

# Refining Authenticated Key Agreement with Strong Adversaries

Joseph Lallemand, Christoph Sprenger, and David Basin

December 14, 2021

## Contents

<b>1</b>	<b>Proving infrastructure</b>	<b>3</b>
1.1	Prover configuration . . . . .	3
1.2	Forward reasoning ("attributes") . . . . .	3
1.3	General results . . . . .	3
1.3.1	Maps . . . . .	3
1.3.2	Set . . . . .	3
1.3.3	Relations . . . . .	4
1.3.4	Lists . . . . .	4
1.3.5	Finite sets . . . . .	4
<b>2</b>	<b>Models, Invariants and Refinements</b>	<b>5</b>
2.1	Specifications, reachability, and behaviours. . . . .	5
2.1.1	Finite behaviours . . . . .	5
2.1.2	Specifications, observability, and implementation . . . . .	6
2.2	Invariants . . . . .	9
2.2.1	Hoare triples . . . . .	9
2.2.2	Characterization of reachability . . . . .	10
2.2.3	Invariant proof rules . . . . .	10
2.3	Refinement . . . . .	11
2.3.1	Relational Hoare tuples . . . . .	11
2.3.2	Refinement proof obligations . . . . .	13
2.3.3	Deriving invariants from refinements . . . . .	14
2.3.4	Refinement of specifications . . . . .	15
<b>3</b>	<b>Message definitions</b>	<b>18</b>
3.1	Messages . . . . .	18
<b>4</b>	<b>Message theory</b>	<b>27</b>
4.1	Message composition . . . . .	27
4.2	Message decomposition . . . . .	29
4.3	Lemmas about combined composition/decomposition . . . . .	31
4.4	Accessible message parts . . . . .	32
4.4.1	Lemmas about combinations with composition and decomposition . . . . .	35
4.5	More lemmas about combinations of closures . . . . .	35

<b>5</b>	<b>Environment: Dolev-Yao Intruder</b>	<b>38</b>
<b>6</b>	<b>Secrecy Model (L0)</b>	<b>39</b>
6.1	State and events . . . . .	39
6.2	Proof of secrecy invariant . . . . .	40
<b>7</b>	<b>Non-injective Agreement (L0)</b>	<b>42</b>
7.1	Signals . . . . .	42
7.2	State and events . . . . .	42
7.3	Non injective agreement invariant . . . . .	43
<b>8</b>	<b>Injective Agreement (L0)</b>	<b>45</b>
8.1	State and events . . . . .	45
8.2	Injective agreement invariant . . . . .	46
8.3	Refinement . . . . .	46
8.4	Derived invariant . . . . .	47
<b>9</b>	<b>Runs</b>	<b>48</b>
9.1	Type definitions . . . . .	48
<b>10</b>	<b>Channel Messages</b>	<b>49</b>
10.1	Channel messages . . . . .	49
10.2	Extract . . . . .	49
10.3	Fake . . . . .	50
10.4	Closure of Dolev-Yao, extract and fake . . . . .	52
10.4.1	<i>dy-fake-msg</i> : returns messages, closure of DY and extr is sufficient . . . . .	52
10.4.2	<i>dy-fake-chan</i> : returns channel messages . . . . .	53
<b>11</b>	<b>Payloads and Support for Channel Message Implementations</b>	<b>55</b>
11.1	Payload messages . . . . .	55
11.2	<i>isLtKey</i> : is a long term key . . . . .	58
11.3	<i>keys-of</i> : the long term keys of an agent . . . . .	58
11.4	<i>Keys-bad</i> : bounds on the attacker’s knowledge of long-term keys. . . . .	59
11.5	<i>broken K</i> : pairs of agents where at least one is compromised. . . . .	61
11.6	<i>Enc-keys-clean S</i> : messages with “clean” symmetric encryptions. . . . .	61
11.7	Sets of messages with particular constructors . . . . .	62
11.7.1	Lemmas for moving message sets out of <i>analz</i> . . . . .	63
<b>12</b>	<b>Assumptions for Channel Message Implementation</b>	<b>65</b>
12.1	First step: basic implementation locale . . . . .	65
12.2	Second step: basic and analyze assumptions . . . . .	66
12.3	Third step: <i>valid-implem</i> . . . . .	68
<b>13</b>	<b>Lemmas Following from Channel Message Implementation Assumptions</b>	<b>69</b>
13.1	Message implementations and abstractions . . . . .	69
13.2	Extractable messages . . . . .	70
13.2.1	Partition <i>I</i> to keep only the extractable messages . . . . .	71
13.2.2	Partition of <i>extractable</i> . . . . .	71

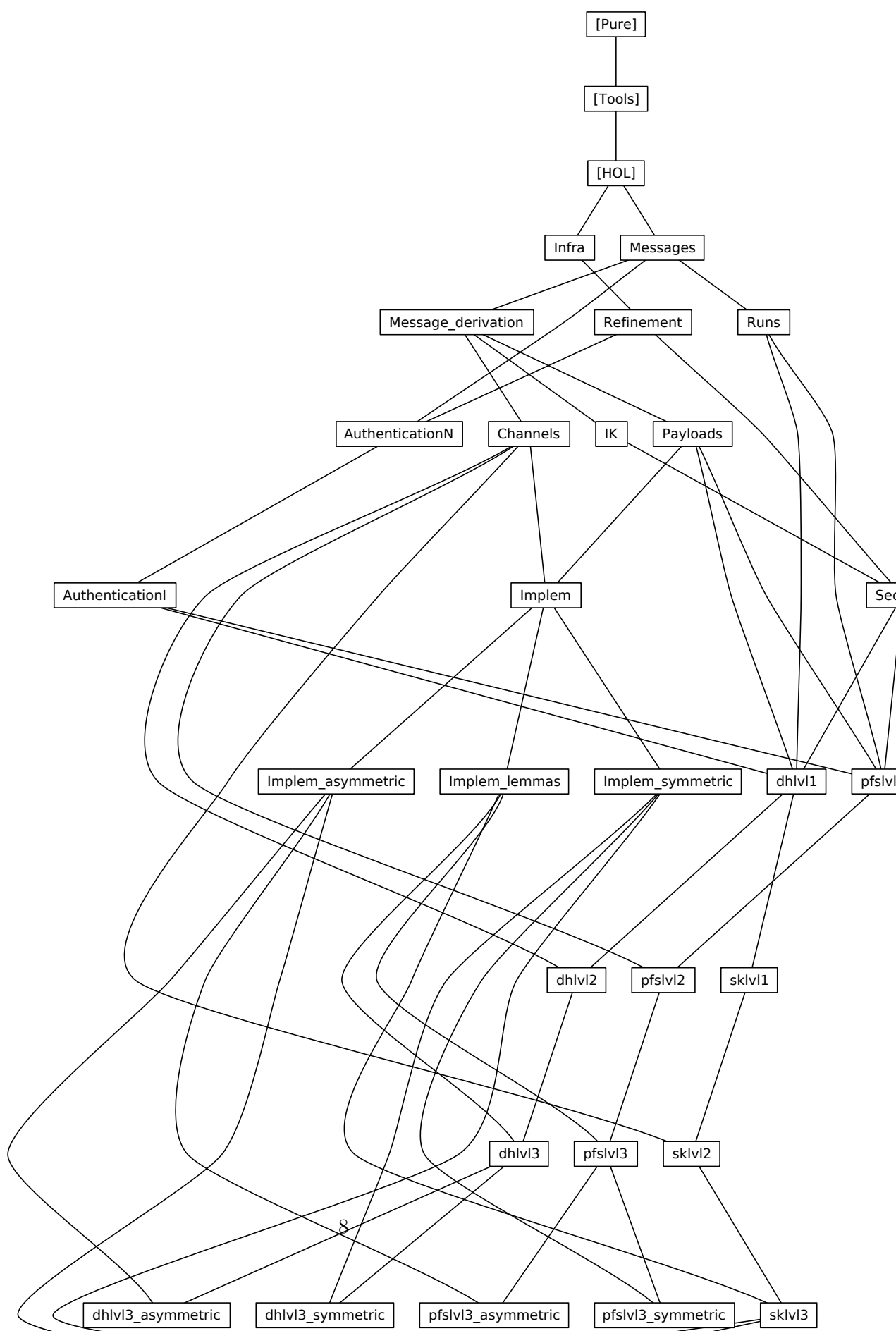
13.3	Lemmas for proving intruder refinement (L2-L3)	71
13.3.1	First: we only keep the extractable messages	72
13.3.2	Only keep the extracted messages (instead of extractable)	72
13.3.3	Keys and Tags can be moved out of the <i>analz</i>	72
13.3.4	Final lemmas, using all the previous ones	72
13.3.5	Partitioning <i>analz ik</i>	73
<b>14</b>	<b>Symmetric Implementation of Channel Messages</b>	<b>74</b>
14.1	Implementation of channel messages	74
14.2	Lemmas to pull implementation sets out of <i>analz</i>	75
14.2.1	Pull <i>implInsecSet</i> out of <i>analz</i>	75
14.3	Pull <i>implConfidSet</i> out of <i>analz</i>	75
14.4	Pull <i>implSecureSet</i> out of <i>analz</i>	76
14.5	Pull <i>implAuthSet</i> out of <i>analz</i>	77
14.6	Locale interpretations	78
<b>15</b>	<b>Asymmetric Implementation of Channel Messages</b>	<b>79</b>
15.1	Implementation of channel messages	79
15.2	Lemmas to pull implementation sets out of <i>analz</i>	80
15.2.1	Pull <i>PairAgentSet</i> out of <i>analz</i>	80
15.2.2	Pull <i>implInsecSet</i> out of <i>analz</i>	80
15.3	Pull <i>implConfidSet</i> out of <i>analz</i>	80
15.4	Pull <i>implAuthSet</i> out of <i>analz</i>	81
15.5	Pull <i>implSecureSet</i> out of <i>analz</i>	81
15.6	Locale interpretations	82
<b>16</b>	<b>Key Transport Protocol with PFS (L1)</b>	<b>84</b>
16.1	State and Events	84
16.2	Refinement: secrecy	88
16.3	Derived invariants: secrecy	90
16.4	Invariants	90
16.4.1	inv1	90
16.4.2	inv2	90
16.4.3	inv3 (derived)	91
16.5	Refinement: injective agreement	91
16.6	Derived invariants: injective agreement	93
<b>17</b>	<b>Key Transport Protocol with PFS (L2)</b>	<b>94</b>
17.1	State and Events	94
17.2	Invariants	99
17.2.1	inv1	99
17.2.2	inv2 (authentication guard)	99
17.2.3	inv3 (authentication guard)	100
17.2.4	inv4	101
17.2.5	inv5	101
17.2.6	inv6	102
17.2.7	inv7	102

17.2.8	inv8	104
17.3	Refinement	104
17.4	Derived invariants	106
<b>18</b>	<b>Key Transport Protocol with PFS (L3 locale)</b>	<b>108</b>
18.1	State and Events	108
18.2	Invariants	112
18.2.1	inv1: No long-term keys as message parts	112
18.2.2	inv2: <i>l3-state.bad s</i> indeed contains "bad" keys	112
18.2.3	inv3	113
18.2.4	inv4: the intruder knows the tags	113
18.2.5	inv5	114
18.2.6	inv6	114
18.2.7	inv7	115
18.2.8	inv8	115
18.2.9	inv9	116
18.3	Refinement	116
18.3.1	Protocol events	117
18.3.2	Intruder events	118
18.3.3	Compromise events	118
18.4	Derived invariants	119
18.4.1	inv10: secrets contain no implementation material	119
18.4.2	Partial secrecy	120
18.4.3	Secrecy	120
18.4.4	Injective agreement	120
<b>19</b>	<b>Key Transport Protocol with PFS (L3, asymmetric implementation)</b>	<b>122</b>
<b>20</b>	<b>Key Transport Protocol with PFS (L3, symmetric implementation)</b>	<b>123</b>
<b>21</b>	<b>Authenticated Diffie Hellman Protocol (L1)</b>	<b>124</b>
21.1	State and Events	124
21.2	Refinement: secrecy	129
21.3	Derived invariants: secrecy	130
21.4	Invariants: <i>Init</i> authenticates <i>Resp</i>	131
21.4.1	inv1	131
21.4.2	inv2	131
21.4.3	inv3 (derived)	132
21.5	Invariants: <i>Resp</i> authenticates <i>Init</i>	132
21.5.1	inv4	132
21.5.2	inv5	133
21.5.3	inv6 (derived)	134
21.6	Refinement: injective agreement ( <i>Init</i> authenticates <i>Resp</i> )	134
21.7	Derived invariants: injective agreement ( <i>Init</i> authenticates <i>Resp</i> )	136
21.8	Refinement: injective agreement ( <i>Resp</i> authenticates <i>Init</i> )	136
21.9	Derived invariants: injective agreement ( <i>Resp</i> authenticates <i>Init</i> )	137

<b>22 Authenticated Diffie-Hellman Protocol (L2)</b>	<b>139</b>
22.1 State and Events	139
22.2 Invariants	145
22.2.1 inv1	145
22.2.2 inv2 (authentication guard)	145
22.2.3 inv3 (authentication guard)	146
22.2.4 inv4	146
22.2.5 inv4'	147
22.2.6 inv5	147
22.2.7 inv6	148
22.2.8 inv7	148
22.2.9 inv8: form of the secrets	150
22.3 Refinement	151
22.4 Derived invariants	153
<b>23 Authenticated Diffie-Hellman Protocol (L3 locale)</b>	<b>155</b>
23.1 State and Events	155
23.2 Invariants	159
23.2.1 inv1: No long-term keys as message parts	159
23.2.2 inv2: <i>l3-state.bad s</i> indeed contains "bad" keys	160
23.2.3 inv3	160
23.2.4 inv4: the intruder knows the tags	161
23.2.5 inv5	162
23.2.6 inv6	162
23.2.7 inv7	163
23.2.8 inv8	163
23.2.9 inv9	164
23.3 Refinement	164
23.3.1 Protocol events	165
23.3.2 Intruder events	166
23.3.3 Compromise events	166
23.4 Derived invariants	167
23.4.1 inv10: secrets contain no implementation material	167
23.4.2 Partial secrecy	168
23.4.3 Secrecy	168
23.4.4 Injective agreement	168
<b>24 Authenticated Diffie-Hellman Protocol (L3, asymmetric)</b>	<b>170</b>
<b>25 Authenticated Diffie-Hellman Protocol (L3, symmetric)</b>	<b>171</b>
<b>26 SKEME Protocol (L1)</b>	<b>172</b>
26.1 State and Events	172
26.2 Refinement: secrecy	177
26.3 Derived invariants: secrecy	179
26.4 Invariants: <i>Init</i> authenticates <i>Resp</i>	179
26.4.1 inv1	179

26.4.2	inv2	180
26.4.3	inv3 (derived)	180
26.5	Invariants: Resp authenticates Init	181
26.5.1	inv4	181
26.5.2	inv5	182
26.5.3	inv6 (derived)	182
26.6	Refinement: injective agreement (Init authenticates Resp)	183
26.7	Derived invariants: injective agreement ( <i>Init</i> authenticates <i>Resp</i> )	184
26.8	Refinement: injective agreement ( <i>Resp</i> authenticates <i>Init</i> )	184
26.9	Derived invariants: injective agreement ( <i>Resp</i> authenticates <i>Init</i> )	186
<b>27</b>	<b>SKEME Protocol (L2)</b>	<b>187</b>
27.1	State and Events	187
27.2	Invariants	194
27.2.1	inv1	194
27.2.2	inv2	194
27.2.3	inv3	195
27.2.4	hmac preservation lemmas	197
27.2.5	inv4 (authentication guard)	198
27.2.6	inv5 (authentication guard)	199
27.2.7	inv6	200
27.2.8	inv6'	201
27.2.9	inv7: form of the secrets	201
27.3	Refinement	202
27.4	Derived invariants	204
<b>28</b>	<b>SKEME Protocol (L3 locale)</b>	<b>206</b>
28.1	State and Events	206
28.2	Invariants	211
28.2.1	inv1: No long-term keys as message parts	211
28.2.2	inv2: <i>l3-state.bad</i> <i>s</i> indeed contains "bad" keys	212
28.2.3	inv3	212
28.2.4	inv4: the intruder knows the tags	213
28.2.5	inv5	213
28.2.6	inv6	214
28.2.7	inv7	214
28.2.8	inv8	215
28.2.9	inv9	216
28.3	Refinement	216
28.3.1	Protocol events	217
28.3.2	Intruder events	218
28.3.3	Compromise events	218
28.4	Derived invariants	219
28.4.1	inv10: secrets contain no implementation material	219
28.4.2	Partial secrecy	220
28.4.3	Secrecy	220
28.4.4	Injective agreement	220

<b>29 SKEME Protocol (L3 with asymmetric implementation)</b>	<b>222</b>
<b>30 SKEME Protocol (L3 with symmetric implementation)</b>	<b>223</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*:  
 $f\ x = \text{Some } y \implies x \in \text{dom } f$   
<proof>

**lemma** *map-definedness-contr*:  
 $\llbracket f\ x = \text{Some } y; z \notin \text{dom } f \rrbracket \implies x \neq z$   
<proof>

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

### 1.3.2 Set

**lemma** *image-image-subset*:  $A \subseteq f^{-1}(f \cdot A)$   
<proof>

### 1.3.3 Relations

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

### 1.3.4 Lists

**lemma** *map-comp*:  $map \ (g \ o \ f) = map \ g \ o \ map \ f$   
*<proof>*

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

**lemma** *take-prefix*:  $\llbracket take \ n \ l = xs \rrbracket \implies \exists xs'. l = xs \ @ \ xs'$   
*<proof>*

### 1.3.5 Finite sets

Cardinality.

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

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

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

**lemma** *finite-zero-cardI* [*intro!*]:  
 $\llbracket A = \{\}; \text{ finite } A \rrbracket \implies card \ A = 0$   
*<proof>*

**lemma** *finite-zero-cardD* [*dest!*]:  
 $\llbracket card \ A = 0; \text{ finite } A \rrbracket \implies A = \{\}$   
*<proof>*

**end**

## 2 Models, Invariants and Refinements

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

### 2.1 Specifications, reachability, and behaviours.

Transition systems are multi-pointed graphs.

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

The inductive set of reachable states.

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

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

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

Behaviours are prefix closed.

**lemma** *beh-immediate-prefix-closed*:  
 $s \# b \in \text{beh } T \Longrightarrow b \in \text{beh } T$   
<proof>

**lemma** *beh-prefix-closed*:  
 $c @ b \in \text{beh } T \Longrightarrow b \in \text{beh } T$   
<proof>

States in behaviours are exactly reachable.

**lemma** *beh-in-reach* [*rule-format*]:  
 $b \in \text{beh } T \Longrightarrow (\forall s \in \text{set } b. s \in \text{reach } T)$   
<proof>

**lemma** *reach-in-beh*:  
**assumes**  $s \in \text{reach } T$  **shows**  $\exists b \in \text{beh } T. s \in \text{set } b$   
<proof>

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

### 2.1.2 Specifications, observability, and implementation

Specifications add an observer function to transition systems.

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

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

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

Observable behaviour and reachability.

**definition**

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

**definition**

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

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

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

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

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

**definition**

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

**where**

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

**definition**

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

**where**

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

**definition**

*observable3* :: [*'s* ⇒ *'o*, *'s set*] ⇒ *bool*

**where**

*observable3 ob P* ≡ *ob- 'ob'P* ⊆ *P* — other direction holds trivially

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

[[*observable ob P*; *ob s = ob s'*; *s' ∈ P*]] ⇒ *s ∈ P*

⟨*proof*⟩

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

⟨*proof*⟩

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

⟨*proof*⟩

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

⟨*proof*⟩

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

**lemma** *image-vimage-adjoints*: (*ob'P* ⊆ *Q*) = (*P* ⊆ *ob-'Q*)

⟨*proof*⟩

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

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

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

**lemma** *image-r-vimage-l*: [[ *Q* ⊆ *ob'P*; *observable ob P* ]] ⇒ *ob-'Q* ⊆ *P*

⟨*proof*⟩

**lemma** *vimage-l-image-r*: [[ *ob-'Q* ⊆ *P*; *Q* ⊆ *range ob* ]] ⇒ *Q* ⊆ *ob'P*

⟨*proof*⟩

Internal and external invariants

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

[[ *reach S* ⊆ *P*; (*obs S*) '*P* ⊆ *Q* ]]

⇒ *oreach S* ⊆ *Q*

⟨*proof*⟩

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

[[ *reach S* ⊆ *P*; *P* ⊆ (*obs S*) - '*Q* ]]

⇒ *oreach S* ⊆ *Q*

⟨*proof*⟩

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

[[ *oreach S* ⊆ *Q*; (*obs S*) - '*Q* ⊆ *P* ]]

⇒ *reach S* ⊆ *P*

⟨*proof*⟩

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

[[ *oreach S* ⊆ *Q*; *Q* ⊆ (*obs S*) '*P*; *observable (obs S) P* ]]

⇒ *reach S* ⊆ *P*

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

Our notion of implementation is inclusion of observable behaviours.

**definition**

*implements* :: [ $'p \Rightarrow 'o, ('s, 'o) \text{ spec}, ('t, 'p) \text{ spec}$ ]  $\Rightarrow \text{bool}$  **where**  
*implements*  $\pi$   $S_a S_c \equiv (\text{map } \pi) ('(\text{obeh } S_c) \subseteq \text{obeh } S_a$

Reflexivity and transitivity

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

$\langle \text{proof} \rangle$

**lemma** *implements-trans:*

$$\begin{aligned} & \llbracket \text{implements } \pi_1 S_1 S_2; \text{implements } \pi_2 S_2 S_3 \rrbracket \\ & \implies \text{implements } (\pi_1 \circ \pi_2) S_1 S_3 \end{aligned}$$

$\langle \text{proof} \rangle$

Preservation of external invariants

**lemma** *implements-oreach:*

$$\text{implements } \pi S_a S_c \implies \pi ('(\text{oreach } S_c) \subseteq \text{oreach } S_a$$

$\langle \text{proof} \rangle$

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

$$\begin{aligned} & \llbracket \text{oreach } S_a \subseteq Q; \text{implements } \pi S_a S_c \rrbracket \\ & \implies \pi ('(\text{oreach } S_c) \subseteq Q \end{aligned}$$

$\langle \text{proof} \rangle$

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

$$\begin{aligned} & \llbracket \text{oreach } S_a \subseteq Q; \pi - 'Q \subseteq P; \text{implements } \pi S_a S_c \rrbracket \\ & \implies \text{oreach } S_c \subseteq P \end{aligned}$$

$\langle \text{proof} \rangle$

Preservation of internal invariants

**lemma** *internal-invariant-translation:*

$$\begin{aligned} & \llbracket \text{reach } S_a \subseteq P_a; P_a \subseteq \text{obs } S_a - 'Q_a; \pi - 'Q_a \subseteq Q; \text{obs } S - 'Q \subseteq P; \\ & \quad \text{implements } \pi S_a S \rrbracket \\ & \implies \text{reach } S \subseteq P \end{aligned}$$

$\langle \text{proof} \rangle$

## 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

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

#### where

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

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

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

Some essential facts about Hoare triples.

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

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

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

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

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

$\{\{\}\} R \{> Q\}$   
 $\langle proof \rangle$

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

$\{P\} R \{> UNIV\}$   
 $\langle proof \rangle$

**lemma** *hoare-conj-right* [intro!]:

$\llbracket \{P\} R \{> Q1\}; \{P\} R \{> Q2\} \rrbracket$   
 $\implies \{P\} R \{> Q1 \cap Q2\}$   
 $\langle proof \rangle$

Special transition relations.

**lemma** *hoare-stop* [simp, intro!]:

$\{P\} \{\}\ \{> Q\}$   
 $\langle proof \rangle$

**lemma** *hoare-skip* [simp, intro!]:

$P \subseteq Q \implies \{P\} Id \{> Q\}$   
 $\langle proof \rangle$

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

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

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

$\{P\} \cup x. R x \{> Q\} = (\forall x. \{P\} R x \{> Q\})$   
 ⟨proof⟩

**lemma** *hoare-apply*:

$\{P\} R \{> Q\} \implies x \in P \implies (x, y) \in R \implies y \in Q$   
 ⟨proof⟩

### 2.2.2 Characterization of reachability

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

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

Useful consequences.

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

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

### 2.2.3 Invariant proof rules

Basic proof rule for invariants.

**lemma** *inv-rule-basic*:

$\llbracket init T \subseteq P; \{P\} (trans T) \{> P\} \rrbracket$   
 $\implies reach T \subseteq P$   
 ⟨proof⟩

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

**lemma** *inv-rule*:

$\llbracket init T \subseteq I; I \subseteq P; \{I\} (trans T) \{> I\} \rrbracket$   
 $\implies reach T \subseteq P$   
 ⟨proof⟩

The following rule is equivalent to the previous one.

**lemma** *INV-rule*:

$\llbracket init T \subseteq I; \{I \cap reach T\} (trans T) \{> I\} \rrbracket$   
 $\implies reach T \subseteq I$   
 ⟨proof⟩

Proof of equivalence.

**lemma** *inv-rule-from-INV-rule*:

$\llbracket init T \subseteq I; I \subseteq P; \{I\} (trans T) \{> I\} \rrbracket$   
 $\implies reach T \subseteq P$   
 ⟨proof⟩



**lemma** *INV-rule-from-inv-rule*:

$$\llbracket \text{init } T \subseteq I; \{I \cap \text{reach } T\} (\text{trans } T) \{> I\} \rrbracket \\ \implies \text{reach } T \subseteq I$$

*<proof>*

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

**lemma** *inv-rule-incr*:

$$\llbracket \text{init } T \subseteq I; \{I \cap J\} (\text{trans } T) \{> I\}; \text{reach } T \subseteq J \rrbracket \\ \implies \text{reach } T \subseteq I$$

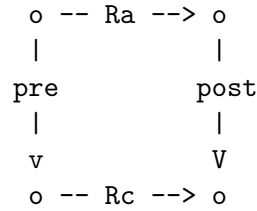
*<proof>*

## 2.3 Refinement

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

### 2.3.1 Relational Hoare tuples

Relational Hoare tuples formalize the following generalized simulation diagram:



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*)  $\equiv \lambda r s. \{(xa, x). ((\lambda x xa. (x, xa) \in r) O O (\lambda x xa. (x, xa) \in s)) xa x\}.$ )

**definition**

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

**where**

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

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

Facts about relational Hoare tuples.

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

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

*<proof>*

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

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

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

**lemma** *Domain-rel-comp* [intro]:  
 $Domain pre \subseteq R \implies Domain (pre \ O \ Rc) \subseteq R$   
 $\langle proof \rangle$

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

Reflexivity and transitivity.

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

**lemma** *rhoare-trans*:  
 $\llbracket \{R1\} T1, T2 \{> R1\}; \{R2\} T2, T3 \{> R2\} \rrbracket$   
 $\implies \{R1 \ O \ R2\} T1, T3 \{> R1 \ O \ R2\}$   
 $\langle proof \rangle$

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

**lemma** *relhoare-conj-right-det*:  
 $\llbracket \{pre\} Ra, Rc \{> post1\}; \{pre\} Ra, Rc \{> post2\};$   
 $\quad \textit{single-valued } Ra \rrbracket$  — only for deterministic *Ra*!  
 $\implies \{pre\} Ra, Rc \{> post1 \cap post2\}$   
 $\langle proof \rangle$

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

Separate rule for cartesian products.

**corollary** *relhoare-cartesian*:  
 $\llbracket \{Domain pre\} Ra \{> I\}; \{Range pre\} Rc \{> J\};$   
 $\quad \{pre\} Ra, Rc \{> post\} \rrbracket$  — any *post*, including *UNIV*!  
 $\implies \{pre\} Ra, Rc \{> I \times J\}$   
 $\langle proof \rangle$

Unions of transition relations.

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

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

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

$$\begin{aligned} & \{pre\} Ra, \bigcup x. Rc x \{> post\} = (\forall x. \{pre\} Ra, Rc x \{> post\}) \\ \langle proof \rangle \end{aligned}$$

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

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

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

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

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

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

Inclusion of abstract transition relations.

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

$$\begin{aligned} & \llbracket \{pre\} Ra', Rc \{> post\}; Ra' \subseteq Ra \rrbracket \\ & \implies \{pre\} Ra, Rc \{> post\} \\ \langle proof \rangle \end{aligned}$$

### 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**

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

**where**

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

**lemma** *PO-refinesI*:

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

$\langle proof \rangle$

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

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

$\implies P$

*<proof>*

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

**lemma** *refine-basic:*

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

*<proof>*

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

The hypotheses state that these state predicates are indeed invariants. Note that the precondition of the invariant preservation hypotheses for  $I$  and  $J$  are strengthened by adding the predicates  $\text{Domain } (R \cap \text{UNIV} \times J)$  and  $\text{Range } (R \cap I \times \text{UNIV})$ , 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:*

$$\llbracket \text{init } Tc \subseteq R \text{ ``}(\text{init } Ta); \text{init } Ta \subseteq I; \text{init } Tc \subseteq J \rrbracket \\ \implies \text{init } Tc \subseteq (R \cap I \times J) \text{ ``}(\text{init } Ta)$$

*<proof>*

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

$$\llbracket \{R \cap I \times J\} (\text{trans } Ta), (\text{trans } Tc) \{> R\}; \\ \{I \cap \text{Domain } (R \cap \text{UNIV} \times J)\} (\text{trans } Ta) \{> I\}; \\ \{J \cap \text{Range } (R \cap I \times \text{UNIV})\} (\text{trans } Tc) \{> J\} \rrbracket \\ \implies \{R \cap I \times J\} (\text{trans } Ta), (\text{trans } Tc) \{> R \cap I \times J\}$$

*<proof>*

This is our main rule for refinements.

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

$$\llbracket \{R \cap I \times J\} (\text{trans } Ta), (\text{trans } Tc) \{> R\}; \\ \{I \cap \text{Domain } (R \cap \text{UNIV} \times J)\} (\text{trans } Ta) \{> I\}; \\ \{J \cap \text{Range } (R \cap I \times \text{UNIV})\} (\text{trans } Tc) \{> J\}; \\ \text{init } Tc \subseteq R \text{ ``}(\text{init } Ta); \\ \text{init } Ta \subseteq I; \text{init } Tc \subseteq J \rrbracket \\ \implies \text{PO-refines } (R \cap I \times J) \text{ } Ta \text{ } Tc$$

*<proof>*

### 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:*

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

*<proof>*

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

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

*<proof>*

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

The following rules are more useful in proofs.

**corollary** *INV-init-from-refinement*:  
 $\llbracket PO\text{-refines } R \text{ } Ta \text{ } Tc; Range \text{ } R \subseteq I \rrbracket$   
 $\implies init \text{ } Tc \subseteq I$   
 $\langle proof \rangle$

**corollary** *INV-trans-from-refinement*:  
 $\llbracket PO\text{-refines } R \text{ } Ta \text{ } Tc; K \subseteq Range \text{ } R; Range \text{ } R \subseteq I \rrbracket$   
 $\implies \{K\} \text{ trans } Tc \{> I\}$   
 $\langle proof \rangle$

**corollary** *INV-from-refinement*:  
 $\llbracket PO\text{-refines } R \text{ } Ta \text{ } Tc; Range \text{ } R \subseteq I \rrbracket$   
 $\implies reach \text{ } Tc \subseteq I$   
 $\langle proof \rangle$

### 2.3.4 Refinement of specifications

Lift relation membership to finite sequences

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

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

For each concrete behaviour there is a related abstract one.

**lemma** *behaviour-refinement*:  
 $\llbracket PO\text{-refines } R \text{ } Ta \text{ } Tc; bc \in beh \text{ } Tc \rrbracket$   
 $\implies \exists ba \in beh \text{ } Ta. (ba, bc) \in seq\text{-lift } R$   
 $\langle proof \rangle$

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

**lemma** *obs-consistent-refl [iff]*:  $obs\text{-consistent } Id \text{ } id \text{ } S \text{ } S$   
 $\langle proof \rangle$

**lemma** *obs-consistent-trans* [intro]:  
 $\llbracket \text{obs-consistent } R1 \text{ } \pi1 \text{ } S1 \text{ } S2; \text{obs-consistent } R2 \text{ } \pi2 \text{ } S2 \text{ } S3 \rrbracket$   
 $\implies \text{obs-consistent } (R1 \text{ } O \text{ } R2) (\pi1 \text{ } o \text{ } \pi2) S1 \text{ } S3$   
 $\langle \text{proof} \rangle$

**lemma** *obs-consistent-empty*:  $\text{obs-consistent } \{ \} \pi \text{ } Sa \text{ } Sc$   
 $\langle \text{proof} \rangle$

**lemma** *obs-consistent-conj1* [intro]:  
 $\text{obs-consistent } R \pi \text{ } Sa \text{ } Sc \implies \text{obs-consistent } (R \cap R') \pi \text{ } Sa \text{ } Sc$   
 $\langle \text{proof} \rangle$

**lemma** *obs-consistent-conj2* [intro]:  
 $\text{obs-consistent } R \pi \text{ } Sa \text{ } Sc \implies \text{obs-consistent } (R' \cap R) \pi \text{ } Sa \text{ } Sc$   
 $\langle \text{proof} \rangle$

**lemma** *obs-consistent-behaviours*:  
 $\llbracket \text{obs-consistent } R \pi \text{ } Sa \text{ } Sc; bc \in \text{beh } Sc; ba \in \text{beh } Sa; (ba, bc) \in \text{seq-lift } R \rrbracket$   
 $\implies \text{map } \pi (\text{map } (\text{obs } Sc) bc) = \text{map } (\text{obs } Sa) ba$   
 $\langle \text{proof} \rangle$

Definition of refinement proof obligations.

**definition**  
 $\text{refines} ::$   
 $\llbracket ('s \times 't) \text{ set}, 'p \Rightarrow 'o, ('s, 'o) \text{ spec}, ('t, 'p) \text{ spec} \rrbracket \Rightarrow \text{bool}$

**where**  
 $\text{refines } R \pi \text{ } Sa \text{ } Sc \equiv \text{obs-consistent } R \pi \text{ } Sa \text{ } Sc \wedge \text{PO-refines } R \text{ } Sa \text{ } Sc$

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

**lemma** *refinesI*:  
 $\llbracket \text{PO-refines } R \text{ } Sa \text{ } Sc; \text{obs-consistent } R \pi \text{ } Sa \text{ } Sc \rrbracket$   
 $\implies \text{refines } R \pi \text{ } Sa \text{ } Sc$   
 $\langle \text{proof} \rangle$

**lemma** *refinesE* [elim]:  
 $\llbracket \text{refines } R \pi \text{ } Sa \text{ } Sc; \llbracket \text{PO-refines } R \text{ } Sa \text{ } Sc; \text{obs-consistent } R \pi \text{ } Sa \text{ } Sc \rrbracket \implies P \rrbracket$   
 $\implies P$   
 $\langle \text{proof} \rangle$

Reflexivity and transitivity of refinement.

**lemma** *refinement-reflexive*:  $\text{refines } Id \text{ } id \text{ } S \text{ } S$   
 $\langle \text{proof} \rangle$

**lemma** *refinement-transitive*:  
 $\llbracket \text{refines } R1 \text{ } \pi1 \text{ } S1 \text{ } S2; \text{refines } R2 \text{ } \pi2 \text{ } S2 \text{ } S3 \rrbracket$   
 $\implies \text{refines } (R1 \text{ } O \text{ } R2) (\pi1 \text{ } o \text{ } \pi2) S1 \text{ } S3$   
 $\langle \text{proof} \rangle$

Soundness of refinement for proving implementation

**lemma** *observable-behaviour-refinement*:

$\llbracket \text{refines } R \text{ pi } Sa \text{ Sc}; bc \in \text{obeh } Sc \rrbracket \implies \text{map } pi \text{ bc} \in \text{obeh } Sa$   
 $\langle \text{proof} \rangle$

**theorem** *refinement-soundness:*

$\text{refines } R \text{ pi } Sa \text{ Sc} \implies \text{implements } pi \text{ Sa } Sc$   
 $\langle \text{proof} \rangle$

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:*

$\text{refines } R \text{ pi } Sa \text{ Sc} \implies \text{refines } (R \cap \text{reach } Sa \times \text{reach } Sc) \text{ pi } Sa \text{ Sc}$   
 $\langle \text{proof} \rangle$

Inheritance of internal invariants through refinements

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

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

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

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

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

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

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

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

**shows**  $\text{reach } Sc \subseteq K$

$\langle \text{proof} \rangle$

**end**

### 3 Message definitions

```
theory Messages
imports Main
begin
```

#### 3.1 Messages

Agents

```
datatype
  agent = Agent nat
```

Nonces

```
typedecl fid-t
```

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

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

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

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

Keys

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

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

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

Messages

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



```

| cHash cmsg
| cTag tag
| cExp cmsg cmsg

```

**fun** *catomic* :: *cmsg*  $\Rightarrow$  *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*  $\Rightarrow$  *cmsg*  $\Rightarrow$  *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  $\implies$  eq c d  $\implies$  eq (cPair a c) (cPair b d)
| Enc[intro]:eq a b  $\implies$  eq c d  $\implies$  eq (cEnc a c) (cEnc b d)
| Aenc[intro]:eq a b  $\implies$  eq c d  $\implies$  eq (cAenc a c) (cAenc b d)
| Sign[intro]:eq a b  $\implies$  eq c d  $\implies$  eq (cSign a c) (cSign b d)
| Hash[intro]:eq a b  $\implies$  eq (cHash a) (cHash b)
| Exp[intro]:eq a b  $\implies$  eq c d  $\implies$  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  $\implies$  eq b c  $\implies$  eq a c
```

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

*<proof>*

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

*<proof>*

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

*<proof>*

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

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

*<proof>*

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

*<proof>*

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

*<proof>*

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

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

Quotient construction

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

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

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

Commutativity of exponents

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

**lift-definition** *atomic* :: *msg*  $\Rightarrow$  *bool* **is** *catomic* ⟨*proof*⟩

## abbreviation

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

**lemma** *atomic-Agent* [simp, intro]:  $atomic\ (Agent\ X) \langle proof \rangle$   
**lemma** *atomic-Tag* [simp, intro]:  $atomic\ (Tag\ X) \langle proof \rangle$   
**lemma** *atomic-Nonce* [simp, intro]:  $atomic\ (Nonce\ X) \langle proof \rangle$   
**lemma** *atomic-Number* [simp, intro]:  $atomic\ (Number\ X) \langle proof \rangle$   
**lemma** *atomic-LtK* [simp, intro]:  $atomic\ (LtK\ X) \langle proof \rangle$   
**lemma** *atomic-EphK* [simp, intro]:  $atomic\ (EphK\ X) \langle proof \rangle$

**lemma** *non-atomic-Pair* [simp]:  $\neg atomic\ (Pair\ x\ y) \langle proof \rangle$   
**lemma** *non-atomic-Enc* [simp]:  $\neg atomic\ (Enc\ x\ y) \langle proof \rangle$   
**lemma** *non-atomic-Aenc* [simp]:  $\neg atomic\ (Aenc\ x\ y) \langle proof \rangle$   
**lemma** *non-atomic-Sign* [simp]:  $\neg atomic\ (Sign\ x\ y) \langle proof \rangle$   
**lemma** *non-atomic-Exp* [simp]:  $\neg atomic\ (Exp\ x\ y) \langle proof \rangle$   
**lemma** *non-atomic-Hash* [simp]:  $\neg atomic\ (Hash\ x) \langle proof \rangle$

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

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

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

**lemma** *Number-Aenc*:  $\text{Number } X \neq \text{Aenc } X' Y' \langle \text{proof} \rangle$

**lemma** *Number-Sign*:  $\text{Number } X \neq \text{Sign } X' Y' \langle \text{proof} \rangle$

**lemma** *Number-Exp*:  $\text{Number } X \neq \text{Exp } X' Y' \langle \text{proof} \rangle$

**lemma** *Hash-Nonce*:  $\text{Hash } X \neq \text{Nonce } X' \langle \text{proof} \rangle$

**lemma** *Hash-Agent*:  $\text{Hash } X \neq \text{Agent } X' \langle \text{proof} \rangle$

**lemma** *Hash-Number*:  $\text{Hash } X \neq \text{Number } X' \langle \text{proof} \rangle$

**lemma** *Hash-Hash*:  $(\text{Hash } X = \text{Hash } X') = (X = X') \langle \text{proof} \rangle$

**lemma** *Hash-Tag*:  $\text{Hash } X \neq \text{Tag } X' \langle \text{proof} \rangle$

**lemma** *Hash-EphK*:  $\text{Hash } X \neq \text{EphK } X' \langle \text{proof} \rangle$

**lemma** *Hash-LtK*:  $\text{Hash } X \neq \text{LtK } X' \langle \text{proof} \rangle$

**lemma** *Hash-Pair*:  $\text{Hash } X \neq \text{Pair } X' Y' \langle \text{proof} \rangle$

**lemma** *Hash-Enc*:  $\text{Hash } X \neq \text{Enc } X' Y' \langle \text{proof} \rangle$

**lemma** *Hash-Aenc*:  $\text{Hash } X \neq \text{Aenc } X' Y' \langle \text{proof} \rangle$

**lemma** *Hash-Sign*:  $\text{Hash } X \neq \text{Sign } X' Y' \langle \text{proof} \rangle$

**lemma** *Hash-Exp*:  $\text{Hash } X \neq \text{Exp } X' Y' \langle \text{proof} \rangle$

**lemma** *Tag-Nonce*:  $\text{Tag } X \neq \text{Nonce } X' \langle \text{proof} \rangle$

**lemma** *Tag-Agent*:  $\text{Tag } X \neq \text{Agent } X' \langle \text{proof} \rangle$

**lemma** *Tag-Number*:  $\text{Tag } X \neq \text{Number } X' \langle \text{proof} \rangle$

**lemma** *Tag-Hash*:  $\text{Tag } X \neq \text{Hash } X' \langle \text{proof} \rangle$

**lemma** *Tag-Tag*:  $(\text{Tag } X = \text{Tag } X') = (X = X') \langle \text{proof} \rangle$

**lemma** *Tag-EphK*:  $\text{Tag } X \neq \text{EphK } X' \langle \text{proof} \rangle$

**lemma** *Tag-LtK*:  $\text{Tag } X \neq \text{LtK } X' \langle \text{proof} \rangle$

**lemma** *Tag-Pair*:  $\text{Tag } X \neq \text{Pair } X' Y' \langle \text{proof} \rangle$

**lemma** *Tag-Enc*:  $\text{Tag } X \neq \text{Enc } X' Y' \langle \text{proof} \rangle$

**lemma** *Tag-Aenc*:  $\text{Tag } X \neq \text{Aenc } X' Y' \langle \text{proof} \rangle$

**lemma** *Tag-Sign*:  $\text{Tag } X \neq \text{Sign } X' Y' \langle \text{proof} \rangle$

**lemma** *Tag-Exp*:  $\text{Tag } X \neq \text{Exp } X' Y' \langle \text{proof} \rangle$

**lemma** *EphK-Nonce*:  $\text{EphK } X \neq \text{Nonce } X' \langle \text{proof} \rangle$

**lemma** *EphK-Agent*:  $\text{EphK } X \neq \text{Agent } X' \langle \text{proof} \rangle$

**lemma** *EphK-Number*:  $\text{EphK } X \neq \text{Number } X' \langle \text{proof} \rangle$

**lemma** *EphK-Hash*:  $\text{EphK } X \neq \text{Hash } X' \langle \text{proof} \rangle$

**lemma** *EphK-Tag*:  $\text{EphK } X \neq \text{Tag } X' \langle \text{proof} \rangle$

**lemma** *EphK-EphK*:  $(\text{EphK } X = \text{EphK } X') = (X = X') \langle \text{proof} \rangle$

**lemma** *EphK-LtK*:  $\text{EphK } X \neq \text{LtK } X' \langle \text{proof} \rangle$

**lemma** *EphK-Pair*:  $\text{EphK } X \neq \text{Pair } X' Y' \langle \text{proof} \rangle$

**lemma** *EphK-Enc*:  $\text{EphK } X \neq \text{Enc } X' Y' \langle \text{proof} \rangle$

**lemma** *EphK-Aenc*:  $\text{EphK } X \neq \text{Aenc } X' Y' \langle \text{proof} \rangle$

**lemma** *EphK-Sign*:  $\text{EphK } X \neq \text{Sign } X' Y' \langle \text{proof} \rangle$

**lemma** *EphK-Exp*:  $\text{EphK } X \neq \text{Exp } X' Y' \langle \text{proof} \rangle$

**lemma** *LtK-Nonce*:  $\text{LtK } X \neq \text{Nonce } X' \langle \text{proof} \rangle$

**lemma** *LtK-Agent*:  $\text{LtK } X \neq \text{Agent } X' \langle \text{proof} \rangle$

**lemma** *LtK-Number*:  $\text{LtK } X \neq \text{Number } X' \langle \text{proof} \rangle$

**lemma** *LtK-Hash*:  $\text{LtK } X \neq \text{Hash } X' \langle \text{proof} \rangle$

**lemma** *LtK-Tag*:  $\text{LtK } X \neq \text{Tag } X' \langle \text{proof} \rangle$

**lemma** *LtK-EphK*:  $\text{LtK } X \neq \text{EphK } X' \langle \text{proof} \rangle$

**lemma** *LtK-LtK*:  $(\text{LtK } X = \text{LtK } X') = (X = X') \langle \text{proof} \rangle$

**lemma** *LtK-Pair*:  $\text{LtK } X \neq \text{Pair } X' Y' \langle \text{proof} \rangle$

**lemma** *LtK-Enc*:  $\text{LtK } X \neq \text{Enc } X' Y' \langle \text{proof} \rangle$

**lemma** *LtK-Aenc*:  $\text{LtK } X \neq \text{Aenc } X' Y' \langle \text{proof} \rangle$

**lemma** *LtK-Sign*:  $LtK\ X \neq Sign\ X'\ Y' \langle proof \rangle$   
**lemma** *LtK-Exp*:  $LtK\ X \neq Exp\ X'\ Y' \langle proof \rangle$

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

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

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

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

**lemma** *Sign-Exp*:  $\text{Sign } X Y \neq \text{Exp } X' Y' \langle \text{proof} \rangle$

**lemma** *Exp-Nonce*:  $\text{Exp } X Y \neq \text{Nonce } X' \langle \text{proof} \rangle$

**lemma** *Exp-Agent*:  $\text{Exp } X Y \neq \text{Agent } X' \langle \text{proof} \rangle$

**lemma** *Exp-Number*:  $\text{Exp } X Y \neq \text{Number } X' \langle \text{proof} \rangle$

**lemma** *Exp-Hash*:  $\text{Exp } X Y \neq \text{Hash } X' \langle \text{proof} \rangle$

**lemma** *Exp-Tag*:  $\text{Exp } X Y \neq \text{Tag } X' \langle \text{proof} \rangle$

**lemma** *Exp-EphK*:  $\text{Exp } X Y \neq \text{EphK } X' \langle \text{proof} \rangle$

**lemma** *Exp-LtK*:  $\text{Exp } X Y \neq \text{LtK } X' \langle \text{proof} \rangle$

**lemma** *Exp-Pair*:  $\text{Exp } X Y \neq \text{Pair } X' Y' \langle \text{proof} \rangle$

**lemma** *Exp-Enc*:  $\text{Exp } X Y \neq \text{Enc } X' Y' \langle \text{proof} \rangle$

**lemma** *Exp-Aenc*:  $\text{Exp } X Y \neq \text{Aenc } X' Y' \langle \text{proof} \rangle$

**lemma** *Exp-Sign*:  $\text{Exp } X Y \neq \text{Sign } X' Y' \langle \text{proof} \rangle$

**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**

$$\text{AuthTag} \equiv \text{Tag } \text{auth}$$
**abbreviation**

$$\text{ConfidTag} \equiv \text{Tag } \text{confid}$$
**abbreviation**

$$\text{SecureTag} \equiv \text{Tag } \text{secure}$$
**abbreviation**

$$\text{Tags} \equiv \text{range } \text{Tag}$$
**abbreviation**

$$\begin{aligned} \text{NonceF} &:: \text{fresh-}t \Rightarrow \text{msg } \mathbf{where} \\ \text{NonceF } N &\equiv \text{Nonce } (\text{nonce-fresh } N) \end{aligned}$$
**abbreviation**

$$\begin{aligned} \text{NonceA} &:: \text{nat} \Rightarrow \text{msg } \mathbf{where} \\ \text{NonceA } N &\equiv \text{Nonce } (\text{nonce-atk } N) \end{aligned}$$
**abbreviation**

$$\begin{aligned} \text{shrK} &:: \text{agent} \Rightarrow \text{agent} \Rightarrow \text{msg } \mathbf{where} \\ \text{shrK } A \ B &\equiv \text{LtK } (\text{sharK } A \ B) \end{aligned}$$
**abbreviation**

$$\begin{aligned} \text{pubK} &:: \text{agent} \Rightarrow \text{msg } \mathbf{where} \\ \text{pubK } A &\equiv \text{LtK } (\text{publK } A) \end{aligned}$$
**abbreviation**

$$\begin{aligned} \text{priK} &:: \text{agent} \Rightarrow \text{msg } \mathbf{where} \\ \text{priK } A &\equiv \text{LtK } (\text{privK } A) \end{aligned}$$
**abbreviation**

$$\begin{aligned} \text{epubK} &:: \text{nonce-}t \Rightarrow \text{msg } \mathbf{where} \\ \text{epubK } N &\equiv \text{EphK } (\text{epublK } N) \end{aligned}$$
**abbreviation**

$$\begin{aligned} \text{epriK} &:: \text{nonce-}t \Rightarrow \text{msg } \mathbf{where} \\ \text{epriK } N &\equiv \text{EphK } (\text{eprivK } N) \end{aligned}$$
**abbreviation**

$$\begin{aligned} \text{epubKF} &:: \text{fresh-}t \Rightarrow \text{msg } \mathbf{where} \\ \text{epubKF } N &\equiv \text{EphK } (\text{epublK } (\text{nonce-fresh } N)) \end{aligned}$$
**abbreviation**

$$\begin{aligned} \text{epriKF} &:: \text{fresh-}t \Rightarrow \text{msg } \mathbf{where} \\ \text{epriKF } N &\equiv \text{EphK } (\text{eprivK } (\text{nonce-fresh } N)) \end{aligned}$$
**abbreviation**

$$\begin{aligned} \text{epubKA} &:: \text{nat} \Rightarrow \text{msg } \mathbf{where} \\ \text{epubKA } N &\equiv \text{EphK } (\text{epublK } (\text{nonce-atk } N)) \end{aligned}$$
**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**

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

**translations**

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

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:**

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

$\langle proof \rangle$

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

$\langle proof \rangle$

**lemma eq-expexpgen:**

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

$\langle proof \rangle$

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

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

$\langle proof \rangle$

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

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

$\langle proof \rangle$

**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]:  $G \subseteq H \Longrightarrow \text{synth } G \subseteq \text{synth } H$   
(proof)

**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)  
(proof)

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

$\langle \text{proof} \rangle$

**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*:  $H \subseteq \text{synth } H$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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

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

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

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

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

## 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]:

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

| *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*:  $G \subseteq H \implies \text{analz}(G) \subseteq \text{analz}(H)$

⟨proof⟩

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

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

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

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

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

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

$\langle \text{proof} \rangle$

**lemma** *analz-cong*:

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

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

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

$\langle \text{proof} \rangle$

**lemma** *analz-insert-cong*:  
 $analz\ H = analz\ H' \implies$   
 $analz\ (insert\ X\ H) = analz\ (insert\ X\ H')$   
 $\langle proof \rangle$

**lemma** *analz-trivial*:  
 $\forall X\ Y. Pair\ X\ Y \notin H \implies$   
 $\forall X\ Y. Enc\ X\ Y \notin H \implies$   
 $\forall X\ Y. Aenc\ X\ Y \notin H \implies$   
 $\forall X\ Y. Sign\ X\ Y \notin H \implies$   
 $analz\ H = H$   
 $\langle proof \rangle$

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

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

Lemmas about *analz* and *insert*.

**lemma** *analz-insert-Agent* [simp]:  
 $analz\ (insert\ (Agent\ A)\ H) = insert\ (Agent\ A)\ (analz\ H)$   
 $\langle proof \rangle$

### 4.3 Lemmas about combined composition/decomposition

**lemma** *synth-analz-incr*:  $H \subseteq synth\ (analz\ H)$   
 $\langle proof \rangle$

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

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

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

**lemma** *lem1*:  
 $Y \in synth\ (analz\ (synth\ G \cup H) \cap (analz\ (G \cup H) \cup synth\ G))$   
 $\implies Y \in synth\ (analz\ (G \cup H))$   
 $\langle proof \rangle$

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

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

**lemma** *analz-synth*:  $analz\ (synth\ H) = analz\ H \cup synth\ H$   
 $\langle proof \rangle$

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

$\langle proof \rangle$

**lemma** *synth-analz-synth* [simp]:  $synth (analz (synth H)) = synth (analz H)$

$\langle proof \rangle$

**lemma** *analz-synth-analz* [simp]:  $analz (synth (analz H)) = synth (analz H)$

$\langle proof \rangle$

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

$\langle proof \rangle$

**lemma** *insert-subset-synth-analz* [simp]:

$X \in synth (analz H) \implies insert X H \subseteq synth (analz H)$

$\langle proof \rangle$

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

**assumes**  $X \in synth (analz H)$

**shows**  $synth (analz (insert X H)) = synth (analz H)$

$\langle proof \rangle$

#### 4.4 Accessible message parts

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

**inductive-set**

*parts* ::  $msg\ set \implies msg\ set$

**for**  $H$  ::  $msg\ set$

**where**

*Inj* [intro]:  $X \in H \implies X \in parts\ H$

| *Fst* [intro]:  $Pair\ X\ Y \in parts\ H \implies X \in parts\ H$

| *Snd* [intro]:  $Pair\ X\ Y \in parts\ H \implies Y \in parts\ H$

| *Dec* [intro]:  $Enc\ X\ Y \in parts\ H \implies X \in parts\ H$

| *Adec* [intro]:  $Aenc\ X\ Y \in parts\ H \implies X \in parts\ H$

| *Sign-getmsg* [intro]:  $Sign\ X\ Y \in parts\ H \implies X \in parts\ H$

Lemmas about accessible message parts.

**lemma** *parts-mono* [mono-set]:  $G \subseteq H \implies parts\ G \subseteq parts\ H$

$\langle proof \rangle$

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

**lemma** *Pair-parts* [elim]:

$\llbracket Pair\ X\ Y \in parts\ H; \llbracket X \in parts\ H; Y \in parts\ H \rrbracket \implies P \rrbracket \implies P$

$\langle proof \rangle$

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

*<proof>*

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

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

*<proof>*

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

*<proof>*

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

*<proof>*

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

*<proof>*

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

*parts Tags = Tags*

*<proof>*

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

*<proof>*

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

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

*<proof>*

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

*<proof>*

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

**assumes** *X  $\in$  G*

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

*<proof>*

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

*<proof>*

**lemma** *parts-insert2:*

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

*<proof>*

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

*<proof>*

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

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

*<proof>*

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

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

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

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

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

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

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

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

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

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

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

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

**lemma** *parts-insert-Enc* [*simp*]:  
 $\text{parts} (\text{insert} (\text{Enc } X \ Y) \ H) = \text{insert} (\text{Enc } X \ Y) (\text{parts } \{X\} \cup \text{parts } H)$   
*<proof>*

**lemma** *parts-insert-Aenc* [*simp*]:  
 $\text{parts} (\text{insert} (\text{Aenc } X \ Y) \ H) = \text{insert} (\text{Aenc } X \ Y) (\text{parts } \{X\} \cup \text{parts } H)$



$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

**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]:  $\text{parts } (\text{analz } H) = \text{parts } H$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**lemma** *Fake-parts-insert*:

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

$\langle \text{proof} \rangle$

**lemma** *Fake-parts-insert-in-Un*:

$$Z \in \text{parts } (\text{insert } X \ H) \implies$$

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

$$Z \in \text{synth } (\text{analz } H) \cup \text{parts } H$$

$\langle \text{proof} \rangle$

**lemma** *analz-conj-parts* [simp]:

$$X \in \text{analz } H \wedge X \in \text{parts } H \longleftrightarrow X \in \text{analz } H$$

$\langle \text{proof} \rangle$

**lemma** *analz-disj-parts* [simp]:

$$X \in \text{analz } H \vee X \in \text{parts } H \longleftrightarrow X \in \text{parts } H$$

$\langle \text{proof} \rangle$

#### 4.5 More lemmas about combinations of closures

Combinations of *synth* and *analz*.

**lemma** *Pair-synth-analz* [simp]:

$\text{Pair } X \ Y \in \text{synth } (\text{analz } H) \longleftrightarrow X \in \text{synth } (\text{analz } H) \wedge Y \in \text{synth } (\text{analz } H)$   
 $\langle \text{proof} \rangle$

**lemma** *Enc-synth-analz*:

$Y \in \text{synth } (\text{analz } H) \implies$   
 $(\text{Enc } X \ Y \in \text{synth } (\text{analz } H)) \longleftrightarrow (X \in \text{synth } (\text{analz } H))$   
 $\langle \text{proof} \rangle$

**lemma** *Hash-synth-analz* [simp]:

$X \notin \text{synth } (\text{analz } H) \implies$   
 $(\text{Hash } (\text{Pair } X \ Y) \in \text{synth } (\text{analz } H)) \longleftrightarrow (\text{Hash } (\text{Pair } X \ Y) \in \text{analz } H)$   
 $\langle \text{proof} \rangle$

**lemma** *gen-analz-insert-eq*:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$analz\ X \subseteq synth\ (analz\ X') \implies analz\ (X \cup Y) \subseteq synth\ (analz\ (X' \cup Y))$   
*<proof>*

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

$analz\ X \subseteq synth\ (analz\ X') \implies analz\ (Y \cup X) \subseteq synth\ (analz\ (Y \cup X'))$   
*<proof>*

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

$analz\ X \subseteq synth\ (analz\ X') \implies analz\ (insert\ Y\ X) \subseteq synth\ (analz\ (insert\ Y\ X'))$   
*<proof>*

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

**assumes**  $Y \in analz\ (insert\ X\ H)$  **and**  $X \in synth\ (analz\ G)$   
**shows**  $Y \in synth\ (analz\ G) \cup analz\ (G \cup H)$   
*<proof>*

**end**

## 5 Environment: Dolev-Yao Intruder

```
theory IK
imports Message-derivation
begin
```

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

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

Dolev-Yao intruder event adds a derived message.

```
definition
  ik-dy :: msg  $\Rightarrow$  ('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'
<proof>
```

```
lemma ik-trans-synth-analz-ik-increasing:
  (s, s')  $\in$  ik-trans  $\implies$  synth (analz (ik s))  $\subseteq$  synth (analz (ik s'))
<proof>
```

```
end
```

## 6 Secrecy Model (L0)

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

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

### 6.1 State and events

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

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

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

#### definition

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

#### where

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

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

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

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

#### definition

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

#### where

```
s0-add-secret m ≡ {(s,s').
```

```
— guard
```

```
m ∉ synth (analz (ik s)) ∧
```

```
— action
```

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

```
}
```

#### definition

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

#### where

```
s0-learn m ≡ {(s,s').
```

```
— guard
```

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

```
— action
```

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

```
}
```

#### definition

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

#### where

```
s0-learn' m ≡ {(s,s').
```

```
— guard
```

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

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

**definition**

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

**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**

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

**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 (\$   
 $\text{init} = s0\text{-init},$   
 $\text{trans} = s0\text{-trans},$   
 $\text{obs} = id$   
 $\left.)$

**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]:  $\text{obs } s0 = id$

$\langle \text{proof} \rangle$

## 6.2 Proof of secrecy invariant

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**lemma**  $s0\text{-anyP-observable}$  [iff]:  $\text{observable } (\text{obs } s0) P$

*<proof>*

**end**

## 7 Non-injective Agreement (L0)

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

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

### 7.1 Signals

signals

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

**fun**  
  *addSignal* :: (*signal*  $\Rightarrow$  *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**

$a0n\text{-init} \equiv \{\langle signals = \lambda s. 0 \rangle\}$

**definition**

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

$a0n \equiv \langle$

$init = a0n\text{-init},$

$trans = a0n\text{-trans},$

$obs = id$

$\rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

### 7.3 Non injective agreement invariant

Invariant: non injective agreement

**definition**

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

**where**

$a0n\text{-agreement} \equiv \{s. \forall A B M.$

$signals s (Commit A B M) > 0 \longrightarrow signals s (Running A B M) > 0$

$\}$

**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**  $a0n\text{-agreement-init [iff]:$

$init a0n \subseteq a0n\text{-agreement}$

$\langle proof \rangle$

**lemma**  $a0n\text{-agreement-trans [iff]:$

$\{a0n\text{-agreement}\} trans a0n \{> a0n\text{-agreement}\}$

$\langle proof \rangle$

**lemma**  $a0n\text{-agreement [iff]: reach } a0n \subseteq a0n\text{-agreement}$

$\langle proof \rangle$

**lemma** *a0n-obs-agreement* [*iff*]:  
    *oreach a0n*  $\subseteq$  *a0n-agreement*  
    ⟨*proof*⟩

**end**

## 8 Injective Agreement (L0)

```
theory AuthenticationI
imports AuthenticationN
begin
```

### 8.1 State and events

```
type-synonym
  a0i-state = a0n-state
```

```
type-synonym
  a0i-obs = a0n-obs
```

```
abbreviation
  a0i-init :: a0n-state set
where
  a0i-init  $\equiv$  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(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  $\equiv$  ( $\lfloor$ 
    init = a0i-init,
    trans = a0i-trans,
    obs = id
   $\rfloor$ )
```

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

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

## 8.2 Injective agreement invariant

**definition**

*a0i-agreement* :: *a0i-state set*

**where**

*a0i-agreement*  $\equiv \{s. \forall A B M.$

$\text{signals } s (\text{Commit } A B M) \leq \text{signals } s (\text{Running } A B M)$

$\}$

**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*

$\langle \text{proof} \rangle$

**lemma** *PO-a0i-agreement-trans* [iff]:

$\{a0i\text{-agreement}\} \text{ trans } a0i \{> a0i\text{-agreement}\}$

$\langle \text{proof} \rangle$

**lemma** *PO-a0i-agreement* [iff]: *reach a0i*  $\subseteq$  *a0i-agreement*

$\langle \text{proof} \rangle$

**lemma** *PO-a0i-obs-agreement* [iff]: *oreach a0i*  $\subseteq$  *a0i-agreement*

$\langle \text{proof} \rangle$

## 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*:

$\{R0n0i\} a0n\text{-running } A B M, a0i\text{-running } A B M \{> R0n0i\}$

$\langle \text{proof} \rangle$

**lemma** *PO-a0i-commit-refines-a0n-commit*:  
 $\{R0n0i\} a0n\text{-commit } A \ B \ M, a0i\text{-commit } A \ B \ M \{> \ R0n0i\}$   
 $\langle \text{proof} \rangle$

**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]:  
 $\text{init } a0i \subseteq R0n0i \text{“}(\text{init } a0n)$   
 $\langle \text{proof} \rangle$

**lemma** *PO-a0i-refines-trans-a0n* [iff]:  
 $\{R0n0i\} \text{trans } a0n, \text{trans } a0i \{> \ R0n0i\}$   
 $\langle \text{proof} \rangle$

**lemma** *PO-obs-consistent* [iff]:  
 $\text{obs-consistent } R0n0i \ \text{med}0n0i \ a0n \ a0i$   
 $\langle \text{proof} \rangle$

**lemma** *PO-a0i-refines-a0n*:  
 $\text{refines } R0n0i \ \text{med}0n0i \ a0n \ a0i$   
 $\langle \text{proof} \rangle$

## 8.4 Derived invariant

**lemma** *agreement-implies-niagreement* [iff]:  $a0i\text{-agreement} \subseteq a0n\text{-agreement}$   
 $\langle \text{proof} \rangle$

**lemma** *PO-a0i-a0n-agreement* [iff]:  $\text{reach } a0i \subseteq a0n\text{-agreement}$   
 $\langle \text{proof} \rangle$

**lemma** *PO-a0i-obs-a0n-agreement* [iff]:  $\text{oreach } a0i \subseteq a0n\text{-agreement}$   
 $\langle \text{proof} \rangle$

**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*  
*<proof>*

**lemma** *in-progress-Some* [*elim*]: *in-progress* *r x*  $\Longrightarrow$   $\exists$  *x*. *r* = *Some x*  
*<proof>*

**lemma** *in-progressS-elt* [*elim*]: *in-progressS* *r S*  $\Longrightarrow$  *x*  $\in$  *S*  $\Longrightarrow$  *in-progress* *r x*  
*<proof>*

**end**

## 10 Channel Messages

```
theory Channels
imports Message-derivation
begin
```

### 10.1 Channel messages

```
datatype chan =
  Chan tag agent agent msg
```

#### abbreviation

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

#### abbreviation

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

#### abbreviation

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

#### abbreviation

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

### 10.2 Extract

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

#### inductive-set

```
extr :: agent set ⇒ msg set ⇒ chan set ⇒ msg set
for bad :: agent set
and IK :: msg set
and H :: chan set
```

#### where

```
extr-Inj:  $M \in IK \implies M \in \text{extr bad IK H}$ 
```

```
| extr-Chan:
```

```
[[ Chan c A B M ∈ H; c = insec ∨ c = auth ∨ A ∈ bad ∨ B ∈ bad ]] ⇒ M ∈ extr bad IK H
```

```
declare extr.intros [intro]
```

```
declare extr.cases [elim]
```

```
lemma extr-empty-chan [simp]:  $\text{extr bad IK \{ \}} = IK$ 
⟨proof⟩
```

```
lemma IK-subset-extr:  $IK \subseteq \text{extr bad IK chan}$ 
```

$\langle \text{proof} \rangle$

**lemma** *extr-mono-chan* [dest]:  $G \subseteq H \implies \text{extr bad IK } G \subseteq \text{extr bad IK } H$

$\langle \text{proof} \rangle$

**lemma** *extr-mono-IK* [dest]:  $IK1 \subseteq IK2 \implies \text{extr bad IK1 } H \subseteq \text{extr bad IK2 } H$

$\langle \text{proof} \rangle$

**lemma** *extr-mono-bad* [dest]:  $\text{bad} \subseteq \text{bad}' \implies \text{extr bad IK } H \subseteq \text{extr bad}' \text{ IK } H$

$\langle \text{proof} \rangle$

**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]:  $\llbracket b \subseteq b'; I \subseteq I'; C \subseteq C' \rrbracket \implies \text{extr } b \text{ I } C \subseteq \text{extr } b' \text{ I}' C'$

$\langle \text{proof} \rangle$

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

**lemma** *extr-insert* [intro]:  $M \in \text{extr bad IK } H \implies M \in \text{extr bad IK } (\text{insert } C \text{ } H)$

$\langle \text{proof} \rangle$

**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  $\text{extr bad IK } H$ )

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**lemma** *extr-insert-IK-eq* [simp]:  $\text{extr bad } (\text{insert } X \text{ } IK) \text{ } CH = \text{insert } X \text{ } (\text{extr bad IK } CH)$

$\langle \text{proof} \rangle$

**lemma** *extr-insert-bad*:

$\text{extr } (\text{insert } A \text{ } \text{bad}) \text{ } IK \text{ } CH \subseteq$

$\text{extr bad IK } CH \cup \{M. \exists B. \text{Confid } A \text{ } B \text{ } M \in CH \vee \text{Confid } B \text{ } A \text{ } M \in CH \vee$

$\text{Secure } A \text{ } B \text{ } M \in CH \vee \text{Secure } B \text{ } A \text{ } M \in CH\}$

$\langle \text{proof} \rangle$

**lemma** *extr-insert-Confid* [simp]:

$A \notin \text{bad} \implies$

$B \notin \text{bad} \implies$

$\text{extr bad IK } (\text{insert } (\text{Confid } A \text{ } B \text{ } X) \text{ } CH) = \text{extr bad IK } CH$

$\langle \text{proof} \rangle$

### 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]:

$\text{bad} \subseteq \text{bad}' \Longrightarrow \text{fake bad IK chan} \subseteq \text{fake bad}' \text{ IK chan}$   
 $\langle \text{proof} \rangle$

**lemma** *fake-mono-ik* [intro]:

$\text{IK} \subseteq \text{IK}' \Longrightarrow \text{fake bad IK chan} \subseteq \text{fake bad IK}' \text{ chan}$   
 $\langle \text{proof} \rangle$

**lemma** *fake-mono-chan* [intro]:

$\text{chan} \subseteq \text{chan}' \Longrightarrow \text{fake bad IK chan} \subseteq \text{fake bad IK chan}'$   
 $\langle \text{proof} \rangle$

**lemma** *fake-mono* [intro]:

$\llbracket \text{bad} \subseteq \text{bad}'; \text{IK} \subseteq \text{IK}'; \text{chan} \subseteq \text{chan}' \rrbracket \Longrightarrow \text{fake bad IK chan} \subseteq \text{fake bad}' \text{ IK}' \text{ chan}'$   
 $\langle \text{proof} \rangle$

**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*:  $\text{chan} \subseteq \text{fake bad IK chan}$

$\langle \text{proof} \rangle$

**lemma** *extr-fake*:

$X \in \text{fake bad IK chan} \Longrightarrow \text{extr bad IK}' \{X\} \subseteq \text{IK} \cup \text{extr bad IK}' \text{ chan}$   
 $\langle \text{proof} \rangle$

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

**lemma** *fake-parts-extr-singleton*:

$X \in \text{fake bad IK chan} \Longrightarrow \text{parts (extr bad IK}' \{X\}) \subseteq \text{parts IK} \cup \text{parts (extr bad IK}' \text{ chan)}$   
 $\langle \text{proof} \rangle$

**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>*

**lemma** *fake-synth-analz-extr*:

**assumes**  $X \in \text{fake bad (synth (analz (extr bad IK CH))) CH}$

**shows**  $\text{synth (analz (extr bad IK (insert X CH)))} = \text{synth (analz (extr bad IK CH))}$

*<proof>*

## 10.4 Closure of Dolev-Yao, extract and fake

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

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

**definition**

$\text{dy-fake-msg} :: \text{agent set} \Rightarrow \text{msg set} \Rightarrow \text{chan set} \Rightarrow \text{msg set}$

**where**

$\text{dy-fake-msg } b \ i \ c = \text{synth (analz (extr } b \ i \ c))$

**lemma** *dy-fake-msg-empty* [simp]:  $\text{dy-fake-msg bad } \{\} \ \{\} = \text{synth } \{\}$

*<proof>*

**lemma** *dy-fake-msg-mono-bad* [dest]:  $\text{bad} \subseteq \text{bad}' \implies \text{dy-fake-msg bad } I \ C \subseteq \text{dy-fake-msg bad}' \ I \ C$

*<proof>*

**lemma** *dy-fake-msg-mono-ik* [dest]:  $G \subseteq H \implies \text{dy-fake-msg bad } G \ C \subseteq \text{dy-fake-msg bad } H \ C$

*<proof>*

**lemma** *dy-fake-msg-mono-chan* [dest]:  $G \subseteq H \implies \text{dy-fake-msg bad } I \ G \subseteq \text{dy-fake-msg bad } I \ H$

*<proof>*

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

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

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

**lemma** *dy-fake-msg-insert* [intro]:

$M \in \text{dy-fake-msg bad } I \ C \implies M \in \text{dy-fake-msg bad } I \ (\text{insert } X \ C)$

*<proof>*

**lemma** *dy-fake-msg-mono* [intro]:

$\llbracket b \subseteq b'; I \subseteq I'; C \subseteq C' \rrbracket \implies \text{dy-fake-msg } b \ I \ C \subseteq \text{dy-fake-msg } b' \ I' \ C'$

*<proof>*

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

**lemma** *dy-fake-msg-insert-chan*:

$x = \text{insec} \vee x = \text{auth} \implies$

$M \in \text{dy-fake-msg bad IK (insert (Chan } x \ A \ B \ M) \ CH)$

$\langle \text{proof} \rangle$

#### 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**

$\text{dy-fake-chan} :: \text{agent set} \Rightarrow \text{msg set} \Rightarrow \text{chan set} \Rightarrow \text{chan set}$

**where**

$\text{dy-fake-chan } b \ i \ c = \text{fake } b \ (\text{dy-fake-msg } b \ i \ c) \ c$

**lemma** *dy-fake-chan-mono-bad* [*intro*]:

$\text{bad} \subseteq \text{bad}' \implies \text{dy-fake-chan bad } I \ C \subseteq \text{dy-fake-chan bad}' \ I \ C$

$\langle \text{proof} \rangle$

**lemma** *dy-fake-chan-mono-ik* [*intro*]:

$T \subseteq T' \implies \text{dy-fake-chan bad } T \ C \subseteq \text{dy-fake-chan bad } T' \ C$

$\langle \text{proof} \rangle$

**lemma** *dy-fake-chan-mono-chan* [*intro*]:

$C \subseteq C' \implies \text{dy-fake-chan bad } T \ C \subseteq \text{dy-fake-chan bad } T \ C'$

$\langle \text{proof} \rangle$

**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**  $\text{dy-fake-chan } b \ I \ C \subseteq \text{dy-fake-chan } b' \ I' \ C'$

$\langle \text{proof} \rangle$

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

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

$\llbracket \text{extr bad IK chan} \subseteq T \rrbracket \implies \text{dy-fake-msg bad IK chan} \subseteq \text{synth (analz } T)$

$\langle \text{proof} \rangle$

**lemma** *dy-fake-chan-mono2*:

$\llbracket \text{extr bad IK chan} \subseteq \text{synth (analz } y); \text{chan} \subseteq \text{fake bad (synth (analz } y)) \ z \rrbracket$

$\implies \text{dy-fake-chan bad IK chan} \subseteq \text{fake bad (synth (analz } y)) \ z$

$\langle \text{proof} \rangle$

**lemma** *extr-subset-dy-fake-msg*:  $\text{extr bad IK chan} \subseteq \text{dy-fake-msg bad IK chan}$

$\langle \text{proof} \rangle$

**lemma** *dy-fake-chan-extr-insert:*

$M \in \text{dy-fake-chan bad IK CH} \implies \text{extr bad IK (insert M CH)} \subseteq \text{dy-fake-msg bad IK CH}$   
*<proof>*

**lemma** *dy-fake-chan-extr-insert-parts:*

$M \in \text{dy-fake-chan bad IK CH} \implies$   
 $\text{parts (extr bad IK (insert M CH))} \subseteq \text{parts (extr bad IK CH)} \cup \text{dy-fake-msg bad IK CH}$   
*<proof>*

**lemma** *dy-fake-msg-extr:*

$\text{extr bad ik chan} \subseteq \text{synth (analz X)} \implies \text{dy-fake-msg bad ik chan} \subseteq \text{synth (analz X)}$   
*<proof>*

**lemma** *extr-insert-dy-fake-msg:*

$M \in \text{dy-fake-msg bad IK CH} \implies \text{extr bad (insert M IK) CH} \subseteq \text{dy-fake-msg bad IK CH}$   
*<proof>*

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

$M \in \text{dy-fake-msg bad IK CH} \implies \text{dy-fake-msg bad (insert M IK) CH} \subseteq \text{dy-fake-msg bad IK CH}$   
*<proof>*

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

$M \in \text{dy-fake-msg bad IK CH} \implies \text{synth (analz (insert M IK))} \subseteq \text{dy-fake-msg bad IK CH}$   
*<proof>*

**lemma** *Fake-insert-dy-fake-msg:*

$M \in \text{dy-fake-msg bad IK CH} \implies$   
 $\text{extr bad IK CH} \subseteq \text{synth (analz X)} \implies$   
 $\text{synth (analz (insert M IK))} \subseteq \text{synth (analz X)}$   
*<proof>*

**lemma** *dy-fake-chan-insert-chan:*

$x = \text{insec} \vee x = \text{auth} \implies$   
 $\text{Chan x A B M} \in \text{dy-fake-chan bad IK (insert (Chan x A B M) CH)}$   
*<proof>*

**lemma** *dy-fake-chan-subset:*

$\text{CH} \subseteq \text{fake bad (dy-fake-msg bad IK CH) CH'} \implies$   
 $\text{dy-fake-chan bad IK CH} \subseteq \text{fake bad (dy-fake-msg bad IK CH) CH'}$   
*<proof>*

**end**

## 11 Payloads and Support for Channel Message Implementations

Definitions and lemmas that do not require the implementations.

```
theory Payloads
imports Message-derivation
begin
```

### 11.1 Payload messages

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

Define set of payloads for basic messages.

```
inductive-set cpayload :: cmsg set where
  cAgent A ∈ cpayload
| cNumber T ∈ cpayload
| cNonce N ∈ cpayload
| cEphK K ∈ cpayload
| X ∈ cpayload ⇒ cHash X ∈ cpayload
| [ X ∈ cpayload; Y ∈ cpayload ] ⇒ cPair X Y ∈ cpayload
| [ X ∈ cpayload; Y ∈ cpayload ] ⇒ cEnc X Y ∈ cpayload
| [ X ∈ cpayload; Y ∈ cpayload ] ⇒ cAenc X Y ∈ cpayload
| [ X ∈ cpayload; Y ∈ cpayload ] ⇒ cSign X Y ∈ cpayload
| [ X ∈ cpayload; Y ∈ cpayload ] ⇒ cExp X Y ∈ cpayload
```

Lift *cpayload* to the quotiented message type.

```
lift-definition payload :: msg set is cpayload <proof>
```

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

```
lemma eq-rep-abs: eq x (Re (Ab x))
<proof>
```

```
lemma eq-cpayload:
  assumes eq x y and x ∈ cpayload
  shows y ∈ cpayload
<proof>
```

```
lemma abs-payload: Ab x ∈ payload ⇔ x ∈ cpayload
<proof>
```

```
lemma abs-cpayload-rep: x ∈ Ab' cpayload ⇔ Re x ∈ cpayload
<proof>
```

```
lemma payload-rep-cpayload: Re x ∈ cpayload ⇔ x ∈ payload
<proof>
```

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

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

$\langle proof \rangle$   
**lemma** *payload-NonceI*:  $Nonce\ N \in payload$   
 $\langle proof \rangle$   
**lemma** *payload-NumberI*:  $Number\ N \in payload$   
 $\langle proof \rangle$   
**lemma** *payload-EphKI*:  $EphK\ X \in payload$   
 $\langle proof \rangle$   
**lemma** *payload-HashI*:  $x \in payload \implies Hash\ x \in payload$   
 $\langle proof \rangle$   
**lemma** *payload-PairI*:  $x \in payload \implies y \in payload \implies Pair\ x\ y \in payload$   
 $\langle proof \rangle$   
**lemma** *payload-EncI*:  $x \in payload \implies y \in payload \implies Enc\ x\ y \in payload$   
 $\langle proof \rangle$   
**lemma** *payload-AencI*:  $x \in payload \implies y \in payload \implies Aenc\ x\ y \in payload$   
 $\langle proof \rangle$   
**lemma** *payload-SignI*:  $x \in payload \implies y \in payload \implies Sign\ x\ y \in payload$   
 $\langle proof \rangle$   
**lemma** *payload-ExpI*:  $x \in payload \implies y \in payload \implies Exp\ x\ y \in payload$   
 $\langle proof \rangle$   
**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*:  $Tag\ X \in payload \implies P$   
 $\langle proof \rangle$   
**lemma** *payload-LtK*:  $LtK\ X \in payload \implies P$   
 $\langle proof \rangle$   
**lemma** *payload-Hash*:  $Hash\ X \in payload \implies (X \in payload \implies P) \implies P$   
 $\langle proof \rangle$   
**lemma** *payload-Pair*:  $Pair\ X\ Y \in payload \implies (X \in payload \implies Y \in payload \implies P) \implies P$   
 $\langle proof \rangle$   
**lemma** *payload-Enc*:  $Enc\ X\ Y \in payload \implies (X \in payload \implies Y \in payload \implies P) \implies P$   
 $\langle proof \rangle$   
**lemma** *payload-Aenc*:  $Aenc\ X\ Y \in payload \implies (X \in payload \implies Y \in payload \implies P) \implies P$   
 $\langle proof \rangle$   
**lemma** *payload-Sign*:  $Sign\ X\ Y \in payload \implies (X \in payload \implies Y \in payload \implies P) \implies P$   
 $\langle proof \rangle$   
**lemma** *payload-Exp*:  $Exp\ X\ Y \in payload \implies (X \in payload \implies Y \in payload \implies P) \implies P$   
 $\langle proof \rangle$   
**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*:  
 $(\bigwedge x. eq\ y\ (cAgent\ x) \implies P) \implies$

$$\begin{aligned}
& (\bigwedge x. \text{eq } y \text{ (cNumber } x) \implies P) \implies \\
& (\bigwedge x. \text{eq } y \text{ (cNonce } x) \implies P) \implies \\
& (\bigwedge x. \text{eq } y \text{ (cLtK } x) \implies P) \implies \\
& (\bigwedge x. \text{eq } y \text{ (cEphK } x) \implies P) \implies \\
& (\bigwedge x x'. \text{eq } y \text{ (cPair } x x') \implies P) \implies \\
& (\bigwedge x x'. \text{eq } y \text{ (cEnc } x x') \implies P) \implies \\
& (\bigwedge x x'. \text{eq } y \text{ (cAenc } x x') \implies P) \implies \\
& (\bigwedge x x'. \text{eq } y \text{ (cSign } x x') \implies P) \implies \\
& (\bigwedge x. \text{eq } y \text{ (cHash } x) \implies P) \implies \\
& (\bigwedge x. \text{eq } y \text{ (cTag } x) \implies P) \implies \\
& (\bigwedge x x'. \text{eq } y \text{ (cExp } x x') \implies P) \implies \\
& P \\
& \langle \text{proof} \rangle
\end{aligned}$$

**lemma** *msg-exhaust*:

$$\begin{aligned}
& (\bigwedge x. y = \text{Agent } x \implies P) \implies \\
& (\bigwedge x. y = \text{Number } x \implies P) \implies \\
& (\bigwedge x. y = \text{Nonce } x \implies P) \implies \\
& (\bigwedge x. y = \text{LtK } x \implies P) \implies \\
& (\bigwedge x. y = \text{EphK } x \implies P) \implies \\
& (\bigwedge x x'. y = \text{Pair } x x' \implies P) \implies \\
& (\bigwedge x x'. y = \text{Enc } x x' \implies P) \implies \\
& (\bigwedge x x'. y = \text{Aenc } x x' \implies P) \implies \\
& (\bigwedge x x'. y = \text{Sign } x x' \implies P) \implies \\
& (\bigwedge x. y = \text{Hash } x \implies P) \implies \\
& (\bigwedge x. y = \text{Tag } x \implies P) \implies \\
& (\bigwedge x x'. y = \text{Exp } x x' \implies P) \implies \\
& P \\
& \langle \text{proof} \rangle
\end{aligned}$$

**lemma** *payload-cases*:

$$\begin{aligned}
& a \in \text{payload} \implies \\
& (\bigwedge A. a = \text{Agent } A \implies P) \implies \\
& (\bigwedge T. a = \text{Number } T \implies P) \implies \\
& (\bigwedge N. a = \text{Nonce } N \implies P) \implies \\
& (\bigwedge K. a = \text{EphK } K \implies P) \implies \\
& (\bigwedge X. a = \text{Hash } X \implies X \in \text{payload} \implies P) \implies \\
& (\bigwedge X Y. a = \text{Pair } X Y \implies X \in \text{payload} \implies Y \in \text{payload} \implies P) \implies \\
& (\bigwedge X Y. a = \text{Enc } X Y \implies X \in \text{payload} \implies Y \in \text{payload} \implies P) \implies \\
& (\bigwedge X Y. a = \text{Aenc } X Y \implies X \in \text{payload} \implies Y \in \text{payload} \implies P) \implies \\
& (\bigwedge X Y. a = \text{Sign } X Y \implies X \in \text{payload} \implies Y \in \text{payload} \implies P) \implies \\
& (\bigwedge X Y. a = \text{Exp } X Y \implies X \in \text{payload} \implies Y \in \text{payload} \implies P) \implies \\
& P \\
& \langle \text{proof} \rangle
\end{aligned}$$

**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]:

$$\llbracket X \in \text{parts } S; S \subseteq \text{payload} \rrbracket \implies X \in \text{payload}$$

$\langle \text{proof} \rangle$

**lemma** *payload-parts-singleton* [*simp*, *dest*]:  
 $\llbracket X \in \text{parts } \{Y\}; Y \in \text{payload} \rrbracket \implies X \in \text{payload}$   
 $\langle \text{proof} \rangle$

**lemma** *payload-analz* [*simp*, *dest*]:  
 $\llbracket X \in \text{analz } S; S \subseteq \text{payload} \rrbracket \implies X \in \text{payload}$   
 $\langle \text{proof} \rangle$

**lemma** *payload-synth-analz*:  
 $\llbracket X \in \text{synth } (\text{analz } S); S \subseteq \text{payload} \rrbracket \implies X \in \text{payload}$   
 $\langle \text{proof} \rangle$

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

**lemma** *synth-payload*:  
 $Y \cap \text{payload} = \{\} \implies \text{synth } (X \cup Y) \subseteq \text{synth } X \cup \text{-payload}$   
 $\langle \text{proof} \rangle$

**lemma** *synth-payload2*:  
 $Y \cap \text{payload} = \{\} \implies \text{synth } (Y \cup X) \subseteq \text{synth } X \cup \text{-payload}$   
 $\langle \text{proof} \rangle$

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

**lemma** *synth-idem-payload*:  
 $X \subseteq \text{synth } Y \cup \text{-payload} \implies \text{synth } X \subseteq \text{synth } Y \cup \text{-payload}$   
 $\langle \text{proof} \rangle$

## 11.2 *isLtKey*: is a long term key

**lemma** *LtKeys-payload* [*dest*]:  $NI \subseteq \text{payload} \implies NI \cap \text{range } \text{LtK} = \{\}$   
 $\langle \text{proof} \rangle$

**lemma** *LtKeys-parts-payload* [*dest*]:  $NI \subseteq \text{payload} \implies \text{parts } NI \cap \text{range } \text{LtK} = \{\}$   
 $\langle \text{proof} \rangle$

**lemma** *LtKeys-parts-payload-singleton* [*elim*]:  $X \in \text{payload} \implies \text{LtK } Y \in \text{parts } \{X\} \implies \text{False}$   
 $\langle \text{proof} \rangle$

**lemma** *parts-of-LtKeys* [*simp*]:  $K \subseteq \text{range } \text{LtK} \implies \text{parts } K = K$   
 $\langle \text{proof} \rangle$

## 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!*]: *keys-of* *A*  $\subseteq$  *range* *LtK*  
 $\langle \text{proof} \rangle$



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

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

**lemma** *shrK-keys-of-2* [*intro!*]:  
 $shrK B A \in keys-of A$   
 ⟨*proof*⟩

**lemma** *priK-keys-of-eq* [*dest*]:  
 $priK B \in keys-of A \implies A = B$   
 ⟨*proof*⟩

**lemma** *shrK-keys-of-eq* [*dest*]:  
 $shrK A B \in keys-of C \implies A = C \vee B = C$   
 ⟨*proof*⟩

**lemma** *def-keys-of* [*dest*]:  
 $K \in keys-of A \implies K = priK A \vee (\exists B. K = shrK A B \vee K = shrK B A)$   
 ⟨*proof*⟩

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

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

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

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

##### **definition**

$Keys-bad :: msg\ set \Rightarrow agent\ set \Rightarrow bool$

##### **where**

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

— basic lemmas

##### **lemma** *Keys-badI*:

$\llbracket IK \cap range\ LtK \subseteq range\ pubK \cup priK\ 'Bad \cup \{shrK\ A\ B \mid A\ B. A \in Bad \vee B \in Bad\};$   
 $range\ pubK \subseteq IK \rrbracket$   
 $\implies Keys-bad\ IK\ Bad$   
 ⟨*proof*⟩

##### **lemma** *Keys-badE* [*elim*]:

$\llbracket Keys-bad\ IK\ Bad;$   
 $\llbracket range\ pubK \subseteq IK;$

$IK \cap \text{range LtK} \subseteq \text{range pubK} \cup \bigcup (\text{keys-of } \text{' Bad})$   
 $\implies P$  ]  
 $\implies P$   
 <proof>

**lemma** *Keys-bad-Ltk [simp]*:  
 $\text{Keys-bad } (IK \cap \text{range LtK}) \text{ Bad} \longleftrightarrow \text{Keys-bad } IK \text{ Bad}$   
 <proof>

**lemma** *Keys-bad-priK-D*:  $\llbracket \text{priK } A \in IK; \text{Keys-bad } IK \text{ Bad} \rrbracket \implies A \in \text{Bad}$   
 <proof>

**lemma** *Keys-bad-shrK-D*:  $\llbracket \text{shrK } A \ B \in IK; \text{Keys-bad } IK \text{ Bad} \rrbracket \implies A \in \text{Bad} \vee B \in \text{Bad}$   
 <proof>

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

interaction with *insert*.

**lemma** *Keys-bad-insert-non-LtK*:  
 $X \notin \text{range LtK} \implies \text{Keys-bad } (\text{insert } X \ IK) \text{ Bad} \longleftrightarrow \text{Keys-bad } IK \text{ Bad}$   
 <proof>

**lemma** *Keys-bad-insert-pubK*:  
 $\llbracket \text{Keys-bad } IK \text{ Bad} \rrbracket \implies \text{Keys-bad } (\text{insert } (\text{pubK } A) \ IK) \text{ Bad}$   
 <proof>

**lemma** *Keys-bad-insert-priK-bad*:  
 $\llbracket \text{Keys-bad } IK \text{ Bad}; A \in \text{Bad} \rrbracket \implies \text{Keys-bad } (\text{insert } (\text{priK } A) \ IK) \text{ Bad}$   
 <proof>

**lemma** *Keys-bad-insert-shrK-bad*:  
 $\llbracket \text{Keys-bad } IK \text{ Bad}; A \in \text{Bad} \vee B \in \text{Bad} \rrbracket \implies \text{Keys-bad } (\text{insert } (\text{shrK } A \ B) \ IK) \text{ Bad}$   
 <proof>

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

**lemma** *Keys-bad-insert-Fake*:  
**assumes** *Keys-bad IK Bad*  
**and** *parts IK  $\cap$  range LtK  $\subseteq$  IK*  
**and** *X  $\in$  synth (analz IK)*  
**shows** *Keys-bad (insert X IK) Bad*  
 <proof>

**lemma** *Keys-bad-insert-keys-of*:  
 $\text{Keys-bad } Ik \text{ Bad} \implies$   
 $\text{Keys-bad } (\text{keys-of } A \cup Ik) (\text{insert } A \ \text{Bad})$   
 <proof>

**lemma** *Keys-bad-insert-payload*:

$Keys\text{-}bad\ Ik\ Bad \implies$   
 $x \in payload \implies$   
 $Keys\text{-}bad\ (insert\ x\ Ik)\ Bad$

$\langle proof \rangle$

### 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**

$broken\ K \equiv \{(A,B) \mid A\ B.\ priK\ A \in K \vee priK\ B \in K \vee shrK\ A\ B \in K \vee shrK\ B\ A \in K\}$

**lemma** *brokenD* [*dest!*]:

$(A, B) \in broken\ K \implies priK\ A \in K \vee priK\ B \in K \vee shrK\ A\ B \in K \vee shrK\ B\ A \in K$

$\langle proof \rangle$

**lemma** *brokenI* [*intro!*]:

$priK\ A \in K \vee priK\ B \in K \vee shrK\ A\ B \in K \vee shrK\ B\ A \in K \implies (A, B) \in broken\ K$

$\langle proof \rangle$

### 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**

$Enc\text{-}keys\text{-}clean :: msg\ set \implies bool$

**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*:

$\forall X\ Y.\ Enc\ X\ Y \in parts\ S \longrightarrow Y \in range\ LtK \cup payload \implies Enc\text{-}keys\text{-}clean\ S$

$\langle proof \rangle$

**lemma** *Enc-keys-clean-mono*:

$Enc\text{-}keys\text{-}clean\ H \implies G \subseteq H \implies Enc\text{-}keys\text{-}clean\ G$  — anti-tone

$\langle proof \rangle$

**lemma** *Enc-keys-clean-Un* [*simp*]:

$Enc\text{-}keys\text{-}clean\ (G \cup H) \longleftrightarrow Enc\text{-}keys\text{-}clean\ G \wedge Enc\text{-}keys\text{-}clean\ H$

$\langle proof \rangle$

**lemma** *Enc-keys-clean-analz*:

$Enc\ X\ K \in analz\ S \implies Enc\text{-}keys\text{-}clean\ S \implies K \in range\ LtK \cup payload$

$\langle proof \rangle$

**lemma** *Enc-keys-clean-Tags* [*simp,intro*]:  $Enc\text{-}keys\text{-}clean\ Tags$

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

## 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**  $parts-PairSet$ :  
 $parts\ (PairSet\ G\ H) \subseteq PairSet\ G\ H \cup parts\ G \cup parts\ H$   
 $\langle proof \rangle$

**lemma**  $parts-EncSet$ :  
 $parts\ (EncSet\ G\ K) \subseteq EncSet\ G\ K \cup PairSet\ (range\ Agent)\ G \cup range\ Agent \cup parts\ G$   
 $\langle proof \rangle$

**lemma**  $parts-AencSet$ :  
 $parts\ (AencSet\ G\ K) \subseteq AencSet\ G\ K \cup PairSet\ (range\ Agent)\ G \cup range\ Agent \cup parts\ G$   
 $\langle proof \rangle$

**lemma**  $parts-SignSet$ :  
 $parts\ (SignSet\ G\ K) \subseteq SignSet\ G\ K \cup PairSet\ (range\ Agent)\ G \cup range\ Agent \cup parts\ G$   
 $\langle proof \rangle$

**lemma**  $parts-HashSet$ :  
 $parts\ (HashSet\ G) \subseteq HashSet\ G$   
 $\langle proof \rangle$

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

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

**lemma**  $Enc$ -keys-clean- $PairSet$ - $Agent$ - $Un$ :  
 $Enc$ -keys-clean  $(G \cup H) \implies Enc$ -keys-clean  $(PairSet\ (Agent'X)\ G \cup H)$   
 $\langle proof \rangle$

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

$Enc\text{-}keys\text{-}clean (G \cup H) \implies Enc\text{-}keys\text{-}clean (PairSet\ Tags\ G \cup H)$   
 ⟨proof⟩

**lemma** *Enc-keys-clean-AencSet-Un:*

$Enc\text{-}keys\text{-}clean (G \cup H) \implies Enc\text{-}keys\text{-}clean (AencSet\ G\ K \cup H)$   
 ⟨proof⟩

**lemma** *Enc-keys-clean-EncSet-Un:*

$K \subseteq range\ LtK \implies Enc\text{-}keys\text{-}clean (G \cup H) \implies Enc\text{-}keys\text{-}clean (EncSet\ G\ K \cup H)$   
 ⟨proof⟩

**lemma** *Enc-keys-clean-SignSet-Un:*

$Enc\text{-}keys\text{-}clean (G \cup H) \implies Enc\text{-}keys\text{-}clean (SignSet\ G\ K \cup H)$   
 ⟨proof⟩

**lemma** *Enc-keys-clean-HashSet-Un:*

$Enc\text{-}keys\text{-}clean (G \cup H) \implies Enc\text{-}keys\text{-}clean (HashSet\ G \cup H)$   
 ⟨proof⟩

**lemmas** *Enc-keys-clean-msgSet-Un =*

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

### 11.7.1 Lemmas for moving message sets out of *analz*

Pull *EncSet* out of *analz*.

**lemma** *analz-Un-EncSet:*

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

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

**lemma** *analz-Un-EncSet2:*

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

Pull *AencSet* out of the *analz*.

**lemma** *analz-Un-AencSet:*

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

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

**lemma** *analz-Un-AencSet2:*

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

Pull *PairSet* out of *analz*.

**lemma** *analz-Un-PairSet*:

$\text{analz } (\text{PairSet } G \ G' \cup H) \subseteq \text{PairSet } G \ G' \cup \text{analz } (G \cup G' \cup H)$   
*<proof>*

**lemma** *analz-Un-SignSet*:

**assumes**  $K \subseteq \text{range } \text{LtK}$  **and**  $\text{Enc-keys-clean } (G \cup H)$

**shows**  $\text{analz } (\text{SignSet } G \ K \cup H) \subseteq \text{SignSet } G \ K \cup \text{analz } (G \cup H)$   
*<proof>*

Pull *Tags* out of *analz*.

**lemma** *analz-Un-Tag*:

**assumes**  $\text{Enc-keys-clean } H$

**shows**  $\text{analz } (\text{Tags } \cup H) \subseteq \text{Tags } \cup \text{analz } H$   
*<proof>*

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

**lemma** *analz-Un-AgentSet*:

**shows**  $\text{analz } (\text{AgentSet } \cup H) \subseteq \text{AgentSet } \cup \text{analz } H$   
*<proof>*

Pull *HashSet* out of *analz*.

**lemma** *analz-Un-HashSet*:

**assumes**  $\text{Enc-keys-clean } H$  **and**  $G \subseteq - \text{payload}$

**shows**  $\text{analz } (\text{HashSet } G \cup H) \subseteq \text{HashSet } G \cup \text{analz } H$   
*<proof>*

**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
<proof>
```

#### 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
<proof>
```

#### 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**  $P$   
 $\langle \text{proof} \rangle$   
**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  $\text{Enc } X Y$  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  $\text{impl*Set}$  out of the  $\text{analz}$  (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*:  $\text{composed } (\text{implem } Z)$

**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*:

$\llbracket G \subseteq \text{payload}; \text{Enc-keys-clean } H; Ag \cap \text{broken } (\text{parts } H \cap \text{range } \text{LtK}) = \{\} \rrbracket$   
 $\implies \text{analz } (\text{implConfidSet } Ag G \cup H) \subseteq \text{synth } (\text{analz } H) \cup \text{-payload}$

**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*:  $X = Y \implies \text{composed } X \longleftrightarrow \text{composed } Y$   
 $\langle \text{proof} \rangle$

**lemma** *implem-Tags-aux*:  $\text{implem } (\text{Chan } x \ A \ B \ M) \notin \text{Tags} \langle \text{proof} \rangle$

**lemma** *implem-Tags* [*simp*]:  $\text{implem } x \notin \text{Tags} \langle \text{proof} \rangle$

**lemma** *implem-LtK-aux*:  $\text{implem } (\text{Chan } x \ A \ B \ M) \neq LtK \ K \langle \text{proof} \rangle$

**lemma** *implem-LtK* [*simp*]:  $\text{implem } x \neq LtK \ K \langle \text{proof} \rangle$

**lemma** *implem-LtK2* [*simp*]:  $\text{implem } x \notin \text{range } LtK \langle \text{proof} \rangle$

**declare** *impl-Impl* [*simp*]

**lemma** *LtK-parts-impl-insert*:

$LtK \ K \in \text{parts } (\text{insert } (\text{implem } (\text{Chan } x \ A \ B \ M)) \ S) \implies M \in \text{payload} \implies LtK \ K \in \text{parts } S$   
 $\langle \text{proof} \rangle$

**declare** *LtK-parts-impl-insert* [*dest*]

**declare** *Enc-keys-clean-valid* [*simp, intro*]

**lemma** *valid-composed* [*simp, dest*]:  $M \in \text{valid} \implies \text{composed } M$   
 $\langle \text{proof} \rangle$

**lemma** *valid-payload* [*dest*]:  $\llbracket X \in \text{valid}; X \in \text{payload} \rrbracket \implies P$   
 $\langle \text{proof} \rangle$

**lemma** *valid-isLtKey* [*dest*]:  $\llbracket X \in \text{valid}; X \in \text{range } LtK \rrbracket \implies P$   
 $\langle \text{proof} \rangle$

**lemma** *analz-valid*:

$H \subseteq \text{payload} \cup \text{valid} \cup \text{range } LtK \cup \text{Tags} \implies$   
 $\text{implem } (\text{Chan } x \ A \ B \ M) \in \text{analz } H \implies$   
 $\text{implem } (\text{Chan } x \ A \ B \ M) \in H$   
 $\langle \text{proof} \rangle$

**lemma** *parts-valid-LtKeys-disjoint*:

$I \subseteq \text{valid} \implies \text{parts } I \cap \text{range } LtK = \{\}$   
 $\langle \text{proof} \rangle$

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

**end**

### 12.3 Third step: *valid-implem*

This extends *semivalid-implem* with four new assumptions, which under certain conditions give information on  $A, B, M$  when  $\text{implXXX } A \ B \ M \in \text{synth } (\text{analz } Z)$ . 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*]:

$\llbracket X \in abs\ H;$

$\bigwedge x\ A\ B\ M. X = Chan\ x\ A\ B\ M \Longrightarrow M \in payload \Longrightarrow implem\ X \in H \Longrightarrow P \rrbracket$

$\Longrightarrow P$

*<proof>*

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

*<proof>*

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

*<proof>*

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

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

*<proof>*

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

*<proof>*

General lemmas about implementations and *local.abs*.

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

*<proof>*

**lemma** *abs-insert-impl* [*simp*]:

$M \in payload \Longrightarrow abs\ (insert\ (implem\ (Chan\ x\ A\ B\ M))\ S) = insert\ (Chan\ x\ A\ B\ M)\ (abs\ S)$

*<proof>*

**lemma** *extr-payload* [*simp*, *intro*]:

$\llbracket X \in extr\ Bad\ NI\ (abs\ I); NI \subseteq payload \rrbracket \Longrightarrow X \in payload$

*<proof>*

**lemma** *abs-Un-LtK*:

$K \subseteq \text{range LtK} \implies \text{abs } (K \cup S) = \text{abs } S$   
 ⟨proof⟩

**lemma** *abs-Un-keys-of* [simp]:

$\text{abs } (\text{keys-of } A \cup S) = \text{abs } S$   
 ⟨proof⟩

Lemmas about *valid* and *local.abs*

**lemma** *abs-validSet*:  $\text{abs } (S \cap \text{valid}) = \text{abs } S$

⟨proof⟩

**lemma** *valid-abs*:  $M \in \text{valid} \implies \exists M'. M' \in \text{abs } \{M\}$

⟨proof⟩

## 13.2 Extractable messages

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

**definition**

$\text{extractable} :: \text{msg set} \Rightarrow \text{msg set} \Rightarrow \text{msg set}$

**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*:  $\text{extractable } K I \subseteq I$

⟨proof⟩

**lemma** *extractableI*:

$\text{implem } (\text{Chan } x A B M) \in I \implies$   
 $x = \text{insec} \vee x = \text{auth} \vee ((x = \text{confid} \vee x = \text{secure}) \wedge (A, B) \in \text{broken } K) \implies$   
 $\text{implem } (\text{Chan } x A B M) \in \text{extractable } K I$

⟨proof⟩

**lemma** *extractableE*:

$X \in \text{extractable } K I \implies$   
 $(\bigwedge A B M. X = \text{implInsec } A B M \implies X \in I \implies P) \implies$   
 $(\bigwedge A B M. X = \text{implAuth } A B M \implies X \in I \implies P) \implies$   
 $(\bigwedge A B M. X = \text{implConfid } A B M \implies X \in I \implies (A, B) \in \text{broken } K \implies P) \implies$   
 $(\bigwedge A B M. X = \text{implSecure } A B M \implies X \in I \implies (A, B) \in \text{broken } K \implies P) \implies$   
 $P$

⟨proof⟩

General lemmas about implementations and extractable.

**lemma** *implem-extractable* [simp]:

$\llbracket \text{Keys-bad } K \text{ Bad}; \text{implem } (\text{Chan } x A B M) \in \text{extractable } K I; M \in \text{payload} \rrbracket$   
 $\implies M \in \text{extr Bad NI } (\text{abs } I)$

$\langle \text{proof} \rangle$

Auxiliary lemmas about extractable messages: they are implementations.

**lemma** *valid-extractable* [simp]:  $I \subseteq \text{valid} \implies \text{extractable } K \ I \subseteq \text{valid}$

$\langle \text{proof} \rangle$

**lemma** *LtKeys-parts-extractable*:

$I \subseteq \text{valid} \implies \text{parts } (\text{extractable } K \ I) \cap \text{range } \text{LtK} = \{\}$

$\langle \text{proof} \rangle$

**lemma** *LtKeys-parts-extractable-elt* [simp]:

$I \subseteq \text{valid} \implies \text{LtK } \text{ltk} \notin \text{parts } (\text{extractable } K \ I)$

$\langle \text{proof} \rangle$

**lemma** *LtKeys-parts-implSecureSet*:

$\text{parts } (\text{implSecureSet } \text{Ag } \text{payload}) \cap \text{range } \text{LtK} = \{\}$

$\langle \text{proof} \rangle$

**lemma** *LtKeys-parts-implSecureSet-elt*:

$\text{LtK } K \notin \text{parts } (\text{implSecureSet } \text{Ag } \text{payload})$

$\langle \text{proof} \rangle$

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

### 13.2.1 Partition $I$ to keep only the extractable messages

Partition the implementation set.

**lemma** *impl-partition*:

$\llbracket NI \subseteq \text{payload}; I \subseteq \text{valid} \rrbracket \implies$

$I \subseteq \text{extractable } K \ I \cup$

$\text{implConfidSet } (\text{UNIV} - \text{broken } K) \ \text{payload} \cup$

$\text{implSecureSet } (\text{UNIV} - \text{broken } K) \ \text{payload}$

$\langle \text{proof} \rangle$

### 13.2.2 Partition of *extractable*

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

**lemma** *extractable-partition*:

$\llbracket \text{Keys-bad } K \ \text{Bad}; NI \subseteq \text{payload}; I \subseteq \text{valid} \rrbracket \implies$

$\text{extractable } K \ I \subseteq$

$\text{implInsecSet } (\text{extr } \text{Bad } NI \ (\text{abs } I)) \cup$

$\text{implConfidSet } \text{UNIV } (\text{extr } \text{Bad } NI \ (\text{abs } I)) \cup$

$\text{implAuthSet } (\text{extr } \text{Bad } NI \ (\text{abs } I)) \cup$

$\text{implSecureSet } \text{UNIV } (\text{extr } \text{Bad } NI \ (\text{abs } I))$

$\langle \text{proof} \rangle$

## 13.3 Lemmas for proving intruder refinement (L2-L3)

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

$\text{synth } (\text{analz } (NI \cup I \cup K \cup \text{Tags}))$   
 $\subseteq \text{synth } (\text{analz } (\text{extr Bad NI } (\text{local.abs } I))) \cup \text{--payload}$

### 13.3.1 First: we only keep the extractable messages

**lemma** *analz-NI-I-K-analz-NI-EI*:  
**assumes**  $HNI: NI \subseteq \text{payload}$   
**and**  $HK: K \subseteq \text{range LtK}$   
**and**  $HI: I \subseteq \text{valid}$   
**shows**  $\text{analz } (NI \cup I \cup K \cup \text{Tags}) \subseteq$   
 $\text{synth } (\text{analz } (NI \cup \text{extractable } K \ I \cup K \cup \text{Tags})) \cup \text{--payload}$   
 $\langle \text{proof} \rangle$

### 13.3.2 Only keep the extracted messages (instead of extractable)

**lemma** *analz-NI-EI-K-synth-analz-NI-E-K*:  
**assumes**  $HNI: NI \subseteq \text{payload}$   
**and**  $HK: K \subseteq \text{range LtK}$   
**and**  $HI: I \subseteq \text{valid}$   
**and**  $Hbad: \text{Keys-bad } K \ \text{Bad}$   
**shows**  $\text{analz } (NI \cup \text{extractable } K \ I \cup K \cup \text{Tags})$   
 $\subseteq \text{synth } (\text{analz } (\text{extr Bad NI } (\text{abs } I) \cup K \cup \text{Tags})) \cup \text{--payload}$   
 $\langle \text{proof} \rangle$

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

**lemma** *analz-LtKeys-Tag*:  
**assumes**  $NI \subseteq \text{payload}$  **and**  $K \subseteq \text{range LtK}$   
**shows**  $\text{analz } (NI \cup K \cup \text{Tags}) \subseteq \text{analz } NI \cup K \cup \text{Tags}$   
 $\langle \text{proof} \rangle$

**lemma** *analz-NI-E-K-analz-NI-E*:  
 $\llbracket NI \subseteq \text{payload}; K \subseteq \text{range LtK}; I \subseteq \text{valid} \rrbracket$   
 $\implies \text{analz } (\text{extr Bad NI } (\text{abs } I) \cup K \cup \text{Tags}) \subseteq \text{analz } (\text{extr Bad NI } (\text{abs } I)) \cup K \cup \text{Tags}$   
 $\langle \text{proof} \rangle$

### 13.3.4 Final lemmas, using all the previous ones

**lemma** *analz-NI-I-K-synth-analz-NI-E*:  
**assumes**  
 $Hbad: \text{Keys-bad } K \ \text{Bad}$  **and**  
 $HNI: NI \subseteq \text{payload}$  **and**  
 $HK: K \subseteq \text{range LtK}$  **and**  
 $HI: I \subseteq \text{valid}$   
**shows**  
 $\text{analz } (NI \cup I \cup K \cup \text{Tags}) \subseteq \text{synth } (\text{analz } (\text{extr Bad NI } (\text{abs } I))) \cup \text{--payload}$   
 $\langle \text{proof} \rangle$

Lemma actually used to prove the refinement.

**lemma** *synth-analz-NI-I-K-synth-analz-NI-E*:  
 $\llbracket \text{Keys-bad } K \ \text{Bad}; NI \subseteq \text{payload}; K \subseteq \text{range LtK}; I \subseteq \text{valid} \rrbracket$   
 $\implies \text{synth } (\text{analz } (NI \cup I \cup K \cup \text{Tags}))$

$\subseteq \text{synth } (\text{analz } (\text{extr Bad NI } (\text{abs I}))) \cup \text{--payload}$   
 $\langle \text{proof} \rangle$

### 13.3.5 Partitioning $\text{analz ik}$

Two lemmas useful for proving the invariant

$\text{analz ik} \subseteq \text{synth } (\text{analz } (\text{ik} \cap \text{payload} \cup \text{ik} \cap \text{valid} \cup \text{ik} \cap \text{range LtK} \cup \text{Tags}))$

**lemma** *analz-Un-partition:*

$\text{analz } S \subseteq \text{synth } (\text{analz } ((S \cap \text{payload}) \cup (S \cap \text{valid}) \cup (S \cap \text{range LtK}) \cup \text{Tags})) \implies$

$H \subseteq \text{payload} \cup \text{valid} \cup \text{range LtK} \implies$

$\text{analz } (H \cup S) \subseteq$

$\text{synth } (\text{analz } (((H \cup S) \cap \text{payload}) \cup ((H \cup S) \cap \text{valid}) \cup ((H \cup S) \cap \text{range LtK}) \cup \text{Tags}))$

$\langle \text{proof} \rangle$

**lemma** *analz-insert-partition:*

$\text{analz } S \subseteq \text{synth } (\text{analz } ((S \cap \text{payload}) \cup (S \cap \text{valid}) \cup (S \cap \text{range LtK}) \cup \text{Tags})) \implies$

$x \in \text{payload} \cup \text{valid} \cup \text{range LtK} \implies$

$\text{analz } (\text{insert } x S) \subseteq$

$\text{synth } (\text{analz } (((\text{insert } x S) \cap \text{payload}) \cup ((\text{insert } x S) \cap \text{valid}) \cup$   
 $((\text{insert } x S) \cap \text{range LtK}) \cup \text{Tags}))$

$\langle \text{proof} \rangle$

**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*  
 $\langle$ *proof* $\rangle$

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  
 $\langle$ *proof* $\rangle$

**lemma** *implAuthSet-implAuthSet-aux*: *sym.implAuthSet* G  $\subseteq$  *implAuthSet-aux* G  
 $\langle$ *proof* $\rangle$

**lemma** *implConfidSet-implConfidSet-aux*: *sym.implConfidSet* Ag G  $\subseteq$  *implConfidSet-aux* Ag G  
 $\langle$ *proof* $\rangle$

**lemma** *implSecureSet-implSecureSet-aux*: *sym.implSecureSet* Ag G  $\subseteq$  *implSecureSet-aux* Ag G  
 $\langle$ *proof* $\rangle$

**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>*

**lemma** *analz-Un-implInsecSet-aux-2*:

$$\begin{aligned} & \text{Enc-keys-clean } (G \cup H) \implies \\ & \text{analz } (\text{implInsecSet-aux } G \cup H) \subseteq \\ & \quad \text{implInsecSet-aux } G \cup \text{Tags} \cup \\ & \quad \text{synth } (\text{analz } (G \cup H)) \end{aligned}$$

*<proof>*

**lemma** *analz-Un-implInsecSet-aux-3*:

$$\begin{aligned} & \text{Enc-keys-clean } (G \cup H) \implies \\ & \text{analz } (\text{implInsecSet-aux } G \cup H) \subseteq \text{synth } (\text{analz } (G \cup H)) \cup \text{--payload} \end{aligned}$$

*<proof>*

**lemma** *analz-Un-implInsecSet*:

$$\begin{aligned} & \text{Enc-keys-clean } (G \cup H) \implies \\ & \text{analz } (\text{sym.implInsecSet } G \cup H) \subseteq \text{synth } (\text{analz } (G \cup H)) \cup \text{--payload} \end{aligned}$$

*<proof>*

### 14.3 Pull *implConfidSet* out of *analz*

**lemma** *analz-Un-implConfidSet-aux-1*:

$$\begin{aligned} & \text{Enc-keys-clean } (G \cup H) \implies \\ & \text{analz } (\text{implConfidSet-aux } Ag G \cup H) \subseteq \\ & \quad \text{implConfidSet-aux } Ag G \cup \text{PairSet } \text{Tags } G \cup \text{Tags} \cup \end{aligned}$$

$analz (G \cup H)$   
 ⟨proof⟩

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

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

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

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

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

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

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

#### 14.4 Pull *implSecureSet* out of *analz*

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

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

$\langle \text{proof} \rangle$

**lemma** *analz-Un-implSecureSet-aux-3:*

$\text{Enc-keys-clean } (G \cup H) \implies$

$\text{analz } (\text{implSecureSet-aux } Ag \ G \cup H) \subseteq \text{synth } (\text{analz } (G \cup H)) \cup \text{-payload}$

$\langle \text{proof} \rangle$

**lemma** *analz-Un-implSecureSet:*

$\text{Enc-keys-clean } (G \cup H) \implies$

$\text{analz } (\text{sym.implSecureSet } Ag \ G \cup H) \subseteq \text{synth } (\text{analz } (G \cup H)) \cup \text{-payload}$

$\langle \text{proof} \rangle$

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

**lemma** *analz-Un-implSecureSet-2-aux-1:*

$\text{Enc-keys-clean } H \implies$

$Ag \cap \text{broken } (\text{parts } H \cap \text{range } LtK) = \{\} \implies$

$\text{analz } (\text{implSecureSet-aux } Ag \ G \cup H) \subseteq \text{implSecureSet-aux } Ag \ G \cup \text{synth } (\text{analz } H)$

$\langle \text{proof} \rangle$

**lemma** *analz-Un-implSecureSet-2-aux-3:*

$\text{Enc-keys-clean } H \implies$

$Ag \cap \text{broken } (\text{parts } H \cap \text{range } LtK) = \{\} \implies$

$\text{analz } (\text{implSecureSet-aux } Ag \ G \cup H) \subseteq \text{synth } (\text{analz } H) \cup \text{-payload}$

$\langle \text{proof} \rangle$

**lemma** *analz-Un-implSecureSet-2:*

$\text{Enc-keys-clean } H \implies$

$Ag \cap \text{broken } (\text{parts } H \cap \text{range } LtK) = \{\} \implies$

$\text{analz } (\text{sym.implSecureSet } Ag \ G \cup H) \subseteq \text{synth } (\text{analz } H) \cup \text{-payload}$

$\langle \text{proof} \rangle$

## 14.5 Pull *implAuthSet* out of *analz*

**lemma** *analz-Un-implAuthSet-aux-1:*

$\text{Enc-keys-clean } (G \cup H) \implies$

$\text{analz } (\text{implAuthSet-aux } G \cup H) \subseteq$

$\text{implAuthSet-aux } G \cup \text{HashSet } (\text{PairSet } (\text{PairSet } \text{Tags } G) (\text{range } (\text{case-prod } \text{shrK}))) \cup$

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

$\langle \text{proof} \rangle$

**lemma** *analz-Un-implAuthSet-aux-2:*

$\text{Enc-keys-clean } (G \cup H) \implies$

$\text{analz } (\text{implAuthSet-aux } G \cup H) \subseteq$

$\text{implAuthSet-aux } G \cup \text{HashSet } (\text{PairSet } (\text{PairSet } \text{Tags } G) (\text{range } (\text{case-prod } \text{shrK}))) \cup$

$\text{synth } (\text{analz } (G \cup H))$

$\langle \text{proof} \rangle$

**lemma** *analz-Un-implAuthSet-aux-3:*

$\text{Enc-keys-clean } (G \cup H) \implies$

$\text{analz } (\text{implAuthSet-aux } G \cup H) \subseteq \text{synth } (\text{analz } (G \cup H)) \cup \text{-payload}$

$\langle \text{proof} \rangle$

**lemma** *analz-Un-implAuthSet:*

*Enc-keys-clean*  $(G \cup H) \implies$   
 $\text{analz } (\text{sym.implAuthSet } G \cup H) \subseteq \text{synth } (\text{analz } (G \cup H)) \cup \text{--payload}$   
 $\langle \text{proof} \rangle$

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

## 14.6 Locale interpretations

**interpretation** *sym: semivalid-implem implem-sym*  
 $\langle \text{proof} \rangle$

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

**lemma** *implInsec-synth-analz*:  
 $H \subseteq \text{payload} \cup \text{sym.valid} \cup \text{range } \text{LtK} \cup \text{Tags} \implies$   
 $\text{sym.implInsec } A B M \in \text{synth } (\text{analz } H) \implies$   
 $\text{sym.implInsec } A B M \in H \vee M \in \text{synth } (\text{analz } H)$   
 $\langle \text{proof} \rangle$

**lemma** *implConfid-synth-analz*:  
 $H \subseteq \text{payload} \cup \text{sym.valid} \cup \text{range } \text{LtK} \cup \text{Tags} \implies$   
 $\text{sym.implConfid } A B M \in \text{synth } (\text{analz } H) \implies$   
 $\text{sym.implConfid } A B M \in H \vee M \in \text{synth } (\text{analz } H)$   
 $\langle \text{proof} \rangle$

**lemma** *implAuth-synth-analz*:  
 $H \subseteq \text{payload} \cup \text{sym.valid} \cup \text{range } \text{LtK} \cup \text{Tags} \implies$   
 $\text{sym.implAuth } A B M \in \text{synth } (\text{analz } H) \implies$   
 $\text{sym.implAuth } A B M \in H \vee (M \in \text{synth } (\text{analz } H) \wedge (A, B) \in \text{broken } H)$   
 $\langle \text{proof} \rangle$

**lemma** *implSecure-synth-analz*:  
 $H \subseteq \text{payload} \cup \text{sym.valid} \cup \text{range } \text{LtK} \cup \text{Tags} \implies$   
 $\text{sym.implSecure } A B M \in \text{synth } (\text{analz } H) \implies$   
 $\text{sym.implSecure } A B M \in H \vee (M \in \text{synth } (\text{analz } H) \wedge (A, B) \in \text{broken } H)$   
 $\langle \text{proof} \rangle$

**interpretation** *sym: valid-implem implem-sym*  
 $\langle \text{proof} \rangle$

**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*  
 $\langle$ *proof* $\rangle$

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  
 $\langle$ *proof* $\rangle$

**lemma** *implAuthSet-implAuthSet-aux*: *asym.implAuthSet* G  $\subseteq$  *implAuthSet-aux* G  
 $\langle$ *proof* $\rangle$

**lemma** *implConfidSet-implConfidSet-aux*: *asym.implConfidSet* Ag G  $\subseteq$  *implConfidSet-aux* Ag G  
 $\langle$ *proof* $\rangle$

**lemma** *implSecureSet-implSecureSet-aux*: *asym.implSecureSet* Ag G  $\subseteq$  *implSecureSet-aux* Ag G  
 $\langle$ *proof* $\rangle$

**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**

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

*<proof>*

### 15.2.2 Pull *implInsecSet* out of *analz*

**lemma** *analz-Un-implInsecSet-aux-aux*:

**assumes** *Enc-keys-clean* ( $G \cup H$ )

**shows**  $\text{analz } (\text{implInsecSet-aux } G \cup H) \subseteq \text{implInsecSet-aux } G \cup \text{Tags} \cup \text{synth } (\text{analz } (G \cup H))$

*<proof>*

**lemma** *analz-Un-implInsecSet-aux*:

*Enc-keys-clean* ( $G \cup H$ )  $\implies$

$$\text{analz } (\text{implInsecSet-aux } G \cup H) \subseteq \text{synth } (\text{analz } (G \cup H)) \cup \text{-payload}$$

*<proof>*

**lemma** *analz-Un-implInsecSet*:

*Enc-keys-clean* ( $G \cup H$ )  $\implies$

$$\text{analz } (\text{asym.implInsecSet } G \cup H) \subseteq \text{synth } (\text{analz } (G \cup H)) \cup \text{-payload}$$

*<proof>*

## 15.3 Pull *implConfidSet* out of *analz*

**lemma** *analz-Un-implConfidSet-aux-aux*:

*Enc-keys-clean* ( $G \cup H$ )  $\implies$

$$\text{analz } (\text{implConfidSet-aux Ag } G \cup H) \subseteq$$

$$\text{implConfidSet-aux Ag } G \cup \text{PairSet AgentSet } G \cup$$

$$\text{synth } (\text{analz } (G \cup H))$$

*<proof>*

**lemma** *analz-Un-implConfidSet-aux:*

$Enc\text{-keys-clean } (G \cup H) \implies$   
 $analz (implConfidSet\text{-aux } Ag \ G \cup H) \subseteq synth (analz (G \cup H)) \cup \text{-payload}$   
 ⟨proof⟩

**lemma** *analz-Un-implConfidSet:*

$Enc\text{-keys-clean } (G \cup H) \implies$   
 $analz (asym.implConfidSet \ Ag \ G \cup H) \subseteq synth (analz (G \cup H)) \cup \text{-payload}$   
 ⟨proof⟩

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

**lemma** *analz-Un-implConfidSet-aux-aux-2:*

$Enc\text{-keys-clean } H \implies$   
 $Ag \cap broken (parts \ H \cap range \ LtK) = \{\} \implies$   
 $analz (implConfidSet\text{-aux } Ag \ G \cup H) \subseteq implConfidSet\text{-aux } Ag \ G \cup synth (analz \ H)$   
 ⟨proof⟩

**lemma** *analz-Un-implConfidSet-aux-2:*

$Enc\text{-keys-clean } H \implies$   
 $Ag \cap broken (parts \ H \cap range \ LtK) = \{\} \implies$   
 $analz (implConfidSet\text{-aux } Ag \ G \cup H) \subseteq synth (analz \ H) \cup \text{-payload}$   
 ⟨proof⟩

**lemma** *analz-Un-implConfidSet-2:*

$Enc\text{-keys-clean } H \implies$   
 $Ag \cap broken (parts \ H \cap range \ LtK) = \{\} \implies$   
 $analz (asym.implConfidSet \ Ag \ G \cup H) \subseteq synth (analz \ H) \cup \text{-payload}$   
 ⟨proof⟩

## 15.4 Pull *implAuthSet* out of *analz*

**lemma** *analz-Un-implAuthSet-aux-aux:*

$Enc\text{-keys-clean } (G \cup H) \implies$   
 $analz (implAuthSet\text{-aux } G \cup H) \subseteq implAuthSet\text{-aux } G \cup synth (analz (G \cup H))$   
 ⟨proof⟩

**lemma** *analz-Un-implAuthSet-aux:*

$Enc\text{-keys-clean } (G \cup H) \implies$   
 $analz (implAuthSet\text{-aux } G \cup H) \subseteq synth (analz (G \cup H)) \cup \text{-payload}$   
 ⟨proof⟩

**lemma** *analz-Un-implAuthSet:*

$Enc\text{-keys-clean } (G \cup H) \implies$   
 $analz (asym.implAuthSet \ G \cup H) \subseteq synth (analz (G \cup H)) \cup \text{-payload}$   
 ⟨proof⟩

## 15.5 Pull *implSecureSet* out of *analz*

**lemma** *analz-Un-implSecureSet-aux-aux:*

$Enc\text{-keys-clean } (G \cup H) \implies$   
 $analz (implSecureSet\text{-aux } Ag \ G \cup H) \subseteq$   
 $implSecureSet\text{-aux } Ag \ G \cup AencSet (PairSet \ Tags \ (PairSet \ AgentSet \ G)) (pubK' (Ag'' \ UNIV)) \cup$   
 $PairSet \ Tags \ (PairSet \ AgentSet \ G) \cup Tags \cup PairSet \ AgentSet \ G \cup$

$\text{synth } (\text{analz } (G \cup H))$   
 $\langle \text{proof} \rangle$

**lemma** *analz-Un-implSecureSet-aux:*

$\text{Enc-keys-clean } (G \cup H) \implies$   
 $\text{analz } (\text{implSecureSet-aux } Ag \ G \cup H) \subseteq \text{synth } (\text{analz } (G \cup H)) \cup \text{-payload}$   
 $\langle \text{proof} \rangle$

**lemma** *analz-Un-implSecureSet:*

$\text{Enc-keys-clean } (G \cup H) \implies$   
 $\text{analz } (\text{asym.implSecureSet } Ag \ G \cup H) \subseteq \text{synth } (\text{analz } (G \cup H)) \cup \text{-payload}$   
 $\langle \text{proof} \rangle$

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

**lemma** *analz-Un-implSecureSet-aux-aux-2:*

$\text{Enc-keys-clean } (G \cup H) \implies$   
 $Ag \cap \text{broken } (\text{parts } H \cap \text{range } LtK) = \{\} \implies$   
 $\text{analz } (\text{implSecureSet-aux } Ag \ G \cup H) \subseteq$   
 $\text{implSecureSet-aux } Ag \ G \cup \text{AencSet } (\text{PairSet } Tags \ (\text{PairSet } AgentSet \ G)) \ (\text{pubK}' (Ag \text{' UNIV})) \cup$   
 $\text{synth } (\text{analz } H)$   
 $\langle \text{proof} \rangle$

**lemma** *analz-Un-implSecureSet-aux-2:*

$\text{Enc-keys-clean } (G \cup H) \implies$   
 $Ag \cap \text{broken } (\text{parts } H \cap \text{range } LtK) = \{\} \implies$   
 $\text{analz } (\text{implSecureSet-aux } Ag \ G \cup H) \subseteq \text{synth } (\text{analz } H) \cup \text{-payload}$   
 $\langle \text{proof} \rangle$

**lemma** *analz-Un-implSecureSet-2:*

$\text{Enc-keys-clean } (G \cup H) \implies$   
 $Ag \cap \text{broken } (\text{parts } H \cap \text{range } LtK) = \{\} \implies$   
 $\text{analz } (\text{asym.implSecureSet } Ag \ G \cup H) \subseteq$   
 $\text{synth } (\text{analz } H) \cup \text{-payload}$   
 $\langle \text{proof} \rangle$

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

## 15.6 Locale interpretations

**interpretation** *asym: semivalid-implem implem-asym*

$\langle \text{proof} \rangle$

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:*

$H \subseteq \text{payload} \cup \text{asym.valid} \cup \text{range } LtK \cup Tags \implies$   
 $\text{asym.implInsec } A \ B \ M \in \text{synth } (\text{analz } H) \implies$   
 $\text{asym.implInsec } A \ B \ M \in I \vee M \in \text{synth } (\text{analz } H)$   
 $\langle \text{proof} \rangle$



**lemma** *implConfid-synth-analz*:

$H \subseteq \text{payload} \cup \text{asym.valid} \cup \text{range LtK} \cup \text{Tags} \implies$   
 $\text{asym.implConfid } A \ B \ M \in \text{synth } (\text{analz } H) \implies$   
 $\text{asym.implConfid } A \ B \ M \in H \vee M \in \text{synth } (\text{analz } H)$

$\langle \text{proof} \rangle$

**lemma** *implAuth-synth-analz*:

$H \subseteq \text{payload} \cup \text{asym.valid} \cup \text{range LtK} \cup \text{Tags} \implies$   
 $\text{asym.implAuth } A \ B \ M \in \text{synth } (\text{analz } H) \implies$   
 $\text{asym.implAuth } A \ B \ M \in H \vee (M \in \text{synth } (\text{analz } H) \wedge (A, B) \in \text{broken } H)$

$\langle \text{proof} \rangle$

**lemma** *implSecure-synth-analz*:

$H \subseteq \text{payload} \cup \text{asym.valid} \cup \text{range LtK} \cup \text{Tags} \implies$   
 $\text{asym.implSecure } A \ B \ M \in \text{synth } (\text{analz } H) \implies$   
 $\text{asym.implSecure } A \ B \ M \in H \vee (M \in \text{synth } (\text{analz } H) \wedge (A, B) \in \text{broken } H)$

$\langle \text{proof} \rangle$

**interpretation** *asym: valid-implem implem-asym*

$\langle \text{proof} \rangle$

**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 partitioning of  $ik$ , since we know that the messages added by the protocol do not contain  $ltk$  keys 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*]:

$role (guessed-runs R) = Resp \implies guessed-frame R xsk = Some (NonceF (R\$sk))$

$\langle proof \rangle$

**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*]:

$run-ended R \implies R = None \implies False$

$\langle proof \rangle$

*test-ended* *s*  $\longleftrightarrow$  the test run has ended in *s*

**abbreviation**

*test-ended* :: '*a l1-state-scheme*  $\Rightarrow$  *bool*

**where**

$test-ended s \equiv run-ended (progress s test)$

a run can emit signals if it involves the same agents as the test run, and if the test run has not ended yet

**definition**

$can\text{-}signal :: 'a\ l1\text{-}state\text{-}scheme \Rightarrow agent \Rightarrow agent \Rightarrow bool$

**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  $pkE$ ,  $skE$
- 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**

$l1\text{-}step1 :: rid\text{-}t \Rightarrow agent \Rightarrow agent \Rightarrow ('a\ l1\text{-}state\text{-}scheme * 'a\ l1\text{-}state\text{-}scheme)\ set$

**where**

$l1\text{-}step1\ Ra\ A\ B \equiv \{(s, s')\}.$   
 — guards:  
 $Ra \notin dom\ (progress\ s) \wedge$   
 $guessed\text{-}runs\ Ra = (\text{role} = Init, \text{owner} = A, \text{partner} = B) \wedge$   
 — actions:  
 $s' = s(\$   
 $\quad progress := (progress\ s)(Ra \mapsto \{xpkE, xskE\})$   
 $\quad \left. \right)$   
 $\}$

**definition**

$l1\text{-}step2 :: rid\text{-}t \Rightarrow agent \Rightarrow agent \Rightarrow msg \Rightarrow ('a\ l1\text{-}state\text{-}scheme * 'a\ l1\text{-}state\text{-}scheme)\ set$

**where**

$l1\text{-}step2\ Rb\ A\ B\ KE \equiv \{(s, s')\}.$   
 — guards:  
 $guessed\text{-}runs\ Rb = (\text{role} = Resp, \text{owner} = B, \text{partner} = A) \wedge$   
 $Rb \notin dom\ (progress\ s) \wedge$   
 $guessed\text{-}frame\ Rb\ xpkE = Some\ KE \wedge$

$(\text{can-signal } s \ A \ B \longrightarrow \text{--- authentication guard}$   
 $(\exists \ Ra. \ \text{guessed-runs } Ra = (\text{role=Init, owner=A, partner=B}) \wedge$   
 $\text{in-progress } (\text{progress } s \ Ra) \ xpkE \wedge \text{guessed-frame } Ra \ xpkE = \text{Some } KE)) \wedge$   
 $(Rb = \text{test} \longrightarrow \text{NonceF } (Rb\$sk) \notin \text{synth } (\text{analz } (ik \ s))) \wedge$   
 $\text{--- actions:}$   
 $s' = s(\text{progress} := (\text{progress } s)(Rb \mapsto \{xpkE, xsk\}),$   
 $\text{secret} := \{x. \ x = \text{NonceF } (Rb\$sk) \wedge Rb = \text{test}\} \cup \text{secret } s,$   
 $\text{signals} := \text{if can-signal } s \ A \ B \ \text{then}$   
 $\text{addSignal } (\text{signals } s) \ (\text{Running } A \ B \ (\langle KE, \text{NonceF } (Rb\$sk) \rangle))$   
 $\text{else}$   
 $\text{signals } s$   
 $\quad \triangleright$   
 $\}$

**definition**

$l1\text{-step3} :: \text{rid-t} \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow \text{msg} \Rightarrow ('a \ l1\text{-state-scheme} * 'a \ l1\text{-state-scheme}) \ \text{set}$

**where**

$l1\text{-step3 } Ra \ A \ B \ K \equiv \{(s, s')\}.$

$\text{--- guards:}$

$\text{guessed-runs } Ra = (\text{role=Init, owner=A, partner=B}) \wedge$

$\text{progress } s \ Ra = \text{Some } \{xpkE, xskE\} \wedge$

$\text{guessed-frame } Ra \ xsk = \text{Some } K \wedge$

$(\text{can-signal } s \ A \ B \longrightarrow \text{--- authentication guard}$

$(\exists \ Rb. \ \text{guessed-runs } Rb = (\text{role=Resp, owner=B, partner=A}) \wedge$

$\text{progress } s \ Rb = \text{Some } \{xpkE, xsk\} \wedge$

$\text{guessed-frame } Rb \ xpkE = \text{Some } (\text{epubKF } (Ra\$kE)) \wedge$

$\text{guessed-frame } Rb \ xsk = \text{Some } K)) \wedge$

$(Ra = \text{test} \longrightarrow K \notin \text{synth } (\text{analz } (ik \ s))) \wedge$

$\text{--- actions:}$

$s' = s(\text{progress} := (\text{progress } s)(Ra \mapsto \{xpkE, xskE, xsk\}),$

$\text{secret} := \{x. \ x = K \wedge Ra = \text{test}\} \cup \text{secret } s,$

$\text{signals} := \text{if can-signal } s \ A \ B \ \text{then}$

$\text{addSignal } (\text{signals } s) \ (\text{Commit } A \ B \ (\langle \text{epubKF } (Ra\$kE), K \rangle))$

$\text{else}$

$\text{signals } s$

$\quad \triangleright$

$\}$

specification

**definition**

$l1\text{-init} :: l1\text{-state} \ \text{set}$

**where**

$l1\text{-init} \equiv \{$

$\text{ik} = \{\},$

$\text{secret} = \{\},$

$\text{progress} = \text{Map.empty},$

$\text{signals} = \lambda x. \ 0$

$\quad \triangleright \}$

**definition**

$l1\text{-trans} :: ('a \ l1\text{-state-scheme} * 'a \ l1\text{-state-scheme}) \ \text{set} \ \text{where}$

$l1\text{-trans} \equiv (\bigcup m \ Ra \ Rb \ A \ B \ K \ KE.$

```

    l1-step1 Ra A B ∪
    l1-step2 Rb A B KE ∪
    l1-step3 Ra A B K ∪
    l1-learn m ∪
    Id
  )

```

**definition**

```

l1 :: (l1-state, l1-obs) spec where
l1 ≡ ⟨
  init = l1-init,
  trans = l1-trans,
  obs = id
⟩

```

**lemmas** l1-defs =

```

l1-def l1-init-def l1-trans-def
l1-learn-def
l1-step1-def l1-step2-def l1-step3-def

```

**lemmas** l1-nostep-defs =

```

l1-def l1-init-def l1-trans-def

```

**lemma** l1-obs-id [simp]: obs l1 = id  
⟨proof⟩

**declare** domIff [iff]

**lemma** run-ended-trans:  
run-ended (progress s R) ⇒  
(s, s') ∈ trans l1 ⇒  
run-ended (progress s' R)  
⟨proof⟩

**declare** domIff [iff del]

**lemma** can-signal-trans:  
can-signal s' A B ⇒  
(s, s') ∈ trans l1 ⇒  
can-signal s A B  
⟨proof⟩

## 16.2 Refinement: secrecy

mediator function

**definition**

```

med01s :: l1-obs ⇒ s0-obs

```

**where**

```

med01s t ≡ ⟨ ik = ik t, secret = secret t ⟩

```

relation between states

**definition**

$R01s :: (s0\text{-state} * l1\text{-state}) \text{ set}$

**where**

$$R01s \equiv \{(s, s') . \\ s = (\text{ik} = \text{ik } s', \text{secret} = \text{secret } s') \\ \}$$

protocol independent events

**lemma** *l1-learn-refines-learn*:

$$\{R01s\} \text{ s0-learn } m, \text{ l1-learn } m \{>R01s\}$$

$\langle \text{proof} \rangle$

protocol events

**lemma** *l1-step1-refines-skip*:

$$\{R01s\} \text{ Id}, \text{ l1-step1 } Ra \ A \ B \{>R01s\}$$

$\langle \text{proof} \rangle$

**lemma** *l1-step2-refines-add-secret-skip*:

$$\{R01s\} \text{ s0-add-secret } (\text{NonceF } (Rb\$sk)) \cup \text{ Id}, \text{ l1-step2 } Rb \ A \ B \ KE \{>R01s\}$$

$\langle \text{proof} \rangle$

**lemma** *l1-step3-refines-add-secret-skip*:

$$\{R01s\} \text{ s0-add-secret } K \cup \text{ Id}, \text{ l1-step3 } Ra \ A \ B \ K \{>R01s\}$$

$\langle \text{proof} \rangle$

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]*:

$$\text{init } l1 \subseteq R01s \text{ “ } (\text{init } s0)$$

$\langle \text{proof} \rangle$

**lemma** *l1-refines-trans-s0 [iff]*:

$$\{R01s\} \text{ trans } s0, \text{ trans } l1 \{> R01s\}$$

$\langle \text{proof} \rangle$

**lemma** *obs-consistent-med01x [iff]*:

$$\text{obs-consistent } R01s \text{ med01s } s0 \ l1$$

$\langle \text{proof} \rangle$

refinement result

**lemma** *l1s-refines-s0 [iff]*:

$$\text{refines} \\ R01s \\ \text{med01s } s0 \ l1$$

$\langle \text{proof} \rangle$

**lemma** *l1-implements-s0 [iff]*: *implements med01s s0 l1*

$\langle \text{proof} \rangle$

### 16.3 Derived invariants: secrecy

abbreviation  $l1\text{-secrecy} \equiv s0\text{-secrecy}$

**lemma**  $l1\text{-obs-secrecy}$  [iff]:  $oreach\ l1 \subseteq l1\text{-secrecy}$   
<proof>

**lemma**  $l1\text{-secrecy}$  [iff]:  $reach\ l1 \subseteq l1\text{-secrecy}$   
<proof>

### 16.4 Invariants

#### 16.4.1 inv1

if a commit signal for a nonce has been emitted, then there is a finished initiator run with this nonce.

**definition**

$l1\text{-inv1} :: l1\text{-state set}$

**where**

$l1\text{-inv1} \equiv \{s. \forall Ra\ A\ B\ K.$   
 $signals\ s\ (Commit\ A\ B\ (epubKF\ (Ra\$kE),\ K))\ >\ 0 \longrightarrow$   
 $guessed\text{-runs}\ Ra = (role=Init,\ owner=A,\ partner=B) \wedge$   
 $progress\ s\ Ra = Some\ \{xpkE,\ xskE,\ xsk\} \wedge$   
 $guessed\text{-frame}\ Ra\ xsk = Some\ K$   
 $\}$

**lemmas**  $l1\text{-inv1I} = l1\text{-inv1-def}$  [THEN setc-def-to-intro, rule-format]

**lemmas**  $l1\text{-inv1E}$  [elim] =  $l1\text{-inv1-def}$  [THEN setc-def-to-elim, rule-format]

**lemmas**  $l1\text{-inv1D} = l1\text{-inv1-def}$  [THEN setc-def-to-dest, rule-format, rotated 1, simplified]

**lemma**  $l1\text{-inv1-init}$  [iff]:

$init\ l1 \subseteq l1\text{-inv1}$

<proof>

**declare**  $domIff$  [iff]

**lemma**  $l1\text{-inv1-trans}$  [iff]:

$\{l1\text{-inv1}\ trans\ l1\ \{>\ l1\text{-inv1}\}$

<proof>

**lemma**  $PO\text{-}l1\text{-inv1}$  [iff]:  $reach\ l1 \subseteq l1\text{-inv1}$

<proof>

#### 16.4.2 inv2

if a responder run knows a nonce, then a running signal for this nonce has been emitted

**definition**

$l1\text{-inv2} :: l1\text{-state set}$

**where**

$l1\text{-inv2} \equiv \{s. \forall\ KE\ A\ B\ Rb.$



$$\begin{aligned}
& \text{guessed-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \longrightarrow \\
& \text{progress } s \ Rb = \text{Some } \{xpkE, xsk\} \longrightarrow \\
& \text{guessed-frame } Rb \ xpkE = \text{Some } KE \longrightarrow \\
& \text{can-signal } s \ A \ B \longrightarrow \\
& \quad \text{signals } s \ (\text{Running } A \ B \ \langle KE, \text{NonceF } (Rb\$sk) \rangle) > 0 \\
& \}
\end{aligned}$$

**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]:

$\text{init } l1 \subseteq l1\text{-inv}2$

$\langle \text{proof} \rangle$

**lemma**  $l1\text{-inv}2\text{-trans}$  [iff]:

$\{l1\text{-inv}2\} \text{ trans } l1 \ \{> \ l1\text{-inv}2\}$

$\langle \text{proof} \rangle$

**lemma**  $PO\text{-}l1\text{-inv}2$  [iff]:  $\text{reach } l1 \subseteq l1\text{-inv}2$

$\langle \text{proof} \rangle$

### 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**

$l1\text{-inv}3 :: l1\text{-state set}$

**where**

$l1\text{-inv}3 \equiv \{s. \forall \ A \ B \ Rb \ Ra.$

$\text{guessed-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \longrightarrow$

$\text{progress } s \ Rb = \text{Some } \{xpkE, xsk\} \longrightarrow$

$\text{guessed-frame } Rb \ xpkE = \text{Some } (\text{epubKF } (Ra\$kE)) \longrightarrow$

$\text{guessed-runs } Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \longrightarrow$

$\text{progress } s \ Ra = \text{Some } \{xpkE, xskE\} \longrightarrow$

$\text{can-signal } s \ A \ B \longrightarrow$

$\quad \text{signals } s \ (\text{Commit } A \ B \ (\langle \text{epubKF } (Ra\$kE), \text{NonceF } (Rb\$sk) \rangle))$

$< \text{signals } s \ (\text{Running } A \ B \ (\langle \text{epubKF } (Ra\$kE), \text{NonceF } (Rb\$sk) \rangle))$

$\}$

**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**  $l1\text{-inv}3\text{-derived}$ :  $l1\text{-inv}1 \cap l1\text{-inv}2 \subseteq l1\text{-inv}3$

$\langle \text{proof} \rangle$

## 16.5 Refinement: injective agreement

mediator function

**definition**

$med01ia :: l1-obs \Rightarrow a0i-obs$

**where**

$med01ia\ t \equiv (\lambda a0n-state. signals = signals\ t)$

relation between states

**definition**

$R01ia :: (a0i-state * l1-state)\ set$

**where**

$R01ia \equiv \{(s, s') .$   
 $\quad a0n-state.signals\ s = signals\ s'$   
 $\}$

protocol independent events

**lemma** *l1-learn-refines-a0-ia-skip*:

$\{R01ia\}\ Id, l1-learn\ m\ \{>R01ia\}$   
 $\langle proof \rangle$

protocol events

**lemma** *l1-step1-refines-a0i-skip*:

$\{R01ia\}\ Id, l1-step1\ Ra\ A\ B\ \{>R01ia\}$   
 $\langle proof \rangle$

**lemma** *l1-step2-refines-a0i-running-skip*:

$\{R01ia\}\ a0i-running\ A\ B\ \langle KE, NonceF\ (Rb\$sk) \rangle \cup Id, l1-step2\ Rb\ A\ B\ KE\ \{>R01ia\}$   
 $\langle proof \rangle$

**lemma** *l1-step3-refines-a0i-commit-skip*:

$\{R01ia \cap (UNIV \times l1-inv3)\}\ a0i-commit\ A\ B\ \langle epubKF\ (Ra\$kE), K \rangle \cup Id, l1-step3\ Ra\ A\ B\ K$   
 $\{>R01ia\}$   
 $\langle proof \rangle$

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]*:

$init\ l1 \subseteq R01ia \iff (init\ a0i)$   
 $\langle proof \rangle$

**lemma** *l1-refines-trans-a0i [iff]*:

$\{R01ia \cap (UNIV \times (l1-inv1 \cap l1-inv2))\}\ trans\ a0i, trans\ l1\ \{>R01ia\}$   
 $\langle proof \rangle$

**lemma** *obs-consistent-med01ia [iff]*:

$obs-consistent\ R01ia\ med01ia\ a0i\ l1$   
 $\langle proof \rangle$

refinement result

**lemma** *l1-refines-a0i* [iff]:

*refines*

$(R01ia \cap (\text{reach } a0i \times (l1\text{-inv1} \cap l1\text{-inv2})))$   
*med01ia a0i l1*

*<proof>*

**lemma** *l1-implements-a0i* [iff]: *implements med01ia a0i l1*

*<proof>*

## 16.6 Derived invariants: injective agreement

**definition**

*l1-agreement* :: ('a l1-state-scheme) set

**where**

*l1-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*

*<proof>*

**lemma** *l1-agreement* [iff]: *reach l1*  $\subseteq$  *l1-agreement*

*<proof>*

**end**

## 17 Key Transport Protocol with PFS (L2)

**theory** *pfslvl2*

**imports** *pfslvl1 Channels*

**begin**

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

### 17.1 State and Events

initial compromise

**consts**

*bad-init* :: *agent set*

**specification** (*bad-init*)

*bad-init-spec*: *test-owner*  $\notin$  *bad-init*  $\wedge$  *test-partner*  $\notin$  *bad-init*  
 $\langle$ *proof* $\rangle$

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

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

}

**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

$s' = s(\text{chan} := \{M\} \cup \text{chan } s)$

}

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*

$\langle$ *proof* $\rangle$

**lemma** *role-comp-inv-eq*:

*y* = *role-comp* *x*  $\longleftrightarrow$  *x* = *role-comp* *y*

$\langle$ *proof* $\rangle$

**definition**

*partners* :: *rid-t* *set*

**where**

*partners*  $\equiv \{R. \textit{partner-runs } \textit{test } R\}$

**lemma** *test-not-partner* [*simp*]:

*test*  $\notin$  *partners*

$\langle$ *proof* $\rangle$

**lemma** *matching-symmetric*:

*matching* *sigma* *sigma'*  $\Longrightarrow$  *matching* *sigma'* *sigma*

$\langle$ *proof* $\rangle$

**lemma** *partner-symmetric*:

*partner-runs* *R* *R'*  $\Longrightarrow$  *partner-runs* *R'* *R*

$\langle$ *proof* $\rangle$

**lemma** *partner-unique*:

*partner-runs* *R* *R''*  $\Longrightarrow$  *partner-runs* *R* *R'*  $\Longrightarrow$  *R'* = *R''*

$\langle$ *proof* $\rangle$

**lemma** *partner-test*:

$R \in \text{partners} \implies \text{partner-runs } R \ R' \implies R' = \text{test}$   
(*proof*)

compromising events

**definition**

$l2\text{-lkr}\text{-others} :: \text{agent} \Rightarrow l2\text{-trans}$

**where**

$l2\text{-lkr}\text{-others } A \equiv \{(s, s')\}.$   
— guards  
 $A \neq \text{test-owner} \wedge$   
 $A \neq \text{test-partner} \wedge$   
— actions  
 $s' = s(\text{bad} := \{A\} \cup \text{bad } s)$   
}

**definition**

$l2\text{-lkr}\text{-actor} :: \text{agent} \Rightarrow l2\text{-trans}$

**where**

$l2\text{-lkr}\text{-actor } A \equiv \{(s, s')\}.$   
— 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  
 $R \neq \text{test} \wedge R \notin \text{partners} \wedge$   
 $\text{in-progress } (\text{progress } s \ R) \ xsk \wedge$   
 $\text{guessed-frame } R \ xsk = \text{Some } K \wedge$   
— 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:

$s' = s \{$

$\text{progress} := (\text{progress } s)(Ra \mapsto \{xpkE, xskE\}),$

$\text{chan} := \{\text{Auth } A \ B \ (\langle \text{Number } 0, \text{epubKF } (Ra\$kE) \rangle)\} \cup (\text{chan } s)$

$\}$

$\}$

**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:

$\text{guessed-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \wedge$

$Rb \notin \text{dom } (\text{progress } s) \wedge$

$\text{guessed-frame } Rb \ xpkE = \text{Some } KE \wedge$

$\text{Auth } A \ B \ (\text{Number } 0, KE) \in \text{chan } s \wedge$

— actions:

$s' = s \{$

$\text{progress} := (\text{progress } s)(Rb \mapsto \{xpkE, xsk\}),$

$\text{chan} := \{\text{Auth } B \ A \ (\text{Aenc } (\text{NonceF } (Rb\$sk)) \ KE)\} \cup (\text{chan } s),$

$\text{signals} := \text{if can-signal } s \ A \ B \ \text{then}$

$\text{addSignal } (\text{signals } s) \ (\text{Running } A \ B \ \langle KE, \text{NonceF } (Rb\$sk) \rangle)$

$\text{else}$

$\text{signals } s,$

$\text{secret} := \{x. x = \text{NonceF } (Rb\$sk) \wedge Rb = \text{test}\} \cup \text{secret } s$

$\}$

$\}$

**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:

$\text{guessed-runs } Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \wedge$

$\text{progress } s \ Ra = \text{Some } \{xpkE, xskE\} \wedge$

$\text{guessed-frame } Ra \ xsk = \text{Some } K \wedge$

$\text{Auth } B \ A \ (\text{Aenc } K \ (\text{epubKF } (Ra\$kE))) \in \text{chan } s \wedge$

— actions:

$s' = s \{ \text{progress} := (\text{progress } s)(Ra \mapsto \{xpkE, xskE, xsk\}),$

$\text{signals} := \text{if can-signal } s \ A \ B \ \text{then}$

$\text{addSignal } (\text{signals } s) \ (\text{Commit } A \ B \ \langle \text{epubKF } (Ra\$kE), K \rangle)$

$\text{else}$

$\text{signals } s,$

$\text{secret} := \{x. x = K \wedge Ra = \text{test}\} \cup \text{secret } s$

$\}$

$\}$

specification

**definition**

$l2\text{-init} :: l2\text{-state set}$

**where**

$l2\text{-init} \equiv \{ \langle \langle$   
   $ik = \{\},$   
   $secret = \{\},$   
   $progress = Map.empty,$   
   $signals = \lambda x. 0,$   
   $chan = \{\},$   
   $bad = bad\text{-init}$   
 $\rangle \rangle$

**definition**

$l2\text{-trans} :: l2\text{-trans where}$

$l2\text{-trans} \equiv (\bigcup m M KE Rb Ra A B K.$

$l2\text{-step1 } Ra A B \cup$

$l2\text{-step2 } Rb A B KE \cup$

$l2\text{-step3 } Ra A B m \cup$

$l2\text{-dy-fake-chan } M \cup$

$l2\text{-dy-fake-msg } m \cup$

$l2\text{-lkr-others } A \cup$

$l2\text{-lkr-after } A \cup$

$l2\text{-skr } Ra K \cup$

$Id$

)

**definition**

$l2 :: (l2\text{-state}, l2\text{-obs}) spec \text{ where}$

$l2 \equiv \langle \langle$

$init = l2\text{-init},$

$trans = l2\text{-trans},$

$obs = id$

$\rangle \rangle$

**lemmas**  $l2\text{-loc-defs} =$

$l2\text{-step1-def } l2\text{-step2-def } l2\text{-step3-def}$

$l2\text{-def } l2\text{-init-def } l2\text{-trans-def}$

$l2\text{-dy-fake-chan-def } l2\text{-dy-fake-msg-def}$

$l2\text{-lkr-after-def } l2\text{-lkr-others-def } l2\text{-skr-def}$

**lemmas**  $l2\text{-defs} = l2\text{-loc-defs } ik\text{-dy-def}$

**lemmas**  $l2\text{-nostep-defs} = l2\text{-def } l2\text{-init-def } l2\text{-trans-def}$

**lemma**  $l2\text{-obs-id [simp]: obs } l2 = id$

$\langle proof \rangle$

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*:  
 $run\text{-}ended\ (progress\ s\ R) \implies$   
 $(s, s') \in trans\ l2 \implies$   
 $run\text{-}ended\ (progress\ s'\ R)$   
 $\langle proof \rangle$   
**declare** *domIff* [*iff del*]

**lemma** *l2-can-signal-trans*:  
 $can\text{-}signal\ s'\ A\ B \implies$   
 $(s, s') \in trans\ l2 \implies$   
 $can\text{-}signal\ s\ A\ B$   
 $\langle proof \rangle$

## 17.2 Invariants

### 17.2.1 inv1

If  $can\text{-}signal\ s\ A\ B$  (i.e.,  $A, B$  are the test session agents and the test is not finished), then  $A, B$  are honest.

#### definition

$l2\text{-}inv1 :: l2\text{-}state\ set$

#### where

$l2\text{-}inv1 \equiv \{s. \forall A\ B.$   
 $can\text{-}signal\ s\ A\ B \longrightarrow$   
 $A \notin bad\ s \wedge B \notin bad\ s$   
 $\}$

**lemmas**  $l2\text{-}inv1I = l2\text{-}inv1\text{-}def\ [THEN\ setc\text{-}def\text{-}to\text{-}intro,\ rule\text{-}format]$

**lemmas**  $l2\text{-}inv1E\ [elim] = l2\text{-}inv1\text{-}def\ [THEN\ setc\text{-}def\text{-}to\text{-}elim,\ rule\text{-}format]$

**lemmas**  $l2\text{-}inv1D = l2\text{-}inv1\text{-}def\ [THEN\ setc\text{-}def\text{-}to\text{-}dest,\ rule\text{-}format,\ rotated\ 1,\ simplified]$

**lemma**  $l2\text{-}inv1\text{-}init\ [iff]$ :

$init\ l2 \subseteq l2\text{-}inv1$

$\langle proof \rangle$

**lemma**  $l2\text{-}inv1\text{-}trans\ [iff]$ :

$\{l2\text{-}inv1\}\ trans\ l2\ \{>\ l2\text{-}inv1\}$

$\langle proof \rangle$

**lemma**  $PO\text{-}l2\text{-}inv1\ [iff]$ :  $reach\ l2 \subseteq l2\text{-}inv1$

$\langle proof \rangle$

### 17.2.2 inv2 (authentication guard)

If  $Auth\ A\ B\ \langle Number\ 0,\ KE \rangle \in chan\ s$  and  $A, B$  are honest then the message has indeed been sent by an initiator run (with the right agents etc.)

#### definition

$l2\text{-}inv2 :: l2\text{-}state\ set$

#### where

$l2\text{-}inv2 \equiv \{s. \forall A\ B\ KE.$

$Auth\ A\ B\ \langle Number\ 0,\ KE \rangle \in chan\ s \longrightarrow$

$$\begin{aligned}
& A \notin \text{bad } s \wedge B \notin \text{bad } s \longrightarrow \\
& (\exists Ra. \\
& \quad \text{guessed-runs } Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \wedge \\
& \quad \text{in-progress } (\text{progress } s Ra) \text{ } xpkE \wedge \\
& \quad KE = \text{epubKF } (Ra\$kE)) \\
& )
\end{aligned}$$

**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]:

$\text{init } l2 \subseteq l2\text{-inv}2$

$\langle \text{proof} \rangle$

**lemma**  $l2\text{-inv}2\text{-trans}$  [iff]:

$\{l2\text{-inv}2\} \text{ trans } l2 \{> l2\text{-inv}2\}$

$\langle \text{proof} \rangle$

**lemma**  $PO\text{-}l2\text{-inv}2$  [iff]:  $\text{reach } l2 \subseteq l2\text{-inv}2$

$\langle \text{proof} \rangle$

### 17.2.3 inv3 (authentication guard)

If  $\text{Auth } B A (\text{Aenc } K (\text{epubKF } (Ra \$ kE))) \in \text{chan } s$  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**

$l2\text{-inv}3 \equiv \{s. \forall Ra A B K.$

$\text{Auth } B A (\text{Aenc } K (\text{epubKF } (Ra \$ kE))) \in \text{chan } s \longrightarrow$

$A \notin \text{bad } s \wedge B \notin \text{bad } s \longrightarrow$

$(\exists Rb.$

$\text{guessed-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \wedge$

$\text{progress } s Rb = \text{Some } \{xpkE, xsk\} \wedge$

$\text{guessed-frame } Rb \text{ } xpkE = \text{Some } (\text{epubKF } (Ra\$kE)) \wedge$

$K = \text{NonceF } (Rb\$sk)$

$)$

$\}$

**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]:

$\text{init } l2 \subseteq l2\text{-inv}3$

$\langle \text{proof} \rangle$

**lemma**  $l2\text{-inv}3\text{-trans}$  [iff]:

$\{l2\text{-inv}3\} \text{ trans } l2 \{> l2\text{-inv}3\}$

$\langle \text{proof} \rangle$

**lemma** *PO-l2-inv3* [iff]: *reach l2*  $\subseteq$  *l2-inv3*  
 ⟨proof⟩

#### 17.2.4 inv4

If the test run is finished and has the session key generated by a run, then this run is also finished.

**definition**

*l2-inv4* :: *l2-state set*

**where**

$l2\text{-inv4} \equiv \{s. \forall Rb.$   
 $in\text{-progress } (progress\ s\ test)\ xsk \longrightarrow$   
 $guessed\text{-frame } test\ xsk = Some\ (NonceF\ (Rb\$sk)) \longrightarrow$   
 $progress\ s\ Rb = Some\ \{xpkE, xsk\}$   
 $\}$

**lemmas** *l2-inv4I* = *l2-inv4-def* [THEN *setc-def-to-intro*, *rule-format*]

**lemmas** *l2-inv4E* [elim] = *l2-inv4-def* [THEN *setc-def-to-elim*, *rule-format*]

**lemmas** *l2-inv4D* = *l2-inv4-def* [THEN *setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

**lemma** *l2-inv4-init* [iff]:

*init l2*  $\subseteq$  *l2-inv4*

⟨proof⟩

**lemma** *l2-inv4-trans* [iff]:

$\{l2\text{-inv4} \cap l2\text{-inv3} \cap l2\text{-inv1}\ trans\ l2\ \{>\ l2\text{-inv4}\}$

⟨proof⟩

**lemma** *PO-l2-inv4* [iff]: *reach l2*  $\subseteq$  *l2-inv4*

⟨proof⟩

#### 17.2.5 inv5

The only confidential or secure messages on the channel have been put there by the attacker.

**definition**

*l2-inv5* :: *l2-state set*

**where**

$l2\text{-inv5} \equiv \{s. \forall A\ B\ M.$   
 $(Confid\ A\ B\ M \in chan\ s \vee Secure\ A\ B\ M \in chan\ s) \longrightarrow$   
 $M \in dy\text{-fake-msg } (bad\ s)\ (ik\ s)\ (chan\ s)$   
 $\}$

**lemmas** *l2-inv5I* = *l2-inv5-def* [THEN *setc-def-to-intro*, *rule-format*]

**lemmas** *l2-inv5E* [elim] = *l2-inv5-def* [THEN *setc-def-to-elim*, *rule-format*]

**lemmas** *l2-inv5D* = *l2-inv5-def* [THEN *setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

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

*init l2*  $\subseteq$  *l2-inv5*

⟨proof⟩

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

$\{l2\text{-inv5}\} \text{ trans } l2 \{> l2\text{-inv5}\}$   
 ⟨proof⟩

**lemma** *PO-l2-inv5* [iff]:  $\text{reach } l2 \subseteq l2\text{-inv5}$   
 ⟨proof⟩

### 17.2.6 inv6

If an initiator  $Ra$  knows a session key  $K$ , then the attacker knows  $Aenc\ K$  ( $epubKF\ (Ra\ \$\ kE)$ ).

#### definition

$l2\text{-inv6} :: l2\text{-state set}$

#### where

$l2\text{-inv6} \equiv \{s. \forall Ra\ K.$   
 $\text{role } (guessed\text{-runs } Ra) = \text{Init} \longrightarrow$   
 $\text{in}\text{-progress } (\text{progress } s\ Ra) \ xsk \longrightarrow$   
 $\text{guessed}\text{-frame } Ra\ xsk = \text{Some } K \longrightarrow$   
 $Aenc\ K\ (epubKF\ (Ra\$kE)) \in \text{extr } (bad\ s)\ (ik\ s)\ (chan\ s)$   
 $\}$

**lemmas**  $l2\text{-inv6I} = l2\text{-inv6}\text{-def } [THEN\ \text{setc}\text{-def}\text{-to}\text{-intro},\ \text{rule}\text{-format}]$

**lemmas**  $l2\text{-inv6E} [elim] = l2\text{-inv6}\text{-def } [THEN\ \text{setc}\text{-def}\text{-to}\text{-elim},\ \text{rule}\text{-format}]$

**lemmas**  $l2\text{-inv6D} = l2\text{-inv6}\text{-def } [THEN\ \text{setc}\text{-def}\text{-to}\text{-dest},\ \text{rule}\text{-format},\ \text{rotated } 1,\ \text{simplified}]$

**lemma**  $l2\text{-inv6}\text{-init}$  [iff]:  
 $\text{init } l2 \subseteq l2\text{-inv6}$   
 ⟨proof⟩

**lemma**  $l2\text{-inv6}\text{-trans}$  [iff]:  
 $\{l2\text{-inv6}\} \text{ trans } l2 \{> l2\text{-inv6}\}$   
 ⟨proof⟩

**lemma** *PO-l2-inv6* [iff]:  $\text{reach } l2 \subseteq l2\text{-inv6}$   
 ⟨proof⟩

### 17.2.7 inv7

Form of the messages in  $\text{extr } (bad\ s)\ (ik\ s)\ (chan\ s) = \text{synth } (\text{analz } \text{generators})$ .

#### abbreviation

$\text{generators} \equiv \text{range } epubK \cup$   
 $\{Aenc\ (NonceF\ (Rb\ \$\ sk))\ (epubKF\ (Ra\$kE)) \mid Ra\ Rb. \exists A\ B.$   
 $\text{guessed}\text{-runs } Ra = (\text{role}=\text{Init},\ \text{owner}=A,\ \text{partner}=B) \wedge$   
 $\text{guessed}\text{-runs } Rb = (\text{role}=\text{Resp},\ \text{owner}=B,\ \text{partner}=A) \wedge$   
 $\text{guessed}\text{-frame } Rb\ xpkE = \text{Some } (epubKF\ (Ra\$kE))\} \cup$   
 $\{NonceF\ (R\ \$\ sk) \mid R. R \neq \text{test} \wedge R \notin \text{partners}\}$

**lemma** *analz-generators*:  $\text{analz } \text{generators} = \text{generators}$   
 ⟨proof⟩

#### definition

$l2\text{-inv7} :: l2\text{-state set}$

**where**

$$\begin{aligned} l2\text{-inv}7 &\equiv \{s. \\ &\quad \text{extr } (\text{bad } s) (\text{ik } s) (\text{chan } s) \subseteq \\ &\quad \text{synth } (\text{analz } (\text{generators})) \\ &\} \end{aligned}$$

**lemmas**  $l2\text{-inv}7I = l2\text{-inv}7\text{-def}$  [*THEN setc-def-to-intro, rule-format*]

**lemmas**  $l2\text{-inv}7E$  [*elim*] =  $l2\text{-inv}7\text{-def}$  [*THEN setc-def-to-elim, rule-format*]

**lemmas**  $l2\text{-inv}7D = l2\text{-inv}7\text{-def}$  [*THEN setc-def-to-dest, rule-format, rotated 1, simplified*]

**lemma**  $l2\text{-inv}7\text{-init}$  [*iff*]:

$$\text{init } l2 \subseteq l2\text{-inv}7$$

*<proof>*

**lemma**  $l2\text{-inv}7\text{-step1}$ :

$$\{l2\text{-inv}7\} l2\text{-step1 } Ra \ A \ B \ \{> \ l2\text{-inv}7\}$$

*<proof>*

**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>*

**lemma**  $l2\text{-inv}7\text{-step3}$ :

$$\{l2\text{-inv}7\} l2\text{-step3 } Rb \ A \ B \ K \ \{> \ l2\text{-inv}7\}$$

*<proof>*

**lemma**  $l2\text{-inv}7\text{-dy-fake-msg}$ :

$$\{l2\text{-inv}7\} l2\text{-dy-fake-msg } M \ \{> \ l2\text{-inv}7\}$$

*<proof>*

**lemma**  $l2\text{-inv}7\text{-dy-fake-chan}$ :

$$\{l2\text{-inv}7\} l2\text{-dy-fake-chan } M \ \{> \ l2\text{-inv}7\}$$

*<proof>*

**lemma**  $l2\text{-inv}7\text{-lkr-others}$ :

$$\{l2\text{-inv}7 \cap l2\text{-inv}5\} l2\text{-lkr-others } A \ \{> \ l2\text{-inv}7\}$$

*<proof>*

**lemma**  $l2\text{-inv}7\text{-lkr-after}$ :

$$\{l2\text{-inv}7 \cap l2\text{-inv}5\} l2\text{-lkr-after } A \ \{> \ l2\text{-inv}7\}$$

*<proof>*

**lemma**  $l2\text{-inv}7\text{-skr}$ :

$$\{l2\text{-inv}7 \cap l2\text{-inv}6\} l2\text{-skr } R \ K \ \{> \ l2\text{-inv}7\}$$

*<proof>*

**lemmas**  $l2\text{-inv}7\text{-trans-aux} =$

$$\begin{aligned} &l2\text{-inv}7\text{-step1 } l2\text{-inv}7\text{-step2 } l2\text{-inv}7\text{-step3} \\ &l2\text{-inv}7\text{-dy-fake-msg } l2\text{-inv}7\text{-dy-fake-chan} \\ &l2\text{-inv}7\text{-lkr-others } l2\text{-inv}7\text{-lkr-after } l2\text{-inv}7\text{-skr} \end{aligned}$$

**lemma**  $l2\text{-inv}7\text{-trans}$  [*iff*]:

$$\{l2\text{-inv}7 \cap l2\text{-inv}1 \cap l2\text{-inv}2 \cap l2\text{-inv}4 \cap l2\text{-inv}5 \cap l2\text{-inv}6\} \text{trans } l2 \ \{> \ l2\text{-inv}7\}$$

$\langle \text{proof} \rangle$

**lemma** *PO-l2-inv7* [iff]: *reach l2*  $\subseteq$  *l2-inv7*

$\langle \text{proof} \rangle$

**lemma** *l2-inv7-aux*:

$\text{NonceF } (R\$sk) \in \text{analz } (ik\ s) \implies s \in \text{l2-inv7} \implies R \neq \text{test} \wedge R \notin \text{partners}$

$\langle \text{proof} \rangle$

### 17.2.8 inv8

Form of the secrets = nonces generated by test or partners

**definition**

*l2-inv8* :: *l2-state set*

**where**

*l2-inv8*  $\equiv$   $\{s.$

$\text{secret } s \subseteq \{ \text{NonceF } (R\$sk) \mid R. R = \text{test} \vee R \in \text{partners} \}$

$\}$

**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*

$\langle \text{proof} \rangle$

**lemma** *l2-inv8-trans* [iff]:

$\{ \text{l2-inv8} \cap \text{l2-inv1} \cap \text{l2-inv3} \}$  *trans l2*  $\{ > \text{l2-inv8} \}$

$\langle \text{proof} \rangle$

**lemma** *PO-l2-inv8* [iff]: *reach l2*  $\subseteq$  *l2-inv8*

$\langle \text{proof} \rangle$

## 17.3 Refinement

mediator function

**definition**

*med12s* :: *l2-obs*  $\Rightarrow$  *l1-obs*

**where**

*med12s t*  $\equiv$  ( $\{$

*ik* = *ik t*,

*secret* = *secret t*,

*progress* = *progress t*,

*signals* = *signals t*

$\})$

relation between states

**definition**

*R12s* :: (*l1-state* \* *l2-state*) *set*

**where**

$$\begin{aligned}
R12s &\equiv \{(s, s') \\
&\quad s = \text{med12s } s' \\
&\}
\end{aligned}$$

**lemmas**  $R12s\text{-defs} = R12s\text{-def } \text{med12s}\text{-def}$

**lemma** *can-signal-R12* [*simp*]:

$$(s1, s2) \in R12s \implies \text{can-signal } s1 \ A \ B \longleftrightarrow \text{can-signal } s2 \ A \ B$$

*<proof>*

protocol events

**lemma** *l2-step1-refines-step1*:

$$\{R12s\} \text{ l1-step1 } Ra \ A \ B, \text{ l2-step1 } Ra \ A \ B \ \{>R12s\}$$

*<proof>*

**lemma** *l2-step2-refines-step2*:

$$\begin{aligned}
&\{R12s \cap UNIV \times (\text{l2-inv1} \cap \text{l2-inv2} \cap \text{l2-inv7})\} \\
&\quad \text{l1-step2 } Rb \ A \ B \ KE, \text{ l2-step2 } Rb \ A \ B \ KE \\
&\quad \{>R12s\}
\end{aligned}$$

*<proof>*

auxiliary lemma needed to prove that the nonce received by the test in step 3 comes from a partner

**lemma** *l2-step3-partners*:

$$\begin{aligned}
&\text{guessed-runs } \text{test} = (\text{role} = \text{Init}, \text{owner} = A, \text{partner} = B) \implies \\
&\quad \text{guessed-frame } \text{test } xsk = \text{Some } (\text{NonceF } (Rb\$sk)) \implies \\
&\quad \text{guessed-runs } Rb = (\text{role} = \text{Resp}, \text{owner} = B, \text{partner} = A) \implies \\
&\quad \text{guessed-frame } Rb \ xpkE = \text{Some } (\text{epubKF } (\text{test } \$ \text{kE})) \implies \\
&\quad Rb \in \text{partners}
\end{aligned}$$

*<proof>*

**lemma** *l2-step3-refines-step3*:

$$\begin{aligned}
&\{R12s \cap UNIV \times (\text{l2-inv1} \cap \text{l2-inv3} \cap \text{l2-inv7})\} \\
&\quad \text{l1-step3 } Ra \ A \ B \ K, \text{ l2-step3 } Ra \ A \ B \ K \\
&\quad \{>R12s\}
\end{aligned}$$

*<proof>*

attacker events

**lemma** *l2-dy-fake-chan-refines-skip*:

$$\{R12s\} \text{ Id}, \text{ l2-dy-fake-chan } M \ \{>R12s\}$$

*<proof>*

**lemma** *l2-dy-fake-msg-refines-learn*:

$$\{R12s \cap UNIV \times \text{l2-inv7} \cap UNIV \times \text{l2-inv8}\} \text{ l1-learn } m, \text{ l2-dy-fake-msg } m \ \{>R12s\}$$

*<proof>*

compromising events

**lemma** *l2-lkr-others-refines-skip*:

$$\{R12s\} \text{ Id}, \text{ l2-lkr-others } A \ \{>R12s\}$$

$\langle \text{proof} \rangle$

**lemma** *l2-lkr-after-refines-skip*:

$\{R12s\} \text{ Id, l2-lkr-after } A \{>R12s\}$

$\langle \text{proof} \rangle$

**lemma** *l2-skr-refines-learn*:

$\{R12s \cap UNIV \times l2\text{-inv}7 \cap UNIV \times l2\text{-inv}6 \cap UNIV \times l2\text{-inv}8\} \text{ l1-learn } K, \text{ l2-skr } R \text{ } K \{>R12s\}$

$\langle \text{proof} \rangle$

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]:

$\text{init } l2 \subseteq R12s \text{ “ (init } l1)$

$\langle \text{proof} \rangle$

**lemma** *l2-refines-trans-l1* [iff]:

$\{R12s \cap (UNIV \times (l2\text{-inv}1 \cap l2\text{-inv}2 \cap l2\text{-inv}3 \cap l2\text{-inv}6 \cap l2\text{-inv}7 \cap l2\text{-inv}8))\} \text{ trans } l1, \text{ trans } l2$   
 $\{> R12s\}$

$\langle \text{proof} \rangle$

**lemma** *PO-obs-consistent-R12s* [iff]:

$\text{obs-consistent } R12s \text{ med}12s \text{ } l1 \text{ } l2$

$\langle \text{proof} \rangle$

**lemma** *l2-refines-l1* [iff]:

*refines*

$(R12s \cap$

$(\text{reach } l1 \times (l2\text{-inv}1 \cap l2\text{-inv}2 \cap l2\text{-inv}3 \cap l2\text{-inv}4 \cap l2\text{-inv}5 \cap l2\text{-inv}6 \cap l2\text{-inv}7 \cap l2\text{-inv}8)))$

$\text{med}12s \text{ } l1 \text{ } l2$

$\langle \text{proof} \rangle$

**lemma** *l2-implements-l1* [iff]:

$\text{implements } \text{med}12s \text{ } l1 \text{ } l2$

$\langle \text{proof} \rangle$

## 17.4 Derived invariants

We want to prove *l2-secretcy*:  $\text{dy-fake-msg } (bad \ s) \ (ik \ s) \ (chan \ s) \cap \text{secret } s = \{\}$  but by refinement we only get *l2-partial-secretcy*:  $\text{synth } (\text{analz } (ik \ s)) \cap \text{secret } s = \{\}$  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-secretcy* would still hold for this new state.

**definition**

*l2-partial-secretcy* :: ('a *l2-state-scheme*) *set*

**where**

*l2-partial-secretcy*  $\equiv \{s. \text{synth } (\text{analz } (ik \ s)) \cap \text{secret } s = \{\}\}$



**lemma** *l2-obs-partial-secrecy* [iff]: *oreach l2*  $\subseteq$  *l2-partial-secrecy*  
*<proof>*

**lemma** *l2-oreach-dy-fake-msg*:

$s \in \text{oreach } l2 \implies x \in \text{dy-fake-msg } (\text{bad } s) (\text{ik } s) (\text{chan } s) \implies s (\text{ik} := \text{insert } x (\text{ik } s)) \in \text{oreach } l2$   
*<proof>*

**definition**

*l2-secrecy* :: ('a *l2-state-scheme*) set

**where**

$\text{l2-secrecy} \equiv \{s. \text{dy-fake-msg } (\text{bad } s) (\text{ik } s) (\text{chan } s) \cap \text{secret } s = \{\}\}$

**lemma** *l2-obs-secrecy* [iff]: *oreach l2*  $\subseteq$  *l2-secrecy*  
*<proof>*

**lemma** *l2-secrecy* [iff]: *reach l2*  $\subseteq$  *l2-secrecy*  
*<proof>*

**abbreviation** *l2-iagreement*  $\equiv$  *l1-iagreement*

**lemma** *l2-obs-iagreement* [iff]: *oreach l2*  $\subseteq$  *l2-iagreement*  
*<proof>*

**lemma** *l2-iagreement* [iff]: *reach l2*  $\subseteq$  *l2-iagreement*  
*<proof>*

**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*

⟨proof⟩

**lemma** *analz-ik-init* [simp]: analz *ik-init* = *ik-init*

⟨proof⟩

**lemma** *abs-ik-init* [iff]: abs *ik-init* = {}

⟨proof⟩

**lemma** *payloadSet-ik-init* [iff]:  $ik\text{-init} \cap \text{payload} = \{\}$   
<proof>

**lemma** *validSet-ik-init* [iff]:  $ik\text{-init} \cap \text{valid} = \{\}$   
<proof>

**definition**

*l3-init* :: *l3-state set*

**where**

$l3\text{-init} \equiv \{$   
   $ik = ik\text{-init},$   
   $secret = \{\},$   
   $progress = \text{Map.empty},$   
   $signals = \lambda x. 0,$   
   $bad = bad\text{-init}$   
 $\}$

**lemmas** *l3-init-defs* = *l3-init-def ik-init-def*

**definition**

*l3-trans* :: *l3-trans*

**where**

$l3\text{-trans} \equiv (\bigcup m M KE Rb Ra A B K.$   
   $l3\text{-step1 } Ra A B \cup$   
   $l3\text{-step2 } Rb A B KE \cup$   
   $l3\text{-step3 } Ra A B m \cup$   
   $l3\text{-dy } M \cup$   
   $l3\text{-lkr-others } A \cup$   
   $l3\text{-lkr-after } A \cup$   
   $l3\text{-skr } Ra K \cup$   
   $Id$   
 $)$

**definition**

*l3* :: (*l3-state*, *l3-obs*) *spec* **where**

$l3 \equiv$  (  
   $init = l3\text{-init},$   
   $trans = l3\text{-trans},$   
   $obs = id$   
)

**lemmas** *l3-loc-defs* =

*l3-step1-def l3-step2-def l3-step3-def*  
*l3-def l3-init-defs l3-trans-def*  
*l3-dy-def*  
*l3-lkr-others-def l3-lkr-after-def l3-skr-def*

**lemmas** *l3-defs* = *l3-loc-defs ik-dy-def*

**lemmas** *l3-nostep-defs* = *l3-def l3-init-def l3-trans-def*

**lemma** *l3-obs-id* [*simp*]: *obs l3 = id*  
 ⟨*proof*⟩

## 18.2 Invariants

### 18.2.1 inv1: No long-term keys as message parts

**definition**

*l3-inv1* :: *l3-state set*

**where**

*l3-inv1* ≡ {*s*.  
   *parts (ik s) ∩ range LtK ⊆ ik s*  
 }

**lemmas** *l3-inv1I* = *l3-inv1-def* [*THEN setc-def-to-intro, rule-format*]

**lemmas** *l3-inv1E* [*elim*] = *l3-inv1-def* [*THEN setc-def-to-elim, rule-format*]

**lemmas** *l3-inv1D* = *l3-inv1-def* [*THEN setc-def-to-dest, rule-format*]

**lemma** *l3-inv1D'* [*dest*]:  $\llbracket LtK\ K \in parts\ (ik\ s); s \in l3-inv1 \rrbracket \implies LtK\ K \in ik\ s$   
 ⟨*proof*⟩

**lemma** *l3-inv1-init* [*iff*]:

*init l3 ⊆ l3-inv1*

⟨*proof*⟩

**lemma** *l3-inv1-trans* [*iff*]:

{*l3-inv1*} *trans l3* {> *l3-inv1*}

⟨*proof*⟩

**lemma** *PO-l3-inv1* [*iff*]:

*reach l3 ⊆ l3-inv1*

⟨*proof*⟩

### 18.2.2 inv2: *l3-state.bad s* indeed contains "bad" keys

**definition**

*l3-inv2* :: *l3-state set*

**where**

*l3-inv2* ≡ {*s*.  
   *Keys-bad (ik s) (bad s)*  
 }

**lemmas** *l3-inv2I* = *l3-inv2-def* [*THEN setc-def-to-intro, rule-format*]

**lemmas** *l3-inv2E* [*elim*] = *l3-inv2-def* [*THEN setc-def-to-elim, rule-format*]

**lemmas** *l3-inv2D* = *l3-inv2-def* [*THEN setc-def-to-dest, rule-format*]

**lemma** *l3-inv2-init* [*simp,intro!*]:

*init l3 ⊆ l3-inv2*

⟨*proof*⟩

**lemma** *l3-inv2-trans* [*simp,intro!*]:

{*l3-inv2* ∩ *l3-inv1*} *trans l3* {> *l3-inv2*}

*<proof>*

**lemma** *PO-l3-inv2* [iff]: *reach l3*  $\subseteq$  *l3-inv2*

*<proof>*

### 18.2.3 inv3

If a message can be analyzed from the intruder knowledge then it can be derived (using synth/analz) from the sets of implementation, non-implementation, and long-term key messages and the tags. That is, intermediate messages are not needed.

**definition**

*l3-inv3* :: *l3-state set*

**where**

*l3-inv3*  $\equiv$  {*s*.

*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*

*<proof>*

**declare** *domIff* [iff del]

Most of the cases in this proof are simple and very similar. The proof could probably be shortened.

**lemma** *l3-inv3-trans* [*simp,intro!*]:

{*l3-inv3*} *trans l3* {> *l3-inv3*}

*<proof>*

**lemma** *PO-l3-inv3* [iff]: *reach l3*  $\subseteq$  *l3-inv3*

*<proof>*

### 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-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*

*<proof>*

**lemma** *l3-inv4-trans* [*simp,intro!*]:

$\{l3\text{-inv4}\} \text{ trans } l3 \{> l3\text{-inv4}\}$

*<proof>*

**lemma** *PO-l3-inv4* [*simp,intro!*]: *reach*  $l3 \subseteq l3\text{-inv4}$

*<proof>*

The remaining invariants are derived from the others. They are not protocol dependent provided the previous invariants hold.

### 18.2.5 inv5

The messages that the L3 DY intruder can derive from the intruder knowledge (using *synth/analz*), are either implementations or intermediate messages or can also be derived by the L2 intruder from the set *extr* (*l3-state.bad s*) (*ik s*  $\cap$  *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*:  $l3\text{-inv2} \cap l3\text{-inv3} \subseteq l3\text{-inv5}$

*<proof>*

**lemma** *PO-l3-inv5* [*simp,intro!*]: *reach*  $l3 \subseteq l3\text{-inv5}$

*<proof>*

### 18.2.6 inv6

If the level 3 intruder can deduce a message implementing an insecure channel message, then:

- either the message is already in the intruder knowledge;
- or the message is constructed, and the payload can also be deduced by the intruder.

**definition**

*l3-inv6* :: *l3-state set*

**where**

$l3\text{-inv6} \equiv \{s. \forall A B M.$

$(\text{implInsec } A B M \in \text{synth } (\text{analz } (\text{ik } s)) \wedge M \in \text{payload}) \longrightarrow$

$(\text{implInsec } A B M \in \text{ik } s \vee M \in \text{synth } (\text{analz } (\text{ik } s)))$



}

**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!*]:

$l3\text{-inv}3 \cap l3\text{-inv}4 \subseteq l3\text{-inv}6$

*<proof>*

**lemma**  $PO\text{-}l3\text{-inv}6$  [*simp,intro!*]:  $reach\ l3 \subseteq l3\text{-inv}6$

*<proof>*

### 18.2.7 inv7

If the level 3 intruder can deduce a message implementing a confidential channel message, then:

- either the message is already in the intruder knowledge;
- or the message is constructed, and the payload can also be deduced by the intruder.

#### definition

$l3\text{-inv}7 :: l3\text{-state set}$

**where**

$l3\text{-inv}7 \equiv \{s. \forall A B M.$

$(implConfid\ A\ B\ M \in synth\ (analz\ (ik\ s)) \wedge M \in payload) \longrightarrow$

$(implConfid\ A\ B\ M \in ik\ s \vee M \in synth\ (analz\ (ik\ s)))$

$\}$

**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!*]:

$l3\text{-inv}3 \cap l3\text{-inv}4 \subseteq l3\text{-inv}7$

*<proof>*

**lemma**  $PO\text{-}l3\text{-inv}7$  [*simp,intro!*]:  $reach\ l3 \subseteq l3\text{-inv}7$

*<proof>*

### 18.2.8 inv8

If the level 3 intruder can deduce a message implementing an authentic channel message then:

- either the message is already in the intruder knowledge;
- or the message is constructed, and in this case the payload can also be deduced by the intruder, and one of the agents is bad.

#### definition

$l3\text{-inv}8 :: l3\text{-state set}$

**where**

$$\begin{aligned}
l3\text{-inv}8 &\equiv \{s. \forall A B M. \\
&\quad (\text{implAuth } A B M \in \text{synth } (\text{analz } (ik\ s)) \wedge M \in \text{payload}) \longrightarrow \\
&\quad (\text{implAuth } A B M \in ik\ s \vee (M \in \text{synth } (\text{analz } (ik\ s)) \wedge (A \in \text{bad } s \vee B \in \text{bad } s))) \\
&\}
\end{aligned}$$

**lemmas**  $l3\text{-inv}8I = l3\text{-inv}8\text{-def}$  [THEN setc-def-to-intro, rule-format]

**lemmas**  $l3\text{-inv}8E = l3\text{-inv}8\text{-def}$  [THEN setc-def-to-elim, rule-format]

**lemmas**  $l3\text{-inv}8D = l3\text{-inv}8\text{-def}$  [THEN setc-def-to-dest, rule-format]

**lemma**  $l3\text{-inv}8\text{-derived}$  [iff]:

$$l3\text{-inv}2 \cap l3\text{-inv}3 \cap l3\text{-inv}4 \subseteq l3\text{-inv}8$$

$\langle\text{proof}\rangle$

**lemma**  $PO\text{-}l3\text{-inv}8$  [iff]:  $\text{reach } l3 \subseteq l3\text{-inv}8$

$\langle\text{proof}\rangle$

### 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**

$l3\text{-inv}9 :: l3\text{-state set}$

**where**

$$\begin{aligned}
l3\text{-inv}9 &\equiv \{s. \forall A B M. \\
&\quad (\text{implSecure } A B M \in \text{synth } (\text{analz } (ik\ s)) \wedge M \in \text{payload}) \longrightarrow \\
&\quad (\text{implSecure } A B M \in ik\ s \vee (M \in \text{synth } (\text{analz } (ik\ s)) \wedge (A \in \text{bad } s \vee B \in \text{bad } s))) \\
&\}
\end{aligned}$$

**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]:

$$l3\text{-inv}2 \cap l3\text{-inv}3 \cap l3\text{-inv}4 \subseteq l3\text{-inv}9$$

$\langle\text{proof}\rangle$

**lemma**  $PO\text{-}l3\text{-inv}9$  [iff]:  $\text{reach } l3 \subseteq l3\text{-inv}9$

$\langle\text{proof}\rangle$

## 18.3 Refinement

mediator function

**definition**

$med23s :: l3\text{-obs} \Rightarrow l2\text{-obs}$

**where**

$med23s\ t \equiv ()$

$ik = ik\ t \cap\ payload,$   
 $secret = secret\ t,$   
 $progress = progress\ t,$   
 $signals = signals\ t,$   
 $chan = abs\ (ik\ t),$   
 $bad = bad\ t$   
 $\rangle$

relation between states

**definition**

$R23s :: (l2\text{-state} * l3\text{-state})\ set$

**where**

$R23s \equiv \{(s, s').$   
 $s = med23s\ s'$   
 $\}$

**lemmas**  $R23s\text{-defs} = R23s\text{-def}\ med23s\text{-def}$

**lemma**  $R23sI$ :

$\llbracket ik\ s = ik\ t \cap\ payload; secret\ s = secret\ t; progress\ s = progress\ t; signals\ s = signals\ t;$   
 $chan\ s = abs\ (ik\ t); l2\text{-state}.bad\ s = bad\ t \rrbracket$   
 $\implies (s, t) \in R23s$   
 $\langle proof \rangle$

**lemma**  $R23sD$ :

$(s, t) \in R23s \implies$   
 $ik\ s = ik\ t \cap\ payload \wedge secret\ s = secret\ t \wedge progress\ s = progress\ t \wedge signals\ s = signals\ t \wedge$   
 $chan\ s = abs\ (ik\ t) \wedge l2\text{-state}.bad\ s = bad\ t$   
 $\langle proof \rangle$

**lemma**  $R23sE$  [elim]:

$\llbracket (s, t) \in R23s;$   
 $\llbracket ik\ s = ik\ t \cap\ payload; secret\ s = secret\ t; progress\ s = progress\ t; signals\ s = signals\ t;$   
 $chan\ s = abs\ (ik\ t); l2\text{-state}.bad\ s = bad\ t \rrbracket \implies P \rrbracket$   
 $\implies P$   
 $\langle proof \rangle$

**lemma**  $can\text{-signal}\text{-}R23$  [simp]:

$(s2, s3) \in R23s \implies$   
 $can\text{-signal}\ s2\ A\ B \longleftrightarrow can\text{-signal}\ s3\ A\ B$   
 $\langle proof \rangle$

### 18.3.1 Protocol events

**lemma**  $l3\text{-step1}\text{-refines}\text{-step1}$ :

$\{R23s\}\ l2\text{-step1}\ Ra\ A\ B, l3\text{-step1}\ Ra\ A\ B\ \{>R23s\}$   
 $\langle proof \rangle$

**lemma**  $l3\text{-step2}\text{-refines}\text{-step2}$ :

$\{R23s\}\ l2\text{-step2}\ Rb\ A\ B\ KE, l3\text{-step2}\ Rb\ A\ B\ KE\ \{>R23s\}$   
 $\langle proof \rangle$

**lemma** *l3-step3-refines-step3*:

$\{R23s\}$  *l2-step3 Ra A B K*, *l3-step3 Ra A B K*  $\{>R23s\}$   
 $\langle proof \rangle$

### 18.3.2 Intruder events

**lemma** *l3-dy-payload-refines-dy-fake-msg*:

$M \in payload \implies$   
 $\{R23s \cap UNIV \times l3-inv5\}$  *l2-dy-fake-msg M*, *l3-dy M*  $\{>R23s\}$   
 $\langle proof \rangle$

**lemma** *l3-dy-valid-refines-dy-fake-chan*:

$\llbracket M \in valid; M' \in abs \{M\} \rrbracket \implies$   
 $\{R23s \cap UNIV \times (l3-inv5 \cap l3-inv6 \cap l3-inv7 \cap l3-inv8 \cap l3-inv9)\}$   
*l2-dy-fake-chan M'*, *l3-dy M*  
 $\{>R23s\}$   
 $\langle proof \rangle$

**lemma** *l3-dy-valid-refines-dy-fake-chan-Un*:

$M \in valid \implies$   
 $\{R23s \cap UNIV \times (l3-inv5 \cap l3-inv6 \cap l3-inv7 \cap l3-inv8 \cap l3-inv9)\}$   
 $\bigcup M'. l2-dy-fake-chan M', l3-dy M$   
 $\{>R23s\}$   
 $\langle proof \rangle$

**lemma** *l3-dy-isLtKey-refines-skip*:

$\{R23s\}$  *Id*, *l3-dy (LtK ltk)*  $\{>R23s\}$   
 $\langle proof \rangle$

**lemma** *l3-dy-others-refines-skip*:

$\llbracket M \notin range LtK; M \notin valid; M \notin payload \rrbracket \implies$   
 $\{R23s\}$  *Id*, *l3-dy M*  $\{>R23s\}$   
 $\langle proof \rangle$

**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*  
 $\{>R23s\}$   
 $\langle proof \rangle$

### 18.3.3 Compromise events

**lemma** *l3-lkr-others-refines-lkr-others*:

$\{R23s\}$  *l2-lkr-others A*, *l3-lkr-others A*  $\{>R23s\}$   
 $\langle proof \rangle$

**lemma** *l3-lkr-after-refines-lkr-after*:

$\{R23s\}$  *l2-lkr-after A*, *l3-lkr-after A*  $\{>R23s\}$   
 $\langle proof \rangle$

**lemma** *l3-skr-refines-skr*:  
 $\{R23s\} \text{ l2-skr } R \ K, \text{ l3-skr } R \ K \ \{>R23s\}$   
 $\langle \text{proof} \rangle$

**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]:  
 $\text{init } l3 \subseteq R23s \ \text{“} (\text{init } l2)$   
 $\langle \text{proof} \rangle$

**lemma** *l3-refines-trans-l2* [iff]:  
 $\{R23s \cap (UNIV \times (l3\text{-inv1} \cap l3\text{-inv2} \cap l3\text{-inv3} \cap l3\text{-inv4}))\} \text{ trans } l2, \text{ trans } l3 \ \{> R23s\}$   
 $\langle \text{proof} \rangle$

**lemma** *PO-obs-consistent-R23s* [iff]:  
 $\text{obs-consistent } R23s \text{ med23s } l2 \ l3$   
 $\langle \text{proof} \rangle$

**lemma** *l3-refines-l2* [iff]:  
 $\text{refines}$   
 $(R23s \cap$   
 $(\text{reach } l2 \times (l3\text{-inv1} \cap l3\text{-inv2} \cap l3\text{-inv3} \cap l3\text{-inv4})))$   
 $\text{med23s } l2 \ l3$   
 $\langle \text{proof} \rangle$

**lemma** *l3-implements-l2* [iff]:  
 $\text{implements med23s } l2 \ l3$   
 $\langle \text{proof} \rangle$

## 18.4 Derived invariants

### 18.4.1 inv10: secrets contain no implementation material

#### definition

$l3\text{-inv10} :: l3\text{-state set}$

#### where

$l3\text{-inv10} \equiv \{s.$   
 $\text{secret } s \subseteq \text{payload}$   
 $\}$

**lemmas**  $l3\text{-inv10I} = l3\text{-inv10-def} \ [THEN \ \text{setc-def-to-intro}, \ \text{rule-format}]$

**lemmas**  $l3\text{-inv10E} = l3\text{-inv10-def} \ [THEN \ \text{setc-def-to-elim}, \ \text{rule-format}]$

**lemmas**  $l3\text{-inv10D} = l3\text{-inv10-def} \ [THEN \ \text{setc-def-to-dest}, \ \text{rule-format}]$

**lemma** *l3-inv10-init* [iff]:

$init\ l3 \subseteq l3\text{-inv}10$   
 ⟨proof⟩

**lemma**  $l3\text{-inv}10\text{-trans}$  [iff]:  
 $\{l3\text{-inv}10\} \text{ trans } l3 \{> l3\text{-inv}10\}$   
 ⟨proof⟩

**lemma**  $PO\text{-}l3\text{-inv}10$  [iff]:  $reach\ l3 \subseteq l3\text{-inv}10$   
 ⟨proof⟩

**lemma**  $l3\text{-obs}\text{-inv}10$  [iff]:  $oreach\ l3 \subseteq l3\text{-inv}10$   
 ⟨proof⟩

### 18.4.2 Partial secrecy

We want to prove  $l3\text{-secrecy}$ , ie  $synth\ (analz\ (ik\ s)) \cap secret\ s = \{\}$ , but by refinement we only get  $l3\text{-partial}\text{-secrecy}$ :  $dy\text{-fake}\text{-msg}\ (l3\text{-state}\text{-bad}\ s)\ (payloadSet\ (ik\ s))\ (local.\text{abs}\ (ik\ s)) \cap secret\ s = \{\}$ . This is fine if secrets contain no implementation material. Then, by  $inv5$ , a message in  $synth\ (analz\ (ik\ s))$  is in  $dy\text{-fake}\text{-msg}\ (l3\text{-state}\text{-bad}\ s)\ (payloadSet\ (ik\ s))\ (local.\text{abs}\ (ik\ s)) \cup -\text{payload}$ , and  $l3\text{-partial}\text{-secrecy}$  proves it is not a secret.

#### definition

$l3\text{-partial}\text{-secrecy} :: ('a\ l3\text{-state}\text{-scheme})\ set$

#### where

$l3\text{-partial}\text{-secrecy} \equiv \{s.\$   
 $dy\text{-fake}\text{-msg}\ (bad\ s)\ (ik\ s \cap payload)\ (abs\ (ik\ s)) \cap secret\ s = \{\}$   
 $\}$

**lemma**  $l3\text{-obs}\text{-partial}\text{-secrecy}$  [iff]:  $oreach\ l3 \subseteq l3\text{-partial}\text{-secrecy}$   
 ⟨proof⟩

### 18.4.3 Secrecy

#### definition

$l3\text{-secrecy} :: ('a\ l3\text{-state}\text{-scheme})\ set$

#### where

$l3\text{-secrecy} \equiv l1\text{-secrecy}$

**lemma**  $l3\text{-obs}\text{-inv}5$ :  $oreach\ l3 \subseteq l3\text{-inv}5$   
 ⟨proof⟩

**lemma**  $l3\text{-obs}\text{-secrecy}$  [iff]:  $oreach\ l3 \subseteq l3\text{-secrecy}$   
 ⟨proof⟩

**lemma**  $l3\text{-secrecy}$  [iff]:  $reach\ l3 \subseteq l3\text{-secrecy}$   
 ⟨proof⟩

### 18.4.4 Injective agreement

**abbreviation**  $l3\text{-iagreement} \equiv l1\text{-iagreement}$

**lemma**  $l3\text{-obs}\text{-iagreement}$  [iff]:  $oreach\ l3 \subseteq l3\text{-iagreement}$

*<proof>*

**lemma** *l3-agreement [iff]: reach l3  $\subseteq$  l3-agreement*  
*<proof>*

**end**  
**end**

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

```
theory pfs_l3-asymmetric
imports pfs_l3 Implem-asymmetric
begin

interpretation pfs_l3-asym: pfs_l3 implem-asym
  <proof>

end
```



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

```
theory pfs_l3-symmetric  
imports pfs_l3 Implem-symmetric  
begin  
  
interpretation pfs_l3-asm: pfs_l3 implem-sym  
<proof>  
  
end
```

## 21 Authenticated Diffie Hellman Protocol (L1)

```
theory dhlvl1
imports Runs Secrecy AuthenticationI Payloads
begin
```

```
declare option.split-asm [split]
```

### 21.1 State and Events

```
consts
```

```
  Nend :: nat
```

```
abbreviation nx :: nat where nx ≡ 2
```

```
abbreviation ny :: nat where ny ≡ 3
```

Proofs break if  $1::'a$  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 partitionning 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]:

$guessed-frame R xEnd = Some End$

$\langle proof \rangle$

**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-obs* = *l1-state*

**abbreviation**

*run-ended* :: *var set option*  $\Rightarrow$  *bool*

**where**

$run-ended r \equiv in-progress r xEnd$

**lemma** *run-ended-not-None* [elim]:

$run\text{-}ended\ R \implies R = None \implies False$   
 ⟨proof⟩

$test\text{-}ended\ s \longleftrightarrow$  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

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

— action

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

}

Protocol events.

- step 1: create  $Ra$ ,  $A$  generates  $nx$ , 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**

$l1\text{-}step1 :: rid\text{-}t \Rightarrow agent \Rightarrow agent \Rightarrow ('a\ l1\text{-}state\text{-}scheme * 'a\ l1\text{-}state\text{-}scheme)\ set$

**where**

$l1\text{-}step1\ Ra\ A\ B \equiv \{(s, s')\}.$

— guards:

$Ra \notin dom\ (progress\ s) \wedge$

$guessed\text{-}runs\ Ra = (role=Init, owner=A, partner=B) \wedge$

— actions:

$s' = s\ ($

$$\begin{array}{l} \text{progress} := (\text{progress } s)(Ra \mapsto \{xnx, xgnx\}) \\ \quad \Downarrow \\ \} \end{array}$$

**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:

$\text{guessed-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=\text{B}, \text{partner}=\text{A}) \wedge$

$Rb \notin \text{dom } (\text{progress } s) \wedge$

$\text{guessed-frame } Rb \ xgnx = \text{Some } gnx \wedge$

$\text{guessed-frame } Rb \ xsk = \text{Some } (\text{Exp } gnx \ (\text{NonceF } (Rb\$ny))) \wedge$

— actions:

$s' = s(\text{progress} := (\text{progress } s)(Rb \mapsto \{xny, xgny, xgnx, xsk\}),$

$\text{signalsInit} := \text{if can-signal } s \ A \ B \ \text{then}$

$\text{addSignal } (\text{signalsInit } s) \ (\text{Running } A \ B \ (\text{Exp } gnx \ (\text{NonceF } (Rb\$ny))))$

$\text{else}$

$\text{signalsInit } s$

$$\begin{array}{l} \quad \Downarrow \\ \} \end{array}$$

**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:

$\text{guessed-runs } Ra = (\text{role}=\text{Init}, \text{owner}=\text{A}, \text{partner}=\text{B}) \wedge$

$\text{progress } s \ Ra = \text{Some } \{xnx, xgnx\} \wedge$

$\text{guessed-frame } Ra \ xgny = \text{Some } gny \wedge$

$\text{guessed-frame } Ra \ xsk = \text{Some } (\text{Exp } gny \ (\text{NonceF } (Ra\$nx))) \wedge$

$(\text{can-signal } s \ A \ B \longrightarrow \text{— authentication guard}$

$(\exists Rb. \text{guessed-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=\text{B}, \text{partner}=\text{A}) \wedge$

$\text{in-progressS } (\text{progress } s \ Rb) \ \{xny, xgnx, xgny, xsk\} \wedge$

$\text{guessed-frame } Rb \ xgny = \text{Some } gny \wedge$

$\text{guessed-frame } Rb \ xgnx = \text{Some } (\text{Exp } \text{Gen } (\text{NonceF } (Ra\$nx)))) \wedge$

$(Ra = \text{test} \longrightarrow \text{Exp } gny \ (\text{NonceF } (Ra\$nx)) \notin \text{synth } (\text{analz } (\text{ik } s))) \wedge$

— actions:

$s' = s(\text{progress} := (\text{progress } s)(Ra \mapsto \{xnx, xgnx, xgny, xsk, xEnd\}),$

$\text{secret} := \{x. x = \text{Exp } gny \ (\text{NonceF } (Ra\$nx)) \wedge Ra = \text{test}\} \cup \text{secret } s,$

$\text{signalsInit} := \text{if can-signal } s \ A \ B \ \text{then}$

$\text{addSignal } (\text{signalsInit } s) \ (\text{Commit } A \ B \ (\text{Exp } gny \ (\text{NonceF } (Ra\$nx))))$

$\text{else}$

$\text{signalsInit } s,$

$\text{signalsResp} := \text{if can-signal } s \ A \ B \ \text{then}$

$\text{addSignal } (\text{signalsResp } s) \ (\text{Running } A \ B \ (\text{Exp } gny \ (\text{NonceF } (Ra\$nx))))$

$\text{else}$

$\text{signalsResp } s$

$$\begin{array}{l} \quad \Downarrow \\ \} \end{array}$$

**definition**

$l1\text{-step4} :: \text{rid}\text{-}t \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow \text{msg} \Rightarrow ('a\ l1\text{-state}\text{-}scheme * 'a\ l1\text{-state}\text{-}scheme)\ \text{set}$

**where**

$l1\text{-step4}\ Rb\ A\ B\ \text{gnx} \equiv \{(s, s')\}.$

— guards:

$\text{guessed}\text{-}runs\ Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \wedge$

$\text{progress}\ s\ Rb = \text{Some}\ \{xny, xgnx, xgny, xsk\} \wedge$

$\text{guessed}\text{-}frame\ Rb\ xgnx = \text{Some}\ \text{gnx} \wedge$

$(\text{can}\text{-}signal\ s\ A\ B \longrightarrow \text{— authentication guard}$

$(\exists\ Ra.\ \text{guessed}\text{-}runs\ Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \wedge$

$\text{in}\text{-}progressS\ (\text{progress}\ s\ Ra)\ \{xnx, xgnx, xgny, xsk, xEnd\} \wedge$

$\text{guessed}\text{-}frame\ Ra\ xgnx = \text{Some}\ \text{gnx} \wedge$

$\text{guessed}\text{-}frame\ Ra\ xgny = \text{Some}\ (\text{Exp}\ \text{Gen}\ (\text{NonceF}\ (Rb\$ny)))) \wedge$

$(Rb = \text{test} \longrightarrow \text{Exp}\ \text{gnx}\ (\text{NonceF}\ (Rb\$ny)) \notin \text{synth}\ (\text{analz}\ (ik\ s))) \wedge$

— actions:

$s' = s(\text{progress} := (\text{progress}\ s)(Rb \mapsto \{xny, xgnx, xgny, xsk, xEnd\}),$

$\text{secret} := \{x.\ x = \text{Exp}\ \text{gnx}\ (\text{NonceF}\ (Rb\$ny)) \wedge Rb = \text{test}\} \cup \text{secret}\ s,$

$\text{signalsResp} := \text{if}\ \text{can}\text{-}signal\ s\ A\ B\ \text{then}$

$\text{addSignal}\ (\text{signalsResp}\ s)\ (\text{Commit}\ A\ B\ (\text{Exp}\ \text{gnx}\ (\text{NonceF}\ (Rb\$ny))))$

$\text{else}$

$\text{signalsResp}\ s$

$\})$

$\}$

Specification.

**definition**

$l1\text{-init} :: l1\text{-state}\ \text{set}$

**where**

$l1\text{-init} \equiv \{(\{$

$ik = \{\},$

$\text{secret} = \{\},$

$\text{progress} = \text{Map.empty},$

$\text{signalsInit} = \lambda x.\ 0,$

$\text{signalsResp} = \lambda x.\ 0$

$\})\ \}$

**definition**

$l1\text{-trans} :: ('a\ l1\text{-state}\text{-}scheme * 'a\ l1\text{-state}\text{-}scheme)\ \text{set}\ \text{where}$

$l1\text{-trans} \equiv (\bigcup m\ Ra\ Rb\ A\ B\ x.$

$l1\text{-step1}\ Ra\ A\ B \cup$

$l1\text{-step2}\ Rb\ A\ B\ x \cup$

$l1\text{-step3}\ Ra\ A\ B\ x \cup$

$l1\text{-step4}\ Rb\ A\ B\ x \cup$

$l1\text{-learn}\ m \cup$

$\text{Id}$

$)$

**definition**

$l1 :: (l1\text{-state}, l1\text{-obs})\ \text{spec}\ \text{where}$

$l1 \equiv ()$

$init = l1-init,$   
 $trans = l1-trans,$   
 $obs = id$   
 $\rangle$

**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*]:  $obs\ l1 = id$   
 $\langle proof \rangle$

**lemma**  $run-ended-trans$ :  
 $run-ended\ (progress\ s\ R) \implies$   
 $(s, s') \in trans\ l1 \implies$   
 $run-ended\ (progress\ s'\ R)$   
 $\langle proof \rangle$

**lemma**  $can-signal-trans$ :  
 $can-signal\ s'\ A\ B \implies$   
 $(s, s') \in trans\ l1 \implies$   
 $can-signal\ s\ A\ B$   
 $\langle proof \rangle$

## 21.2 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**  $l1-learn-refines-learn$ :  
 $\{R01s\}\ s0-learn\ m, l1-learn\ m \{>R01s\}$   
 $\langle proof \rangle$

Protocol events.

**lemma**  $l1-step1-refines-skip$ :

$\{R01s\} Id, l1\text{-step1 } Ra A B \{>R01s\}$   
 $\langle proof \rangle$

**lemma** *l1-step2-refines-skip*:  
 $\{R01s\} Id, l1\text{-step2 } Rb A B gnx \{>R01s\}$   
 $\langle proof \rangle$

**lemma** *l1-step3-refines-add-secret-skip*:  
 $\{R01s\} s0\text{-add-secret } (Exp gny (NonceF (Ra\$nx))) \cup Id, l1\text{-step3 } Ra A B gny \{>R01s\}$   
 $\langle proof \rangle$

**lemma** *l1-step4-refines-add-secret-skip*:  
 $\{R01s\} s0\text{-add-secret } (Exp gnx (NonceF (Rb\$ny))) \cup Id, l1\text{-step4 } Rb A B gnx \{>R01s\}$   
 $\langle proof \rangle$

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)*  
 $\langle proof \rangle$

**lemma** *l1-refines-trans-s0 [iff]*:  
 $\{R01s\} trans s0, trans l1 \{> R01s\}$   
 $\langle proof \rangle$

**lemma** *obs-consistent-med01x [iff]*:  
*obs-consistent R01s med01s s0 l1*  
 $\langle proof \rangle$

Refinement result.

**lemma** *l1s-refines-s0 [iff]*:  
*refines*  
*R01s*  
*med01s s0 l1*  
 $\langle proof \rangle$

**lemma** *l1-implements-s0 [iff]*: *implements med01s s0 l1*  
 $\langle proof \rangle$

### 21.3 Derived invariants: secrecy

**abbreviation** *l1-secrecy  $\equiv$  s0-secrecy*

**lemma** *l1-obs-secrecy [iff]*: *oreach l1  $\subseteq$  l1-secrecy*  
 $\langle proof \rangle$



**lemma** *l1-secrecy* [iff]: *reach l1*  $\subseteq$  *l1-secrecy*  
 ⟨proof⟩

## 21.4 Invariants: *Init* authenticates *Resp*

### 21.4.1 inv1

If an initiator commit signal exists for  $(g^{ny})Ra \$ nx$  then *Ra* is *Init*, has passed step 3, and has  $(g \hat{ny}) \wedge (Ra \$ nx)$  as the key in its frame.

**definition**

*l1-inv1* :: *l1-state set*

**where**

*l1-inv1*  $\equiv$   $\{s. \forall Ra A B gny.$   
 $signalsInit\ s\ (Commit\ A\ B\ (Exp\ gny\ (NonceF\ (Ra\ \$\ nx)))) > 0 \longrightarrow$   
 $guessed-runs\ Ra = (\text{role}=Init, \text{owner}=A, \text{partner}=B) \wedge$   
 $progress\ s\ Ra = Some\ \{xnx, xgnx, xgny, xsk, xEnd\} \wedge$   
 $guessed-frame\ Ra\ xsk = Some\ (Exp\ gny\ (NonceF\ (Ra\ \$\ nx)))$   
 $\}$

**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*

⟨proof⟩

**lemma** *l1-inv1-trans* [iff]:

$\{l1-inv1\}$  *trans* *l1*  $\{> l1-inv1\}$

⟨proof⟩

**lemma** *PO-l1-inv1* [iff]: *reach l1*  $\subseteq$  *l1-inv1*

⟨proof⟩

### 21.4.2 inv2

If a *Resp* run *Rb* has passed step 2 then (if possible) an initiator running signal has been emitted.

**definition**

*l1-inv2* :: *l1-state set*

**where**

*l1-inv2*  $\equiv$   $\{s. \forall gnx A B Rb.$   
 $guessed-runs\ Rb = (\text{role}=Resp, \text{owner}=B, \text{partner}=A) \longrightarrow$   
 $in-progressS\ (progress\ s\ Rb)\ \{xny, xgnx, xgny, xsk\} \longrightarrow$   
 $guessed-frame\ Rb\ xgnx = Some\ gnx \longrightarrow$   
 $can-signal\ s\ A\ B \longrightarrow$   
 $signalsInit\ s\ (Running\ A\ B\ (Exp\ gnx\ (NonceF\ (Rb\ \$\ ny)))) > 0$   
 $\}$

**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**  $l1\text{-inv}2D = l1\text{-inv}2\text{-def}$  [THEN setc-def-to-dest, rule-format, rotated 1, simplified]

**lemma**  $l1\text{-inv}2\text{-init}$  [iff]:

$init\ l1 \subseteq l1\text{-inv}2$

<proof>

**lemma**  $l1\text{-inv}2\text{-trans}$  [iff]:

$\{l1\text{-inv}2\}\ trans\ l1\ \{>\ l1\text{-inv}2\}$

<proof>

**lemma**  $PO\text{-}l1\text{-inv}2$  [iff]:  $reach\ l1 \subseteq l1\text{-inv}2$

<proof>

### 21.4.3 inv3 (derived)

If an *Init* run before step 3 and a *Resp* run after step 2 both know the same half-keys (more or less), then the number of *Init* running signals for the key is strictly greater than the number of *Init* commit signals. (actually, there are 0 commit and 1 running).

**definition**

$l1\text{-inv}3 :: l1\text{-state\ set}$

**where**

$l1\text{-inv}3 \equiv \{s. \forall\ A\ B\ Rb\ Ra\ gny.$

$guessed\text{-runs}\ Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \longrightarrow$

$in\text{-progress}S\ (\text{progress}\ s\ Rb)\ \{xny, xgnx, xgny, xsk\} \longrightarrow$

$guessed\text{-frame}\ Rb\ xgny = \text{Some}\ gny \longrightarrow$

$guessed\text{-frame}\ Rb\ xgnx = \text{Some}\ (\text{Exp}\ \text{Gen}\ (\text{Nonce}F\ (Ra\$nx))) \longrightarrow$

$guessed\text{-runs}\ Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \longrightarrow$

$\text{progress}\ s\ Ra = \text{Some}\ \{xnx, xgnx\} \longrightarrow$

$\text{can}\text{-signal}\ s\ A\ B \longrightarrow$

$\text{signalsInit}\ s\ (\text{Commit}\ A\ B\ (\text{Exp}\ gny\ (\text{Nonce}F\ (Ra\$nx))))$

$<\ \text{signalsInit}\ s\ (\text{Running}\ A\ B\ (\text{Exp}\ gny\ (\text{Nonce}F\ (Ra\$nx))))$

$\}$

**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**  $l1\text{-inv}3\text{-derived}$ :  $l1\text{-inv}1 \cap l1\text{-inv}2 \subseteq l1\text{-inv}3$

<proof>

## 21.5 Invariants: *Resp* authenticates *Init*

### 21.5.1 inv4

If a *Resp* commit signal exists for  $(g^{nx})Rb\ \$\ ny$  then *Rb* is *Resp*, has finished its run, and has  $(g^{nx})Rb\ \$\ ny$  as the key in its frame.

**definition**

$l1\text{-inv}4 :: l1\text{-state\ set}$

**where**

$l1\text{-inv}4 \equiv \{s. \forall\ Rb\ A\ B\ gnx.$

$$\begin{aligned} & \text{signalsResp } s \text{ (Commit } A \ B \ (\text{Exp } gnx \ (\text{NonceF } (Rb\$ny)))) > 0 \longrightarrow \\ & \text{guessed-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \wedge \\ & \text{progress } s \ Rb = \text{Some } \{xny, xgnx, xgny, xsk, xEnd\} \wedge \\ & \text{guessed-frame } Rb \ xgnx = \text{Some } gnx \\ & \} \end{aligned}$$

**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]:

$\text{init } l1 \subseteq l1\text{-inv}4$

$\langle \text{proof} \rangle$

**declare**  $\text{domIff}$  [iff]

**lemma**  $l1\text{-inv}4\text{-trans}$  [iff]:

$\{l1\text{-inv}4\} \text{ trans } l1 \ \{> \ l1\text{-inv}4\}$

$\langle \text{proof} \rangle$

**declare**  $\text{domIff}$  [iff del]

**lemma**  $PO\text{-}l1\text{-inv}4$  [iff]:  $\text{reach } l1 \subseteq l1\text{-inv}4$

$\langle \text{proof} \rangle$

### 21.5.2 inv5

If an *Init* run  $Ra$  has passed step3 then (if possible) a *Resp* running signal has been emitted.

**definition**

$l1\text{-inv}5 :: l1\text{-state set}$

**where**

$l1\text{-inv}5 \equiv \{s. \forall \ gny \ A \ B \ Ra.$

$\text{guessed-runs } Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \longrightarrow$

$\text{in-progressS } (\text{progress } s \ Ra) \ \{xnx, xgnx, xgny, xsk, xEnd\} \longrightarrow$

$\text{guessed-frame } Ra \ xgny = \text{Some } gny \longrightarrow$

$\text{can-signal } s \ A \ B \longrightarrow$

$\text{signalsResp } s \ (\text{Running } A \ B \ (\text{Exp } gny \ (\text{NonceF } (Ra\$nx)))) > 0$

$\}$

**lemmas**  $l1\text{-inv}5I = l1\text{-inv}5\text{-def}$  [THEN setc-def-to-intro, rule-format]

**lemmas**  $l1\text{-inv}5E$  [elim] =  $l1\text{-inv}5\text{-def}$  [THEN setc-def-to-elim, rule-format]

**lemmas**  $l1\text{-inv}5D = l1\text{-inv}5\text{-def}$  [THEN setc-def-to-dest, rule-format, rotated 1, simplified]

**lemma**  $l1\text{-inv}5\text{-init}$  [iff]:

$\text{init } l1 \subseteq l1\text{-inv}5$

$\langle \text{proof} \rangle$

**lemma**  $l1\text{-inv}5\text{-trans}$  [iff]:

$\{l1\text{-inv}5\} \text{ trans } l1 \ \{> \ l1\text{-inv}5\}$

$\langle \text{proof} \rangle$

**lemma** *PO-l1-inv5* [iff]: *reach l1*  $\subseteq$  *l1-inv5*  
 ⟨proof⟩

### 21.5.3 inv6 (derived)

If a *Resp* run before step 4 and an *Init* run after step 3 both know the same half-keys (more or less), then the number of *Resp* running signals for the key is strictly greater than the number of *Resp* commit signals. (actually, there are 0 commit and 1 running).

**definition**

*l1-inv6* :: *l1-state set*

**where**

*l1-inv6*  $\equiv$  {*s*.  $\forall$  *A B Rb Ra gnx*.  
 guessed-runs *Ra* = (*role=Init*, *owner=A*, *partner=B*)  $\longrightarrow$   
 in-progressS (*progress s Ra*) {*xnx*, *xgnx*, *xgny*, *xsk*, *xEnd*}  $\longrightarrow$   
 guessed-frame *Ra xgnx* = *Some gnx*  $\longrightarrow$   
 guessed-frame *Ra xgny* = *Some (Exp Gen (NonceF (Rb\$ny)))*  $\longrightarrow$   
 guessed-runs *Rb* = (*role=Resp*, *owner=B*, *partner=A*)  $\longrightarrow$   
 progress *s Rb* = *Some {xny, xgnx, xgny, xsk}*  $\longrightarrow$   
 can-signal *s A B*  $\longrightarrow$   
 signalsResp *s (Commit A B (Exp gnx (NonceF (Rb\$ny))))*  
 < *signalsResp s (Running A B (Exp gnx (NonceF (Rb\$ny))))*  
 }

**lemmas** *l1-inv6I* = *l1-inv6-def* [THEN *setc-def-to-intro*, *rule-format*]

**lemmas** *l1-inv6E* [elim] = *l1-inv6-def* [THEN *setc-def-to-elim*, *rule-format*]

**lemmas** *l1-inv6D* = *l1-inv6-def* [THEN *setc-def-to-dest*, *rule-format*, *rotated 1*, *simplified*]

**lemma** *l1-inv6-derived*:

*l1-inv4*  $\cap$  *l1-inv5*  $\subseteq$  *l1-inv6*

⟨proof⟩

## 21.6 Refinement: injective agreement (*Init* authenticates *Resp*)

Mediator function.

**definition**

*med01iai* :: *l1-obs*  $\Rightarrow$  *a0i-obs*

**where**

*med01iai t*  $\equiv$  (*a0n-state.signals* = *signalsInit t*)

Relation between states.

**definition**

*R01iai* :: (*a0i-state* \* *l1-state*) *set*

**where**

*R01iai*  $\equiv$  {(*s, s'*).  
*a0n-state.signals s* = *signalsInit s'*  
 }

Protocol-independent events.

**lemma** *l1-learn-refines-a0-ia-skip-i*:

{*R01iai*} *Id*, *l1-learn m* {>*R01iai*}

$\langle \text{proof} \rangle$

Protocol events.

**lemma** *l1-step1-refines-a0i-skip-i*:  
 $\{R01iai\} Id, l1\text{-step1 } Ra A B \{>R01iai\}$   
 $\langle \text{proof} \rangle$

**lemma** *l1-step2-refines-a0i-running-skip-i*:  
 $\{R01iai\} a0i\text{-running } A B (Exp\ gnx\ (NonceF\ (Rb\$ny))) \cup Id, l1\text{-step2 } Rb A B\ gnx \{>R01iai\}$   
 $\langle \text{proof} \rangle$

**lemma** *l1-step3-refines-a0i-commit-skip-i*:  
 $\{R01iai \cap (UNIV \times l1\text{-inv3})\}$   
 $a0i\text{-commit } A B (Exp\ gny\ (NonceF\ (Ra\$nx))) \cup Id,$   
 $l1\text{-step3 } Ra A B\ gny$   
 $\{>R01iai\}$   
 $\langle \text{proof} \rangle$

**lemma** *l1-step4-refines-a0i-skip-i*:  
 $\{R01iai\} Id, l1\text{-step4 } Rb A B\ gnx \{>R01iai\}$   
 $\langle \text{proof} \rangle$

Refinement proof.

**lemmas** *l1-trans-refines-a0i-trans-i* =  
*l1-learn-refines-a0-ia-skip-i*  
*l1-step1-refines-a0i-skip-i* *l1-step2-refines-a0i-running-skip-i*  
*l1-step3-refines-a0i-commit-skip-i* *l1-step4-refines-a0i-skip-i*

**lemma** *l1-refines-init-a0i-i* [iff]:  
 $init\ l1 \subseteq R01iai \text{ “ } (init\ a0i)$   
 $\langle \text{proof} \rangle$

**lemma** *l1-refines-trans-a0i-i* [iff]:  
 $\{R01iai \cap (UNIV \times (l1\text{-inv1} \cap l1\text{-inv2}))\} trans\ a0i, trans\ l1 \{> R01iai\}$   
 $\langle \text{proof} \rangle$

**lemma** *obs-consistent-med01iai* [iff]:  
 $obs\text{-consistent } R01iai\ med01iai\ a0i\ l1$   
 $\langle \text{proof} \rangle$

Refinement result.

**lemma** *l1-refines-a0i-i* [iff]:  
*refines*  
 $(R01iai \cap (reach\ a0i \times (l1\text{-inv1} \cap l1\text{-inv2})))$   
 $med01iai\ a0i\ l1$   
 $\langle \text{proof} \rangle$

**lemma** *l1-implements-a0i-i* [iff]: *implements med01iai a0i l1*  
 $\langle \text{proof} \rangle$

## 21.7 Derived invariants: injective agreement (*Init* authenticates *Resp*)

### definition

$l1\text{-iagreement-Init} :: ('a\ l1\text{-state-scheme})\ set$

### where

$l1\text{-iagreement-Init} \equiv \{s. \forall A\ B\ N.$   
 $\quad signalsInit\ s\ (Commit\ A\ B\ N) \leq signalsInit\ s\ (Running\ A\ B\ N)$   
 $\}$

**lemmas**  $l1\text{-iagreement-Init}I = l1\text{-iagreement-Init-def}$  [*THEN setc-def-to-intro, rule-format*]

**lemmas**  $l1\text{-iagreement-Init}E$  [*elim*] =  $l1\text{-iagreement-Init-def}$  [*THEN setc-def-to-elim, rule-format*]

**lemma**  $l1\text{-obs-iagreement-Init}$  [*iff*]:  $oreach\ l1 \subseteq l1\text{-iagreement-Init}$   
 $\langle proof \rangle$

**lemma**  $l1\text{-iagreement-Init}$  [*iff*]:  $reach\ l1 \subseteq l1\text{-iagreement-Init}$   
 $\langle proof \rangle$

## 21.8 Refinement: injective agreement (*Resp* authenticates *Init*)

Mediator function.

### definition

$med01iar :: l1\text{-obs} \Rightarrow a0i\text{-obs}$

### where

$med01iar\ t \equiv (a0n\text{-state}.signals = signalsResp\ t)$

Relation between states.

### definition

$R01iar :: (a0i\text{-state} * l1\text{-state})\ set$

### where

$R01iar \equiv \{(s, s').$   
 $\quad a0n\text{-state}.signals\ s = signalsResp\ s'$   
 $\}$

Protocol-independent events.

**lemma**  $l1\text{-learn-refines-a0i-skip-r}$ :  
 $\{R01iar\}\ Id, l1\text{-learn}\ m\ \{>R01iar\}$   
 $\langle proof \rangle$

Protocol events.

**lemma**  $l1\text{-step1-refines-a0i-skip-r}$ :  
 $\{R01iar\}\ Id, l1\text{-step1}\ Ra\ A\ B\ \{>R01iar\}$   
 $\langle proof \rangle$

**lemma**  $l1\text{-step2-refines-a0i-skip-r}$ :  
 $\{R01iar\}\ Id, l1\text{-step2}\ Rb\ A\ B\ gnx\ \{>R01iar\}$   
 $\langle proof \rangle$

**lemma**  $l1\text{-step3-refines-a0i-running-skip-r}$ :  
 $\{R01iar\}\ a0i\text{-running}\ A\ B\ (Exp\ gny\ (NonceF\ (Ra\$nx))) \cup Id, l1\text{-step3}\ Ra\ A\ B\ gny\ \{>R01iar\}$

$\langle \text{proof} \rangle$

**lemma** *l1-step4-refines-a0i-commit-skip-r*:

$\{R01iar \cap UNIV \times l1\text{-inv6}\}$   
 $a0i\text{-commit } A B (Exp\ gnx\ (NonceF\ (Rb\$ny))) \cup Id,$   
 $l1\text{-step4 } Rb\ A\ B\ gnx$   
 $\{>R01iar\}$

$\langle \text{proof} \rangle$

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]:

$init\ l1 \subseteq R01iar \text{ “ } (init\ a0i)$

$\langle \text{proof} \rangle$

**lemma** *l1-refines-trans-a0i-r* [iff]:

$\{R01iar \cap (UNIV \times (l1\text{-inv4} \cap l1\text{-inv5}))\} \text{ trans } a0i, \text{ trans } l1 \{> R01iar\}$

$\langle \text{proof} \rangle$

**lemma** *obs-consistent-med01iar* [iff]:

$obs\text{-consistent } R01iar\ med01iar\ a0i\ l1$

$\langle \text{proof} \rangle$

Refinement result.

**lemma** *l1-refines-a0i-r* [iff]:

$refines$   
 $(R01iar \cap (reach\ a0i \times (l1\text{-inv4} \cap l1\text{-inv5})))$   
 $med01iar\ a0i\ l1$

$\langle \text{proof} \rangle$

**lemma** *l1-implements-a0i-r* [iff]: *implements med01iar a0i l1*

$\langle \text{proof} \rangle$

## 21.9 Derived invariants: injective agreement (*Resp* authenticates *Init*)

**definition**

$l1\text{-iagreement-Resp} :: ('a\ l1\text{-state-scheme})\ set$

**where**

$l1\text{-iagreement-Resp} \equiv \{s. \forall A\ B\ N.$   
 $signalsResp\ s\ (Commit\ A\ B\ N) \leq signalsResp\ s\ (Running\ A\ B\ N)$   
 $\}$

**lemmas** *l1-iagreement-RespI* = *l1-iagreement-Resp-def* [THEN *setc-def-to-intro*, *rule-format*]

**lemmas** *l1-iagreement-RespE* [elim] = *l1-iagreement-Resp-def* [THEN *setc-def-to-elim*, *rule-format*]

**lemma** *l1-obs-iagreement-Resp* [iff]: *oreach l1  $\subseteq$  l1-iagreement-Resp*

*<proof>*

**lemma** *l1-agreement-Resp [iff]: reach l1  $\subseteq$  l1-agreement-Resp*  
*<proof>*

**end**



## 22 Authenticated Diffie-Hellman Protocol (L2)

```
theory dhlvl2
imports dhlvl1 Channels
begin
```

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

### 22.1 State and Events

Initial compromise.

```
consts
  bad-init :: agent set
```

```
specification (bad-init)
  bad-init-spec: test-owner  $\notin$  bad-init  $\wedge$  test-partner  $\notin$  bad-init
  <proof>
```

Level 2 state.

```
record l2-state =
  l1-state +
  chan :: chan set
  bad :: agent set
```

```
type-synonym l2-obs = l2-state
```

```
type-synonym
  l2-pred = l2-state set
```

```
type-synonym
  l2-trans = (l2-state  $\times$  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*

$\langle$ *proof* $\rangle$

**lemma** *role-comp-inv-eq*:

*y* = *role-comp* *x*  $\longleftrightarrow$  *x* = *role-comp* *y*

$\langle$ *proof* $\rangle$

**definition**

*partners* :: *rid-t* *set*

**where**

*partners*  $\equiv \{R. \textit{partner-runs } \textit{test } R\}$

**lemma** *test-not-partner* [*simp*]:

*test*  $\notin$  *partners*

$\langle$ *proof* $\rangle$

**lemma** *matching-symmetric*:

*matching* *sigma* *sigma'*  $\Longrightarrow$  *matching* *sigma'* *sigma*

$\langle$ *proof* $\rangle$

**lemma** *partner-symmetric*:

*partner-runs* *R* *R'*  $\Longrightarrow$  *partner-runs* *R'* *R*

$\langle$ *proof* $\rangle$

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⟩

**lemma** *partner-test*:

$R \in partners \implies partner\text{-runs } R R' \implies R' = test$   
 ⟨proof⟩

Compromising events.

**definition**

$l2\text{-lkr-others} :: agent \Rightarrow l2\text{-trans}$

**where**

$l2\text{-lkr-others } A \equiv \{(s, s')\}.$   
 — guards  
 $A \neq test\text{-owner} \wedge$   
 $A \neq test\text{-partner} \wedge$   
 — actions  
 $s' = s(\text{bad} := \{A\} \cup \text{bad } s)$   
 }

**definition**

$l2\text{-lkr-actor} :: agent \Rightarrow l2\text{-trans}$

**where**

$l2\text{-lkr-actor } A \equiv \{(s, s')\}.$   
 — guards  
 $A = test\text{-owner} \wedge$   
 $A \neq test\text{-partner} \wedge$   
 — actions  
 $s' = s(\text{bad} := \{A\} \cup \text{bad } s)$   
 }

**definition**

$l2\text{-lkr-after} :: agent \Rightarrow l2\text{-trans}$

**where**

$l2\text{-lkr-after } A \equiv \{(s, s')\}.$   
 — guards  
 $test\text{-ended } s \wedge$   
 — actions  
 $s' = s(\text{bad} := \{A\} \cup \text{bad } s)$   
 }

**definition**

$l2\text{-skr} :: rid\text{-t} \Rightarrow msg \Rightarrow l2\text{-trans}$

**where**

$l2\text{-skr } R K \equiv \{(s, s')\}.$   
 — guards  
 $R \neq test \wedge R \notin partners \wedge$   
 $in\text{-progress } (progress\ s\ R)\ xsk \wedge$   
 $guessed\text{-frame } R\ xsk = Some\ K \wedge$   
 — actions  
 $s' = s(\text{ik} := \{K\} \cup \text{ik } s)$   
 }

Protocol events:

- step 1: create  $Ra$ ,  $A$  generates  $nx$ , computes and insecurely sends  $g^{nx}$
- step 2: create  $Rb$ ,  $B$  receives  $g^{nx}$  insecurely, generates  $ny$ , computes  $g^{ny}$ , authentically sends  $(g^{ny}, g^{nx})$ , computes  $g^{nx} * ny$ , emits a running signal for  $Init$ ,  $g^{nx} * ny$
- step 3:  $A$  receives  $g^{ny}$  and  $g^{nx}$  authentically, sends  $(g^{nx}, g^{ny})$  authentically, computes  $g^{ny} * nx$ , emits a commit signal for  $Init$ ,  $g^{ny} * nx$ , a running signal for  $Resp$ ,  $g^{ny} * nx$ , declares the secret  $g^{ny} * nx$
- step 4:  $B$  receives  $g^{nx}$  and  $g^{ny}$  authentically, emits a commit signal for  $Resp$ ,  $g^{nx} * ny$ , declares the secret  $g^{nx} * ny$

**definition**

$l2\text{-step1} :: rid\text{-}t \Rightarrow agent \Rightarrow agent \Rightarrow l2\text{-}trans$

**where**

$l2\text{-step1 } Ra A B \equiv \{(s, s')\}.$

— guards:

$Ra \notin dom (progress\ s) \wedge$

$guessed\text{-}runs\ Ra = \langle role=Init, owner=A, partner=B \rangle \wedge$

— actions:

$s' = s \langle$

$progress := (progress\ s)(Ra \mapsto \{xnx, xgnx\}),$

$chan := \{Insec\ A\ B\ (Exp\ Gen\ (NonceF\ (Ra\$nx)))\} \cup (chan\ s)$

$\rangle$

$\}$

**definition**

$l2\text{-step2} :: rid\text{-}t \Rightarrow agent \Rightarrow agent \Rightarrow msg \Rightarrow l2\text{-}trans$

**where**

$l2\text{-step2 } Rb A B gnx \equiv \{(s, s')\}.$

— guards:

$guessed\text{-}runs\ Rb = \langle role=Resp, owner=B, partner=A \rangle \wedge$

$Rb \notin dom (progress\ s) \wedge$

$guessed\text{-}frame\ Rb\ xgnx = Some\ gnx \wedge$

$guessed\text{-}frame\ Rb\ xsk = Some\ (Exp\ gnx\ (NonceF\ (Rb\$ny))) \wedge$

$Insec\ A\ B\ gnx \in chan\ s \wedge$

— actions:

$s' = s \langle progress := (progress\ s)(Rb \mapsto \{xny, xgny, xgnx, xsk\}),$

$chan := \{Auth\ B\ A\ \langle Number\ 0, Exp\ Gen\ (NonceF\ (Rb\$ny)), gnx \rangle\} \cup (chan\ s),$

$signalsInit := if\ can\text{-}signal\ s\ A\ B\ then$

$addSignal\ (signalsInit\ s)\ (Running\ A\ B\ (Exp\ gnx\ (NonceF\ (Rb\$ny))))$

$else$

$signalsInit\ s$

$\rangle$

$\}$

**definition**

$l2\text{-step3} :: rid\text{-}t \Rightarrow agent \Rightarrow agent \Rightarrow msg \Rightarrow l2\text{-}trans$

**where**

$l2\text{-step3 } Ra A B gny \equiv \{(s, s')\}.$

— guards:

```

guessed-runs Ra = ( $\text{role}=\text{Init}$ ,  $\text{owner}=\text{A}$ ,  $\text{partner}=\text{B}$ )  $\wedge$ 
progress s Ra =  $\text{Some}$  { $xn_x$ ,  $xgn_x$ }  $\wedge$ 
guessed-frame Ra  $xgn_y$  =  $\text{Some}$   $gn_y$   $\wedge$ 
guessed-frame Ra  $xsk$  =  $\text{Some}$  ( $\text{Exp } gn_y$  ( $\text{NonceF}$  ( $\text{Ra}\$nx$ )))  $\wedge$ 
Auth B A ( $\text{Number } 0$ ,  $gn_y$ ,  $\text{Exp Gen}$  ( $\text{NonceF}$  ( $\text{Ra}\$nx$ )))  $\in$   $\text{chan } s$   $\wedge$ 
— actions:
s' = s( $\text{progress} :=$  ( $\text{progress } s$ )( $\text{Ra} \mapsto$  { $xn_x$ ,  $xgn_x$ ,  $xgn_y$ ,  $xsk$ ,  $xEnd$ }),
    $\text{chan} :=$  { $\text{Auth } A B$  ( $\text{Number } 1$ ,  $\text{Exp Gen}$  ( $\text{NonceF}$  ( $\text{Ra}\$nx$ )),  $gn_y$ )}  $\cup$   $\text{chan } s$ ,
    $\text{secret} :=$  { $x. x = \text{Exp } gn_y$  ( $\text{NonceF}$  ( $\text{Ra}\$nx$ ))  $\wedge$   $\text{Ra} = \text{test}$ }  $\cup$   $\text{secret } s$ ,
    $\text{signalsInit} :=$   $\text{if can-signal } s A B$  then
        $\text{addSignal}$  ( $\text{signalsInit } s$ ) ( $\text{Commit } A B$  ( $\text{Exp } gn_y$  ( $\text{NonceF}$  ( $\text{Ra}\$nx$ ))))
   else
        $\text{signalsInit } s$ ,
    $\text{signalsResp} :=$   $\text{if can-signal } s A B$  then
        $\text{addSignal}$  ( $\text{signalsResp } s$ ) ( $\text{Running } A B$  ( $\text{Exp } gn_y$  ( $\text{NonceF}$  ( $\text{Ra}\$nx$ ))))
   else
        $\text{signalsResp } s$ 
)
}

```

### definition

$l2\text{-step}_4 :: \text{rid-t} \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow \text{msg} \Rightarrow l2\text{-trans}$

### where

$l2\text{-step}_4 Rb A B gn_x \equiv \{(s, s')\}$ .

— guards:

$\text{guessed-runs } Rb = (\text{role}=\text{Resp}$ ,  $\text{owner}=\text{B}$ ,  $\text{partner}=\text{A}) \wedge$

$\text{progress } s Rb = \text{Some}$  { $xn_y$ ,  $xgn_x$ ,  $xgn_y$ ,  $xsk$ }  $\wedge$

$\text{guessed-frame } Rb xgn_x = \text{Some } gn_x \wedge$

$\text{Auth } A B$  ( $\text{Number } 1$ ,  $gn_x$ ,  $\text{Exp Gen}$  ( $\text{NonceF}$  ( $\text{Rb}\$ny$ )))  $\in$   $\text{chan } s$   $\wedge$

— actions:

$s' = s(\text{progress} :=$  ( $\text{progress } s$ )( $\text{Rb} \mapsto$  { $xn_y$ ,  $xgn_x$ ,  $xgn_y$ ,  $xsk$ ,  $xEnd$ }),

$\text{secret} :=$  { $x. x = \text{Exp } gn_x$  ( $\text{NonceF}$  ( $\text{Rb}\$ny$ ))  $\wedge$   $\text{Rb} = \text{test}$ }  $\cup$   $\text{secret } s$ ,

$\text{signalsResp} :=$   $\text{if can-signal } s A B$  then

$\text{addSignal}$  ( $\text{signalsResp } s$ ) ( $\text{Commit } A B$  ( $\text{Exp } gn_x$  ( $\text{NonceF}$  ( $\text{Rb}\$ny$ ))))

else

$\text{signalsResp } s$

)

Specification.

### definition

$l2\text{-init} :: l2\text{-state set}$

### where

$l2\text{-init} \equiv \{$

$ik = \{\},$

$\text{secret} = \{\},$

$\text{progress} = \text{Map.empty},$

$\text{signalsInit} = \lambda x. 0,$

$\text{signalsResp} = \lambda x. 0,$

$\text{chan} = \{\},$

$\text{bad} = \text{bad-init}$

)}

**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**

$l2 :: (l2\text{-state}, l2\text{-obs})$  *spec* **where**  
 $l2 \equiv ()$   
   $init = l2\text{-init},$   
   $trans = l2\text{-trans},$   
   $obs = id$   
)

**lemmas**  $l2\text{-loc-defs} =$

$l2\text{-step1-def } l2\text{-step2-def } l2\text{-step3-def } l2\text{-step4-def}$   
 $l2\text{-def } l2\text{-init-def } l2\text{-trans-def}$   
 $l2\text{-dy-fake-chan-def } l2\text{-dy-fake-msg-def}$   
 $l2\text{-lkr-after-def } l2\text{-lkr-others-def } l2\text{-skr-def}$

**lemmas**  $l2\text{-defs} = l2\text{-loc-defs ik-dy-def}$

**lemmas**  $l2\text{-nostep-defs} = l2\text{-def } l2\text{-init-def } l2\text{-trans-def}$

**lemma**  $l2\text{-obs-id}$  [*simp*]:  $obs\ l2 = id$

$\langle proof \rangle$

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**  $l2\text{-run-ended-trans}$ :

$run\text{-ended} (progress\ s\ R) \implies$   
   $(s, s') \in trans\ l2 \implies$   
   $run\text{-ended} (progress\ s'\ R)$

$\langle proof \rangle$

**declare**  $domIff$  [*iff del*]

**lemma**  $l2\text{-can-signal-trans}$ :

$can\text{-signal}\ s'\ A\ B \implies$   
 $(s, s') \in trans\ l2 \implies$

*can-signal s A B*  
 ⟨proof⟩

## 22.2 Invariants

### 22.2.1 inv1

If *can-signal s A B* (i.e.,  $A, B$  are the test session agents and the test is not finished), then  $A$  and  $B$  are honest.

#### definition

*l2-inv1* :: *l2-state set*

#### where

$$\begin{aligned} l2\text{-inv1} &\equiv \{s. \forall A B. \\ &\quad \text{can-signal } s \ A \ B \longrightarrow \\ &\quad A \notin \text{bad } s \wedge B \notin \text{bad } s \\ &\quad \} \end{aligned}$$

**lemmas** *l2-inv1I* = *l2-inv1-def* [*THEN setc-def-to-intro, rule-format*]

**lemmas** *l2-inv1E* [*elim*] = *l2-inv1-def* [*THEN setc-def-to-elim, rule-format*]

**lemmas** *l2-inv1D* = *l2-inv1-def* [*THEN setc-def-to-dest, rule-format, rotated 1, simplified*]

**lemma** *l2-inv1-init* [*iff*]:

*init l2*  $\subseteq$  *l2-inv1*

⟨proof⟩

**lemma** *l2-inv1-trans* [*iff*]:

$\{l2\text{-inv1}\}$  *trans* *l2*  $\{> l2\text{-inv1}\}$

⟨proof⟩

**lemma** *PO-l2-inv1* [*iff*]: *reach l2*  $\subseteq$  *l2-inv1*

⟨proof⟩

### 22.2.2 inv2 (authentication guard)

If *Auth B A* ⟨*Number 0, gny, Exp Gen (NonceF (Ra \$ nx))*⟩  $\in$  *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

$$\begin{aligned} l2\text{-inv2} &\equiv \{s. \forall Ra \ A \ B \ gny. \\ &\quad \text{Auth } B \ A \ \langle \text{Number } 0, \text{gny}, \text{Exp Gen (NonceF (Ra \$ nx))} \rangle \in \text{chan } s \longrightarrow \\ &\quad A \notin \text{bad } s \wedge B \notin \text{bad } s \longrightarrow \\ &\quad (\exists Rb. \text{guessed-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \wedge \\ &\quad \quad \text{in-progressS } (\text{progress } s \ Rb) \ \{xny, xgnx, xgny, xsk\} \wedge \\ &\quad \quad \text{gny} = \text{Exp Gen (NonceF (Rb \$ ny))} \wedge \\ &\quad \quad \text{guessed-frame } Rb \ xgnx = \text{Some (Exp Gen (NonceF (Ra \$ nx)))) \\ &\quad \} \end{aligned}$$

**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*

*<proof>*

**lemma** *l2-inv2-trans* [iff]:

{*l2-inv2*} *trans l2* {> *l2-inv2*}

*<proof>*

**lemma** *PO-l2-inv2* [iff]: *reach l2*  $\subseteq$  *l2-inv2*

*<proof>*

### 22.2.3 inv3 (authentication guard)

If *Auth A B* (*Number 1, gnx, Exp Gen (NonceF (Rb\$ny))*)  $\in$  *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**

*l2-inv3*  $\equiv$  {*s.  $\forall$  Rb A B gnx.*

*Auth A B* (*Number 1, gnx, Exp Gen (NonceF (Rb\$ny))*)  $\in$  *chan s*  $\longrightarrow$

*A*  $\notin$  *bad s*  $\wedge$  *B*  $\notin$  *bad s*  $\longrightarrow$

( $\exists$  *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 gnx*  $\wedge$   
*guessed-frame Ra xgny = Some (Exp Gen (NonceF (Rb\$ny)))*)  
 }

**lemmas** *l2-inv3I* = *l2-inv3-def* [*THEN setc-def-to-intro, rule-format*]

**lemmas** *l2-inv3E* [*elim*] = *l2-inv3-def* [*THEN setc-def-to-elim, rule-format*]

**lemmas** *l2-inv3D* = *l2-inv3-def* [*THEN setc-def-to-dest, rule-format, rotated 1, simplified*]

**lemma** *l2-inv3-init* [iff]:

*init l2*  $\subseteq$  *l2-inv3*

*<proof>*

**lemma** *l2-inv3-trans* [iff]:

{*l2-inv3*} *trans l2* {> *l2-inv3*}

*<proof>*

**lemma** *PO-l2-inv3* [iff]: *reach l2*  $\subseteq$  *l2-inv3*

*<proof>*

### 22.2.4 inv4

For an initiator, the session key is always *gny* <sup>$\hat{n}$</sup> *x*.

**definition**

*l2-inv4* :: *l2-state set*

**where**

*l2-inv4*  $\equiv$  {*s.  $\forall$  Ra A B gny.*

*guessed-runs Ra =* (*role=Init, owner=A, partner=B*)  $\longrightarrow$

*in-progress* (*progress s Ra*) *xsk*  $\longrightarrow$



$$\begin{aligned} & \text{guessed-frame } Ra \text{ } xgny = \text{Some } gny \longrightarrow \\ & \text{guessed-frame } Ra \text{ } xsk = \text{Some } (\text{Exp } gny (\text{NonceF } (Ra\$nx))) \\ & \} \end{aligned}$$

**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]:

$\text{init } l2 \subseteq l2\text{-inv4}$

$\langle \text{proof} \rangle$

**lemma**  $l2\text{-inv4}\text{-trans}$  [iff]:

$\{l2\text{-inv4}\} \text{ trans } l2 \{> l2\text{-inv4}\}$

$\langle \text{proof} \rangle$

**lemma**  $PO\text{-}l2\text{-inv4}$  [iff]:  $\text{reach } l2 \subseteq l2\text{-inv4}$

$\langle \text{proof} \rangle$

### 22.2.5 inv4'

For a responder, the session key is always  $gnx \hat{=} ny$ .

**definition**

$l2\text{-inv4}' :: l2\text{-state set}$

**where**

$l2\text{-inv4}' \equiv \{s. \forall Rb \ A \ B \ gnx.$

$\text{guessed-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \longrightarrow$

$\text{in-progress } (\text{progress } s \ Rb) \ xsk \longrightarrow$

$\text{guessed-frame } Rb \ xgnx = \text{Some } gnx \longrightarrow$

$\text{guessed-frame } Rb \ xsk = \text{Some } (\text{Exp } gnx (\text{NonceF } (Rb\$ny)))$

$\}$

**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]:

$\text{init } l2 \subseteq l2\text{-inv4}'$

$\langle \text{proof} \rangle$

**lemma**  $l2\text{-inv4}'\text{-trans}$  [iff]:

$\{l2\text{-inv4}'\} \text{ trans } l2 \{> l2\text{-inv4}'\}$

$\langle \text{proof} \rangle$

**lemma**  $PO\text{-}l2\text{-inv4}'$  [iff]:  $\text{reach } l2 \subseteq l2\text{-inv4}'$

$\langle \text{proof} \rangle$

### 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**

$$\begin{aligned}
l2\text{-inv5} &\equiv \{s. \forall A B M. \\
&(\text{Confid } A B M \in \text{chan } s \vee \text{Secure } A B M \in \text{chan } s) \longrightarrow \\
&M \in \text{dy-fake-msg } (\text{bad } s) (\text{ik } s) (\text{chan } s) \\
&\}
\end{aligned}$$

**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]:

$\text{init } l2 \subseteq l2\text{-inv5}$

$\langle \text{proof} \rangle$

**lemma**  $l2\text{-inv5-trans}$  [iff]:

$\{l2\text{-inv5}\} \text{ trans } l2 \{> l2\text{-inv5}\}$

$\langle \text{proof} \rangle$

**lemma**  $PO\text{-}l2\text{-inv5}$  [iff]:  $\text{reach } l2 \subseteq l2\text{-inv5}$

$\langle \text{proof} \rangle$

### 22.2.7 inv6

For a run  $R$  (with any role), the session key always has the form *something* <sup>$n$</sup>  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]:

$\text{init } l2 \subseteq l2\text{-inv6}$

$\langle \text{proof} \rangle$

**lemma**  $l2\text{-inv6-trans}$  [iff]:

$\{l2\text{-inv6}\} \text{ trans } l2 \{> l2\text{-inv6}\}$

$\langle \text{proof} \rangle$

**lemma**  $PO\text{-}l2\text{-inv6}$  [iff]:  $\text{reach } l2 \subseteq l2\text{-inv6}$

$\langle \text{proof} \rangle$

### 22.2.8 inv7

Form of the messages in  $\text{extr } (\text{bad } s) (\text{ik } s) (\text{chan } s) = \text{synth } (\text{analz generators})$ .

**abbreviation**

$$\text{generators} \equiv \{x. \exists N. x = \text{Exp Gen (Nonce N)}\} \cup \\ \{\text{Exp } y \text{ (NonceF (R\$N))} \mid y N R. R \neq \text{test} \wedge R \notin \text{partners}\}$$
**lemma** *analz-generators*: *analz generators = generators**<proof>***definition***l2-inv7* :: *l2-state set***where**

$$\text{l2-inv7} \equiv \{s. \\ \text{extr (bad } s) \text{ (ik } s) \text{ (chan } s) \subseteq \\ \text{synth (analz (generators))}\}$$
**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**<proof>***lemma** *l2-inv7-step1*:*{l2-inv7}* *l2-step1 Ra A B*  $\{>$  *l2-inv7**<proof>***lemma** *l2-inv7-step2*:*{l2-inv7}* *l2-step2 Rb A B gnx*  $\{>$  *l2-inv7**<proof>***lemma** *l2-inv7-step3*:*{l2-inv7}* *l2-step3 Ra A B gny*  $\{>$  *l2-inv7**<proof>***lemma** *l2-inv7-step4*:*{l2-inv7}* *l2-step4 Rb A B gnx*  $\{>$  *l2-inv7**<proof>***lemma** *l2-inv7-dy-fake-msg*:*{l2-inv7}* *l2-dy-fake-msg M*  $\{>$  *l2-inv7**<proof>***lemma** *l2-inv7-dy-fake-chan*:*{l2-inv7}* *l2-dy-fake-chan M*  $\{>$  *l2-inv7**<proof>***lemma** *l2-inv7-lkr-others*:*{l2-inv7} \cap* *l2-inv5* *l2-lkr-others A*  $\{>$  *l2-inv7**<proof>***lemma** *l2-inv7-lkr-after*:*{l2-inv7} \cap* *l2-inv5* *l2-lkr-after A*  $\{>$  *l2-inv7*

$\langle \text{proof} \rangle$

**lemma** *l2-inv7-skr*:

$\{l2\text{-inv}7 \cap l2\text{-inv}6\} l2\text{-skr } R K \{> l2\text{-inv}7\}$

$\langle \text{proof} \rangle$

**lemmas** *l2-inv7-trans-aux* =

*l2-inv7-step1 l2-inv7-step2 l2-inv7-step3 l2-inv7-step4*  
*l2-inv7-dy-fake-msg l2-inv7-dy-fake-chan*  
*l2-inv7-lkr-others l2-inv7-lkr-after l2-inv7-skr*

**lemma** *l2-inv7-trans* [iff]:

$\{l2\text{-inv}7 \cap l2\text{-inv}5 \cap l2\text{-inv}6\} \text{trans } l2 \{> l2\text{-inv}7\}$

$\langle \text{proof} \rangle$

**lemma** *PO-l2-inv7* [iff]: *reach l2*  $\subseteq$  *l2-inv7*

$\langle \text{proof} \rangle$

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**

$l2\text{-inv}8 \equiv \{s.$   
 $\text{secret } s \subseteq \{Exp (Exp \text{ Gen } (NonceF (R\$N))) (NonceF (R'\$N')) \mid N N' R R'.$   
 $R = \text{test} \wedge R' \in \text{partners}\}$   
 $\}$

**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*

$\langle \text{proof} \rangle$

Steps 3 and 4 are the hard part.

**lemma** *l2-inv8-step3*:

$\{l2\text{-inv}8 \cap l2\text{-inv}1 \cap l2\text{-inv}2 \cap l2\text{-inv}4'\} l2\text{-step}3 Ra A B gny \{> l2\text{-inv}8\}$

$\langle \text{proof} \rangle$

**lemma** *l2-inv8-step4*:

$\{l2\text{-inv}8 \cap l2\text{-inv}1 \cap l2\text{-inv}3 \cap l2\text{-inv}4 \cap l2\text{-inv}4'\} l2\text{-step}4 Rb A B gnx \{> l2\text{-inv}8\}$

$\langle \text{proof} \rangle$

**lemma** *l2-inv8-trans* [iff]:

$\{l2\text{-inv}8 \cap l2\text{-inv}1 \cap l2\text{-inv}2 \cap l2\text{-inv}3 \cap l2\text{-inv}4 \cap l2\text{-inv}4'\} \text{ trans } l2 \{> l2\text{-inv}8\}$   
 $\langle \text{proof} \rangle$

**lemma** *PO-l2-inv8 [iff]: reach l2  $\subseteq$  l2-inv8*  
 $\langle \text{proof} \rangle$

Auxiliary destruction rule for inv8.

**lemma** *Exp-Exp-Gen-synth:*

$\text{Exp } (\text{Exp Gen } X) Y \in \text{synth } H \implies \text{Exp } (\text{Exp Gen } X) Y \in H \vee X \in \text{synth } H \vee Y \in \text{synth } H$   
 $\langle \text{proof} \rangle$

**lemma** *l2-inv8-aux:*

$s \in l2\text{-inv}8 \implies$   
 $x \in \text{secret } s \implies$   
 $x \notin \text{synth } (\text{analz generators})$   
 $\langle \text{proof} \rangle$

## 22.3 Refinement

Mediator function.

**definition**

$\text{med}12s :: l2\text{-obs} \Rightarrow l1\text{-obs}$

**where**

$\text{med}12s \ t \equiv \langle$   
 $\text{ik} = \text{ik } t,$   
 $\text{secret} = \text{secret } t,$   
 $\text{progress} = \text{progress } t,$   
 $\text{signalsInit} = \text{signalsInit } t,$   
 $\text{signalsResp} = \text{signalsResp } t$   
 $\rangle$

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]:*

$(s1, s2) \in R12s \implies$   
 $\text{can-signal } s1 \ A \ B \longleftrightarrow \text{can-signal } s2 \ A \ B$   
 $\langle \text{proof} \rangle$

Protocol events.

**lemma** *l2-step1-refines-step1:*

$\{R12s\} \ l1\text{-step1 } Ra \ A \ B, \ l2\text{-step1 } Ra \ A \ B \ \{> R12s\}$   
 $\langle \text{proof} \rangle$

**lemma** *l2-step2-refines-step2*:  
 $\{R12s\}$  *l1-step2* *Rb A B gnx*, *l2-step2* *Rb A B gnx*  $\{>R12s\}$   
 $\langle$ *proof* $\rangle$

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*:  
 $\{R12s \cap UNIV \times (l2-inv1 \cap l2-inv2 \cap l2-inv4' \cap l2-inv7 \cap l2-inv8)\}$   
*l1-step3* *Ra A B gny*, *l2-step3* *Ra A B gny*  
 $\{>R12s\}$   
 $\langle$ *proof* $\rangle$

**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* *Rb A B gnx*, *l2-step4* *Rb A B gnx*  
 $\{>R12s\}$   
 $\langle$ *proof* $\rangle$

Attacker events.

**lemma** *l2-dy-fake-chan-refines-skip*:  
 $\{R12s\}$  *Id*, *l2-dy-fake-chan* *M*  $\{>R12s\}$   
 $\langle$ *proof* $\rangle$

**lemma** *l2-dy-fake-msg-refines-learn*:  
 $\{R12s \cap UNIV \times (l2-inv7 \cap l2-inv8)\}$  *l1-learn* *m*, *l2-dy-fake-msg* *m*  $\{>R12s\}$   
 $\langle$ *proof* $\rangle$

Compromising events.

**lemma** *l2-lkr-others-refines-skip*:  
 $\{R12s\}$  *Id*, *l2-lkr-others* *A*  $\{>R12s\}$   
 $\langle$ *proof* $\rangle$

**lemma** *l2-lkr-after-refines-skip*:  
 $\{R12s\}$  *Id*, *l2-lkr-after* *A*  $\{>R12s\}$   
 $\langle$ *proof* $\rangle$

**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*  $\{>R12s\}$   
 $\langle$ *proof* $\rangle$

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*)  
 $\langle$ *proof* $\rangle$

**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*  
 $\{> R12s\}$   
 ⟨proof⟩

**lemma** *PO-obs-consistent-R12s* [iff]:  
*obs-consistent R12s med12s l1 l2*  
 ⟨proof⟩

**lemma** *l2-refines-l1* [iff]:  
*refines*  
 $(R12s \cap$   
 $(reach\ l1 \times (l2-inv1 \cap l2-inv2 \cap l2-inv3 \cap l2-inv4 \cap l2-inv4' \cap l2-inv5 \cap$   
 $l2-inv6 \cap l2-inv7 \cap l2-inv8)))$   
*med12s l1 l2*  
 ⟨proof⟩

**lemma** *l2-implements-l1* [iff]:  
*implements med12s l1 l2*  
 ⟨proof⟩

## 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-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-partial-secrecy*  
 ⟨proof⟩

**lemma** *l2-oreach-dy-fake-msg*:  
 $\llbracket s \in oreach\ l2; x \in dy\text{-fake-msg } (bad\ s) (ik\ s) (chan\ s) \rrbracket$   
 $\implies s (ik := insert\ x\ (ik\ s)) \in oreach\ l2$   
 ⟨proof⟩

**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-secrecy*

*<proof>*

**lemma** *l2-secrecy* [iff]: *reach l2*  $\subseteq$  *l2-secrecy*

*<proof>*

**abbreviation** *l2-iagreement-Init*  $\equiv$  *l1-iagreement-Init*

**lemma** *l2-obs-iagreement-Init* [iff]: *oreach l2*  $\subseteq$  *l2-iagreement-Init*

*<proof>*

**lemma** *l2-iagreement-Init* [iff]: *reach l2*  $\subseteq$  *l2-iagreement-Init*

*<proof>*

**abbreviation** *l2-iagreement-Resp*  $\equiv$  *l1-iagreement-Resp*

**lemma** *l2-obs-iagreement-Resp* [iff]: *oreach l2*  $\subseteq$  *l2-iagreement-Resp*

*<proof>*

**lemma** *l2-iagreement-Resp* [iff]: *reach l2*  $\subseteq$  *l2-iagreement-Resp*

*<proof>*

**end**



## 23 Authenticated Diffie-Hellman Protocol (L3 locale)

```
theory dhlvl3
imports dhlvl2 Implem-lemmas
begin
```

### 23.1 State and Events

Level 3 state.

(The types have to be defined outside the locale.)

```
record l3-state = l1-state +
  bad :: agent set
```

```
type-synonym l3-obs = l3-state
```

```
type-synonym
  l3-pred = l3-state set
```

```
type-synonym
  l3-trans = (l3-state × l3-state) set
```

Attacker event.

```
definition
  l3-dy :: msg ⇒ l3-trans
where
  l3-dy ≡ ik-dy
```

Compromise events.

```
definition
  l3-lkr-others :: agent ⇒ l3-trans
where
  l3-lkr-others A ≡ {(s,s').
    — guards
    A ≠ test-owner ∧
    A ≠ test-partner ∧
    — actions
    s' = s(bad := {A} ∪ bad s,
           ik := keys-of A ∪ ik s)
  }
```

```
definition
  l3-lkr-actor :: agent ⇒ l3-trans
where
  l3-lkr-actor A ≡ {(s,s').
    — guards
    A = test-owner ∧
    A ≠ test-partner ∧
    — actions
    s' = s(bad := {A} ∪ bad s,
           ik := keys-of A ∪ ik s)
  }
```

**definition**

$$l3-lkr\text{-after} :: \text{agent} \Rightarrow l3\text{-trans}$$
**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:

$s' = s(\text{progress} := (\text{progress } s)(\text{Rb} \mapsto \{xny, xgnx, xgny, xsk, xEnd\}),$

$\text{secret} := \{x. x = \text{Exp gn}x (\text{NonceF (Rb\$ny)}) \wedge \text{Rb} = \text{test}\} \cup \text{secret } s,$

$\text{signalsResp} := \text{if can-signal } s \text{ A B then}$

$\text{addSignal } (\text{signalsResp } s) (\text{Commit } A \text{ B } (\text{Exp gn}x (\text{NonceF (Rb\$ny)})))$

$\text{else}$

$\text{signalsResp } s$

$\rangle$

}

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]:  $\text{parts ik-init} = \text{ik-init}$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**lemma**  $\text{abs-ik-init}$  [iff]:  $\text{abs ik-init} = \{\}$

$\langle \text{proof} \rangle$

**lemma**  $\text{payloadSet-ik-init}$  [iff]:  $\text{ik-init} \cap \text{payload} = \{\}$

$\langle \text{proof} \rangle$

**lemma**  $\text{validSet-ik-init}$  [iff]:  $\text{ik-init} \cap \text{valid} = \{\}$

$\langle \text{proof} \rangle$

**definition**

$l3\text{-init} :: l3\text{-state set}$

**where**

$l3\text{-init} \equiv \{$

$\text{ik} = \text{ik-init},$

$\text{secret} = \{\},$

$\text{progress} = \text{Map.empty},$

$\text{signalsInit} = \lambda x. 0,$

$signalsResp = \lambda x. 0,$   
 $bad = bad-init$   
 $\})$

**lemmas**  $l3-init-defs = l3-init-def\ ik-init-def$

**definition**

$l3-trans :: l3-trans$

**where**

$l3-trans \equiv (\bigcup M\ X\ Rb\ Ra\ A\ B\ K.$   
 $l3-step1\ Ra\ A\ B \cup$   
 $l3-step2\ Rb\ A\ B\ X \cup$   
 $l3-step3\ Ra\ A\ B\ X \cup$   
 $l3-step4\ Rb\ A\ B\ X \cup$   
 $l3-dy\ M \cup$   
 $l3-lkr-others\ A \cup$   
 $l3-lkr-after\ A \cup$   
 $l3-skr\ Ra\ K \cup$   
 $Id$   
 $)$

**definition**

$l3 :: (l3-state, l3-obs)\ spec\ \mathbf{where}$   
 $l3 \equiv ()$   
 $init = l3-init,$   
 $trans = l3-trans,$   
 $obs = id$   
 $\})$

**lemmas**  $l3-loc-defs =$

$l3-step1-def\ l3-step2-def\ l3-step3-def\ l3-step4-def$   
 $l3-def\ l3-init-defs\ l3-trans-def$   
 $l3-dy-def$   
 $l3-lkr-others-def\ l3-lkr-after-def\ l3-skr-def$

**lemmas**  $l3-defs = l3-loc-defs\ ik-dy-def$

**lemmas**  $l3-nostep-defs = l3-def\ l3-init-def\ l3-trans-def$

**lemma**  $l3-obs-id [simp]: obs\ l3 = id$

$\langle proof \rangle$

## 23.2 Invariants

### 23.2.1 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**  $l3\text{-inv}1I = l3\text{-inv}1\text{-def}$  [*THEN setc-def-to-intro, rule-format*]  
**lemmas**  $l3\text{-inv}1E$  [*elim*] =  $l3\text{-inv}1\text{-def}$  [*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*]:  $\llbracket LtK K \in parts (ik s); s \in l3\text{-inv}1 \rrbracket \implies LtK K \in ik s$   
 $\langle proof \rangle$

**lemma**  $l3\text{-inv}1\text{-init}$  [*iff*]:  
 $init\ l3 \subseteq l3\text{-inv}1$   
 $\langle proof \rangle$

**lemma**  $l3\text{-inv}1\text{-trans}$  [*iff*]:  
 $\{l3\text{-inv}1\}$  *trans*  $l3$   $\{> l3\text{-inv}1\}$   
 $\langle proof \rangle$

**lemma**  $PO\text{-}l3\text{-inv}1$  [*iff*]:  
 $reach\ l3 \subseteq l3\text{-inv}1$   
 $\langle proof \rangle$

### 23.2.2 inv2: $l3\text{-state.bad}$ $s$ 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!*]:  
 $init\ l3 \subseteq l3\text{-inv}2$   
 $\langle proof \rangle$

**lemma**  $l3\text{-inv}2\text{-trans}$  [*simp,intro!*]:  
 $\{l3\text{-inv}2 \cap l3\text{-inv}1\}$  *trans*  $l3$   $\{> l3\text{-inv}2\}$   
 $\langle proof \rangle$

**lemma**  $PO\text{-}l3\text{-inv}2$  [*iff*]:  $reach\ l3 \subseteq l3\text{-inv}2$   
 $\langle proof \rangle$

### 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**

$l3\text{-inv}3 :: l3\text{-state set}$

**where**

```
l3-inv3 ≡ {s.  
  analz (ik s) ⊆  
  synth (analz ((ik s ∩ payload) ∪ ((ik s) ∩ valid) ∪ (ik s ∩ range LtK) ∪ Tags))  
}
```

**lemmas**  $l3\text{-inv}3I = l3\text{-inv}3\text{-def}$  [THEN setc-def-to-intro, rule-format]

**lemmas**  $l3\text{-inv}3E = l3\text{-inv}3\text{-def}$  [THEN setc-def-to-elim, rule-format]

**lemmas**  $l3\text{-inv}3D = l3\text{-inv}3\text{-def}$  [THEN setc-def-to-dest, rule-format]

**lemma**  $l3\text{-inv}3\text{-init}$  [iff]:

$init\ l3 \subseteq l3\text{-inv}3$

⟨proof⟩

**declare**  $domIff$  [iff del]

Most of the cases in this proof are simple and very similar. The proof could probably be shortened.

**lemma**  $l3\text{-inv}3\text{-trans}$  [simp,intro!]:

$\{l3\text{-inv}3\}$  trans  $l3 \{> l3\text{-inv}3\}$

⟨proof⟩

**lemma**  $PO\text{-}l3\text{-inv}3$  [iff]:  $reach\ l3 \subseteq l3\text{-inv}3$

⟨proof⟩

### 23.2.4 inv4: the intruder knows the tags

**definition**

$l3\text{-inv}4 :: l3\text{-state set}$

**where**

$l3\text{-inv}4 \equiv \{s.$

$Tags \subseteq ik\ s$

$\}$

**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!]:

$init\ l3 \subseteq l3\text{-inv}4$

⟨proof⟩

**lemma**  $l3\text{-inv}4\text{-trans}$  [simp,intro!]:

$\{l3\text{-inv}4\}$  trans  $l3 \{> l3\text{-inv}4\}$

⟨proof⟩

**lemma**  $PO\text{-}l3\text{-inv}4$  [simp,intro!]:  $reach\ l3 \subseteq l3\text{-inv}4$

⟨proof⟩

The remaining invariants are derived from the others. They are not protocol dependent provided the previous invariants hold.

### 23.2.5 inv5

The messages that the L3 DY intruder can derive from the intruder knowledge (using *synth/analz*), are either implementations or intermediate messages or can also be derived by the L2 intruder from the set *extr* (*l3-state.bad s*) (*ik s*  $\cap$  *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

$$\begin{aligned} l3-inv5 \equiv \{ & s. \\ & synth\ (analz\ (ik\ s)) \subseteq \\ & dy-fake-msg\ (bad\ s)\ (ik\ s \cap\ payload)\ (abs\ (ik\ s)) \cup\ -payload \\ & \} \end{aligned}$$

**lemmas** *l3-inv5I* = *l3-inv5-def* [*THEN setc-def-to-intro, rule-format*]

**lemmas** *l3-inv5E* = *l3-inv5-def* [*THEN setc-def-to-elim, rule-format*]

**lemmas** *l3-inv5D* = *l3-inv5-def* [*THEN setc-def-to-dest, rule-format*]

**lemma** *l3-inv5-derived*: *l3-inv2*  $\cap$  *l3-inv3*  $\subseteq$  *l3-inv5*  
 $\langle proof \rangle$

**lemma** *PO-l3-inv5* [*simp,intro!*]: *reach l3*  $\subseteq$  *l3-inv5*  
 $\langle proof \rangle$

### 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

$$\begin{aligned} l3-inv6 \equiv \{ & s. \forall\ A\ B\ M. \\ & (implInsec\ A\ B\ M \in\ synth\ (analz\ (ik\ s)) \wedge\ M \in\ payload) \longrightarrow \\ & (implInsec\ A\ B\ M \in\ ik\ s \vee\ M \in\ synth\ (analz\ (ik\ s))) \\ & \} \end{aligned}$$

**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*  $\cap$  *l3-inv4*  $\subseteq$  *l3-inv6*  
 $\langle proof \rangle$

**lemma** *PO-l3-inv6* [*simp,intro!*]: *reach l3*  $\subseteq$  *l3-inv6*  
 $\langle proof \rangle$



### 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

$l3\text{-inv7} :: l3\text{-state set}$

#### 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**  $l3\text{-inv7}I = l3\text{-inv7}\text{-def}$  [THEN setc-def-to-intro, rule-format]

**lemmas**  $l3\text{-inv7}E = l3\text{-inv7}\text{-def}$  [THEN setc-def-to-elim, rule-format]

**lemmas**  $l3\text{-inv7}D = l3\text{-inv7}\text{-def}$  [THEN setc-def-to-dest, rule-format]

**lemma**  $l3\text{-inv7}\text{-derived}$  [simp,intro!]:

$l3\text{-inv3} \cap l3\text{-inv4} \subseteq l3\text{-inv7}$

$\langle\text{proof}\rangle$

**lemma**  $PO\text{-}l3\text{-inv7}$  [simp,intro!]:  $\text{reach } l3 \subseteq l3\text{-inv7}$

$\langle\text{proof}\rangle$

### 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

$l3\text{-inv8} :: l3\text{-state set}$

#### where

$$l3\text{-inv8} \equiv \{s. \forall A B M. \\ (\text{implAuth } A B M \in \text{synth } (\text{analz } (ik\ s)) \wedge M \in \text{payload}) \longrightarrow \\ (\text{implAuth } A B M \in ik\ s \vee (M \in \text{synth } (\text{analz } (ik\ s)) \wedge (A \in \text{bad } s \vee B \in \text{bad } s))) \\ \}$$

**lemmas**  $l3\text{-inv8}I = l3\text{-inv8}\text{-def}$  [THEN setc-def-to-intro, rule-format]

**lemmas**  $l3\text{-inv8}E = l3\text{-inv8}\text{-def}$  [THEN setc-def-to-elim, rule-format]

**lemmas**  $l3\text{-inv8}D = l3\text{-inv8}\text{-def}$  [THEN setc-def-to-dest, rule-format]

**lemma**  $l3\text{-inv8}\text{-derived}$  [iff]:

$l3\text{-inv2} \cap l3\text{-inv3} \cap l3\text{-inv4} \subseteq l3\text{-inv8}$

*<proof>*

**lemma** *PO-l3-inv8* [iff]: *reach l3*  $\subseteq$  *l3-inv8*

*<proof>*

### 23.2.9 inv9

If the level 3 intruder can deduce a message implementing a secure channel message then either:

- the message is already in the intruder knowledge, or
- the message is constructed, and in this case the payload can also be deduced by the intruder, and one of the agents is bad.

**definition**

*l3-inv9* :: *l3-state set*

**where**

*l3-inv9*  $\equiv$   $\{s. \forall A B M.$

$(\text{implSecure } A B M \in \text{synth } (\text{analz } (\text{ik } s)) \wedge M \in \text{payload}) \longrightarrow$

$(\text{implSecure } A B M \in \text{ik } s \vee (M \in \text{synth } (\text{analz } (\text{ik } s)) \wedge (A \in \text{bad } s \vee B \in \text{bad } s)))$

$\}$

**lemmas** *l3-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*  $\cap$  *l3-inv3*  $\cap$  *l3-inv4*  $\subseteq$  *l3-inv9*

*<proof>*

**lemma** *PO-l3-inv9* [iff]: *reach l3*  $\subseteq$  *l3-inv9*

*<proof>*

## 23.3 Refinement

Mediator function.

**definition**

*med23s* :: *l3-obs*  $\Rightarrow$  *l2-obs*

**where**

*med23s t*  $\equiv$  ( $\mid$

*ik* = *ik t*  $\cap$  *payload*,

*secret* = *secret t*,

*progress* = *progress t*,

*signalsInit* = *signalsInit t*,

*signalsResp* = *signalsResp t*,

*chan* = *abs (ik t)*,

*bad* = *bad t*

$\mid$ )

Relation between states.

**definition**

$R23s :: (l2\text{-state} * l3\text{-state}) \text{ set}$

**where**

$$\begin{aligned} R23s &\equiv \{(s, s') \\ &\quad s = \text{med}23s\ s' \\ &\quad \} \end{aligned}$$

**lemmas**  $R23s\text{-defs} = R23s\text{-def}\ \text{med}23s\text{-def}$

**lemma**  $R23sI$ :

$$\begin{aligned} &\llbracket ik\ s = ik\ t \cap \text{payload}; \text{secret}\ s = \text{secret}\ t; \text{progress}\ s = \text{progress}\ t; \\ &\quad \text{signalsInit}\ s = \text{signalsInit}\ t; \text{signalsResp}\ s = \text{signalsResp}\ t; \\ &\quad \text{chan}\ s = \text{abs}\ (ik\ t); l2\text{-state}.bad\ s = bad\ t \rrbracket \\ &\implies (s, t) \in R23s \\ &\langle \text{proof} \rangle \end{aligned}$$

**lemma**  $R23sD$ :

$$\begin{aligned} (s, t) \in R23s &\implies \\ &ik\ s = ik\ t \cap \text{payload} \wedge \text{secret}\ s = \text{secret}\ t \wedge \text{progress}\ s = \text{progress}\ t \wedge \\ &\quad \text{signalsInit}\ s = \text{signalsInit}\ t \wedge \text{signalsResp}\ s = \text{signalsResp}\ t \wedge \\ &\quad \text{chan}\ s = \text{abs}\ (ik\ t) \wedge l2\text{-state}.bad\ s = bad\ t \\ &\langle \text{proof} \rangle \end{aligned}$$

**lemma**  $R23sE$  [*elim*]:

$$\begin{aligned} &\llbracket (s, t) \in R23s; \\ &\quad \llbracket ik\ s = ik\ t \cap \text{payload}; \text{secret}\ s = \text{secret}\ t; \text{progress}\ s = \text{progress}\ t; \\ &\quad \quad \text{signalsInit}\ s = \text{signalsInit}\ t; \text{signalsResp}\ s = \text{signalsResp}\ t; \\ &\quad \quad \text{chan}\ s = \text{abs}\ (ik\ t); l2\text{-state}.bad\ s = bad\ t \rrbracket \implies P \rrbracket \\ &\implies P \\ &\langle \text{proof} \rangle \end{aligned}$$

**lemma**  $\text{can-signal-R}23$  [*simp*]:

$$\begin{aligned} (s2, s3) \in R23s &\implies \\ &\text{can-signal}\ s2\ A\ B \longleftrightarrow \text{can-signal}\ s3\ A\ B \\ &\langle \text{proof} \rangle \end{aligned}$$

### 23.3.1 Protocol events

**lemma**  $l3\text{-step1-refines-step1}$ :

$$\{R23s\}\ l2\text{-step1}\ Ra\ A\ B, l3\text{-step1}\ Ra\ A\ B\ \{>R23s\}$$

$\langle \text{proof} \rangle$

**lemma**  $l3\text{-step2-refines-step2}$ :

$$\{R23s\}\ l2\text{-step2}\ Rb\ A\ B\ gnx, l3\text{-step2}\ Rb\ A\ B\ gnx\ \{>R23s\}$$

$\langle \text{proof} \rangle$

**lemma**  $l3\text{-step3-refines-step3}$ :

$$\{R23s\}\ l2\text{-step3}\ Ra\ A\ B\ gny, l3\text{-step3}\ Ra\ A\ B\ gny\ \{>R23s\}$$

$\langle \text{proof} \rangle$

**lemma**  $l3\text{-step4-refines-step4}$ :

$$\{R23s\}\ l2\text{-step4}\ Rb\ A\ B\ gnx, l3\text{-step4}\ Rb\ A\ B\ gnx\ \{>R23s\}$$

$\langle \text{proof} \rangle$

### 23.3.2 Intruder events

**lemma** *l3-dy-payload-refines-dy-fake-msg:*

$$M \in \text{payload} \implies \{R23s \cap UNIV \times l3\text{-inv}5\} l2\text{-dy-fake-msg } M, l3\text{-dy } M \{>R23s\}$$

*<proof>*

**lemma** *l3-dy-valid-refines-dy-fake-chan:*

$$\llbracket M \in \text{valid}; M' \in \text{abs } \{M\} \rrbracket \implies \{R23s \cap UNIV \times (l3\text{-inv}5 \cap l3\text{-inv}6 \cap l3\text{-inv}7 \cap l3\text{-inv}8 \cap l3\text{-inv}9)\} l2\text{-dy-fake-chan } M', l3\text{-dy } M \{>R23s\}$$

*<proof>*

**lemma** *l3-dy-valid-refines-dy-fake-chan-Un:*

$$M \in \text{valid} \implies \{R23s \cap UNIV \times (l3\text{-inv}5 \cap l3\text{-inv}6 \cap l3\text{-inv}7 \cap l3\text{-inv}8 \cap l3\text{-inv}9)\} \bigcup M'. l2\text{-dy-fake-chan } M', l3\text{-dy } M \{>R23s\}$$

*<proof>*

**lemma** *l3-dy-isLtKey-refines-skip:*

$$\{R23s\} Id, l3\text{-dy } (LtK \text{ ltk}) \{>R23s\}$$

*<proof>*

**lemma** *l3-dy-others-refines-skip:*

$$\llbracket M \notin \text{range } LtK; M \notin \text{valid}; M \notin \text{payload} \rrbracket \implies \{R23s\} Id, l3\text{-dy } M \{>R23s\}$$

*<proof>*

**lemma** *l3-dy-refines-dy-fake-msg-dy-fake-chan-skip:*

$$\{R23s \cap UNIV \times (l3\text{-inv}5 \cap l3\text{-inv}6 \cap l3\text{-inv}7 \cap l3\text{-inv}8 \cap l3\text{-inv}9)\} l2\text{-dy-fake-msg } M \cup (\bigcup M'. l2\text{-dy-fake-chan } M') \cup Id, l3\text{-dy } M \{>R23s\}$$

*<proof>*

### 23.3.3 Compromise events

**lemma** *l3-lkr-others-refines-lkr-others:*

$$\{R23s\} l2\text{-lkr-others } A, l3\text{-lkr-others } A \{>R23s\}$$

*<proof>*

**lemma** *l3-lkr-after-refines-lkr-after:*

$$\{R23s\} l2\text{-lkr-after } A, l3\text{-lkr-after } A \{>R23s\}$$

*<proof>*

**lemma** *l3-skr-refines-skr:*

$$\{R23s\} l2\text{-skr } R \ K, l3\text{-skr } R \ K \{>R23s\}$$

*<proof>*

**lemmas** *l3-trans-refines-l2-trans =*  
*l3-step1-refines-step1 l3-step2-refines-step2 l3-step3-refines-step3 l3-step4-refines-step4*  
*l3-dy-refines-dy-fake-msg-dy-fake-chan-skip*  
*l3-lkr-others-refines-lkr-others l3-lkr-after-refines-lkr-after l3-skr-refines-skr*

**lemma** *l3-refines-init-l2 [iff]:*  
*init l3  $\subseteq$  R23s “ (init l2)*  
*<proof>*

**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>*

**lemma** *PO-obs-consistent-R23s [iff]:*  
*obs-consistent R23s med23s l2 l3*  
*<proof>*

**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*  
*<proof>*

**lemma** *l3-implements-l2 [iff]:*  
*implements med23s l2 l3*  
*<proof>*

## 23.4 Derived invariants

### 23.4.1 inv10: secrets contain no implementation material

#### definition

*l3-inv10 :: l3-state set*

#### where

*l3-inv10  $\equiv$  {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*  
*<proof>*

**lemma** *l3-inv10-trans [iff]:*

$\{l3\text{-inv}10\}$  *trans*  $l3 \{> l3\text{-inv}10\}$   
 $\langle\text{proof}\rangle$

**lemma** *PO-l3-inv10* [*iff*]: *reach*  $l3 \subseteq l3\text{-inv}10$   
 $\langle\text{proof}\rangle$

**lemma** *l3-obs-inv10* [*iff*]: *oreach*  $l3 \subseteq l3\text{-inv}10$   
 $\langle\text{proof}\rangle$

### 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*  $\equiv \{s.$

$\text{dy-fake-msg}(bad\ s)\ (ik\ s \cap \text{payload})\ (\text{abs}(ik\ s)) \cap \text{secret}\ s = \{\}$   
 $\}$

**lemma** *l3-obs-partial-secrecy* [*iff*]: *oreach*  $l3 \subseteq l3\text{-partial-secrecy}$   
 $\langle\text{proof}\rangle$

### 23.4.3 Secrecy

#### definition

*l3-secrecy* :: ('a *l3-state-scheme*) *set*

#### where

*l3-secrecy*  $\equiv l1\text{-secrecy}$

**lemma** *l3-obs-inv5*: *oreach*  $l3 \subseteq l3\text{-inv}5$   
 $\langle\text{proof}\rangle$

**lemma** *l3-obs-secrecy* [*iff*]: *oreach*  $l3 \subseteq l3\text{-secrecy}$   
 $\langle\text{proof}\rangle$

**lemma** *l3-secrecy* [*iff*]: *reach*  $l3 \subseteq l3\text{-secrecy}$   
 $\langle\text{proof}\rangle$

### 23.4.4 Injective agreement

**abbreviation** *l3-iagreement-Init*  $\equiv l1\text{-iagreement-Init}$

**lemma** *l3-obs-iagreement-Init* [*iff*]: *oreach*  $l3 \subseteq l3\text{-iagreement-Init}$   
 $\langle\text{proof}\rangle$

**lemma** *l3-iagreement-Init* [*iff*]: *reach*  $l3 \subseteq l3\text{-iagreement-Init}$   
 $\langle\text{proof}\rangle$

**abbreviation**  $l3\text{-iagreement-Resp} \equiv l1\text{-iagreement-Resp}$

**lemma**  $l3\text{-obs-iagreement-Resp}$  [iff]:  $\text{oreach } l3 \subseteq l3\text{-iagreement-Resp}$   
*<proof>*

**lemma**  $l3\text{-iagreement-Resp}$  [iff]:  $\text{reach } l3 \subseteq l3\text{-iagreement-Resp}$   
*<proof>*

**end**

**end**

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

```
theory dhlvl3-asymmetric
imports dhlvl3 Implem-asymmetric
begin

interpretation dhlvl3-asm: dhlvl3 implem-asm
  <proof>

end
```



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

```
theory dhlvl3-symmetric
imports dhlvl3 Implem-symmetric
begin

interpretation dhlvl3-sym: dhlvl3 implem-sym
  <proof>

end
```

## 26 SKEME Protocol (L1)

```
theory sklvl1
imports 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 ltkeys in their payload and are therefore valid implementations.
3. We also ensure that the values generated by the frame owner are correctly guessed.
4. The new frame extends the previous one (from *Key-Agreement-Strong-Adversaries.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]:
```

$role (guessed-runs R) = Init \implies guessed-frame R xni = Some (NonceF (R\$ni))$   
 $guessed-frame-Resp-xny [simp]:$   
 $role (guessed-runs R) = Resp \implies guessed-frame R xny = Some (NonceF (R\$ny))$   
 $guessed-frame-Resp-xgny [simp]:$   
 $role (guessed-runs R) = Resp \implies guessed-frame R xgny = Some (Exp Gen (NonceF (R\$ny)))$   
 $guessed-frame-Resp-xnr [simp]:$   
 $role (guessed-runs R) = Resp \implies guessed-frame R xnr = Some (NonceF (R\$nr))$   
 $guessed-frame-xEnd [simp]:$   
 $guessed-frame R xEnd = Some End$   
 $guessed-frame-eq [simp]:$   
 $x \in \{xnx, xny, xgnx, xgny, xsk, xEnd\} \implies dhlvl1.guessed-frame R x = guessed-frame R x$   
 $\langle proof \rangle$

**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**

$skl1-step1 :: rid-t \Rightarrow agent \Rightarrow agent \Rightarrow ('a skl1-state-scheme * 'a skl1-state-scheme) set$

**where**

$skl1-step1 Ra A B \equiv \{(s, s') .$   
— guards:  
 $Ra \notin dom (progress s) \wedge$   
 $guessed-runs Ra = (role=Init, owner=A, partner=B) \wedge$   
— actions:  
 $s' = s \{$   
 $progress := (progress s)(Ra \mapsto \{xnx, xni, xgnx\})$   
 $\}$   
 $\}$

**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*  $Rb\ A\ B\ Ni\ gnx \equiv \{(s, s')\}$ .

— guards:

*guessed-runs*  $Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \wedge$   
*Rb*  $\notin \text{dom}(\text{progress } s) \wedge$   
*guessed-frame*  $Rb\ xgnx = \text{Some } gnx \wedge$   
*guessed-frame*  $Rb\ xni = \text{Some } Ni \wedge$   
*guessed-frame*  $Rb\ xsk = \text{Some } (\text{Exp } gnx (\text{NonceF } (Rb\$ny))) \wedge$

— actions:

$s' = s(\text{progress} := (\text{progress } s)(Rb \mapsto \{xny, xni, xnr, xgny, xgnx, xsk\}),$   
*signalsInit* :=  
  if *can-signal*  $s\ A\ B$  then  
    *addSignal* (*signalsInit*  $s$ )  
    (*Running*  $A\ B\ \langle Ni, \text{NonceF } (Rb\$nr), \text{Exp } gnx (\text{NonceF } (Rb\$ny)) \rangle$ )  
  else  
    *signalsInit*  $s,$   
*signalsInit2* :=  
  if *can-signal*  $s\ A\ B$  then  
    *addSignal* (*signalsInit2*  $s$ ) (*Running*  $A\ B\ (\text{Exp } gnx (\text{NonceF } (Rb\$ny)))$ )  
  else  
    *signalsInit2*  $s$   
 $\rangle$   
 $\}$

### definition

*skl1-step3* ::  
 $\text{rid-t} \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow \text{msg} \Rightarrow \text{msg} \Rightarrow ('a\ \text{skl1-state-scheme} * 'a\ \text{skl1-state-scheme})\ \text{set}$

### where

*skl1-step3*  $Ra\ A\ B\ Nr\ gny \equiv \{(s, s')\}$ .

— guards:

*guessed-runs*  $Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \wedge$   
*progress*  $s\ Ra = \text{Some } \{xnx, xni, xgnx\} \wedge$   
*guessed-frame*  $Ra\ xgny = \text{Some } gny \wedge$   
*guessed-frame*  $Ra\ xnr = \text{Some } Nr \wedge$   
*guessed-frame*  $Ra\ xsk = \text{Some } (\text{Exp } gny (\text{NonceF } (Ra\$nx))) \wedge$   
(*can-signal*  $s\ A\ B \longrightarrow$  — authentication guard  
 $(\exists Rb. \text{guessed-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \wedge$   
*in-progressS* (*progress*  $s\ Rb$ )  $\{xny, xni, xnr, xgnx, xgny, xsk\} \wedge$   
*guessed-frame*  $Rb\ xgny = \text{Some } gny \wedge$   
*guessed-frame*  $Rb\ xnr = \text{Some } Nr \wedge$   
*guessed-frame*  $Rb\ xni = \text{Some } (\text{NonceF } (Ra\$ni)) \wedge$   
*guessed-frame*  $Rb\ xgnx = \text{Some } (\text{Exp } \text{Gen } (\text{NonceF } (Ra\$nx)))) \wedge$   
 $(Ra = \text{test} \longrightarrow \text{Exp } gny (\text{NonceF } (Ra\$nx)) \notin \text{synth } (\text{analz } (ik\ s))) \wedge$

— actions:

$s' = s(\text{progress} := (\text{progress } s)(Ra \mapsto \{xnx, xni, xnr, xgnx, xgny, xsk, xEnd\}),$   
*secret* :=  $\{x. x = \text{Exp } gny (\text{NonceF } (Ra\$nx)) \wedge Ra = \text{test}\} \cup \text{secret } s,$   
*signalsInit* :=  
  if *can-signal*  $s\ A\ B$  then  
    *addSignal* (*signalsInit*  $s$ )  
    (*Commit*  $A\ B\ \langle \text{NonceF } (Ra\$ni), Nr, \text{Exp } gny (\text{NonceF } (Ra\$nx)) \rangle$ )  
  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*  $\Rightarrow$  *agent*  $\Rightarrow$  *agent*  $\Rightarrow$  *msg*  $\Rightarrow$  *msg*  $\Rightarrow$  ('a *skl1-state-scheme* \* 'a *skl1-state-scheme*) *set*

**where**

*skl1-step4* *Rb A B Ni gnz*  $\equiv$   $\{(s, s')\}$ .

— guards:

*guessed-runs Rb* =  $(\text{role}=\text{Resp}, \text{owner}=\text{B}, \text{partner}=\text{A}) \wedge$

*progress s Rb* =  $\text{Some} \{xny, xni, xnr, xgnx, xgny, xsk\} \wedge$

*guessed-frame Rb xgnx* =  $\text{Some gnz} \wedge$

*guessed-frame Rb xni* =  $\text{Some Ni} \wedge$

(*can-signal s A B*  $\longrightarrow$  — authentication guard

$(\exists \text{Ra. } \text{guessed-runs Ra} = (\text{role}=\text{Init}, \text{owner}=\text{A}, \text{partner}=\text{B}) \wedge$

*in-progressS* (*progress s Ra*)  $\{xnx, xni, xnr, xgnx, xgny, xsk, xEnd\} \wedge$

*guessed-frame Ra xgnx* =  $\text{Some gnz} \wedge$

*guessed-frame Ra xni* =  $\text{Some Ni} \wedge$

*guessed-frame Ra xnr* =  $\text{Some} (\text{NonceF} (\text{Rb}\$nr)) \wedge$

*guessed-frame Ra xgny* =  $\text{Some} (\text{Exp Gen} (\text{NonceF} (\text{Rb}\$ny)))) \wedge$

(*Rb* = *test*  $\longrightarrow$  *Exp gnz* (*NonceF* (*Rb*\$ny))  $\notin$  *synth* (*analz* (*ik s*)))  $\wedge$

— actions:

*s'* = *s* | *progress* := (*progress s*) (*Rb*  $\mapsto$   $\{xny, xni, xnr, xgnx, xgny, xsk, xEnd\}$ ),

*secret* :=  $\{x. x = \text{Exp gnz} (\text{NonceF} (\text{Rb}\$ny)) \wedge \text{Rb} = \text{test}\} \cup \text{secret } s,$

*signalsResp* :=

if *can-signal s A B* then

addSignal (*signalsResp s*)

(Commit A B (Ni, NonceF (Rb\$nr), Exp gnz (NonceF (Rb\$ny))))

else

signalsResp s,

*signalsResp2* :=

if *can-signal s A B* then

addSignal (*signalsResp2 s*) (Commit A B (Exp gnz (NonceF (Rb\$ny))))

else

signalsResp2 s

)

}

Specification.

**definition**

*skl1-trans* :: ('a *skl1-state-scheme* \* 'a *skl1-state-scheme*) *set* **where**  
*skl1-trans*  $\equiv$  ( $\bigcup m$  *Ra* *Rb* *A* *B* *x* *y*.  
  *skl1-step1* *Ra* *A* *B*  $\cup$   
  *skl1-step2* *Rb* *A* *B* *x* *y*  $\cup$   
  *skl1-step3* *Ra* *A* *B* *x* *y*  $\cup$   
  *skl1-step4* *Rb* *A* *B* *x* *y*  $\cup$   
  *l1-learn* *m*  $\cup$   
  *Id*  
)

**definition**

*skl1-init* :: *skl1-state* *set*

**where**

*skl1-init*  $\equiv$  { (  
  *ik* = {},  
  *secret* = {},  
  *progress* = *Map.empty*,  
  *signalsInit* =  $\lambda x. 0$ ,  
  *signalsResp* =  $\lambda x. 0$ ,  
  *signalsInit2* =  $\lambda x. 0$ ,  
  *signalsResp2* =  $\lambda x. 0$   
  )  
}

**definition**

*skl1* :: (*skl1-state*, *skl1-obs*) *spec* **where**  
*skl1*  $\equiv$  (  
  *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*

$\langle$ *proof* $\rangle$

**lemma** *run-ended-trans*:

*run-ended* (*progress s R*)  $\implies$   
(*s*, *s'*)  $\in$  *trans skl1*  $\implies$   
*run-ended* (*progress s' R*)

$\langle$ *proof* $\rangle$

**lemma** *can-signal-trans*:

*can-signal s' A B*  $\implies$   
*(s, s') ∈ trans skl1*  $\implies$   
*can-signal s A B*

*<proof>*

## 26.2 Refinement: secrecy

**fun** *option-inter* :: *var set*  $\Rightarrow$  *var set option*  $\Rightarrow$  *var set option*

**where**

*option-inter S (Some x) = Some (x ∩ S)*

*|option-inter S None = None*

**definition** *med-progress* :: *progress-t*  $\Rightarrow$  *progress-t*

**where**

*med-progress r*  $\equiv$   $\lambda R. \text{option-inter } \{xnx, xny, xgnx, xgny, xsk, xEnd\} (r R)$

**lemma** *med-progress-upd* [*simp*]:

*med-progress (r(R  $\mapsto$  S)) = (med-progress r) (R  $\mapsto$  S  $\cap$  {xnx, xny, xgnx, xgny, xsk, xEnd})*

*<proof>*

**lemma** *med-progress-Some*:

*r x = Some s*  $\implies$  *med-progress r x = Some (s  $\cap$  {xnx, xny, xgnx, xgny, xsk, xEnd})*

*<proof>*

**lemma** *med-progress-None* [*simp*]: *med-progress r x = None*  $\longleftrightarrow$  *r x = None*

*<proof>*

**lemma** *med-progress-Some2* [*dest*]:

*med-progress r x = Some y*  $\implies$   $\exists z. r x = Some z \wedge y = z \cap \{xnx, xny, xgnx, xgny, xsk, xEnd\}$

*<proof>*

**lemma** *med-progress-dom* [*simp*]: *dom (med-progress r) = dom r*

*<proof>*

**lemma** *med-progress-empty* [*simp*]: *med-progress Map.empty = Map.empty*

*<proof>*

Mediator function.

**definition**

*med11* :: *skl1-obs*  $\Rightarrow$  *l1-obs*

**where**

*med11 t*  $\equiv$  (*ik = ik t*,  
*secret = secret t*,  
*progress = med-progress (progress t)*,  
*signalsInit = signalsInit2 t*,  
*signalsResp = signalsResp2 t*)

relation between states

**definition**

*R11* :: (*l1-state* \* *skl1-state*) *set*

**where**

$$\begin{aligned}
R11 &\equiv \{(s, s') \\
&\quad s = \text{med11 } s' \\
&\quad \}
\end{aligned}$$

**lemmas**  $R11\text{-defs} = R11\text{-def } \text{med11}\text{-def}$

**lemma**  $\text{in-progress-med-progress}$ :

$$\begin{aligned}
&x \in \{xnx, xny, xgnx, xgny, xsk, xEnd\} \\
&\implies \text{in-progress } (\text{med-progress } r \ R) \ x \longleftrightarrow \text{in-progress } (r \ R) \ x \\
\langle \text{proof} \rangle
\end{aligned}$$

**lemma**  $\text{in-progressS-eq}$ :  $\text{in-progressS } S \ S' \longleftrightarrow (S \neq \text{None} \wedge (\forall x \in S'. \text{in-progress } S \ x))$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{in-progressS-med-progress}$ :

$$\begin{aligned}
&\text{in-progressS } (r \ R) \ S \\
&\implies \text{in-progressS } (\text{med-progress } r \ R) \ (S \cap \{xnx, xny, xgnx, xgny, xsk, xEnd\}) \\
\langle \text{proof} \rangle
\end{aligned}$$

**lemma**  $\text{can-signal-R11}$  [*simp*]:

$$\begin{aligned}
&(s1, s2) \in R11 \implies \\
&\quad \text{can-signal } s1 \ A \ B \longleftrightarrow \text{can-signal } s2 \ A \ B \\
\langle \text{proof} \rangle
\end{aligned}$$

Protocol-independent events.

**lemma**  $\text{skl1-learn-refines-learn}$ :

$$\begin{aligned}
&\{R11\} \ \text{l1-learn } m, \ \text{l1-learn } m \ \{>R11\} \\
\langle \text{proof} \rangle
\end{aligned}$$

Protocol events.

**lemma**  $\text{skl1-step1-refines-step1}$ :

$$\begin{aligned}
&\{R11\} \ \text{l1-step1 } Ra \ A \ B, \ \text{skl1-step1 } Ra \ A \ B \ \{>R11\} \\
\langle \text{proof} \rangle
\end{aligned}$$

**lemma**  $\text{skl1-step2-refines-step2}$ :

$$\begin{aligned}
&\{R11\} \ \text{l1-step2 } Rb \ A \ B \ gnx, \ \text{skl1-step2 } Rb \ A \ B \ Ni \ gnx \ \{>R11\} \\
\langle \text{proof} \rangle
\end{aligned}$$

**lemma**  $\text{skl1-step3-refines-step3}$ :

$$\begin{aligned}
&\{R11\} \ \text{l1-step3 } Ra \ A \ B \ gny, \ \text{skl1-step3 } Ra \ A \ B \ Nr \ gny \ \{>R11\} \\
\langle \text{proof} \rangle
\end{aligned}$$

**lemma**  $\text{skl1-step4-refines-step4}$ :

$$\begin{aligned}
&\{R11\} \ \text{l1-step4 } Rb \ A \ B \ gnx, \ \text{skl1-step4 } Rb \ A \ B \ Ni \ gnx \ \{>R11\} \\
\langle \text{proof} \rangle
\end{aligned}$$

Refinement proof.

**lemmas**  $\text{skl1-trans-refines-l1-trans} =$

$$\begin{aligned}
&\text{skl1-learn-refines-learn} \\
&\text{skl1-step1-refines-step1 } \text{skl1-step2-refines-step2} \\
&\text{skl1-step3-refines-step3 } \text{skl1-step4-refines-step4}
\end{aligned}$$



**lemma** *skl1-refines-init-l1* [iff]:  
 $init\ skl1 \subseteq R11 \text{ “ } (init\ l1)$   
 ⟨proof⟩

**lemma** *skl1-refines-trans-l1* [iff]:  
 $\{R11\} trans\ l1, trans\ skl1 \{> R11\}$   
 ⟨proof⟩

**lemma** *obs-consistent-med11* [iff]:  
 $obs-consistent\ R11\ med11\ l1\ skl1$   
 ⟨proof⟩

Refinement result.

**lemma** *skl1-refines-l1* [iff]:  
 $refines$   
 $R11$   
 $med11\ l1\ skl1$   
 ⟨proof⟩

**lemma** *skl1-implements-l1* [iff]:  $implements\ med11\ l1\ skl1$   
 ⟨proof⟩

## 26.3 Derived invariants: secrecy

**lemma** *skl1-obs-secrecy* [iff]:  $oreach\ skl1 \subseteq s0-secrecy$   
 ⟨proof⟩

**lemma** *skl1-secrecy* [iff]:  $reach\ skl1 \subseteq s0-secrecy$   
 ⟨proof⟩

## 26.4 Invariants: *Init* authenticates *Resp*

### 26.4.1 *inv1*

If an initiator commit signal exists for  $Ra\ \$\ ni, Nr, (g^{ny})Ra\ \$\ nx$ , then  $Ra$  is *Init*, has passed step 3, and has the nonce  $Nr$ , and  $(g \hat{ny}) \hat{\sim}(Ra\$nx)$  as the key in its frame.

#### definition

$skl1-inv1 :: skl1-state\ set$

#### where

$skl1-inv1 \equiv \{s. \forall Ra\ A\ B\ gny\ Nr.$   
 $signalsInit\ s\ (Commit\ A\ B\ \langle NonceF\ (Ra\$ni),\ Nr,\ Exp\ gny\ (NonceF\ (Ra\$nx)) \rangle) > 0 \longrightarrow$   
 $guessed-runs\ Ra = \langle role=Init,\ owner=A,\ partner=B \rangle \wedge$   
 $progress\ s\ Ra = Some\ \{xnx,\ xni,\ xnr,\ xgnx,\ xgny,\ xsk,\ xEnd\} \wedge$   
 $guessed-frame\ Ra\ xnr = Some\ Nr \wedge$   
 $guessed-frame\ Ra\ xsk = Some\ (Exp\ gny\ (NonceF\ (Ra\$nx)))$   
 $\}$

**lemmas**  $skl1-inv1I = skl1-inv1-def\ [THEN\ setc-def-to-intro,\ rule-format]$

**lemmas**  $skl1-inv1E [elim] = skl1-inv1-def\ [THEN\ setc-def-to-elim,\ rule-format]$

**lemmas**  $skl1\text{-}inv1D = skl1\text{-}inv1\text{-}def$  [*THEN setc-def-to-dest, rule-format, rotated 1, simplified*]

**lemma**  $skl1\text{-}inv1\text{-}init$  [*iff*]:

$init\ skl1 \subseteq skl1\text{-}inv1$

*<proof>*

**lemma**  $skl1\text{-}inv1\text{-}trans$  [*iff*]:

$\{skl1\text{-}inv1\} trans\ skl1 \{>\ skl1\text{-}inv1\}$

*<proof>*

**lemma**  $PO\text{-}skl1\text{-}inv1$  [*iff*]:  $reach\ skl1 \subseteq skl1\text{-}inv1$

*<proof>*

### 26.4.2 inv2

If a *Resp* run *Rb* has passed step 2 then (if possible) an initiator running signal has been emitted.

**definition**

$skl1\text{-}inv2 :: skl1\text{-}state\ set$

**where**

$skl1\text{-}inv2 \equiv \{s. \forall\ gnx\ A\ B\ Rb\ Ni.$

$guessed\text{-}runs\ Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \longrightarrow$

$in\text{-}progressS\ (\text{progress}\ s\ Rb)\ \{xny, xni, xnr, xgnx, xgny, xsk\} \longrightarrow$

$guessed\text{-}frame\ Rb\ xgnx = \text{Some}\ gnx \longrightarrow$

$guessed\text{-}frame\ Rb\ xni = \text{Some}\ Ni \longrightarrow$

$can\text{-}signal\ s\ A\ B \longrightarrow$

$signalsInit\ s\ (\text{Running}\ A\ B\ (Ni, \text{NonceF}\ (Rb\$nr), \text{Exp}\ gnx\ (\text{NonceF}\ (Rb\$ny)))) > 0$

$\}$

**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*]:

$init\ skl1 \subseteq skl1\text{-}inv2$

*<proof>*

**lemma**  $skl1\text{-}inv2\text{-}trans$  [*iff*]:

$\{skl1\text{-}inv2\} trans\ skl1 \{>\ skl1\text{-}inv2\}$

*<proof>*

**lemma**  $PO\text{-}skl1\text{-}inv2$  [*iff*]:  $reach\ skl1 \subseteq skl1\text{-}inv2$

*<proof>*

### 26.4.3 inv3 (derived)

If an *Init* run before step 3 and a *Resp* run after step 2 both know the same half-keys and nonces (more or less), then the number of *Init* running signals for the key is strictly greater than the number of *Init* commit signals. (actually, there are 0 commit and 1 running).

**definition**

$skl1\text{-}inv3 :: skl1\text{-}state\ set$

**where**

$skl1\text{-}inv3 \equiv \{s. \forall A B Rb Ra gny Nr.$   
 $\quad guessed\text{-}runs\ Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \longrightarrow$   
 $\quad in\text{-}progressS\ (\text{progress } s\ Rb) \{xny, xni, xnr, xgnx, xgny, xsk\} \longrightarrow$   
 $\quad guessed\text{-}frame\ Rb\ xgny = \text{Some } gny \longrightarrow$   
 $\quad guessed\text{-}frame\ Rb\ xnr = \text{Some } Nr \longrightarrow$   
 $\quad guessed\text{-}frame\ Rb\ xni = \text{Some } (\text{NonceF } (Ra\$ni)) \longrightarrow$   
 $\quad guessed\text{-}frame\ Rb\ xgnx = \text{Some } (\text{Exp Gen } (\text{NonceF } (Ra\$nx))) \longrightarrow$   
 $\quad guessed\text{-}runs\ Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \longrightarrow$   
 $\quad progress\ s\ Ra = \text{Some } \{xnx, xgnx, xni\} \longrightarrow$   
 $\quad can\text{-}signal\ s\ A\ B \longrightarrow$   
 $\quad \quad signalsInit\ s\ (\text{Commit } A\ B\ \langle \text{NonceF } (Ra\$ni), Nr, \text{Exp } gny\ (\text{NonceF } (Ra\$nx)) \rangle)$   
 $\quad \quad < \text{signalsInit } s\ (\text{Running } A\ B\ \langle \text{NonceF } (Ra\$ni), Nr, \text{Exp } gny\ (\text{NonceF } (Ra\$nx)) \rangle)$   
 $\quad \left. \right\}$

**lemmas**  $skl1\text{-}inv3I = skl1\text{-}inv3\text{-}def\ [THEN\ setc\text{-}def\text{-}to\text{-}intro,\ rule\text{-}format]$

**lemmas**  $skl1\text{-}inv3E\ [elim] = skl1\text{-}inv3\text{-}def\ [THEN\ setc\text{-}def\text{-}to\text{-}elim,\ rule\text{-}format]$

**lemmas**  $skl1\text{-}inv3D = skl1\text{-}inv3\text{-}def\ [THEN\ setc\text{-}def\text{-}to\text{-}dest,\ rule\text{-}format,\ rotated\ 1,\ simplified]$

**lemma**  $skl1\text{-}inv3\text{-}derived: skl1\text{-}inv1 \cap skl1\text{-}inv2 \subseteq skl1\text{-}inv3$

$\langle proof \rangle$

## 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**

$skl1\text{-}inv4 :: skl1\text{-}state\ set$

**where**

$skl1\text{-}inv4 \equiv \{s. \forall Rb A B gnx Ni.$   
 $\quad signalsResp\ s\ (\text{Commit } A\ B\ \langle Ni, \text{NonceF } (Rb\$nr), \text{Exp } gnx\ (\text{NonceF } (Rb\$ny)) \rangle) > 0 \longrightarrow$   
 $\quad guessed\text{-}runs\ Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \wedge$   
 $\quad progress\ s\ Rb = \text{Some } \{xny, xni, xnr, xgnx, xgny, xsk, xEnd\} \wedge$   
 $\quad guessed\text{-}frame\ Rb\ xgnx = \text{Some } gnx \wedge$   
 $\quad guessed\text{-}frame\ Rb\ xni = \text{Some } Ni$   
 $\quad \left. \right\}$

**lemmas**  $skl1\text{-}inv4I = skl1\text{-}inv4\text{-}def\ [THEN\ setc\text{-}def\text{-}to\text{-}intro,\ rule\text{-}format]$

**lemmas**  $skl1\text{-}inv4E\ [elim] = skl1\text{-}inv4\text{-}def\ [THEN\ setc\text{-}def\text{-}to\text{-}elim,\ rule\text{-}format]$

**lemmas**  $skl1\text{-}inv4D = skl1\text{-}inv4\text{-}def\ [THEN\ setc\text{-}def\text{-}to\text{-}dest,\ rule\text{-}format,\ rotated\ 1,\ simplified]$

**lemma**  $skl1\text{-}inv4\text{-}init\ [iff]:$

$init\ skl1 \subseteq skl1\text{-}inv4$

$\langle proof \rangle$

**lemma**  $skl1\text{-}inv4\text{-}trans\ [iff]:$

$\{skl1\text{-}inv4\}\ trans\ skl1 \{> skl1\text{-}inv4\}$

$\langle \text{proof} \rangle$

**lemma** *PO-skl1-inv4* [iff]:  $\text{reach skl1} \subseteq \text{skl1-inv4}$

$\langle \text{proof} \rangle$

### 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**

$\text{skl1-inv5} \equiv \{s. \forall \text{ gny } A \ B \ Ra \ Nr.$   
   $\text{guessed-runs } Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \longrightarrow$   
   $\text{in-progressS } (\text{progress } s \ Ra) \ \{xnx, xni, xnr, xgnx, xgny, xsk, xEnd\} \longrightarrow$   
   $\text{guessed-frame } Ra \ xgny = \text{Some } \text{gny} \longrightarrow$   
   $\text{guessed-frame } Ra \ xnr = \text{Some } \text{Nr} \longrightarrow$   
   $\text{can-signal } s \ A \ B \longrightarrow$   
   $\text{signalsResp } s \ (\text{Running } A \ B \ \langle \text{NonceF } (Ra\$ni), \text{Nr}, \text{Exp } \text{gny} \ (\text{NonceF } (Ra\$nx)) \rangle) > 0$   
   $\}$

**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]:

$\text{init skl1} \subseteq \text{skl1-inv5}$

$\langle \text{proof} \rangle$

**lemma** *skl1-inv5-trans* [iff]:

$\{\text{skl1-inv5}\} \text{ trans skl1 } \{> \text{skl1-inv5}\}$

$\langle \text{proof} \rangle$

**lemma** *PO-skl1-inv5* [iff]:  $\text{reach skl1} \subseteq \text{skl1-inv5}$

$\langle \text{proof} \rangle$

### 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**

$\text{skl1-inv6} \equiv \{s. \forall \ A \ B \ Rb \ Ra \ gnx \ Ni.$   
   $\text{guessed-runs } Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \longrightarrow$   
   $\text{in-progressS } (\text{progress } s \ Ra) \ \{xnx, xni, xnr, xgnx, xgny, xsk, xEnd\} \longrightarrow$   
   $\text{guessed-frame } Ra \ xgnx = \text{Some } \text{gnx} \longrightarrow$   
   $\text{guessed-frame } Ra \ xni = \text{Some } \text{Ni} \longrightarrow$   
   $\text{guessed-frame } Ra \ xgny = \text{Some } (\text{Exp } \text{Gen } (\text{NonceF } (Rb\$ny))) \longrightarrow$   
   $\text{guessed-frame } Ra \ xnr = \text{Some } (\text{NonceF } (Rb\$nr)) \longrightarrow$

$$\begin{aligned}
& \text{guessed-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \longrightarrow \\
& \text{progress } s \ Rb = \text{Some } \{xny, xni, xnr, xgnx, xgny, xsk\} \longrightarrow \\
& \text{can-signal } s \ A \ B \longrightarrow \\
& \quad \text{signalsResp } s \ (\text{Commit } A \ B \ \langle Ni, \text{NonceF } (Rb\$nr), \text{Exp } gnx \ (\text{NonceF } (Rb\$ny)) \rangle) \\
& \quad < \text{signalsResp } s \ (\text{Running } A \ B \ \langle Ni, \text{NonceF } (Rb\$nr), \text{Exp } gnx \ (\text{NonceF } (Rb\$ny)) \rangle) \\
& \quad \}
\end{aligned}$$

**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*:

$$\text{skl1-inv4} \cap \text{skl1-inv5} \subseteq \text{skl1-inv6}$$

*<proof>*

## 26.6 Refinement: injective agreement (Init authenticates Resp)

Mediator function.

**definition**

$$\text{med0sk1iai} :: \text{skl1-obs} \Rightarrow \text{a0i-obs}$$

**where**

$$\text{med0sk1iai } t \equiv (\text{a0n-state.signals} = \text{signalsInit } t)$$

Relation between states.

**definition**

$$R0sk1iai :: (\text{a0i-state} * \text{skl1-state}) \text{ set}$$

**where**

$$\begin{aligned}
R0sk1iai & \equiv \{(s, s') \\
& \quad \text{a0n-state.signals } s = \text{signalsInit } s' \\
& \quad \}
\end{aligned}$$

Protocol-independent events.

**lemma** *skl1-learn-refines-a0-ia-skip-i*:

$$\{R0sk1iai\} \text{ Id, l1-learn } m \ \{>R0sk1iai\}$$

*<proof>*

Protocol events.

**lemma** *skl1-step1-refines-a0i-skip-i*:

$$\{R0sk1iai\} \text{ Id, skl1-step1 } Ra \ A \ B \ \{>R0sk1iai\}$$

*<proof>*

**lemma** *skl1-step2-refines-a0i-running-skip-i*:

$$\begin{aligned}
\{R0sk1iai\} \ \text{a0i-running } A \ B \ \langle Ni, \text{NonceF } (Rb\$nr), \text{Exp } gnx \ (\text{NonceF } (Rb\$ny)) \rangle \cup \text{Id,} \\
\text{skl1-step2 } Rb \ A \ B \ Ni \ gnx \ \{>R0sk1iai\}
\end{aligned}$$

*<proof>*

**lemma** *skl1-step3-refines-a0i-commit-skip-i*:

$$\begin{aligned}
\{R0sk1iai \cap (\text{UNIV} \times \text{skl1-inv3})\} \\
\text{a0i-commit } A \ B \ \langle \text{NonceF } (Ra\$ni), Nr, \text{Exp } gny \ (\text{NonceF } (Ra\$nx)) \rangle \cup \text{Id,} \\
\text{skl1-step3 } Ra \ A \ B \ Nr \ gny
\end{aligned}$$

$\{>R0sk1iai\}$   
 $\langle proof \rangle$

**lemma** *skl1-step4-refines-a0i-skip-i*:  
 $\{R0sk1iai\} Id, skl1-step4 Rb A B Ni gnx \{>R0sk1iai\}$   
 $\langle proof \rangle$

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* “ (*init a0i*)  
 $\langle proof \rangle$

**lemma** *skl1-refines-trans-a0i-i [iff]*:  
 $\{R0sk1iai \cap (UNIV \times (skl1-inv1 \cap skl1-inv2))\}$  *trans a0i, trans skl1*  $\{> R0sk1iai\}$   
 $\langle proof \rangle$

**lemma** *obs-consistent-med01iai [iff]*:  
*obs-consistent R0sk1iai med0sk1iai a0i skl1*  
 $\langle proof \rangle$

refinement result

**lemma** *skl1-refines-a0i-i [iff]*:  
*refines*  
 $(R0sk1iai \cap (reach a0i \times (skl1-inv1 \cap skl1-inv2)))$   
*med0sk1iai a0i skl1*  
 $\langle proof \rangle$

**lemma** *skl1-implements-a0i-i [iff]*: *implements med0sk1iai a0i skl1*  
 $\langle proof \rangle$

## 26.7 Derived invariants: injective agreement (*Init* authenticates *Resp*)

**lemma** *skl1-obs-ia-greement-Init [iff]*: *oreach skl1*  $\subseteq$  *l1-ia-greement-Init*  
 $\langle proof \rangle$

**lemma** *skl1-ia-greement-Init [iff]*: *reach skl1*  $\subseteq$  *l1-ia-greement-Init*  
 $\langle proof \rangle$

## 26.8 Refinement: injective agreement (*Resp* authenticates *Init*)

Mediator function.

**definition**  
*med0sk1iar* :: *skl1-obs*  $\Rightarrow$  *a0i-obs*  
**where**  
*med0sk1iar t*  $\equiv$  (*a0n-state.signals* = *signalsResp t*)

Relation between states.

**definition**

$R0sk1iar :: (a0i\text{-state} * skl1\text{-state}) \text{ set}$

**where**

$R0sk1iar \equiv \{(s, s') \mid$   
 $a0n\text{-state.signals } s = \text{signalsResp } s'$   
 $\}$

Protocol independent events.

**lemma** *skl1-learn-refines-a0-ia-skip-r*:

$\{R0sk1iar\} Id, l1\text{-learn } m \{>R0sk1iar\}$   
 $\langle \text{proof} \rangle$

Protocol events.

**lemma** *skl1-step1-refines-a0i-skip-r*:

$\{R0sk1iar\} Id, skl1\text{-step1 } Ra A B \{>R0sk1iar\}$   
 $\langle \text{proof} \rangle$

**lemma** *skl1-step2-refines-a0i-skip-r*:

$\{R0sk1iar\} Id, skl1\text{-step2 } Rb A B Ni gnx \{>R0sk1iar\}$   
 $\langle \text{proof} \rangle$

**lemma** *skl1-step3-refines-a0i-running-skip-r*:

$\{R0sk1iar\}$   
 $a0i\text{-running } A B \langle \text{NonceF } (Ra\$ni), Nr, \text{Exp gny } (\text{NonceF } (Ra\$nx)) \rangle \cup Id,$   
 $skl1\text{-step3 } Ra A B Nr gny$   
 $\{>R0sk1iar\}$   
 $\langle \text{proof} \rangle$

**lemma** *skl1-step4-refines-a0i-commit-skip-r*:

$\{R0sk1iar \cap UNIV \times skl1\text{-inv6}\}$   
 $a0i\text{-commit } A B \langle Ni, \text{NonceF } (Rb\$nr), \text{Exp gnx } (\text{NonceF } (Rb\$ny)) \rangle \cup Id,$   
 $skl1\text{-step4 } Rb A B Ni gnx$   
 $\{>R0sk1iar\}$   
 $\langle \text{proof} \rangle$

Refinement proof.

**lemmas** *skl1-trans-refines-a0i-trans-r* =

$skl1\text{-learn-refines-a0-ia-skip-r}$   
 $skl1\text{-step1-refines-a0i-skip-r } skl1\text{-step2-refines-a0i-skip-r}$   
 $skl1\text{-step3-refines-a0i-running-skip-r } skl1\text{-step4-refines-a0i-commit-skip-r}$

**lemma** *skl1-refines-init-a0i-r* [iff]:

$init\ skl1 \subseteq R0sk1iar$  “  $(init\ a0i)$   
 $\langle \text{proof} \rangle$

**lemma** *skl1-refines-trans-a0i-r* [iff]:

$\{R0sk1iar \cap (UNIV \times (skl1\text{-inv4} \cap skl1\text{-inv5}))\} \text{trans } a0i, \text{trans } skl1 \{> R0sk1iar\}$   
 $\langle \text{proof} \rangle$

**lemma** *obs-consistent-med0sk1iar* [iff]:  
  *obs-consistent R0sk1iar med0sk1iar a0i skl1*  
  ⟨proof⟩

Refinement result.

**lemma** *skl1-refines-a0i-r* [iff]:  
  *refines*  
  (*R0sk1iar*  $\cap$  (*reach a0i*  $\times$  (*skl1-inv4*  $\cap$  *skl1-inv5*)))  
  *med0sk1iar a0i skl1*  
  ⟨proof⟩

**lemma** *skl1-implements-a0i-r* [iff]: *implements med0sk1iar a0i skl1*  
  ⟨proof⟩

## 26.9 Derived invariants: injective agreement (*Resp* authenticates *Init*)

**lemma** *skl1-obs-iagreement-Resp* [iff]: *oreach skl1*  $\subseteq$  *l1-iagreement-Resp*  
  ⟨proof⟩

**lemma** *skl1-iagreement-Resp* [iff]: *reach skl1*  $\subseteq$  *l1-iagreement-Resp*  
  ⟨proof⟩

**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  
  <proof>
```

Level 2 state.

```
record l2-state =
```

```
  skl1-state +  
  chan :: chan set  
  bad :: agent set
```

```
type-synonym l2-obs = l2-state
```

```
type-synonym
```

```
  l2-pred = l2-state set
```

```
type-synonym
```

```
  l2-trans = (l2-state  $\times$  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*

$\langle$ *proof* $\rangle$

**lemma** *role-comp-inv-eq*:

*y* = *role-comp* *x*  $\longleftrightarrow$  *x* = *role-comp* *y*

$\langle$ *proof* $\rangle$

**definition**

*partners* :: *rid-t* *set*

**where**

*partners*  $\equiv \{R. \textit{partner-runs } \textit{test } R\}$

**lemma** *test-not-partner* [*simp*]:

*test*  $\notin$  *partners*

$\langle$ *proof* $\rangle$

**lemma** *matching-symmetric*:

*matching* *sigma* *sigma'*  $\Longrightarrow$  *matching* *sigma'* *sigma*

$\langle$ *proof* $\rangle$

**lemma** *partner-symmetric*:

*partner-runs* *R* *R'*  $\Longrightarrow$  *partner-runs* *R'* *R*

$\langle$ *proof* $\rangle$

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⟩

**lemma** *partner-test*:

$R \in partners \implies partner\text{-runs } R R' \implies R' = test$   
 ⟨proof⟩

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  
 $s' = s(\text{bad} := \{A\} \cup \text{bad } s)$   
 }

**definition**

$l2\text{-lkr}\text{-actor} :: agent \Rightarrow l2\text{-trans}$

**where**

$l2\text{-lkr}\text{-actor } A \equiv \{(s, s')\}.$   
 — guards  
 $A = test\text{-owner} \wedge$   
 $A \neq test\text{-partner} \wedge$   
 — actions  
 $s' = s(\text{bad} := \{A\} \cup \text{bad } s)$   
 }

**definition**

$l2\text{-lkr}\text{-after} :: agent \Rightarrow l2\text{-trans}$

**where**

$l2\text{-lkr}\text{-after } A \equiv \{(s, s')\}.$   
 — guards  
 $test\text{-ended } s \wedge$   
 — actions  
 $s' = s(\text{bad} := \{A\} \cup \text{bad } s)$   
 }

**definition**

$l2\text{-skr} :: rid\text{-t} \Rightarrow msg \Rightarrow l2\text{-trans}$

**where**

$l2\text{-skr } R K \equiv \{(s, s')\}.$   
 — guards  
 $R \neq test \wedge R \notin partners \wedge$   
 $in\text{-progress } (progress\ s\ R)\ xsk \wedge$   
 $guessed\text{-frame } R\ xsk = Some\ K \wedge$   
 — actions  
 $s' = s(\text{ik} := \{K\} \cup \text{ik } s)$   
 }

Protocol events (with  $K = H(ni, nr)$ ):

- step 1: create  $Ra$ ,  $A$  generates  $nx$  and  $ni$ , confidentially sends  $ni$ , computes and insecurely sends  $g^{nx}$
- step 2: create  $Rb$ ,  $B$  receives  $ni$  (confidentially) and  $g^{nx}$  (insecurely), generates  $ny$  and  $nr$ , confidentially sends  $nr$ , insecurely sends  $g^{ny}$  and  $MAC_K(g^{nx}, g^{ny}, B, A)$  computes  $g^{nx * ny}$ , emits a running signal for  $Init, ni, nr, g^{nx * ny}$
- step 3:  $A$  receives  $nr$  confidentially, and  $g^{ny}$  and the MAC insecurely, sends  $MAC_K(g^{ny}, g^{nx}, A, B)$  insecurely, computes  $g^{ny * nx}$ , emits a commit signal for  $Init, ni, nr, g^{ny * nx}$ , a running signal for  $Resp, ni, nr, g^{ny * nx}$ , declares the secret  $g^{ny * nx}$
- step 4:  $B$  receives the MAC insecurely, emits a commit signal for  $Resp, ni, nr, g^{nx * ny}$ , declares the secret  $g^{nx * ny}$

**definition**

$l2\text{-step1} :: rid\text{-}t \Rightarrow agent \Rightarrow agent \Rightarrow l2\text{-}trans$

**where**

$l2\text{-step1 } Ra \ A \ B \equiv \{(s, s')\}.$

— guards:

$Ra \notin dom \ (progress \ s) \wedge$   
 $guessed\text{-}runs \ Ra = \langle role=Init, owner=A, partner=B \rangle \wedge$

— actions:

$s' = s \langle$   
 $progress := (progress \ s)(Ra \mapsto \{xnx, xni, xgnx\}),$   
 $chan := \{Confid \ A \ B \ (NonceF \ (Ra\$ni))\} \cup$   
 $\{\{Insec \ A \ B \ (Exp \ Gen \ (NonceF \ (Ra\$nx)))\}\} \cup$   
 $(chan \ s)$

$\rangle$   
 $\}$

**definition**

$l2\text{-step2} :: rid\text{-}t \Rightarrow agent \Rightarrow agent \Rightarrow msg \Rightarrow msg \Rightarrow l2\text{-}trans$

**where**

$l2\text{-step2 } Rb \ A \ B \ Ni \ gnx \equiv \{(s, s')\}.$

— guards:

$guessed\text{-}runs \ Rb = \langle role=Resp, owner=B, partner=A \rangle \wedge$   
 $Rb \notin dom \ (progress \ s) \wedge$   
 $guessed\text{-}frame \ Rb \ xgnx = Some \ gnx \wedge$   
 $guessed\text{-}frame \ Rb \ xni = Some \ Ni \wedge$   
 $guessed\text{-}frame \ Rb \ xsk = Some \ (Exp \ gnx \ (NonceF \ (Rb\$ny))) \wedge$   
 $Confid \ A \ B \ Ni \in chan \ s \wedge$   
 $Insec \ A \ B \ gnx \in chan \ s \wedge$

— actions:

$s' = s \langle$   $progress := (progress \ s)(Rb \mapsto \{xny, xni, xnr, xgny, xgnx, xsk\}),$   
 $chan := \{Confid \ B \ A \ (NonceF \ (Rb\$nr))\} \cup$   
 $\{\{Insec \ B \ A$   
 $\langle Exp \ Gen \ (NonceF \ (Rb\$ny)),$   
 $hmac \ \langle Number \ 0, \ gnx, \ Exp \ Gen \ (NonceF \ (Rb\$ny)), \ Agent \ B, \ Agent \ A \rangle$   
 $(Hash \ \langle Ni, \ NonceF \ (Rb\$nr) \rangle)\} \} \cup$   
 $(chan \ s),$   
 $signalsInit :=$

```

    if can-signal s A B then
      addSignal (signalsInit s)
        (Running A B ⟨Ni, NonceF (Rb$nr), Exp gnx (NonceF (Rb$ny))⟩)
    else
      signalsInit s,
signalsInit2 :=
  if can-signal s A B then
    addSignal (signalsInit2 s) (Running A B (Exp gnx (NonceF (Rb$ny))))
  else
    signalsInit2 s
}

```

**definition**

$l2\text{-step3} :: \text{rid-t} \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow \text{msg} \Rightarrow \text{msg} \Rightarrow l2\text{-trans}$

**where**

$l2\text{-step3 Ra A B Nr gny} \equiv \{(s, s')\}.$

— guards:

$\text{guessed-runs Ra} = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \wedge$

$\text{progress s Ra} = \text{Some } \{xnx, xni, xgnx\} \wedge$

$\text{guessed-frame Ra xgny} = \text{Some gny} \wedge$

$\text{guessed-frame Ra xnr} = \text{Some Nr} \wedge$

$\text{guessed-frame Ra xsk} = \text{Some } (\text{Exp gny } (\text{NonceF } (Ra\$nx))) \wedge$

$\text{Confid B A Nr} \in \text{chan s} \wedge$

$\text{Insec B A } \langle \text{gny}, \text{hmac } \langle \text{Number } 0, \text{Exp Gen } (\text{NonceF } (Ra\$nx)), \text{gny}, \text{Agent B}, \text{Agent A} \rangle$

$(\text{Hash } \langle \text{NonceF } (Ra\$ni), \text{Nr} \rangle) \rangle \in \text{chan s} \wedge$

— actions:

$s' = s \{ \text{progress} := (\text{progress s})(Ra \mapsto \{xnx, xni, xnr, xgnx, xgny, xsk, xEnd\}),$

$\text{chan} := \{ \text{Insec A B}$

$(\text{hmac } \langle \text{Number } 1, \text{gny}, \text{Exp Gen } (\text{NonceF } (Ra\$nx)), \text{Agent A}, \text{Agent B} \rangle$

$(\text{Hash } \langle \text{NonceF } (Ra\$ni), \text{Nr} \rangle) \}$

$\cup \text{chan s},$

$\text{secret} := \{x. x = \text{Exp gny } (\text{NonceF } (Ra\$nx)) \wedge Ra = \text{test}\} \cup \text{secret s},$

$\text{signalsInit} :=$

$\text{if can-signal s A B then}$

$\text{addSignal } (\text{signalsInit s})$

$(\text{Commit A B } \langle \text{NonceF } (Ra\$ni), \text{Nr}, \text{Exp gny } (\text{NonceF } (Ra\$nx))) \rangle)$

$\text{else}$

$\text{signalsInit s},$

$\text{signalsInit2} :=$

$\text{if can-signal s A B then}$

$\text{addSignal } (\text{signalsInit2 s}) (\text{Commit A B } (\text{Exp gny } (\text{NonceF } (Ra\$nx))))$

$\text{else}$

$\text{signalsInit2 s},$

$\text{signalsResp} :=$

$\text{if can-signal s A B then}$

$\text{addSignal } (\text{signalsResp s})$

$(\text{Running A B } \langle \text{NonceF } (Ra\$ni), \text{Nr}, \text{Exp gny } (\text{NonceF } (Ra\$nx))) \rangle)$

$\text{else}$

$\text{signalsResp s},$

$\text{signalsResp2} :=$

$\text{if can-signal s A B then}$

```

      addSignal (signalsResp2 s) (Running A B (Exp gny (NonceF (Ra$nx))))
    else
      signalsResp2 s
  }
}

```

**definition**

$l2\text{-step4} :: \text{rid-t} \Rightarrow \text{agent} \Rightarrow \text{agent} \Rightarrow \text{msg} \Rightarrow \text{msg} \Rightarrow l2\text{-trans}$

**where**

$l2\text{-step4} \text{ Rb A B Ni gn}x \equiv \{(s, s')\}.$

— guards:

$\text{guessed-runs Rb} = (\text{role}=\text{Resp}, \text{owner}=\text{B}, \text{partner}=\text{A}) \wedge$

$\text{progress s Rb} = \text{Some } \{xny, xni, xnr, xgnx, xgny, xsk\} \wedge$

$\text{guessed-frame Rb xgn}x = \text{Some gn}x \wedge$

$\text{guessed-frame Rb xni} = \text{Some Ni} \wedge$

$\text{Insec A B (hmac } \langle \text{Number 1, Exp Gen (NonceF (Rb\$ny)), gn}x, \text{Agent A, Agent B} \rangle$   
 $\quad (\text{Hash } \langle \text{Ni, NonceF (Rb\$nr)} \rangle)) \in \text{chan s} \wedge$

— actions:

$s' = s \{ \text{progress} := (\text{progress s})(\text{Rb} \mapsto \{xny, xni, xnr, xgnx, xgny, xsk, xEnd\}),$

$\text{secret} := \{x. x = \text{Exp gn}x (\text{NonceF (Rb\$ny)}) \wedge \text{Rb} = \text{test}\} \cup \text{secret s},$

$\text{signalsResp} :=$

$\text{if can-signal s A B then}$

$\text{addSignal (signalsResp s)}$

$\quad (\text{Commit A B } \langle \text{Ni, NonceF (Rb\$nr)}, \text{Exp gn}x (\text{NonceF (Rb\$ny)}) \rangle)$

$\text{else}$

$\text{signalsResp s},$

$\text{signalsResp2} :=$

$\text{if can-signal s A B then}$

$\text{addSignal (signalsResp2 s) (Commit A B (Exp gn}x (\text{NonceF (Rb\$ny)}))$

$\text{else}$

$\text{signalsResp2 s}$

```

  }
}

```

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-trans :: l2-trans where
l2-trans ≡ (⋃ m M X Rb Ra A B K Y.
  l2-step1 Ra A B ∪
  l2-step2 Rb A B X Y ∪
  l2-step3 Ra A B X Y ∪
  l2-step4 Rb A B X Y ∪
  l2-dy-fake-chan M ∪
  l2-dy-fake-msg m ∪
  l2-lkr-others A ∪
  l2-lkr-after A ∪
  l2-skr Ra K ∪
  Id
)

```

**definition**

```

l2 :: (l2-state, l2-obs) spec where
l2 ≡ ⟨
  init = l2-init,
  trans = l2-trans,
  obs = id
⟩

```

**lemmas** *l2-loc-defs* =

```

l2-step1-def l2-step2-def l2-step3-def l2-step4-def
l2-def l2-init-def l2-trans-def
l2-dy-fake-chan-def l2-dy-fake-msg-def
l2-lkr-after-def l2-lkr-others-def l2-skr-def

```

**lemmas** *l2-defs* = *l2-loc-defs ik-dy-def*

**lemmas** *l2-nostep-defs* = *l2-def l2-init-def l2-trans-def*

**lemmas** *l2-step-defs* =

```

l2-step1-def l2-step2-def l2-step3-def l2-step4-def
l2-dy-fake-chan-def l2-dy-fake-msg-def l2-lkr-after-def l2-lkr-others-def l2-skr-def

```

**lemma** *l2-obs-id* [*simp*]: *obs l2* = *id*

⟨*proof*⟩

Once a run is finished, it stays finished, therefore if the test is not finished at some point then it was not finished before either.

**declare** *domIff* [*iff*]

**lemma** *l2-run-ended-trans*:

```

run-ended (progress s R) ⇒
  (s, s') ∈ trans l2 ⇒
  run-ended (progress s' R)

```

⟨*proof*⟩

**declare** *domIff* [*iff del*]

**lemma** *l2-can-signal-trans*:

```

can-signal s' A B ⇒
  (s, s') ∈ trans l2 ⇒
  can-signal s A B

```

$\langle \text{proof} \rangle$

**lemma** *in-progressS-trans*:

$\text{in-progressS } (\text{progress } s \ R) \ S \implies (s, s') \in \text{trans } l2 \implies \text{in-progressS } (\text{progress } s' \ R) \ S$

$\langle \text{proof} \rangle$

## 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**

$l2\text{-inv1} :: l2\text{-state set}$

**where**

$l2\text{-inv1} \equiv \{s. \forall A \ B. \\ \text{can-signal } s \ A \ B \longrightarrow \\ A \notin \text{bad } s \wedge B \notin \text{bad } s \\ \}$

**lemmas**  $l2\text{-inv1I} = l2\text{-inv1-def} \ [THEN \ \text{setc-def-to-intro}, \ \text{rule-format}]$

**lemmas**  $l2\text{-inv1E} \ [elim] = l2\text{-inv1-def} \ [THEN \ \text{setc-def-to-elim}, \ \text{rule-format}]$

**lemmas**  $l2\text{-inv1D} = l2\text{-inv1-def} \ [THEN \ \text{setc-def-to-dest}, \ \text{rule-format}, \ \text{rotated } 1, \ \text{simplified}]$

**lemma**  $l2\text{-inv1-init} \ [iff]$ :

$\text{init } l2 \subseteq l2\text{-inv1}$

$\langle \text{proof} \rangle$

**lemma**  $l2\text{-inv1-trans} \ [iff]$ :

$\{l2\text{-inv1}\} \ \text{trans } l2 \ \{> \ l2\text{-inv1}\}$

$\langle \text{proof} \rangle$

**lemma**  $PO\text{-}l2\text{-inv1} \ [iff]$ :  $\text{reach } l2 \subseteq l2\text{-inv1}$

$\langle \text{proof} \rangle$

### 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**

$l2\text{-inv2} :: l2\text{-state set}$

**where**

$l2\text{-inv2} \equiv \{s. \forall R. \\ \text{in-progress } (\text{progress } s \ R) \ xsk \longrightarrow \\ (\exists \ X \ N. \\ \text{guessed-frame } R \ xsk = \text{Some } (\text{Exp } X \ (\text{NonceF } (R\ \$N)))) \\ \}$

**lemmas**  $l2\text{-inv2I} = l2\text{-inv2-def} \ [THEN \ \text{setc-def-to-intro}, \ \text{rule-format}]$

**lemmas**  $l2\text{-inv2E} \ [elim] = l2\text{-inv2-def} \ [THEN \ \text{setc-def-to-elim}, \ \text{rule-format}]$

**lemmas**  $l2\text{-inv2D} = l2\text{-inv2-def} \ [THEN \ \text{setc-def-to-dest}, \ \text{rule-format}, \ \text{rotated } 1, \ \text{simplified}]$



**lemma** *l2-inv2-init* [iff]:

$init\ l2 \subseteq l2\text{-inv2}$

$\langle proof \rangle$

**lemma** *l2-inv2-trans* [iff]:

$\{l2\text{-inv2}\} trans\ l2\ \{>\ l2\text{-inv2}\}$

$\langle proof \rangle$

**lemma** *PO-l2-inv2* [iff]:  $reach\ l2 \subseteq l2\text{-inv2}$

$\langle proof \rangle$

### 27.2.3 inv3

**definition**

$bad\text{-runs}\ s = \{R. owner\ (guessed\text{-runs}\ R) \in bad\ s \vee partner\ (guessed\text{-runs}\ R) \in bad\ s\}$

**abbreviation**

$generators :: l2\text{-state} \Rightarrow msg\ set$

**where**

$generators\ s \equiv$

— from the *insec* messages in steps 1 2

$\{x. \exists N. x = Exp\ Gen\ (Nonce\ N)\} \cup$

— from the opened *confid* messages in steps 1 2

$\{x. \exists R \in bad\text{-runs}\ s. x = NonceF\ (R\$ni) \vee x = NonceF\ (R\$nr)\} \cup$

— from the *insec* messages in steps 2 3

$\{x. \exists y\ y'\ z. x = hmac\ \langle y, y' \rangle\ (Hash\ z)\} \cup$

— from the *skr*

$\{Exp\ y\ (NonceF\ (R\$N)) \mid y\ N\ R. R \neq test \wedge R \notin partners\}$

**lemma** *analz-generators*:  $analz\ (generators\ s) = generators\ s$

$\langle proof \rangle$

**definition**

$faked\text{-chan}\text{-msgs} :: l2\text{-state} \Rightarrow chan\ set$

**where**

$faked\text{-chan}\text{-msgs}\ s =$

$\{Chan\ x\ A\ B\ M \mid x\ A\ B\ M. M \in synth\ (analz\ (extr\ (bad\ s)\ (ik\ s)\ (chan\ s)))\}$

**definition**

$chan\text{-generators} :: chan\ set$

**where**

$chan\text{-generators} = \{x. \exists n\ R. \text{— the messages that can't be opened}$

$x = Confid\ (owner\ (guessed\text{-runs}\ R))\ (partner\ (guessed\text{-runs}\ R))\ (NonceF\ (R\$n)) \wedge$

$(n = ni \vee n = nr)$

$\}$

**definition**

$l2\text{-inv3} :: l2\text{-state}\ set$

**where**

$l2\text{-inv3} \equiv \{s.$

$extr\ (bad\ s)\ (ik\ s)\ (chan\ s) \subseteq synth\ (analz\ (generators\ s)) \wedge$

$chan\ s \subseteq faked\ chan\ msgs\ s \cup chan\ generators$   
 $\}$

**lemmas**  $l2\ inv3\ aux\ defs = faked\ chan\ msgs\ def\ chan\ generators\ def$

**lemmas**  $l2\ inv3I = l2\ inv3\ def\ [THEN\ setc\ def\ to\ intro,\ rule\ format]$

**lemmas**  $l2\ inv3E = l2\ inv3\ def\ [THEN\ setc\ def\ to\ elim,\ rule\ format]$

**lemmas**  $l2\ inv3D = l2\ inv3\ def\ [THEN\ setc\ def\ to\ dest,\ rule\ format,\ rotated\ 1,\ simplified]$

**lemma**  $l2\ inv3\ init\ [iff]:$

$init\ l2 \subseteq l2\ inv3$

$\langle proof \rangle$

**lemma**  $l2\ inv3\ step1:$

$\{l2\ inv3\}\ l2\ step1\ Ra\ A\ B\ \{>\ l2\ inv3\}$

$\langle proof \rangle$

**lemma**  $l2\ inv3\ step2:$

$\{l2\ inv3\}\ l2\ step2\ Rb\ A\ B\ Ni\ gnx\ \{>\ l2\ inv3\}$

$\langle proof \rangle$

**lemma**  $l2\ inv3\ step3:$

$\{l2\ inv3\}\ l2\ step3\ Ra\ A\ B\ Nr\ gny\ \{>\ l2\ inv3\}$

$\langle proof \rangle$

**lemma**  $l2\ inv3\ step4:$

$\{l2\ inv3\}\ l2\ step4\ Rb\ A\ B\ Ni\ gnx\ \{>\ l2\ inv3\}$

$\langle proof \rangle$

**lemma**  $l2\ inv3\ dy\ fake\ msg:$

$\{l2\ inv3\}\ l2\ dy\ fake\ msg\ M\ \{>\ l2\ inv3\}$

$\langle proof \rangle$

**lemma**  $l2\ inv3\ dy\ fake\ chan:$

$\{l2\ inv3\}\ l2\ dy\ fake\ chan\ M\ \{>\ l2\ inv3\}$

$\langle proof \rangle$

**lemma**  $l2\ inv3\ lkr\ others:$

$\{l2\ inv3\}\ l2\ lkr\ others\ A\ \{>\ l2\ inv3\}$

$\langle proof \rangle$

**lemma**  $l2\ inv3\ lkr\ after:$

$\{l2\ inv3\}\ l2\ lkr\ after\ A\ \{>\ l2\ inv3\}$

$\langle proof \rangle$

**lemma**  $l2\ inv3\ skr:$

$\{l2\ inv3 \cap l2\ inv2\}\ l2\ skr\ R\ K\ \{>\ l2\ inv3\}$

$\langle proof \rangle$

**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]:  
 $\{l2\text{-inv3} \cap l2\text{-inv2}\} \text{ trans } l2 \{> l2\text{-inv3}\}$   
 ⟨proof⟩

**lemma** *PO-l2-inv3* [iff]: *reach*  $l2 \subseteq l2\text{-inv3}$   
 ⟨proof⟩

Auxiliary dest rule for inv3.

**lemmas** *l2-inv3D-aux* =  
*l2-inv3D* [THEN *conjunct1*,  
 THEN [2] *subset-trans*,  
 THEN *synth-analz-mono, simplified*,  
 THEN [2] *rev-subsetD, rotated 1, OF IK-subset-extr*]

**lemma** *l2-inv3D-HashNonce1*:  
 $s \in l2\text{-inv3} \implies$   
 $\text{Hash} (\text{NonceF} (R\$N), X) \in \text{synth} (\text{analz} (\text{extr} (\text{bad } s) (\text{ik } s) (\text{chan } s))) \implies$   
 $R \in \text{bad-runs } s$   
 ⟨proof⟩

**lemma** *l2-inv3D-HashNonce2*:  
 $s \in l2\text{-inv3} \implies$   
 $\text{Hash} (X, \text{NonceF} (R\$N)) \in \text{synth} (\text{analz} (\text{extr} (\text{bad } s) (\text{ik } s) (\text{chan } s))) \implies$   
 $R \in \text{bad-runs } s$   
 ⟨proof⟩

#### 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*:  
 $\text{hmac } X K \in \text{synth} (\text{analz } H) \implies K \notin \text{synth} (\text{analz } H) \implies \text{hmac } X K \in \text{analz } H$   
 ⟨proof⟩

**lemma** *parts-exp* [simp]:  $\text{parts} \{Exp X Y\} = \{Exp X Y\}$   
 ⟨proof⟩

**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  
 $(s, s') \in l2\text{-step1 } Ra A B \cup l2\text{-step4 } Rb A B Ni gnx \cup l2\text{-skr } R KK \cup$   
 $l2\text{-dy-fake-msg } M \cup l2\text{-dy-fake-chan } MM \implies$   
 $\text{hmac } X K \in \text{parts} (\text{extr} (\text{bad } s) (\text{ik } s) (\text{chan } s))$   
 ⟨proof⟩

**lemma** *hmac-trans-2*:  
 $\text{hmac } X K \in \text{parts} (\text{extr} (\text{bad } s') (\text{ik } s') (\text{chan } s')) \implies$

$(s, s') \in l2\text{-step2 } Rb \ A \ B \ Ni \ gnx \implies$   
 $hmac \ X \ K \in parts \ (extr \ (bad \ s) \ (ik \ s) \ (chan \ s)) \vee$   
 $(X = \langle Number \ 0, \ gnx, \ Exp \ Gen \ (NonceF \ (Rb\$ny)), \ Agent \ B, \ Agent \ A \rangle \wedge$   
 $K = Hash \ \langle Ni, \ NonceF \ (Rb\$nr) \rangle \wedge$   
 $guessed\text{-runs } Rb = \langle role=Resp, \ owner=B, \ partner=A \rangle \wedge$   
 $progress \ s' \ Rb = Some \ \{xny, \ xni, \ xnr, \ xgnx, \ xgny, \ xsk\} \wedge$   
 $guessed\text{-frame } Rb \ xgnx = Some \ gnx \wedge$   
 $guessed\text{-frame } Rb \ xni = Some \ Ni \ )$   
 $\langle proof \rangle$

**lemma** *hmac-trans-3*:

$hmac \ X \ K \in parts \ (extr \ (bad \ s') \ (ik \ s') \ (chan \ s')) \implies$   
 $(s, s') \in l2\text{-step3 } Ra \ A \ B \ Nr \ gny \implies$   
 $hmac \ X \ K \in parts \ (extr \ (bad \ s) \ (ik \ s) \ (chan \ s)) \vee$   
 $(X = \langle Number \ 1, \ gny, \ Exp \ Gen \ (NonceF \ (Ra\$nx)), \ Agent \ A, \ Agent \ B \rangle \wedge$   
 $K = Hash \ \langle NonceF \ (Ra\$ni), \ Nr \rangle \wedge$   
 $guessed\text{-runs } Ra = \langle role=Init, \ owner=A, \ partner=B \rangle \wedge$   
 $progress \ s' \ Ra = Some \ \{xnx, \ xni, \ xnr, \ xgnx, \ xgny, \ xsk, \ xEnd\} \wedge$   
 $guessed\text{-frame } Ra \ xgny = Some \ gny \wedge$   
 $guessed\text{-frame } Ra \ xnr = Some \ Nr \ )$   
 $\langle proof \rangle$

**lemma** *hmac-trans-lkr-aux*:

$hmac \ X \ K \in parts \ \{M. \ \exists \ x \ A \ B. \ Chan \ x \ A \ B \ M \in \ chan \ s\} \implies$   
 $K \notin synth \ (analz \ (extr \ (bad \ s) \ (ik \ s) \ (chan \ s))) \implies$   
 $s \in l2\text{-inv3} \implies$   
 $hmac \ X \ K \in parts \ (extr \ (bad \ s) \ (ik \ s) \ (chan \ s))$   
 $\langle proof \rangle$

**lemma** *hmac-trans-lkr*:

$hmac \ X \ K \in parts \ (extr \ (bad \ s') \ (ik \ s') \ (chan \ s')) \implies$   
 $K \notin synth \ (analz \ (extr \ (bad \ s) \ (ik \ s) \ (chan \ s))) \implies$   
 $s \in l2\text{-inv3} \implies$   
 $(s, s') \in l2\text{-lkr-others } A \cup l2\text{-lkr-after } A \implies$   
 $hmac \ X \ K \in parts \ (extr \ (bad \ s) \ (ik \ s) \ (chan \ s))$   
 $\langle proof \rangle$

**lemmas** *hmac-trans = hmac-trans-1-4-skr-extr-fake hmac-trans-lkr hmac-trans-2 hmac-trans-3*

### 27.2.5 inv4 (authentication guard)

If HMAC is *parts (extr (bad s) (ik s) (chan s))* and *A, B* are honest then the message has indeed been sent by a responder run (etc).

**definition**

*l2-inv4* :: *l2-state set*

**where**

$l2\text{-inv4} \equiv \{s. \ \forall \ Ra \ A \ B \ gny \ Nr.$   
 $hmac \ \langle Number \ 0, \ Exp \ Gen \ (NonceF \ (Ra\$nx)), \ gny, \ Agent \ B, \ Agent \ A \rangle$   
 $(Hash \ \langle NonceF \ (Ra\$ni), \ Nr \rangle) \in parts \ (extr \ (bad \ s) \ (ik \ s) \ (chan \ s)) \implies$   
 $guessed\text{-runs } Ra = \langle role=Init, \ owner=A, \ partner=B \rangle \implies$   
 $A \notin bad \ s \wedge B \notin bad \ s \implies$

$$\begin{aligned}
& (\exists Rb. \text{ guessed-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \wedge \\
& \quad \text{in-progressS } (\text{progress } s \text{ } Rb) \{xny, xni, xnr, xgnx, xgny, xsk\} \wedge \\
& \quad \text{guessed-frame } Rb \text{ } xgny = \text{Some } gny \wedge \\
& \quad \text{guessed-frame } Rb \text{ } xnr = \text{Some } Nr \wedge \\
& \quad \text{guessed-frame } Rb \text{ } xni = \text{Some } (\text{NonceF } (Ra\$ni)) \wedge \\
& \quad \text{guessed-frame } Rb \text{ } xgnx = \text{Some } (\text{Exp Gen } (\text{NonceF } (Ra\$nx)))) \\
& \}
\end{aligned}$$

**lemmas**  $l2\text{-inv}4I = l2\text{-inv}4\text{-def}$  [THEN setc-def-to-intro, rule-format]

**lemmas**  $l2\text{-inv}4E$  [elim] =  $l2\text{-inv}4\text{-def}$  [THEN setc-def-to-elim, rule-format]

**lemmas**  $l2\text{-inv}4D = l2\text{-inv}4\text{-def}$  [THEN setc-def-to-dest, rule-format, rotated 1, simplified]

**lemma**  $l2\text{-inv}4\text{-init}$  [iff]:

$\text{init } l2 \subseteq l2\text{-inv}4$

$\langle \text{proof} \rangle$

**lemma**  $l2\text{-inv}4\text{-trans}$  [iff]:

$\{l2\text{-inv}4 \cap l2\text{-inv}2 \cap l2\text{-inv}3\} \text{ trans } l2 \{> l2\text{-inv}4\}$

$\langle \text{proof} \rangle$

**lemma**  $PO\text{-}l2\text{-inv}4$  [iff]:  $\text{reach } l2 \subseteq l2\text{-inv}4$

$\langle \text{proof} \rangle$

**lemma**  $\text{auth-guard-step}3$ :

$s \in l2\text{-inv}4 \implies$

$s \in l2\text{-inv}1 \implies$

$\text{Insec } B \ A \ \langle gny, \text{hmac } \langle \text{Number } 0, \text{Exp Gen } (\text{NonceF } (Ra\$nx)), gny, \text{Agent } B, \text{Agent } A \rangle \rangle$   
 $\quad (\text{Hash } \langle \text{NonceF } (Ra\$ni), Nr \rangle)$

$\in \text{chan } s \implies$

$\text{guessed-runs } Ra = (\text{role}=\text{Init}, \text{owner}=A, \text{partner}=B) \implies$

$\text{can-signal } s \ A \ B \implies$

$(\exists Rb. \text{ guessed-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \wedge$   
 $\quad \text{in-progressS } (\text{progress } s \text{ } Rb) \{xny, xni, xnr, xgnx, xgny, xsk\} \wedge$   
 $\quad \text{guessed-frame } Rb \text{ } xgny = \text{Some } gny \wedge$   
 $\quad \text{guessed-frame } Rb \text{ } xnr = \text{Some } Nr \wedge$   
 $\quad \text{guessed-frame } Rb \text{ } xni = \text{Some } (\text{NonceF } (Ra\$ni)) \wedge$   
 $\quad \text{guessed-frame } Rb \text{ } xgnx = \text{Some } (\text{Exp Gen } (\text{NonceF } (Ra\$nx))))$

$\langle \text{proof} \rangle$

### 27.2.6 inv5 (authentication guard)

If MAC is in  $\text{parts } (\text{extr } (\text{bad } s) (\text{ik } s) (\text{chan } s))$  and  $A, B$  are honest then the message has indeed been sent by an initiator run (etc).

**definition**

$l2\text{-inv}5 :: l2\text{-state set}$

**where**

$l2\text{-inv}5 \equiv \{s. \forall Rb \ A \ B \ gnx \ Ni.$

$\quad \text{hmac } \langle \text{Number } 1, \text{Exp Gen } (\text{NonceF } (Rb\$ny)), gnx, \text{Agent } A, \text{Agent } B \rangle$

$\quad (\text{Hash } \langle Ni, \text{NonceF } (Rb\$nr) \rangle) \in \text{parts } (\text{extr } (\text{bad } s) (\text{ik } s) (\text{chan } s)) \implies$

$\quad \text{guessed-runs } Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \implies$

$\quad A \notin \text{bad } s \wedge B \notin \text{bad } s \implies$

$$\begin{aligned}
& (\exists Ra. \text{ guessed-runs } Ra = (\text{role=Init, owner=A, partner=B}) \wedge \\
& \quad \text{in-progressS } (\text{progress } s \text{ Ra}) \{xnx, xni, xnr, xgnx, xgny, xsk, xEnd\} \wedge \\
& \quad \text{guessed-frame } Ra \text{ xgnx} = \text{Some } gnx \wedge \\
& \quad \text{guessed-frame } Ra \text{ xni} = \text{Some } Ni \wedge \\
& \quad \text{guessed-frame } Ra \text{ xnr} = \text{Some } (\text{NonceF } (Rb\$nr)) \wedge \\
& \quad \text{guessed-frame } Ra \text{ xgny} = \text{Some } (\text{Exp Gen } (\text{NonceF } (Rb\$ny)))) \\
& \}
\end{aligned}$$

**lemmas**  $l2\text{-inv5I} = l2\text{-inv5-def } [THEN \text{ setc-def-to-intro, rule-format}]$

**lemmas**  $l2\text{-inv5E } [elim] = l2\text{-inv5-def } [THEN \text{ setc-def-to-elim, rule-format}]$

**lemmas**  $l2\text{-inv5D} = l2\text{-inv5-def } [THEN \text{ setc-def-to-dest, rule-format, rotated 1, simplified}]$

**lemma**  $l2\text{-inv5-init } [iff]:$

$\text{init } l2 \subseteq l2\text{-inv5}$

$\langle \text{proof} \rangle$

**lemma**  $l2\text{-inv5-trans } [iff]:$

$\{l2\text{-inv5} \cap l2\text{-inv2} \cap l2\text{-inv3}\} \text{ trans } l2 \{> l2\text{-inv5}\}$

$\langle \text{proof} \rangle$

**lemma**  $PO\text{-}l2\text{-inv5 } [iff]: \text{ reach } l2 \subseteq l2\text{-inv5}$

$\langle \text{proof} \rangle$

**lemma**  $\text{auth-guard-step4}:$

$s \in l2\text{-inv5} \implies$

$s \in l2\text{-inv1} \implies$

$\text{Insec } A \ B \ (\text{hmac } \langle \text{Number } 1, \text{Exp Gen } (\text{NonceF } (Rb\$ny)), gnx, \text{Agent } A, \text{Agent } B \rangle$   
 $\quad (\text{Hash } \langle Ni, \text{NonceF } (Rb\$nr) \rangle))$

$\in \text{chan } s \implies$

$\text{guessed-runs } Rb = (\text{role=Resp, owner=B, partner=A}) \implies$

$\text{can-signal } s \ A \ B \implies$

$$\begin{aligned}
& (\exists Ra. \text{ guessed-runs } Ra = (\text{role=Init, owner=A, partner=B}) \wedge \\
& \quad \text{in-progressS } (\text{progress } s \text{ Ra}) \{xnx, xni, xnr, xgnx, xgny, xsk, xEnd\} \wedge \\
& \quad \text{guessed-frame } Ra \text{ xgnx} = \text{Some } gnx \wedge \\
& \quad \text{guessed-frame } Ra \text{ xni} = \text{Some } Ni \wedge \\
& \quad \text{guessed-frame } Ra \text{ xnr} = \text{Some } (\text{NonceF } (Rb\$nr)) \wedge \\
& \quad \text{guessed-frame } Ra \text{ xgny} = \text{Some } (\text{Exp Gen } (\text{NonceF } (Rb\$ny)))) \\
& \}
\end{aligned}$$

$\langle \text{proof} \rangle$

## 27.2.7 inv6

For an initiator, the session key is always  $gny^{nx}$ .

**definition**

$l2\text{-inv6} :: l2\text{-state set}$

**where**

$l2\text{-inv6} \equiv \{s. \forall Ra \ A \ B \ gny.$

$\text{guessed-runs } Ra = (\text{role=Init, owner=A, partner=B}) \longrightarrow$

$\text{in-progress } (\text{progress } s \text{ Ra}) \ xsk \longrightarrow$

$\text{guessed-frame } Ra \text{ xgny} = \text{Some } gny \longrightarrow$

$\text{guessed-frame } Ra \ xsk = \text{Some } (\text{Exp } gny \ (\text{NonceF } (Ra\$nx)))$

$\}$

**lemmas**  $l2\text{-inv6}I = l2\text{-inv6}\text{-def}$  [*THEN setc-def-to-intro, rule-format*]  
**lemmas**  $l2\text{-inv6}E$  [*elim*] =  $l2\text{-inv6}\text{-def}$  [*THEN setc-def-to-elim, rule-format*]  
**lemmas**  $l2\text{-inv6}D = l2\text{-inv6}\text{-def}$  [*THEN setc-def-to-dest, rule-format, rotated 1, simplified*]

**lemma**  $l2\text{-inv6}\text{-init}$  [*iff*]:  
 $init\ l2 \subseteq l2\text{-inv6}$   
 $\langle proof \rangle$

**lemma**  $l2\text{-inv6}\text{-trans}$  [*iff*]:  
 $\{l2\text{-inv6}\} \text{ trans } l2 \{> l2\text{-inv6}\}$   
 $\langle proof \rangle$

**lemma**  $PO\text{-}l2\text{-inv6}$  [*iff*]:  $reach\ l2 \subseteq l2\text{-inv6}$   
 $\langle proof \rangle$

### 27.2.8 inv6'

For a responder, the session key is always  $gnx^{ny}$ .

#### definition

$l2\text{-inv6}' :: l2\text{-state set}$

#### where

$l2\text{-inv6}' \equiv \{s. \forall Rb\ A\ B\ gnx.$   
 $guessed\text{-runs}\ Rb = (\text{role}=\text{Resp}, \text{owner}=B, \text{partner}=A) \longrightarrow$   
 $in\text{-progress}\ (\text{progress}\ s\ Rb)\ xsk \longrightarrow$   
 $guessed\text{-frame}\ Rb\ xgnx = \text{Some}\ gnx \longrightarrow$   
 $guessed\text{-frame}\ Rb\ xsk = \text{Some}\ (\text{Exp}\ gnx\ (\text{NonceF}\ (Rb\$ny)))$   
 $\}$

**lemmas**  $l2\text{-inv6}'I = l2\text{-inv6}'\text{-def}$  [*THEN setc-def-to-intro, rule-format*]  
**lemmas**  $l2\text{-inv6}'E$  [*elim*] =  $l2\text{-inv6}'\text{-def}$  [*THEN setc-def-to-elim, rule-format*]  
**lemmas**  $l2\text{-inv6}'D = l2\text{-inv6}'\text{-def}$  [*THEN setc-def-to-dest, rule-format, rotated 1, simplified*]

**lemma**  $l2\text{-inv6}'\text{-init}$  [*iff*]:  
 $init\ l2 \subseteq l2\text{-inv6}'$   
 $\langle proof \rangle$

**lemma**  $l2\text{-inv6}'\text{-trans}$  [*iff*]:  
 $\{l2\text{-inv6}'\} \text{ trans } l2 \{> l2\text{-inv6}'\}$   
 $\langle proof \rangle$

**lemma**  $PO\text{-}l2\text{-inv6}'$  [*iff*]:  $reach\ l2 \subseteq l2\text{-inv6}'$   
 $\langle proof \rangle$

### 27.2.9 inv7: form of the secrets

#### definition

$l2\text{-inv7} :: l2\text{-state set}$

#### where

$l2\text{-inv7} \equiv \{s.$   
 $secret\ s \subseteq \{\text{Exp}\ (\text{Exp}\ \text{Gen}\ (\text{NonceF}\ (R\$N)))\ (\text{NonceF}\ (R'\$N'))\ |\ N\ N'\ R\ R'.$   
 $R = \text{test} \wedge R' \in \text{partners} \wedge (N=nx \vee N=ny) \wedge (N'=nx \vee N'=ny)\}$

}

**lemmas**  $l2\text{-inv}7I = l2\text{-inv}7\text{-def}$  [THEN setc-def-to-intro, rule-format]

**lemmas**  $l2\text{-inv}7E$  [elim] =  $l2\text{-inv}7\text{-def}$  [THEN setc-def-to-elim, rule-format]

**lemmas**  $l2\text{-inv}7D = l2\text{-inv}7\text{-def}$  [THEN setc-def-to-dest, rule-format, rotated 1, simplified]

**lemma**  $l2\text{-inv}7\text{-init}$  [iff]:

$init\ l2 \subseteq l2\text{-inv}7$

<proof>

Steps 3 and 4 are the hard part.

**lemma**  $l2\text{-inv}7\text{-step}3$ :

$\{l2\text{-inv}7 \cap l2\text{-inv}1 \cap l2\text{-inv}4 \cap l2\text{-inv}6'\} l2\text{-step}3\ Ra\ A\ B\ Nr\ gny\ \{>\ l2\text{-inv}7\}$

<proof>

**lemma**  $l2\text{-inv}7\text{-step}4$ :

$\{l2\text{-inv}7 \cap l2\text{-inv}1 \cap l2\text{-inv}5 \cap l2\text{-inv}6 \cap l2\text{-inv}6'\} l2\text{-step}4\ Rb\ A\ B\ Ni\ gnx\ \{>\ l2\text{-inv}7\}$

<proof>

**lemma**  $l2\text{-inv}7\text{-trans}$  [iff]:

$\{l2\text{-inv}7 \cap l2\text{-inv}1 \cap l2\text{-inv}4 \cap l2\text{-inv}5 \cap l2\text{-inv}6 \cap l2\text{-inv}6'\} trans\ l2\ \{>\ l2\text{-inv}7\}$

<proof>

**lemma**  $PO\text{-}l2\text{-inv}7$  [iff]:  $reach\ l2 \subseteq l2\text{-inv}7$

<proof>

auxiliary dest rule for inv7

**lemma**  $Exp\text{-}Exp\text{-}Gen\text{-}synth$ :

$Exp\ (Exp\ Gen\ X)\ Y \in synth\ H \implies Exp\ (Exp\ Gen\ X)\ Y \in H \vee X \in synth\ H \vee Y \in synth\ H$

<proof>

**lemma**  $l2\text{-inv}7\text{-aux}$ :

$s \in l2\text{-inv}7 \implies$

$x \in secret\ s \implies$

$x \notin synth\ (analz\ (generators\ s))$

<proof>

## 27.3 Refinement

Mediator function.

**definition**

$med12s :: l2\text{-obs} \Rightarrow skl1\text{-obs}$

**where**

$med12s\ t \equiv ()$

$ik = ik\ t,$

$secret = secret\ t,$

$progress = progress\ t,$

$signalsInit = signalsInit\ t,$

$signalsResp = signalsResp\ t,$

$signalsInit2 = signalsInit2\ t,$

$signalsResp2 = signalsResp2\ t$



)

Relation between states.

**definition**

$R12s :: (skl1\text{-state} * l2\text{-state}) \text{ set}$

**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*]:

$(s1, s2) \in R12s \implies$   
 $can\text{-signal}\ s1\ A\ B \longleftrightarrow can\text{-signal}\ s2\ A\ B$

$\langle proof \rangle$

Protocol events.

**lemma**  $l2\text{-step1}\text{-refines}\text{-step1}$ :

$\{R12s\}\ skl1\text{-step1}\ Ra\ A\ B, l2\text{-step1}\ Ra\ A\ B \{>R12s\}$

$\langle proof \rangle$

**lemma**  $l2\text{-step2}\text{-refines}\text{-step2}$ :

$\{R12s\}\ skl1\text{-step2}\ Rb\ A\ B\ Ni\ gnx, l2\text{-step2}\ Rb\ A\ B\ Ni\ gnx \{>R12s\}$

$\langle proof \rangle$

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**  $l2\text{-step3}\text{-refines}\text{-step3}$ :

$\{R12s \cap UNIV \times (l2\text{-inv1} \cap l2\text{-inv3} \cap l2\text{-inv4} \cap l2\text{-inv6}' \cap l2\text{-inv7})\}$   
 $skl1\text{-step3}\ Ra\ A\ B\ Nr\ gny, l2\text{-step3}\ Ra\ A\ B\ Nr\ gny$   
 $\{>R12s\}$

$\langle proof \rangle$

**lemma**  $l2\text{-step4}\text{-refines}\text{-step4}$ :

$\{R12s \cap UNIV \times (l2\text{-inv1} \cap l2\text{-inv3} \cap l2\text{-inv5} \cap l2\text{-inv6} \cap l2\text{-inv6}' \cap l2\text{-inv7})\}$   
 $skl1\text{-step4}\ Rb\ A\ B\ Ni\ gnx, l2\text{-step4}\ Rb\ A\ B\ Ni\ gnx$   
 $\{>R12s\}$

$\langle proof \rangle$

attacker events

**lemma**  $l2\text{-dy}\text{-fake}\text{-chan}\text{-refines}\text{-skip}$ :

$\{R12s\}\ Id, l2\text{-dy}\text{-fake}\text{-chan}\ M \{>R12s\}$

$\langle proof \rangle$

**lemma**  $l2\text{-dy}\text{-fake}\text{-msg}\text{-refines}\text{-learn}$ :

$\{R12s \cap UNIV \times (l2\text{-inv3} \cap l2\text{-inv7})\}\ l1\text{-learn}\ m, l2\text{-dy}\text{-fake}\text{-msg}\ m \{>R12s\}$

$\langle proof \rangle$

compromising events

**lemma** *l2-lkr-others-refines-skip*:  
 $\{R12s\} Id, l2-lkr-others A \{>R12s\}$   
 $\langle proof \rangle$

**lemma** *l2-lkr-after-refines-skip*:  
 $\{R12s\} Id, l2-lkr-after A \{>R12s\}$   
 $\langle proof \rangle$

**lemma** *l2-skr-refines-learn*:  
 $\{R12s \cap UNIV \times (l2-inv2 \cap l2-inv3 \cap l2-inv7)\} l1-learn K, l2-skr R K \{>R12s\}$   
 $\langle proof \rangle$

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]*:  
 $init\ l2 \subseteq R12s \text{ “ } (init\ skl1)$   
 $\langle proof \rangle$

**lemma** *l2-refines-trans-l1 [iff]*:  
 $\{R12s \cap (UNIV \times (l2-inv1 \cap l2-inv2 \cap l2-inv3 \cap l2-inv4 \cap l2-inv5 \cap$   
 $l2-inv6 \cap l2-inv6' \cap l2-inv7))\}$   
 $trans\ skl1, trans\ l2$   
 $\{> R12s\}$   
 $\langle proof \rangle$

**lemma** *PO-obs-consistent-R12s [iff]*:  
 $obs-consistent\ R12s\ med12s\ skl1\ l2$   
 $\langle proof \rangle$

**lemma** *l2-refines-l1 [iff]*:  
 $refines$   
 $(R12s \cap$   
 $(reach\ skl1 \times (l2-inv1 \cap l2-inv2 \cap l2-inv3 \cap l2-inv4 \cap l2-inv5 \cap$   
 $l2-inv6 \cap l2-inv6' \cap l2-inv7)))$   
 $med12s\ skl1\ l2$   
 $\langle proof \rangle$

**lemma** *l2-implements-l1 [iff]*:  
 $implements\ med12s\ skl1\ l2$   
 $\langle proof \rangle$

## 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**

$l2\text{-partial}\text{-secrecy} :: ('a\ l2\text{-state}\text{-scheme})\ set$

**where**

$l2\text{-partial}\text{-secrecy} \equiv \{s.\ synth\ (analz\ (ik\ s)) \cap\ secret\ s = \{\}\}$

**lemma**  $l2\text{-obs}\text{-partial}\text{-secrecy}$  [iff]:  $oreach\ l2 \subseteq l2\text{-partial}\text{-secrecy}$   
 <proof>

**lemma**  $l2\text{-oreach}\text{-dy}\text{-fake}\text{-msg}$ :

$\llbracket s \in oreach\ l2; x \in dy\text{-fake}\text{-msg}\ (bad\ s)\ (ik\ s)\ (chan\ s) \rrbracket$

$\implies s\ (ik := insert\ x\ (ik\ s)) \in oreach\ l2$

<proof>

**definition**

$l2\text{-secrecy} :: ('a\ l2\text{-state}\text{-scheme})\ set$

**where**

$l2\text{-secrecy} \equiv \{s.\ dy\text{-fake}\text{-msg}\ (bad\ s)\ (ik\ s)\ (chan\ s) \cap\ secret\ s = \{\}\}$

**lemma**  $l2\text{-obs}\text{-secrecy}$  [iff]:  $oreach\ l2 \subseteq l2\text{-secrecy}$   
 <proof>

**lemma**  $l2\text{-secrecy}$  [iff]:  $reach\ l2 \subseteq l2\text{-secrecy}$   
 <proof>

**abbreviation**  $l2\text{-iagreement}\text{-Init} \equiv l1\text{-iagreement}\text{-Init}$

**lemma**  $l2\text{-obs}\text{-iagreement}\text{-Init}$  [iff]:  $oreach\ l2 \subseteq l2\text{-iagreement}\text{-Init}$   
 <proof>

**lemma**  $l2\text{-iagreement}\text{-Init}$  [iff]:  $reach\ l2 \subseteq l2\text{-iagreement}\text{-Init}$   
 <proof>

**abbreviation**  $l2\text{-iagreement}\text{-Resp} \equiv l1\text{-iagreement}\text{-Resp}$

**lemma**  $l2\text{-obs}\text{-iagreement}\text{-Resp}$  [iff]:  $oreach\ l2 \subseteq l2\text{-iagreement}\text{-Resp}$   
 <proof>

**lemma**  $l2\text{-iagreement}\text{-Resp}$  [iff]:  $reach\ l2 \subseteq l2\text{-iagreement}\text{-Resp}$   
 <proof>

**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 = Some (Exp gny (NonceF (Ra\$nx)))*  $\wedge$   
*implConfid B A Nr*  $\in ik\ s \wedge$   
*implInsec B A*  $\langle gny, hmac \langle Number\ 0, Exp\ Gen\ (NonceF\ (Ra\$nx)), gny, Agent\ B, Agent\ A \rangle$   
 $(Hash \langle NonceF\ (Ra\$ni), Nr \rangle) \in ik\ s \wedge$   
— actions:  
*s' = s* (*progress := (progress s)(Ra*  $\mapsto \{xnx, xni, xnr, xgnx, xgny, xsk, xEnd\}$ ),  
*ik := {implInsec A B (hmac*  $\langle Number\ 1, gny, Exp\ Gen\ (NonceF\ (Ra\$nx)), Agent\ A, Agent$   
*B)*  
 $(Hash \langle NonceF\ (Ra\$ni), Nr \rangle) \}$   $\cup ik\ s,$   
*secret := {x. x = Exp gny (NonceF (Ra\$nx))*  $\wedge Ra = test\}$   $\cup secret\ s,$   
*signalsInit :=*  
*if can-signal s A B then*  
*addSignal (signalsInit s)*  
 $(Commit\ A\ B \langle NonceF\ (Ra\$ni), Nr, Exp\ gny\ (NonceF\ (Ra\$nx)) \rangle)$   
*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)*  
 $(Running\ A\ B \langle NonceF\ (Ra\$ni), Nr, Exp\ gny\ (NonceF\ (Ra\$nx)) \rangle)$   
*else*  
*signalsResp s,*  
*signalsResp2 :=*  
*if can-signal s A B then*  
*addSignal (signalsResp2 s)*  $(Running\ A\ B\ (Exp\ gny\ (NonceF\ (Ra\$nx))))$   
*else*  
*signalsResp2 s*  
*})*  
*}*

**definition**

*l3-step4* :: *rid-t*  $\Rightarrow agent \Rightarrow agent \Rightarrow msg \Rightarrow msg \Rightarrow l3-trans$

**where**

*l3-step4 Rb A B Ni gnz*  $\equiv \{(s, s')\}.$

— guards:

*guessed-runs Rb* =  $(role=Resp, owner=B, partner=A) \wedge$

*progress s Rb* = *Some*  $\{xny, xni, xnr, xgnx, xgny, xsk\} \wedge$

*guessed-frame Rb xgnz* = *Some gnz*  $\wedge$

*guessed-frame Rb xni* = *Some Ni*  $\wedge$

*implInsec A B (hmac*  $\langle Number\ 1, Exp\ Gen\ (NonceF\ (Rb\$ny)), gnz, Agent\ A, Agent\ B \rangle$

$(Hash \langle Ni, NonceF\ (Rb\$nr) \rangle) \in ik\ s \wedge$

— actions:

*s' = s* (*progress := (progress s)(Rb*  $\mapsto \{xny, xni, xnr, xgnz, xgny, xsk, xEnd\}$ ),

*secret := {x. x = Exp gnz (NonceF (Rb\$ny))*  $\wedge Rb = test\}$   $\cup secret\ s,$

*signalsResp :=*

*if can-signal s A B then*

```

      addSignal (signalsResp s)
        (Commit A B ⟨Ni, NonceF (Rb$nr), Exp gnx (NonceF (Rb$ny))⟩)
    else
      signalsResp s,
signalsResp2 :=
  if can-signal s A B then
    addSignal (signalsResp2 s) (Commit A B (Exp gnx (NonceF (Rb$ny))))
  else
    signalsResp2 s
  }
}

```

Specification.

Initial compromise.

**definition**

*ik-init* :: msg set

**where**

$ik-init \equiv \{priK\ C \mid C. C \in bad-init\} \cup \{pubK\ A \mid A. True\} \cup$   
 $\{shrK\ A\ B \mid A\ B. A \in bad-init \vee B \in bad-init\} \cup Tags$

lemmas about *ik-init*

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

**lemma** *analz-ik-init* [simp]: *analz ik-init = ik-init*  
 ⟨proof⟩

**lemma** *abs-ik-init* [iff]: *abs ik-init = {}*  
 ⟨proof⟩

**lemma** *payloadSet-ik-init* [iff]: *ik-init ∩ payload = {}*  
 ⟨proof⟩

**lemma** *validSet-ik-init* [iff]: *ik-init ∩ valid = {}*  
 ⟨proof⟩

**definition**

*l3-init* :: l3-state set

**where**

$l3-init \equiv \{ \mid$   
 $ik = ik-init,$   
 $secret = \{\},$   
 $progress = Map.empty,$   
 $signalsInit = \lambda x. 0,$   
 $signalsResp = \lambda x. 0,$   
 $signalsInit2 = \lambda x. 0,$   
 $signalsResp2 = \lambda x. 0,$   
 $bad = bad-init$   
 $\}$

**lemmas** *l3-init-defs = l3-init-def ik-init-def*



**definition** $l3\text{-trans} :: l3\text{-trans}$ **where**

$$\begin{aligned}
l3\text{-trans} &\equiv (\bigcup M N X Rb Ra A B K. \\
&\quad l3\text{-step1 } Ra A B \cup \\
&\quad l3\text{-step2 } Rb A B N X \cup \\
&\quad l3\text{-step3 } Ra A B N X \cup \\
&\quad l3\text{-step4 } Rb A B N X \cup \\
&\quad l3\text{-dy } M \cup \\
&\quad l3\text{-lkr-others } A \cup \\
&\quad l3\text{-lkr-after } A \cup \\
&\quad l3\text{-skr } Ra K \cup \\
&\quad Id \\
&.)
\end{aligned}$$
**definition** $l3 :: (l3\text{-state}, l3\text{-obs}) \text{ spec}$  **where**

$$\begin{aligned}
l3 &\equiv \langle \\
&\quad \text{init} = l3\text{-init}, \\
&\quad \text{trans} = l3\text{-trans}, \\
&\quad \text{obs} = id \\
&\rangle
\end{aligned}$$
**lemmas**  $l3\text{-loc-defs} =$ 

$$\begin{aligned}
&l3\text{-step1-def } l3\text{-step2-def } l3\text{-step3-def } l3\text{-step4-def} \\
&l3\text{-def } l3\text{-init-defs } l3\text{-trans-def} \\
&l3\text{-dy-def} \\
&l3\text{-lkr-others-def } l3\text{-lkr-after-def } l3\text{-skr-def}
\end{aligned}$$
**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*]:  $obs \ l3 = id$  $\langle \text{proof} \rangle$ 

## 28.2 Invariants

### 28.2.1 inv1: No long-term keys as message parts

**definition** $l3\text{-inv1} :: l3\text{-state set}$ **where**

$$\begin{aligned}
l3\text{-inv1} &\equiv \{s. \\
&\quad \text{parts } (ik \ s) \cap \text{range } LtK \subseteq ik \ s \\
&\}
\end{aligned}$$
**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**  $l3\text{-inv1D} = l3\text{-inv1-def}$  [*THEN setc-def-to-dest, rule-format*]

**lemma** *l3-inv1D'* [*dest*]:  $\llbracket LtK\ K \in parts\ (ik\ s); s \in l3-inv1 \rrbracket \implies LtK\ K \in ik\ s$   
 <proof>

**lemma** *l3-inv1-init* [*iff*]:  
 $init\ l3 \subseteq l3-inv1$   
 <proof>

**lemma** *l3-inv1-trans* [*iff*]:  
 $\{l3-inv1\}\ trans\ l3\ \{>\ l3-inv1\}$   
 <proof>

**lemma** *PO-l3-inv1* [*iff*]:  
 $reach\ l3 \subseteq l3-inv1$   
 <proof>

### 28.2.2 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!*]:  
 $init\ l3 \subseteq l3-inv2$   
 <proof>

**lemma** *l3-inv2-trans* [*simp,intro!*]:  
 $\{l3-inv2 \cap l3-inv1\}\ trans\ l3\ \{>\ l3-inv2\}$   
 <proof>

**lemma** *PO-l3-inv2* [*iff*]:  $reach\ l3 \subseteq l3-inv2$   
 <proof>

### 28.2.3 inv3

If a message can be analyzed from the intruder knowledge then it can be derived (using *synth/analz*) from the sets of implementation, non-implementation, and long-term key messages and the tags. That is, intermediate messages are not needed.

#### definition

*l3-inv3* :: *l3-state set*

#### where

$l3-inv3 \equiv \{s.$   
 $analz\ (ik\ s) \subseteq$   
 $synth\ (analz\ ((ik\ s \cap payload) \cup ((ik\ s) \cap valid) \cup (ik\ s \cap range\ LtK) \cup Tags))$   
 $\}$

**lemmas**  $l3\text{-inv}3I = l3\text{-inv}3\text{-def}$  [*THEN setc-def-to-intro, rule-format*]  
**lemmas**  $l3\text{-inv}3E = l3\text{-inv}3\text{-def}$  [*THEN setc-def-to-elim, rule-format*]  
**lemmas**  $l3\text{-inv}3D = l3\text{-inv}3\text{-def}$  [*THEN setc-def-to-dest, rule-format*]

**lemma**  $l3\text{-inv}3\text{-init}$  [*iff*]:  
 $init\ l3 \subseteq l3\text{-inv}3$   
 $\langle proof \rangle$

**declare**  $domIff$  [*iff del*]

Most of the cases in this proof are simple and very similar. The proof could probably be shortened.

**lemma**  $l3\text{-inv}3\text{-trans}$  [*simp,intro!*]:  
 $\{l3\text{-inv}3\}$  *trans*  $l3 \{> l3\text{-inv}3\}$   
 $\langle proof \rangle$

**lemma**  $PO\text{-}l3\text{-inv}3$  [*iff*]:  $reach\ l3 \subseteq l3\text{-inv}3$   
 $\langle proof \rangle$

#### 28.2.4 inv4: the intruder knows the tags

**definition**

$l3\text{-inv}4 :: l3\text{-state set}$

**where**

$l3\text{-inv}4 \equiv \{s.$   
 $Tags \subseteq ik\ s$   
 $\}$

**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!*]:  
 $init\ l3 \subseteq l3\text{-inv}4$   
 $\langle proof \rangle$

**lemma**  $l3\text{-inv}4\text{-trans}$  [*simp,intro!*]:  
 $\{l3\text{-inv}4\}$  *trans*  $l3 \{> l3\text{-inv}4\}$   
 $\langle proof \rangle$

**lemma**  $PO\text{-}l3\text{-inv}4$  [*simp,intro!*]:  $reach\ l3 \subseteq l3\text{-inv}4$   
 $\langle proof \rangle$

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  $\text{extr } (l3\text{-state.}bad\ s) (ik\ s \cap \text{payload}) (local.abs\ (ik\ s))$ , that is, given the non-implementation messages and the abstractions of (implementation) messages in the intruder knowledge.

**definition**

$l3\text{-inv5} :: l3\text{-state set}$

**where**

$l3\text{-inv5} \equiv \{s.$   
 $\quad \text{synth } (analz\ (ik\ s)) \subseteq$   
 $\quad \text{dy-fake-msg } (bad\ s) (ik\ s \cap \text{payload}) (abs\ (ik\ s)) \cup \text{-payload}$   
 $\quad \}$

**lemmas**  $l3\text{-inv5I} = l3\text{-inv5-def } [THEN\ setc\text{-def-to-intro, rule-format}]$

**lemmas**  $l3\text{-inv5E} = l3\text{-inv5-def } [THEN\ setc\text{-def-to-elim, rule-format}]$

**lemmas**  $l3\text{-inv5D} = l3\text{-inv5-def } [THEN\ setc\text{-def-to-dest, rule-format}]$

**lemma**  $l3\text{-inv5-derived: } l3\text{-inv2} \cap l3\text{-inv3} \subseteq l3\text{-inv5}$

$\langle\text{proof}\rangle$

**lemma**  $PO\text{-}l3\text{-inv5 } [simp,intro!]: \text{reach } l3 \subseteq l3\text{-inv5}$

$\langle\text{proof}\rangle$

### 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**

$l3\text{-inv6} :: l3\text{-state set}$

**where**

$l3\text{-inv6} \equiv \{s. \forall A B M.$   
 $\quad (\text{implInsec } A B M \in \text{synth } (analz\ (ik\ s)) \wedge M \in \text{payload}) \longrightarrow$   
 $\quad (\text{implInsec } A B M \in ik\ s \vee M \in \text{synth } (analz\ (ik\ s)))$   
 $\quad \}$

**lemmas**  $l3\text{-inv6I} = l3\text{-inv6-def } [THEN\ setc\text{-def-to-intro, rule-format}]$

**lemmas**  $l3\text{-inv6E} = l3\text{-inv6-def } [THEN\ setc\text{-def-to-elim, rule-format}]$

**lemmas**  $l3\text{-inv6D} = l3\text{-inv6-def } [THEN\ setc\text{-def-to-dest, rule-format}]$

**lemma**  $l3\text{-inv6-derived } [simp,intro!]:$

$l3\text{-inv3} \cap l3\text{-inv4} \subseteq l3\text{-inv6}$

$\langle\text{proof}\rangle$

**lemma**  $PO\text{-}l3\text{-inv6 } [simp,intro!]: \text{reach } l3 \subseteq l3\text{-inv6}$

$\langle\text{proof}\rangle$

### 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**

$l3\text{-inv}7 \equiv \{s. \forall A B M. \\ (implConfid A B M \in synth (analz (ik s)) \wedge M \in payload) \longrightarrow \\ (implConfid A B M \in ik s \vee M \in synth (analz (ik s))) \\ \}$

**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!]:

$l3\text{-inv}3 \cap l3\text{-inv}4 \subseteq l3\text{-inv}7$

$\langle proof \rangle$

**lemma**  $PO\text{-}l3\text{-inv}7$  [simp,intro!]:  $reach\ l3 \subseteq l3\text{-inv}7$

$\langle proof \rangle$

### 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**

$l3\text{-inv}8 :: l3\text{-state set}$

**where**

$l3\text{-inv}8 \equiv \{s. \forall A B M. \\ (implAuth A B M \in synth (analz (ik s)) \wedge M \in payload) \longrightarrow \\ (implAuth A B M \in ik s \vee (M \in synth (analz (ik s)) \wedge (A \in bad\ s \vee B \in bad\ s))) \\ \}$

**lemmas**  $l3\text{-inv}8I = l3\text{-inv}8\text{-def}$  [THEN setc-def-to-intro, rule-format]

**lemmas**  $l3\text{-inv}8E = l3\text{-inv}8\text{-def}$  [THEN setc-def-to-elim, rule-format]

**lemmas**  $l3\text{-inv}8D = l3\text{-inv}8\text{-def}$  [THEN setc-def-to-dest, rule-format]

**lemma**  $l3\text{-inv}8\text{-derived}$  [iff]:

$l3\text{-inv}2 \cap l3\text{-inv}3 \cap l3\text{-inv}4 \subseteq l3\text{-inv}8$

$\langle proof \rangle$

**lemma**  $PO\text{-}l3\text{-inv}8$  [iff]:  $reach\ l3 \subseteq l3\text{-inv}8$

$\langle proof \rangle$

### 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

$$l3\text{-inv9} \equiv \{s. \forall A B M. \\ (\text{implSecure } A B M \in \text{synth } (\text{analz } (ik\ s)) \wedge M \in \text{payload}) \longrightarrow \\ (\text{implSecure } A B M \in ik\ s \vee (M \in \text{synth } (\text{analz } (ik\ s)) \wedge (A \in \text{bad } s \vee B \in \text{bad } s))) \\ \}$$

**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**  $l3\text{-inv9-derived } [iff]:$

$l3\text{-inv2} \cap l3\text{-inv3} \cap l3\text{-inv4} \subseteq l3\text{-inv9}$

$\langle\text{proof}\rangle$

**lemma**  $PO\text{-}l3\text{-inv9 } [iff]:\ \text{reach } l3 \subseteq l3\text{-inv9}$

$\langle\text{proof}\rangle$

## 28.3 Refinement

Mediator function.

#### definition

$med23s :: l3\text{-obs} \Rightarrow l2\text{-obs}$

#### where

$$med23s\ t \equiv (\ \\ ik = ik\ t \cap \text{payload}, \\ secret = secret\ t, \\ progress = progress\ t, \\ signalsInit = signalsInit\ t, \\ signalsResp = signalsResp\ t, \\ signalsInit2 = signalsInit2\ t, \\ signalsResp2 = signalsResp2\ t, \\ chan = \text{abs } (ik\ t), \\ bad = bad\ t \\ )$$

Relation between states.

#### definition

$R23s :: (l2\text{-state} * l3\text{-state})\ \text{set}$

#### where

$R23s \equiv \{(s, s')\}.$

$$\begin{array}{l} s = \text{med23s } s' \\ \} \end{array}$$

**lemmas**  $R23s\text{-defs} = R23s\text{-def } \text{med23s}\text{-def}$

**lemma**  $R23sI$ :

$$\begin{array}{l} \llbracket ik \ s = ik \ t \cap \text{payload}; \text{secret } s = \text{secret } t; \text{progress } s = \text{progress } t; \\ \text{signalsInit } s = \text{signalsInit } t; \text{signalsResp } s = \text{signalsResp } t; \\ \text{signalsInit2 } s = \text{signalsInit2 } t; \text{signalsResp2 } s = \text{signalsResp2 } t; \\ \text{chan } s = \text{abs } (ik \ t); l2\text{-state.bad } s = \text{bad } t \rrbracket \\ \implies (s, t) \in R23s \\ \langle \text{proof} \rangle \end{array}$$

**lemma**  $R23sD$ :

$$\begin{array}{l} (s, t) \in R23s \implies \\ ik \ s = ik \ t \cap \text{payload} \wedge \text{secret } s = \text{secret } t \wedge \text{progress } s = \text{progress } t \wedge \\ \text{signalsInit } s = \text{signalsInit } t \wedge \text{signalsResp } s = \text{signalsResp } t \wedge \\ \text{signalsInit2 } s = \text{signalsInit2 } t \wedge \text{signalsResp2 } s = \text{signalsResp2 } t \wedge \\ \text{chan } s = \text{abs } (ik \ t) \wedge l2\text{-state.bad } s = \text{bad } t \\ \langle \text{proof} \rangle \end{array}$$

**lemma**  $R23sE$  [*elim*]:

$$\begin{array}{l} \llbracket (s, t) \in R23s; \\ \llbracket ik \ s = ik \ t \cap \text{payload}; \text{secret } s = \text{secret } t; \text{progress } s = \text{progress } t; \\ \text{signalsInit } s = \text{signalsInit } t; \text{signalsResp } s = \text{signalsResp } t; \\ \text{signalsInit2 } s = \text{signalsInit2 } t; \text{signalsResp2 } s = \text{signalsResp2 } t; \\ \text{chan } s = \text{abs } (ik \ t); l2\text{-state.bad } s = \text{bad } t \rrbracket \implies P \rrbracket \\ \implies P \\ \langle \text{proof} \rangle \end{array}$$

**lemma**  $\text{can-signal-R23}$  [*simp*]:

$$\begin{array}{l} (s2, s3) \in R23s \implies \\ \text{can-signal } s2 \ A \ B \longleftrightarrow \text{can-signal } s3 \ A \ B \\ \langle \text{proof} \rangle \end{array}$$

### 28.3.1 Protocol events

**lemma**  $l3\text{-step1-refines-step1}$ :

$$\{R23s\} \ l2\text{-step1 } Ra \ A \ B, \ l3\text{-step1 } Ra \ A \ B \ \{>R23s\} \\ \langle \text{proof} \rangle$$

**lemma**  $l3\text{-step2-refines-step2}$ :

$$\{R23s\} \ l2\text{-step2 } Rb \ A \ B \ Ni \ gnx, \ l3\text{-step2 } Rb \ A \ B \ Ni \ gnx \ \{>R23s\} \\ \langle \text{proof} \rangle$$

**lemma**  $l3\text{-step3-refines-step3}$ :

$$\{R23s\} \ l2\text{-step3 } Ra \ A \ B \ Nr \ gny, \ l3\text{-step3 } Ra \ A \ B \ Nr \ gny \ \{>R23s\} \\ \langle \text{proof} \rangle$$

**lemma**  $l3\text{-step4-refines-step4}$ :

$$\{R23s\} \ l2\text{-step4 } Rb \ A \ B \ Ni \ gnx, \ l3\text{-step4 } Rb \ A \ B \ Ni \ gnx \ \{>R23s\} \\ \langle \text{proof} \rangle$$

### 28.3.2 Intruder events

**lemma** *l3-dy-payload-refines-dy-fake-msg*:

$$M \in \text{payload} \implies \{R23s \cap UNIV \times l3\text{-inv}5\} \text{ l2-dy-fake-msg } M, \text{ l3-dy } M \{>R23s\}$$

*<proof>*

**lemma** *l3-dy-valid-refines-dy-fake-chan*:

$$\llbracket M \in \text{valid}; M' \in \text{abs } \{M\} \rrbracket \implies \{R23s \cap UNIV \times (l3\text{-inv}5 \cap l3\text{-inv}6 \cap l3\text{-inv}7 \cap l3\text{-inv}8 \cap l3\text{-inv}9)\} \text{ l2-dy-fake-chan } M', \text{ l3-dy } M \{>R23s\}$$

*<proof>*

**lemma** *l3-dy-valid-refines-dy-fake-chan-Un*:

$$M \in \text{valid} \implies \{R23s \cap UNIV \times (l3\text{-inv}5 \cap l3\text{-inv}6 \cap l3\text{-inv}7 \cap l3\text{-inv}8 \cap l3\text{-inv}9)\} \bigcup M'. \text{ l2-dy-fake-chan } M', \text{ l3-dy } M \{>R23s\}$$

*<proof>*

**lemma** *l3-dy-isLtKey-refines-skip*:

$$\{R23s\} \text{ Id}, \text{ l3-dy } (\text{LtK } \text{ltk}) \{>R23s\}$$

*<proof>*

**lemma** *l3-dy-others-refines-skip*:

$$\llbracket M \notin \text{range } \text{LtK}; M \notin \text{valid}; M \notin \text{payload} \rrbracket \implies \{R23s\} \text{ Id}, \text{ l3-dy } M \{>R23s\}$$

*<proof>*

**lemma** *l3-dy-refines-dy-fake-msg-dy-fake-chan-skip*:

$$\{R23s \cap UNIV \times (l3\text{-inv}5 \cap l3\text{-inv}6 \cap l3\text{-inv}7 \cap l3\text{-inv}8 \cap l3\text{-inv}9)\} \text{ l2-dy-fake-msg } M \cup (\bigcup M'. \text{ l2-dy-fake-chan } M') \cup \text{Id}, \text{ l3-dy } M \{>R23s\}$$

*<proof>*

### 28.3.3 Compromise events

**lemma** *l3-lkr-others-refines-lkr-others*:

$$\{R23s\} \text{ l2-lkr-others } A, \text{ l3-lkr-others } A \{>R23s\}$$

*<proof>*

**lemma** *l3-lkr-after-refines-lkr-after*:

$$\{R23s\} \text{ l2-lkr-after } A, \text{ l3-lkr-after } A \{>R23s\}$$

*<proof>*

**lemma** *l3-skr-refines-skr*:

$$\{R23s\} \text{ l2-skr } R \ K, \text{ l3-skr } R \ K \{>R23s\}$$

*<proof>*



**lemmas** *l3-trans-refines-l2-trans* =  
*l3-step1-refines-step1 l3-step2-refines-step2 l3-step3-refines-step3 l3-step4-refines-step4*  
*l3-dy-refines-dy-fake-msg-dy-fake-chan-skip*  
*l3-lkr-others-refines-lkr-others l3-lkr-after-refines-lkr-after l3-skr-refines-skr*

**lemma** *l3-refines-init-l2* [iff]:  
 $init\ l3 \subseteq R23s$  “ ( $init\ l2$ )  
 ⟨proof⟩

**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⟩

**lemma** *PO-obs-consistent-R23s* [iff]:  
 $obs-consistent\ R23s\ med23s\ l2\ l3$   
 ⟨proof⟩

**lemma** *l3-refines-l2* [iff]:  
 $refines$   
 $(R23s \cap$   
 $(reach\ l2 \times (l3-inv1 \cap l3-inv2 \cap l3-inv3 \cap l3-inv4)))$   
 $med23s\ l2\ l3$   
 ⟨proof⟩

**lemma** *l3-implements-l2* [iff]:  
 $implements\ med23s\ l2\ l3$   
 ⟨proof⟩

## 28.4 Derived invariants

### 28.4.1 inv10: secrets contain no implementation material

#### definition

$l3-inv10 :: l3-state\ set$

#### where

$l3-inv10 \equiv \{s.$   
 $secret\ s \subseteq payload$   
 $\}$

**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]:  
 $init\ l3 \subseteq l3-inv10$   
 ⟨proof⟩

**lemma** *l3-inv10-trans* [iff]:

$\{l3\text{-inv}10\}$  *trans*  $l3$   $\{> l3\text{-inv}10\}$   
 $\langle$ *proof* $\rangle$

**lemma** *PO-l3-inv10* [*iff*]: *reach*  $l3 \subseteq l3\text{-inv}10$   
 $\langle$ *proof* $\rangle$

**lemma** *l3-obs-inv10* [*iff*]: *oreach*  $l3 \subseteq l3\text{-inv}10$   
 $\langle$ *proof* $\rangle$

### 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*  $\equiv \{s.$

$\text{dy-fake-msg } (\text{bad } s) (ik\ s \cap \text{payload}) (\text{abs } (ik\ s)) \cap \text{secret } s = \{\}$   
 $\}$

**lemma** *l3-obs-partial-secrecy* [*iff*]: *oreach*  $l3 \subseteq l3\text{-partial-secrecy}$   
 $\langle$ *proof* $\rangle$

### 28.4.3 Secrecy

#### definition

*l3-secrecy* :: ('a *l3-state-scheme*) *set*

#### where

*l3-secrecy*  $\equiv l1\text{-secrecy}$

**lemma** *l3-obs-inv5*: *oreach*  $l3 \subseteq l3\text{-inv}5$   
 $\langle$ *proof* $\rangle$

**lemma** *l3-obs-secrecy* [*iff*]: *oreach*  $l3 \subseteq l3\text{-secrecy}$   
 $\langle$ *proof* $\rangle$

**lemma** *l3-secrecy* [*iff*]: *reach*  $l3 \subseteq l3\text{-secrecy}$   
 $\langle$ *proof* $\rangle$

### 28.4.4 Injective agreement

**abbreviation** *l3-iagreement-Init*  $\equiv l1\text{-iagreement-Init}$

**lemma** *l3-obs-iagreement-Init* [*iff*]: *oreach*  $l3 \subseteq l3\text{-iagreement-Init}$   
 $\langle$ *proof* $\rangle$

**lemma** *l3-iagreement-Init* [*iff*]: *reach*  $l3 \subseteq l3\text{-iagreement-Init}$   
 $\langle$ *proof* $\rangle$

**abbreviation**  $l3\text{-iagreement-Resp} \equiv l1\text{-iagreement-Resp}$

**lemma**  $l3\text{-obs-iagreement-Resp}$  [iff]:  $\text{oreach } l3 \subseteq l3\text{-iagreement-Resp}$   
 $\langle \text{proof} \rangle$

**lemma**  $l3\text{-iagreement-Resp}$  [iff]:  $\text{reach } l3 \subseteq l3\text{-iagreement-Resp}$   
 $\langle \text{proof} \rangle$

**end**

**end**

## 29 SKEME Protocol (L3 with asymmetric implementation)

**theory** *sklv3-asymmetric*

**imports** *sklv3 Implem-asymmetric*

**begin**

**interpretation** *sklv3-asym: sklv3 implem-asym*  
*<proof>*

**end**

### 30 SKEME Protocol (L3 with symmetric implementation)

```
theory sklv3-symmetric  
imports sklv3 Implem-symmetric  
begin  
  
interpretation sklv3-sym: sklv3 implem-sym  
<proof>  
  
end
```