

# Refining Authenticated Key Agreement with Strong Adversaries

Joseph Lallemand, Christoph Sprenger, and David Basin

February 23, 2021

## Contents

<b>1</b>	<b>Proving infrastructure</b>	<b>9</b>
1.1	Prover configuration . . . . .	9
1.2	Forward reasoning (“attributes”) . . . . .	9
1.3	General results . . . . .	9
1.3.1	Maps . . . . .	9
1.3.2	Set . . . . .	9
1.3.3	Relations . . . . .	10
1.3.4	Lists . . . . .	10
1.3.5	Finite sets . . . . .	10
<b>2</b>	<b>Models, Invariants and Refinements</b>	<b>11</b>
2.1	Specifications, reachability, and behaviours. . . . .	11
2.1.1	Finite behaviours . . . . .	11
2.1.2	Specifications, observability, and implementation . . . . .	12
2.2	Invariants . . . . .	15
2.2.1	Hoare triples . . . . .	15
2.2.2	Characterization of reachability . . . . .	16
2.2.3	Invariant proof rules . . . . .	16
2.3	Refinement . . . . .	17
2.3.1	Relational Hoare tuples . . . . .	17
2.3.2	Refinement proof obligations . . . . .	20
2.3.3	Deriving invariants from refinements . . . . .	21
2.3.4	Refinement of specifications . . . . .	22
<b>3</b>	<b>Message definitions</b>	<b>25</b>
3.1	Messages . . . . .	25
<b>4</b>	<b>Message theory</b>	<b>35</b>
4.1	Message composition . . . . .	35
4.2	Message decomposition . . . . .	37
4.3	Lemmas about combined composition/decomposition . . . . .	40
4.4	Accessible message parts . . . . .	41
4.4.1	Lemmas about combinations with composition and decomposition . . . . .	44
4.5	More lemmas about combinations of closures . . . . .	45

<b>5</b>	<b>Environment: Dolev-Yao Intruder</b>	<b>48</b>
<b>6</b>	<b>Secrecy Model (L0)</b>	<b>49</b>
6.1	State and events . . . . .	49
6.2	Proof of secrecy invariant . . . . .	50
<b>7</b>	<b>Non-injective Agreement (L0)</b>	<b>52</b>
7.1	Signals . . . . .	52
7.2	State and events . . . . .	52
7.3	Non injective agreement invariant . . . . .	53
<b>8</b>	<b>Injective Agreement (L0)</b>	<b>55</b>
8.1	State and events . . . . .	55
8.2	Injective agreement invariant . . . . .	56
8.3	Refinement . . . . .	56
8.4	Derived invariant . . . . .	57
<b>9</b>	<b>Runs</b>	<b>58</b>
9.1	Type definitions . . . . .	58
<b>10</b>	<b>Channel Messages</b>	<b>59</b>
10.1	Channel messages . . . . .	59
10.2	Extract . . . . .	59
10.3	Fake . . . . .	60
10.4	Closure of Dolev-Yao, extract and fake . . . . .	62
10.4.1	<i>dy-fake-msg</i> : returns messages, closure of DY and extr is sufficient . . . . .	62
10.4.2	<i>dy-fake-chan</i> : returns channel messages . . . . .	63
<b>11</b>	<b>Payloads and Support for Channel Message Implementations</b>	<b>66</b>
11.1	Payload messages . . . . .	66
11.2	<i>isLtKey</i> : is a long term key . . . . .	69
11.3	<i>keys-of</i> : the long term keys of an agent . . . . .	70
11.4	<i>Keys-bad</i> : bounds on the attacker’s knowledge of long-term keys. . . . .	71
11.5	<i>broken K</i> : pairs of agents where at least one is compromised. . . . .	72
11.6	<i>Enc-keys-clean S</i> : messages with “clean” symmetric encryptions. . . . .	73
11.7	Sets of messages with particular constructors . . . . .	73
11.7.1	Lemmas for moving message sets out of <i>analz</i> . . . . .	75
<b>12</b>	<b>Assumptions for Channel Message Implementation</b>	<b>81</b>
12.1	First step: basic implementation locale . . . . .	81
12.2	Second step: basic and analyze assumptions . . . . .	82
12.3	Third step: <i>valid-imlem</i> . . . . .	84
<b>13</b>	<b>Lemmas Following from Channel Message Implementation Assumptions</b>	<b>86</b>
13.1	Message implementations and abstractions . . . . .	86
13.2	Extractable messages . . . . .	87
13.2.1	Partition <i>I</i> to keep only the extractable messages . . . . .	88
13.2.2	Partition of <i>extractable</i> . . . . .	88

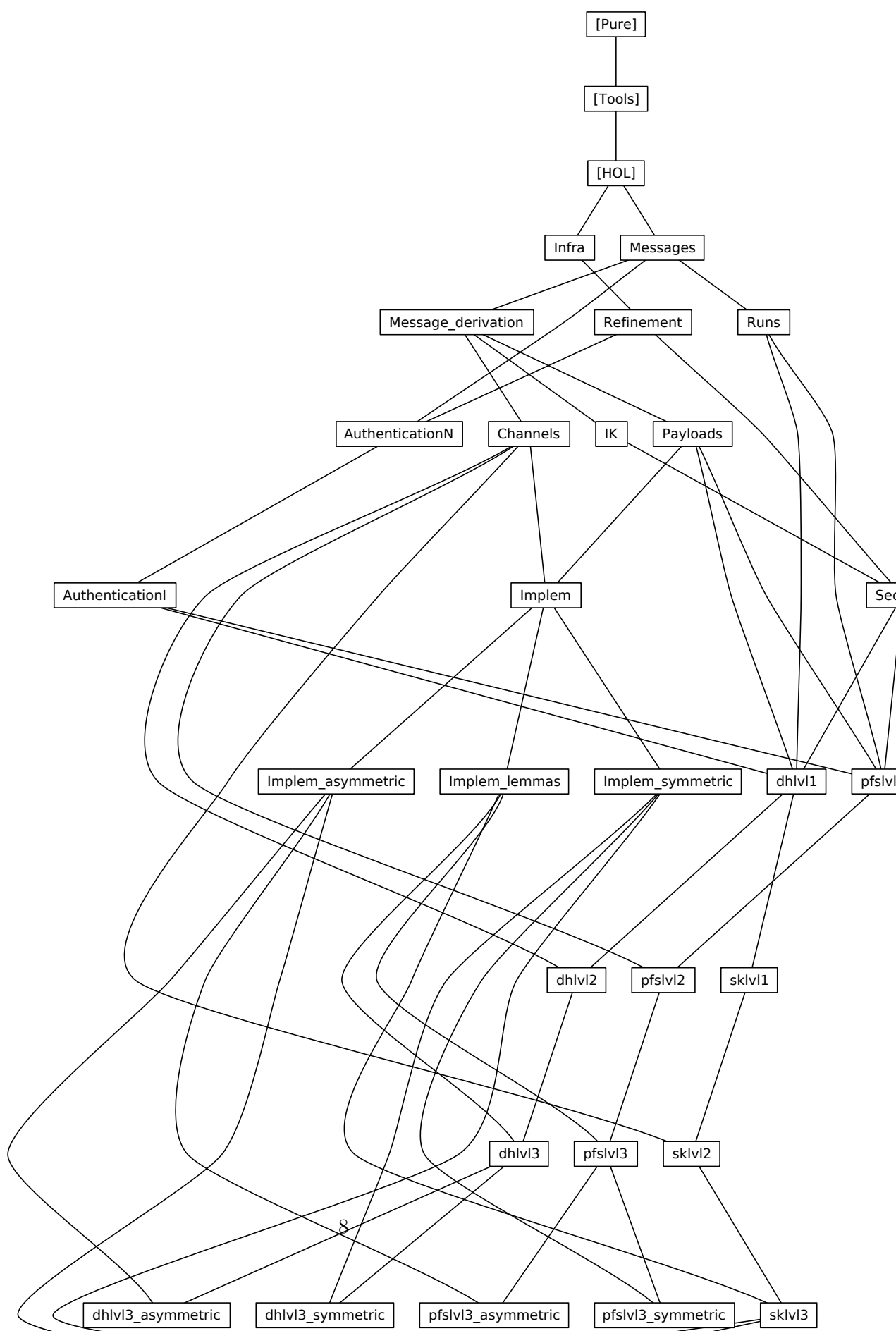
13.3	Lemmas for proving intruder refinement (L2-L3)	89
13.3.1	First: we only keep the extractable messages	89
13.3.2	Only keep the extracted messages (instead of extractable)	90
13.3.3	Keys and Tags can be moved out of the <i>analz</i>	92
13.3.4	Final lemmas, using all the previous ones	93
13.3.5	Partitioning <i>analz ik</i>	94
<b>14</b>	<b>Symmetric Implementation of Channel Messages</b>	<b>95</b>
14.1	Implementation of channel messages	95
14.2	Lemmas to pull implementation sets out of <i>analz</i>	96
14.2.1	Pull <i>implInsecSet</i> out of <i>analz</i>	96
14.3	Pull <i>implConfidSet</i> out of <i>analz</i>	98
14.4	Pull <i>implSecureSet</i> out of <i>analz</i>	99
14.5	Pull <i>implAuthSet</i> out of <i>analz</i>	100
14.6	Locale interpretations	102
<b>15</b>	<b>Asymmetric Implementation of Channel Messages</b>	<b>106</b>
15.1	Implementation of channel messages	106
15.2	Lemmas to pull implementation sets out of <i>analz</i>	107
15.2.1	Pull <i>PairAgentSet</i> out of <i>analz</i>	107
15.2.2	Pull <i>implInsecSet</i> out of <i>analz</i>	107
15.3	Pull <i>implConfidSet</i> out of <i>analz</i>	108
15.4	Pull <i>implAuthSet</i> out of <i>analz</i>	109
15.5	Pull <i>implSecureSet</i> out of <i>analz</i>	109
15.6	Locale interpretations	111
<b>16</b>	<b>Key Transport Protocol with PFS (L1)</b>	<b>115</b>
16.1	State and Events	115
16.2	Refinement: secrecy	120
16.3	Derived invariants: secrecy	121
16.4	Invariants	121
16.4.1	inv1	121
16.4.2	inv2	122
16.4.3	inv3 (derived)	123
16.5	Refinement: injective agreement	123
16.6	Derived invariants: injective agreement	125
<b>17</b>	<b>Key Transport Protocol with PFS (L2)</b>	<b>126</b>
17.1	State and Events	126
17.2	Invariants	131
17.2.1	inv1	131
17.2.2	inv2 (authentication guard)	132
17.2.3	inv3 (authentication guard)	133
17.2.4	inv4	134
17.2.5	inv5	134
17.2.6	inv6	135
17.2.7	inv7	135

17.2.8	inv8	139
17.3	Refinement	140
17.4	Derived invariants	142
<b>18</b>	<b>Key Transport Protocol with PFS (L3 locale)</b>	<b>144</b>
18.1	State and Events	144
18.2	Invariants	148
18.2.1	inv1: No long-term keys as message parts	148
18.2.2	inv2: <i>l3-state.bad s</i> indeed contains "bad" keys	148
18.2.3	inv3	149
18.2.4	inv4: the intruder knows the tags	150
18.2.5	inv5	151
18.2.6	inv6	151
18.2.7	inv7	152
18.2.8	inv8	153
18.2.9	inv9	154
18.3	Refinement	154
18.3.1	Protocol events	155
18.3.2	Intruder events	156
18.3.3	Compromise events	157
18.4	Derived invariants	158
18.4.1	inv10: secrets contain no implementation material	158
18.4.2	Partial secrecy	159
18.4.3	Secrecy	159
18.4.4	Injective agreement	159
<b>19</b>	<b>Key Transport Protocol with PFS (L3, asymmetric implementation)</b>	<b>161</b>
<b>20</b>	<b>Key Transport Protocol with PFS (L3, symmetric implementation)</b>	<b>162</b>
<b>21</b>	<b>Authenticated Diffie Hellman Protocol (L1)</b>	<b>163</b>
21.1	State and Events	163
21.2	Refinement: secrecy	168
21.3	Derived invariants: secrecy	170
21.4	Invariants: <i>Init</i> authenticates <i>Resp</i>	170
21.4.1	inv1	170
21.4.2	inv2	171
21.4.3	inv3 (derived)	171
21.5	Invariants: <i>Resp</i> authenticates <i>Init</i>	172
21.5.1	inv4	172
21.5.2	inv5	173
21.5.3	inv6 (derived)	173
21.6	Refinement: injective agreement ( <i>Init</i> authenticates <i>Resp</i> )	174
21.7	Derived invariants: injective agreement ( <i>Init</i> authenticates <i>Resp</i> )	176
21.8	Refinement: injective agreement ( <i>Resp</i> authenticates <i>Init</i> )	177
21.9	Derived invariants: injective agreement ( <i>Resp</i> authenticates <i>Init</i> )	178

<b>22 Authenticated Diffie-Hellman Protocol (L2)</b>	<b>180</b>
22.1 State and Events	180
22.2 Invariants	186
22.2.1 inv1	186
22.2.2 inv2 (authentication guard)	187
22.2.3 inv3 (authentication guard)	188
22.2.4 inv4	188
22.2.5 inv4'	189
22.2.6 inv5	190
22.2.7 inv6	190
22.2.8 inv7	191
22.2.9 inv8: form of the secrets	193
22.3 Refinement	195
22.4 Derived invariants	199
<b>23 Authenticated Diffie-Hellman Protocol (L3 locale)</b>	<b>201</b>
23.1 State and Events	201
23.2 Invariants	205
23.2.1 inv1: No long-term keys as message parts	205
23.2.2 inv2: <i>l3-state.bad s</i> indeed contains "bad" keys	206
23.2.3 inv3	207
23.2.4 inv4: the intruder knows the tags	208
23.2.5 inv5	209
23.2.6 inv6	209
23.2.7 inv7	210
23.2.8 inv8	211
23.2.9 inv9	211
23.3 Refinement	212
23.3.1 Protocol events	213
23.3.2 Intruder events	214
23.3.3 Compromise events	215
23.4 Derived invariants	216
23.4.1 inv10: secrets contain no implementation material	216
23.4.2 Partial secrecy	217
23.4.3 Secrecy	217
23.4.4 Injective agreement	218
<b>24 Authenticated Diffie-Hellman Protocol (L3, asymmetric)</b>	<b>219</b>
<b>25 Authenticated Diffie-Hellman Protocol (L3, symmetric)</b>	<b>220</b>
<b>26 SKEME Protocol (L1)</b>	<b>221</b>
26.1 State and Events	221
26.2 Refinement: secrecy	226
26.3 Derived invariants: secrecy	228
26.4 Invariants: <i>Init</i> authenticates <i>Resp</i>	229
26.4.1 inv1	229

26.4.2	inv2	229
26.4.3	inv3 (derived)	230
26.5	Invariants: Resp authenticates Init	231
26.5.1	inv4	231
26.5.2	inv5	231
26.5.3	inv6 (derived)	232
26.6	Refinement: injective agreement (Init authenticates Resp)	233
26.7	Derived invariants: injective agreement ( <i>Init</i> authenticates <i>Resp</i> )	235
26.8	Refinement: injective agreement ( <i>Resp</i> authenticates <i>Init</i> )	235
26.9	Derived invariants: injective agreement ( <i>Resp</i> authenticates <i>Init</i> )	237
<b>27</b>	<b>SKEME Protocol (L2)</b>	<b>238</b>
27.1	State and Events	238
27.2	Invariants	245
27.2.1	inv1	245
27.2.2	inv2	246
27.2.3	inv3	247
27.2.4	hmac preservation lemmas	250
27.2.5	inv4 (authentication guard)	252
27.2.6	inv5 (authentication guard)	254
27.2.7	inv6	257
27.2.8	inv6'	257
27.2.9	inv7: form of the secrets	258
27.3	Refinement	260
27.4	Derived invariants	264
<b>28</b>	<b>SKEME Protocol (L3 locale)</b>	<b>266</b>
28.1	State and Events	266
28.2	Invariants	271
28.2.1	inv1: No long-term keys as message parts	271
28.2.2	inv2: <i>l3-state.bad</i> <i>s</i> indeed contains "bad" keys	272
28.2.3	inv3	273
28.2.4	inv4: the intruder knows the tags	274
28.2.5	inv5	275
28.2.6	inv6	275
28.2.7	inv7	276
28.2.8	inv8	277
28.2.9	inv9	277
28.3	Refinement	278
28.3.1	Protocol events	279
28.3.2	Intruder events	280
28.3.3	Compromise events	281
28.4	Derived invariants	282
28.4.1	inv10: secrets contain no implementation material	282
28.4.2	Partial secrecy	283
28.4.3	Secrecy	283
28.4.4	Injective agreement	283

<b>29 SKEME Protocol (L3 with asymmetric implementation)</b>	<b>285</b>
<b>30 SKEME Protocol (L3 with symmetric implementation)</b>	<b>286</b>





# 1 Proving infrastructure

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

## 1.1 Prover configuration

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

## 1.2 Forward reasoning (“attributes”)

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

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

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

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

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

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

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

**lemmas** *rel-def-to-intro* = *setc-def-to-intro* [**where**  $x = (s, t)$  **for**  $s\ t$ ]

**lemmas** *rel-def-to-dest* = *setc-def-to-dest* [**where**  $x = (s, t)$  **for**  $s\ t$ ]

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

## 1.3 General results

### 1.3.1 Maps

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

**lemma** *map-definedness*:  
 $f\ x = \text{Some } y \implies x \in \text{dom } f$   
**by** (*simp add: domIff*)

**lemma** *map-definedness-contr*:  
 $\llbracket f\ x = \text{Some } y; z \notin \text{dom } f \rrbracket \implies x \neq z$   
**by** (*auto simp add: domIff*)

**lemmas** *dom-lemmas* = *map-definedness map-definedness-contr*

### 1.3.2 Set

**lemma** *vimage-image-subset*:  $A \subseteq f^{-1}(f \cdot A)$   
**by** (*auto simp add: image-def vimage-def*)

### 1.3.3 Relations

**lemma** *Image-compose* [*simp*]:  
  $(R1 \ O \ R2) \text{``}A = R2 \text{``}(R1 \text{``}A)$   
**by** (*auto*)

### 1.3.4 Lists

— Do NOT add the following equation to the simpset! (looping)

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

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

**lemma** *take-prefix*:  $\llbracket take \ n \ l = xs \rrbracket \implies \exists xs'. l = xs \ @ \ xs'$   
**by** (*induct l arbitrary: n xs, auto*)  
 (*rename-tac n, case-tac n, auto*)

### 1.3.5 Finite sets

Cardinality.

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

**lemma** *finite-positive-cardI* [*intro!*]:  
  $\llbracket A \neq \{\}; \text{finite } A \rrbracket \implies 0 < card \ A$   
**by** (*auto*)

**lemma** *finite-positive-cardD* [*dest!*]:  
  $\llbracket 0 < card \ A; \text{finite } A \rrbracket \implies A \neq \{\}$   
**by** (*auto*)

**lemma** *finite-zero-cardI* [*intro!*]:  
  $\llbracket A = \{\}; \text{finite } A \rrbracket \implies card \ A = 0$   
**by** (*auto*)

**lemma** *finite-zero-cardD* [*dest!*]:  
  $\llbracket card \ A = 0; \text{finite } A \rrbracket \implies A = \{\}$   
**by** (*auto*)

**end**

## 2 Models, Invariants and Refinements

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

### 2.1 Specifications, reachability, and behaviours.

Transition systems are multi-pointed graphs.

**record** *'s TS* =  
*init* :: *'s set*  
*trans* :: (*'s* × *'s*) *set*

The inductive set of reachable states.

**inductive-set**  
*reach* :: (*'s, 'a*) *TS-scheme* ⇒ *'s set*  
**for** *T* :: (*'s, 'a*) *TS-scheme*  
**where**  
*r-init* [*intro*]:  $s \in \text{init } T \implies s \in \text{reach } T$   
*r-trans* [*intro*]:  $\llbracket (s, t) \in \text{trans } T; s \in \text{reach } T \rrbracket \implies t \in \text{reach } T$

#### 2.1.1 Finite behaviours

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

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

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

Behaviours are prefix closed.

**lemma** *beh-immediate-prefix-closed*:  
 $s \# b \in \text{beh } T \implies b \in \text{beh } T$   
**by** (*erule beh-non-empty, auto*)

**lemma** *beh-prefix-closed*:  
 $c @ b \in \text{beh } T \implies b \in \text{beh } T$   
**by** (*induct c, auto dest!: beh-immediate-prefix-closed*)

States in behaviours are exactly reachable.

**lemma** *beh-in-reach* [*rule-format*]:  
 $b \in \text{beh } T \implies (\forall s \in \text{set } b. s \in \text{reach } T)$   
**by** (*erule beh.induct*) (*auto*)

**lemma** *reach-in-beh*:  
**assumes**  $s \in \text{reach } T$  **shows**  $\exists b \in \text{beh } T. s \in \text{set } b$   
**using** *assms*  
**proof** (*induction s rule: reach.induct*)

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

**lemma** *reach-equiv-beh-states*:  $\text{reach } T = \bigcup (\text{set}'(\text{beh } T))$   
**by** (*auto intro!: reach-in-beh beh-in-reach*)

## 2.1.2 Specifications, observability, and implementation

Specifications add an observer function to transition systems.

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

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

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

Observable behaviour and reachability.

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

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

**lemma** *oreach-equiv-obeh-states*:  
 $\text{oreach } S = \bigcup (\text{set}'(\text{obeh } S))$   
**by** (*auto simp add: reach-equiv-beh-states oreach-def obeh-def*)

**lemma** *obeh-pi-translation*:  
 $(\text{map } \text{pi})'(\text{obeh } S) = \text{obeh } (S(| \text{obs} := \text{pi } o (\text{obs } S) |))$   
**by** (*auto simp add: obeh-def image-comp*)

**lemma** *oreach-pi-translation*:  
 $\text{pi}'(\text{oreach } S) = \text{oreach } (S(| \text{obs} := \text{pi } o (\text{obs } S) |))$   
**by** (*auto simp add: oreach-def*)

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

**definition**

$observable :: ['s \Rightarrow 'o, 's\ set] \Rightarrow bool$

**where**

$observable\ ob\ P \equiv \forall s\ s'.\ ob\ s = ob\ s' \longrightarrow s' \in P \longrightarrow s \in P$

**definition**

$observable2 :: ['s \Rightarrow 'o, 's\ set] \Rightarrow bool$

**where**

$observable2\ ob\ P \equiv \exists Q.\ P = ob-'Q$

**definition**

$observable3 :: ['s \Rightarrow 'o, 's\ set] \Rightarrow bool$

**where**

$observable3\ ob\ P \equiv ob-'ob'P \subseteq P$  — other direction holds trivially

**lemma** *observableE* [elim]:

$\llbracket observable\ ob\ P; ob\ s = ob\ s'; s' \in P \rrbracket \Longrightarrow s \in P$

**by** (*unfold observable-def*) (*fast*)

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

**by** (*unfold observable-def observable2-def*) (*auto*)

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

**by** (*unfold observable3-def observable2-def*) (*auto*)

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

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

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

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

**by** *auto*

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

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

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

**lemma** *image-r-vimage-l*:  $\llbracket Q \subseteq ob'P; observable\ ob\ P \rrbracket \Longrightarrow ob-'Q \subseteq P$ 

**by** (*auto*)

**lemma** *vimage-l-image-r*:  $\llbracket ob-'Q \subseteq P; Q \subseteq range\ ob \rrbracket \Longrightarrow Q \subseteq ob'P$ 

**by** (*drule image-mono [where f=ob], auto*)

Internal and external invariants

**lemma** *external-from-internal-invariant*:

$\llbracket reach\ S \subseteq P; (obs\ S)'P \subseteq Q \rrbracket$

$\Longrightarrow oreach\ S \subseteq Q$

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

**lemma** *external-from-internal-invariant-vimage*:

$\llbracket reach\ S \subseteq P; P \subseteq (obs\ S)-'Q \rrbracket$

$\implies \text{oreach } S \subseteq Q$   
**by** (*erule external-from-internal-invariant*) (*auto*)

**lemma** *external-to-internal-invariant-vimage*:

$\llbracket \text{oreach } S \subseteq Q; (\text{obs } S) \text{--}'Q \subseteq P \rrbracket$   
 $\implies \text{reach } S \subseteq P$

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

**lemma** *external-to-internal-invariant*:

$\llbracket \text{oreach } S \subseteq Q; Q \subseteq (\text{obs } S) \text{'P}; \text{observable } (\text{obs } S) P \rrbracket$   
 $\implies \text{reach } S \subseteq P$

**by** (*erule external-to-internal-invariant-vimage*) (*auto*)

**lemma** *external-equiv-internal-invariant-vimage*:

$\llbracket P = (\text{obs } S) \text{--}'Q \rrbracket$   
 $\implies (\text{oreach } S \subseteq Q) = (\text{reach } S \subseteq P)$

**by** (*fast intro: external-from-internal-invariant-vimage*  
*external-to-internal-invariant-vimage*  
*del: subsetI*)

**lemma** *external-equiv-internal-invariant*:

$\llbracket (\text{obs } S) \text{'P} = Q; \text{observable } (\text{obs } S) P \rrbracket$   
 $\implies (\text{oreach } S \subseteq Q) = (\text{reach } S \subseteq P)$

**by** (*rule external-equiv-internal-invariant-vimage*) (*auto*)

Our notion of implementation is inclusion of observable behaviours.

**definition**

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

Reflexivity and transitivity

**lemma** *implements-refl*: *implements id S S*

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

**lemma** *implements-trans*:

$\llbracket \text{implements } \text{pi1 } S1 S2; \text{implements } \text{pi2 } S2 S3 \rrbracket$   
 $\implies \text{implements } (\text{pi1 } \circ \text{pi2}) S1 S3$

**by** (*fastforce simp add: implements-def image-subset-iff*)

Preservation of external invariants

**lemma** *implements-oreach*:

*implements pi Sa Sc*  $\implies \text{pi}'(\text{oreach } Sc) \subseteq \text{oreach } Sa$

**by** (*auto simp add: implements-def oreach-equiv-obeh-states dest!: subsetD*)

**lemma** *external-invariant-preservation*:

$\llbracket \text{oreach } Sa \subseteq Q; \text{implements } \text{pi } Sa Sc \rrbracket$   
 $\implies \text{pi}'(\text{oreach } Sc) \subseteq Q$

**by** (*rule subset-trans [OF implements-oreach]*) (*auto*)

**lemma** *external-invariant-translation*:

$\llbracket \text{oreach } Sa \subseteq Q; pi - 'Q \subseteq P; \text{implements } pi \text{ } Sa \text{ } Sc \rrbracket$   
 $\implies \text{oreach } Sc \subseteq P$   
**apply** (rule subset-trans [OF vimage-image-subset, of pi])  
**apply** (rule subset-trans [where B=pi-'Q])  
**apply** (intro vimage-mono external-invariant-preservation, auto)  
**done**

Preservation of internal invariants

**lemma** *internal-invariant-translation*:  
 $\llbracket \text{reach } Sa \subseteq Pa; Pa \subseteq \text{obs } Sa - 'Qa; pi - 'Qa \subseteq Q; \text{obs } S - 'Q \subseteq P; \text{implements } pi \text{ } Sa \text{ } S \rrbracket$   
 $\implies \text{reach } S \subseteq P$   
**by** (rule external-from-internal-invariant-vimage [ THEN external-invariant-translation, THEN external-to-internal-invariant-vimage])

## 2.2 Invariants

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

### 2.2.1 Hoare triples

**definition**

$PO\text{-hoare} :: [ 's \text{ set}, ( 's \times 's) \text{ set}, 's \text{ set}] \Rightarrow \text{bool}$   
 $((\exists \{-\} - \{> -\}) [0, 0, 0] 90)$

**where**

$\{pre\} R \{> post\} \equiv R 'pre \subseteq post$

**lemmas**  $PO\text{-hoare-defs} = PO\text{-hoare-def Image-def}$

**lemma**  $\{P\} R \{> Q\} = (\forall s t. s \in P \longrightarrow (s, t) \in R \longrightarrow t \in Q)$   
**by** (auto simp add: PO-hoare-defs)

Some essential facts about Hoare triples.

**lemma** *hoare-conseq-left* [intro]:

$\llbracket \{P'\} R \{> Q\}; P \subseteq P' \rrbracket$   
 $\implies \{P\} R \{> Q\}$

**by** (auto simp add: PO-hoare-defs)

**lemma** *hoare-conseq-right*:

$\llbracket \{P\} R \{> Q'\}; Q' \subseteq Q \rrbracket$   
 $\implies \{P\} R \{> Q\}$

**by** (auto simp add: PO-hoare-defs)

**lemma** *hoare-false-left* [simp]:

$\{\{\}\} R \{> Q\}$

**by** (auto simp add: PO-hoare-defs)

**lemma** *hoare-true-right* [simp]:

$\{P\} R \{> UNIV\}$

**by** (auto simp add: PO-hoare-defs)

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

Special transition relations.

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

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

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

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

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

### 2.2.2 Characterization of reachability

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

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

Useful consequences.

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

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

### 2.2.3 Invariant proof rules

Basic proof rule for invariants.

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

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

**lemma** *inv-rule*:



```

[[ init T ⊆ I; I ⊆ P; {I} (trans T) {> I} ]]
⇒ reach T ⊆ P
apply (rule subset-trans, auto) — strengthen goal
apply (erule reach.induct, auto simp add: PO-hoare-def)
done

```

The following rule is equivalent to the previous one.

```

lemma INV-rule:
[[ init T ⊆ I; {I ∩ reach T} (trans T) {> I} ]]
⇒ reach T ⊆ I
by (safe, erule reach.induct, auto simp add: PO-hoare-defs)

```

Proof of equivalence.

```

lemma inv-rule-from-INV-rule:
[[ init T ⊆ I; I ⊆ P; {I} (trans T) {> I} ]]
⇒ reach T ⊆ P
apply (rule subset-trans, auto del: subsetI)
apply (rule INV-rule, auto)
done

```

```

lemma INV-rule-from-inv-rule:
[[ init T ⊆ I; {I ∩ reach T} (trans T) {> I} ]]
⇒ reach T ⊆ I
by (rule-tac I=I ∩ reach T in inv-rule, auto)

```

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

```

lemma inv-rule-incr:
[[ init T ⊆ I; {I ∩ J} (trans T) {> I}; reach T ⊆ J ]]
⇒ reach T ⊆ I
by (rule INV-rule, auto)

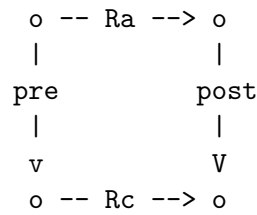
```

## 2.3 Refinement

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

### 2.3.1 Relational Hoare tuples

Relational Hoare tuples formalize the following generalized simulation diagram:



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

**definition**

$PO\text{-rhoare} ::$   
 $[(\text{'s} \times \text{'t}) \text{ set}, (\text{'s} \times \text{'s}) \text{ set}, (\text{'t} \times \text{'t}) \text{ set}, (\text{'s} \times \text{'t}) \text{ set}] \Rightarrow \text{bool}$   
 $((\{ \{ - \} -, - \{ > - \} \} [0, 0, 0] 90)$

**where**

$\{pre\} Ra, Rc \{> post\} \equiv pre O Rc \subseteq Ra O post$

**lemmas**  $PO\text{-rhoare-defs} = PO\text{-rhoare-def relcomp-unfold}$

Facts about relational Hoare tuples.

**lemma**  $relhoare\text{-conseq-left}$  [intro]:

$\llbracket \{pre'\} Ra, Rc \{> post'\}; pre \subseteq pre' \rrbracket$   
 $\implies \{pre\} Ra, Rc \{> post\}$

**by** (auto simp add:  $PO\text{-rhoare-defs}$  dest!: subsetD)

**lemma**  $relhoare\text{-conseq-right}$ :

— do NOT declare [intro]

$\llbracket \{pre\} Ra, Rc \{> post'\}; post' \subseteq post \rrbracket$   
 $\implies \{pre\} Ra, Rc \{> post\}$

**by** (auto simp add:  $PO\text{-rhoare-defs}$ )

**lemma**  $relhoare\text{-false-left}$  [simp]:

— do NOT declare [intro]

$\{ \{ \} \} Ra, Rc \{> post\}$

**by** (auto simp add:  $PO\text{-rhoare-defs}$ )

**lemma**  $relhoare\text{-true-right}$  [simp]:

— not true in general

$\{pre\} Ra, Rc \{> UNIV\} = (\text{Domain } (pre O Rc) \subseteq \text{Domain } Ra)$

**by** (auto simp add:  $PO\text{-rhoare-defs}$ )

**lemma**  $\text{Domain-rel-comp}$  [intro]:

$\text{Domain } pre \subseteq R \implies \text{Domain } (pre O Rc) \subseteq R$

**by** (auto simp add:  $\text{Domain-def}$ )

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

**by** (auto simp add:  $PO\text{-rhoare-def}$ )

Reflexivity and transitivity.

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

**by** (auto simp add:  $PO\text{-rhoare-defs}$ )

**lemma**  $rhoare\text{-trans}$ :

$\llbracket \{R1\} T1, T2 \{> R1\}; \{R2\} T2, T3 \{> R2\} \rrbracket$   
 $\implies \{R1 O R2\} T1, T3 \{> R1 O R2\}$

**apply** (auto simp add:  $PO\text{-rhoare-def del: subsetI}$ )

**apply** (drule subset-refl [THEN relcomp-mono, **where**  $r=R1$ ])

**apply** (drule subset-refl [THEN [2] relcomp-mono, **where**  $s=R2$ ])

**apply** (auto simp add:  $O\text{-assoc del: subsetI}$ )

**done**

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

**lemma** *relhoare-conj-right-det*:

$$\begin{aligned} & \llbracket \{pre\} Ra, Rc \{> post1\}; \{pre\} Ra, Rc \{> post2\}; \\ & \quad \text{single-valued } Ra \rrbracket \quad \text{— only for deterministic } Ra! \\ \implies & \{pre\} Ra, Rc \{> post1 \cap post2\} \end{aligned}$$

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

**lemma** *relhoare-conj-right-cartesian* [*intro*]:

$$\begin{aligned} & \llbracket \{Domain\ pre\} Ra \{> I\}; \{Range\ pre\} Rc \{> J\}; \\ & \quad \{pre\} Ra, Rc \{> post\} \rrbracket \\ \implies & \{pre\} Ra, Rc \{> post \cap I \times J\} \end{aligned}$$

**by** (*force simp add: PO-rhoare-defs PO-hoare-defs Domain-def Range-def*)

Separate rule for cartesian products.

**corollary** *relhoare-cartesian*:

$$\begin{aligned} & \llbracket \{Domain\ pre\} Ra \{> I\}; \{Range\ pre\} Rc \{> J\}; \\ & \quad \{pre\} Ra, Rc \{> post\} \rrbracket \quad \text{— any } post, \text{ including } UNIV! \\ \implies & \{pre\} Ra, Rc \{> I \times J\} \end{aligned}$$

**by** (*auto intro: relhoare-conseq-right*)

Unions of transition relations.

**lemma** *relhoare-concrete-Un* [*simp*]:

$$\begin{aligned} & \{pre\} Ra, Rc1 \cup Rc2 \{> post\} \\ & = (\{pre\} Ra, Rc1 \{> post\} \wedge \{pre\} Ra, Rc2 \{> post\}) \end{aligned}$$

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

**apply** (*auto dest!: subsetD*)

**done**

**lemma** *relhoare-concrete-UN* [*simp*]:

$$\{pre\} Ra, \bigcup x. Rc\ x \{> post\} = (\forall x. \{pre\} Ra, Rc\ x \{> post\})$$

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

**apply** (*auto dest!: subsetD*)

**done**

**lemma** *relhoare-abstract-Un-left* [*intro*]:

$$\begin{aligned} & \llbracket \{pre\} Ra1, Rc \{> post\} \rrbracket \\ \implies & \{pre\} Ra1 \cup Ra2, Rc \{> post\} \end{aligned}$$

**by** (*auto simp add: PO-rhoare-defs*)

**lemma** *relhoare-abstract-Un-right* [*intro*]:

$$\begin{aligned} & \llbracket \{pre\} Ra2, Rc \{> post\} \rrbracket \\ \implies & \{pre\} Ra1 \cup Ra2, Rc \{> post\} \end{aligned}$$

**by** (*auto simp add: PO-rhoare-defs*)

**lemma** *relhoare-abstract-UN* [*intro*]: — ! might be too aggressive? INDEED.

$$\begin{aligned} & \llbracket \{pre\} Ra\ x, Rc \{> post\} \rrbracket \\ \implies & \{pre\} \bigcup x. Ra\ x, Rc \{> post\} \end{aligned}$$

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

**apply** (*auto dest!: subsetD*)

**done**

Inclusion of abstract transition relations.

**lemma** *relhoare-abstract-trans-weak* [intro]:

$\llbracket \{pre\} Ra', Rc \{> post\}; Ra' \subseteq Ra \rrbracket$

$\implies \{pre\} Ra, Rc \{> post\}$

**by** (*auto simp add:PO-rhoare-defs*)

### 2.3.2 Refinement proof obligations

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

**definition**

*PO-refines* ::

$[( 's \times 't) \text{ set}, ('s, 'a) \text{ TS-scheme}, ('t, 'b) \text{ TS-scheme}] \Rightarrow \text{bool}$

**where**

$PO\text{-refines } R \text{ } Ta \text{ } Tc \equiv ($   
 $\quad \text{init } Tc \subseteq R''(\text{init } Ta)$   
 $\quad \wedge \{R\} (\text{trans } Ta), (\text{trans } Tc) \{> R\}$   
 $\quad )$

**lemma** *PO-refinesI*:

$\llbracket \text{init } Tc \subseteq R''(\text{init } Ta); \{R\} (\text{trans } Ta), (\text{trans } Tc) \{> R\} \rrbracket \implies PO\text{-refines } R \text{ } Ta \text{ } Tc$

**by** (*simp add: PO-refines-def*)

**lemma** *PO-refinesE* [elim]:

$\llbracket PO\text{-refines } R \text{ } Ta \text{ } Tc; \llbracket \text{init } Tc \subseteq R''(\text{init } Ta); \{R\} (\text{trans } Ta), (\text{trans } Tc) \{> R\} \rrbracket \implies P \rrbracket$   
 $\implies P$

**by** (*simp add: PO-refines-def*)

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

**lemma** *refine-basic*:

$\llbracket \text{init } Tc \subseteq R''(\text{init } Ta); \{R\} (\text{trans } Ta), (\text{trans } Tc) \{> R\} \rrbracket$

$\implies PO\text{-refines } R \text{ } Ta \text{ } Tc$

**by** (*simp add: PO-refines-def*)

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

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

**lemma** *refine-init-using-invariants*:

$\llbracket \text{init } Tc \subseteq R''(\text{init } Ta); \text{init } Ta \subseteq I; \text{init } Tc \subseteq J \rrbracket$

$\implies \text{init } Tc \subseteq (R \cap I \times J)''(\text{init } Ta)$

**by** (*auto simp add: Image-def dest!: bspec subsetD*)

**lemma** *refine-trans-using-invariants*:

$\llbracket \{R \cap I \times J\} (\text{trans } Ta), (\text{trans } Tc) \{> R\};$

$$\begin{aligned} & \{I \cap \text{Domain } (R \cap \text{UNIV} \times J)\} (\text{trans } Ta) \{> I\}; \\ & \{J \cap \text{Range } (R \cap I \times \text{UNIV})\} (\text{trans } Tc) \{> J\} \parallel \\ \implies & \{R \cap I \times J\} (\text{trans } Ta), (\text{trans } Tc) \{> R \cap I \times J\} \\ \text{by } & (\text{rule relhoare-conj-right-cartesian}) (\text{auto}) \end{aligned}$$

This is our main rule for refinements.

**lemma** *refine-using-invariants*:

$$\begin{aligned} & \parallel \{R \cap I \times J\} (\text{trans } Ta), (\text{trans } Tc) \{> R\}; \\ & \{I \cap \text{Domain } (R \cap \text{UNIV} \times J)\} (\text{trans } Ta) \{> I\}; \\ & \{J \cap \text{Range } (R \cap I \times \text{UNIV})\} (\text{trans } Tc) \{> J\}; \\ & \text{init } Tc \subseteq R^{\text{c}}(\text{init } Ta); \\ & \text{init } Ta \subseteq I; \text{init } Tc \subseteq J \parallel \\ \implies & \text{PO-refines } (R \cap I \times J) \text{ } Ta \text{ } Tc \\ \text{by } & (\text{unfold PO-refines-def}) \\ & (\text{intro refine-init-using-invariants refine-trans-using-invariants conjI}) \end{aligned}$$

### 2.3.3 Deriving invariants from refinements

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

**lemma** *PO-refines-implies-Range-init*:

$$\text{PO-refines } R \text{ } Ta \text{ } Tc \implies \text{init } Tc \subseteq \text{Range } R$$

**by** (*auto simp add: PO-refines-def*)

**lemma** *PO-refines-implies-Range-trans*:

$$\text{PO-refines } R \text{ } Ta \text{ } Tc \implies \{\text{Range } R\} \text{trans } Tc \{> \text{Range } R\}$$

**by** (*auto simp add: PO-refines-def PO-rhoare-def PO-hoare-def*)

**lemma** *PO-refines-implies-Range-invariant*:

$$\text{PO-refines } R \text{ } Ta \text{ } Tc \implies \text{reach } Tc \subseteq \text{Range } R$$

**by** (*rule INV-rule*)  
*(auto intro!: PO-refines-implies-Range-init*  
*PO-refines-implies-Range-trans)*

The following rules are more useful in proofs.

**corollary** *INV-init-from-refinement*:

$$\begin{aligned} & \parallel \text{PO-refines } R \text{ } Ta \text{ } Tc; \text{Range } R \subseteq I \parallel \\ \implies & \text{init } Tc \subseteq I \\ \text{by } & (\text{drule PO-refines-implies-Range-init}, \text{ auto}) \end{aligned}$$

**corollary** *INV-trans-from-refinement*:

$$\begin{aligned} & \parallel \text{PO-refines } R \text{ } Ta \text{ } Tc; K \subseteq \text{Range } R; \text{Range } R \subseteq I \parallel \\ \implies & \{K\} \text{trans } Tc \{> I\} \\ \text{apply } & (\text{drule PO-refines-implies-Range-trans}) \\ \text{apply } & (\text{auto intro: hoare-conseq-right}) \\ \text{done} & \end{aligned}$$

**corollary** *INV-from-refinement*:

$$\begin{aligned} & \parallel \text{PO-refines } R \text{ } Ta \text{ } Tc; \text{Range } R \subseteq I \parallel \\ \implies & \text{reach } Tc \subseteq I \\ \text{by } & (\text{drule PO-refines-implies-Range-invariant}, \text{ fast}) \end{aligned}$$

### 2.3.4 Refinement of specifications

Lift relation membership to finite sequences

**inductive-set**

*seq-lift* :: ('s × 't) set ⇒ ('s list × 't list) set  
**for** *R* :: ('s × 't) set

**where**

*sl-nil* [iff]: ([], []) ∈ *seq-lift R*  
| *sl-cons* [intro]:  
[[ (xs, ys) ∈ *seq-lift R*; (x, y) ∈ *R* ]] ⇒ (x#xs, y#ys) ∈ *seq-lift R*

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

For each concrete behaviour there is a related abstract one.

**lemma** *behaviour-refinement*:

[[ *PO-refines R Ta Tc*; bc ∈ *beh Tc* ]]  
⇒ ∃ ba ∈ *beh Ta*. (ba, bc) ∈ *seq-lift R*

**apply** (*erule beh.induct*, *auto*)

— case: singleton

**apply** (*clarsimp simp add: PO-refines-def Image-def*)

**apply** (*drule subsetD*, *auto*)

— case: cons; first construct related abstract state

**apply** (*erule sl-cons-right-invert*, *clarsimp*)

**apply** (*rename-tac s bc s' ba t*)

— now construct abstract transition

**apply** (*auto simp add: PO-refines-def PO-rhoare-def*)

**apply** (*thin-tac X ⊆ Y for X Y*)

**apply** (*drule subsetD*, *auto*)

**done**

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

**definition**

*obs-consistent* ::  
[(('s × 't) set, 'p ⇒ 'o, ('s, 'o) spec, ('t, 'p) spec)] ⇒ bool

**where**

*obs-consistent R pi Sa Sc* ≡ (∀ s t. (s, t) ∈ *R* → *pi* (*obs Sc t*) = *obs Sa s*)

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

**by** (*simp add: obs-consistent-def*)

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

[[ *obs-consistent R1 pi1 S1 S2*; *obs-consistent R2 pi2 S2 S3* ]]  
⇒ *obs-consistent (R1 O R2) (pi1 o pi2) S1 S3*

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

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

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

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

$obs\text{-consistent } R \text{ pi } Sa \text{ Sc} \implies obs\text{-consistent } (R \cap R') \text{ pi } Sa \text{ Sc}$   
**by** (*auto simp add: obs-consistent-def*)

**lemma** *obs-consistent-conj2* [*intro*]:  
 $obs\text{-consistent } R \text{ pi } Sa \text{ Sc} \implies obs\text{-consistent } (R' \cap R) \text{ pi } Sa \text{ Sc}$   
**by** (*auto simp add: obs-consistent-def*)

**lemma** *obs-consistent-behaviours*:  
 $\llbracket obs\text{-consistent } R \text{ pi } Sa \text{ Sc}; bc \in beh \text{ Sc}; ba \in beh \text{ Sa}; (ba, bc) \in seq\text{-lift } R \rrbracket$   
 $\implies map \text{ pi } (map (obs \text{ Sc}) bc) = map (obs \text{ Sa}) ba$   
**by** (*erule seq-lift.induct*) (*auto simp add: obs-consistent-def*)

Definition of refinement proof obligations.

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

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

**lemma** *refinesI*:  
 $\llbracket PO\text{-refines } R \text{ Sa Sc}; obs\text{-consistent } R \text{ pi Sa Sc} \rrbracket$   
 $\implies refines \text{ R pi Sa Sc}$   
**by** (*simp add: refines-def*)

**lemma** *refinesE* [*elim*]:  
 $\llbracket refines \text{ R pi Sa Sc}; \llbracket PO\text{-refines } R \text{ Sa Sc}; obs\text{-consistent } R \text{ pi Sa Sc} \rrbracket \implies P \rrbracket$   
 $\implies P$   
**by** (*simp add: refines-def*)

Reflexivity and transitivity of refinement.

**lemma** *refinement-reflexive*:  $refines \text{ Id id } S \text{ S}$   
**by** (*auto simp add: refines-defs*)

**lemma** *refinement-transitive*:  
 $\llbracket refines \text{ R1 pi1 S1 S2}; refines \text{ R2 pi2 S2 S3} \rrbracket$   
 $\implies refines (\text{R1 } O \text{ R2}) (\text{pi1 } o \text{ pi2}) \text{ S1 S3}$   
**apply** (*auto simp add: refines-defs del: subsetI*  
*intro: rhoare-trans*)  
**apply** (*fastforce dest: Image-mono*)  
**done**

Soundness of refinement for proving implementation

**lemma** *observable-behaviour-refinement*:  
 $\llbracket refines \text{ R pi Sa Sc}; bc \in obeh \text{ Sc} \rrbracket \implies map \text{ pi } bc \in obeh \text{ Sa}$   
**by** (*auto simp add: refines-def obeh-def image-def*  
*dest!: behaviour-refinement obs-consistent-behaviours*)

**theorem** *refinement-soundness*:  
 $refines \text{ R pi Sa Sc} \implies implements \text{ pi Sa Sc}$

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

Extended versions of refinement proof rules including observations

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

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

**lemma** *refines-reachable-strengthening:*

*refines R pi Sa Sc  $\implies$  refines (R  $\cap$  reach Sa  $\times$  reach Sc) pi Sa Sc*

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

Inheritance of internal invariants through refinements

**lemma** *INV-init-from-Refinement:*

$\llbracket \text{refines } R \text{ pi Sa Sc; Range } R \subseteq I \rrbracket \implies \text{init Sc} \subseteq I$

**by** (*blast intro: INV-init-from-refinement*)

**lemma** *INV-trans-from-Refinement:*

$\llbracket \text{refines } R \text{ pi Sa Sc; } K \subseteq \text{Range } R; \text{Range } R \subseteq I \rrbracket \implies \{K\} \text{TS.trans Sc } \{> I\}$

**by** (*blast intro: INV-trans-from-refinement*)

**lemma** *INV-from-Refinement-basic:*

$\llbracket \text{refines } R \text{ pi Sa Sc; Range } R \subseteq I \rrbracket \implies \text{reach Sc} \subseteq I$

**by** (*rule INV-from-refinement*) *blast*

**lemma** *INV-from-Refinement-using-invariants:*

**assumes** *refines R pi Sa Sc Range (R  $\cap$  I  $\times$  J)  $\subseteq$  K*

*reach Sa  $\subseteq$  I reach Sc  $\subseteq$  J*

**shows** *reach Sc  $\subseteq$  K*

**proof** (*rule INV-from-Refinement-basic*)

**show** *refines (R  $\cap$  reach Sa  $\times$  reach Sc) pi Sa Sc using *assms(1)**

**by** (*rule refines-reachable-strengthening*)

**next**

**show** *Range (R  $\cap$  reach Sa  $\times$  reach Sc)  $\subseteq$  K using *assms(2-4)* by *blast**

**qed**

**end**



### 3 Message definitions

```
theory Messages
imports Main
begin
```

#### 3.1 Messages

Agents

```
datatype
  agent = Agent nat
```

Nonces

```
typedecl fid-t
```

```
datatype fresh-t =
  mk-fresh fid-t nat (infixr $ 65)
```

```
fun fid :: fresh-t  $\Rightarrow$  fid-t where
  fid (f $ n) = f
```

```
fun num :: fresh-t  $\Rightarrow$  nat where
  num (f $ n) = n
```

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

Keys

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

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

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

Messages

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

```

| cHash cmsg
| cTag tag
| cExp cmsg cmsg

```

**fun** *catomic* :: *cmsg* ⇒ *bool*

**where**

```

  catomic (cAgent -) = True
| catomic (cNumber -) = True
| catomic (cNonce -) = True
| catomic (cLtK -) = True
| catomic (cEphK -) = True
| catomic (cTag -) = True
| catomic - = False

```

**inductive** *eq* :: *cmsg* ⇒ *cmsg* ⇒ *bool*

**where**

— equations

```

  Permute [intro]:eq (cExp (cExp a b) c) (cExp (cExp a c) b)

```

— closure by context

```

| Tag[intro]: eq (cTag t) (cTag t)
| Agent[intro]: eq (cAgent A) (cAgent A)
| Nonce[intro]:eq (cNonce x) (cNonce x)
| Number[intro]:eq (cNumber x) (cNumber x)
| LtK[intro]:eq (cLtK x) (cLtK x)
| EphK[intro]:eq (cEphK x) (cEphK x)
| Pair[intro]:eq a b ⇒ eq c d ⇒ eq (cPair a c) (cPair b d)
| Enc[intro]:eq a b ⇒ eq c d ⇒ eq (cEnc a c) (cEnc b d)
| Aenc[intro]:eq a b ⇒ eq c d ⇒ eq (cAenc a c) (cAenc b d)
| Sign[intro]:eq a b ⇒ eq c d ⇒ eq (cSign a c) (cSign b d)
| Hash[intro]:eq a b ⇒ eq (cHash a) (cHash b)
| Exp[intro]:eq a b ⇒ eq c d ⇒ eq (cExp a c) (cExp b d)

```

— reflexive closure is not needed here because the context closure implies it

— symmetric closure is not needed as it is easier to include equations in both directions

— transitive closure

```

| Tr[intro]: eq a b ⇒ eq b c ⇒ eq a c

```

**lemma** *eq-sym*:  $eq\ a\ b \longleftrightarrow eq\ b\ a$

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

**lemma** *eq-Sym* [intro]:  $eq\ a\ b \implies eq\ b\ a$

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

**lemma** *eq-refl* [simp, intro]:  $eq\ a\ a$

**by** (*induction a, auto*)

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

**lemma** *eq-Number*:  $eq\ (cNumber\ N)\ y \implies y = cNumber\ N$

**by** (*induction cNumber N y rule: eq.induct, auto*)

**lemma** *eq-Agent*:  $eq\ (cAgent\ A)\ y \implies y = cAgent\ A$

**by** (*induction cAgent A y rule: eq.induct, auto*)

**lemma** *eq-Nonce*:  $eq\ (cNonce\ N)\ y \implies y = cNonce\ N$

**by** (*induction cNonce N y rule: eq.induct, auto*)

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

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

Quotient construction

**quotient-type** *msg* = *cmsg* / *eq*  
**morphisms** *Re Ab*  
**by** (*auto simp add:equivp-def*)

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

**lift-definition**  $Exp :: msg \Rightarrow msg \Rightarrow msg$  **is**  $cExp$  **by** *auto*

**lift-definition**  $Tag :: tag \Rightarrow msg$  **is**  $cTag$  **by**  $-$

**lift-definition**  $Hash :: msg \Rightarrow msg$  **is**  $cHash$  **by** *auto*

**lift-definition**  $Sign :: msg \Rightarrow msg \Rightarrow msg$  **is**  $cSign$  **by** *auto*

**lemmas**  $msg-defs =$

*Agent-def Number-def Nonce-def LtK-def EphK-def Pair-def*  
*Enc-def Aenc-def Exp-def Hash-def Tag-def Sign-def*

Commutativity of exponents

**lemma**  $permute-exp$  [*simp*]:  $Exp (Exp X Y) Z = Exp (Exp X Z) Y$   
**by** (*transfer, auto*)

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

**abbreviation**

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

**lemma**  $atomic-Agent$  [*simp, intro*]:  $atomic (Agent X)$  **by** (*transfer, auto*)

**lemma**  $atomic-Tag$  [*simp, intro*]:  $atomic (Tag X)$  **by** (*transfer, auto*)

**lemma**  $atomic-Nonce$  [*simp, intro*]:  $atomic (Nonce X)$  **by** (*transfer, auto*)

**lemma**  $atomic-Number$  [*simp, intro*]:  $atomic (Number X)$  **by** (*transfer, auto*)

**lemma**  $atomic-LtK$  [*simp, intro*]:  $atomic (LtK X)$  **by** (*transfer, auto*)

**lemma**  $atomic-EphK$  [*simp, intro*]:  $atomic (EphK X)$  **by** (*transfer, auto*)

**lemma**  $non-atomic-Pair$  [*simp*]:  $\neg atomic (Pair x y)$  **by** (*transfer, auto*)

**lemma**  $non-atomic-Enc$  [*simp*]:  $\neg atomic (Enc x y)$  **by** (*transfer, auto*)

**lemma**  $non-atomic-Aenc$  [*simp*]:  $\neg atomic (Aenc x y)$  **by** (*transfer, auto*)

**lemma**  $non-atomic-Sign$  [*simp*]:  $\neg atomic (Sign x y)$  **by** (*transfer, auto*)

**lemma**  $non-atomic-Exp$  [*simp*]:  $\neg atomic (Exp x y)$  **by** (*transfer, auto*)

**lemma**  $non-atomic-Hash$  [*simp*]:  $\neg atomic (Hash x)$  **by** (*transfer, auto*)

**lemma**  $Nonce-Nonce$ :  $(Nonce X = Nonce X') = (X = X')$  **by** *transfer auto*

**lemma**  $Nonce-Agent$ :  $Nonce X \neq Agent X'$  **by** *transfer auto*

**lemma**  $Nonce-Number$ :  $Nonce X \neq Number X'$  **by** *transfer auto*

**lemma**  $Nonce-Hash$ :  $Nonce X \neq Hash X'$  **by** *transfer auto*

**lemma**  $Nonce-Tag$ :  $Nonce X \neq Tag X'$  **by** *transfer auto*

**lemma**  $Nonce-EphK$ :  $Nonce X \neq EphK X'$  **by** *transfer auto*

**lemma**  $Nonce-LtK$ :  $Nonce X \neq LtK X'$  **by** *transfer auto*

**lemma**  $Nonce-Pair$ :  $Nonce X \neq Pair X' Y'$  **by** *transfer auto*

**lemma**  $Nonce-Enc$ :  $Nonce X \neq Enc X' Y'$  **by** *transfer auto*

**lemma**  $Nonce-Aenc$ :  $Nonce X \neq Aenc X' Y'$  **by** *transfer auto*

**lemma**  $Nonce-Sign$ :  $Nonce X \neq Sign X' Y'$  **by** *transfer auto*

**lemma**  $Nonce-Exp$ :  $Nonce X \neq Exp X' Y'$  **by** *transfer auto*

**lemma**  $Agent-Nonce$ :  $Agent X \neq Nonce X'$  **by** *transfer auto*

**lemma**  $Agent-Agent$ :  $(Agent X = Agent X') = (X = X')$  **by** *transfer auto*

**lemma**  $Agent-Number$ :  $Agent X \neq Number X'$  **by** *transfer auto*

**lemma**  $Agent-Hash$ :  $Agent X \neq Hash X'$  **by** *transfer auto*

**lemma**  $Agent-Tag$ :  $Agent X \neq Tag X'$  **by** *transfer auto*

**lemma**  $Agent-EphK$ :  $Agent X \neq EphK X'$  **by** *transfer auto*

**lemma**  $Agent-LtK$ :  $Agent X \neq LtK X'$  **by** *transfer auto*

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

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

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

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

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

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

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

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

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

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

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

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

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

**lemma** *Sign-Nonce*:  $Sign\ X\ Y \neq Nonce\ X'$  **by** *transfer auto*

**lemma** *Sign-Agent*:  $Sign\ X\ Y \neq Agent\ X'$  **by** *transfer auto*

**lemma** *Sign-Number*:  $Sign\ X\ Y \neq Number\ X'$  **by** *transfer auto*

**lemma** *Sign-Hash*:  $Sign\ X\ Y \neq Hash\ X'$  **by** *transfer auto*

**lemma** *Sign-Tag*:  $Sign\ X\ Y \neq Tag\ X'$  **by** *transfer auto*

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

**lemma** *Sign-LtK*:  $Sign\ X\ Y \neq LtK\ X'$  **by** *transfer auto*

**lemma** *Sign-Pair*:  $Sign\ X\ Y \neq Pair\ X'\ Y'$  **by** *transfer auto*

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

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

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

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

**lemma** *Exp-Nonce*:  $Exp\ X\ Y \neq Nonce\ X'$  **by** *transfer auto*

**lemma** *Exp-Agent*:  $Exp\ X\ Y \neq Agent\ X'$  **by** *transfer auto*

**lemma** *Exp-Number*:  $Exp\ X\ Y \neq Number\ X'$  **by** *transfer auto*

**lemma** *Exp-Hash*:  $Exp\ X\ Y \neq Hash\ X'$  **by** *transfer auto*

**lemma** *Exp-Tag*:  $Exp\ X\ Y \neq Tag\ X'$  **by** *transfer auto*

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

**lemma** *Exp-LtK*:  $Exp\ X\ Y \neq LtK\ X'$  **by** *transfer auto*

**lemma** *Exp-Pair*:  $Exp\ X\ Y \neq Pair\ X'\ Y'$  **by** *transfer auto*

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

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

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

**lemmas** *msg-inject* [*iff*, *induct-simp*] =

*Nonce-Nonce Agent-Agent Number-Number Hash-Hash Tag-Tag EphK-EphK LtK-LtK  
Pair-Pair Enc-Enc Aenc-Aenc Sign-Sign*

**lemmas** *msg-distinct* [*simp*, *induct-simp*] =

*Nonce-Agent Nonce-Number Nonce-Hash Nonce-Tag Nonce-EphK Nonce-LtK Nonce-Pair  
Nonce-Enc Nonce-Aenc Nonce-Sign Nonce-Exp  
Agent-Nonce Agent-Number Agent-Hash Agent-Tag Agent-EphK Agent-LtK Agent-Pair  
Agent-Enc Agent-Aenc Agent-Sign Agent-Exp  
Number-Nonce Number-Agent Number-Hash Number-Tag Number-EphK Number-LtK  
Number-Pair Number-Enc Number-Aenc Number-Sign Number-Exp  
Hash-Nonce Hash-Agent Hash-Number Hash-Tag Hash-EphK Hash-LtK Hash-Pair  
Hash-Enc Hash-Aenc Hash-Sign Hash-Exp  
Tag-Nonce Tag-Agent Tag-Number Tag-Hash Tag-EphK Tag-LtK Tag-Pair  
Tag-Enc Tag-Aenc Tag-Sign Tag-Exp  
EphK-Nonce EphK-Agent EphK-Number EphK-Hash EphK-Tag EphK-LtK EphK-Pair  
EphK-Enc EphK-Aenc EphK-Sign EphK-Exp  
LtK-Nonce LtK-Agent LtK-Number LtK-Hash LtK-Tag LtK-EphK LtK-Pair  
LtK-Enc LtK-Aenc LtK-Sign LtK-Exp  
Pair-Nonce Pair-Agent Pair-Number Pair-Hash Pair-Tag Pair-EphK Pair-LtK  
Pair-Enc Pair-Aenc Pair-Sign Pair-Exp  
Enc-Nonce Enc-Agent Enc-Number Enc-Hash Enc-Tag Enc-EphK Enc-LtK Enc-Pair  
Enc-Aenc Enc-Sign Enc-Exp*

*Aenc-Nonce Aenc-Agent Aenc-Number Aenc-Hash Aenc-Tag Aenc-EphK Aenc-LtK*  
*Aenc-Pair Aenc-Enc Aenc-Sign Aenc-Exp*  
*Sign-Nonce Sign-Agent Sign-Number Sign-Hash Sign-Tag Sign-EphK Sign-LtK*  
*Sign-Pair Sign-Enc Sign-Aenc Sign-Exp*  
*Exp-Nonce Exp-Agent Exp-Number Exp-Hash Exp-Tag Exp-EphK Exp-LtK Exp-Pair*  
*Exp-Enc Exp-Aenc Exp-Sign*

**consts** *Ngen* :: *nat*

**abbreviation** *Gen*  $\equiv$  *Number Ngen*

**abbreviation** *cGen*  $\equiv$  *cNumber Ngen*

**abbreviation**

*InsecTag*  $\equiv$  *Tag insec*

**abbreviation**

*AuthTag*  $\equiv$  *Tag auth*

**abbreviation**

*ConfidTag*  $\equiv$  *Tag confid*

**abbreviation**

*SecureTag*  $\equiv$  *Tag secure*

**abbreviation**

*Tags*  $\equiv$  *range Tag*

**abbreviation**

*NonceF* :: *fresh-t*  $\Rightarrow$  *msg* **where**

*NonceF N*  $\equiv$  *Nonce (nonce-fresh N)*

**abbreviation**

*NonceA* :: *nat*  $\Rightarrow$  *msg* **where**

*NonceA N*  $\equiv$  *Nonce (nonce-atk N)*

**abbreviation**

*shrK* :: *agent*  $\Rightarrow$  *agent*  $\Rightarrow$  *msg* **where**

*shrK A B*  $\equiv$  *LtK (sharK A B)*

**abbreviation**

*pubK* :: *agent*  $\Rightarrow$  *msg* **where**

*pubK A*  $\equiv$  *LtK (publK A)*

**abbreviation**

*priK* :: *agent*  $\Rightarrow$  *msg* **where**

*priK A*  $\equiv$  *LtK (privK A)*

**abbreviation**

*epubK* :: *nonce-t*  $\Rightarrow$  *msg* **where**

*epubK N*  $\equiv$  *EphK (epublK N)*

**abbreviation**

*epriK* :: *nonce-t*  $\Rightarrow$  *msg* **where**



$epriK N \equiv EphK (eprivK N)$

**abbreviation**

$epubKF :: fresh-t \Rightarrow msg$  **where**  
 $epubKF N \equiv EphK (epubK (nonce-fresh N))$

**abbreviation**

$epriKF :: fresh-t \Rightarrow msg$  **where**  
 $epriKF N \equiv EphK (eprivK (nonce-fresh N))$

**abbreviation**

$epubKA :: nat \Rightarrow msg$  **where**  
 $epubKA N \equiv EphK (epubK (nonce-atk N))$

**abbreviation**

$epriKA :: nat \Rightarrow msg$  **where**  
 $epriKA N \equiv EphK (eprivK (nonce-atk N))$

Concrete syntax: messages appear as  $\langle A, B, NA \rangle$ , etc...

**syntax**

$-MTuple :: [ 'a, args ] \Rightarrow 'a * 'b \quad ((\lambda(-, / -)))$

**translations**

$\langle x, y, z \rangle \Leftrightarrow \langle x, \langle y, z \rangle \rangle$   
 $\langle x, y \rangle \Leftrightarrow CONST Pair x y$

hash macs

**abbreviation**

$hmac :: msg \Rightarrow msg \Rightarrow msg$  **where**  
 $hmac M K \equiv Hash \langle M, K \rangle$

recover some kind of injectivity for Exp

**lemma eq-expgen:**

$eq X Y \Longrightarrow (\forall X'. X = cExp cGen X' \longrightarrow (\exists Z. Y = (cExp cGen Z) \wedge eq X' Z)) \wedge$   
 $(\forall Y'. Y = cExp cGen Y' \longrightarrow (\exists Z. X = (cExp cGen Z) \wedge eq Y' Z))$

**by** (*erule eq.induct, auto elim!: Tr*)

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

**by** (*transfer, auto dest: eq-expgen*)

**lemma eq-expexpgen:**

$eq X Y \Longrightarrow (\forall X' X''. X = cExp (cExp cGen X') X'' \longrightarrow$   
 $(\exists Y' Y''. Y = cExp (cExp cGen Y') Y'' \wedge$   
 $((eq X' Y' \wedge eq X'' Y'') \vee (eq X' Y'' \wedge eq X'' Y'))))$

**apply** (*erule eq.induct, simp-all*)

**apply** (*((drule eq-expgen)+, force)*)

**apply** (*auto, blast+*)

**done**

**lemma Exp-Exp-Gen-inj:**

$Exp (Exp Gen X) X' = Z \Longrightarrow$   
 $(\exists Y Y'. Z = Exp (Exp Gen Y) Y' \wedge ((X = Y \wedge X' = Y') \vee (X = Y' \wedge X' = Y)))$

**by** (*transfer*, *auto dest: eq-expexpgen*)

**lemma** *Exp-Exp-Gen-inj2*:

$Exp (Exp Gen X) X' = Exp Z Y' \implies$

$(Y' = X \wedge Z = Exp Gen X') \vee (Y' = X' \wedge Z = Exp Gen X)$

**apply** (*transfer*, *auto*)

**apply** (*drule eq-expexpgen*, *auto*)+

**done**

**end**

## 4 Message theory

```
theory Message-derivation
imports Messages
begin
```

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

### 4.1 Message composition

Dolev-Yao message synthesis.

**inductive-set**

```
synth :: msg set  $\Rightarrow$  msg set
for H :: msg set
```

**where**

```
Ax [intro]: X  $\in$  H  $\Longrightarrow$  X  $\in$  synth H
| Agent [simp, intro]: Agent A  $\in$  synth H
| Number [simp, intro]: Number n  $\in$  synth H
| NonceA [simp, intro]: NonceA n  $\in$  synth H
| EpubKA [simp, intro]: epubKA n  $\in$  synth H
| EpriKA [simp, intro]: epriKA n  $\in$  synth H
| Hash [intro]: X  $\in$  synth H  $\Longrightarrow$  Hash X  $\in$  synth H
| Pair [intro]: X  $\in$  synth H  $\Longrightarrow$  Y  $\in$  synth H  $\Longrightarrow$  (Pair X Y)  $\in$  synth H
| Enc [intro]: X  $\in$  synth H  $\Longrightarrow$  Y  $\in$  synth H  $\Longrightarrow$  (Enc X Y)  $\in$  synth H
| Aenc [intro]: X  $\in$  synth H  $\Longrightarrow$  Y  $\in$  synth H  $\Longrightarrow$  (Aenc X Y)  $\in$  synth H
| Sign [intro]: X  $\in$  synth H  $\Longrightarrow$  Y  $\in$  synth H  $\Longrightarrow$  Sign X Y  $\in$  synth H
| Exp [intro]: X  $\in$  synth H  $\Longrightarrow$  Y  $\in$  synth H  $\Longrightarrow$  (Exp X Y)  $\in$  synth H
```

Lemmas about Dolev-Yao message synthesis.

```
lemma synth-mono [mono-set]: G  $\subseteq$  H  $\Longrightarrow$  synth G  $\subseteq$  synth H
by (auto, erule synth.induct, auto)
```

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

— [elim!] slows down certain proofs, e.g.,  $\llbracket \text{synth } H \cap B \subseteq \{\} \rrbracket \Longrightarrow P$

```
inductive-cases NonceF-synth: NonceF n  $\in$  synth H
```

```
inductive-cases LtK-synth: LtK K  $\in$  synth H
```

```
inductive-cases EpubKF-synth: epubKF K  $\in$  synth H
```

```
inductive-cases EpriKF-synth: epriKF K  $\in$  synth H
```

```
inductive-cases Hash-synth: Hash X  $\in$  synth H
```

```
inductive-cases Pair-synth: Pair X Y  $\in$  synth H
```

```
inductive-cases Enc-synth: Enc X K  $\in$  synth H
```

```
inductive-cases Aenc-synth: Aenc X K  $\in$  synth H
```

```
inductive-cases Sign-synth: Sign X K  $\in$  synth H
```

```
inductive-cases Tag-synth: Tag t  $\in$  synth H
```

```
lemma EpriK-synth [elim]: epriK K  $\in$  synth H  $\Longrightarrow$ 
  epriK K  $\in$  H  $\vee$  ( $\exists$  N. epriK K = epriKA N)
by (cases K, auto elim: EpriKF-synth)
```

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

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

**lemmas** *synth-inversion* [*elim*] =

*NonceF-synth LtK-synth EpubKF-synth EpriKF-synth Hash-synth Pair-synth  
Enc-synth Aenc-synth Sign-synth Tag-synth*

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

**by** *blast*

**lemma** *synth-Int1*:  $x \in \text{synth } (A \cap B) \implies x \in \text{synth } A$

**by** (*erule synth.induct*) (*auto*)

**lemma** *synth-Int2*:  $x \in \text{synth } (A \cap B) \implies x \in \text{synth } B$

**by** (*erule synth.induct*) (*auto*)

**lemma** *synth-Int*:  $x \in \text{synth } (A \cap B) \implies x \in \text{synth } A \cap \text{synth } B$

**by** (*blast intro: synth-Int1 synth-Int2*)

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

**by** (*intro Un-least synth-mono Un-upper1 Un-upper2*)

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

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

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

**by** (*erule synth.induct, blast+*)

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

**by** *blast*

**lemma** *synth-subset-iff*:  $\text{synth } G \subseteq \text{synth } H \longleftrightarrow G \subseteq \text{synth } H$

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

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

**by** (*drule synth-mono, blast*)

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

**by** (*erule synth-trans, blast*)

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

**by** *blast*

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

**by** *blast*

**lemma** *EpubKF-synth-eq* [*simp*]:  $(\text{epubKF } K \in \text{synth } H) = (\text{epubKF } K \in H)$

**by** *blast*

**lemma** *EpriKF-synth-eq* [*simp*]:  $(\text{epriKF } K \in \text{synth } H) = (\text{epriKF } K \in H)$

**by** *blast*

**lemma** *Enc-synth-eq1* [*simp*]:

$$K \notin \text{synth } H \implies (\text{Enc } X \ K \in \text{synth } H) = (\text{Enc } X \ K \in H)$$

**by** *blast*

**lemma** *Enc-synth-eq2* [*simp*]:

$$X \notin \text{synth } H \implies (\text{Enc } X \ K \in \text{synth } H) = (\text{Enc } X \ K \in H)$$

**by** *blast*

**lemma** *Aenc-synth-eq1* [*simp*]:

$$K \notin \text{synth } H \implies (\text{Aenc } X \ K \in \text{synth } H) = (\text{Aenc } X \ K \in H)$$

**by** *blast*

**lemma** *Aenc-synth-eq2* [*simp*]:

$$X \notin \text{synth } H \implies (\text{Aenc } X \ K \in \text{synth } H) = (\text{Aenc } X \ K \in H)$$

**by** *blast*

**lemma** *Sign-synth-eq1* [*simp*]:

$$K \notin \text{synth } H \implies (\text{Sign } X \ K \in \text{synth } H) = (\text{Sign } X \ K \in H)$$

**by** *blast*

**lemma** *Sign-synth-eq2* [*simp*]:

$$X \notin \text{synth } H \implies (\text{Sign } X \ K \in \text{synth } H) = (\text{Sign } X \ K \in H)$$

**by** *blast*

## 4.2 Message decomposition

Dolev-Yao message decomposition using known keys.

**inductive-set**

*analz* :: *msg set*  $\Rightarrow$  *msg set*

**for** *H* :: *msg set*

**where**

*Ax* [*intro*]:  $X \in H \implies X \in \text{analz } H$

| *Fst*:  $\text{Pair } X \ Y \in \text{analz } H \implies X \in \text{analz } H$

| *Snd*:  $\text{Pair } X \ Y \in \text{analz } H \implies Y \in \text{analz } H$

| *Dec* [*dest*]:

$\llbracket \text{Enc } X \ Y \in \text{analz } H; Y \in \text{synth } (\text{analz } H) \rrbracket \implies X \in \text{analz } H$

| *Adec-lt* [*dest*]:

$\llbracket \text{Aenc } X \ (\text{LtK } (\text{pubK } Y)) \in \text{analz } H; \text{priK } Y \in \text{analz } H \rrbracket \implies X \in \text{analz } H$

| *Adec-eph* [*dest*]:

$\llbracket \text{Aenc } X \ (\text{EphK } (\text{epublK } Y)) \in \text{analz } H; \text{epriK } Y \in \text{synth } (\text{analz } H) \rrbracket \implies X \in \text{analz } H$

| *Sign-getmsg* [*dest*]:

$\text{Sign } X \ (\text{priK } Y) \in \text{analz } H \implies \text{pubK } Y \in \text{analz } H \implies X \in \text{analz } H$

Lemmas about Dolev-Yao message decomposition.

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

**by** (*safe*, *erule analz.induct*) (*auto dest: Fst Snd synth-Int2*)

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

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

$\llbracket \text{Pair } X \ Y \in \text{analz } H; \llbracket X \in \text{analz } H; Y \in \text{analz } H \rrbracket \implies P \rrbracket \implies P$

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

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

**by** (*safe, erule analz.induct*) (*blast+*)

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

**by** *auto*

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

**by** (*induction X rule: analz.induct*) (*auto dest: synth-monotone*)

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

**by** *auto*

**lemma** *analz-Un*:  $\text{analz } G \cup \text{analz } H \subseteq \text{analz } (G \cup H)$

**by** (*intro Un-least analz-mono Un-upper1 Un-upper2*)

**lemma** *analz-insertI*:  $X \in \text{analz } H \implies X \in \text{analz } (\text{insert } Y H)$

**by** (*blast intro: analz-monotone*)

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

**by** (*blast intro: analz-monotone*)

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

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

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

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

**by** (*drule analz-mono*) *blast*

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

**by** (*erule analz-trans*) *blast*

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

**by** (*blast intro: analz-cut analz-insertI*)

**lemma** *analz-subset-cong*:

$\text{analz } G \subseteq \text{analz } G' \implies$

$\text{analz } H \subseteq \text{analz } H' \implies$

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

**apply** *simp*

**apply** (*iprover intro: conjI subset-trans analz-mono Un-upper1 Un-upper2*)

**done**

**lemma** *analz-cong*:

$\text{analz } G = \text{analz } G' \implies$

$\text{analz } H = \text{analz } H' \implies$

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

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

**lemma** *analz-insert-cong*:

$analz\ H = analz\ H' \implies$

$analz\ (insert\ X\ H) = analz\ (insert\ X\ H')$

**by** (*force simp only: insert-def intro!: analz-cong*)

**lemma** *analz-trivial*:

$\forall X\ Y. Pair\ X\ Y \notin H \implies$

$\forall X\ Y. Enc\ X\ Y \notin H \implies$

$\forall X\ Y. Aenc\ X\ Y \notin H \implies$

$\forall X\ Y. Sign\ X\ Y \notin H \implies$

$analz\ H = H$

**apply** *safe*

**apply** (*erule analz.induct, blast+*)

**done**

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

**apply** (*intro equalityI analz-subset-cong*)**+**

**apply** *simp-all*

**done**

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

**by** (*subst Un-commute, auto*)**+**

Lemmas about *analz* and *insert*.

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

$analz\ (insert\ (Agent\ A)\ H) = insert\ (Agent\ A)\ (analz\ H)$

**apply** (*rule analz-insert-eq-I*)

**apply** (*erule analz.induct*)

**thm** *analz.induct*

**apply** *fastforce*

**apply** *fastforce*

**apply** *fastforce*

**defer** 1

**apply** *fastforce*

**defer** 1

**apply** *fastforce*

**apply** (*rename-tac x X Y*)

**apply** (*subgoal-tac Y  $\in$  synth (analz H), auto*)

**apply** (*thin-tac Enc X Y  $\in$  Z for Z*)**+**

**apply** (*drule synth-Int2, auto*)

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

**apply** (*rename-tac X Y*)

**apply** (*subgoal-tac eprK Y  $\in$  synth (analz H), auto*)

**apply** (*thin-tac Aenc X (epubK Y)  $\in$  Z for Z*)**+**

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

**done**

### 4.3 Lemmas about combined composition/decomposition

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

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

**lemma** *synth-analz-mono*:  $G \subseteq H \implies \text{synth} (\text{analz } G) \subseteq \text{synth} (\text{analz } H)$   
**by** (*blast intro!*: *analz-mono synth-mono*)

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

**lemma** *lem1*:

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

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

**lemma** *analz-synth-Un*:  $\text{analz} (\text{synth } G \cup H) = \text{analz} (G \cup H) \cup \text{synth } G$

**proof** (*intro equalityI subsetI*)

**fix** *x*

**assume**  $x \in \text{analz} (\text{synth } G \cup H)$

**thus**  $x \in \text{analz} (G \cup H) \cup \text{synth } G$

**by** (*induction x rule: analz.induct*)

(*auto simp add: lem2 intro: analz-monotone Fst Snd Dec Adec-eph Adec-lt Sign-getmsg*  
*dest: lem1*)

**next**

**fix** *x*

**assume**  $x \in \text{analz} (G \cup H) \cup \text{synth } G$

**thus**  $x \in \text{analz} (\text{synth } G \cup H)$

**by** (*blast intro: analz-monotone*)

**qed**

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

**by** (*rule analz-synth-Un [where H={}, simplified]*)

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

**by** (*subst Un-commute, auto simp add: analz-synth-Un*)<sup>+</sup>

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

**by** (*auto del:subsetI*) (*auto simp add: synth-subset-iff analz-synth*)

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

**by** (*auto simp add: analz-synth*)

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

**by** (*simp only: analz-synth-analz*) *simp*



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

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

#### 4.4 Accessible message parts

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

**inductive-set**  
 $\text{parts} :: \text{msg set} \Rightarrow \text{msg set}$   
**for**  $H :: \text{msg set}$   
**where**  
 $\text{Inj}$  [*intro*]:  $X \in H \implies X \in \text{parts } H$   
 $\text{Fst}$  [*intro*]:  $\text{Pair } X Y \in \text{parts } H \implies X \in \text{parts } H$   
 $\text{Snd}$  [*intro*]:  $\text{Pair } X Y \in \text{parts } H \implies Y \in \text{parts } H$   
 $\text{Dec}$  [*intro*]:  $\text{Enc } X Y \in \text{parts } H \implies X \in \text{parts } H$   
 $\text{Adec}$  [*intro*]:  $\text{Aenc } X Y \in \text{parts } H \implies X \in \text{parts } H$   
 $\text{Sign-getmsg}$  [*intro*]:  $\text{Sign } X Y \in \text{parts } H \implies X \in \text{parts } H$

Lemmas about accessible message parts.

**lemma** *parts-mono* [*mono-set*]:  $G \subseteq H \implies \text{parts } G \subseteq \text{parts } H$   
**by** (*auto, erule parts.induct, auto*)

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

**lemma** *Pair-parts* [*elim*]:  
 $\llbracket \text{Pair } X Y \in \text{parts } H; \llbracket X \in \text{parts } H; Y \in \text{parts } H \rrbracket \implies P \rrbracket \implies P$   
**by** *blast*

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

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

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

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

**lemma** *parts-atomic* [*simp*]: *atomic x*  $\implies$  *parts {x} = {x}*

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

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

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

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

**by** *simp*

**lemma** *parts-Tags* [*simp*]:

*parts Tags = Tags*

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

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

**by** (*erule parts.induct*, *blast+*)

**lemma** *parts-Agents* [*simp*]:

*parts (Agent' G) = Agent' G*

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

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

**proof**

**show** *parts (G  $\cup$  H)  $\subseteq$  parts G  $\cup$  parts H*

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

**next**

**show** *parts G  $\cup$  parts H  $\subseteq$  parts (G  $\cup$  H)*

**by** (*intro Un-least parts-mono Un-upper1 Un-upper2*)

**qed**

**lemma** *parts-insert-subset-Un*:

**assumes**  $X \in G$

**shows** *parts (insert X H)  $\subseteq$  parts G  $\cup$  parts H*

**proof** –

**from** *assms* **have** *parts (insert X H)  $\subseteq$  parts (G  $\cup$  H)* **by** (*blast intro!: parts-mono*)

**thus** *?thesis* **by** *simp*

**qed**

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

**by** (*blast intro!: parts-insert-subset-Un intro: parts-monotone*)

**lemma** *parts-insert2*:

*parts (insert X (insert Y H)) = parts {X}  $\cup$  parts {Y}  $\cup$  parts H*

**apply** (*simp add: Un-assoc*)

**apply** (*simp add: parts-insert [symmetric]*)

**done**

**lemma** *parts-UN* [*simp*]: *parts ( $\bigcup x \in A. H x$ ) = ( $\bigcup x \in A. \text{parts}(H x)$ )* (**is**  $?X = ?Y$ )

**proof**

**show**  $?X \subseteq ?Y$  **by** (*rule subsetI*, *erule parts.induct*) (*blast+*)

**next**

**show**  $?Y \subseteq ?X$  **by** (*intro UN-least parts-mono UN-upper*)  
**qed**

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

**lemma** *parts-insert-subset*:  $\text{insert } X \text{ (parts } H) \subseteq \text{parts (insert } X H)$   
**by** (*blast intro: parts-mono [THEN [2] rev-subsetD]*)

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

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

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

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

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

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

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

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

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

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

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

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

$parts (insert (Hash X) H) = insert (Hash X) (parts H)$   
**by** (rule parts-insert-eq-I, erule parts.induct, auto)

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

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

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

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

#### 4.4.1 Lemmas about combinations with composition and decomposition

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

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

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

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

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

done

**lemma** *parts-synth* [simp]:  $\text{parts} (\text{synth } H) = \text{parts } H \cup \text{synth } H$   
**apply** (rule equalityI)  
**apply** (rule subsetI)  
**apply** (erule parts.induct)  
**apply** (blast intro: synth-increasing [THEN parts-mono, THEN subsetD])  
**done**

**lemma** *Fake-parts-insert*:  
 $X \in \text{synth} (\text{analz } H) \implies \text{parts} (\text{insert } X H) \subseteq \text{synth} (\text{analz } H) \cup \text{parts } H$   
**apply** (erule parts-insert-subset-Un)  
**apply** (simp (no-asm-use))  
**apply** blast  
**done**

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

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

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

## 4.5 More lemmas about combinations of closures

Combinations of *synth* and *analz*.

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

**lemma** *Enc-synth-analz*:  
 $Y \in \text{synth} (\text{analz } H) \implies$   
 $(\text{Enc } X Y \in \text{synth} (\text{analz } H)) \longleftrightarrow (X \in \text{synth} (\text{analz } H))$   
**by** blast

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

**lemma** *gen-analz-insert-eq*:  
 $\llbracket X \in \text{analz } G; G \subseteq H \rrbracket \implies \text{analz} (\text{insert } X H) = \text{analz } H$

**by** (*blast intro: analz-cut analz-insertI analz-monotone*)

**lemma** *synth-analz-insert-eq*:

$\llbracket X \in \text{synth}(\text{analz } G); G \subseteq H \rrbracket \implies \text{synth}(\text{analz}(\text{insert } X H)) = \text{synth}(\text{analz } H)$

**by** (*blast intro!: synth-analz-insert synth-analz-monotone*)

**lemma** *Fake-parts-sing*:

$X \in \text{synth}(\text{analz } H) \implies \text{parts } \{X\} \subseteq \text{synth}(\text{analz } H) \cup \text{parts } H$

**apply** (*rule subset-trans*)

**apply** (*erule-tac [2] Fake-parts-insert*)

**apply** (*rule parts-mono, blast*)

**done**

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

**lemma** *analz-hash-nonce [simp]*:

$\text{analz } \{M. \exists N. M = \text{Hash}(\text{Nonce } N)\} = \{M. \exists N. M = \text{Hash}(\text{Nonce } N)\}$

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

**lemma** *synth-analz-hash-nonce [simp]*:

$\text{NonceF } N \notin \text{synth}(\text{analz } \{M. \exists N. M = \text{Hash}(\text{Nonce } N)\})$

**by** *auto*

**lemma** *synth-analz-idem-mono*:

$S \subseteq \text{synth}(\text{analz } S') \implies \text{synth}(\text{analz } S) \subseteq \text{synth}(\text{analz } S')$

**by** (*frule synth-analz-mono, auto*)

**lemmas** *synth-analz-idem-monoI =*

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

**lemma** *analz-synth-subset*:

$\text{analz } X \subseteq \text{synth}(\text{analz } X') \implies$

$\text{analz } Y \subseteq \text{synth}(\text{analz } Y') \implies$

$\text{analz } (X \cup Y) \subseteq \text{synth}(\text{analz } (X' \cup Y'))$

**proof** –

**assume**  $\text{analz } X \subseteq \text{synth}(\text{analz } X')$

**then have**  $HX:\text{analz } X \subseteq \text{analz}(\text{synth}(\text{analz } X'))$  **by** *blast*

**assume**  $\text{analz } Y \subseteq \text{synth}(\text{analz } Y')$

**then have**  $HY:\text{analz } Y \subseteq \text{analz}(\text{synth}(\text{analz } Y'))$  **by** *blast*

**from** *analz-subset-cong [OF HX HY]*

**have**  $\text{analz } (X \cup Y) \subseteq \text{analz}(\text{synth}(\text{analz } X') \cup \text{synth}(\text{analz } Y'))$

**by** *blast*

**also have**  $\dots \subseteq \text{analz}(\text{synth}(\text{analz } X' \cup \text{analz } Y'))$

**by** (*intro analz-mono synth-Un*)

**also have**  $\dots \subseteq \text{analz}(\text{synth}(\text{analz } (X' \cup Y')))$

**by** (*intro synth-mono analz-mono analz-Un*)

**also have**  $\dots \subseteq \text{synth}(\text{analz } (X' \cup Y'))$

**by** *auto*

**finally show**  $\text{analz } (X \cup Y) \subseteq \text{synth}(\text{analz } (X' \cup Y'))$

**by** *auto*

qed

**lemma** *analz-synth-subset-Un1* :

$\text{analz } X \subseteq \text{synth } (\text{analz } X') \implies \text{analz } (X \cup Y) \subseteq \text{synth } (\text{analz } (X' \cup Y))$

**using** *analz-synth-subset* **by** *blast*

**lemma** *analz-synth-subset-Un2* :

$\text{analz } X \subseteq \text{synth } (\text{analz } X') \implies \text{analz } (Y \cup X) \subseteq \text{synth } (\text{analz } (Y \cup X'))$

**using** *analz-synth-subset* **by** *blast*

**lemma** *analz-synth-insert*:

$\text{analz } X \subseteq \text{synth } (\text{analz } X') \implies \text{analz } (\text{insert } Y X) \subseteq \text{synth } (\text{analz } (\text{insert } Y X'))$

**by** (*metis analz-synth-subset-Un2 insert-def*)

**lemma** *Fake-analz-insert-Un*:

**assumes**  $Y \in \text{analz } (\text{insert } X H)$  **and**  $X \in \text{synth } (\text{analz } G)$

**shows**  $Y \in \text{synth } (\text{analz } G) \cup \text{analz } (G \cup H)$

**proof** –

**from** *assms* **have**  $Y \in \text{analz } (\text{synth } (\text{analz } G) \cup H)$

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

$\text{analz-mono [THEN synth-mono, THEN [2] rev-subsetD]}$ )

**thus** *?thesis* **by** (*simp add: sup.commute*)

qed

end

## 5 Environment: Dolev-Yao Intruder

```

theory IK
imports Message-derivation
begin

```

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

```

record ik-state =
  ik :: msg set

```

Dolev-Yao intruder event adds a derived message.

```

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

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

```

```

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

```

```

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

```

```

lemma ik-trans-ik-increasing: (s, s')  $\in$  ik-trans  $\implies$  ik s  $\subseteq$  ik s'
by (auto simp add: ik-trans-defs)

```

```

lemma ik-trans-synth-analz-ik-increasing:
  (s, s')  $\in$  ik-trans  $\implies$  synth (analz (ik s))  $\subseteq$  synth (analz (ik s'))
by (simp only: synth-analz-mono ik-trans-ik-increasing)

```

```

end

```



## 6 Secrecy Model (L0)

```
theory Secrecy
imports Refinement IK
begin
```

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

### 6.1 State and events

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

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

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

#### definition

```
s0-secrecy :: 'a s0-state-scheme set
```

#### where

```
s0-secrecy ≡ {s. synth (analz (ik s)) ∩ secret s = {}}
```

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

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

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

#### definition

```
s0-add-secret :: msg ⇒ ('a s0-state-scheme * 'a s0-state-scheme) set
```

#### where

```
s0-add-secret m ≡ {(s,s').
```

```
— guard
```

```
m ∉ synth (analz (ik s)) ∧
```

```
— action
```

```
s' = s(secret := insert m (secret s))
```

```
}
```

#### definition

```
s0-learn :: msg ⇒ ('a s0-state-scheme * 'a s0-state-scheme) set
```

#### where

```
s0-learn m ≡ {(s,s').
```

```
— guard
```

```
s(ik := insert m (ik s)) ∈ s0-secrecy ∧
```

```
— action
```

```
s' = s(ik := insert m (ik s))
```

```
}
```

#### definition

```
s0-learn' :: msg ⇒ ('a s0-state-scheme * 'a s0-state-scheme) set
```

#### where

```
s0-learn' m ≡ {(s,s').
```

```
— guard
```

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

— action  
 $s' = s \{ ik := \text{insert } m \ (ik \ s) \}$

**definition**

$s0\text{-trans} :: ('a \ s0\text{-state-scheme} * 'a \ s0\text{-state-scheme}) \text{ set}$

**where**

$s0\text{-trans} \equiv (\bigcup m. \ s0\text{-add-secret } m) \cup (\bigcup m. \ s0\text{-learn } m) \cup Id$

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

**definition**

$s0\text{-init} :: 'a \ s0\text{-state-scheme} \text{ set}$

**where**

$s0\text{-init} \equiv s0\text{-secrecy}$

**type-synonym**

$s0\text{-obs} = s0\text{-state}$

**definition**

$s0 :: (s0\text{-state}, s0\text{-obs}) \text{ spec}$  **where**

$s0 \equiv \{$   
 $\text{init} = s0\text{-init},$   
 $\text{trans} = s0\text{-trans},$   
 $\text{obs} = id$   
 $\}$

**lemmas**  $s0\text{-defs} = s0\text{-def } s0\text{-init-def } s0\text{-trans-def } s0\text{-add-secret-def } s0\text{-learn-def}$

**lemmas**  $s0\text{-all-defs} = s0\text{-defs } ik\text{-trans-defs}$

**lemma**  $s0\text{-obs-id}$  [*simp*]:  $\text{obs } s0 = id$

**by** (*simp add: s0-def*)

## 6.2 Proof of secrecy invariant

**lemma**  $s0\text{-secrecy-init}$  [*iff*]:  $\text{init } s0 \subseteq s0\text{-secrecy}$

**by** (*simp add: s0-defs*)

**lemma**  $s0\text{-secrecy-trans}$  [*simp, intro*]:  $\{s0\text{-secrecy}\} \text{ trans } s0 \{> s0\text{-secrecy}\}$

**apply** (*auto simp add: s0-all-defs PO-hoare-defs intro!: s0-secrecyI*)

**apply** (*auto*)

**done**

**lemma**  $s0\text{-secrecy}$  [*iff*]:  $\text{reach } s0 \subseteq s0\text{-secrecy}$

**by** (*rule inv-rule-basic, auto*)

**lemma**  $s0\text{-obs-secrecy}$  [*iff*]:  $\text{oreach } s0 \subseteq s0\text{-secrecy}$

**by** (*rule external-from-internal-invariant*) (*auto del: subsetI*)

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

**end**

## 7 Non-injective Agreement (L0)

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

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

### 7.1 Signals

signals

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

**fun**  
  *addSignal* :: (*signal*  $\Rightarrow$  *nat*)  $\Rightarrow$  *signal*  $\Rightarrow$  *signal*  $\Rightarrow$  *nat*  
**where**  
  *addSignal sigs s* = *sigs* (*s* := *sigs s* + 1)

### 7.2 State and events

level 0 non-injective agreement

**record** *a0n-state* =  
  *signals* :: *signal*  $\Rightarrow$  *nat*   — multi-set of signals

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

Events

**definition**  
  *a0n-running* :: *agent*  $\Rightarrow$  *agent*  $\Rightarrow$  *msg*  $\Rightarrow$  (*a0n-state*  $\times$  *a0n-state*) *set*  
**where**  
  *a0n-running A B M*  $\equiv$   $\{(s, s')\}$ .  
    — action  
    *s'* = *s*(*signals* := *addSignal* (*signals s*) (*Running A B M*))  
  }

**definition**  
  *a0n-commit* :: *agent*  $\Rightarrow$  *agent*  $\Rightarrow$  *msg*  $\Rightarrow$  (*a0n-state*  $\times$  *a0n-state*) *set*  
**where**  
  *a0n-commit A B M*  $\equiv$   $\{(s, s')\}$ .  
    — guard  
    *signals s* (*Running A B M*) > 0  $\wedge$   
    — action  
    *s'* = *s*(*signals* := *addSignal* (*signals s*) (*Commit A B M*))  
  }

**definition**  
  *a0n-trans* :: (*a0n-state*  $\times$  *a0n-state*) *set* **where**  
  *a0n-trans*  $\equiv$   $(\bigcup A B M. a0n-running A B M) \cup (\bigcup A B M. a0n-commit A B M) \cup Id$

Level 0 state

**definition**

$a0n\text{-init} :: a0n\text{-state set}$

**where**

$a0n\text{-init} \equiv \{(\text{signals} = \lambda s. 0)\}$

**definition**

$a0n :: (a0n\text{-state}, a0n\text{-obs}) \text{ spec}$  **where**

$a0n \equiv \langle$   
   $\text{init} = a0n\text{-init},$   
   $\text{trans} = a0n\text{-trans},$   
   $\text{obs} = \text{id}$

$\rangle$

**lemmas**  $a0n\text{-defs} =$

$a0n\text{-def } a0n\text{-init}\text{-def } a0n\text{-trans}\text{-def}$   
 $a0n\text{-running}\text{-def } a0n\text{-commit}\text{-def}$

**lemma**  $a0n\text{-obs-id}$  [*simp*]:  $\text{obs } a0n = \text{id}$

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

**lemma**  $a0n\text{-anyP-observable}$  [*iff*]:  $\text{observable } (\text{obs } a0n) P$

**by** (*auto*)

### 7.3 Non injective agreement invariant

Invariant: non injective agreement

**definition**

$a0n\text{-agreement} :: a0n\text{-state set}$

**where**

$a0n\text{-agreement} \equiv \{s. \forall A B M.$   
   $\text{signals } s (\text{Commit } A B M) > 0 \longrightarrow \text{signals } s (\text{Running } A B M) > 0$   
   $\}$

**lemmas**  $a0n\text{-agreement}I = a0n\text{-agreement}\text{-def}$  [*THEN setc-def-to-intro, rule-format*]

**lemmas**  $a0n\text{-agreement}E$  [*elim*] =  $a0n\text{-agreement}\text{-def}$  [*THEN setc-def-to-elim, rule-format*]

**lemmas**  $a0n\text{-agreement}D = a0n\text{-agreement}\text{-def}$  [*THEN setc-def-to-dest, rule-format, rotated 2*]

**lemma**  $a0n\text{-agreement-init}$  [*iff*]:

$\text{init } a0n \subseteq a0n\text{-agreement}$

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

**lemma**  $a0n\text{-agreement-trans}$  [*iff*]:

$\{a0n\text{-agreement}\} \text{trans } a0n \{> a0n\text{-agreement}\}$

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

**lemma**  $a0n\text{-agreement}$  [*iff*]:  $\text{reach } a0n \subseteq a0n\text{-agreement}$

**by** (*rule inv-rule-basic*) (*auto*)

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

```
end
```

## 8 Injective Agreement (L0)

```
theory AuthenticationI
imports AuthenticationN
begin
```

### 8.1 State and events

```
type-synonym
  a0i-state = a0n-state
```

```
type-synonym
  a0i-obs = a0n-obs
```

```
abbreviation
  a0i-init :: a0n-state set
where
  a0i-init  $\equiv$  a0n-init
```

```
abbreviation
  a0i-running :: agent  $\Rightarrow$  agent  $\Rightarrow$  msg  $\Rightarrow$  (a0i-state  $\times$  a0i-state) set
where
  a0i-running  $\equiv$  a0n-running
```

```
lemmas a0i-running-def = a0n-running-def
```

```
definition
  a0i-commit :: agent  $\Rightarrow$  agent  $\Rightarrow$  msg  $\Rightarrow$  (a0i-state  $\times$  a0i-state) set
where
  a0i-commit A B M  $\equiv$  {(s, s') .
    — guard
      signals s (Commit A B M) < signals s (Running A B M)  $\wedge$ 
    — actions:
      s' = s[signals := addSignal (signals s) (Commit A B M)]
  }
```

```
definition
  a0i-trans :: (a0i-state  $\times$  a0i-state) set where
  a0i-trans  $\equiv$  ( $\bigcup$  A B M. a0i-running A B M)  $\cup$  ( $\bigcup$  A B M. a0i-commit A B M)  $\cup$  Id
```

```
definition
  a0i :: (a0i-state, a0i-obs) spec where
  a0i  $\equiv$  [
    init = a0i-init,
    trans = a0i-trans,
    obs = id
  ]
```

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

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

by (simp add: a0i-def)

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

## 8.2 Injective agreement invariant

**definition**

*a0i-agreement* :: *a0i-state set*

**where**

*a0i-agreement*  $\equiv$  {*s*.  $\forall A B M$ .  
*signals s (Commit A B M)*  $\leq$  *signals s (Running A B M)*  
}

**lemmas** *a0i-agreementI* =

*a0i-agreement-def* [THEN *setc-def-to-intro*, *rule-format*]

**lemmas** *a0i-agreementE* [elim] =

*a0i-agreement-def* [THEN *setc-def-to-elim*, *rule-format*]

**lemmas** *a0i-agreementD* =

*a0i-agreement-def* [THEN *setc-def-to-dest*, *rule-format*, *rotated 1*]

**lemma** *PO-a0i-agreement-init* [iff]:

*init a0i*  $\subseteq$  *a0i-agreement*

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

**lemma** *PO-a0i-agreement-trans* [iff]:

{*a0i-agreement*} *trans a0i* { $>$  *a0i-agreement*}

**apply** (auto simp add: *PO-hoare-defs a0i-defs intro!*: *a0i-agreementI*)

**apply** (auto *dest*: *a0i-agreementD intro*: *le-SucI*)

**done**

**lemma** *PO-a0i-agreement* [iff]: *reach a0i*  $\subseteq$  *a0i-agreement*

by (*rule inv-rule-basic*) (auto)

**lemma** *PO-a0i-obs-agreement* [iff]: *oreach a0i*  $\subseteq$  *a0i-agreement*

**apply** (*rule external-from-internal-invariant*, *fast*)

**apply** (*subst a0i-def*, *auto*)

**done**

## 8.3 Refinement

**definition**

*med0n0i* :: *a0i-obs*  $\Rightarrow$  *a0i-obs*

**where**

*med0n0i*  $\equiv$  *id*

**definition**

*R0n0i* :: (*a0n-state*  $\times$  *a0i-state*) *set*

**where**

*R0n0i*  $\equiv$  *Id*



**lemma** *PO-a0i-running-refines-a0n-running*:  
 $\{R0n0i\} a0n\text{-running } A \ B \ M, a0i\text{-running } A \ B \ M \ \{> \ R0n0i\}$   
**by** (*unfold R0n0i-def*) (*rule relhoare-refl*)

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

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

**lemma** *PO-a0i-refines-init-a0n [iff]*:  
 $init \ a0i \subseteq R0n0i \text{“}(init \ a0n)$   
**by** (*auto simp add: R0n0i-def a0i-defs*)

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

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

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

## 8.4 Derived invariant

**lemma** *agreement-implies-niagreement [iff]*:  $a0i\text{-agreement} \subseteq a0n\text{-agreement}$   
**apply** (*auto intro!: a0n-agreementI*)  
**apply** (*drule a0i-agreementD, drule order.strict-trans2, auto*)  
**done**

**lemma** *PO-a0i-a0n-agreement [iff]*:  $reach \ a0i \subseteq a0n\text{-agreement}$   
**by** (*rule subset-trans, rule, rule*)

**lemma** *PO-a0i-obs-a0n-agreement [iff]*:  $oreach \ a0i \subseteq a0n\text{-agreement}$   
**by** (*rule subset-trans, rule, rule*)

**end**

## 9 Runs

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

### 9.1 Type definitions

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

**datatype** *var* = *Var nat*

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

**type-synonym**  
*frame* = *var*  $\rightarrow$  *msg*

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

**type-synonym**  
*progress-t* = *rid-t*  $\rightarrow$  *var set*

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

**fun**  
  *in-progressS* :: *var set option*  $\Rightarrow$  *var set*  $\Rightarrow$  *bool*  
**where**  
  *in-progressS* (*Some S*) *S'* = (*S'*  $\subseteq$  *S*)  
  | *in-progressS* *None* *S'* = *False*

**lemma** *in-progress-dom* [*elim*]: *in-progress* (*r R*) *x*  $\Longrightarrow$  *R*  $\in$  *dom r*  
**by** (*cases r R, auto*)

**lemma** *in-progress-Some* [*elim*]: *in-progress* *r x*  $\Longrightarrow$   $\exists$  *x*. *r* = *Some x*  
**by** (*cases r, auto*)

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

**end**

## 10 Channel Messages

```
theory Channels
imports Message-derivation
begin
```

### 10.1 Channel messages

```
datatype chan =
  Chan tag agent agent msg
```

#### abbreviation

```
Insec :: [agent, agent, msg] ⇒ chan where
Insec ≡ Chan insecurity
```

#### abbreviation

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

#### abbreviation

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

#### abbreviation

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

### 10.2 Extract

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

#### inductive-set

```
extr :: agent set ⇒ msg set ⇒ chan set ⇒ msg set
for bad :: agent set
and IK :: msg set
and H :: chan set
```

#### where

```
extr-Inj:  $M \in IK \implies M \in \text{extr bad IK H}$ 
```

```
| extr-Chan:
```

```
[[ Chan c A B M ∈ H; c = insecurity ∨ c = auth ∨ A ∈ bad ∨ B ∈ bad ]] ⇒ M ∈ extr bad IK H
```

```
declare extr.intros [intro]
```

```
declare extr.cases [elim]
```

```
lemma extr-empty-chan [simp]: extr bad IK {} = IK
by (auto)
```

```
lemma IK-subset-extr:  $IK \subseteq \text{extr bad IK chan}$ 
```

**by** (*auto*)

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

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

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

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

**lemma** *extr-mono* [*intro*]:  $\llbracket b \subseteq b'; I \subseteq I'; C \subseteq C' \rrbracket \implies \text{extr } b \text{ I } C \subseteq \text{extr } b' \text{ I}' C'$   
**by** (*force*)

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

**lemma** *extr-insert* [*intro*]:  $M \in \text{extr bad IK } H \implies M \in \text{extr bad IK } (\text{insert } C \text{ } H)$   
**by** (*auto*)

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

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

**lemma** *extr-insert-IK-eq* [*simp*]:  $\text{extr bad } (\text{insert } X \text{ } IK) \text{ } CH = \text{insert } X \text{ } (\text{extr bad IK } CH)$   
**by** (*auto*)

**lemma** *extr-insert-bad*:  
 $\text{extr } (\text{insert } A \text{ } \text{bad}) \text{ } IK \text{ } CH \subseteq$   
 $\text{extr bad IK } CH \cup \{M. \exists B. \text{Confid } A \text{ } B \text{ } M \in CH \vee \text{Confid } B \text{ } A \text{ } M \in CH \vee$   
     $\text{Secure } A \text{ } B \text{ } M \in CH \vee \text{Secure } B \text{ } A \text{ } M \in CH\}$   
**by** (*rule, erule extr.induct, auto intro: tag.exhaust*)

**lemma** *extr-insert-Confid* [*simp*]:  
 $A \notin \text{bad} \implies$   
 $B \notin \text{bad} \implies$   
 $\text{extr bad IK } (\text{insert } (\text{Confid } A \text{ } B \text{ } X) \text{ } CH) = \text{extr bad IK } CH$   
**by** *auto*

### 10.3 Fake

The set of channel messages that an attacker can fake given a set of compromised agents, a set of crypto messages and a set of channel messages. The second rule states that an attacker can

fake an insecure or confidential messages or a channel message with a compromised endpoint using a payload that he knows.

**inductive-set**

*fake* :: *agent set*  $\Rightarrow$  *msg set*  $\Rightarrow$  *chan set*  $\Rightarrow$  *chan set*  
**for** *bad* :: *agent set*  
**and** *IK* :: *msg set*  
**and** *chan* :: *chan set*

**where**

*fake-Inj*:  $M \in \text{chan} \Longrightarrow M \in \text{fake bad IK chan}$   
| *fake-New*:  
 $\llbracket M \in \text{IK}; c = \text{insec} \vee c = \text{confid} \vee A \in \text{bad} \vee B \in \text{bad} \rrbracket$   
 $\Longrightarrow \text{Chan } c \ A \ B \ M \in \text{fake bad IK chan}$

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

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

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

**lemma** *fake-mono-bad* [intro]:

$\text{bad} \subseteq \text{bad}' \Longrightarrow \text{fake bad IK chan} \subseteq \text{fake bad}' \text{ IK chan}$

**by** (*auto*)

**lemma** *fake-mono-ik* [intro]:

$\text{IK} \subseteq \text{IK}' \Longrightarrow \text{fake bad IK chan} \subseteq \text{fake bad IK}' \text{ chan}$

**by** (*auto*)

**lemma** *fake-mono-chan* [intro]:

$\text{chan} \subseteq \text{chan}' \Longrightarrow \text{fake bad IK chan} \subseteq \text{fake bad IK chan}'$

**by** (*auto*)

**lemma** *fake-mono* [intro]:

$\llbracket \text{bad} \subseteq \text{bad}'; \text{IK} \subseteq \text{IK}'; \text{chan} \subseteq \text{chan}' \rrbracket \Longrightarrow \text{fake bad IK chan} \subseteq \text{fake bad}' \text{ IK}' \text{ chan}'$

**by** (*auto, erule fake.cases, auto*)

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

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

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

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

**lemma** *chan-subset-fake*:  $\text{chan} \subseteq \text{fake bad IK chan}$

**by** *auto*

**lemma** *extr-fake*:

$X \in \text{fake bad IK chan} \Longrightarrow \text{extr bad IK}' \{X\} \subseteq \text{IK} \cup \text{extr bad IK}' \text{ chan}$

**by** *auto*

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

**lemma** *fake-parts-extr-singleton*:

$X \in \text{fake bad IK chan} \Longrightarrow \text{parts (extr bad IK}' \{X\}) \subseteq \text{parts IK} \cup \text{parts (extr bad IK}' \text{ chan)}$

**by** (*rule extr-fake* [THEN *parts-mono, simplified*])

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

**lemma** *fake-parts-extr-insert*:

**assumes**  $X \in \text{fake bad IK CH}$

**shows**  $\text{parts (extr bad IK' (insert X CH))} \subseteq \text{parts (extr bad IK' CH)} \cup \text{parts IK}$

**proof** –

**have**  $\text{parts (extr bad IK' (insert X CH))} \subseteq \text{parts (extr bad IK' \{X\})} \cup \text{parts (extr bad IK' CH)}$

**by** (*auto simp: extr-insert-chan-eq* [where CH=CH])

**also have**  $\dots \subseteq \text{parts (extr bad IK' CH)} \cup \text{parts IK}$  **using** *assms*

**by** (*auto dest!: fake-parts-extr-singleton*)

**finally show** *?thesis* .

**qed**

**lemma** *fake-synth-analz-extr*:

**assumes**  $X \in \text{fake bad (synth (analz (extr bad IK CH))) CH}$

**shows**  $\text{synth (analz (extr bad IK (insert X CH)))} = \text{synth (analz (extr bad IK CH))}$

**using** *assms*

**proof** (*intro equalityI*)

**have**  $\text{synth (analz (extr bad IK (insert X CH)))}$

$\subseteq \text{synth (analz (extr bad IK \{X\} \cup extr bad IK CH))}$

**by** – (*rule synth-analz-mono, auto*)

**also have**  $\dots \subseteq \text{synth (analz (synth (analz (extr bad IK CH)) \cup extr bad IK CH))}$  **using** *assms*

**by** – (*rule synth-analz-mono, auto*)

**also have**  $\dots \subseteq \text{synth (analz (synth (analz (extr bad IK CH))))}$

**by** – (*rule synth-analz-mono, auto*)

**also have**  $\dots \subseteq \text{synth (analz (extr bad IK CH))}$  **by** *simp*

**finally show**  $\text{synth (analz (extr bad IK (insert X CH)))} \subseteq \text{synth (analz (extr bad IK CH))}$  .

**next**

**have**  $\text{extr bad IK CH} \subseteq \text{extr bad IK (insert X CH)}$

**by** *auto*

**then show**  $\text{synth (analz (extr bad IK CH))} \subseteq \text{synth (analz (extr bad IK (insert X CH)))}$

**by** – (*rule synth-analz-mono, auto*)

**qed**

## 10.4 Closure of Dolev-Yao, extract and fake

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

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

**definition**

$\text{dy-fake-msg} :: \text{agent set} \Rightarrow \text{msg set} \Rightarrow \text{chan set} \Rightarrow \text{msg set}$

**where**

$\text{dy-fake-msg } b \ i \ c = \text{synth (analz (extr } b \ i \ c))$

**lemma** *dy-fake-msg-empty* [*simp*]:  $\text{dy-fake-msg bad \{ \} \{ \}} = \text{synth \{ \}}$

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

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

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

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

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

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

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

**lemma** *dy-fake-msg-insert* [intro]:

$M \in dy\text{-fake-msg } bad \ I \ C \implies M \in dy\text{-fake-msg } bad \ I \ (insert \ X \ C)$

**by** (*auto*)

**lemma** *dy-fake-msg-mono* [intro]:

$\llbracket b \subseteq b'; I \subseteq I'; C \subseteq C' \rrbracket \implies dy\text{-fake-msg } b \ I \ C \subseteq dy\text{-fake-msg } b' \ I' \ C'$

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

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

**lemma** *dy-fake-msg-insert-chan*:

$x = insec \vee x = auth \implies$

$M \in dy\text{-fake-msg } bad \ IK \ (insert \ (Chan \ x \ A \ B \ M) \ CH)$

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

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

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

**definition**

*dy-fake-chan* :: *agent set*  $\Rightarrow$  *msg set*  $\Rightarrow$  *chan set*  $\Rightarrow$  *chan set*

**where**

*dy-fake-chan* *b i c* = *fake b (dy-fake-msg b i c) c*

**lemma** *dy-fake-chan-mono-bad* [intro]:

$bad \subseteq bad' \implies dy\text{-fake-chan } bad \ I \ C \subseteq dy\text{-fake-chan } bad' \ I \ C$

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

**lemma** *dy-fake-chan-mono-ik* [intro]:

$T \subseteq T' \implies dy\text{-fake-chan } bad \ T \ C \subseteq dy\text{-fake-chan } bad \ T' \ C$

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

**lemma** *dy-fake-chan-mono-chan* [intro]:

$C \subseteq C' \implies dy\text{-fake-chan } bad \ T \ C \subseteq dy\text{-fake-chan } bad \ T \ C'$

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

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

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

**lemma** *dy-fake-chan-mono* [intro]:  
 assumes  $b \subseteq b'$  and  $I \subseteq I'$  and  $C \subseteq C'$   
 shows *dy-fake-chan*  $b I C \subseteq$  *dy-fake-chan*  $b' I' C'$   
**proof** –  
 have *dy-fake-chan*  $b I C \subseteq$  *dy-fake-chan*  $b' I C$  **using**  $\langle b \subseteq b' \rangle$  **by** *auto*  
 also have  $\dots \subseteq$  *dy-fake-chan*  $b' I' C$  **using**  $\langle I \subseteq I' \rangle$  **by** *auto*  
 also have  $\dots \subseteq$  *dy-fake-chan*  $b' I' C'$  **using**  $\langle C \subseteq C' \rangle$  **by** *auto*  
 finally show ?thesis .  
**qed**

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

**lemma** *dy-fake-msg-subset-synth-analz*:  
 $\llbracket \text{extr bad IK chan} \subseteq T \rrbracket \implies$  *dy-fake-msg* *bad IK chan*  $\subseteq$  *synth* (*analz*  $T$ )  
**by** (*auto simp add: dy-fake-msg-def synth-analz-mono*)

**lemma** *dy-fake-chan-mono2*:  
 $\llbracket \text{extr bad IK chan} \subseteq \text{synth}(\text{analz } y); \text{chan} \subseteq \text{fake bad}(\text{synth}(\text{analz } y)) z \rrbracket$   
 $\implies$  *dy-fake-chan* *bad IK chan*  $\subseteq$  *fake bad* (*synth* (*analz*  $y$ ))  $z$   
**apply** (*auto simp add: dy-fake-chan-def, erule fake.cases, auto*)  
**apply** (*auto intro!: fake-New dest!: dy-fake-msg-subset-synth-analz*)  
**done**

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

**lemma** *dy-fake-chan-extr-insert*:  
 $M \in$  *dy-fake-chan* *bad IK CH*  $\implies$  *extr bad IK* (*insert*  $M CH$ )  $\subseteq$  *dy-fake-msg* *bad IK CH*  
**by** (*auto simp add: dy-fake-chan-def dy-fake-msg-def dest: fake-synth-analz-extr*)

**lemma** *dy-fake-chan-extr-insert-parts*:  
 $M \in$  *dy-fake-chan* *bad IK CH*  $\implies$   
 $\text{parts}(\text{extr bad IK}(\text{insert } M CH)) \subseteq \text{parts}(\text{extr bad IK } CH) \cup \text{dy-fake-msg bad IK } CH$   
**by** (*drule dy-fake-chan-extr-insert [THEN parts-mono], auto simp add: dy-fake-msg-def*)

**lemma** *dy-fake-msg-extr*:  
 $\text{extr bad ik chan} \subseteq \text{synth}(\text{analz } X) \implies$  *dy-fake-msg* *bad ik chan*  $\subseteq$  *synth* (*analz*  $X$ )  
**by** (*drule synth-analz-mono*) (*auto simp add: dy-fake-msg-def*)

**lemma** *extr-insert-dy-fake-msg*:  
 $M \in$  *dy-fake-msg* *bad IK CH*  $\implies$  *extr bad* (*insert*  $M IK$ )  $CH \subseteq$  *dy-fake-msg* *bad IK CH*  
**by** (*auto simp add: dy-fake-msg-def*)

**lemma** *dy-fake-msg-insert-dy-fake-msg*:  
 $M \in$  *dy-fake-msg* *bad IK CH*  $\implies$  *dy-fake-msg* *bad* (*insert*  $M IK$ )  $CH \subseteq$  *dy-fake-msg* *bad IK CH*  
**by** (*drule synth-analz-mono [OF extr-insert-dy-fake-msg], auto simp add: dy-fake-msg-def*)

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



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

**lemma** Fake-insert-dy-fake-msg:

$M \in \text{dy-fake-msg bad IK CH} \implies$   
 $\text{extr bad IK CH} \subseteq \text{synth (analz X)} \implies$   
 $\text{synth (analz (insert M IK))} \subseteq \text{synth (analz X)}$

**by** (auto dest!: synth-analz-insert-dy-fake-msg dy-fake-msg-extr)

**lemma** dy-fake-chan-insert-chan:

$x = \text{insec} \vee x = \text{auth} \implies$

$\text{Chan } x \ A \ B \ M \in \text{dy-fake-chan bad IK (insert (Chan } x \ A \ B \ M) \ CH)$

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

**lemma** dy-fake-chan-subset:

$\text{CH} \subseteq \text{fake bad (dy-fake-msg bad IK CH)} \ \text{CH}' \implies$

$\text{dy-fake-chan bad IK CH} \subseteq \text{fake bad (dy-fake-msg bad IK CH)} \ \text{CH}'$

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

**end**

## 11 Payloads and Support for Channel Message Implementations

Definitions and lemmas that do not require the implementations.

```
theory Payloads
imports Message-derivation
begin
```

### 11.1 Payload messages

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

Define set of payloads for basic messages.

```
inductive-set cpayload :: cmsg set where
  cAgent A ∈ cpayload
| cNumber T ∈ cpayload
| cNonce N ∈ cpayload
| cEphK K ∈ cpayload
| X ∈ cpayload ⇒ cHash X ∈ cpayload
| [ X ∈ cpayload; Y ∈ cpayload ] ⇒ cPair X Y ∈ cpayload
| [ X ∈ cpayload; Y ∈ cpayload ] ⇒ cEnc X Y ∈ cpayload
| [ X ∈ cpayload; Y ∈ cpayload ] ⇒ cAenc X Y ∈ cpayload
| [ X ∈ cpayload; Y ∈ cpayload ] ⇒ cSign X Y ∈ cpayload
| [ X ∈ cpayload; Y ∈ cpayload ] ⇒ cExp X Y ∈ cpayload
```

Lift *cpayload* to the quotiented message type.

**lift-definition** *payload* :: msg set is *cpayload* by –

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

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

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

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

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

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

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

```

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

```

```

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

```

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

```

declare cpayload.cases[elim]
lemma payload-Tag: Tag X  $\in$  payload  $\implies$  P
apply (auto simp add: payload-def msg-defs msg.abs-eq-iff eq-sym)
apply (auto dest!: eq-cpayload simp add: abs-cpayload-rep)
done

lemma payload-LtK: LtK X  $\in$  payload  $\implies$  P
apply (auto simp add: payload-def msg-defs msg.abs-eq-iff eq-sym)
apply (auto dest!: eq-cpayload simp add: abs-cpayload-rep)
done

lemma payload-Hash: Hash X  $\in$  payload  $\implies$  (X  $\in$  payload  $\implies$  P)  $\implies$  P
apply (auto simp add: payload-def msg-defs msg.abs-eq-iff eq-sym)
apply (auto dest!: eq-cpayload simp add: abs-cpayload-rep)
done

lemma payload-Pair: Pair X Y  $\in$  payload  $\implies$  (X  $\in$  payload  $\implies$  Y  $\in$  payload  $\implies$  P)  $\implies$  P
apply (auto simp add: payload-def msg-defs msg.abs-eq-iff eq-sym)
apply (auto dest!: eq-cpayload simp add: abs-cpayload-rep)
done

lemma payload-Enc: Enc X Y  $\in$  payload  $\implies$  (X  $\in$  payload  $\implies$  Y  $\in$  payload  $\implies$  P)  $\implies$  P
apply (auto simp add: payload-def msg-defs msg.abs-eq-iff eq-sym)
apply (auto dest!: eq-cpayload simp add: abs-cpayload-rep)
done

lemma payload-Aenc: Aenc X Y  $\in$  payload  $\implies$  (X  $\in$  payload  $\implies$  Y  $\in$  payload  $\implies$  P)  $\implies$  P
apply (auto simp add: payload-def msg-defs msg.abs-eq-iff eq-sym)
apply (auto dest!: eq-cpayload simp add: abs-cpayload-rep)
done

```

**lemma** *payload-Sign*:  $Sign\ X\ Y \in payload \implies (X \in payload \implies Y \in payload \implies P) \implies P$   
**apply** (*auto simp add: payload-def msg-defs msg.abs-eq-iff eq-sym*)  
**apply** (*auto dest!: eq-cpayload simp add: abs-cpayload-rep*)

**done**

**lemma** *payload-Exp*:  $Exp\ X\ Y \in payload \implies (X \in payload \implies Y \in payload \implies P) \implies P$

**apply** (*auto simp add: payload-def Exp-def msg.abs-eq-iff eq-sym*)

**apply** (*auto dest!: eq-cpayload simp add: abs-cpayload-rep*)

**done**

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

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

**lemmas** *payload-inductive-cases* =

*payload-Tag payload-LtK payload-Hash*

*payload-Pair payload-Enc payload-Aenc payload-Sign payload-Exp*

**lemma** *eq-exhaust*:

$(\bigwedge x. eq\ y\ (cAgent\ x) \implies P) \implies$

$(\bigwedge x. eq\ y\ (cNumber\ x) \implies P) \implies$

$(\bigwedge x. eq\ y\ (cNonce\ x) \implies P) \implies$

$(\bigwedge x. eq\ y\ (cLtK\ x) \implies P) \implies$

$(\bigwedge x. eq\ y\ (cEphK\ x) \implies P) \implies$

$(\bigwedge x\ x'. eq\ y\ (cPair\ x\ x') \implies P) \implies$

$(\bigwedge x\ x'. eq\ y\ (cEnc\ x\ x') \implies P) \implies$

$(\bigwedge x\ x'. eq\ y\ (cAenc\ x\ x') \implies P) \implies$

$(\bigwedge x\ x'. eq\ y\ (cSign\ x\ x') \implies P) \implies$

$(\bigwedge x. eq\ y\ (cHash\ x) \implies P) \implies$

$(\bigwedge x. eq\ y\ (cTag\ x) \implies P) \implies$

$(\bigwedge x\ x'. eq\ y\ (cExp\ x\ x') \implies P) \implies$

$P$

**apply** (*cases y*)

**apply** (*meson Messages.eq-refl*)<sup>+</sup>

**done**

**lemma** *msg-exhaust*:

$(\bigwedge x. y = Agent\ x \implies P) \implies$

$(\bigwedge x. y = Number\ x \implies P) \implies$

$(\bigwedge x. y = Nonce\ x \implies P) \implies$

$(\bigwedge x. y = LtK\ x \implies P) \implies$

$(\bigwedge x. y = EphK\ x \implies P) \implies$

$(\bigwedge x\ x'. y = Pair\ x\ x' \implies P) \implies$

$(\bigwedge x\ x'. y = Enc\ x\ x' \implies P) \implies$

$(\bigwedge x\ x'. y = Aenc\ x\ x' \implies P) \implies$

$(\bigwedge x\ x'. y = Sign\ x\ x' \implies P) \implies$

$(\bigwedge x. y = Hash\ x \implies P) \implies$

$(\bigwedge x. y = Tag\ x \implies P) \implies$

$(\bigwedge x\ x'. y = Exp\ x\ x' \implies P) \implies$

$P$

**apply** *transfer*

**apply** (*erule eq-exhaust, auto*)

**done**

**lemma** *payload-cases*:

$a \in \text{payload} \implies$   
 $(\wedge A. a = \text{Agent } A \implies P) \implies$   
 $(\wedge T. a = \text{Number } T \implies P) \implies$   
 $(\wedge N. a = \text{Nonce } N \implies P) \implies$   
 $(\wedge K. a = \text{EphK } K \implies P) \implies$   
 $(\wedge X. a = \text{Hash } X \implies X \in \text{payload} \implies P) \implies$   
 $(\wedge X Y. a = \text{Pair } X Y \implies X \in \text{payload} \implies Y \in \text{payload} \implies P) \implies$   
 $(\wedge X Y. a = \text{Enc } X Y \implies X \in \text{payload} \implies Y \in \text{payload} \implies P) \implies$   
 $(\wedge X Y. a = \text{Aenc } X Y \implies X \in \text{payload} \implies Y \in \text{payload} \implies P) \implies$   
 $(\wedge X Y. a = \text{Sign } X Y \implies X \in \text{payload} \implies Y \in \text{payload} \implies P) \implies$   
 $(\wedge X Y. a = \text{Exp } X Y \implies X \in \text{payload} \implies Y \in \text{payload} \implies P) \implies$   
 $P$

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

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

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

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

**lemma** *payload-parts* [*simp, dest*]:

$\llbracket X \in \text{parts } S; S \subseteq \text{payload} \rrbracket \implies X \in \text{payload}$

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

**lemma** *payload-parts-singleton* [*simp, dest*]:

$\llbracket X \in \text{parts } \{Y\}; Y \in \text{payload} \rrbracket \implies X \in \text{payload}$

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

**lemma** *payload-analz* [*simp, dest*]:

$\llbracket X \in \text{analz } S; S \subseteq \text{payload} \rrbracket \implies X \in \text{payload}$

**by** (*auto dest: analz-into-parts*)

**lemma** *payload-synth-analz*:

$\llbracket X \in \text{synth } (\text{analz } S); S \subseteq \text{payload} \rrbracket \implies X \in \text{payload}$

**by** (*erule synth.induct*) (*auto intro: payload-analz*)

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

**lemma** *synth-payload*:

$Y \cap \text{payload} = \{\} \implies \text{synth } (X \cup Y) \subseteq \text{synth } X \cup \text{-payload}$

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

**lemma** *synth-payload2*:

$Y \cap \text{payload} = \{\} \implies \text{synth } (Y \cup X) \subseteq \text{synth } X \cup \text{-payload}$

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

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

**lemma** *synth-idem-payload*:

$X \subseteq \text{synth } Y \cup \text{-payload} \implies \text{synth } X \subseteq \text{synth } Y \cup \text{-payload}$

**by** (*auto dest: synth-mono subset-trans* [*OF - synth-payload*])

## 11.2 *isLtKey*: is a long term key

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

**by** (*auto*)

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

**by** (*auto*)

**lemma** *LtKeys-parts-payload-singleton* [*elim*]:  $X \in \text{payload} \implies LtK Y \in \text{parts } \{X\} \implies \text{False}$

**by** (*auto*)

**lemma** *parts-of-LtKeys* [*simp*]:  $K \subseteq \text{range } LtK \implies \text{parts } K = K$

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

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

**definition**

*keys-of* :: *agent*  $\Rightarrow$  *msg set*

**where**

*keys-of* *A*  $\equiv$  *insert* (*priK* *A*) {*shrK* *B* *C* | *B* *C*. *B* = *A*  $\vee$  *C* = *A*}

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

**by** (*auto simp add: keys-of-def*)

**lemma** *priK-keys-of* [*intro!*]:

*priK* *A*  $\in$  *keys-of* *A*

**by** (*simp add: keys-of-def*)

**lemma** *shrK-keys-of-1* [*intro!*]:

*shrK* *A* *B*  $\in$  *keys-of* *A*

**by** (*simp add: keys-of-def*)

**lemma** *shrK-keys-of-2* [*intro!*]:

*shrK* *B* *A*  $\in$  *keys-of* *A*

**by** (*simp add: keys-of-def*)

**lemma** *priK-keys-of-eq* [*dest*]:

*priK* *B*  $\in$  *keys-of* *A*  $\implies A = B$

**by** (*simp add: keys-of-def*)

**lemma** *shrK-keys-of-eq* [*dest*]:

*shrK* *A* *B*  $\in$  *keys-of* *C*  $\implies A = C \vee B = C$

**by** (*simp add: keys-of-def*)

**lemma** *def-keys-of* [*dest*]:

$K \in \text{keys-of } A \implies K = \text{priK } A \vee (\exists B. K = \text{shrK } A B \vee K = \text{shrK } B A)$

**by** (*auto simp add: keys-of-def*)

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

**by** (*auto intro!: parts-of-LtKeys*)

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

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

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

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

### definition

$Keys\text{-}bad :: msg\ set \Rightarrow agent\ set \Rightarrow bool$

### where

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

— basic lemmas

### lemma *Keys-badI*:

$\llbracket IK \cap range\ LtK \subseteq range\ pubK \cup priK\ 'Bad \cup \{shrK\ A\ B \mid A\ B.\ A \in Bad \vee B \in Bad\};$   
 $range\ pubK \subseteq IK \rrbracket$   
 $\Longrightarrow Keys\text{-}bad\ IK\ Bad$

**by** (*auto simp add: Keys-bad-def*)

### lemma *Keys-badE* [*elim*]:

$\llbracket Keys\text{-}bad\ IK\ Bad;$   
 $\llbracket range\ pubK \subseteq IK;$   
 $IK \cap range\ LtK \subseteq range\ pubK \cup \bigcup (keys\text{-}of\ 'Bad) \rrbracket$   
 $\Longrightarrow P \rrbracket$   
 $\Longrightarrow P$

**by** (*auto simp add: Keys-bad-def*)

### lemma *Keys-bad-Ltk* [*simp*]:

$Keys\text{-}bad\ (IK \cap range\ LtK)\ Bad \longleftrightarrow Keys\text{-}bad\ IK\ Bad$

**by** (*auto simp add: Keys-bad-def*)

### lemma *Keys-bad-priK-D*: $\llbracket priK\ A \in IK; Keys\text{-}bad\ IK\ Bad \rrbracket \Longrightarrow A \in Bad$

**by** (*auto simp add: Keys-bad-def*)

### lemma *Keys-bad-shrK-D*: $\llbracket shrK\ A\ B \in IK; Keys\text{-}bad\ IK\ Bad \rrbracket \Longrightarrow A \in Bad \vee B \in Bad$

**by** (*auto simp add: Keys-bad-def*)

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

interaction with *insert*.

### lemma *Keys-bad-insert-non-LtK*:

$X \notin range\ LtK \Longrightarrow Keys\text{-}bad\ (insert\ X\ IK)\ Bad \longleftrightarrow Keys\text{-}bad\ IK\ Bad$

**by** (*auto simp add: Keys-bad-def*)

### lemma *Keys-bad-insert-pubK*:

$\llbracket Keys\text{-}bad\ IK\ Bad \rrbracket \Longrightarrow Keys\text{-}bad\ (insert\ (pubK\ A)\ IK)\ Bad$

**by** (*auto simp add: Keys-bad-def*)

### lemma *Keys-bad-insert-priK-bad*:

$\llbracket Keys\text{-}bad\ IK\ Bad; A \in Bad \rrbracket \Longrightarrow Keys\text{-}bad\ (insert\ (priK\ A)\ IK)\ Bad$

**by** (*auto simp add: Keys-bad-def*)

**lemma** *Keys-bad-insert-shrK-bad*:  
 $\llbracket \text{Keys-bad IK Bad}; A \in \text{Bad} \vee B \in \text{Bad} \rrbracket \implies \text{Keys-bad } (\text{insert } (\text{shrK } A \ B) \ \text{IK}) \ \text{Bad}$   
**by** (*auto simp add: Keys-bad-def*)

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

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

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

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

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

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

### definition

*broken* :: *msg set  $\implies$  (agent \* agent) set*

### where

*broken K  $\equiv$  {(A,B) | A B. priK A  $\in$  K  $\vee$  priK B  $\in$  K  $\vee$  shrK A B  $\in$  K  $\vee$  shrK B A  $\in$  K}*

**lemma** *brokenD* [*dest!*]:  
 $(A, B) \in \text{broken } K \implies \text{priK } A \in K \vee \text{priK } B \in K \vee \text{shrK } A \ B \in K \vee \text{shrK } B \ A \in K$   
**by** (*simp add: broken-def*)

**lemma** *brokenI* [*intro!*]:  
 $\text{priK } A \in K \vee \text{priK } B \in K \vee \text{shrK } A \ B \in K \vee \text{shrK } B \ A \in K \implies (A, B) \in \text{broken } K$   
**by** (*auto simp add: broken-def*)



## 11.6 *Enc-keys-clean S*: messages with “clean” symmetric encryptions.

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

### definition

$Enc\text{-}keys\text{-}clean :: msg\ set \Rightarrow bool$

### where

$Enc\text{-}keys\text{-}clean\ S \equiv \forall X\ Y. Enc\ X\ Y \in parts\ S \longrightarrow Y \in range\ LtK \cup payload$

### lemma *Enc-keys-cleanI*:

$\forall X\ Y. Enc\ X\ Y \in parts\ S \longrightarrow Y \in range\ LtK \cup payload \Longrightarrow Enc\text{-}keys\text{-}clean\ S$

by (*simp add: Enc-keys-clean-def*)

— general lemmas about *Enc-keys-clean*

### lemma *Enc-keys-clean-mono*:

$Enc\text{-}keys\text{-}clean\ H \Longrightarrow G \subseteq H \Longrightarrow Enc\text{-}keys\text{-}clean\ G$  — anti-tone

by (*auto simp add: Enc-keys-clean-def dest!: parts-monotone [where G=G]*)

### lemma *Enc-keys-clean-Un [simp]*:

$Enc\text{-}keys\text{-}clean\ (G \cup H) \longleftrightarrow Enc\text{-}keys\text{-}clean\ G \wedge Enc\text{-}keys\text{-}clean\ H$

by (*auto simp add: Enc-keys-clean-def*)

— from *Enc-keys-clean S*, the property on *parts S* also holds for *analz S*

### lemma *Enc-keys-clean-analz*:

$Enc\ X\ K \in analz\ S \Longrightarrow Enc\text{-}keys\text{-}clean\ S \Longrightarrow K \in range\ LtK \cup payload$

by (*auto simp add: Enc-keys-clean-def dest: analz-into-parts*)

— *Enc-keys-clean* and different types of messages

### lemma *Enc-keys-clean-Tags [simp,intro]*: $Enc\text{-}keys\text{-}clean\ Tags$

by (*auto simp add: Enc-keys-clean-def*)

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

by (*auto simp add: Enc-keys-clean-def*)

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

by (*auto simp add: Enc-keys-clean-def*)

## 11.7 Sets of messages with particular constructors

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

### abbreviation *AgentSet* :: $msg\ set$

where  $AgentSet \equiv range\ Agent$

### abbreviation *PairSet* :: $msg\ set \Rightarrow msg\ set \Rightarrow msg\ set$

where  $PairSet\ G\ H \equiv \{Pair\ X\ Y \mid X\ Y. X \in G \wedge Y \in H\}$

### abbreviation *EncSet* :: $msg\ set \Rightarrow msg\ set \Rightarrow msg\ set$

where  $EncSet\ G\ K \equiv \{Enc\ X\ Y \mid X\ Y. X \in G \wedge Y \in K\}$

### abbreviation *AencSet* :: $msg\ set \Rightarrow msg\ set \Rightarrow msg\ set$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**lemma** *Enc-keys-clean-HashSet-Un:*

*Enc-keys-clean (G ∪ H) ⇒ Enc-keys-clean (HashSet G ∪ H)*

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

**lemmas** *Enc-keys-clean-msgSet-Un =*

*Enc-keys-clean-PairSet-Tag-Un Enc-keys-clean-PairSet-Agent-Un*

*Enc-keys-clean-EncSet-Un Enc-keys-clean-AencSet-Un*

*Enc-keys-clean-SignSet-Un Enc-keys-clean-HashSet-Un*

### 11.7.1 Lemmas for moving message sets out of *analz*

Pull *EncSet* out of *analz*.

**lemma** *analz-Un-EncSet:*

**assumes**  $K \subseteq \text{range LtK}$  **and** *Enc-keys-clean (G ∪ H)*

**shows**  $\text{analz (EncSet G K} \cup H) \subseteq \text{EncSet G K} \cup \text{analz (G} \cup H)$

**proof**

**fix**  $X$

**assume**  $X \in \text{analz (EncSet G K} \cup H)$

**thus**  $X \in \text{EncSet G K} \cup \text{analz (G} \cup H)$

**proof** (*induction X rule: analz.induct*)

**case** (*Dec Y K'*)

**from** *Dec.IH(1)* **show** *?case*

**proof**

**assume**  $\text{Enc Y K}' \in \text{analz (G} \cup H)$

**have**  $K' \in \text{synth (analz (G} \cup H))$

**proof** –

**have**  $K' \in \text{range LtK} \cup \text{payload}$  **using**  $\langle \text{Enc Y K}' \in \text{analz (G} \cup H) \rangle \text{ assms(2)}$

**by** (*blast dest: Enc-keys-clean-analz*)

**moreover**

**have**  $K' \in \text{synth (EncSet G K} \cup \text{analz (G} \cup H))$  **using** *Dec.IH(2)*

**by** (*auto simp add: Collect-disj-eq dest: synth-Int2*)

**moreover**

**hence**  $K' \in \text{synth (analz (G} \cup H)) \cup \text{–payload}$  **using** *assms(1)*

**by** (*blast dest!: synth-payload2 [THEN [2] rev-subsetD]*)

**ultimately show** *?thesis* **by** *auto*

**qed**

**thus** *?case* **using**  $\langle \text{Enc Y K}' \in \text{analz (G} \cup H) \rangle$  **by** *auto*

**qed** *auto*

**next**

**case** (*Adec-eph Y K'*)

**thus** *?case* **by** (*auto dest!: EpriK-synth*)

**qed** (*auto*)

**qed**

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

**lemma** *analz-Un-EncSet2:*

**assumes** *Enc-keys-clean H* **and**  $K \subseteq \text{range LtK}$  **and**  $K \cap \text{synth (analz H)} = \{\}$

**shows**  $\text{analz (EncSet G K} \cup H) \subseteq \text{EncSet G K} \cup \text{analz H}$

**proof**

**fix**  $X$

```

assume  $X \in \text{analz} (\text{EncSet } G \ K \cup H)$ 
thus  $X \in \text{EncSet } G \ K \cup \text{analz } H$ 
proof (induction X rule: analz.induct)
  case ( $\text{Dec } Y \ K'$ )
  from  $\text{Dec.IH}(1)$  show  $?case$ 
  proof
    assume  $\text{Enc } Y \ K' \in \text{analz } H$ 
    moreover have  $K' \in \text{synth} (\text{analz } H)$ 
    proof –
      have  $K' \in \text{range } \text{LtK} \cup \text{payload}$  using  $\langle \text{Enc } Y \ K' \in \text{analz } H \rangle \text{ assms}(1)$ 
      by (auto dest: Enc-keys-clean-analz)
      moreover
      from  $\text{Dec.IH}(2)$  have  $H: K' \in \text{synth} (\text{EncSet } G \ K \cup \text{analz } H)$ 
      by (auto simp add: Collect-disj-eq dest: synth-Int2)
      moreover
      hence  $K' \in \text{synth} (\text{analz } H) \cup \text{–payload}$ 
      proof (rule synth-payload2 [THEN [2] rev-subsetD], auto elim!: payload-Enc)
        fix  $X \ Y$ 
        assume  $Y \in K \ Y \in \text{payload}$ 
        with  $\langle K \subseteq \text{range } \text{LtK} \rangle$  obtain  $KK$  where  $Y = \text{LtK } KK$  by auto
        with  $\langle Y \in \text{payload} \rangle$  show  $\text{False}$  by auto
      qed
    ultimately
    show  $?thesis$  by auto
  qed
  ultimately show  $?case$  by auto
next
  assume  $\text{Enc } Y \ K' \in \text{EncSet } G \ K$ 
  moreover hence  $K' \in K$  by auto
  moreover with  $\langle K \subseteq \text{range } \text{LtK} \rangle$  obtain  $KK$  where  $K' = \text{LtK } KK$  by auto
  moreover with  $\text{Dec.IH}(2)$  have  $K' \in \text{analz } H$ 
  by (auto simp add: Collect-disj-eq dest: synth-Int2)
  ultimately show  $?case$  using  $\langle K \cap \text{synth} (\text{analz } H) = \{\} \rangle$  by auto
qed
next
  case ( $\text{Adec-eph } Y \ K'$ )
  thus  $?case$  by (auto dest!: EpriK-synth)
qed (insert assms(2), auto)
qed

```

Pull  $\text{AencSet}$  out of the  $\text{analz}$ .

**lemma** *analz-Un-AencSet*:

**assumes**  $K \subseteq \text{range } \text{LtK}$  **and**  $\text{Enc-keys-clean } (G \cup H)$   
**shows**  $\text{analz} (\text{AencSet } G \ K \cup H) \subseteq \text{AencSet } G \ K \cup \text{analz} (G \cup H)$

```

proof
  fix  $X$ 
  assume  $X \in \text{analz} (\text{AencSet } G \ K \cup H)$ 
  thus  $X \in \text{AencSet } G \ K \cup \text{analz} (G \cup H)$ 
  proof (induction X rule: analz.induct)
    case ( $\text{Dec } Y \ K'$ )
    from  $\text{Dec.IH}(1)$  have  $\text{Enc } Y \ K' \in \text{analz} (G \cup H)$  by auto
    moreover have  $K' \in \text{synth} (\text{analz} (G \cup H))$ 
    proof –

```

```

have  $K' \in \text{range } LtK \cup \text{payload}$  using  $\langle \text{Enc } Y \ K' \in \text{analz } (G \cup H) \rangle \text{ assms}(2)$ 
  by (blast dest: Enc-keys-clean-analz)
moreover
have  $K' \in \text{synth } (AencSet \ G \ K \cup \text{analz } (G \cup H))$  using Dec.IH(2)
  by (auto simp add: Collect-disj-eq dest: synth-Int2)
moreover
hence  $K' \in \text{synth } (\text{analz } (G \cup H)) \cup \text{--payload}$  using assms(1)
  by (blast dest!: synth-payload2 [THEN [2] rev-subsetD])
ultimately
show ?thesis by auto
qed
ultimately show ?case by auto
next
  case (Adec-eph Y K')
  thus ?case by (auto dest!: EpriK-synth)
qed auto
qed

```

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

**lemma** *analz-Un-AencSet2*:

```

assumes Enc-keys-clean H and  $\text{pri}K'Ag \cap \text{synth } (\text{analz } H) = \{\}$ 
shows  $\text{analz } (AencSet \ G \ (\text{pub}K'Ag) \cup H) \subseteq AencSet \ G \ (\text{pub}K'Ag) \cup \text{analz } H$ 
proof
  fix  $X$ 
  assume  $X \in \text{analz } (AencSet \ G \ (\text{pub}K'Ag) \cup H)$ 
  thus  $X \in AencSet \ G \ (\text{pub}K'Ag) \cup \text{analz } H$ 
  proof (induction X rule: analz.induct)
    case (Dec Y K')
    from Dec.IH(1) have  $\text{Enc } Y \ K' \in \text{analz } H$  by auto
    moreover have  $K' \in \text{synth } (\text{analz } H)$ 
    proof –
      have  $K' \in \text{range } LtK \cup \text{payload}$  using  $\langle \text{Enc } Y \ K' \in \text{analz } H \rangle \text{ assms}(1)$ 
      by (auto dest: Enc-keys-clean-analz)
      moreover
      from Dec.IH(2) have  $H: K' \in \text{synth } (AencSet \ G \ (\text{pub}K'Ag) \cup \text{analz } H)$ 
      by (auto simp add: Collect-disj-eq dest: synth-Int2)
      moreover
      hence  $K' \in \text{synth } (\text{analz } H) \cup \text{--payload}$ 
      by (auto dest: synth-payload2 [THEN [2] rev-subsetD])
      ultimately
      show ?thesis by auto
    qed
    ultimately show ?case by auto
  next
    case (Adec-eph Y K')
    thus ?case by (auto dest!: EpriK-synth)
  qed (insert assms(2), auto)
qed

```

Pull *PairSet* out of *analz*.

**lemma** *analz-Un-PairSet*:

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

```

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

```

— move the *SignSet* out of the *analz*

**lemma** *analz-Un-SignSet*:

**assumes**  $K \subseteq \text{range } \text{LtK}$  **and** *Enc-keys-clean*  $(G \cup H)$

**shows**  $\text{analz} (\text{SignSet } G \ K \cup H) \subseteq \text{SignSet } G \ K \cup \text{analz} (G \cup H)$

**proof**

```

  fix X
  assume  $X \in \text{analz} (\text{SignSet } G \ K \cup H)$ 
  thus  $X \in \text{SignSet } G \ K \cup \text{analz} (G \cup H)$ 
  proof (induct X rule: analz.induct)
    case (Dec Y K')
    from Dec.hyps(2) have  $\text{Enc } Y \ K' \in \text{analz} (G \cup H)$  by auto
    moreover have  $K' \in \text{synth} (\text{analz} (G \cup H))$ 
    proof –
      have  $K' \in \text{range } \text{LtK} \cup \text{payload}$  using  $\langle \text{Enc } Y \ K' \in \text{analz} (G \cup H) \rangle$  assms(2)
      by (blast dest: Enc-keys-clean-analz)
    moreover
    from Dec.hyps(3) have  $K' \in \text{synth} (\text{SignSet } G \ K \cup \text{analz} (G \cup H))$ 
    by (auto simp add: Collect-disj-eq dest: synth-Int2)
    moreover
    hence  $K' \in \text{synth} (\text{analz} (G \cup H)) \cup \text{–payload}$  using assms(1)
    by (blast dest!: synth-payload2 [THEN [2] rev-subsetD])
    ultimately
    show ?thesis by auto
  qed
  ultimately show ?case by auto
  next
  case (Adec-eph Y K)
  thus ?case by (auto dest!: EpriK-synth)
qed auto
qed

```

Pull *Tags* out of *analz*.

**lemma** *analz-Un-Tag*:

```

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

```

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

```

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

```

Pull *HashSet* out of *analz*.

**lemma** *analz-Un-HashSet*:

**assumes** *Enc-keys-clean H* **and**  $G \subseteq - \text{payload}$

**shows**  $\text{analz } (\text{HashSet } G \cup H) \subseteq \text{HashSet } G \cup \text{analz } H$

**proof**

**fix**  $X$

**assume**  $X \in \text{analz } (\text{HashSet } G \cup H)$

**thus**  $X \in \text{HashSet } G \cup \text{analz } H$

**proof** (*induction X rule: analz.induct*)

**case** (*Dec Y K'*)

**from** *Dec.IH(1)* **have**  $\text{Enc } Y \ K' \in \text{analz } H$  **by** *auto*

**moreover** **have**  $K' \in \text{synth } (\text{analz } H)$

**proof** –

**have**  $K' \in \text{range } \text{LtK} \cup \text{payload}$  **using**  $\langle \text{Enc } Y \ K' \in \text{analz } H \rangle$  *assms(1)*

**by** (*auto dest: Enc-keys-clean-analz*)

**thus** *?thesis*

**proof**

**assume**  $K' \in \text{range } \text{LtK}$

**then obtain**  $KK$  **where**  $K' = \text{LtK } KK$  **by** *auto*

**moreover**

**with** *Dec.IH(2)* **show** *?thesis*

**by** (*auto simp add: Collect-disj-eq dest: synth-Int2*)

**next**

**assume**  $K' \in \text{payload}$

**moreover**

**from** *assms* **have**  $\text{HashSet } G \cap \text{payload} = \{\}$  **by** *auto*

**moreover from** *Dec.IH(2)* **have**  $K' \in \text{synth } (\text{HashSet } G \cup \text{analz } H)$

**by** (*auto simp add: Collect-disj-eq dest: synth-Int2*)

**ultimately**

**have**  $K' \in \text{synth } (\text{analz } H) \cup -\text{payload}$

**by** (*auto dest: synth-payload2 [THEN [2] rev-subsetD]*)

**with**  $\langle K' \in \text{payload} \rangle$  **show** *?thesis* **by** *auto*

**qed**

**qed**

**ultimately show** *?case* **by** *auto*

**next**

**case** (*Adec-eph Y K'*)

**thus** *?case* **by** (*auto dest!: EpriK-synth*)

**qed** (*insert assms(2), auto*)

**qed**

**end**



## 12 Assumptions for Channel Message Implementation

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

```
theory Implem
imports Channels Payloads
begin
```

### 12.1 First step: basic implementation locale

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

```
locale basic-implem =
  fixes implem :: chan  $\Rightarrow$  msg
begin
```

```
abbreviation implInsec A B M  $\equiv$  implem (Insec A B M)
abbreviation implConfid A B M  $\equiv$  implem (Confid A B M)
abbreviation implAuth A B M  $\equiv$  implem (Auth A B M)
abbreviation implSecure A B M  $\equiv$  implem (Secure A B M)
```

```
abbreviation implInsecSet :: msg set  $\Rightarrow$  msg set
where implInsecSet G  $\equiv$  {implInsec A B M | A B M. M  $\in$  G}
```

```
abbreviation implConfidSet :: (agent * agent) set  $\Rightarrow$  msg set  $\Rightarrow$  msg set
where implConfidSet Ag G  $\equiv$  {implConfid A B M | A B M. (A, B)  $\in$  Ag  $\wedge$  M  $\in$  G}
```

```
abbreviation implAuthSet :: msg set  $\Rightarrow$  msg set
where implAuthSet G  $\equiv$  {implAuth A B M | A B M. M  $\in$  G}
```

```
abbreviation implSecureSet :: (agent * agent) set  $\Rightarrow$  msg set  $\Rightarrow$  msg set
where implSecureSet Ag G  $\equiv$  {implSecure A B M | A B M. (A, B)  $\in$  Ag  $\wedge$  M  $\in$  G}
```

#### definition

```
valid :: msg set
```

#### where

```
valid  $\equiv$  {implem (Chan x A B M) | x A B M. M  $\in$  payload}
```

#### lemma *validI*:

```
M  $\in$  payload  $\Longrightarrow$  implem (Chan x A B M)  $\in$  valid
```

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

#### lemma *validE*:

```
X  $\in$  valid  $\Longrightarrow$  ( $\bigwedge$  x A B M. X = implem (Chan x A B M)  $\Longrightarrow$  M  $\in$  payload  $\Longrightarrow$  P)  $\Longrightarrow$  P
```

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

#### lemma *valid-cases*:

```
fixes X P
```

```
assumes X  $\in$  valid
```

```
( $\bigwedge$  A B M. X = implInsec A B M  $\Longrightarrow$  M  $\in$  payload  $\Longrightarrow$  P)
```

```
( $\bigwedge$  A B M. X = implConfid A B M  $\Longrightarrow$  M  $\in$  payload  $\Longrightarrow$  P)
```

```
( $\bigwedge$  A B M. X = implAuth A B M  $\Longrightarrow$  M  $\in$  payload  $\Longrightarrow$  P)
```

$(\bigwedge A B M. X = \text{implSecure } A B M \implies M \in \text{payload} \implies P)$   
**shows**  $P$   
**proof** –  
**from** *assms* **have**  $(\bigwedge x A B M. X = \text{implem } (\text{Chan } x A B M) \implies M \in \text{payload} \implies P) \implies P$   
**by** (*auto elim: validE*)  
**moreover from** *assms* **have**  $\bigwedge x A B M. X = \text{implem } (\text{Chan } x A B M) \implies M \in \text{payload} \implies P$   
**proof** –  
**fix**  $x A B M$   
**assume**  $X = \text{implem } (\text{Chan } x A B M) M \in \text{payload}$   
**with** *assms* **show**  $P$  **by** (*cases x, auto*)  
**qed**  
**ultimately show** *?thesis* .  
**qed**  
**end**

## 12.2 Second step: basic and analyze assumptions

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

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

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

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

**begin**

— declare some attributes and abbreviations for the hypotheses  
— and prove some simple consequences of the hypotheses

**declare** *impl-inj* [*simp*]

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

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

**lemma** *composed-arg-cong*:  $X = Y \implies \text{composed } X \longleftrightarrow \text{composed } Y$

**by** (*rule arg-cong*)

**lemma** *implem-Tags-aux*:  $\text{implem } (\text{Chan } x \ A \ B \ M) \notin \text{Tags}$  **by** (*cases x, auto dest: composed-arg-cong*)

**lemma** *implem-Tags* [*simp*]:  $\text{implem } x \notin \text{Tags}$  **by** (*cases x, auto simp add: implem-Tags-aux*)

**lemma** *implem-LtK-aux*:  $\text{implem } (\text{Chan } x \ A \ B \ M) \neq LtK \ K$  **by** (*cases x, auto dest: composed-arg-cong*)

**lemma** *implem-LtK* [*simp*]:  $\text{implem } x \neq LtK \ K$  **by** (*cases x, auto simp add: implem-LtK-aux*)

**lemma** *implem-LtK2* [*simp*]:  $\text{implem } x \notin \text{range } LtK$  **by** (*cases x, auto simp add: implem-LtK-aux*)

**declare** *impl-Impl* [*simp*]

**lemma** *LtK-parts-impl-insert*:

$LtK \ K \in \text{parts } (\text{insert } (\text{implem } (\text{Chan } x \ A \ B \ M)) \ S) \implies M \in \text{payload} \implies LtK \ K \in \text{parts } S$

**apply** (*simp add: parts-insert [of - S], clarify*)

**apply** (*auto dest: validI LtK-parts-impl*)

**done**

**declare** *LtK-parts-impl-insert* [*dest*]

**declare** *Enc-keys-clean-valid* [*simp, intro*]

**lemma** *valid-composed* [*simp, dest*]:  $M \in \text{valid} \implies \text{composed } M$

**by** (*auto elim: validE*)

— lemmas: valid/payload are mutually exclusive

**lemma** *valid-payload* [*dest*]:  $\llbracket X \in \text{valid}; X \in \text{payload} \rrbracket \implies P$   
**by** (*auto elim!: validE*)

— *valid/LtK* are mutually exclusive

**lemma** *valid-isLtKey* [*dest*]:  $\llbracket X \in \text{valid}; X \in \text{range LtK} \rrbracket \implies P$   
**by** (*auto*)

**lemma** *analz-valid*:

$H \subseteq \text{payload} \cup \text{valid} \cup \text{range LtK} \cup \text{Tags} \implies$

$\text{implem } (\text{Chan } x \ A \ B \ M) \in \text{analz } H \implies$

$\text{implem } (\text{Chan } x \ A \ B \ M) \in H$

**apply** (*drule analz-into-parts*,

*drule parts-monotone [of - H payload  $\cup$  H  $\cap$  valid  $\cup$  range LtK  $\cup$  Tags], auto)*

**apply** (*drule parts-singleton, auto elim!:validE dest: parts-impl-inj*)

**done**

**lemma** *parts-valid-LtKeys-disjoint*:

$I \subseteq \text{valid} \implies \text{parts } I \cap \text{range LtK} = \{\}$

**apply** (*safe, drule parts-singleton, clarsimp*)

**apply** (*auto dest: subsetD LtK-parts-impl*)

**done**

**lemma** *analz-LtKeys*:

$H \subseteq \text{payload} \cup \text{valid} \cup \text{range LtK} \cup \text{Tags} \implies$

$\text{analz } H \cap \text{range LtK} \subseteq H$

**apply** *auto*

**apply** (*drule analz-into-parts, drule parts-monotone [of - H payload  $\cup$  valid  $\cup$  H  $\cap$  range LtK  $\cup$  Tags], auto*)

**apply** (*drule parts-singleton, auto elim!:validE dest: parts-impl-inj*)

**done**

**end**

### 12.3 Third step: *valid-implem*

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

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

— Synthesize assumptions: conditions on payloads  $M$  implied by derivable

— channel messages with payload  $M$ .

**assumes** *implInsec-synth-analz*:

$H \subseteq \text{payload} \cup \text{valid} \cup \text{range LtK} \cup \text{Tags} \implies$

$\text{implInsec } A \ B \ M \in \text{synth } (\text{analz } H) \implies$

$\text{implInsec } A \ B \ M \in H \vee M \in \text{synth } (\text{analz } H)$

**and** *implConfid-synth-analz*:

$H \subseteq \text{payload} \cup \text{valid} \cup \text{range LtK} \cup \text{Tags} \implies$

$\text{implConfid } A \ B \ M \in \text{synth } (\text{analz } H) \implies$

$\text{implConfid } A \ B \ M \in H \vee M \in \text{synth } (\text{analz } H)$

**and** *implAuth-synth-analz*:

$H \subseteq \text{payload} \cup \text{valid} \cup \text{range LtK} \cup \text{Tags} \implies$   
 $\text{implAuth } A \ B \ M \in \text{synth } (\text{analz } H) \implies$   
 $\text{implAuth } A \ B \ M \in H \vee (M \in \text{synth } (\text{analz } H) \wedge (A, B) \in \text{broken } H)$   
**and** *implSecure-synth-analz*:  
 $H \subseteq \text{payload} \cup \text{valid} \cup \text{range LtK} \cup \text{Tags} \implies$   
 $\text{implSecure } A \ B \ M \in \text{synth } (\text{analz } H) \implies$   
 $\text{implSecure } A \ B \ M \in H \vee (M \in \text{synth } (\text{analz } H) \wedge (A, B) \in \text{broken } H)$   
**end**

## 13 Lemmas Following from Channel Message Implementation Assumptions

```
theory Implem-lemmas
imports Implem
begin
```

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

```
context semivalid-implem
begin
```

### 13.1 Message implementations and abstractions

Abstracting a set of messages into channel messages.

**definition**

$abs :: msg\ set \Rightarrow chan\ set$

**where**

$abs\ S \equiv \{ Chan\ x\ A\ B\ M \mid x\ A\ B\ M. M \in payload \wedge implem\ (Chan\ x\ A\ B\ M) \in S \}$

**lemma** *absE* [*elim*]:

$\llbracket X \in abs\ H;$   
 $\bigwedge x\ A\ B\ M. X = Chan\ x\ A\ B\ M \Longrightarrow M \in payload \Longrightarrow implem\ X \in H \Longrightarrow P \rrbracket$   
 $\Longrightarrow P$

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

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

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

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

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

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

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

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

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

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

General lemmas about implementations and *local.abs*.

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

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

**lemma** *abs-insert-impl* [*simp*]:

$M \in payload \Longrightarrow abs\ (insert\ (implem\ (Chan\ x\ A\ B\ M))\ S) = insert\ (Chan\ x\ A\ B\ M)\ (abs\ S)$

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

**lemma** *extr-payload* [*simp*, *intro*]:

$\llbracket X \in extr\ Bad\ NI\ (abs\ I); NI \subseteq payload \rrbracket \Longrightarrow X \in payload$

**by** (*erule extr.induct, blast, auto*)

**lemma** *abs-Un-LtK*:

$K \subseteq \text{range LtK} \implies \text{abs } (K \cup S) = \text{abs } S$

**by** (*auto simp add: abs-Un-eq*)

**lemma** *abs-Un-keys-of* [*simp*]:

$\text{abs } (\text{keys-of } A \cup S) = \text{abs } S$

**by** (*auto intro!: abs-Un-LtK*)

Lemmas about *valid* and *local.abs*

**lemma** *abs-validSet*:  $\text{abs } (S \cap \text{valid}) = \text{abs } S$

**by** (*auto elim: absE intro: validI*)

**lemma** *valid-abs*:  $M \in \text{valid} \implies \exists M'. M' \in \text{abs } \{M\}$

**by** (*auto elim: validE*)

## 13.2 Extractable messages

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

**definition**

$\text{extractable} :: \text{msg set} \Rightarrow \text{msg set} \Rightarrow \text{msg set}$

**where**

$\text{extractable } K I \equiv$

$\{\text{implInsec } A B M \mid A B M. \text{implInsec } A B M \in I\} \cup$

$\{\text{implAuth } A B M \mid A B M. \text{implAuth } A B M \in I\} \cup$

$\{\text{implConfid } A B M \mid A B M. \text{implConfid } A B M \in I \wedge (A, B) \in \text{broken } K\} \cup$

$\{\text{implSecure } A B M \mid A B M. \text{implSecure } A B M \in I \wedge (A, B) \in \text{broken } K\}$

**lemma** *extractable-red*:  $\text{extractable } K I \subseteq I$

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

**lemma** *extractableI*:

$\text{implem } (\text{Chan } x A B M) \in I \implies$

$x = \text{insec} \vee x = \text{auth} \vee ((x = \text{confid} \vee x = \text{secure}) \wedge (A, B) \in \text{broken } K) \implies$

$\text{implem } (\text{Chan } x A B M) \in \text{extractable } K I$

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

**lemma** *extractableE*:

$X \in \text{extractable } K I \implies$

$(\bigwedge A B M. X = \text{implInsec } A B M \implies X \in I \implies P) \implies$

$(\bigwedge A B M. X = \text{implAuth } A B M \implies X \in I \implies P) \implies$

$(\bigwedge A B M. X = \text{implConfid } A B M \implies X \in I \implies (A, B) \in \text{broken } K \implies P) \implies$

$(\bigwedge A B M. X = \text{implSecure } A B M \implies X \in I \implies (A, B) \in \text{broken } K \implies P) \implies$

$P$

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

General lemmas about implementations and extractable.

**lemma** *implem-extractable* [*simp*]:

$\llbracket \text{Keys-bad } K \text{ Bad}; \text{implem } (\text{Chan } x A B M) \in \text{extractable } K I; M \in \text{payload} \rrbracket$

$\implies M \in \text{extr } \text{Bad } \text{NI } (\text{abs } I)$

by (erule extractableE, auto)

Auxiliary lemmas about extractable messages: they are implementations.

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

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

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

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

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

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

### 13.2.1 Partition $I$ to keep only the extractable messages

Partition the implementation set.

**lemma** *impl-partition*:  
 $\llbracket NI \subseteq \text{payload}; I \subseteq \text{valid} \rrbracket \implies$   
 $I \subseteq \text{extractable } K \ I \cup$   
 $\text{implConfidSet } (\text{UNIV} - \text{broken } K) \ \text{payload} \cup$   
 $\text{implSecureSet } (\text{UNIV} - \text{broken } K) \ \text{payload}$   
 by (auto dest!: subsetD [where A=I] elim!: valid-cases intro: extractableI)

### 13.2.2 Partition of *extractable*

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

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



### 13.3 Lemmas for proving intruder refinement (L2-L3)

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

$$\begin{aligned} & \text{synth } (\text{analz } (NI \cup I \cup K \cup \text{Tags})) \\ & \subseteq \text{synth } (\text{analz } (\text{extr } \text{Bad } NI \text{ } (\text{local.abs } I))) \cup \text{--payload} \end{aligned}$$

#### 13.3.1 First: we only keep the extractable messages

— the *synth* is probably not needed

**lemma** *analz-NI-I-K-analz-NI-EI*:

**assumes** *HNI*:  $NI \subseteq \text{payload}$

**and** *HK*:  $K \subseteq \text{range } \text{LtK}$

**and** *HI*:  $I \subseteq \text{valid}$

**shows**  $\text{analz } (NI \cup I \cup K \cup \text{Tags}) \subseteq$

$\text{synth } (\text{analz } (NI \cup \text{extractable } K \ I \cup K \cup \text{Tags})) \cup \text{--payload}$

**proof** —

**from** *HNI HI*

**have**  $\text{analz } (NI \cup I \cup K \cup \text{Tags}) \subseteq$

$\text{analz } (NI \cup (\text{extractable } K \ I \cup$   
 $\text{implConfidSet } (UNIV - \text{broken } K) \ \text{payload} \cup$   
 $\text{implSecureSet } (UNIV - \text{broken } K) \ \text{payload})$   
 $\cup K \cup \text{Tags})$

**by** (*intro analz-mono Un-mono impl-partition, simp-all*)

**also have**  $\dots \subseteq \text{analz } (\text{implConfidSet } (UNIV - \text{broken } K) \ \text{payload} \cup$   
 $(\text{implSecureSet } (UNIV - \text{broken } K) \ \text{payload} \cup$   
 $(\text{extractable } K \ I \cup NI \cup K \cup \text{Tags})))$

**by** (*auto*)

**also have**  $\dots \subseteq \text{synth } (\text{analz } (\text{implSecureSet } (UNIV - \text{broken } K) \ \text{payload} \cup$   
 $(\text{extractable } K \ I \cup NI \cup K \cup \text{Tags}))) \cup \text{--payload}$

**proof** (*rule analz-Un-implConfidSet-2*)

**show**  $\text{Enc-keys-clean } (\text{implSecureSet } (UNIV - \text{broken } K) \ \text{payload} \cup$   
 $(\text{extractable } K \ I \cup NI \cup K \cup \text{Tags}))$

**by** (*auto simp add: HNI HI HK intro: validI*)

**next**

**from** *HK HI HNI*

**show**  $(UNIV - \text{broken } K) \cap$

$\text{broken } (\text{parts } ($   
 $\text{implSecureSet } (UNIV - \text{broken } K) \ \text{payload} \cup$   
 $(\text{extractable } K \ I \cup NI \cup K \cup \text{Tags})) \cap \text{range } \text{LtK}) = \{\}$

**by** (*auto simp add: LtKeys-parts*)

$\text{LtKeys-parts-implSecureSet-elt } [\text{where } \text{Ag} = - \text{broken } K, \text{ simplified}]$

**qed** (*auto*)

**also have**  $\dots \subseteq \text{synth } (\text{analz } (\text{extractable } K \ I \cup NI \cup K \cup \text{Tags})) \cup \text{--payload}$

**proof** (*rule Un-least, rule synth-idem-payload*)

**show**  $\text{analz } (\text{implSecureSet } (UNIV - \text{broken } K) \ \text{payload} \cup$   
 $(\text{extractable } K \ I \cup NI \cup K \cup \text{Tags}))$

$\subseteq \text{synth } (\text{analz } (\text{extractable } K \ I \cup NI \cup K \cup \text{Tags})) \cup \text{--payload}$

**proof** (*rule analz-Un-implSecureSet-2*)

**show**  $\text{Enc-keys-clean } (\text{extractable } K \ I \cup NI \cup K \cup \text{Tags})$

**using** *HNI HK HI by auto*

**next**

```

from HI HK HNI
show (UNIV – broken K)  $\cap$ 
      broken (parts (extractable K I  $\cup$  NI  $\cup$  K  $\cup$  Tags)  $\cap$  range LtK) = {}
by (auto simp add: LtKeys-parts)
qed (auto)
next
show –payload  $\subseteq$  synth (analz (extractable K I  $\cup$  NI  $\cup$  K  $\cup$  Tags))  $\cup$  –payload
by auto
qed
also have ...  $\subseteq$  synth (analz (NI  $\cup$  extractable K I  $\cup$  K  $\cup$  Tags))  $\cup$  –payload
by (simp add: sup.left-commute sup-commute)
finally show ?thesis .
qed

```

### 13.3.2 Only keep the extracted messages (instead of extractable)

```

lemma analz-NI-EI-K-synth-analz-NI-E-K:
assumes HNI: NI  $\subseteq$  payload
and HK: K  $\subseteq$  range LtK
and HI: I  $\subseteq$  valid
and Hbad: Keys-bad K Bad
shows analz (NI  $\cup$  extractable K I  $\cup$  K  $\cup$  Tags)
   $\subseteq$  synth (analz (extr Bad NI (abs I)  $\cup$  K  $\cup$  Tags))  $\cup$  –payload
proof –
from HNI HI Hbad
have analz (NI  $\cup$  extractable K I  $\cup$  K  $\cup$  Tags)  $\subseteq$ 
  analz (NI  $\cup$  (implInsecSet (extr Bad NI (abs I))  $\cup$ 
    implConfidSet UNIV (extr Bad NI (abs I))  $\cup$ 
    implAuthSet (extr Bad NI (abs I))  $\cup$ 
    implSecureSet UNIV (extr Bad NI (abs I)))  $\cup$ 
    K  $\cup$  Tags)
by (intro analz-mono Un-mono extractable-partition) (auto)

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

```

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

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

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

**lemma** *analz-LtKeys-Tag*:

**assumes**  $\text{NI} \subseteq \text{payload}$  **and**  $\text{K} \subseteq \text{range LtK}$

**shows**  $\text{analz} (\text{NI} \cup \text{K} \cup \text{Tags}) \subseteq \text{analz NI} \cup \text{K} \cup \text{Tags}$

**proof**

**fix**  $X$

**assume**  $H: X \in \text{analz} (\text{NI} \cup \text{K} \cup \text{Tags})$

**thus**  $X \in \text{analz NI} \cup \text{K} \cup \text{Tags}$

**proof** (*induction X rule: analz.induct*)

**case** (*Dec X Y*)

**hence**  $\text{Enc X Y} \in \text{payload}$  **using** *assms* **by** *auto*

**moreover**

**from** *Dec.IH(2)* **have**  $Y \in \text{synth} (\text{analz NI} \cup (\text{K} \cup \text{Tags}))$

**by** (*auto simp add: Collect-disj-eq dest!: synth-Int2*)

**ultimately show** *?case* **using** *Dec.IH(1) assms(2)*

**by** (*auto dest!: synth-payload [THEN [2] rev-subsetD]*)

**next**

**case** (*Adec-lt X Y*)

**hence**  $\text{Aenc X (pubK Y)} \in \text{payload} \cup \text{range LtK} \cup \text{Tags}$  **using** *assms*

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

**lemma** *analz-NI-E-K-analz-NI-E*:

$\llbracket NI \subseteq \text{payload}; K \subseteq \text{range LtK}; I \subseteq \text{valid} \rrbracket$   
 $\implies \text{analz} (\text{extr Bad NI} (abs I) \cup K \cup \text{Tags}) \subseteq \text{analz} (\text{extr Bad NI} (abs I)) \cup K \cup \text{Tags}$   
 by (*rule analz-LtKeys-Tag*) *auto*

### 13.3.4 Final lemmas, using all the previous ones

**lemma** *analz-NI-I-K-synth-analz-NI-E*:

**assumes**

*Hbad: Keys-bad K Bad* and

*HNI: NI ⊆ payload* and

*HK: K ⊆ range LtK* and

*HI: I ⊆ valid*

**shows**

$\text{analz} (NI \cup I \cup K \cup \text{Tags}) \subseteq \text{synth} (\text{analz} (\text{extr Bad NI} (abs I))) \cup \text{--payload}$

**proof** –

**from** *HNI HK HI* **have**  $\text{analz} (NI \cup I \cup K \cup \text{Tags}) \subseteq$   
 $\text{synth} (\text{analz} (NI \cup \text{extractable K I} \cup K \cup \text{Tags})) \cup \text{--payload}$

by (*rule analz-NI-I-K-analz-NI-EI*)

**also have**  $\dots \subseteq \text{synth} (\text{analz} (\text{extr Bad NI} (abs I) \cup K \cup \text{Tags})) \cup \text{--payload}$

**proof** (*rule Un-least, simp-all*)

**from** *Hbad HNI HK HI* **have**  $\text{analz} (NI \cup \text{extractable K I} \cup K \cup \text{Tags}) \subseteq$   
 $\text{synth} (\text{analz} (\text{extr Bad NI} (abs I) \cup K \cup \text{Tags})) \cup \text{--payload}$

by (*intro analz-NI-EI-K-synth-analz-NI-E-K*)

**then show**  $\text{synth} (\text{analz} (NI \cup \text{extractable K I} \cup K \cup \text{Tags})) \subseteq$   
 $\text{synth} (\text{analz} (\text{extr Bad NI} (abs I) \cup K \cup \text{Tags})) \cup \text{--payload}$

by (*rule synth-idem-payload*)

qed

**also have**  $\dots \subseteq \text{synth} (\text{analz} (\text{extr Bad NI} (abs I))) \cup \text{--payload}$

**proof** (*rule Un-least, simp-all*)

**from** *HNI HK HI* **have**  $\text{analz} (\text{extr Bad NI} (abs I) \cup K \cup \text{Tags}) \subseteq$   
 $\text{analz} (\text{extr Bad NI} (abs I)) \cup K \cup \text{Tags}$

by (*rule analz-NI-E-K-analz-NI-E*)

**also from** *HK* **have**  $\dots \subseteq \text{analz} (\text{extr Bad NI} (abs I)) \cup \text{--payload}$

by *auto*

**also have**  $\dots \subseteq \text{synth} (\text{analz} (\text{extr Bad NI} (abs I))) \cup \text{--payload}$

by *auto*

**finally show**  $\text{synth} (\text{analz} (\text{extr Bad NI} (abs I) \cup K \cup \text{Tags})) \subseteq$   
 $\text{synth} (\text{analz} (\text{extr Bad NI} (abs I))) \cup \text{--payload}$

by (*rule synth-idem-payload*)

qed

**finally show** *?thesis* .  
**qed**

Lemma actually used to prove the refinement.

**lemma** *synth-analz-NI-I-K-synth-analz-NI-E*:  
 $\llbracket \text{Keys-bad } K \text{ Bad}; NI \subseteq \text{payload}; K \subseteq \text{range LtK}; I \subseteq \text{valid} \rrbracket$   
 $\implies \text{synth } (\text{analz } (NI \cup I \cup K \cup \text{Tags}))$   
 $\subseteq \text{synth } (\text{analz } (\text{extr Bad NI } (\text{abs I}))) \cup \text{-payload}$   
**by** (*intro synth-idem-payload analz-NI-I-K-synth-analz-NI-E*) (*assumption+*)

### 13.3.5 Partitioning *analz ik*

Two lemmas useful for proving the invariant

$\text{analz ik} \subseteq \text{synth } (\text{analz } (ik \cap \text{payload} \cup ik \cap \text{valid} \cup ik \cap \text{range LtK} \cup \text{Tags}))$

**lemma** *analz-Un-partition*:  
 $\text{analz } S \subseteq \text{synth } (\text{analz } ((S \cap \text{payload}) \cup (S \cap \text{valid}) \cup (S \cap \text{range LtK}) \cup \text{Tags})) \implies$   
 $H \subseteq \text{payload} \cup \text{valid} \cup \text{range LtK} \implies$   
 $\text{analz } (H \cup S) \subseteq$   
 $\text{synth } (\text{analz } (((H \cup S) \cap \text{payload}) \cup ((H \cup S) \cap \text{valid}) \cup ((H \cup S) \cap \text{range LtK}) \cup \text{Tags}))$

**proof** –

**assume**  $H \subseteq \text{payload} \cup \text{valid} \cup \text{range LtK}$

**then have**  $HH:H = (H \cap \text{payload}) \cup (H \cap \text{valid}) \cup (H \cap \text{range LtK})$

**by** *auto*

**assume** *HA*:

$\text{analz } S \subseteq \text{synth } (\text{analz } ((S \cap \text{payload}) \cup (S \cap \text{valid}) \cup (S \cap \text{range LtK}) \cup \text{Tags}))$

**then have**

$\text{analz } (H \cup S) \subseteq$

$\text{synth } (\text{analz } (H \cup ((S \cap \text{payload}) \cup (S \cap \text{valid}) \cup (S \cap \text{range LtK}) \cup \text{Tags})))$

**by** (*rule analz-synth-subset-Un2*)

**also with** *HH* **have**

$\dots \subseteq \text{synth } (\text{analz } (((H \cap \text{payload}) \cup (H \cap \text{valid}) \cup (H \cap \text{range LtK})) \cup$   
 $((S \cap \text{payload}) \cup (S \cap \text{valid}) \cup (S \cap \text{range LtK}) \cup \text{Tags})))$

**by** *auto*

**also have**  $\dots = \text{synth } (\text{analz } (((H \cup S) \cap \text{payload}) \cup ((H \cup S) \cap \text{valid}) \cup$   
 $((H \cup S) \cap \text{range LtK}) \cup \text{Tags}))$

**by** (*simp add: Un-left-commute sup commute Int-Un-distrib2*)

**finally show** *?thesis* .

**qed**

**lemma** *analz-insert-partition*:  
 $\text{analz } S \subseteq \text{synth } (\text{analz } ((S \cap \text{payload}) \cup (S \cap \text{valid}) \cup (S \cap \text{range LtK}) \cup \text{Tags})) \implies$   
 $x \in \text{payload} \cup \text{valid} \cup \text{range LtK} \implies$   
 $\text{analz } (\text{insert } x \text{ } S) \subseteq$   
 $\text{synth } (\text{analz } (((\text{insert } x \text{ } S) \cap \text{payload}) \cup ((\text{insert } x \text{ } S) \cap \text{valid}) \cup$   
 $((\text{insert } x \text{ } S) \cap \text{range LtK}) \cup \text{Tags}))$

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

**end**

**end**

## 14 Symmetric Implementation of Channel Messages

```
theory Implem-symmetric
imports Implem
begin
```

### 14.1 Implementation of channel messages

```
fun implem-sym :: chan  $\Rightarrow$  msg where
  implem-sym (Insec A B M) =  $\langle$ InsecTag, Agent A, Agent B, M $\rangle$ 
| implem-sym (Confid A B M) = Enc  $\langle$ ConfidTag, M $\rangle$  (shrK A B)
| implem-sym (Auth A B M) =  $\langle$ M, hmac  $\langle$ AuthTag, M $\rangle$  (shrK A B) $\rangle$ 
| implem-sym (Secure A B M) = Enc  $\langle$ SecureTag, M $\rangle$  (shrK A B)
```

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

```
interpretation sym: basic-implem implem-sym
done
```

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

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

```
abbreviation implInsecSet-aux :: msg set  $\Rightarrow$  msg set where
  implInsecSet-aux G  $\equiv$  PairSet Tags (PairSet (range Agent) (PairSet (range Agent) G))
```

```
abbreviation implAuthSet-aux :: msg set  $\Rightarrow$  msg set where
  implAuthSet-aux G  $\equiv$  PairSet G (HashSet (PairSet (PairSet Tags G) (range (case-prod shrK))))
```

```
abbreviation implConfidSet-aux :: (agent * agent) set  $\Rightarrow$  msg set  $\Rightarrow$  msg set where
  implConfidSet-aux Ag G  $\equiv$  EncSet (PairSet Tags G) (case-prod shrK'Ag)
```

```
abbreviation implSecureSet-aux :: (agent * agent) set  $\Rightarrow$  msg set  $\Rightarrow$  msg set where
  implSecureSet-aux Ag G  $\equiv$  EncSet (PairSet Tags G) (case-prod shrK'Ag)
```

These auxiliary definitions are overapproximations.

```
lemma implInsecSet-implInsecSet-aux: sym.implInsecSet G  $\subseteq$  implInsecSet-aux G
by auto
```

```
lemma implAuthSet-implAuthSet-aux: sym.implAuthSet G  $\subseteq$  implAuthSet-aux G
by (auto, auto)
```

```
lemma implConfidSet-implConfidSet-aux: sym.implConfidSet Ag G  $\subseteq$  implConfidSet-aux Ag G
by (auto)
```

```
lemma implSecureSet-implSecureSet-aux: sym.implSecureSet Ag G  $\subseteq$  implSecureSet-aux Ag G
by (auto)
```

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

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

## 14.2 Lemmas to pull implementation sets out of *analz*

All these proofs are similar:

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

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

### 14.2.1 Pull *implInsecSet* out of *analz*

**lemma** *analz-Un-implInsecSet-aux-1*:

$$\begin{aligned} & \text{Enc-keys-clean } (G \cup H) \implies \\ & \text{analz } (\text{implInsecSet-aux } G \cup H) \subseteq \\ & \quad \text{implInsecSet-aux } G \cup \text{Tags} \cup \\ & \quad \text{PairSet } (\text{range Agent}) (\text{PairSet } (\text{range Agent}) G) \cup \\ & \quad \text{PairSet } (\text{range Agent}) G \cup \\ & \quad \text{analz } (\text{range Agent} \cup G \cup (\text{range Agent} \cup H)) \end{aligned}$$

**proof** –

**assume**  $H: \text{Enc-keys-clean } (G \cup H)$

**have**  $\text{analz } (\text{implInsecSet-aux } G \cup H) \subseteq \text{implInsecSet-aux } G \cup$   
 $\text{analz } (\text{Tags} \cup \text{PairSet } (\text{range Agent}) (\text{PairSet } (\text{range Agent}) G) \cup H)$

**by** (rule *analz-Un-PairSet*)

**also have**  $\dots = \text{implInsecSet-aux } G \cup$

$$\text{analz } (\text{Tags} \cup (\text{PairSet } (\text{range Agent}) (\text{PairSet } (\text{range Agent}) G) \cup H))$$

**by** (*simp only: Un-assoc*)

**also have**  $\dots \subseteq \text{implInsecSet-aux } G \cup$

$$(\text{Tags} \cup \text{analz } (\text{PairSet } (\text{range Agent}) (\text{PairSet } (\text{range Agent}) G) \cup H))$$

**by** (rule *Un-mono, blast, rule analz-Un-Tag, blast intro: H*)

**also have**  $\dots = \text{implInsecSet-aux } G \cup \text{Tags} \cup$

$$\text{analz } (\text{PairSet } (\text{range Agent}) (\text{PairSet } (\text{range Agent}) G) \cup H)$$

**by** *auto*

**also have**  $\dots \subseteq \text{implInsecSet-aux } G \cup \text{Tags} \cup (\text{PairSet } (\text{range Agent}) (\text{PairSet } (\text{range Agent}) G) \cup$   
 $\text{analz } (\text{range Agent} \cup \text{PairSet } (\text{range Agent}) G \cup H))$

**by** (rule *Un-mono, blast, rule analz-Un-PairSet*)

**also have**  $\dots = \text{implInsecSet-aux } G \cup \text{Tags} \cup \text{PairSet } (\text{range Agent}) (\text{PairSet } (\text{range Agent}) G) \cup$   
 $\text{analz } (\text{PairSet } (\text{range Agent}) G \cup (\text{range Agent} \cup H))$

**by** (*auto simp add: Un-assoc Un-commute*)

**also have**  $\dots \subseteq \text{implInsecSet-aux } G \cup \text{Tags} \cup \text{PairSet } (\text{range Agent}) (\text{PairSet } (\text{range Agent}) G) \cup$   
 $(\text{PairSet } (\text{range Agent}) G \cup \text{analz } (\text{range Agent} \cup G \cup (\text{range Agent} \cup H)))$

**by** (rule *Un-mono, blast, rule analz-Un-PairSet*)

**also have**  $\dots = \text{implInsecSet-aux } G \cup \text{Tags} \cup (\text{PairSet } (\text{range Agent}) (\text{PairSet } (\text{range Agent}) G) \cup$   
 $\text{PairSet } (\text{range Agent}) G) \cup \text{analz } (\text{range Agent} \cup G \cup (\text{range Agent} \cup H))$



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

**lemma** *analz-Un-implInsecSet-aux-2:*

$Enc\text{-}keys\text{-}clean (G \cup H) \implies$   
 $analz (implInsecSet\text{-}aux G \cup H) \subseteq$   
 $implInsecSet\text{-}aux G \cup Tags \cup$   
 $synth (analz (G \cup H))$

**proof** –

assume  $H:Enc\text{-}keys\text{-}clean (G \cup H)$

have  $HH:PairSet (range Agent) (PairSet (range Agent) G) \cup$   
 $PairSet (range Agent) G \subseteq synth (analz (G \cup H))$

by auto

have  $HHH:analz (range Agent \cup G \cup (range Agent \cup H)) \subseteq synth (analz (G \cup H))$

**proof** –

have  $analz (range Agent \cup G \cup (range Agent \cup H)) \subseteq$   
 $synth (analz (range Agent \cup G \cup (range Agent \cup H)))$

by auto

also have  $\dots = synth (analz (synth (range Agent \cup G \cup (range Agent \cup H))))$  by auto

also have  $\dots \subseteq synth (analz (synth (G \cup H)))$

**proof** (rule synth-analz-mono)

have  $range Agent \cup G \cup (range Agent \cup H) \subseteq synth (G \cup H)$  by auto

then have  $synth (range Agent \cup G \cup (range Agent \cup H)) \subseteq synth (synth (G \cup H))$

by (rule synth-mono)

then show  $synth (range Agent \cup G \cup (range Agent \cup H)) \subseteq synth (G \cup H)$  by auto

qed

also have  $\dots = synth (analz (G \cup H))$  by auto

finally show ?thesis .

qed

from  $H$  have

$analz (implInsecSet\text{-}aux G \cup H) \subseteq$   
 $implInsecSet\text{-}aux G \cup Tags \cup PairSet (range Agent) (PairSet (range Agent) G) \cup$   
 $PairSet (range Agent) G \cup analz (range Agent \cup G \cup (range Agent \cup H))$

by (rule analz-Un-implInsecSet-aux-1)

also have  $\dots = implInsecSet\text{-}aux G \cup Tags \cup$

$(PairSet (range Agent) (PairSet (range Agent) G) \cup$

$PairSet (range Agent) G) \cup analz (range Agent \cup G \cup (range Agent \cup H))$

by (simp only: Un-assoc Un-commute)

also have  $\dots \subseteq implInsecSet\text{-}aux G \cup Tags \cup synth (analz (G \cup H)) \cup$

$synth (analz (G \cup H))$

by ((rule Un-mono)+, auto simp add: HH HHH)

finally show ?thesis by auto

qed

**lemma** *analz-Un-implInsecSet-aux-3:*

$Enc\text{-}keys\text{-}clean (G \cup H) \implies$

$analz (implInsecSet\text{-}aux G \cup H) \subseteq synth (analz (G \cup H)) \cup \text{-payload}$

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

**lemma** *analz-Un-implInsecSet:*

$Enc\text{-}keys\text{-}clean (G \cup H) \implies$

$analz (sym.implInsecSet G \cup H) \subseteq synth (analz (G \cup H)) \cup \text{-payload}$

```

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

```

### 14.3 Pull *implConfidSet* out of *analz*

**lemma** *analz-Un-implConfidSet-aux-1*:

$$\begin{aligned} & \text{Enc-keys-clean } (G \cup H) \implies \\ & \text{analz } (\text{implConfidSet-aux } Ag \ G \cup H) \subseteq \\ & \text{implConfidSet-aux } Ag \ G \cup \text{PairSet } Tags \ G \cup Tags \cup \\ & \text{analz } (G \cup H) \end{aligned}$$

**proof** –

```

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

```

**qed**

**lemma** *analz-Un-implConfidSet-aux-2*:

$$\begin{aligned} & \text{Enc-keys-clean } (G \cup H) \implies \\ & \text{analz } (\text{implConfidSet-aux } Ag \ G \cup H) \subseteq \\ & \text{implConfidSet-aux } Ag \ G \cup \text{PairSet } Tags \ G \cup Tags \cup \\ & \text{synth } (\text{analz } (G \cup H)) \end{aligned}$$

**proof** –

```

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

```

**qed**

**lemma** *analz-Un-implConfidSet-aux-3*:

$$\begin{aligned} & \text{Enc-keys-clean } (G \cup H) \implies \\ & \text{analz } (\text{implConfidSet-aux } Ag \ G \cup H) \subseteq \text{synth } (\text{analz } (G \cup H)) \cup \text{-payload} \\ & \text{by (rule subset-trans [OF analz-Un-implConfidSet-aux-2], auto)} \end{aligned}$$

**lemma** *analz-Un-implConfidSet*:

$$\begin{aligned} & \text{Enc-keys-clean } (G \cup H) \implies \\ & \text{analz } (\text{sym.implConfidSet } Ag \ G \cup H) \subseteq \text{synth } (\text{analz } (G \cup H)) \cup \text{-payload} \end{aligned}$$

```

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

```

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

**lemma** *analz-Un-implConfidSet-2-aux-1*:

*Enc-keys-clean H*  $\implies$

$Ag \cap broken (parts H \cap range LtK) = \{\} \implies$

$analz (implConfidSet-aux Ag G \cup H) \subseteq implConfidSet-aux Ag G \cup synth (analz H)$

**apply** (rule *subset-trans* [*OF analz-Un-EncSet2*], *simp*)

**apply** (auto dest:*analz-into-parts*)

**done**

**lemma** *analz-Un-implConfidSet-2-aux-3*:

*Enc-keys-clean H*  $\implies$

$Ag \cap broken (parts H \cap range LtK) = \{\} \implies$

$analz (implConfidSet-aux Ag G \cup H) \subseteq synth (analz H) \cup \text{-payload}$

**by** (rule *subset-trans* [*OF analz-Un-implConfidSet-2-aux-1*], auto)

**lemma** *analz-Un-implConfidSet-2*:

*Enc-keys-clean H*  $\implies$

$Ag \cap broken (parts H \cap range LtK) = \{\} \implies$

$analz (sym.implConfidSet Ag G \cup H) \subseteq synth (analz H) \cup \text{-payload}$

**apply** (rule *subset-trans* [*of - analz (implConfidSet-aux Ag G \cup H) -*])

**apply** (rule *analz-mono*, rule *Un-mono*, blast intro!: *implSet-implSet-aux*, *simp*)

**using** *analz-Un-implConfidSet-2-aux-3* **apply** auto

**done**

#### 14.4 Pull *implSecureSet* out of *analz*

**lemma** *analz-Un-implSecureSet-aux-1*:

*Enc-keys-clean (G \cup H)*  $\implies$

$analz (implSecureSet-aux Ag G \cup H) \subseteq$

$implSecureSet-aux Ag G \cup PairSet Tags G \cup Tags \cup$

$analz (G \cup H)$

**proof** –

**assume** *H:Enc-keys-clean (G \cup H)*

**have**  $analz (implSecureSet-aux Ag G \cup H) \subseteq$

$implSecureSet-aux Ag G \cup analz (PairSet Tags G \cup H)$

**by** (rule *analz-Un-EncSet*, *fast*, blast intro: *H*)

**also have**  $\dots \subseteq implSecureSet-aux Ag G \cup (PairSet Tags G \cup analz (Tags \cup G \cup H))$

**by** (rule *Un-mono*, blast, rule *analz-Un-PairSet*)

**also have**  $\dots = implSecureSet-aux Ag G \cup PairSet Tags G \cup analz (Tags \cup (G \cup H))$

**by** (*simp only: Un-assoc*)

**also have**  $\dots \subseteq implSecureSet-aux Ag G \cup PairSet Tags G \cup (Tags \cup analz (G \cup H))$

**by** (rule *Un-mono*, blast, rule *analz-Un-Tag*, blast intro: *H*)

**finally show** *?thesis* **by** auto

**qed**

**lemma** *analz-Un-implSecureSet-aux-2*:

*Enc-keys-clean (G \cup H)*  $\implies$

$analz (implSecureSet-aux Ag G \cup H) \subseteq$

$implSecureSet-aux Ag G \cup PairSet Tags G \cup Tags \cup$

$synth (analz (G \cup H))$

**proof** –

**assume** *H:Enc-keys-clean (G \cup H)*

**from**  $H$  **have**  $\text{analz } (\text{implSecureSet-aux } Ag \ G \cup H) \subseteq$   
 $\text{implSecureSet-aux } Ag \ G \cup \text{PairSet } Tags \ G \cup Tags \cup \text{analz } (G \cup H)$   
**by** (rule  $\text{analz-Un-implSecureSet-aux-1}$ )  
**also have**  $\dots \subseteq \text{implSecureSet-aux } Ag \ G \cup \text{PairSet } Tags \ G \cup Tags \cup \text{synth } (\text{analz } (G \cup H))$   
**by** *auto*  
**finally show** *?thesis* **by** *auto*  
**qed**

**lemma**  $\text{analz-Un-implSecureSet-aux-3}$ :  
 $\text{Enc-keys-clean } (G \cup H) \implies$   
 $\text{analz } (\text{implSecureSet-aux } Ag \ G \cup H) \subseteq \text{synth } (\text{analz } (G \cup H)) \cup \text{-payload}$   
**by** (rule  $\text{subset-trans [OF analz-Un-implSecureSet-aux-2]}$ , *auto*)

**lemma**  $\text{analz-Un-implSecureSet}$ :  
 $\text{Enc-keys-clean } (G \cup H) \implies$   
 $\text{analz } (\text{sym.implSecureSet } Ag \ G \cup H) \subseteq \text{synth } (\text{analz } (G \cup H)) \cup \text{-payload}$   
**apply** (rule  $\text{subset-trans [of - analz (implSecureSet-aux } Ag \ G \cup H) -]}$ )  
**apply** (rule  $\text{analz-mono}$ , rule  $\text{Un-mono}$ ,  $\text{blast intro! : implSet-implSet-aux, simp}$ )  
**using**  $\text{analz-Un-implSecureSet-aux-3}$  **apply** *blast*  
**done**

Pull  $\text{implSecureSet}$  out of  $\text{analz}$ , 2nd case, where the agents are honest.

**lemma**  $\text{analz-Un-implSecureSet-2-aux-1}$ :  
 $\text{Enc-keys-clean } H \implies$   
 $Ag \cap \text{broken } (\text{parts } H \cap \text{range } LtK) = \{\} \implies$   
 $\text{analz } (\text{implSecureSet-aux } Ag \ G \cup H) \subseteq \text{implSecureSet-aux } Ag \ G \cup \text{synth } (\text{analz } H)$   
**apply** (rule  $\text{subset-trans [OF analz-Un-EncSet2]}$ , *simp*)  
**apply** (*auto dest:analz-into-parts*)  
**done**

**lemma**  $\text{analz-Un-implSecureSet-2-aux-3}$ :  
 $\text{Enc-keys-clean } H \implies$   
 $Ag \cap \text{broken } (\text{parts } H \cap \text{range } LtK) = \{\} \implies$   
 $\text{analz } (\text{implSecureSet-aux } Ag \ G \cup H) \subseteq \text{synth } (\text{analz } H) \cup \text{-payload}$   
**by** (rule  $\text{subset-trans [OF analz-Un-implSecureSet-2-aux-1]}$ , *auto*)

**lemma**  $\text{analz-Un-implSecureSet-2}$ :  
 $\text{Enc-keys-clean } H \implies$   
 $Ag \cap \text{broken } (\text{parts } H \cap \text{range } LtK) = \{\} \implies$   
 $\text{analz } (\text{sym.implSecureSet } Ag \ G \cup H) \subseteq \text{synth } (\text{analz } H) \cup \text{-payload}$   
**apply** (rule  $\text{subset-trans [of - analz (implSecureSet-aux } Ag \ G \cup H) -]}$ )  
**apply** (rule  $\text{analz-mono}$ , rule  $\text{Un-mono}$ ,  $\text{blast intro! : implSet-implSet-aux, simp}$ )  
**using**  $\text{analz-Un-implSecureSet-2-aux-3}$  **apply** *auto*  
**done**

## 14.5 Pull $\text{implAuthSet}$ out of $\text{analz}$

**lemma**  $\text{analz-Un-implAuthSet-aux-1}$ :  
 $\text{Enc-keys-clean } (G \cup H) \implies$   
 $\text{analz } (\text{implAuthSet-aux } G \cup H) \subseteq$   
 $\text{implAuthSet-aux } G \cup \text{HashSet } (\text{PairSet } (\text{PairSet } Tags \ G) \ (\text{range } (\text{case-prod } \text{shrK}))) \cup$   
 $\text{analz } (G \cup H)$   
**proof** –

**assume**  $H:Enc\text{-}keys\text{-}clean (G \cup H)$   
**have**  $analz (implAuthSet\text{-}aux G \cup H) \subseteq implAuthSet\text{-}aux G \cup$   
 $analz (G \cup HashSet (PairSet (PairSet Tags G) (range (case\text{-}prod shrK))) \cup H)$   
**by** (rule *analz-Un-PairSet*)  
**also have**  $\dots = implAuthSet\text{-}aux G \cup$   
 $analz (HashSet (PairSet (PairSet Tags G) (range (case\text{-}prod shrK))) \cup G \cup H)$   
**by** (simp only: *Un-assoc Un-commute*)  
**also have**  $\dots = implAuthSet\text{-}aux G \cup$   
 $analz (HashSet (PairSet (PairSet Tags G) (range (case\text{-}prod shrK))) \cup (G \cup H))$   
**by** (simp only: *Un-assoc*)  
**also have**  
 $\dots \subseteq implAuthSet\text{-}aux G \cup$   
 $(HashSet (PairSet (PairSet Tags G) (range (case\text{-}prod shrK))) \cup$   
 $analz (G \cup H))$   
**by** (rule *Un-mono, blast, rule analz-Un-HashSet, blast intro: H, auto*)  
**also have**  $\dots = implAuthSet\text{-}aux G \cup$   
 $HashSet (PairSet (PairSet Tags G) (range (case\text{-}prod shrK))) \cup$   
 $analz (G \cup H)$   
**by** *auto*  
**finally show** *?thesis* **by** *auto*  
**qed**

**lemma** *analz-Un-implAuthSet-aux-2:*  
 $Enc\text{-}keys\text{-}clean (G \cup H) \implies$   
 $analz (implAuthSet\text{-}aux G \cup H) \subseteq$   
 $implAuthSet\text{-}aux G \cup HashSet (PairSet (PairSet Tags G) (range (case\text{-}prod shrK))) \cup$   
 $synth (analz (G \cup H))$

**proof** –  
**assume**  $H:Enc\text{-}keys\text{-}clean (G \cup H)$   
**from**  $H$  **have**  
 $analz (implAuthSet\text{-}aux G \cup H) \subseteq$   
 $implAuthSet\text{-}aux G \cup$   
 $HashSet (PairSet (PairSet Tags G) (range (case\text{-}prod shrK))) \cup$   
 $analz (G \cup H)$   
**by** (rule *analz-Un-implAuthSet-aux-1*)  
**also have**  
 $\dots \subseteq implAuthSet\text{-}aux G \cup$   
 $HashSet (PairSet (PairSet Tags G) (range (case\text{-}prod shrK))) \cup$   
 $synth (analz (G \cup H))$   
**by** *auto*  
**finally show** *?thesis* **by** *auto*  
**qed**

**lemma** *analz-Un-implAuthSet-aux-3:*  
 $Enc\text{-}keys\text{-}clean (G \cup H) \implies$   
 $analz (implAuthSet\text{-}aux G \cup H) \subseteq synth (analz (G \cup H)) \cup \text{-}payload$   
**by** (rule *subset-trans [OF analz-Un-implAuthSet-aux-2], auto*)

**lemma** *analz-Un-implAuthSet:*  
 $Enc\text{-}keys\text{-}clean (G \cup H) \implies$   
 $analz (sym.implAuthSet G \cup H) \subseteq synth (analz (G \cup H)) \cup \text{-}payload$   
**apply** (rule *subset-trans [of - analz (implAuthSet-aux G \cup H) -]*)  
**apply** (rule *analz-mono, rule Un-mono, blast intro!: implSet-implSet-aux, simp*)

**using** *analz-Un-implAuthSet-aux-3* **apply** *blast*  
**done**

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

## 14.6 Locale interpretations

**interpretation** *sym: semivalid-implem implem-sym*

**proof** (*unfold-locales*)

**fix**  $x A B M x' A' B' M'$

**show**  $\text{implem-sym } (\text{Chan } x A B M) = \text{implem-sym } (\text{Chan } x' A' B' M') \longleftrightarrow$   
 $x = x' \wedge A = A' \wedge B = B' \wedge M = M'$

**by** (*cases x, (cases x', auto)+*)

**next**

**fix**  $M' M x x' A A' B B'$

**assume**  $H: M' \in \text{payload}$

**then have**  $A1: \bigwedge y. y \in \text{parts } \{M'\} \implies y \in \text{payload}$

**and**  $A2: \bigwedge y. M' = y \implies y \in \text{payload}$  **by** *auto*

**assume**  $\text{implem-sym } (\text{Chan } x A B M) \in \text{parts } \{\text{implem-sym } (\text{Chan } x' A' B' M')\}$

**then show**  $x = x' \wedge A = A' \wedge B = B' \wedge M = M'$

**by** (*cases x, (cases x', auto dest!: A1 A2)+*)

**next**

**fix**  $I$

**assume**  $I \subseteq \text{sym.valid}$

**then show** *Enc-keys-clean I*

**proof** (*simp add: Enc-keys-clean-def, intro allI impI*)

**fix**  $X Y$

**assume**  $\text{Enc } X Y \in \text{parts } I$

**obtain**  $x A B M$  **where**  $M \in \text{payload}$  **and**  $\text{Enc } X Y \in \text{parts } \{\text{implem-sym } (\text{Chan } x A B M)\}$

**using** *parts-singleton [OF (Enc X Y ∈ parts I)] (I ⊆ sym.valid)*

**by** (*auto elim!: sym.validE*)

**then show**  $Y \in \text{range LtK} \vee Y \in \text{payload}$  **by** (*cases x, auto*)

**qed**

**next**

**fix**  $Z$

**show** *composed (implem-sym Z)*

**proof** (*cases Z, simp*)

**fix**  $x A B M$

**show** *composed (implem-sym (Chan x A B M))* **by** (*cases x, auto*)

**qed**

**next**

**fix**  $x A B M$

**show**  $\text{implem-sym } (\text{Chan } x A B M) \notin \text{payload}$

**by** (*cases x, auto*)

**next**

**fix**  $X K$

**assume**  $X \in \text{sym.valid}$

**then obtain**  $x A B M$  **where**  $M \in \text{payload}$   $X = \text{implem-sym } (\text{Chan } x A B M)$

**by** (*auto elim: sym.validE*)

**then show**  $\text{LtK } K \notin \text{parts } \{X\}$

**by** (*cases x, auto*)

**next**

```

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

```

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

```

lemma implInsec-synth-analz:
   $H \subseteq \text{payload} \cup \text{sym.valid} \cup \text{range } \text{LtK} \cup \text{Tags} \implies$ 
   $\text{sym.implInsec } A B M \in \text{synth } (\text{analz } H) \implies$ 
   $\text{sym.implInsec } A B M \in H \vee M \in \text{synth } (\text{analz } H)$ 
apply (erule synth.cases, auto)
done

```

**lemma** *implConfid-synth-analz*:

$H \subseteq \text{payload} \cup \text{sym.valid} \cup \text{range LtK} \cup \text{Tags} \implies$   
 $\text{sym.implConfid } A \ B \ M \in \text{synth } (\text{analz } H) \implies$   
 $\text{sym.implConfid } A \ B \ M \in H \vee M \in \text{synth } (\text{analz } H)$

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

— 1 subgoal

**apply** (*frule sym.analz-valid [where x=confid], auto*)

**done**

**lemma** *implAuth-synth-analz*:

$H \subseteq \text{payload} \cup \text{sym.valid} \cup \text{range LtK} \cup \text{Tags} \implies$   
 $\text{sym.implAuth } A \ B \ M \in \text{synth } (\text{analz } H) \implies$   
 $\text{sym.implAuth } A \ B \ M \in H \vee (M \in \text{synth } (\text{analz } H) \wedge (A, B) \in \text{broken } H)$

**using** [*simproc del: defined-all*] **proof** (*erule synth.cases, simp-all*)

**fix**  $X$

**assume**  $H$ :  $H \subseteq \text{payload} \cup \text{sym.valid} \cup \text{range LtK} \cup \text{Tags}$

**assume**  $H'$ :  $\langle M, \text{hmac } \langle \text{AuthTag}, M \rangle (\text{shrK } A \ B) \rangle = X \ X \in \text{analz } H$

**hence**  $\text{sym.implAuth } A \ B \ M \in \text{analz } H$  **by** *auto*

**with**  $H$  **have**  $\text{sym.implAuth } A \ B \ M \in H$  **by** (*rule sym.analz-valid*)

**with**  $H'$  **show**  $X \in H \vee M \in \text{synth } (\text{analz } H) \wedge (A, B) \in \text{broken } H$

**by** *auto*

**next**

**fix**  $X \ Y$

**assume**  $H$ :  $H \subseteq \text{payload} \cup \text{sym.valid} \cup \text{range LtK} \cup \text{Tags}$

**assume**  $H'$ :  $M = X \wedge \text{hmac } \langle \text{AuthTag}, M \rangle (\text{shrK } A \ B) = Y \ Y \in \text{synth } (\text{analz } H)$

**hence**  $\text{hmac } \langle \text{AuthTag}, M \rangle (\text{shrK } A \ B) \in \text{synth } (\text{analz } H)$  **by** *auto*

**then have**  $\text{hmac } \langle \text{AuthTag}, M \rangle (\text{shrK } A \ B) \in \text{analz } H \vee$

$\text{shrK } A \ B \in \text{synth } (\text{analz } H)$

**by** (*rule synth.cases, auto*)

**then show**  $\langle X, \text{hmac } \langle \text{AuthTag}, X \rangle (\text{shrK } A \ B) \rangle \in H \vee (A, B) \in \text{broken } H$

**proof**

**assume**  $\text{shrK } A \ B \in \text{synth } (\text{analz } H)$

**with**  $H$  **have**  $(A, B) \in \text{broken } H$  **by** (*auto dest:sym.analz-LtKeys*)

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

**next**

**assume**  $\text{hmac } \langle \text{AuthTag}, M \rangle (\text{shrK } A \ B) \in \text{analz } H$

**hence**  $\text{hmac } \langle \text{AuthTag}, M \rangle (\text{shrK } A \ B) \in \text{parts } H$  **by** (*rule analz-into-parts*)

**with**  $H$  **have**  $\text{hmac } \langle \text{AuthTag}, M \rangle (\text{shrK } A \ B) \in \text{parts } H$

**by** (*auto dest!:payload-parts elim!:payload-Hash*)

**from**  $H$  **obtain**  $Z$  **where**  $Z \in H$  **and**  $H''$ :  $\text{hmac } \langle \text{AuthTag}, M \rangle (\text{shrK } A \ B) \in \text{parts } \{Z\}$

**using** *parts-singleton* [*OF*  $\langle \text{hmac } \langle \text{AuthTag}, M \rangle (\text{shrK } A \ B) \in \text{parts } H \rangle$ ] **by** *blast*

**moreover with**  $H$  **have**  $Z \in \text{sym.valid}$  **by** (*auto dest!: subsetD*)

**moreover with**  $H''$  **have**  $Z = \text{sym.implAuth } A \ B \ M$

**by** (*auto*) (*erule sym.valid-cases, auto*)

**ultimately have**  $\text{sym.implAuth } A \ B \ M \in H$  **by** *auto*

**with**  $H'$  **show** *?thesis* **by** *auto*

**qed**

**qed**

**lemma** *implSecure-synth-analz*:

$H \subseteq \text{payload} \cup \text{sym.valid} \cup \text{range LtK} \cup \text{Tags} \implies$   
 $\text{sym.implSecure } A \ B \ M \in \text{synth } (\text{analz } H) \implies$   
 $\text{sym.implSecure } A \ B \ M \in H \vee (M \in \text{synth } (\text{analz } H) \wedge (A, B) \in \text{broken } H)$



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

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

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

**apply** (*auto dest:sym.analz-LtKeys*)

**done**

**interpretation** *sym: valid-implem implem-sym*

**proof** (*unfold-locales*)

**fix** *H A B M*

**assume**  $H \subseteq \text{payload} \cup \text{sym.valid} \cup \text{range LtK} \cup \text{Tags}$

$\text{implem-sym } (\text{Insec } A B M) \in \text{synth } (\text{analz } H)$

**then show**  $\text{implem-sym } (\text{Insec } A B M) \in H \vee M \in \text{synth } (\text{analz } H)$

**by** (*rule implInsec-synth-analz*)

**next**

**fix** *H A B M*

**assume**  $H \subseteq \text{payload} \cup \text{sym.valid} \cup \text{range LtK} \cup \text{Tags}$

$\text{implem-sym } (\text{Confid } A B M) \in \text{synth } (\text{analz } H)$

**then show**  $\text{implem-sym } (\text{Confid } A B M) \in H \vee M \in \text{synth } (\text{analz } H)$

**by** (*rule implConfid-synth-analz*)

**next**

**fix** *H A B M*

**assume**  $H \subseteq \text{payload} \cup \text{sym.valid} \cup \text{range LtK} \cup \text{Tags}$

$\text{implem-sym } (\text{Auth } A B M) \in \text{synth } (\text{analz } H)$

**then show**  $\text{implem-sym } (\text{Auth } A B M) \in H \vee$

$M \in \text{synth } (\text{analz } H) \wedge (A, B) \in \text{broken } H$

**by** (*rule implAuth-synth-analz*)

**next**

**fix** *H A B M*

**assume**  $H \subseteq \text{payload} \cup \text{sym.valid} \cup \text{range LtK} \cup \text{Tags}$

$\text{implem-sym } (\text{Secure } A B M) \in \text{synth } (\text{analz } H)$

**then show**  $\text{implem-sym } (\text{Secure } A B M) \in H \vee$

$M \in \text{synth } (\text{analz } H) \wedge (A, B) \in \text{broken } H$

**by** (*rule implSecure-synth-analz*)

**qed**

**end**

## 15 Asymmetric Implementation of Channel Messages

```
theory Implem-asymmetric
imports Implem
begin
```

### 15.1 Implementation of channel messages

```
fun implem-asym :: chan  $\Rightarrow$  msg where
  | implem-asym (Insec A B M) =  $\langle$ InsecTag, Agent A, Agent B, M $\rangle$ 
  | implem-asym (Confid A B M) = Aenc  $\langle$ Agent A, M $\rangle$  (pubK B)
  | implem-asym (Auth A B M) = Sign  $\langle$ Agent B, M $\rangle$  (priK A)
  | implem-asym (Secure A B M) = Sign (Aenc  $\langle$ SecureTag, Agent A, M $\rangle$  (pubK B)) (priK A)
```

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

```
interpretation asym: basic-implem implem-asym
done
```

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

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

```
abbreviation implInsecSet-aux :: msg set  $\Rightarrow$  msg set
where implInsecSet-aux G  $\equiv$  PairSet Tags (PairSet AgentSet (PairSet AgentSet G))
```

```
abbreviation implAuthSet-aux :: msg set  $\Rightarrow$  msg set
where implAuthSet-aux G  $\equiv$  SignSet (PairSet AgentSet G) (range priK)
```

```
abbreviation implConfidSet-aux :: (agent * agent) set  $\Rightarrow$  msg set  $\Rightarrow$  msg set
where implConfidSet-aux Ag G  $\equiv$  AencSet (PairSet AgentSet G) (pubK' (Ag "UNIV"))
```

```
abbreviation implSecureSet-aux :: (agent * agent) set  $\Rightarrow$  msg set  $\Rightarrow$  msg set
where implSecureSet-aux Ag G  $\equiv$ 
  SignSet (AencSet (PairSet Tags (PairSet AgentSet G)) (pubK' (Ag "UNIV"))) (range priK)
```

These auxiliary definitions are overapproximations.

```
lemma implInsecSet-implInsecSet-aux: asym.implInsecSet G  $\subseteq$  implInsecSet-aux G
by auto
```

```
lemma implAuthSet-implAuthSet-aux: asym.implAuthSet G  $\subseteq$  implAuthSet-aux G
by auto
```

```
lemma implConfidSet-implConfidSet-aux: asym.implConfidSet Ag G  $\subseteq$  implConfidSet-aux Ag G
by (auto, blast)
```

```
lemma implSecureSet-implSecureSet-aux: asym.implSecureSet Ag G  $\subseteq$  implSecureSet-aux Ag G
by (auto, blast)
```

```
lemmas implSet-implSet-aux =
  implInsecSet-implInsecSet-aux implAuthSet-implAuthSet-aux
```

*implConfidSet-implConfidSet-aux implSecureSet-implSecureSet-aux*

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

## 15.2 Lemmas to pull implementation sets out of *analz*

All these proofs are similar:

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

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

### 15.2.1 Pull *PairAgentSet* out of *analz*

**lemma** *analz-Un-PairAgentSet*:

**shows**

$analz (PairSet AgentSet G \cup H) \subseteq PairSet AgentSet G \cup AgentSet \cup analz (G \cup H)$

**proof** –

**have**  $analz (PairSet AgentSet G \cup H) \subseteq PairSet AgentSet G \cup analz (AgentSet \cup G \cup H)$

**by** (rule *analz-Un-PairSet*)

**also have**  $\dots \subseteq PairSet AgentSet G \cup AgentSet \cup analz (G \cup H)$

**apply** (*simp only: Un-assoc*)

**apply** (*intro Un-mono analz-Un-AgentSet, fast*)

**done**

**finally show** *?thesis* .

**qed**

### 15.2.2 Pull *implInsecSet* out of *analz*

**lemma** *analz-Un-implInsecSet-aux-aux*:

**assumes** *Enc-keys-clean* ( $G \cup H$ )

**shows**  $analz (implInsecSet-aux G \cup H) \subseteq implInsecSet-aux G \cup Tags \cup synth (analz (G \cup H))$

**proof** –

**have**  $analz (implInsecSet-aux G \cup H) \subseteq$

$implInsecSet-aux G \cup analz (Tags \cup PairSet AgentSet (PairSet AgentSet G) \cup H)$

**by** (rule *analz-Un-PairSet*)

**also have**  $\dots \subseteq implInsecSet-aux G \cup Tags \cup analz (PairSet AgentSet (PairSet AgentSet G) \cup H)$

**using** *assms*

**apply** –

**apply** (*simp only: Un-assoc, rule Un-mono, fast*)

**apply** (rule *analz-Un-Tag, blast*)

**done**

**also have**  $\dots \subseteq implInsecSet-aux G \cup Tags \cup PairSet AgentSet (PairSet AgentSet G) \cup AgentSet \cup analz (PairSet AgentSet G \cup H)$

**apply** –

**apply** (*simp only: Un-assoc, (rule Un-mono, fast)+*)  
**apply** (*simp only: Un-assoc [symmetric], rule analz-Un-PairAgentSet*)  
**done**  
**also have**  
 $\dots \subseteq \text{implInsecSet-aux } G \cup \text{Tags} \cup \text{PairSet AgentSet } (\text{PairSet AgentSet } G) \cup \text{AgentSet}$   
 $\cup \text{PairSet AgentSet } G \cup \text{AgentSet} \cup \text{analz } (G \cup H)$   
**apply** –  
**apply** (*simp only: Un-assoc, (rule Un-mono, fast)+*)  
**apply** (*simp only: Un-assoc [symmetric], rule analz-Un-PairAgentSet*)  
**done**  
**also have**  $\dots \subseteq \text{implInsecSet-aux } G \cup \text{Tags} \cup \text{synth } (\text{analz } (G \cup H))$   
**apply** –  
**apply** (*simp only: Un-assoc, (rule Un-mono, fast)+, auto*)  
**done**  
**finally show** *?thesis* .  
**qed**

**lemma** *analz-Un-implInsecSet-aux:*  
 $\text{Enc-keys-clean } (G \cup H) \implies$   
 $\text{analz } (\text{implInsecSet-aux } G \cup H) \subseteq \text{synth } (\text{analz } (G \cup H)) \cup \text{-payload}$   
**by** (*rule subset-trans [OF analz-Un-implInsecSet-aux-aux], auto*)

**lemma** *analz-Un-implInsecSet:*  
 $\text{Enc-keys-clean } (G \cup H) \implies$   
 $\text{analz } (\text{asym.implInsecSet } G \cup H) \subseteq \text{synth } (\text{analz } (G \cup H)) \cup \text{-payload}$   
**apply** (*rule subset-trans [of - analz (implInsecSet-aux G U H) -]*)  
**apply** (*rule analz-mono, rule Un-mono, blast intro!: implSet-implSet-aux, simp*)  
**apply** (*blast dest: analz-Un-implInsecSet-aux*)  
**done**

### 15.3 Pull *implConfidSet* out of *analz*

**lemma** *analz-Un-implConfidSet-aux-aux:*  
 $\text{Enc-keys-clean } (G \cup H) \implies$   
 $\text{analz } (\text{implConfidSet-aux Ag } G \cup H) \subseteq$   
 $\text{implConfidSet-aux Ag } G \cup \text{PairSet AgentSet } G \cup$   
 $\text{synth } (\text{analz } (G \cup H))$   
**apply** (*rule subset-trans [OF analz-Un-AencSet], blast, blast*)  
**apply** (*simp only: Un-assoc, rule Un-mono, simp*)  
**apply** (*rule subset-trans [OF analz-Un-PairAgentSet], blast*)  
**done**

**lemma** *analz-Un-implConfidSet-aux:*  
 $\text{Enc-keys-clean } (G \cup H) \implies$   
 $\text{analz } (\text{implConfidSet-aux Ag } G \cup H) \subseteq \text{synth } (\text{analz } (G \cup H)) \cup \text{-payload}$   
**by** (*rule subset-trans [OF analz-Un-implConfidSet-aux-aux], auto*)

**lemma** *analz-Un-implConfidSet:*  
 $\text{Enc-keys-clean } (G \cup H) \implies$   
 $\text{analz } (\text{asym.implConfidSet Ag } G \cup H) \subseteq \text{synth } (\text{analz } (G \cup H)) \cup \text{-payload}$   
**apply** (*rule subset-trans [of - analz (implConfidSet-aux Ag G U H) -]*)  
**apply** (*rule analz-mono, rule Un-mono, blast intro!: implSet-implSet-aux, simp*)  
**using** *analz-Un-implConfidSet-aux* **apply** *blast*

**done**

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

**lemma** *analz-Un-implConfidSet-aux-aux-2*:

*Enc-keys-clean H*  $\implies$

$Ag \cap broken (parts H \cap range LtK) = \{\} \implies$

$analz (implConfidSet-aux Ag G \cup H) \subseteq implConfidSet-aux Ag G \cup synth (analz H)$

**apply** (rule *subset-trans* [*OF analz-Un-AencSet2*], *simp*)

**apply** (auto dest:*analz-into-parts*)

**done**

**lemma** *analz-Un-implConfidSet-aux-2*:

*Enc-keys-clean H*  $\implies$

$Ag \cap broken (parts H \cap range LtK) = \{\} \implies$

$analz (implConfidSet-aux Ag G \cup H) \subseteq synth (analz H) \cup -payload$

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

**lemma** *analz-Un-implConfidSet-2*:

*Enc-keys-clean H*  $\implies$

$Ag \cap broken (parts H \cap range LtK) = \{\} \implies$

$analz (asym.implConfidSet Ag G \cup H) \subseteq synth (analz H) \cup -payload$

**apply** (rule *subset-trans* [*of - analz (implConfidSet-aux Ag G \cup H) -*])

**apply** (rule *analz-mono*, rule *Un-mono*, *blast intro!*: *implSet-implSet-aux*, *simp*)

**using** *analz-Un-implConfidSet-aux-2* **apply** auto

**done**

## 15.4 Pull *implAuthSet* out of *analz*

**lemma** *analz-Un-implAuthSet-aux-aux*:

*Enc-keys-clean (G \cup H)*  $\implies$

$analz (implAuthSet-aux G \cup H) \subseteq implAuthSet-aux G \cup synth (analz (G \cup H))$

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

**apply** (rule *Un-mono*, *blast*)

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

**done**

**lemma** *analz-Un-implAuthSet-aux*:

*Enc-keys-clean (G \cup H)*  $\implies$

$analz (implAuthSet-aux G \cup H) \subseteq synth (analz (G \cup H)) \cup -payload$

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

**lemma** *analz-Un-implAuthSet*:

*Enc-keys-clean (G \cup H)*  $\implies$

$analz (asym.implAuthSet G \cup H) \subseteq synth (analz (G \cup H)) \cup -payload$

**apply** (rule *subset-trans* [*of - analz (implAuthSet-aux G \cup H) -*])

**apply** (rule *analz-mono*, rule *Un-mono*, *blast intro!*: *implSet-implSet-aux*, *simp*)

**using** *analz-Un-implAuthSet-aux* **apply** *blast*

**done**

## 15.5 Pull *implSecureSet* out of *analz*

**lemma** *analz-Un-implSecureSet-aux-aux*:

*Enc-keys-clean (G \cup H)*  $\implies$

$analz (implSecureSet-aux Ag G \cup H) \subseteq$   
 $implSecureSet-aux Ag G \cup AencSet (PairSet Tags (PairSet AgentSet G)) (pubK' (Ag^{UNIV})) \cup$   
 $PairSet Tags (PairSet AgentSet G) \cup Tags \cup PairSet AgentSet G \cup$   
 $synth (analz (G \cup H))$   
**apply** (rule subset-trans [OF analz-Un-SignSet], blast, blast)  
**apply** (simp only: Un-assoc, rule Un-mono, simp)  
**apply** (rule subset-trans [OF analz-Un-AencSet], blast, blast)  
**apply** (rule Un-mono, simp)  
**apply** (rule subset-trans [OF analz-Un-PairSet], rule Un-mono, simp, simp only: Un-assoc)  
**apply** (rule subset-trans [OF analz-Un-Tag], blast)  
**apply** (rule Un-mono, simp)  
**apply** (rule subset-trans [OF analz-Un-PairAgentSet], blast)  
**done**

**lemma** analz-Un-implSecureSet-aux:

$Enc-keys-clean (G \cup H) \implies$   
 $analz (implSecureSet-aux Ag G \cup H) \subseteq synth (analz (G \cup H)) \cup -payload$   
**by** (rule subset-trans [OF analz-Un-implSecureSet-aux-aux], auto)

**lemma** analz-Un-implSecureSet:

$Enc-keys-clean (G \cup H) \implies$   
 $analz (asym.implSecureSet Ag G \cup H) \subseteq synth (analz (G \cup H)) \cup -payload$   
**apply** (rule subset-trans [of - analz (implSecureSet-aux Ag G \cup H) -])  
**apply** (rule analz-mono, rule Un-mono, blast intro!: implSet-implSet-aux, simp)  
**using** analz-Un-implSecureSet-aux **apply** blast  
**done**

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

**lemma** analz-Un-implSecureSet-aux-aux-2:

$Enc-keys-clean (G \cup H) \implies$   
 $Ag \cap broken (parts H \cap range LtK) = \{\} \implies$   
 $analz (implSecureSet-aux Ag G \cup H) \subseteq$   
 $implSecureSet-aux Ag G \cup AencSet (PairSet Tags (PairSet AgentSet G)) (pubK' (Ag^{UNIV})) \cup$   
 $synth (analz H)$   
**apply** (rule subset-trans [OF analz-Un-SignSet], blast, blast)  
**apply** (simp only: Un-assoc, rule Un-mono, simp)  
**apply** (rule subset-trans [OF analz-Un-AencSet2], simp)  
**apply** (auto dest: analz-into-parts)  
**done**

**lemma** analz-Un-implSecureSet-aux-2:

$Enc-keys-clean (G \cup H) \implies$   
 $Ag \cap broken (parts H \cap range LtK) = \{\} \implies$   
 $analz (implSecureSet-aux Ag G \cup H) \subseteq synth (analz H) \cup -payload$   
**by** (rule subset-trans [OF analz-Un-implSecureSet-aux-aux-2], auto)

**lemma** analz-Un-implSecureSet-2:

$Enc-keys-clean (G \cup H) \implies$   
 $Ag \cap broken (parts H \cap range LtK) = \{\} \implies$   
 $analz (asym.implSecureSet Ag G \cup H) \subseteq$   
 $synth (analz H) \cup -payload$   
**apply** (rule subset-trans [of - analz (implSecureSet-aux Ag G \cup H) -])  
**apply** (rule analz-mono, rule Un-mono, blast intro!: implSet-implSet-aux, simp)

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

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

## 15.6 Locale interpretations

**interpretation** *asym*: *semivalid-implem implem-asym*

**proof** (*unfold-locales*)

**fix**  $x A B M x' A' B' M'$

**show**  $\text{implem-asym } (\text{Chan } x A B M) = \text{implem-asym } (\text{Chan } x' A' B' M') \longleftrightarrow$   
 $x = x' \wedge A = A' \wedge B = B' \wedge M = M'$

**by** (*cases x, (cases x', auto)+*)

**next**

**fix**  $M' M x x' A A' B B'$

**assume**  $M' \in \text{payload } \text{implem-asym } (\text{Chan } x A B M) \in \text{parts } \{\text{implem-asym } (\text{Chan } x' A' B' M')\}$

**then show**  $x = x' \wedge A = A' \wedge B = B' \wedge M = M'$

**by** (*cases x, auto, (cases x', auto)+*)

**next**

**fix**  $I$

**assume**  $I \subseteq \text{asym.valid}$

**then show** *Enc-keys-clean I*

**proof** (*simp add: Enc-keys-clean-def, intro allI impI*)

**fix**  $X Y$

**assume**  $\text{Enc } X Y \in \text{parts } I$

**obtain**  $x A B M$  **where**  $M \in \text{payload}$  **and**  $\text{Enc } X Y \in \text{parts } \{\text{implem-asym } (\text{Chan } x A B M)\}$

**using** *parts-singleton [OF (Enc X Y ∈ parts I)] (I ⊆ asym.valid)*

**by** (*auto elim!: asym.validE*)

**then show**  $Y \in \text{range } \text{LtK} \vee Y \in \text{payload}$  **by** (*cases x, auto*)

**qed**

**next**

**fix**  $Z$

**show** *composed (implem-asym Z)*

**proof** (*cases Z, simp*)

**fix**  $x A B M$

**show** *composed (implem-asym (Chan x A B M))* **by** (*cases x, auto*)

**qed**

**next**

**fix**  $x A B M$

**show**  $\text{implem-asym } (\text{Chan } x A B M) \notin \text{payload}$

**by** (*cases x, auto*)

**next**

**fix**  $X K$

**assume**  $X \in \text{asym.valid}$

**then obtain**  $x A B M$  **where**  $M \in \text{payload}$   $X = \text{implem-asym } (\text{Chan } x A B M)$

**by** (*auto elim!: asym.validE*)

**then show**  $\text{LtK } K \notin \text{parts } \{X\}$

**by** (*cases x, auto*)

**next**

**fix**  $G H$

**assume**  $G \subseteq \text{payload } \text{Enc-keys-clean } H$

**hence** *Enc-keys-clean (G ∪ H)* **by** (*auto intro: Enc-keys-clean-Un*)

**then show**  $\text{analz } (\{\text{implem-asym } (\text{Insec } A B M) \mid A B M. M \in G\} \cup H) \subseteq$

```

      synth (analz (G ∪ H)) ∪ - payload
    by (rule analz-Un-implInsecSet)
next
fix G H
assume G ⊆ payload Enc-keys-clean H
hence Enc-keys-clean (G ∪ H) by (auto intro: Enc-keys-clean-Un)
then show analz ({imlem-asym (Auth A B M) | A B M. M ∈ G} ∪ H) ⊆
  synth (analz (G ∪ H)) ∪ - payload
  by (rule analz-Un-implAuthSet)
next
fix G H Ag
assume G ⊆ payload Enc-keys-clean H
hence Enc-keys-clean (G ∪ H) by (auto intro: Enc-keys-clean-Un)
then show analz ({imlem-asym (Confid A B M) | A B M. (A, B) ∈ Ag ∧ M ∈ G} ∪ H) ⊆
  synth (analz (G ∪ H)) ∪ - payload
  by (rule analz-Un-implConfidSet)
next
fix G H Ag
assume G ⊆ payload Enc-keys-clean H
hence Enc-keys-clean (G ∪ H) by (auto intro: Enc-keys-clean-Un)
then show analz ({imlem-asym (Secure A B M) | A B M. (A, B) ∈ Ag ∧ M ∈ G} ∪ H) ⊆
  synth (analz (G ∪ H)) ∪ - payload
  by (rule analz-Un-implSecureSet)
next
fix G H Ag
assume G ⊆ payload
assume Enc-keys-clean H
moreover assume Ag ∩ broken (parts H ∩ range LtK) = {}
ultimately show analz ({imlem-asym (Confid A B M) | A B M. (A, B) ∈ Ag ∧ M ∈ G} ∪ H) ⊆
  synth (analz H) ∪ - payload
  by (rule analz-Un-implConfidSet-2)
next
fix G H Ag
assume G ⊆ payload Enc-keys-clean H
hence Enc-keys-clean (G ∪ H) by (auto intro: Enc-keys-clean-Un)
moreover assume Ag ∩ broken (parts H ∩ range LtK) = {}
ultimately
  show analz ({imlem-asym (Secure A B M) | A B M. (A, B) ∈ Ag ∧ M ∈ G} ∪ H) ⊆
    synth (analz H) ∪ - payload
  by (rule analz-Un-implSecureSet-2)
qed

```

Third step: *valid-imlem*. The lemmas giving conditions on  $M$ ,  $A$  and  $B$  for

$implXXX A B M \in synth (analz Z)$

.

**lemma** *implInsec-synth-analz*:

$H \subseteq payload \cup asym.valid \cup range LtK \cup Tags \implies$   
 $asym.implInsec A B M \in synth (analz H) \implies$   
 $asym.implInsec A B M \in I \vee M \in synth (analz H)$

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

**done**



**lemma** *implConfid-synth-analz*:

$H \subseteq \text{payload} \cup \text{asym.valid} \cup \text{range LtK} \cup \text{Tags} \implies$   
 $\text{asym.implConfid } A \ B \ M \in \text{synth } (\text{analz } H) \implies$   
 $\text{asym.implConfid } A \ B \ M \in H \vee M \in \text{synth } (\text{analz } H)$

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

**apply** (*frule asym.analz-valid* [**where**  $x=\text{confid}$ ], *auto*)

**done**

**lemma** *implAuth-synth-analz*:

$H \subseteq \text{payload} \cup \text{asym.valid} \cup \text{range LtK} \cup \text{Tags} \implies$   
 $\text{asym.implAuth } A \ B \ M \in \text{synth } (\text{analz } H) \implies$   
 $\text{asym.implAuth } A \ B \ M \in H \vee (M \in \text{synth } (\text{analz } H) \wedge (A, B) \in \text{broken } H)$

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

**apply** (*frule asym.analz-valid* [**where**  $x=\text{auth}$ ], *auto*)

**apply** (*frule asym.analz-valid* [**where**  $x=\text{auth}$ ], *auto*)

**apply** (*blast dest: asym.analz-LtKeys*)

**done**

**lemma** *implSecure-synth-analz*:

$H \subseteq \text{payload} \cup \text{asym.valid} \cup \text{range LtK} \cup \text{Tags} \implies$   
 $\text{asym.implSecure } A \ B \ M \in \text{synth } (\text{analz } H) \implies$   
 $\text{asym.implSecure } A \ B \ M \in H \vee (M \in \text{synth } (\text{analz } H) \wedge (A, B) \in \text{broken } H)$

**using** [*simproc del: defined-all*] **proof** (*erule synth.cases*, *simp-all*)

**fix**  $X$

**assume**  $H: H \subseteq \text{payload} \cup \text{asym.valid} \cup \text{range LtK} \cup \text{Tags}$

**assume**  $H': \text{Sign } (Aenc \langle \text{SecureTag}, \text{Agent } A, M \rangle (\text{pubK } B)) (\text{priK } A) = X$   
 $X \in \text{analz } H$

**hence**  $\text{asym.implSecure } A \ B \ M \in \text{analz } H$  **by** *auto*

**with**  $H$  **have**  $\text{asym.implSecure } A \ B \ M \in H$  **by** (*rule asym.analz-valid*)

**with**  $H'$  **show**  $X \in H \vee M \in \text{synth } (\text{analz } H) \wedge (A, B) \in \text{broken } H$

**by** *auto*

**next**

**fix**  $X \ Y$

**assume**  $H: H \subseteq \text{payload} \cup \text{asym.valid} \cup \text{range LtK} \cup \text{Tags}$

**assume**  $H': Aenc \langle \text{SecureTag}, \text{Agent } A, M \rangle (\text{pubK } B) = X \wedge \text{priK } A = Y$   
 $X \in \text{synth } (\text{analz } H) \ Y \in \text{synth } (\text{analz } H)$

**hence**  $\text{priK } A \in \text{analz } H$  **by** *auto*

**with**  $H$  **have**  $H\text{Agents}: (A, B) \in \text{broken } H$  **by** (*auto dest: asym.analz-LtKeys*)

**from**  $H'$  **have**  $Aenc \langle \text{SecureTag}, \text{Agent } A, M \rangle (\text{pubK } B) \in \text{synth } (\text{analz } H)$  **by** *auto*

**then have**  $Aenc \langle \text{SecureTag}, \text{Agent } A, M \rangle (\text{pubK } B) \in \text{analz } H \vee$

$M \in \text{synth } (\text{analz } H)$

**by** (*rule synth.cases*, *auto*)

**then show**  $\text{Sign } X \ Y \in H \vee M \in \text{synth } (\text{analz } H) \wedge (A, B) \in \text{broken } H$

**proof**

**assume**  $M \in \text{synth } (\text{analz } H)$

**with**  $H\text{Agents}$  **show** *?thesis* **by** *auto*

**next**

**assume**  $Aenc \langle \text{SecureTag}, \text{Agent } A, M \rangle (\text{pubK } B) \in \text{analz } H$

**hence**  $Aenc \langle \text{SecureTag}, \text{Agent } A, M \rangle (\text{pubK } B) \in \text{parts } H$  **by** (*rule analz-into-parts*)

**from**  $H$  **obtain**  $Z$  **where**

$Z \in H$  **and**  $H'': Aenc \langle \text{SecureTag}, \text{Agent } A, M \rangle (\text{pubK } B) \in \text{parts } \{Z\}$

**using** *parts-singleton* [*OF*  $\langle Aenc \langle \text{SecureTag}, \text{Agent } A, M \rangle (\text{pubK } B) \in \text{parts } H \rangle$ ]

by blast  
 moreover with  $H$  have  $Z \in \text{asym.valid}$  by (auto dest!: subsetD)  
 moreover with  $H''$  have  $Z = \text{asym.implSecure } A \ B \ M$   
 by (auto) (erule asym.valid-cases, auto)  
 ultimately have  $\text{asym.implSecure } A \ B \ M \in H$  by auto  
 with  $H'$  show ?thesis by auto  
 qed  
 qed

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

## 16 Key Transport Protocol with PFS (L1)

```
theory pfsbl1
imports Runs Secrecy AuthenticationI Payloads
begin
```

```
declare option.split-asm [split]
declare domIff [simp, iff del]
```

### 16.1 State and Events

```
consts
  sk :: nat
  kE :: nat
  Nend :: nat
```

Proofs break if  $1::'a$  is used, because *simp* replaces it with *Suc 0*...

```
abbreviation
  xpkE  $\equiv$  Var 0
```

```
abbreviation
  xskE  $\equiv$  Var 2
```

```
abbreviation
  xsk  $\equiv$  Var 3
```

```
abbreviation
  xEnd  $\equiv$  Var 4
```

```
abbreviation
  End  $\equiv$  Number Nend
```

domain of each role (protocol dependent)

```
fun domain :: role-t  $\Rightarrow$  var set where
  domain Init = {xpkE, xskE, xsk}
| domain Resp = {xpkE, xsk}
```

```
consts
  test :: rid-t
```

```
consts
  guessed-runs :: rid-t  $\Rightarrow$  run-t
  guessed-frame :: rid-t  $\Rightarrow$  frame
```

specification of the guessed frame

1. Domain

2. Well-typedness. The messages in the frame of a run never contain implementation material even if the agents of the run are dishonest. Therefore we consider only well-typed frames. This is notably required for the session key compromise; it also helps proving the partitionning of ik, since we know that the messages added by the protocol do not contain ltkeys in their payload and are therefore valid implementations.
3. We also ensure that the values generated by the frame owner are correctly guessed.

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

**abbreviation**  
 $test-owner \equiv owner\ (guessed-runs\ test)$

**abbreviation**  
 $test-partner \equiv partner\ (guessed-runs\ test)$

level 1 state

**record** *l1-state* =  
 $s0-state +$   
 $progress :: progress-t$   
 $signals :: signal \Rightarrow nat$

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

**abbreviation**  
 $run-ended :: var\ set\ option \Rightarrow bool$   
**where**  
 $run-ended\ r \equiv in-progress\ r\ xsk$

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

*test-ended*  $s \longleftrightarrow$  the test run has ended in  $s$

**abbreviation**

*test-ended*  $:: 'a\ l1\text{-state-scheme} \Rightarrow \text{bool}$

**where**

*test-ended*  $s \equiv \text{run-ended } (\text{progress } s \text{ test})$

a run can emit signals if it involves the same agents as the test run, and if the test run has not ended yet

**definition**

*can-signal*  $:: 'a\ l1\text{-state-scheme} \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow \text{bool}$

**where**

*can-signal*  $s\ A\ B \equiv$

$((A = \text{test-owner} \wedge B = \text{test-partner}) \vee (B = \text{test-owner} \wedge A = \text{test-partner})) \wedge$

$\neg \text{test-ended } s$

events

**definition**

*l1-learn*  $:: \text{msg} \Rightarrow ('a\ l1\text{-state-scheme} * 'a\ l1\text{-state-scheme})\ \text{set}$

**where**

*l1-learn*  $m \equiv \{(s, s')\}.$

— guard

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

— action

$s' = s\ (\text{ik} := \text{ik } s \cup \{m\})$

}

protocol events

- step 1: create  $Ra$ ,  $A$  generates  $pkE$ ,  $skE$
- step 2: create  $Rb$ ,  $B$  reads  $pkE$  authentically, generates  $K$ , emits a running signal for  $A$ ,  $B$ ,  $(pkE, K)$
- step 3:  $A$  reads  $K$  and  $pkE$  authentically, emits a commit signal for  $A$ ,  $B$ ,  $(pkE, K)$

**definition**

*l1-step1*  $:: \text{rid-t} \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow ('a\ l1\text{-state-scheme} * 'a\ l1\text{-state-scheme})\ \text{set}$

**where**

*l1-step1*  $Ra\ A\ B \equiv \{(s, s')\}.$

— guards:

$Ra \notin \text{dom } (\text{progress } s) \wedge$

$\text{guessed-runs } Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \wedge$

— actions:

$s' = s(\text{progress} := (\text{progress } s)(Ra \mapsto \{xpkE, xskE\}))$

}

**definition**

*l1-step2*  $:: \text{rid-t} \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow \text{msg} \Rightarrow ('a\ l1\text{-state-scheme} * 'a\ l1\text{-state-scheme})\ \text{set}$



```

signals =  $\lambda x. 0$ 
})

```

**definition**

```

l1-trans :: ('a l1-state-scheme * 'a l1-state-scheme) set where
l1-trans  $\equiv$  ( $\bigcup m Ra Rb A B K KE.$ 
  l1-step1 Ra A B  $\cup$ 
  l1-step2 Rb A B KE  $\cup$ 
  l1-step3 Ra A B K  $\cup$ 
  l1-learn m  $\cup$ 
  Id
)

```

**definition**

```

l1 :: (l1-state, l1-obs) spec where
l1  $\equiv$  ()
  init = l1-init,
  trans = l1-trans,
  obs = id
)

```

**lemmas** l1-defs =

```

l1-def l1-init-def l1-trans-def
l1-learn-def
l1-step1-def l1-step2-def l1-step3-def

```

**lemmas** l1-nostep-defs =

```

l1-def l1-init-def l1-trans-def

```

**lemma** l1-obs-id [simp]: obs l1 = id

**by** (simp add: l1-def)

**declare** domIff [iff]

**lemma** run-ended-trans:

```

run-ended (progress s R)  $\implies$ 
(s, s')  $\in$  trans l1  $\implies$ 
run-ended (progress s' R)

```

**apply** (auto simp add: l1-nostep-defs)

**apply** (simp add: l1-defs ik-dy-def, fast ?)+

**done**

**declare** domIff [iff del]

**lemma** can-signal-trans:

```

can-signal s' A B  $\implies$ 
(s, s')  $\in$  trans l1  $\implies$ 
can-signal s A B

```

**by** (auto simp add: can-signal-def run-ended-trans)

## 16.2 Refinement: secrecy

mediator function

**definition**

$med01s :: l1-obs \Rightarrow s0-obs$

**where**

$med01s\ t \equiv (\ ik = ik\ t, secret = secret\ t\ )$

relation between states

**definition**

$R01s :: (s0-state * l1-state)\ set$

**where**

$R01s \equiv \{(s, s') .$   
 $s = (\ ik = ik\ s', secret = secret\ s')$   
 $\}$

protocol independent events

**lemma** *l1-learn-refines-learn:*

$\{R01s\}\ s0-learn\ m, l1-learn\ m\ \{>R01s\}$

**apply** (*simp add: PO-rhoare-defs R01s-def*)

**apply** *auto*

**apply** (*simp add: l1-defs s0-defs s0-secrecy-def*)

**done**

protocol events

**lemma** *l1-step1-refines-skip:*

$\{R01s\}\ Id, l1-step1\ Ra\ A\ B\ \{>R01s\}$

**by** (*auto simp add: PO-rhoare-defs R01s-def l1-step1-def*)

**lemma** *l1-step2-refines-add-secret-skip:*

$\{R01s\}\ s0-add-secret\ (NonceF\ (Rb\$sk)) \cup Id, l1-step2\ Rb\ A\ B\ KE\ \{>R01s\}$

**apply** (*auto simp add: PO-rhoare-defs R01s-def s0-add-secret-def*)

**apply** (*auto simp add: l1-step2-def*)

**done**

**lemma** *l1-step3-refines-add-secret-skip:*

$\{R01s\}\ s0-add-secret\ K \cup Id, l1-step3\ Ra\ A\ B\ K\ \{>R01s\}$

**apply** (*auto simp add: PO-rhoare-defs R01s-def s0-add-secret-def*)

**apply** (*auto simp add: l1-step3-def*)

**done**

refinement proof

**lemmas** *l1-trans-refines-s0-trans =*

*l1-learn-refines-learn*

*l1-step1-refines-skip l1-step2-refines-add-secret-skip l1-step3-refines-add-secret-skip*

**lemma** *l1-refines-init-s0 [iff]:*

$init\ l1 \subseteq R01s\ \text{“}\ (init\ s0)$

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



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

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

refinement result

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

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

### 16.3 Derived invariants: secrecy

**abbreviation** *l1-secrecy*  $\equiv$  *s0-secrecy*

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

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

### 16.4 Invariants

#### 16.4.1 inv1

if a commit signal for a nonce has been emitted, then there is a finished initiator run with this nonce.

**definition**

*l1-inv1* :: l1-state set

**where**

$l1-inv1 \equiv \{s. \forall Ra A B K.$   
 signals s (Commit A B  $\langle epubKF (Ra\$kE), K \rangle$ )  $> 0 \longrightarrow$   
 guessed-runs Ra = (role=Init, owner=A, partner=B)  $\wedge$   
 progress s Ra = Some {xpkE, xskE, xsk}  $\wedge$   
 guessed-frame Ra xsk = Some K  
 $\}$

**lemmas** *l1-inv1I* = l1-inv1-def [THEN setc-def-to-intro, rule-format]

**lemmas** *l1-inv1E* [elim] = l1-inv1-def [THEN setc-def-to-elim, rule-format]

**lemmas**  $l1\text{-inv}1D = l1\text{-inv}1\text{-def}$  [THEN setc-def-to-dest, rule-format, rotated 1, simplified]

**lemma**  $l1\text{-inv}1\text{-init}$  [iff]:

$init\ l1 \subseteq l1\text{-inv}1$

**by** (auto simp add:  $l1\text{-def}$   $l1\text{-init}\text{-def}$   $l1\text{-inv}1\text{-def}$ )

**declare**  $domIff$  [iff]

**lemma**  $l1\text{-inv}1\text{-trans}$  [iff]:

$\{l1\text{-inv}1\}$  trans  $l1 \{> l1\text{-inv}1\}$

**apply** (auto simp add: PO-hoare-defs  $l1\text{-nostep}\text{-defs}$  intro!:  $l1\text{-inv}1I$ )

**apply** (auto simp add:  $l1\text{-defs}$  ik-dy-def  $l1\text{-inv}1\text{-def}$ )

**done**

**lemma**  $PO\text{-}l1\text{-inv}1$  [iff]: reach  $l1 \subseteq l1\text{-inv}1$

**by** (rule inv-rule-basic) (auto)

## 16.4.2 inv2

if a responder run knows a nonce, then a running signal for this nonce has been emitted

**definition**

$l1\text{-inv}2 :: l1\text{-state set}$

**where**

$l1\text{-inv}2 \equiv \{s. \forall KE\ A\ B\ Rb.$

$guessed\text{-runs}\ Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \longrightarrow$

$progress\ s\ Rb = \text{Some}\ \{xpkE, xsk\} \longrightarrow$

$guessed\text{-frame}\ Rb\ xpkE = \text{Some}\ KE \longrightarrow$

$can\text{-signal}\ s\ A\ B \longrightarrow$

$signals\ s\ (\text{Running}\ A\ B\ \langle KE, \text{NonceF}\ (Rb\$sk)\rangle) > 0$

$\}$

**lemmas**  $l1\text{-inv}2I = l1\text{-inv}2\text{-def}$  [THEN setc-def-to-intro, rule-format]

**lemmas**  $l1\text{-inv}2E$  [elim] =  $l1\text{-inv}2\text{-def}$  [THEN setc-def-to-elim, rule-format]

**lemmas**  $l1\text{-inv}2D = l1\text{-inv}2\text{-def}$  [THEN setc-def-to-dest, rule-format, rotated 1, simplified]

**lemma**  $l1\text{-inv}2\text{-init}$  [iff]:

$init\ l1 \subseteq l1\text{-inv}2$

**by** (auto simp add:  $l1\text{-def}$   $l1\text{-init}\text{-def}$   $l1\text{-inv}2\text{-def}$ )

**lemma**  $l1\text{-inv}2\text{-trans}$  [iff]:

$\{l1\text{-inv}2\}$  trans  $l1 \{> l1\text{-inv}2\}$

**apply** (auto simp add: PO-hoare-defs intro!:  $l1\text{-inv}2I$ )

**apply** (drule can-signal-trans, assumption)

**apply** (auto simp add:  $l1\text{-nostep}\text{-defs}$ )

**apply** (auto simp add:  $l1\text{-defs}$  ik-dy-def  $l1\text{-inv}2\text{-def}$ )

**done**

**lemma**  $PO\text{-}l1\text{-inv}2$  [iff]: reach  $l1 \subseteq l1\text{-inv}2$

**by** (rule inv-rule-basic) (auto)

### 16.4.3 inv3 (derived)

if an unfinished initiator run and a finished responder run both know the same nonce, then the number of running signals for this nonce is strictly greater than the number of commit signals. (actually, there are 0 commit and 1 running)

#### definition

$l1\text{-inv3} :: l1\text{-state set}$

#### where

$$\begin{aligned}
 l1\text{-inv3} &\equiv \{s. \forall A B Rb Ra. \\
 &\text{guessed-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \longrightarrow \\
 &\text{progress } s Rb = \text{Some } \{xpkE, xsk\} \longrightarrow \\
 &\text{guessed-frame } Rb \ xpkE = \text{Some } (\text{epubKF } (Ra\$kE)) \longrightarrow \\
 &\text{guessed-runs } Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \longrightarrow \\
 &\text{progress } s Ra = \text{Some } \{xpkE, xskE\} \longrightarrow \\
 &\text{can-signal } s A B \longrightarrow \\
 &\text{signals } s (\text{Commit } A B (\langle \text{epubKF } (Ra\$kE), \text{NonceF } (Rb\$sk) \rangle)) \\
 &< \text{signals } s (\text{Running } A B (\langle \text{epubKF } (Ra\$kE), \text{NonceF } (Rb\$sk) \rangle)) \\
 &\}
 \end{aligned}$$

**lemmas**  $l1\text{-inv3I} = l1\text{-inv3-def}$  [THEN setc-def-to-intro, rule-format]

**lemmas**  $l1\text{-inv3E}$  [elim] =  $l1\text{-inv3-def}$  [THEN setc-def-to-elim, rule-format]

**lemmas**  $l1\text{-inv3D} = l1\text{-inv3-def}$  [THEN setc-def-to-dest, rule-format, rotated 1, simplified]

**lemma**  $l1\text{-inv3-derived}$ :  $l1\text{-inv1} \cap l1\text{-inv2} \subseteq l1\text{-inv3}$

**apply** (auto intro!:  $l1\text{-inv3I}$ )

**apply** (auto dest!:  $l1\text{-inv2D}$ )

**apply** (rename-tac  $x A B Rb Ra$ )

**apply** (case-tac signals  $x (\text{Commit } A B (\langle \text{epubKF } (Ra \$ kE), \text{NonceF } (Rb \$ sk) \rangle)) > 0$ , auto)

**apply** (fastforce dest:  $l1\text{-inv1D}$  elim: equalityE)

**done**

## 16.5 Refinement: injective agreement

mediator function

#### definition

$med01ia :: l1\text{-obs} \Rightarrow a0i\text{-obs}$

#### where

$med01ia t \equiv (\text{a0n-state.signals} = \text{signals } t)$

relation between states

#### definition

$R01ia :: (a0i\text{-state} * l1\text{-state}) \text{ set}$

#### where

$$\begin{aligned}
 R01ia &\equiv \{(s, s'). \\
 &\text{a0n-state.signals } s = \text{signals } s' \\
 &\}
 \end{aligned}$$

protocol independent events

**lemma**  $l1\text{-learn-refines-a0-ia-skip}$ :

$\{R01ia\} Id, l1\text{-learn } m \{>R01ia\}$

**apply** (auto simp add: PO-rhoare-defs  $R01ia\text{-def}$ )

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

protocol events

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

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

**lemma** *l1-step3-refines-a0i-commit-skip*:  
 $\{R01ia \cap (UNIV \times l1-inv3)\} a0i-commit A B \langle epubKF (Ra\$kE), K \rangle \cup Id, l1-step3 Ra A B K \{>R01ia\}$   
**apply** (*auto simp add: PO-rhoare-defs R01ia-def*)  
**apply** (*auto simp add: l1-step3-def a0i-commit-def*)  
**apply** (*force elim!: l1-inv3E*)  
**done**

refinement proof

**lemmas** *l1-trans-refines-a0i-trans =*  
*l1-learn-refines-a0-ia-skip*  
*l1-step1-refines-a0i-skip l1-step2-refines-a0i-running-skip l1-step3-refines-a0i-commit-skip*

**lemma** *l1-refines-init-a0i [iff]*:  
 $init\ l1 \subseteq R01ia \text{ “ } (init\ a0i)$   
**by** (*auto simp add: R01ia-def a0i-defs l1-defs*)

**lemma** *l1-refines-trans-a0i [iff]*:  
 $\{R01ia \cap (UNIV \times (l1-inv1 \cap l1-inv2))\} trans\ a0i, trans\ l1 \{> R01ia\}$

**proof** –  
**let**  $?pre' = R01ia \cap (UNIV \times l1-inv3)$   
**show**  $?thesis$  (**is**  $\{?pre\} ?t1, ?t2 \{> ?post\}$ )  
**proof** (*rule relhoare-conseq-left*)  
**show**  $?pre \subseteq ?pre'$   
**using** *l1-inv3-derived* **by** *blast*  
**next**  
**show**  $\{?pre'\} ?t1, ?t2 \{> ?post\}$   
**apply** (*auto simp add: a0i-def l1-def a0i-trans-def l1-trans-def*)  
**prefer** 2 **using** *l1-step2-refines-a0i-running-skip* **apply** (*simp add: PO-rhoare-defs, blast*)  
**prefer** 2 **using** *l1-step3-refines-a0i-commit-skip* **apply** (*simp add: PO-rhoare-defs, blast*)  
**apply** (*blast intro!: l1-trans-refines-a0i-trans*)  
**done**  
**qed**  
**qed**

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

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

refinement result

**lemma** *l1-refines-a0i* [*iff*]:

*refines*  
( $R01ia \cap (\text{reach } a0i \times (l1\text{-inv1} \cap l1\text{-inv2}))$ )  
*med01ia a0i l1*

**by** (*rule Refinement-using-invariants, auto*)

**lemma** *l1-implements-a0i* [*iff*]: *implements med01ia a0i l1*

**by** (*rule refinement-soundness*) (*fast*)

## 16.6 Derived invariants: injective agreement

**definition**

*l1-agreement* :: (*'a l1-state-scheme*) *set*

**where**

$l1\text{-agreement} \equiv \{s. \forall A B N. \text{signals } s (\text{Commit } A B N) \leq \text{signals } s (\text{Running } A B N)\}$

**lemmas** *l1-agreementI* = *l1-agreement-def* [*THEN setc-def-to-intro, rule-format*]

**lemmas** *l1-agreementE* [*elim*] = *l1-agreement-def* [*THEN setc-def-to-elim, rule-format*]

**lemma** *l1-obs-agreement* [*iff*]: *oreach l1*  $\subseteq$  *l1-agreement*

**apply** (*rule external-invariant-translation*

[*OF PO-a0i-obs-agreement - l1-implements-a0i*])

**apply** (*auto simp add: med01ia-def l1-agreement-def a0i-agreement-def*)

**done**

**lemma** *l1-agreement* [*iff*]: *reach l1*  $\subseteq$  *l1-agreement*

**by** (*rule external-to-internal-invariant* [*OF l1-obs-agreement*], *auto*)

**end**

## 17 Key Transport Protocol with PFS (L2)

```
theory pfsvl2
imports pfsvl1 Channels
begin
```

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

### 17.1 State and Events

initial compromise

**consts**

```
bad-init :: agent set
```

**specification** (*bad-init*)

```
bad-init-spec: test-owner  $\notin$  bad-init  $\wedge$  test-partner  $\notin$  bad-init
```

**by** *auto*

level 2 state

**record** *l2-state* =

```
l1-state +
chan :: chan set
bad :: agent set
```

**type-synonym** *l2-obs* = *l2-state*

**type-synonym**

```
l2-pred = l2-state set
```

**type-synonym**

```
l2-trans = (l2-state  $\times$  l2-state) set
```

attacker events

**definition**

```
l2-dy-fake-msg :: msg  $\Rightarrow$  l2-trans
```

**where**

```
l2-dy-fake-msg m  $\equiv$  {(s, s').
  — guards
  m  $\in$  dy-fake-msg (bad s) (ik s) (chan s)  $\wedge$ 
  — actions
  s' = s(ik := {m}  $\cup$  ik s)
}
```

**definition**

```
l2-dy-fake-chan :: chan  $\Rightarrow$  l2-trans
```

**where**

```
l2-dy-fake-chan M  $\equiv$  {(s, s').
  — guards
  M  $\in$  dy-fake-chan (bad s) (ik s) (chan s)  $\wedge$ 
  — actions
  s' = s(chan := {M}  $\cup$  chan s)
```

```

}

partnering
fun
  role-comp :: role-t ⇒ role-t
where
  role-comp Init = Resp
| role-comp Resp = Init

definition
  matching :: frame ⇒ frame ⇒ bool
where
  matching sigma sigma' ≡ ∀ x. x ∈ dom sigma ∩ dom sigma' ⟶ sigma x = sigma' x

definition
  partner-runs :: rid-t ⇒ rid-t ⇒ bool
where
  partner-runs R R' ≡
    role (guessed-runs R) = role-comp (role (guessed-runs R')) ∧
    owner (guessed-runs R) = partner (guessed-runs R') ∧
    owner (guessed-runs R') = partner (guessed-runs R) ∧
    matching (guessed-frame R) (guessed-frame R')

lemma role-comp-inv [simp]:
  role-comp (role-comp x) = x
by (cases x, auto)

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

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

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

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

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

lemma partner-unique:
  partner-runs R R'' ⟹ partner-runs R R' ⟹ R' = R''
proof –

```

```

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

```

**lemma** *partner-test*:

$R \in \text{partners} \implies \text{partner-runs } R \ R' \implies R' = \text{test}$

**by** (*auto intro!: partner-unique simp add: partners-def partner-symmetric*)

compromising events

**definition**

*l2-lkr-others* :: *agent*  $\Rightarrow$  *l2-trans*

**where**

*l2-lkr-others*  $A \equiv \{(s, s')\}$ .  
 — guards  
 $A \neq \text{test-owner} \wedge$   
 $A \neq \text{test-partner} \wedge$   
 — actions  
 $s' = s(\text{bad} := \{A\} \cup \text{bad } s)$   
 $\}$

**definition**

*l2-lkr-actor* :: *agent*  $\Rightarrow$  *l2-trans*

**where**

*l2-lkr-actor*  $A \equiv \{(s, s')\}$ .  
 — guards  
 $A = \text{test-owner} \wedge$   
 $A \neq \text{test-partner} \wedge$



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

**definition**

$l2\text{-lkr}\text{-after} :: \text{agent} \Rightarrow l2\text{-trans}$

**where**

$l2\text{-lkr}\text{-after } A \equiv \{(s, s')\}.$   
 — guards  
 $\text{test-ended } s \wedge$   
 — actions  
 $s' = s(\text{bad} := \{A\} \cup \text{bad } s)$   
 $\}$

**definition**

$l2\text{-skr} :: \text{rid-t} \Rightarrow \text{msg} \Rightarrow l2\text{-trans}$

**where**

$l2\text{-skr } R K \equiv \{(s, s')\}.$   
 — guards  
 $R \neq \text{test} \wedge R \notin \text{partners} \wedge$   
 $\text{in-progress } (\text{progress } s R) \text{ xsk} \wedge$   
 $\text{guessed-frame } R \text{ xsk} = \text{Some } K \wedge$   
 — actions  
 $s' = s(\text{ik} := \{K\} \cup \text{ik } s)$   
 $\}$

protocol events

**definition**

$l2\text{-step1} :: \text{rid-t} \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow l2\text{-trans}$

**where**

$l2\text{-step1 } Ra A B \equiv \{(s, s')\}.$   
 — guards:  
 $Ra \notin \text{dom } (\text{progress } s) \wedge$   
 $\text{guessed-runs } Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \wedge$   
 — actions:  
 $s' = s(\text{progress} := (\text{progress } s)(Ra \mapsto \{xpKE, xskE\}),$   
 $\text{chan} := \{\text{Auth } A B \langle \text{Number } 0, \text{epubKF } (Ra\$kE) \rangle\} \cup (\text{chan } s)$   
 $\text{)})$   
 $\}$

**definition**

$l2\text{-step2} :: \text{rid-t} \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow \text{msg} \Rightarrow l2\text{-trans}$

**where**

$l2\text{-step2 } Rb A B KE \equiv \{(s, s')\}.$   
 — guards:  
 $\text{guessed-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \wedge$   
 $Rb \notin \text{dom } (\text{progress } s) \wedge$   
 $\text{guessed-frame } Rb \text{ xpKE} = \text{Some } KE \wedge$   
 $\text{Auth } A B \langle \text{Number } 0, KE \rangle \in \text{chan } s \wedge$   
 — actions:  
 $s' = s(\text{progress} := (\text{progress } s)(Rb \mapsto \{xpKE, xsk\}),$

$$\begin{aligned}
& \text{chan} := \{ \text{Auth } B \ A \ ( \text{Aenc } ( \text{NonceF } ( \text{Rb}\$sk ) ) \ KE ) \} \cup ( \text{chan } s ), \\
& \text{signals} := \text{if can-signal } s \ A \ B \ \text{then} \\
& \quad \text{addSignal } ( \text{signals } s ) \ ( \text{Running } A \ B \ \langle \text{KE}, \text{NonceF } ( \text{Rb}\$sk ) \rangle ) \\
& \quad \text{else} \\
& \quad \text{signals } s, \\
& \text{secret} := \{ x. x = \text{NonceF } ( \text{Rb}\$sk ) \wedge \text{Rb} = \text{test} \} \cup \text{secret } s \\
& \quad \Downarrow \\
& \}
\end{aligned}$$

**definition**

$l2\text{-step3} :: \text{rid-t} \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow \text{msg} \Rightarrow l2\text{-trans}$

**where**

$l2\text{-step3 } Ra \ A \ B \ K \equiv \{ (s, s') .$

— guards:

$\text{guessed-runs } Ra = ( \text{role} = \text{Init}, \text{owner} = A, \text{partner} = B ) \wedge$

$\text{progress } s \ Ra = \text{Some } \{ xpkE, xskE \} \wedge$

$\text{guessed-frame } Ra \ xsk = \text{Some } K \wedge$

$\text{Auth } B \ A \ ( \text{Aenc } K \ ( \text{epubKF } ( \text{Ra}\$kE ) ) ) \in \text{chan } s \wedge$

— actions:

$s' = s ( \text{progress} := ( \text{progress } s ) ( Ra \mapsto \{ xpkE, xskE, xsk \} ),$

$\text{signals} := \text{if can-signal } s \ A \ B \ \text{then}$

$\quad \text{addSignal } ( \text{signals } s ) \ ( \text{Commit } A \ B \ \langle \text{epubKF } ( \text{Ra}\$kE ), K \rangle )$

$\text{else}$

$\text{signals } s,$

$\text{secret} := \{ x. x = K \wedge \text{Ra} = \text{test} \} \cup \text{secret } s$

$\Downarrow$

$\}$

specification

**definition**

$l2\text{-init} :: l2\text{-state set}$

**where**

$l2\text{-init} \equiv \{ ($

$\text{ik} = \{ \},$

$\text{secret} = \{ \},$

$\text{progress} = \text{Map.empty},$

$\text{signals} = \lambda x. 0,$

$\text{chan} = \{ \},$

$\text{bad} = \text{bad-init}$

$\Downarrow \}$

**definition**

$l2\text{-trans} :: l2\text{-trans where}$

$l2\text{-trans} \equiv ( \bigcup m \ M \ KE \ Rb \ Ra \ A \ B \ K .$

$\text{l2-step1 } Ra \ A \ B \cup$

$\text{l2-step2 } Rb \ A \ B \ KE \cup$

$\text{l2-step3 } Ra \ A \ B \ m \cup$

$\text{l2-dy-fake-chan } M \cup$

$\text{l2-dy-fake-msg } m \cup$

$\text{l2-lkr-others } A \cup$

$\text{l2-lkr-after } A \cup$

$\text{l2-skr } Ra \ K \cup$

*Id*  
)

**definition**

*l2* :: (*l2-state*, *l2-obs*) spec **where**  
*l2* ≡ (  
  *init* = *l2-init*,  
  *trans* = *l2-trans*,  
  *obs* = *id*  
)

**lemmas** *l2-loc-defs* =

*l2-step1-def l2-step2-def l2-step3-def*  
*l2-def l2-init-def l2-trans-def*  
*l2-dy-fake-chan-def l2-dy-fake-msg-def*  
*l2-lkr-after-def l2-lkr-others-def l2-skr-def*

**lemmas** *l2-defs* = *l2-loc-defs ik-dy-def*

**lemmas** *l2-nostep-defs* = *l2-def l2-init-def l2-trans-def*

**lemma** *l2-obs-id* [*simp*]: *obs l2* = *id*  
**by** (*simp add: l2-def*)

Once a run is finished, it stays finished, therefore if the test is not finished at some point then it was not finished before either

**declare** *domIff* [*iff*]

**lemma** *l2-run-ended-trans*:

*run-ended* (*progress s R*) ⇒  
(*s*, *s'*) ∈ *trans l2* ⇒  
*run-ended* (*progress s' R*)

**apply** (*auto simp add: l2-nostep-defs*)

**apply** (*simp add: l2-defs, fast ?*)<sup>+</sup>

**done**

**declare** *domIff* [*iff del*]

**lemma** *l2-can-signal-trans*:

*can-signal s' A B* ⇒  
(*s*, *s'*) ∈ *trans l2* ⇒  
*can-signal s A B*

**by** (*auto simp add: can-signal-def l2-run-ended-trans*)

## 17.2 Invariants

### 17.2.1 inv1

If *can-signal s A B* (i.e., *A*, *B* are the test session agents and the test is not finished), then *A*, *B* are honest.

**definition**

*l2-inv1* :: *l2-state set*

**where**

$$\begin{aligned} l2\text{-inv1} &\equiv \{s. \forall A B. \\ &\quad \text{can-signal } s \ A \ B \longrightarrow \\ &\quad A \notin \text{bad } s \wedge B \notin \text{bad } s \\ &\} \end{aligned}$$

**lemmas**  $l2\text{-inv1I} = l2\text{-inv1-def}$  [THEN setc-def-to-intro, rule-format]

**lemmas**  $l2\text{-inv1E}$  [elim] =  $l2\text{-inv1-def}$  [THEN setc-def-to-elim, rule-format]

**lemmas**  $l2\text{-inv1D} = l2\text{-inv1-def}$  [THEN setc-def-to-dest, rule-format, rotated 1, simplified]

**lemma**  $l2\text{-inv1-init}$  [iff]:

$\text{init } l2 \subseteq l2\text{-inv1}$

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

**lemma**  $l2\text{-inv1-trans}$  [iff]:

$\{l2\text{-inv1}\} \text{ trans } l2 \ \{> \ l2\text{-inv1}\}$

**proof** (auto simp add: PO-hoare-defs intro!: l2-inv1I del: conjI)

**fix**  $s' \ s :: l2\text{-state}$

**fix**  $A \ B$

**assume**  $HI:s \in l2\text{-inv1}$

**assume**  $HT:(s, s') \in \text{trans } l2$

**assume**  $\text{can-signal } s' \ A \ B$

**with**  $HT$  **have**  $HS:\text{can-signal } s \ A \ B$

**by** (auto simp add: l2-can-signal-trans)

**with**  $HI$  **have**  $A \notin \text{bad } s \wedge B \notin \text{bad } s$

**by** fast

**with**  $HS \ HT$  **show**  $A \notin \text{bad } s' \wedge B \notin \text{bad } s'$

**by** (auto simp add: l2-nostep-defs can-signal-def)

(simp-all add: l2-defs)

**qed**

**lemma**  $PO\text{-}l2\text{-inv1}$  [iff]:  $\text{reach } l2 \subseteq l2\text{-inv1}$

**by** (rule inv-rule-basic) (auto)

## 17.2.2 inv2 (authentication guard)

If  $\text{Auth } A \ B \ \langle \text{Number } 0, \text{KE} \rangle \in \text{chan } s$  and  $A, B$  are honest then the message has indeed been sent by an initiator run (with the right agents etc.)

**definition**

$l2\text{-inv2} :: l2\text{-state set}$

**where**

$l2\text{-inv2} \equiv \{s. \forall A \ B \ \text{KE}.$

$\text{Auth } A \ B \ \langle \text{Number } 0, \text{KE} \rangle \in \text{chan } s \longrightarrow$

$A \notin \text{bad } s \wedge B \notin \text{bad } s \longrightarrow$

$(\exists \text{ Ra}.$

$\text{guessed-runs } \text{Ra} = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \wedge$

$\text{in-progress } (\text{progress } s \ \text{Ra}) \ \text{xpke} \wedge$

$\text{KE} = \text{epubKF } (\text{Ra}\$\text{ke})$

$\}$

**lemmas**  $l2\text{-inv2I} = l2\text{-inv2-def}$  [THEN setc-def-to-intro, rule-format]

**lemmas**  $l2\text{-inv2E}$  [elim] =  $l2\text{-inv2-def}$  [THEN setc-def-to-elim, rule-format]

**lemmas**  $l2\text{-inv}2D = l2\text{-inv}2\text{-def}$  [*THEN setc-def-to-dest, rule-format, rotated 1, simplified*]

**lemma**  $l2\text{-inv}2\text{-init}$  [*iff*]:

$init\ l2 \subseteq l2\text{-inv}2$

**by** (*auto simp add: l2-def l2-init-def l2-inv2-def*)

**lemma**  $l2\text{-inv}2\text{-trans}$  [*iff*]:

$\{l2\text{-inv}2\}\ trans\ l2\ \{>\ l2\text{-inv}2\}$

**apply** (*auto simp add: PO-hoare-defs l2-nostep-defs intro!: l2-inv2I*)

**apply** (*auto simp add: l2-defs dy-fake-chan-def*)

**apply** *force+*

**done**

**lemma**  $PO\text{-}l2\text{-inv}2$  [*iff*]:  $reach\ l2 \subseteq l2\text{-inv}2$

**by** (*rule inv-rule-basic*) (*auto*)

### 17.2.3 inv3 (authentication guard)

If  $Auth\ B\ A\ (Aenc\ K\ (epubKF\ (Ra\ \$\ kE))) \in chan\ s$  and  $A, B$  are honest then the message has indeed been sent by a responder run (etc).

**definition**

$l2\text{-inv}3 :: l2\text{-state\ set}$

**where**

$l2\text{-inv}3 \equiv \{s. \forall\ Ra\ A\ B\ K.$

$Auth\ B\ A\ (Aenc\ K\ (epubKF\ (Ra\ \$\ kE))) \in chan\ s \longrightarrow$

$A \notin bad\ s \wedge B \notin bad\ s \longrightarrow$

$(\exists\ Rb.$

$guessed\text{-runs}\ Rb = (\downarrow role=Resp, owner=B, partner=A) \wedge$

$progress\ s\ Rb = Some\ \{xpkE, xsk\} \wedge$

$guessed\text{-frame}\ Rb\ xpkE = Some\ (epubKF\ (Ra\ \$\ kE)) \wedge$

$K = NonceF\ (Rb\ \$\ sk)$

$)$

$\}$

**lemmas**  $l2\text{-inv}3I = l2\text{-inv}3\text{-def}$  [*THEN setc-def-to-intro, rule-format*]

**lemmas**  $l2\text{-inv}3E$  [*elim*] =  $l2\text{-inv}3\text{-def}$  [*THEN setc-def-to-elim, rule-format*]

**lemmas**  $l2\text{-inv}3D = l2\text{-inv}3\text{-def}$  [*THEN setc-def-to-dest, rule-format, rotated 1, simplified*]

**lemma**  $l2\text{-inv}3\text{-init}$  [*iff*]:

$init\ l2 \subseteq l2\text{-inv}3$

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

**lemma**  $l2\text{-inv}3\text{-trans}$  [*iff*]:

$\{l2\text{-inv}3\}\ trans\ l2\ \{>\ l2\text{-inv}3\}$

**apply** (*auto simp add: PO-hoare-defs l2-nostep-defs intro!: l2-inv3I*)

**apply** (*auto simp add: l2-defs dy-fake-chan-def*)

**apply** (*simp-all add: domIff*)

**apply** *force+*

**done**

**lemma**  $PO\text{-}l2\text{-inv}3$  [*iff*]:  $reach\ l2 \subseteq l2\text{-inv}3$

**by** (*rule inv-rule-basic*) (*auto*)

### 17.2.4 inv4

If the test run is finished and has the session key generated by a run, then this run is also finished.

**definition**

$l2\text{-inv}4 :: l2\text{-state set}$

**where**

$l2\text{-inv}4 \equiv \{s. \forall Rb.$   
 $in\text{-progress } (progress\ s\ test)\ xsk \longrightarrow$   
 $guessed\text{-frame } test\ xsk = Some\ (NonceF\ (Rb\$sk)) \longrightarrow$   
 $progress\ s\ Rb = Some\ \{xpkE, xsk\}$   
 $\}$

**lemmas**  $l2\text{-inv}4I = l2\text{-inv}4\text{-def } [THEN\ setc\text{-def-to-intro, rule-format}]$

**lemmas**  $l2\text{-inv}4E [elim] = l2\text{-inv}4\text{-def } [THEN\ setc\text{-def-to-elim, rule-format}]$

**lemmas**  $l2\text{-inv}4D = l2\text{-inv}4\text{-def } [THEN\ setc\text{-def-to-dest, rule-format, rotated 1, simplified}]$

**lemma**  $l2\text{-inv}4\text{-init } [iff]:$

$init\ l2 \subseteq l2\text{-inv}4$

**by**  $(auto\ simp\ add: l2\text{-def } l2\text{-init-def } l2\text{-inv}4\text{-def})$

**lemma**  $l2\text{-inv}4\text{-trans } [iff]:$

$\{l2\text{-inv}4 \cap l2\text{-inv}3 \cap l2\text{-inv}1\}\ trans\ l2\ \{>\ l2\text{-inv}4\}$

**apply**  $(auto\ simp\ add: PO\text{-hoare-defs } l2\text{-nostep-defs } intro!: l2\text{-inv}4I)$

**apply**  $(auto\ simp\ add: l2\text{-defs } dy\text{-fake-chan-def})$

**apply**  $(auto\ dest!: l2\text{-inv}4D\ simp\ add: domIff\ can\text{-signal-def})$

**apply**  $(auto\ dest!: l2\text{-inv}3D\ intro!: l2\text{-inv}1D\ simp\ add: can\text{-signal-def})$

**done**

**lemma**  $PO\text{-}l2\text{-inv}4 [iff]: reach\ l2 \subseteq l2\text{-inv}4$

**by**  $(rule\text{-tac } J=l2\text{-inv}3 \cap l2\text{-inv}1\ \mathbf{in}\ inv\text{-rule-incr})\ (auto)$

### 17.2.5 inv5

The only confidential or secure messages on the channel have been put there by the attacker.

**definition**

$l2\text{-inv}5 :: l2\text{-state set}$

**where**

$l2\text{-inv}5 \equiv \{s. \forall A\ B\ M.$   
 $(Confid\ A\ B\ M \in chan\ s \vee Secure\ A\ B\ M \in chan\ s) \longrightarrow$   
 $M \in dy\text{-fake-msg } (bad\ s)\ (ik\ s)\ (chan\ s)$   
 $\}$

**lemmas**  $l2\text{-inv}5I = l2\text{-inv}5\text{-def } [THEN\ setc\text{-def-to-intro, rule-format}]$

**lemmas**  $l2\text{-inv}5E [elim] = l2\text{-inv}5\text{-def } [THEN\ setc\text{-def-to-elim, rule-format}]$

**lemmas**  $l2\text{-inv}5D = l2\text{-inv}5\text{-def } [THEN\ setc\text{-def-to-dest, rule-format, rotated 1, simplified}]$

**lemma**  $l2\text{-inv}5\text{-init } [iff]:$

$init\ l2 \subseteq l2\text{-inv}5$

**by**  $(auto\ simp\ add: l2\text{-def } l2\text{-init-def } l2\text{-inv}5\text{-def})$

**lemma**  $l2\text{-inv}5\text{-trans } [iff]:$

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

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

### 17.2.6 inv6

If an initiator  $Ra$  knows a session key  $K$ , then the attacker knows  $Aenc\ K\ (epubKF\ (Ra\ \$\ kE))$ .

#### definition

$l2\text{-inv6} :: l2\text{-state set}$

#### where

$l2\text{-inv6} \equiv \{s. \forall Ra\ K.$   
 $\text{role } (guessed\text{-runs } Ra) = \text{Init} \longrightarrow$   
 $\text{in\_progress } (\text{progress } s\ Ra)\ xsk \longrightarrow$   
 $\text{guessed\_frame } Ra\ xsk = \text{Some } K \longrightarrow$   
 $Aenc\ K\ (epubKF\ (Ra\$kE)) \in \text{extr } (\text{bad } s)\ (ik\ s)\ (\text{chan } s)$   
 $\}$

**lemmas** l2-inv6I = l2-inv6-def [THEN setc-def-to-intro, rule-format]

**lemmas** l2-inv6E [elim] = l2-inv6-def [THEN setc-def-to-elim, rule-format]

**lemmas** l2-inv6D = l2-inv6-def [THEN setc-def-to-dest, rule-format, rotated 1, simplified]

**lemma** l2-inv6-init [iff]:

$\text{init } l2 \subseteq l2\text{-inv6}$

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

**lemma** l2-inv6-trans [iff]:

$\{l2\text{-inv6}\} \text{ trans } l2 \{> l2\text{-inv6}\}$

**apply** (auto simp add: PO-hoare-defs l2-nostep-defs intro!: l2-inv6I)

**apply** (auto simp add: l2-defs dy-fake-chan-def dest: l2-inv6D)

**done**

**lemma** PO-l2-inv6 [iff]: reach l2  $\subseteq$  l2-inv6

**by** (rule inv-rule-basic) (auto)

### 17.2.7 inv7

Form of the messages in  $\text{extr } (\text{bad } s)\ (ik\ s)\ (\text{chan } s) = \text{synth } (\text{analz } \text{generators})$ .

#### abbreviation

$\text{generators} \equiv \text{range } epubK \cup$   
 $\{Aenc\ (NonceF\ (Rb\ \$\ sk))\ (epubKF\ (Ra\$kE)) \mid Ra\ Rb. \exists A\ B.$   
 $\text{guessed\_runs } Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \wedge$   
 $\text{guessed\_runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \wedge$   
 $\text{guessed\_frame } Rb\ xpkE = \text{Some } (epubKF\ (Ra\$kE))\} \cup$   
 $\{NonceF\ (R\ \$\ sk) \mid R. R \neq \text{test} \wedge R \notin \text{partners}\}$

**lemma** analz-generators: analz generators = generators

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

**definition**

$l2\text{-inv}7 :: l2\text{-state set}$

**where**

$l2\text{-inv}7 \equiv \{s.$   
 $\text{extr } (bad\ s) (ik\ s) (chan\ s) \subseteq$   
 $\text{synth } (analz\ (generators))$   
 $\}$

**lemmas**  $l2\text{-inv}7I = l2\text{-inv}7\text{-def } [THEN\ setc\text{-def-to-intro, rule-format}]$

**lemmas**  $l2\text{-inv}7E [elim] = l2\text{-inv}7\text{-def } [THEN\ setc\text{-def-to-elim, rule-format}]$

**lemmas**  $l2\text{-inv}7D = l2\text{-inv}7\text{-def } [THEN\ setc\text{-def-to-dest, rule-format, rotated 1, simplified}]$

**lemma**  $l2\text{-inv}7\text{-init } [iff]:$

$init\ l2 \subseteq l2\text{-inv}7$

by (auto simp add:  $l2\text{-def } l2\text{-init-def } l2\text{-inv}7\text{-def}$ )

**lemma**  $l2\text{-inv}7\text{-step1}:$

$\{l2\text{-inv}7\} l2\text{-step1 } Ra\ A\ B \{> l2\text{-inv}7\}$

apply (auto simp add:  $PO\text{-hoare-defs } l2\text{-defs intro!:$   $l2\text{-inv}7I$ )

apply (auto dest:  $l2\text{-inv}7D [THEN [2] rev-subsetD]$ )+

done

**lemma**  $l2\text{-inv}7\text{-step2}:$

$\{l2\text{-inv}1 \cap l2\text{-inv}2 \cap l2\text{-inv}4 \cap l2\text{-inv}7\} l2\text{-step2 } Rb\ A\ B\ KE \{> l2\text{-inv}7\}$

**proof** (auto simp add:  $PO\text{-hoare-defs } l2\text{-nostep-defs intro!:$   $l2\text{-inv}7I$ )

fix  $s' s :: l2\text{-state}$

fix  $x$

assume  $Hx: x \in \text{extr } (bad\ s') (ik\ s') (chan\ s')$

assume  $Hi: s \in l2\text{-inv}7$

assume  $Hi': s \in l2\text{-inv}1$

assume  $Hi'': s \in l2\text{-inv}2$

assume  $Hi''': s \in l2\text{-inv}4$

assume  $Hs: (s, s') \in l2\text{-step2 } Rb\ A\ B\ KE$

from  $Hx\ Hi\ Hs$  show  $x \in \text{synth } (analz\ (generators))$

proof (auto simp add:  $l2\text{-defs dest: } l2\text{-inv}7D [THEN [2] rev-subsetD]$ )

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

assume  $Hc: Auth\ A\ B \langle Number\ 0, KE \rangle \in chan\ s$

assume  $HRb: guessed\text{-runs } Rb = (\text{role} = Resp, \text{owner} = B, \text{partner} = A)$   
 $gessed\text{-frame } Rb\ xpkE = Some\ KE$

assume  $Hcs: can\text{-signal } s\ A\ B$

from  $Hcs\ Hi'$  have  $A \notin bad\ s \wedge B \notin bad\ s$

by auto

with  $Hc\ Hi''$  obtain  $Ra$  where  $KE = epubKF\ (Ra\ \$kE)$

and  $gessed\text{-runs } Ra = (\text{role} = Init, \text{owner} = A, \text{partner} = B)$

by (auto dest:  $l2\text{-inv}2D$ )

with  $HRb$  show  $Aenc\ (NonceF\ (Rb\ \$sk))\ KE \in \text{synth } (analz\ generators)$

by blast

next



second case:  $\neg \text{can-signal } s \ A \ B$ . We show that  $Rb$  is not test and not a partner: -  $Rb$  is not test because in that case test is not finished and  $A, B$  are the test agents, thus  $\text{can-signal } s \ A \ B - Rb$  is not a partner for the same reason therefore the message added to the channel can be constructed from  $\{\text{NonceF } (R \ \$ \ sk) \mid R. R \neq \text{test} \wedge R \notin \text{partners}\}$

```

assume  $Hc: \text{Auth } A \ B \langle \text{Number } 0, KE \rangle \in \text{chan } s$ 
assume  $Hcs: \neg \text{can-signal } s \ A \ B$ 
assume  $HRb: Rb \notin \text{dom } (\text{progress } s)$ 
            $\text{guessed-runs } Rb = (\text{role} = \text{Resp}, \text{owner} = B, \text{partner} = A)$ 
from  $Hcs \ HRb$  have  $Rb \neq \text{test}$ 
  by (auto simp add: can-signal-def domIff)
moreover from  $HRb \ Hi''' \ Hcs$  have  $Rb \notin \text{partners}$ 
  by (clarify, auto simp add: partners-def partner-runs-def can-signal-def matching-def domIff)
moreover from  $Hc \ Hi$  have  $KE \in \text{synth } (\text{analz } (\text{generators}))$ 
  by auto
ultimately show  $Aenc \ (\text{NonceF } (Rb \ \$ \ sk)) \ KE \in \text{synth } (\text{analz } (\text{generators}))$ 
  by blast
qed
qed

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

done

**lemma** *l2-inv7-skr*:

$\{l2\text{-inv}7 \cap l2\text{-inv}6\}$  *l2-skr*  $R K \{> l2\text{-inv}7\}$

**proof** (*auto simp add: PO-hoare-defs l2-defs extr-insert-IK-eq intro!: l2-inv7I,*  
*auto elim: l2-inv7D [THEN subsetD]*)

**fix**  $s$

**assume** *HRtest*:  $R \neq \text{test } R \notin \text{partners}$

**assume** *Hi*:  $s \in l2\text{-inv}7$

**assume** *Hi'*:  $s \in l2\text{-inv}6$

**assume** *HRsk*: *in-progress* (*progress*  $s R$ ) *xsk* *guessed-frame*  $R \text{ } xsk = \text{Some } K$

**show**  $K \in \text{synth } (\text{analz } \text{generators})$

**proof** (*cases role (guessed-runs R)*)

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

**assume** *HRI*: *role* (*guessed-runs*  $R$ ) = *Init*

**with** *HRsk Hi Hi'* **have**  $Aenc K (\text{epubKF } (R\$kE)) \in \text{synth } (\text{analz } \text{generators})$

**by** (*auto dest!: l2-inv7D*)

**then have**  $K \in \text{synth } (\text{analz } \text{generators}) \vee Aenc K (\text{epubKF } (R\$kE)) \in \text{generators}$

**by** (*rule synth.cases, simp-all, simp add: analz-generators*)

**with** *HRsk* **show** *?thesis*

**proof** *auto*

**fix**  $Rb A B$

**assume** *HR*: *guessed-runs*  $R = (\text{role} = \text{Init}, \text{owner} = A, \text{partner} = B)$   
*guessed-frame*  $R \text{ } xsk = \text{Some } (\text{NonceF } (Rb \$ sk))$

**assume** *HRb*: *guessed-runs*  $Rb = (\text{role} = \text{Resp}, \text{owner} = B, \text{partner} = A)$   
*guessed-frame*  $Rb \text{ } xpkE = \text{Some } (\text{epubKF } (R \$ kE))$

**from** *HR HRb* **have** *partner-runs*  $Rb R$

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

**with** *HRtest* **have**  $Rb \neq \text{test} \wedge Rb \notin \text{partners}$

**by** (*auto dest: partner-test, simp add: partners-def*)

**then show**  $\text{NonceF } (Rb \$ sk) \in \text{analz } \text{generators}$

**by** *blast*

**qed**

**next**

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

**assume** *HRI*: *role* (*guessed-runs*  $R$ ) = *Resp*

**with** *HRsk HRtest* **show** *?thesis*

**by** *auto*

**qed**

**qed**

**lemmas** *l2-inv7-trans-aux* =

*l2-inv7-step1 l2-inv7-step2 l2-inv7-step3*

*l2-inv7-dy-fake-msg l2-inv7-dy-fake-chan*

*l2-inv7-lkr-others l2-inv7-lkr-after l2-inv7-skr*

**lemma** *l2-inv7-trans* [*iff*]:

$\{l2\text{-inv}7 \cap l2\text{-inv}1 \cap l2\text{-inv}2 \cap l2\text{-inv}4 \cap l2\text{-inv}5 \cap l2\text{-inv}6\}$  *trans*  $l2 \{> l2\text{-inv}7\}$   
**by** (*auto simp add: l2-nostep-defs intro:l2-inv7-trans-aux*)

**lemma** *PO-l2-inv7 [iff]: reach*  $l2 \subseteq l2\text{-inv}7$   
**by** (*rule-tac J=l2-inv1 \cap l2-inv2 \cap l2-inv4 \cap l2-inv5 \cap l2-inv6 in inv-rule-incr*) (*auto*)

**lemma** *l2-inv7-aux:*

$NonceF (R\$sk) \in \text{analz } (ik\ s) \implies s \in l2\text{-inv}7 \implies R \neq \text{test} \wedge R \notin \text{partners}$

**proof** –

**assume**  $H:s \in l2\text{-inv}7$  **and**  $H':NonceF (R\$sk) \in \text{analz } (ik\ s)$

**then have**  $H'':NonceF (R\$sk) \in \text{analz } (\text{extr } (\text{bad } s) (ik\ s) (\text{chan } s))$

**by** (*auto elim: analz-monotone*)

**from**  $H$  **have**  $\text{analz } (\text{extr } (\text{bad } s) (ik\ s) (\text{chan } s)) \subseteq \text{analz } (\text{synth } (\text{analz } \text{generators}))$

**by** (*blast dest: analz-mono intro: l2-inv7D*)

**with**  $H''$  **have**  $NonceF (R\$sk) \in \text{analz } \text{generators}$

**by** *auto*

**then have**  $NonceF (R\$sk) \in \text{generators}$

**by** (*simp add: analz-generators*)

**then show** *?thesis*

**by** *auto*

**qed**

### 17.2.8 inv8

Form of the secrets = nonces generated by test or partners

**definition**

$l2\text{-inv}8 :: l2\text{-state set}$

**where**

$l2\text{-inv}8 \equiv \{s.$

$\text{secret } s \subseteq \{NonceF (R\$sk) \mid R. R = \text{test} \vee R \in \text{partners}\}$

$\}$

**lemmas**  $l2\text{-inv}8I = l2\text{-inv}8\text{-def}$  [*THEN setc-def-to-intro, rule-format*]

**lemmas**  $l2\text{-inv}8E$  [*elim*] =  $l2\text{-inv}8\text{-def}$  [*THEN setc-def-to-elim, rule-format*]

**lemmas**  $l2\text{-inv}8D = l2\text{-inv}8\text{-def}$  [*THEN setc-def-to-dest, rule-format, rotated 1, simplified*]

**lemma**  $l2\text{-inv}8\text{-init}$  [*iff*]:

$\text{init } l2 \subseteq l2\text{-inv}8$

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

**lemma**  $l2\text{-inv}8\text{-trans}$  [*iff*]:

$\{l2\text{-inv}8 \cap l2\text{-inv}1 \cap l2\text{-inv}3\}$  *trans*  $l2 \{> l2\text{-inv}8\}$

**supply** [*simproc del: defined-all*]

**apply** (*auto simp add: PO-hoare-defs l2-nostep-defs intro!: l2-inv8I*)

**apply** (*auto simp add: l2-defs dy-fake-chan-def*)

**apply** (*auto simp add: partners-def partner-runs-def matching-def dest!: l2-inv3D*)

**apply** (*simp-all add: can-signal-def*)

**done**

**lemma** *PO-l2-inv8 [iff]: reach*  $l2 \subseteq l2\text{-inv}8$

**by** (*rule-tac J=l2-inv1 \cap l2-inv3 in inv-rule-incr*) (*auto*)

### 17.3 Refinement

mediator function

**definition**

$med12s :: l2\text{-obs} \Rightarrow l1\text{-obs}$

**where**

$med12s\ t \equiv (\!$   
 $ik = ik\ t,$   
 $secret = secret\ t,$   
 $progress = progress\ t,$   
 $signals = signals\ t$   
 $\!)$

relation between states

**definition**

$R12s :: (l1\text{-state} * l2\text{-state})\ set$

**where**

$R12s \equiv \{(s, s') .$   
 $s = med12s\ s'$   
 $\}$

**lemmas**  $R12s\text{-defs} = R12s\text{-def}\ med12s\text{-def}$

**lemma**  $can\text{-signal}\text{-}R12\ [simp]:$

$(s1, s2) \in R12s \Longrightarrow$   
 $can\text{-signal}\ s1\ A\ B \longleftrightarrow can\text{-signal}\ s2\ A\ B$

**by**  $(auto\ simp\ add: can\text{-signal}\text{-def}\ R12s\text{-defs})$

protocol events

**lemma**  $l2\text{-step1}\text{-refines}\text{-step1}:$

$\{R12s\}\ l1\text{-step1}\ Ra\ A\ B, l2\text{-step1}\ Ra\ A\ B \{>R12s\}$

**by**  $(auto\ simp\ add: PO\text{-rhoare}\text{-defs}\ R12s\text{-defs}\ l1\text{-step1}\text{-def}\ l2\text{-step1}\text{-def})$

**lemma**  $l2\text{-step2}\text{-refines}\text{-step2}:$

$\{R12s \cap UNIV \times (l2\text{-inv1} \cap l2\text{-inv2} \cap l2\text{-inv7})\}$   
 $l1\text{-step2}\ Rb\ A\ B\ KE, l2\text{-step2}\ Rb\ A\ B\ KE$   
 $\{>R12s\}$

**apply**  $(auto\ simp\ add: PO\text{-rhoare}\text{-defs}\ R12s\text{-defs}\ l1\text{-step2}\text{-def},\ simp\text{-all}\ add: l2\text{-step2}\text{-def})$

**apply**  $(auto\ dest!: l2\text{-inv7}\text{-aux}\ l2\text{-inv2D})$

**done**

auxiliary lemma needed to prove that the nonce received by the test in step 3 comes from a partner

**lemma**  $l2\text{-step3}\text{-partners}:$

$guessed\text{-runs}\ test = (\!role = Init, owner = A, partner = B) \Longrightarrow$   
 $guessed\text{-frame}\ test\ xsk = Some\ (NonceF\ (Rb\$sk)) \Longrightarrow$   
 $guessed\text{-runs}\ Rb = (\!role = Resp, owner = B, partner = A) \Longrightarrow$   
 $guessed\text{-frame}\ Rb\ xpkE = Some\ (epubKF\ (test\ \$\ kE)) \Longrightarrow$   
 $Rb \in partners$

**by**  $(auto\ simp\ add: partners\text{-def}\ partner\text{-runs}\text{-def}\ matching\text{-def})$

**lemma** *l2-step3-refines-step3*:  
 $\{R12s \cap UNIV \times (l2-inv1 \cap l2-inv3 \cap l2-inv7)\}$   
 $l1-step3\ Ra\ A\ B\ K, l2-step3\ Ra\ A\ B\ K$   
 $\{>R12s\}$   
**supply**  $[[simproc\ del:\ defined-all]]$   
**apply**  $(auto\ simp\ add:\ PO-rhoare-defs\ R12s-defs\ l1-step3-def, simp-all\ add:\ l2-step3-def)$   
**apply**  $(auto\ dest!:\ l2-inv3D\ l2-inv7-aux\ intro:l2-step3-partners)$   
**apply**  $(auto\ simp\ add:\ can-signal-def)$   
**done**

attacker events

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

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

compromising events

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

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

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

refinement proof

**lemmas** *l2-trans-refines-l1-trans =*  
*l2-dy-fake-msg-refines-learn\ l2-dy-fake-chan-refines-skip*

*l2-lkr-others-refines-skip l2-lkr-after-refines-skip l2-skr-refines-learn  
l2-step1-refines-step1 l2-step2-refines-step2 l2-step3-refines-step3*

**lemma** *l2-refines-init-l1* [iff]:  
*init l2*  $\subseteq$  *R12s* “ (*init l1*)  
**by** (*auto simp add: R12s-defs l1-defs l2-loc-defs*)

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

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

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

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

## 17.4 Derived invariants

We want to prove *l2-secrecy*: *dy-fake-msg (bad s) (ik s) (chan s)  $\cap$  secret s = {}* but by refinement we only get *l2-partial-secrecy*: *synth (analz (ik s))  $\cap$  secret s = {}* This is fine, since a message in *dy-fake-msg (bad s) (ik s) (chan s)* could be added to *ik s*, and *l2-partial-secrecy* would still hold for this new state.

### definition

*l2-partial-secrecy* :: (*'a l2-state-scheme*) *set*

### where

*l2-partial-secrecy*  $\equiv \{s. \text{synth } (analz\ (ik\ s)) \cap \text{secret } s = \{\}\}$

**lemma** *l2-obs-partial-secrecy* [iff]: *oreach l2*  $\subseteq$  *l2-partial-secrecy*

**apply** (*rule external-invariant-translation*

[*OF l1-obs-secrecy - l2-implements-l1*])

**apply** (*auto simp add: med12s-def s0-secrecy-def l2-partial-secrecy-def*)

**done**

**lemma** *l2-oreach-dy-fake-msg*:

*s*  $\in$  *oreach l2*  $\implies$  *x*  $\in$  *dy-fake-msg (bad s) (ik s) (chan s)*  $\implies$  *s* (*ik := insert x (ik s)*)  $\in$  *oreach l2*

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

**apply** *blast*

**done**

**definition**

$l2\text{-secrecy} :: ('a\ l2\text{-state-scheme})\ set$

**where**

$l2\text{-secrecy} \equiv \{s.\ dy\text{-fake-msg}\ (bad\ s)\ (ik\ s)\ (chan\ s) \cap\ secret\ s = \{\}\}$

**lemma**  $l2\text{-obs-secrecy}$  [iff]:  $oreach\ l2 \subseteq l2\text{-secrecy}$

**apply** (auto simp add: l2-secrecy-def)

**apply** (drule l2-oreach-dy-fake-msg, simp-all)

**apply** (drule l2-obs-partial-secrecy [THEN [2] rev-subsetD], simp add: l2-partial-secrecy-def)

**apply** blast

**done**

**lemma**  $l2\text{-secrecy}$  [iff]:  $reach\ l2 \subseteq l2\text{-secrecy}$

**by** (rule external-to-internal-invariant [OF l2-obs-secrecy], auto)

**abbreviation**  $l2\text{-iagreement} \equiv l1\text{-iagreement}$

**lemma**  $l2\text{-obs-iagreement}$  [iff]:  $oreach\ l2 \subseteq l2\text{-iagreement}$

**apply** (rule external-invariant-translation

[OF l1-obs-iagreement - l2-implements-l1])

**apply** (auto simp add: med12s-def l1-iagreement-def)

**done**

**lemma**  $l2\text{-iagreement}$  [iff]:  $reach\ l2 \subseteq l2\text{-iagreement}$

**by** (rule external-to-internal-invariant [OF l2-obs-iagreement], auto)

**end**

## 18 Key Transport Protocol with PFS (L3 locale)

```
theory pfsvl3
imports pfsvl2 Implem-lemmas
begin
```

### 18.1 State and Events

Level 3 state

(The types have to be defined outside the locale.)

```
record l3-state = l1-state +
  bad :: agent set
```

```
type-synonym l3-obs = l3-state
```

```
type-synonym
  l3-pred = l3-state set
```

```
type-synonym
  l3-trans = (l3-state × l3-state) set
```

attacker event

```
definition
  l3-dy :: msg ⇒ l3-trans
where
  l3-dy ≡ ik-dy
```

compromise events

```
definition
  l3-lkr-others :: agent ⇒ l3-trans
where
  l3-lkr-others A ≡ {(s,s').
    — guards
    A ≠ test-owner ∧
    A ≠ test-partner ∧
    — actions
    s' = s(bad := {A} ∪ bad s,
           ik := keys-of A ∪ ik s)
  }
```

```
definition
  l3-lkr-actor :: agent ⇒ l3-trans
where
  l3-lkr-actor A ≡ {(s,s').
    — guards
    A = test-owner ∧
    A ≠ test-partner ∧
    — actions
    s' = s(bad := {A} ∪ bad s,
           ik := keys-of A ∪ ik s)
  }
```



**definition**

$$l3\text{-lkr}\text{-after} :: \text{agent} \Rightarrow l3\text{-trans}$$
**where**

$$l3\text{-lkr}\text{-after} A \equiv \{(s, s') .$$

- guards
- $test\text{-ended } s \wedge$
- actions
- $s' = s(\text{bad} := \{A\} \cup \text{bad } s,$
- $ik := \text{keys-of } A \cup ik \ s)$

$$\}$$
**definition**

$$l3\text{-skr} :: \text{rid-t} \Rightarrow \text{msg} \Rightarrow l3\text{-trans}$$
**where**

$$l3\text{-skr} R K \equiv \{(s, s') .$$

- guards
- $R \neq test \wedge R \notin \text{partners} \wedge$
- $in\text{-progress } (\text{progress } s \ R) \ xsk \wedge$
- $guessed\text{-frame } R \ xsk = \text{Some } K \wedge$
- actions
- $s' = s(ik := \{K\} \cup ik \ s)$

$$\}$$

New locale for the level 3 protocol

This locale does not add new assumptions, it is only used to separate the level 3 protocol from the implementation locale.

**locale**  $pfslvl3 = \text{valid-implem}$

**begin**

protocol events

**definition**

$$l3\text{-step1} :: \text{rid-t} \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow l3\text{-trans}$$
**where**

$$l3\text{-step1} Ra A B \equiv \{(s, s') .$$

- guards:
- $Ra \notin \text{dom } (\text{progress } s) \wedge$
- $guessed\text{-runs } Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \wedge$
- actions:
- $s' = s($
- $\text{progress} := (\text{progress } s)(Ra \mapsto \{xpkE, xskE\}),$
- $ik := \{\text{implAuth } A \ B \ \langle \text{Number } 0, \text{epubKF } (Ra\$kE)\rangle\} \cup (ik \ s)$
- $)$

$$\}$$
**definition**

$$l3\text{-step2} :: \text{rid-t} \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow \text{msg} \Rightarrow l3\text{-trans}$$
**where**

$$l3\text{-step2} Rb A B KE \equiv \{(s, s') .$$

- guards:
- $guessed\text{-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \wedge$
- $Rb \notin \text{dom } (\text{progress } s) \wedge$

```

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

```

**definition**

*l3-step3* :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *msg* ⇒ *l3-trans*

**where**

*l3-step3* Ra A B K ≡ {(s, s').

— guards:

*guessed-runs* Ra = (|role=Init, owner=A, partner=B|) ∧

*progress* s Ra = Some {xpkE, xskE} ∧

*guessed-frame* Ra xsk = Some K ∧

*implAuth* B A (Aenc K (epubKF (Ra\$skE))) ∈ ik s ∧

— actions:

s' = s(| *progress* := (progress s)(Ra ↦ {xpkE, xskE, xsk}),

*signals* := if can-signal s A B then

addSignal (signals s) (Commit A B ⟨epubKF (Ra\$skE), K⟩)

else

*signals* s,

*secret* := {x. x = K ∧ Ra = test} ∪ secret s

|)

}

specification

initial compromise

**definition**

*ik-init* :: *msg set*

**where**

*ik-init* ≡ {priK C | C. C ∈ bad-init} ∪ {pubK A | A. True} ∪  
 {shrK A B | A B. A ∈ bad-init ∨ B ∈ bad-init} ∪ Tags

lemmas about *ik-init*

**lemma** *parts-ik-init* [simp]: parts *ik-init* = *ik-init*

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

**lemma** *analz-ik-init* [simp]: analz *ik-init* = *ik-init*

**by** (auto dest: analz-into-parts)

**lemma** *abs-ik-init* [iff]: abs *ik-init* = {}

**apply** (auto elim!: absE)

**apply** (auto simp add: ik-init-def)



**lemma** *l3-obs-id* [*simp*]: *obs l3 = id*  
**by** (*simp add: l3-def*)

## 18.2 Invariants

### 18.2.1 inv1: No long-term keys as message parts

**definition**

*l3-inv1* :: *l3-state set*

**where**

*l3-inv1*  $\equiv$  {*s*.  
*parts (ik s)  $\cap$  range LtK  $\subseteq$  ik s*  
}

**lemmas** *l3-inv1I* = *l3-inv1-def* [*THEN setc-def-to-intro, rule-format*]

**lemmas** *l3-inv1E* [*elim*] = *l3-inv1-def* [*THEN setc-def-to-elim, rule-format*]

**lemmas** *l3-inv1D* = *l3-inv1-def* [*THEN setc-def-to-dest, rule-format*]

**lemma** *l3-inv1D'* [*dest*]:  $\llbracket \text{LtK } K \in \text{parts } (ik\ s); s \in l3\text{-inv1} \rrbracket \implies \text{LtK } K \in ik\ s$   
**by** (*auto simp add: l3-inv1-def*)

**lemma** *l3-inv1-init* [*iff*]:

*init l3  $\subseteq$  l3-inv1*

**by** (*auto simp add: l3-def l3-init-def intro!:l3-inv1I*)

**lemma** *l3-inv1-trans* [*iff*]:

{*l3-inv1*} *trans l3*  $\{>$  *l3-inv1*}

**apply** (*auto simp add: PO-hoare-defs l3-nostep-defs intro!: l3-inv1I*)

**apply** (*auto simp add: l3-defs dy-fake-msg-def dy-fake-chan-def*

*parts-insert* [**where** *H=ik -*] *parts-insert* [**where** *H=insert - (ik -)*]  
*dest!*: *Fake-parts-insert*)

**apply** (*auto dest:analz-into-parts*)

**done**

**lemma** *PO-l3-inv1* [*iff*]:

*reach l3  $\subseteq$  l3-inv1*

**by** (*rule inv-rule-basic*) (*auto*)

### 18.2.2 inv2: *l3-state.bad s* indeed contains "bad" keys

**definition**

*l3-inv2* :: *l3-state set*

**where**

*l3-inv2*  $\equiv$  {*s*.  
*Keys-bad (ik s) (bad s)*  
}

**lemmas** *l3-inv2I* = *l3-inv2-def* [*THEN setc-def-to-intro, rule-format*]

**lemmas** *l3-inv2E* [*elim*] = *l3-inv2-def* [*THEN setc-def-to-elim, rule-format*]

**lemmas** *l3-inv2D* = *l3-inv2-def* [*THEN setc-def-to-dest, rule-format*]

**lemma** *l3-inv2-init* [*simp,intro!*]:  
*init l3*  $\subseteq$  *l3-inv2*  
**by** (*auto simp add: l3-def l3-init-defs intro!:l3-inv2I Keys-badI*)

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

4 subgoals: *dy*, *lkr\**, *skr*

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

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

### 18.2.3 inv3

If a message can be analyzed from the intruder knowledge then it can be derived (using *synth/analz*) from the sets of implementation, non-implementation, and long-term key messages and the tags. That is, intermediate messages are not needed.

#### definition

*l3-inv3*  $::$  *l3-state set*

#### where

*l3-inv3*  $\equiv$   $\{s.$   
 $\text{analz } (ik\ s) \subseteq$   
 $\text{synth } (\text{analz } ((ik\ s \cap \text{payload}) \cup ((ik\ s) \cap \text{valid}) \cup (ik\ s \cap \text{range } LtK) \cup \text{Tags}))$   
 $\}$

**lemmas** *l3-inv3I* = *l3-inv3-def* [*THEN setc-def-to-intro, rule-format*]

**lemmas** *l3-inv3E* = *l3-inv3-def* [*THEN setc-def-to-elim, rule-format*]

**lemmas** *l3-inv3D* = *l3-inv3-def* [*THEN setc-def-to-dest, rule-format*]

**lemma** *l3-inv3-init* [*iff*]:  
*init l3*  $\subseteq$  *l3-inv3*  
**apply** (*auto simp add: l3-def l3-init-def intro!: l3-inv3I*)  
**apply** (*auto simp add: ik-init-def intro!: synth-increasing [THEN [2] rev-subsetD]*)  
**done**

**declare** *domIff* [*iff del*]

Most of the cases in this proof are simple and very similar. The proof could probably be shortened.

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

```

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

lemma PO-l3-inv3 [iff]: reach l3  $\subseteq$  l3-inv3
by (rule inv-rule-basic) (auto)

```

#### 18.2.4 inv4: the intruder knows the tags

**definition**

*l3-inv4* :: *l3-state set*

**where**

*l3-inv4*  $\equiv$  {*s*.  
 Tags  $\subseteq$  *ik s*  
 }

**lemmas**  $l3\text{-inv}4I = l3\text{-inv}4\text{-def}$  [THEN setc-def-to-intro, rule-format]  
**lemmas**  $l3\text{-inv}4E$  [elim] =  $l3\text{-inv}4\text{-def}$  [THEN setc-def-to-elim, rule-format]  
**lemmas**  $l3\text{-inv}4D = l3\text{-inv}4\text{-def}$  [THEN setc-def-to-dest, rule-format]

**lemma**  $l3\text{-inv}4\text{-init}$  [simp,intro!]:  
 $init\ l3 \subseteq l3\text{-inv}4$   
**by** (auto simp add:  $l3\text{-def}$   $l3\text{-init-def}$   $ik\text{-init-def}$  intro!:  $l3\text{-inv}4I$ )

**lemma**  $l3\text{-inv}4\text{-trans}$  [simp,intro!]:  
 $\{l3\text{-inv}4\}$  trans  $l3$   $\{> l3\text{-inv}4\}$   
**apply** (auto simp add: PO-hoare-defs  $l3\text{-nostep-defs}$  intro!:  $l3\text{-inv}4I$ )  
**apply** (auto simp add:  $l3\text{-defs}$   $dy\text{-fake-chan-def}$   $dy\text{-fake-msg-def}$ )  
**done**

**lemma**  $PO\text{-}l3\text{-inv}4$  [simp,intro!]:  $reach\ l3 \subseteq l3\text{-inv}4$   
**by** (rule inv-rule-basic) (auto)

The remaining invariants are derived from the others. They are not protocol dependent provided the previous invariants hold.

### 18.2.5 inv5

The messages that the L3 DY intruder can derive from the intruder knowledge (using *synth/analz*), are either implementations or intermediate messages or can also be derived by the L2 intruder from the set  $extr\ (l3\text{-state}\ .bad\ s)\ (ik\ s \cap payload)\ (local.\ abs\ (ik\ s))$ , that is, given the non-implementation messages and the abstractions of (implementation) messages in the intruder knowledge.

#### definition

$l3\text{-inv}5 :: l3\text{-state}\ set$

#### where

$l3\text{-inv}5 \equiv \{s.$   
 $synth\ (analz\ (ik\ s)) \subseteq$   
 $dy\text{-fake-msg}\ (bad\ s)\ (ik\ s \cap payload)\ (abs\ (ik\ s)) \cup \text{-payload}$   
 $\}$

**lemmas**  $l3\text{-inv}5I = l3\text{-inv}5\text{-def}$  [THEN setc-def-to-intro, rule-format]  
**lemmas**  $l3\text{-inv}5E = l3\text{-inv}5\text{-def}$  [THEN setc-def-to-elim, rule-format]  
**lemmas**  $l3\text{-inv}5D = l3\text{-inv}5\text{-def}$  [THEN setc-def-to-dest, rule-format]

**lemma**  $l3\text{-inv}5\text{-derived}$ :  $l3\text{-inv}2 \cap l3\text{-inv}3 \subseteq l3\text{-inv}5$   
**by** (auto simp add: abs-validSet  $dy\text{-fake-msg-def}$  intro!:  $l3\text{-inv}5I$   
 $dest!$ :  $l3\text{-inv}3D$  [THEN synth-mono, THEN [2] rev-subsetD]  
 $dest!$ :  $synth\text{-analz-NI-I-K-synth-analz-NI-E}$  [THEN [2] rev-subsetD])

**lemma**  $PO\text{-}l3\text{-inv}5$  [simp,intro!]:  $reach\ l3 \subseteq l3\text{-inv}5$   
**using**  $l3\text{-inv}5\text{-derived}$   $PO\text{-}l3\text{-inv}2$   $PO\text{-}l3\text{-inv}3$   
**by** blast

### 18.2.6 inv6

If the level 3 intruder can deduce a message implementing an insecure channel message, then:

- either the message is already in the intruder knowledge;
- or the message is constructed, and the payload can also be deduced by the intruder.

**definition**

*l3-inv6* :: *l3-state set*

**where**

$l3\text{-inv6} \equiv \{s. \forall A B M. \\ (\text{implInsec } A B M \in \text{synth } (\text{analz } (ik\ s)) \wedge M \in \text{payload}) \longrightarrow \\ (\text{implInsec } A B M \in ik\ s \vee M \in \text{synth } (\text{analz } (ik\ s)))\}$

**lemmas** *l3-inv6I* = *l3-inv6-def* [*THEN setc-def-to-intro, rule-format*]

**lemmas** *l3-inv6E* = *l3-inv6-def* [*THEN setc-def-to-elim, rule-format*]

**lemmas** *l3-inv6D* = *l3-inv6-def* [*THEN setc-def-to-dest, rule-format*]

**lemma** *l3-inv6-derived* [*simp,intro!*]:

$l3\text{-inv3} \cap l3\text{-inv4} \subseteq l3\text{-inv6}$

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

1 subgoal

**apply** (*drule synth-mono, simp, drule subsetD, assumption*)

**apply** (*auto dest!*: *implInsec-synth-analz* [*rotated 1, where H=- ∪ -*])

**apply** (*auto dest!*: *synth-analz-monotone* [*of - - ∪ - ik -*])

**done**

**lemma** *PO-l3-inv6* [*simp,intro!*]:  $reach\ l3 \subseteq l3\text{-inv6}$

**using** *l3-inv6-derived PO-l3-inv3 PO-l3-inv4*

**by** (*blast*)

### 18.2.7 inv7

If the level 3 intruder can deduce a message implementing a confidential channel message, then:

- either the message is already in the intruder knowledge;
- or the message is constructed, and the payload can also be deduced by the intruder.

**definition**

*l3-inv7* :: *l3-state set*

**where**

$l3\text{-inv7} \equiv \{s. \forall A B M. \\ (\text{implConfid } A B M \in \text{synth } (\text{analz } (ik\ s)) \wedge M \in \text{payload}) \longrightarrow \\ (\text{implConfid } A B M \in ik\ s \vee M \in \text{synth } (\text{analz } (ik\ s)))\}$

**lemmas** *l3-inv7I* = *l3-inv7-def* [*THEN setc-def-to-intro, rule-format*]

**lemmas** *l3-inv7E* = *l3-inv7-def* [*THEN setc-def-to-elim, rule-format*]

**lemmas** *l3-inv7D* = *l3-inv7-def* [*THEN setc-def-to-dest, rule-format*]

**lemma** *l3-inv7-derived* [*simp,intro!*]:



$l3\text{-inv}3 \cap l3\text{-inv}4 \subseteq l3\text{-inv}7$   
**apply** (auto intro!: l3-inv7I dest!: l3-inv3D)

1 subgoal

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

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

### 18.2.8 inv8

If the level 3 intruder can deduce a message implementing an authentic channel message then:

- either the message is already in the intruder knowledge;
- or the message is constructed, and in this case the payload can also be deduced by the intruder, and one of the agents is bad.

#### definition

$l3\text{-inv}8 :: l3\text{-state set}$

**where**

$l3\text{-inv}8 \equiv \{s. \forall A B M. \\
(\text{implAuth } A B M \in \text{synth } (\text{analz } (\text{ik } s)) \wedge M \in \text{payload}) \longrightarrow \\
(\text{implAuth } A B M \in \text{ik } s \vee (M \in \text{synth } (\text{analz } (\text{ik } s)) \wedge (A \in \text{bad } s \vee B \in \text{bad } s)))\}$

**lemmas** l3-inv8I = l3-inv8-def [THEN setc-def-to-intro, rule-format]

**lemmas** l3-inv8E = l3-inv8-def [THEN setc-def-to-elim, rule-format]

**lemmas** l3-inv8D = l3-inv8-def [THEN setc-def-to-dest, rule-format]

**lemma** l3-inv8-derived [iff]:

$l3\text{-inv}2 \cap l3\text{-inv}3 \cap l3\text{-inv}4 \subseteq l3\text{-inv}8$

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

2 subgoals: M is deducible and the agents are bad

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

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

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

### 18.2.9 inv9

If the level 3 intruder can deduce a message implementing a secure channel message then:

- either the message is already in the intruder knowledge;
- or the message is constructed, and in this case the payload can also be deduced by the intruder, and one of the agents is bad.

#### definition

$l3\text{-inv9} :: l3\text{-state set}$

#### where

$l3\text{-inv9} \equiv \{s. \forall A B M.$

$(\text{implSecure } A B M \in \text{synth } (\text{analz } (ik\ s)) \wedge M \in \text{payload}) \longrightarrow$

$(\text{implSecure } A B M \in ik\ s \vee (M \in \text{synth } (\text{analz } (ik\ s)) \wedge (A \in \text{bad } s \vee B \in \text{bad } s)))$

$\}$

**lemmas**  $l3\text{-inv9I} = l3\text{-inv9-def } [THEN\ \text{setc-def-to-intro},\ \text{rule-format}]$

**lemmas**  $l3\text{-inv9E} = l3\text{-inv9-def } [THEN\ \text{setc-def-to-elim},\ \text{rule-format}]$

**lemmas**  $l3\text{-inv9D} = l3\text{-inv9-def } [THEN\ \text{setc-def-to-dest},\ \text{rule-format}]$

**lemma**  $l3\text{-inv9-derived } [iff]:$

$l3\text{-inv2} \cap l3\text{-inv3} \cap l3\text{-inv4} \subseteq l3\text{-inv9}$

**apply** ( $\text{auto intro!}:\ l3\text{-inv9I dest!}:\ l3\text{-inv3D } l3\text{-inv2D}$ )

2 subgoals: M is deducible and the agents are bad

**apply** ( $\text{drule } \text{synth-mono},\ \text{simp},\ \text{drule } \text{subsetD},\ \text{assumption}$ )

**apply** ( $\text{auto dest!}:\ \text{implSecure-synth-analz } [\text{rotated } 1,\ \text{where } H=-\cup-] \text{ elim!}:\ \text{synth-analz-monotone}$ )

**apply** ( $\text{drule } \text{synth-mono},\ \text{simp},\ \text{drule } \text{subsetD},\ \text{assumption}$ )

**apply** ( $\text{auto dest!}:\ \text{implSecure-synth-analz } [\text{rotated } 1,\ \text{where } H=-\cup-]$ )

**done**

**lemma**  $PO\text{-}l3\text{-inv9 } [iff]:\ \text{reach } l3 \subseteq l3\text{-inv9}$

**using**  $l3\text{-inv9-derived}$

$PO\text{-}l3\text{-inv3 } PO\text{-}l3\text{-inv2 } PO\text{-}l3\text{-inv4}$

**by**  $\text{blast}$

## 18.3 Refinement

mediator function

#### definition

$med23s :: l3\text{-obs} \Rightarrow l2\text{-obs}$

#### where

$med23s\ t \equiv ()$

$ik = ik\ t \cap \text{payload},$

$\text{secret} = \text{secret } t,$

$\text{progress} = \text{progress } t,$

$\text{signals} = \text{signals } t,$

$\text{chan} = \text{abs } (ik\ t),$

$\text{bad} = \text{bad } t$

$()$

relation between states

**definition**

$R23s :: (l2\text{-state} * l3\text{-state}) \text{ set}$

**where**

$R23s \equiv \{(s, s').$   
 $s = \text{med}23s\ s'$   
 $\}$

**lemmas**  $R23s\text{-defs} = R23s\text{-def}\ \text{med}23s\text{-def}$

**lemma**  $R23sI$ :

$\llbracket ik\ s = ik\ t \cap \text{payload}; \text{secret}\ s = \text{secret}\ t; \text{progress}\ s = \text{progress}\ t; \text{signals}\ s = \text{signals}\ t;$   
 $\text{chan}\ s = \text{abs}\ (ik\ t); l2\text{-state}.bad\ s = bad\ t \rrbracket$   
 $\implies (s, t) \in R23s$

**by**  $(\text{auto}\ \text{simp}\ \text{add}: R23s\text{-def}\ \text{med}23s\text{-def})$

**lemma**  $R23sD$ :

$(s, t) \in R23s \implies$   
 $ik\ s = ik\ t \cap \text{payload} \wedge \text{secret}\ s = \text{secret}\ t \wedge \text{progress}\ s = \text{progress}\ t \wedge \text{signals}\ s = \text{signals}\ t \wedge$   
 $\text{chan}\ s = \text{abs}\ (ik\ t) \wedge l2\text{-state}.bad\ s = bad\ t$

**by**  $(\text{auto}\ \text{simp}\ \text{add}: R23s\text{-def}\ \text{med}23s\text{-def})$

**lemma**  $R23sE$   $[elim]$ :

$\llbracket (s, t) \in R23s;$   
 $\llbracket ik\ s = ik\ t \cap \text{payload}; \text{secret}\ s = \text{secret}\ t; \text{progress}\ s = \text{progress}\ t; \text{signals}\ s = \text{signals}\ t;$   
 $\text{chan}\ s = \text{abs}\ (ik\ t); l2\text{-state}.bad\ s = bad\ t \rrbracket \implies P \rrbracket$   
 $\implies P$

**by**  $(\text{auto}\ \text{simp}\ \text{add}: R23s\text{-def}\ \text{med}23s\text{-def})$

**lemma**  $can\text{-signal}\text{-}R23$   $[simp]$ :

$(s2, s3) \in R23s \implies$   
 $can\text{-signal}\ s2\ A\ B \longleftrightarrow can\text{-signal}\ s3\ A\ B$

**by**  $(\text{auto}\ \text{simp}\ \text{add}: can\text{-signal}\text{-def})$

### 18.3.1 Protocol events

**lemma**  $l3\text{-step1}\text{-refines}\text{-step1}$ :

$\{R23s\}\ l2\text{-step1}\ Ra\ A\ B, l3\text{-step1}\ Ra\ A\ B\ \{>R23s\}$

**apply**  $(\text{auto}\ \text{simp}\ \text{add}: PO\text{-rhoare}\text{-defs}\ R23s\text{-defs})$

**apply**  $(\text{auto}\ \text{simp}\ \text{add}: l3\text{-defs}\ l2\text{-step1}\text{-def})$

**done**

**lemma**  $l3\text{-step2}\text{-refines}\text{-step2}$ :

$\{R23s\}\ l2\text{-step2}\ Rb\ A\ B\ KE, l3\text{-step2}\ Rb\ A\ B\ KE\ \{>R23s\}$

**apply**  $(\text{auto}\ \text{simp}\ \text{add}: PO\text{-rhoare}\text{-defs}\ R23s\text{-defs}\ l2\text{-step2}\text{-def})$

**apply**  $(\text{auto}\ \text{simp}\ \text{add}: l3\text{-step2}\text{-def})$

**done**

**lemma**  $l3\text{-step3}\text{-refines}\text{-step3}$ :

$\{R23s\}\ l2\text{-step3}\ Ra\ A\ B\ K, l3\text{-step3}\ Ra\ A\ B\ K\ \{>R23s\}$

**apply**  $(\text{auto}\ \text{simp}\ \text{add}: PO\text{-rhoare}\text{-defs}\ R23s\text{-defs}\ l2\text{-step3}\text{-def})$

**apply**  $(\text{auto}\ \text{simp}\ \text{add}: l3\text{-step3}\text{-def})$

done

### 18.3.2 Intruder events

**lemma** *l3-dy-payload-refines-dy-fake-msg*:

$M \in \text{payload} \implies$

$\{R23s \cap UNIV \times l3\text{-inv}5\} \text{ l2-dy-fake-msg } M, \text{ l3-dy } M \{>R23s\}$

**apply** (*auto simp add: PO-rhoare-defs R23s-defs*)

**apply** (*auto simp add: l3-defs l2-dy-fake-msg-def dest: l3-inv5D*)

done

**lemma** *l3-dy-valid-refines-dy-fake-chan*:

$\llbracket M \in \text{valid}; M' \in \text{abs } \{M\} \rrbracket \implies$

$\{R23s \cap UNIV \times (l3\text{-inv}5 \cap l3\text{-inv}6 \cap l3\text{-inv}7 \cap l3\text{-inv}8 \cap l3\text{-inv}9)\}$

$\text{ l2-dy-fake-chan } M', \text{ l3-dy } M$

$\{>R23s\}$

**apply** (*auto simp add: PO-rhoare-defs R23s-defs, simp add: l2-dy-fake-chan-def*)

**apply** (*auto simp add: l3-defs*)

1 subgoal

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

Insec

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

Confid

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

Auth

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

Secure

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

done

**lemma** *l3-dy-valid-refines-dy-fake-chan-Un*:

$M \in \text{valid} \implies$

$\{R23s \cap UNIV \times (l3\text{-inv}5 \cap l3\text{-inv}6 \cap l3\text{-inv}7 \cap l3\text{-inv}8 \cap l3\text{-inv}9)\}$

$\bigcup M'. \text{ l2-dy-fake-chan } M', \text{ l3-dy } M$

$\{>R23s\}$

**by** (*auto dest: valid-abs intro: l3-dy-valid-refines-dy-fake-chan*)

**lemma** *l3-dy-isLtKey-refines-skip*:

$\{R23s\} \text{ Id, l3-dy } (LtK \text{ ltk}) \{>R23s\}$

**apply** (*auto simp add: PO-rhoare-defs R23s-defs l3-defs*)

**apply** (*auto elim!: absE*)

done

**lemma** *l3-dy-others-refines-skip*:

$\llbracket M \notin \text{range } LtK; M \notin \text{valid}; M \notin \text{payload} \rrbracket \implies$

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

**lemma** *l3-dy-refines-dy-fake-msg-dy-fake-chan-skip*:  
 $\{R23s \cap UNIV \times (l3-inv5 \cap l3-inv6 \cap l3-inv7 \cap l3-inv8 \cap l3-inv9)\}$   
 $l2-dy-fake-msg M \cup (\bigcup M'. l2-dy-fake-chan M') \cup Id, l3-dy M$   
 $\{>R23s\}$   
**by** (*cases M  $\in$  payload  $\cup$  valid  $\cup$  range LtK*)  
(*auto dest: l3-dy-payload-refines-dy-fake-msg l3-dy-valid-refines-dy-fake-chan-Un*  
*intro: l3-dy-isLtKey-refines-skip dest!: l3-dy-others-refines-skip*)

### 18.3.3 Compromise events

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

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

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

**lemmas** *l3-trans-refines-l2-trans =*  
*l3-step1-refines-step1 l3-step2-refines-step2 l3-step3-refines-step3*  
*l3-dy-refines-dy-fake-msg-dy-fake-chan-skip*  
*l3-lkr-others-refines-lkr-others l3-lkr-after-refines-lkr-after l3-skr-refines-skr*

**lemma** *l3-refines-init-l2 [iff]*:  
*init l3  $\subseteq$  R23s* “ (*init l2*)  
**by** (*auto simp add: R23s-defs l2-defs l3-def l3-init-def*)

**lemma** *l3-refines-trans-l2 [iff]*:  
 $\{R23s \cap (UNIV \times (l3-inv1 \cap l3-inv2 \cap l3-inv3 \cap l3-inv4))\}$  *trans l2*, *trans l3*  $\{> R23s\}$   
**proof** –  
**let** *?pre' = R23s  $\cap$  (UNIV  $\times$  (l3-inv5  $\cap$  l3-inv6  $\cap$  l3-inv7  $\cap$  l3-inv8  $\cap$  l3-inv9))*  
**show** *?thesis (is  $\{?pre\}$  ?t1, ?t2  $\{>?post\}$ )*

```

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

```

```

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

```

```

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

```

```

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

```

## 18.4 Derived invariants

### 18.4.1 inv10: secrets contain no implementation material

#### definition

*l3-inv10* :: *l3-state set*

#### where

```

l3-inv10  $\equiv$  {s.
  secret s  $\subseteq$  payload
}

```

**lemmas** *l3-inv10I* = *l3-inv10-def* [THEN setc-def-to-intro, rule-format]

**lemmas** *l3-inv10E* = *l3-inv10-def* [THEN setc-def-to-elim, rule-format]

**lemmas** *l3-inv10D* = *l3-inv10-def* [THEN setc-def-to-dest, rule-format]

**lemma** *l3-inv10-init* [iff]:

*init* l3  $\subseteq$  *l3-inv10*

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

**lemma** *l3-inv10-trans* [iff]:

{*l3-inv10*} trans l3 {> *l3-inv10*}

**apply** (auto simp add: PO-hoare-defs l3-nostep-defs)

**apply** (auto simp add: l3-defs l3-inv10-def)

**done**

**lemma** PO-l3-inv10 [iff]: reach l3  $\subseteq$  *l3-inv10*

**by** (rule inv-rule-basic) (auto)

**lemma** *l3-obs-inv10* [iff]: *oreach*  $l3 \subseteq l3\text{-inv10}$   
**by** (*auto simp add: oreach-def*)

### 18.4.2 Partial secrecy

We want to prove *l3-secrecy*, ie  $\text{synth} (\text{analz} (ik\ s)) \cap \text{secret}\ s = \{\}$ , but by refinement we only get *l3-partial-secrecy*:  $\text{dy-fake-msg} (l3\text{-state.bad}\ s) (\text{payloadSet} (ik\ s)) (\text{local.abs} (ik\ s)) \cap \text{secret}\ s = \{\}$ . This is fine if secrets contain no implementation material. Then, by *inv5*, a message in  $\text{synth} (\text{analz} (ik\ s))$  is in  $\text{dy-fake-msg} (l3\text{-state.bad}\ s) (\text{payloadSet} (ik\ s)) (\text{local.abs} (ik\ s)) \cup - \text{payload}$ , and *l3-partial-secrecy* proves it is not a secret.

**definition**

*l3-partial-secrecy* :: ('a *l3-state-scheme*) *set*

**where**

*l3-partial-secrecy*  $\equiv \{s.$   
 $\text{dy-fake-msg} (\text{bad}\ s) (ik\ s \cap \text{payload}) (\text{abs} (ik\ s)) \cap \text{secret}\ s = \{\}$   
 $\}$

**lemma** *l3-obs-partial-secrecy* [iff]: *oreach*  $l3 \subseteq l3\text{-partial-secrecy}$

**apply** (*rule external-invariant-translation [OF l2-obs-secrecy - l3-implements-l2]*)

**apply** (*auto simp add: med23s-def l2-secrecy-def l3-partial-secrecy-def*)

**done**

### 18.4.3 Secrecy

**definition**

*l3-secrecy* :: ('a *l3-state-scheme*) *set*

**where**

*l3-secrecy*  $\equiv l1\text{-secrecy}$

**lemma** *l3-obs-inv5*: *oreach*  $l3 \subseteq l3\text{-inv5}$

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

**lemma** *l3-obs-secrecy* [iff]: *oreach*  $l3 \subseteq l3\text{-secrecy}$

**apply** (*rule, frule l3-obs-inv5 [THEN [2] rev-subsetD], frule l3-obs-inv10 [THEN [2] rev-subsetD]*)

**apply** (*auto simp add: med23s-def l2-secrecy-def l3-secrecy-def s0-secrecy-def l3-inv10-def*)

**using** *l3-partial-secrecy-def* **apply** (*blast dest!: l3-inv5D subsetD [OF l3-obs-partial-secrecy]*)

**done**

**lemma** *l3-secrecy* [iff]: *reach*  $l3 \subseteq l3\text{-secrecy}$

**by** (*rule external-to-internal-invariant [OF l3-obs-secrecy], auto*)

### 18.4.4 Injective agreement

**abbreviation** *l3-iagreement*  $\equiv l1\text{-iagreement}$

**lemma** *l3-obs-iagreement* [iff]: *oreach*  $l3 \subseteq l3\text{-iagreement}$

**apply** (*rule external-invariant-translation [OF l2-obs-iagreement - l3-implements-l2]*)

**apply** (*auto simp add: med23s-def l1-iagreement-def*)

**done**

**lemma** *l3-agreement* [*iff*]: *reach l3*  $\subseteq$  *l3-agreement*  
**by** (*rule external-to-internal-invariant* [*OF l3-obs-agreement*], *auto*)

**end**  
**end**



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

```
theory pfslvl3-asymmetric
imports pfslvl3 Implem-asymmetric
begin

interpretation pfslvl3-asym: pfslvl3 implem-asym
by (unfold-locales)

end
```

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

```
theory pfs_l3-symmetric  
imports pfs_l3 Implem-symmetric  
begin  
  
interpretation pfs_l3-asym: pfs_l3 implem-sym  
by (unfold-locales)  
  
end
```

## 21 Authenticated Diffie Hellman Protocol (L1)

```
theory dhlvl1
imports Runs Secrecy AuthenticationI Payloads
begin
```

```
declare option.split-asm [split]
```

### 21.1 State and Events

```
consts
```

```
  Nend :: nat
```

```
abbreviation nx :: nat where nx ≡ 2
```

```
abbreviation ny :: nat where ny ≡ 3
```

Proofs break if  $1::'a$  is used, because *simp* replaces it with *Suc 0*...

```
abbreviation
```

```
  xEnd ≡ Var 0
```

```
abbreviation
```

```
  xnx ≡ Var 2
```

```
abbreviation
```

```
  xny ≡ Var 3
```

```
abbreviation
```

```
  xsk ≡ Var 4
```

```
abbreviation
```

```
  xgnx ≡ Var 5
```

```
abbreviation
```

```
  xgny ≡ Var 6
```

```
abbreviation
```

```
  End ≡ Number Nend
```

Domain of each role (protocol dependent).

```
fun domain :: role-t ⇒ var set where
```

```
  domain Init = {xnx, xgnx, xgny, xsk, xEnd}
```

```
| domain Resp = {xny, xgnx, xgny, xsk, xEnd}
```

```
consts
```

```
  test :: rid-t
```

```
consts
```

```
  guessed-runs :: rid-t ⇒ run-t
```

```
  guessed-frame :: rid-t ⇒ frame
```

Specification of the guessed frame:

1. Domain
2. Well-typedness. The messages in the frame of a run never contain implementation material even if the agents of the run are dishonest. Therefore we consider only well-typed frames. This is notably required for the session key compromise; it also helps proving the partitioning of ik, since we know that the messages added by the protocol do not contain lkeys in their payload and are therefore valid implementations.
3. We also ensure that the values generated by the frame owner are correctly guessed.

**specification** (*guessed-frame*)

*guessed-frame-dom-spec* [simp]:

$dom\ (guessed-frame\ R) = domain\ (role\ (guessed-runs\ R))$

*guessed-frame-payload-spec* [simp, elim]:

$guessed-frame\ R\ x = Some\ y \implies y \in payload$

*guessed-frame-Init-xnx* [simp]:

$role\ (guessed-runs\ R) = Init \implies guessed-frame\ R\ xnx = Some\ (NonceF\ (R\$nx))$

*guessed-frame-Init-xgnx* [simp]:

$role\ (guessed-runs\ R) = Init \implies guessed-frame\ R\ xgnx = Some\ (Exp\ Gen\ (NonceF\ (R\$nx)))$

*guessed-frame-Resp-xny* [simp]:

$role\ (guessed-runs\ R) = Resp \implies guessed-frame\ R\ xny = Some\ (NonceF\ (R\$ny))$

*guessed-frame-Resp-xgny* [simp]:

$role\ (guessed-runs\ R) = Resp \implies guessed-frame\ R\ xgny = Some\ (Exp\ Gen\ (NonceF\ (R\$ny)))$

*guessed-frame-xEnd* [simp]:

$guessed-frame\ R\ xEnd = Some\ End$

**apply** (rule *exI* [of -

$\lambda R.$

if  $role\ (guessed-runs\ R) = Init$  then

$[xnx \mapsto NonceF\ (R\$nx), xgnx \mapsto Exp\ Gen\ (NonceF\ (R\$nx)), xgny \mapsto End,$   
 $xsk \mapsto End, xEnd \mapsto End]$

else

$[xny \mapsto NonceF\ (R\$ny), xgnx \mapsto End, xgny \mapsto Exp\ Gen\ (NonceF\ (R\$ny)),$   
 $xsk \mapsto End, xEnd \mapsto End]$ ,

*auto simp add: domIff intro: role-t.exhaust*)

**done**

**abbreviation**

$test-owner \equiv owner\ (guessed-runs\ test)$

**abbreviation**

$test-partner \equiv partner\ (guessed-runs\ test)$

Level 1 state.

**record** *l1-state* =

*s0-state* +

*progress* :: *progress-t*

*signalsInit* :: *signal*  $\Rightarrow$  *nat*

*signalsResp* :: *signal*  $\Rightarrow$  *nat*

**type-synonym**  $l1\text{-obs} = l1\text{-state}$

**abbreviation**

$run\text{-ended} :: var\ set\ option \Rightarrow bool$

**where**

$run\text{-ended}\ r \equiv in\text{-progress}\ r\ xEnd$

**lemma**  $run\text{-ended}\text{-not}\text{-None}$  [elim]:

$run\text{-ended}\ R \Longrightarrow R = None \Longrightarrow False$

**by** (fast dest:  $in\text{-progress}\text{-Some}$ )

$test\text{-ended}\ s \longleftrightarrow$  the test run has ended in  $s$ .

**abbreviation**

$test\text{-ended} :: 'a\ l1\text{-state}\text{-scheme} \Rightarrow bool$

**where**

$test\text{-ended}\ s \equiv run\text{-ended}\ (progress\ s\ test)$

A run can emit signals if it involves the same agents as the test run, and if the test run has not ended yet.

**definition**

$can\text{-signal} :: 'a\ l1\text{-state}\text{-scheme} \Rightarrow agent \Rightarrow agent \Rightarrow bool$

**where**

$can\text{-signal}\ s\ A\ B \equiv$

$((A = test\text{-owner} \wedge B = test\text{-partner}) \vee (B = test\text{-owner} \wedge A = test\text{-partner})) \wedge$

$\neg test\text{-ended}\ s$

Events.

**definition**

$l1\text{-learn} :: msg \Rightarrow ('a\ l1\text{-state}\text{-scheme} * 'a\ l1\text{-state}\text{-scheme})\ set$

**where**

$l1\text{-learn}\ m \equiv \{(s, s')\}.$

— guard

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

— action

$s' = s\ (ik := ik\ s \cup \{m\})$

}

Protocol events.

- step 1: create  $Ra$ ,  $A$  generates  $nx$ , computes  $g^{nx}$
- step 2: create  $Rb$ ,  $B$  reads  $g^{nx}$  insecurely, generates  $ny$ , computes  $g^{ny}$ , computes  $g^{nx * ny}$ , emits a running signal for  $Init$ ,  $g^{nx * ny}$
- step 3:  $A$  reads  $g^{ny}$  and  $g^{nx}$  authentically, computes  $g^{ny * nx}$ , emits a commit signal for  $Init$ ,  $g^{ny * nx}$ , a running signal for  $Resp$ ,  $g^{ny * nx}$ , declares the secret  $g^{ny * nx}$
- step 4:  $B$  reads  $g^{nx}$  and  $g^{ny}$  authentically, emits a commit signal for  $Resp$ ,  $g^{nx * ny}$ , declares the secret  $g^{nx * ny}$

**definition**

$l1\text{-step1} :: \text{rid-}t \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow ('a \text{ l1-state-scheme} * 'a \text{ l1-state-scheme}) \text{ set}$

**where**

$l1\text{-step1 } Ra \ A \ B \equiv \{(s, s')\}.$

— guards:

$Ra \notin \text{dom } (\text{progress } s) \wedge$   
 $\text{guessed-runs } Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \wedge$

— actions:

$s' = s(\langle$   
 $\quad \text{progress} := (\text{progress } s)(Ra \mapsto \{xnx, xgnx\})$   
 $\quad \rangle$   
 $\}$

**definition**

$l1\text{-step2} :: \text{rid-}t \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow \text{msg} \Rightarrow ('a \text{ l1-state-scheme} * 'a \text{ l1-state-scheme}) \text{ set}$

**where**

$l1\text{-step2 } Rb \ A \ B \ gnx \equiv \{(s, s')\}.$

— guards:

$\text{guessed-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \wedge$   
 $Rb \notin \text{dom } (\text{progress } s) \wedge$   
 $\text{guessed-frame } Rb \ xgnx = \text{Some } gnx \wedge$   
 $\text{guessed-frame } Rb \ xsk = \text{Some } (\text{Exp } gnx (\text{NonceF } (Rb\$ny))) \wedge$

— actions:

$s' = s(\langle$   
 $\quad \text{progress} := (\text{progress } s)(Rb \mapsto \{xny, xgny, xgnx, xsk\}),$   
 $\quad \text{signalsInit} := \text{if can-signal } s \ A \ B \ \text{then}$   
 $\quad \quad \text{addSignal } (\text{signalsInit } s) \ (\text{Running } A \ B \ (\text{Exp } gnx (\text{NonceF } (Rb\$ny))))$   
 $\quad \text{else}$   
 $\quad \quad \text{signalsInit } s$   
 $\quad \rangle$   
 $\}$

**definition**

$l1\text{-step3} :: \text{rid-}t \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow \text{msg} \Rightarrow ('a \text{ l1-state-scheme} * 'a \text{ l1-state-scheme}) \text{ set}$

**where**

$l1\text{-step3 } Ra \ A \ B \ gny \equiv \{(s, s')\}.$

— guards:

$\text{guessed-runs } Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \wedge$   
 $\text{progress } s \ Ra = \text{Some } \{xnx, xgnx\} \wedge$   
 $\text{guessed-frame } Ra \ xgny = \text{Some } gny \wedge$   
 $\text{guessed-frame } Ra \ xsk = \text{Some } (\text{Exp } gny (\text{NonceF } (Ra\$nx))) \wedge$   
 $(\text{can-signal } s \ A \ B \longrightarrow \text{— authentication guard}$   
 $\quad (\exists Rb. \text{guessed-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \wedge$   
 $\quad \quad \text{in-progressS } (\text{progress } s \ Rb) \ \{xny, xgnx, xgny, xsk\} \wedge$   
 $\quad \quad \text{guessed-frame } Rb \ xgny = \text{Some } gny \wedge$   
 $\quad \quad \text{guessed-frame } Rb \ xgnx = \text{Some } (\text{Exp } \text{Gen } (\text{NonceF } (Ra\$nx)))) \wedge$   
 $(Ra = \text{test} \longrightarrow \text{Exp } gny (\text{NonceF } (Ra\$nx)) \notin \text{synth } (\text{analz } (\text{ik } s))) \wedge$

— actions:

$s' = s(\langle$   
 $\quad \text{progress} := (\text{progress } s)(Ra \mapsto \{xnx, xgnx, xgny, xsk, xEnd\}),$   
 $\quad \text{secret} := \{x. x = \text{Exp } gny (\text{NonceF } (Ra\$nx)) \wedge Ra = \text{test}\} \cup \text{secret } s,$   
 $\quad \text{signalsInit} := \text{if can-signal } s \ A \ B \ \text{then}$   
 $\quad \quad \text{addSignal } (\text{signalsInit } s) \ (\text{Commit } A \ B \ (\text{Exp } gny (\text{NonceF } (Ra\$nx))))$   
 $\quad \rangle$

```

      else
        signalsInit s,
signalsResp := if can-signal s A B then
  addSignal (signalsResp s) (Running A B (Exp gny (NonceF (Ra$nx))))
else
  signalsResp s
    }
  }
}

```

**definition**

$l1\text{-step4} :: \text{rid-t} \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow \text{msg} \Rightarrow ('a \text{ l1-state-scheme} * 'a \text{ l1-state-scheme}) \text{ set}$

**where**

$l1\text{-step4} \text{ Rb A B gnx} \equiv \{(s, s')\}.$

— guards:

$\text{guessed-runs Rb} = (\text{role}=\text{Resp}, \text{owner}=\text{B}, \text{partner}=\text{A}) \wedge$

$\text{progress s Rb} = \text{Some } \{xny, xgnx, xgny, xsk\} \wedge$

$\text{guessed-frame Rb xgnx} = \text{Some gnx} \wedge$

$(\text{can-signal s A B} \longrightarrow \text{— authentication guard}$

$(\exists \text{ Ra. guessed-runs Ra} = (\text{role}=\text{Init}, \text{owner}=\text{A}, \text{partner}=\text{B}) \wedge$

$\text{in-progressS (progress s Ra) } \{xnx, xgnx, xgny, xsk, xEnd\} \wedge$

$\text{guessed-frame Ra xgnx} = \text{Some gnx} \wedge$

$\text{guessed-frame Ra xgny} = \text{Some (Exp Gen (NonceF (Rb$ny)))) \wedge$

$(\text{Rb} = \text{test} \longrightarrow \text{Exp gnx (NonceF (Rb$ny))} \notin \text{synth (analz (ik s))}) \wedge$

— actions:

$s' = s(\text{progress} := (\text{progress s})(\text{Rb} \mapsto \{xny, xgnx, xgny, xsk, xEnd\}),$

$\text{secret} := \{x. x = \text{Exp gnx (NonceF (Rb$ny))} \wedge \text{Rb} = \text{test}\} \cup \text{secret s},$

$\text{signalsResp} := \text{if can-signal s A B then}$

$\text{addSignal (signalsResp s) (Commit A B (Exp gnx (NonceF (Rb$ny))))}$

$\text{else}$

$\text{signalsResp s}$

```

    }
  }
}

```

Specification.

**definition**

$l1\text{-init} :: \text{l1-state set}$

**where**

$l1\text{-init} \equiv \{ \{$

$\text{ik} = \{\},$

$\text{secret} = \{\},$

$\text{progress} = \text{Map.empty},$

$\text{signalsInit} = \lambda x. 0,$

$\text{signalsResp} = \lambda x. 0$

$\} \}$

**definition**

$l1\text{-trans} :: ('a \text{ l1-state-scheme} * 'a \text{ l1-state-scheme}) \text{ set}$  **where**

$l1\text{-trans} \equiv (\bigcup m \text{ Ra Rb A B } x.$

$l1\text{-step1 Ra A B} \cup$

$l1\text{-step2 Rb A B } x \cup$

```

    l1-step3 Ra A B x ∪
    l1-step4 Rb A B x ∪
    l1-learn m ∪
    Id
  )

```

**definition**

```

l1 :: (l1-state, l1-obs) spec where
l1 ≡ ⟨
  init = l1-init,
  trans = l1-trans,
  obs = id
⟩

```

**lemmas** *l1-defs* =

```

l1-def l1-init-def l1-trans-def
l1-learn-def
l1-step1-def l1-step2-def l1-step3-def l1-step4-def

```

**lemmas** *l1-nostep-defs* =

```

l1-def l1-init-def l1-trans-def

```

**lemma** *l1-obs-id* [*simp*]: *obs l1 = id*

**by** (*simp add: l1-def*)

**lemma** *run-ended-trans*:

```

run-ended (progress s R) ⇒
(s, s') ∈ trans l1 ⇒
run-ended (progress s' R)

```

**apply** (*auto simp add: l1-nostep-defs*)

**apply** (*auto simp add: l1-defs ik-dy-def*)

**done**

**lemma** *can-signal-trans*:

```

can-signal s' A B ⇒
(s, s') ∈ trans l1 ⇒
can-signal s A B

```

**by** (*auto simp add: can-signal-def run-ended-trans*)

## 21.2 Refinement: secrecy

Mediator function.

**definition**

```

med01s :: l1-obs ⇒ s0-obs

```

**where**

```

med01s t ≡ ⟨ ik = ik t, secret = secret t ⟩

```

Relation between states.

**definition**

```

R01s :: (s0-state * l1-state) set

```

**where**



$$R01s \equiv \{(s, s').$$

$$s = (ik = ik\ s', secret = secret\ s')$$

$$\}$$

Protocol independent events.

**lemma** *l1-learn-refines-learn*:  
 $\{R01s\}$  *s0-learn m, l1-learn m*  $\{>R01s\}$   
**apply** (*simp add: PO-rhoare-defs R01s-def*)  
**apply** *auto*  
**apply** (*simp add: l1-defs s0-defs s0-secrecy-def*)  
**done**

Protocol events.

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

**lemma** *l1-step2-refines-skip*:  
 $\{R01s\}$  *Id, l1-step2 Rb A B gnx*  $\{>R01s\}$   
**apply** (*auto simp add: PO-rhoare-defs R01s-def*)  
**apply** (*auto simp add: l1-step2-def*)  
**done**

**lemma** *l1-step3-refines-add-secret-skip*:  
 $\{R01s\}$  *s0-add-secret (Exp gny (NonceF (Ra\$nx)))*  $\cup Id, l1-step3 Ra A B gny$   $\{>R01s\}$   
**apply** (*auto simp add: PO-rhoare-defs R01s-def s0-add-secret-def*)  
**apply** (*auto simp add: l1-step3-def*)  
**done**

**lemma** *l1-step4-refines-add-secret-skip*:  
 $\{R01s\}$  *s0-add-secret (Exp gnx (NonceF (Rb\$ny)))*  $\cup Id, l1-step4 Rb A B gnx$   $\{>R01s\}$   
**apply** (*auto simp add: PO-rhoare-defs R01s-def s0-add-secret-def*)  
**apply** (*auto simp add: l1-step4-def*)  
**done**

Refinement proof.

**lemmas** *l1-trans-refines-s0-trans =*  
*l1-learn-refines-learn*  
*l1-step1-refines-skip l1-step2-refines-skip*  
*l1-step3-refines-add-secret-skip l1-step4-refines-add-secret-skip*

**lemma** *l1-refines-init-s0 [iff]*:  
*init l1*  $\subseteq R01s$  “ (*init s0*)  
**by** (*auto simp add: R01s-def s0-defs l1-defs s0-secrecy-def*)

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

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

Refinement result.

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

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

### 21.3 Derived invariants: secrecy

**abbreviation** *l1-secrecy*  $\equiv$  *s0-secrecy*

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

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

### 21.4 Invariants: *Init* authenticates *Resp*

#### 21.4.1 *inv1*

If an initiator commit signal exists for  $(g^{ny})Ra \$ nx$  then *Ra* is *Init*, has passed step 3, and has  $(g \hat{ny}) \wedge (Ra \$ nx)$  as the key in its frame.

**definition**

*l1-inv1* :: *l1-state set*

**where**

*l1-inv1*  $\equiv$   $\{s. \forall Ra A B gny.$   
*signalsInit* *s* (*Commit A B (Exp gny (NonceF (Ra \$ nx)))*)  $> 0 \longrightarrow$   
*guessed-runs* *Ra* = ( $\langle$ *role=Init, owner=A, partner=B* $\rangle$ )  $\wedge$   
*progress* *s* *Ra* = *Some* {*xnx, xgnx, xgny, xsk, xEnd*}  $\wedge$   
*guessed-frame* *Ra* *xsk* = *Some* (*Exp gny (NonceF (Ra \$ nx))*)  
 $\}$

**lemmas** *l1-inv1I* = *l1-inv1-def* [*THEN setc-def-to-intro, rule-format*]

**lemmas** *l1-inv1E* [*elim*] = *l1-inv1-def* [*THEN setc-def-to-elim, rule-format*]

**lemmas** *l1-inv1D* = *l1-inv1-def* [*THEN setc-def-to-dest, rule-format, rotated 1, simplified*]

**lemma** *l1-inv1-init* [iff]:

*init l1  $\subseteq$  l1-inv1*

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

**lemma** *l1-inv1-trans* [iff]:  
 {*l1-inv1*} *trans l1* {> *l1-inv1*}  
**apply** (*auto simp add: PO-hoare-defs l1-nostep-defs intro!: l1-inv1I*)  
**apply** (*auto simp add: l1-defs ik-dy-def l1-inv1-def dest: Exp-Exp-Gen-inj2 [OF sym]*)  
**done**

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

### 21.4.2 inv2

If a *Resp* run *Rb* has passed step 2 then (if possible) an initiator running signal has been emitted.

#### definition

*l1-inv2* :: *l1-state set*

#### where

*l1-inv2*  $\equiv$  {*s*.  $\forall$  *gnx A B Rb*.  
*guessed-runs Rb* = (*role=Resp, owner=B, partner=A*)  $\longrightarrow$   
*in-progressS* (*progress s Rb*) {*xny, xgnx, xgny, xsk*}  $\longrightarrow$   
*guessed-frame Rb xgnx* = *Some gnx*  $\longrightarrow$   
*can-signal s A B*  $\longrightarrow$   
*signalsInit s* (*Running A B* (*Exp gnx* (*NonceF* (*Rb\$ny*))))  $> 0$   
}

**lemmas** *l1-inv2I* = *l1-inv2-def* [*THEN setc-def-to-intro, rule-format*]

**lemmas** *l1-inv2E* [*elim*] = *l1-inv2-def* [*THEN setc-def-to-elim, rule-format*]

**lemmas** *l1-inv2D* = *l1-inv2-def* [*THEN setc-def-to-dest, rule-format, rotated 1, simplified*]

**lemma** *l1-inv2-init* [iff]:  
*init l1*  $\subseteq$  *l1-inv2*  
**by** (*auto simp add: l1-def l1-init-def l1-inv2-def*)

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

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

### 21.4.3 inv3 (derived)

If an *Init* run before step 3 and a *Resp* run after step 2 both know the same half-keys (more or less), then the number of *Init* running signals for the key is strictly greater than the number of *Init* commit signals. (actually, there are 0 commit and 1 running).

#### definition

$l1\text{-inv}3 :: l1\text{-state set}$

**where**

$l1\text{-inv}3 \equiv \{s. \forall A B Rb Ra gny.$   
 $\text{guessed-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \longrightarrow$   
 $\text{in-progress}S (\text{progress } s Rb) \{xny, xgnx, xgny, xsk\} \longrightarrow$   
 $\text{guessed-frame } Rb \text{ } xgny = \text{Some } gny \longrightarrow$   
 $\text{guessed-frame } Rb \text{ } xgnx = \text{Some } (\text{Exp Gen } (\text{NonceF } (Ra\$nx))) \longrightarrow$   
 $\text{guessed-runs } Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \longrightarrow$   
 $\text{progress } s Ra = \text{Some } \{xnx, xgnx\} \longrightarrow$   
 $\text{can-signal } s A B \longrightarrow$   
 $\text{signalsInit } s (\text{Commit } A B (\text{Exp } gny (\text{NonceF } (Ra\$nx))))$   
 $< \text{signalsInit } s (\text{Running } A B (\text{Exp } gny (\text{NonceF } (Ra\$nx))))$   
 $\}$

**lemmas**  $l1\text{-inv}3I = l1\text{-inv}3\text{-def}$  [*THEN setc-def-to-intro, rule-format*]

**lemmas**  $l1\text{-inv}3E$  [*elim*] =  $l1\text{-inv}3\text{-def}$  [*THEN setc-def-to-elim, rule-format*]

**lemmas**  $l1\text{-inv}3D = l1\text{-inv}3\text{-def}$  [*THEN setc-def-to-dest, rule-format, rotated 1, simplified*]

**lemma**  $l1\text{-inv}3\text{-derived}: l1\text{-inv}1 \cap l1\text{-inv}2 \subseteq l1\text{-inv}3$

**apply** (*auto intro: l1-inv3I*)

**apply** (*auto dest!: l1-inv2D*)

**apply** (*rename-tac x A B Rb Ra*)

**apply** (*case-tac*)

$\text{signalsInit } x (\text{Commit } A B (\text{Exp } (\text{Exp Gen } (\text{NonceF } (Rb \$ ny))) (\text{NonceF } (Ra \$ nx)))) > 0$ , *auto*)

**apply** (*fastforce dest: l1-inv1D elim: equalityE*)

**done**

## 21.5 Invariants: *Resp* authenticates *Init*

### 21.5.1 inv4

If a *Resp* commit signal exists for  $(g^{nx})Rb \$ ny$  then *Rb* is *Resp*, has finished its run, and has  $(g^{nx})Rb \$ ny$  as the key in its frame.

**definition**

$l1\text{-inv}4 :: l1\text{-state set}$

**where**

$l1\text{-inv}4 \equiv \{s. \forall Rb A B gnx.$   
 $\text{signalsResp } s (\text{Commit } A B (\text{Exp } gnx (\text{NonceF } (Rb\$ny)))) > 0 \longrightarrow$   
 $\text{guessed-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \wedge$   
 $\text{progress } s Rb = \text{Some } \{xny, xgnx, xgny, xsk, xEnd\} \wedge$   
 $\text{guessed-frame } Rb \text{ } xgnx = \text{Some } gnx$   
 $\}$

**lemmas**  $l1\text{-inv}4I = l1\text{-inv}4\text{-def}$  [*THEN setc-def-to-intro, rule-format*]

**lemmas**  $l1\text{-inv}4E$  [*elim*] =  $l1\text{-inv}4\text{-def}$  [*THEN setc-def-to-elim, rule-format*]

**lemmas**  $l1\text{-inv}4D = l1\text{-inv}4\text{-def}$  [*THEN setc-def-to-dest, rule-format, rotated 1, simplified*]

**lemma**  $l1\text{-inv}4\text{-init}$  [*iff*]:

$\text{init } l1 \subseteq l1\text{-inv}4$

**by** (*auto simp add: l1-def l1-init-def l1-inv4-def*)

**declare** *domIff* [*iff*]

**lemma** *l1-inv4-trans* [*iff*]:

{*l1-inv4*} *trans l1* {> *l1-inv4*}

**apply** (*auto simp add: PO-hoare-defs l1-nostep-defs intro!: l1-inv4I*)

**apply** (*auto simp add: l1-inv4-def l1-defs ik-dy-def dest: Exp-Exp-Gen-inj2 [OF sym]*)

**done**

**declare** *domIff* [*iff del*]

**lemma** *PO-l1-inv4* [*iff*]: *reach l1*  $\subseteq$  *l1-inv4*

**by** (*rule inv-rule-basic*) (*auto*)

### 21.5.2 inv5

If an *Init* run *Ra* has passed step3 then (if possible) a *Resp* running signal has been emitted.

**definition**

*l1-inv5* :: *l1-state set*

**where**

*l1-inv5*  $\equiv$  {*s*.  $\forall$  *gny A B Ra*.

*guessed-runs Ra* = (*role=Init, owner=A, partner=B*)  $\longrightarrow$

*in-progressS* (*progress s Ra*) {*xnx, xgnx, xgny, xsk, xEnd*}  $\longrightarrow$

*guessed-frame Ra xgny* = *Some gny*  $\longrightarrow$

*can-signal s A B*  $\longrightarrow$

*signalsResp s* (*Running A B (Exp gny (NonceF (Ra\$nx)))*) > 0

}

**lemmas** *l1-inv5I* = *l1-inv5-def* [*THEN setc-def-to-intro, rule-format*]

**lemmas** *l1-inv5E* [*elim*] = *l1-inv5-def* [*THEN setc-def-to-elim, rule-format*]

**lemmas** *l1-inv5D* = *l1-inv5-def* [*THEN setc-def-to-dest, rule-format, rotated 1, simplified*]

**lemma** *l1-inv5-init* [*iff*]:

*init l1*  $\subseteq$  *l1-inv5*

**by** (*auto simp add: l1-def l1-init-def l1-inv5-def*)

**lemma** *l1-inv5-trans* [*iff*]:

{*l1-inv5*} *trans l1* {> *l1-inv5*}

**apply** (*auto simp add: PO-hoare-defs intro!: l1-inv5I*)

**apply** (*drule can-signal-trans, assumption*)

**apply** (*auto simp add: l1-nostep-defs*)

**apply** (*auto simp add: l1-defs ik-dy-def l1-inv5-def*)

**done**

**lemma** *PO-l1-inv5* [*iff*]: *reach l1*  $\subseteq$  *l1-inv5*

**by** (*rule inv-rule-basic*) (*auto*)

### 21.5.3 inv6 (derived)

If a *Resp* run before step 4 and an *Init* run after step 3 both know the same half-keys (more or less), then the number of *Resp* running signals for the key is strictly greater than the number of *Resp* commit signals. (actually, there are 0 commit and 1 running).

**definition**

$l1\text{-inv6} :: l1\text{-state set}$

**where**

$l1\text{-inv6} \equiv \{s. \forall A B Rb Ra gnx.$   
 $\text{guessed-runs } Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \longrightarrow$   
 $\text{in-progressS } (\text{progress } s Ra) \{xnx, xgnx, xgny, xsk, xEnd\} \longrightarrow$   
 $\text{guessed-frame } Ra \text{ } xgnx = \text{Some } gnx \longrightarrow$   
 $\text{guessed-frame } Ra \text{ } xgny = \text{Some } (\text{Exp Gen } (\text{NonceF } (Rb\$ny))) \longrightarrow$   
 $\text{guessed-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \longrightarrow$   
 $\text{progress } s Rb = \text{Some } \{xny, xgnx, xgny, xsk\} \longrightarrow$   
 $\text{can-signal } s A B \longrightarrow$   
 $\text{signalsResp } s (\text{Commit } A B (\text{Exp } gnx (\text{NonceF } (Rb\$ny))))$   
 $< \text{signalsResp } s (\text{Running } A B (\text{Exp } gnx (\text{NonceF } (Rb\$ny))))$   
 $\}$

**lemmas**  $l1\text{-inv6I} = l1\text{-inv6-def } [THEN \text{setc-def-to-intro}, \text{rule-format}]$

**lemmas**  $l1\text{-inv6E } [elim] = l1\text{-inv6-def } [THEN \text{setc-def-to-elim}, \text{rule-format}]$

**lemmas**  $l1\text{-inv6D} = l1\text{-inv6-def } [THEN \text{setc-def-to-dest}, \text{rule-format}, \text{rotated } 1, \text{simplified}]$

**lemma**  $l1\text{-inv6-derived}$ :

$l1\text{-inv4} \cap l1\text{-inv5} \subseteq l1\text{-inv6}$

**proof** (*auto intro!*:  $l1\text{-inv6I}$ )

**fix**  $s::l1\text{-state}$  **fix**  $A B Rb Ra$

**assume**  $HRun:\text{guessed-runs } Ra = (\text{role} = \text{Init}, \text{owner} = A, \text{partner} = B)$   
 $\text{in-progressS } (\text{progress } s Ra) \{xnx, xgnx, xgny, xsk, xEnd\}$   
 $\text{guessed-frame } Ra \text{ } xgny = \text{Some } (\text{Exp Gen } (\text{NonceF } (Rb \$ ny)))$   
 $\text{can-signal } s A B$

**assume**  $HRb:\text{progress } s Rb = \text{Some } \{xny, xgnx, xgny, xsk\}$

**assume**  $I4:s \in l1\text{-inv4}$

**assume**  $I5:s \in l1\text{-inv5}$

**from**  $I4 HRb$

**have**  $\text{signalsResp } s (\text{Commit } A B (\text{Exp } (\text{Exp Gen } (\text{NonceF } (Rb \$ ny))) (\text{NonceF } (Ra \$ nx)))) > 0$   
 $\implies \text{False}$

**proof** (*auto dest!*:  $l1\text{-inv4D}$ )

**assume**  $\{xny, xgnx, xgny, xsk, xEnd\} = \{xny, xgnx, xgny, xsk\}$

**thus** *?thesis* **by force**

**qed**

**then have**

$HC:\text{signalsResp } s (\text{Commit } A B (\text{Exp } (\text{Exp Gen } (\text{NonceF } (Rb \$ ny))) (\text{NonceF } (Ra \$ nx)))) = 0$

**by auto**

**from**  $I5 HRun$

**have**  $\text{signalsResp } s (\text{Running } A B (\text{Exp } (\text{Exp Gen } (\text{NonceF } (Rb \$ ny))) (\text{NonceF } (Ra \$ nx)))) > 0$   
**by** (*auto dest!*:  $l1\text{-inv5D}$ )

**with**  $HC$  **show**

$\text{signalsResp } s (\text{Commit } A B (\text{Exp } (\text{Exp Gen } (\text{NonceF } (Rb \$ ny))) (\text{NonceF } (Ra \$ nx))))$   
 $< \text{signalsResp } s (\text{Running } A B (\text{Exp } (\text{Exp Gen } (\text{NonceF } (Rb \$ ny))) (\text{NonceF } (Ra \$ nx))))$

**by auto**

**qed**

## 21.6 Refinement: injective agreement (*Init* authenticates *Resp*)

Mediator function.

**definition**

$med01iai :: l1\text{-obs} \Rightarrow a0i\text{-obs}$

**where**

$med01iai\ t \equiv \langle a0n\text{-state}.signals = signalsInit\ t \rangle$

Relation between states.

**definition**

$R01iai :: (a0i\text{-state} * l1\text{-state})\ set$

**where**

$R01iai \equiv \{(s, s') \mid a0n\text{-state}.signals\ s = signalsInit\ s'\}$

Protocol-independent events.

**lemma**  $l1\text{-learn-refines-}a0i\text{-ia-skip-i}$ :

$\{R01iai\}\ Id, l1\text{-learn}\ m\ \{>R01iai\}$

**apply** ( $auto\ simp\ add: PO\text{-rhoare-defs}\ R01iai\text{-def}$ )

**apply** ( $simp\ add: l1\text{-learn-def}$ )

**done**

Protocol events.

**lemma**  $l1\text{-step1-refines-}a0i\text{-skip-i}$ :

$\{R01iai\}\ Id, l1\text{-step1}\ Ra\ A\ B\ \{>R01iai\}$

**by** ( $auto\ simp\ add: PO\text{-rhoare-defs}\ R01iai\text{-def}\ l1\text{-step1-def}$ )

**lemma**  $l1\text{-step2-refines-}a0i\text{-running-skip-i}$ :

$\{R01iai\}\ a0i\text{-running}\ A\ B\ (Exp\ gnx\ (NonceF\ (Rb\$ny))) \cup Id, l1\text{-step2}\ Rb\ A\ B\ gnx\ \{>R01iai\}$

**by** ( $auto\ simp\ add: PO\text{-rhoare-defs}\ R01iai\text{-def}, simp\text{-all}\ add: l1\text{-step2-def}\ a0i\text{-running-def}, auto$ )

**lemma**  $l1\text{-step3-refines-}a0i\text{-commit-skip-i}$ :

$\{R01iai \cap (UNIV \times l1\text{-inv3})\}$   
 $a0i\text{-commit}\ A\ B\ (Exp\ gny\ (NonceF\ (Ra\$nx))) \cup Id,$   
 $l1\text{-step3}\ Ra\ A\ B\ gny$   
 $\{>R01iai\}$

**apply** ( $auto\ simp\ add: PO\text{-rhoare-defs}\ R01iai\text{-def}$ )

**apply** ( $auto\ simp\ add: l1\text{-step3-def}\ a0i\text{-commit-def}$ )

**apply** ( $force\ elim!: l1\text{-inv3E}$ )<sup>+</sup>

**done**

**lemma**  $l1\text{-step4-refines-}a0i\text{-skip-i}$ :

$\{R01iai\}\ Id, l1\text{-step4}\ Rb\ A\ B\ gnx\ \{>R01iai\}$

**by** ( $auto\ simp\ add: PO\text{-rhoare-defs}\ R01iai\text{-def}, auto\ simp\ add: l1\text{-step4-def}$ )

Refinement proof.

**lemmas**  $l1\text{-trans-refines-}a0i\text{-trans-i} =$ 

$l1\text{-learn-refines-}a0i\text{-ia-skip-i}$   
 $l1\text{-step1-refines-}a0i\text{-skip-i}\ l1\text{-step2-refines-}a0i\text{-running-skip-i}$   
 $l1\text{-step3-refines-}a0i\text{-commit-skip-i}\ l1\text{-step4-refines-}a0i\text{-skip-i}$

**lemma**  $l1\text{-refines-init-}a0i\text{-i}$  [iff]:

$init\ l1 \subseteq R01iai \text{ “ } (init\ a0i)$

by (auto simp add: R01iai-def a0i-defs l1-defs)

**lemma** *l1-refines-trans-a0i-i* [iff]:

$\{R01iai \cap (UNIV \times (l1-inv1 \cap l1-inv2))\}$  trans a0i, trans l1  $\{> R01iai\}$

**proof** –

let  $?pre' = R01iai \cap (UNIV \times l1-inv3)$

**show** ?thesis (is  $\{?pre\}$  ?t1, ?t2  $\{> ?post\}$ )

**proof** (rule relhoare-conseq-left)

show  $?pre \subseteq ?pre'$

using l1-inv3-derived by blast

next

show  $\{?pre'\}$  ?t1, ?t2  $\{> ?post\}$

apply (auto simp add: a0i-def l1-def a0i-trans-def l1-trans-def)

prefer 2 using l1-step2-refines-a0i-running-skip-i apply (simp add: PO-rhoare-defs, blast)

prefer 2 using l1-step3-refines-a0i-commit-skip-i apply (simp add: PO-rhoare-defs, blast)

apply (blast intro!:l1-trans-refines-a0i-trans-i)+

done

qed

qed

**lemma** *obs-consistent-med01iai* [iff]:

obs-consistent R01iai med01iai a0i l1

by (auto simp add: obs-consistent-def R01iai-def med01iai-def)

Refinement result.

**lemma** *l1-refines-a0i-i* [iff]:

refines

$(R01iai \cap (reach\ a0i \times (l1-inv1 \cap l1-inv2)))$

med01iai a0i l1

by (rule Refinement-using-invariants, auto)

**lemma** *l1-implements-a0i-i* [iff]: implements med01iai a0i l1

by (rule refinement-soundness) (fast)

## 21.7 Derived invariants: injective agreement (*Init* authenticates *Resp*)

**definition**

*l1-iagreement-Init* :: ('a l1-state-scheme) set

**where**

$l1-iagreement-Init \equiv \{s. \forall A B N.$

$signalsInit\ s\ (Commit\ A\ B\ N) \leq signalsInit\ s\ (Running\ A\ B\ N)$

$\}$

**lemmas** *l1-iagreement-InitI* = *l1-iagreement-Init-def* [THEN setc-def-to-intro, rule-format]

**lemmas** *l1-iagreement-InitE* [elim] = *l1-iagreement-Init-def* [THEN setc-def-to-elim, rule-format]

**lemma** *l1-obs-iagreement-Init* [iff]: oreach l1  $\subseteq l1-iagreement-Init$

**apply** (rule external-invariant-translation

[OF PO-a0i-obs-agreement - l1-implements-a0i-i])

**apply** (auto simp add: med01iai-def l1-iagreement-Init-def a0i-agreement-def)



done

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

## 21.8 Refinement: injective agreement (*Resp* authenticates *Init*)

Mediator function.

**definition**

*med01iar* ::  $l1\text{-obs} \Rightarrow a0i\text{-obs}$

**where**

*med01iar*  $t \equiv (\lambda a0n\text{-state}. \text{signals} = \text{signalsResp } t)$

Relation between states.

**definition**

*R01iar* ::  $(a0i\text{-state} * l1\text{-state}) \text{ set}$

**where**

*R01iar*  $\equiv \{(s, s').$   
 $a0n\text{-state}. \text{signals } s = \text{signalsResp } s'$   
 $\}$

Protocol-independent events.

**lemma** *l1-learn-refines-a0-ia-skip-r*:

$\{R01iar\} \text{ Id, } l1\text{-learn } m \{>R01iar\}$

**apply** (auto simp add: *PO-rhoare-defs R01iar-def*)

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

done

Protocol events.

**lemma** *l1-step1-refines-a0i-skip-r*:

$\{R01iar\} \text{ Id, } l1\text{-step1 } Ra \ A \ B \{>R01iar\}$

**by** (auto simp add: *PO-rhoare-defs R01iar-def l1-step1-def*)

**lemma** *l1-step2-refines-a0i-skip-r*:

$\{R01iar\} \text{ Id, } l1\text{-step2 } Rb \ A \ B \ gnx \{>R01iar\}$

**by** (auto simp add: *PO-rhoare-defs R01iar-def, auto simp add:l1-step2-def*)

**lemma** *l1-step3-refines-a0i-running-skip-r*:

$\{R01iar\} \ a0i\text{-running } A \ B \ (\text{Exp } gny \ (\text{NonceF } (Ra\$nx))) \cup \text{ Id, } l1\text{-step3 } Ra \ A \ B \ gny \{>R01iar\}$

**by** (auto simp add: *PO-rhoare-defs R01iar-def, simp-all add: l1-step3-def a0i-running-def, auto*)

**lemma** *l1-step4-refines-a0i-commit-skip-r*:

$\{R01iar \cap UNIV \times l1\text{-inv6}\}$

$a0i\text{-commit } A \ B \ (\text{Exp } gnx \ (\text{NonceF } (Rb\$ny))) \cup \text{ Id,}$

$l1\text{-step4 } Rb \ A \ B \ gnx$

$\{>R01iar\}$

**apply** (auto simp add: *PO-rhoare-defs R01iar-def*)

**apply** (auto simp add: *l1-step4-def a0i-commit-def*)

**apply** (auto dest!: *l1-inv6D* [rotated 1])

done

Refinement proofs.

**lemmas** *l1-trans-refines-a0i-trans-r* =  
*l1-learn-refines-a0-ia-skip-r*  
*l1-step1-refines-a0i-skip-r l1-step2-refines-a0i-skip-r*  
*l1-step3-refines-a0i-running-skip-r l1-step4-refines-a0i-commit-skip-r*

**lemma** *l1-refines-init-a0i-r* [iff]:  
 $init\ l1 \subseteq R01iar \text{ “ (init } a0i)$   
**by** (*auto simp add: R01iar-def a0i-defs l1-defs*)

**lemma** *l1-refines-trans-a0i-r* [iff]:  
 $\{R01iar \cap (UNIV \times (l1-inv4 \cap l1-inv5))\} \text{ trans } a0i, \text{ trans } l1 \{> R01iar\}$   
**proof** –  
**let**  $?pre' = R01iar \cap (UNIV \times l1-inv6)$   
**show**  $?thesis \text{ (is } \{?pre\} ?t1, ?t2 \{> ?post\})$   
**proof** (*rule relhoare-conseq-left*)  
**show**  $?pre \subseteq ?pre'$   
**using** *l1-inv6-derived* **by** *blast*  
**next**  
**show**  $\{?pre'\} ?t1, ?t2 \{> ?post\}$   
**apply** (*auto simp add: a0i-def l1-def a0i-trans-def l1-trans-def*)  
**prefer** 3 **using** *l1-step3-refines-a0i-running-skip-r* **apply** (*simp add: PO-rhoare-defs, blast*)  
**prefer** 3 **using** *l1-step4-refines-a0i-commit-skip-r* **apply** (*simp add: PO-rhoare-defs, blast*)  
**apply** (*blast intro!:l1-trans-refines-a0i-trans-r*)  
**done**  
**qed**  
**qed**

**lemma** *obs-consistent-med01iar* [iff]:  
 $obs-consistent\ R01iar\ med01iar\ a0i\ l1$   
**by** (*auto simp add: obs-consistent-def R01iar-def med01iar-def*)

Refinement result.

**lemma** *l1-refines-a0i-r* [iff]:  
 $refines$   
 $(R01iar \cap (reach\ a0i \times (l1-inv4 \cap l1-inv5)))$   
 $med01iar\ a0i\ l1$   
**by** (*rule Refinement-using-invariants, auto*)

**lemma** *l1-implements-a0i-r* [iff]: *implements med01iar a0i l1*  
**by** (*rule refinement-soundness*) (*fast*)

## 21.9 Derived invariants: injective agreement (*Resp* authenticates *Init*)

**definition**

$l1-agreement-Resp :: ('a\ l1-state-scheme)\ set$

**where**

$l1-agreement-Resp \equiv \{s. \forall A\ B\ N.$

$signalsResp\ s\ (Commit\ A\ B\ N) \leq signalsResp\ s\ (Running\ A\ B\ N)$

$\}$

**lemmas**  $l1\text{-iagreement-RespI} = l1\text{-iagreement-Resp-def}$  [THEN setc-def-to-intro, rule-format]  
**lemmas**  $l1\text{-iagreement-RespE}$  [elim] =  $l1\text{-iagreement-Resp-def}$  [THEN setc-def-to-elim, rule-format]

**lemma**  $l1\text{-obs-iagreement-Resp}$  [iff]:  $oreach\ l1 \subseteq l1\text{-iagreement-Resp}$   
**apply** (rule external-invariant-translation  
[OF PO-a0i-obs-agreement - l1-implements-a0i-r])  
**apply** (auto simp add: med01iar-def l1-iagreement-Resp-def a0i-agreement-def)  
**done**

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

**end**

## 22 Authenticated Diffie-Hellman Protocol (L2)

```
theory dhlvl2
imports dhlvl1 Channels
begin
```

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

### 22.1 State and Events

Initial compromise.

```
consts
```

```
  bad-init :: agent set
```

```
specification (bad-init)
```

```
  bad-init-spec: test-owner  $\notin$  bad-init  $\wedge$  test-partner  $\notin$  bad-init
```

```
by auto
```

Level 2 state.

```
record l2-state =
```

```
  l1-state +
```

```
  chan :: chan set
```

```
  bad :: agent set
```

```
type-synonym l2-obs = l2-state
```

```
type-synonym
```

```
  l2-pred = l2-state set
```

```
type-synonym
```

```
  l2-trans = (l2-state  $\times$  l2-state) set
```

Attacker events.

```
definition
```

```
  l2-dy-fake-msg :: msg  $\Rightarrow$  l2-trans
```

```
where
```

```
  l2-dy-fake-msg m  $\equiv$   $\{(s, s')\}$ .
```

```
  — guards
```

```
  m  $\in$  dy-fake-msg (bad s) (ik s) (chan s)  $\wedge$ 
```

```
  — actions
```

```
  s' = s(ik :=  $\{m\} \cup$  ik s)
```

```
}
```

```
definition
```

```
  l2-dy-fake-chan :: chan  $\Rightarrow$  l2-trans
```

```
where
```

```
  l2-dy-fake-chan M  $\equiv$   $\{(s, s')\}$ .
```

```
  — guards
```

```
  M  $\in$  dy-fake-chan (bad s) (ik s) (chan s)  $\wedge$ 
```

```
  — actions
```

```
  s' = s(chan :=  $\{M\} \cup$  chan s)
```

}

Partnering.

**fun**

*role-comp* :: *role-t*  $\Rightarrow$  *role-t*

**where**

*role-comp* *Init* = *Resp*

| *role-comp* *Resp* = *Init*

**definition**

*matching* :: *frame*  $\Rightarrow$  *frame*  $\Rightarrow$  *bool*

**where**

*matching* *sigma* *sigma'*  $\equiv \forall x. x \in \text{dom } \textit{sigma} \cap \text{dom } \textit{sigma}' \longrightarrow \textit{sigma } x = \textit{sigma}' x$

**definition**

*partner-runs* :: *rid-t*  $\Rightarrow$  *rid-t*  $\Rightarrow$  *bool*

**where**

*partner-runs* *R* *R'*  $\equiv$

*role* (*guessed-runs* *R*) = *role-comp* (*role* (*guessed-runs* *R'*))  $\wedge$

*owner* (*guessed-runs* *R*) = *partner* (*guessed-runs* *R'*)  $\wedge$

*owner* (*guessed-runs* *R'*) = *partner* (*guessed-runs* *R*)  $\wedge$

*matching* (*guessed-frame* *R*) (*guessed-frame* *R'*)

**lemma** *role-comp-inv* [*simp*]:

*role-comp* (*role-comp* *x*) = *x*

**by** (*cases* *x*, *auto*)

**lemma** *role-comp-inv-eq*:

*y* = *role-comp* *x*  $\longleftrightarrow$  *x* = *role-comp* *y*

**by** (*auto elim!*: *role-comp.elims* [*OF sym*])

**definition**

*partners* :: *rid-t* *set*

**where**

*partners*  $\equiv \{R. \textit{partner-runs } \textit{test } R\}$

**lemma** *test-not-partner* [*simp*]:

*test*  $\notin$  *partners*

**by** (*auto simp add*: *partners-def partner-runs-def*, *cases* *role* (*guessed-runs* *test*), *auto*)

**lemma** *matching-symmetric*:

*matching* *sigma* *sigma'*  $\implies$  *matching* *sigma'* *sigma*

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

**lemma** *partner-symmetric*:

*partner-runs* *R* *R'*  $\implies$  *partner-runs* *R'* *R*

**by** (*auto simp add*: *partner-runs-def matching-symmetric*)

The unicity of the partner is actually protocol dependent: it only holds if there are generated fresh nonces (which identify the runs) in the frames.

**lemma** *partner-unique*:

$partner\text{-}runs\ R\ R'' \implies partner\text{-}runs\ R\ R' \implies R' = R''$

**proof** –

**assume**  $H'$ : $partner\text{-}runs\ R\ R'$

**then have**  $Hm'$ :  $matching\ (guessed\text{-}frame\ R)\ (guessed\text{-}frame\ R')$

by (*auto simp add: partner-runs-def*)

**assume**  $H''$ : $partner\text{-}runs\ R\ R''$

**then have**  $Hm''$ :  $matching\ (guessed\text{-}frame\ R)\ (guessed\text{-}frame\ R'')$

by (*auto simp add: partner-runs-def*)

**show**  $?thesis$

**proof** (*cases role (guessed-runs R')*)

**case**  $Init$

**with**  $H'$   $partner\text{-}symmetric\ [OF\ H']$  **have**  $Hrole:role\ (guessed\text{-}runs\ R) = Resp$   
 $role\ (guessed\text{-}runs\ R'') = Init$

by (*auto simp add: partner-runs-def*)

**with**  $Init\ Hm'$  **have**  $guessed\text{-}frame\ R\ xgnx = Some\ (Exp\ Gen\ (NonceF\ (R'\$nx)))$

by (*simp add: matching-def*)

**moreover from**  $Hrole\ Hm''$  **have**  $guessed\text{-}frame\ R\ xgnx = Some\ (Exp\ Gen\ (NonceF\ (R''\$nx)))$

by (*simp add: matching-def*)

**ultimately show**  $?thesis$  by (*auto dest: Exp-Gen-inj*)

**next**

**case**  $Resp$

**with**  $H'$   $partner\text{-}symmetric\ [OF\ H']$  **have**  $Hrole:role\ (guessed\text{-}runs\ R) = Init$   
 $role\ (guessed\text{-}runs\ R'') = Resp$

by (*auto simp add: partner-runs-def*)

**with**  $Resp\ Hm'$  **have**  $guessed\text{-}frame\ R\ xgny = Some\ (Exp\ Gen\ (NonceF\ (R'\$ny)))$

by (*simp add: matching-def*)

**moreover from**  $Hrole\ Hm''$  **have**  $guessed\text{-}frame\ R\ xgny = Some\ (Exp\ Gen\ (NonceF\ (R''\$ny)))$

by (*simp add: matching-def*)

**ultimately show**  $?thesis$  by (*auto dest: Exp-Gen-inj*)

**qed**

**qed**

**lemma**  $partner\text{-}test$ :

$R \in partners \implies partner\text{-}runs\ R\ R' \implies R' = test$

by (*auto intro!: partner-unique simp add: partners-def partner-symmetric*)

Compromising events.

**definition**

$l2\text{-lkr}\text{-others} :: agent \Rightarrow l2\text{-trans}$

**where**

$l2\text{-lkr}\text{-others}\ A \equiv \{(s, s')\}.$

— guards

$A \neq test\text{-}owner \wedge$

$A \neq test\text{-}partner \wedge$

— actions

$s' = s(bad := \{A\} \cup bad\ s)$

}

**definition**

$l2\text{-lkr}\text{-actor} :: agent \Rightarrow l2\text{-trans}$

**where**

$l2\text{-lkr}\text{-actor}\ A \equiv \{(s, s')\}.$

— guards

$$\begin{array}{l}
A = \text{test-owner} \wedge \\
A \neq \text{test-partner} \wedge \\
\text{— actions} \\
s' = s(\text{bad} := \{A\} \cup \text{bad } s) \\
\}
\end{array}$$

**definition**

$$l2\text{-lkr-after} :: \text{agent} \Rightarrow l2\text{-trans}$$

**where**

$$\begin{array}{l}
l2\text{-lkr-after } A \equiv \{(s, s')\}. \\
\text{— guards} \\
\text{test-ended } s \wedge \\
\text{— actions} \\
s' = s(\text{bad} := \{A\} \cup \text{bad } s) \\
\}
\end{array}$$

**definition**

$$l2\text{-skr} :: \text{rid-t} \Rightarrow \text{msg} \Rightarrow l2\text{-trans}$$

**where**

$$\begin{array}{l}
l2\text{-skr } R \ K \equiv \{(s, s')\}. \\
\text{— guards} \\
R \neq \text{test} \wedge R \notin \text{partners} \wedge \\
\text{in-progress } (\text{progress } s \ R) \ xsk \wedge \\
\text{guessed-frame } R \ xsk = \text{Some } K \wedge \\
\text{— actions} \\
s' = s(\text{ik} := \{K\} \cup \text{ik } s) \\
\}
\end{array}$$

Protocol events:

- step 1: create  $Ra$ ,  $A$  generates  $nx$ , computes and insecurely sends  $g^{nx}$
- step 2: create  $Rb$ ,  $B$  receives  $g^{nx}$  insecurely, generates  $ny$ , computes  $g^{ny}$ , authentically sends  $(g^{ny}, g^{nx})$ , computes  $g^{nx} * ny$ , emits a running signal for  $Init$ ,  $g^{nx} * ny$
- step 3:  $A$  receives  $g^{ny}$  and  $g^{nx}$  authentically, sends  $(g^{nx}, g^{ny})$  authentically, computes  $g^{ny} * nx$ , emits a commit signal for  $Init$ ,  $g^{ny} * nx$ , a running signal for  $Resp$ ,  $g^{ny} * nx$ , declares the secret  $g^{ny} * nx$
- step 4:  $B$  receives  $g^{nx}$  and  $g^{ny}$  authentically, emits a commit signal for  $Resp$ ,  $g^{nx} * ny$ , declares the secret  $g^{nx} * ny$

**definition**

$$l2\text{-step1} :: \text{rid-t} \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow l2\text{-trans}$$

**where**

$$\begin{array}{l}
l2\text{-step1 } Ra \ A \ B \equiv \{(s, s')\}. \\
\text{— guards:} \\
Ra \notin \text{dom } (\text{progress } s) \wedge \\
\text{guessed-runs } Ra = (\text{role=Init, owner=A, partner=B}) \wedge \\
\text{— actions:} \\
s' = s( \\
\quad \text{progress} := (\text{progress } s)(Ra \mapsto \{xnx, xgnx\}), \\
\quad \text{chan} := \{\text{Insec } A \ B \ (\text{Exp } \text{Gen } (\text{NonceF } (Ra\$nx)))\} \cup (\text{chan } s) \\
)
\end{array}$$

$\rangle$   
 $\}$

**definition**

$l2\text{-step2} :: \text{rid-}t \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow \text{msg} \Rightarrow l2\text{-trans}$

**where**

$l2\text{-step2 } Rb \ A \ B \ gnx \equiv \{(s, s')\}.$

— guards:

$gessed\text{-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \wedge$

$Rb \notin \text{dom } (\text{progress } s) \wedge$

$gessed\text{-frame } Rb \ xgnx = \text{Some } gnx \wedge$

$gessed\text{-frame } Rb \ xsk = \text{Some } (\text{Exp } gnx \ (\text{NonceF } (Rb\$ny))) \wedge$

$\text{Insec } A \ B \ gnx \in \text{chan } s \wedge$

— actions:

$s' = s(\text{progress} := (\text{progress } s)(Rb \mapsto \{xny, xgny, xgnx, xsk\}),$

$\text{chan} := \{\text{Auth } B \ A \ \langle \text{Number } 0, \text{Exp } \text{Gen } (\text{NonceF } (Rb\$ny)), gnx \rangle\} \cup (\text{chan } s),$

$\text{signalsInit} := \text{if can-signal } s \ A \ B \ \text{then}$

$\text{addSignal } (\text{signalsInit } s) \ (\text{Running } A \ B \ (\text{Exp } gnx \ (\text{NonceF } (Rb\$ny))))$

$\text{else}$

$\text{signalsInit } s$

$\rangle$   
 $\}$

**definition**

$l2\text{-step3} :: \text{rid-}t \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow \text{msg} \Rightarrow l2\text{-trans}$

**where**

$l2\text{-step3 } Ra \ A \ B \ gny \equiv \{(s, s')\}.$

— guards:

$gessed\text{-runs } Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \wedge$

$\text{progress } s \ Ra = \text{Some } \{xnx, xgnx\} \wedge$

$gessed\text{-frame } Ra \ xgny = \text{Some } gny \wedge$

$gessed\text{-frame } Ra \ xsk = \text{Some } (\text{Exp } gny \ (\text{NonceF } (Ra\$nx))) \wedge$

$\text{Auth } B \ A \ \langle \text{Number } 0, gny, \text{Exp } \text{Gen } (\text{NonceF } (Ra\$nx)) \rangle \in \text{chan } s \wedge$

— actions:

$s' = s(\text{progress} := (\text{progress } s)(Ra \mapsto \{xnx, xgnx, xgny, xsk, xEnd\}),$

$\text{chan} := \{\text{Auth } A \ B \ \langle \text{Number } 1, \text{Exp } \text{Gen } (\text{NonceF } (Ra\$nx)), gny \rangle\} \cup \text{chan } s,$

$\text{secret} := \{x. x = \text{Exp } gny \ (\text{NonceF } (Ra\$nx)) \wedge Ra = \text{test}\} \cup \text{secret } s,$

$\text{signalsInit} := \text{if can-signal } s \ A \ B \ \text{then}$

$\text{addSignal } (\text{signalsInit } s) \ (\text{Commit } A \ B \ (\text{Exp } gny \ (\text{NonceF } (Ra\$nx))))$

$\text{else}$

$\text{signalsInit } s,$

$\text{signalsResp} := \text{if can-signal } s \ A \ B \ \text{then}$

$\text{addSignal } (\text{signalsResp } s) \ (\text{Running } A \ B \ (\text{Exp } gny \ (\text{NonceF } (Ra\$nx))))$

$\text{else}$

$\text{signalsResp } s$

$\rangle$   
 $\}$

**definition**

$l2\text{-step4} :: \text{rid-}t \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow \text{msg} \Rightarrow l2\text{-trans}$



**where**

$l2\text{-step4 } Rb \ A \ B \ gnx \equiv \{(s, s')\}.$

— guards:

$guessed\text{-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \wedge$

$progress \ s \ Rb = \text{Some } \{xny, xgnx, xgny, xsk\} \wedge$

$guessed\text{-frame } Rb \ xgnx = \text{Some } gnx \wedge$

$Auth \ A \ B \ \langle \text{Number } 1, gnx, \text{Exp Gen } (\text{NonceF } (Rb\$ny)) \rangle \in \text{chan } s \wedge$

— actions:

$s' = s(\text{progress} := (\text{progress } s)(Rb \mapsto \{xny, xgnx, xgny, xsk, xEnd\}),$

$\text{secret} := \{x. x = \text{Exp } gnx \ (\text{NonceF } (Rb\$ny)) \wedge Rb = \text{test}\} \cup \text{secret } s,$

$\text{signalsResp} := \text{if can-signal } s \ A \ B \ \text{then}$

$\text{addSignal } (\text{signalsResp } s) \ (\text{Commit } A \ B \ (\text{Exp } gnx \ (\text{NonceF } (Rb\$ny))))$

$\text{else}$

$\text{signalsResp } s$

$\})$

$\}$

Specification.

**definition**

$l2\text{-init} :: l2\text{-state set}$

**where**

$l2\text{-init} \equiv \{ (\text{$

$ik = \{\},$

$\text{secret} = \{\},$

$\text{progress} = \text{Map.empty},$

$\text{signalsInit} = \lambda x. 0,$

$\text{signalsResp} = \lambda x. 0,$

$\text{chan} = \{\},$

$\text{bad} = \text{bad-init}$

$\})$

**definition**

$l2\text{-trans} :: l2\text{-trans where}$

$l2\text{-trans} \equiv (\bigcup m \ M \ X \ Rb \ Ra \ A \ B \ K.$

$l2\text{-step1 } Ra \ A \ B \cup$

$l2\text{-step2 } Rb \ A \ B \ X \cup$

$l2\text{-step3 } Ra \ A \ B \ X \cup$

$l2\text{-step4 } Rb \ A \ B \ X \cup$

$l2\text{-dy-fake-chan } M \cup$

$l2\text{-dy-fake-msg } m \cup$

$l2\text{-lkr-others } A \cup$

$l2\text{-lkr-after } A \cup$

$l2\text{-skr } Ra \ K \cup$

$Id$

$)$

**definition**

$l2 :: (l2\text{-state}, l2\text{-obs}) \text{ spec where}$

$l2 \equiv (\text{$

$\text{init} = l2\text{-init},$

$\text{trans} = l2\text{-trans},$

```

    obs = id
  )

```

```

lemmas l2-loc-defs =
  l2-step1-def l2-step2-def l2-step3-def l2-step4-def
  l2-def l2-init-def l2-trans-def
  l2-dy-fake-chan-def l2-dy-fake-msg-def
  l2-lkr-after-def l2-lkr-others-def l2-skr-def

```

```

lemmas l2-defs = l2-loc-defs ik-dy-def

```

```

lemmas l2-nostep-defs = l2-def l2-init-def l2-trans-def

```

```

lemma l2-obs-id [simp]: obs l2 = id
by (simp add: l2-def)

```

Once a run is finished, it stays finished, therefore if the test is not finished at some point then it was not finished before either.

```

declare domIff [iff]
lemma l2-run-ended-trans:
  run-ended (progress s R)  $\implies$ 
  (s, s')  $\in$  trans l2  $\implies$ 
  run-ended (progress s' R)
apply (auto simp add: l2-nostep-defs)
apply (auto simp add: l2-defs)
done
declare domIff [iff del]

```

```

lemma l2-can-signal-trans:
  can-signal s' A B  $\implies$ 
  (s, s')  $\in$  trans l2  $\implies$ 
  can-signal s A B
by (auto simp add: can-signal-def l2-run-ended-trans)

```

## 22.2 Invariants

### 22.2.1 inv1

If *can-signal*  $s A B$  (i.e.,  $A, B$  are the test session agents and the test is not finished), then  $A$  and  $B$  are honest.

```

definition
  l2-inv1 :: l2-state set
where
  l2-inv1  $\equiv$  {s.  $\forall A B.$ 
    can-signal s A B  $\implies$ 
    A  $\notin$  bad s  $\wedge$  B  $\notin$  bad s
  }

```

```

lemmas l2-inv1I = l2-inv1-def [THEN setc-def-to-intro, rule-format]
lemmas l2-inv1E [elim] = l2-inv1-def [THEN setc-def-to-elim, rule-format]
lemmas l2-inv1D = l2-inv1-def [THEN setc-def-to-dest, rule-format, rotated 1, simplified]

```

**lemma** *l2-inv1-init* [iff]:  
*init l2*  $\subseteq$  *l2-inv1*  
**by** (*auto simp add: l2-def l2-init-def l2-inv1-def can-signal-def bad-init-spec*)

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

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

## 22.2.2 inv2 (authentication guard)

If *Auth B A*  $\langle$ *Number 0, gny, Exp Gen (NonceF (Ra \$ nx))* $\rangle \in$  *chan s* and *A, B* are honest then the message has indeed been sent by a responder run (etc).

### definition

*l2-inv2* :: *l2-state set*

### where

*l2-inv2*  $\equiv$   $\{s. \forall Ra A B gny.$   
*Auth B A*  $\langle$ *Number 0, gny, Exp Gen (NonceF (Ra\$nx))* $\rangle \in$  *chan s*  $\longrightarrow$   
*A*  $\notin$  *bad s*  $\wedge$  *B*  $\notin$  *bad s*  $\longrightarrow$   
 $(\exists Rb. \text{guessed-runs } Rb = (\text{role}=\text{Resp, owner}=B, \text{partner}=A) \wedge$   
*in-progressS* (*progress s Rb*)  $\{xny, xgnx, xgny, xsk\} \wedge$   
*gny* = *Exp Gen (NonceF (Rb\$ny))*  $\wedge$   
*guessed-frame Rb xgnx* = *Some (Exp Gen (NonceF (Ra\$nx)))*)  
 $\}$

**lemmas** *l2-inv2I* = *l2-inv2-def* [*THEN setc-def-to-intro, rule-format*]

**lemmas** *l2-inv2E* [*elim*] = *l2-inv2-def* [*THEN setc-def-to-elim, rule-format*]

**lemmas** *l2-inv2D* = *l2-inv2-def* [*THEN setc-def-to-dest, rule-format, rotated 1, simplified*]

**lemma** *l2-inv2-init* [iff]:  
*init l2*  $\subseteq$  *l2-inv2*  
**by** (*auto simp add: l2-def l2-init-def l2-inv2-def*)

**lemma** *l2-inv2-trans* [iff]:  
*{l2-inv2}* *trans l2*  $\{>$  *l2-inv2*  
**apply** (*auto simp add: PO-hoare-defs l2-nostep-defs intro!: l2-inv2I*)

**apply** (*auto simp add: l2-defs dy-fake-chan-def*)  
**apply** *force+*  
**done**

**lemma** *PO-l2-inv2 [iff]: reach l2  $\subseteq$  l2-inv2*  
**by** (*rule inv-rule-basic*) (*auto*)

### 22.2.3 inv3 (authentication guard)

If *Auth A B*  $\langle$ Number 1, *gnx*, *Exp Gen (NonceF (Rb\$ny)) $\rangle \in$  *chan s* and *A*, *B* are honest then the message has indeed been sent by an initiator run (etc).*

**definition**

*l2-inv3* :: *l2-state set*

**where**

$$\begin{aligned} l2\text{-inv3} \equiv \{ & s. \forall Rb A B gnx. \\ & \text{Auth } A B \langle \text{Number } 1, gnx, \text{Exp Gen (NonceF (Rb\$ny))} \rangle \in \text{chan } s \longrightarrow \\ & A \notin \text{bad } s \wedge B \notin \text{bad } s \longrightarrow \\ & (\exists Ra. \text{guessed-runs } Ra = (\text{role=Init, owner=A, partner=B}) \wedge \\ & \quad \text{in-progressS (progress } s \text{ Ra) } \{xnx, xgnx, xgny, xsk, xEnd\} \wedge \\ & \quad \text{guessed-frame } Ra \text{ } xgnx = \text{Some } gnx \wedge \\ & \quad \text{guessed-frame } Ra \text{ } xgny = \text{Some (Exp Gen (NonceF (Rb\$ny)))}) \\ & \} \end{aligned}$$

**lemmas** *l2-inv3I = l2-inv3-def [THEN setc-def-to-intro, rule-format]*

**lemmas** *l2-inv3E [elim] = l2-inv3-def [THEN setc-def-to-elim, rule-format]*

**lemmas** *l2-inv3D = l2-inv3-def [THEN setc-def-to-dest, rule-format, rotated 1, simplified]*

**lemma** *l2-inv3-init [iff]:*

*init l2  $\subseteq$  l2-inv3*

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

**lemma** *l2-inv3-trans [iff]:*

$\{l2\text{-inv3}\} \text{ trans } l2 \{> l2\text{-inv3}\}$

**apply** (*auto simp add: PO-hoare-defs l2-nostep-defs intro!: l2-inv3I*)

**apply** (*auto simp add: l2-defs dy-fake-chan-def*)

**apply** (*simp-all add: domIff insert-ident*)

**apply** *force+*

**done**

**lemma** *PO-l2-inv3 [iff]: reach l2  $\subseteq$  l2-inv3*

**by** (*rule inv-rule-basic*) (*auto*)

### 22.2.4 inv4

For an initiator, the session key is always  $gny \hat{=} nx$ .

**definition**

*l2-inv4* :: *l2-state set*

**where**

$$\begin{aligned} l2\text{-inv4} \equiv \{ & s. \forall Ra A B gny. \\ & \text{guessed-runs } Ra = (\text{role=Init, owner=A, partner=B}) \longrightarrow \\ & \text{in-progress (progress } s \text{ Ra) } xsk \longrightarrow \end{aligned}$$

```

    guessed-frame Ra xgny = Some gny  $\longrightarrow$ 
    guessed-frame Ra xsk = Some (Exp gny (NonceF (Ra$nx)))
  }

```

**lemmas**  $l2\text{-inv4}I = l2\text{-inv4}\text{-def}$  [THEN setc-def-to-intro, rule-format]

**lemmas**  $l2\text{-inv4}E$  [elim] =  $l2\text{-inv4}\text{-def}$  [THEN setc-def-to-elim, rule-format]

**lemmas**  $l2\text{-inv4}D = l2\text{-inv4}\text{-def}$  [THEN setc-def-to-dest, rule-format, rotated 1, simplified]

**lemma**  $l2\text{-inv4}\text{-init}$  [iff]:

$init\ l2 \subseteq l2\text{-inv4}$

**by** (auto simp add:  $l2\text{-def}$   $l2\text{-init}\text{-def}$   $l2\text{-inv4}\text{-def}$ )

**lemma**  $l2\text{-inv4}\text{-trans}$  [iff]:

$\{l2\text{-inv4}\}$  trans  $l2 \{> l2\text{-inv4}\}$

**apply** (auto simp add: PO-hoare-defs  $l2\text{-nostep}\text{-defs}$  intro!:  $l2\text{-inv4}I$ )

**apply** (auto simp add:  $l2\text{-defs}$  dy-fake-chan-def)

**done**

**lemma**  $PO\text{-}l2\text{-inv4}$  [iff]: reach  $l2 \subseteq l2\text{-inv4}$

**by** (rule inv-rule-basic) (auto)

### 22.2.5 inv4'

For a responder, the session key is always  $gnx \hat{=} ny$ .

**definition**

$l2\text{-inv4}' :: l2\text{-state set}$

**where**

$l2\text{-inv4}' \equiv \{s. \forall Rb\ A\ B\ gnx.$

$guessed\text{-runs}\ Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \longrightarrow$

$in\text{-progress}\ (\text{progress}\ s\ Rb)\ xsk \longrightarrow$

$guessed\text{-frame}\ Rb\ xgnx = \text{Some}\ gnx \longrightarrow$

$guessed\text{-frame}\ Rb\ xsk = \text{Some}\ (\text{Exp}\ gnx\ (\text{NonceF}\ (Rb\$ny)))$

$\}$

**lemmas**  $l2\text{-inv4}'I = l2\text{-inv4}'\text{-def}$  [THEN setc-def-to-intro, rule-format]

**lemmas**  $l2\text{-inv4}'E$  [elim] =  $l2\text{-inv4}'\text{-def}$  [THEN setc-def-to-elim, rule-format]

**lemmas**  $l2\text{-inv4}'D = l2\text{-inv4}'\text{-def}$  [THEN setc-def-to-dest, rule-format, rotated 1, simplified]

**lemma**  $l2\text{-inv4}'\text{-init}$  [iff]:

$init\ l2 \subseteq l2\text{-inv4}'$

**by** (auto simp add:  $l2\text{-def}$   $l2\text{-init}\text{-def}$   $l2\text{-inv4}'\text{-def}$ )

**lemma**  $l2\text{-inv4}'\text{-trans}$  [iff]:

$\{l2\text{-inv4}'\}$  trans  $l2 \{> l2\text{-inv4}'\}$

**apply** (auto simp add: PO-hoare-defs  $l2\text{-nostep}\text{-defs}$  intro!:  $l2\text{-inv4}'I$ )

**apply** (auto simp add:  $l2\text{-defs}$  dy-fake-chan-def)

**done**

**lemma**  $PO\text{-}l2\text{-inv4}'$  [iff]: reach  $l2 \subseteq l2\text{-inv4}'$

**by** (rule inv-rule-basic) (auto)

### 22.2.6 inv5

The only confidential or secure messages on the channel have been put there by the attacker.

#### definition

$l2\text{-inv5} :: l2\text{-state set}$

#### where

$$l2\text{-inv5} \equiv \{s. \forall A B M. \\ (\text{Confid } A B M \in \text{chan } s \vee \text{Secure } A B M \in \text{chan } s) \longrightarrow \\ M \in \text{dy-fake-msg } (\text{bad } s) (\text{ik } s) (\text{chan } s) \\ \}$$

**lemmas**  $l2\text{-inv5I} = l2\text{-inv5-def}$  [*THEN setc-def-to-intro, rule-format*]

**lemmas**  $l2\text{-inv5E}$  [*elim*] =  $l2\text{-inv5-def}$  [*THEN setc-def-to-elim, rule-format*]

**lemmas**  $l2\text{-inv5D} = l2\text{-inv5-def}$  [*THEN setc-def-to-dest, rule-format, rotated 1, simplified*]

**lemma**  $l2\text{-inv5-init}$  [*iff*]:

$\text{init } l2 \subseteq l2\text{-inv5}$

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

**lemma**  $l2\text{-inv5-trans}$  [*iff*]:

$\{l2\text{-inv5}\} \text{ trans } l2 \{> l2\text{-inv5}\}$

**apply** (*auto simp add: PO-hoare-defs l2-nostep-defs intro!: l2-inv5I*)

**apply** (*auto simp add: l2-defs dy-fake-chan-def intro: l2-inv5D dy-fake-msg-monotone*)

**done**

**lemma**  $PO\text{-}l2\text{-inv5}$  [*iff*]:  $\text{reach } l2 \subseteq l2\text{-inv5}$

**by** (*rule inv-rule-basic*) (*auto*)

### 22.2.7 inv6

For a run  $R$  (with any role), the session key always has the form  $\text{something}^n$  where  $n$  is a nonce generated by  $R$ .

#### definition

$l2\text{-inv6} :: l2\text{-state set}$

#### where

$$l2\text{-inv6} \equiv \{s. \forall R. \\ \text{in-progress } (\text{progress } s R) \text{ xsk} \longrightarrow \\ (\exists X N. \\ \text{guessed-frame } R \text{ xsk} = \text{Some } (\text{Exp } X (\text{NonceF } (R\$N)))) \\ \}$$

**lemmas**  $l2\text{-inv6I} = l2\text{-inv6-def}$  [*THEN setc-def-to-intro, rule-format*]

**lemmas**  $l2\text{-inv6E}$  [*elim*] =  $l2\text{-inv6-def}$  [*THEN setc-def-to-elim, rule-format*]

**lemmas**  $l2\text{-inv6D} = l2\text{-inv6-def}$  [*THEN setc-def-to-dest, rule-format, rotated 1, simplified*]

**lemma**  $l2\text{-inv6-init}$  [*iff*]:

$\text{init } l2 \subseteq l2\text{-inv6}$

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

**lemma**  $l2\text{-inv6-trans}$  [*iff*]:

$\{l2\text{-inv6}\} \text{ trans } l2 \{> l2\text{-inv6}\}$

**apply** (*auto simp add: PO-hoare-defs l2-nostep-defs intro!: l2-inv6I*)

**apply** (*auto simp add: l2-defs dy-fake-chan-def dest: l2-inv6D*)  
**done**

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

### 22.2.8 inv7

Form of the messages in *extr* (*bad s*) (*ik s*) (*chan s*) = *synth* (*analz generators*).

#### abbreviation

$$\text{generators} \equiv \{x. \exists N. x = \text{Exp Gen (Nonce N)}\} \cup \\ \{\text{Exp } y \text{ (NonceF (R\$N))} \mid y \text{ N R. } R \neq \text{test} \wedge R \notin \text{partners}\}$$

**lemma** *analz-generators: analz generators = generators*  
**by** (*rule, rule, erule analz.induct, auto*)

#### definition

*l2-inv7* :: *l2-state set*

#### where

$$\text{l2-inv7} \equiv \{s. \\ \text{extr (bad s) (ik s) (chan s)} \subseteq \\ \text{synth (analz (generators))}\}$$

**lemmas** *l2-inv7I = l2-inv7-def [THEN setc-def-to-intro, rule-format]*

**lemmas** *l2-inv7E [elim] = l2-inv7-def [THEN setc-def-to-elim, rule-format]*

**lemmas** *l2-inv7D = l2-inv7-def [THEN setc-def-to-dest, rule-format, rotated 1, simplified]*

**lemma** *l2-inv7-init [iff]:*

*init l2  $\subseteq$  l2-inv7*

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

**lemma** *l2-inv7-step1:*

*{l2-inv7} l2-step1 Ra A B {> l2-inv7}*

**apply** (*auto simp add: PO-hoare-defs l2-defs intro!: l2-inv7I*)

**apply** (*auto intro: synth-analz-increasing*)

**done**

**lemma** *l2-inv7-step2:*

*{l2-inv7} l2-step2 Rb A B gnx {> l2-inv7}*

**apply** (*auto simp add: PO-hoare-defs l2-nostep-defs intro!: l2-inv7I, auto simp add: l2-defs*)

**apply** (*auto intro: synth-analz-increasing*)

**done**

**lemma** *l2-inv7-step3:*

*{l2-inv7} l2-step3 Ra A B gny {> l2-inv7}*

**apply** (*auto simp add: PO-hoare-defs l2-nostep-defs intro!: l2-inv7I, auto simp add: l2-defs*)

**apply** (*auto intro: synth-analz-increasing*)

**apply** (*auto dest:l2-inv7D [THEN [2] rev-subsetD] dest!:extr-Chan*)

**done**

**lemma** *l2-inv7-step4:*

$\{l2\text{-inv}7\}$   $l2\text{-step}4$   $Rb$   $A$   $B$   $gx$   $\{> l2\text{-inv}7\}$   
**by** (*auto simp add: PO-hoare-defs l2-nostep-defs intro!: l2-inv7I, auto simp add: l2-defs*)

**lemma**  $l2\text{-inv}7\text{-dy-fake-msg}$ :

$\{l2\text{-inv}7\}$   $l2\text{-dy-fake-msg}$   $M$   $\{> l2\text{-inv}7\}$

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

**lemma**  $l2\text{-inv}7\text{-dy-fake-chan}$ :

$\{l2\text{-inv}7\}$   $l2\text{-dy-fake-chan}$   $M$   $\{> l2\text{-inv}7\}$

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

**lemma**  $l2\text{-inv}7\text{-lkr-others}$ :

$\{l2\text{-inv}7 \cap l2\text{-inv}5\}$   $l2\text{-lkr-others}$   $A$   $\{> l2\text{-inv}7\}$

**apply** (*auto simp add: PO-hoare-defs l2-defs  
intro!: l2-inv7I  
dest!: extr-insert-bad [THEN [2] rev-subsetD]  
elim!: l2-inv7E l2-inv5E*)  
**apply** (*auto dest: dy-fake-msg-extr [THEN [2] rev-subsetD]*)  
**done**

**lemma**  $l2\text{-inv}7\text{-lkr-after}$ :

$\{l2\text{-inv}7 \cap l2\text{-inv}5\}$   $l2\text{-lkr-after}$   $A$   $\{> l2\text{-inv}7\}$

**apply** (*auto simp add: PO-hoare-defs l2-defs  
intro!: l2-inv7I  
dest!: extr-insert-bad [THEN [2] rev-subsetD]  
elim!: l2-inv7E l2-inv5E*)  
**apply** (*auto dest: dy-fake-msg-extr [THEN [2] rev-subsetD]*)  
**done**

**lemma**  $l2\text{-inv}7\text{-skr}$ :

$\{l2\text{-inv}7 \cap l2\text{-inv}6\}$   $l2\text{-skr}$   $R$   $K$   $\{> l2\text{-inv}7\}$

**apply** (*auto simp add: PO-hoare-defs l2-defs intro!: l2-inv7I*)  
**apply** (*auto simp add: extr-insert-IK-eq dest!: l2-inv6D*)  
**apply** (*auto intro: synth-analz-increasing*)  
**done**

**lemmas**  $l2\text{-inv}7\text{-trans-aux} =$

$l2\text{-inv}7\text{-step}1$   $l2\text{-inv}7\text{-step}2$   $l2\text{-inv}7\text{-step}3$   $l2\text{-inv}7\text{-step}4$   
 $l2\text{-inv}7\text{-dy-fake-msg}$   $l2\text{-inv}7\text{-dy-fake-chan}$   
 $l2\text{-inv}7\text{-lkr-others}$   $l2\text{-inv}7\text{-lkr-after}$   $l2\text{-inv}7\text{-skr}$

**lemma**  $l2\text{-inv}7\text{-trans}$  [*iff*]:

$\{l2\text{-inv}7 \cap l2\text{-inv}5 \cap l2\text{-inv}6\}$   $\text{trans } l2$   $\{> l2\text{-inv}7\}$

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

**lemma**  $PO\text{-}l2\text{-inv}7$  [*iff*]:  $\text{reach } l2 \subseteq l2\text{-inv}7$

**by** (*rule-tac J=l2-inv5  $\cap$  l2-inv6 in inv-rule-incr*) (*auto*)



Auxiliary dest rule for inv7.

**lemmas** *l2-inv7D-aux* =  
*l2-inv7D* [THEN [2] *subset-trans*, THEN *synth-analz-mono*, *simplified*,  
 THEN [2] *rev-subsetD*, *rotated 1*, OF *IK-subset-extr*]

### 22.2.9 inv8: form of the secrets

**definition**

*l2-inv8* :: *l2-state set*

**where**

*l2-inv8*  $\equiv$  {  
*secret s*  $\subseteq$  {*Exp (Exp Gen (NonceF (R\$N))) (NonceF (R'\$N'))* | *N N' R R'*  
*R = test  $\wedge$  R'  $\in$  partners*}  
 }

**lemmas** *l2-inv8I* = *l2-inv8-def* [THEN *setc-def-to-intro*, *rule-format*]

**lemmas** *l2-inv8E* [*elim*] = *l2-inv8-def* [THEN *setc-def-to-elim*, *rule-format*]

**lemmas** *l2-inv8D* = *l2-inv8-def* [THEN *setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

**lemma** *l2-inv8-init* [*iff*]:

*init l2*  $\subseteq$  *l2-inv8*

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

Steps 3 and 4 are the hard part.

**lemma** *l2-inv8-step3*:

{*l2-inv8*  $\cap$  *l2-inv1*  $\cap$  *l2-inv2*  $\cap$  *l2-inv4'*} *l2-step3 Ra A B gny* {> *l2-inv8*}

**proof** (*auto simp add: PO-hoare-defs intro!: l2-inv8I*)

**fix** *s s'* :: *l2-state* **fix** *x*

**assume** *Hi:s*  $\in$  *l2-inv1* *s*  $\in$  *l2-inv8* *s*  $\in$  *l2-inv2* *s*  $\in$  *l2-inv4'*

**assume** *Ht:(s, s')*  $\in$  *l2-step3 Ra A B gny*

**assume** *Hs:x*  $\in$  *secret s'*

**from** *Hs Ht* **have** *x*  $\in$  *secret s*  $\vee$  (*Ra = test*  $\wedge$  *x = Exp gny (NonceF (Ra\$nx))*)

**by** (*auto simp add: l2-defs*)

**with** *Hi Ht*

**show**  $\exists N N' R'. x = Exp (Exp Gen (NonceF (R' $ N'))) (NonceF (test $ N)) \wedge R' \in partners$

**proof** (*auto dest: l2-inv8D simp add: l2-defs*)

**assume** *Hx: x = Exp gny (NonceF (test \$ nx))*

**assume** *can-signal s A B*

**with** *Hi* **have** *HA: A*  $\notin$  *bad s*  $\wedge$  *B*  $\notin$  *bad s*

**by** *auto*

**assume** *Htest: guessed-runs test = (role = Init, owner = A, partner = B)*

*guessed-frame test xgny = Some gny*

*guessed-frame test xsk = Some (Exp gny (NonceF (test \$ nx)))*

**assume** *Auth B A* (*Number 0, gny, Exp Gen (NonceF (test \$ nx))*)  $\in$  *chan s*

**with** *Hi HA* **obtain** *Rb* **where** *HRb:*

*guessed-runs Rb = (role=Resp, owner=B, partner=A)*

*in-progressS (progress s Rb) {xny, xgnx, xgny, xsk}*

*gny = Exp Gen (NonceF (Rb\$ny))*

*guessed-frame Rb xgnx = Some (Exp Gen (NonceF (test\$nx)))*

**by** (*auto dest!: l2-inv2D*)

**with** *Hi*

**have** *guessed-frame Rb xsk = Some (Exp (Exp Gen (NonceF (Rb\$ny))) (NonceF (test\$nx)))*

by (*auto dest: l2-inv4'D*)  
**with** *HRb Htest* **have** *Rb*  $\in$  *partners*  
 by (*auto simp add: partners-def partner-runs-def matching-def*)  
**with** *HRb* **have** *Exp gny* (*NonceF* (*test* \$ *nx*)) =  
     *Exp* (*Exp Gen* (*NonceF* (*Rb* \$ *ny*))) (*NonceF* (*test* \$ *nx*))  $\wedge$  *Rb*  $\in$  *partners*  
 by *auto*  
**then show**  $\exists N N' R'$ .  
     *Exp gny* (*NonceF* (*test* \$ *nx*)) = *Exp* (*Exp Gen* (*NonceF* (*R'* \$ *N'*))) (*NonceF* (*test* \$ *N*))  $\wedge$   
     *R'*  $\in$  *partners*  
 by *blast*  
**qed** (*auto simp add: can-signal-def*)  
**qed**

**lemma** *l2-inv8-step4*:

$\{l2\text{-inv}8 \cap l2\text{-inv}1 \cap l2\text{-inv}3 \cap l2\text{-inv}4 \cap l2\text{-inv}4'\} l2\text{-step}4 \text{ } Rb \ A \ B \ gnx \ \{> \ l2\text{-inv}8\}$

**proof** (*auto simp add: PO-hoare-defs intro!: l2-inv8I*)

**fix** *s s'* :: *l2-state* **fix** *x*

**assume** *Hi:s*  $\in$  *l2-inv1* *s*  $\in$  *l2-inv8* *s*  $\in$  *l2-inv3* *s*  $\in$  *l2-inv4* *s*  $\in$  *l2-inv4'*

**assume** *Ht:(s, s')*  $\in$  *l2-step4* *Rb A B gnx*

**assume** *Hs:x*  $\in$  *secret s'*

**from** *Hs Ht* **have** *x*  $\in$  *secret s*  $\vee$  (*Rb = test*  $\wedge$  *x = Exp gnx* (*NonceF* (*Rb*\$*ny*)))

by (*auto simp add: l2-defs*)

**with** *Hi Ht*

**show**  $\exists N N' R'$ . *x = Exp* (*Exp Gen* (*NonceF* (*R'* \$ *N'*))) (*NonceF* (*test* \$ *N*))  $\wedge$  *R'*  $\in$  *partners*

**proof** (*auto dest: l2-inv8D simp add: l2-defs*)

**assume** *Hx: x = Exp gnx* (*NonceF* (*test* \$ *ny*))

**assume** *can-signal s A B*

**with** *Hi* **have** *HA: A*  $\notin$  *bad s*  $\wedge$  *B*  $\notin$  *bad s*

by *auto*

**assume** *Htest: guessed-runs test =* ( $\backslash$ *role = Resp, owner = B, partner = A*)  
     *guessed-frame test xgnx = Some gnx*

**assume** *progress s test = Some* {*xny, xgnx, xgny, xsk*}

**with** *Htest Hi* **have** *Htest': guessed-frame test xsk = Some* (*Exp gnx* (*NonceF* (*test* \$ *ny*)))

by (*auto dest: l2-inv4'D*)

**assume** *Auth A B* (*Number* (*Suc 0*), *gnx, Exp Gen* (*NonceF* (*test* \$ *ny*)))  $\in$  *chan s*

**with** *Hi HA* **obtain** *Ra* **where** *HRa*:

*guessed-runs Ra =* ( $\backslash$ *role=Init, owner=A, partner=B*)

*in-progressS* (*progress s Ra*) {*xnx, xgnx, xgny, xsk, xEnd*}

*gnx = Exp Gen* (*NonceF* (*Ra*\$*nx*))

*guessed-frame Ra xgny = Some* (*Exp Gen* (*NonceF* (*test*\$*ny*)))

by (*auto dest!: l2-inv3D*)

**with** *Hi*

**have** *guessed-frame Ra xsk = Some* (*Exp* (*Exp Gen* (*NonceF* (*Ra*\$*nx*))) (*NonceF* (*test*\$*ny*)))

by (*auto dest: l2-inv4D*)

**with** *HRa Htest Htest'* **have** *Ra*  $\in$  *partners*

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

**with** *HRa* **have** *Exp gnx* (*NonceF* (*test* \$ *ny*)) =

*Exp* (*Exp Gen* (*NonceF* (*Ra* \$ *nx*))) (*NonceF* (*test* \$ *ny*))  $\wedge$  *Ra*  $\in$  *partners*

by *auto*

**then show**  $\exists N N' R'$ .

*Exp gnx* (*NonceF* (*test* \$ *ny*)) = *Exp* (*Exp Gen* (*NonceF* (*R'* \$ *N'*))) (*NonceF* (*test* \$ *N*))  $\wedge$

*R'*  $\in$  *partners*

by *auto*

**qed** (*auto simp add: can-signal-def*)  
**qed**

**lemma** *l2-inv8-trans* [*iff*]:  
 $\{l2\text{-inv}8 \cap l2\text{-inv}1 \cap l2\text{-inv}2 \cap l2\text{-inv}3 \cap l2\text{-inv}4 \cap l2\text{-inv}4'\} \text{ trans } l2 \{> l2\text{-inv}8\}$   
**apply** (*auto simp add: l2-nostep-defs intro!: l2-inv8-step3 l2-inv8-step4*)  
**apply** (*auto simp add: PO-hoare-defs intro!: l2-inv8I*)  
**apply** (*auto simp add: l2-defs dy-fake-chan-def dest: l2-inv8D*)  
**done**

**lemma** *PO-l2-inv8* [*iff*]: *reach*  $l2 \subseteq l2\text{-inv}8$   
**by** (*rule-tac J=l2-inv1 \cap l2-inv2 \cap l2-inv3 \cap l2-inv4 \cap l2-inv4'* **in** *inv-rule-incr*) (*auto*)

Auxiliary destruction rule for inv8.

**lemma** *Exp-Exp-Gen-synth*:  
 $\text{Exp } (\text{Exp Gen } X) Y \in \text{synth } H \implies \text{Exp } (\text{Exp Gen } X) Y \in H \vee X \in \text{synth } H \vee Y \in \text{synth } H$   
**by** (*erule synth.cases, auto dest: Exp-Exp-Gen-inj2*)

**lemma** *l2-inv8-aux*:  
 $s \in l2\text{-inv}8 \implies$   
 $x \in \text{secret } s \implies$   
 $x \notin \text{synth } (\text{analz generators})$   
**apply** (*auto simp add: analz-generators dest!: l2-inv8D [THEN [2] rev-subsetD]*)  
**apply** (*auto dest!: Exp-Exp-Gen-synth Exp-Exp-Gen-inj2*)  
**done**

## 22.3 Refinement

Mediator function.

**definition**  
 $\text{med}12s :: l2\text{-obs} \Rightarrow l1\text{-obs}$   
**where**  
 $\text{med}12s t \equiv (\$   
 $\text{ik} = \text{ik } t,$   
 $\text{secret} = \text{secret } t,$   
 $\text{progress} = \text{progress } t,$   
 $\text{signalsInit} = \text{signalsInit } t,$   
 $\text{signalsResp} = \text{signalsResp } t$   
 $\left.)$

Relation between states.

**definition**  
 $R12s :: (l1\text{-state} * l2\text{-state}) \text{ set}$   
**where**  
 $R12s \equiv \{(s, s').$   
 $s = \text{med}12s s'$   
 $\}$

**lemmas**  $R12s\text{-defs} = R12s\text{-def } \text{med}12s\text{-def}$

**lemma** *can-signal-R12* [*simp*]:  
 $(s1, s2) \in R12s \implies$   
 $can\text{-}signal\ s1\ A\ B \longleftrightarrow can\text{-}signal\ s2\ A\ B$   
**by** (*auto simp add: can-signal-def R12s-defs*)

Protocol events.

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

**lemma** *l2-step2-refines-step2*:  
 $\{R12s\}\ l1\text{-}step2\ Rb\ A\ B\ gnx, l2\text{-}step2\ Rb\ A\ B\ gnx\ \{\>R12s\}$   
**by** (*auto simp add: PO-rhoare-defs R12s-defs l1-step2-def, simp-all add: l2-step2-def*)

For step3 and 4, we prove the level 1 guard, i.e., "the future session key is not in *synth* (*analz* (*ik s*))" using the fact that *inv8* also holds for the future state in which the session key is already in *secret s*.

**lemma** *l2-step3-refines-step3*:  
 $\{R12s \cap UNIV \times (l2\text{-}inv1 \cap l2\text{-}inv2 \cap l2\text{-}inv4' \cap l2\text{-}inv7 \cap l2\text{-}inv8)\}$   
 $l1\text{-}step3\ Ra\ A\ B\ gny, l2\text{-}step3\ Ra\ A\ B\ gny$   
 $\{\>R12s\}$

**proof** (*auto simp add: PO-rhoare-defs R12s-defs*)  
**fix**  $s\ s'$   
**assume**  $Hi: s \in l2\text{-}inv1\ s \in l2\text{-}inv2\ s \in l2\text{-}inv4'\ s \in l2\text{-}inv7$   
**assume**  $Ht: (s, s') \in l2\text{-}step3\ Ra\ A\ B\ gny$   
**assume**  $s \in l2\text{-}inv8$   
**with**  $Hi\ Ht\ l2\text{-}inv8\text{-}step3$  **have**  $Hi':s' \in l2\text{-}inv8$   
**by** (*auto simp add: PO-hoare-defs, blast*)  
**from**  $Ht$  **have**  $Ra = test \longrightarrow Exp\ gny\ (NonceF\ (Ra\$nx)) \in secret\ s'$   
**by** (*auto simp add: l2-defs*)  
**with**  $Hi'$  **have**  $Ra = test \longrightarrow Exp\ gny\ (NonceF\ (Ra\$nx)) \notin synth\ (analz\ generators)$   
**by** (*auto dest: l2-inv8-aux*)  
**with**  $Hi$  **have**  $G2: Ra = test \longrightarrow Exp\ gny\ (NonceF\ (Ra\$nx)) \notin synth\ (analz\ (ik\ s))$   
**by** (*auto dest!: l2-inv7D-aux*)  
**from**  $Ht\ Hi$  **have**  $G1:$   
 $can\text{-}signal\ s\ A\ B \longrightarrow (\exists\ Rb.\ guessed\text{-}runs\ Rb = (\text{role}=Resp, owner=B, partner=A) \wedge$   
 $in\text{-}progressS\ (progress\ s\ Rb)\ \{xny, xgnx, xgny, xsk\} \wedge$   
 $gny = Exp\ Gen\ (NonceF\ (Rb\$ny)) \wedge$   
 $guessed\text{-}frame\ Rb\ xgnx = Some\ (Exp\ Gen\ (NonceF\ (Ra\$nx))))$   
**by** (*auto dest!: l2-inv2D [rotated 2] simp add: l2-defs*)

**with**  $Ht\ G1\ G2$  **show**  
 $(\text{!}\ ik = ik\ s, secret = secret\ s, progress = progress\ s,$   
 $signalsInit = signalsInit\ s, signalsResp = signalsResp\ s\ \text{!}),$   
 $(\text{!}\ ik = ik\ s', secret = secret\ s', progress = progress\ s',$   
 $signalsInit = signalsInit\ s', signalsResp = signalsResp\ s'\ \text{!})$   
 $\in l1\text{-}step3\ Ra\ A\ B\ gny$   
**apply** (*auto simp add: l2-step3-def, auto simp add: l1-step3-def*)  
**apply** (*auto simp add: can-signal-def*)  
**done**

qed

**lemma** *l2-step4-refines-step4*:

$\{R12s \cap UNIV \times (l2-inv1 \cap l2-inv3 \cap l2-inv4 \cap l2-inv4' \cap l2-inv7 \cap l2-inv8)\}$   
 $l1-step4 \text{ Rb A B gn}x, l2-step4 \text{ Rb A B gn}x$   
 $\{>R12s\}$

**proof** (*auto simp add: PO-rhoare-defs R12s-defs*)

**fix**  $s \ s'$

**assume**  $Hi:s \in l2-inv1 \ s \in l2-inv3 \ s \in l2-inv4 \ s \in l2-inv4' \ s \in l2-inv7$

**assume**  $Ht: (s, s') \in l2-step4 \text{ Rb A B gn}x$

**assume**  $s \in l2-inv8$

**with**  $Hi \ Ht \ l2-inv8-step4$  **have**  $Hi':s' \in l2-inv8$

**by** (*auto simp add: PO-hoare-defs, blast*)

**from**  $Ht$  **have**  $Rb = test \longrightarrow Exp \ gn}x \ (NonceF \ (Rb\$ny)) \in secret \ s'$

**by** (*auto simp add: l2-defs*)

**with**  $Hi'$  **have**  $Rb = test \longrightarrow Exp \ gn}x \ (NonceF \ (Rb\$ny)) \notin synth \ (analz \ generators)$

**by** (*auto dest: l2-inv8-aux*)

**with**  $Hi$  **have**  $G2:Rb = test \longrightarrow Exp \ gn}x \ (NonceF \ (Rb\$ny)) \notin synth \ (analz \ (ik \ s))$

**by** (*auto dest!: l2-inv7D-aux*)

**from**  $Ht \ Hi$  **have**  $G1$ :

$can-signal \ s \ A \ B \longrightarrow (\exists \ Ra. \ guessed-runs \ Ra = (\role=Init, \ owner=A, \ partner=B)) \wedge$   
 $in-progressS \ (progress \ s \ Ra) \ \{xnx, \ xgnx, \ xgny, \ xsk, \ xEnd\} \wedge$   
 $guessed-frame \ Ra \ xgnx = Some \ gn}x \wedge$   
 $guessed-frame \ Ra \ xgny = Some \ (Exp \ Gen \ (NonceF \ (Rb\$ny)))$

**by** (*auto dest!: l2-inv3D [rotated 2] simp add: l2-defs*)

**with**  $Ht \ G1 \ G2$  **show**

$(\ () \ ik = ik \ s, \ secret = secret \ s, \ progress = progress \ s,$   
 $signalsInit = signalsInit \ s, \ signalsResp = signalsResp \ s \ ),$   
 $(\ () \ ik = ik \ s', \ secret = secret \ s', \ progress = progress \ s',$   
 $signalsInit = signalsInit \ s', \ signalsResp = signalsResp \ s' \ ))$   
 $\in l1-step4 \ \text{Rb A B gn}x$

**apply** (*auto simp add: l2-step4-def, auto simp add: l1-step4-def*)

**apply** (*auto simp add: can-signal-def*)

**done**

**qed**

Attacker events.

**lemma** *l2-dy-fake-chan-refines-skip*:

$\{R12s\} \ Id, \ l2-dy-fake-chan \ M \ \{>R12s\}$

**by** (*auto simp add: PO-rhoare-defs R12s-defs l2-defs*)

**lemma** *l2-dy-fake-msg-refines-learn*:

$\{R12s \cap UNIV \times (l2-inv7 \cap l2-inv8)\} \ l1-learn \ m, \ l2-dy-fake-msg \ m \ \{>R12s\}$

**apply** (*auto simp add: PO-rhoare-defs R12s-defs l2-loc-defs l1-defs*)

**apply** (*drule Fake-insert-dy-fake-msg, erule l2-inv7D*)

**apply** (*auto dest!: l2-inv8-aux*)

**done**

Compromising events.

**lemma** *l2-lkr-others-refines-skip*:

$\{R12s\} \ Id, \ l2-lkr-others \ A \ \{>R12s\}$

**by** (*auto simp add: PO-rhoare-defs R12s-defs l2-loc-defs l1-defs*)

**lemma** *l2-lkr-after-refines-skip*:

$\{R12s\}$  *Id*, *l2-lkr-after A*  $\{>R12s\}$

**by** (*auto simp add: PO-rhoare-defs R12s-defs l2-loc-defs l1-defs*)

**lemma** *l2-skr-refines-learn*:

$\{R12s \cap UNIV \times l2-inv7 \cap UNIV \times l2-inv6 \cap UNIV \times l2-inv8\}$  *l1-learn K*, *l2-skr R K*  $\{>R12s\}$

**proof** (*auto simp add: PO-rhoare-defs R12s-defs l2-loc-defs l1-defs*)

**fix** *s* :: *l2-state* **fix** *x*

**assume** *H*:*s*  $\in$  *l2-inv7* *s*  $\in$  *l2-inv6*

*R*  $\notin$  *partners* *R*  $\neq$  *test in-progress* (*progress s R*) *xsk* *guessed-frame R xsk* = *Some K*

**assume** *Hx*:*x*  $\in$  *synth* (*analz* (*insert K* (*ik s*)))

**assume** *x*  $\in$  *secret s* *s*  $\in$  *l2-inv8*

**then obtain** *R R' N N'* **where** *Hx'*:*x* = *Exp* (*Exp Gen* (*NonceF* (*R\$N*))) (*NonceF* (*R'\$N'*))

*R* = *test*  $\wedge$  *R'*  $\in$  *partners*

**by** (*auto dest!: l2-inv8D subsetD*)

**from** *H* **have** *s* (*ik* := *insert K* (*ik s*))  $\in$  *l2-inv7*

**by** (*auto intro: hoare-apply [OF l2-inv7-skr] simp add: l2-defs*)

**with** *Hx* **have** *x*  $\in$  *synth* (*analz* (*generators*))

**by** (*auto dest: l2-inv7D-aux*)

**with** *Hx'* **show** *False*

**by** (*auto dest!: Exp-Exp-Gen-synth dest: Exp-Exp-Gen-inj2 simp add: analz-generators*)

**qed**

Refinement proof.

**lemmas** *l2-trans-refines-l1-trans* =

*l2-dy-fake-msg-refines-learn l2-dy-fake-chan-refines-skip*

*l2-lkr-others-refines-skip l2-lkr-after-refines-skip l2-skr-refines-learn*

*l2-step1-refines-step1 l2-step2-refines-step2 l2-step3-refines-step3 l2-step4-refines-step4*

**lemma** *l2-refines-init-l1* [*iff*]:

*init l2*  $\subseteq$  *R12s* “ (*init l1*)

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

**lemma** *l2-refines-trans-l1* [*iff*]:

$\{R12s \cap (UNIV \times (l2-inv1 \cap l2-inv2 \cap l2-inv3 \cap l2-inv4 \cap l2-inv4' \cap$   
 $l2-inv6 \cap l2-inv7 \cap l2-inv8))\}$

*trans l1*, *trans l2*

$\{> R12s\}$

**by** (*auto 0 3 simp add: l1-def l2-def l1-trans-def l2-trans-def*

*intro: l2-trans-refines-l1-trans*)

**lemma** *PO-obs-consistent-R12s* [*iff*]:

*obs-consistent R12s med12s l1 l2*

**by** (*auto simp add: obs-consistent-def R12s-def med12s-def l2-defs*)

**lemma** *l2-refines-l1* [*iff*]:

*refines*

(*R12s*  $\cap$

(*reach l1*  $\times$  (*l2-inv1*  $\cap$  *l2-inv2*  $\cap$  *l2-inv3*  $\cap$  *l2-inv4*  $\cap$  *l2-inv4'*  $\cap$  *l2-inv5*  $\cap$   
 $l2-inv6 \cap l2-inv7 \cap l2-inv8)))$

*med12s l1 l2*

by (rule Refinement-using-invariants, auto)

**lemma** *l2-implements-l1* [iff]:

*implements med12s l1 l2*

by (rule refinement-soundness) (auto)

## 22.4 Derived invariants

We want to prove *l2-secrecy*:  $dy\text{-fake-msg } (bad\ s) (ik\ s) (chan\ s) \cap secret\ s = \{\}$  but by refinement we only get *l2-partial-secrecy*:  $synth\ (analz\ (ik\ s)) \cap secret\ s = \{\}$  This is fine, since a message in  $dy\text{-fake-msg } (bad\ s) (ik\ s) (chan\ s)$  could be added to  $ik\ s$ , and *l2-partial-secrecy* would still hold for this new state.

**definition**

*l2-partial-secrecy* :: ('a l2-state-scheme) set

**where**

*l2-partial-secrecy*  $\equiv \{s.\ synth\ (analz\ (ik\ s)) \cap secret\ s = \{\}\}$

**lemma** *l2-obs-partial-secrecy* [iff]: *oreach*  $l2 \subseteq l2\text{-partial-secrecy}$

**apply** (rule external-invariant-translation

[OF *l1-obs-secrecy* - *l2-implements-l1*])

**apply** (auto simp add: *med12s-def* *s0-secrecy-def* *l2-partial-secrecy-def*)

**done**

**lemma** *l2-oreach-dy-fake-msg*:

$\llbracket s \in \text{oreach } l2; x \in dy\text{-fake-msg } (bad\ s) (ik\ s) (chan\ s) \rrbracket$

$\implies s \llbracket ik := insert\ x\ (ik\ s) \rrbracket \in \text{oreach } l2$

**apply** (auto simp add: *oreach-def*, rule, *simp-all*,

*simp add: l2-def l2-trans-def l2-dy-fake-msg-def*)

**apply** *blast*

**done**

**definition**

*l2-secrecy* :: ('a l2-state-scheme) set

**where**

*l2-secrecy*  $\equiv \{s.\ dy\text{-fake-msg } (bad\ s) (ik\ s) (chan\ s) \cap secret\ s = \{\}\}$

**lemma** *l2-obs-secrecy* [iff]: *oreach*  $l2 \subseteq l2\text{-secrecy}$

**apply** (auto simp add: *l2-secrecy-def*)

**apply** (drule *l2-oreach-dy-fake-msg*, *simp-all*)

**apply** (drule *l2-obs-partial-secrecy* [THEN [2] *rev-subsetD*], *simp add: l2-partial-secrecy-def*)

**apply** *blast*

**done**

**lemma** *l2-secrecy* [iff]: *reach*  $l2 \subseteq l2\text{-secrecy}$

by (rule external-to-internal-invariant [OF *l2-obs-secrecy*], auto)

**abbreviation** *l2-iagreement-Init*  $\equiv l1\text{-iagreement-Init}$

**lemma** *l2-obs-iagreement-Init* [iff]: oreach  $l2 \subseteq l2\text{-iagreement-Init}$   
**apply** (rule *external-invariant-translation*  
 [OF *l1-obs-iagreement-Init - l2-implements-l1*])  
**apply** (auto simp add: *med12s-def l1-iagreement-Init-def*)  
**done**

**lemma** *l2-iagreement-Init* [iff]: reach  $l2 \subseteq l2\text{-iagreement-Init}$   
**by** (rule *external-to-internal-invariant* [OF *l2-obs-iagreement-Init*], auto)

**abbreviation** *l2-iagreement-Resp*  $\equiv$  *l1-iagreement-Resp*

**lemma** *l2-obs-iagreement-Resp* [iff]: oreach  $l2 \subseteq l2\text{-iagreement-Resp}$   
**apply** (rule *external-invariant-translation*  
 [OF *l1-obs-iagreement-Resp - l2-implements-l1*])  
**apply** (auto simp add: *med12s-def l1-iagreement-Resp-def*)  
**done**

**lemma** *l2-iagreement-Resp* [iff]: reach  $l2 \subseteq l2\text{-iagreement-Resp}$   
**by** (rule *external-to-internal-invariant* [OF *l2-obs-iagreement-Resp*], auto)

**end**



## 23 Authenticated Diffie-Hellman Protocol (L3 locale)

```
theory dhlvl3
imports dhlvl2 Implem-lemmas
begin
```

### 23.1 State and Events

Level 3 state.

(The types have to be defined outside the locale.)

```
record l3-state = l1-state +
  bad :: agent set
```

```
type-synonym l3-obs = l3-state
```

```
type-synonym
  l3-pred = l3-state set
```

```
type-synonym
  l3-trans = (l3-state × l3-state) set
```

Attacker event.

```
definition
  l3-dy :: msg ⇒ l3-trans
where
  l3-dy ≡ ik-dy
```

Compromise events.

```
definition
  l3-lkr-others :: agent ⇒ l3-trans
where
  l3-lkr-others A ≡ {(s,s').
    — guards
    A ≠ test-owner ∧
    A ≠ test-partner ∧
    — actions
    s' = s(bad := {A} ∪ bad s,
           ik := keys-of A ∪ ik s)
  }
```

```
definition
  l3-lkr-actor :: agent ⇒ l3-trans
where
  l3-lkr-actor A ≡ {(s,s').
    — guards
    A = test-owner ∧
    A ≠ test-partner ∧
    — actions
    s' = s(bad := {A} ∪ bad s,
           ik := keys-of A ∪ ik s)
  }
```

**definition**

$$l3-lkr\text{-after} :: \text{agent} \Rightarrow l3\text{-trans}$$
**where**

$$l3-lkr\text{-after } A \equiv \{(s, s') .$$

— guards  
 $test\text{-ended } s \wedge$   
 — actions  
 $s' = s(\text{bad} := \{A\} \cup \text{bad } s,$   
 $ik := \text{keys-of } A \cup ik \ s)$   
 $\}$

**definition**

$$l3-skr :: \text{rid-t} \Rightarrow \text{msg} \Rightarrow l3\text{-trans}$$
**where**

$$l3-skr \ R \ K \equiv \{(s, s') .$$

— guards  
 $R \neq test \wedge R \notin \text{partners} \wedge$   
 $in\text{-progress } (\text{progress } s \ R) \ xsk \wedge$   
 $guessed\text{-frame } R \ xsk = \text{Some } K \wedge$   
 — actions  
 $s' = s(ik := \{K\} \cup ik \ s)$   
 $\}$

New locale for the level 3 protocol. This locale does not add new assumptions, it is only used to separate the level 3 protocol from the implementation locale.

**locale**  $dhlvl3 = \text{valid-implem}$

**begin**

Protocol events:

- step 1: create  $Ra$ ,  $A$  generates  $nx$ , computes and insecurely sends  $g^{nx}$
- step 2: create  $Rb$ ,  $B$  receives  $g^{nx}$  insecurely, generates  $ny$ , computes  $g^{ny}$ , authentically sends  $(g^{ny}, g^{nx})$ , computes  $g^{nx} * ny$ , emits a running signal for  $Init$ ,  $g^{nx} * ny$
- step 3:  $A$  receives  $g^{ny}$  and  $g^{nx}$  authentically, sends  $(g^{nx}, g^{ny})$  authentically, computes  $g^{ny} * nx$ , emits a commit signal for  $Init$ ,  $g^{ny} * nx$ , a running signal for  $Resp$ ,  $g^{ny} * nx$ , declares the secret  $g^{ny} * nx$
- step 4:  $B$  receives  $g^{nx}$  and  $g^{ny}$  authentically, emits a commit signal for  $Resp$ ,  $g^{nx} * ny$ , declares the secret  $g^{nx} * ny$

**definition**

$$l3\text{-step1} :: \text{rid-t} \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow l3\text{-trans}$$
**where**

$$l3\text{-step1 } Ra \ A \ B \equiv \{(s, s') .$$

— guards:  
 $Ra \notin \text{dom } (\text{progress } s) \wedge$   
 $guessed\text{-runs } Ra = (\text{role}=Init, \text{owner}=A, \text{partner}=B) \wedge$   
 — actions:  
 $s' = s(\$

```

    progress := (progress s)(Ra ↦ {xnx, xgnx}),
    ik := {implInsec A B (Exp Gen (NonceF (Ra$nx)))} ∪ ik s
  )
}

```

**definition**

*l3-step2* :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *msg* ⇒ *l3-trans*

**where**

*l3-step2* *Rb A B gnx* ≡ {(*s*, *s'*).

— guards:

*guessed-runs Rb* = (|*role=Resp*, *owner=B*, *partner=A*|) ∧

*Rb* ∉ *dom* (*progress s*) ∧

*guessed-frame Rb xgnx* = *Some gnx* ∧

*guessed-frame Rb xsk* = *Some (Exp gnx (NonceF (Rb\$ny)))* ∧

*implInsec A B gnx* ∈ *ik s* ∧

— actions:

*s'* = *s*(| *progress* := (*progress s*)(*Rb* ↦ {*xny*, *xgny*, *xgnx*, *xsk*}),

*ik* := {*implAuth B A* ⟨*Number 0*, *Exp Gen (NonceF (Rb\$ny))*, *gnx*⟩} ∪ *ik s*,

*signalsInit* := *if can-signal s A B then*

*addSignal (signalsInit s) (Running A B (Exp gnx (NonceF (Rb\$ny))))*

*else*

*signalsInit s*

)

}

**definition**

*l3-step3* :: *rid-t* ⇒ *agent* ⇒ *agent* ⇒ *msg* ⇒ *l3-trans*

**where**

*l3-step3* *Ra A B gny* ≡ {(*s*, *s'*).

— guards:

*guessed-runs Ra* = (|*role=Init*, *owner=A*, *partner=B*|) ∧

*progress s Ra* = *Some {xnx, xgnx}* ∧

*guessed-frame Ra xgny* = *Some gny* ∧

*guessed-frame Ra xsk* = *Some (Exp gny (NonceF (Ra\$nx)))* ∧

*implAuth B A* ⟨*Number 0*, *gny*, *Exp Gen (NonceF (Ra\$nx))*⟩ ∈ *ik s* ∧

— actions:

*s'* = *s*(| *progress* := (*progress s*)(*Ra* ↦ {*xnx*, *xgnx*, *xgny*, *xsk*, *xEnd*}),

*ik* := {*implAuth A B* ⟨*Number 1*, *Exp Gen (NonceF (Ra\$nx))*, *gny*⟩} ∪ *ik s*,

*secret* := {*x*. *x* = *Exp gny (NonceF (Ra\$nx))* ∧ *Ra* = *test*} ∪ *secret s*,

*signalsInit* := *if can-signal s A B then*

*addSignal (signalsInit s) (Commit A B (Exp gny (NonceF (Ra\$nx))))*

*else*

*signalsInit s*,

*signalsResp* := *if can-signal s A B then*

*addSignal (signalsResp s) (Running A B (Exp gny (NonceF (Ra\$nx))))*

*else*

*signalsResp s*

)

}

**definition**

$l3\text{-step4} :: \text{rid-}t \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow \text{msg} \Rightarrow l3\text{-trans}$

**where**

$l3\text{-step4} \text{ Rb } A \text{ B } \text{gnx} \equiv \{(s, s')\}.$

— guards:

$\text{guessed-runs } \text{Rb} = (\text{role}=\text{Resp}, \text{owner}=\text{B}, \text{partner}=\text{A}) \wedge$

$\text{progress } s \text{ Rb} = \text{Some } \{xny, xgnx, xgny, xsk\} \wedge$

$\text{guessed-frame } \text{Rb } xgnx = \text{Some } \text{gnx} \wedge$

$\text{implAuth } A \text{ B } \langle \text{Number } 1, \text{gnx}, \text{Exp Gen (NonceF (Rb\$ny))} \rangle \in \text{ik } s \wedge$

— actions:

$s' = s(\text{progress} := (\text{progress } s)(\text{Rb} \mapsto \{xny, xgnx, xgny, xsk, xEnd\}),$

$\text{secret} := \{x. x = \text{Exp gn}x (\text{NonceF (Rb\$ny)}) \wedge \text{Rb} = \text{test}\} \cup \text{secret } s,$

$\text{signalsResp} := \text{if can-signal } s \text{ A B then}$

$\text{addSignal (signalsResp } s) (\text{Commit } A \text{ B (Exp gn}x (\text{NonceF (Rb\$ny)})))$

$\text{else}$

$\text{signalsResp } s$

$\})$

$\}$

Specification.

Initial compromise.

**definition**

$\text{ik-init} :: \text{msg set}$

**where**

$\text{ik-init} \equiv \{\text{priK } C \mid C. C \in \text{bad-init}\} \cup \{\text{pubK } A \mid A. \text{True}\} \cup$

$\{\text{shrK } A \text{ B} \mid A \text{ B}. A \in \text{bad-init} \vee B \in \text{bad-init}\} \cup \text{Tags}$

Lemmas about  $\text{ik-init}$ .

**lemma**  $\text{parts-ik-init [simp]}$ :  $\text{parts ik-init} = \text{ik-init}$

**by** ( $\text{auto elim!}$ :  $\text{parts.induct}$ ,  $\text{auto simp add}$ :  $\text{ik-init-def}$ )

**lemma**  $\text{analz-ik-init [simp]}$ :  $\text{analz ik-init} = \text{ik-init}$

**by** ( $\text{auto dest}$ :  $\text{analz-into-parts}$ )

**lemma**  $\text{abs-ik-init [iff]}$ :  $\text{abs ik-init} = \{\}$

**apply** ( $\text{auto elim!}$ :  $\text{absE}$ )

**apply** ( $\text{auto simp add}$ :  $\text{ik-init-def}$ )

**done**

**lemma**  $\text{payloadSet-ik-init [iff]}$ :  $\text{ik-init} \cap \text{payload} = \{\}$

**by** ( $\text{auto simp add}$ :  $\text{ik-init-def}$ )

**lemma**  $\text{validSet-ik-init [iff]}$ :  $\text{ik-init} \cap \text{valid} = \{\}$

**by** ( $\text{auto simp add}$ :  $\text{ik-init-def}$ )

**definition**

$l3\text{-init} :: l3\text{-state set}$

**where**

$l3\text{-init} \equiv \{$

$\text{ik} = \text{ik-init},$

$\text{secret} = \{\},$

```

progress = Map.empty,
signalsInit = λx. 0,
signalsResp = λx. 0,
bad = bad-init
})

```

**lemmas**  $l3\text{-init-defs} = l3\text{-init-def } ik\text{-init-def}$

**definition**

$l3\text{-trans} :: l3\text{-trans}$

**where**

```

l3-trans ≡ (⋃ M X Rb Ra A B K.
  l3-step1 Ra A B ∪
  l3-step2 Rb A B X ∪
  l3-step3 Ra A B X ∪
  l3-step4 Rb A B X ∪
  l3-dy M ∪
  l3-lkr-others A ∪
  l3-lkr-after A ∪
  l3-skr Ra K ∪
  Id
)

```

**definition**

$l3 :: (l3\text{-state}, l3\text{-obs}) \text{ spec}$  **where**

```

l3 ≡ ()
  init = l3-init,
  trans = l3-trans,
  obs = id
)

```

**lemmas**  $l3\text{-loc-defs} =$

```

l3-step1-def l3-step2-def l3-step3-def l3-step4-def
l3-def l3-init-defs l3-trans-def
l3-dy-def
l3-lkr-others-def l3-lkr-after-def l3-skr-def

```

**lemmas**  $l3\text{-defs} = l3\text{-loc-defs } ik\text{-dy-def}$

**lemmas**  $l3\text{-nostep-defs} = l3\text{-def } l3\text{-init-def } l3\text{-trans-def}$

**lemma**  $l3\text{-obs-id}$  [simp]:  $obs\ l3 = id$

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

## 23.2 Invariants

### 23.2.1 inv1: No long-term keys as message parts

**definition**

$l3\text{-inv1} :: l3\text{-state } set$

**where**

$l3\text{-inv1} ≡ \{s.$

parts (ik s)  $\cap$  range LtK  $\subseteq$  ik s  
}

**lemmas** l3-inv1I = l3-inv1-def [THEN setc-def-to-intro, rule-format]

**lemmas** l3-inv1E [elim] = l3-inv1-def [THEN setc-def-to-elim, rule-format]

**lemmas** l3-inv1D = l3-inv1-def [THEN setc-def-to-dest, rule-format]

**lemma** l3-inv1D' [dest]:  $\llbracket$  LtK K  $\in$  parts (ik s); s  $\in$  l3-inv1  $\rrbracket \implies$  LtK K  $\in$  ik s  
**by** (auto simp add: l3-inv1-def)

**lemma** l3-inv1-init [iff]:

init l3  $\subseteq$  l3-inv1

**by** (auto simp add: l3-def l3-init-def intro!:l3-inv1I)

**lemma** l3-inv1-trans [iff]:

{l3-inv1} trans l3  $\{>$  l3-inv1}

**apply** (auto simp add: PO-hoare-defs l3-nostep-defs intro!: l3-inv1I)

**apply** (auto simp add: l3-defs dy-fake-msg-def dy-fake-chan-def

parts-insert [where H=ik -] parts-insert [where H=insert - (ik -)])

dest!: Fake-parts-insert)

**apply** (auto dest:analz-into-parts)

**done**

**lemma** PO-l3-inv1 [iff]:

reach l3  $\subseteq$  l3-inv1

**by** (rule inv-rule-basic) (auto)

### 23.2.2 inv2: l3-state.bad s indeed contains "bad" keys

**definition**

l3-inv2 :: l3-state set

**where**

l3-inv2  $\equiv$  {s.

Keys-bad (ik s) (bad s)

}

**lemmas** l3-inv2I = l3-inv2-def [THEN setc-def-to-intro, rule-format]

**lemmas** l3-inv2E [elim] = l3-inv2-def [THEN setc-def-to-elim, rule-format]

**lemmas** l3-inv2D = l3-inv2-def [THEN setc-def-to-dest, rule-format]

**lemma** l3-inv2-init [simp,intro!]:

init l3  $\subseteq$  l3-inv2

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

**lemma** l3-inv2-trans [simp,intro!]:

{l3-inv2  $\cap$  l3-inv1} trans l3  $\{>$  l3-inv2}

**apply** (auto simp add: PO-hoare-defs l3-nostep-defs intro!: l3-inv2I)

**apply** (auto simp add: l3-defs dy-fake-msg-def dy-fake-chan-def)

4 subgoals: dy, lkr\*, skr

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

**apply** (auto intro!: Keys-bad-insert-payload)

done

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

### 23.2.3 inv3

If a message can be analyzed from the intruder knowledge then it can be derived (using *synth/analz*) from the sets of implementation, non-implementation, and long-term key messages and the tags. That is, intermediate messages are not needed.

**definition**

*l3-inv3* :: *l3-state set*

**where**

*l3-inv3*  $\equiv$  {  
  *analz (ik s)*  $\subseteq$   
  *synth (analz ((ik s  $\cap$  payload)  $\cup$  ((ik s)  $\cap$  valid)  $\cup$  (ik s  $\cap$  range LtK)  $\cup$  Tags))*  
}

**lemmas** *l3-inv3I* = *l3-inv3-def* [*THEN setc-def-to-intro, rule-format*]

**lemmas** *l3-inv3E* = *l3-inv3-def* [*THEN setc-def-to-elim, rule-format*]

**lemmas** *l3-inv3D* = *l3-inv3-def* [*THEN setc-def-to-dest, rule-format*]

**lemma** *l3-inv3-init* [*iff*]:

*init l3*  $\subseteq$  *l3-inv3*

**apply** (*auto simp add: l3-def l3-init-def intro!: l3-inv3I*)

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

done

**declare** *domIff* [*iff del*]

Most of the cases in this proof are simple and very similar. The proof could probably be shortened.

**lemma** *l3-inv3-trans* [*simp,intro!*]:

{*l3-inv3*} *trans l3* { $>$  *l3-inv3*}

**proof** (*simp add: l3-nostep-defs, safe*)

**fix** *Ra A B*

**show** {*l3-inv3*} *l3-step1 Ra A B* { $>$  *l3-inv3*}

**apply** (*auto simp add: PO-hoare-def l3-defs intro!: l3-inv3I dest!: l3-inv3D*)

**apply** (*auto intro!: validI dest!: analz-insert-partition [THEN [2] rev-subsetD]*)

done

**next**

**fix** *Rb A B gnx*

**show** {*l3-inv3*} *l3-step2 Rb A B gnx* { $>$  *l3-inv3*}

**apply** (*auto simp add: PO-hoare-def l3-defs intro!: l3-inv3I dest!: l3-inv3D*)

**apply** (*auto intro!: validI dest!: analz-insert-partition [THEN [2] rev-subsetD]*)

done

**next**

**fix** *Ra A B gny*

**show** {*l3-inv3*} *l3-step3 Ra A B gny* { $>$  *l3-inv3*}

**apply** (*auto simp add: PO-hoare-def l3-defs intro!: l3-inv3I dest!: l3-inv3D*)

**apply** (*auto intro!: validI dest!: analz-insert-partition [THEN [2] rev-subsetD]*)

done

```

next
  fix Rb A B gnx
  show {l3-inv3} l3-step4 Rb A B gnx {> l3-inv3}
  apply (auto simp add: PO-hoare-def l3-defs intro!: l3-inv3I dest!: l3-inv3D)
  done
next
  fix m
  show {l3-inv3} l3-dy m {> l3-inv3}
  apply (auto simp add: PO-hoare-def l3-defs dy-fake-chan-def dy-fake-msg-def
    intro!: l3-inv3I dest!: l3-inv3D)
  apply (drule synth-analz-insert)
  apply (blast intro: synth-analz-monotone dest: synth-monotone)
  done
next
  fix A
  show {l3-inv3} l3-lkr-others A {> l3-inv3}
  apply (auto simp add: PO-hoare-def l3-defs intro!: l3-inv3I dest!: l3-inv3D)
  apply (drule analz-Un-partition [of - keys-of A], auto)
  done
next
  fix A
  show {l3-inv3} l3-lkr-after A {> l3-inv3}
  apply (auto simp add: PO-hoare-def l3-defs intro!: l3-inv3I dest!: l3-inv3D)
  apply (drule analz-Un-partition [of - keys-of A], auto)
  done
next
  fix R K
  show {l3-inv3} l3-skr R K {> l3-inv3}
  apply (auto simp add: PO-hoare-def l3-defs intro!: l3-inv3I dest!: l3-inv3D)
  apply (auto dest!: analz-insert-partition [THEN [2] rev-subsetD])
  done
qed

```

**lemma** *PO-l3-inv3* [iff]:  $\text{reach } l3 \subseteq l3\text{-inv3}$   
**by** (rule *inv-rule-basic*) (auto)

### 23.2.4 inv4: the intruder knows the tags

**definition**

*l3-inv4* :: *l3-state set*

**where**

$l3\text{-inv4} \equiv \{s.$   
 $\text{Tags} \subseteq ik\ s$   
 $\}$

**lemmas** *l3-inv4I* = *l3-inv4-def* [THEN *setc-def-to-intro*, *rule-format*]

**lemmas** *l3-inv4E* [*elim*] = *l3-inv4-def* [THEN *setc-def-to-elim*, *rule-format*]

**lemmas** *l3-inv4D* = *l3-inv4-def* [THEN *setc-def-to-dest*, *rule-format*]

**lemma** *l3-inv4-init* [*simp,intro!*]:

$\text{init } l3 \subseteq l3\text{-inv4}$

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



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

**lemma** *PO-l3-inv4* [*simp,intro!*]:  $\text{reach } l3 \subseteq l3\text{-inv4}$   
**by** (*rule inv-rule-basic*) (*auto*)

The remaining invariants are derived from the others. They are not protocol dependent provided the previous invariants hold.

### 23.2.5 inv5

The messages that the L3 DY intruder can derive from the intruder knowledge (using *synth/analz*), are either implementations or intermediate messages or can also be derived by the L2 intruder from the set *extr* (*l3-state.bad s*) (*ik s*  $\cap$  *payload*) (*local.abs (ik s)*), that is, given the non-implementation messages and the abstractions of (implementation) messages in the intruder knowledge.

#### definition

*l3-inv5* :: *l3-state set*

#### where

$l3\text{-inv5} \equiv \{s.$   
 $\text{synth } (\text{analz } (\text{ik } s)) \subseteq$   
 $\text{dy-fake-msg } (\text{bad } s) (\text{ik } s \cap \text{payload}) (\text{abs } (\text{ik } s)) \cup \text{-payload}$   
 $\}$

**lemmas** *l3-inv5I* = *l3-inv5-def* [*THEN setc-def-to-intro, rule-format*]

**lemmas** *l3-inv5E* = *l3-inv5-def* [*THEN setc-def-to-elim, rule-format*]

**lemmas** *l3-inv5D* = *l3-inv5-def* [*THEN setc-def-to-dest, rule-format*]

**lemma** *l3-inv5-derived*:  $l3\text{-inv2} \cap l3\text{-inv3} \subseteq l3\text{-inv5}$

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

**lemma** *PO-l3-inv5* [*simp,intro!*]:  $\text{reach } l3 \subseteq l3\text{-inv5}$

**using** *l3-inv5-derived PO-l3-inv2 PO-l3-inv3*

**by** *blast*

### 23.2.6 inv6

If the level 3 intruder can deduce a message implementing an insecure channel message, then:

- either the message is already in the intruder knowledge;
- or the message is constructed, and the payload can also be deduced by the intruder.

#### definition

*l3-inv6* :: *l3-state set*

#### where

```

l3-inv6 ≡ {s. ∀ A B M.
  (implInsec A B M ∈ synth (analz (ik s))) ∧ M ∈ payload) →
  (implInsec A B M ∈ ik s ∨ M ∈ synth (analz (ik s)))
}

```

```

lemmas l3-inv6I = l3-inv6-def [THEN setc-def-to-intro, rule-format]
lemmas l3-inv6E = l3-inv6-def [THEN setc-def-to-elim, rule-format]
lemmas l3-inv6D = l3-inv6-def [THEN setc-def-to-dest, rule-format]

```

**lemma** *l3-inv6-derived* [*simp,intro!*]:

*l3-inv3* ∩ *l3-inv4* ⊆ *l3-inv6*

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

1 subgoal

```

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

```

**lemma** *PO-l3-inv6* [*simp,intro!*]: *reach l3* ⊆ *l3-inv6*

**using** *l3-inv6-derived PO-l3-inv3 PO-l3-inv4*

**by** (*blast*)

### 23.2.7 inv7

If the level 3 intruder can deduce a message implementing a confidential channel message, then either

- the message is already in the intruder knowledge, or
- the message is constructed, and the payload can also be deduced by the intruder.

**definition**

*l3-inv7* :: *l3-state set*

**where**

```

l3-inv7 ≡ {s. ∀ A B M.
  (implConfid A B M ∈ synth (analz (ik s))) ∧ M ∈ payload) →
  (implConfid A B M ∈ ik s ∨ M ∈ synth (analz (ik s)))
}

```

**lemmas** *l3-inv7I* = *l3-inv7-def* [*THEN setc-def-to-intro, rule-format*]

**lemmas** *l3-inv7E* = *l3-inv7-def* [*THEN setc-def-to-elim, rule-format*]

**lemmas** *l3-inv7D* = *l3-inv7-def* [*THEN setc-def-to-dest, rule-format*]

**lemma** *l3-inv7-derived* [*simp,intro!*]:

*l3-inv3* ∩ *l3-inv4* ⊆ *l3-inv7*

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

1 subgoal

```

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

```

done

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

### 23.2.8 inv8

If the level 3 intruder can deduce a message implementing an authentic channel message then either

- the message is already in the intruder knowledge, or
- the message is constructed, and in this case the payload can also be deduced by the intruder, and one of the agents is bad.

**definition**

*l3-inv8* :: *l3-state set*

**where**

$l3-inv8 \equiv \{s. \forall A B M.$   
   $(implAuth A B M \in synth (analz (ik s)) \wedge M \in payload) \longrightarrow$   
   $(implAuth A B M \in ik s \vee (M \in synth (analz (ik s)) \wedge (A \in bad s \vee B \in bad s)))$   
}

**lemmas** *l3-inv8I* = *l3-inv8-def* [*THEN setc-def-to-intro, rule-format*]

**lemmas** *l3-inv8E* = *l3-inv8-def* [*THEN setc-def-to-elim, rule-format*]

**lemmas** *l3-inv8D* = *l3-inv8-def* [*THEN setc-def-to-dest, rule-format*]

**lemma** *l3-inv8-derived* [*iff*]:

$l3-inv2 \cap l3-inv3 \cap l3-inv4 \subseteq l3-inv8$

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

2 subgoals: M is deducible and the agents are bad

**apply** (*drule synth-mono, simp, drule subsetD, assumption*)

**apply** (*auto dest!*: *implAuth-synth-analz* [*rotated 1, where H=-  $\cup$  -*] *elim!*: *synth-analz-monotone*)

**apply** (*drule synth-mono, simp, drule subsetD, assumption*)

**apply** (*auto dest!*: *implAuth-synth-analz* [*rotated 1, where H=-  $\cup$  -*] *elim!*: *synth-analz-monotone*)

done

**lemma** *PO-l3-inv8* [*iff*]: *reach l3*  $\subseteq$  *l3-inv8*

**using** *l3-inv8-derived*

*PO-l3-inv3 PO-l3-inv2 PO-l3-inv4*

**by** *blast*

### 23.2.9 inv9

If the level 3 intruder can deduce a message implementing a secure channel message then either:

- the message is already in the intruder knowledge, or

- the message is constructed, and in this case the payload can also be deduced by the intruder, and one of the agents is bad.

**definition**

$l3\text{-inv}9 :: l3\text{-state set}$

**where**

$l3\text{-inv}9 \equiv \{s. \forall A B M.$   
 $(\text{implSecure } A B M \in \text{synth } (\text{analz } (ik\ s)) \wedge M \in \text{payload}) \longrightarrow$   
 $(\text{implSecure } A B M \in ik\ s \vee (M \in \text{synth } (\text{analz } (ik\ s)) \wedge (A \in \text{bad } s \vee B \in \text{bad } s)))$   
 $\}$

**lemmas**  $l3\text{-inv}9I = l3\text{-inv}9\text{-def}$  [*THEN setc-def-to-intro, rule-format*]

**lemmas**  $l3\text{-inv}9E = l3\text{-inv}9\text{-def}$  [*THEN setc-def-to-elim, rule-format*]

**lemmas**  $l3\text{-inv}9D = l3\text{-inv}9\text{-def}$  [*THEN setc-def-to-dest, rule-format*]

**lemma**  $l3\text{-inv}9\text{-derived}$  [*iff*]:

$l3\text{-inv}2 \cap l3\text{-inv}3 \cap l3\text{-inv}4 \subseteq l3\text{-inv}9$

**apply** (*auto intro!*:  $l3\text{-inv}9I$  *dest!*: $l3\text{-inv}3D$   $l3\text{-inv}2D$ )

2 subgoals: M is deducible and the agents are bad

**apply** (*drule synth-mono, simp, drule subsetD, assumption*)

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

**apply** (*drule synth-mono, simp, drule subsetD, assumption*)

**apply** (*auto dest!*: *implSecure-synth-analz* [*rotated 1, where H=- ∪ -*])

**done**

**lemma**  $PO\text{-}l3\text{-inv}9$  [*iff*]:  $\text{reach } l3 \subseteq l3\text{-inv}9$

**using**  $l3\text{-inv}9\text{-derived}$

$PO\text{-}l3\text{-inv}3$   $PO\text{-}l3\text{-inv}2$   $PO\text{-}l3\text{-inv}4$

**by** *blast*

### 23.3 Refinement

Mediator function.

**definition**

$med23s :: l3\text{-obs} \Rightarrow l2\text{-obs}$

**where**

$med23s\ t \equiv (\$   
 $ik = ik\ t \cap \text{payload},$   
 $\text{secret} = \text{secret } t,$   
 $\text{progress} = \text{progress } t,$   
 $\text{signalsInit} = \text{signalsInit } t,$   
 $\text{signalsResp} = \text{signalsResp } t,$   
 $\text{chan} = \text{abs } (ik\ t),$   
 $\text{bad} = \text{bad } t$   
 $\left.)$

Relation between states.

**definition**

$R23s :: (l2\text{-state} * l3\text{-state}) \text{ set}$

**where**

$R23s \equiv \{(s, s') .$   
 $s = \text{med}23s\ s'$   
 $\}$

**lemmas**  $R23s\text{-defs} = R23s\text{-def}\ \text{med}23s\text{-def}$

**lemma**  $R23sI$ :

$\llbracket ik\ s = ik\ t \cap \text{payload}; \text{secret}\ s = \text{secret}\ t; \text{progress}\ s = \text{progress}\ t;$   
 $\text{signalsInit}\ s = \text{signalsInit}\ t; \text{signalsResp}\ s = \text{signalsResp}\ t;$   
 $\text{chan}\ s = \text{abs}\ (ik\ t); l2\text{-state}.\text{bad}\ s = \text{bad}\ t \rrbracket$

$\implies (s, t) \in R23s$

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

**lemma**  $R23sD$ :

$(s, t) \in R23s \implies$   
 $ik\ s = ik\ t \cap \text{payload} \wedge \text{secret}\ s = \text{secret}\ t \wedge \text{progress}\ s = \text{progress}\ t \wedge$   
 $\text{signalsInit}\ s = \text{signalsInit}\ t \wedge \text{signalsResp}\ s = \text{signalsResp}\ t \wedge$   
 $\text{chan}\ s = \text{abs}\ (ik\ t) \wedge l2\text{-state}.\text{bad}\ s = \text{bad}\ t$

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

**lemma**  $R23sE$  [*elim*]:

$\llbracket (s, t) \in R23s;$   
 $\llbracket ik\ s = ik\ t \cap \text{payload}; \text{secret}\ s = \text{secret}\ t; \text{progress}\ s = \text{progress}\ t;$   
 $\text{signalsInit}\ s = \text{signalsInit}\ t; \text{signalsResp}\ s = \text{signalsResp}\ t;$   
 $\text{chan}\ s = \text{abs}\ (ik\ t); l2\text{-state}.\text{bad}\ s = \text{bad}\ t \rrbracket \implies P \rrbracket$

$\implies P$

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

**lemma**  $\text{can-signal-R}23$  [*simp*]:

$(s2, s3) \in R23s \implies$   
 $\text{can-signal}\ s2\ A\ B \longleftrightarrow \text{can-signal}\ s3\ A\ B$

**by** (*auto simp add: can-signal-def*)

### 23.3.1 Protocol events

**lemma**  $l3\text{-step1-refines-step1}$ :

$\{R23s\}\ l2\text{-step1}\ Ra\ A\ B, l3\text{-step1}\ Ra\ A\ B\ \{>R23s\}$

**apply** (*auto simp add: PO-rhoare-defs R23s-defs*)

**apply** (*auto simp add: l3-defs l2-step1-def*)

**done**

**lemma**  $l3\text{-step2-refines-step2}$ :

$\{R23s\}\ l2\text{-step2}\ Rb\ A\ B\ \text{gnx}, l3\text{-step2}\ Rb\ A\ B\ \text{gnx}\ \{>R23s\}$

**apply** (*auto simp add: PO-rhoare-defs R23s-defs l2-step2-def*)

**apply** (*auto simp add: l3-step2-def*)

**done**

**lemma**  $l3\text{-step3-refines-step3}$ :

$\{R23s\}\ l2\text{-step3}\ Ra\ A\ B\ \text{gny}, l3\text{-step3}\ Ra\ A\ B\ \text{gny}\ \{>R23s\}$

**apply** (*auto simp add: PO-rhoare-defs R23s-defs l2-step3-def*)

**apply** (*auto simp add: l3-step3-def*)

done

**lemma** *l3-step4-refines-step4*:

$\{R23s\}$  *l2-step4* *Rb A B gnx*, *l3-step4* *Rb A B gnx*  $\{>R23s\}$

**apply** (*auto simp add: PO-rhoare-defs R23s-defs l2-step4-def*)

**apply** (*auto simp add: l3-step4-def*)

done

### 23.3.2 Intruder events

**lemma** *l3-dy-payload-refines-dy-fake-msg*:

$M \in \text{payload} \implies$

$\{R23s \cap UNIV \times l3\text{-inv}5\}$  *l2-dy-fake-msg* *M*, *l3-dy* *M*  $\{>R23s\}$

**apply** (*auto simp add: PO-rhoare-defs R23s-defs*)

**apply** (*auto simp add: l3-defs l2-dy-fake-msg-def dest: l3-inv5D*)

done

**lemma** *l3-dy-valid-refines-dy-fake-chan*:

$\llbracket M \in \text{valid}; M' \in \text{abs } \{M\} \rrbracket \implies$

$\{R23s \cap UNIV \times (l3\text{-inv}5 \cap l3\text{-inv}6 \cap l3\text{-inv}7 \cap l3\text{-inv}8 \cap l3\text{-inv}9)\}$

*l2-dy-fake-chan* *M'*, *l3-dy* *M*

$\{>R23s\}$

**apply** (*auto simp add: PO-rhoare-defs R23s-defs, simp add: l2-dy-fake-chan-def*)

**apply** (*auto simp add: l3-defs*)

1 subgoal

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

Insec

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

Confid

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

Auth

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

Secure

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

done

**lemma** *l3-dy-valid-refines-dy-fake-chan-Un*:

$M \in \text{valid} \implies$

$\{R23s \cap UNIV \times (l3\text{-inv}5 \cap l3\text{-inv}6 \cap l3\text{-inv}7 \cap l3\text{-inv}8 \cap l3\text{-inv}9)\}$

$\bigcup M'. l2\text{-dy-fake-chan } M', l3\text{-dy } M$

$\{>R23s\}$

**by** (*auto dest: valid-abs intro: l3-dy-valid-refines-dy-fake-chan*)

**lemma** *l3-dy-isLtKey-refines-skip*:

$\{R23s\}$  *Id*, *l3-dy* (*LtK ltk*)  $\{>R23s\}$

**apply** (*auto simp add: PO-rhoare-defs R23s-defs l3-defs*)  
**apply** (*auto elim!: absE*)  
**done**

**lemma** *l3-dy-others-refines-skip*:  
 $\llbracket M \notin \text{range } LtK; M \notin \text{valid}; M \notin \text{payload} \rrbracket \implies$   
 $\{R23s\} \text{Id}, l3\text{-dy } M \{>R23s\}$   
**apply** (*auto simp add: PO-rhoare-defs R23s-defs*)  
**apply** (*auto simp add: l3-defs*)  
**apply** (*auto elim!: absE intro: validI*)  
**done**

**lemma** *l3-dy-refines-dy-fake-msg-dy-fake-chan-skip*:  
 $\{R23s \cap UNIV \times (l3\text{-inv5} \cap l3\text{-inv6} \cap l3\text{-inv7} \cap l3\text{-inv8} \cap l3\text{-inv9})\}$   
 $l2\text{-dy-fake-msg } M \cup (\bigcup M'. l2\text{-dy-fake-chan } M') \cup \text{Id}, l3\text{-dy } M$   
 $\{>R23s\}$   
**by** (*cases*  $M \in \text{payload} \cup \text{valid} \cup \text{range } LtK$ )  
(*auto dest: l3-dy-payload-refines-dy-fake-msg l3-dy-valid-refines-dy-fake-chan-Un*  
*intro: l3-dy-isLtKey-refines-skip dest!: l3-dy-others-refines-skip*)

### 23.3.3 Compromise events

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

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

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

**lemmas** *l3-trans-refines-l2-trans =*  
*l3-step1-refines-step1 l3-step2-refines-step2 l3-step3-refines-step3 l3-step4-refines-step4*  
*l3-dy-refines-dy-fake-msg-dy-fake-chan-skip*  
*l3-lkr-others-refines-lkr-others l3-lkr-after-refines-lkr-after l3-skr-refines-skr*

**lemma** *l3-refines-init-l2 [iff]*:  
 $\text{init } l3 \subseteq R23s \text{ “ (init } l2)$   
**by** (*auto simp add: R23s-defs l2-defs l3-def l3-init-def*)

**lemma** *l3-refines-trans-l2* [iff]:  
 $\{R23s \cap (UNIV \times (l3-inv1 \cap l3-inv2 \cap l3-inv3 \cap l3-inv4))\}$  *trans l2, trans l3*  $\{> R23s\}$   
**proof** –  
**let**  $?pre' = R23s \cap (UNIV \times (l3-inv5 \cap l3-inv6 \cap l3-inv7 \cap l3-inv8 \cap l3-inv9))$   
**show** *?thesis* (**is**  $\{?pre\}$  *?t1, ?t2*  $\{> ?post\}$ )  
**proof** (*rule relhoare-conseq-left*)  
**show**  $?pre \subseteq ?pre'$   
**using** *l3-inv5-derived l3-inv6-derived l3-inv7-derived l3-inv8-derived l3-inv9-derived*  
**by** *blast*  
**next**  
**show**  $\{?pre'\}$  *?t1, ?t2*  $\{> ?post\}$   
**by** (*auto simp add: l2-def l3-def l2-trans-def l3-trans-def*  
*intro!: l3-trans-refines-l2-trans*)  
**qed**  
**qed**

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

**lemma** *l3-refines-l2* [iff]:  
*refines*  
 $(R23s \cap$   
 $(reach\ l2 \times (l3-inv1 \cap l3-inv2 \cap l3-inv3 \cap l3-inv4)))$   
 $med23s\ l2\ l3$   
**by** (*rule Refinement-using-invariants, auto*)

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

## 23.4 Derived invariants

### 23.4.1 inv10: secrets contain no implementation material

#### definition

*l3-inv10* :: *l3-state set*

#### where

$l3-inv10 \equiv \{s.$   
 $secret\ s \subseteq payload$   
 $\}$

**lemmas** *l3-inv10I* = *l3-inv10-def* [THEN *setc-def-to-intro, rule-format*]

**lemmas** *l3-inv10E* = *l3-inv10-def* [THEN *setc-def-to-elim, rule-format*]

**lemmas** *l3-inv10D* = *l3-inv10-def* [THEN *setc-def-to-dest, rule-format*]

**lemma** *l3-inv10-init* [iff]:

$init\ l3 \subseteq l3-inv10$

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

**lemma** *l3-inv10-trans* [iff]:



```

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

```

```

lemma PO-l3-inv10 [iff]: reach l3  $\subseteq$  l3-inv10
by (rule inv-rule-basic) (auto)

```

```

lemma l3-obs-inv10 [iff]: oreach l3  $\subseteq$  l3-inv10
by (auto simp add: oreach-def)

```

### 23.4.2 Partial secrecy

We want to prove *l3-secrecy*, i.e.,  $\text{synth}(\text{analz}(ik\ s)) \cap \text{secret}\ s = \{\}$ , but by refinement we only get *l3-partial-secrecy*:  $\text{dy-fake-msg}(l3\text{-state}.bad\ s)\ (\text{payloadSet}(ik\ s))\ (\text{local.abs}(ik\ s)) \cap \text{secret}\ s = \{\}$ . This is fine if secrets contain no implementation material. Then, by *inv5*, a message in  $\text{synth}(\text{analz}(ik\ s))$  is in  $\text{dy-fake-msg}(l3\text{-state}.bad\ s)\ (\text{payloadSet}(ik\ s))\ (\text{local.abs}(ik\ s)) \cup -\ \text{payload}$ , and *l3-partial-secrecy* proves it is not a secret.

#### definition

```

  l3-partial-secrecy :: ('a l3-state-scheme) set
where
  l3-partial-secrecy  $\equiv$  {s.
    dy-fake-msg (bad s) (ik s  $\cap$  payload) (abs (ik s))  $\cap$  secret s = {}}
}

```

```

lemma l3-obs-partial-secrecy [iff]: oreach l3  $\subseteq$  l3-partial-secrecy
apply (rule external-invariant-translation [OF l2-obs-secrecy - l3-implements-l2])
apply (auto simp add: med23s-def l2-secrecy-def l3-partial-secrecy-def)
done

```

### 23.4.3 Secrecy

#### definition

```

  l3-secrecy :: ('a l3-state-scheme) set
where
  l3-secrecy  $\equiv$  l1-secrecy

```

```

lemma l3-obs-inv5: oreach l3  $\subseteq$  l3-inv5
by (auto simp add: oreach-def)

```

```

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

```

```

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

```

#### 23.4.4 Injective agreement

**abbreviation** *l3-agreement-Init*  $\equiv$  *l1-agreement-Init*

**lemma** *l3-obs-iagreement-Init* [*iff*]: *oreach*  $l3 \subseteq l3\text{-iagreement-Init}$   
**apply** (*rule external-invariant-translation*  
    [*OF l2-obs-iagreement-Init - l3-implements-l2*])  
**apply** (*auto simp add: med23s-def l1-iagreement-Init-def*)  
**done**

**lemma** *l3-iagreement-Init* [*iff*]: *reach*  $l3 \subseteq l3\text{-iagreement-Init}$   
**by** (*rule external-to-internal-invariant [OF l3-obs-iagreement-Init]*, *auto*)

**abbreviation** *l3-iagreement-Resp*  $\equiv$  *l1-iagreement-Resp*

**lemma** *l3-obs-iagreement-Resp* [*iff*]: *oreach*  $l3 \subseteq l3\text{-iagreement-Resp}$   
**apply** (*rule external-invariant-translation*  
    [*OF l2-obs-iagreement-Resp - l3-implements-l2*])  
**apply** (*auto simp add: med23s-def l1-iagreement-Resp-def*)  
**done**

**lemma** *l3-iagreement-Resp* [*iff*]: *reach*  $l3 \subseteq l3\text{-iagreement-Resp}$   
**by** (*rule external-to-internal-invariant [OF l3-obs-iagreement-Resp]*, *auto*)

**end**  
**end**

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

```
theory dhlvl3-asymmetric
imports dhlvl3 Implem-asymmetric
begin

interpretation dhlvl3-asm: dhlvl3 implem-asm
by (unfold-locales)

end
```

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

```
theory dhlvl3-symmetric
imports dhlvl3 Implem-symmetric
begin

interpretation dhlvl3-sym: dhlvl3 implem-sym
by (unfold-locales)

end
```

## 26 SKEME Protocol (L1)

```
theory sklvl1
imports dhvl1
begin
```

```
declare option.split-asm [split]
```

### 26.1 State and Events

```
abbreviation ni :: nat where ni  $\equiv$  4
```

```
abbreviation nr :: nat where nr  $\equiv$  5
```

Proofs break if  $1::'a$  is used, because *simp* replaces it with *Suc 0*...

```
abbreviation
```

```
  xni  $\equiv$  Var 7
```

```
abbreviation
```

```
  xnr  $\equiv$  Var 8
```

Domain of each role (protocol-dependent).

```
fun domain :: role-t  $\Rightarrow$  var set where
```

```
  domain Init = {xnx, xni, xnr, xgnx, xgny, xsk, xEnd}
```

```
| domain Resp = {xny, xni, xnr, xgnx, xgny, xsk, xEnd}
```

```
consts
```

```
  guessed-frame :: rid-t  $\Rightarrow$  frame
```

Specification of the guessed frame:

1. Domain.
2. Well-typedness. The messages in the frame of a run never contain implementation material even if the agents of the run are dishonest. Therefore we consider only well-typed frames. This is notably required for the session key compromise; it also helps proving the partitioning of ik, since we know that the messages added by the protocol do not contain lkeys in their payload and are therefore valid implementations.
3. We also ensure that the values generated by the frame owner are correctly guessed.
4. The new frame extends the previous one (from *Key-Agreement-Strong-Adversaries.dhvl1*)

```
specification (guessed-frame)
```

```
  guessed-frame-dom-spec [simp]:
```

```
    dom (guessed-frame R) = domain (role (guessed-runs R))
```

```
  guessed-frame-payload-spec [simp, elim]:
```

```
    guessed-frame R x = Some y  $\implies$  y  $\in$  payload
```

```
  guessed-frame-Init-xnx [simp]:
```

```
    role (guessed-runs R) = Init  $\implies$  guessed-frame R xnx = Some (NonceF (R$nx))
```

```
  guessed-frame-Init-xgnx [simp]:
```

```
    role (guessed-runs R) = Init  $\implies$  guessed-frame R xgnx = Some (Exp Gen (NonceF (R$nx)))
```

```
  guessed-frame-Init-xni [simp]:
```

```

    role (guessed-runs R) = Init  $\implies$  guessed-frame R xni = Some (NonceF (R$ni))
guessed-frame-Resp-xny [simp]:
    role (guessed-runs R) = Resp  $\implies$  guessed-frame R xny = Some (NonceF (R$ny))
guessed-frame-Resp-xgny [simp]:
    role (guessed-runs R) = Resp  $\implies$  guessed-frame R xgny = Some (Exp Gen (NonceF (R$ny)))
guessed-frame-Resp-xnr [simp]:
    role (guessed-runs R) = Resp  $\implies$  guessed-frame R xnr = Some (NonceF (R$nr))
guessed-frame-xEnd [simp]:
    guessed-frame R xEnd = Some End
guessed-frame-eq [simp]:
    x  $\in$  {xnx, xny, xgnx, xgny, xsk, xEnd}  $\implies$  dhlvl1.guessed-frame R x = guessed-frame R x
apply (rule exI [of -
     $\lambda R.$ 
    if role (guessed-runs R) = Init then
      (dhlvl1.guessed-frame R) (xni  $\mapsto$  NonceF (R$ni), xnr  $\mapsto$  End)
    else
      (dhlvl1.guessed-frame R) (xnr  $\mapsto$  NonceF (R$nr), xni  $\mapsto$  End)],
    auto simp add: domIff intro: role-t.exhaust)
done

```

```

record skl1-state =
  l1-state +
  signalsInit2 :: signal  $\Rightarrow$  nat
  signalsResp2 :: signal  $\Rightarrow$  nat

```

**type-synonym** skl1-obs = skl1-state

Protocol events:

- step 1: create  $Ra$ ,  $A$  generates  $nx$  and  $ni$ , computes  $g^{nx}$
- step 2: create  $Rb$ ,  $B$  reads  $ni$  and  $g^{nx}$  insecurely, generates  $ny$  and  $nr$ , computes  $g^{ny}$ , computes  $g^{nx} * ny$ , emits a running signal for  $Init$ ,  $ni$ ,  $nr$ ,  $g^{nx} * ny$
- step 3:  $A$  reads  $g^{ny}$  and  $g^{nx}$  authentically, computes  $g^{ny} * nx$ , emits a commit signal for  $Init$ ,  $ni$ ,  $nr$ ,  $g^{ny} * nx$ , a running signal for  $Resp$ ,  $ni$ ,  $nr$ ,  $g^{ny} * nx$ , declares the secret  $g^{ny} * nx$
- step 4:  $B$  reads  $nr$ ,  $ni$ ,  $g^{nx}$  and  $g^{ny}$  authentically, emits a commit signal for  $Resp$ ,  $ni$ ,  $nr$ ,  $g^{nx} * ny$ , declares the secret  $g^{nx} * ny$

**definition**

$skl1\text{-}step1 :: rid\text{-}t \Rightarrow agent \Rightarrow agent \Rightarrow ('a\ skl1\text{-}state\text{-}scheme * 'a\ skl1\text{-}state\text{-}scheme)\ set$

**where**

$skl1\text{-}step1\ Ra\ A\ B \equiv \{(s, s')\}.$

— guards:

$Ra \notin dom\ (progress\ s) \wedge$   
 $guessed\text{-}runs\ Ra = (\text{role} = Init, \text{owner} = A, \text{partner} = B) \wedge$

— actions:

$s' = s(\text{progress} := (progress\ s)(Ra \mapsto \{xnx, xni, xgnx\}))$   
 $\})$

}

**definition**

*skl1-step2* ::

*rid-t*  $\Rightarrow$  *agent*  $\Rightarrow$  *agent*  $\Rightarrow$  *msg*  $\Rightarrow$  *msg*  $\Rightarrow$  ('a *skl1-state-scheme* \* 'a *skl1-state-scheme*) *set*

**where**

*skl1-step2* *Rb A B Ni gn $x$*   $\equiv$  {(*s*, *s'*}.

— guards:

*guessed-runs* *Rb* = ( $\downarrow$ *role=Resp*, *owner=B*, *partner=A*)  $\wedge$

*Rb*  $\notin$  *dom* (*progress* *s*)  $\wedge$

*guessed-frame* *Rb* *xgn $x$*  = *Some gn $x$*   $\wedge$

*guessed-frame* *Rb* *xni* = *Some Ni*  $\wedge$

*guessed-frame* *Rb* *xsk* = *Some (Exp gn $x$  (NonceF (Rb\$ny)))*  $\wedge$

— actions:

*s'* = *s*(*progress* := (*progress* *s*)(*Rb*  $\mapsto$  {*xny*, *xni*, *xnr*, *xgny*, *xgn $x$* , *xsk*}),

*signalsInit* :=

if *can-signal* *s* *A B* then

addSignal (*signalsInit* *s*)

(*Running* *A B* (*Ni*, *NonceF* (*Rb*\$*nr*), *Exp gn $x$  (NonceF (Rb\$ny))*))

else

*signalsInit* *s*,

*signalsInit2* :=

if *can-signal* *s* *A B* then

addSignal (*signalsInit2* *s*) (*Running* *A B* (*Exp gn $x$  (NonceF (Rb\$ny))*))

else

*signalsInit2* *s*

)

}

**definition**

*skl1-step3* ::

*rid-t*  $\Rightarrow$  *agent*  $\Rightarrow$  *agent*  $\Rightarrow$  *msg*  $\Rightarrow$  *msg*  $\Rightarrow$  ('a *skl1-state-scheme* \* 'a *skl1-state-scheme*) *set*

**where**

*skl1-step3* *Ra A B Nr gn $y$*   $\equiv$  {(*s*, *s'*}.

— guards:

*guessed-runs* *Ra* = ( $\downarrow$ *role=Init*, *owner=A*, *partner=B*)  $\wedge$

*progress* *s* *Ra* = *Some* {*xnx*, *xni*, *xgn $x$* }  $\wedge$

*guessed-frame* *Ra* *xgny* = *Some gn $y$*   $\wedge$

*guessed-frame* *Ra* *xnr* = *Some Nr*  $\wedge$

*guessed-frame* *Ra* *xsk* = *Some (Exp gn $y$  (NonceF (Ra\$nx)))*  $\wedge$

(*can-signal* *s* *A B*  $\longrightarrow$  — authentication guard

( $\exists$  *Rb*. *guessed-runs* *Rb* = ( $\downarrow$ *role=Resp*, *owner=B*, *partner=A*)  $\wedge$

*in-progressS* (*progress* *s* *Rb*) {*xny*, *xni*, *xnr*, *xgn $x$* , *xgny*, *xsk*}  $\wedge$

*guessed-frame* *Rb* *xgny* = *Some gn $y$*   $\wedge$

*guessed-frame* *Rb* *xnr* = *Some Nr*  $\wedge$

*guessed-frame* *Rb* *xni* = *Some (NonceF (Ra\$ni))*  $\wedge$

*guessed-frame* *Rb* *xgn $x$*  = *Some (Exp Gen (NonceF (Ra\$nx)))*))  $\wedge$

(*Ra* = *test*  $\longrightarrow$  *Exp gn $y$  (NonceF (Ra\$nx))*  $\notin$  *synth* (*analz* (*ik* *s*)))  $\wedge$

— actions:

*s'* = *s*(*progress* := (*progress* *s*)(*Ra*  $\mapsto$  {*xnx*, *xni*, *xnr*, *xgn $x$* , *xgny*, *xsk*, *xEnd*}),

*secret* := {*x*. *x* = *Exp gn $y$  (NonceF (Ra\$nx))*  $\wedge$  *Ra* = *test*}  $\cup$  *secret* *s*,

```

signalsInit :=
  if can-signal s A B then
    addSignal (signalsInit s)
      (Commit A B ⟨NonceF (Ra$ni), Nr, Exp gny (NonceF (Ra$nx))⟩)
  else
    signalsInit s,
signalsInit2 :=
  if can-signal s A B then
    addSignal (signalsInit2 s) (Commit A B (Exp gny (NonceF (Ra$nx))))
  else
    signalsInit2 s,
signalsResp :=
  if can-signal s A B then
    addSignal (signalsResp s)
      (Running A B ⟨NonceF (Ra$ni), Nr, Exp gny (NonceF (Ra$nx))⟩)
  else
    signalsResp s,
signalsResp2 :=
  if can-signal s A B then
    addSignal (signalsResp2 s) (Running A B (Exp gny (NonceF (Ra$nx))))
  else
    signalsResp2 s
}

```

**definition**

*skl1-step4* ::  
*rid-t* ⇒ *agent* ⇒ *agent* ⇒ *msg* ⇒ *msg* ⇒ (*'a skl1-state-scheme* \* *'a skl1-state-scheme*) *set*

**where**

*skl1-step4* *Rb A B Ni gnx* ≡ {(*s*, *s'*).

— guards:

*guessed-runs Rb* = (|*role=Resp*, *owner=B*, *partner=A*) ∧

*progress s Rb* = *Some* {*xny*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*} ∧

*guessed-frame Rb xgnx* = *Some gnx* ∧

*guessed-frame Rb xni* = *Some Ni* ∧

(*can-signal s A B* → — authentication guard

(∃ *Ra*. *guessed-runs Ra* = (|*role=Init*, *owner=A*, *partner=B*) ∧

*in-progressS* (*progress s Ra*) {*xnx*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*, *xEnd*} ∧

*guessed-frame Ra xgnx* = *Some gnx* ∧

*guessed-frame Ra xni* = *Some Ni* ∧

*guessed-frame Ra xnr* = *Some (NonceF (Rb\$nr))* ∧

*guessed-frame Ra xgny* = *Some (Exp Gen (NonceF (Rb\$ny)))))* ∧

(*Rb = test* → *Exp gnx (NonceF (Rb\$ny))* ∉ *synth (analz (ik s))*) ∧

— actions:

*s'* = *s*(| *progress* := (*progress s*)(*Rb* ↦ {*xny*, *xni*, *xnr*, *xgnx*, *xgny*, *xsk*, *xEnd*}),

*secret* := {*x*. *x* = *Exp gnx (NonceF (Rb\$ny))* ∧ *Rb = test*} ∪ *secret s*,

*signalsResp* :=

if *can-signal s A B* then

addSignal (*signalsResp s*)

(Commit A B ⟨*Ni*, *NonceF (Rb\$nr)*, *Exp gnx (NonceF (Rb\$ny))*⟩)

else



```

      signalsResp s,
signalsResp2 :=
  if can-signal s A B then
    addSignal (signalsResp2 s) (Commit A B (Exp gnx (NonceF (Rb$ny))))
  else
    signalsResp2 s
    )
}

```

Specification.

**definition**

```

skl1-trans :: ('a skl1-state-scheme * 'a skl1-state-scheme) set where
skl1-trans ≡ (∪ m Ra Rb A B x y.
  skl1-step1 Ra A B ∪
  skl1-step2 Rb A B x y ∪
  skl1-step3 Ra A B x y ∪
  skl1-step4 Rb A B x y ∪
  l1-learn m ∪
  Id
)

```

**definition**

*skl1-init* :: *skl1-state set*

**where**

```

skl1-init ≡ { ()
  ik = {},
  secret = {},
  progress = Map.empty,
  signalsInit = λx. 0,
  signalsResp = λx. 0,
  signalsInit2 = λx. 0,
  signalsResp2 = λx. 0
}

```

**definition**

*skl1* :: (*skl1-state, skl1-obs*) *spec* **where**

```

skl1 ≡ ()
  init = skl1-init,
  trans = skl1-trans,
  obs = id
)

```

**lemmas** *skl1-defs* =

```

skl1-def skl1-init-def skl1-trans-def
l1-learn-def
skl1-step1-def skl1-step2-def skl1-step3-def skl1-step4-def

```

**lemmas** *skl1-nostep-defs* =

```

skl1-def skl1-init-def skl1-trans-def

```

**lemma** *skl1-obs-id* [*simp*]: *obs skl1 = id*

**by** (*simp add: skl1-def*)

**lemma** *run-ended-trans*:  
 $run\text{-}ended\ (progress\ s\ R) \implies$   
 $(s, s') \in trans\ skl1 \implies$   
 $run\text{-}ended\ (progress\ s'\ R)$   
**by** (*auto simp add: skl1-nostep-defs*)  
*(auto simp add: skl1-defs ik-dy-def domIff)*

**lemma** *can-signal-trans*:  
 $can\text{-}signal\ s'\ A\ B \implies$   
 $(s, s') \in trans\ skl1 \implies$   
 $can\text{-}signal\ s\ A\ B$   
**by** (*auto simp add: can-signal-def run-ended-trans*)

## 26.2 Refinement: secrecy

**fun** *option-inter* :: *var set*  $\Rightarrow$  *var set option*  $\Rightarrow$  *var set option*  
**where**  
 $option\text{-}inter\ S\ (Some\ x) = Some\ (x \cap S)$   
 $option\text{-}inter\ S\ None = None$

**definition** *med-progress* :: *progress-t*  $\Rightarrow$  *progress-t*  
**where**  
 $med\text{-}progress\ r \equiv \lambda R. option\text{-}inter\ \{xnx, xny, xgnx, xgny, xsk, xEnd\}\ (r\ R)$

**lemma** *med-progress-upd* [*simp*]:  
 $med\text{-}progress\ (r(R \mapsto S)) = (med\text{-}progress\ r)\ (R \mapsto S \cap \{xnx, xny, xgnx, xgny, xsk, xEnd\})$   
**by** (*auto simp add: med-progress-def*)

**lemma** *med-progress-Some*:  
 $r\ x = Some\ s \implies med\text{-}progress\ r\ x = Some\ (s \cap \{xnx, xny, xgnx, xgny, xsk, xEnd\})$   
**by** (*auto simp add: med-progress-def*)

**lemma** *med-progress-None* [*simp*]:  $med\text{-}progress\ r\ x = None \longleftrightarrow r\ x = None$   
**by** (*cases r x, auto simp add: med-progress-def*)

**lemma** *med-progress-Some2* [*dest*]:  
 $med\text{-}progress\ r\ x = Some\ y \implies \exists z. r\ x = Some\ z \wedge y = z \cap \{xnx, xny, xgnx, xgny, xsk, xEnd\}$   
**by** (*cases r x, auto simp add: med-progress-def*)

**lemma** *med-progress-dom* [*simp*]:  $dom\ (med\text{-}progress\ r) = dom\ r$   
**apply** (*auto simp add: domIff med-progress-def*)  
**apply** (*rename-tac x y, case-tac r x, auto*)  
**done**

**lemma** *med-progress-empty* [*simp*]:  $med\text{-}progress\ Map.empty = Map.empty$   
**by** (*rule ext, auto*)

Mediator function.

**definition**  
 $med11 :: skl1\text{-}obs \Rightarrow l1\text{-}obs$   
**where**

$med11\ t \equiv (\text{ik} = \text{ik}\ t,$   
 $\text{secret} = \text{secret}\ t,$   
 $\text{progress} = \text{med-progress}(\text{progress}\ t),$   
 $\text{signalsInit} = \text{signalsInit2}\ t,$   
 $\text{signalsResp} = \text{signalsResp2}\ t)$

relation between states

**definition**

$R11 :: (\text{l1-state} * \text{skl1-state})\ \text{set}$

**where**

$R11 \equiv \{(s, s').$   
 $\quad s = \text{med11}\ s'$   
 $\quad \}$

**lemmas**  $R11\text{-defs} = R11\text{-def}\ \text{med11}\text{-def}$

**lemma**  $\text{in-progress-med-progress}$ :

$x \in \{xnx, xny, xgnx, xgny, xsk, xEnd\}$

$\implies \text{in-progress}(\text{med-progress}\ r\ R)\ x \longleftrightarrow \text{in-progress}(r\ R)\ x$

**by** ( $\text{cases}\ r\ R, \text{auto}$ )

( $\text{cases}\ \text{med-progress}\ r\ R, \text{auto}$ ) $+$

**lemma**  $\text{in-progressS-eq}$ :  $\text{in-progressS}\ S\ S' \longleftrightarrow (S \neq \text{None} \wedge (\forall x \in S'. \text{in-progress}\ S\ x))$

**by** ( $\text{cases}\ S, \text{auto}$ )

**lemma**  $\text{in-progressS-med-progress}$ :

$\text{in-progressS}(r\ R)\ S$

$\implies \text{in-progressS}(\text{med-progress}\ r\ R)\ (S \cap \{xnx, xny, xgnx, xgny, xsk, xEnd\})$

**by** ( $\text{auto}\ \text{simp}\ \text{add:}\ \text{in-progressS-eq}\ \text{in-progress-med-progress}$ )

**lemma**  $\text{can-signal-R11}$  [ $\text{simp}$ ]:

$(s1, s2) \in R11 \implies$

$\text{can-signal}\ s1\ A\ B \longleftrightarrow \text{can-signal}\ s2\ A\ B$

**by** ( $\text{auto}\ \text{simp}\ \text{add:}\ \text{can-signal-def}\ R11\text{-defs}\ \text{in-progress-med-progress}$ )

Protocol-independent events.

**lemma**  $\text{skl1-learn-refines-learn}$ :

$\{R11\}\ \text{l1-learn}\ m, \text{l1-learn}\ m\ \{>R11\}$

**by** ( $\text{auto}\ \text{simp}\ \text{add:}\ \text{PO-rhoare-defs}\ R11\text{-defs}$ )

( $\text{simp}\ \text{add:}\ \text{l1-defs}$ )

Protocol events.

**lemma**  $\text{skl1-step1-refines-step1}$ :

$\{R11\}\ \text{l1-step1}\ Ra\ A\ B, \text{skl1-step1}\ Ra\ A\ B\ \{>R11\}$

**by** ( $\text{auto}\ \text{simp}\ \text{add:}\ \text{PO-rhoare-defs}\ R11\text{-defs}\ \text{l1-step1-def}\ \text{skl1-step1-def}$ )

**lemma**  $\text{skl1-step2-refines-step2}$ :

$\{R11\}\ \text{l1-step2}\ Rb\ A\ B\ gnx, \text{skl1-step2}\ Rb\ A\ B\ Ni\ gnx\ \{>R11\}$

**by** ( $\text{auto}\ \text{simp}\ \text{add:}\ \text{PO-rhoare-defs}\ R11\text{-defs}\ \text{l1-step2-def}$ )

( $\text{auto}\ \text{simp}\ \text{add:}\ \text{skl1-step2-def}$ )

**lemma** *skl1-step3-refines-step3*:  
 $\{R11\}$  *l1-step3* *Ra A B gny*, *skl1-step3 Ra A B Nr gny*  $\{>R11\}$   
**apply** (*auto simp add: PO-rhoare-defs R11-defs l1-step3-def*)  
**apply** (*auto simp add: skl1-step3-def, auto dest: med-progress-Some*)  
**apply** (*drule in-progressS-med-progress, auto*)  
**done**

**lemma** *skl1-step4-refines-step4*:  
 $\{R11\}$  *l1-step4* *Rb A B gnx*, *skl1-step4 Rb A B Ni gnx*  $\{>R11\}$   
**apply** (*auto simp add: PO-rhoare-defs R11-defs l1-step4-def*)  
**apply** (*auto simp add: skl1-step4-def, auto dest: med-progress-Some*)  
**apply** (*drule in-progressS-med-progress, auto*)  
**done**

Refinement proof.

**lemmas** *skl1-trans-refines-l1-trans* =  
*skl1-learn-refines-learn*  
*skl1-step1-refines-step1 skl1-step2-refines-step2*  
*skl1-step3-refines-step3 skl1-step4-refines-step4*

**lemma** *skl1-refines-init-l1* [*iff*]:  
*init skl1*  $\subseteq$  *R11* “ (*init l1*)  
**by** (*auto simp add: R11-defs l1-defs skl1-defs*)

**lemma** *skl1-refines-trans-l1* [*iff*]:  
 $\{R11\}$  *trans l1*, *trans skl1*  $\{> R11\}$   
**by** (*auto 0 3 simp add: l1-def skl1-def l1-trans-def skl1-trans-def*  
*intro: skl1-trans-refines-l1-trans*)

**lemma** *obs-consistent-med11* [*iff*]:  
*obs-consistent R11 med11 l1 skl1*  
**by** (*auto simp add: obs-consistent-def R11-defs*)

Refinement result.

**lemma** *skl1-refines-l1* [*iff*]:  
*refines*  
*R11*  
*med11 l1 skl1*  
**by** (*auto simp add:refines-def PO-refines-def*)

**lemma** *skl1-implements-l1* [*iff*]: *implements med11 l1 skl1*  
**by** (*rule refinement-soundness*) (*fast*)

### 26.3 Derived invariants: secrecy

**lemma** *skl1-obs-secrecy* [*iff*]: *oreach skl1*  $\subseteq$  *s0-secrecy*  
**apply** (*rule external-invariant-translation [OF l1-obs-secrecy - skl1-implements-l1]*)  
**apply** (*auto simp add: med11-def s0-secrecy-def*)  
**done**

**lemma** *skl1-secrecy* [*iff*]: *reach skl1*  $\subseteq$  *s0-secrecy*

by (rule external-to-internal-invariant [OF skl1-obs-secrecy], auto)

## 26.4 Invariants: *Init* authenticates *Resp*

### 26.4.1 inv1

If an initiator commit signal exists for  $Ra \$ ni, Nr, (g^{ny})Ra \$ nx$ , then  $Ra$  is *Init*, has passed step 3, and has the nonce  $Nr$ , and  $(g \hat{ny}) \wedge (Ra \$ nx)$  as the key in its frame.

**definition**

$skl1\text{-}inv1 :: skl1\text{-}state\ set$

**where**

$$\begin{aligned}
 skl1\text{-}inv1 \equiv \{s. \forall Ra\ A\ B\ gny\ Nr. \\
 &signalsInit\ s\ (Commit\ A\ B\ \langle NonceF\ (Ra\ \$ni),\ Nr,\ Exp\ gny\ (NonceF\ (Ra\ \$nx)) \rangle) > 0 \longrightarrow \\
 &guessed\text{-}runs\ Ra = (\text{role}=\text{Init},\ \text{owner}=A,\ \text{partner}=B) \wedge \\
 &progress\ s\ Ra = Some\ \{xnx,\ xni,\ xnr,\ xgnx,\ xgny,\ xsk,\ xEnd\} \wedge \\
 &guessed\text{-}frame\ Ra\ xnr = Some\ Nr \wedge \\
 &guessed\text{-}frame\ Ra\ xsk = Some\ (Exp\ gny\ (NonceF\ (Ra\ \$nx))) \\
 &\}
 \end{aligned}$$

**lemmas**  $skl1\text{-}inv1I = skl1\text{-}inv1\text{-}def\ [THEN\ setc\text{-}def\text{-}to\text{-}intro,\ rule\text{-}format]$

**lemmas**  $skl1\text{-}inv1E\ [elim] = skl1\text{-}inv1\text{-}def\ [THEN\ setc\text{-}def\text{-}to\text{-}elim,\ rule\text{-}format]$

**lemmas**  $skl1\text{-}inv1D = skl1\text{-}inv1\text{-}def\ [THEN\ setc\text{-}def\text{-}to\text{-}dest,\ rule\text{-}format,\ rotated\ 1,\ simplified]$

**lemma**  $skl1\text{-}inv1\text{-}init\ [iff]:$

$init\ skl1 \subseteq skl1\text{-}inv1$

by (auto simp add: skl1-def skl1-init-def skl1-inv1-def)

**lemma**  $skl1\text{-}inv1\text{-}trans\ [iff]:$

$\{skl1\text{-}inv1\}\ trans\ skl1\ \{>\ skl1\text{-}inv1\}$

**apply** (auto simp add: PO-hoare-defs skl1-nostep-defs intro!: skl1-inv1I)

**apply** (auto simp add: skl1-defs ik-dy-def skl1-inv1-def domIff dest: Exp-Exp-Gen-inj2 [OF sym])

**done**

**lemma**  $PO\text{-}skl1\text{-}inv1\ [iff]:\ reach\ skl1 \subseteq skl1\text{-}inv1$

by (rule inv-rule-basic) (auto)

### 26.4.2 inv2

If a *Resp* run  $Rb$  has passed step 2 then (if possible) an initiator running signal has been emitted.

**definition**

$skl1\text{-}inv2 :: skl1\text{-}state\ set$

**where**

$$\begin{aligned}
 skl1\text{-}inv2 \equiv \{s. \forall gnx\ A\ B\ Rb\ Ni. \\
 &guessed\text{-}runs\ Rb = (\text{role}=\text{Resp},\ \text{owner}=B,\ \text{partner}=A) \longrightarrow \\
 &in\text{-}progressS\ (progress\ s\ Rb)\ \{xny,\ xni,\ xnr,\ xgnx,\ xgny,\ xsk\} \longrightarrow \\
 &guessed\text{-}frame\ Rb\ xgnx = Some\ gnx \longrightarrow \\
 &guessed\text{-}frame\ Rb\ xni = Some\ Ni \longrightarrow \\
 &can\text{-}signal\ s\ A\ B \longrightarrow \\
 &signalsInit\ s\ (Running\ A\ B\ \langle Ni,\ NonceF\ (Rb\ \$nr),\ Exp\ gnx\ (NonceF\ (Rb\ \$ny)) \rangle) > 0 \\
 &\}
 \end{aligned}$$

**lemmas**  $skl1\text{-}inv2I = skl1\text{-}inv2\text{-}def$  [THEN setc-def-to-intro, rule-format]  
**lemmas**  $skl1\text{-}inv2E$  [elim] =  $skl1\text{-}inv2\text{-}def$  [THEN setc-def-to-elim, rule-format]  
**lemmas**  $skl1\text{-}inv2D = skl1\text{-}inv2\text{-}def$  [THEN setc-def-to-dest, rule-format, rotated 1, simplified]

**lemma**  $skl1\text{-}inv2\text{-}init$  [iff]:  
 $init\ skl1 \subseteq skl1\text{-}inv2$   
**by** (auto simp add:  $skl1\text{-}def\ skl1\text{-}init\text{-}def\ skl1\text{-}inv2\text{-}def$ )

**lemma**  $skl1\text{-}inv2\text{-}trans$  [iff]:  
 $\{skl1\text{-}inv2\}$  trans  $skl1 \{> skl1\text{-}inv2\}$   
**apply** (auto simp add: PO-hoare-defs intro!:  $skl1\text{-}inv2I$ )  
**apply** (drule can-signal-trans, assumption)  
**apply** (auto simp add:  $skl1\text{-}nostep\text{-}defs$ )  
**apply** (auto simp add:  $skl1\text{-}defs\ ik\text{-}dy\text{-}def\ skl1\text{-}inv2\text{-}def$ )  
**done**

**lemma**  $PO\text{-}skl1\text{-}inv2$  [iff]:  $reach\ skl1 \subseteq skl1\text{-}inv2$   
**by** (rule inv-rule-basic) (auto)

### 26.4.3 inv3 (derived)

If an *Init* run before step 3 and a *Resp* run after step 2 both know the same half-keys and nonces (more or less), then the number of *Init* running signals for the key is strictly greater than the number of *Init* commit signals. (actually, there are 0 commit and 1 running).

#### definition

$skl1\text{-}inv3 :: skl1\text{-}state\ set$

#### where

$skl1\text{-}inv3 \equiv \{s. \forall A B Rb Ra gny Nr.$   
 $guessed\text{-}runs\ Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \longrightarrow$   
 $in\text{-}progressS\ (\text{progress}\ s\ Rb) \{xny, xni, xnr, xgnx, xgny, xsk\} \longrightarrow$   
 $guessed\text{-}frame\ Rb\ xgny = \text{Some}\ gny \longrightarrow$   
 $guessed\text{-}frame\ Rb\ xnr = \text{Some}\ Nr \longrightarrow$   
 $guessed\text{-}frame\ Rb\ xni = \text{Some}\ (\text{NonceF}\ (Ra\$ni)) \longrightarrow$   
 $guessed\text{-}frame\ Rb\ xgnx = \text{Some}\ (\text{Exp}\ \text{Gen}\ (\text{NonceF}\ (Ra\$nx))) \longrightarrow$   
 $guessed\text{-}runs\ Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \longrightarrow$   
 $progress\ s\ Ra = \text{Some}\ \{xnx, xgnx, xni\} \longrightarrow$   
 $can\text{-}signal\ s\ A\ B \longrightarrow$   
 $signalsInit\ s\ (\text{Commit}\ A\ B\ \langle \text{NonceF}\ (Ra\$ni), Nr, \text{Exp}\ gny\ (\text{NonceF}\ (Ra\$nx)) \rangle)$   
 $<\ signalsInit\ s\ (\text{Running}\ A\ B\ \langle \text{NonceF}\ (Ra\$ni), Nr, \text{Exp}\ gny\ (\text{NonceF}\ (Ra\$nx)) \rangle)$   
 $\}$

**lemmas**  $skl1\text{-}inv3I = skl1\text{-}inv3\text{-}def$  [THEN setc-def-to-intro, rule-format]  
**lemmas**  $skl1\text{-}inv3E$  [elim] =  $skl1\text{-}inv3\text{-}def$  [THEN setc-def-to-elim, rule-format]  
**lemmas**  $skl1\text{-}inv3D = skl1\text{-}inv3\text{-}def$  [THEN setc-def-to-dest, rule-format, rotated 1, simplified]

**lemma**  $skl1\text{-}inv3\text{-}derived$ :  $skl1\text{-}inv1 \cap skl1\text{-}inv2 \subseteq skl1\text{-}inv3$   
**apply** (auto intro!:  $skl1\text{-}inv3I$ )  
**apply** (auto dest!:  $skl1\text{-}inv2D$ )  
**apply** (rename-tac  $x\ A\ B\ Rb\ Ra$ )  
**apply** (case-tac

```

signalsInit x (Commit A B
  ⟨NonceF (Ra $ ni), NonceF (Rb $ nr),
  Exp (Exp Gen (NonceF (Rb $ ny))) (NonceF (Ra $ nx))⟩) > 0, auto)
apply (fastforce dest: skl1-inv1D elim: equalityE)
done

```

## 26.5 Invariants: Resp authenticates Init

### 26.5.1 inv4

If a *Resp* commit signal exists for  $Ni$ ,  $Rb \$ nr$ ,  $(g^{nx})Rb \$ ny$  then  $Rb$  is *Resp*, has finished its run, and has the nonce  $Ni$  and  $(g^{nx})Rb \$ ny$  as the key in its frame.

**definition**

$skl1-inv4 :: skl1-state\ set$

**where**

```

skl1-inv4 ≡ {s. ∀ Rb A B gnx Ni.
  signalsResp s (Commit A B ⟨Ni, NonceF (Rb$nr), Exp gnx (NonceF (Rb$ny))⟩) > 0 →
  guessed-runs Rb = ⟨role=Resp, owner=B, partner=A⟩ ∧
  progress s Rb = Some {xny, xni, xnr, xgnx, xgny, xsk, xEnd} ∧
  guessed-frame Rb xgnx = Some gnx ∧
  guessed-frame Rb xni = Some Ni
}

```

**lemmas**  $skl1-inv4I = skl1-inv4-def [THEN setc-def-to-intro, rule-format]$

**lemmas**  $skl1-inv4E [elim] = skl1-inv4-def [THEN setc-def-to-elim, rule-format]$

**lemmas**  $skl1-inv4D = skl1-inv4-def [THEN setc-def-to-dest, rule-format, rotated 1, simplified]$

**lemma**  $skl1-inv4-init [iff]$ :

$init\ skl1 \subseteq skl1-inv4$

**by** (auto simp add: skl1-def skl1-init-def skl1-inv4-def)

**lemma**  $skl1-inv4-trans [iff]$ :

$\{skl1-inv4\}\ trans\ skl1\ \{>\ skl1-inv4\}$

**apply** (auto simp add: PO-hoare-defs skl1-nostep-defs intro!: skl1-inv4I)

**apply** (auto simp add: skl1-inv4-def skl1-defs ik-dy-def domIff dest: Exp-Exp-Gen-inj2 [OF sym])

**done**

**lemma**  $PO-skl1-inv4 [iff]$ :  $reach\ skl1 \subseteq skl1-inv4$

**by** (rule inv-rule-basic) (auto)

### 26.5.2 inv5

If an *Init* run  $Ra$  has passed step3 then (if possible) a *Resp* running signal has been emitted.

**definition**

$skl1-inv5 :: skl1-state\ set$

**where**

```

skl1-inv5 ≡ {s. ∀ gny A B Ra Nr.
  guessed-runs Ra = ⟨role=Init, owner=A, partner=B⟩ →
  in-progressS (progress s Ra) {xnx, xni, xnr, xgnx, xgny, xsk, xEnd} →
  guessed-frame Ra xgny = Some gny →
  guessed-frame Ra xnr = Some Nr →
}

```

```

    can-signal s A B  $\longrightarrow$ 
      signalsResp s (Running A B  $\langle$ NonceF (Ra$ni), Nr, Exp gny (NonceF (Ra$nx)) $\rangle$ ) > 0
  }

```

```

lemmas skl1-inv5I = skl1-inv5-def [THEN setc-def-to-intro, rule-format]
lemmas skl1-inv5E [elim] = skl1-inv5-def [THEN setc-def-to-elim, rule-format]
lemmas skl1-inv5D = skl1-inv5-def [THEN setc-def-to-dest, rule-format, rotated 1, simplified]

```

```

lemma skl1-inv5-init [iff]:
  init skl1  $\subseteq$  skl1-inv5
by (auto simp add: skl1-def skl1-init-def skl1-inv5-def)

```

```

lemma skl1-inv5-trans [iff]:
  {skl1-inv5} trans skl1  $\{>$  skl1-inv5}
apply (auto simp add: PO-hoare-defs intro!: skl1-inv5I)
apply (drule can-signal-trans, assumption)
apply (auto simp add: skl1-nostep-defs)
apply (auto simp add: skl1-defs ik-dy-def dest: skl1-inv5D)
done

```

```

lemma PO-skl1-inv5 [iff]: reach skl1  $\subseteq$  skl1-inv5
by (rule inv-rule-basic) (auto)

```

### 26.5.3 inv6 (derived)

If a *Resp* run before step 4 and an *Init* run after step 3 both know the same half-keys (more or less), then the number of *Resp* running signals for the key is strictly greater than the number of *Resp* commit signals. (actually, there are 0 commit and 1 running).

#### definition

*skl1-inv6* :: *skl1-state set*

#### where

```

skl1-inv6  $\equiv$  {s.  $\forall$  A B Rb Ra gnx Ni.
  guessed-runs Ra = ( $\!$ role=Init, owner=A, partner=B)  $\longrightarrow$ 
  in-progressS (progress s Ra) {xnx, xni, xnr, xgnx, xgny, xsk, xEnd}  $\longrightarrow$ 
  guessed-frame Ra xgnx = Some gnx  $\longrightarrow$ 
  guessed-frame Ra xni = Some Ni  $\longrightarrow$ 
  guessed-frame Ra xgny = Some (Exp Gen (NonceF (Rb$ny)))  $\longrightarrow$ 
  guessed-frame Ra xnr = Some (NonceF (Rb$nr))  $\longrightarrow$ 
  guessed-runs Rb = ( $\!$ role=Resp, owner=B, partner=A)  $\longrightarrow$ 
  progress s Rb = Some {xny, xni, xnr, xgnx, xgny, xsk}  $\longrightarrow$ 
  can-signal s A B  $\longrightarrow$ 
  signalsResp s (Commit A B  $\langle$ Ni, NonceF (Rb$nr), Exp gnx (NonceF (Rb$ny)) $\rangle$ )
  < signalsResp s (Running A B  $\langle$ Ni, NonceF (Rb$nr), Exp gnx (NonceF (Rb$ny)) $\rangle$ )
}

```

```

lemmas skl1-inv6I = skl1-inv6-def [THEN setc-def-to-intro, rule-format]
lemmas skl1-inv6E [elim] = skl1-inv6-def [THEN setc-def-to-elim, rule-format]
lemmas skl1-inv6D = skl1-inv6-def [THEN setc-def-to-dest, rule-format, rotated 1, simplified]

```

```

lemma skl1-inv6-derived:
  skl1-inv4  $\cap$  skl1-inv5  $\subseteq$  skl1-inv6

```



```

proof (auto intro!: skl1-inv6I)
  fix s::skl1-state fix A B Rb Ra
  assume HRun:guessed-runs Ra = ( $\lambda$ role = Init, owner = A, partner = B)
    in-progressS (progress s Ra) {xnx, xni, xnr, xgnx, xgny, xsk, xEnd}
    guessed-frame Ra xgny = Some (Exp Gen (NonceF (Rb $ ny)))
    guessed-frame Ra xnr = Some (NonceF (Rb $ nr))
    can-signal s A B
  assume HRb: progress s Rb = Some {xny, xni, xnr, xgnx, xgny, xsk}
  assume I4:s  $\in$  skl1-inv4
  assume I5:s  $\in$  skl1-inv5
  from I4 HRb have signalsResp s (Commit A B  $\langle$ NonceF (Ra$ni), NonceF (Rb$nr),
    Exp (Exp Gen (NonceF (Rb $ ny))) (NonceF (Ra $ nx))))  $> 0 \implies$  False
  proof (auto dest!: skl1-inv4D)
    assume {xny, xni, xnr, xgnx, xgny, xsk, xEnd} = {xny, xni, xnr, xgnx, xgny, xsk}
    thus ?thesis by force
  qed
  then have HC:signalsResp s (Commit A B  $\langle$ NonceF (Ra$ni), NonceF (Rb$nr),
    Exp (Exp Gen (NonceF (Rb $ ny))) (NonceF (Ra $ nx)))) = 0
  by auto
  from I5 HRun have signalsResp s (Running A B  $\langle$ NonceF (Ra$ni), NonceF (Rb$nr),
    Exp (Exp Gen (NonceF (Rb $ ny))) (NonceF (Ra $ nx))))  $> 0$ 
  by (auto dest!: skl1-inv5D)
  with HC show signalsResp s (Commit A B  $\langle$ NonceF (Ra$ni), NonceF (Rb$nr),
    Exp (Exp Gen (NonceF (Rb $ ny))) (NonceF (Ra $ nx))))
  < signalsResp s (Running A B  $\langle$ NonceF (Ra$ni), NonceF (Rb$nr),
    Exp (Exp Gen (NonceF (Rb $ ny))) (NonceF (Ra $ nx))))
  by auto
qed

```

## 26.6 Refinement: injective agreement (Init authenticates Resp)

Mediator function.

**definition**

$med0sk1iai :: skl1-obs \Rightarrow a0i-obs$

**where**

$med0sk1iai t \equiv (\lambda a0n-state. signals = signalsInit t)$

Relation between states.

**definition**

$R0sk1iai :: (a0i-state * skl1-state)$  set

**where**

$R0sk1iai \equiv \{(s, s').$   
 $a0n-state.signals s = signalsInit s'$   
 $\}$

Protocol-independent events.

**lemma**  $skl1-learn-refines-a0-ia-skip-i:$

$\{R0sk1iai\} Id, l1-learn m \{>R0sk1iai\}$

**apply** (auto simp add: PO-rhoare-defs R0sk1iai-def)

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

**done**

Protocol events.

**lemma** *skl1-step1-refines-a0i-skip-i*:  
 $\{R0sk1iai\} Id, skl1-step1 Ra A B \{>R0sk1iai\}$   
**by** (*auto simp add: PO-rhoare-defs R0sk1iai-def skl1-step1-def*)

**lemma** *skl1-step2-refines-a0i-running-skip-i*:  
 $\{R0sk1iai\} a0i-running A B \langle Ni, NonceF (Rb\$nr), Exp gnx (NonceF (Rb\$ny)) \rangle \cup Id,$   
 $skl1-step2 Rb A B Ni gnx \{>R0sk1iai\}$   
**by** (*auto simp add: PO-rhoare-defs R0sk1iai-def,*  
*simp-all add: skl1-step2-def a0i-running-def, auto*)

**lemma** *skl1-step3-refines-a0i-commit-skip-i*:  
 $\{R0sk1iai \cap (UNIV \times skl1-inv3)\}$   
 $a0i-commit A B \langle NonceF (Ra\$ni), Nr, Exp gny (NonceF (Ra\$nx)) \rangle \cup Id,$   
 $skl1-step3 Ra A B Nr gny$   
 $\{>R0sk1iai\}$   
**apply** (*auto simp add: PO-rhoare-defs R0sk1iai-def*)  
**apply** (*auto simp add: skl1-step3-def a0i-commit-def*)  
**apply** (*frule skl1-inv3D, auto*)  
**done**

**lemma** *skl1-step4-refines-a0i-skip-i*:  
 $\{R0sk1iai\} Id, skl1-step4 Rb A B Ni gnx \{>R0sk1iai\}$   
**by** (*auto simp add: PO-rhoare-defs R0sk1iai-def, auto simp add: skl1-step4-def*)

refinement proof

**lemmas** *skl1-trans-refines-a0i-trans-i =*  
*skl1-learn-refines-a0-ia-skip-i*  
*skl1-step1-refines-a0i-skip-i skl1-step2-refines-a0i-running-skip-i*  
*skl1-step3-refines-a0i-commit-skip-i skl1-step4-refines-a0i-skip-i*

**lemma** *skl1-refines-init-a0i-i [iff]*:  
 $init skl1 \subseteq R0sk1iai \text{ “ } (init a0i)$   
**by** (*auto simp add: R0sk1iai-def a0i-defs skl1-defs*)

**lemma** *skl1-refines-trans-a0i-i [iff]*:  
 $\{R0sk1iai \cap (UNIV \times (skl1-inv1 \cap skl1-inv2))\} trans a0i, trans skl1 \{> R0sk1iai\}$   
**proof** –  
**let**  $?pre' = R0sk1iai \cap (UNIV \times skl1-inv3)$   
**show**  $?thesis (is \{?pre\} ?t1, ?t2 \{> ?post\})$   
**proof** (*rule relhoare-conseq-left*)  
**show**  $?pre \subseteq ?pre'$   
**using** *skl1-inv3-derived* **by** *blast*  
**next**  
**show**  $\{?pre'\} ?t1, ?t2 \{> ?post\}$   
**apply** (*auto simp add: a0i-def skl1-def a0i-trans-def skl1-trans-def*)  
**prefer** 2 **using** *skl1-step2-refines-a0i-running-skip-i* **apply** (*simp add: PO-rhoare-defs, blast*)  
**prefer** 2 **using** *skl1-step3-refines-a0i-commit-skip-i* **apply** (*simp add: PO-rhoare-defs, blast*)  
**apply** (*blast intro!: skl1-trans-refines-a0i-trans-i*)  
**done**

qed  
qed

**lemma** *obs-consistent-med01iai* [iff]:  
  *obs-consistent R0sk1iai med0sk1iai a0i skl1*  
**by** (*auto simp add: obs-consistent-def R0sk1iai-def med0sk1iai-def*)

refinement result

**lemma** *skl1-refines-a0i-i* [iff]:  
  *refines*  
  (*R0sk1iai*  $\cap$  (*reach a0i*  $\times$  (*skl1-inv1*  $\cap$  *skl1-inv2*)))  
  *med0sk1iai a0i skl1*  
**by** (*rule Refinement-using-invariants, auto*)

**lemma** *skl1-implements-a0i-i* [iff]: *implements med0sk1iai a0i skl1*  
**by** (*rule refinement-soundness*) (*fast*)

## 26.7 Derived invariants: injective agreement (*Init* authenticates *Resp*)

**lemma** *skl1-obs-iagreement-Init* [iff]: *oreach skl1*  $\subseteq$  *l1-iagreement-Init*  
**apply** (*rule external-invariant-translation*  
  [*OF PO-a0i-obs-agreement - skl1-implements-a0i-i*])  
**apply** (*auto simp add: med0sk1iai-def l1-iagreement-Init-def a0i-agreement-def*)  
**done**

**lemma** *skl1-iagreement-Init* [iff]: *reach skl1*  $\subseteq$  *l1-iagreement-Init*  
**by** (*rule external-to-internal-invariant [OF skl1-obs-iagreement-Init], auto*)

## 26.8 Refinement: injective agreement (*Resp* authenticates *Init*)

Mediator function.

**definition**  
  *med0sk1iar* :: *skl1-obs*  $\Rightarrow$  *a0i-obs*  
**where**  
  *med0sk1iar t*  $\equiv$  ( $\lambda a0n\text{-state}.\text{signals} = \text{signalsResp } t$ )

Relation between states.

**definition**  
  *R0sk1iar* :: (*a0i-state* \* *skl1-state*) *set*  
**where**  
  *R0sk1iar*  $\equiv$   $\{(s, s') .$   
    *a0n-state.signals s = signalsResp s'*  
  }

Protocol independent events.

**lemma** *skl1-learn-refines-a0-ia-skip-r*:  
  {*R0sk1iar*} *Id*, *l1-learn m*  $\{>$  *R0sk1iar*  
**apply** (*auto simp add: PO-rhoare-defs R0sk1iar-def*)  
**apply** (*simp add: l1-learn-def*)  
**done**

Protocol events.

**lemma** *skl1-step1-refines-a0i-skip-r*:

$\{R0sk1iar\} Id, skl1-step1 Ra A B \{>R0sk1iar\}$

**by** (*auto simp add: PO-rhoare-defs R0sk1iar-def skl1-step1-def*)

**lemma** *skl1-step2-refines-a0i-skip-r*:

$\{R0sk1iar\} Id, skl1-step2 Rb A B Ni gnx \{>R0sk1iar\}$

**by** (*auto simp add: PO-rhoare-defs R0sk1iar-def, auto simp add:skl1-step2-def*)

**lemma** *skl1-step3-refines-a0i-running-skip-r*:

$\{R0sk1iar\}$

$a0i-running A B \langle NonceF (Ra\$ni), Nr, Exp gny (NonceF (Ra\$nx)) \rangle \cup Id,$

$skl1-step3 Ra A B Nr gny$

$\{>R0sk1iar\}$

**by** (*auto simp add: PO-rhoare-defs R0sk1iar-def,*

*simp-all add: skl1-step3-def a0i-running-def, auto*)

**lemma** *skl1-step4-refines-a0i-commit-skip-r*:

$\{R0sk1iar \cap UNIV \times skl1-inv6\}$

$a0i-commit A B \langle Ni, NonceF (Rb\$nr), Exp gnx (NonceF (Rb\$ny)) \rangle \cup Id,$

$skl1-step4 Rb A B Ni gnx$

$\{>R0sk1iar\}$

**apply** (*auto simp add: PO-rhoare-defs R0sk1iar-def*)

**apply** (*auto simp add: skl1-step4-def a0i-commit-def*)

**apply** (*auto dest!: skl1-inv6D [rotated 1]*)

**done**

Refinement proof.

**lemmas** *skl1-trans-refines-a0i-trans-r =*

*skl1-learn-refines-a0-ia-skip-r*

*skl1-step1-refines-a0i-skip-r skl1-step2-refines-a0i-skip-r*

*skl1-step3-refines-a0i-running-skip-r skl1-step4-refines-a0i-commit-skip-r*

**lemma** *skl1-refines-init-a0i-r [iff]*:

*init skl1*  $\subseteq$  *R0sk1iar* “ (*init a0i*)

**by** (*auto simp add: R0sk1iar-def a0i-defs skl1-defs*)

**lemma** *skl1-refines-trans-a0i-r [iff]*:

$\{R0sk1iar \cap (UNIV \times (skl1-inv4 \cap skl1-inv5))\} trans a0i, trans skl1 \{> R0sk1iar\}$

**proof** –

**let** *?pre'* = *R0sk1iar*  $\cap$  (*UNIV*  $\times$  *skl1-inv6*)

**show** *?thesis* (*is*  $\{?pre\} ?t1, ?t2 \{> ?post\}$ )

**proof** (*rule relhoare-conseq-left*)

**show** *?pre*  $\subseteq$  *?pre'*

**using** *skl1-inv6-derived* **by** *blast*

**next**

**show**  $\{?pre'\} ?t1, ?t2 \{> ?post\}$

**apply** (*auto simp add: a0i-def skl1-def a0i-trans-def skl1-trans-def*)

**prefer** 3 **using** *skl1-step3-refines-a0i-running-skip-r* **apply** (*simp add: PO-rhoare-defs, blast*)

**prefer** 3 **using** *skl1-step4-refines-a0i-commit-skip-r* **apply** (*simp add: PO-rhoare-defs, blast*)

```

    apply (blast intro!:skl1-trans-refines-a0i-trans-r)+
  done
qed
qed

```

```

lemma obs-consistent-med0sk1iar [iff]:
  obs-consistent R0sk1iar med0sk1iar a0i skl1
by (auto simp add: obs-consistent-def R0sk1iar-def med0sk1iar-def)

```

Refinement result.

```

lemma skl1-refines-a0i-r [iff]:
  refines
  (R0sk1iar  $\cap$  (reach a0i  $\times$  (skl1-inv4  $\cap$  skl1-inv5)))
  med0sk1iar a0i skl1
by (rule Refinement-using-invariants, auto)

```

```

lemma skl1-implements-a0i-r [iff]: implements med0sk1iar a0i skl1
by (rule refinement-soundness) (fast)

```

## 26.9 Derived invariants: injective agreement (*Resp* authenticates *Init*)

```

lemma skl1-obs-iagreement-Resp [iff]: oreach skl1  $\subseteq$  l1-iagreement-Resp
apply (rule external-invariant-translation
  [OF PO-a0i-obs-agreement - skl1-implements-a0i-r])
apply (auto simp add: med0sk1iar-def l1-iagreement-Resp-def a0i-agreement-def)
done

```

```

lemma skl1-iagreement-Resp [iff]: reach skl1  $\subseteq$  l1-iagreement-Resp
by (rule external-to-internal-invariant [OF skl1-obs-iagreement-Resp], auto)

```

**end**

## 27 SKEME Protocol (L2)

```
theory sklv2
imports sklv1 Channels
begin
```

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

### 27.1 State and Events

Initial compromise.

```
consts
```

```
  bad-init :: agent set
```

```
specification (bad-init)
```

```
  bad-init-spec: test-owner  $\notin$  bad-init  $\wedge$  test-partner  $\notin$  bad-init
```

```
by auto
```

Level 2 state.

```
record l2-state =
```

```
  skl1-state +
```

```
  chan :: chan set
```

```
  bad :: agent set
```

```
type-synonym l2-obs = l2-state
```

```
type-synonym
```

```
  l2-pred = l2-state set
```

```
type-synonym
```

```
  l2-trans = (l2-state  $\times$  l2-state) set
```

Attacker events.

```
definition
```

```
  l2-dy-fake-msg :: msg  $\Rightarrow$  l2-trans
```

```
where
```

```
  l2-dy-fake-msg m  $\equiv$   $\{(s, s')\}$ .
```

```
  — guards
```

```
  m  $\in$  dy-fake-msg (bad s) (ik s) (chan s)  $\wedge$ 
```

```
  — actions
```

```
  s' = s(ik := {m}  $\cup$  ik s)
```

```
}
```

```
definition
```

```
  l2-dy-fake-chan :: chan  $\Rightarrow$  l2-trans
```

```
where
```

```
  l2-dy-fake-chan M  $\equiv$   $\{(s, s')\}$ .
```

```
  — guards
```

```
  M  $\in$  dy-fake-chan (bad s) (ik s) (chan s)  $\wedge$ 
```

```
  — actions
```

```
  s' = s(chan := {M}  $\cup$  chan s)
```

}

Partnering.

**fun**

*role-comp* :: *role-t*  $\Rightarrow$  *role-t*

**where**

*role-comp* *Init* = *Resp*

| *role-comp* *Resp* = *Init*

**definition**

*matching* :: *frame*  $\Rightarrow$  *frame*  $\Rightarrow$  *bool*

**where**

*matching* *sigma* *sigma'*  $\equiv \forall x. x \in \text{dom } \textit{sigma} \cap \text{dom } \textit{sigma}' \longrightarrow \textit{sigma } x = \textit{sigma}' x$

**definition**

*partner-runs* :: *rid-t*  $\Rightarrow$  *rid-t*  $\Rightarrow$  *bool*

**where**

*partner-runs* *R* *R'*  $\equiv$

*role* (*guessed-runs* *R*) = *role-comp* (*role* (*guessed-runs* *R'*))  $\wedge$

*owner* (*guessed-runs* *R*) = *partner* (*guessed-runs* *R'*)  $\wedge$

*owner* (*guessed-runs* *R'*) = *partner* (*guessed-runs* *R*)  $\wedge$

*matching* (*guessed-frame* *R*) (*guessed-frame* *R'*)

**lemma** *role-comp-inv* [*simp*]:

*role-comp* (*role-comp* *x*) = *x*

**by** (*cases* *x*, *auto*)

**lemma** *role-comp-inv-eq*:

*y* = *role-comp* *x*  $\longleftrightarrow$  *x* = *role-comp* *y*

**by** (*auto elim!*: *role-comp.elims* [*OF sym*])

**definition**

*partners* :: *rid-t* *set*

**where**

*partners*  $\equiv \{R. \textit{partner-runs test } R\}$

**lemma** *test-not-partner* [*simp*]:

*test*  $\notin$  *partners*

**by** (*auto simp add*: *partners-def partner-runs-def*, *cases* *role* (*guessed-runs test*), *auto*)

**lemma** *matching-symmetric*:

*matching* *sigma* *sigma'*  $\implies$  *matching* *sigma'* *sigma*

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

**lemma** *partner-symmetric*:

*partner-runs* *R* *R'*  $\implies$  *partner-runs* *R'* *R*

**by** (*auto simp add*: *partner-runs-def matching-symmetric*)

The unicity of the partner is actually protocol dependent: it only holds if there are generated fresh nonces (which identify the runs) in the frames

**lemma** *partner-unique*:

$partner\text{-}runs\ R\ R'' \implies partner\text{-}runs\ R\ R' \implies R' = R''$

**proof** –

**assume**  $H'$ : $partner\text{-}runs\ R\ R'$

**then have**  $Hm'$ :  $matching\ (guessed\text{-}frame\ R)\ (guessed\text{-}frame\ R')$

by (*auto simp add: partner-runs-def*)

**assume**  $H''$ : $partner\text{-}runs\ R\ R''$

**then have**  $Hm''$ :  $matching\ (guessed\text{-}frame\ R)\ (guessed\text{-}frame\ R'')$

by (*auto simp add: partner-runs-def*)

**show** *?thesis*

**proof** (*cases role (guessed-runs R')*)

**case** *Init*

**with**  $H'$  *partner-symmetric* [*OF H'*] **have**  $Hrole:role\ (guessed\text{-}runs\ R) = Resp$   
 $role\ (guessed\text{-}runs\ R'') = Init$

by (*auto simp add: partner-runs-def*)

**with** *Init*  $Hm'$  **have**  $guessed\text{-}frame\ R\ xgnx = Some\ (Exp\ Gen\ (NonceF\ (R'\$nx)))$

by (*simp add: matching-def*)

**moreover from**  $Hrole\ Hm''$  **have**  $guessed\text{-}frame\ R\ xgnx = Some\ (Exp\ Gen\ (NonceF\ (R''\$nx)))$

by (*simp add: matching-def*)

**ultimately show** *?thesis* by (*auto dest: Exp-Gen-inj*)

**next**

**case** *Resp*

**with**  $H'$  *partner-symmetric* [*OF H'*] **have**  $Hrole:role\ (guessed\text{-}runs\ R) = Init$   
 $role\ (guessed\text{-}runs\ R'') = Resp$

by (*auto simp add: partner-runs-def*)

**with** *Resp*  $Hm'$  **have**  $guessed\text{-}frame\ R\ xgny = Some\ (Exp\ Gen\ (NonceF\ (R'\$ny)))$

by (*simp add: matching-def*)

**moreover from**  $Hrole\ Hm''$  **have**  $guessed\text{-}frame\ R\ xgny = Some\ (Exp\ Gen\ (NonceF\ (R''\$ny)))$

by (*simp add: matching-def*)

**ultimately show** *?thesis* by (*auto dest: Exp-Gen-inj*)

**qed**

**qed**

**lemma** *partner-test*:

$R \in partners \implies partner\text{-}runs\ R\ R' \implies R' = test$

by (*auto intro!: partner-unique simp add: partners-def partner-symmetric*)

compromising events

**definition**

$l2\text{-lkr}\text{-others} :: agent \Rightarrow l2\text{-trans}$

**where**

$l2\text{-lkr}\text{-others}\ A \equiv \{(s, s')\}.$

— guards

$A \neq test\text{-}owner \wedge$

$A \neq test\text{-}partner \wedge$

— actions

$s' = s(bad := \{A\} \cup bad\ s)$

}

**definition**

$l2\text{-lkr}\text{-actor} :: agent \Rightarrow l2\text{-trans}$

**where**

$l2\text{-lkr}\text{-actor}\ A \equiv \{(s, s')\}.$

— guards



$$\begin{array}{l}
A = \text{test-owner} \wedge \\
A \neq \text{test-partner} \wedge \\
\text{— actions} \\
s' = s(\text{bad} := \{A\} \cup \text{bad } s) \\
\}
\end{array}$$

**definition**

$$l2\text{-lkr-after} :: \text{agent} \Rightarrow l2\text{-trans}$$

**where**

$$\begin{array}{l}
l2\text{-lkr-after } A \equiv \{(s, s')\}. \\
\text{— guards} \\
\text{test-ended } s \wedge \\
\text{— actions} \\
s' = s(\text{bad} := \{A\} \cup \text{bad } s) \\
\}
\end{array}$$

**definition**

$$l2\text{-skr} :: \text{rid-t} \Rightarrow \text{msg} \Rightarrow l2\text{-trans}$$

**where**

$$\begin{array}{l}
l2\text{-skr } R \ K \equiv \{(s, s')\}. \\
\text{— guards} \\
R \neq \text{test} \wedge R \notin \text{partners} \wedge \\
\text{in-progress } (\text{progress } s \ R) \ xsk \wedge \\
\text{guessed-frame } R \ xsk = \text{Some } K \wedge \\
\text{— actions} \\
s' = s(\text{ik} := \{K\} \cup \text{ik } s) \\
\}
\end{array}$$

Protocol events (with  $K = H(ni, nr)$ ):

- step 1: create  $Ra$ ,  $A$  generates  $nx$  and  $ni$ , confidentially sends  $ni$ , computes and insecurely sends  $g^{nx}$
- step 2: create  $Rb$ ,  $B$  receives  $ni$  (confidentially) and  $g^{nx}$  (insecurely), generates  $ny$  and  $nr$ , confidentially sends  $nr$ , insecurely sends  $g^{ny}$  and  $MAC_K(g^{nx}, g^{ny}, B, A)$  computes  $g^{nx * ny}$ , emits a running signal for  $Init, ni, nr, g^{nx * ny}$
- step 3:  $A$  receives  $nr$  confidentially, and  $g^{ny}$  and the MAC insecurely, sends  $MAC_K(g^{ny}, g^{nx}, A, B)$  insecurely, computes  $g^{ny * nx}$ , emits a commit signal for  $Init, ni, nr, g^{ny * nx}$ , a running signal for  $Resp, ni, nr, g^{ny * nx}$ , declares the secret  $g^{ny * nx}$
- step 4:  $B$  receives the MAC insecurely, emits a commit signal for  $Resp, ni, nr, g^{nx * ny}$ , declares the secret  $g^{nx * ny}$

**definition**

$$l2\text{-step1} :: \text{rid-t} \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow l2\text{-trans}$$

**where**

$$\begin{array}{l}
l2\text{-step1 } Ra \ A \ B \equiv \{(s, s')\}. \\
\text{— guards:} \\
Ra \notin \text{dom } (\text{progress } s) \wedge \\
\text{guessed-runs } Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \wedge \\
\text{— actions:} \\
s' = s(
\end{array}$$

```

    progress := (progress s)(Ra ↦ {xnx, xni, xgnx}),
    chan := {Confid A B (NonceF (Ra$ni))} ∪
           ({Insec A B (Exp Gen (NonceF (Ra$nx)))} ∪
            (chan s))
  }
}

```

**definition**

$l2\text{-step2} :: \text{rid-}t \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow \text{msg} \Rightarrow \text{msg} \Rightarrow l2\text{-trans}$

**where**

$l2\text{-step2} \text{ Rb } A \text{ B } Ni \text{ gnx} \equiv \{(s, s')\}.$

— guards:

$gessed\text{-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \wedge$   
 $Rb \notin \text{dom } (\text{progress } s) \wedge$   
 $gessed\text{-frame } Rb \text{ xgnx} = \text{Some } gnx \wedge$   
 $gessed\text{-frame } Rb \text{ xni} = \text{Some } Ni \wedge$   
 $gessed\text{-frame } Rb \text{ xsk} = \text{Some } (\text{Exp } gnx \text{ (NonceF (Rb$ny))}) \wedge$   
 $\text{Confid } A \text{ B } Ni \in \text{chan } s \wedge$   
 $\text{Insec } A \text{ B } gnx \in \text{chan } s \wedge$

— actions:

$s' = s(\text{progress} := (\text{progress } s)(Rb \mapsto \{xny, xni, xnr, xgny, xgnx, xsk\}),$   
 $\text{chan} := \{\text{Confid } B \text{ A } (\text{NonceF } (Rb\$nr))\} \cup$   
 $\{\text{Insec } B \text{ A}$   
 $\langle \text{Exp } \text{Gen } (\text{NonceF } (Rb\$ny)),$   
 $\text{hmac } \langle \text{Number } 0, \text{gnx}, \text{Exp } \text{Gen } (\text{NonceF } (Rb\$ny)), \text{Agent } B, \text{Agent } A \rangle$   
 $(\text{Hash } \langle Ni, \text{NonceF } (Rb\$nr) \rangle \rangle \} \cup$   
 $(\text{chan } s)),$   
 $\text{signalsInit} :=$   
 $\text{if can-signal } s \text{ A } B \text{ then}$   
 $\text{addSignal } (\text{signalsInit } s)$   
 $(\text{Running } A \text{ B } \langle Ni, \text{NonceF } (Rb\$nr), \text{Exp } gnx \text{ (NonceF } (Rb\$ny)) \rangle)$   
 $\text{else}$   
 $\text{signalsInit } s,$   
 $\text{signalsInit2} :=$   
 $\text{if can-signal } s \text{ A } B \text{ then}$   
 $\text{addSignal } (\text{signalsInit2 } s) (\text{Running } A \text{ B } (\text{Exp } gnx \text{ (NonceF } (Rb\$ny))))$   
 $\text{else}$   
 $\text{signalsInit2 } s$

**definition**

$l2\text{-step3} :: \text{rid-}t \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow \text{msg} \Rightarrow \text{msg} \Rightarrow l2\text{-trans}$

**where**

$l2\text{-step3} \text{ Ra } A \text{ B } Nr \text{ gny} \equiv \{(s, s')\}.$

— guards:

$gessed\text{-runs } Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \wedge$   
 $\text{progress } s \text{ Ra} = \text{Some } \{xnx, xni, xgnx\} \wedge$   
 $gessed\text{-frame } Ra \text{ xgny} = \text{Some } gny \wedge$   
 $gessed\text{-frame } Ra \text{ xnr} = \text{Some } Nr \wedge$   
 $gessed\text{-frame } Ra \text{ xsk} = \text{Some } (\text{Exp } gny \text{ (NonceF } (Ra$nx))) \wedge$

$Confid\ B\ A\ Nr \in chan\ s \wedge$   
 $Insec\ B\ A\ \langle gny, hmac\ \langle Number\ 0, Exp\ Gen\ (NonceF\ (Ra\$nx)), gny, Agent\ B, Agent\ A\rangle$   
 $\quad (Hash\ \langle NonceF\ (Ra\$ni), Nr\rangle)\rangle \in chan\ s \wedge$   
— actions:  
 $s' = s(\mid progress := (progress\ s)(Ra \mapsto \{xnx, xni, xnr, xgnx, xgny, xsk, xEnd\}),$   
 $\quad chan := \{Insec\ A\ B$   
 $\quad \quad (hmac\ \langle Number\ 1, gny, Exp\ Gen\ (NonceF\ (Ra\$nx)), Agent\ A, Agent\ B\rangle$   
 $\quad \quad (Hash\ \langle NonceF\ (Ra\$ni), Nr\rangle))\}$   
 $\quad \cup\ chan\ s,$   
 $secret := \{x. x = Exp\ gny\ (NonceF\ (Ra\$nx)) \wedge Ra = test\} \cup secret\ s,$   
 $signalsInit :=$   
 $\quad if\ can\ signal\ s\ A\ B\ then$   
 $\quad \quad addSignal\ (signalsInit\ s)$   
 $\quad \quad \quad (Commit\ A\ B\ \langle NonceF\ (Ra\$ni), Nr, Exp\ gny\ (NonceF\ (Ra\$nx))\rangle)$   
 $\quad else$   
 $\quad \quad signalsInit\ s,$   
 $signalsInit2 :=$   
 $\quad if\ can\ signal\ s\ A\ B\ then$   
 $\quad \quad addSignal\ (signalsInit2\ s)\ (Commit\ A\ B\ (Exp\ gny\ (NonceF\ (Ra\$nx))))$   
 $\quad else$   
 $\quad \quad signalsInit2\ s,$   
 $signalsResp :=$   
 $\quad if\ can\ signal\ s\ A\ B\ then$   
 $\quad \quad addSignal\ (signalsResp\ s)$   
 $\quad \quad \quad (Running\ A\ B\ \langle NonceF\ (Ra\$ni), Nr, Exp\ gny\ (NonceF\ (Ra\$nx))\rangle)$   
 $\quad else$   
 $\quad \quad signalsResp\ s,$   
 $signalsResp2 :=$   
 $\quad if\ can\ signal\ s\ A\ B\ then$   
 $\quad \quad addSignal\ (signalsResp2\ s)\ (Running\ A\ B\ (Exp\ gny\ (NonceF\ (Ra\$nx))))$   
 $\quad else$   
 $\quad \quad signalsResp2\ s$   
 $\mid$   
 $\}$

### definition

$l2\text{-step4} :: rid\text{-}t \Rightarrow agent \Rightarrow agent \Rightarrow msg \Rightarrow msg \Rightarrow l2\text{-}trans$

### where

$l2\text{-step4}\ Rb\ A\ B\ Ni\ gnx \equiv \{(s, s')\}.$

— guards:

$guessed\text{-}runs\ Rb = (\mid role=Resp, owner=B, partner=A) \wedge$

$progress\ s\ Rb = Some\ \{xny, xni, xnr, xgnx, xgny, xsk\} \wedge$

$guessed\text{-}frame\ Rb\ xgnx = Some\ gnx \wedge$

$guessed\text{-}frame\ Rb\ xni = Some\ Ni \wedge$

$Insec\ A\ B\ (hmac\ \langle Number\ 1, Exp\ Gen\ (NonceF\ (Rb\$ny)), gnx, Agent\ A, Agent\ B\rangle$

$\quad (Hash\ \langle Ni, NonceF\ (Rb\$nr)\rangle)) \in chan\ s \wedge$

— actions:

$s' = s(\mid progress := (progress\ s)(Rb \mapsto \{xny, xni, xnr, xgnx, xgny, xsk, xEnd\}),$

$secret := \{x. x = Exp\ gnx\ (NonceF\ (Rb\$ny)) \wedge Rb = test\} \cup secret\ s,$

$signalsResp :=$

$\quad if\ can\ signal\ s\ A\ B\ then$

```

      addSignal (signalsResp s)
        (Commit A B ⟨Ni, NonceF (Rb$nr), Exp gnx (NonceF (Rb$ny))⟩)
    else
      signalsResp s,
signalsResp2 :=
  if can-signal s A B then
    addSignal (signalsResp2 s) (Commit A B (Exp gnx (NonceF (Rb$ny))))
  else
    signalsResp2 s
  )
}

```

specification

**definition**

$l2\text{-init} :: l2\text{-state set}$

**where**

```

l2-init ≡ { ()
  ik = {},
  secret = {},
  progress = Map.empty,
  signalsInit = λx. 0,
  signalsResp = λx. 0,
  signalsInit2 = λx. 0,
  signalsResp2 = λx. 0,
  chan = {},
  bad = bad-init
}

```

**definition**

$l2\text{-trans} :: l2\text{-trans where}$

```

l2-trans ≡ (⋃ m M X Rb Ra A B K Y.
  l2-step1 Ra A B ∪
  l2-step2 Rb A B X Y ∪
  l2-step3 Ra A B X Y ∪
  l2-step4 Rb A B X Y ∪
  l2-dy-fake-chan M ∪
  l2-dy-fake-msg m ∪
  l2-lkr-others A ∪
  l2-lkr-after A ∪
  l2-skr Ra K ∪
  Id
)

```

**definition**

$l2 :: (l2\text{-state}, l2\text{-obs}) spec where$

```

l2 ≡ ()
  init = l2-init,
  trans = l2-trans,
  obs = id
)

```

**lemmas**  $l2\text{-loc-defs} =$

*l2-step1-def l2-step2-def l2-step3-def l2-step4-def*  
*l2-def l2-init-def l2-trans-def*  
*l2-dy-fake-chan-def l2-dy-fake-msg-def*  
*l2-lkr-after-def l2-lkr-others-def l2-skr-def*

**lemmas** *l2-defs = l2-loc-defs ik-dy-def*

**lemmas** *l2-nostep-defs = l2-def l2-init-def l2-trans-def*

**lemmas** *l2-step-defs =*

*l2-step1-def l2-step2-def l2-step3-def l2-step4-def*  
*l2-dy-fake-chan-def l2-dy-fake-msg-def l2-lkr-after-def l2-lkr-others-def l2-skr-def*

**lemma** *l2-obs-id [simp]: obs l2 = id*

**by** (*simp add: l2-def*)

Once a run is finished, it stays finished, therefore if the test is not finished at some point then it was not finished before either.

**declare** *domIff [iff]*

**lemma** *l2-run-ended-trans:*

*run-ended (progress s R)  $\implies$*

*(s, s')  $\in$  trans l2  $\implies$*

*run-ended (progress s' R)*

**apply** (*auto simp add: l2-nostep-defs*)

**apply** (*auto simp add: l2-defs*)

**done**

**declare** *domIff [iff del]*

**lemma** *l2-can-signal-trans:*

*can-signal s' A B  $\implies$*

*(s, s')  $\in$  trans l2  $\implies$*

*can-signal s A B*

**by** (*auto simp add: can-signal-def l2-run-ended-trans*)

**lemma** *in-progressS-trans:*

*in-progressS (progress s R) S  $\implies$  (s, s')  $\in$  trans l2  $\implies$  in-progressS (progress s' R) S*

**apply** (*auto simp add: l2-nostep-defs*)

**apply** (*auto simp add: l2-defs domIff*)

**done**

## 27.2 Invariants

### 27.2.1 inv1

If *can-signal s A B* (i.e., *A, B* are the test session agents and the test is not finished), then *A, B* are honest.

**definition**

*l2-inv1 :: l2-state set*

**where**

*l2-inv1  $\equiv$  {s.  $\forall A B.$*

*can-signal s A B  $\implies$*

*A  $\notin$  bad s  $\wedge$  B  $\notin$  bad s*

*}*

**lemmas**  $l2\text{-inv}1I = l2\text{-inv}1\text{-def}$  [*THEN setc-def-to-intro, rule-format*]  
**lemmas**  $l2\text{-inv}1E$  [*elim*] =  $l2\text{-inv}1\text{-def}$  [*THEN setc-def-to-elim, rule-format*]  
**lemmas**  $l2\text{-inv}1D = l2\text{-inv}1\text{-def}$  [*THEN setc-def-to-dest, rule-format, rotated 1, simplified*]

**lemma**  $l2\text{-inv}1\text{-init}$  [*iff*]:  
 $init\ l2 \subseteq l2\text{-inv}1$   
**by** (*auto simp add: l2-def l2-init-def l2-inv1-def can-signal-def bad-init-spec*)

**lemma**  $l2\text{-inv}1\text{-trans}$  [*iff*]:  
 $\{l2\text{-inv}1\}\ trans\ l2\ \{>\ l2\text{-inv}1\}$   
**proof** (*auto simp add: PO-hoare-defs intro!: l2-inv1I del: conjI*)  
**fix**  $s'\ s :: l2\text{-state}$   
**fix**  $A\ B$   
**assume**  $HI:s \in l2\text{-inv}1$   
**assume**  $HT:(s, s') \in trans\ l2$   
**assume**  $can\text{-signal}\ s'\ A\ B$   
**with**  $HT$  **have**  $HS:can\text{-signal}\ s\ A\ B$   
**by** (*auto simp add: l2-can-signal-trans*)  
**with**  $HI$  **have**  $A \notin bad\ s \wedge B \notin bad\ s$   
**by** *fast*  
**with**  $HS\ HT$  **show**  $A \notin bad\ s' \wedge B \notin bad\ s'$   
**by** (*auto simp add: l2-nostep-defs can-signal-def*)  
(*simp-all add: l2-defs*)  
**qed**

**lemma**  $PO\text{-}l2\text{-inv}1$  [*iff*]:  $reach\ l2 \subseteq l2\text{-inv}1$   
**by** (*rule inv-rule-basic*) (*auto*)

## 27.2.2 inv2

For a run  $R$  (with any role), the session key is always *something* <sup>$n$</sup>  where  $n$  is a nonce generated by  $R$ .

### definition

$l2\text{-inv}2 :: l2\text{-state}\ set$

### where

$l2\text{-inv}2 \equiv \{s. \forall R.$   
 $in\text{-progress}\ (progress\ s\ R)\ xsk \longrightarrow$   
 $(\exists\ X\ N.$   
 $guessed\text{-frame}\ R\ xsk = Some\ (Exp\ X\ (NonceF\ (R\$N))))$   
 $\}$

**lemmas**  $l2\text{-inv}2I = l2\text{-inv}2\text{-def}$  [*THEN setc-def-to-intro, rule-format*]  
**lemmas**  $l2\text{-inv}2E$  [*elim*] =  $l2\text{-inv}2\text{-def}$  [*THEN setc-def-to-elim, rule-format*]  
**lemmas**  $l2\text{-inv}2D = l2\text{-inv}2\text{-def}$  [*THEN setc-def-to-dest, rule-format, rotated 1, simplified*]

**lemma**  $l2\text{-inv}2\text{-init}$  [*iff*]:  
 $init\ l2 \subseteq l2\text{-inv}2$   
**by** (*auto simp add: l2-def l2-init-def l2-inv2-def*)

**lemma**  $l2\text{-inv}2\text{-trans}$  [*iff*]:  
 $\{l2\text{-inv}2\}\ trans\ l2\ \{>\ l2\text{-inv}2\}$

**apply** (*auto simp add: PO-hoare-defs l2-nostep-defs intro!: l2-inv2I*)  
**apply** (*auto simp add: l2-defs dy-fake-chan-def dest: l2-inv2D*)  
**done**

**lemma** *PO-l2-inv2 [iff]: reach l2  $\subseteq$  l2-inv2*  
**by** (*rule inv-rule-basic*) (*auto*)

### 27.2.3 inv3

#### definition

*bad-runs s = {R. owner (guessed-runs R)  $\in$  bad s  $\vee$  partner (guessed-runs R)  $\in$  bad s}*

#### abbreviation

*generators :: l2-state  $\Rightarrow$  msg set*

#### where

*generators s  $\equiv$*   
— from the *insec* messages in steps 1 2  
 $\{x. \exists N. x = \text{Exp Gen (Nonce N)}\} \cup$   
— from the opened *confid* messages in steps 1 2  
 $\{x. \exists R \in \text{bad-runs } s. x = \text{NonceF (R\$ni)} \vee x = \text{NonceF (R\$nr)}\} \cup$   
— from the *insec* messages in steps 2 3  
 $\{x. \exists y y' z. x = \text{hmac } \langle y, y' \rangle (\text{Hash } z)\} \cup$   
— from the *skr*  
 $\{\text{Exp } y (\text{NonceF (R\$N)}) \mid y N R. R \neq \text{test} \wedge R \notin \text{partners}\}$

**lemma** *analz-generators: analz (generators s) = generators s*  
**by** (*rule, rule, erule analz.induct*) (*auto*)

#### definition

*faked-chan-msgs :: l2-state  $\Rightarrow$  chan set*

#### where

*faked-chan-msgs s =*  
 $\{\text{Chan } x A B M \mid x A B M. M \in \text{synth (analz (extr (bad s) (ik s) (chan s)))}\}$

#### definition

*chan-generators :: chan set*

#### where

*chan-generators = {x.  $\exists n R$ . — the messages that can't be opened*  
 $x = \text{Confid (owner (guessed-runs R)) (partner (guessed-runs R)) (NonceF (R\$n))} \wedge$   
 $(n = \text{ni} \vee n = \text{nr})$   
 $\}$

#### definition

*l2-inv3 :: l2-state set*

#### where

*l2-inv3  $\equiv$  {s.*  
 $\text{extr (bad s) (ik s) (chan s)} \subseteq \text{synth (analz (generators s))} \wedge$   
 $\text{chan } s \subseteq \text{faked-chan-msgs } s \cup \text{chan-generators}$   
 $\}$

**lemmas** *l2-inv3-aux-defs = faked-chan-msgs-def chan-generators-def*

**lemmas**  $l2\text{-inv}3I = l2\text{-inv}3\text{-def}$  [*THEN setc-def-to-intro, rule-format*]  
**lemmas**  $l2\text{-inv}3E = l2\text{-inv}3\text{-def}$  [*THEN setc-def-to-elim, rule-format*]  
**lemmas**  $l2\text{-inv}3D = l2\text{-inv}3\text{-def}$  [*THEN setc-def-to-dest, rule-format, rotated 1, simplified*]

**lemma**  $l2\text{-inv}3\text{-init}$  [*iff*]:  
 $init\ l2 \subseteq l2\text{-inv}3$   
**by** (*auto simp add: l2-def l2-init-def l2-inv3-def*)

**lemma**  $l2\text{-inv}3\text{-step1}$ :  
 $\{l2\text{-inv}3\}\ l2\text{-step1}\ Ra\ A\ B\ \{\>\ l2\text{-inv}3\}$   
**apply** (*auto simp add: PO-hoare-defs l2-nostep-defs intro!: l2-inv3I*)  
**apply** (*auto simp add: l2-defs bad-runs-def intro: synth-analz-increasing dest!: l2-inv3D*)  
**apply** (*auto simp add: l2-inv3-aux-defs intro: synth-analz-monotone*  
 $dest!: subsetD$  [**where**  $A=chan\ -$ ])  
**done**

**lemma**  $l2\text{-inv}3\text{-step2}$ :  
 $\{l2\text{-inv}3\}\ l2\text{-step2}\ Rb\ A\ B\ Ni\ gnx\ \{\>\ l2\text{-inv}3\}$   
**apply** (*auto simp add: PO-hoare-defs l2-nostep-defs intro!: l2-inv3I*)  
**apply** (*auto simp add: l2-defs*)  
**apply** (*auto simp add: bad-runs-def intro: synth-analz-increasing dest!: l2-inv3D*)  
**apply** (*auto simp add: l2-inv3-aux-defs*) — SLOW, ca. 30s  
**apply** (*blast intro: synth-analz-monotone analz.intros insert-iff synth-analz-increasing*  
 $dest!: subsetD$  [**where**  $A=chan\ -$ ])+  
**done**

**lemma**  $l2\text{-inv}3\text{-step3}$ :  
 $\{l2\text{-inv}3\}\ l2\text{-step3}\ Ra\ A\ B\ Nr\ gny\ \{\>\ l2\text{-inv}3\}$   
**apply** (*auto simp add: PO-hoare-defs l2-nostep-defs intro!: l2-inv3I*)  
**apply** (*auto simp add: l2-defs bad-runs-def intro: synth-analz-increasing dest!: l2-inv3D*)  
**apply** (*auto simp add: l2-inv3-aux-defs*)  
**apply** (*blast intro: synth-analz-monotone dest!: subsetD* [**where**  $A=chan\ -$ ])+  
**done**

**lemma**  $l2\text{-inv}3\text{-step4}$ :  
 $\{l2\text{-inv}3\}\ l2\text{-step4}\ Rb\ A\ B\ Ni\ gnx\ \{\>\ l2\text{-inv}3\}$   
**apply** (*auto simp add: PO-hoare-defs l2-nostep-defs intro!: l2-inv3I, auto simp add: l2-defs*)  
**apply** (*auto simp add: bad-runs-def intro: synth-analz-increasing dest!: l2-inv3D*)  
**apply** (*auto simp add: l2-inv3-aux-defs dest!: subsetD* [**where**  $A=chan\ -$ ])  
**done**

**lemma**  $l2\text{-inv}3\text{-dy-fake-msg}$ :  
 $\{l2\text{-inv}3\}\ l2\text{-dy-fake-msg}\ M\ \{\>\ l2\text{-inv}3\}$   
**apply** (*auto simp add: PO-hoare-defs l2-defs extr-insert-IK-eq*  
 $intro!: l2\text{-inv}3I$   
 $elim!: l2\text{-inv}3E\ dy\text{-fake-msg-extr}$  [*THEN* [2]  $rev\text{-subsetD}$ ])  
**apply** (*auto intro!: fake-New*  
 $intro: synth\text{-analz-increasing}\ fake\text{-monotone}\ dy\text{-fake-msg-monotone}$   
 $dy\text{-fake-msg-insert-chan}$   
 $simp\ add: bad\text{-runs-def}\ elim!: l2\text{-inv}3E$ )  
**apply** (*auto simp add: l2-inv3-aux-defs intro: synth-analz-monotone*  
 $dest!: subsetD$  [**where**  $A=chan\ -$ ])



done

**lemma** *l2-inv3-dy-fake-chan:*

{*l2-inv3*} *l2-dy-fake-chan* *M* {> *l2-inv3*}

**apply** (*auto simp add: PO-hoare-defs l2-defs*

*intro!*: *l2-inv3I*

*elim!*: *l2-inv3E*)

**apply** (*auto intro: synth-analz-increasing simp add: bad-runs-def elim!*: *l2-inv3E*

*dest:dy-fake-msg-extr [THEN [2] rev-subsetD]*

*dy-fake-chan-extr-insert [THEN [2] rev-subsetD]*

*dy-fake-chan-mono2*)

**apply** (*simp add: l2-inv3-aux-defs dy-fake-chan-def dy-fake-msg-def,*

*erule fake.cases, simp-all*)

**apply** (*auto simp add: l2-inv3-aux-defs elim!*: *synth-analz-monotone*

*dest!*: *subsetD [where A=chan -]*)

done

**lemma** *l2-inv3-lkr-others:*

{*l2-inv3*} *l2-lkr-others* *A* {> *l2-inv3*}

**apply** (*auto simp add: PO-hoare-defs l2-defs*

*intro!*: *l2-inv3I*

*dest!*: *extr-insert-bad [THEN [2] rev-subsetD]*

*elim!*: *l2-inv3E*)

**apply** (*auto simp add: l2-inv3-aux-defs bad-runs-def*

*intro: synth-analz-increasing synth-analz-monotone*)

**apply** (*drule synth-analz-mono [where G=extr - - -], auto,*

*(drule rev-subsetD [where A=chan -], simp)+, auto intro: synth-analz-increasing,*

*drule rev-subsetD [where A=synth (analz (extr - - -))],*

*auto intro: synth-analz-monotone*)+

done

**lemma** *l2-inv3-lkr-after:*

{*l2-inv3*} *l2-lkr-after* *A* {> *l2-inv3*}

**apply** (*auto simp add: PO-hoare-defs l2-defs intro!*: *l2-inv3I*

*dest!*: *extr-insert-bad [THEN [2] rev-subsetD]*

*elim!*: *l2-inv3E*)

**apply** (*auto simp add: l2-inv3-aux-defs bad-runs-def*

*intro: synth-analz-increasing synth-analz-monotone*)

**apply** (*drule synth-analz-mono [where G=extr - - -], auto,*

*(drule rev-subsetD [where A=chan -], simp)+, auto intro: synth-analz-increasing,*

*drule rev-subsetD [where A=synth (analz (extr - - -))],*

*auto intro: synth-analz-monotone*)+

done

**lemma** *l2-inv3-skr:*

{*l2-inv3*  $\cap$  *l2-inv2*} *l2-skr* *R* *K* {> *l2-inv3*}

**apply** (*auto simp add: PO-hoare-defs l2-defs intro!*: *l2-inv3I dest!*: *l2-inv2D*)

**apply** (*auto simp add: l2-inv3-aux-defs bad-runs-def intro: synth-analz-increasing*

*elim!*: *l2-inv3E*)

**apply** (*blast intro: synth-analz-monotone dest!*: *subsetD [where A=chan -]*)+

done

**lemmas** *l2-inv3-trans-aux* =  
*l2-inv3-step1 l2-inv3-step2 l2-inv3-step3 l2-inv3-step4*  
*l2-inv3-dy-fake-msg l2-inv3-dy-fake-chan*  
*l2-inv3-lkr-others l2-inv3-lkr-after l2-inv3-skr*

**lemma** *l2-inv3-trans* [iff]:  
 $\{l2\text{-inv3} \cap l2\text{-inv2}\} \text{ trans } l2 \{> l2\text{-inv3}\}$   
**by** (*auto simp add: l2-nostep-defs intro:l2-inv3-trans-aux*)

**lemma** *PO-l2-inv3* [iff]: *reach l2*  $\subseteq$  *l2-inv3*  
**by** (*rule-tac J=l2-inv2 in inv-rule-incr*) (*auto*)

Auxiliary dest rule for *inv3*.

**lemmas** *l2-inv3D-aux* =  
*l2-inv3D [THEN conjunct1,*  
*THEN [2] subset-trans,*  
*THEN synth-analz-mono, simplified,*  
*THEN [2] rev-subsetD, rotated 1, OF IK-subset-extr]*

**lemma** *l2-inv3D-HashNonce1*:  
 $s \in l2\text{-inv3} \implies$   
 $\text{Hash} (\text{NonceF} (R\$N), X) \in \text{synth} (\text{analz} (\text{extr} (\text{bad } s) (\text{ik } s) (\text{chan } s))) \implies$   
 $R \in \text{bad-runs } s$   
**apply** (*drule l2-inv3D, auto, drule synth-analz-monotone, auto simp add: analz-generators*)  
**apply** (*erule synth.cases, auto*)  
**done**

**lemma** *l2-inv3D-HashNonce2*:  
 $s \in l2\text{-inv3} \implies$   
 $\text{Hash} (X, \text{NonceF} (R\$N)) \in \text{synth} (\text{analz} (\text{extr} (\text{bad } s) (\text{ik } s) (\text{chan } s))) \implies$   
 $R \in \text{bad-runs } s$   
**apply** (*drule l2-inv3D, auto, drule synth-analz-monotone, auto simp add: analz-generators*)  
**apply** (*erule synth.cases, auto*)  
**done**

#### 27.2.4 hmac preservation lemmas

If  $(s, s') \in TS.\text{trans } l2$  then the MACs (with secret keys) that the attacker knows in  $s'$  (overapproximated by those in  $\text{parts} (\text{extr} (\text{bad } s') (\text{ik } s') (\text{chan } s'))$ ) are already known in  $s$ , except in the case of the steps 2 and 3 of the protocol.

**lemma** *hmac-key-unknown*:  
 $\text{hmac } X K \in \text{synth} (\text{analz } H) \implies K \notin \text{synth} (\text{analz } H) \implies \text{hmac } X K \in \text{analz } H$   
**by** (*erule synth.cases, auto*)

**lemma** *parts-exp* [simp]:  $\text{parts} \{Exp X Y\} = \{Exp X Y\}$   
**by** (*auto,erule parts.induct, auto*)

**lemma** *hmac-trans-1-4-skr-extr-fake*:  
 $\text{hmac } X K \in \text{parts} (\text{extr} (\text{bad } s') (\text{ik } s') (\text{chan } s')) \implies$   
 $K \notin \text{synth} (\text{analz} (\text{extr} (\text{bad } s) (\text{ik } s) (\text{chan } s))) \implies$  — necessary for the *dy-fake-msg* case  
 $s \in l2\text{-inv2} \implies$  — necessary for the *skr* case

$(s, s') \in l2\text{-step1 } Ra \ A \ B \cup l2\text{-step4 } Rb \ A \ B \ Ni \ gnx \cup l2\text{-skr } R \ KK \cup$   
 $l2\text{-dy-fake-msg } M \cup l2\text{-dy-fake-chan } MM \implies$   
 $hmac \ X \ K \in parts \ (extr \ (bad \ s) \ (ik \ s) \ (chan \ s))$   
**apply** (auto simp add: l2-defs parts-insert [where  $H=extr \ - \ - \ -$ ]  
parts-insert [where  $H=insert \ - \ (extr \ - \ - \ -)$ ])  
**apply** (auto dest!: l2-inv2D)  
**apply** (auto dest!: dy-fake-chan-extr-insert-parts [THEN [2] rev-subsetD]  
parts-monotone [of  $- \ \{M\}$  synth (analz (extr (bad s) (ik s) (chan s)))],  
auto simp add: dy-fake-msg-def)  
**done**

**lemma** hmac-trans-2:

$hmac \ X \ K \in parts \ (extr \ (bad \ s') \ (ik \ s') \ (chan \ s')) \implies$   
 $(s, s') \in l2\text{-step2 } Rb \ A \ B \ Ni \ gnx \implies$   
 $hmac \ X \ K \in parts \ (extr \ (bad \ s) \ (ik \ s) \ (chan \ s)) \vee$   
 $(X = \langle Number \ 0, \ gnx, \ Exp \ Gen \ (NonceF \ (Rb\$ny)), \ Agent \ B, \ Agent \ A \rangle \wedge$   
 $K = Hash \ \langle Ni, \ NonceF \ (Rb\$nr) \rangle \wedge$   
 $guessed\text{-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \wedge$   
 $progress \ s' \ Rb = Some \ \{xny, \ xni, \ xnr, \ xgnx, \ xgny, \ xsk\} \wedge$   
 $guessed\text{-frame } Rb \ xgnx = Some \ gnx \wedge$   
 $guessed\text{-frame } Rb \ xni = Some \ Ni \ )$   
**apply** (auto simp add: l2-defs parts-insert [where  $H=extr \ - \ - \ -$ ]  
parts-insert [where  $H=insert \ - \ (extr \ - \ - \ -)$ ])  
**done**

**lemma** hmac-trans-3:

$hmac \ X \ K \in parts \ (extr \ (bad \ s') \ (ik \ s') \ (chan \ s')) \implies$   
 $(s, s') \in l2\text{-step3 } Ra \ A \ B \ Nr \ gny \implies$   
 $hmac \ X \ K \in parts \ (extr \ (bad \ s) \ (ik \ s) \ (chan \ s)) \vee$   
 $(X = \langle Number \ 1, \ gny, \ Exp \ Gen \ (NonceF \ (Ra\$nx)), \ Agent \ A, \ Agent \ B \rangle \wedge$   
 $K = Hash \ \langle NonceF \ (Ra\$ni), \ Nr \rangle \wedge$   
 $guessed\text{-runs } Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \wedge$   
 $progress \ s' \ Ra = Some \ \{xnx, \ xni, \ xnr, \ xgnx, \ xgny, \ xsk, \ xEnd\} \wedge$   
 $guessed\text{-frame } Ra \ xgny = Some \ gny \wedge$   
 $guessed\text{-frame } Ra \ xnr = Some \ Nr \ )$   
**apply** (auto simp add: l2-defs parts-insert [where  $H=extr \ - \ - \ -$ ]  
parts-insert [where  $H=insert \ - \ (extr \ - \ - \ -)$ ])  
**done**

**lemma** hmac-trans-lkr-aux:

$hmac \ X \ K \in parts \ \{M. \ \exists \ x \ A \ B. \ Chan \ x \ A \ B \ M \in \ chan \ s\} \implies$   
 $K \notin synth \ (analz \ (extr \ (bad \ s) \ (ik \ s) \ (chan \ s))) \implies$   
 $s \in l2\text{-inv3} \implies$   
 $hmac \ X \ K \in parts \ (extr \ (bad \ s) \ (ik \ s) \ (chan \ s))$

**proof** –

**assume**  $A:K \notin synth \ (analz \ (extr \ (bad \ s) \ (ik \ s) \ (chan \ s))) \ s \in l2\text{-inv3}$   
**assume**  $hmac \ X \ K \in parts \ \{M. \ \exists \ x \ A \ B. \ Chan \ x \ A \ B \ M \in \ chan \ s\}$   
**then obtain**  $x \ A \ B \ M$  **where**  $H:hmac \ X \ K \in parts \ \{M\}$  **and**  $H':Chan \ x \ A \ B \ M \in \ chan \ s$   
**by** (auto dest: parts-singleton)  
**assume**  $s \in l2\text{-inv3}$   
**with**  $H'$  **have**  $M \in range \ Nonce \vee M \in synth \ (analz \ (extr \ (bad \ s) \ (ik \ s) \ (chan \ s)))$   
**by** (auto simp add: l2-inv3-aux-defs dest!: l2-inv3D, auto)  
**with**  $H$  **show** ?thesis

**proof** (*auto*)  
**assume**  $M \in \text{synth} (\text{analz} (\text{extr} (\text{bad } s) (\text{ik } s) (\text{chan } s)))$   
**then have**  $\{M\} \subseteq \text{synth} (\text{analz} (\text{extr} (\text{bad } s) (\text{ik } s) (\text{chan } s)))$  **by** (*auto*)  
**then have parts**  $\{M\} \subseteq \text{parts} (\text{synth} (\text{analz} (\text{extr} (\text{bad } s) (\text{ik } s) (\text{chan } s))))$   
**by** (*rule parts-mono*)  
**with**  $H$  **have**  $\text{hmac } X K \in \text{parts} (\text{synth} (\text{analz} (\text{extr} (\text{bad } s) (\text{ik } s) (\text{chan } s))))$  **by** *auto*  
**with**  $A$  **show** *?thesis* **by** *auto*  
**qed**  
**qed**

**lemma** *hmac-trans-lkr*:  
 $\text{hmac } X K \in \text{parts} (\text{extr} (\text{bad } s') (\text{ik } s') (\text{chan } s')) \implies$   
 $K \notin \text{synth} (\text{analz} (\text{extr} (\text{bad } s) (\text{ik } s) (\text{chan } s))) \implies$   
 $s \in \text{l2-inv3} \implies$   
 $(s, s') \in \text{l2-lkr-others } A \cup \text{l2-lkr-after } A \implies$   
 $\text{hmac } X K \in \text{parts} (\text{extr} (\text{bad } s) (\text{ik } s) (\text{chan } s))$   
**apply** (*auto simp add: l2-defs*  
*dest!: parts-monotone [OF - extr-insert-bad]*)  
**apply** (*auto intro: parts-monotone intro!hmac-trans-lkr-aux*)  
**done**

**lemmas** *hmac-trans* = *hmac-trans-1-4-skr-extr-fake hmac-trans-lkr hmac-trans-2 hmac-trans-3*

### 27.2.5 inv4 (authentication guard)

If HMAC is  $\text{parts} (\text{extr} (\text{bad } s) (\text{ik } s) (\text{chan } s))$  and  $A, B$  are honest then the message has indeed been sent by a responder run (etc).

#### definition

*l2-inv4* :: *l2-state set*

#### where

$\text{l2-inv4} \equiv \{s. \forall Ra A B gny Nr.$

$\text{hmac} \langle \text{Number } 0, \text{Exp Gen} (\text{NonceF} (\text{Ra}\$nx)), gny, \text{Agent } B, \text{Agent } A \rangle$   
 $(\text{Hash} \langle \text{NonceF} (\text{Ra}\$ni), Nr \rangle) \in \text{parts} (\text{extr} (\text{bad } s) (\text{ik } s) (\text{chan } s)) \longrightarrow$   
 $\text{guessed-runs } Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \longrightarrow$   
 $A \notin \text{bad } s \wedge B \notin \text{bad } s \longrightarrow$   
 $(\exists Rb. \text{guessed-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \wedge$   
 $\text{in-progressS} (\text{progress } s Rb) \{xny, xni, xnr, xgnx, xgny, xsk\} \wedge$   
 $\text{guessed-frame } Rb \text{ xgny} = \text{Some } gny \wedge$   
 $\text{guessed-frame } Rb \text{ xnr} = \text{Some } Nr \wedge$   
 $\text{guessed-frame } Rb \text{ xni} = \text{Some} (\text{NonceF} (\text{Ra}\$ni)) \wedge$   
 $\text{guessed-frame } Rb \text{ xgnx} = \text{Some} (\text{Exp Gen} (\text{NonceF} (\text{Ra}\$nx))))$   
 $\}$

**lemmas** *l2-inv4I* = *l2-inv4-def [THEN setc-def-to-intro, rule-format]*

**lemmas** *l2-inv4E* [*elim*] = *l2-inv4-def [THEN setc-def-to-elim, rule-format]*

**lemmas** *l2-inv4D* = *l2-inv4-def [THEN setc-def-to-dest, rule-format, rotated 1, simplified]*

**lemma** *l2-inv4-init* [*iff*]:

$\text{init } l2 \subseteq \text{l2-inv4}$

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

**lemma** *l2-inv4-trans* [iff]:  
 $\{l2\text{-inv}4 \cap l2\text{-inv}2 \cap l2\text{-inv}3\} \text{ trans } l2 \{> l2\text{-inv}4\}$

**proof** (*auto simp add: PO-hoare-defs intro!: l2-inv4I*)  
**fix**  $s' s :: l2\text{-state}$   
**fix**  $Ra A B gny Nr$   
**assume**  $HH\text{parts}:\text{hmac} \langle \text{Number } 0, \text{Exp Gen} (\text{NonceF} (Ra \$ nx)), gny, \text{Agent } B, \text{Agent } A \rangle$   
 $(\text{Hash} \langle \text{NonceF} (Ra \$ ni), Nr \rangle)$   
 $\in \text{parts} (\text{extr} (\text{bad } s') (ik\ s') (\text{chan } s'))$   
**assume**  $HRa: \text{guessed-runs } Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B)$   
**assume**  $Hi:s \in l2\text{-inv}4\ s \in l2\text{-inv}2\ s \in l2\text{-inv}3$   
**assume**  $Ht:(s, s') \in \text{trans } l2$   
**assume**  $A \notin \text{bad } s'\ B \notin \text{bad } s'$   
**with**  $Ht$  **have**  $Hb:A \notin \text{bad } s\ B \notin \text{bad } s$   
**by** (*auto simp add: l2-nostep-defs*) (*simp-all add: l2-defs*)  
**with**  $HRa\ Hi$   
**have**  $HH:\text{Hash} \langle \text{NonceF} (Ra \$ ni), Nr \rangle \notin \text{synth} (\text{analz} (\text{extr} (\text{bad } s) (ik\ s) (\text{chan } s)))$   
**by** (*auto dest!: l2-inv3D-HashNonce1 simp add: bad-runs-def*)  
**from**  $Ht\ Hi\ HH\text{parts}\ HH$   
**have**  $\text{hmac} \langle \text{Number } 0, \text{Exp Gen} (\text{NonceF} (Ra \$ nx)), gny, \text{Agent } B, \text{Agent } A \rangle$   
 $(\text{Hash} \langle \text{NonceF} (Ra \$ ni), Nr \rangle) \in \text{parts} (\text{extr} (\text{bad } s) (ik\ s) (\text{chan } s)) \vee$   
 $(\exists Rb. (s, s') \in l2\text{-step}2\ Rb\ A\ B\ (\text{NonceF} (Ra \$ ni))\ (\text{Exp Gen} (\text{NonceF} (Ra \$ nx))) \wedge$   
 $gny = \text{Exp Gen} (\text{NonceF} (Rb \$ ny)) \wedge$   
 $Nr = \text{NonceF} (Rb \$ nr) \wedge$   
 $\text{guessed-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \wedge$   
 $\text{progress } s'\ Rb = \text{Some} \{xny, xni, xnr, xgnx, xgny, xsk\} \wedge$   
 $\text{guessed-frame } Rb\ xgnx = \text{Some} (\text{Exp Gen} (\text{NonceF} (Ra \$ nx))) \wedge$   
 $\text{guessed-frame } Rb\ xni = \text{Some} (\text{NonceF} (Ra \$ ni)))$   
**apply** (*auto simp add: l2-nostep-defs*)

**apply** (*drule hmac-trans-1-4-skr-extr-fake, auto*)  
**apply** (*drule hmac-trans-2, auto*)  
**apply** (*drule hmac-trans-3, auto*)  
**apply** (*drule hmac-trans-1-4-skr-extr-fake, auto*)  
**apply** (*drule hmac-trans-1-4-skr-extr-fake, auto*)  
**apply** (*drule hmac-trans-1-4-skr-extr-fake, auto*)  
**apply** (*drule hmac-trans-lkr, auto*)  
**apply** (*drule hmac-trans-lkr, auto*)  
**apply** (*drule hmac-trans-1-4-skr-extr-fake, auto*)  
**done**

**then show**  $\exists Rb. \text{guessed-runs } Rb = (\text{role} = \text{Resp}, \text{owner} = B, \text{partner} = A) \wedge$   
 $\text{in-progress}S (\text{progress } s'\ Rb) \{xny, xni, xnr, xgnx, xgny, xsk\} \wedge$   
 $\text{guessed-frame } Rb\ xgny = \text{Some } gny \wedge$   
 $\text{guessed-frame } Rb\ xnr = \text{Some } Nr \wedge$   
 $\text{guessed-frame } Rb\ xni = \text{Some} (\text{NonceF} (Ra \$ ni)) \wedge$   
 $\text{guessed-frame } Rb\ xgnx = \text{Some} (\text{Exp Gen} (\text{NonceF} (Ra \$ nx)))$

**proof** (*auto*)  
**assume**  
 $\text{hmac} \langle \text{Number } 0, \text{Exp Gen} (\text{NonceF} (Ra \$ nx)), gny, \text{Agent } B, \text{Agent } A \rangle$   
 $(\text{hmac} (\text{NonceF} (Ra \$ ni))\ Nr)$   
 $\in \text{parts} (\text{extr} (\text{bad } s) (ik\ s) (\text{chan } s))$   
**with**  $Hi\ Hb\ HRa$  **obtain**  $Rb$  **where**  
 $HRb: \text{guessed-runs } Rb = (\text{role} = \text{Resp}, \text{owner} = B, \text{partner} = A)$   
 $\text{in-progress}S (\text{progress } s\ Rb) \{xny, xni, xnr, xgnx, xgny, xsk\}$

guessed-frame  $Rb$   $xgny = \text{Some } gny$   
 guessed-frame  $Rb$   $xnr = \text{Some } Nr$   
 guessed-frame  $Rb$   $xni = \text{Some } (\text{NonceF } (Ra \$ ni))$   
 guessed-frame  $Rb$   $xgnx = \text{Some } (\text{Exp Gen } (\text{NonceF } (Ra \$ nx)))$   
**by** (*auto dest!*:  $l2\text{-inv}4D$ )  
**with**  $Ht$  **have**  $in\text{-progress}S$  ( $progress\ s'\ Rb$ )  $\{xny, xni, xnr, xgnx, xgny, xsk\}$   
**by** (*auto elim*:  $in\text{-progress}S\text{-trans}$ )  
**with**  $HRb$  **show** *?thesis* **by** *auto*  
**qed**  
**qed**

**lemma**  $PO\text{-}l2\text{-inv}4$  [*iff*]:  $reach\ l2 \subseteq l2\text{-inv}4$   
**by** (*rule-tac*  $J=l2\text{-inv}2 \cap l2\text{-inv}3$  **in**  $inv\text{-rule-incr}$ ) (*auto*)

**lemma**  $auth\text{-guard}\text{-step}3$ :

$s \in l2\text{-inv}4 \implies$   
 $s \in l2\text{-inv}1 \implies$   
 $Insec\ B\ A \langle gny, hmac \langle \text{Number } 0, \text{Exp Gen } (\text{NonceF } (Ra \$ nx)), gny, \text{Agent } B, \text{Agent } A \rangle$   
 $(\text{Hash } \langle \text{NonceF } (Ra \$ ni), Nr \rangle) \rangle$   
 $\in chan\ s \implies$   
 $guessed\text{-runs } Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \implies$   
 $can\text{-signal } s\ A\ B \implies$   
 $(\exists\ Rb. guessed\text{-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \wedge$   
 $in\text{-progress}S\ (progress\ s\ Rb)\ \{xny, xni, xnr, xgnx, xgny, xsk\} \wedge$   
 $guessed\text{-frame } Rb\ xgny = \text{Some } gny \wedge$   
 $guessed\text{-frame } Rb\ xnr = \text{Some } Nr \wedge$   
 $guessed\text{-frame } Rb\ xni = \text{Some } (\text{NonceF } (Ra \$ ni)) \wedge$   
 $guessed\text{-frame } Rb\ xgnx = \text{Some } (\text{Exp Gen } (\text{NonceF } (Ra \$ nx))))$

**proof** –

**assume**  $s \in l2\text{-inv}1\ can\text{-signal } s\ A\ B$   
**hence**  $Hb:A \notin bad\ s\ B \notin bad\ s$  **by** *auto*  
**assume**  $Insec\ B\ A \langle gny, hmac \langle \text{Number } 0, \text{Exp Gen } (\text{NonceF } (Ra \$ nx)), gny, \text{Agent } B, \text{Agent } A \rangle$   
 $(\text{Hash } \langle \text{NonceF } (Ra \$ ni), Nr \rangle) \rangle \in chan\ s$

**hence**  $HH$ :

$hmac \langle \text{Number } 0, \text{Exp Gen } (\text{NonceF } (Ra \$ nx)), gny, \text{Agent } B, \text{Agent } A \rangle$   
 $(\text{Hash } \langle \text{NonceF } (Ra \$ ni), Nr \rangle)$   
 $\in parts\ (extr\ (bad\ s)\ (ik\ s)\ (chan\ s))$  **by** *auto*

**assume**  $s \in l2\text{-inv}4\ guessed\text{-runs } Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B)$

**with**  $Hb\ HH$  **show** *?thesis* **by** *auto*

**qed**

## 27.2.6 $inv5$ (authentication guard)

If MAC is in  $parts\ (extr\ (bad\ s)\ (ik\ s)\ (chan\ s))$  and  $A, B$  are honest then the message has indeed been sent by an initiator run (etc).

**definition**

$l2\text{-inv}5 :: l2\text{-state set}$

**where**

$l2\text{-inv}5 \equiv \{s. \forall\ Rb\ A\ B\ gnx\ Ni.$

$hmac \langle \text{Number } 1, \text{Exp Gen } (\text{NonceF } (Rb \$ ny)), gnx, \text{Agent } A, \text{Agent } B \rangle$   
 $(\text{Hash } \langle Ni, \text{NonceF } (Rb \$ nr) \rangle) \in parts\ (extr\ (bad\ s)\ (ik\ s)\ (chan\ s)) \implies$

$gessed\text{-}runs\ Rb = \langle role=Resp, owner=B, partner=A \rangle \longrightarrow$   
 $A \notin bad\ s \wedge B \notin bad\ s \longrightarrow$   
 $(\exists Ra. gessed\text{-}runs\ Ra = \langle role=Init, owner=A, partner=B \rangle \wedge$   
 $in\text{-}progressS\ (progress\ s\ Ra)\ \{xnx, xni, xnr, xgnx, xgny, xsk, xEnd\} \wedge$   
 $gessed\text{-}frame\ Ra\ xgnx = Some\ gnx \wedge$   
 $gessed\text{-}frame\ Ra\ xni = Some\ Ni \wedge$   
 $gessed\text{-}frame\ Ra\ xnr = Some\ (NonceF\ (Rb\$nr)) \wedge$   
 $gessed\text{-}frame\ Ra\ xgny = Some\ (Exp\ Gen\ (NonceF\ (Rb\$ny))))$   
 $\}$

**lemmas**  $l2\text{-}inv5I = l2\text{-}inv5\text{-}def\ [THEN\ setc\text{-}def\text{-}to\text{-}intro, rule\text{-}format]$

**lemmas**  $l2\text{-}inv5E [elim] = l2\text{-}inv5\text{-}def\ [THEN\ setc\text{-}def\text{-}to\text{-}elim, rule\text{-}format]$

**lemmas**  $l2\text{-}inv5D = l2\text{-}inv5\text{-}def\ [THEN\ setc\text{-}def\text{-}to\text{-}dest, rule\text{-}format, rotated\ 1, simplified]$

**lemma**  $l2\text{-}inv5\text{-}init\ [iff]:$

$init\ l2 \subseteq l2\text{-}inv5$

**by**  $(auto\ simp\ add: l2\text{-}def\ l2\text{-}init\text{-}def\ l2\text{-}inv5\text{-}def)$

**lemma**  $l2\text{-}inv5\text{-}trans\ [iff]:$

$\{l2\text{-}inv5 \cap l2\text{-}inv2 \cap l2\text{-}inv3\}\ trans\ l2\ \{>\ l2\text{-}inv5\}$

**proof**  $(auto\ simp\ add: PO\text{-}hoare\text{-}defs\ intro!: l2\text{-}inv5I)$

**fix**  $s'\ s :: l2\text{-}state$

**fix**  $Rb\ A\ B\ gnx\ Ni$

**assume**  $HHparts:hmac\ \langle Number\ (Suc\ 0), Exp\ Gen\ (NonceF\ (Rb\$ny)), gnx, Agent\ A, Agent\ B \rangle$   
 $(Hash\ \langle Ni, NonceF\ (Rb\$nr) \rangle)$

$\in parts\ (extr\ (bad\ s')\ (ik\ s')\ (chan\ s'))$

**assume**  $HRb: gessed\text{-}runs\ Rb = \langle role=Resp, owner=B, partner=A \rangle$

**assume**  $Hi: s \in l2\text{-}inv5\ s \in l2\text{-}inv2\ s \in l2\text{-}inv3$

**assume**  $Ht: (s, s') \in trans\ l2$

**assume**  $A \notin bad\ s'\ B \notin bad\ s'$

**with**  $Ht$  **have**  $Hb: A \notin bad\ s\ B \notin bad\ s$

**by**  $(auto\ simp\ add: l2\text{-}nostep\text{-}defs)\ (simp\text{-}all\ add: l2\text{-}defs)$

**with**  $HRb\ Hi$  **have**  $HH: Hash\ \langle Ni, NonceF\ (Rb\$nr) \rangle \notin synth\ (analz\ (extr\ (bad\ s)\ (ik\ s)\ (chan\ s)))$

**by**  $(auto\ dest!: l2\text{-}inv3D\text{-}HashNonce2\ simp\ add: bad\text{-}runs\text{-}def)$

**from**  $Ht\ Hi\ HHparts\ HH$  **have**  $hmac\ \langle Number\ 1, Exp\ Gen\ (NonceF\ (Rb\$ny)), gnx, Agent\ A, Agent\ B \rangle$

$(Hash\ \langle Ni, NonceF\ (Rb\$nr) \rangle) \in parts\ (extr\ (bad\ s)\ (ik\ s)\ (chan\ s)) \vee$

$(\exists Ra. (s, s') \in l2\text{-}step3\ Ra\ A\ B\ (NonceF\ (Rb\$nr))\ (Exp\ Gen\ (NonceF\ (Rb\$ny))) \wedge$

$gnx = Exp\ Gen\ (NonceF\ (Ra\$nx)) \wedge$

$Ni = NonceF\ (Ra\$ni) \wedge$

$gessed\text{-}runs\ Ra = \langle role=Init, owner=A, partner=B \rangle \wedge$

$progress\ s'\ Ra = Some\ \{xnx, xni, xnr, xgnx, xgny, xsk, xEnd\} \wedge$

$gessed\text{-}frame\ Ra\ xgny = Some\ (Exp\ Gen\ (NonceF\ (Rb\$ny))) \wedge$

$gessed\text{-}frame\ Ra\ xnr = Some\ (NonceF\ (Rb\$nr))$

**apply**  $(auto\ simp\ add: l2\text{-}nostep\text{-}defs)$

**apply**  $(drule\ hmac\text{-}trans\text{-}1\text{-}4\text{-}skr\text{-}extr\text{-}fake, auto)$

**apply**  $(drule\ hmac\text{-}trans\text{-}2, auto)$

**apply**  $(drule\ hmac\text{-}trans\text{-}3, auto)$

**apply**  $(drule\ hmac\text{-}trans\text{-}1\text{-}4\text{-}skr\text{-}extr\text{-}fake, auto)$

**apply**  $(drule\ hmac\text{-}trans\text{-}1\text{-}4\text{-}skr\text{-}extr\text{-}fake, auto)$

**apply**  $(drule\ hmac\text{-}trans\text{-}1\text{-}4\text{-}skr\text{-}extr\text{-}fake, auto)$

**apply**  $(drule\ hmac\text{-}trans\text{-}lkr, auto)$

**apply** (*drule hmac-trans-lkr, auto*)  
**apply** (*drule hmac-trans-1-4-skr-extr-fake, auto*)  
**done**  
**then show**  $\exists Ra. \text{ guessed-runs } Ra = (\text{role=Init, owner=A, partner=B}) \wedge$   
 $\text{in-progressS } (\text{progress } s' Ra) \{xnx, xni, xnr, xgnx, xgny, xsk, xEnd\} \wedge$   
 $\text{guessed-frame } Ra \ xgnx = \text{Some } gnx \wedge$   
 $\text{guessed-frame } Ra \ xni = \text{Some } Ni \wedge$   
 $\text{guessed-frame } Ra \ xnr = \text{Some } (\text{NonceF } (Rb\$nr)) \wedge$   
 $\text{guessed-frame } Ra \ xgny = \text{Some } (\text{Exp Gen } (\text{NonceF } (Rb\$ny)))$   
**proof** (*auto*)  
**assume**  
 $\text{hmac } \langle \text{Number } (Suc \ 0), \text{Exp Gen } (\text{NonceF } (Rb\$ny)), gnx, \text{Agent } A, \text{Agent } B \rangle$   
 $(\text{Hash } \langle Ni, \text{NonceF } (Rb\$nr) \rangle)$   
 $\in \text{parts } (\text{extr } (\text{bad } s) (\text{ik } s) (\text{chan } s))$   
**with** *Hi Hb HRb obtain Ra where HRA:guessed-runs Ra = (role=Init, owner=A, partner=B)*  
 $\text{in-progressS } (\text{progress } s Ra) \{xnx, xni, xnr, xgnx, xgny, xsk, xEnd\}$   
 $\text{guessed-frame } Ra \ xgnx = \text{Some } gnx$   
 $\text{guessed-frame } Ra \ xni = \text{Some } Ni$   
 $\text{guessed-frame } Ra \ xnr = \text{Some } (\text{NonceF } (Rb\$nr))$   
 $\text{guessed-frame } Ra \ xgny = \text{Some } (\text{Exp Gen } (\text{NonceF } (Rb\$ny)))$   
**by** (*auto dest!: l2-inv5D*)  
**with** *Ht have in-progressS (progress s' Ra) {xnx, xni, xnr, xgnx, xgny, xsk, xEnd}*  
**by** (*auto elim: in-progressS-trans*)  
**with HRA show ?thesis by auto**  
**qed**  
**qed**

**lemma** *PO-l2-inv5 [iff]: reach l2  $\subseteq$  l2-inv5*  
**by** (*rule-tac J=l2-inv2  $\cap$  l2-inv3 in inv-rule-incr*) (*auto*)

**lemma** *auth-guard-step4:*

$s \in l2\text{-inv5} \implies$

$s \in l2\text{-inv1} \implies$

$\text{Insec } A \ B \ (\text{hmac } \langle \text{Number } 1, \text{Exp Gen } (\text{NonceF } (Rb\$ny)), gnx, \text{Agent } A, \text{Agent } B \rangle$   
 $(\text{Hash } \langle Ni, \text{NonceF } (Rb\$nr) \rangle))$

$\in \text{chan } s \implies$

$\text{guessed-runs } Rb = (\text{role=Resp, owner=B, partner=A}) \implies$

$\text{can-signal } s \ A \ B \implies$

$(\exists Ra. \text{ guessed-runs } Ra = (\text{role=Init, owner=A, partner=B}) \wedge$   
 $\text{in-progressS } (\text{progress } s Ra) \{xnx, xni, xnr, xgnx, xgny, xsk, xEnd\} \wedge$   
 $\text{guessed-frame } Ra \ xgnx = \text{Some } gnx \wedge$   
 $\text{guessed-frame } Ra \ xni = \text{Some } Ni \wedge$   
 $\text{guessed-frame } Ra \ xnr = \text{Some } (\text{NonceF } (Rb\$nr)) \wedge$   
 $\text{guessed-frame } Ra \ xgny = \text{Some } (\text{Exp Gen } (\text{NonceF } (Rb\$ny)))$ )

**proof** –

**assume**  $s \in l2\text{-inv1} \ \text{can-signal } s \ A \ B$

**hence**  $Hb:A \notin \text{bad } s \ B \notin \text{bad } s$  **by** *auto*

**assume**  $\text{Insec } A \ B \ (\text{hmac } \langle \text{Number } 1, \text{Exp Gen } (\text{NonceF } (Rb\$ny)), gnx, \text{Agent } A, \text{Agent } B \rangle$   
 $(\text{Hash } \langle Ni, \text{NonceF } (Rb\$nr) \rangle)) \in \text{chan } s$

**hence** *HH:*

$\text{hmac } \langle \text{Number } 1, \text{Exp Gen } (\text{NonceF } (Rb\$ny)), gnx, \text{Agent } A, \text{Agent } B \rangle$   
 $(\text{Hash } \langle Ni, \text{NonceF } (Rb\$nr) \rangle)$



$\in$  parts (extr (bad s) (ik s) (chan s)) by auto  
**assume**  $s \in l2\text{-inv}5$  guessed-runs  $Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A)$   
**with**  $Hb$   $HH$  **show** ?thesis by auto  
**qed**

### 27.2.7 inv6

For an initiator, the session key is always  $gny^{nx}$ .

#### definition

$l2\text{-inv}6 :: l2\text{-state set}$

#### where

$l2\text{-inv}6 \equiv \{s. \forall Ra A B gny.$   
 guessed-runs  $Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \longrightarrow$   
 in-progress (progress s  $Ra$ )  $xsk \longrightarrow$   
 guessed-frame  $Ra$   $xgny = \text{Some } gny \longrightarrow$   
 guessed-frame  $Ra$   $xsk = \text{Some } (\text{Exp } gny (\text{NonceF } (Ra\$nx)))$   
 $\}$

**lemmas**  $l2\text{-inv}6I = l2\text{-inv}6\text{-def}$  [THEN setc-def-to-intro, rule-format]

**lemmas**  $l2\text{-inv}6E$  [elim] =  $l2\text{-inv}6\text{-def}$  [THEN setc-def-to-elim, rule-format]

**lemmas**  $l2\text{-inv}6D = l2\text{-inv}6\text{-def}$  [THEN setc-def-to-dest, rule-format, rotated 1, simplified]

**lemma**  $l2\text{-inv}6\text{-init}$  [iff]:

$\text{init } l2 \subseteq l2\text{-inv}6$

**by** (auto simp add:  $l2\text{-def}$   $l2\text{-init-def}$   $l2\text{-inv}6\text{-def}$ )

**lemma**  $l2\text{-inv}6\text{-trans}$  [iff]:

$\{l2\text{-inv}6\}$  trans  $l2 \{> l2\text{-inv}6\}$

**apply** (auto simp add: PO-hoare-defs  $l2\text{-nostep-defs}$  intro!:  $l2\text{-inv}6I$ )

**apply** (auto simp add:  $l2\text{-defs}$  dy-fake-chan-def)

**done**

**lemma**  $PO\text{-}l2\text{-inv}6$  [iff]:  $\text{reach } l2 \subseteq l2\text{-inv}6$

**by** (rule inv-rule-basic) (auto)

### 27.2.8 inv6'

For a responder, the session key is always  $gnx^{ny}$ .

#### definition

$l2\text{-inv}6' :: l2\text{-state set}$

#### where

$l2\text{-inv}6' \equiv \{s. \forall Rb A B gnx.$   
 guessed-runs  $Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \longrightarrow$   
 in-progress (progress s  $Rb$ )  $xsk \longrightarrow$   
 guessed-frame  $Rb$   $xgnx = \text{Some } gnx \longrightarrow$   
 guessed-frame  $Rb$   $xsk = \text{Some } (\text{Exp } gnx (\text{NonceF } (Rb\$ny)))$   
 $\}$

**lemmas**  $l2\text{-inv}6'I = l2\text{-inv}6'\text{-def}$  [THEN setc-def-to-intro, rule-format]

**lemmas**  $l2\text{-inv}6'E$  [elim] =  $l2\text{-inv}6'\text{-def}$  [THEN setc-def-to-elim, rule-format]

**lemmas**  $l2\text{-inv}6'D = l2\text{-inv}6'\text{-def}$  [THEN setc-def-to-dest, rule-format, rotated 1, simplified]

**lemma** *l2-inv6'-init* [iff]:  
*init*  $l2 \subseteq l2\text{-inv6}'$   
**by** (*auto simp add: l2-def l2-init-def l2-inv6'-def*)

**lemma** *l2-inv6'-trans* [iff]:  
 $\{l2\text{-inv6}'\}$  *trans*  $l2 \{> l2\text{-inv6}'\}$   
**apply** (*auto simp add: PO-hoare-defs l2-nostep-defs intro!: l2-inv6'I*)  
**apply** (*auto simp add: l2-defs dy-fake-chan-def*)  
**done**

**lemma** *PO-l2-inv6'* [iff]: *reach*  $l2 \subseteq l2\text{-inv6}'$   
**by** (*rule inv-rule-basic*) (*auto*)

### 27.2.9 inv7: form of the secrets

#### definition

*l2-inv7* :: *l2-state set*

#### where

$l2\text{-inv7} \equiv \{s.$   
 $\text{secret } s \subseteq \{Exp (Exp \text{ Gen } (NonceF (R\$N))) (NonceF (R'\$N')) \mid N N' R R'.$   
 $R = \text{test} \wedge R' \in \text{partners} \wedge (N = nx \vee N = ny) \wedge (N' = nx \vee N' = ny)\}$   
 $\}$

**lemmas** *l2-inv7I* = *l2-inv7-def* [THEN *setc-def-to-intro*, *rule-format*]

**lemmas** *l2-inv7E* [elim] = *l2-inv7-def* [THEN *setc-def-to-elim*, *rule-format*]

**lemmas** *l2-inv7D* = *l2-inv7-def* [THEN *setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

**lemma** *l2-inv7-init* [iff]:  
*init*  $l2 \subseteq l2\text{-inv7}$   
**by** (*auto simp add: l2-def l2-init-def l2-inv7-def*)

Steps 3 and 4 are the hard part.

**lemma** *l2-inv7-step3*:  
 $\{l2\text{-inv7} \cap l2\text{-inv1} \cap l2\text{-inv4} \cap l2\text{-inv6}'\}$  *l2-step3*  $Ra A B Nr gny \{> l2\text{-inv7}\}$   
**proof** (*auto simp add: PO-hoare-defs intro!: l2-inv7I*)  
**fix**  $s s' :: l2\text{-state}$  **fix**  $x$   
**assume**  $Hi:s \in l2\text{-inv1} \ s \in l2\text{-inv7} \ s \in l2\text{-inv4} \ s \in l2\text{-inv6}'$   
**assume**  $Ht:(s, s') \in l2\text{-step3} \ Ra A B Nr gny$   
**assume**  $Hs:x \in \text{secret } s'$   
**from**  $Hs \ Ht$  **have**  $x \in \text{secret } s \vee (Ra = \text{test} \wedge x = Exp \ gny \ (NonceF \ (Ra\$nx)))$   
**by** (*auto simp add: l2-defs*)  
**with**  $Hi \ Ht$   
**show**  $\exists N N' R'. x = Exp \ (Exp \ \text{Gen} \ (NonceF \ (R' \$ N'))) \ (NonceF \ (\text{test} \$ N)) \wedge$   
 $R' \in \text{partners} \wedge (N = nx \vee N = ny) \wedge (N' = nx \vee N' = ny)$   
**proof** (*auto dest: l2-inv7D simp add: l2-defs*)  
**assume**  $Htest: \text{guessed-runs } \text{test} = (\text{role} = \text{Init}, \text{owner} = A, \text{partner} = B)$   
 $\text{guessed-frame } \text{test} \ xgny = \text{Some } gny$   
 $\text{guessed-frame } \text{test} \ xnr = \text{Some } Nr$   
 $\text{guessed-frame } \text{test} \ xsk = \text{Some} \ (Exp \ gny \ (NonceF \ (\text{test} \$ nx)))$   
**assume**  
 $\text{Insec } B \ A \ \langle gny, \text{hmac} \ \langle \text{Number } 0, Exp \ \text{Gen} \ (NonceF \ (\text{test} \$ nx)), gny, \text{Agent } B, \text{Agent } A \rangle$   
 $\ (\text{Hash} \ \langle NonceF \ (\text{test} \$ ni), Nr \rangle))$   
 $\in \text{chan } s$

*can-signal s A B*  
**with** *Htest Hi obtain Rb where HRb:*  
*guessed-runs Rb = (|role=Resp, owner=B, partner=A|)*  
*in-progressS (progress s Rb) {xny, xni, xnr, xgnx, xgny, xsk}*  
*gny = Exp Gen (NonceF (Rb\$ny))*  
*Nr = NonceF (Rb\$nr)*  
*guessed-frame Rb xni = Some (NonceF (test\$ni))*  
*guessed-frame Rb xgnx = Some (Exp Gen (NonceF (test\$nx)))*  
**by** (*auto dest!: auth-guard-step3*)  
**with** *Hi*  
**have** *guessed-frame Rb xsk = Some (Exp (Exp Gen (NonceF (Rb\$ny))) (NonceF (test\$nx)))*  
**by** (*auto dest: l2-inv6'D*)  
**with** *HRb Htest have Rb ∈ partners*  
**by** (*auto simp add: partners-def partner-runs-def, simp add: matching-def*)  
**with** *HRb have Exp gny (NonceF (test \$ nx)) =*  
*Exp (Exp Gen (NonceF (Rb \$ ny))) (NonceF (test \$ nx)) ∧ Rb ∈ partners*  
**by** *auto*  
**then show**  $\exists N N' R'$   
*Exp gny (NonceF (test \$ nx)) = Exp (Exp Gen (NonceF (R' \$ N'))) (NonceF (test \$ N)) ∧*  
*R' ∈ partners ∧ (N = nx ∨ N = ny) ∧ (N' = nx ∨ N' = ny)*  
**by** *blast*  
**qed** (*auto simp add: can-signal-def*)  
**qed**

**lemma** *l2-inv7-step4:*

$\{l2\text{-inv}7 \cap l2\text{-inv}1 \cap l2\text{-inv}5 \cap l2\text{-inv}6 \cap l2\text{-inv}6'\} l2\text{-step}4 Rb A B Ni gnx \{> l2\text{-inv}7\}$

**proof** (*auto simp add: PO-hoare-defs intro!: l2-inv7I*)

**fix**  $s s' :: l2\text{-state}$  **fix**  $x$

**assume**  $Hi:s \in l2\text{-inv}1 \ s \in l2\text{-inv}7 \ s \in l2\text{-inv}5 \ s \in l2\text{-inv}6 \ s \in l2\text{-inv}6'$

**assume**  $Ht:(s, s') \in l2\text{-step}4 Rb A B Ni gnx$

**assume**  $Hs:x \in \text{secret } s'$

**from**  $Hs Ht$  **have**  $x \in \text{secret } s \vee (Rb = \text{test} \wedge x = \text{Exp } gnx \text{ (NonceF (Rb\$ny))})$

**by** (*auto simp add: l2-defs*)

**with**  $Hi Ht$

**show**  $\exists N N' R'. x = \text{Exp (Exp Gen (NonceF (R' \$ N'))) (NonceF (test \$ N))} \wedge R' \in \text{partners}$   
 $\wedge (N = nx \vee N = ny) \wedge (N' = nx \vee N' = ny)$

**proof** (*auto dest: l2-inv7D simp add: l2-defs*)

**assume**  $Htest: \text{guessed-runs test} = (|role = \text{Resp}, owner = B, partner = A|)$

*guessed-frame test xgnx = Some gnx*

*guessed-frame test xni = Some Ni*

**assume**  $\text{progress } s \text{ test} = \text{Some } \{xny, xni, xnr, xgnx, xgny, xsk\}$

**with**  $Htest Hi$  **have**  $Htest': \text{guessed-frame test } xsk = \text{Some (Exp } gnx \text{ (NonceF (test \$ ny)))}$

**by** (*auto dest: l2-inv6'D*)

**assume**

$\text{Insec } A B \text{ (hmac (Suc 0), Exp Gen (NonceF (test\$ny)), gnx, Agent A, Agent B)}$   
 $\text{(Hash (Ni, NonceF (test \$ nr)))}$

$\in \text{chan } s$

*can-signal s A B*

**with**  $Hi Htest$  **obtain**  $Ra$  **where**  $HRA:$

*guessed-runs Ra = (|role=Init, owner=A, partner=B|)*

*in-progressS (progress s Ra) {xnx, xni, xnr, xgnx, xgny, xsk, xEnd}*

*gnx = Exp Gen (NonceF (Ra\$nx))*

*Ni = NonceF (Ra\$ni)*

```

    guessed-frame Ra xgny = Some (Exp Gen (NonceF (test$ny)))
    guessed-frame Ra xnr = Some (NonceF (test$nr))
    by (auto dest!: auth-guard-step4)
  with Hi
  have guessed-frame Ra xsk = Some (Exp (Exp Gen (NonceF (Ra$nx))) (NonceF (test$ny)))
    by (auto dest: l2-inv6D)
  with HRa Htest Htest' have Ra ∈ partners
    by (auto simp add: partners-def partner-runs-def, simp add: matching-def)
  with HRa have Exp gnz (NonceF (test $ ny)) =
      Exp (Exp Gen (NonceF (Ra $ nx))) (NonceF (test $ ny)) ∧ Ra ∈ partners
    by auto
  then show ∃ N N' R'.
    Exp gnz (NonceF (test $ ny))
    = Exp (Exp Gen (NonceF (R' $ N'))) (NonceF (test $ N)) ∧
      R' ∈ partners ∧ (N = nx ∨ N = ny) ∧ (N' = nx ∨ N' = ny)
    by auto
  qed (auto simp add: can-signal-def)
qed

```

**lemma** *l2-inv7-trans* [iff]:  
 $\{l2\text{-inv}7 \cap l2\text{-inv}1 \cap l2\text{-inv}4 \cap l2\text{-inv}5 \cap l2\text{-inv}6 \cap l2\text{-inv}6'\} \text{ trans } l2 \{> l2\text{-inv}7\}$   
**apply** (auto simp add: l2-nostep-defs intro!: l2-inv7-step3 l2-inv7-step4)  
**apply** (auto simp add: PO-hoare-defs intro!: l2-inv7I)  
**apply** (auto simp add: l2-defs dy-fake-chan-def dest: l2-inv7D)  
**done**

**lemma** *PO-l2-inv7* [iff]: *reach*  $l2 \subseteq l2\text{-inv}7$   
**by** (rule-tac  $J=l2\text{-inv}1 \cap l2\text{-inv}4 \cap l2\text{-inv}5 \cap l2\text{-inv}6 \cap l2\text{-inv}6'$  **in** *inv-rule-incr*) (auto)

auxiliary dest rule for inv7

**lemma** *Exp-Exp-Gen-synth*:  
 $\text{Exp (Exp Gen X) } Y \in \text{synth } H \implies \text{Exp (Exp Gen X) } Y \in H \vee X \in \text{synth } H \vee Y \in \text{synth } H$   
**by** (erule *synth.cases*, auto dest: *Exp-Exp-Gen-inj2*)

**lemma** *l2-inv7-aux*:  
 $s \in l2\text{-inv}7 \implies$   
 $x \in \text{secret } s \implies$   
 $x \notin \text{synth (analz (generators } s))$   
**apply** (auto simp add: analz-generators dest!: l2-inv7D [THEN [2] rev-subsetD])  
**apply** (auto dest!: *Exp-Exp-Gen-synth Exp-Exp-Gen-inj2*)  
**done**

## 27.3 Refinement

Mediator function.

**definition**  
 $\text{med}12s :: l2\text{-obs} \Rightarrow \text{skl1-obs}$   
**where**  
 $\text{med}12s \ t \equiv ()$   
 $\text{ik} = \text{ik } t,$   
 $\text{secret} = \text{secret } t,$

```

progress = progress t,
signalsInit = signalsInit t,
signalsResp = signalsResp t,
signalsInit2 = signalsInit2 t,
signalsResp2 = signalsResp2 t
  )

```

Relation between states.

**definition**

$R12s :: (skl1\text{-state} * l2\text{-state}) \text{ set}$

**where**

```

R12s ≡ {(s,s').
  s = med12s s'
}

```

**lemmas**  $R12s\text{-defs} = R12s\text{-def med12s\text{-def}}$

**lemma** *can-signal-R12* [*simp*]:

```

(s1, s2) ∈ R12s ⇒
  can-signal s1 A B ⇔ can-signal s2 A B

```

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

Protocol events.

**lemma** *l2-step1-refines-step1*:

$\{R12s\} skl1\text{-step1 Ra A B, l2-step1 Ra A B \{>R12s\}$

**by** (*auto simp add: PO-rhoare-defs R12s-defs skl1-step1-def l2-step1-def*)

**lemma** *l2-step2-refines-step2*:

$\{R12s\} skl1\text{-step2 Rb A B Ni gnx, l2-step2 Rb A B Ni gnx \{>R12s\}$

**by** (*auto simp add: PO-rhoare-defs R12s-defs skl1-step2-def, simp-all add: l2-step2-def*)

for step3 and 4, we prove the level 1 guard, i.e., "the future session key is not in *synth* (*analz* (*ik s*))", using the fact that *inv8* also holds for the future state in which the session key is already in *secret s*

**lemma** *l2-step3-refines-step3*:

```

{R12s ∩ UNIV × (l2-inv1 ∩ l2-inv3 ∩ l2-inv4 ∩ l2-inv6' ∩ l2-inv7)}
  skl1-step3 Ra A B Nr gny, l2-step3 Ra A B Nr gny
  {>R12s}

```

**proof** (*auto simp add: PO-rhoare-defs R12s-defs*)

**fix**  $s s'$

**assume**  $Hi:s ∈ l2\text{-inv1 } s ∈ l2\text{-inv4 } s ∈ l2\text{-inv6}'$

**assume**  $Ht: (s, s') ∈ l2\text{-step3 Ra A B Nr gny}$

**assume**  $s ∈ l2\text{-inv7 } s ∈ l2\text{-inv3}$

**with**  $Hi Ht l2\text{-inv7-step3 } l2\text{-inv3-step3}$  **have**  $Hi':s' ∈ l2\text{-inv7 } s' ∈ l2\text{-inv3}$

**by** (*auto simp add: PO-rhoare-defs, blast, blast*)

**from**  $Ht$  **have**  $Ra = test \longrightarrow Exp gny (NonceF (Ra\$nx)) ∈ secret s'$

**by** (*auto simp add: l2-defs*)

**with**  $Hi'$  **have**  $Ra = test \longrightarrow Exp gny (NonceF (Ra\$nx)) \notin synth (analz (generators s'))$

**by** (*auto dest: l2-inv7-aux*)

**with**  $Hi'$  **have**  $G2:Ra = test \longrightarrow Exp gny (NonceF (Ra\$nx)) \notin synth (analz (ik s'))$

**by** (*auto dest!: l2-inv3D-aux*)

**from** *Ht Hi* **have** *G1*:

*can-signal* *s A B*  $\longrightarrow$   $(\exists Rb. \text{guessed-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \wedge$   
*in-progressS* (*progress* *s Rb*) {*xny, xni, xnr, xgnx, xgny, xsk*}  $\wedge$   
*gny* = *Exp Gen* (*NonceF* (*Rb\$ny*))  $\wedge$   
*Nr* = *NonceF* (*Rb\$nr*)  $\wedge$   
*guessed-frame* *Rb xgnx* = *Some* (*Exp Gen* (*NonceF* (*Ra\$nx*)))  $\wedge$   
*guessed-frame* *Rb xni* = *Some* (*NonceF* (*Ra\$ni*)))

**by** (*auto dest!*: *auth-guard-step3 simp add: l2-defs*)

**with** *Ht G1 G2* **show**

$(\text{!}ik = ik\ s, \text{secret} = \text{secret}\ s, \text{progress} = \text{progress}\ s,$   
*signalsInit* = *signalsInit* *s*, *signalsResp* = *signalsResp* *s*,  
*signalsInit2* = *signalsInit2* *s*, *signalsResp2* = *signalsResp2* *s*),  
 $(\text{!}ik = ik\ s', \text{secret} = \text{secret}\ s', \text{progress} = \text{progress}\ s',$   
*signalsInit* = *signalsInit* *s'*, *signalsResp* = *signalsResp* *s'*,  
*signalsInit2* = *signalsInit2* *s'*, *signalsResp2* = *signalsResp2* *s'*)  
 $\in \text{skl1-step3 } Ra\ A\ B\ Nr\ gny$

**apply** (*auto simp add: l2-step3-def, auto simp add: skl1-step3-def*)

**apply** (*auto simp add: can-signal-def*)

**done**

**qed**

**lemma** *l2-step4-refines-step4*:

$\{R12s \cap UNIV \times (l2-inv1 \cap l2-inv3 \cap l2-inv5 \cap l2-inv6 \cap l2-inv6' \cap l2-inv7)\}$   
*skl1-step4* *Rb A B Ni gnx, l2-step4* *Rb A B Ni gnx*  
 $\{>R12s\}$

**proof** (*auto simp add: PO-rhoare-defs R12s-defs*)

**fix** *s s'*

**assume** *Hi:s*  $\in$  *l2-inv1* *s*  $\in$  *l2-inv5* *s*  $\in$  *l2-inv6* *s*  $\in$  *l2-inv6'*

**assume** *Ht:* (*s, s'*)  $\in$  *l2-step4* *Rb A B Ni gnx*

**assume** *s*  $\in$  *l2-inv7* *s*  $\in$  *l2-inv3*

**with** *Hi Ht l2-inv7-step4 l2-inv3-step4* **have** *Hi':s'*  $\in$  *l2-inv7* *s'*  $\in$  *l2-inv3*

**by** (*auto simp add: PO-hoare-defs, blast, blast*)

**from** *Ht* **have** *Rb = test*  $\longrightarrow$  *Exp gnx* (*NonceF* (*Rb\$ny*))  $\in$  *secret* *s'*

**by** (*auto simp add: l2-defs*)

**with** *Hi'* **have** *Rb = test*  $\longrightarrow$  *Exp gnx* (*NonceF* (*Rb\$ny*))  $\notin$  *synth* (*analz* (*generators* *s'*))

**by** (*auto dest: l2-inv7-aux*)

**with** *Hi'* **have** *G2:Rb = test*  $\longrightarrow$  *Exp gnx* (*NonceF* (*Rb\$ny*))  $\notin$  *synth* (*analz* (*ik* *s'*))

**by** (*auto dest!: l2-inv3D-aux*)

**from** *Ht Hi* **have** *G1*:

*can-signal* *s A B*  $\longrightarrow$   $(\exists Ra. \text{guessed-runs } Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \wedge$   
*in-progressS* (*progress* *s Ra*) {*xnx, xni, xnr, xgnx, xgny, xsk, xEnd*}  $\wedge$   
*guessed-frame* *Ra xgnx* = *Some* *gnx*  $\wedge$   
*guessed-frame* *Ra xni* = *Some* *Ni*  $\wedge$   
*guessed-frame* *Ra xgny* = *Some* (*Exp Gen* (*NonceF* (*Rb\$ny*)))  $\wedge$   
*guessed-frame* *Ra xnr* = *Some* (*NonceF* (*Rb\$nr*)))

**by** (*auto dest!*: *auth-guard-step4 simp add: l2-defs*)

**with** *Ht G1 G2* **show**

$(\text{!}ik = ik\ s, \text{secret} = \text{secret}\ s, \text{progress} = \text{progress}\ s,$   
*signalsInit* = *signalsInit* *s*, *signalsResp* = *signalsResp* *s*,  
*signalsInit2* = *signalsInit2* *s*, *signalsResp2* = *signalsResp2* *s*),  
 $(\text{!}ik = ik\ s', \text{secret} = \text{secret}\ s', \text{progress} = \text{progress}\ s',$   
*signalsInit* = *signalsInit* *s'*, *signalsResp* = *signalsResp* *s'*,  
*signalsInit2* = *signalsInit2* *s'*, *signalsResp2* = *signalsResp2* *s'*)

$\in \text{skl1-step4 } Rb \ A \ B \ Ni \ gnx$   
**apply** (*auto simp add: l2-step4-def, auto simp add: skl1-step4-def*)  
**apply** (*auto simp add: can-signal-def*)  
**done**  
**qed**

attacker events

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

**lemma** *l2-dy-fake-msg-refines-learn*:  
 $\{R12s \cap UNIV \times (l2-inv3 \cap l2-inv7)\} \text{ l1-learn } m, \text{ l2-dy-fake-msg } m \ \{>R12s\}$   
**apply** (*auto simp add: PO-rhoare-defs R12s-defs l2-loc-defs l1-defs*)  
**apply** (*drule Fake-insert-dy-fake-msg, erule l2-inv3D [THEN conjunct1]*)  
**apply** (*auto dest!: l2-inv7-aux*)  
**done**

compromising events

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

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

**lemma** *l2-skr-refines-learn*:  
 $\{R12s \cap UNIV \times (l2-inv2 \cap l2-inv3 \cap l2-inv7)\} \text{ l1-learn } K, \text{ l2-skr } R \ K \ \{>R12s\}$   
**proof** (*auto simp add: PO-rhoare-defs R12s-defs l2-loc-defs l1-defs*)  
**fix**  $s :: \text{l2-state}$  **fix**  $x$   
**assume**  $H$ :  
 $s \in l2-inv2 \ s \in l2-inv3$   
 $R \notin \text{partners } R \neq \text{test in-progress } (\text{progress } s \ R) \ xsk \ \text{guessed-frame } R \ xsk = \text{Some } K$   
**assume**  $Hx: x \in \text{synth } (\text{analz } (\text{insert } K \ (ik \ s)))$   
**assume**  $x \in \text{secret } s \ s \in l2-inv7$   
**then obtain**  $R \ R' \ N \ N'$  **where**  $Hx': x = \text{Exp } (\text{Exp } \text{Gen } (\text{NonceF } (R\$N))) \ (\text{NonceF } (R'\$N'))$   
 $R = \text{test} \wedge R' \in \text{partners} \wedge (N=nx \vee N=ny) \wedge (N'=nx \vee N'=ny)$   
**by** (*auto dest!: l2-inv7D subsetD*)  
**from**  $H$  **have**  $s \ (ik := \text{insert } K \ (ik \ s)) \in l2-inv3$   
**by** (*auto intro: hoare-apply [OF l2-inv3-skr] simp add: l2-defs*)  
**with**  $Hx$  **have**  $x \in \text{synth } (\text{analz } (\text{generators } (s \ (ik := \text{insert } K \ (ik \ s)))))$   
**by** (*auto dest: l2-inv3D-aux*)  
**with**  $Hx'$  **show**  $\text{False}$   
**by** (*auto dest!: Exp-Exp-Gen-synth dest: Exp-Exp-Gen-inj2 simp add: analz-generators*)  
**qed**

Refinement proof.

**lemmas** *l2-trans-refines-l1-trans =*  
*l2-dy-fake-msg-refines-learn l2-dy-fake-chan-refines-skip*  
*l2-lkr-others-refines-skip l2-lkr-after-refines-skip l2-skr-refines-learn*

*l2-step1-refines-step1 l2-step2-refines-step2 l2-step3-refines-step3 l2-step4-refines-step4*

**lemma** *l2-refines-init-l1* [iff]:  
*init l2*  $\subseteq$  *R12s* “ (*init skl1*)  
**by** (*auto simp add: R12s-defs skl1-defs l2-loc-defs*)

**lemma** *l2-refines-trans-l1* [iff]:  
 $\{R12s \cap (UNIV \times (l2-inv1 \cap l2-inv2 \cap l2-inv3 \cap l2-inv4 \cap l2-inv5 \cap$   
 $l2-inv6 \cap l2-inv6' \cap l2-inv7))\}$   
*trans skl1, trans l2*  
 $\{> R12s\}$   
**by** (*auto 0 3 simp add: skl1-def l2-def skl1-trans-def l2-trans-def*  
*intro!: l2-trans-refines-l1-trans*)

**lemma** *PO-obs-consistent-R12s* [iff]:  
*obs-consistent R12s med12s skl1 l2*  
**by** (*auto simp add: obs-consistent-def R12s-def med12s-def l2-defs*)

**lemma** *l2-refines-l1* [iff]:  
*refines*  
 $(R12s \cap$   
 $(reach\ skl1 \times (l2-inv1 \cap l2-inv2 \cap l2-inv3 \cap l2-inv4 \cap l2-inv5 \cap$   
 $l2-inv6 \cap l2-inv6' \cap l2-inv7)))$   
*med12s skl1 l2*  
**by** (*rule Refinement-using-invariants, auto*)

**lemma** *l2-implements-l1* [iff]:  
*implements med12s skl1 l2*  
**by** (*rule refinement-soundness*) (*auto*)

## 27.4 Derived invariants

We want to prove *l2-secrecy*: *dy-fake-msg (bad s) (ik s) (chan s)  $\cap$  secret s = {}* but by refinement we only get *l2-partial-secrecy*: *synth (analz (ik s))  $\cap$  secret s = {}* This is fine, since a message in *dy-fake-msg (bad s) (ik s) (chan s)* could be added to *ik s*, and *l2-partial-secrecy* would still hold for this new state.

### definition

*l2-partial-secrecy* :: (*'a l2-state-scheme*) *set*

### where

*l2-partial-secrecy*  $\equiv \{s. \text{synth } (analz \ (ik \ s)) \cap \text{secret } s = \{\}\}$

**lemma** *l2-obs-partial-secrecy* [iff]: *oreach l2*  $\subseteq$  *l2-partial-secrecy*

**apply** (*rule external-invariant-translation*

[*OF skl1-obs-secrecy - l2-implements-l1*])

**apply** (*auto simp add: med12s-def s0-secrecy-def l2-partial-secrecy-def*)

**done**

**lemma** *l2-oreach-dy-fake-msg*:

$\llbracket s \in \text{oreach } l2; x \in \text{dy-fake-msg } (bad \ s) \ (ik \ s) \ (chan \ s) \rrbracket$

$\implies s \ (ik := \text{insert } x \ (ik \ s)) \in \text{oreach } l2$

**apply** (*auto simp add: oreach-def, rule, simp-all,*



*simp add: l2-def l2-trans-def l2-dy-fake-msg-def*

**apply** *blast*  
**done**

**definition**

*l2-secrecy* :: ('a *l2-state-scheme*) *set*

**where**

*l2-secrecy*  $\equiv \{s. \text{dy-fake-msg } (\text{bad } s) \ (\text{ik } s) \ (\text{chan } s) \cap \text{secret } s = \{\}\}$

**lemma** *l2-obs-secrecy* [*iff*]: *oreach*  $l2 \subseteq l2\text{-secrecy}$

**apply** (*auto simp add:l2-secrecy-def*)

**apply** (*drule l2-oreach-dy-fake-msg, simp-all*)

**apply** (*drule l2-obs-partial-secrecy [THEN [2] rev-subsetD], simp add: l2-partial-secrecy-def*)

**apply** *blast*

**done**

**lemma** *l2-secrecy* [*iff*]: *reach*  $l2 \subseteq l2\text{-secrecy}$

**by** (*rule external-to-internal-invariant [OF l2-obs-secrecy], auto*)

**abbreviation** *l2-iagreement-Init*  $\equiv l1\text{-iagreement-Init}$

**lemma** *l2-obs-iagreement-Init* [*iff*]: *oreach*  $l2 \subseteq l2\text{-iagreement-Init}$

**apply** (*rule external-invariant-translation*

*[OF skl1-obs-iagreement-Init - l2-implements-l1]*)

**apply** (*auto simp add: med12s-def l1-iagreement-Init-def*)

**done**

**lemma** *l2-iagreement-Init* [*iff*]: *reach*  $l2 \subseteq l2\text{-iagreement-Init}$

**by** (*rule external-to-internal-invariant [OF l2-obs-iagreement-Init], auto*)

**abbreviation** *l2-iagreement-Resp*  $\equiv l1\text{-iagreement-Resp}$

**lemma** *l2-obs-iagreement-Resp* [*iff*]: *oreach*  $l2 \subseteq l2\text{-iagreement-Resp}$

**apply** (*rule external-invariant-translation*

*[OF skl1-obs-iagreement-Resp - l2-implements-l1]*)

**apply** (*auto simp add: med12s-def l1-iagreement-Resp-def*)

**done**

**lemma** *l2-iagreement-Resp* [*iff*]: *reach*  $l2 \subseteq l2\text{-iagreement-Resp}$

**by** (*rule external-to-internal-invariant [OF l2-obs-iagreement-Resp], auto*)

**end**

## 28 SKEME Protocol (L3 locale)

```
theory sklv3  
imports sklv2 Implem-lemmas  
begin
```

### 28.1 State and Events

Level 3 state.

(The types have to be defined outside the locale.)

```
record l3-state = skl1-state +  
  bad :: agent set
```

```
type-synonym l3-obs = l3-state
```

```
type-synonym  
  l3-pred = l3-state set
```

```
type-synonym  
  l3-trans = (l3-state  $\times$  l3-state) set
```

attacker event

```
definition  
  l3-dy :: msg  $\Rightarrow$  l3-trans  
where  
  l3-dy  $\equiv$  ik-dy
```

Compromise events.

```
definition  
  l3-lkr-others :: agent  $\Rightarrow$  l3-trans  
where  
  l3-lkr-others A  $\equiv$  {(s,s').  
    — guards  
    A  $\neq$  test-owner  $\wedge$   
    A  $\neq$  test-partner  $\wedge$   
    — actions  
    s' = s(bad := {A}  $\cup$  bad s,  
           ik := keys-of A  $\cup$  ik s)  
  }
```

```
definition  
  l3-lkr-actor :: agent  $\Rightarrow$  l3-trans  
where  
  l3-lkr-actor A  $\equiv$  {(s,s').  
    — guards  
    A = test-owner  $\wedge$   
    A  $\neq$  test-partner  $\wedge$   
    — actions  
    s' = s(bad := {A}  $\cup$  bad s,  
           ik := keys-of A  $\cup$  ik s)  
  }
```

**definition**

$$l3-lkr\text{-after} :: \text{agent} \Rightarrow l3\text{-trans}$$
**where**

$$l3-lkr\text{-after } A \equiv \{(s, s') .$$

— guards  
 $test\text{-ended } s \wedge$   
 — actions  
 $s' = s(\text{bad} := \{A\} \cup \text{bad } s,$   
 $ik := \text{keys-of } A \cup ik \ s)$   
 $\}$

**definition**

$$l3-skr :: \text{rid-t} \Rightarrow \text{msg} \Rightarrow l3\text{-trans}$$
**where**

$$l3-skr \ R \ K \equiv \{(s, s') .$$

— guards  
 $R \neq test \wedge R \notin \text{partners} \wedge$   
 $in\text{-progress } (\text{progress } s \ R) \ xsk \wedge$   
 $guessed\text{-frame } R \ xsk = \text{Some } K \wedge$   
 — actions  
 $s' = s(ik := \{K\} \cup ik \ s)$   
 $\}$

New locale for the level 3 protocol. This locale does not add new assumptions, it is only used to separate the level 3 protocol from the implementation locale.

**locale**  $skl3 = \text{valid-implem}$

**begin**

Protocol events (with  $K = H(ni, nr)$ ):

- step 1: create  $Ra$ ,  $A$  generates  $nx$  and  $ni$ , confidentially sends  $ni$ , computes and insecurely sends  $g^{nx}$
- step 2: create  $Rb$ ,  $B$  receives  $ni$  (confidentially) and  $g^{nx}$  (insecurely), generates  $ny$  and  $nr$ , confidentially sends  $nr$ , insecurely sends  $g^{ny}$  and  $MAC_K(g^{nx}, g^{ny}, B, A)$  computes  $g^{nx * ny}$ , emits a running signal for  $Init, ni, nr, g^{nx * ny}$
- step 3:  $A$  receives  $nr$  confidentially, and  $g^{ny}$  and the MAC insecurely, sends  $MAC_K(g^{ny}, g^{nx}, A, B)$  insecurely, computes  $g^{ny * nx}$ , emits a commit signal for  $Init, ni, nr, g^{ny * nx}$ , a running signal for  $Resp, ni, nr, g^{ny * nx}$ , declares the secret  $g^{ny * nx}$
- step 4:  $B$  receives the MAC insecurely, emits a commit signal for  $Resp, ni, nr, g^{nx * ny}$ , declares the secret  $g^{nx * ny}$

**definition**

$$l3\text{-step1} :: \text{rid-t} \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow l3\text{-trans}$$
**where**

$$l3\text{-step1 } Ra \ A \ B \equiv \{(s, s') .$$

— guards:  
 $Ra \notin \text{dom } (\text{progress } s) \wedge$   
 $guessed\text{-runs } Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \wedge$

— actions:  
 $s' = s(\langle$   
 $\text{progress} := (\text{progress } s)(Ra \mapsto \{xnx, xni, xgnx\}),$   
 $ik := \{\text{implConfid } A \ B \ (\text{NonceF } (Ra\$ni))\} \cup$   
 $\{\text{implInsec } A \ B \ (\text{Exp Gen } (\text{NonceF } (Ra\$nx)))\} \cup$   
 $(ik \ s)$   
 $\rangle$   
 $\}$

**definition**

$l3\text{-step2} :: \text{rid-}t \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow \text{msg} \Rightarrow \text{msg} \Rightarrow l3\text{-trans}$

**where**

$l3\text{-step2 } Rb \ A \ B \ Ni \ gnx \equiv \{(s, s')\}.$

— guards:

$\text{guessed-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \wedge$   
 $Rb \notin \text{dom } (\text{progress } s) \wedge$   
 $\text{guessed-frame } Rb \ xgnx = \text{Some } gnx \wedge$   
 $\text{guessed-frame } Rb \ xni = \text{Some } Ni \wedge$   
 $\text{guessed-frame } Rb \ xsk = \text{Some } (\text{Exp } gnx \ (\text{NonceF } (Rb\$ny))) \wedge$   
 $\text{implConfid } A \ B \ Ni \in ik \ s \wedge$   
 $\text{implInsec } A \ B \ gnx \in ik \ s \wedge$

— actions:

$s' = s(\langle$   
 $\text{progress} := (\text{progress } s)(Rb \mapsto \{xny, xni, xnr, xgny, xgnx, xsk\}),$   
 $ik := \{\text{implConfid } B \ A \ (\text{NonceF } (Rb\$nr))\} \cup$   
 $\{\text{implInsec } B \ A \ (\text{Exp Gen } (\text{NonceF } (Rb\$ny))),$   
 $\text{hmac } \langle \text{Number } 0, gnx, \text{Exp Gen } (\text{NonceF } (Rb\$ny)), \text{Agent } B, \text{Agent } A \rangle$   
 $(\text{Hash } \langle Ni, \text{NonceF } (Rb\$nr) \rangle))\} \cup$   
 $(ik \ s),$   
 $\text{signalsInit} :=$   
 $\text{if can-signal } s \ A \ B \ \text{then}$   
 $\text{addSignal } (\text{signalsInit } s)$   
 $(\text{Running } A \ B \ \langle Ni, \text{NonceF } (Rb\$nr), \text{Exp } gnx \ (\text{NonceF } (Rb\$ny)) \rangle)$   
 $\text{else}$   
 $\text{signalsInit } s,$   
 $\text{signalsInit2} :=$   
 $\text{if can-signal } s \ A \ B \ \text{then}$   
 $\text{addSignal } (\text{signalsInit2 } s) (\text{Running } A \ B \ (\text{Exp } gnx \ (\text{NonceF } (Rb\$ny))))$   
 $\text{else}$   
 $\text{signalsInit2 } s$   
 $\rangle$   
 $\}$

**definition**

$l3\text{-step3} :: \text{rid-}t \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow \text{msg} \Rightarrow \text{msg} \Rightarrow l3\text{-trans}$

**where**

$l3\text{-step3 } Ra \ A \ B \ Nr \ gny \equiv \{(s, s')\}.$

— guards:

$\text{guessed-runs } Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \wedge$   
 $\text{progress } s \ Ra = \text{Some } \{xnx, xni, xgnx\} \wedge$   
 $\text{guessed-frame } Ra \ xgny = \text{Some } gny \wedge$   
 $\text{guessed-frame } Ra \ xnr = \text{Some } Nr \wedge$

*guessed-frame*  $Ra\ xsk = \text{Some} (\text{Exp } gny (\text{NonceF } (Ra\$nx))) \wedge$   
*implConfd*  $B\ A\ Nr \in ik\ s \wedge$   
*implInsec*  $B\ A\ \langle gny, hmac\ \langle \text{Number } 0, \text{Exp Gen } (\text{NonceF } (Ra\$nx)), gny, \text{Agent } B, \text{Agent } A \rangle$   
 $\quad (\text{Hash } \langle \text{NonceF } (Ra\$ni), Nr \rangle) \rangle \in ik\ s \wedge$   
— actions:  
 $s' = s(\mid \text{progress} := (\text{progress } s)(Ra \mapsto \{xnx, xni, xnr, xgnx, xgny, xsk, xEnd\}),$   
 $\quad ik := \{\text{implInsec } A\ B\ (hmac\ \langle \text{Number } 1, gny, \text{Exp Gen } (\text{NonceF } (Ra\$nx)), \text{Agent } A, \text{Agent } B)$   
 $\quad (\text{Hash } \langle \text{NonceF } (Ra\$ni), Nr \rangle)\} \cup ik\ s,$   
 $\text{secret} := \{x. x = \text{Exp } gny (\text{NonceF } (Ra\$nx)) \wedge Ra = \text{test}\} \cup \text{secret } s,$   
*signalsInit* :=  
 $\quad \text{if can-signal } s\ A\ B\ \text{then}$   
 $\quad \quad \text{addSignal } (\text{signalsInit } s)$   
 $\quad \quad (\text{Commit } A\ B\ \langle \text{NonceF } (Ra\$ni), Nr, \text{Exp } gny (\text{NonceF } (Ra\$nx)) \rangle)$   
 $\quad \text{else}$   
 $\quad \quad \text{signalsInit } s,$   
*signalsInit2* :=  
 $\quad \text{if can-signal } s\ A\ B\ \text{then}$   
 $\quad \quad \text{addSignal } (\text{signalsInit2 } s) (\text{Commit } A\ B\ (\text{Exp } gny (\text{NonceF } (Ra\$nx))))$   
 $\quad \text{else}$   
 $\quad \quad \text{signalsInit2 } s,$   
*signalsResp* :=  
 $\quad \text{if can-signal } s\ A\ B\ \text{then}$   
 $\quad \quad \text{addSignal } (\text{signalsResp } s)$   
 $\quad \quad (\text{Running } A\ B\ \langle \text{NonceF } (Ra\$ni), Nr, \text{Exp } gny (\text{NonceF } (Ra\$nx)) \rangle)$   
 $\quad \text{else}$   
 $\quad \quad \text{signalsResp } s,$   
*signalsResp2* :=  
 $\quad \text{if can-signal } s\ A\ B\ \text{then}$   
 $\quad \quad \text{addSignal } (\text{signalsResp2 } s) (\text{Running } A\ B\ (\text{Exp } gny (\text{NonceF } (Ra\$nx))))$   
 $\quad \text{else}$   
 $\quad \quad \text{signalsResp2 } s$   
 $\mid$   
 $\}$

**definition**

$l3\text{-step4} :: \text{rid-t} \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow \text{msg} \Rightarrow \text{msg} \Rightarrow l3\text{-trans}$

**where**

$l3\text{-step4 } Rb\ A\ B\ Ni\ gnx \equiv \{(s, s')\}.$

— guards:

*guessed-runs*  $Rb = (\mid \text{role} = \text{Resp}, \text{owner} = B, \text{partner} = A) \wedge$

*progress*  $s\ Rb = \text{Some } \{xny, xni, xnr, xgnx, xgny, xsk\} \wedge$

*guessed-frame*  $Rb\ xgnx = \text{Some } gnx \wedge$

*guessed-frame*  $Rb\ xni = \text{Some } Ni \wedge$

*implInsec*  $A\ B\ (hmac\ \langle \text{Number } 1, \text{Exp Gen } (\text{NonceF } (Rb\$ny)), gnx, \text{Agent } A, \text{Agent } B)$

$\quad (\text{Hash } \langle Ni, \text{NonceF } (Rb\$nr) \rangle) \in ik\ s \wedge$

— actions:

$s' = s(\mid \text{progress} := (\text{progress } s)(Rb \mapsto \{xny, xni, xnr, xgnx, xgny, xsk, xEnd\}),$

$\quad \text{secret} := \{x. x = \text{Exp } gnx (\text{NonceF } (Rb\$ny)) \wedge Rb = \text{test}\} \cup \text{secret } s,$

$\quad \text{signalsResp} :=$

$\quad \quad \text{if can-signal } s\ A\ B\ \text{then}$

```

      addSignal (signalsResp s)
        (Commit A B ⟨Ni, NonceF (Rb$nr), Exp gnx (NonceF (Rb$ny))⟩)
    else
      signalsResp s,
signalsResp2 :=
  if can-signal s A B then
    addSignal (signalsResp2 s) (Commit A B (Exp gnx (NonceF (Rb$ny))))
  else
    signalsResp2 s
  }
}

```

Specification.

Initial compromise.

**definition**

*ik-init* :: msg set

**where**

$ik-init \equiv \{priK\ C \mid C. C \in bad-init\} \cup \{pubK\ A \mid A. True\} \cup$   
 $\{shrK\ A\ B \mid A\ B. A \in bad-init \vee B \in bad-init\} \cup Tags$

lemmas about *ik-init*

**lemma** *parts-ik-init* [simp]: parts *ik-init* = *ik-init*

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

**lemma** *analz-ik-init* [simp]: analz *ik-init* = *ik-init*

**by** (auto dest: analz-into-parts)

**lemma** *abs-ik-init* [iff]: abs *ik-init* = {}

**apply** (auto elim!: absE)

**apply** (auto simp add: ik-init-def)

**done**

**lemma** *payloadSet-ik-init* [iff]: *ik-init*  $\cap$  payload = {}

**by** (auto simp add: ik-init-def)

**lemma** *validSet-ik-init* [iff]: *ik-init*  $\cap$  valid = {}

**by** (auto simp add: ik-init-def)

**definition**

*l3-init* :: l3-state set

**where**

$l3-init \equiv \{ \langle$   
 $ik = ik-init,$   
 $secret = \{\},$   
 $progress = Map.empty,$   
 $signalsInit = \lambda x. 0,$   
 $signalsResp = \lambda x. 0,$   
 $signalsInit2 = \lambda x. 0,$   
 $signalsResp2 = \lambda x. 0,$   
 $bad = bad-init$   
 $\rangle \}$

**lemmas**  $l3\text{-init-defs} = l3\text{-init-def } ik\text{-init-def}$

**definition**

$l3\text{-trans} :: l3\text{-trans}$

**where**

$l3\text{-trans} \equiv (\bigcup M N X Rb Ra A B K.$   
 $l3\text{-step1 } Ra A B \cup$   
 $l3\text{-step2 } Rb A B N X \cup$   
 $l3\text{-step3 } Ra A B N X \cup$   
 $l3\text{-step4 } Rb A B N X \cup$   
 $l3\text{-dy } M \cup$   
 $l3\text{-lkr-others } A \cup$   
 $l3\text{-lkr-after } A \cup$   
 $l3\text{-skr } Ra K \cup$   
 $Id$   
 $)$

**definition**

$l3 :: (l3\text{-state}, l3\text{-obs}) \text{ spec}$  **where**

$l3 \equiv \langle$   
 $init = l3\text{-init},$   
 $trans = l3\text{-trans},$   
 $obs = id$   
 $\rangle$

**lemmas**  $l3\text{-loc-defs} =$

$l3\text{-step1-def } l3\text{-step2-def } l3\text{-step3-def } l3\text{-step4-def}$   
 $l3\text{-def } l3\text{-init-defs } l3\text{-trans-def}$   
 $l3\text{-dy-def}$   
 $l3\text{-lkr-others-def } l3\text{-lkr-after-def } l3\text{-skr-def}$

**lemmas**  $l3\text{-defs} = l3\text{-loc-defs } ik\text{-dy-def}$

**lemmas**  $l3\text{-nostep-defs} = l3\text{-def } l3\text{-init-def } l3\text{-trans-def}$

**lemma**  $l3\text{-obs-id}$  [*simp*]:  $obs \ l3 = id$

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

## 28.2 Invariants

### 28.2.1 inv1: No long-term keys as message parts

**definition**

$l3\text{-inv1} :: l3\text{-state set}$

**where**

$l3\text{-inv1} \equiv \{s.$   
 $parts (ik \ s) \cap range \ LtK \subseteq ik \ s$   
 $\}$

**lemmas**  $l3\text{-inv1I} = l3\text{-inv1-def}$  [*THEN setc-def-to-intro, rule-format*]

**lemmas**  $l3\text{-inv1E}$  [*elim*] =  $l3\text{-inv1-def}$  [*THEN setc-def-to-elim, rule-format*]

**lemmas**  $l3\text{-inv}1D = l3\text{-inv}1\text{-def}$  [*THEN setc-def-to-dest, rule-format*]

**lemma**  $l3\text{-inv}1D'$  [*dest*]:  $\llbracket LtK K \in parts (ik s); s \in l3\text{-inv}1 \rrbracket \implies LtK K \in ik s$   
**by** (*auto simp add: l3-inv1-def*)

**lemma**  $l3\text{-inv}1\text{-init}$  [*iff*]:  
 $init\ l3 \subseteq l3\text{-inv}1$   
**by** (*auto simp add: l3-def l3-init-def intro!:l3-inv1I*)

**lemma**  $l3\text{-inv}1\text{-trans}$  [*iff*]:  
 $\{l3\text{-inv}1\} trans\ l3 \{>\ l3\text{-inv}1\}$   
**apply** (*auto simp add: PO-hoare-defs l3-nostep-defs intro!: l3-inv1I*)  
**apply** (*auto simp add: l3-defs dy-fake-msg-def dy-fake-chan-def*  
 $parts\text{-insert} [where\ H=ik\ -] parts\text{-insert} [where\ H=insert\ -\ (ik\ -)]$   
 $dest!: Fake\text{-parts}\text{-insert}$ )  
**apply** (*auto dest:analz-into-parts*)  
**done**

**lemma**  $PO\text{-}l3\text{-inv}1$  [*iff*]:  
 $reach\ l3 \subseteq l3\text{-inv}1$   
**by** (*rule inv-rule-basic*) (*auto*)

## 28.2.2 inv2: $l3\text{-state}\text{-bad}$ $s$ indeed contains "bad" keys

**definition**

$l3\text{-inv}2 :: l3\text{-state}\ set$

**where**

$l3\text{-inv}2 \equiv \{s.$   
 $Keys\text{-bad}\ (ik\ s)\ (bad\ s)$   
 $\}$

**lemmas**  $l3\text{-inv}2I = l3\text{-inv}2\text{-def}$  [*THEN setc-def-to-intro, rule-format*]  
**lemmas**  $l3\text{-inv}2E$  [*elim*] =  $l3\text{-inv}2\text{-def}$  [*THEN setc-def-to-elim, rule-format*]  
**lemmas**  $l3\text{-inv}2D = l3\text{-inv}2\text{-def}$  [*THEN setc-def-to-dest, rule-format*]

**lemma**  $l3\text{-inv}2\text{-init}$  [*simp,intro!*]:  
 $init\ l3 \subseteq l3\text{-inv}2$   
**by** (*auto simp add: l3-def l3-init-defs intro!:l3-inv2I Keys-badI*)

**lemma**  $l3\text{-inv}2\text{-trans}$  [*simp,intro!*]:  
 $\{l3\text{-inv}2 \cap l3\text{-inv}1\} trans\ l3 \{>\ l3\text{-inv}2\}$   
**apply** (*auto simp add: PO-hoare-defs l3-nostep-defs intro!: l3-inv2I*)  
**apply** (*auto simp add: l3-defs dy-fake-msg-def dy-fake-chan-def*)

4 subgoals:  $dy$ ,  $lkr^*$ ,  $skr$ .

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

**lemma**  $PO\text{-}l3\text{-inv}2$  [*iff*]:  $reach\ l3 \subseteq l3\text{-inv}2$   
**by** (*rule-tac J=l3-inv1 in inv-rule-incr*) (*auto*)



### 28.2.3 inv3

If a message can be analyzed from the intruder knowledge then it can be derived (using *synth/analz*) from the sets of implementation, non-implementation, and long-term key messages and the tags. That is, intermediate messages are not needed.

#### definition

$l3\text{-inv3} :: l3\text{-state set}$

#### where

```

l3-inv3 ≡ {s.
  analz (ik s) ⊆
  synth (analz ((ik s ∩ payload) ∪ ((ik s) ∩ valid) ∪ (ik s ∩ range LtK) ∪ Tags))
}
```

**lemmas**  $l3\text{-inv3}I = l3\text{-inv3}\text{-def}$  [THEN setc-def-to-intro, rule-format]

**lemmas**  $l3\text{-inv3}E = l3\text{-inv3}\text{-def}$  [THEN setc-def-to-elim, rule-format]

**lemmas**  $l3\text{-inv3}D = l3\text{-inv3}\text{-def}$  [THEN setc-def-to-dest, rule-format]

**lemma**  $l3\text{-inv3}\text{-init}$  [iff]:

$init\ l3 \subseteq l3\text{-inv3}$

**apply** (auto simp add: l3-def l3-init-def intro!: l3-inv3I)

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

**done**

**declare**  $domIff$  [iff del]

Most of the cases in this proof are simple and very similar. The proof could probably be shortened.

**lemma**  $l3\text{-inv3}\text{-trans}$  [simp,intro!]:

$\{l3\text{-inv3}\} \text{ trans } l3 \ \{> \ l3\text{-inv3}\}$

**proof** (simp add: l3-nostep-defs, safe)

**fix**  $Ra\ A\ B$

**show**  $\{l3\text{-inv3}\} \text{ l3-step1 } Ra\ A\ B \ \{> \ l3\text{-inv3}\}$

**apply** (auto simp add: PO-hoare-def l3-defs intro!: l3-inv3I dest!: l3-inv3D)

**apply** (auto intro!: validI dest!: analz-insert-partition [THEN [2] rev-subsetD])

**done**

**next**

**fix**  $Rb\ A\ B\ Ni\ gnx$

**show**  $\{l3\text{-inv3}\} \text{ l3-step2 } Rb\ A\ B\ Ni\ gnx \ \{> \ l3\text{-inv3}\}$

**apply** (auto simp add: PO-hoare-def l3-defs intro!: l3-inv3I dest!: l3-inv3D)

**apply** (auto intro!: validI dest!: analz-insert-partition [THEN [2] rev-subsetD])

**done**

**next**

**fix**  $Ra\ A\ B\ Nr\ gny$

**show**  $\{l3\text{-inv3}\} \text{ l3-step3 } Ra\ A\ B\ Nr\ gny \ \{> \ l3\text{-inv3}\}$

**apply** (auto simp add: PO-hoare-def l3-defs intro!: l3-inv3I dest!: l3-inv3D)

**apply** (auto intro!: validI dest!: analz-insert-partition [THEN [2] rev-subsetD])

**done**

**next**

**fix**  $Rb\ A\ B\ Ni\ gnx$

**show**  $\{l3\text{-inv3}\} \text{ l3-step4 } Rb\ A\ B\ Ni\ gnx \ \{> \ l3\text{-inv3}\}$

**apply** (auto simp add: PO-hoare-def l3-defs intro!: l3-inv3I dest!: l3-inv3D)

**done**

```

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

```

**lemma** *PO-l3-inv3 [iff]: reach l3  $\subseteq$  l3-inv3*  
**by** (rule inv-rule-basic) (auto)

#### 28.2.4 inv4: the intruder knows the tags

**definition**

*l3-inv4* :: *l3-state set*

**where**

```

l3-inv4  $\equiv$  {s.
  Tags  $\subseteq$  ik s
}

```

**lemmas** *l3-inv4I = l3-inv4-def [THEN setc-def-to-intro, rule-format]*

**lemmas** *l3-inv4E [elim] = l3-inv4-def [THEN setc-def-to-elim, rule-format]*

**lemmas** *l3-inv4D = l3-inv4-def [THEN setc-def-to-dest, rule-format]*

**lemma** *l3-inv4-init [simp,intro!]:*

*init l3  $\subseteq$  l3-inv4*

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

**lemma** *l3-inv4-trans [simp,intro!]:*

{*l3-inv4*} trans l3 {> *l3-inv4*}

**apply** (auto simp add: PO-hoare-defs l3-nostep-defs intro!: l3-inv4I)

**apply** (auto simp add: l3-defs dy-fake-chan-def dy-fake-msg-def)

**done**

**lemma** *PO-l3-inv4* [*simp,intro!*]:  $reach\ l3 \subseteq l3\text{-inv4}$   
**by** (*rule inv-rule-basic*) (*auto*)

The remaining invariants are derived from the others. They are not protocol dependent provided the previous invariants hold.

### 28.2.5 inv5

The messages that the L3 DY intruder can derive from the intruder knowledge (using *synth/analz*), are either implementations or intermediate messages or can also be derived by the L2 intruder from the set *extr* (*l3-state.bad s*) (*ik s*  $\cap$  *payload*) (*local.abs (ik s)*), that is, given the non-implementation messages and the abstractions of (implementation) messages in the intruder knowledge.

**definition**

*l3-inv5* :: *l3-state set*

**where**

*l3-inv5*  $\equiv$  {  
*synth (analz (ik s))*  $\subseteq$   
*dy-fake-msg (bad s) (ik s*  $\cap$  *payload) (abs (ik s))*  $\cup$  *-payload*  
}

**lemmas** *l3-inv5I* = *l3-inv5-def* [*THEN setc-def-to-intro, rule-format*]

**lemmas** *l3-inv5E* = *l3-inv5-def* [*THEN setc-def-to-elim, rule-format*]

**lemmas** *l3-inv5D* = *l3-inv5-def* [*THEN setc-def-to-dest, rule-format*]

**lemma** *l3-inv5-derived*:  $l3\text{-inv2} \cap l3\text{-inv3} \subseteq l3\text{-inv5}$

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

**lemma** *PO-l3-inv5* [*simp,intro!*]:  $reach\ l3 \subseteq l3\text{-inv5}$

**using** *l3-inv5-derived PO-l3-inv2 PO-l3-inv3*

**by** *blast*

### 28.2.6 inv6

If the level 3 intruder can deduce a message implementing an insecure channel message, then either:

- the message is already in the intruder knowledge, or
- the message is constructed, and the payload can also be deduced by the intruder.

**definition**

*l3-inv6* :: *l3-state set*

**where**

*l3-inv6*  $\equiv$  {*s*.  $\forall$  *A B M*.  
(*implInsec A B M*  $\in$  *synth (analz (ik s))*  $\wedge$  *M*  $\in$  *payload*)  $\longrightarrow$   
(*implInsec A B M*  $\in$  *ik s*  $\vee$  *M*  $\in$  *synth (analz (ik s))*)  
}

**lemmas**  $l3\text{-inv}6I = l3\text{-inv}6\text{-def}$  [*THEN setc-def-to-intro, rule-format*]  
**lemmas**  $l3\text{-inv}6E = l3\text{-inv}6\text{-def}$  [*THEN setc-def-to-elim, rule-format*]  
**lemmas**  $l3\text{-inv}6D = l3\text{-inv}6\text{-def}$  [*THEN setc-def-to-dest, rule-format*]

**lemma**  $l3\text{-inv}6\text{-derived}$  [*simp,intro!*]:

$l3\text{-inv}3 \cap l3\text{-inv}4 \subseteq l3\text{-inv}6$

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

1 subgoal

**apply** (*drule synth-mono, simp, drule subsetD, assumption*)

**apply** (*auto dest!: implInsec-synth-analz* [*rotated 1, where H=- ∪ -*])

**apply** (*auto dest!: synth-analz-monotone* [*of - - ∪ - ik -*])

**done**

**lemma**  $PO\text{-}l3\text{-inv}6$  [*simp,intro!*]: *reach*  $l3 \subseteq l3\text{-inv}6$

**using**  $l3\text{-inv}6\text{-derived}$   $PO\text{-}l3\text{-inv}3$   $PO\text{-}l3\text{-inv}4$

**by** (*blast*)

## 28.2.7 inv7

If the level 3 intruder can deduce a message implementing a confidential channel message, then either:

- the message is already in the intruder knowledge, or
- the message is constructed, and the payload can also be deduced by the intruder.

**definition**

$l3\text{-inv}7 :: l3\text{-state set}$

**where**

$l3\text{-inv}7 \equiv \{s. \forall A B M.$

$(\text{implConfid } A B M \in \text{synth } (\text{analz } (\text{ik } s)) \wedge M \in \text{payload}) \longrightarrow$

$(\text{implConfid } A B M \in \text{ik } s \vee M \in \text{synth } (\text{analz } (\text{ik } s)))$

$\}$

**lemmas**  $l3\text{-inv}7I = l3\text{-inv}7\text{-def}$  [*THEN setc-def-to-intro, rule-format*]

**lemmas**  $l3\text{-inv}7E = l3\text{-inv}7\text{-def}$  [*THEN setc-def-to-elim, rule-format*]

**lemmas**  $l3\text{-inv}7D = l3\text{-inv}7\text{-def}$  [*THEN setc-def-to-dest, rule-format*]

**lemma**  $l3\text{-inv}7\text{-derived}$  [*simp,intro!*]:

$l3\text{-inv}3 \cap l3\text{-inv}4 \subseteq l3\text{-inv}7$

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

1 subgoal

**apply** (*drule synth-mono, simp, drule subsetD, assumption*)

**apply** (*auto dest!: implConfid-synth-analz* [*rotated 1, where H=- ∪ -*])

**apply** (*auto dest!: synth-analz-monotone* [*of - - ∪ - ik -*])

**done**

**lemma**  $PO\text{-}l3\text{-inv}7$  [*simp,intro!*]: *reach*  $l3 \subseteq l3\text{-inv}7$

**using**  $l3\text{-inv}7\text{-derived}$   $PO\text{-}l3\text{-inv}3$   $PO\text{-}l3\text{-inv}4$

**by** (*blast*)

### 28.2.8 inv8

If the level 3 intruder can deduce a message implementing an authentic channel message then either:

- the message is already in the intruder knowledge, or
- the message is constructed, and in this case the payload can also be deduced by the intruder, and one of the agents is bad.

#### definition

$l3\text{-inv8} :: l3\text{-state set}$

#### where

$$l3\text{-inv8} \equiv \{s. \forall A B M. \\ (implAuth A B M \in synth (analz (ik s)) \wedge M \in payload) \longrightarrow \\ (implAuth A B M \in ik s \vee (M \in synth (analz (ik s)) \wedge (A \in bad s \vee B \in bad s))) \\ \}$$

**lemmas**  $l3\text{-inv8I} = l3\text{-inv8-def [THEN setc-def-to-intro, rule-format]$

**lemmas**  $l3\text{-inv8E} = l3\text{-inv8-def [THEN setc-def-to-elim, rule-format]$

**lemmas**  $l3\text{-inv8D} = l3\text{-inv8-def [THEN setc-def-to-dest, rule-format]$

**lemma**  $l3\text{-inv8-derived [iff]:$

$l3\text{-inv2} \cap l3\text{-inv3} \cap l3\text{-inv4} \subseteq l3\text{-inv8}$

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

2 subgoals: M is deducible and the agents are bad

**apply** (*drule synth-mono, simp, drule subsetD, assumption*)

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

**apply** (*drule synth-mono, simp, drule subsetD, assumption*)

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

**done**

**lemma**  $PO\text{-}l3\text{-inv8 [iff]: reach l3} \subseteq l3\text{-inv8}$

**using**  $l3\text{-inv8-derived}$

$PO\text{-}l3\text{-inv3 PO\text{-}l3\text{-inv2 PO\text{-}l3\text{-inv4}$

**by** *blast*

### 28.2.9 inv9

If the level 3 intruder can deduce a message implementing a secure channel message then either:

- the message is already in the intruder knowledge, or
- the message is constructed, and in this case the payload can also be deduced by the intruder, and one of the agents is bad.

#### definition

$l3\text{-inv9} :: l3\text{-state set}$

#### where

```

l3-inv9 ≡ {s. ∀ A B M.
  (implSecure A B M ∈ synth (analz (ik s))) ∧ M ∈ payload) →
  (implSecure A B M ∈ ik s ∨ (M ∈ synth (analz (ik s))) ∧ (A ∈ bad s ∨ B ∈ bad s))
}

```

```

lemmas l3-inv9I = l3-inv9-def [THEN setc-def-to-intro, rule-format]
lemmas l3-inv9E = l3-inv9-def [THEN setc-def-to-elim, rule-format]
lemmas l3-inv9D = l3-inv9-def [THEN setc-def-to-dest, rule-format]

```

```

lemma l3-inv9-derived [iff]:
  l3-inv2 ∩ l3-inv3 ∩ l3-inv4 ⊆ l3-inv9
apply (auto intro!: l3-inv9I dest!:l3-inv3D l3-inv2D)

```

2 subgoals: *M* is deducible and the agents are bad.

```

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

```

```

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

```

```

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

```

## 28.3 Refinement

Mediator function.

```

definition
  med23s :: l3-obs ⇒ l2-obs
where
  med23s t ≡ (
    ik = ik t ∩ payload,
    secret = secret t,
    progress = progress t,
    signalsInit = signalsInit t,
    signalsResp = signalsResp t,
    signalsInit2 = signalsInit2 t,
    signalsResp2 = signalsResp2 t,
    chan = abs (ik t),
    bad = bad t
  )

```

Relation between states.

```

definition
  R23s :: (l2-state * l3-state) set
where
  R23s ≡ {(s, s').
    s = med23s s'
  }

```

**lemmas** *R23s-defs* = *R23s-def med23s-def*

**lemma** *R23sI*:

$\llbracket ik\ s = ik\ t \cap payload; secret\ s = secret\ t; progress\ s = progress\ t;$   
 $signalsInit\ s = signalsInit\ t; signalsResp\ s = signalsResp\ t;$   
 $signalsInit2\ s = signalsInit2\ t; signalsResp2\ s = signalsResp2\ t;$   
 $chan\ s = abs\ (ik\ t); l2-state.bad\ s = bad\ t \rrbracket$

$\implies (s, t) \in R23s$

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

**lemma** *R23sD*:

$(s, t) \in R23s \implies$

$ik\ s = ik\ t \cap payload \wedge secret\ s = secret\ t \wedge progress\ s = progress\ t \wedge$   
 $signalsInit\ s = signalsInit\ t \wedge signalsResp\ s = signalsResp\ t \wedge$   
 $signalsInit2\ s = signalsInit2\ t \wedge signalsResp2\ s = signalsResp2\ t \wedge$   
 $chan\ s = abs\ (ik\ t) \wedge l2-state.bad\ s = bad\ t$

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

**lemma** *R23sE* [*elim*]:

$\llbracket (s, t) \in R23s;$

$\llbracket ik\ s = ik\ t \cap payload; secret\ s = secret\ t; progress\ s = progress\ t;$   
 $signalsInit\ s = signalsInit\ t; signalsResp\ s = signalsResp\ t;$   
 $signalsInit2\ s = signalsInit2\ t; signalsResp2\ s = signalsResp2\ t;$   
 $chan\ s = abs\ (ik\ t); l2-state.bad\ s = bad\ t \rrbracket \implies P \rrbracket$

$\implies P$

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

**lemma** *can-signal-R23* [*simp*]:

$(s2, s3) \in R23s \implies$

$can-signal\ s2\ A\ B \longleftrightarrow can-signal\ s3\ A\ B$

**by** (*auto simp add: can-signal-def*)

### 28.3.1 Protocol events

**lemma** *l3-step1-refines-step1*:

$\{R23s\}\ l2-step1\ Ra\ A\ B, l3-step1\ Ra\ A\ B\ \{>R23s\}$

**apply** (*auto simp add: PO-rhoare-defs R23s-defs*)

**apply** (*auto simp add: l3-defs l2-step1-def*)

**done**

**lemma** *l3-step2-refines-step2*:

$\{R23s\}\ l2-step2\ Rb\ A\ B\ Ni\ gnx, l3-step2\ Rb\ A\ B\ Ni\ gnx\ \{>R23s\}$

**apply** (*auto simp add: PO-rhoare-defs R23s-defs l2-step2-def*)

**apply** (*auto simp add: l3-step2-def*)

**done**

**lemma** *l3-step3-refines-step3*:

$\{R23s\}\ l2-step3\ Ra\ A\ B\ Nr\ gny, l3-step3\ Ra\ A\ B\ Nr\ gny\ \{>R23s\}$

**apply** (*auto simp add: PO-rhoare-defs R23s-defs l2-step3-def*)

**apply** (*auto simp add: l3-step3-def*)

**done**

**lemma** *l3-step4-refines-step4*:  
 $\{R23s\}$  *l2-step4* *Rb A B Ni gn<sub>x</sub>*, *l3-step4* *Rb A B Ni gn<sub>x</sub>*  $\{>R23s\}$   
**apply** (*auto simp add: PO-rhoare-defs R23s-defs l2-step4-def*)  
**apply** (*auto simp add: l3-step4-def*)  
**done**

### 28.3.2 Intruder events

**lemma** *l3-dy-payload-refines-dy-fake-msg*:  
 $M \in \text{payload} \implies$   
 $\{R23s \cap UNIV \times l3\text{-inv}5\}$  *l2-dy-fake-msg* *M*, *l3-dy* *M*  $\{>R23s\}$   
**apply** (*auto simp add: PO-rhoare-defs R23s-defs*)  
**apply** (*auto simp add: l3-defs l2-dy-fake-msg-def dest: l3-inv5D*)  
**done**

**lemma** *l3-dy-valid-refines-dy-fake-chan*:  
 $\llbracket M \in \text{valid}; M' \in \text{abs } \{M\} \rrbracket \implies$   
 $\{R23s \cap UNIV \times (l3\text{-inv}5 \cap l3\text{-inv}6 \cap l3\text{-inv}7 \cap l3\text{-inv}8 \cap l3\text{-inv}9)\}$   
*l2-dy-fake-chan* *M'*, *l3-dy* *M*  
 $\{>R23s\}$   
**apply** (*auto simp add: PO-rhoare-defs R23s-defs, simp add: l2-dy-fake-chan-def*)  
**apply** (*auto simp add: l3-defs*)

1 subgoal

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

Insec

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

Confid

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

Auth

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

Secure

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

**done**

**lemma** *l3-dy-valid-refines-dy-fake-chan-Un*:  
 $M \in \text{valid} \implies$   
 $\{R23s \cap UNIV \times (l3\text{-inv}5 \cap l3\text{-inv}6 \cap l3\text{-inv}7 \cap l3\text{-inv}8 \cap l3\text{-inv}9)\}$   
 $\bigcup M'. l2\text{-dy-fake-chan } M', l3\text{-dy } M$   
 $\{>R23s\}$   
**by** (*auto dest: valid-abs intro: l3-dy-valid-refines-dy-fake-chan*)

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



**apply** (*auto elim!*: *absE*)  
**done**

**lemma** *l3-dy-others-refines-skip*:  
 $\llbracket M \notin \text{range } LtK; M \notin \text{valid}; M \notin \text{payload} \rrbracket \implies$   
 $\{R23s\} Id, l3\text{-dy } M \{>R23s\}$   
**apply** (*auto simp add*: *PO-rhoare-defs R23s-defs*)  
**apply** (*auto simp add*: *l3-defs*)  
**apply** (*auto elim!*: *absE intro: validI*)  
**done**

**lemma** *l3-dy-refines-dy-fake-msg-dy-fake-chan-skip*:  
 $\{R23s \cap UNIV \times (l3\text{-inv5} \cap l3\text{-inv6} \cap l3\text{-inv7} \cap l3\text{-inv8} \cap l3\text{-inv9})\}$   
 $l2\text{-dy-fake-msg } M \cup (\bigcup M'. l2\text{-dy-fake-chan } M') \cup Id, l3\text{-dy } M$   
 $\{>R23s\}$   
**by** (*cases*  $M \in \text{payload} \cup \text{valid} \cup \text{range } LtK$ )  
(*auto dest*: *l3-dy-payload-refines-dy-fake-msg l3-dy-valid-refines-dy-fake-chan-Un*  
*intro: l3-dy-isLtKey-refines-skip dest!: l3-dy-others-refines-skip*)

### 28.3.3 Compromise events

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

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

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

**lemmas** *l3-trans-refines-l2-trans =*  
*l3-step1-refines-step1 l3-step2-refines-step2 l3-step3-refines-step3 l3-step4-refines-step4*  
*l3-dy-refines-dy-fake-msg-dy-fake-chan-skip*  
*l3-lkr-others-refines-lkr-others l3-lkr-after-refines-lkr-after l3-skr-refines-skr*

**lemma** *l3-refines-init-l2 [iff]*:  
 $\text{init } l3 \subseteq R23s \text{ “ (init } l2)$   
**by** (*auto simp add*: *R23s-defs l2-defs l3-def l3-init-def*)

**lemma** *l3-refines-trans-l2* [*iff*]:  
 $\{R23s \cap (UNIV \times (l3-inv1 \cap l3-inv2 \cap l3-inv3 \cap l3-inv4))\}$  *trans l2, trans l3*  $\{> R23s\}$   
**proof** –  
**let**  $?pre' = R23s \cap (UNIV \times (l3-inv5 \cap l3-inv6 \cap l3-inv7 \cap l3-inv8 \cap l3-inv9))$   
**show** *?thesis* (**is**  $\{?pre\}$  *?t1, ?t2*  $\{> ?post\}$ )  
**proof** (*rule relhoare-conseq-left*)  
**show**  $?pre \subseteq ?pre'$   
**using** *l3-inv5-derived l3-inv6-derived l3-inv7-derived l3-inv8-derived l3-inv9-derived*  
**by** *blast*  
**next**  
**show**  $\{?pre'\}$  *?t1, ?t2*  $\{> ?post\}$   
**by** (*auto simp add: l2-def l3-def l2-trans-def l3-trans-def*  
*intro!: l3-trans-refines-l2-trans*)  
**qed**  
**qed**

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

**lemma** *l3-refines-l2* [*iff*]:  
*refines*  
 $(R23s \cap$   
 $(reach\ l2 \times (l3-inv1 \cap l3-inv2 \cap l3-inv3 \cap l3-inv4)))$   
 $med23s\ l2\ l3$   
**by** (*rule Refinement-using-invariants, auto*)

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

## 28.4 Derived invariants

### 28.4.1 inv10: secrets contain no implementation material

#### definition

*l3-inv10* :: *l3-state set*

#### where

$l3-inv10 \equiv \{s.$   
*secret*  $s \subseteq payload$   
 $\}$

**lemmas** *l3-inv10I = l3-inv10-def* [*THEN setc-def-to-intro, rule-format*]

**lemmas** *l3-inv10E = l3-inv10-def* [*THEN setc-def-to-elim, rule-format*]

**lemmas** *l3-inv10D = l3-inv10-def* [*THEN setc-def-to-dest, rule-format*]

**lemma** *l3-inv10-init* [*iff*]:

*init l3*  $\subseteq l3-inv10$

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

**lemma** *l3-inv10-trans* [*iff*]:

$\{l3-inv10\}$  *trans l3*  $\{> l3-inv10\}$

```

apply (auto simp add: PO-hoare-defs l3-nostep-defs)
apply (auto simp add: l3-defs l3-inv10-def)
done

```

```

lemma PO-l3-inv10 [iff]: reach l3  $\subseteq$  l3-inv10
by (rule inv-rule-basic) (auto)

```

```

lemma l3-obs-inv10 [iff]: oreach l3  $\subseteq$  l3-inv10
by (auto simp add: oreach-def)

```

### 28.4.2 Partial secrecy

We want to prove *l3-secrecy*, ie  $\text{synth} (\text{analz} (ik\ s)) \cap \text{secret}\ s = \{\}$ , but by refinement we only get *l3-partial-secrecy*:  $\text{dy-fake-msg} (l3\text{-state}.bad\ s) (\text{payloadSet} (ik\ s)) (\text{local.abs} (ik\ s)) \cap \text{secret}\ s = \{\}$ . This is fine if secrets contain no implementation material. Then, by *inv5*, a message in  $\text{synth} (\text{analz} (ik\ s))$  is in  $\text{dy-fake-msg} (l3\text{-state}.bad\ s) (\text{payloadSet} (ik\ s)) (\text{local.abs} (ik\ s)) \cup -\text{payload}$ , and *l3-partial-secrecy* proves it is not a secret.

#### definition

```

l3-partial-secrecy :: ('a l3-state-scheme) set
where
  l3-partial-secrecy  $\equiv$  {s.
    dy-fake-msg (bad s) (ik s  $\cap$  payload) (abs (ik s))  $\cap$  secret s = {}
  }

```

```

lemma l3-obs-partial-secrecy [iff]: oreach l3  $\subseteq$  l3-partial-secrecy
apply (rule external-invariant-translation [OF l2-obs-secrecy - l3-implements-l2])
apply (auto simp add: med23s-def l2-secrecy-def l3-partial-secrecy-def)
done

```

### 28.4.3 Secrecy

#### definition

```

l3-secrecy :: ('a l3-state-scheme) set
where
  l3-secrecy  $\equiv$  l1-secrecy

```

```

lemma l3-obs-inv5: oreach l3  $\subseteq$  l3-inv5
by (auto simp add: oreach-def)

```

```

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

```

```

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

```

### 28.4.4 Injective agreement

```

abbreviation l3-iagreement-Init  $\equiv$  l1-iagreement-Init

```

**lemma** *l3-obs-iagreement-Init* [iff]: oreach  $l3 \subseteq l3\text{-iagreement-Init}$   
**apply** (rule *external-invariant-translation*  
[OF *l2-obs-iagreement-Init - l3-implements-l2*])  
**apply** (auto simp add: *med23s-def l1-iagreement-Init-def*)  
**done**

**lemma** *l3-iagreement-Init* [iff]: reach  $l3 \subseteq l3\text{-iagreement-Init}$   
**by** (rule *external-to-internal-invariant* [OF *l3-obs-iagreement-Init*], auto)

**abbreviation** *l3-iagreement-Resp*  $\equiv$  *l1-iagreement-Resp*

**lemma** *l3-obs-iagreement-Resp* [iff]: oreach  $l3 \subseteq l3\text{-iagreement-Resp}$   
**apply** (rule *external-invariant-translation*  
[OF *l2-obs-iagreement-Resp - l3-implements-l2*])  
**apply** (auto simp add: *med23s-def l1-iagreement-Resp-def*)  
**done**

**lemma** *l3-iagreement-Resp* [iff]: reach  $l3 \subseteq l3\text{-iagreement-Resp}$   
**by** (rule *external-to-internal-invariant* [OF *l3-obs-iagreement-Resp*], auto)

**end**  
**end**

## 29 SKEME Protocol (L3 with asymmetric implementation)

```
theory sklv3-asymmetric  
imports sklv3 Implem-asymmetric  
begin  
  
interpretation sklv3-asym: sklv3 implem-asym  
by (unfold-locales)  
  
end
```

### 30 SKEME Protocol (L3 with symmetric implementation)

```
theory sklv3-symmetric  
imports sklv3 Implem-symmetric  
begin  
  
interpretation sklv3-sym: sklv3 implem-sym  
by (unfold-locales)  
  
end
```