder nonce
s1 = *s*⦇
runs := (*runs s*)(*Rs* ↦ (*Serv*, [*A*, *B*], [*aNon Na*])),
chan := {*Secure Sv A* (*Msg* [*aNon Na*, *aAgt B*, *aKey Kab*]),
Secure Sv B (*Msg* [*aKey Kab*, *aAgt A*])} ∪ *chan s*
⦈
}

definition — by *A*, refines *m1a-step4*
m2-step4 :: [*rid-t*, *agent*, *agent*, *nonce*, *key*] ⇒ *m2-trans*
where
m2-step4 Ra A B Na Kab ≡ {(*s*, *s1*).

— guards:
runs s Ra = *Some* (*Init*, [*A*, *B*], []) ∧
Na = *Ra*$*na* ∧

Secure Sv A (*Msg* [*aNon Na*, *aAgt B*, *aKey Kab*]) ∈ *chan s* ∧ — recv msg 2

— actions:
— record session key
s1 = *s*⦇
runs := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [*aKey Kab*]))
⦈
}

definition — by *B*, refines *m1-step5*
m2-step5 :: [*rid-t*, *agent*, *agent*, *nonce*, *key*] ⇒ *m2-trans*
where
m2-step5 Rb A B Nb Kab ≡ {(*s*, *s1*).

— guards:
runs s Rb = *Some* (*Resp*, [*A*, *B*], []) ∧
Nb = *Rb*$*nb* ∧

Secure Sv B (*Msg* [*aKey Kab*, *aAgt A*]) ∈ *chan s* ∧ — recv msg 3

— actions:
— record session key
s1 = *s*⦇
runs := (*runs s*)(*Rb* ↦ (*Resp*, [*A*, *B*], [*aKey Kab*])),
chan := *insert* (*dAuth Kab* (*Msg* [*aNon Nb*])) (*chan s*)
⦈
}

definition — by *A*, refines *m1-step6*
m2-step6 :: [*rid-t*, *agent*, *agent*, *nonce*, *nonce*, *key*] ⇒ *m2-trans*
where
m2-step6 Ra A B Na Nb Kab ≡ {(*s*, *s'*).
runs s Ra = *Some* (*Init*, [*A*, *B*], [*aKey Kab*]) ∧ — key recv'd before
Na = *Ra*$*na* ∧

dAuth Kab (*Msg* [*aNon Nb*]) ∈ *chan s* ∧ — receive *M4*

— actions:
$s' = s$⦇
$runs := (runs\ s)(Ra \mapsto (Init, [A, B], [aKey\ Kab, aNon\ Nb]))$,
$chan := insert\ (dAuth\ Kab\ (Msg\ [aNon\ Nb, aNon\ Nb]))\ (chan\ s)$
⦈
}

definition — by B, refines *m1-step6*
m2-step7 :: [*rid-t*, *agent*, *agent*, *nonce*, *key*] ⇒ *m2-trans*
where
m2-step7 Rb A B Nb Kab ≡ {(*s*, *s′*).
runs s Rb = *Some* (*Resp*, [*A*, *B*], [*aKey Kab*]) ∧ — key recv'd before
Nb = *Rb*$*nb* ∧

dAuth Kab (*Msg* [*aNon Nb*, *aNon Nb*]) ∈ *chan s* ∧ — receive *M5*

— actions: (redundant) update local state marks successful termination
$s' = s$⦇
runs := (*runs s*)(*Rb* ↦ (*Resp*, [*A*, *B*], [*aKey Kab*, *END*]))
⦈
}

Intruder fake event.

definition — refines *m1-leak*
m2-leak :: [*rid-t*, *rid-t*, *rid-t*, *agent*, *agent*] ⇒ *m2-trans*
where
m2-leak Rs Ra Rb A B ≡ {(*s*, *s1*).
— guards:
runs s Rs = *Some* (*Serv*, [*A*, *B*], [*aNon* (*Ra*$*na*)]) ∧
runs s Ra = *Some* (*Init*, [*A*, *B*], [*aKey* (*sesK* (*Rs*$*sk*)), *aNon* (*Rb*$*nb*)]) ∧
runs s Rb = *Some* (*Resp*, [*A*, *B*], [*aKey* (*sesK* (*Rs*$*sk*)), *END*]) ∧

— actions:
s1 = *s*⦇ *leak* := *insert* (*sesK* (*Rs*$*sk*), *Ra*$*na*, *Rb*$*nb*) (*leak s*),
chan := *insert* (*Insec undefined undefined* (*Msg* [*aKey* (*sesK* (*Rs*$*sk*))])) (*chan s*) ⦈
}

definition — refines *Id*
m2-fake :: *m2-trans*
where
m2-fake ≡ {(*s*, *s1*).

— actions:
s1 = *s*⦇
chan := *fake ik0* (*dom* (*runs s*)) (*chan s*)
⦈
}

3.10.3 Transition system

definition
m2-init :: *m2-pred*

where
  m2-init ≡ { (|
    runs = *Map.empty*,
    leak = *corrKey* × {*undefined*} × {*undefined*},
    chan = {} |)
  }

definition
  m2-trans :: *m2-trans* **where**
  m2-trans ≡ (⋃ *A B Ra Rb Rs Na Nb Kab*.
    m2-step1 Ra A B Na ∪
    m2-step2 Rb A B ∪
    m2-step3 Rs A B Na Kab ∪
    m2-step4 Ra A B Na Kab ∪
    m2-step5 Rb A B Nb Kab ∪
    m2-step6 Ra A B Na Nb Kab ∪
    m2-step7 Rb A B Nb Kab ∪
    m2-leak Rs Ra Rb A B ∪
    m2-fake ∪
    Id
)

definition
  m2 :: (*m2-state*, *m2-obs*) *spec* **where**
  m2 ≡ (|
    init = *m2-init*,
    trans = *m2-trans*,
    obs = *m2-obs*
  |)

lemmas *m2-loc-defs* =
  m2-def m2-init-def m2-trans-def m2-obs-def
  m2-step1-def m2-step2-def m2-step3-def m2-step4-def m2-step5-def
  m2-step6-def m2-step7-def m2-leak-def m2-fake-def

lemmas *m2-defs* = *m2-loc-defs m1-defs*

3.10.4 Invariants

inv1: Key definedness

All session keys in channel messages stem from existing runs.

definition
  m2-inv1-keys :: *m2-pred*
where
  m2-inv1-keys ≡ {*s*. ∀ *R*.
    aKey (*sesK* (*R*$*sk*)) ∈ *atoms* (*chan s*) ∨ *sesK* (*R*$*sk*) ∈ *Domain* (*leak s*) ⟶
      R ∈ *dom* (*runs s*)
  }

lemmas *m2-inv1-keysI* = *m2-inv1-keys-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv1-keysE* [*elim*] = *m2-inv1-keys-def* [*THEN setc-def-to-elim*, *rule-format*]

lemmas *m2-inv1-keysD* = *m2-inv1-keys-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

Invariance proof.

lemma *PO-m2-inv1-keys-init* [*iff*]:
init m2 ⊆ *m2-inv1-keys*
⟨*proof*⟩

lemma *PO-m2-inv1-keys-trans* [*iff*]:
{*m2-inv1-keys*} *trans m2* {> *m2-inv1-keys*}
⟨*proof*⟩

lemma *PO-m2-inv1-keys* [*iff*]: *reach m2* ⊆ *m2-inv1-keys*
⟨*proof*⟩

inv2: Definedness of used keys

definition
m2-inv2-keys-for :: *m2-pred*
where
m2-inv2-keys-for ≡ {*s*. ∀ *R*.
sesK (*R$sk*) ∈ *keys-for* (*chan s*) ⟶ *R* ∈ *dom* (*runs s*)
}

lemmas *m2-inv2-keys-forI* = *m2-inv2-keys-for-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv2-keys-forE* [*elim*] = *m2-inv2-keys-for-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv2-keys-forD* = *m2-inv2-keys-for-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

Invariance proof.

lemma *PO-m2-inv2-keys-for-init* [*iff*]:
init m2 ⊆ *m2-inv2-keys-for*
⟨*proof*⟩

lemma *PO-m2-inv2-keys-for-trans* [*iff*]:
{*m2-inv2-keys-for* ∩ *m2-inv1-keys*} *trans m2* {> *m2-inv2-keys-for*}
⟨*proof*⟩

lemma *PO-m2-inv2-keys-for* [*iff*]: *reach m2* ⊆ *m2-inv2-keys-for*
⟨*proof*⟩

Useful application of invariant.

lemma *m2-inv2-keys-for--extr-insert-key*:
⟦ *R* ∉ *dom* (*runs s*); *s* ∈ *m2-inv2-keys-for* ⟧
⟹ *extr* (*insert* (*aKey* (*sesK* (*R$sk*))) *T*) (*chan s*) = *insert* (*aKey* (*sesK* (*R$sk*))) (*extr T* (*chan s*))
⟨*proof*⟩

inv2b: leaked keys include corrupted ones

definition
m2-inv2b-corrKey-leaked :: *m2-pred*
where
m2-inv2b-corrKey-leaked ≡ {*s*. ∀ *K*.
K ∈ *corrKey* ⟶ *K* ∈ *Domain* (*leak s*)

}

lemmas *m2-inv2b-corrKey-leakedI* = *m2-inv2b-corrKey-leaked-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv2b-corrKey-leakedE* [*elim*] = *m2-inv2b-corrKey-leaked-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv2b-corrKey-leakedD* = *m2-inv2b-corrKey-leaked-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

Invariance proof.

lemma *PO-m2-inv2b-corrKey-leaked-init* [*iff*]:
init m2 ⊆ *m2-inv2b-corrKey-leaked*
⟨*proof*⟩

lemma *PO-m2-inv2b-corrKey-leaked-trans* [*iff*]:
{*m2-inv2b-corrKey-leaked* ∩ *m2-inv1-keys*} *trans m2* {> *m2-inv2b-corrKey-leaked*}
⟨*proof*⟩

lemma *PO-m2-inv2b-corrKey-leaked* [*iff*]: *reach m2* ⊆ *m2-inv2b-corrKey-leaked*
⟨*proof*⟩

inv3a: Session key compromise

A L2 version of a session key comprise invariant. Roughly, it states that adding a set of keys *KK* to the parameter *T* of *extr* does not help the intruder to extract keys other than those in *KK* or extractable without adding *KK*.

definition
m2-inv3a-sesK-compr :: *m2-state set*
where
m2-inv3a-sesK-compr ≡ {*s*. ∀ *K KK*.
~~*KK* ⊆ *range sesK* ⟶~~
aKey K ∈ *extr* (*aKey*'*KK* ∪ *ik0*) (*chan s*) ⟷ (*K* ∈ *KK* ∨ *aKey K* ∈ *extr ik0* (*chan s*))
}

lemmas *m2-inv3a-sesK-comprI* = *m2-inv3a-sesK-compr-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv3a-sesK-comprE* [*elim*] = *m2-inv3a-sesK-compr-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv3a-sesK-comprD* = *m2-inv3a-sesK-compr-def* [*THEN setc-def-to-dest*, *rule-format*]

Additional lemma to get the keys in front

lemmas *insert-commute-aKey* = *insert-commute* [**where** *x*=*aKey K* **for** *K*]

lemmas *m2-inv3a-sesK-compr-simps* =
m2-inv3a-sesK-comprD
m2-inv3a-sesK-comprD [**where** *KK*={*Kab*} **for** *Kab*, *simplified*]
m2-inv3a-sesK-comprD [**where** *KK*=*insert Kab KK* **for** *Kab KK*, *simplified*]
insert-commute-aKey

lemma *PO-m2-inv3a-sesK-compr-init* [*iff*]:
init m2 ⊆ *m2-inv3a-sesK-compr*
⟨*proof*⟩

lemma *PO-m2-inv3a-sesK-compr-trans* [*iff*]:

{*m2-inv3a-sesK-compr*} *trans m2* {> *m2-inv3a-sesK-compr*}
⟨*proof*⟩

lemma *PO-m2-inv3a-sesK-compr* [*iff*]: *reach m2* ⊆ *m2-inv3a-sesK-compr*
⟨*proof*⟩

inv3b: Session key compromise for nonces

A variant of the above for nonces. Roughly, it states that adding a set of keys *KK* to the parameter *T* of *extr* does not help the intruder to extract more nonces than those extractable without adding *KK*.

NOTE: This lemma is only needed at the next refinement level.

definition
m2-inv3b-sesK-compr-non :: *m2-state set*
where
m2-inv3b-sesK-compr-non ≡ {*s*. ∀ *N KK*.
~~*KK* ⊆ *range sesK* ⟶~~
aNon N ∈ *extr* (*aKey*‘*KK* ∪ *ik0*) (*chan s*) ⟷ *aNon N* ∈ *extr ik0* (*chan s*)
}

lemmas *m2-inv3b-sesK-compr-nonI* = *m2-inv3b-sesK-compr-non-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv3b-sesK-compr-nonE* [*elim*] = *m2-inv3b-sesK-compr-non-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv3b-sesK-compr-nonD* = *m2-inv3b-sesK-compr-non-def* [*THEN setc-def-to-dest*, *rule-format*]

lemmas *m2-inv3b-sesK-compr-non-simps* =
m2-inv3b-sesK-compr-nonD
m2-inv3b-sesK-compr-nonD [**where** *KK*={*Kab*} **for** *Kab*, *simplified*]
m2-inv3b-sesK-compr-nonD [**where** *KK*=*insert Kab KK* **for** *Kab KK*, *simplified*]
insert-commute-aKey — to get the keys to the front

lemma *PO-m2-inv3b-sesK-compr-non-init* [*iff*]:
init m2 ⊆ *m2-inv3b-sesK-compr-non*
⟨*proof*⟩

lemma *PO-m2-inv3b-sesK-compr-non-trans* [*iff*]:
{*m2-inv3b-sesK-compr-non*} *trans m2* {> *m2-inv3b-sesK-compr-non*}
⟨*proof*⟩

lemma *PO-m2-inv3b-sesK-compr-non* [*iff*]: *reach m2* ⊆ *m2-inv3b-sesK-compr-non*
⟨*proof*⟩

inv3: Lost session keys

inv3: Lost session keys were generated by the server for at least one bad agent. This invariant is needed in the proof of the strengthening of the authorization guards in steps 4 and 5 (e.g., (*Kab*, *A*) ∈ *azC* (*runs s*) for the initiator's step4).

definition
m2-inv3-extrKey :: *m2-state set*

where
  m2-inv3-extrKey ≡ {*s*. ∀ *K*.
    aKey K ∈ *extr ik0* (*chan s*) ⟶ *K* ∉ *corrKey* ⟶
      (∃ *R A′ B′ Na′*. *K* = *sesK* (*R$sk*) ∧
        runs s R = *Some* (*Serv*, [*A′*, *B′*], [*aNon Na′*]) ∧
        (*A′* ∈ *bad* ∨ *B′* ∈ *bad* ∨ (∃ *Nb′*. (*K*, *Na′*, *Nb′*) ∈ *leak s*)))
  }

lemmas *m2-inv3-extrKeyI* = *m2-inv3-extrKey-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv3-extrKeyE* [*elim*] = *m2-inv3-extrKey-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv3-extrKeyD* = *m2-inv3-extrKey-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

lemma *PO-m2-inv3-extrKey-init* [*iff*]:
  init m2 ⊆ *m2-inv3-extrKey*
⟨*proof*⟩

lemma *PO-m2-inv3-extrKey-trans* [*iff*]:
  {*m2-inv3-extrKey* ∩ *m2-inv3a-sesK-compr*} *trans m2* {> *m2-inv3-extrKey*}
⟨*proof*⟩

lemma *PO-m2-inv3-extrKey* [*iff*]: *reach m2* ⊆ *m2-inv3-extrKey*
⟨*proof*⟩

inv4: Secure channel and message 2

inv4: Secure messages to honest agents and server state; one variant for each of M2 and M3. Note that the one for M2 is stronger than the one for M3.

definition
  m2-inv4-M2 :: *m2-pred*
where
  m2-inv4-M2 ≡ {*s*. ∀ *A B Na Kab*.
    Secure Sv A (*Msg* [*aNon Na*, *aAgt B*, *aKey Kab*]) ∈ *chan s* ⟶ *A* ∈ *good* ⟶
      (∃ *Rs*. *Kab* = *sesK* (*Rs$sk*) ∧ *runs s Rs* = *Some* (*Serv*, [*A*, *B*], [*aNon Na*]))
  }

lemmas *m2-inv4-M2I* = *m2-inv4-M2-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv4-M2E* [*elim*] = *m2-inv4-M2-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv4-M2D* = *m2-inv4-M2-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

Invariance proof.

lemma *PO-m2-inv4-M2-init* [*iff*]:
  init m2 ⊆ *m2-inv4-M2*
⟨*proof*⟩

lemma *PO-m2-inv4-M2-trans* [*iff*]:
  {*m2-inv4-M2*} *trans m2* {> *m2-inv4-M2*}
⟨*proof*⟩

lemma *PO-m2-inv4-M2* [*iff*]: *reach m2* ⊆ *m2-inv4-M2*
⟨*proof*⟩

inv4b: Secure channel and message 3

definition
m2-inv4-M3 :: *m2-pred*
where
m2-inv4-M3 ≡ {*s*. ∀ *A B Kab*.
Secure Sv B (*Msg* [*aKey Kab*, *aAgt A*]) ∈ *chan s* ⟶ *B* ∈ *good* ⟶
(∃ *Rs Na*. *Kab* = *sesK* (*Rs*$*sk*) ∧ *runs s Rs* = *Some* (*Serv*, [*A*, *B*], [*aNon Na*]))
}

lemmas *m2-inv4-M3I* = *m2-inv4-M3-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv4-M3E* [*elim*] = *m2-inv4-M3-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv4-M3D* = *m2-inv4-M3-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

Invariance proof.

lemma *PO-m2-inv4-M3-init* [*iff*]:
init m2 ⊆ *m2-inv4-M3*
⟨*proof*⟩

lemma *PO-m2-inv4-M3-trans* [*iff*]:
{*m2-inv4-M3*} *trans m2* {> *m2-inv4-M3*}
⟨*proof*⟩

lemma *PO-m2-inv4-M3* [*iff*]: *reach m2* ⊆ *m2-inv4-M3*
⟨*proof*⟩

Consequence needed in proof of inv8/step5

lemma *m2-inv4-M2-M3-unique-names*:
assumes
Secure Sv A′ (*Msg* [*aNon Na*, *aAgt B′*, *aKey Kab*]) ∈ *chan s*
Secure Sv B (*Msg* [*aKey Kab*, *aAgt A*]) ∈ *chan s aKey Kab* ∉ *extr ik0* (*chan s*)
s ∈ *m2-inv4-M2 s* ∈ *m2-inv4-M3*
shows
A = *A′* ∧ *B* = *B′*
⟨*proof*⟩

More consequences of invariants. Needed in ref/step4 and ref/step5 respectively to show the strengthening of the authorization guards.

lemma *m2-inv34-M2-authorized*:
assumes *Secure Sv A* (*Msg* [*aNon N*, *aAgt B*, *aKey K*]) ∈ *chan s*
s ∈ *m2-inv4-M2 s* ∈ *m2-inv3-extrKey s* ∈ *m2-inv2b-corrKey-leaked*
K ∉ *Domain* (*leak s*)
shows (*K*, *A*) ∈ *azC* (*runs s*)
⟨*proof*⟩

lemma *m2-inv34-M3-authorized*:
assumes *Secure Sv B* (*Msg* [*aKey K*, *aAgt A*]) ∈ *chan s*
s ∈ *m2-inv4-M3 s* ∈ *m2-inv3-extrKey s* ∈ *m2-inv2b-corrKey-leaked*
K ∉ *Domain* (*leak s*)
shows (*K*, *B*) ∈ *azC* (*runs s*)
⟨*proof*⟩

inv5 (derived): Key secrecy for server

inv5: Key secrecy from server perspective. This invariant links the abstract notion of key secrecy to the intruder key knowledge.

definition
  m2-inv5-ikk-sv :: *m2-pred*
where
  m2-inv5-ikk-sv ≡ {*s*. ∀ *Rs A B Na al*.
    runs s Rs = *Some* (*Serv*, [*A*, *B*], *aNon Na* # *al*) ⟶ *A* ∈ *good* ⟶ *B* ∈ *good* ⟶
    aKey (*sesK* (*Rs*$*sk*)) ∈ *extr ik0* (*chan s*) ⟶
      (∃ *Nb'*. (*sesK* (*Rs*$*sk*), *Na*, *Nb'*) ∈ *leak s*)
  }

lemmas *m2-inv5-ikk-svI* = *m2-inv5-ikk-sv-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv5-ikk-svE* [*elim*] = *m2-inv5-ikk-sv-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv5-ikk-svD* = *m2-inv5-ikk-sv-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

Invariance proof. This invariant follows from *m2-inv3-extrKey*.

lemma *m2-inv5-ikk-sv-derived*:
  s ∈ *m2-inv3-extrKey* ⟹ *s* ∈ *m2-inv5-ikk-sv*
⟨*proof*⟩

lemma *PO-m2-inv5-ikk-sv* [*iff*]: *reach m2* ⊆ *m2-inv5-ikk-sv*
⟨*proof*⟩

inv6 (derived): Key secrecy for initiator

This invariant is derivable (see below).

definition
  m2-inv6-ikk-init :: *m2-pred*
where
  m2-inv6-ikk-init ≡ {*s*. ∀ *Ra K A B al*.
    runs s Ra = *Some* (*Init*, [*A*, *B*], *aKey K* # *al*) ⟶ *A* ∈ *good* ⟶ *B* ∈ *good* ⟶
    aKey K ∈ *extr ik0* (*chan s*) ⟶
      (∃ *Nb'*. (*K*, *Ra* $ *na*, *Nb'*) ∈ *leak s*)
  }

lemmas *m2-inv6-ikk-initI* = *m2-inv6-ikk-init-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv6-ikk-initE* [*elim*] = *m2-inv6-ikk-init-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv6-ikk-initD* = *m2-inv6-ikk-init-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

inv7 (derived): Key secrecy for responder

This invariant is derivable (see below).

definition
  m2-inv7-ikk-resp :: *m2-pred*
where
  m2-inv7-ikk-resp ≡ {*s*. ∀ *Rb K A B al*.
    runs s Rb = *Some* (*Resp*, [*A*, *B*], *aKey K* # *al*) ⟶ *A* ∈ *good* ⟶ *B* ∈ *good* ⟶
    aKey K ∈ *extr ik0* (*chan s*) ⟶
      K ∈ *Domain* (*leak s*)

}

lemmas *m2-inv7-ikk-respI* = *m2-inv7-ikk-resp-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv7-ikk-respE* [*elim*] = *m2-inv7-ikk-resp-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv7-ikk-respD* = *m2-inv7-ikk-resp-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

inv8: Relating M2 and M4 to the responder state

This invariant relates messages M2 and M4 to the responder's state. It is required in the refinement of step 6 to prove that the initiator agrees with the responder on (A, B, Nb, Kab).

definition
m2-inv8-M4 :: *m2-pred*
where
m2-inv8-M4 ≡ {*s*. ∀ *Kab A B Na Nb*.
Secure Sv A (*Msg* [*aNon Na*, *aAgt B*, *aKey Kab*]) ∈ *chan s* ⟶
dAuth Kab (*Msg* [*aNon Nb*]) ∈ *chan s* ⟶
aKey Kab ∉ *extr ik0* (*chan s*) ⟶
(∃ *Rb*. *Nb* = *Rb$nb* ∧ (∃ *al*. *runs s Rb* = *Some* (*Resp*, [*A*, *B*], *aKey Kab* # *al*)))
}

lemmas *m2-inv8-M4I* = *m2-inv8-M4-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv8-M4E* [*elim*] = *m2-inv8-M4-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv8-M4D* = *m2-inv8-M4-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

Invariance proof.

lemma *PO-m2-inv8-M4-step1*:
{*m2-inv8-M4*} *m2-step1 Ra A B Na* {> *m2-inv8-M4*}
⟨*proof*⟩

lemma *PO-m2-inv8-M4-step2*:
{*m2-inv8-M4*} *m2-step2 Rb A B* {> *m2-inv8-M4*}
⟨*proof*⟩

lemma *PO-m2-inv8-M4-step3*:
{*m2-inv8-M4* ∩ *m2-inv2-keys-for*} *m2-step3 Rs A B Na Kab* {> *m2-inv8-M4*}
⟨*proof*⟩

lemma *PO-m2-inv8-M4-step4*:
{*m2-inv8-M4*} *m2-step4 Ra A B Na Kab* {> *m2-inv8-M4*}
⟨*proof*⟩

lemma *PO-m2-inv8-M4-step5*:
{*m2-inv8-M4* ∩ *m2-inv4-M3* ∩ *m2-inv4-M2*}
m2-step5 Rb A B Nb Kab
{> *m2-inv8-M4*}
⟨*proof*⟩

lemma *PO-m2-inv8-M4-step6*:
{*m2-inv8-M4*} *m2-step6 Ra A B Na Nb Kab* {> *m2-inv8-M4*}
⟨*proof*⟩

lemma *PO-m2-inv8-M4-step7*:
{*m2-inv8-M4*} *m2-step7 Rb A B Nb Kab* {> *m2-inv8-M4*}
⟨*proof*⟩

lemma *PO-m2-inv8-M4-leak*:
{*m2-inv8-M4* ∩ *m2-inv3a-sesK-compr*} *m2-leak Rs Ra Rb A B* {> *m2-inv8-M4*}
⟨*proof*⟩

lemma *PO-m2-inv8-M4-fake*:
{*m2-inv8-M4*} *m2-fake* {> *m2-inv8-M4*}
⟨*proof*⟩

All together now..

lemmas *PO-m2-inv8-M4-lemmas* =
PO-m2-inv8-M4-step1 PO-m2-inv8-M4-step2 PO-m2-inv8-M4-step3
PO-m2-inv8-M4-step4 PO-m2-inv8-M4-step5 PO-m2-inv8-M4-step6
PO-m2-inv8-M4-step7 PO-m2-inv8-M4-leak PO-m2-inv8-M4-fake

lemma *PO-m2-inv8-M4-init* [*iff*]:
init m2 ⊆ *m2-inv8-M4*
⟨*proof*⟩

lemma *PO-m2-inv8-M4-trans* [*iff*]:
{*m2-inv8-M4* ∩ *m2-inv4-M3* ∩ *m2-inv4-M2* ∩ *m2-inv3a-sesK-compr* ∩ *m2-inv2-keys-for*}
trans m2
{> *m2-inv8-M4*}
⟨*proof*⟩

lemma *PO-m2-inv8-M4* [*iff*]: *reach m2* ⊆ *m2-inv8-M4*
⟨*proof*⟩

inv8a: Relating the initiator state to M2

definition
m2-inv8a-init-M2 :: *m2-pred*
where
m2-inv8a-init-M2 ≡ {*s*. ∀ *Ra A B Kab al*.
runs s Ra = *Some* (*Init*, [*A*, *B*], *aKey Kab* # *al*) ⟶
Secure Sv A (*Msg* [*aNon* (*Ra$na*), *aAgt B*, *aKey Kab*]) ∈ *chan s*
}

lemmas *m2-inv8a-init-M2I* = *m2-inv8a-init-M2-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv8a-init-M2E* [*elim*] = *m2-inv8a-init-M2-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv8a-init-M2D* = *m2-inv8a-init-M2-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

Invariance proof.

lemma *PO-m2-inv8a-init-M2-init* [*iff*]:
init m2 ⊆ *m2-inv8a-init-M2*
⟨*proof*⟩

lemma *PO-m2-inv8a-init-M2-trans* [*iff*]:
{*m2-inv8a-init-M2*}

trans m2
{> *m2-inv8a-init-M2*}
⟨*proof*⟩

lemma *PO-m2-inv8a-init-M2* [*iff*]: *reach m2* ⊆ *m2-inv8a-init-M2*
⟨*proof*⟩

inv9a: Relating the responder state to M3

definition
m2-inv9a-resp-M3 :: *m2-pred*
where
m2-inv9a-resp-M3 ≡ {*s*. ∀ *Rb A B Kab al*.
runs s Rb = *Some* (*Resp*, [*A*, *B*], *aKey Kab* # *al*) ⟶
Secure Sv B (*Msg* [*aKey Kab*, *aAgt A*]) ∈ *chan s*
}

lemmas *m2-inv9a-resp-M3I* = *m2-inv9a-resp-M3-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv9a-resp-M3E* [*elim*] = *m2-inv9a-resp-M3-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv9a-resp-M3D* = *m2-inv9a-resp-M3-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

Invariance proof.

lemma *PO-m2-inv9a-resp-M3-init* [*iff*]:
init m2 ⊆ *m2-inv9a-resp-M3*
⟨*proof*⟩

lemma *PO-m2-inv9a-resp-M3-trans* [*iff*]:
{*m2-inv9a-resp-M3*}
trans m2
{> *m2-inv9a-resp-M3*}
⟨*proof*⟩

lemma *PO-m2-inv9a-resp-M3* [*iff*]: *reach m2* ⊆ *m2-inv9a-resp-M3*
⟨*proof*⟩

inv9: Relating M3 and M5 to the initiator state

This invariant relates message M5 to the initiator's state. It is required in step 7 of the refinement to prove that the initiator agrees with the responder on (A, B, Nb, Kab).

definition
m2-inv9-M5 :: *m2-pred*
where
m2-inv9-M5 ≡ {*s*. ∀ *Kab A B Nb*.
Secure Sv B (*Msg* [*aKey Kab*, *aAgt A*]) ∈ *chan s* ⟶
dAuth Kab (*Msg* [*aNon Nb*, *aNon Nb*]) ∈ *chan s* ⟶
aKey Kab ∉ *extr ik0* (*chan s*) ⟶
(∃ *Ra*. *runs s Ra* = *Some* (*Init*, [*A*, *B*], [*aKey Kab*, *aNon Nb*]))
}

lemmas *m2-inv9-M5I* = *m2-inv9-M5-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv9-M5E* [*elim*] = *m2-inv9-M5-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv9-M5D* = *m2-inv9-M5-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

Invariance proof.

lemma *PO-m2-inv9-M5-step1*:
{*m2-inv9-M5*} *m2-step1 Ra A B Na* {> *m2-inv9-M5*}
⟨*proof*⟩

lemma *PO-m2-inv9-M5-step2*:
{*m2-inv9-M5*} *m2-step2 Rb A B* {> *m2-inv9-M5*}
⟨*proof*⟩

lemma *PO-m2-inv9-M5-step3*:
{*m2-inv9-M5* ∩ *m2-inv2-keys-for*} *m2-step3 Rs A B Na Kab* {> *m2-inv9-M5*}
⟨*proof*⟩

lemma *PO-m2-inv9-M5-step4*:
{*m2-inv9-M5*} *m2-step4 Ra A B Na Kab* {> *m2-inv9-M5*}
⟨*proof*⟩

lemma *PO-m2-inv9-M5-step5*:
{*m2-inv9-M5*} *m2-step5 Rb A B Nb Kab* {> *m2-inv9-M5*}
⟨*proof*⟩

lemma *PO-m2-inv9-M5-step6*:
{*m2-inv9-M5* ∩ *m2-inv8a-init-M2* ∩ *m2-inv9a-resp-M3* ∩ *m2-inv4-M2* ∩ *m2-inv4-M3*}
m2-step6 Ra A B Na Nb Kab
{> *m2-inv9-M5*}
⟨*proof*⟩

lemma *PO-m2-inv9-M5-step7*:
{*m2-inv9-M5*} *m2-step7 Rb A B Nb Kab* {> *m2-inv9-M5*}
⟨*proof*⟩

lemma *PO-m2-inv9-M5-leak*:
{*m2-inv9-M5* ∩ *m2-inv3a-sesK-compr*} *m2-leak Rs Ra Rb A B* {> *m2-inv9-M5*}
⟨*proof*⟩

lemma *PO-m2-inv9-M5-fake*:
{*m2-inv9-M5*} *m2-fake* {> *m2-inv9-M5*}
⟨*proof*⟩

All together now.

lemmas *PO-m2-inv9-M5-lemmas* =
PO-m2-inv9-M5-step1 PO-m2-inv9-M5-step2 PO-m2-inv9-M5-step3
PO-m2-inv9-M5-step4 PO-m2-inv9-M5-step5 PO-m2-inv9-M5-step6
PO-m2-inv9-M5-step7 PO-m2-inv9-M5-leak PO-m2-inv9-M5-fake

lemma *PO-m2-inv9-M5-init* [*iff*]:
init m2 ⊆ *m2-inv9-M5*
⟨*proof*⟩

lemma *PO-m2-inv9-M5-trans* [*iff*]:
{*m2-inv9-M5* ∩ *m2-inv8a-init-M2* ∩ *m2-inv9a-resp-M3* ∩
m2-inv4-M2 ∩ *m2-inv4-M3* ∩ *m2-inv3a-sesK-compr* ∩ *m2-inv2-keys-for*}
trans m2
{> *m2-inv9-M5*}
⟨*proof*⟩

lemma *PO-m2-inv9-M5* [*iff*]: *reach m2* ⊆ *m2-inv9-M5*
⟨*proof*⟩

3.10.5 Refinement

The simulation relation. This is a pure superposition refinement.

definition
R12 :: (*m1-state* × *m2-state*) *set* **where**
R12 ≡ {(*s*, *t*). *runs s* = *runs t* ∧ *leak s* = *leak t*}

The mediator function projects on the local states.

definition
med21 :: *m2-obs* ⇒ *m1-obs* **where**
med21 o2 = (| *runs* = *runs o2*, *leak* = *leak o2* |)

Refinement proof.

lemma *PO-m2-step1-refines-m1-step1*:
{*R12*}
(*m1-step1 Ra A B Na*), (*m2-step1 Ra A B Na*)
{> *R12*}
⟨*proof*⟩

lemma *PO-m2-step2-refines-m1-step2*:
{*R12*}
(*m1-step2 Rb A B*), (*m2-step2 Rb A B*)
{> *R12*}
⟨*proof*⟩

lemma *PO-m2-step3-refines-m1-step3*:
{*R12*}
(*m1-step3 Rs A B Na Kab*), (*m2-step3 Rs A B Na Kab*)
{> *R12*}
⟨*proof*⟩

lemma *PO-m2-step4-refines-m1-step4*:
{*R12* ∩ *UNIV* × (*m2-inv4-M2* ∩ *m2-inv3-extrKey* ∩ *m2-inv2b-corrKey-leaked*)}
(*m1-step4 Ra A B Na Kab*), (*m2-step4 Ra A B Na Kab*)
{> *R12*}
⟨*proof*⟩

lemma *PO-m2-step5-refines-m1-step5*:
{*R12* ∩ *UNIV* × (*m2-inv4-M3* ∩ *m2-inv3-extrKey* ∩ *m2-inv2b-corrKey-leaked*)}
(*m1-step5 Rb A B Nb Kab*), (*m2-step5 Rb A B Nb Kab*)
{> *R12*}

⟨*proof*⟩

lemma *PO-m2-step6-refines-m1-step6*:
{*R12* ∩ *UNIV* × (*m2-inv8a-init-M2* ∩ *m2-inv8-M4* ∩ *m2-inv6-ikk-init*)}
(*m1-step6 Ra A B Na Nb Kab*), (*m2-step6 Ra A B Na Nb Kab*)
{> *R12*}
⟨*proof*⟩

lemma *PO-m2-step7-refines-m1-step7*:
{*R12* ∩ *UNIV* × (*m2-inv9-M5* ∩ *m2-inv9a-resp-M3* ∩ *m2-inv7-ikk-resp*)}
(*m1-step7 Rb A B Nb Kab*), (*m2-step7 Rb A B Nb Kab*)
{> *R12*}
⟨*proof*⟩

lemma *PO-m2-leak-refines-leak*:
{*R12*}
m1-leak Rs Ra Rb A B, *m2-leak Rs Ra Rb A B*
{> *R12*}
⟨*proof*⟩

lemma *PO-m2-fake-refines-skip*:
{*R12*}
Id, *m2-fake*
{> *R12*}
⟨*proof*⟩

Consequences of simulation relation and invariants.

lemma *m2-inv6-ikk-init-derived*:
assumes (*s*, *t*) ∈ *R12 s* ∈ *m1-inv2i-serv t* ∈ *m2-inv5-ikk-sv*
shows *t* ∈ *m2-inv6-ikk-init*
⟨*proof*⟩

lemma *m2-inv7-ikk-resp-derived*:
assumes (*s*, *t*) ∈ *R12 s* ∈ *m1-inv2r-serv t* ∈ *m2-inv5-ikk-sv*
shows *t* ∈ *m2-inv7-ikk-resp*
⟨*proof*⟩

All together now...

lemmas *PO-m2-trans-refines-m1-trans* =
PO-m2-step1-refines-m1-step1 PO-m2-step2-refines-m1-step2
PO-m2-step3-refines-m1-step3 PO-m2-step4-refines-m1-step4
PO-m2-step5-refines-m1-step5 PO-m2-step6-refines-m1-step6
PO-m2-step7-refines-m1-step7 PO-m2-leak-refines-leak
PO-m2-fake-refines-skip

lemma *PO-m2-refines-init-m1* [*iff*]:
init m2 ⊆ *R12*‘‘(*init m1*)
⟨*proof*⟩

lemma *PO-m2-refines-trans-m1* [*iff*]:
{*R12* ∩
(*reach m1* ×

(*m2-inv9-M5* ∩ *m2-inv8a-init-M2* ∩ *m2-inv9a-resp-M3* ∩ *m2-inv8-M4* ∩
m2-inv4-M3 ∩ *m2-inv4-M2* ∩ *m2-inv3a-sesK-compr* ∩ *m2-inv3-extrKey* ∩ *m2-inv2b-corrKey-leaked*))}

(*trans m1*), (*trans m2*)
{> *R12*}
⟨*proof*⟩

lemma *PO-obs-consistent-R12* [*iff*]:
obs-consistent R12 med21 m1 m2
⟨*proof*⟩

Refinement result.

lemma *m2-refines-m1* [*iff*]:
refines
(*R12* ∩
(*reach m1* ×
(*m2-inv9-M5* ∩ *m2-inv8a-init-M2* ∩ *m2-inv9a-resp-M3* ∩ *m2-inv8-M4* ∩
m2-inv4-M3 ∩ *m2-inv4-M2* ∩ *m2-inv3a-sesK-compr* ∩ *m2-inv3-extrKey* ∩
m2-inv2b-corrKey-leaked ∩ *m2-inv2-keys-for* ∩ *m2-inv1-keys*)))
med21 m1 m2
⟨*proof*⟩

lemma *m2-implements-m1* [*iff*]:
implements med21 m1 m2
⟨*proof*⟩

3.10.6 Inherited and derived invariants

Show preservation of invariants *m1-inv2i-serv* and *m1-inv2r-serv* from *m1*.

lemma *PO-m2-sat-m1-inv2i-serv* [*iff*]: *reach m2* ⊆ *m1-inv2i-serv*
⟨*proof*⟩

lemma *PO-m2-sat-m1-inv2r-serv* [*iff*]: *reach m2* ⊆ *m1-inv2r-serv*
⟨*proof*⟩

Now we derive the additional invariants for the initiator and the responder (see above for the definitions).

lemma *PO-m2-inv6-init-ikk* [*iff*]: *reach m2* ⊆ *m2-inv6-ikk-init*
⟨*proof*⟩

lemma *PO-m2-inv6-resp-ikk* [*iff*]: *reach m2* ⊆ *m2-inv7-ikk-resp*
⟨*proof*⟩

end

3.11 Needham-Schroeder Shared Key, "parallel" variant (L3)

theory *m3-nssk-par* **imports** *m2-nssk ../Refinement/Message*

begin

We model an abstract version of the Needham-Schroeder Shared Key protocol:

$$\begin{array}{lll} \text{M1.} & A \rightarrow S: & A, B, Na \\ \text{M2.} & S \rightarrow A: & \{Na, B, Kab, \{Kab, A\}_{Kbs}\}_{Kas} \\ \text{M3.} & A \rightarrow B: & \{Kab, A\}_{Kbs} \\ \text{M4.} & B \rightarrow A: & \{Nb\}_{Kab} \\ \text{M5.} & A \rightarrow B: & \{Nb - 1\}_{Kab} \end{array}$$

We model a "parallel" version of the NSSK protocol:

$$\begin{array}{lll} \text{M1.} & A \rightarrow S: & A, B, Na \\ \text{M2.} & S \rightarrow A: & \{Na, B, Kab\}_{Kas} \\ \text{M3.} & S \rightarrow B: & \{Kab, A\}_{Kbs} \\ \text{M4.} & B \rightarrow A: & \{Nb\}_{Kab} \\ \text{M5.} & A \rightarrow B: & \{Nb - 1\}_{Kab} \end{array}$$

Proof tool configuration. Avoid annoying automatic unfolding of *dom*.

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

3.11.1 Setup

Now we can define the initial key knowledge.

overloading *ltkeySetup′* ≡ *ltkeySetup* **begin**
definition *ltkeySetup-def*: *ltkeySetup′* ≡ {(*sharK C*, *A*) | *C A*. *A* = *C* ∨ *A* = *Sv*}
end

lemma *corrKey-shrK-bad* [*simp*]: *corrKey* = *shrK‘bad*
⟨*proof*⟩

3.11.2 State

The secure channels are star-shaped to/from the server. Therefore, we have only one agent in the relation.

record *m3-state* = *m1-state* +
  IK :: *msg set* — intruder knowledge

Observable state: agent's local state.

type-synonym
  m3-obs = *m2-obs*

definition
  m3-obs :: *m3-state* ⇒ *m3-obs* **where**
  m3-obs s ≡ (| *runs* = *runs s*, *leak* = *leak s* |)

type-synonym
  m3-pred = *m3-state set*

type-synonym
  m3-trans = (*m3-state* × *m3-state*) *set*

3.11.3 Events

Protocol events.

definition — by A, refines *m2-step1*
  m3-step1 :: [*rid-t*, *agent*, *agent*, *nonce*] ⇒ *m3-trans*
where
  m3-step1 Ra A B Na ≡ {(*s*, *s1*).

    — guards:
    $Ra \notin dom\ (runs\ s) \wedge$ — *Ra* is fresh
    $Na = Ra\$na \wedge$ — generate nonce *Na*

    — actions:
    s1 = *s*⦇
      runs := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [])),
      IK := *insert* ⦃*Agent A*, *Agent B*, *Nonce Na*⦄ (*IK s*) — send msg 1
    ⦈
  }

definition — by B, refines *m2-step2*
  m3-step2 :: [*rid-t*, *agent*, *agent*] ⇒ *m3-trans*
where
  m3-step2 Rb A B ≡ {(*s*, *s1*).

    — guards:
    $Rb \notin dom\ (runs\ s) \wedge$ — *Rb* is fresh

    — actions:
    — create responder thread
    s1 = *s*⦇
      runs := (*runs s*)(*Rb* ↦ (*Resp*, [*A*, *B*], []))
    ⦈
  }

definition — by *Server*, refines *m2-step3*
  m3-step3 :: [*rid-t*, *agent*, *agent*, *nonce*, *key*] ⇒ *m3-trans*
where
  m3-step3 Rs A B Na Kab ≡ {(*s*, *s1*).
    — guards:
    $Rs \notin dom\ (runs\ s) \wedge$ — fresh server run
    $Kab = sesK\ (Rs\$sk) \wedge$ — fresh session key

    ⦃*Agent A*, *Agent B*, *Nonce Na*⦄ ∈ *IK s* ∧ — recv msg 1

    — actions:
    — record session key and send messages 2 and 3
    — note that last field in server record is for responder nonce
    s1 = *s*⦇
      runs := (*runs s*)(*Rs* ↦ (*Serv*, [*A*, *B*], [*aNon Na*])),
      IK := {*Crypt* (*shrK A*) ⦃*Nonce Na*, *Agent B*, *Key Kab*⦄,
            Crypt (*shrK B*) ⦃*Key Kab*, *Agent A*⦄} ∪ *IK s*
    ⦈

}

definition — by A, refines *m2-step4*
m3-step4 :: [*rid-t*, *agent*, *agent*, *nonce*, *key*] ⇒ *m3-trans*
where
m3-step4 Ra A B Na Kab ≡ {(*s*, *s1*).

— guards:
runs s Ra = *Some* (*Init*, [*A*, *B*], []) ∧
Na = *Ra*$*na* ∧

Crypt (*shrK A*) ⦃*Nonce Na*, *Agent B*, *Key Kab*⦄ ∈ *IK s* ∧ — recv msg 2

— actions:
— record session key
s1 = *s*⦇
runs := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [*aKey Kab*]))
⦈
}

definition — by B, refines *m2-step5*
m3-step5 :: [*rid-t*, *agent*, *agent*, *nonce*, *key*] ⇒ *m3-trans*
where
m3-step5 Rb A B Nb Kab ≡ {(*s*, *s1*).

— guards:
runs s Rb = *Some* (*Resp*, [*A*, *B*], []) ∧
Nb = *Rb*$*nb* ∧

Crypt (*shrK B*) ⦃*Key Kab*, *Agent A*⦄ ∈ *IK s* ∧ — recv msg 3

— actions:
— record session key
s1 = *s*⦇
runs := (*runs s*)(*Rb* ↦ (*Resp*, [*A*, *B*], [*aKey Kab*])),
IK := *insert* (*Crypt Kab* (*Nonce Nb*)) (*IK s*)
⦈
}

definition — by A, refines *m2-step6*
m3-step6 :: [*rid-t*, *agent*, *agent*, *nonce*, *nonce*, *key*] ⇒ *m3-trans*
where
m3-step6 Ra A B Na Nb Kab ≡ {(*s*, *s'*).
runs s Ra = *Some* (*Init*, [*A*, *B*], [*aKey Kab*]) ∧ — key recv'd before
Na = *Ra*$*na* ∧

Crypt Kab (*Nonce Nb*) ∈ *IK s* ∧ — receive *M4*

— actions:
s' = *s*⦇
runs := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [*aKey Kab*, *aNon Nb*])),
IK := *insert* (*Crypt Kab* ⦃*Nonce Nb*, *Nonce Nb*⦄) (*IK s*)
⦈

}

definition — by B, refines *m2-step6*
m3-step7 :: [*rid-t*, *agent*, *agent*, *nonce*, *key*] ⇒ *m3-trans*
where
m3-step7 Rb A B Nb Kab ≡ {(s, s′).
runs s Rb = *Some* (*Resp*, [A, B], [*aKey Kab*]) ∧ — key recv'd before
Nb = *Rb$nb* ∧

Crypt Kab {|*Nonce Nb*, *Nonce Nb*|} ∈ *IK s* ∧ — receive *M5*

— actions: (redundant) update local state marks successful termination
s′ = s(|
runs := (*runs s*)(*Rb* ↦ (*Resp*, [A, B], [*aKey Kab*, *END*]))
|)
}

Session key compromise.

definition — refines *m2-leak*
m3-leak :: [*rid-t*, *rid-t*, *rid-t*, *agent*, *agent*] ⇒ *m3-trans*
where
m3-leak Rs Ra Rb A B ≡ {(s, s1).
— guards:
runs s Rs = *Some* (*Serv*, [A, B], [*aNon* (*Ra$na*)]) ∧
runs s Ra = *Some* (*Init*, [A, B], [*aKey* (*sesK* (*Rs$sk*)), *aNon* (*Rb$nb*)]) ∧
runs s Rb = *Some* (*Resp*, [A, B], [*aKey* (*sesK* (*Rs$sk*)), *END*]) ∧

— actions:
— record session key as leaked and add it to intruder knowledge
s1 = s(| *leak* := *insert* (*sesK* (*Rs$sk*), *Ra$na*, *Rb$nb*) (*leak s*),
IK := *insert* (*Key* (*sesK* (*Rs$sk*))) (*IK s*) |)
}

Intruder fake event.

definition — refines *m2-fake*
m3-DY-fake :: *m3-trans*
where
m3-DY-fake ≡ {(s, s1).

— actions:
s1 = s(|
IK := *synth* (*analz* (*IK s*))
|)
}

3.11.4 Transition system

definition
m3-init :: *m3-state set*
where
m3-init ≡ { (|
runs = *Map.empty*,
leak = *shrK‘bad* × {*undefined*} × {*undefined*},

    IK = *Key`shrK`bad*
  ⦈ }

definition
  m3-trans :: (*m3-state* × *m3-state*) *set* **where**
  m3-trans ≡ (⋃ *Ra Rb Rs A B Na Nb Kab*.
    m3-step1 Ra A B Na ∪
    m3-step2 Rb A B ∪
    m3-step3 Rs A B Na Kab ∪
    m3-step4 Ra A B Na Kab ∪
    m3-step5 Rb A B Nb Kab ∪
    m3-step6 Ra A B Na Nb Kab ∪
    m3-step7 Rb A B Nb Kab ∪
    m3-leak Rs Ra Rb A B ∪
    m3-DY-fake ∪
    Id
)

definition
  m3 :: (*m3-state*, *m3-obs*) *spec* **where**
  m3 ≡ ⦇
    init = *m3-init*,
    trans = *m3-trans*,
    obs = *m3-obs*
  ⦈

lemmas *m3-defs* =
  m3-def m3-init-def m3-trans-def m3-obs-def
  m3-step1-def m3-step2-def m3-step3-def m3-step4-def m3-step5-def
  m3-step6-def m3-step7-def m3-leak-def m3-DY-fake-def

3.11.5 Invariants

Specialized injection that we can apply more aggressively.

lemmas *analz-Inj-IK* = *analz.Inj* [**where** *H*=*IK s* **for** *s*]
lemmas *parts-Inj-IK* = *parts.Inj* [**where** *H*=*IK s* **for** *s*]

declare *parts-Inj-IK* [*dest*!]

declare *analz-into-parts* [*dest*]

inv1: Secrecy of pre-distributed shared keys

inv1: Secrecy of long-term keys

definition
  m3-inv1-lkeysec :: *m3-state set*
where
  m3-inv1-lkeysec ≡ {*s*. ∀ *C*.
    (*Key* (*shrK C*) ∈ *parts* (*IK s*) ⟶ *C* ∈ *bad*) ∧
    (*C* ∈ *bad* ⟶ *Key* (*shrK C*) ∈ *IK s*)
  }

lemmas *m3-inv1-lkeysecI* = *m3-inv1-lkeysec-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv1-lkeysecE* [*elim*] = *m3-inv1-lkeysec-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv1-lkeysecD* = *m3-inv1-lkeysec-def* [*THEN setc-def-to-dest*, *rule-format*]

Invariance proof.

lemma *PO-m3-inv1-lkeysec-init* [*iff*]:
  init m3 ⊆ *m3-inv1-lkeysec*
⟨*proof*⟩

lemma *PO-m3-inv1-lkeysec-trans* [*iff*]:
  {*m3-inv1-lkeysec*} *trans m3* {> *m3-inv1-lkeysec*}
⟨*proof*⟩

lemma *PO-m3-inv1-lkeysec* [*iff*]: *reach m3* ⊆ *m3-inv1-lkeysec*
⟨*proof*⟩

Useful simplifier lemmas

lemma *m3-inv1-lkeysec-for-parts* [*simp*]:
  ⟦ *s* ∈ *m3-inv1-lkeysec* ⟧ ⟹ *Key* (*shrK C*) ∈ *parts* (*IK s*) ⟷ *C* ∈ *bad*
⟨*proof*⟩

lemma *m3-inv1-lkeysec-for-analz* [*simp*]:
  ⟦ *s* ∈ *m3-inv1-lkeysec* ⟧ ⟹ *Key* (*shrK C*) ∈ *analz* (*IK s*) ⟷ *C* ∈ *bad*
⟨*proof*⟩

inv7a: Session keys not used to encrypt other session keys

Session keys are not used to encrypt other keys. Proof requires generalization to sets of session keys.

NOTE: This invariant will be derived from the corresponding L2 invariant using the simulation relation.

definition
  m3-inv7a-sesK-compr :: *m3-pred*
where
  m3-inv7a-sesK-compr ≡ {*s*. ∀ *K KK*.
    KK ⊆ *range sesK* ⟶
    (*Key K* ∈ *analz* (*Key*‘*KK* ∪ (*IK s*))) = (*K* ∈ *KK* ∨ *Key K* ∈ *analz* (*IK s*))
  }

lemmas *m3-inv7a-sesK-comprI* = *m3-inv7a-sesK-compr-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv7a-sesK-comprE* = *m3-inv7a-sesK-compr-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv7a-sesK-comprD* = *m3-inv7a-sesK-compr-def* [*THEN setc-def-to-dest*, *rule-format*]

Additional lemma

lemmas *insert-commute-Key* = *insert-commute* [**where** *x*=*Key K* **for** *K*]

lemmas *m3-inv7a-sesK-compr-simps* =
  m3-inv7a-sesK-comprD
  m3-inv7a-sesK-comprD [**where** *KK*={*Kab*} **for** *Kab*, *simplified*]
  m3-inv7a-sesK-comprD [**where** *KK*=*insert Kab KK* **for** *Kab KK*, *simplified*]
  insert-commute-Key

3.11.6 Refinement

Message abstraction and simulation relation

Abstraction function on sets of messages.

inductive-set
 abs-msg :: *msg set* $\Rightarrow$ *chmsg set*
 for *H* :: *msg set*
where
 am-M1:
  $\{\!|$*Agent A*, *Agent B*, *Nonce Na*$|\!\}$ $\in$ *H*
 $\Longrightarrow$ *Insec A B* (*Msg* [*aNon Na*]) $\in$ *abs-msg H*
| *am-M2*:
  Crypt (*shrK C*) $\{\!|$*Nonce N*, *Agent B*, *Key K*$|\!\}$ $\in$ *H*
 $\Longrightarrow$ *Secure Sv C* (*Msg* [*aNon N*, *aAgt B*, *aKey K*]) $\in$ *abs-msg H*
| *am-M3*:
  Crypt (*shrK C*) $\{\!|$*Key K*, *Agent A*$|\!\}$ $\in$ *H*
 $\Longrightarrow$ *Secure Sv C* (*Msg* [*aKey K*, *aAgt A*]) $\in$ *abs-msg H*
| *am-M4*:
  Crypt K (*Nonce N*) $\in$ *H*
 $\Longrightarrow$ *dAuth K* (*Msg* [*aNon N*]) $\in$ *abs-msg H*
| *am-M5*:
  Crypt K $\{\!|$*Nonce N*, *Nonce N'*$|\!\}$ $\in$ *H*
 $\Longrightarrow$ *dAuth K* (*Msg* [*aNon N*, *aNon N'*]) $\in$ *abs-msg H*

R23: The simulation relation. This is a data refinement of the insecure and secure channels of refinement 2.

definition
 R23-msgs :: (*m2-state* $\times$ *m3-state*) *set* **where**
 R23-msgs $\equiv$ $\{(s, t).$ *abs-msg* (*parts* (*IK t*)) $\subseteq$ *chan s*$\}$

definition
 R23-keys :: (*m2-state* $\times$ *m3-state*) *set* **where** — equivalence!
 R23-keys $\equiv$ $\{(s, t). \forall$ *KK K*. *KK* $\subseteq$ *range sesK* $\longrightarrow$
  Key K $\in$ *analz* (*Key‘KK* $\cup$ *IK t*) $\longleftrightarrow$ *aKey K* $\in$ *extr* (*aKey‘KK* $\cup$ *ik0*) (*chan s*)
 $\}$

definition
 R23-non :: (*m2-state* $\times$ *m3-state*) *set* **where** — only an implication!
 R23-non $\equiv$ $\{(s, t). \forall$ *KK N*. *KK* $\subseteq$ *range sesK* $\longrightarrow$
  Nonce N $\in$ *analz* (*Key‘KK* $\cup$ *IK t*) $\longrightarrow$ *aNon N* $\in$ *extr* (*aKey‘KK* $\cup$ *ik0*) (*chan s*)
 $\}$

definition
 R23-pres :: (*m2-state* $\times$ *m3-state*) *set* **where**
 R23-pres $\equiv$ $\{(s, t).$ *runs s* = *runs t* $\wedge$ *leak s* = *leak t*$\}$

definition
 R23 :: (*m2-state* $\times$ *m3-state*) *set* **where**
 R23 $\equiv$ *R23-msgs* $\cap$ *R23-keys* $\cap$ *R23-non* $\cap$ *R23-pres*

lemmas *R23-defs* =
 R23-def R23-msgs-def R23-keys-def R23-non-def R23-pres-def

The mediator function is the identity here.

definition
med32 :: *m3-obs* ⇒ *m2-obs* **where**
med32 ≡ *id*

lemmas *R23-msgsI* = *R23-msgs-def* [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-msgsE* [*elim*] = *R23-msgs-def* [*THEN rel-def-to-elim*, *simplified*, *rule-format*]

lemmas *R23-keysI* = *R23-keys-def* [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-keysE* [*elim*] = *R23-keys-def* [*THEN rel-def-to-elim*, *simplified*, *rule-format*]
lemmas *R23-keysD* = *R23-keys-def* [*THEN rel-def-to-dest*, *simplified*, *rule-format*]

lemmas *R23-nonI* = *R23-non-def* [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-nonE* [*elim*] = *R23-non-def* [*THEN rel-def-to-elim*, *simplified*, *rule-format*]
lemmas *R23-nonD* = *R23-non-def* [*THEN rel-def-to-dest*, *simplified*, *rule-format*, *rotated 2*]

lemmas *R23-presI* = *R23-pres-def* [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-presE* [*elim*] = *R23-pres-def* [*THEN rel-def-to-elim*, *simplified*, *rule-format*]

lemmas *R23-intros* = *R23-msgsI R23-keysI R23-nonI R23-presI*

Further lemmas: general lemma for simplifier and different instantiations.

lemmas *R23-keys-simps* =
R23-keysD
R23-keysD [**where** *KK*={}, *simplified*]
R23-keysD [**where** *KK*={*K'*} **for** *K'*, *simplified*]
R23-keysD [**where** *KK*=*insert K' KK* **for** *K' KK*, *simplified*, *OF - conjI*]

lemmas *R23-non-dests* =
R23-nonD
R23-nonD [**where** *KK*={}, *simplified*]
R23-nonD [**where** *KK*={*K*} **for** *K*, *simplified*]
R23-nonD [**where** *KK*=*insert K KK* **for** *K KK*, *simplified*, *OF - - conjI*]

General lemmas

General facts about *abs-msg*

declare *abs-msg.intros* [*intro*!]
declare *abs-msg.cases* [*elim*!]

lemma *abs-msg-empty*: *abs-msg* {} = {}
⟨*proof*⟩

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

lemma *abs-msg-mono* [*elim*]:
⟦ *m* ∈ *abs-msg G*; *G* ⊆ *H* ⟧ ⟹ *m* ∈ *abs-msg H*
⟨*proof*⟩

lemma *abs-msg-insert-mono* [*intro*]:
  $[\![\ m \in \textit{abs-msg}\ H\ ]\!] \Longrightarrow m \in \textit{abs-msg}\ (\textit{insert}\ m'\ H)$
⟨*proof*⟩

Facts about *abs-msg* concerning abstraction of fakeable messages. This is crucial for proving the refinement of the intruder event.

lemma *abs-msg-DY-subset-fakeable*:
  $[\![\ (s, t) \in \textit{R23-msgs};\ (s, t) \in \textit{R23-keys};\ (s, t) \in \textit{R23-non};\ t \in \textit{m3-inv1-lkeysec}\ ]\!]$
  $\Longrightarrow \textit{abs-msg}\ (\textit{synth}\ (\textit{analz}\ (\textit{IK}\ t))) \subseteq \textit{fake}\ \textit{ik0}\ (\textit{dom}\ (\textit{runs}\ s))\ (\textit{chan}\ s)$
⟨*proof*⟩

Refinement proof

Pair decomposition. These were set to `elim!`, which is too agressive here.

declare *MPair-analz* [*rule del*, *elim*]
declare *MPair-parts* [*rule del*, *elim*]

Protocol events.

lemma *PO-m3-step1-refines-m2-step1*:
  {*R23*}
    (*m2-step1 Ra A B Na*), (*m3-step1 Ra A B Na*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-step2-refines-m2-step2*:
  {*R23*}
    (*m2-step2 Rb A B*), (*m3-step2 Rb A B*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-step3-refines-m2-step3*:
  {*R23* ∩ (*m2-inv3a-sesK-compr*) × (*m3-inv7a-sesK-compr* ∩ *m3-inv1-lkeysec*)}
    (*m2-step3 Rs A B Na Kab*), (*m3-step3 Rs A B Na Kab*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-step4-refines-m2-step4*:
  {*R23*}
    (*m2-step4 Ra A B Na Kab*), (*m3-step4 Ra A B Na Kab*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-step5-refines-m2-step5*:
  {*R23*}
    (*m2-step5 Rb A B Nb Kab*), (*m3-step5 Rb A B Nb Kab*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-step6-refines-m2-step6*:

{*R23*}
  (*m2-step6 Ra A B Na Nb Kab*), (*m3-step6 Ra A B Na Nb Kab*)
 {> *R23*}
⟨*proof*⟩

lemma *PO-m3-step7-refines-m2-step7*:
{*R23*}
  (*m2-step7 Rb A B Nb Kab*), (*m3-step7 Rb A B Nb Kab*)
 {> *R23*}
⟨*proof*⟩

Intruder events.

lemma *PO-m3-leak-refines-m2-leak*:
{*R23*}
  (*m2-leak Rs Ra Rb A B*), (*m3-leak Rs Ra Rb A B*)
 {>*R23*}
⟨*proof*⟩

lemma *PO-m3-DY-fake-refines-m2-fake*:
{*R23* ∩ *UNIV* × *m3-inv1-lkeysec*}
  m2-fake, *m3-DY-fake*
 {> *R23*}
⟨*proof*⟩

All together now...

lemmas *PO-m3-trans-refines-m2-trans* =
PO-m3-step1-refines-m2-step1 PO-m3-step2-refines-m2-step2
PO-m3-step3-refines-m2-step3 PO-m3-step4-refines-m2-step4
PO-m3-step5-refines-m2-step5 PO-m3-step6-refines-m2-step6
PO-m3-step7-refines-m2-step7 PO-m3-leak-refines-m2-leak
PO-m3-DY-fake-refines-m2-fake

lemma *PO-m3-refines-init-m2* [*iff*]:
init m3 ⊆ *R23*‘‘(*init m2*)
⟨*proof*⟩

lemma *PO-m3-refines-trans-m2* [*iff*]:
{*R23* ∩ (*m2-inv3a-sesK-compr*) × (*m3-inv7a-sesK-compr* ∩ *m3-inv1-lkeysec*)}
  (*trans m2*), (*trans m3*)
 {> *R23*}
⟨*proof*⟩

lemma *PO-m3-observation-consistent* [*iff*]:
obs-consistent R23 med32 m2 m3
⟨*proof*⟩

Refinement result.

lemma *m3-refines-m2* [*iff*]:
refines (*R23* ∩ *m2-inv3a-sesK-compr* × *m3-inv1-lkeysec*)
  med32 m2 m3
⟨*proof*⟩

lemma *m3-implements-m2* [*iff*]:
  implements med32 m2 m3
⟨*proof*⟩

3.11.7 Inherited invariants

inv4 (derived): Key secrecy for initiator

definition
  m3-inv4-ikk-init :: *m3-state set*
where
  m3-inv4-ikk-init ≡ {*s*. ∀ *Ra K A B al*.
    runs s Ra = *Some* (*Init*, [*A*, *B*], *aKey K* # *al*) ⟶ *A* ∈ *good* ⟶ *B* ∈ *good* ⟶
    Key K ∈ *analz* (*IK s*) ⟶
      (∃ *Nb*′. (*K*, *Ra* $ *na*, *Nb*′) ∈ *leak s*)
  }

lemmas *m3-inv4-ikk-initI* = *m3-inv4-ikk-init-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv4-ikk-initE* [*elim*] = *m3-inv4-ikk-init-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv4-ikk-initD* = *m3-inv4-ikk-init-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

lemma *PO-m3-inv4-ikk-init*: *reach m3* ⊆ *m3-inv4-ikk-init*
⟨*proof*⟩

inv5 (derived): Key secrecy for responder

definition
  m3-inv5-ikk-resp :: *m3-state set*
where
  m3-inv5-ikk-resp ≡ {*s*. ∀ *Rb K A B al*.
    runs s Rb = *Some* (*Resp*, [*A*, *B*], *aKey K* # *al*) ⟶ *A* ∈ *good* ⟶ *B* ∈ *good* ⟶
    Key K ∈ *analz* (*IK s*) ⟶
      K ∈ *Domain* (*leak s*)
  }

lemmas *m3-inv5-ikk-respI* = *m3-inv5-ikk-resp-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv5-ikk-respE* [*elim*] = *m3-inv5-ikk-resp-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv5-ikk-respD* = *m3-inv5-ikk-resp-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

lemma *PO-m3-inv4-ikk-resp*: *reach m3* ⊆ *m3-inv5-ikk-resp*
⟨*proof*⟩

end

3.12 Needham-Schroeder Shared Key (L3)

theory *m3-nssk* **imports** *m2-nssk* ../*Refinement*/*Message*
begin

We model an abstract version of the Needham-Schroeder Shared Key protocol:

$$\begin{array}{lll}
\text{M1.} & A \to S: & A, B, Na \\
\text{M2.} & S \to A: & \{Na, B, Kab, \{Kab, A\}_{Kbs}\}_{Kas} \\
\text{M3.} & A \to B: & \{Kab, A\}_{Kbs} \\
\text{M4.} & B \to A: & \{Nb\}_{Kab} \\
\text{M5.} & A \to B: & \{Nb - 1\}_{Kab}
\end{array}$$

This refinement works with a single insecure channel and introduces the full Dolev-Yao intruder.

Proof tool configuration. Avoid annoying automatic unfolding of *dom*.

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

3.12.1 Setup

Now we can define the initial key knowledge.

overloading *ltkeySetup'* ≡ *ltkeySetup* **begin**
definition *ltkeySetup-def*: *ltkeySetup'* ≡ {(*sharK C*, *A*) | *C A*. *A* = *C* ∨ *A* = *Sv*}
end

lemma *corrKey-shrK-bad* [*simp*]: *corrKey* = *shrK‘bad*
⟨*proof*⟩

3.12.2 State

The secure channels are star-shaped to/from the server. Therefore, we have only one agent in the relation.

record *m3-state* = *m1-state* +
  IK :: *msg set* — intruder knowledge

Observable state: agent's local state.

type-synonym
  m3-obs = *m2-obs*

definition
  m3-obs :: *m3-state* ⇒ *m3-obs* **where**
  m3-obs s ≡ (| *runs* = *runs s*, *leak* = *leak s* |)

type-synonym
  m3-pred = *m3-state set*

type-synonym
  m3-trans = (*m3-state* × *m3-state*) *set*

3.12.3 Events

Protocol events.

definition — by *A*, refines *m2-step1*
  m3-step1 :: [*rid-t*, *agent*, *agent*, *nonce*] ⇒ *m3-trans*

where
  m3-step1 Ra A B Na ≡ {(*s*, *s1*).

    — guards:
    Ra ∉ *dom* (*runs s*) ∧ — *Ra* is fresh
    Na = *Ra*\$*na* ∧ — generate nonce *Na*

    — actions:
    s1 = *s*⦇
      runs := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [])),
      IK := *insert* ⦃*Agent A*, *Agent B*, *Nonce Na*⦄ (*IK s*) — send msg 1
    ⦈
  }

definition — by *B*, refines *m2-step2*
  m3-step2 :: [*rid-t*, *agent*, *agent*] ⇒ *m3-trans*
where
  m3-step2 Rb A B ≡ {(*s*, *s1*).

    — guards:
    Rb ∉ *dom* (*runs s*) ∧ — *Rb* is fresh

    — actions:
    — create responder thread
    s1 = *s*⦇
      runs := (*runs s*)(*Rb* ↦ (*Resp*, [*A*, *B*], []))
    ⦈
  }

definition — by *Server*, refines *m2-step3*
  m3-step3 :: [*rid-t*, *agent*, *agent*, *nonce*, *key*] ⇒ *m3-trans*
where
  m3-step3 Rs A B Na Kab ≡ {(*s*, *s1*).
    — guards:
    Rs ∉ *dom* (*runs s*) ∧ — fresh server run
    Kab = *sesK* (*Rs*\$*sk*) ∧ — fresh session key

    ⦃*Agent A*, *Agent B*, *Nonce Na*⦄ ∈ *IK s* ∧ — recv msg 1

    — actions:
    — record session key and send messages 2 and 3
    — note that last field in server record is for responder nonce
    s1 = *s*⦇
      runs := (*runs s*)(*Rs* ↦ (*Serv*, [*A*, *B*], [*aNon Na*])),
      IK := *insert*
          (*Crypt* (*shrK A*)
            ⦃*Nonce Na*, *Agent B*, *Key Kab*,
              Crypt (*shrK B*) ⦃*Key Kab*, *Agent A*⦄⦄)
          (*IK s*)
    ⦈
  }

definition — by *A*, refines *m2-step4*

$m3\text{-}step4 :: [rid\text{-}t, agent, agent, nonce, key, msg] \Rightarrow m3\text{-}trans$
where
$m3\text{-}step4\ Ra\ A\ B\ Na\ Kab\ X \equiv \{(s, s1).$

— guards:
$runs\ s\ Ra = Some\ (Init, [A, B], []) \wedge$
$Na = Ra\$na \wedge$

$Crypt\ (shrK\ A)\ \{\!|Nonce\ Na, Agent\ B, Key\ Kab, X|\!\} \in IK\ s \wedge$ — recv msg 2

— actions:
— record session key, and forward X
$s1 = s(\!|$
$runs := (runs\ s)(Ra \mapsto (Init, [A, B], [aKey\ Kab])),$
$IK := insert\ X\ (IK\ s)$
$|\!)$
}

definition — by B, refines $m2\text{-}step5$
$m3\text{-}step5 :: [rid\text{-}t, agent, agent, nonce, key] \Rightarrow m3\text{-}trans$
where
$m3\text{-}step5\ Rb\ A\ B\ Nb\ Kab \equiv \{(s, s1).$

— guards:
$runs\ s\ Rb = Some\ (Resp, [A, B], []) \wedge$
$Nb = Rb\$nb \wedge$

$Crypt\ (shrK\ B)\ \{\!|Key\ Kab, Agent\ A|\!\} \in IK\ s \wedge$ — recv msg 3

— actions:
— record session key
$s1 = s(\!|$
$runs := (runs\ s)(Rb \mapsto (Resp, [A, B], [aKey\ Kab])),$
$IK := insert\ (Crypt\ Kab\ (Nonce\ Nb))\ (IK\ s)$
$|\!)$
}

definition — by A, refines $m2\text{-}step6$
$m3\text{-}step6 :: [rid\text{-}t, agent, agent, nonce, nonce, key] \Rightarrow m3\text{-}trans$
where
$m3\text{-}step6\ Ra\ A\ B\ Na\ Nb\ Kab \equiv \{(s, s').$
$runs\ s\ Ra = Some\ (Init, [A, B], [aKey\ Kab]) \wedge$ — key recv'd before
$Na = Ra\$na \wedge$

$Crypt\ Kab\ (Nonce\ Nb) \in IK\ s \wedge$ — receive $M4$

— actions:
$s' = s(\!|$
$runs := (runs\ s)(Ra \mapsto (Init, [A, B], [aKey\ Kab, aNon\ Nb])),$
$IK := insert\ (Crypt\ Kab\ \{\!|Nonce\ Nb, Nonce\ Nb|\!\})\ (IK\ s)$
$|\!)$
}

definition — by B, refines *m2-step6*
m3-step7 :: [*rid-t*, *agent*, *agent*, *nonce*, *key*] ⇒ *m3-trans*
where
m3-step7 Rb A B Nb Kab ≡ {(*s*, *s'*).
runs s Rb = *Some* (*Resp*, [*A*, *B*], [*aKey Kab*]) ∧ — key recv'd before
Nb = *Rb$nb* ∧

Crypt Kab {|*Nonce Nb*, *Nonce Nb*|} ∈ *IK s* ∧ — receive *M5*

— actions: (redundant) update local state marks successful termination
s' = *s*(|
runs := (*runs s*)(*Rb* ↦ (*Resp*, [*A*, *B*], [*aKey Kab*, *END*]))
|)
}

Session key compromise.

definition — refines *m2-leak*
m3-leak :: [*rid-t*, *rid-t*, *rid-t*, *agent*, *agent*] ⇒ *m3-trans*
where
m3-leak Rs Ra Rb A B ≡ {(*s*, *s1*).
— guards:
runs s Rs = *Some* (*Serv*, [*A*, *B*], [*aNon* (*Ra$na*)]) ∧
runs s Ra = *Some* (*Init*, [*A*, *B*], [*aKey* (*sesK* (*Rs$sk*)), *aNon* (*Rb$nb*)]) ∧
runs s Rb = *Some* (*Resp*, [*A*, *B*], [*aKey* (*sesK* (*Rs$sk*)), *END*]) ∧

— actions:
— record session key as leaked and add it to intruder knowledge
s1 = *s*(| *leak* := *insert* (*sesK* (*Rs$sk*), *Ra$na*, *Rb$nb*) (*leak s*),
IK := *insert* (*Key* (*sesK* (*Rs$sk*))) (*IK s*) |)
}

Intruder fake event.

definition — refines *m2-fake*
m3-DY-fake :: *m3-trans*
where
m3-DY-fake ≡ {(*s*, *s1*).

— actions:
s1 = *s*(|
IK := *synth* (*analz* (*IK s*))
|)
}

3.12.4 Transition system

definition
m3-init :: *m3-state set*
where
m3-init ≡ { (|
runs = *Map.empty*,
leak = *shrK‘bad* × {*undefined*} × {*undefined*},
IK = *Key‘shrK‘bad*
|) }

definition
m3-trans :: (*m3-state* × *m3-state*) *set* **where**
m3-trans ≡ (⋃ *Ra Rb Rs A B Na Nb Kab X*.
m3-step1 Ra A B Na ∪
m3-step2 Rb A B ∪
m3-step3 Rs A B Na Kab ∪
m3-step4 Ra A B Na Kab X ∪
m3-step5 Rb A B Nb Kab ∪
m3-step6 Ra A B Na Nb Kab ∪
m3-step7 Rb A B Nb Kab ∪
m3-leak Rs Ra Rb A B ∪
m3-DY-fake ∪
Id
)

definition
m3 :: (*m3-state*, *m3-obs*) *spec* **where**
m3 ≡ (|
init = *m3-init*,
trans = *m3-trans*,
obs = *m3-obs*
|)

lemmas *m3-defs* =
m3-def m3-init-def m3-trans-def m3-obs-def
m3-step1-def m3-step2-def m3-step3-def m3-step4-def m3-step5-def
m3-step6-def m3-step7-def m3-leak-def m3-DY-fake-def

3.12.5 Invariants

Specialized injection that we can apply more aggressively.

lemmas *analz-Inj-IK* = *analz.Inj* [**where** *H*=*IK s* **for** *s*]
lemmas *parts-Inj-IK* = *parts.Inj* [**where** *H*=*IK s* **for** *s*]

declare *parts-Inj-IK* [*dest*!]

declare *analz-into-parts* [*dest*]

inv1: Secrecy of pre-distributed shared keys

inv1: Secrecy of long-term keys

definition
m3-inv1-lkeysec :: *m3-state set*
where
m3-inv1-lkeysec ≡ {*s*. ∀ *C*.
(*Key* (*shrK C*) ∈ *parts* (*IK s*) ⟶ *C* ∈ *bad*) ∧
(*C* ∈ *bad* ⟶ *Key* (*shrK C*) ∈ *IK s*)
}

lemmas *m3-inv1-lkeysecI* = *m3-inv1-lkeysec-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv1-lkeysecE* [*elim*] = *m3-inv1-lkeysec-def* [*THEN setc-def-to-elim*, *rule-format*]

lemmas *m3-inv1-lkeysecD* = *m3-inv1-lkeysec-def* [*THEN setc-def-to-dest*, *rule-format*]

Invariance proof.

lemma *PO-m3-inv1-lkeysec-init* [*iff*]:
  init m3 ⊆ *m3-inv1-lkeysec*
⟨*proof*⟩

lemma *PO-m3-inv1-lkeysec-trans* [*iff*]:
  {*m3-inv1-lkeysec*} *trans m3* {> *m3-inv1-lkeysec*}
⟨*proof*⟩

lemma *PO-m3-inv1-lkeysec* [*iff*]: *reach m3* ⊆ *m3-inv1-lkeysec*
⟨*proof*⟩

Useful simplifier lemmas

lemma *m3-inv1-lkeysec-for-parts* [*simp*]:
  ⟦ *s* ∈ *m3-inv1-lkeysec* ⟧ ⟹ *Key* (*shrK C*) ∈ *parts* (*IK s*) ⟷ *C* ∈ *bad*
⟨*proof*⟩

lemma *m3-inv1-lkeysec-for-analz* [*simp*]:
  ⟦ *s* ∈ *m3-inv1-lkeysec* ⟧ ⟹ *Key* (*shrK C*) ∈ *analz* (*IK s*) ⟷ *C* ∈ *bad*
⟨*proof*⟩

inv2: Ticket shape for honestly encrypted M2

definition
  m3-inv2-ticket :: *m3-state set*
where
  m3-inv2-ticket ≡ {*s*. ∀ *A B N K X*.
    A ∉ *bad* ⟶
    Crypt (*shrK A*) ⦃*Nonce N*, *Agent B*, *Key K*, *X*⦄ ∈ *parts* (*IK s*) ⟶
      X = *Crypt* (*shrK B*) ⦃*Key K*, *Agent A*⦄ ∧ *K* ∈ *range sesK*
  }

lemmas *m3-inv2-ticketI* =
  m3-inv2-ticket-def [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv2-ticketE* [*elim*] =
  m3-inv2-ticket-def [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv2-ticketD* =
  m3-inv2-ticket-def [*THEN setc-def-to-dest*, *rule-format*, *rotated* −*1*]

Invariance proof.

lemma *PO-m3-inv2-ticket-init* [*iff*]:
  init m3 ⊆ *m3-inv2-ticket*
⟨*proof*⟩

lemma *PO-m3-inv2-ticket-trans* [*iff*]:
  {*m3-inv2-ticket* ∩ *m3-inv1-lkeysec*} *trans m3* {> *m3-inv2-ticket*}
⟨*proof*⟩

lemma *PO-m3-inv2-ticket* [*iff*]: *reach m3* ⊆ *m3-inv2-ticket*
⟨*proof*⟩

inv3: Session keys not used to encrypt other session keys

Session keys are not used to encrypt other keys. Proof requires generalization to sets of session keys.

NOTE: For NSSK, this invariant cannot be inherited from the corresponding L2 invariant. The simulation relation is only an implication not an equivalence.

definition
  m3-inv3-sesK-compr :: *m3-state set*
where
  m3-inv3-sesK-compr ≡ {*s*. ∀ *K KK*.
    KK ⊆ *range sesK* ⟶
    (*Key K* ∈ *analz* (*Key*'*KK* ∪ (*IK s*))) = (*K* ∈ *KK* ∨ *Key K* ∈ *analz* (*IK s*))
  }

lemmas *m3-inv3-sesK-comprI* = *m3-inv3-sesK-compr-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv3-sesK-comprE* = *m3-inv3-sesK-compr-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv3-sesK-comprD* = *m3-inv3-sesK-compr-def* [*THEN setc-def-to-dest*, *rule-format*]

Additional lemma

lemmas *insert-commute-Key* = *insert-commute* [**where** *x*=*Key K* **for** *K*]

lemmas *m3-inv3-sesK-compr-simps* =
  m3-inv3-sesK-comprD
  m3-inv3-sesK-comprD [**where** *KK*={*Kab*} **for** *Kab*, *simplified*]
  m3-inv3-sesK-comprD [**where** *KK*=*insert Kab KK* **for** *Kab KK*, *simplified*]
  insert-commute-Key

Invariance proof.

lemma *PO-m3-inv3-sesK-compr-step4*:
  {*m3-inv3-sesK-compr* ∩ *m3-inv2-ticket* ∩ *m3-inv1-lkeysec*}
    m3-step4 Ra A B Na Kab X
  {> *m3-inv3-sesK-compr*}
⟨*proof*⟩

All together now.

lemmas *PO-m3-inv3-sesK-compr-trans-lemmas* =
  PO-m3-inv3-sesK-compr-step4

lemma *PO-m3-inv3-sesK-compr-init* [*iff*]:
  init m3 ⊆ *m3-inv3-sesK-compr*
⟨*proof*⟩

lemma *PO-m3-inv3-sesK-compr-trans* [*iff*]:
  {*m3-inv3-sesK-compr* ∩ *m3-inv2-ticket* ∩ *m3-inv1-lkeysec*}
    trans m3
  {> *m3-inv3-sesK-compr*}
⟨*proof*⟩

lemma *PO-m3-inv3-sesK-compr* [*iff*]: *reach m3* ⊆ *m3-inv3-sesK-compr*
⟨*proof*⟩

3.12.6 Refinement

Message abstraction and simulation relation

Abstraction function on sets of messages.

inductive-set
  abs-msg :: *msg set* ⇒ *chmsg set*
  for *H* :: *msg set*
where
  am-M1:
    ⦃*Agent A*, *Agent B*, *Nonce Na*⦄ ∈ *H*
  ⟹ *Insec A B* (*Msg* [*aNon Na*]) ∈ *abs-msg H*
| *am-M2*:
    Crypt (*shrK C*) ⦃*Nonce N*, *Agent B*, *Key K*, *X*⦄ ∈ *H*
  ⟹ *Secure Sv C* (*Msg* [*aNon N*, *aAgt B*, *aKey K*]) ∈ *abs-msg H*
| *am-M3*:
    Crypt (*shrK C*) ⦃*Key K*, *Agent A*⦄ ∈ *H*
  ⟹ *Secure Sv C* (*Msg* [*aKey K*, *aAgt A*]) ∈ *abs-msg H*
| *am-M4*:
    Crypt K (*Nonce N*) ∈ *H*
  ⟹ *dAuth K* (*Msg* [*aNon N*]) ∈ *abs-msg H*
| *am-M5*:
    Crypt K ⦃*Nonce N*, *Nonce N'*⦄ ∈ *H*
  ⟹ *dAuth K* (*Msg* [*aNon N*, *aNon N'*]) ∈ *abs-msg H*

R23: The simulation relation. This is a data refinement of the insecure and secure channels of refinement 2.

definition
  R23-msgs :: (*m2-state* × *m3-state*) *set* **where**
  R23-msgs ≡ {(*s*, *t*). *abs-msg* (*parts* (*IK t*)) ⊆ *chan s*}

definition
  R23-keys :: (*m2-state* × *m3-state*) *set* **where** — only an implication!
  R23-keys ≡ {(*s*, *t*). ∀ *KK K*. *KK* ⊆ *range sesK* ⟶
    Key K ∈ *analz* (*Key*‘*KK* ∪ *IK t*) ⟶ *aKey K* ∈ *extr* (*aKey*‘*KK* ∪ *ik0*) (*chan s*)
  }

definition
  R23-non :: (*m2-state* × *m3-state*) *set* **where** — only an implication!
  R23-non ≡ {(*s*, *t*). ∀ *KK N*. *KK* ⊆ *range sesK* ⟶
    Nonce N ∈ *analz* (*Key*‘*KK* ∪ *IK t*) ⟶ *aNon N* ∈ *extr* (*aKey*‘*KK* ∪ *ik0*) (*chan s*)
  }

definition
  R23-pres :: (*m2-state* × *m3-state*) *set* **where**
  R23-pres ≡ {(*s*, *t*). *runs s* = *runs t* ∧ *leak s* = *leak t*}

definition
  R23 :: (*m2-state* × *m3-state*) *set* **where**
  R23 ≡ *R23-msgs* ∩ *R23-keys* ∩ *R23-non* ∩ *R23-pres*

lemmas *R23-defs* =
  R23-def R23-msgs-def R23-keys-def R23-non-def R23-pres-def

The mediator function is the identity here.

definition
med32 :: *m3-obs* ⇒ *m2-obs* **where**
med32 ≡ *id*

lemmas *R23-msgsI* = *R23-msgs-def* [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-msgsE* [*elim*] = *R23-msgs-def* [*THEN rel-def-to-elim*, *simplified*, *rule-format*]
lemmas *R23-msgsE'* [*elim*] =
R23-msgs-def [*THEN rel-def-to-dest*, *simplified*, *rule-format*, *THEN subsetD*]

lemmas *R23-keysI* = *R23-keys-def* [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-keysE* [*elim*] = *R23-keys-def* [*THEN rel-def-to-elim*, *simplified*, *rule-format*]
lemmas *R23-keysD* = *R23-keys-def* [*THEN rel-def-to-dest*, *simplified*, *rule-format*, *rotated 2*]

lemmas *R23-nonI* = *R23-non-def* [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-nonE* [*elim*] = *R23-non-def* [*THEN rel-def-to-elim*, *simplified*, *rule-format*]
lemmas *R23-nonD* = *R23-non-def* [*THEN rel-def-to-dest*, *simplified*, *rule-format*, *rotated 2*]

lemmas *R23-presI* = *R23-pres-def* [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-presE* [*elim*] = *R23-pres-def* [*THEN rel-def-to-elim*, *simplified*, *rule-format*]

lemmas *R23-intros* = *R23-msgsI R23-keysI R23-nonI R23-presI*

Further lemmas: general lemma for simplifier and different instantiations.

lemmas *R23-keys-dests* =
R23-keysD
R23-keysD [**where** *KK*={}, *simplified*]
R23-keysD [**where** *KK*={*K*} **for** *K*, *simplified*]
R23-keysD [**where** *KK*=*insert K KK* **for** *K KK*, *simplified*, *OF - - conjI*]

lemmas *R23-non-dests* =
R23-nonD
R23-nonD [**where** *KK*={}, *simplified*]
R23-nonD [**where** *KK*={*K*} **for** *K*, *simplified*]
R23-nonD [**where** *KK*=*insert K KK* **for** *K KK*, *simplified*, *OF - - conjI*]

lemmas *R23-dests* = *R23-keys-dests R23-non-dests*

General lemmas

General facts about *abs-msg*

declare *abs-msg.intros* [*intro*!]
declare *abs-msg.cases* [*elim*!]

lemma *abs-msg-empty*: *abs-msg* {} = {}
⟨*proof*⟩

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

lemma *abs-msg-mono* [*elim*]:
  ⟦ $m \in$ *abs-msg G*; $G \subseteq H$ ⟧ $\Longrightarrow m \in$ *abs-msg H*
⟨*proof*⟩

lemma *abs-msg-insert-mono* [*intro*]:
  ⟦ $m \in$ *abs-msg H* ⟧ $\Longrightarrow m \in$ *abs-msg* (*insert* m' *H*)
⟨*proof*⟩

Facts about *abs-msg* concerning abstraction of fakeable messages. This is crucial for proving the refinement of the intruder event.

lemma *abs-msg-DY-subset-fakeable*:
  ⟦ $(s, t) \in$ *R23-msgs*; $(s, t) \in$ *R23-keys*; $(s, t) \in$ *R23-non*; $t \in$ *m3-inv1-lkeysec* ⟧
  $\Longrightarrow$ *abs-msg* (*synth* (*analz* (*IK t*))) $\subseteq$ *fake ik0* (*dom* (*runs s*)) (*chan s*)
⟨*proof*⟩

Refinement proof

Pair decomposition. These were set to `elim!`, which is too agressive here.

declare *MPair-analz* [*rule del*, *elim*]
declare *MPair-parts* [*rule del*, *elim*]

Protocol events.

lemma *PO-m3-step1-refines-m2-step1*:
  {*R23*}
    (*m2-step1 Ra A B Na*), (*m3-step1 Ra A B Na*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-step2-refines-m2-step2*:
  {*R23*}
    (*m2-step2 Rb A B*), (*m3-step2 Rb A B*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-step3-refines-m2-step3*:
  {*R23* ∩ (*m2-inv3a-sesK-compr*) × (*m3-inv3-sesK-compr* ∩ *m3-inv1-lkeysec*)}
    (*m2-step3 Rs A B Na Kab*), (*m3-step3 Rs A B Na Kab*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-step4-refines-m2-step4*:
  {*R23* ∩ (*m2-inv3b-sesK-compr-non*)
      × (*m3-inv3-sesK-compr* ∩ *m3-inv2-ticket* ∩ *m3-inv1-lkeysec*)}
    (*m2-step4 Ra A B Na Kab*), (*m3-step4 Ra A B Na Kab X*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-step5-refines-m2-step5*:
  {*R23*}

(*m2-step5 Rb A B Nb Kab*), (*m3-step5 Rb A B Nb Kab*)
{> *R23*}
⟨*proof*⟩

lemma *PO-m3-step6-refines-m2-step6*:
{*R23*}
(*m2-step6 Ra A B Na Nb Kab*), (*m3-step6 Ra A B Na Nb Kab*)
{> *R23*}
⟨*proof*⟩

lemma *PO-m3-step7-refines-m2-step7*:
{*R23*}
(*m2-step7 Rb A B Nb Kab*), (*m3-step7 Rb A B Nb Kab*)
{> *R23*}
⟨*proof*⟩

Intruder events.

lemma *PO-m3-leak-refines-m2-leak*:
{*R23*}
m2-leak Rs Ra Rb A B, *m3-leak Rs Ra Rb A B*
{> *R23*}
⟨*proof*⟩

lemma *PO-m3-DY-fake-refines-m2-fake*:
{*R23* ∩ *UNIV* × *m3-inv1-lkeysec*}
m2-fake, *m3-DY-fake*
{> *R23*}
⟨*proof*⟩

All together now...

lemmas *PO-m3-trans-refines-m2-trans* =
PO-m3-step1-refines-m2-step1 PO-m3-step2-refines-m2-step2
PO-m3-step3-refines-m2-step3 PO-m3-step4-refines-m2-step4
PO-m3-step5-refines-m2-step5 PO-m3-step6-refines-m2-step6
PO-m3-step7-refines-m2-step7 PO-m3-leak-refines-m2-leak
PO-m3-DY-fake-refines-m2-fake

lemma *PO-m3-refines-init-m2* [*iff*]:
init m3 ⊆ *R23*‘‘(*init m2*)
⟨*proof*⟩

lemma *PO-m3-refines-trans-m2* [*iff*]:
{*R23* ∩ (*m2-inv3a-sesK-compr* ∩ *m2-inv3b-sesK-compr-non*)
× (*m3-inv3-sesK-compr* ∩ *m3-inv2-ticket* ∩ *m3-inv1-lkeysec*)}
(*trans m2*), (*trans m3*)
{> *R23*}
⟨*proof*⟩

lemma *PO-m3-observation-consistent* [*iff*]:
obs-consistent R23 med32 m2 m3
⟨*proof*⟩

Refinement result.

lemma *m3-refines-m2* [*iff*]:
refines
($R23 \cap$ (*m2-inv3a-sesK-compr* $\cap$ *m2-inv3b-sesK-compr-non*)
$\times$ (*m3-inv3-sesK-compr* $\cap$ *m3-inv2-ticket* $\cap$ *m3-inv1-lkeysec*))
med32 m2 m3
⟨*proof*⟩

lemma *m3-implements-m2* [*iff*]:
implements med32 m2 m3
⟨*proof*⟩

3.12.7 Inherited invariants

inv4 (derived): Key secrecy for initiator

definition
m3-inv4-ikk-init :: *m3-state set*
where
m3-inv4-ikk-init $\equiv \{s.\ \forall\, Ra\ K\ A\ B\ al.$
$runs\ s\ Ra = Some\ (Init,\ [A,\ B],\ aKey\ K\ \#\ al) \longrightarrow A \in good \longrightarrow B \in good \longrightarrow$
$Key\ K \in analz\ (IK\ s) \longrightarrow$
$(\exists\, Nb'.\ (K,\ Ra\ \$\ na,\ Nb') \in leak\ s)$
$\}$

lemmas *m3-inv4-ikk-initI* = *m3-inv4-ikk-init-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv4-ikk-initE* [*elim*] = *m3-inv4-ikk-init-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv4-ikk-initD* = *m3-inv4-ikk-init-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

lemma *PO-m3-inv4-ikk-init*: *reach m3* $\subseteq$ *m3-inv4-ikk-init*
⟨*proof*⟩

inv5 (derived): Key secrecy for responder

definition
m3-inv5-ikk-resp :: *m3-state set*
where
m3-inv5-ikk-resp $\equiv \{s.\ \forall\, Rb\ K\ A\ B\ al.$
$runs\ s\ Rb = Some\ (Resp,\ [A,\ B],\ aKey\ K\ \#\ al) \longrightarrow A \in good \longrightarrow B \in good \longrightarrow$
$Key\ K \in analz\ (IK\ s) \longrightarrow$
$K \in Domain\ (leak\ s)$
$\}$

lemmas *m3-inv5-ikk-respI* = *m3-inv5-ikk-resp-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv5-ikk-respE* [*elim*] = *m3-inv5-ikk-resp-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv5-ikk-respD* = *m3-inv5-ikk-resp-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

lemma *PO-m3-inv4-ikk-resp*: *reach m3* $\subseteq$ *m3-inv5-ikk-resp*
⟨*proof*⟩

end

3.13 Abstract Denning-Sacco protocol (L1)

theory *m1-ds* **imports** *m1-keydist-inrn ../Refinement/a0n-agree*
begin

We augment the basic abstract key distribution model such that the server sends a timestamp along with the session key. We check the timestamp's validity to ensure recentness of the session key.

We establish one refinement for this model, namely that this model refines the basic authenticated key transport model *m1-keydist-inrn*, which guarantees non-injective agreement with the server on the session key and the server-generated timestamp.

3.13.1 State

We extend the basic key distribution by adding timestamps. We add a clock variable modeling the current time. The frames, runs, and observations remain the same as in the previous model, but we will use the *nat list*'s to store timestamps.

type-synonym
time = *nat* — for clock and timestamps

consts
Ls :: *time* — life time for session keys

State and observations

record
m1-state = *m1x-state* +
clk :: *time*

type-synonym
m1-obs = *m1-state*

type-synonym
'x m1-pred = *'x m1-state-scheme set*

type-synonym
'x m1-trans = (*'x m1-state-scheme* × *'x m1-state-scheme*) *set*

Instantiate parameters regarding list of freshness identifiers stored at server.

overloading *is-len'* ≡ *is-len* *rs-len'* ≡ *rs-len* **begin**
definition *is-len-def* [*simp*]: *is-len'* ≡ *1*::*nat*
definition *rs-len-def* [*simp*]: *rs-len'* ≡ *1*::*nat*
end

3.13.2 Events

definition — by *A*, refines *m1x-step1*
m1-step1 :: [*rid-t*, *agent*, *agent*] ⇒ *'x m1-trans*
where
m1-step1 ≡ *m1a-step1*

definition — by *B*, refines *m1x-step2*

$m1\text{-}step2 :: [rid\text{-}t, agent, agent] \Rightarrow {'x}\ m1\text{-}trans$
where
$m1\text{-}step2 \equiv m1a\text{-}step2$

definition — by *Sv*, refines *m1x-step3*
$m1\text{-}step3 :: [rid\text{-}t, agent, agent, key, time] \Rightarrow {'x}\ m1\text{-}trans$
where
$m1\text{-}step3\ Rs\ A\ B\ Kab\ Ts \equiv \{(s, s').$
— new guards:
$Ts = clk\ s\ \wedge$ — fresh timestamp

— rest as before:
$(s, s') \in m1a\text{-}step3\ Rs\ A\ B\ Kab\ [aNum\ Ts]$
$\}$

definition — by *A*, refines *m1x-step5*
$m1\text{-}step4 :: [rid\text{-}t, agent, agent, key, time] \Rightarrow {'x}\ m1\text{-}trans$
where
$m1\text{-}step4\ Ra\ A\ B\ Kab\ Ts \equiv \{(s, s').$
— new guards:
$clk\ s < Ts + Ls\ \wedge$ — ensure session key recentness

— rest as before
$(s, s') \in m1a\text{-}step4\ Ra\ A\ B\ Kab\ [aNum\ Ts]$
$\}$

definition — by *B*, refines *m1x-step4*
$m1\text{-}step5 :: [rid\text{-}t, agent, agent, key, time] \Rightarrow {'x}\ m1\text{-}trans$
where
$m1\text{-}step5\ Rb\ A\ B\ Kab\ Ts \equiv \{(s, s').$
— new guards:
— ensure freshness of session key
$clk\ s < Ts + Ls\ \wedge$

— rest as before
$(s, s') \in m1a\text{-}step5\ Rb\ A\ B\ Kab\ [aNum\ Ts]$
$\}$

definition — refines *skip*
$m1\text{-}tick :: time \Rightarrow {'x}\ m1\text{-}trans$
where
$m1\text{-}tick\ T \equiv \{(s, s').$
$s' = s(\!|\ clk := clk\ s + T\ |\!)$
$\}$

definition — by attacker, refines *m1x-leak*
$m1\text{-}leak :: [rid\text{-}t] \Rightarrow {'x}\ m1\text{-}trans$
where
$m1\text{-}leak \equiv m1a\text{-}leak$

3.13.3 Specification

definition

m1-init :: *unit m1-pred*
where
m1-init ≡ { ⦇ *runs* = *Map.empty*, *leak* = *corrKey*, *clk* = *0* ⦈ }

definition
m1-trans :: *'x m1-trans* **where**
m1-trans ≡ (⋃ *A B Ra Rb Rs Kab Ts T*.
m1-step1 Ra A B ∪
m1-step2 Rb A B ∪
m1-step3 Rs A B Kab Ts ∪
m1-step4 Ra A B Kab Ts ∪
m1-step5 Rb A B Kab Ts ∪
m1-tick T ∪
m1-leak Rs ∪
Id
)

definition
m1 :: (*m1-state*, *m1-obs*) *spec* **where**
m1 ≡ ⦇
init = *m1-init*,
trans = *m1-trans*,
obs = *id*
⦈

lemmas *m1-loc-defs* =
m1-def m1-init-def m1-trans-def
m1-step1-def m1-step2-def m1-step3-def m1-step4-def m1-step5-def
m1-leak-def m1-tick-def

lemmas *m1-defs* = *m1-loc-defs m1a-defs*

lemma *m1-obs-id* [*simp*]: *obs m1* = *id*
⟨*proof*⟩

3.13.4 Invariants

inv0: Finite domain

There are only finitely many runs. This is needed to establish the responder/initiator agreement.

definition
m1-inv0-fin :: *'x m1-pred*
where
m1-inv0-fin ≡ {*s*. *finite* (*dom* (*runs s*))}

lemmas *m1-inv0-finI* = *m1-inv0-fin-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m1-inv0-finE* [*elim*] = *m1-inv0-fin-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m1-inv0-finD* = *m1-inv0-fin-def* [*THEN setc-def-to-dest*, *rule-format*]

Invariance proofs.

lemma *PO-m1-inv0-fin-init* [*iff*]:

init m1 ⊆ *m1-inv0-fin*
⟨*proof*⟩

lemma *PO-m1-inv0-fin-trans* [*iff*]:
{*m1-inv0-fin*} *trans m1* {> *m1-inv0-fin*}
⟨*proof*⟩

lemma *PO-m1-inv0-fin* [*iff*]: *reach m1* ⊆ *m1-inv0-fin*
⟨*proof*⟩

3.13.5 Refinement of *m1a*

Simulation relation

R1a1: The simulation relation and mediator function are identities.

definition
med1a1 :: *m1-obs* ⇒ *m1a-obs* **where**
med1a1 t ≡ (| *runs* = *runs t*, *leak* = *leak t* |)

definition
R1a1 :: (*m1a-state* × *m1-state*) *set* **where**
R1a1 ≡ {(*s*, *t*). *s* = *med1a1 t*}

lemmas *R1a1-defs* = *R1a1-def med1a1-def*

Refinement proof

lemma *PO-m1-step1-refines-m1a-step1*:
{*R1a1*}
(*m1a-step1 Ra A B*), (*m1-step1 Ra A B*)
{> *R1a1*}
⟨*proof*⟩

lemma *PO-m1-step2-refines-m1a-step2*:
{*R1a1*}
(*m1a-step2 Rb A B*), (*m1-step2 Rb A B*)
{> *R1a1*}
⟨*proof*⟩

lemma *PO-m1-step3-refines-m1a-step3*:
{*R1a1*}
(*m1a-step3 Rs A B Kab* [*aNum Ts*]), (*m1-step3 Rs A B Kab Ts*)
{> *R1a1*}
⟨*proof*⟩

lemma *PO-m1-step4-refines-m1a-step4*:
{*R1a1*}
(*m1a-step4 Ra A B Kab* [*aNum Ts*]), (*m1-step4 Ra A B Kab Ts*)
{> *R1a1*}
⟨*proof*⟩

lemma *PO-m1-step5-refines-m1a-step5*:
{*R1a1*}

(*m1a-step5 Rb A B Kab* [*aNum Ts*]), (*m1-step5 Rb A B Kab Ts*)
{> *R1a1*}
⟨*proof*⟩

lemma *PO-m1-leak-refines-m1a-leak*:
{*R1a1*}
(*m1a-leak Rs*), (*m1-leak Rs*)
{> *R1a1*}
⟨*proof*⟩

lemma *PO-m1-tick-refines-m1a-skip*:
{*R1a1*}
Id, (*m1-tick T*)
{> *R1a1*}
⟨*proof*⟩

All together now...

lemmas *PO-m1-trans-refines-m1a-trans* =
PO-m1-step1-refines-m1a-step1 PO-m1-step2-refines-m1a-step2
PO-m1-step3-refines-m1a-step3 PO-m1-step4-refines-m1a-step4
PO-m1-step5-refines-m1a-step5 PO-m1-leak-refines-m1a-leak
PO-m1-tick-refines-m1a-skip

lemma *PO-m1-refines-init-m1a* [*iff*]:
init m1 ⊆ *R1a1*‘‘(*init m1a*)
⟨*proof*⟩

lemma *PO-m1-refines-trans-m1a* [*iff*]:
{*R1a1*}
(*trans m1a*), (*trans m1*)
{> *R1a1*}
⟨*proof*⟩

Observation consistency.

lemma *obs-consistent-med1a1* [*iff*]:
obs-consistent R1a1 med1a1 m1a m1
⟨*proof*⟩

Refinement result.

lemma *PO-m1-refines-m1a* [*iff*]:
refines R1a1 med1a1 m1a m1
⟨*proof*⟩

lemma *m1-implements-m1*: *implements med1a1 m1a m1*
⟨*proof*⟩

end

3.14 Abstract Denning-Sacco protocol (L2)

theory *m2-ds* **imports** *m1-ds ../Refinement/Channels*

begin

We model an abstract version of the Denning-Sacco protocol:

$$\begin{array}{lll} \text{M1.} & A \rightarrow S: & A, B \\ \text{M2.} & S \rightarrow A: & \{B, Kab, T, \{Kab, A, T\}_{Kbs}\}_{Kas} \\ \text{M3.} & A \rightarrow B: & \{Kab, A, T\}_{Kbs} \end{array}$$

This refinement introduces channels with security properties. We model a parallel version of the DS protocol:

$$\begin{array}{lll} \text{M1.} & A \rightarrow S: & A, B \\ \text{M2a.} & S \rightarrow A: & \{B, Kab, T\}_{Kas} \\ \text{M2b.} & S \rightarrow B: & \{Kab, A, T\}_{Kbs} \end{array}$$

Message 1 is sent over an insecure channel, the other two message over secure channels.

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

3.14.1 State

State and observations

record *m2-state* = *m1-state* +
  chan :: *chmsg set* — channel messages

type-synonym
  m2-obs = *m1-state*

definition
  m2-obs :: *m2-state* ⇒ *m2-obs* **where**
  m2-obs s ≡ (|
    runs = *runs s*,
    leak = *leak s*,
    clk = *clk s*
  |)

type-synonym
  m2-pred = *m2-state set*

type-synonym
  m2-trans = (*m2-state* × *m2-state*) *set*

3.14.2 Events

Protocol events.

definition — by *A*, refines *m1a-step1*
  m2-step1 :: [*rid-t*, *agent*, *agent*] ⇒ *m2-trans*
where
  m2-step1 Ra A B ≡ {(*s*, *s1*).

    — guards:
    Ra ∉ *dom* (*runs s*) ∧ — *Ra* is fresh

— actions:
— create initiator thread and send message 1
s1 = *s*(|
runs := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [])),
chan := *insert* (*Insec A B* (*Msg* [])) (*chan s*)   — send M1
|)
}

definition — by *B*, refines *m1e-step2*
m2-step2 :: [*rid-t*, *agent*, *agent*] ⇒ *m2-trans*
where
m2-step2 ≡ *m1-step2*

definition — by *Server*, refines *m1e-step3*
m2-step3 :: [*rid-t*, *agent*, *agent*, *key*, *time*] ⇒ *m2-trans*
where
m2-step3 Rs A B Kab Ts ≡ {(*s*, *s1*).

— guards:
Rs ∉ *dom* (*runs s*) ∧   — fresh server run
Kab = *sesK* (*Rs*$*sk*) ∧   — fresh session key
Ts = *clk s* ∧   — fresh timestamp

Insec A B (*Msg* []) ∈ *chan s* ∧   — recv M1

— actions:
— record key and send messages 2 and 3
s1 = *s*(|
runs := (*runs s*)(*Rs* ↦ (*Serv*, [*A*, *B*], [*aNum Ts*])),
chan := {*Secure Sv A* (*Msg* [*aAgt B*, *aKey Kab*, *aNum Ts*]),   — send *M2a*/*b*
Secure Sv B (*Msg* [*aKey Kab*, *aAgt A*, *aNum Ts*])} ∪ *chan s*
|)
}

definition — by *A*, refines *m1e-step4*
m2-step4 :: [*rid-t*, *agent*, *agent*, *key*, *time*] ⇒ *m2-trans*
where
m2-step4 Ra A B Kab Ts ≡ {(*s*, *s1*).

— guards:
runs s Ra = *Some* (*Init*, [*A*, *B*], []) ∧
Secure Sv A (*Msg* [*aAgt B*, *aKey Kab*, *aNum Ts*]) ∈ *chan s* ∧ — recv *M2a*

clk s < *Ts* + *Ls* ∧   — ensure key freshness

— actions:
— record session key
s1 = *s*(|
runs := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [*aKey Kab*, *aNum Ts*]))
|)
}

definition — by *B*, refines *m1e-step5*

m2-step5 :: [*rid-t*, *agent*, *agent*, *key*, *time*] ⇒ *m2-trans*
where
m2-step5 Rb A B Kab Ts ≡ {(*s*, *s1*).

— guards:
runs s Rb = *Some* (*Resp*, [*A*, *B*], []) ∧
Secure Sv B (*Msg* [*aKey Kab*, *aAgt A*, *aNum Ts*]) ∈ *chan s* ∧ — recv *M2b*

— ensure freshness of session key
clk s < *Ts* + *Ls* ∧

— actions:
— record session key
s1 = *s*(|
runs := (*runs s*)(*Rb* ↦ (*Resp*, [*A*, *B*], [*aKey Kab*, *aNum Ts*]))
|)
}

Clock tick event

definition — refines *m1-tick*
m2-tick :: *time* ⇒ *m2-trans*
where
m2-tick ≡ *m1-tick*

Session key compromise.

definition — refines *m1-leak*
m2-leak :: *rid-t* ⇒ *m2-trans*
where
m2-leak Rs ≡ {(*s*, *s1*).
— guards:
Rs ∈ *dom* (*runs s*) ∧
fst (*the* (*runs s Rs*)) = *Serv* ∧ — compromise server run *Rs*

— actions:
— record session key as leaked;
— intruder sends himself an insecure channel message containing the key
s1 = *s*(| *leak* := *insert* (*sesK* (*Rs*$*sk*)) (*leak s*),
chan := *insert* (*Insec undefined undefined* (*Msg* [*aKey* (*sesK* (*Rs*$*sk*))])) (*chan s*) |)
}

Intruder fake event (new).

definition — refines *Id*
m2-fake :: *m2-trans*
where
m2-fake ≡ {(*s*, *s1*).

— actions:
s1 = *s*(|
— close under fakeable messages
chan := *fake ik0* (*dom* (*runs s*)) (*chan s*)
|)
}

3.14.3 Transition system

definition
  m2-init :: *m2-pred*
where
  m2-init ≡ { ⦇
    runs = *Map.empty*,
    leak = *corrKey*,
    clk = *0*,
    chan = {}   — *Channels.ik0* contains *aKey‘corrKey*
  ⦈ }

definition
  m2-trans :: *m2-trans* **where**
  m2-trans ≡ (⋃ *A B Ra Rb Rs Kab Ts T*.
    m2-step1 Ra A B ∪
    m2-step2 Rb A B ∪
    m2-step3 Rs A B Kab Ts ∪
    m2-step4 Ra A B Kab Ts ∪
    m2-step5 Rb A B Kab Ts ∪
    m2-tick T ∪
    m2-leak Rs ∪
    m2-fake ∪
    Id
)

definition
  m2 :: (*m2-state*, *m2-obs*) *spec* **where**
  m2 ≡ ⦇
   init = *m2-init*,
   trans = *m2-trans*,
   obs = *m2-obs*
  ⦈

lemmas *m2-loc-defs* =
  m2-def m2-init-def m2-trans-def m2-obs-def
  m2-step1-def m2-step2-def m2-step3-def m2-step4-def m2-step5-def
  m2-tick-def m2-leak-def m2-fake-def

lemmas *m2-defs* = *m2-loc-defs m1-defs*

3.14.4 Invariants and simulation relation

inv3a: Session key compromise

A L2 version of a session key comprise invariant. Roughly, it states that adding a set of keys *KK* to the parameter *T* of *extr* does not help the intruder to extract keys other than those in *KK* or extractable without adding *KK*.

definition
  m2-inv3a-sesK-compr :: *m2-state set*
where
  m2-inv3a-sesK-compr ≡ {*s*. ∀ *K KK*.
    aKey K ∈ *extr* (*aKey‘KK* ∪ *ik0*) (*chan s*) ⟷ (*K* ∈ *KK* ∨ *aKey K* ∈ *extr ik0* (*chan s*))

}

lemmas *m2-inv3a-sesK-comprI* =
m2-inv3a-sesK-compr-def [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv3a-sesK-comprE* [*elim*] =
m2-inv3a-sesK-compr-def [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv3a-sesK-comprD* =
m2-inv3a-sesK-compr-def [*THEN setc-def-to-dest*, *rule-format*]

Additional lemma

lemmas *insert-commute-aKey* = *insert-commute* [**where** *x=aKey K* **for** *K*]

lemmas *m2-inv3a-sesK-compr-simps* =
m2-inv3a-sesK-comprD
m2-inv3a-sesK-comprD [**where** *KK=insert Kab KK* **for** *Kab KK*, *simplified*]
m2-inv3a-sesK-comprD [**where** *KK*={*Kab*} **for** *Kab*, *simplified*]
insert-commute-aKey — to get the keys to the front

lemma *PO-m2-inv3a-sesK-compr-init* [*iff*]:
init m2 ⊆ *m2-inv3a-sesK-compr*
⟨*proof*⟩

lemma *PO-m2-inv3a-sesK-compr-trans* [*iff*]:
{*m2-inv3a-sesK-compr*} *trans m2* {> *m2-inv3a-sesK-compr*}
⟨*proof*⟩

lemma *PO-m2-inv3a-sesK-compr* [*iff*]: *reach m2* ⊆ *m2-inv3a-sesK-compr*
⟨*proof*⟩

inv3: Extracted session keys

inv3: Extracted non-leaked session keys were generated by the server for at least one bad agent. This invariant is needed in the proof of the strengthening of the authorization guards in steps 4 and 5 (e.g., (*Kab*, *A*) ∈ *azC* (*runs s*) for the initiator's step4).

definition
m2-inv3-extrKey :: *m2-state set*
where
m2-inv3-extrKey ≡ {*s*. ∀ *K*.
aKey K ∈ *extr ik0* (*chan s*) ⟶ *K* ∉ *leak s* ⟶ — was: *K* ∉ *corrKey* ⟶
(∃ *R A′ B′ Ts′*. *K* = *sesK* (*R$sk*) ∧
runs s R = *Some* (*Serv*, [*A′*, *B′*], [*aNum Ts′*]) ∧
(*A′* ∈ *bad* ∨ *B′* ∈ *bad*))
}

lemmas *m2-inv3-extrKeyI* =
m2-inv3-extrKey-def [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv3-extrKeyE* [*elim*] =
m2-inv3-extrKey-def [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv3-extrKeyD* =
m2-inv3-extrKey-def [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

lemma *PO-m2-inv3-extrKey-init* [*iff*]:

init m2 ⊆ *m2-inv3-extrKey*
⟨*proof*⟩

lemma *PO-m2-inv3-extrKey-trans* [*iff*]:
{*m2-inv3-extrKey* ∩ *m2-inv3a-sesK-compr*}
trans m2
{> *m2-inv3-extrKey*}
⟨*proof*⟩

lemma *PO-m2-inv3-extrKey* [*iff*]: *reach m2* ⊆ *m2-inv3-extrKey*
⟨*proof*⟩

inv4: Messages M2a/M2b for good agents and server state

inv4: Secure messages to honest agents and server state; one variant for each of M2a and M2b. These invariants establish guard strengthening for server authentication by the initiator and the responder.

definition
m2-inv4-M2a :: *m2-state set*
where
m2-inv4-M2a ≡ {*s*. ∀ *A B Kab Ts*.
Secure Sv A (*Msg* [*aAgt B*, *aKey Kab*, *aNum Ts*]) ∈ *chan s* ⟶ *A* ∈ *good* ⟶
(∃ *Rs*. *Kab* = *sesK* (*Rs*$*sk*) ∧
runs s Rs = *Some* (*Serv*, [*A*, *B*], [*aNum Ts*]))
}

definition
m2-inv4-M2b :: *m2-state set*
where
m2-inv4-M2b ≡ {*s*. ∀ *A B Kab Ts*.
Secure Sv B (*Msg* [*aKey Kab*, *aAgt A*, *aNum Ts*]) ∈ *chan s* ⟶ *B* ∈ *good* ⟶
(∃ *Rs*. *Kab* = *sesK* (*Rs*$*sk*) ∧
runs s Rs = *Some* (*Serv*, [*A*, *B*], [*aNum Ts*]))
}

lemmas *m2-inv4-M2aI* =
m2-inv4-M2a-def [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv4-M2aE* [*elim*] =
m2-inv4-M2a-def [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv4-M2aD* =
m2-inv4-M2a-def [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

lemmas *m2-inv4-M2bI* = *m2-inv4-M2b-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv4-M2bE* [*elim*] =
m2-inv4-M2b-def [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv4-M2bD* =
m2-inv4-M2b-def [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

Invariance proofs.

lemma *PO-m2-inv4-M2a-init* [*iff*]:
init m2 ⊆ *m2-inv4-M2a*

⟨*proof*⟩

lemma *PO-m2-inv4-M2a-trans* [*iff*]:
{*m2-inv4-M2a*} *trans m2* {> *m2-inv4-M2a*}
⟨*proof*⟩

lemma *PO-m2-inv4-M2a* [*iff*]: *reach m2* ⊆ *m2-inv4-M2a*
⟨*proof*⟩

lemma *PO-m2-inv4-M2b-init* [*iff*]:
init m2 ⊆ *m2-inv4-M2b*
⟨*proof*⟩

lemma *PO-m2-inv4-M2b-trans* [*iff*]:
{*m2-inv4-M2b*} *trans m2* {> *m2-inv4-M2b*}
⟨*proof*⟩

lemma *PO-m2-inv4-M2b* [*iff*]: *reach m2* ⊆ *m2-inv4-M2b*
⟨*proof*⟩

Consequence needed in proof of inv8/step5 and inv9/step4: The session key uniquely identifies other fields in M2a and M2b, provided it is secret.

lemma *m2-inv4-M2a-M2b-match*:
⟦ *Secure Sv A′* (*Msg* [*aAgt B′*, *aKey Kab*, *aNum Ts′*]) ∈ *chan s*;
Secure Sv B (*Msg* [*aKey Kab*, *aAgt A*, *aNum Ts*]) ∈ *chan s*;
aKey Kab ∉ *extr ik0* (*chan s*); *s* ∈ *m2-inv4-M2a*; *s* ∈ *m2-inv4-M2b* ⟧
⟹ *A* = *A′* ∧ *B* = *B′* ∧ *Ts* = *Ts′*
⟨*proof*⟩

More consequences of invariants. Needed in ref/step4 and ref/step5 respectively to show the strengthening of the authorization guards.

lemma *m2-inv34-M2a-authorized*:
assumes *Secure Sv A* (*Msg* [*aAgt B*, *aKey K*, *aNum T*]) ∈ *chan s*
s ∈ *m2-inv3-extrKey s* ∈ *m2-inv4-M2a K* ∉ *leak s*
shows (*K*, *A*) ∈ *azC* (*runs s*)
⟨*proof*⟩

lemma *m2-inv34-M2b-authorized*:
assumes *Secure Sv B* (*Msg* [*aKey K*, *aAgt A*, *aNum T*]) ∈ *chan s*
s ∈ *m2-inv3-extrKey s* ∈ *m2-inv4-M2b K* ∉ *leak s*
shows (*K*, *B*) ∈ *azC* (*runs s*)
⟨*proof*⟩

inv5: Key secrecy for server

inv5: Key secrecy from server perspective. This invariant links the abstract notion of key secrecy to the intruder key knowledge.

definition
m2-inv5-ikk-sv :: *m2-state set*
where

m2-inv5-ikk-sv ≡ {*s*. ∀ *R A B al*.
runs s R = *Some* (*Serv*, [*A*, *B*], *al*) ⟶ *A* ∈ *good* ⟶ *B* ∈ *good* ⟶
aKey (*sesK* (*R$sk*)) ∈ *extr ik0* (*chan s*) ⟶
sesK (*R$sk*) ∈ *leak s*
}

lemmas *m2-inv5-ikk-svI* =
m2-inv5-ikk-sv-def [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv5-ikk-svE* [*elim*] =
m2-inv5-ikk-sv-def [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv5-ikk-svD* =
m2-inv5-ikk-sv-def [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

Invariance proof.

lemma *PO-m2-inv5-ikk-sv-init* [*iff*]:
init m2 ⊆ *m2-inv5-ikk-sv*
⟨*proof*⟩

lemma *PO-m2-inv5-ikk-sv-trans* [*iff*]:
{*m2-inv5-ikk-sv* ∩ *m2-inv3a-sesK-compr* ∩ *m2-inv3-extrKey*}
trans m2
{> *m2-inv5-ikk-sv*}
⟨*proof*⟩

lemma *PO-m2-inv5-ikk-sv* [*iff*]: *reach m2* ⊆ *m2-inv5-ikk-sv*
⟨*proof*⟩

inv6/7: Key secrecy for initiator and responder

These invariants are derivable.

definition
m2-inv6-ikk-init :: *m2-state set*
where
m2-inv6-ikk-init ≡ {*s*. ∀ *A B Ra K Ts nl*.
runs s Ra = *Some* (*Init*, [*A*, *B*], *aKey K* # *aNum Ts* # *nl*) ⟶
A ∈ *good* ⟶ *B* ∈ *good* ⟶ *aKey K* ∈ *extr ik0* (*chan s*) ⟶
K ∈ *leak s*
}

lemmas *m2-inv6-ikk-initI* = *m2-inv6-ikk-init-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv6-ikk-initE* [*elim*] = *m2-inv6-ikk-init-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv6-ikk-initD* = *m2-inv6-ikk-init-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

definition
m2-inv7-ikk-resp :: *m2-state set*
where
m2-inv7-ikk-resp ≡ {*s*. ∀ *A B Rb K Ts nl*.
runs s Rb = *Some* (*Resp*, [*A*, *B*], *aKey K* # *aNum Ts* # *nl*) ⟶
A ∈ *good* ⟶ *B* ∈ *good* ⟶ *aKey K* ∈ *extr ik0* (*chan s*) ⟶
K ∈ *leak s*
}

lemmas *m2-inv7-ikk-respI* = *m2-inv7-ikk-resp-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m2-inv7-ikk-respE* [*elim*] = *m2-inv7-ikk-resp-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m2-inv7-ikk-respD* = *m2-inv7-ikk-resp-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated 1*]

3.14.5 Refinement

The simulation relation. This is a pure superposition refinement.

definition
R12 :: (*m1-state* × *m2-state*) *set* **where**
R12 ≡ {(*s*, *t*). *runs s* = *runs t* ∧ *leak s* = *leak t* ∧ *clk s* = *clk t*}

The mediator function is the identity.

definition
med21 :: *m2-obs* ⇒ *m1-obs* **where**
med21 = *id*

Refinement proof.

lemma *PO-m2-step1-refines-m1-step1*:
{*R12*}
(*m1-step1 Ra A B*), (*m2-step1 Ra A B*)
{> *R12*}
⟨*proof*⟩

lemma *PO-m2-step2-refines-m1-step2*:
{*R12*}
(*m1-step2 Rb A B*), (*m2-step2 Rb A B*)
{> *R12*}
⟨*proof*⟩

lemma *PO-m2-step3-refines-m1-step3*:
{*R12*}
(*m1-step3 Rs A B Kab Ts*), (*m2-step3 Rs A B Kab Ts*)
{> *R12*}
⟨*proof*⟩

lemma *PO-m2-step4-refines-m1-step4*:
{*R12* ∩ *UNIV* × (*m2-inv4-M2a* ∩ *m2-inv3-extrKey*)}
(*m1-step4 Ra A B Kab Ts*), (*m2-step4 Ra A B Kab Ts*)
{> *R12*}
⟨*proof*⟩

lemma *PO-m2-step5-refines-m1-step5*:
{*R12* ∩ *UNIV* × (*m2-inv4-M2b* ∩ *m2-inv3-extrKey*)} — REMOVED!: *m2-inv5-ikk-sv*
(*m1-step5 Rb A B Kab Ts*), (*m2-step5 Rb A B Kab Ts*)
{> *R12*}
⟨*proof*⟩

lemma *PO-m2-tick-refines-m1-tick*:
{*R12*}
(*m1-tick T*), (*m2-tick T*)
{> *R12*}

⟨*proof*⟩

lemma *PO-m2-leak-refines-m1-leak*:
{*R12*}
(*m1-leak Rs*), (*m2-leak Rs*)
{> *R12*}
⟨*proof*⟩

lemma *PO-m2-fake-refines-skip*:
{*R12*}
Id, *m2-fake*
{> *R12*}
⟨*proof*⟩

All together now...

lemmas *PO-m2-trans-refines-m1-trans* =
PO-m2-step1-refines-m1-step1 PO-m2-step2-refines-m1-step2
PO-m2-step3-refines-m1-step3 PO-m2-step4-refines-m1-step4
PO-m2-step5-refines-m1-step5 PO-m2-tick-refines-m1-tick
PO-m2-leak-refines-m1-leak PO-m2-fake-refines-skip

lemma *PO-m2-refines-init-m1* [*iff*]:
init m2 ⊆ *R12*``(*init m1*)
⟨*proof*⟩

lemma *PO-m2-refines-trans-m1* [*iff*]:
{*R12* ∩
UNIV × (*m2-inv4-M2b* ∩ *m2-inv4-M2a* ∩ *m2-inv3-extrKey*)}
(*trans m1*), (*trans m2*)
{> *R12*}
⟨*proof*⟩

lemma *PO-obs-consistent-R12* [*iff*]:
obs-consistent R12 med21 m1 m2
⟨*proof*⟩

Refinement result.

lemma *m2-refines-m1* [*iff*]:
refines
(*R12* ∩
(*UNIV* × (*m2-inv4-M2b* ∩ *m2-inv4-M2a* ∩ *m2-inv3-extrKey* ∩ *m2-inv3a-sesK-compr*)))
med21 m1 m2
⟨*proof*⟩

lemma *m2-implements-m1* [*iff*]:
implements med21 m1 m2
⟨*proof*⟩

3.14.6 Inherited and derived invariants

end

3.15 Denning-Sacco, direct variant (L3)

theory *m3-ds-par* **imports** *m2-ds ../Refinement/Message*
begin

We model a direct implementation of the channel-based Denning-Sacco protocol at Level 2. In this version, there is no ticket forwarding.

$$\begin{array}{lll} \text{M1.} & A \to S: & A, B \\ \text{M2a.} & S \to A: & \{Kab, B, Ts\}_{Kas} \\ \text{M2b.} & S \to B: & \{Kab, A, Ts\}_{Kbs} \end{array}$$

Proof tool configuration. Avoid annoying automatic unfolding of *dom*.

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

3.15.1 Setup

Now we can define the initial key knowledge.

overloading *ltkeySetup′* ≡ *ltkeySetup* **begin**
definition *ltkeySetup-def*: *ltkeySetup′* ≡ {(*sharK C*, *A*) | *C A*. *A* = *C* ∨ *A* = *Sv*}
end

lemma *corrKey-shrK-bad* [*simp*]: *corrKey* = *shrK‘bad*
⟨*proof*⟩

3.15.2 State

The secure channels are star-shaped to/from the server. Therefore, we have only one agent in the relation.

record *m3-state* = *m1-state* +
  IK :: *msg set* — intruder knowledge

Observable state: *runs*, *leak*, *clk*, and *cache*.

type-synonym
  m3-obs = *m2-obs*

definition
  m3-obs :: *m3-state* ⇒ *m3-obs* **where**
  m3-obs s ≡ (| *runs* = *runs s*, *leak* = *leak s*, *clk* = *clk s* |)

type-synonym
  m3-pred = *m3-state set*

type-synonym
  m3-trans = (*m3-state* × *m3-state*) *set*

3.15.3 Events

Protocol events.

definition — by *A*, refines *m2-step1*

m3-step1 :: [*rid-t*, *agent*, *agent*] ⇒ *m3-trans*
where
m3-step1 Ra A B ≡ {(*s*, *s1*).
— guards:
Ra ∉ *dom* (*runs s*) ∧ — *Ra* is fresh

— actions:
s1 = *s*⦇
runs := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [])),
IK := *insert* ⦃*Agent A*, *Agent B*⦄ (*IK s*) — send *M1*
⦈
}

definition — by *B*, refines *m2-step2*
m3-step2 :: [*rid-t*, *agent*, *agent*] ⇒ *m3-trans*
where
m3-step2 ≡ *m1-step2*

definition — by *Server*, refines *m2-step3*
m3-step3 :: [*rid-t*, *agent*, *agent*, *key*, *time*] ⇒ *m3-trans*
where
m3-step3 Rs A B Kab Ts ≡ {(*s*, *s1*).
— guards:
Rs ∉ *dom* (*runs s*) ∧ — fresh server run
Kab = *sesK* (*Rs$sk*) ∧ — fresh session key

⦃*Agent A*, *Agent B*⦄ ∈ *IK s* ∧ — recv *M1*
Ts = *clk s* ∧ — fresh timestamp

— actions:
— record session key and send *M2*
s1 = *s*⦇
runs := (*runs s*)(*Rs* ↦ (*Serv*, [*A*, *B*], [*aNum Ts*])), — send *M2a*, *M2b*
IK := *insert* (*Crypt* (*shrK A*) ⦃*Agent B*, *Key Kab*, *Number Ts*⦄)
(*insert* (*Crypt* (*shrK B*) ⦃*Key Kab*, *Agent A*, *Number Ts*⦄) (*IK s*))
⦈
}

definition — by *A*, refines *m2-step4*
m3-step4 :: [*rid-t*, *agent*, *agent*, *key*, *time*] ⇒ *m3-trans*
where
m3-step4 Ra A B Kab Ts ≡ {(*s*, *s1*).

— guards:
runs s Ra = *Some* (*Init*, [*A*, *B*], []) ∧ — key not yet recv'd

Crypt (*shrK A*) — recv *M2*
⦃*Agent B*, *Key Kab*, *Number Ts*⦄ ∈ *IK s* ∧

— check freshness of session key
clk s < *Ts* + *Ls* ∧

— actions:

— record session key
s1 = *s*⦇
runs := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [*aKey Kab*, *aNum Ts*]))
⦈
}

definition — by *B*, refines *m2-step5*
m3-step5 :: [*rid-t*, *agent*, *agent*, *key*, *time*] ⇒ *m3-trans*
where
m3-step5 Rb A B Kab Ts ≡ {(*s*, *s1*).
— guards:
runs s Rb = *Some* (*Resp*, [*A*, *B*], []) ∧ — key not yet recv'd

Crypt (*shrK B*) ⦃*Key Kab*, *Agent A*, *Number Ts*⦄ ∈ *IK s* ∧ — recv *M3*

— ensure freshness of session key; replays with fresh authenticator ok!
clk s < *Ts* + *Ls* ∧

— actions:
— record session key
s1 = *s*⦇
runs := (*runs s*)(*Rb* ↦ (*Resp*, [*A*, *B*], [*aKey Kab*, *aNum Ts*]))
⦈
}

Clock tick event

definition — refines *m2-tick*
m3-tick :: *time* ⇒ *m3-trans*
where
m3-tick ≡ *m1-tick*

Session key compromise.

definition — refines *m2-leak*
m3-leak :: *rid-t* ⇒ *m3-trans*
where
m3-leak Rs ≡ {(*s*, *s1*).
— guards:
Rs ∈ *dom* (*runs s*) ∧
fst (*the* (*runs s Rs*)) = *Serv* ∧ — compromise server run *Rs*

— actions:
— record session key as leaked and add it to intruder knowledge
s1 = *s*⦇ *leak* := *insert* (*sesK* (*Rs*$*sk*)) (*leak s*),
IK := *insert* (*Key* (*sesK* (*Rs*$*sk*))) (*IK s*) ⦈
}

Intruder fake event. The following "Dolev-Yao" event generates all intruder-derivable messages.

definition — refines *m2-fake*
m3-DY-fake :: *m3-trans*
where
m3-DY-fake ≡ {(*s*, *s1*).

— actions:
s1 = *s*⦇ *IK* := *synth* (*analz* (*IK* *s*)) ⦈ — take DY closure
}

3.15.4 Transition system

definition
m3-init :: *m3-pred*
where
m3-init ≡ { ⦇
runs = *Map.empty*,
leak = *shrK*'*bad*,
clk = *0*,
IK = *Key*'*shrK*'*bad*
⦈ }

definition
m3-trans :: *m3-trans* **where**
m3-trans ≡ (⋃ *A B Ra Rb Rs Kab Ts T*.
m3-step1 Ra A B ∪
m3-step2 Rb A B ∪
m3-step3 Rs A B Kab Ts ∪
m3-step4 Ra A B Kab Ts ∪
m3-step5 Rb A B Kab Ts ∪
m3-tick T ∪
m3-leak Rs ∪
m3-DY-fake ∪
Id
)

definition
m3 :: (*m3-state*, *m3-obs*) *spec* **where**
m3 ≡ ⦇
init = *m3-init*,
trans = *m3-trans*,
obs = *m3-obs*
⦈

lemmas *m3-loc-defs* =
m3-def m3-init-def m3-trans-def m3-obs-def
m3-step1-def m3-step2-def m3-step3-def m3-step4-def m3-step5-def
m3-tick-def m3-leak-def m3-DY-fake-def

lemmas *m3-defs* = *m3-loc-defs m2-defs*

3.15.5 Invariants

Specialized injection that we can apply more aggressively.

lemmas *analz-Inj-IK* = *analz.Inj* [**where** *H*=*IK s* **for** *s*]
lemmas *parts-Inj-IK* = *parts.Inj* [**where** *H*=*IK s* **for** *s*]

declare *parts-Inj-IK* [*dest*!]

declare *analz-into-parts* [*dest*]

inv1: Secrecy of pre-distributed shared keys

definition
  m3-inv1-lkeysec :: *m3-pred*
where
  m3-inv1-lkeysec ≡ {*s*. ∀ *C*.
    (*Key* (*shrK C*) ∈ *parts* (*IK s*) ⟶ *C* ∈ *bad*) ∧
    (*C* ∈ *bad* ⟶ *Key* (*shrK C*) ∈ *IK s*)
  }

lemmas *m3-inv1-lkeysecI* = *m3-inv1-lkeysec-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv1-lkeysecE* [*elim*] = *m3-inv1-lkeysec-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv1-lkeysecD* = *m3-inv1-lkeysec-def* [*THEN setc-def-to-dest*, *rule-format*]

Invariance proof.

lemma *PO-m3-inv1-lkeysec-init* [*iff*]:
  init m3 ⊆ *m3-inv1-lkeysec*
⟨*proof*⟩

lemma *PO-m3-inv1-lkeysec-trans* [*iff*]:
  {*m3-inv1-lkeysec*} *trans m3* {> *m3-inv1-lkeysec*}
⟨*proof*⟩

lemma *PO-m3-inv1-lkeysec* [*iff*]: *reach m3* ⊆ *m3-inv1-lkeysec*
⟨*proof*⟩

Useful simplifier lemmas

lemma *m3-inv1-lkeysec-for-parts* [*simp*]:
  ⟦ *s* ∈ *m3-inv1-lkeysec* ⟧ ⟹ *Key* (*shrK C*) ∈ *parts* (*IK s*) ⟷ *C* ∈ *bad*
⟨*proof*⟩

lemma *m3-inv1-lkeysec-for-analz* [*simp*]:
  ⟦ *s* ∈ *m3-inv1-lkeysec* ⟧ ⟹ *Key* (*shrK C*) ∈ *analz* (*IK s*) ⟷ *C* ∈ *bad*
⟨*proof*⟩

inv3: Session keys not used to encrypt other session keys

Session keys are not used to encrypt other keys. Proof requires generalization to sets of session keys.

NOTE: This invariant will be derived from the corresponding L2 invariant using the simulation relation.

definition
  m3-inv3-sesK-compr :: *m3-pred*
where
  m3-inv3-sesK-compr ≡ {*s*. ∀ *K KK*.
    KK ⊆ *range sesK* ⟶
    (*Key K* ∈ *analz* (*Key*‘*KK* ∪ (*IK s*))) = (*K* ∈ *KK* ∨ *Key K* ∈ *analz* (*IK s*))

}

lemmas *m3-inv3-sesK-comprI* = *m3-inv3-sesK-compr-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv3-sesK-comprE* = *m3-inv3-sesK-compr-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv3-sesK-comprD* = *m3-inv3-sesK-compr-def* [*THEN setc-def-to-dest*, *rule-format*]

Additional lemma

lemmas *insert-commute-Key* = *insert-commute* [**where** *x*=*Key K* **for** *K*]

lemmas *m3-inv3-sesK-compr-simps* =
m3-inv3-sesK-comprD
m3-inv3-sesK-comprD [**where** *KK*=*insert Kab KK* **for** *Kab KK*, *simplified*]
m3-inv3-sesK-comprD [**where** *KK*={*Kab*} **for** *Kab*, *simplified*]
insert-commute-Key

3.15.6 Refinement

Message abstraction and simulation relation

Abstraction function on sets of messages.

inductive-set
abs-msg :: *msg set* ⇒ *chmsg set*
for *H* :: *msg set*
where
am-M1:
{|*Agent A*, *Agent B*|} ∈ *H*
⟹ *Insec A B* (*Msg* []) ∈ *abs-msg H*
| *am-M2a*:
Crypt (*shrK C*) {|*Agent B*, *Key K*, *Number T*|} ∈ *H*
⟹ *Secure Sv C* (*Msg* [*aAgt B*, *aKey K*, *aNum T*]) ∈ *abs-msg H*
| *am-M2b*:
Crypt (*shrK C*) {|*Key K*, *Agent A*, *Number T*|} ∈ *H*
⟹ *Secure Sv C* (*Msg* [*aKey K*, *aAgt A*, *aNum T*]) ∈ *abs-msg H*

R23: The simulation relation. This is a data refinement of the insecure and secure channels of refinement 2.

definition
R23-msgs :: (*m2-state* × *m3-state*) *set* **where**
R23-msgs ≡ {(*s*, *t*). *abs-msg* (*parts* (*IK t*)) ⊆ *chan s* }

definition
R23-keys :: (*m2-state* × *m3-state*) *set* **where**
R23-keys ≡ {(*s*, *t*). ∀ *KK K*. *KK* ⊆ *range sesK* ⟶
Key K ∈ *analz* (*Key*‘*KK* ∪ (*IK t*)) ⟷ *aKey K* ∈ *extr* (*aKey*‘*KK* ∪ *ik0*) (*chan s*)
}

definition
R23-pres :: (*m2-state* × *m3-state*) *set* **where**
R23-pres ≡ {(*s*, *t*). *runs s* = *runs t* ∧ *leak s* = *leak t* ∧ *clk s* = *clk t*}

definition
R23 :: (*m2-state* × *m3-state*) *set* **where**

R23 ≡ *R23-msgs* ∩ *R23-keys* ∩ *R23-pres*

lemmas *R23-defs* =
R23-def R23-msgs-def R23-keys-def R23-pres-def

The mediator function is the identity here.

definition
med32 :: *m3-obs* ⇒ *m2-obs* **where**
med32 ≡ *id*

lemmas *R23-msgsI* = *R23-msgs-def* [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-msgsE* [*elim*] = *R23-msgs-def* [*THEN rel-def-to-elim*, *simplified*, *rule-format*]

lemmas *R23-keysI* = *R23-keys-def* [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-keysE* [*elim*] = *R23-keys-def* [*THEN rel-def-to-elim*, *simplified*, *rule-format*]

lemmas *R23-presI* = *R23-pres-def* [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-presE* [*elim*] = *R23-pres-def* [*THEN rel-def-to-elim*, *simplified*, *rule-format*]

lemmas *R23-intros* = *R23-msgsI R23-keysI R23-presI*

Simplifier lemmas for various instantiations (for keys).

lemmas *R23-keys-simp* = *R23-keys-def* [*THEN rel-def-to-dest*, *simplified*, *rule-format*]
lemmas *R23-keys-simps* =
R23-keys-simp
R23-keys-simp [**where** *KK*={}, *simplified*]
R23-keys-simp [**where** *KK*={*K'*} **for** *K'*, *simplified*]
R23-keys-simp [**where** *KK*=*insert K' KK* **for** *K' KK*, *simplified*, *OF - conjI*]

General lemmas

General facts about *abs-msg*

declare *abs-msg.intros* [*intro*!]
declare *abs-msg.cases* [*elim*!]

lemma *abs-msg-empty*: *abs-msg* {} = {}
⟨*proof*⟩

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

lemma *abs-msg-mono* [*elim*]:
⟦ *m* ∈ *abs-msg G*; *G* ⊆ *H* ⟧ ⟹ *m* ∈ *abs-msg H*
⟨*proof*⟩

lemma *abs-msg-insert-mono* [*intro*]:
⟦ *m* ∈ *abs-msg H* ⟧ ⟹ *m* ∈ *abs-msg* (*insert m' H*)
⟨*proof*⟩

Facts about *abs-msg* concerning abstraction of fakeable messages. This is crucial for proving the refinement of the intruder event.

lemma *abs-msg-DY-subset-fakeable*:
  ⟦ (*s*, *t*) ∈ *R23-msgs*; (*s*, *t*) ∈ *R23-keys*; *t* ∈ *m3-inv1-lkeysec* ⟧
  ⟹ *abs-msg* (*synth* (*analz* (*IK t*))) ⊆ *fake ik0* (*dom* (*runs s*)) (*chan s*)
⟨*proof*⟩

Refinement proof

Pair decomposition. These were set to `elim!`, which is too agressive here.

declare *MPair-analz* [*rule del*, *elim*]
declare *MPair-parts* [*rule del*, *elim*]

Protocol events.

lemma *PO-m3-step1-refines-m2-step1*:
  {*R23*}
    (*m2-step1 Ra A B*), (*m3-step1 Ra A B*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-step2-refines-m2-step2*:
  {*R23*}
    (*m2-step2 Rb A B*), (*m3-step2 Rb A B*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-step3-refines-m2-step3*:
  {*R23* ∩ (*m2-inv3a-sesK-compr*) × (*m3-inv3-sesK-compr* ∩ *m3-inv1-lkeysec*)}
    (*m2-step3 Rs A B Kab Ts*), (*m3-step3 Rs A B Kab Ts*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-step4-refines-m2-step4*:
  {*R23* ∩ *UNIV* × (*m3-inv1-lkeysec*)}
    (*m2-step4 Ra A B Kab Ts*), (*m3-step4 Ra A B Kab Ts*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-step5-refines-m2-step5*:
  {*R23*}
    (*m2-step5 Rb A B Kab Ts*), (*m3-step5 Rb A B Kab Ts*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-tick-refines-m2-tick*:
  {*R23*}
    (*m2-tick T*), (*m3-tick T*)
  {>*R23*}
⟨*proof*⟩

Intruder events.

lemma *PO-m3-leak-refines-m2-leak*:
{*R23*}
(*m2-leak Rs*), (*m3-leak Rs*)
{>*R23*}
⟨*proof*⟩

lemma *PO-m3-DY-fake-refines-m2-fake*:
{*R23* ∩ *UNIV* × (*m3-inv1-lkeysec*)}
m2-fake, *m3-DY-fake*
{> *R23*}
⟨*proof*⟩

All together now...

lemmas *PO-m3-trans-refines-m2-trans* =
PO-m3-step1-refines-m2-step1 PO-m3-step2-refines-m2-step2
PO-m3-step3-refines-m2-step3 PO-m3-step4-refines-m2-step4
PO-m3-step5-refines-m2-step5 PO-m3-tick-refines-m2-tick
PO-m3-leak-refines-m2-leak PO-m3-DY-fake-refines-m2-fake

lemma *PO-m3-refines-init-m2* [*iff*]:
init m3 ⊆ *R23*``(*init m2*)
⟨*proof*⟩

lemma *PO-m3-refines-trans-m2* [*iff*]:
{*R23* ∩ (*m2-inv3a-sesK-compr*) × (*m3-inv3-sesK-compr* ∩ *m3-inv1-lkeysec*)}
(*trans m2*), (*trans m3*)
{> *R23*}
⟨*proof*⟩

lemma *PO-m3-observation-consistent* [*iff*]:
obs-consistent R23 med32 m2 m3
⟨*proof*⟩

Refinement result.

lemma *m3-refines-m2* [*iff*]:
refines
(*R23* ∩ (*m2-inv3a-sesK-compr*) × (*m3-inv1-lkeysec*))
med32 m2 m3
⟨*proof*⟩

lemma *m3-implements-m2* [*iff*]:
implements med32 m2 m3
⟨*proof*⟩

end

3.16 Denning-Sacco protocol (L3)

theory *m3-ds* **imports** *m2-ds ../Refinement/Message*
begin

We model the Denning-Sacco protocol:

$$\begin{array}{lll} \text{M1.} & A \rightarrow S: & A, B \\ \text{M2.} & S \rightarrow A: & \{Kab, B, Ts, Na, \{Kab, A, Ts\}_{Kbs}\}_{Kas} \\ \text{M3.} & A \rightarrow B: & \{Kab, A, Ts\}_{Kbs} \end{array}$$

Proof tool configuration. Avoid annoying automatic unfolding of *dom*.

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

3.16.1 Setup

Now we can define the initial key knowledge.

overloading *ltkeySetup'* ≡ *ltkeySetup* **begin**
definition *ltkeySetup-def*: *ltkeySetup'* ≡ {(*sharK C*, *A*) | *C A*. *A* = *C* ∨ *A* = *Sv*}
end

lemma *corrKey-shrK-bad* [*simp*]: *corrKey* = *shrK‘bad*
⟨*proof*⟩

3.16.2 State

The secure channels are star-shaped to/from the server. Therefore, we have only one agent in the relation.

record *m3-state* = *m1-state* +
  IK :: *msg set* — intruder knowledge

Observable state: *runs*, *leak*, *clk*, and *cache*.

type-synonym
  m3-obs = *m2-obs*

definition
  m3-obs :: *m3-state* ⇒ *m3-obs* **where**
  m3-obs s ≡ ⦇ *runs* = *runs s*, *leak* = *leak s*, *clk* = *clk s* ⦈

type-synonym
  m3-pred = *m3-state set*

type-synonym
  m3-trans = (*m3-state* × *m3-state*) *set*

3.16.3 Events

Protocol events.

definition — by *A*, refines *m2-step1*
  m3-step1 :: [*rid-t*, *agent*, *agent*] ⇒ *m3-trans*
where
  m3-step1 Ra A B ≡ {(*s*, *s1*).
    — guards:
    Ra ∉ *dom* (*runs s*) ∧ — *Ra* is fresh

— actions:
s1 = *s*⦇
runs := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [])),
IK := *insert* ⦃*Agent A*, *Agent B*⦄ (*IK s*) — send *M1*
⦈
}

definition — by *B*, refines *m2-step2*
m3-step2 :: [*rid-t*, *agent*, *agent*] ⇒ *m3-trans*
where
m3-step2 ≡ *m1-step2*

definition — by *Server*, refines *m2-step3*
m3-step3 :: [*rid-t*, *agent*, *agent*, *key*, *time*] ⇒ *m3-trans*
where
m3-step3 Rs A B Kab Ts ≡ {(*s*, *s1*).
— guards:
Rs ∉ *dom* (*runs s*) ∧ — fresh server run
Kab = *sesK* (*Rs*$*sk*) ∧ — fresh session key

⦃*Agent A*, *Agent B*⦄ ∈ *IK s* ∧ — recv *M1*
Ts = *clk s* ∧ — fresh timestamp

— actions:
— record session key and send *M2*
s1 = *s*⦇
runs := (*runs s*)(*Rs* ↦ (*Serv*, [*A*, *B*], [*aNum Ts*])),
IK := *insert* (*Crypt* (*shrK A*) — send *M2*
⦃*Key Kab*, *Agent B*, *Number Ts*,
Crypt (*shrK B*) ⦃*Key Kab*, *Agent A*, *Number Ts*⦄⦄)
(*IK s*)
⦈
}

definition — by *A*, refines *m2-step4*
m3-step4 :: [*rid-t*, *agent*, *agent*, *key*, *time*, *msg*] ⇒ *m3-trans*
where
m3-step4 Ra A B Kab Ts X ≡ {(*s*, *s1*).

— guards:
runs s Ra = *Some* (*Init*, [*A*, *B*], []) ∧ — key not yet recv'd

Crypt (*shrK A*) — recv *M2*
⦃*Key Kab*, *Agent B*, *Number Ts*, *X*⦄ ∈ *IK s* ∧

— check freshness of session key
clk s < *Ts* + *Ls* ∧

— actions:
— record session key and send *M3*
s1 = *s*⦇
runs := (*runs s*)(*Ra* ↦ (*Init*, [*A*, *B*], [*aKey Kab*, *aNum Ts*])),

IK := *insert X* (*IK s*) — send *M3*
⦈
}

definition — by *B*, refines *m2-step5*
m3-step5 :: [*rid-t*, *agent*, *agent*, *key*, *time*] ⇒ *m3-trans*
where
m3-step5 Rb A B Kab Ts ≡ {(*s*, *s1*).
— guards:
runs s Rb = *Some* (*Resp*, [*A*, *B*], []) ∧ — key not yet recv'd

Crypt (*shrK B*) ⦃*Key Kab*, *Agent A*, *Number Ts*⦄ ∈ *IK s* ∧ — recv *M3*

— ensure freshness of session key; replays with fresh authenticator ok!
clk s < *Ts* + *Ls* ∧

— actions:
— record session key
s1 = *s*⦇
runs := (*runs s*)(*Rb* ↦ (*Resp*, [*A*, *B*], [*aKey Kab*, *aNum Ts*]))
⦈
}

Clock tick event

definition — refines *m2-tick*
m3-tick :: *time* ⇒ *m3-trans*
where
m3-tick ≡ *m1-tick*

Session key compromise.

definition — refines *m2-leak*
m3-leak :: *rid-t* ⇒ *m3-trans*
where
m3-leak Rs ≡ {(*s*, *s1*).
— guards:
Rs ∈ *dom* (*runs s*) ∧
fst (*the* (*runs s Rs*)) = *Serv* ∧ — compromise server run *Rs*

— actions:
— record session key as leaked and add it to intruder knowledge
s1 = *s*⦇ *leak* := *insert* (*sesK* (*Rs*$*sk*)) (*leak s*),
IK := *insert* (*Key* (*sesK* (*Rs*$*sk*))) (*IK s*) ⦈
}

Intruder fake event. The following "Dolev-Yao" event generates all intruder-derivable messages.

definition — refines *m2-fake*
m3-DY-fake :: *m3-trans*
where
m3-DY-fake ≡ {(*s*, *s1*).

— actions:

s1 = *s*⦇ *IK* := *synth* (*analz* (*IK s*)) ⦈ — take DY closure
}

3.16.4 Transition system

definition
m3-init :: *m3-pred*
where
m3-init ≡ { ⦇
runs = *Map.empty*,
leak = *shrK*‘*bad*,
clk = *0*,
IK = *Key*‘*shrK*‘*bad*
⦈ }

definition
m3-trans :: *m3-trans* **where**
m3-trans ≡ (⋃ *A B Ra Rb Rs Kab Ts T X*.
m3-step1 Ra A B ∪
m3-step2 Rb A B ∪
m3-step3 Rs A B Kab Ts ∪
m3-step4 Ra A B Kab Ts X ∪
m3-step5 Rb A B Kab Ts ∪
m3-tick T ∪
m3-leak Rs ∪
m3-DY-fake ∪
Id
)

definition
m3 :: (*m3-state*, *m3-obs*) *spec* **where**
m3 ≡ ⦇
init = *m3-init*,
trans = *m3-trans*,
obs = *m3-obs*
⦈

lemmas *m3-loc-defs* =
m3-def m3-init-def m3-trans-def m3-obs-def
m3-step1-def m3-step2-def m3-step3-def m3-step4-def m3-step5-def
m3-tick-def m3-leak-def m3-DY-fake-def

lemmas *m3-defs* = *m3-loc-defs m2-defs*

3.16.5 Invariants

Specialized injection that we can apply more aggressively.

lemmas *analz-Inj-IK* = *analz.Inj* [**where** *H*=*IK s* **for** *s*]
lemmas *parts-Inj-IK* = *parts.Inj* [**where** *H*=*IK s* **for** *s*]

declare *parts-Inj-IK* [*dest*!]

declare *analz-into-parts* [*dest*]

inv1: Secrecy of pre-distributed shared keys

definition
  m3-inv1-lkeysec :: *m3-pred*
where
  m3-inv1-lkeysec ≡ {*s*. ∀ *C*.
    (*Key* (*shrK C*) ∈ *parts* (*IK s*) ⟶ *C* ∈ *bad*) ∧
    (*C* ∈ *bad* ⟶ *Key* (*shrK C*) ∈ *IK s*)
  }

lemmas *m3-inv1-lkeysecI* = *m3-inv1-lkeysec-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv1-lkeysecE* [*elim*] = *m3-inv1-lkeysec-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv1-lkeysecD* = *m3-inv1-lkeysec-def* [*THEN setc-def-to-dest*, *rule-format*]

Invariance proof.

lemma *PO-m3-inv1-lkeysec-init* [*iff*]:
  init m3 ⊆ *m3-inv1-lkeysec*
⟨*proof*⟩

lemma *PO-m3-inv1-lkeysec-trans* [*iff*]:
  {*m3-inv1-lkeysec*} *trans m3* {> *m3-inv1-lkeysec*}
⟨*proof*⟩

lemma *PO-m3-inv1-lkeysec* [*iff*]: *reach m3* ⊆ *m3-inv1-lkeysec*
⟨*proof*⟩

Useful simplifier lemmas

lemma *m3-inv1-lkeysec-for-parts* [*simp*]:
  ⟦ *s* ∈ *m3-inv1-lkeysec* ⟧ ⟹ *Key* (*shrK C*) ∈ *parts* (*IK s*) ⟷ *C* ∈ *bad*
⟨*proof*⟩

lemma *m3-inv1-lkeysec-for-analz* [*simp*]:
  ⟦ *s* ∈ *m3-inv1-lkeysec* ⟧ ⟹ *Key* (*shrK C*) ∈ *analz* (*IK s*) ⟷ *C* ∈ *bad*
⟨*proof*⟩

inv2: Ticket shape for honestly encrypted M2

definition
  m3-inv2-ticket :: *m3-pred*
where
  m3-inv2-ticket ≡ {*s*. ∀ *A B T K X*.
    A ∉ *bad* ⟶
    Crypt (*shrK A*) ⦃*Key K*, *Agent B*, *Number T*, *X*⦄ ∈ *parts* (*IK s*) ⟶
      X = *Crypt* (*shrK B*) ⦃*Key K*, *Agent A*, *Number T*⦄ ∧ *K* ∈ *range sesK*
  }

lemmas *m3-inv2-ticketI* = *m3-inv2-ticket-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv2-ticketE* [*elim*] = *m3-inv2-ticket-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv2-ticketD* = *m3-inv2-ticket-def* [*THEN setc-def-to-dest*, *rule-format*, *rotated −1*]

Invariance proof.

lemma *PO-m3-inv2-ticket-init* [*iff*]:
init m3 ⊆ *m3-inv2-ticket*
⟨*proof*⟩

lemma *PO-m3-inv2-ticket-trans* [*iff*]:
{*m3-inv2-ticket* ∩ *m3-inv1-lkeysec*} *trans m3* {> *m3-inv2-ticket*}
⟨*proof*⟩

lemma *PO-m3-inv2-ticket* [*iff*]: *reach m3* ⊆ *m3-inv2-ticket*
⟨*proof*⟩

inv3: Session keys not used to encrypt other session keys

Session keys are not used to encrypt other keys. Proof requires generalization to sets of session keys.

definition
m3-inv3-sesK-compr :: *m3-pred*
where
m3-inv3-sesK-compr ≡ {*s*. ∀ *K KK*.
KK ⊆ *range sesK* ⟶
(*Key K* ∈ *analz* (*Key'KK* ∪ (*IK s*))) = (*K* ∈ *KK* ∨ *Key K* ∈ *analz* (*IK s*))
}

lemmas *m3-inv3-sesK-comprI* = *m3-inv3-sesK-compr-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *m3-inv3-sesK-comprE* = *m3-inv3-sesK-compr-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *m3-inv3-sesK-comprD* = *m3-inv3-sesK-compr-def* [*THEN setc-def-to-dest*, *rule-format*]

Additional lemma

lemmas *insert-commute-Key* = *insert-commute* [**where** *x*=*Key K* **for** *K*]

lemmas *m3-inv3-sesK-compr-simps* =
m3-inv3-sesK-comprD
m3-inv3-sesK-comprD [**where** *KK*={*Kab*} **for** *Kab*, *simplified*]
m3-inv3-sesK-comprD [**where** *KK*=*insert Kab KK* **for** *Kab KK*, *simplified*]
insert-commute-Key — to get the keys to the front

Invariance proof.

lemma *PO-m3-inv3-sesK-compr-step4*:
{*m3-inv3-sesK-compr* ∩ *m3-inv2-ticket* ∩ *m3-inv1-lkeysec*}
m3-step4 Ra A B Kab Ts X
{> *m3-inv3-sesK-compr*}
⟨*proof*⟩

All together now.

lemmas *PO-m3-inv3-sesK-compr-trans-lemmas* =
PO-m3-inv3-sesK-compr-step4

lemma *PO-m3-inv3-sesK-compr-init* [*iff*]:
init m3 ⊆ *m3-inv3-sesK-compr*
⟨*proof*⟩

lemma *PO-m3-inv3-sesK-compr-trans* [*iff*]:
 {*m3-inv3-sesK-compr* ∩ *m3-inv2-ticket* ∩ *m3-inv1-lkeysec*}
  trans m3
 {> *m3-inv3-sesK-compr*}
⟨*proof*⟩

lemma *PO-m3-inv3-sesK-compr* [*iff*]: *reach m3* ⊆ *m3-inv3-sesK-compr*
⟨*proof*⟩

3.16.6 Refinement

Message abstraction and simulation relation

Abstraction function on sets of messages.

inductive-set
 abs-msg :: *msg set* ⇒ *chmsg set*
 for *H* :: *msg set*
where
 am-M1:
  ⦃*Agent A*, *Agent B*⦄ ∈ *H*
 ⟹ *Insec A B* (*Msg* []) ∈ *abs-msg H*
| *am-M2a*:
  Crypt (*shrK C*) ⦃*Key K*, *Agent B*, *Number T*, *X*⦄ ∈ *H*
 ⟹ *Secure Sv C* (*Msg* [*aAgt B*, *aKey K*, *aNum T*]) ∈ *abs-msg H*
| *am-M2b*:
  Crypt (*shrK C*) ⦃*Key K*, *Agent A*, *Number T*⦄ ∈ *H*
 ⟹ *Secure Sv C* (*Msg* [*aKey K*, *aAgt A*, *aNum T*]) ∈ *abs-msg H*

R23: The simulation relation. This is a data refinement of the insecure and secure channels of refinement 2.

definition
 R23-msgs :: (*m2-state* × *m3-state*) *set* **where**
 R23-msgs ≡ {(*s*, *t*). *abs-msg* (*parts* (*IK t*)) ⊆ *chan s* }

definition
 R23-keys :: (*m2-state* × *m3-state*) *set* **where**
 R23-keys ≡ {(*s*, *t*). ∀ *KK K*. *KK* ⊆ *range sesK* ⟶
  Key K ∈ *analz* (*Key*‘*KK* ∪ (*IK t*)) ⟶ *aKey K* ∈ *extr* (*aKey*‘*KK* ∪ *ik0*) (*chan s*)
 }

definition
 R23-pres :: (*m2-state* × *m3-state*) *set* **where**
 R23-pres ≡ {(*s*, *t*). *runs s* = *runs t* ∧ *clk s* = *clk t* ∧ *leak s* = *leak t*}

definition
 R23 :: (*m2-state* × *m3-state*) *set* **where**
 R23 ≡ *R23-msgs* ∩ *R23-keys* ∩ *R23-pres*

lemmas *R23-defs* =
 R23-def R23-msgs-def R23-keys-def R23-pres-def

The mediator function is the identity here.

definition
med32 :: *m3-obs* ⇒ *m2-obs* **where**
med32 ≡ *id*

lemmas *R23-msgsI* = *R23-msgs-def* [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-msgsE* [*elim*] = *R23-msgs-def* [*THEN rel-def-to-elim*, *simplified*, *rule-format*]

lemmas *R23-keysI* = *R23-keys-def* [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-keysE* [*elim*] = *R23-keys-def* [*THEN rel-def-to-elim*, *simplified*, *rule-format*]

lemmas *R23-presI* = *R23-pres-def* [*THEN rel-def-to-intro*, *simplified*, *rule-format*]
lemmas *R23-presE* [*elim*] = *R23-pres-def* [*THEN rel-def-to-elim*, *simplified*, *rule-format*]

lemmas *R23-intros* = *R23-msgsI R23-keysI R23-presI*

Lemmas for various instantiations (for keys).

lemmas *R23-keys-dest* = *R23-keys-def* [*THEN rel-def-to-dest*, *simplified*, *rule-format*, *rotated 2*]
lemmas *R23-keys-dests* =
R23-keys-dest
R23-keys-dest [**where** *KK*={}, *simplified*]
R23-keys-dest [**where** *KK*={*K′*} **for** *K′*, *simplified*]
R23-keys-dest [**where** *KK*=*insert K′ KK* **for** *K′ KK*, *simplified*, *OF - - conjI*]

General lemmas

General facts about *abs-msg*

declare *abs-msg.intros* [*intro*!]
declare *abs-msg.cases* [*elim*!]

lemma *abs-msg-empty*: *abs-msg* {} = {}
⟨*proof*⟩

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

lemma *abs-msg-mono* [*elim*]:
⟦ *m* ∈ *abs-msg G*; *G* ⊆ *H* ⟧ ⟹ *m* ∈ *abs-msg H*
⟨*proof*⟩

lemma *abs-msg-insert-mono* [*intro*]:
⟦ *m* ∈ *abs-msg H* ⟧ ⟹ *m* ∈ *abs-msg* (*insert m′ H*)
⟨*proof*⟩

Facts about *abs-msg* concerning abstraction of fakeable messages. This is crucial for proving the refinement of the intruder event.

lemma *abs-msg-DY-subset-fakeable*:
⟦ (*s*, *t*) ∈ *R23-msgs*; (*s*, *t*) ∈ *R23-keys*; (*s*, *t*) ∈ *R23-non*; *t* ∈ *m3-inv1-lkeysec* ⟧
⟹ *abs-msg* (*synth* (*analz* (*IK t*))) ⊆ *fake ik0* (*dom* (*runs s*)) (*chan s*)
⟨*proof*⟩

Refinement proof

Pair decomposition. These were set to `elim!`, which is too agressive here.

declare *MPair-analz* [*rule del*, *elim*]
declare *MPair-parts* [*rule del*, *elim*]

Protocol events.

lemma *PO-m3-step1-refines-m2-step1*:
  {*R23*}
    (*m2-step1 Ra A B*), (*m3-step1 Ra A B*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-step2-refines-m2-step2*:
  {*R23*}
    (*m2-step2 Rb A B*), (*m3-step2 Rb A B*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-step3-refines-m2-step3*:
  {*R23* ∩ (*m2-inv3a-sesK-compr*) × (*m3-inv3-sesK-compr* ∩ *m3-inv1-lkeysec*)}
    (*m2-step3 Rs A B Kab Ts*), (*m3-step3 Rs A B Kab Ts*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-step4-refines-m2-step4*:
  {*R23* ∩
    UNIV × (*m3-inv3-sesK-compr* ∩ *m3-inv2-ticket* ∩ *m3-inv1-lkeysec*)}
    (*m2-step4 Ra A B Kab Ts*), (*m3-step4 Ra A B Kab Ts X*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-step5-refines-m2-step5*:
  {*R23*}
    (*m2-step5 Rb A B Kab Ts*), (*m3-step5 Rb A B Kab Ts*)
  {> *R23*}
⟨*proof*⟩

lemma *PO-m3-tick-refines-m2-tick*:
  {*R23*}
   (*m2-tick T*), (*m3-tick T*)
  {>*R23*}
⟨*proof*⟩

Intruder events.

lemma *PO-m3-leak-refines-m2-leak*:
  {*R23*}
   (*m2-leak Rs*), (*m3-leak Rs*)
  {>*R23*}
⟨*proof*⟩

lemma *PO-m3-DY-fake-refines-m2-fake*:

{*R23* ∩ *UNIV* × (*m3-inv1-lkeysec*)}
m2-fake, *m3-DY-fake*
{> *R23*}
⟨*proof*⟩

All together now...

lemmas *PO-m3-trans-refines-m2-trans* =
PO-m3-step1-refines-m2-step1 PO-m3-step2-refines-m2-step2
PO-m3-step3-refines-m2-step3 PO-m3-step4-refines-m2-step4
PO-m3-step5-refines-m2-step5 PO-m3-tick-refines-m2-tick
PO-m3-leak-refines-m2-leak PO-m3-DY-fake-refines-m2-fake

lemma *PO-m3-refines-init-m2* [*iff*]:
init m3 ⊆ *R23*``(*init m2*)
⟨*proof*⟩

lemma *PO-m3-refines-trans-m2* [*iff*]:
{*R23* ∩
(*m2-inv3a-sesK-compr*) × (*m3-inv3-sesK-compr* ∩ *m3-inv2-ticket* ∩ *m3-inv1-lkeysec*)}
(*trans m2*), (*trans m3*)
{> *R23*}
⟨*proof*⟩

lemma *PO-m3-observation-consistent* [*iff*]:
obs-consistent R23 med32 m2 m3
⟨*proof*⟩

Refinement result.

lemma *m3-refines-m2* [*iff*]:
refines
(*R23* ∩ (*m2-inv3a-sesK-compr*) × (*m3-inv3-sesK-compr* ∩ *m3-inv2-ticket* ∩ *m3-inv1-lkeysec*))
med32 m2 m3
⟨*proof*⟩

lemma *m3-implements-m2* [*iff*]:
implements med32 m2 m3
⟨*proof*⟩

end