

# Security Protocols

Giampaolo Bella, Frederic Blanqui, Lawrence C. Paulson et al.

April 19, 2009

## Contents

<b>1</b>	<b>Theory of Agents and Messages for Security Protocols</b>	<b>12</b>
1.0.1	Inductive Definition of All Parts” of a Message . . . . .	13
1.0.2	Inverse of keys . . . . .	13
1.1	keysFor operator . . . . .	13
1.2	Inductive relation ”parts” . . . . .	14
1.2.1	Unions . . . . .	15
1.2.2	Idempotence and transitivity . . . . .	15
1.2.3	Rewrite rules for pulling out atomic messages . . . . .	16
1.3	Inductive relation ”analz” . . . . .	17
1.3.1	General equational properties . . . . .	17
1.3.2	Rewrite rules for pulling out atomic messages . . . . .	18
1.3.3	Idempotence and transitivity . . . . .	19
1.4	Inductive relation ”synth” . . . . .	20
1.4.1	Unions . . . . .	21
1.4.2	Idempotence and transitivity . . . . .	21
1.4.3	Combinations of parts, analz and synth . . . . .	22
1.4.4	For reasoning about the Fake rule in traces . . . . .	22
1.5	HPair: a combination of Hash and MPair . . . . .	23
1.5.1	Freeness . . . . .	23
1.5.2	Specialized laws, proved in terms of those for Hash and MPair . . . . .	24
1.6	Tactics useful for many protocol proofs . . . . .	25
<b>2</b>	<b>Theory of Events for Security Protocols</b>	<b>25</b>
2.1	Function <i>knows</i> . . . . .	27
2.2	Knowledge of Agents . . . . .	28
2.2.1	Useful for case analysis on whether a hash is a spoof or not	30
2.3	Asymmetric Keys . . . . .	30
2.4	Basic properties of <i>pubK</i> and $\lambda A. invKey (pubK A)$ . . . . .	31
2.5	”Image” equations that hold for injective functions . . . . .	31
2.6	Symmetric Keys . . . . .	32
2.7	Initial States of Agents . . . . .	33
2.8	Function <i>knows Spy</i> . . . . .	34
2.9	Fresh Nonces . . . . .	35
2.10	Supply fresh nonces for possibility theorems . . . . .	35
2.11	Specialized Rewriting for Theorems About <i>analz</i> and Image . . . . .	36
2.12	Specialized Methods for Possibility Theorems . . . . .	36

<b>3</b>	<b>Needham-Schroeder Shared-Key Protocol and the Issues Property</b>	<b>36</b>
3.1	Inductive proofs about <i>ns_shared</i>	38
3.1.1	Forwarding lemmas, to aid simplification	38
3.1.2	Lemmas concerning the form of items passed in messages	38
3.1.3	Session keys are not used to encrypt other session keys	39
3.1.4	The session key K uniquely identifies the message	39
3.1.5	Crucial secrecy property: Spy doesn't see the keys sent in NS2	40
3.2	Guarantees available at various stages of protocol	40
3.3	Lemmas for reasoning about predicate "Issues"	41
3.4	Guarantees of non-injective agreement on the session key, and of key distribution. They also express forms of freshness of certain messages, namely that agents were alive after something happened.	42
<b>4</b>	<b>The Kerberos Protocol, BAN Version</b>	<b>43</b>
4.1	Lemmas for reasoning about predicate "Issues"	45
4.2	Lemmas concerning the form of items passed in messages	46
4.3	Non-temporal guarantees, explicitly relying on non-occurrence of oops events - refined below by temporal guarantees	48
4.4	Temporal guarantees, relying on a temporal check that insures that no oops event occurred. These are available in the sense of goal availability	50
4.5	Treatment of the key distribution goal using trace inspection. All guarantees are in non-temporal form, hence non available, though their temporal form is trivial to derive. These guarantees also convey a stronger form of authentication - non-injective agreement on the session key	51
<b>5</b>	<b>The Kerberos Protocol, BAN Version, with Gets event</b>	<b>52</b>
5.1	Lemmas concerning the form of items passed in messages	56
5.2	Non-temporal guarantees, explicitly relying on non-occurrence of oops events - refined below by temporal guarantees	58
5.3	Temporal guarantees, relying on a temporal check that insures that no oops event occurred. These are available in the sense of goal availability	60
5.4	Combined guarantees of key distribution and non-injective agreement on the session keys	61
<b>6</b>	<b>The Kerberos Protocol, Version IV</b>	<b>62</b>
6.1	Lemmas about lists, for reasoning about Issues	66
6.2	Lemmas about <i>authKeys</i>	67
6.3	Forwarding Lemmas	67
6.4	Lemmas for reasoning about predicate "before"	68
6.5	Regularity Lemmas	69
6.6	Authenticity theorems: confirm origin of sensitive messages	71
6.7	Reliability: friendly agents send something if something else happened	73
6.8	Unicity Theorems	74
6.9	Lemmas About the Predicate <i>AKcryptSK</i>	75

6.10	Secrecy Theorems . . . . .	77
6.11	Parties authentication: each party verifies "the identity of another party who generated some data" (quoted from Neuman and Ts'o). . . . .	81
6.12	Key distribution guarantees An agent knows a session key if he used it to issue a cipher. These guarantees also convey a stronger form of authentication - non-injective agreement on the session key	82
<b>7</b>	<b>The Kerberos Protocol, Version IV</b>	<b>86</b>
7.1	Lemmas about reception event . . . . .	90
7.2	Lemmas about <i>authKeys</i> . . . . .	90
7.3	Forwarding Lemmas . . . . .	91
7.4	Regularity Lemmas . . . . .	92
7.5	Authenticity theorems: confirm origin of sensitive messages . . .	93
7.6	Reliability: friendly agents send something if something else happened . . . . .	96
7.7	Unicity Theorems . . . . .	97
7.8	Lemmas About the Predicate <i>AKcryptSK</i> . . . . .	98
7.9	Secrecy Theorems . . . . .	100
7.10	2. Parties' strong authentication: non-injective agreement on the session key. The same guarantees also express key distribution, hence their names . . . . .	103
<b>8</b>	<b>The Kerberos Protocol, Version V</b>	<b>105</b>
8.1	Lemmas about lists, for reasoning about Issues . . . . .	109
8.2	Lemmas about <i>authKeys</i> . . . . .	110
8.3	Forwarding Lemmas . . . . .	110
8.4	Regularity Lemmas . . . . .	111
8.5	Authenticity theorems: confirm origin of sensitive messages . . .	112
8.6	Reliability: friendly agents send something if something else happened . . . . .	114
8.7	Unicity Theorems . . . . .	116
8.8	Lemmas About the Predicate <i>AKcryptSK</i> . . . . .	116
8.9	Secrecy Theorems . . . . .	118
8.10	Parties authentication: each party verifies "the identity of another party who generated some data" (quoted from Neuman and Ts'o). . . . .	121
8.11	Parties' knowledge of session keys. An agent knows a session key if he used it to issue a cipher. These guarantees can be interpreted both in terms of key distribution and of non-injective agreement on the session key. . . . .	122
8.12	Novel guarantees, never studied before. Because honest agents always say the right timestamp in authenticators, we can prove unicity guarantees based exactly on timestamps. Classical unicity guarantees are based on nonces. Of course assuming the agent to be different from the Spy, rather than not in bad, would suffice below. Similar guarantees must also hold of Kerberos IV. . . . .	124

<b>9 The Original Otway-Rees Protocol</b>	<b>126</b>
9.1 Towards Secrecy: Proofs Involving <i>analz</i>	128
9.2 Authenticity properties relating to NA	128
9.3 Authenticity properties relating to NB	130
<b>10 The Otway-Rees Protocol as Modified by Abadi and Needham</b>	<b>132</b>
10.1 Proofs involving <i>analz</i>	134
10.2 Authenticity properties relating to NA	135
10.3 Authenticity properties relating to NB	136
<b>11 The Otway-Rees Protocol: The Faulty BAN Version</b>	<b>137</b>
11.1 For reasoning about the encrypted portion of messages	139
11.2 Proofs involving <i>analz</i>	139
11.3 Attempting to prove stronger properties	140
<b>12 Bella's version of the Otway-Rees protocol</b>	<b>141</b>
12.1 Proofs involving <i>analz</i>	143
<b>13 The Woo-Lam Protocol</b>	<b>146</b>
<b>14 The Otway-Bull Recursive Authentication Protocol</b>	<b>149</b>
<b>15 The Yahalom Protocol</b>	<b>154</b>
15.1 Regularity Lemmas for Yahalom	156
15.2 Secrecy Theorems	157
15.2.1 Security Guarantee for A upon receiving YM3	158
15.2.2 Security Guarantees for B upon receiving YM4	158
15.2.3 Towards proving secrecy of Nonce NB	159
15.2.4 The Nonce NB uniquely identifies B's message.	160
15.2.5 A nonce value is never used both as NA and as NB	160
15.3 Authenticating B to A	162
15.4 Authenticating A to B using the certificate <i>Crypt K (Nonce NB)</i>	162
<b>16 The Yahalom Protocol, Variant 2</b>	<b>163</b>
16.1 Inductive Proofs	164
16.2 Crucial Secrecy Property: Spy Does Not See Key <i>KAB</i>	166
16.3 Security Guarantee for A upon receiving YM3	166
16.4 Security Guarantee for B upon receiving YM4	167
16.5 Authenticating B to A	167
16.6 Authenticating A to B	168
<b>17 The Yahalom Protocol: A Flawed Version</b>	<b>169</b>
17.1 Regularity Lemmas for Yahalom	170
17.2 For reasoning about the encrypted portion of messages	170
17.3 Secrecy Theorems	171
17.4 Session keys are not used to encrypt other session keys	171
17.5 Security Guarantee for A upon receiving YM3	172
17.6 Security Guarantees for B upon receiving YM4	172
17.7 The Flaw in the Model	172
17.8 Basic Lemmas	175
17.9 About NRO: Validity for B	176

17.10 About NRR: Validity for $A$	177
17.11 Proofs About $sub\_K$	177
17.12 Proofs About $con\_K$	178
17.13 Proving fairness	178
<b>18 Verifying the Needham-Schroeder Public-Key Protocol</b>	<b>180</b>
<b>19 Verifying the Needham-Schroeder-Lowe Public-Key Protocol</b>	<b>183</b>
19.1 Authenticity properties obtained from NS2	184
19.2 Authenticity properties obtained from NS2	184
19.3 Overall guarantee for B	185
<b>20 Bijections <math>\mathbb{N} \rightarrow \mathbb{N}^2</math> and <math>\mathbb{N} \rightarrow \mathbb{Z}</math></b>	<b>185</b>
20.1 A bijection between $\mathbb{N}$ and $\mathbb{N}^2$	185
20.2 A bijection between $\mathbb{N}$ and $\mathbb{Z}$	186
<b>21 The TLS Protocol: Transport Layer Security</b>	<b>187</b>
21.1 Protocol Proofs	191
21.2 Inductive proofs about $tls$	192
21.2.1 Properties of items found in Notes	193
21.2.2 Protocol goal: if B receives CertVerify, then A sent it	193
21.2.3 Unicity results for PMS, the pre-master-secret	194
21.3 Secrecy Theorems	194
21.3.1 Protocol goal: $serverK(Na, Nb, M)$ and $clientK(Na, Nb, M)$ remain secure	195
21.3.2 Weakening the Oops conditions for leakage of $clientK$	196
21.3.3 Weakening the Oops conditions for leakage of $serverK$	196
21.3.4 Protocol goals: if A receives ServerFinished, then B is present and has used the quoted values PA, PB, etc. Note that it is up to A to compare PA with what she originally sent.	197
21.3.5 Protocol goal: if B receives any message encrypted with $clientK$ then A has sent it	197
21.3.6 Protocol goal: if B receives ClientFinished, and if B is able to check a CertVerify from A, then A has used the quoted values PA, PB, etc. Even this one requires A to be uncompromised.	198
<b>22 The Certified Electronic Mail Protocol by Abadi et al.</b>	<b>198</b>
22.1 Proving Confidentiality Results	202
22.2 The Guarantees for Sender and Recipient	203
<b>23 Theory of Events for Security Protocols that use smartcards</b>	<b>204</b>
23.1 Function <i>knows</i>	206
23.2 Knowledge of Agents	208
<b>24 Theory of smartcards</b>	<b>211</b>
24.1 Basic properties of $shrK$	213
24.2 Function "knows"	214
24.3 Fresh nonces	215
24.4 Supply fresh nonces for possibility theorems.	215

24.5	Specialized Rewriting for Theorems About <i>analz</i> and Image . . .	216
24.6	Tactics for possibility theorems . . . . .	216
<b>25</b>	<b>Original Shoup-Rubin protocol</b>	<b>217</b>
<b>26</b>	<b>Bella's modification of the Shoup-Rubin protocol</b>	<b>236</b>
<b>27</b>	<b>Extensions to Standard Theories</b>	<b>254</b>
27.1	Extensions to Theory <i>Set</i> . . . . .	254
27.2	Extensions to Theory <i>List</i> . . . . .	255
27.2.1	"remove l x" erase the first element of "l" equal to "x" . .	255
27.3	Extensions to Theory <i>Message</i> . . . . .	255
27.3.1	declarations for tactics . . . . .	255
27.3.2	extract the agent number of an Agent message . . . . .	255
27.3.3	messages that are pairs . . . . .	255
27.3.4	well-foundedness of messages . . . . .	256
27.3.5	lemmas on keysFor . . . . .	256
27.3.6	lemmas on parts . . . . .	256
27.3.7	lemmas on synth . . . . .	257
27.3.8	lemmas on analz . . . . .	257
27.3.9	lemmas on parts, synth and analz . . . . .	258
27.3.10	greatest nonce used in a message . . . . .	258
27.3.11	sets of keys . . . . .	258
27.3.12	keys a priori necessary for decrypting the messages of G .	259
27.3.13	only the keys necessary for G are useful in analz . . . . .	259
27.4	Extensions to Theory <i>Event</i> . . . . .	259
27.4.1	general protocol properties . . . . .	259
27.4.2	lemma on knows . . . . .	260
27.4.3	knows without initState . . . . .	260
27.4.4	decomposition of knows into knows' and initState . . . . .	261
27.4.5	knows' is finite . . . . .	261
27.4.6	monotonicity of knows . . . . .	261
27.4.7	maximum knowledge an agent can have includes messages sent to the agent . . . . .	262
27.4.8	basic facts about <i>knows_max</i> . . . . .	262
27.4.9	used without initState . . . . .	263
27.4.10	monotonicity of used . . . . .	263
27.4.11	lemmas on used and knows . . . . .	264
27.4.12	a nonce or key in a message cannot equal a fresh nonce or key . . . . .	265
27.4.13	message of an event . . . . .	265
<b>28</b>	<b>Decomposition of Analz into two parts</b>	<b>265</b>
28.1	messages that do not contribute to analz . . . . .	265
28.2	basic facts about <i>pparts</i> . . . . .	265
28.3	facts about <i>pparts</i> and <i>parts</i> . . . . .	267
28.4	facts about <i>pparts</i> and <i>analz</i> . . . . .	267
28.5	messages that contribute to analz . . . . .	267
28.6	basic facts about <i>kparts</i> . . . . .	267
28.7	facts about <i>kparts</i> and <i>parts</i> . . . . .	269

28.8 facts about <i>kparts</i> and <i>analz</i> . . . . .	269
28.9 <i>analz</i> is <i>pparts</i> + <i>analz</i> of <i>kparts</i> . . . . .	270
<b>29 Protocol-Independent Confidentiality Theorem on Nonces</b> . . . . .	<b>270</b>
29.1 basic facts about <i>guard</i> . . . . .	270
29.2 guarded sets . . . . .	271
29.3 basic facts about <i>Guard</i> . . . . .	271
29.4 set obtained by decrypting a message . . . . .	273
29.5 number of Crypt's in a message . . . . .	273
29.6 basic facts about <i>crypt_nb</i> . . . . .	273
29.7 number of Crypt's in a message list . . . . .	273
29.8 basic facts about <i>cnb</i> . . . . .	273
29.9 list of <i>kparts</i> . . . . .	274
29.10 list corresponding to "decrypt" . . . . .	274
29.11 basic facts about <i>decrypt'</i> . . . . .	274
29.12 if the analyse of a finite guarded set gives <i>n</i> then it must also gives one of the keys of <i>Ks</i> . . . . .	274
29.13 if the analyse of a finite guarded set and a (possibly infinite) set of keys gives <i>n</i> then it must also gives <i>Ks</i> . . . . .	274
29.14 Extensions to Theory <i>Public</i> . . . . .	275
29.14.1 signature . . . . .	275
29.14.2 agent associated to a key . . . . .	275
29.14.3 basic facts about <i>initState</i> . . . . .	275
29.14.4 sets of private keys . . . . .	275
29.14.5 sets of good keys . . . . .	276
29.14.6 greatest nonce used in a trace, 0 if there is no nonce . . . . .	276
29.14.7 function giving a new nonce . . . . .	276
29.15 Proofs About Guarded Messages . . . . .	276
29.15.1 small hack necessary because <i>priK</i> is defined as the inverse of <i>pubK</i> . . . . .	276
29.15.2 guardedness results . . . . .	276
29.15.3 regular protocols . . . . .	277
<b>30 Lists of Messages and Lists of Agents</b> . . . . .	<b>277</b>
30.1 Implementation of Lists by Messages . . . . .	277
30.1.1 nil is represented by any message which is not a pair . . . . .	277
30.1.2 induction principle . . . . .	278
30.1.3 head . . . . .	278
30.1.4 tail . . . . .	278
30.1.5 length . . . . .	278
30.1.6 membership . . . . .	278
30.1.7 delete an element . . . . .	278
30.1.8 concatenation . . . . .	279
30.1.9 replacement . . . . .	279
30.1.10 <i>i</i> th element . . . . .	279
30.1.11 insertion . . . . .	279
30.1.12 truncation . . . . .	279
30.2 Agent Lists . . . . .	280
30.2.1 set of well-formed agent-list messages . . . . .	280
30.2.2 basic facts about agent lists . . . . .	280

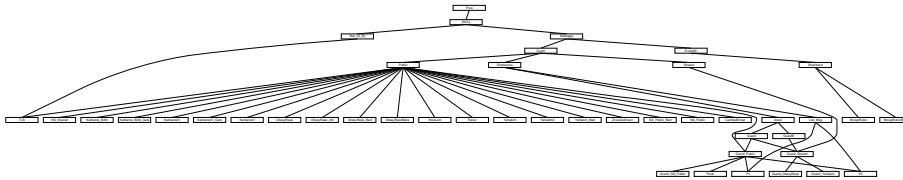
<b>31 Protocol P1</b>	<b>280</b>
31.1 Protocol Definition	280
31.1.1 offer chaining: B chains his offer for A with the head offer of L for sending it to C	280
31.1.2 agent whose key is used to sign an offer	281
31.1.3 nonce used in an offer	281
31.1.4 next shop	281
31.1.5 anchor of the offer list	281
31.1.6 request event	282
31.1.7 propose event	282
31.1.8 protocol	283
31.1.9 Composition of Traces	283
31.1.10 Valid Offer Lists	283
31.1.11 basic properties of valid	283
31.1.12 offers of an offer list	283
31.1.13 the originator can get the offers	283
31.1.14 list of offers	283
31.1.15 list of agents whose keys are used to sign a list of offers	284
31.1.16 builds a trace from an itinerary	284
31.1.17 there is a trace in which the originator receives a valid answer	284
31.2 properties of protocol P1	285
31.2.1 strong forward integrity: except the last one, no offer can be modified	285
31.2.2 insertion resilience: except at the beginning, no offer can be inserted	285
31.2.3 truncation resilience: only shop i can truncate at offer i	285
31.2.4 declarations for tactics	285
31.2.5 get components of a message	285
31.2.6 general properties of p1	285
31.2.7 private keys are safe	286
31.2.8 general guardedness properties	286
31.2.9 guardedness of messages	287
31.2.10 Nonce uniqueness	287
31.2.11 requests are guarded	287
31.2.12 propositions are guarded	288
31.2.13 data confidentiality: no one other than the originator can decrypt the offers	288
31.2.14 non repudiability: an offer signed by B has been sent by B	288
<b>32 Protocol P2</b>	<b>289</b>
32.1 Protocol Definition	289
32.1.1 offer chaining: B chains his offer for A with the head offer of L for sending it to C	289
32.1.2 agent whose key is used to sign an offer	289
32.1.3 nonce used in an offer	289
32.1.4 next shop	290
32.1.5 anchor of the offer list	290
32.1.6 request event	290
32.1.7 propose event	290



32.1.8	protocol	291
32.1.9	valid offer lists	291
32.1.10	basic properties of valid	291
32.1.11	list of offers	291
32.2	Properties of Protocol P2	292
32.3	strong forward integrity: except the last one, no offer can be modified	292
32.4	insertion resilience: except at the beginning, no offer can be inserted	292
32.5	truncation resilience: only shop i can truncate at offer i	292
32.6	declarations for tactics	292
32.7	get components of a message	292
32.8	general properties of p2	292
32.9	private keys are safe	293
32.10	general guardedness properties	293
32.11	guardedness of messages	294
32.12	Nonce uniqueness	294
32.13	requests are guarded	295
32.14	propositions are guarded	295
32.15	data confidentiality: no one other than the originator can decrypt the offers	295
32.16	forward privacy: only the originator can know the identity of the shops	295
32.17	non repudiability: an offer signed by B has been sent by B	296
<b>33</b>	<b>Needham-Schroeder-Lowe Public-Key Protocol</b>	<b>296</b>
33.1	messages used in the protocol	296
33.2	definition of the protocol	297
33.3	declarations for tactics	297
33.4	general properties of nsp	297
33.5	nonce are used only once	298
33.6	guardedness of NA	298
33.7	guardedness of NB	298
33.8	Agents' Authentication	298
<b>34</b>	<b>protocol-independent confidentiality theorem on keys</b>	<b>299</b>
34.1	basic facts about <i>guardK</i>	299
34.2	guarded sets	300
34.3	basic facts about <i>GuardK</i>	300
34.4	set obtained by decrypting a message	301
34.5	number of Crypt's in a message	301
34.6	basic facts about <i>crypt_nb</i>	302
34.7	number of Crypt's in a message list	302
34.8	basic facts about <i>cnb</i>	302
34.9	list of kparts	302
34.10	list corresponding to "decrypt"	302
34.11	basic facts about <i>decrypt'</i>	303
34.12	Basic properties of shrK	304
34.13	Function "knows"	304
34.14	Fresh nonces	305
34.15	Supply fresh nonces for possibility theorems.	305

34.16	Specialized Rewriting for Theorems About <code>analz</code> and Image . . .	305
34.17	Tactics for possibility theorems . . . . .	306
<b>35</b>	<b>lemmas on guarded messages for protocols with symmetric keys</b>	<b>306</b>
35.1	Extensions to Theory <code>Shared</code> . . . . .	306
35.1.1	a little abbreviation . . . . .	306
35.1.2	agent associated to a key . . . . .	307
35.1.3	basic facts about <code>initState</code> . . . . .	307
35.1.4	sets of symmetric keys . . . . .	307
35.1.5	sets of good keys . . . . .	307
35.2	Proofs About Guarded Messages . . . . .	308
35.2.1	small hack . . . . .	308
35.2.2	guardedness results on nonces . . . . .	308
35.2.3	guardedness results on keys . . . . .	308
35.2.4	regular protocols . . . . .	309
<b>36</b>	<b>Otway-Rees Protocol</b>	<b>309</b>
36.1	messages used in the protocol . . . . .	309
36.2	definition of the protocol . . . . .	310
36.3	declarations for tactics . . . . .	311
36.4	general properties of <code>or</code> . . . . .	311
36.5	<code>or</code> is regular . . . . .	311
36.6	guardedness of <code>KAB</code> . . . . .	311
36.7	guardedness of <code>NB</code> . . . . .	312
<b>37</b>	<b>Yahalom Protocol</b>	<b>312</b>
37.1	messages used in the protocol . . . . .	312
37.2	definition of the protocol . . . . .	313
37.3	declarations for tactics . . . . .	313
37.4	general properties of <code>ya</code> . . . . .	313
37.5	guardedness of <code>KAB</code> . . . . .	313
37.6	session keys are not symmetric keys . . . . .	314
37.7	<code>ya2'</code> implies <code>ya1'</code> . . . . .	314
37.8	uniqueness of <code>NB</code> . . . . .	314
37.9	<code>ya3'</code> implies <code>ya2'</code> . . . . .	314
37.10	<code>ya3'</code> implies <code>ya3</code> . . . . .	314
37.11	guardedness of <code>NB</code> . . . . .	315
<b>38</b>	<b>Other Protocol-Independent Results</b>	<b>315</b>
38.1	protocols . . . . .	315
38.2	substitutions . . . . .	315
38.3	nonces generated by a rule . . . . .	316
38.4	traces generated by a protocol . . . . .	316
38.5	general properties . . . . .	317
38.6	types . . . . .	317
38.7	introduction of a fresh guarded nonce . . . . .	317
38.8	safe keys . . . . .	318
38.9	guardedness preservation . . . . .	319
38.10	monotonic <code>keyfun</code> . . . . .	319
38.11	guardedness theorem . . . . .	319

38.12	useful properties for guardedness . . . . .	319
38.13	unicity . . . . .	319
38.14	Needham-Schroeder-Lowe . . . . .	320
38.15	general properties . . . . .	321
38.16	guardedness for NSL . . . . .	322
38.17	unicity for NSL . . . . .	322



# 1 Theory of Agents and Messages for Security Protocols

```
theory Message
imports Main
begin
```

```
lemma [simp] : "A  $\cup$  (B  $\cup$  A) = B  $\cup$  A"
<proof>
```

```
types
  key = nat
```

```
consts
  all_symmetric :: bool          — true if all keys are symmetric
  invKey         :: "key=>key"    — inverse of a symmetric key
```

```
specification (invKey)
  invKey [simp]: "invKey (invKey K) = K"
  invKey_symmetric: "all_symmetric --> invKey = id"
  <proof>
```

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

```
constdefs
  symKeys :: "key set"
  "symKeys == {K. invKey K = K}"
```

```
datatype — We allow any number of friendly agents
  agent = Server | Friend nat | Spy
```

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

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

```
syntax
  "@MTuple"      :: "'a, args] => 'a * 'b"      ("(2{_,/ _})")
```

```
syntax (xsymbols)
  "@MTuple"      :: "'a, args] => 'a * 'b"      ("(2{_,/ _})")
```

```
translations
  "{|x, y, z|}" == "{|x, {|y, z|}|}"
  "{|x, y|}"    == "MPair x y"
```

**constdefs**

```

HPair :: "[msg,msg] => msg"
      — Message Y paired with a MAC computed with the help of X
      "Hash[X] Y == {| Hash{|X,Y|}, Y|}"

keysFor :: "msg set => key set"
      — Keys useful to decrypt elements of a message set
      "keysFor H == invKey ' {K. ∃X. Crypt K X ∈ H}"

```

**1.0.1 Inductive Definition of All Parts” of a Message****inductive\_set**

```

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

```

Monotonicity

```

lemma parts_mono: "G ⊆ H ==> parts(G) ⊆ parts(H)"
<proof>

```

Equations hold because constructors are injective.

```

lemma Friend_image_eq [simp]: "(Friend x ∈ Friend'A) = (x:A)"
<proof>

```

```

lemma Key_image_eq [simp]: "(Key x ∈ Key'A) = (x∈A)"
<proof>

```

```

lemma Nonce_Key_image_eq [simp]: "(Nonce x ∉ Key'A)"
<proof>

```

**1.0.2 Inverse of keys**

```

lemma invKey_eq [simp]: "(invKey K = invKey K') = (K=K')"
<proof>

```

**1.1 keysFor operator**

```

lemma keysFor_empty [simp]: "keysFor {} = {}"
<proof>

```

```

lemma keysFor_Un [simp]: "keysFor (H ∪ H') = keysFor H ∪ keysFor H'"
<proof>

```

```

lemma keysFor_UN [simp]: "keysFor (⋃ i∈A. H i) = (⋃ i∈A. keysFor (H i))"
<proof>

```

Monotonicity

```

lemma keysFor_mono: "G ⊆ H ==> keysFor(G) ⊆ keysFor(H)"
<proof>

```

```
lemma keysFor_insert_Agent [simp]: "keysFor (insert (Agent A) H) = keysFor
H"
<proof>
```

```
lemma keysFor_insert_Nonce [simp]: "keysFor (insert (Nonce N) H) = keysFor
H"
<proof>
```

```
lemma keysFor_insert_Number [simp]: "keysFor (insert (Number N) H) = keysFor
H"
<proof>
```

```
lemma keysFor_insert_Key [simp]: "keysFor (insert (Key K) H) = keysFor H"
<proof>
```

```
lemma keysFor_insert_Hash [simp]: "keysFor (insert (Hash X) H) = keysFor
H"
<proof>
```

```
lemma keysFor_insert_MPair [simp]: "keysFor (insert {|X,Y|} H) = keysFor
H"
<proof>
```

```
lemma keysFor_insert_Crypt [simp]:
  "keysFor (insert (Crypt K X) H) = insert (invKey K) (keysFor H)"
<proof>
```

```
lemma keysFor_image_Key [simp]: "keysFor (Key'E) = {}"
<proof>
```

```
lemma Crypt_imp_invKey_keysFor: "Crypt K X ∈ H ==> invKey K ∈ keysFor H"
<proof>
```

## 1.2 Inductive relation "parts"

```
lemma MPair_parts:
  "[| {|X,Y|} ∈ parts H;
    [| X ∈ parts H; Y ∈ parts H |] ==> P |] ==> P"
<proof>
```

```
declare MPair_parts [elim!] parts.Body [dest!]
```

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

```
lemma parts_increasing: "H ⊆ parts(H)"
<proof>
```

```
lemmas parts_insertI = subset_insertI [THEN parts_mono, THEN subsetD, standard]
```

```
lemma parts_empty [simp]: "parts{} = {}"
<proof>
```

```
lemma parts_emptyE [elim!]: "X ∈ parts{} ==> P"
<proof>
```

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

```
lemma parts_singleton: "X ∈ parts H ==> ∃ Y ∈ H. X ∈ parts {Y}"
<proof>
```

### 1.2.1 Unions

```
lemma parts_Un_subset1: "parts(G) ∪ parts(H) ⊆ parts(G ∪ H)"
<proof>
```

```
lemma parts_Un_subset2: "parts(G ∪ H) ⊆ parts(G) ∪ parts(H)"
<proof>
```

```
lemma parts_Un [simp]: "parts(G ∪ H) = parts(G) ∪ parts(H)"
<proof>
```

```
lemma parts_insert: "parts (insert X H) = parts {X} ∪ parts H"
<proof>
```

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

```
lemma parts_insert2:
  "parts (insert X (insert Y H)) = parts {X} ∪ parts {Y} ∪ parts H"
<proof>
```

```
lemma parts_UN_subset1: "(⋃ x ∈ A. parts(H x)) ⊆ parts(⋃ x ∈ A. H x)"
<proof>
```

```
lemma parts_UN_subset2: "parts(⋃ x ∈ A. H x) ⊆ (⋃ x ∈ A. parts(H x))"
<proof>
```

```
lemma parts_UN [simp]: "parts(⋃ x ∈ A. H x) = (⋃ x ∈ A. parts(H x))"
<proof>
```

Added to simplify arguments to parts, analz and synth. NOTE: the UN versions are no longer used!

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

```
lemmas in_parts_UnE = parts_Un [THEN equalityD1, THEN subsetD, THEN UnE]
```

```
declare in_parts_UnE [elim!]
```

```
lemma parts_insert_subset: "insert X (parts H) ⊆ parts(insert X H)"
<proof>
```

### 1.2.2 Idempotence and transitivity

```
lemma parts_partsD [dest!]: "X ∈ parts (parts H) ==> X ∈ parts H"
<proof>
```

```
lemma parts_idem [simp]: "parts (parts H) = parts H"
```

*<proof>*

**lemma** parts\_subset\_iff [simp]: "(parts G  $\subseteq$  parts H) = (G  $\subseteq$  parts H)"  
*<proof>*

**lemma** parts\_trans: "[| X  $\in$  parts G; G  $\subseteq$  parts H |] ==> X  $\in$  parts H"  
*<proof>*

Cut

**lemma** parts\_cut:  
 "[| Y  $\in$  parts (insert X G); X  $\in$  parts H |] ==> Y  $\in$  parts (G  $\cup$  H)"  
*<proof>*

**lemma** parts\_cut\_eq [simp]: "X  $\in$  parts H ==> parts (insert X H) = parts H"  
*<proof>*

### 1.2.3 Rewrite rules for pulling out atomic messages

**lemmas** parts\_insert\_eq\_I = equalityI [OF subsetI parts\_insert\_subset]

**lemma** parts\_insert\_Agent [simp]:  
 "parts (insert (Agent agt) H) = insert (Agent agt) (parts H)"  
*<proof>*

**lemma** parts\_insert\_Nonce [simp]:  
 "parts (insert (Nonce N) H) = insert (Nonce N) (parts H)"  
*<proof>*

**lemma** parts\_insert\_Number [simp]:  
 "parts (insert (Number N) H) = insert (Number N) (parts H)"  
*<proof>*

**lemma** parts\_insert\_Key [simp]:  
 "parts (insert (Key K) H) = insert (Key K) (parts H)"  
*<proof>*

**lemma** parts\_insert\_Hash [simp]:  
 "parts (insert (Hash X) H) = insert (Hash X) (parts H)"  
*<proof>*

**lemma** parts\_insert\_Crypt [simp]:  
 "parts (insert (Crypt K X) H) = insert (Crypt K X) (parts (insert X H))"  
*<proof>*

**lemma** parts\_insert\_MPair [simp]:  
 "parts (insert {|X,Y|} H) =  
 insert {|X,Y|} (parts (insert X (insert Y H)))"  
*<proof>*

**lemma** parts\_image\_Key [simp]: "parts (Key'N) = Key'N"  
*<proof>*

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



```
lemma msg_Nonce_supply: "∃ N. ∀ n. N ≤ n --> Nonce n ∉ parts {msg}"
<proof>
```

### 1.3 Inductive relation "analz"

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

```
inductive_set
  analz :: "msg set => msg set"
  for H :: "msg set"
  where
    Inj [intro,simp] : "X ∈ H ==> X ∈ analz H"
  | Fst : "{|X,Y|} ∈ analz H ==> X ∈ analz H"
  | Snd : "{|X,Y|} ∈ analz H ==> Y ∈ analz H"
  | Decrypt [dest]:
    "[|Crypt K X ∈ analz H; Key(invKey K): analz H|] ==> X ∈ analz
H"
```

Monotonicity; Lemma 1 of Lowe's paper

```
lemma analz_mono: "G ⊆ H ==> analz(G) ⊆ analz(H)"
<proof>
```

Making it safe speeds up proofs

```
lemma MPair_analz [elim!]:
  "[| {|X,Y|} ∈ analz H;
    [| X ∈ analz H; Y ∈ analz H |] ==> P
  |] ==> P"
<proof>
```

```
lemma analz_increasing: "H ⊆ analz(H)"
<proof>
```

```
lemma analz_subset_parts: "analz H ⊆ parts H"
<proof>
```

```
lemmas analz_into_parts = analz_subset_parts [THEN subsetD, standard]
```

```
lemmas not_parts_not_analz = analz_subset_parts [THEN contra_subsetD, standard]
```

```
lemma parts_analz [simp]: "parts (analz H) = parts H"
<proof>
```

```
lemma analz_parts [simp]: "analz (parts H) = parts H"
<proof>
```

```
lemmas analz_insertI = subset_insertI [THEN analz_mono, THEN [2] rev_subsetD,
standard]
```

#### 1.3.1 General equational properties

```
lemma analz_empty [simp]: "analz{} = {}"
```

*<proof>*

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

**lemma** *analz\_Un*: "*analz*(*G*)  $\cup$  *analz*(*H*)  $\subseteq$  *analz*(*G*  $\cup$  *H*)"

*<proof>*

**lemma** *analz\_insert*: "*insert* *X* (*analz* *H*)  $\subseteq$  *analz*(*insert* *X* *H*)"

*<proof>*

### 1.3.2 Rewrite rules for pulling out atomic messages

**lemmas** *analz\_insert\_eq\_I* = *equalityI* [*OF* *subsetI* *analz\_insert*]

**lemma** *analz\_insert\_Agent* [*simp*]:

"*analz* (*insert* (*Agent* *agt*) *H*) = *insert* (*Agent* *agt*) (*analz* *H*)"

*<proof>*

**lemma** *analz\_insert\_Nonce* [*simp*]:

"*analz* (*insert* (*Nonce* *N*) *H*) = *insert* (*Nonce* *N*) (*analz* *H*)"

*<proof>*

**lemma** *analz\_insert\_Number* [*simp*]:

"*analz* (*insert* (*Number* *N*) *H*) = *insert* (*Number* *N*) (*analz* *H*)"

*<proof>*

**lemma** *analz\_insert\_Hash* [*simp*]:

"*analz* (*insert* (*Hash* *X*) *H*) = *insert* (*Hash* *X*) (*analz* *H*)"

*<proof>*

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

**lemma** *analz\_insert\_Key* [*simp*]:

"*K*  $\notin$  *keysFor* (*analz* *H*) ==>

*analz* (*insert* (*Key* *K*) *H*) = *insert* (*Key* *K*) (*analz* *H*)"

*<proof>*

**lemma** *analz\_insert\_MPair* [*simp*]:

"*analz* (*insert* {|*X*,*Y*|} *H*) =

*insert* {|*X*,*Y*|} (*analz* (*insert* *X* (*insert* *Y* *H*)))"

*<proof>*

Can pull out enCrypted message if the Key is not known

**lemma** *analz\_insert\_Crypt*:

"*Key* (*invKey* *K*)  $\notin$  *analz* *H*

==> *analz* (*insert* (*Crypt* *K* *X*) *H*) = *insert* (*Crypt* *K* *X*) (*analz* *H*)"

*<proof>*

**lemma** *lemma1*: "*Key* (*invKey* *K*)  $\in$  *analz* *H* ==>

*analz* (*insert* (*Crypt* *K* *X*) *H*)  $\subseteq$

*insert* (*Crypt* *K* *X*) (*analz* (*insert* *X* *H*))"

*<proof>*

**lemma** *lemma2*: "*Key* (*invKey* *K*)  $\in$  *analz* *H* ==>

```

      insert (Crypt K X) (analz (insert X H))  $\subseteq$ 
      analz (insert (Crypt K X) H)"
<proof>

```

```

lemma analz_insert_Decrypt:
  "Key (invKey K)  $\in$  analz H ==>
   analz (insert (Crypt K X) H) =
   insert (Crypt K X) (analz (insert X H))"
<proof>

```

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

```

lemma analz_Crypt_if [simp]:
  "analz (insert (Crypt K X) H) =
   (if (Key (invKey K)  $\in$  analz H)
    then insert (Crypt K X) (analz (insert X H))
    else insert (Crypt K X) (analz H))"
<proof>

```

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

```

lemma analz_insert_Crypt_subset:
  "analz (insert (Crypt K X) H)  $\subseteq$ 
   insert (Crypt K X) (analz (insert X H))"
<proof>

```

```

lemma analz_image_Key [simp]: "analz (Key'N) = Key'N"
<proof>

```

### 1.3.3 Idempotence and transitivity

```

lemma analz_analzD [dest!]: "X  $\in$  analz (analz H) ==> X  $\in$  analz H"
<proof>

```

```

lemma analz_idem [simp]: "analz (analz H) = analz H"
<proof>

```

```

lemma analz_subset_iff [simp]: "(analz G  $\subseteq$  analz H) = (G  $\subseteq$  analz H)"
<proof>

```

```

lemma analz_trans: "[| X  $\in$  analz G; G  $\subseteq$  analz H |] ==> X  $\in$  analz H"
<proof>

```

Cut; Lemma 2 of Lowe

```

lemma analz_cut: "[| Y  $\in$  analz (insert X H); X  $\in$  analz H |] ==> Y  $\in$  analz H"
<proof>

```

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

```

lemma analz_insert_eq: "X  $\in$  analz H ==> analz (insert X H) = analz H"

```

*<proof>*

A congruence rule for "analz"

```
lemma analz_subset_cong:
  "[| analz G ⊆ analz G'; analz H ⊆ analz H' |]
   ==> analz (G ∪ H) ⊆ analz (G' ∪ H')"
```

*<proof>*

```
lemma analz_cong:
  "[| analz G = analz G'; analz H = analz H' |]
   ==> analz (G ∪ H) = analz (G' ∪ H')"
```

*<proof>*

```
lemma analz_insert_cong:
  "analz H = analz H' ==> analz(insert X H) = analz(insert X H')"
```

*<proof>*

If there are no pairs or encryptions then analz does nothing

```
lemma analz_trivial:
  "[| ∀X Y. {|X,Y|} ∉ H; ∀X K. Crypt K X ∉ H |] ==> analz H = H"
```

*<proof>*

These two are obsolete (with a single Spy) but cost little to prove...

```
lemma analz_UN_analz_lemma:
  "X ∈ analz (⋃ i ∈ A. analz (H i)) ==> X ∈ analz (⋃ i ∈ A. H i)"
```

*<proof>*

```
lemma analz_UN_analz [simp]: "analz (⋃ i ∈ A. analz (H i)) = analz (⋃ i ∈ A.
H i)"
```

*<proof>*

## 1.4 Inductive relation "synth"

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

```
inductive_set
  synth :: "msg set => msg set"
  for H :: "msg set"
  where
    Inj [intro]: "X ∈ H ==> X ∈ synth H"
  | Agent [intro]: "Agent agt ∈ synth H"
  | Number [intro]: "Number n ∈ synth H"
  | Hash [intro]: "X ∈ synth H ==> Hash X ∈ synth H"
  | MPair [intro]: "[| X ∈ synth H; Y ∈ synth H |] ==> {|X,Y|} ∈ synth
H"
  | Crypt [intro]: "[| X ∈ synth H; Key(K) ∈ H |] ==> Crypt K X ∈ synth
H"
```

Monotonicity

```
lemma synth_mono: "G ⊆ H ==> synth(G) ⊆ synth(H)"
```

*<proof>*

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

```
inductive_cases Nonce_synth [elim!]: "Nonce n ∈ synth H"
inductive_cases Key_synth [elim!]: "Key K ∈ synth H"
inductive_cases Hash_synth [elim!]: "Hash X ∈ synth H"
inductive_cases MPair_synth [elim!]: "{|X,Y|} ∈ synth H"
inductive_cases Crypt_synth [elim!]: "Crypt K X ∈ synth H"
```

**lemma** *synth\_increasing*: " $H \subseteq \text{synth}(H)$ "

*<proof>*

#### 1.4.1 Unions

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

**lemma** *synth\_Un*: " $\text{synth}(G) \cup \text{synth}(H) \subseteq \text{synth}(G \cup H)$ "

*<proof>*

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

*<proof>*

#### 1.4.2 Idempotence and transitivity

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

*<proof>*

**lemma** *synth\_idem*: " $\text{synth} (\text{synth } H) = \text{synth } H$ "

*<proof>*

**lemma** *synth\_subset\_iff* [*simp*]: " $(\text{synth } G \subseteq \text{synth } H) = (G \subseteq \text{synth } H)$ "

*<proof>*

**lemma** *synth\_trans*: " $[| X \in \text{synth } G; G \subseteq \text{synth } H |] \implies X \in \text{synth } H$ "

*<proof>*

Cut; Lemma 2 of Lowe

**lemma** *synth\_cut*: " $[| Y \in \text{synth} (\text{insert } X H); X \in \text{synth } H |] \implies Y \in \text{synth } H$ "

*<proof>*

**lemma** *Agent\_synth* [*simp*]: " $\text{Agent } A \in \text{synth } H$ "

*<proof>*

**lemma** *Number\_synth* [*simp*]: " $\text{Number } n \in \text{synth } H$ "

*<proof>*

**lemma** *Nonce\_synth\_eq* [*simp*]: " $(\text{Nonce } N \in \text{synth } H) = (\text{Nonce } N \in H)$ "

*<proof>*

**lemma** *Key\_synth\_eq* [*simp*]: " $(\text{Key } K \in \text{synth } H) = (\text{Key } K \in H)$ "

*<proof>*

**lemma** *Crypt\_synth\_eq [simp]:*  
 "Key  $K \notin H \implies (\text{Crypt } K \ X \in \text{synth } H) = (\text{Crypt } K \ X \in H)"$   
*<proof>*

**lemma** *keysFor\_synth [simp]:*  
 "keysFor (synth H) = keysFor H  $\cup$  invKey '{K. Key K  $\in$  H}"  
*<proof>*

### 1.4.3 Combinations of parts, analz and synth

**lemma** *parts\_synth [simp]:* "parts (synth H) = parts H  $\cup$  synth H"  
*<proof>*

**lemma** *analz\_analz\_Un [simp]:* "analz (analz G  $\cup$  H) = analz (G  $\cup$  H)"  
*<proof>*

**lemma** *analz\_synth\_Un [simp]:* "analz (synth G  $\cup$  H) = analz (G  $\cup$  H)  $\cup$  synth G"  
*<proof>*

**lemma** *analz\_synth [simp]:* "analz (synth H) = analz H  $\cup$  synth H"  
*<proof>*

### 1.4.4 For reasoning about the Fake rule in traces

**lemma** *parts\_insert\_subset\_Un:* " $X \in G \implies \text{parts}(\text{insert } X \ H) \subseteq \text{parts } G \cup \text{parts } H"$   
*<proof>*

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

**lemma** *Fake\_parts\_insert:*  
 " $X \in \text{synth } (\text{analz } H) \implies$   
 $\text{parts } (\text{insert } X \ H) \subseteq \text{synth } (\text{analz } H) \cup \text{parts } H"$   
*<proof>*

**lemma** *Fake\_parts\_insert\_in\_Un:*  
 " $[Z \in \text{parts } (\text{insert } X \ H); \ X: \text{synth } (\text{analz } H)]$   
 $\implies Z \in \text{synth } (\text{analz } H) \cup \text{parts } H"$   
*<proof>*

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

**lemma** *Fake\_analz\_insert:*  
 " $X \in \text{synth } (\text{analz } G) \implies$   
 $\text{analz } (\text{insert } X \ H) \subseteq \text{synth } (\text{analz } G) \cup \text{analz } (G \cup H)"$   
*<proof>*

**lemma** *analz\_conj\_parts [simp]:*  
 " $(X \in \text{analz } H \ \& \ X \in \text{parts } H) = (X \in \text{analz } H)"$   
*<proof>*

```

lemma analz_disj_parts [simp]:
  "(X ∈ analz H | X ∈ parts H) = (X ∈ parts H)"
<proof>

```

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

```

lemma MPair_synth_analz [iff]:
  "({|X,Y|} ∈ synth (analz H)) =
   (X ∈ synth (analz H) & Y ∈ synth (analz H))"
<proof>

```

```

lemma Crypt_synth_analz:
  "[| Key K ∈ analz H; Key (invKey K) ∈ analz H |]
   ==> (Crypt K X ∈ synth (analz H)) = (X ∈ synth (analz H))"
<proof>

```

```

lemma Hash_synth_analz [simp]:
  "X ∉ synth (analz H)
   ==> (Hash{|X,Y|} ∈ synth (analz H)) = (Hash{|X,Y|} ∈ analz H)"
<proof>

```

## 1.5 HPair: a combination of Hash and MPair

### 1.5.1 Freeness

```

lemma Agent_neq_HPPair: "Agent A ~= Hash[X] Y"
<proof>

```

```

lemma Nonce_neq_HPPair: "Nonce N ~= Hash[X] Y"
<proof>

```

```

lemma Number_neq_HPPair: "Number N ~= Hash[X] Y"
<proof>

```

```

lemma Key_neq_HPPair: "Key K ~= Hash[X] Y"
<proof>

```

```

lemma Hash_neq_HPPair: "Hash Z ~= Hash[X] Y"
<proof>

```

```

lemma Crypt_neq_HPPair: "Crypt K X' ~= Hash[X] Y"
<proof>

```

```

lemmas HPair_neqs = Agent_neq_HPPair Nonce_neq_HPPair Number_neq_HPPair
          Key_neq_HPPair Hash_neq_HPPair Crypt_neq_HPPair

```

```

declare HPair_neqs [iff]
declare HPair_neqs [symmetric, iff]

```

```

lemma HPair_eq [iff]: "(Hash[X'] Y' = Hash[X] Y) = (X' = X & Y'=Y)"
<proof>

```

```

lemma MPair_eq_HPPair [iff]:

```

```
"({|X',Y'|} = Hash[X] Y) = (X' = Hash{|X,Y|} & Y'=Y)"
⟨proof⟩
```

```
lemma HPair_eq_MPair [iff]:
  "(Hash[X] Y = {|X',Y'|}) = (X' = Hash{|X,Y|} & Y'=Y)"
⟨proof⟩
```

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

```
lemma keysFor_insert_HPair [simp]: "keysFor (insert (Hash[X] Y) H) = keysFor
H"
⟨proof⟩
```

```
lemma parts_insert_HPair [simp]:
  "parts (insert (Hash[X] Y) H) =
  insert (Hash[X] Y) (insert (Hash{|X,Y|}) (parts (insert Y H)))"
⟨proof⟩
```

```
lemma analz_insert_HPair [simp]:
  "analz (insert (Hash[X] Y) H) =
  insert (Hash[X] Y) (insert (Hash{|X,Y|}) (analz (insert Y H)))"
⟨proof⟩
```

```
lemma HPair_synth_analz [simp]:
  "X ∉ synth (analz H)
  ==> (Hash[X] Y ∈ synth (analz H)) =
  (Hash {|X, Y|} ∈ analz H & Y ∈ synth (analz H))"
⟨proof⟩
```

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

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

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

```
lemmas pushKeys [standard] =
  insert_commute [of "Key K" "Agent C"]
  insert_commute [of "Key K" "Nonce N"]
  insert_commute [of "Key K" "Number N"]
  insert_commute [of "Key K" "Hash X"]
  insert_commute [of "Key K" "MPair X Y"]
  insert_commute [of "Key K" "Crypt X K"]
```

```
lemmas pushCrypts [standard] =
  insert_commute [of "Crypt X K" "Agent C"]
  insert_commute [of "Crypt X K" "Agent C"]
  insert_commute [of "Crypt X K" "Nonce N"]
  insert_commute [of "Crypt X K" "Number N"]
  insert_commute [of "Crypt X K" "Hash X'"]
  insert_commute [of "Crypt X K" "MPair X' Y"]
```

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

```
lemmas pushes = pushKeys pushCrypts
```



## 1.6 Tactics useful for many protocol proofs

*<ML>*

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

```
declare o_def [simp]
```

```
lemma Crypt_notin_image_Key [simp]: "Crypt K X ∉ Key ' A"
<proof>
```

```
lemma Hash_notin_image_Key [simp] : "Hash X ∉ Key ' A"
<proof>
```

```
lemma synth_analz_mono: "G ⊆ H ==> synth (analz(G)) ⊆ synth (analz(H))"
<proof>
```

```
lemma Fake_analz_eq [simp]:
  "X ∈ synth (analz H) ==> synth (analz (insert X H)) = synth (analz H)"
<proof>
```

Two generalizations of `analz_insert_eq`

```
lemma gen_analz_insert_eq [rule_format]:
  "X ∈ analz H ==> ALL G. H ⊆ G --> analz (insert X G) = analz G"
<proof>
```

```
lemma synth_analz_insert_eq [rule_format]:
  "X ∈ synth (analz H)
   ==> ALL G. H ⊆ G --> (Key K ∈ analz (insert X G)) = (Key K ∈ analz
G)"
<proof>
```

```
lemma Fake_parts_sing:
  "X ∈ synth (analz H) ==> parts{X} ⊆ synth (analz H) ∪ parts H"
<proof>
```

```
lemmas Fake_parts_sing_imp_Un = Fake_parts_sing [THEN [2] rev_subsetD]
```

*<ML>*

```
end
```

## 2 Theory of Events for Security Protocols

```
theory Event imports Message begin
```

```
consts
  initState :: "agent => msg set"
```

```
datatype
  event = Says agent agent msg
```

```

/ Gets agent      msg
/ Notes agent     msg

```

**consts**

```

bad      :: "agent set"
knows    :: "agent => event list => msg set"

```

The constant "spies" is retained for compatibility's sake

**abbreviation** (input)

```

spies    :: "event list => msg set" where
"spies == knows Spy"

```

Spy has access to his own key for spoof messages, but Server is secure

**specification** (bad)

```

Spy_in_bad      [iff]: "Spy ∈ bad"
Server_not_bad [iff]: "Server ∉ bad"
⟨proof⟩

```

**primrec**

```

knows_Nil:  "knows A [] = initState A"
knows_Cons:
  "knows A (ev # evs) =
    (if A = Spy then
      (case ev of
        Says A' B X => insert X (knows Spy evs)
      | Gets A' X => knows Spy evs
      | Notes A' X =>
          if A' ∈ bad then insert X (knows Spy evs) else knows Spy evs)
    else
      (case ev of
        Says A' B X =>
          if A'=A then insert X (knows A evs) else knows A evs
      | Gets A' X =>
          if A'=A then insert X (knows A evs) else knows A evs
      | Notes A' X =>
          if A'=A then insert X (knows A evs) else knows A evs)))"

```

**consts**

```

used :: "event list => msg set"

```

**primrec**

```

used_Nil:  "used [] = (UN B. parts (initState B))"
used_Cons: "used (ev # evs) =
  (case ev of
    Says A B X => parts {X} ∪ used evs
  | Gets A X   => used evs
  | Notes A X  => parts {X} ∪ used evs)"

```

— The case for *Gets* seems anomalous, but *Gets* always follows *Says* in real protocols. Seems difficult to change. See *Gets\_correct* in theory *Guard/Extensions.thy*.

**lemma** *Notes\_imp\_used* [rule\_format]: "Notes A X  $\in$  set evs  $\rightarrow$  X  $\in$  used evs"  
 <proof>

**lemma** *Says\_imp\_used* [rule\_format]: "Says A B X  $\in$  set evs  $\rightarrow$  X  $\in$  used evs"  
 <proof>

## 2.1 Function knows

**lemmas** *parts\_insert\_knows\_A* = parts\_insert [of \_ "knows A evs", standard]

**lemma** *knows\_Spy\_Says* [simp]:  
 "knows Spy (Says A B X # evs) = insert X (knows Spy evs)"  
 <proof>

Letting the Spy see "bad" agents' notes avoids redundant case-splits on whether  
 A = Spy and whether A  $\in$  bad

**lemma** *knows\_Spy\_Notes* [simp]:  
 "knows Spy (Notes A X # evs) =  
 (if A:bad then insert X (knows Spy evs) else knows Spy evs)"  
 <proof>

**lemma** *knows\_Spy\_Gets* [simp]: "knows Spy (Gets A X # evs) = knows Spy evs"  
 <proof>

**lemma** *knows\_Spy\_subset\_knows\_Spy\_Says*:  
 "knows Spy evs  $\subseteq$  knows Spy (Says A B X # evs)"  
 <proof>

**lemma** *knows\_Spy\_subset\_knows\_Spy\_Notes*:  
 "knows Spy evs  $\subseteq$  knows Spy (Notes A X # evs)"  
 <proof>

**lemma** *knows\_Spy\_subset\_knows\_Spy\_Gets*:  
 "knows Spy evs  $\subseteq$  knows Spy (Gets A X # evs)"  
 <proof>

Spy sees what is sent on the traffic

**lemma** *Says\_imp\_knows\_Spy* [rule\_format]:  
 "Says A B X  $\in$  set evs  $\rightarrow$  X  $\in$  knows Spy evs"  
 <proof>

**lemma** *Notes\_imp\_knows\_Spy* [rule\_format]:  
 "Notes A X  $\in$  set evs  $\rightarrow$  A: bad  $\rightarrow$  X  $\in$  knows Spy evs"  
 <proof>

Elimination rules: derive contradictions from old Says events containing items  
 known to be fresh

**lemmas** *knows\_Spy\_partsEs* =  
 Says\_imp\_knows\_Spy [THEN parts.Inj, THEN revcut\_rl, standard]  
 parts.Body [THEN revcut\_rl, standard]

**lemmas** *Says\_imp\_analz\_Spy* = Says\_imp\_knows\_Spy [THEN analz.Inj]

Compatibility for the old "spies" function

```

lemmas spies_partsEs = knows_Spy_partsEs
lemmas Says_imp_spies = Says_imp_knows_Spy
lemmas parts_insert_spies = parts_insert_knows_A [of _ Spy]

```

## 2.2 Knowledge of Agents

```

lemma knows_Says: "knows A (Says A B X # evs) = insert X (knows A evs)"
<proof>

```

```

lemma knows_Notes: "knows A (Notes A X # evs) = insert X (knows A evs)"
<proof>

```

```

lemma knows_Gets:
  "A ≠ Spy --> knows A (Gets A X # evs) = insert X (knows A evs)"
<proof>

```

```

lemma knows_subset_knows_Says: "knows A evs ⊆ knows A (Says A' B X # evs)"
<proof>

```

```

lemma knows_subset_knows_Notes: "knows A evs ⊆ knows A (Notes A' X # evs)"
<proof>

```

```

lemma knows_subset_knows_Gets: "knows A evs ⊆ knows A (Gets A' X # evs)"
<proof>

```

Agents know what they say

```

lemma Says_imp_knows [rule_format]: "Says A B X ∈ set evs --> X ∈ knows
A evs"
<proof>

```

Agents know what they note

```

lemma Notes_imp_knows [rule_format]: "Notes A X ∈ set evs --> X ∈ knows
A evs"
<proof>

```

Agents know what they receive

```

lemma Gets_imp_knows_agents [rule_format]:
  "A ≠ Spy --> Gets A X ∈ set evs --> X ∈ knows A evs"
<proof>

```

What agents DIFFERENT FROM Spy know was either said, or noted, or got, or known initially

```

lemma knows_imp_Says_Gets_Notes_initState [rule_format]:
  "[| X ∈ knows A evs; A ≠ Spy |] ==> EX B.
    Says A B X ∈ set evs | Gets A X ∈ set evs | Notes A X ∈ set evs | X ∈
    initState A"
<proof>

```

What the Spy knows – for the time being – was either said or noted, or known initially

```

lemma knows_Spy_imp_Says_Notes_initState [rule_format]:

```

```

    "[| X ∈ knows Spy evs |] ==> EX A B.
    Says A B X ∈ set evs | Notes A X ∈ set evs | X ∈ initState Spy"
  <proof>

```

```

lemma parts_knows_Spy_subset_used: "parts (knows Spy evs) ⊆ used evs"
  <proof>

```

```

lemmas usedI = parts_knows_Spy_subset_used [THEN subsetD, intro]

```

```

lemma initState_into_used: "X ∈ parts (initState B) ==> X ∈ used evs"
  <proof>

```

```

lemma used_Says [simp]: "used (Says A B X # evs) = parts{X} ∪ used evs"
  <proof>

```

```

lemma used_Notes [simp]: "used (Notes A X # evs) = parts{X} ∪ used evs"
  <proof>

```

```

lemma used_Gets [simp]: "used (Gets A X # evs) = used evs"
  <proof>

```

```

lemma used_nil_subset: "used [] ⊆ used evs"
  <proof>

```

NOTE REMOVAL—laws above are cleaner, as they don't involve "case"

```

declare knows_Cons [simp del]
        used_Nil [simp del] used_Cons [simp del]

```

For proving theorems of the form  $X \notin \text{analz}(\text{knows Spy evs}) \longrightarrow P$  New events added by induction to "evs" are discarded. Provided this information isn't needed, the proof will be much shorter, since it will omit complicated reasoning about *analz*.

```

lemmas analz_mono_contra =
  knows_Spy_subset_knows_Spy_Says [THEN analz_mono, THEN contra_subsetD]
  knows_Spy_subset_knows_Spy_Notes [THEN analz_mono, THEN contra_subsetD]
  knows_Spy_subset_knows_Spy_Gets [THEN analz_mono, THEN contra_subsetD]

```

```

lemma knows_subset_knows_Cons: "knows A evs ⊆ knows A (e # evs)"
  <proof>

```

```

lemma initState_subset_knows: "initState A ⊆ knows A evs"
  <proof>

```

For proving *new\_keys\_not\_used*

```

lemma keysFor_parts_insert:
  "[| K ∈ keysFor (parts (insert X G)); X ∈ synth (analz H) |]
  ==> K ∈ keysFor (parts (G ∪ H)) | Key (invKey K) ∈ parts H"
  <proof>

```

```

lemmas analz_impI = impI [where P = "Y ∉ analz (knows Spy evs)", standard]

```

<ML>

### 2.2.1 Useful for case analysis on whether a hash is a spoof or not

```
lemmas syan_impI = impI [where P = "Y  $\notin$  synth (analz (knows Spy evs))",
standard]
```

```
 $\langle ML \rangle$ 
```

```
end
```

```
theory Public imports Event begin
```

```
lemma invKey_K: "K  $\in$  symKeys ==> invKey K = K"
 $\langle proof \rangle$ 
```

## 2.3 Asymmetric Keys

```
datatype keymode = Signature | Encryption
```

```
consts
```

```
  publicKey :: "[keymode, agent] => key"
```

```
abbreviation
```

```
  pubEK :: "agent => key" where
    "pubEK == publicKey Encryption"
```

```
abbreviation
```

```
  pubSK :: "agent => key" where
    "pubSK == publicKey Signature"
```

```
abbreviation
```

```
  privateKey :: "[keymode, agent] => key" where
    "privateKey b A == invKey (publicKey b A)"
```

```
abbreviation
```

```
  priEK :: "agent => key" where
    "priEK A == privateKey Encryption A"
```

```
abbreviation
```

```
  priSK :: "agent => key" where
    "priSK A == privateKey Signature A"
```

These abbreviations give backward compatibility. They represent the simple situation where the signature and encryption keys are the same.

```
abbreviation
```

```
  pubK :: "agent => key" where
    "pubK A == pubEK A"
```

```
abbreviation
```

```
  priK :: "agent => key" where
    "priK A == invKey (pubEK A)"
```

By freeness of agents, no two agents have the same key. Since *True*  $\neq$  *False*,

no agent has identical signing and encryption keys

**specification** ( $\text{publicKey}$ )

```
injective_publicKey:
  "publicKey b A = publicKey c A' ==> b=c & A=A'"
  <proof>
```

**axioms**

```
privateKey_neq_publicKey [iff]: "privateKey b A ≠ publicKey c A'"
```

```
lemmas publicKey_neq_privateKey = privateKey_neq_publicKey [THEN not_sym]
declare publicKey_neq_privateKey [iff]
```

## 2.4 Basic properties of $\text{pubK}$ and $\lambda A. \text{invKey } (\text{pubK } A)$

```
lemma publicKey_inject [iff]: "(publicKey b A = publicKey c A') = (b=c &
A=A')"
  <proof>
```

```
lemma not_symKeys_pubK [iff]: "publicKey b A ∉ symKeys"
  <proof>
```

```
lemma not_symKeys_priK [iff]: "privateKey b A ∉ symKeys"
  <proof>
```

```
lemma symKey_neq_priEK: "K ∈ symKeys ==> K ≠ priEK A"
  <proof>
```

```
lemma symKeys_neq_imp_neq: "(K ∈ symKeys) ≠ (K' ∈ symKeys) ==> K ≠ K'"
  <proof>
```

```
lemma symKeys_invKey_iff [iff]: "(invKey K ∈ symKeys) = (K ∈ symKeys)"
  <proof>
```

```
lemma analz_symKeys_Decrypt:
  "[| Crypt K X ∈ analz H; K ∈ symKeys; Key K ∈ analz H |]
   ==> X ∈ analz H"
  <proof>
```

## 2.5 "Image" equations that hold for injective functions

```
lemma invKey_image_eq [simp]: "(invKey x ∈ invKey 'A) = (x ∈ A)"
  <proof>
```

```
lemma publicKey_image_eq [simp]:
  "(publicKey b x ∈ publicKey c ' AA) = (b=c & x ∈ AA)"
  <proof>
```

```
lemma privateKey_notin_image_publicKey [simp]: "privateKey b x ∉ publicKey
c ' AA"
  <proof>
```

```

lemma privateKey_image_eq [simp]:
  "(privateKey b A ∈ invKey ' publicKey c ' AS) = (b=c & A∈AS)"
  <proof>

```

```

lemma publicKey_notin_image_privateKey [simp]: "publicKey b A ∉ invKey '
publicKey c ' AS"
  <proof>

```

## 2.6 Symmetric Keys

For some protocols, it is convenient to equip agents with symmetric as well as asymmetric keys. The theory *Shared* assumes that all keys are symmetric.

```

consts
  shrK      :: "agent => key"      — long-term shared keys

```

```

specification (shrK)
  inj_shrK: "inj shrK"
  — No two agents have the same long-term key
  <proof>

```

```

axioms
  sym_shrK [iff]: "shrK X ∈ symKeys" — All shared keys are symmetric

```

Injectiveness: Agents' long-term keys are distinct.

```

lemmas shrK_injective = inj_shrK [THEN inj_eq]
declare shrK_injective [iff]

```

```

lemma invKey_shrK [simp]: "invKey (shrK A) = shrK A"
  <proof>

```

```

lemma analz_shrK_Decrypt:
  "[| Crypt (shrK A) X ∈ analz H; Key(shrK A) ∈ analz H |] ==> X ∈ analz
H"
  <proof>

```

```

lemma analz_Decrypt':
  "[| Crypt K X ∈ analz H; K ∈ symKeys; Key K ∈ analz H |] ==> X ∈ analz
H"
  <proof>

```

```

lemma priK_neq_shrK [iff]: "shrK A ≠ privateKey b C"
  <proof>

```

```

lemmas shrK_neq_priK = priK_neq_shrK [THEN not_sym]
declare shrK_neq_priK [simp]

```

```

lemma pubK_neq_shrK [iff]: "shrK A ≠ publicKey b C"
  <proof>

```

```

lemmas shrK_neq_pubK = pubK_neq_shrK [THEN not_sym]
declare shrK_neq_pubK [simp]

```

```

lemma priEK_noteq_shrK [simp]: "priEK A ≠ shrK B"

```



*<proof>*

**lemma** *publicKey\_notin\_image\_shrK* [simp]: "publicKey b x  $\notin$  shrK ' AA"  
*<proof>*

**lemma** *privateKey\_notin\_image\_shrK* [simp]: "privateKey b x  $\notin$  shrK ' AA"  
*<proof>*

**lemma** *shrK\_notin\_image\_publicKey* [simp]: "shrK x  $\notin$  publicKey b ' AA"  
*<proof>*

**lemma** *shrK\_notin\_image\_privateKey* [simp]: "shrK x  $\notin$  invKey ' publicKey b  
 ' AA"  
*<proof>*

**lemma** *shrK\_image\_eq* [simp]: "(shrK x  $\in$  shrK ' AA) = (x  $\in$  AA)"  
*<proof>*

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

**declare** *invKey\_K* [simp]

## 2.7 Initial States of Agents

Note: for all practical purposes, all that matters is the initial knowledge of the Spy. All other agents are automata, merely following the protocol.

**primrec**

```

initState_Server:
  "initState Server      =
    {Key (priEK Server), Key (priSK Server)}  $\cup$ 
    (Key ' range pubEK)  $\cup$  (Key ' range pubSK)  $\cup$  (Key ' range shrK)"

initState_Friend:
  "initState (Friend i) =
    {Key (priEK(Friend i)), Key (priSK(Friend i)), Key (shrK(Friend i))}
 $\cup$ 
    (Key ' range pubEK)  $\cup$  (Key ' range pubSK)"

initState_Spy:
  "initState Spy          =
    (Key ' invKey ' pubEK ' bad)  $\cup$  (Key ' invKey ' pubSK ' bad)  $\cup$ 
    (Key ' shrK ' bad)  $\cup$ 
    (Key ' range pubEK)  $\cup$  (Key ' range pubSK)"

```

These lemmas allow reasoning about *used evs* rather than *knows Spy evs*, which is useful when there are private Notes. Because they depend upon the definition of *initState*, they cannot be moved up.

**lemma** *used\_parts\_subset\_parts* [rule\_format]:  
 " $\forall X \in \text{used evs. parts } \{X\} \subseteq \text{used evs}$ "  
*<proof>*

**lemma** *MPair\_used\_D*: " $\{|X, Y|\} \in \text{used } H \implies X \in \text{used } H \ \& \ Y \in \text{used } H$ "  
*<proof>*

There was a similar theorem in Event.thy, so perhaps this one can be moved up if proved directly by induction.

```
lemma MPair_used [elim!]:
  "[| {X,Y} ∈ used H;
    [| X ∈ used H; Y ∈ used H |] ==> P |]
  ==> P"
<proof>
```

Rewrites should not refer to `initState (Friend i)` because that expression is not in normal form.

```
lemma keysFor_parts_initState [simp]: "keysFor (parts (initState C)) = {}"
<proof>
```

```
lemma Crypt_notin_initState: "Crypt K X ∉ parts (initState B)"
<proof>
```

```
lemma Crypt_notin_used_empty [simp]: "Crypt K X ∉ used []"
<proof>
```

```
lemma shrK_in_initState [iff]: "Key (shrK A) ∈ initState A"
<proof>
```

```
lemma shrK_in_knows [iff]: "Key (shrK A) ∈ knows A evs"
<proof>
```

```
lemma shrK_in_used [iff]: "Key (shrK A) ∈ used evs"
<proof>
```

```
lemma Key_not_used [simp]: "Key K ∉ used evs ==> K ∉ range shrK"
<proof>
```

```
lemma shrK_neq: "Key K ∉ used evs ==> shrK B ≠ K"
<proof>
```

```
lemmas neq_shrK = shrK_neq [THEN not_sym]
declare neq_shrK [simp]
```

## 2.8 Function `knows Spy`

```
lemma not_SignatureE [elim!]: "b ≠ Signature ==> b = Encryption"
<proof>
```

Agents see their own private keys!

```
lemma priK_in_initState [iff]: "Key (privateKey b A) ∈ initState A"
<proof>
```

Agents see all public keys!

**lemma** *publicKey\_in\_initState* [iff]: "Key (publicKey b A) ∈ initState B"  
 ⟨proof⟩

All public keys are visible

**lemma** *spies\_pubK* [iff]: "Key (publicKey b A) ∈ spies evs"  
 ⟨proof⟩

**lemmas** *analz\_spies\_pubK* = *spies\_pubK* [THEN *analz.Inj*]  
**declare** *analz\_spies\_pubK* [iff]

Spy sees private keys of bad agents!

**lemma** *Spy\_spies\_bad\_privateKey* [intro!]:  
 "A ∈ bad ==> Key (privateKey b A) ∈ spies evs"  
 ⟨proof⟩

Spy sees long-term shared keys of bad agents!

**lemma** *Spy\_spies\_bad\_shrK* [intro!]:  
 "A ∈ bad ==> Key (shrK A) ∈ spies evs"  
 ⟨proof⟩

**lemma** *publicKey\_into\_used* [iff] : "Key (publicKey b A) ∈ used evs"  
 ⟨proof⟩

**lemma** *privateKey\_into\_used* [iff]: "Key (privateKey b A) ∈ used evs"  
 ⟨proof⟩

**lemma** *Crypt\_Spy\_analz\_bad*:  
 "[| Crypt (shrK A) X ∈ analz (knows Spy evs); A ∈ bad |]  
 ==> X ∈ analz (knows Spy evs)"  
 ⟨proof⟩

## 2.9 Fresh Nonces

**lemma** *Nonce\_notin\_initState* [iff]: "Nonce N ∉ parts (initState B)"  
 ⟨proof⟩

**lemma** *Nonce\_notin\_used\_empty* [simp]: "Nonce N ∉ used []"  
 ⟨proof⟩

## 2.10 Supply fresh nonces for possibility theorems

In any trace, there is an upper bound N on the greatest nonce in use

**lemma** *Nonce\_supply\_lemma*: "EX N. ALL n. N ≤ n --> Nonce n ∉ used evs"  
 ⟨proof⟩

**lemma** *Nonce\_supply1*: "EX N. Nonce N ∉ used evs"  
 ⟨proof⟩

**lemma** *Nonce\_supply*: "Nonce (@ N. Nonce N ∉ used evs) ∉ used evs"  
 ⟨proof⟩

## 2.11 Specialized Rewriting for Theorems About *analz* and Image

**lemma** *insert\_Key\_singleton*: "insert (Key K) H = Key ' {K} Un H"  
 <proof>

**lemma** *insert\_Key\_image*: "insert (Key K) (Key'KK  $\cup$  C) = Key ' (insert K KK)  $\cup$  C"  
 <proof>

**lemma** *Crypt\_imp\_keysFor* : "[|Crypt K X  $\in$  H; K  $\in$  symKeys|] ==> K  $\in$  keysFor H"  
 <proof>

Lemma for the trivial direction of the if-and-only-if of the Session Key Compromise Theorem

**lemma** *analz\_image\_freshK\_lemma*:  
 "(Key K  $\in$  analz (Key'nE  $\cup$  H)) --> (K  $\in$  nE | Key K  $\in$  analz H) ==>  
 (Key K  $\in$  analz (Key'nE  $\cup$  H)) = (K  $\in$  nE | Key K  $\in$  analz H)"  
 <proof>

**lemmas** *analz\_image\_freshK\_simps* =  
*simp\_thms mem\_simps* — these two allow its use with only:  
*disj\_comms*  
*image\_insert [THEN sym] image\_Un [THEN sym] empty\_subsetI insert\_subset*  
*analz\_insert\_eq Un\_upper2 [THEN analz\_mono, THEN subsetD]*  
*insert\_Key\_singleton*  
*Key\_not\_used insert\_Key\_image Un\_assoc [THEN sym]*

<ML>

## 2.12 Specialized Methods for Possibility Theorems

<ML>

end

## 3 Needham-Schroeder Shared-Key Protocol and the Issues Property

**theory** *NS\_Shared* imports *Public* begin

From page 247 of Burrows, Abadi and Needham (1989). A Logic of Authentication. Proc. Royal Soc. 426

**constdefs**

*Issues* :: "[agent, agent, msg, event list] => bool"  
 ("\_ Issues \_ with \_ on \_")  
 "A Issues B with X on evs ==

```

 $\exists Y. \text{Says } A \ B \ Y \in \text{set evs} \ \& \ X \in \text{parts } \{Y\} \ \&$ 
 $X \notin \text{parts } (\text{spies } (\text{takeWhile } (\% z. z \neq \text{Says } A \ B \ Y) \ (\text{rev evs})))$ "

inductive_set ns_shared :: "event list set"
where

  Nil: "[ ] ∈ ns_shared"

  / Fake: "[[evsf ∈ ns_shared; X ∈ synth (analz (spies evsf))]]
    ⇒ Says Spy B X # evsf ∈ ns_shared"

  / NS1: "[[evs1 ∈ ns_shared; Nonce NA ∉ used evs1]
    ⇒ Says A Server {Agent A, Agent B, Nonce NA} # evs1 ∈ ns_shared"

  / NS2: "[[evs2 ∈ ns_shared; Key KAB ∉ used evs2; KAB ∈ symKeys;
    Says A' Server {Agent A, Agent B, Nonce NA} ∈ set evs2]
    ⇒ Says Server A
      (Crypt (shrK A)
        {Nonce NA, Agent B, Key KAB,
          (Crypt (shrK B) {Key KAB, Agent A})})
      # evs2 ∈ ns_shared"

  / NS3: "[[evs3 ∈ ns_shared; A ≠ Server;
    Says S A (Crypt (shrK A) {Nonce NA, Agent B, Key K, X}) ∈ set evs3;
    Says A Server {Agent A, Agent B, Nonce NA} ∈ set evs3]
    ⇒ Says A B X # evs3 ∈ ns_shared"

  / NS4: "[[evs4 ∈ ns_shared; Nonce NB ∉ used evs4; K ∈ symKeys;
    Says A' B (Crypt (shrK B) {Key K, Agent A}) ∈ set evs4]
    ⇒ Says B A (Crypt K (Nonce NB)) # evs4 ∈ ns_shared"

  / NS5: "[[evs5 ∈ ns_shared; K ∈ symKeys;
    Says B' A (Crypt K (Nonce NB)) ∈ set evs5;
    Says S A (Crypt (shrK A) {Nonce NA, Agent B, Key K, X})
      ∈ set evs5]
    ⇒ Says A B (Crypt K {Nonce NB, Nonce NB}) # evs5 ∈ ns_shared"

  / Ops: "[[evso ∈ ns_shared; Says B A (Crypt K (Nonce NB)) ∈ set evso;
    Says Server A (Crypt (shrK A) {Nonce NA, Agent B, Key K, X})
      ∈ set evso]
    ⇒ Notes Spy {Nonce NA, Nonce NB, Key K} # evso ∈ ns_shared"

declare Says_imp_knows_Spy [THEN parts.Inj, dest]
declare parts.Body [dest]
declare Fake_parts_insert_in_Un [dest]
declare analz_into_parts [dest]

```

**declare** *image\_eq\_UN* [simp]

A "possibility property": there are traces that reach the end

**lemma** "[| A ≠ Server; Key K ∉ used []; K ∈ symKeys |]  
 ==> ∃ N. ∃ evs ∈ ns\_shared.  
       Says A B (Crypt K {Nonce N, Nonce N}) ∈ set evs"  
 <proof>

### 3.1 Inductive proofs about *ns\_shared*

#### 3.1.1 Forwarding lemmas, to aid simplification

For reasoning about the encrypted portion of message NS3

**lemma** *NS3\_msg\_in\_parts\_spies*:  
 "Says S A (Crypt KA {N, B, K, X}) ∈ set evs ==> X ∈ parts (spies evs)"  
 <proof>

For reasoning about the Oops message

**lemma** *Oops\_parts\_spies*:  
 "Says Server A (Crypt (shrK A) {NA, B, K, X}) ∈ set evs  
       ==> K ∈ parts (spies evs)"  
 <proof>

Theorems of the form  $X \notin \text{parts } (\text{knows } \text{Spy } \text{evs})$  imply that NOBODY sends messages containing  $X$

Spy never sees another agent's shared key! (unless it's bad at start)

**lemma** *Spy\_see\_shrK* [simp]:  
 "evs ∈ ns\_shared ==> (Key (shrK A) ∈ parts (spies evs)) = (A ∈ bad)"  
 <proof>

**lemma** *Spy\_analz\_shrK* [simp]:  
 "evs ∈ ns\_shared ==> (Key (shrK A) ∈ analz (spies evs)) = (A ∈ bad)"  
 <proof>

Nobody can have used non-existent keys!

**lemma** *new\_keys\_not\_used* [simp]:  
 "[| Key K ∉ used evs; K ∈ symKeys; evs ∈ ns\_shared |]  
 ==> K ∉ keysFor (parts (spies evs))"  
 <proof>

#### 3.1.2 Lemmas concerning the form of items passed in messages

Describes the form of  $K$ ,  $X$  and  $K'$  when the Server sends this message.

**lemma** *Says\_Server\_message\_form*:  
 "[| Says Server A (Crypt K' {N, Agent B, Key K, X}) ∈ set evs;  
   evs ∈ ns\_shared |]  
 ==> K ∉ range shrK ∧  
       X = (Crypt (shrK B) {Key K, Agent A}) ∧  
       K' = shrK A"  
 <proof>

If the encrypted message appears then it originated with the Server

**lemma** *A\_trusts\_NS2*:

```
"[[Crypt (shrK A) {NA, Agent B, Key K, X} ∈ parts (spies evs);
  A ∉ bad; evs ∈ ns_shared]]
  ⇒ Says Server A (Crypt (shrK A) {NA, Agent B, Key K, X}) ∈ set evs"
⟨proof⟩
```

**lemma** *cert\_A\_form*:

```
"[[Crypt (shrK A) {NA, Agent B, Key K, X} ∈ parts (spies evs);
  A ∉ bad; evs ∈ ns_shared]]
  ⇒ K ∉ range shrK ∧ X = (Crypt (shrK B) {Key K, Agent A})"
⟨proof⟩
```

EITHER describes the form of X when the following message is sent, OR reduces it to the Fake case. Use *Says\_Server\_message\_form* if applicable.

**lemma** *Says\_S\_message\_form*:

```
"[[Says S A (Crypt (shrK A) {Nonce NA, Agent B, Key K, X}) ∈ set evs;
  evs ∈ ns_shared]]
  ⇒ (K ∉ range shrK ∧ X = (Crypt (shrK B) {Key K, Agent A}))
     ∨ X ∈ analz (spies evs)"
⟨proof⟩
```

NOT useful in this form, but it says that session keys are not used to encrypt messages containing other keys, in the actual protocol. We require that agents should behave like this subsequently also.

```
lemma "[[evs ∈ ns_shared; Kab ∉ range shrK]] ⇒
  (Crypt KAB X) ∈ parts (spies evs) ∧
  Key K ∈ parts {X} ⟶ Key K ∈ parts (spies evs)"
⟨proof⟩
```

### 3.1.3 Session keys are not used to encrypt other session keys

The equality makes the induction hypothesis easier to apply

**lemma** *analz\_image\_freshK* [rule\_format]:

```
"evs ∈ ns_shared ⇒
  ∀ K KK. KK ⊆ - (range shrK) ⟶
    (Key K ∈ analz (Key KK ∪ (spies evs))) =
    (K ∈ KK ∨ Key K ∈ analz (spies evs))"
⟨proof⟩
```

**lemma** *analz\_insert\_freshK*:

```
"[[evs ∈ ns_shared; KAB ∉ range shrK]] ⇒
  (Key K ∈ analz (insert (Key KAB) (spies evs))) =
  (K = KAB ∨ Key K ∈ analz (spies evs))"
⟨proof⟩
```

### 3.1.4 The session key K uniquely identifies the message

In messages of this form, the session key uniquely identifies the rest

**lemma** *unique\_session\_keys*:

```
"[[Says Server A (Crypt (shrK A) {NA, Agent B, Key K, X}) ∈ set evs;
  Says Server A' (Crypt (shrK A') {NA', Agent B', Key K, X'}) ∈ set
  evs;
```

$evs \in ns\_shared \implies A=A' \wedge NA=NA' \wedge B=B' \wedge X = X'$

*<proof>*

### 3.1.5 Crucial secrecy property: Spy doesn't see the keys sent in NS2

Beware of *[rule\_format]* and the universal quantifier!

**lemma** *secrecy\_lemma*:

$\llbracket \text{Says Server } A \text{ (Crypt (shrK } A) \{NA, \text{Agent } B, \text{Key } K, \text{Crypt (shrK } B) \{Key } K, \text{Agent } A\}\})} \in \text{set } evs;$

$A \notin \text{bad}; B \notin \text{bad}; evs \in ns\_shared \rrbracket$

$\implies (\forall NB. \text{Notes Spy } \{NA, NB, \text{Key } K\} \notin \text{set } evs) \longrightarrow$

$\text{Key } K \notin \text{analz (spies } evs)$

*<proof>*

Final version: Server's message in the most abstract form

**lemma** *Spy\_not\_see\_encrypted\_key*:

$\llbracket \text{Says Server } A \text{ (Crypt } K' \{NA, \text{Agent } B, \text{Key } K, X\})} \in \text{set } evs;$

$\forall NB. \text{Notes Spy } \{NA, NB, \text{Key } K\} \notin \text{set } evs;$

$A \notin \text{bad}; B \notin \text{bad}; evs \in ns\_shared \rrbracket$

$\implies \text{Key } K \notin \text{analz (spies } evs)$

*<proof>*

## 3.2 Guarantees available at various stages of protocol

If the encrypted message appears then it originated with the Server

**lemma** *B\_trusts\_NS3*:

$\llbracket \text{Crypt (shrK } B) \{Key } K, \text{Agent } A\} \in \text{parts (spies } evs);$

$B \notin \text{bad}; evs \in ns\_shared \rrbracket$

$\implies \exists NA. \text{Says Server } A$

$(\text{Crypt (shrK } A) \{NA, \text{Agent } B, \text{Key } K,$

$\text{Crypt (shrK } B) \{Key } K, \text{Agent } A\}\})$

$\in \text{set } evs$

*<proof>*

**lemma** *A\_trusts\_NS4\_lemma [rule\_format]*:

$evs \in ns\_shared \implies$

$\text{Key } K \notin \text{analz (spies } evs) \longrightarrow$

$\text{Says Server } A \text{ (Crypt (shrK } A) \{NA, \text{Agent } B, \text{Key } K, X\})} \in \text{set } evs \longrightarrow$

$\text{Crypt } K \text{ (Nonce } NB) \in \text{parts (spies } evs) \longrightarrow$

$\text{Says } B \text{ } A \text{ (Crypt } K \text{ (Nonce } NB))} \in \text{set } evs$

*<proof>*

This version no longer assumes that K is secure

**lemma** *A\_trusts\_NS4*:

$\llbracket \text{Crypt } K \text{ (Nonce } NB) \in \text{parts (spies } evs);$

$\text{Crypt (shrK } A) \{NA, \text{Agent } B, \text{Key } K, X\} \in \text{parts (spies } evs);$

$\forall NB. \text{Notes Spy } \{NA, NB, \text{Key } K\} \notin \text{set } evs;$

$A \notin \text{bad}; B \notin \text{bad}; evs \in ns\_shared \rrbracket$

$\implies \text{Says } B \text{ } A \text{ (Crypt } K \text{ (Nonce } NB))} \in \text{set } evs$

*<proof>*



If the session key has been used in NS4 then somebody has forwarded component X in some instance of NS4. Perhaps an interesting property, but not needed (after all) for the proofs below.

**theorem** *NS4\_implies\_NS3* [rule\_format]:

```
"evs ∈ ns_shared ⇒
  Key K ∉ analz (spies evs) →
  Says Server A (Crypt (shrK A) {NA, Agent B, Key K, X}) ∈ set evs →
  Crypt K (Nonce NB) ∈ parts (spies evs) →
  (∃ A'. Says A' B X ∈ set evs)"
```

⟨proof⟩

**lemma** *B\_trusts\_NS5\_lemma* [rule\_format]:

```
"[B ∉ bad; evs ∈ ns_shared] ⇒
  Key K ∉ analz (spies evs) →
  Says Server A
    (Crypt (shrK A) {NA, Agent B, Key K,
                     Crypt (shrK B) {Key K, Agent A}}) ∈ set evs →
  Crypt K {Nonce NB, Nonce NB} ∈ parts (spies evs) →
  Says A B (Crypt K {Nonce NB, Nonce NB}) ∈ set evs"
```

⟨proof⟩

Very strong Oops condition reveals protocol's weakness

**lemma** *B\_trusts\_NS5*:

```
"[Crypt K {Nonce NB, Nonce NB} ∈ parts (spies evs);
  Crypt (shrK B) {Key K, Agent A} ∈ parts (spies evs);
  ∀ NA NB. Notes Spy {NA, NB, Key K} ∉ set evs;
  A ∉ bad; B ∉ bad; evs ∈ ns_shared]
⇒ Says A B (Crypt K {Nonce NB, Nonce NB}) ∈ set evs"
```

⟨proof⟩

Unaltered so far wrt original version

### 3.3 Lemmas for reasoning about predicate "Issues"

**lemma** *spies\_Says\_rev*: "spies (evs @ [Says A B X]) = insert X (spies evs)"

⟨proof⟩

**lemma** *spies\_Gets\_rev*: "spies (evs @ [Gets A X]) = spies evs"

⟨proof⟩

**lemma** *spies\_Notes\_rev*: "spies (evs @ [Notes A X]) =  
(if A:bad then insert X (spies evs) else spies evs)"

⟨proof⟩

**lemma** *spies\_evs\_rev*: "spies evs = spies (rev evs)"

⟨proof⟩

**lemmas** *parts\_spies\_evs\_revD2* = spies\_evs\_rev [THEN equalityD2, THEN parts\_mono]

**lemma** *spies\_takeWhile*: "spies (takeWhile P evs) ≤ spies evs"

⟨proof⟩

**lemmas** *parts\_spies\_takeWhile\_mono* = spies\_takeWhile [THEN parts\_mono]

### 3.4 Guarantees of non-injective agreement on the session key, and of key distribution. They also express forms of freshness of certain messages, namely that agents were alive after something happened.

**lemma** *B\_Issues\_A*:  
 "[[ Says B A (Crypt K (Nonce Nb)) ∈ set evs;  
   Key K ∉ analz (spies evs);  
   A ∉ bad; B ∉ bad; evs ∈ ns\_shared ]]  
 ⇒ B Issues A with (Crypt K (Nonce Nb)) on evs"  
 <proof>

Tells A that B was alive after she sent him the session key. The session key must be assumed confidential for this deduction to be meaningful, but that assumption can be relaxed by the appropriate argument.

Precisely, the theorem guarantees (to A) key distribution of the session key to B. It also guarantees (to A) non-injective agreement of B with A on the session key. Both goals are available to A in the sense of Goal Availability.

**lemma** *A\_authenticates\_and\_keydist\_to\_B*:  
 "[[ Crypt K (Nonce NB) ∈ parts (spies evs);  
   Crypt (shrK A) {NA, Agent B, Key K, X} ∈ parts (spies evs);  
   Key K ∉ analz (knows Spy evs);  
   A ∉ bad; B ∉ bad; evs ∈ ns\_shared ]]  
 ⇒ B Issues A with (Crypt K (Nonce NB)) on evs"  
 <proof>

**lemma** *A\_trusts\_NS5*:  
 "[[ Crypt K {Nonce NB, Nonce NB} ∈ parts (spies evs);  
   Crypt (shrK A) {Nonce NA, Agent B, Key K, X} ∈ parts (spies evs);  
   Key K ∉ analz (spies evs);  
   A ∉ bad; B ∉ bad; evs ∈ ns\_shared ]]  
 ⇒ Says A B (Crypt K {Nonce NB, Nonce NB}) ∈ set evs"  
 <proof>

**lemma** *A\_Issues\_B*:  
 "[[ Says A B (Crypt K {Nonce NB, Nonce NB}) ∈ set evs;  
   Key K ∉ analz (spies evs);  
   A ∉ bad; B ∉ bad; evs ∈ ns\_shared ]]  
 ⇒ A Issues B with (Crypt K {Nonce NB, Nonce NB}) on evs"  
 <proof>

Tells B that A was alive after B issued NB.

Precisely, the theorem guarantees (to B) key distribution of the session key to A. It also guarantees (to B) non-injective agreement of A with B on the session key. Both goals are available to B in the sense of Goal Availability.

**lemma** *B\_authenticates\_and\_keydist\_to\_A*:  
 "[[ Crypt K {Nonce NB, Nonce NB} ∈ parts (spies evs);  
   Crypt (shrK B) {Key K, Agent A} ∈ parts (spies evs);  
   Key K ∉ analz (spies evs);  
   A ∉ bad; B ∉ bad; evs ∈ ns\_shared ]]  
 ⇒ A Issues B with (Crypt K {Nonce NB, Nonce NB}) on evs"  
 <proof>

end

## 4 The Kerberos Protocol, BAN Version

**theory** *Kerberos\_BAN* **imports** *Public* **begin**

From page 251 of Burrows, Abadi and Needham (1989). A Logic of Authentication. Proc. Royal Soc. 426

Confidentiality (secrecy) and authentication properties are also given in a temporal version: strong guarantees in a little abstracted - but very realistic - model.

**consts**

*sesKlife* :: nat

*authlife* :: nat

The ticket should remain fresh for two journeys on the network at least

**specification** (*sesKlife*)

*sesKlife\_LB* [iff]: " $2 \leq \text{sesKlife}$ "  
 $\langle \text{proof} \rangle$

The authenticator only for one journey

**specification** (*authlife*)

*authlife\_LB* [iff]: " $\text{authlife} \neq 0$ "  
 $\langle \text{proof} \rangle$

**abbreviation**

*CT* :: "event list  $\Rightarrow$  nat" **where**  
 $\text{"CT == length "}$

**abbreviation**

*expiredK* :: "[nat, event list]  $\Rightarrow$  bool" **where**  
 $\text{"expiredK T evs == sesKlife + T < CT evs"}$

**abbreviation**

*expiredA* :: "[nat, event list]  $\Rightarrow$  bool" **where**  
 $\text{"expiredA T evs == authlife + T < CT evs"}$

**constdefs**

*Issues* :: "[agent, agent, msg, event list]  $\Rightarrow$  bool"  
 $\text{"_ Issues _ with _ on _"}$   
 $\text{"A Issues B with X on evs ==}$   
 $\exists Y. \text{Says A B Y} \in \text{set evs} \ \& \ X \in \text{parts \{Y\}} \ \&$   
 $X \notin \text{parts (spies (takeWhile (\% z. z} \neq \text{Says A B Y) (rev evs)))}"$

```

before :: "[event, event list] => event list" ("before _ on _")
"before ev on evs == takeWhile (% z. z ~= ev) (rev evs)"

```

```

Unique :: "[event, event list] => bool" ("Unique _ on _")
"Unique ev on evs ==
  ev ∉ set (tl (dropWhile (% z. z ≠ ev) evs))"

```

```

inductive_set bankerberos :: "event list set"
where

```

```

  Nil: "[] ∈ bankerberos"

```

```

/ Fake: "[ evsf ∈ bankerberos; X ∈ synth (analz (spies evsf)) ]
  ⇒ Says Spy B X # evsf ∈ bankerberos"

```

```

/ BK1: "[ evs1 ∈ bankerberos ]
  ⇒ Says A Server {Agent A, Agent B} # evs1
  ∈ bankerberos"

```

```

/ BK2: "[ evs2 ∈ bankerberos; Key K ∉ used evs2; K ∈ symKeys;
  Says A' Server {Agent A, Agent B} ∈ set evs2 ]
  ⇒ Says Server A
    (Crypt (shrK A)
      {Number (CT evs2), Agent B, Key K,
       (Crypt (shrK B) {Number (CT evs2), Agent A, Key K})})
  # evs2 ∈ bankerberos"

```

```

/ BK3: "[ evs3 ∈ bankerberos;
  Says S A (Crypt (shrK A) {Number Tk, Agent B, Key K, Ticket})
  ∈ set evs3;
  Says A Server {Agent A, Agent B} ∈ set evs3;
  ¬ expiredK Tk evs3 ]
  ⇒ Says A B {Ticket, Crypt K {Agent A, Number (CT evs3)}}
  # evs3 ∈ bankerberos"

```

```

/ BK4: "[ evs4 ∈ bankerberos;
  Says A' B {(Crypt (shrK B) {Number Tk, Agent A, Key K}),
    (Crypt K {Agent A, Number Ta})} : set evs4;
  ¬ expiredK Tk evs4; ¬ expiredA Ta evs4 ]
  ⇒ Says B A (Crypt K (Number Ta)) # evs4
  ∈ bankerberos"

```

```

/ Oops: "[ evso ∈ bankerberos;
  Says Server A (Crypt (shrK A) {Number Tk, Agent B, Key K, Ticket})
  ∈ set evso;
  expiredK Tk evso ]

```

$$\implies \text{Notes Spy } \{ \text{Number Tk, Key K} \} \# \text{ evso} \in \text{bankerberos}$$

```

declare Says_imp_knows_Spy [THEN parts.Inj, dest]
declare parts.Body [dest]
declare analz_into_parts [dest]
declare Fake_parts_insert_in_Un [dest]

```

A "possibility property": there are traces that reach the end.

```

lemma "[Key K \notin used []; K \in symKeys]
  \implies \exists Timestamp. \exists evs \in bankerberos.
    Says B A (Crypt K (Number Timestamp))
      \in set evs"

```

*<proof>*

#### 4.1 Lemmas for reasoning about predicate "Issues"

```

lemma spies_Says_rev: "spies (evs @ [Says A B X]) = insert X (spies evs)"
<proof>

```

```

lemma spies_Gets_rev: "spies (evs @ [Gets A X]) = spies evs"
<proof>

```

```

lemma spies_Notes_rev: "spies (evs @ [Notes A X]) =
  (if A:bad then insert X (spies evs) else spies evs)"
<proof>

```

```

lemma spies_evs_rev: "spies evs = spies (rev evs)"
<proof>

```

```

lemmas parts_spies_evs_revD2 = spies_evs_rev [THEN equalityD2, THEN parts_mono]

```

```

lemma spies_takeWhile: "spies (takeWhile P evs) <= spies evs"
<proof>

```

```

lemmas parts_spies_takeWhile_mono = spies_takeWhile [THEN parts_mono]

```

Lemmas for reasoning about predicate "before"

```

lemma used_Says_rev: "used (evs @ [Says A B X]) = parts {X} \cup (used evs)"
<proof>

```

```

lemma used_Notes_rev: "used (evs @ [Notes A X]) = parts {X} \cup (used evs)"
<proof>

```

```

lemma used_Gets_rev: "used (evs @ [Gets B X]) = used evs"
<proof>

```

```

lemma used_evs_rev: "used evs = used (rev evs)"
<proof>

```

```

lemma used_takeWhile_used [rule_format]:
  "x : used (takeWhile P X) --> x : used X"
<proof>

```

**lemma** *set\_evs\_rev*: "set evs = set (rev evs)"  
 <proof>

**lemma** *takeWhile\_void* [rule\_format]:  
 "x  $\notin$  set evs  $\longrightarrow$  takeWhile ( $\lambda z. z \neq x$ ) evs = evs"  
 <proof>

Forwarding Lemma for reasoning about the encrypted portion of message BK3

**lemma** *BK3\_msg\_in\_parts\_spies*:  
 "Says S A (Crypt KA {Timestamp, B, K, X})  $\in$  set evs  
 $\implies X \in$  parts (spies evs)"  
 <proof>

**lemma** *Ops\_parts\_spies*:  
 "Says Server A (Crypt (shrK A) {Timestamp, B, K, X})  $\in$  set evs  
 $\implies K \in$  parts (spies evs)"  
 <proof>

Spy never sees another agent's shared key! (unless it's bad at start)

**lemma** *Spy\_see\_shrK* [simp]:  
 "evs  $\in$  bankerberos  $\implies$  (Key (shrK A)  $\in$  parts (spies evs)) = (A  $\in$  bad)"  
 <proof>

**lemma** *Spy\_analz\_shrK* [simp]:  
 "evs  $\in$  bankerberos  $\implies$  (Key (shrK A)  $\in$  analz (spies evs)) = (A  $\in$  bad)"  
 <proof>

**lemma** *Spy\_see\_shrK\_D* [dest!]:  
 "[[ Key (shrK A)  $\in$  parts (spies evs);  
 evs  $\in$  bankerberos ]]  $\implies$  A:bad"  
 <proof>

**lemmas** *Spy\_analz\_shrK\_D* = analz\_subset\_parts [THEN subsetD, THEN Spy\_see\_shrK\_D, dest!]

Nobody can have used non-existent keys!

**lemma** *new\_keys\_not\_used* [simp]:  
 "[[ Key K  $\notin$  used evs; K  $\in$  symKeys; evs  $\in$  bankerberos ]]  
 $\implies K \notin$  keysFor (parts (spies evs))"  
 <proof>

## 4.2 Lemmas concerning the form of items passed in messages

Describes the form of K, X and K' when the Server sends this message.

**lemma** *Says\_Server\_message\_form*:  
 "[[ Says Server A (Crypt K' {Number Tk, Agent B, Key K, Ticket})  
 $\in$  set evs; evs  $\in$  bankerberos ]]  
 $\implies K' =$  shrK A & K  $\notin$  range shrK &  
 Ticket = (Crypt (shrK B) {Number Tk, Agent A, Key K}) &  
 Key K  $\notin$  used(before

```

      Says Server A (Crypt K' {Number Tk, Agent B, Key K, Ticket})
      on evs) &
Tk = CT(before
  Says Server A (Crypt K' {Number Tk, Agent B, Key K, Ticket})
  on evs)"
⟨proof⟩

```

If the encrypted message appears then it originated with the Server PROVIDED that A is NOT compromised! This allows A to verify freshness of the session key.

**lemma** *Kab\_authentic*:

```

"[[ Crypt (shrK A) {Number Tk, Agent B, Key K, X}
   ∈ parts (spies evs);
   A ∉ bad; evs ∈ bankerberos ]]
⇒ Says Server A (Crypt (shrK A) {Number Tk, Agent B, Key K, X})
   ∈ set evs"
⟨proof⟩

```

If the TICKET appears then it originated with the Server

FRESHNESS OF THE SESSION KEY to B

**lemma** *ticket\_authentic*:

```

"[[ Crypt (shrK B) {Number Tk, Agent A, Key K} ∈ parts (spies evs);
   B ∉ bad; evs ∈ bankerberos ]]
⇒ Says Server A
   (Crypt (shrK A) {Number Tk, Agent B, Key K,
    Crypt (shrK B) {Number Tk, Agent A, Key K}})
   ∈ set evs"
⟨proof⟩

```

EITHER describes the form of X when the following message is sent, OR reduces it to the Fake case. Use *Says\_Server\_message\_form* if applicable.

**lemma** *Says\_S\_message\_form*:

```

"[[ Says S A (Crypt (shrK A) {Number Tk, Agent B, Key K, X})
   ∈ set evs;
   evs ∈ bankerberos ]]
⇒ (K ∉ range shrK & X = (Crypt (shrK B) {Number Tk, Agent A, Key K}))
   | X ∈ analz (spies evs)"
⟨proof⟩

```

Session keys are not used to encrypt other session keys

**lemma** *analz\_image\_freshK* [rule\_format (no\_asm)]:

```

"evs ∈ bankerberos ⇒
∀ K KK. KK ⊆ - (range shrK) →
  (Key K ∈ analz (Key'KK Un (spies evs))) =
  (K ∈ KK | Key K ∈ analz (spies evs))"
⟨proof⟩

```

**lemma** *analz\_insert\_freshK*:

```

"[[ evs ∈ bankerberos; KAB ∉ range shrK ]] ⇒
  (Key K ∈ analz (insert (Key KAB) (spies evs))) =
  (K = KAB | Key K ∈ analz (spies evs))"

```

*<proof>*

The session key K uniquely identifies the message

**lemma** *unique\_session\_keys*:

```
"[[ Says Server A
  (Crypt (shrK A) {Number Tk, Agent B, Key K, X}) ∈ set evs;
  Says Server A'
  (Crypt (shrK A') {Number Tk', Agent B', Key K, X'}) ∈ set evs;
  evs ∈ bankerberos ]] ⇒ A=A' & Tk=Tk' & B=B' & X = X'"
```

*<proof>*

**lemma** *Server\_Unique*:

```
"[[ Says Server A
  (Crypt (shrK A) {Number Tk, Agent B, Key K, Ticket}) ∈ set evs;
  evs ∈ bankerberos ]] ⇒
  Unique Says Server A (Crypt (shrK A) {Number Tk, Agent B, Key K, Ticket})
  on evs"
```

*<proof>*

### 4.3 Non-temporal guarantees, explicitly relying on non-occurrence of oops events - refined below by temporal guarantees

Non temporal treatment of confidentiality

Lemma: the session key sent in msg BK2 would be lost by oops if the spy could see it!

**lemma** *lemma\_conf [rule\_format (no\_asm)]*:

```
"[[ A ∉ bad; B ∉ bad; evs ∈ bankerberos ]]
⇒ Says Server A
  (Crypt (shrK A) {Number Tk, Agent B, Key K,
                  Crypt (shrK B) {Number Tk, Agent A, Key K}})
  ∈ set evs →
  Key K ∈ analz (spies evs) → Notes Spy {Number Tk, Key K} ∈ set evs"
```

*<proof>*

Confidentiality for the Server: Spy does not see the keys sent in msg BK2 as long as they have not expired.

**lemma** *Confidentiality\_S*:

```
"[[ Says Server A
  (Crypt K' {Number Tk, Agent B, Key K, Ticket}) ∈ set evs;
  Notes Spy {Number Tk, Key K} ∉ set evs;
  A ∉ bad; B ∉ bad; evs ∈ bankerberos
  ]] ⇒ Key K ∉ analz (spies evs)"
```

*<proof>*

Confidentiality for Alice

**lemma** *Confidentiality\_A*:

```
"[[ Crypt (shrK A) {Number Tk, Agent B, Key K, X} ∈ parts (spies evs);
  Notes Spy {Number Tk, Key K} ∉ set evs;
  A ∉ bad; B ∉ bad; evs ∈ bankerberos
  ]] ⇒ Key K ∉ analz (spies evs)"
```



### 4.3 Non-temporal guarantees, explicitly relying on non-occurrence of oops events - refined below by temporal guarantees

*<proof>*

Confidentiality for Bob

**lemma** Confidentiality\_B:

```
"[ Crypt (shrK B) {Number Tk, Agent A, Key K}
  ∈ parts (spies evs);
  Notes Spy {Number Tk, Key K} ∉ set evs;
  A ∉ bad; B ∉ bad; evs ∈ bankerberos
] ⇒ Key K ∉ analz (spies evs)"
```

*<proof>*

Non temporal treatment of authentication

Lemmas *lemma\_A* and *lemma\_B* in fact are common to both temporal and non-temporal treatments

**lemma** lemma\_A [rule\_format]:

```
"[ A ∉ bad; B ∉ bad; evs ∈ bankerberos ]
⇒
  Key K ∉ analz (spies evs) →
  Says Server A (Crypt (shrK A) {Number Tk, Agent B, Key K, X})
  ∈ set evs →
  Crypt K {Agent A, Number Ta} ∈ parts (spies evs) →
  Says A B {X, Crypt K {Agent A, Number Ta}}
  ∈ set evs"
```

*<proof>*

**lemma** lemma\_B [rule\_format]:

```
"[ B ∉ bad; evs ∈ bankerberos ]
⇒ Key K ∉ analz (spies evs) →
  Says Server A (Crypt (shrK A) {Number Tk, Agent B, Key K, X})
  ∈ set evs →
  Crypt K (Number Ta) ∈ parts (spies evs) →
  Says B A (Crypt K (Number Ta)) ∈ set evs"
```

*<proof>*

The "r" suffix indicates theorems where the confidentiality assumptions are relaxed by the corresponding arguments.

Authentication of A to B

**lemma** B\_authenticates\_A\_r:

```
"[ Crypt K {Agent A, Number Ta} ∈ parts (spies evs);
  Crypt (shrK B) {Number Tk, Agent A, Key K} ∈ parts (spies evs);
  Notes Spy {Number Tk, Key K} ∉ set evs;
  A ∉ bad; B ∉ bad; evs ∈ bankerberos ]
⇒ Says A B {Crypt (shrK B) {Number Tk, Agent A, Key K},
  Crypt K {Agent A, Number Ta}} ∈ set evs"
```

*<proof>*

Authentication of B to A

**lemma** A\_authenticates\_B\_r:

```
"[ Crypt K (Number Ta) ∈ parts (spies evs);
  Crypt (shrK A) {Number Tk, Agent B, Key K, X} ∈ parts (spies evs);
  Notes Spy {Number Tk, Key K} ∉ set evs;
  A ∉ bad; B ∉ bad; evs ∈ bankerberos ]
```

$\Rightarrow \text{Says } B \ A \ (\text{Crypt } K \ (\text{Number } Ta)) \in \text{set evs}"$   
 $\langle \text{proof} \rangle$

**lemma** *B\_authenticates\_A*:

"[[ Crypt K {Agent A, Number Ta} ∈ parts (spies evs);  
 Crypt (shrK B) {Number Tk, Agent A, Key K} ∈ parts (spies evs);  
 Key K ∉ analz (spies evs);  
 A ∉ bad; B ∉ bad; evs ∈ bankerberos ]]  
 $\Rightarrow \text{Says } A \ B \ \{\text{Crypt (shrK B) \{Number Tk, Agent A, Key K\},}$   
 $\text{Crypt } K \ \{\text{Agent A, Number Ta}\}\} \in \text{set evs}"$

$\langle \text{proof} \rangle$

**lemma** *A\_authenticates\_B*:

"[[ Crypt K (Number Ta) ∈ parts (spies evs);  
 Crypt (shrK A) {Number Tk, Agent B, Key K, X} ∈ parts (spies evs);  
 Key K ∉ analz (spies evs);  
 A ∉ bad; B ∉ bad; evs ∈ bankerberos ]]  
 $\Rightarrow \text{Says } B \ A \ (\text{Crypt } K \ (\text{Number } Ta)) \in \text{set evs}"$

$\langle \text{proof} \rangle$

#### 4.4 Temporal guarantees, relying on a temporal check that insures that no oops event occurred. These are available in the sense of goal availability

Temporal treatment of confidentiality

Lemma: the session key sent in msg BK2 would be EXPIRED if the spy could see it!

**lemma** *lemma\_conf\_temporal [rule\_format (no\_asm)]*:

"[[ A ∉ bad; B ∉ bad; evs ∈ bankerberos ]]  
 $\Rightarrow \text{Says Server } A$   
 $(\text{Crypt (shrK A) \{Number Tk, Agent B, Key K,}$   
 $\text{Crypt (shrK B) \{Number Tk, Agent A, Key K\}\})$   
 $\in \text{set evs} \longrightarrow$   
 $\text{Key } K \in \text{analz (spies evs)} \longrightarrow \text{expiredK Tk evs}"$

$\langle \text{proof} \rangle$

Confidentiality for the Server: Spy does not see the keys sent in msg BK2 as long as they have not expired.

**lemma** *Confidentiality\_S\_temporal*:

"[[ Says Server A  
 $(\text{Crypt } K' \ \{\text{Number } T, \text{Agent } B, \text{Key } K, X\}) \in \text{set evs};$   
 $\neg \text{expiredK } T \text{ evs};$   
 A ∉ bad; B ∉ bad; evs ∈ bankerberos  
 ]]  
 $\Rightarrow \text{Key } K \notin \text{analz (spies evs)"}$

$\langle \text{proof} \rangle$

Confidentiality for Alice

**lemma** *Confidentiality\_A\_temporal*:

"[[ Crypt (shrK A) {Number T, Agent B, Key K, X} ∈ parts (spies evs);  
 $\neg \text{expiredK } T \text{ evs};$   
 A ∉ bad; B ∉ bad; evs ∈ bankerberos

#### 4.5 Treatment of the key distribution goal using trace inspection. All guarantees are in non-temporal form, hence n

]]  $\Rightarrow$  Key  $K \notin \text{analz}(\text{spies evs})$ "  
 <proof>

Confidentiality for Bob

**lemma** Confidentiality\_B\_temporal:

"[[ Crypt (shrK B) {Number Tk, Agent A, Key K}  
    $\in \text{parts}(\text{spies evs})$ ;  
    $\neg \text{expiredK Tk evs}$ ;  
    $A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{bankerberos}$   
 ]]  $\Rightarrow$  Key  $K \notin \text{analz}(\text{spies evs})$ "  
 <proof>

Temporal treatment of authentication

Authentication of A to B

**lemma** B\_authenticates\_A\_temporal:

"[[ Crypt K {Agent A, Number Ta}  $\in \text{parts}(\text{spies evs})$ ;  
   Crypt (shrK B) {Number Tk, Agent A, Key K}  
    $\in \text{parts}(\text{spies evs})$ ;  
    $\neg \text{expiredK Tk evs}$ ;  
    $A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{bankerberos}$  ]]  
 $\Rightarrow$  Says A B {Crypt (shrK B) {Number Tk, Agent A, Key K},  
   Crypt K {Agent A, Number Ta}}  $\in \text{set evs}$ "  
 <proof>

Authentication of B to A

**lemma** A\_authenticates\_B\_temporal:

"[[ Crypt K (Number Ta)  $\in \text{parts}(\text{spies evs})$ ;  
   Crypt (shrK A) {Number Tk, Agent B, Key K, X}  
    $\in \text{parts}(\text{spies evs})$ ;  
    $\neg \text{expiredK Tk evs}$ ;  
    $A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{bankerberos}$  ]]  
 $\Rightarrow$  Says B A (Crypt K (Number Ta))  $\in \text{set evs}$ "  
 <proof>

#### 4.5 Treatment of the key distribution goal using trace inspection. All guarantees are in non-temporal form, hence non available, though their temporal form is trivial to derive. These guarantees also convey a stronger form of authentication - non-injective agreement on the session key

**lemma** B\_Issues\_A:

"[[ Says B A (Crypt K (Number Ta))  $\in \text{set evs}$ ;  
   Key  $K \notin \text{analz}(\text{spies evs})$ ;  
    $A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{bankerberos}$  ]]  
 $\Rightarrow$  B Issues A with (Crypt K (Number Ta)) on evs"  
 <proof>

**lemma** A\_authenticates\_and\_keydist\_to\_B:

"[[ Crypt K (Number Ta)  $\in \text{parts}(\text{spies evs})$ ;  
   Crypt (shrK A) {Number Tk, Agent B, Key K, X}  $\in \text{parts}(\text{spies evs})$ ;

```

      Key K ∉ analz (spies evs);
      A ∉ bad; B ∉ bad; evs ∈ bankerberos ]
    ⇒ B Issues A with (Crypt K (Number Ta)) on evs"
  ⟨proof⟩

```

```

lemma A_Issues_B:
  "[[ Says A B {Ticket, Crypt K {Agent A, Number Ta}}
    ∈ set evs;
    Key K ∉ analz (spies evs);
    A ∉ bad; B ∉ bad; evs ∈ bankerberos ]
  ⇒ A Issues B with (Crypt K {Agent A, Number Ta}) on evs"
  ⟨proof⟩

```

```

lemma B_authenticates_and_keydist_to_A:
  "[[ Crypt K {Agent A, Number Ta} ∈ parts (spies evs);
    Crypt (shrK B) {Number Tk, Agent A, Key K} ∈ parts (spies evs);
    Key K ∉ analz (spies evs);
    A ∉ bad; B ∉ bad; evs ∈ bankerberos ]
  ⇒ A Issues B with (Crypt K {Agent A, Number Ta}) on evs"
  ⟨proof⟩

```

**end**

## 5 The Kerberos Protocol, BAN Version, with Gets event

**theory** Kerberos\_BAN\_Gets **imports** Public **begin**

From page 251 of Burrows, Abadi and Needham (1989). A Logic of Authentication. Proc. Royal Soc. 426

Confidentiality (secrecy) and authentication properties rely on temporal checks: strong guarantees in a little abstracted - but very realistic - model.

**consts**

```

  sesKlife    :: nat

```

```

  authlife    :: nat

```

The ticket should remain fresh for two journeys on the network at least

The Gets event causes longer traces for the protocol to reach its end

```

specification (sesKlife)
  sesKlife_LB [iff]: "4 ≤ sesKlife"
  ⟨proof⟩

```

The authenticator only for one journey

The Gets event causes longer traces for the protocol to reach its end

**specification** (*authlife*)

```
authlife_LB [iff]:    "2 ≤ authlife"
  ⟨proof⟩
```

**abbreviation**

```
CT :: "event list=>nat" where
  "CT == length"
```

**abbreviation**

```
expiredK :: "[nat, event list] => bool" where
  "expiredK T evs == sesKlife + T < CT evs"
```

**abbreviation**

```
expiredA :: "[nat, event list] => bool" where
  "expiredA T evs == authlife + T < CT evs"
```

**constdefs**

```
before :: "[event, event list] => event list" ("before _ on _")
  "before ev on evs == takeWhile (% z. z ~= ev) (rev evs)"
```

```
Unique :: "[event, event list] => bool" ("Unique _ on _")
  "Unique ev on evs ==
   ev ∉ set (tl (dropWhile (% z. z ≠ ev) evs))"
```

**inductive\_set** *bankerb\_gets* :: "event list set"

**where**

```
Nil:    "[] ∈ bankerb_gets"

/ Fake: "[ evsf ∈ bankerb_gets; X ∈ synth (analz (knows Spy evsf)) ]
  ⇒ Says Spy B X # evsf ∈ bankerb_gets"

/ Reception: "[ evsr ∈ bankerb_gets; Says A B X ∈ set evsr ]
  ⇒ Gets B X # evsr ∈ bankerb_gets"

/ BK1: "[ evs1 ∈ bankerb_gets ]
  ⇒ Says A Server {Agent A, Agent B} # evs1
    ∈ bankerb_gets"

/ BK2: "[ evs2 ∈ bankerb_gets; Key K ∉ used evs2; K ∈ symKeys;
  Gets Server {Agent A, Agent B} ∈ set evs2 ]
  ⇒ Says Server A
    (Crypt (shrK A)
      {Number (CT evs2), Agent B, Key K,
       (Crypt (shrK B) {Number (CT evs2), Agent A, Key K})})
```

```

# evs2 ∈ bankerb_gets"

/ BK3: "[ evs3 ∈ bankerb_gets;
  Gets A (Crypt (shrK A) {Number Tk, Agent B, Key K, Ticket})
  ∈ set evs3;
  Says A Server {Agent A, Agent B} ∈ set evs3;
  ¬ expiredK Tk evs3 ]
⇒ Says A B {Ticket, Crypt K {Agent A, Number (CT evs3)}}
  # evs3 ∈ bankerb_gets"

/ BK4: "[ evs4 ∈ bankerb_gets;
  Gets B {(Crypt (shrK B) {Number Tk, Agent A, Key K}),
  (Crypt K {Agent A, Number Ta}) } : set evs4;
  ¬ expiredK Tk evs4; ¬ expiredA Ta evs4 ]
⇒ Says B A (Crypt K (Number Ta)) # evs4
  ∈ bankerb_gets"

/ Oops: "[ evso ∈ bankerb_gets;
  Says Server A (Crypt (shrK A) {Number Tk, Agent B, Key K, Ticket})
  ∈ set evso;
  expiredK Tk evso ]
⇒ Notes Spy {Number Tk, Key K} # evso ∈ bankerb_gets"

declare Says_imp_knows_Spy [THEN parts.Inj, dest]
declare parts.Body [dest]
declare analz_into_parts [dest]
declare Fake_parts_insert_in_Un [dest]
declare knows_Spy_partsEs [elim]

A "possibility property": there are traces that reach the end.

lemma "[Key K ∉ used []; K ∈ symKeys]
  ⇒ ∃ Timestamp. ∃ evs ∈ bankerb_gets.
    Says B A (Crypt K (Number Timestamp))
    ∈ set evs"
<proof>

Lemmas about reception invariant: if a message is received it certainly was sent

lemma Gets_imp_Says :
  "[ Gets B X ∈ set evs; evs ∈ bankerb_gets ] ⇒ ∃ A. Says A B X ∈ set
  evs"
<proof>

lemma Gets_imp_knows_Spy:
  "[ Gets B X ∈ set evs; evs ∈ bankerb_gets ] ⇒ X ∈ knows Spy evs"
<proof>

lemma Gets_imp_knows_Spy_parts[dest]:
  "[ Gets B X ∈ set evs; evs ∈ bankerb_gets ] ⇒ X ∈ parts (knows Spy
  evs)"
<proof>

```

**lemma** *Gets\_imp\_knows*:

"[[ Gets B X ∈ set evs; evs ∈ bankerb\_gets ]] ⇒ X ∈ knows B evs"  
 <proof>

**lemma** *Gets\_imp\_knows\_analz*:

"[[ Gets B X ∈ set evs; evs ∈ bankerb\_gets ]] ⇒ X ∈ analz (knows B evs)"  
 <proof>

Lemmas for reasoning about predicate "before"

**lemma** *used\_Says\_rev*: "used (evs @ [Says A B X]) = parts {X} ∪ (used evs)"  
 <proof>

**lemma** *used\_Notes\_rev*: "used (evs @ [Notes A X]) = parts {X} ∪ (used evs)"  
 <proof>

**lemma** *used\_Gets\_rev*: "used (evs @ [Gets B X]) = used evs"  
 <proof>

**lemma** *used\_evs\_rev*: "used evs = used (rev evs)"  
 <proof>

**lemma** *used\_takeWhile\_used* [rule\_format]:  
 "x : used (takeWhile P X) --> x : used X"  
 <proof>

**lemma** *set\_evs\_rev*: "set evs = set (rev evs)"  
 <proof>

**lemma** *takeWhile\_void* [rule\_format]:  
 "x ∉ set evs → takeWhile (λz. z ≠ x) evs = evs"  
 <proof>

Forwarding Lemma for reasoning about the encrypted portion of message BK3

**lemma** *BK3\_msg\_in\_parts\_knows\_Spy*:  
 "[[ Gets A (Crypt KA {Timestamp, B, K, X}) ∈ set evs; evs ∈ bankerb\_gets ]]  
 ⇒ X ∈ parts (knows Spy evs)"  
 <proof>

**lemma** *Oops\_parts\_knows\_Spy*:  
 "Says Server A (Crypt (shrK A) {Timestamp, B, K, X}) ∈ set evs  
 ⇒ K ∈ parts (knows Spy evs)"  
 <proof>

Spy never sees another agent's shared key! (unless it's bad at start)

**lemma** *Spy\_see\_shrK* [simp]:  
 "evs ∈ bankerb\_gets ⇒ (Key (shrK A) ∈ parts (knows Spy evs)) = (A  
 ∈ bad)"  
 <proof>

**lemma** *Spy\_analz\_shrK* [simp]:

" $evs \in \text{bankerb\_gets} \implies (\text{Key } (\text{shrK } A) \in \text{analz } (\text{knows Spy } evs)) = (A \in \text{bad})$ "  
 $\langle \text{proof} \rangle$

**lemma** *Spy\_see\_shrK\_D* [*dest!*]:  
 "[ $\text{Key } (\text{shrK } A) \in \text{parts } (\text{knows Spy } evs);$   
 $evs \in \text{bankerb\_gets}$  ]  $\implies A:\text{bad}$ "  
 $\langle \text{proof} \rangle$

**lemmas** *Spy\_analz\_shrK\_D* = *analz\_subset\_parts* [*THEN subsetD, THEN Spy\_see\_shrK\_D, dest!*]

Nobody can have used non-existent keys!

**lemma** *new\_keys\_not\_used* [*simp*]:  
 "[ $\text{Key } K \notin \text{used } evs; K \in \text{symKeys}; evs \in \text{bankerb\_gets}$  ]  
 $\implies K \notin \text{keysFor } (\text{parts } (\text{knows Spy } evs))$ "  
 $\langle \text{proof} \rangle$

## 5.1 Lemmas concerning the form of items passed in messages

Describes the form of K, X and K' when the Server sends this message.

**lemma** *Says\_Server\_message\_form*:  
 "[ $\text{Says Server } A (\text{Crypt } K' \{ \text{Number } Tk, \text{Agent } B, \text{Key } K, \text{Ticket} \})$   
 $\in \text{set } evs; evs \in \text{bankerb\_gets}$  ]  
 $\implies K' = \text{shrK } A \ \& \ K \notin \text{range shrK} \ \&$   
 $\text{Ticket} = (\text{Crypt } (\text{shrK } B) \{ \text{Number } Tk, \text{Agent } A, \text{Key } K \}) \ \&$   
 $\text{Key } K \notin \text{used}(\text{before}$   
 $\quad \text{Says Server } A (\text{Crypt } K' \{ \text{Number } Tk, \text{Agent } B, \text{Key } K, \text{Ticket} \})$   
 $\quad \text{on } evs) \ \&$   
 $\text{Tk} = \text{CT}(\text{before}$   
 $\quad \text{Says Server } A (\text{Crypt } K' \{ \text{Number } Tk, \text{Agent } B, \text{Key } K, \text{Ticket} \})$   
 $\quad \text{on } evs)$ "  
 $\langle \text{proof} \rangle$

If the encrypted message appears then it originated with the Server PROVIDED that A is NOT compromised! This allows A to verify freshness of the session key.

**lemma** *Kab\_authentic*:  
 "[ $\text{Crypt } (\text{shrK } A) \{ \text{Number } Tk, \text{Agent } B, \text{Key } K, X \}$   
 $\in \text{parts } (\text{knows Spy } evs);$   
 $A \notin \text{bad}; evs \in \text{bankerb\_gets}$  ]  
 $\implies \text{Says Server } A (\text{Crypt } (\text{shrK } A) \{ \text{Number } Tk, \text{Agent } B, \text{Key } K, X \})$   
 $\in \text{set } evs$ "  
 $\langle \text{proof} \rangle$

If the TICKET appears then it originated with the Server

FRESHNESS OF THE SESSION KEY to B

**lemma** *ticket\_authentic*:  
 "[ $\text{Crypt } (\text{shrK } B) \{ \text{Number } Tk, \text{Agent } A, \text{Key } K \} \in \text{parts } (\text{knows Spy } evs);$   
 $B \notin \text{bad}; evs \in \text{bankerb\_gets}$  ]



$\implies$  Says Server A  
 (Crypt (shrK A) {Number Tk, Agent B, Key K,  
 Crypt (shrK B) {Number Tk, Agent A, Key K}})  
 $\in \text{set evs}$   
 <proof>

EITHER describes the form of X when the following message is sent, OR reduces it to the Fake case. Use *Says\_Server\_message\_form* if applicable.

**lemma** *Gets\_Server\_message\_form*:

"[ Gets A (Crypt (shrK A) {Number Tk, Agent B, Key K, X})  
 $\in \text{set evs};$   
 $\text{evs} \in \text{bankerb\_gets}$  ]  
 $\implies (K \notin \text{range shrK} \ \& \ X = (\text{Crypt (shrK B) \{Number Tk, Agent A, Key K\}}))$   
 $\mid X \in \text{analz (knows Spy evs)}$ "  
 <proof>

Reliability guarantees: honest agents act as we expect

**lemma** *BK3\_imp\_Gets*:

"[ Says A B {Ticket, Crypt K {Agent A, Number Ta}}  $\in \text{set evs};$   
 $A \notin \text{bad}; \text{evs} \in \text{bankerb\_gets}$  ]  
 $\implies \exists \text{Tk. Gets A (Crypt (shrK A) \{Number Tk, Agent B, Key K, Ticket\})}$   
 $\in \text{set evs}$ "  
 <proof>

**lemma** *BK4\_imp\_Gets*:

"[ Says B A (Crypt K (Number Ta))  $\in \text{set evs};$   
 $B \notin \text{bad}; \text{evs} \in \text{bankerb\_gets}$  ]  
 $\implies \exists \text{Tk. Gets B \{Crypt (shrK B) \{Number Tk, Agent A, Key K\},}$   
 $\text{Crypt K \{Agent A, Number Ta\}} \in \text{set evs}$ "  
 <proof>

**lemma** *Gets\_A\_knows\_K*:

"[ Gets A (Crypt (shrK A) {Number Tk, Agent B, Key K, Ticket})  $\in \text{set evs};$   
 $\text{evs} \in \text{bankerb\_gets}$  ]  
 $\implies \text{Key K} \in \text{analz (knows A evs)}$ "  
 <proof>

**lemma** *Gets\_B\_knows\_K*:

"[ Gets B {Crypt (shrK B) {Number Tk, Agent A, Key K},  
 Crypt K {Agent A, Number Ta}}  $\in \text{set evs};$   
 $\text{evs} \in \text{bankerb\_gets}$  ]  
 $\implies \text{Key K} \in \text{analz (knows B evs)}$ "  
 <proof>

Session keys are not used to encrypt other session keys

**lemma** *analz\_image\_freshK [rule\_format (no\_asm)]*:

"evs  $\in \text{bankerb\_gets} \implies$   
 $\forall K \text{ KK. } \text{KK} \subseteq - (\text{range shrK}) \longrightarrow$   
 $(\text{Key K} \in \text{analz (Key'KK Un (knows Spy evs))}) =$   
 $(K \in \text{KK} \mid \text{Key K} \in \text{analz (knows Spy evs)})$ "  
 <proof>

```

lemma analz_insert_freshK:
  "[[ evs ∈ bankerb_gets; KAB ∉ range shrK ]] ⇒
   (Key K ∈ analz (insert (Key KAB) (knows Spy evs))) =
   (K = KAB | Key K ∈ analz (knows Spy evs))"
<proof>

```

The session key K uniquely identifies the message

```

lemma unique_session_keys:
  "[[ Says Server A
     (Crypt (shrK A) {Number Tk, Agent B, Key K, X}) ∈ set evs;
   Says Server A'
     (Crypt (shrK A') {Number Tk', Agent B', Key K, X'}) ∈ set evs;
   evs ∈ bankerb_gets ]] ⇒ A=A' & Tk=Tk' & B=B' & X = X'"
<proof>

```

```

lemma unique_session_keys_Gets:
  "[[ Gets A
     (Crypt (shrK A) {Number Tk, Agent B, Key K, X}) ∈ set evs;
   Gets A
     (Crypt (shrK A) {Number Tk', Agent B', Key K, X'}) ∈ set evs;
   A ∉ bad; evs ∈ bankerb_gets ]] ⇒ Tk=Tk' & B=B' & X = X'"
<proof>

```

```

lemma Server_Unique:
  "[[ Says Server A
     (Crypt (shrK A) {Number Tk, Agent B, Key K, Ticket}) ∈ set evs;
   evs ∈ bankerb_gets ]] ⇒
   Unique Says Server A (Crypt (shrK A) {Number Tk, Agent B, Key K, Ticket})
   on evs"
<proof>

```

## 5.2 Non-temporal guarantees, explicitly relying on non-occurrence of oops events - refined below by temporal guarantees

Non temporal treatment of confidentiality

Lemma: the session key sent in msg BK2 would be lost by oops if the spy could see it!

```

lemma lemma_conf [rule_format (no_asm)]:
  "[[ A ∉ bad; B ∉ bad; evs ∈ bankerb_gets ]
   ⇒ Says Server A
     (Crypt (shrK A) {Number Tk, Agent B, Key K,
                      Crypt (shrK B) {Number Tk, Agent A, Key K}})
   ∈ set evs ⇒
   Key K ∈ analz (knows Spy evs) ⇒ Notes Spy {Number Tk, Key K} ∈ set
   evs"
<proof>

```

Confidentiality for the Server: Spy does not see the keys sent in msg BK2 as long as they have not expired.

```

lemma Confidentiality_S:

```

## 5.2 Non-temporal guarantees, explicitly relying on non-occurrence of oops events - refined below by temporal guarantees

```

    "[ Says Server A
      (Crypt K' {|Number Tk, Agent B, Key K, Ticket|}) ∈ set evs;
      Notes Spy {|Number Tk, Key K|} ∉ set evs;
      A ∉ bad; B ∉ bad; evs ∈ bankerb_gets
    ] ⇒ Key K ∉ analz (knows Spy evs)"
  <proof>

```

Confidentiality for Alice

**lemma Confidentiality\_A:**

```

    "[ Crypt (shrK A) {|Number Tk, Agent B, Key K, X|} ∈ parts (knows Spy evs);
      Notes Spy {|Number Tk, Key K|} ∉ set evs;
      A ∉ bad; B ∉ bad; evs ∈ bankerb_gets
    ] ⇒ Key K ∉ analz (knows Spy evs)"
  <proof>

```

Confidentiality for Bob

**lemma Confidentiality\_B:**

```

    "[ Crypt (shrK B) {|Number Tk, Agent A, Key K|}
      ∈ parts (knows Spy evs);
      Notes Spy {|Number Tk, Key K|} ∉ set evs;
      A ∉ bad; B ∉ bad; evs ∈ bankerb_gets
    ] ⇒ Key K ∉ analz (knows Spy evs)"
  <proof>

```

Non temporal treatment of authentication

Lemmas *lemma\_A* and *lemma\_B* in fact are common to both temporal and non-temporal treatments

**lemma lemma\_A [rule\_format]:**

```

    "[ A ∉ bad; B ∉ bad; evs ∈ bankerb_gets ]
    ⇒
      Key K ∉ analz (knows Spy evs) →
      Says Server A (Crypt (shrK A) {|Number Tk, Agent B, Key K, X|})
      ∈ set evs →
      Crypt K {|Agent A, Number Ta|} ∈ parts (knows Spy evs) →
      Says A B {|X, Crypt K {|Agent A, Number Ta|}|}
      ∈ set evs"
  <proof>

```

**lemma lemma\_B [rule\_format]:**

```

    "[ B ∉ bad; evs ∈ bankerb_gets ]
    ⇒ Key K ∉ analz (knows Spy evs) →
      Says Server A (Crypt (shrK A) {|Number Tk, Agent B, Key K, X|})
      ∈ set evs →
      Crypt K (Number Ta) ∈ parts (knows Spy evs) →
      Says B A (Crypt K (Number Ta)) ∈ set evs"
  <proof>

```

The "r" suffix indicates theorems where the confidentiality assumptions are relaxed by the corresponding arguments.

Authentication of A to B

**lemma B\_authenticates\_A\_r:**

```

    "[ Crypt K {|Agent A, Number Ta|} ∈ parts (knows Spy evs);

```

```

    Crypt (shrK B) {Number Tk, Agent A, Key K} ∈ parts (knows Spy evs);
    Notes Spy {Number Tk, Key K} ∉ set evs;
    A ∉ bad; B ∉ bad; evs ∈ bankerb_gets ]
  ⇒ Says A B {Crypt (shrK B) {Number Tk, Agent A, Key K},
               Crypt K {Agent A, Number Ta}} ∈ set evs"
<proof>

```

Authentication of B to A

```

lemma A_authenticates_B_r:
  "[ Crypt K (Number Ta) ∈ parts (knows Spy evs);
    Crypt (shrK A) {Number Tk, Agent B, Key K, X} ∈ parts (knows Spy evs);
    Notes Spy {Number Tk, Key K} ∉ set evs;
    A ∉ bad; B ∉ bad; evs ∈ bankerb_gets ]
  ⇒ Says B A (Crypt K (Number Ta)) ∈ set evs"
<proof>

```

```

lemma B_authenticates_A:
  "[ Crypt K {Agent A, Number Ta} ∈ parts (spies evs);
    Crypt (shrK B) {Number Tk, Agent A, Key K} ∈ parts (spies evs);
    Key K ∉ analz (spies evs);
    A ∉ bad; B ∉ bad; evs ∈ bankerb_gets ]
  ⇒ Says A B {Crypt (shrK B) {Number Tk, Agent A, Key K},
               Crypt K {Agent A, Number Ta}} ∈ set evs"
<proof>

```

```

lemma A_authenticates_B:
  "[ Crypt K (Number Ta) ∈ parts (spies evs);
    Crypt (shrK A) {Number Tk, Agent B, Key K, X} ∈ parts (spies evs);
    Key K ∉ analz (spies evs);
    A ∉ bad; B ∉ bad; evs ∈ bankerb_gets ]
  ⇒ Says B A (Crypt K (Number Ta)) ∈ set evs"
<proof>

```

### 5.3 Temporal guarantees, relying on a temporal check that insures that no oops event occurred. These are available in the sense of goal availability

Temporal treatment of confidentiality

Lemma: the session key sent in msg BK2 would be EXPIRED if the spy could see it!

```

lemma lemma_conf_temporal [rule_format (no_asm)]:
  "[ A ∉ bad; B ∉ bad; evs ∈ bankerb_gets ]
  ⇒ Says Server A
      (Crypt (shrK A) {Number Tk, Agent B, Key K,
                      Crypt (shrK B) {Number Tk, Agent A, Key K}})
    ∈ set evs →
    Key K ∈ analz (knows Spy evs) → expiredK Tk evs"
<proof>

```

Confidentiality for the Server: Spy does not see the keys sent in msg BK2 as long as they have not expired.

lemma Confidentiality\_S\_temporal:

#### 5.4 Combined guarantees of key distribution and non-injective agreement on the session keys61

```

"[[ Says Server A
  (Crypt K' {Number T, Agent B, Key K, X}) ∈ set evs;
  ¬ expiredK T evs;
  A ∉ bad; B ∉ bad; evs ∈ bankerb_gets
]] ⇒ Key K ∉ analz (knows Spy evs)"
⟨proof⟩

```

Confidentiality for Alice

**lemma** Confidentiality\_A\_temporal:

```

"[[ Crypt (shrK A) {Number T, Agent B, Key K, X} ∈ parts (knows Spy evs);
  ¬ expiredK T evs;
  A ∉ bad; B ∉ bad; evs ∈ bankerb_gets
]] ⇒ Key K ∉ analz (knows Spy evs)"
⟨proof⟩

```

Confidentiality for Bob

**lemma** Confidentiality\_B\_temporal:

```

"[[ Crypt (shrK B) {Number Tk, Agent A, Key K}
  ∈ parts (knows Spy evs);
  ¬ expiredK Tk evs;
  A ∉ bad; B ∉ bad; evs ∈ bankerb_gets
]] ⇒ Key K ∉ analz (knows Spy evs)"
⟨proof⟩

```

Temporal treatment of authentication

Authentication of A to B

**lemma** B\_authenticates\_A\_temporal:

```

"[[ Crypt K {Agent A, Number Ta} ∈ parts (knows Spy evs);
  Crypt (shrK B) {Number Tk, Agent A, Key K}
  ∈ parts (knows Spy evs);
  ¬ expiredK Tk evs;
  A ∉ bad; B ∉ bad; evs ∈ bankerb_gets ]]
⇒ Says A B {Crypt (shrK B) {Number Tk, Agent A, Key K},
  Crypt K {Agent A, Number Ta}} ∈ set evs"
⟨proof⟩

```

Authentication of B to A

**lemma** A\_authenticates\_B\_temporal:

```

"[[ Crypt K (Number Ta) ∈ parts (knows Spy evs);
  Crypt (shrK A) {Number Tk, Agent B, Key K, X}
  ∈ parts (knows Spy evs);
  ¬ expiredK Tk evs;
  A ∉ bad; B ∉ bad; evs ∈ bankerb_gets ]]
⇒ Says B A (Crypt K (Number Ta)) ∈ set evs"
⟨proof⟩

```

#### 5.4 Combined guarantees of key distribution and non-injective agreement on the session keys

**lemma** B\_authenticates\_and\_keydist\_to\_A:

```

"[[ Gets B {Crypt (shrK B) {Number Tk, Agent A, Key K},
  Crypt K {Agent A, Number Ta}} ∈ set evs;

```

```

    Key K ∉ analz (spies evs);
    A ∉ bad; B ∉ bad; evs ∈ bankerb_gets ]
  ⇒ Says A B {Crypt (shrK B) {Number Tk, Agent A, Key K}},
           Crypt K {Agent A, Number Ta}} ∈ set evs
  ∧ Key K ∈ analz (knows A evs)"
⟨proof⟩

lemma A_authenticates_and_keydist_to_B:
  "[ Gets A (Crypt (shrK A) {Number Tk, Agent B, Key K, Ticket}) ∈ set
  evs;
    Gets A (Crypt K (Number Ta)) ∈ set evs;
    Key K ∉ analz (spies evs);
    A ∉ bad; B ∉ bad; evs ∈ bankerb_gets ]
  ⇒ Says B A (Crypt K (Number Ta)) ∈ set evs
  ∧ Key K ∈ analz (knows B evs)"
⟨proof⟩

```

end

## 6 The Kerberos Protocol, Version IV

theory *KerberosIV* imports *Public* begin

The "u" prefix indicates theorems referring to an updated version of the protocol. The "r" suffix indicates theorems where the confidentiality assumptions are relaxed by the corresponding arguments.

**abbreviation**

*Kas* :: agent where "Kas == Server"

**abbreviation**

*Tgs* :: agent where "Tgs == Friend 0"

**axioms**

*Tgs\_not\_bad* [iff]: "*Tgs* ∉ bad"

— *Tgs* is secure — we already know that *Kas* is secure

**constdefs**

```

authKeys :: "event list => key set"
"authKeys evs == {authK. ∃ A Peer Ta. Says Kas A
                      (Crypt (shrK A) {Key authK, Agent Peer, Number Ta,
                      (Crypt (shrK Peer) {Agent A, Agent Peer, Key authK, Number
Ta})
                      }) ∈ set evs}"

```

*Issues* :: "[agent, agent, msg, event list] => bool"

```

      ("_ Issues _ with _ on _")
    "A Issues B with X on evs ==
       $\exists Y. \text{Says } A \ B \ Y \in \text{set evs} \ \& \ X \in \text{parts } \{Y\} \ \& \\
      X \notin \text{parts } (\text{spies } (\text{takeWhile } (\% z. z \neq \text{Says } A \ B \ Y) (\text{rev evs}))))"$ 

    before :: "[event, event list] => event list" ("before _ on _")
    "before ev on evs == takeWhile (% z. z ~= ev) (rev evs)"

    Unique :: "[event, event list] => bool" ("Unique _ on _")
    "Unique ev on evs ==
      ev  $\notin$  set (tl (dropWhile (% z. z  $\neq$  ev) evs))"

consts

    authKlife    :: nat

    servKlife    :: nat

    authlife     :: nat

    replylife    :: nat

specification (authKlife)
    authKlife_LB [iff]: "2  $\leq$  authKlife"
    <proof>

specification (servKlife)
    servKlife_LB [iff]: "2 + authKlife  $\leq$  servKlife"
    <proof>

specification (authlife)
    authlife_LB [iff]: "Suc 0  $\leq$  authlife"
    <proof>

specification (replylife)
    replylife_LB [iff]: "Suc 0  $\leq$  replylife"
    <proof>

abbreviation

    CT :: "event list=>nat" where
    "CT == length"

abbreviation
    expiredAK :: "[nat, event list] => bool" where
    "expiredAK Ta evs == authKlife + Ta < CT evs"

abbreviation

```

```

expiredSK :: "[nat, event list] => bool" where
  "expiredSK Ts evs == servKlife + Ts < CT evs"

```

#### abbreviation

```

expiredA :: "[nat, event list] => bool" where
  "expiredA T evs == authlife + T < CT evs"

```

#### abbreviation

```

valid :: "[nat, nat] => bool" ("valid _ wrt _") where
  "valid T1 wrt T2 == T1 <= replylife + T2"

```

#### constdefs

```

AKcryptSK :: "[key, key, event list] => bool"
  "AKcryptSK authK servK evs ==
    $\exists A B Ts.$ 
   Says Tgs A (Crypt authK
                  $\{\{$ Key servK, Agent B, Number Ts,
                 Crypt (shrK B)  $\{\{$ Agent A, Agent B, Key servK, Number
Ts $\}\}$ 
    $\in$  set evs"

```

```

inductive_set kerbIV :: "event list set"
  where

```

```

  Nil: "[]  $\in$  kerbIV"

```

```

  / Fake: " $\llbracket$  evsf  $\in$  kerbIV;  $X \in$  synth (analz (spies evsf))  $\rrbracket$ 
     $\implies$  Says Spy B X # evsf  $\in$  kerbIV"

```

```

  / K1: " $\llbracket$  evs1  $\in$  kerbIV  $\rrbracket$ 
     $\implies$  Says A Kas  $\{\{$ Agent A, Agent Tgs, Number (CT evs1) $\}\}$  # evs1
     $\in$  kerbIV"

```

```

  / K2: " $\llbracket$  evs2  $\in$  kerbIV; Key authK  $\notin$  used evs2; authK  $\in$  symKeys;
    Says A' Kas  $\{\{$ Agent A, Agent Tgs, Number T1 $\}\}$   $\in$  set evs2  $\rrbracket$ 
     $\implies$  Says Kas A
      (Crypt (shrK A)  $\{\{$ Key authK, Agent Tgs, Number (CT evs2),
        (Crypt (shrK Tgs)  $\{\{$ Agent A, Agent Tgs, Key authK,
          Number (CT evs2) $\}\}$   $\rrbracket$ ) # evs2  $\in$  kerbIV"

```



```

/ K3: "[ evs3 ∈ kerbIV;
      Says A Kas {Agent A, Agent Tgs, Number T1} ∈ set evs3;
      Says Kas' A (Crypt (shrK A) {Key authK, Agent Tgs, Number Ta,
      authTicket}) ∈ set evs3;
      valid Ta wrt T1
    ]
    ⇒ Says A Tgs {authTicket,
                  (Crypt authK {Agent A, Number (CT evs3)}),
                  Agent B} # evs3 ∈ kerbIV"

```

```

/ K4: "[ evs4 ∈ kerbIV; Key servK ∉ used evs4; servK ∈ symKeys;
      B ≠ Tgs; authK ∈ symKeys;
      Says A' Tgs {
        (Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK,
        Number Ta}),
        (Crypt authK {Agent A, Number T2}), Agent B}
      ∈ set evs4;
      ¬ expiredAK Ta evs4;
      ¬ expiredA T2 evs4;
      servKlife + (CT evs4) ≤ authKlife + Ta
    ]
    ⇒ Says Tgs A
      (Crypt authK {Key servK, Agent B, Number (CT evs4),
        Crypt (shrK B) {Agent A, Agent B, Key servK,
        Number (CT evs4)}})
      # evs4 ∈ kerbIV"

```

```

/ K5: "[ evs5 ∈ kerbIV; authK ∈ symKeys; servK ∈ symKeys;
      Says A Tgs
        {authTicket, Crypt authK {Agent A, Number T2},
        Agent B}
      ∈ set evs5;
      Says Tgs' A
        (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
      ∈ set evs5;
      valid Ts wrt T2 ]
    ⇒ Says A B {servTicket,
                  Crypt servK {Agent A, Number (CT evs5)}}
      # evs5 ∈ kerbIV"

```

```

/ K6: "[ evs6 ∈ kerbIV;

```

```

    Says A' B {
      (Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}),
      (Crypt servK {Agent A, Number T3})
    }
    ∈ set evs6;
    ¬ expiredSK Ts evs6;
    ¬ expiredA T3 evs6
  ]
  ⇒ Says B A (Crypt servK (Number T3))
    # evs6 ∈ kerbIV"

/ Oops1: "[ evs01 ∈ kerbIV; A ≠ Spy;
  Says Kas A
    (Crypt (shrK A) {Key authK, Agent Tgs, Number Ta,
                     authTicket}) ∈ set evs01;
  expiredAK Ta evs01 ]
  ⇒ Says A Spy {Agent A, Agent Tgs, Number Ta, Key authK}
    # evs01 ∈ kerbIV"

/ Oops2: "[ evs02 ∈ kerbIV; A ≠ Spy;
  Says Tgs A
    (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
    ∈ set evs02;
  expiredSK Ts evs02 ]
  ⇒ Says A Spy {Agent A, Agent B, Number Ts, Key servK}
    # evs02 ∈ kerbIV"

declare Says_imp_knows_Spy [THEN parts.Inj, dest]
declare parts.Body [dest]
declare analz_into_parts [dest]
declare Fake_parts_insert_in_Un [dest]

```

## 6.1 Lemmas about lists, for reasoning about Issues

**lemma** spies\_Says\_rev: "spies (evs @ [Says A B X]) = insert X (spies evs)"  
 <proof>

**lemma** spies\_Gets\_rev: "spies (evs @ [Gets A X]) = spies evs"  
 <proof>

**lemma** spies\_Notes\_rev: "spies (evs @ [Notes A X]) =  
 (if A:bad then insert X (spies evs) else spies evs)"  
 <proof>

**lemma** spies\_evs\_rev: "spies evs = spies (rev evs)"  
 <proof>

**lemmas** parts\_spies\_evs\_revD2 = spies\_evs\_rev [THEN equalityD2, THEN parts\_mono]

**lemma** spies\_takeWhile: "spies (takeWhile P evs) <= spies evs"  
 <proof>

**lemmas** parts\_spies\_takeWhile\_mono = spies\_takeWhile [THEN parts\_mono]

## 6.2 Lemmas about authKeys

**lemma** authKeys\_empty: "authKeys [] = {}"  
 <proof>

**lemma** authKeys\_not\_insert:  
 "( $\forall$  A Ta akey Peer.  
   ev  $\neq$  Says Kas A (Crypt (shrK A) {akey, Agent Peer, Ta,  
     (Crypt (shrK Peer) {Agent A, Agent Peer, akey, Ta})})  
    $\implies$  authKeys (ev # evs) = authKeys evs"  
 <proof>

**lemma** authKeys\_insert:  
 "authKeys  
   (Says Kas A (Crypt (shrK A) {Key K, Agent Peer, Number Ta,  
     (Crypt (shrK Peer) {Agent A, Agent Peer, Key K, Number Ta})}) # evs)  
   = insert K (authKeys evs)"  
 <proof>

**lemma** authKeys\_simp:  
 "K  $\in$  authKeys  
   (Says Kas A (Crypt (shrK A) {Key K', Agent Peer, Number Ta,  
     (Crypt (shrK Peer) {Agent A, Agent Peer, Key K', Number Ta})}) # evs)  
    $\implies$  K = K' | K  $\in$  authKeys evs"  
 <proof>

**lemma** authKeysI:  
 "Says Kas A (Crypt (shrK A) {Key K, Agent Tgs, Number Ta,  
   (Crypt (shrK Tgs) {Agent A, Agent Tgs, Key K, Number Ta})})  $\in$  set evs  
    $\implies$  K  $\in$  authKeys evs"  
 <proof>

**lemma** authKeys\_used: "K  $\in$  authKeys evs  $\implies$  Key K  $\in$  used evs"  
 <proof>

## 6.3 Forwarding Lemmas

–For reasoning about the encrypted portion of message K3–

**lemma** K3\_msg\_in\_parts\_spies:  
 "Says Kas' A (Crypt KeyA {authK, Peer, Ta, authTicket})  
    $\in$  set evs  $\implies$  authTicket  $\in$  parts (spies evs)"  
 <proof>

**lemma** Ops\_range\_spies1:  
 "[[ Says Kas A (Crypt KeyA {Key authK, Peer, Ta, authTicket})

$$\begin{aligned} & \in \text{set evs} ; \\ & \text{evs} \in \text{kerbIV} \implies \text{authK} \notin \text{range shrK} \ \& \ \text{authK} \in \text{symKeys} \end{aligned}$$
 $\langle \text{proof} \rangle$

–For reasoning about the encrypted portion of message K5–

**lemma** *K5\_msg\_in\_parts\_spies*:  

$$\begin{aligned} & \text{"Says Tgs' A (Crypt authK \{servK, Agent B, Ts, servTicket\})} \\ & \quad \in \text{set evs} \implies \text{servTicket} \in \text{parts (spies evs)} \end{aligned}$$
 $\langle \text{proof} \rangle$

**lemma** *Ops\_range\_spies2*:  

$$\begin{aligned} & \text{"[ Says Tgs A (Crypt authK \{Key servK, Agent B, Ts, servTicket\})} \\ & \quad \in \text{set evs} ; \\ & \quad \text{evs} \in \text{kerbIV} \implies \text{servK} \notin \text{range shrK} \ \& \ \text{servK} \in \text{symKeys} \end{aligned}$$
 $\langle \text{proof} \rangle$

**lemma** *Says\_ticket\_parts*:  

$$\begin{aligned} & \text{"Says S A (Crypt K \{SesKey, B, TimeStamp, Ticket\})} \in \text{set evs} \\ & \implies \text{Ticket} \in \text{parts (spies evs)} \end{aligned}$$
 $\langle \text{proof} \rangle$

**lemma** *Spy\_see\_shrK [simp]*:  

$$\text{"evs} \in \text{kerbIV} \implies (\text{Key (shrK A)} \in \text{parts (spies evs)}) = (A \in \text{bad})"$$
 $\langle \text{proof} \rangle$

**lemma** *Spy\_analz\_shrK [simp]*:  

$$\text{"evs} \in \text{kerbIV} \implies (\text{Key (shrK A)} \in \text{analz (spies evs)}) = (A \in \text{bad})"$$
 $\langle \text{proof} \rangle$

**lemma** *Spy\_see\_shrK\_D [dest!]*:  

$$\text{"[ Key (shrK A) \in parts (spies evs); evs} \in \text{kerbIV} \implies A:\text{bad} \text{"}$$
 $\langle \text{proof} \rangle$

**lemmas** *Spy\_analz\_shrK\_D* = *analz\_subset\_parts [THEN subsetD, THEN Spy\_see\_shrK\_D, dest!]*

Nobody can have used non-existent keys!

**lemma** *new\_keys\_not\_used [simp]*:  

$$\begin{aligned} & \text{"[Key K} \notin \text{used evs; K} \in \text{symKeys; evs} \in \text{kerbIV}] \\ & \implies K \notin \text{keysFor (parts (spies evs))} \end{aligned}$$
 $\langle \text{proof} \rangle$

**lemma** *new\_keys\_not\_analz*:  

$$\begin{aligned} & \text{"[evs} \in \text{kerbIV; K} \in \text{symKeys; Key K} \notin \text{used evs}] \\ & \implies K \notin \text{keysFor (analz (spies evs))} \end{aligned}$$
 $\langle \text{proof} \rangle$

## 6.4 Lemmas for reasoning about predicate "before"

**lemma** *used\_Says\_rev*:  $\text{"used (evs @ [Says A B X])} = \text{parts \{X\} } \cup (\text{used evs})"$   
 $\langle \text{proof} \rangle$

**lemma** *used\_Notes\_rev*:  $\text{"used (evs @ [Notes A X])} = \text{parts \{X\} } \cup (\text{used evs})"$

*<proof>*

**lemma** *used\_Gets\_rev*: "used (evs @ [Gets B X]) = used evs"

*<proof>*

**lemma** *used\_evs\_rev*: "used evs = used (rev evs)"

*<proof>*

**lemma** *used\_takeWhile\_used* [rule\_format]:  
"x : used (takeWhile P X) --> x : used X"

*<proof>*

**lemma** *set\_evs\_rev*: "set evs = set (rev evs)"

*<proof>*

**lemma** *takeWhile\_void* [rule\_format]:  
"x ∉ set evs → takeWhile (λz. z ≠ x) evs = evs"

*<proof>*

## 6.5 Regularity Lemmas

These concern the form of items passed in messages

Describes the form of all components sent by Kas

**lemma** *Says\_Kas\_message\_form*:

"[ Says Kas A (Crypt K {Key authK, Agent Peer, Number Ta, authTicket})  
  ∈ set evs;  
  evs ∈ kerbIV ] ⇒  
K = shrK A & Peer = Tgs &  
authK ∉ range shrK & authK ∈ authKeys evs & authK ∈ symKeys &  
authTicket = (Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta})  
&  
Key authK ∉ used(before  
  Says Kas A (Crypt K {Key authK, Agent Peer, Number Ta, authTicket})  
  on evs) &  
Ta = CT (before  
  Says Kas A (Crypt K {Key authK, Agent Peer, Number Ta, authTicket})  
  on evs)"

*<proof>*

**lemma** *SesKey\_is\_session\_key*:

"[ Crypt (shrK Tgs\_B) {Agent A, Agent Tgs\_B, Key SesKey, Number T}  
  ∈ parts (spies evs); Tgs\_B ∉ bad;  
  evs ∈ kerbIV ]  
⇒ SesKey ∉ range shrK"

*<proof>*

**lemma** *authTicket\_authentic*:

"[ Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}  
  ∈ parts (spies evs);  
  evs ∈ kerbIV ]

$\Rightarrow$  Says Kas A (Crypt (shrK A)  $\{\text{Key authK, Agent Tgs, Number Ta,}$   
 Crypt (shrK Tgs)  $\{\text{Agent A, Agent Tgs, Key authK, Number Ta}\}\}$ )  
 $\in \text{set evs}$ "  
 <proof>

**lemma** authTicket\_crypt\_authK:  
 "[ Crypt (shrK Tgs)  $\{\text{Agent A, Agent Tgs, Key authK, Number Ta}\}$   
 $\in \text{parts (spies evs)}$ ;  
 $\text{evs} \in \text{kerbIV}$  ]  
 $\Rightarrow \text{authK} \in \text{authKeys evs}$ "  
 <proof>

Describes the form of servK, servTicket and authK sent by Tgs

**lemma** Says\_Tgs\_message\_form:  
 "[ Says Tgs A (Crypt authK  $\{\text{Key servK, Agent B, Number Ts, servTicket}\}$ )  
 $\in \text{set evs}$ ;  
 $\text{evs} \in \text{kerbIV}$  ]  
 $\Rightarrow B \neq \text{Tgs} \ \&$   
 $\text{authK} \notin \text{range shrK} \ \& \ \text{authK} \in \text{authKeys evs} \ \& \ \text{authK} \in \text{symKeys} \ \&$   
 $\text{servK} \notin \text{range shrK} \ \& \ \text{servK} \notin \text{authKeys evs} \ \& \ \text{servK} \in \text{symKeys} \ \&$   
 $\text{servTicket} = (\text{Crypt (shrK B)} \ \{\text{Agent A, Agent B, Key servK, Number Ts}\})$   
 $\&$   
 $\text{Key servK} \notin \text{used (before}$   
 $\text{Says Tgs A (Crypt authK } \{\text{Key servK, Agent B, Number Ts, servTicket}\})$   
 $\text{on evs}) \ \&$   
 $\text{Ts} = \text{CT(before}$   
 $\text{Says Tgs A (Crypt authK } \{\text{Key servK, Agent B, Number Ts, servTicket}\})$   
 $\text{on evs})$  "  
 <proof>

**lemma** authTicket\_form:  
 "[ Crypt (shrK A)  $\{\text{Key authK, Agent Tgs, Ta, authTicket}\}$   
 $\in \text{parts (spies evs)}$ ;  
 $A \notin \text{bad}$ ;  
 $\text{evs} \in \text{kerbIV}$  ]  
 $\Rightarrow \text{authK} \notin \text{range shrK} \ \& \ \text{authK} \in \text{symKeys} \ \&$   
 $\text{authTicket} = \text{Crypt (shrK Tgs)} \ \{\text{Agent A, Agent Tgs, Key authK, Ta}\}$ "  
 <proof>

This form holds also over an authTicket, but is not needed below.

**lemma** servTicket\_form:  
 "[ Crypt authK  $\{\text{Key servK, Agent B, Ts, servTicket}\}$   
 $\in \text{parts (spies evs)}$ ;  
 $\text{Key authK} \notin \text{analz (spies evs)}$ ;  
 $\text{evs} \in \text{kerbIV}$  ]  
 $\Rightarrow \text{servK} \notin \text{range shrK} \ \& \ \text{servK} \in \text{symKeys} \ \&$   
 $(\exists A. \text{servTicket} = \text{Crypt (shrK B)} \ \{\text{Agent A, Agent B, Key servK, Ts}\})$ "  
 <proof>

Essentially the same as authTicket\_form

**lemma** Says\_kas\_message\_form:  
 "[ Says Kas' A (Crypt (shrK A)  
 $\{\text{Key authK, Agent Tgs, Ta, authTicket}\}) \in \text{set evs}$ ;  
 ]

```

    evs ∈ kerbIV ]
  ⇒ authK ∉ range shrK & authK ∈ symKeys &
    authTicket =
      Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Ta}
    / authTicket ∈ analz (spies evs)"
⟨proof⟩

```

**lemma** Says\_tgs\_message\_form:

```

  "[ Says Tgs' A (Crypt authK {Key servK, Agent B, Ts, servTicket})
    ∈ set evs; authK ∈ symKeys;
    evs ∈ kerbIV ]
  ⇒ servK ∉ range shrK &
    (∃ A. servTicket =
      Crypt (shrK B) {Agent A, Agent B, Key servK, Ts})
    / servTicket ∈ analz (spies evs)"
⟨proof⟩

```

## 6.6 Authenticity theorems: confirm origin of sensitive messages

**lemma** authK\_authentic:

```

  "[ Crypt (shrK A) {Key authK, Peer, Ta, authTicket}
    ∈ parts (spies evs);
    A ∉ bad; evs ∈ kerbIV ]
  ⇒ Says Kas A (Crypt (shrK A) {Key authK, Peer, Ta, authTicket})
    ∈ set evs"
⟨proof⟩

```

If a certain encrypted message appears then it originated with Tgs

**lemma** servK\_authentic:

```

  "[ Crypt authK {Key servK, Agent B, Number Ts, servTicket}
    ∈ parts (spies evs);
    Key authK ∉ analz (spies evs);
    authK ∉ range shrK;
    evs ∈ kerbIV ]
  ⇒ ∃ A. Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
    ∈ set evs"
⟨proof⟩

```

**lemma** servK\_authentic\_bis:

```

  "[ Crypt authK {Key servK, Agent B, Number Ts, servTicket}
    ∈ parts (spies evs);
    Key authK ∉ analz (spies evs);
    B ≠ Tgs;
    evs ∈ kerbIV ]
  ⇒ ∃ A. Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
    ∈ set evs"
⟨proof⟩

```

Authenticity of servK for B

**lemma** servTicket\_authentic\_Tgs:

```

  "[ Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}
    ∈ parts (spies evs); B ≠ Tgs; B ∉ bad;

```

```

    evs ∈ kerbIV ]
⇒ ∃ authK.
  Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts,
    Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}})
    ∈ set evs"
⟨proof⟩

```

Anticipated here from next subsection

**lemma** *K4\_imp\_K2*:

```

"[[ Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
  ∈ set evs; evs ∈ kerbIV]]
⇒ ∃ Ta. Says Kas A
  (Crypt (shrK A)
    {Key authK, Agent Tgs, Number Ta,
    Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}})
  ∈ set evs"
⟨proof⟩

```

Anticipated here from next subsection

**lemma** *u\_K4\_imp\_K2*:

```

"[[ Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
  ∈ set evs; evs ∈ kerbIV]]
⇒ ∃ Ta. (Says Kas A (Crypt (shrK A) {Key authK, Agent Tgs, Number Ta,
  Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}})
    ∈ set evs
    & servKlife + Ts ≤ authKlife + Ta)"
⟨proof⟩

```

**lemma** *servTicket\_authentic\_Kas*:

```

"[[ Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}
  ∈ parts (spies evs); B ≠ Tgs; B ∉ bad;
  evs ∈ kerbIV ]
⇒ ∃ authK Ta.
  Says Kas A
    (Crypt (shrK A) {Key authK, Agent Tgs, Number Ta,
    Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}})
  ∈ set evs"
⟨proof⟩

```

**lemma** *u\_servTicket\_authentic\_Kas*:

```

"[[ Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}
  ∈ parts (spies evs); B ≠ Tgs; B ∉ bad;
  evs ∈ kerbIV ]
⇒ ∃ authK Ta. Says Kas A (Crypt(shrK A) {Key authK, Agent Tgs, Number Ta,
  Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}})
  ∈ set evs
  & servKlife + Ts ≤ authKlife + Ta"
⟨proof⟩

```

**lemma** *servTicket\_authentic*:

```

"[[ Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}
  ∈ parts (spies evs); B ≠ Tgs; B ∉ bad;
  evs ∈ kerbIV ]
⇒ ∃ Ta authK.

```



## 6.7 Reliability: friendly agents send something if something else happened 73

```

    Says Kas A (Crypt (shrK A) {Key authK, Agent Tgs, Number Ta,
                          Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number
Ta}}))
      ∈ set evs
    & Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts,
                          Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}})
      ∈ set evs"
⟨proof⟩

```

```

lemma u_servTicket_authentic:
  "[ Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}
    ∈ parts (spies evs); B ≠ Tgs; B ∉ bad;
    evs ∈ kerbIV ]
  ⇒ ∃ Ta authK.
    (Says Kas A (Crypt (shrK A) {Key authK, Agent Tgs, Number Ta,
                              Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number
Ta}}))
      ∈ set evs
    & Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts,
                              Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}})
      ∈ set evs
    & servKlife + Ts ≤ authKlife + Ta)"
⟨proof⟩

```

```

lemma u_NotexpiredSK_NotexpiredAK:
  "[ ¬ expiredSK Ts evs; servKlife + Ts ≤ authKlife + Ta ]
  ⇒ ¬ expiredAK Ta evs"
⟨proof⟩

```

## 6.7 Reliability: friendly agents send something if something else happened

```

lemma K3_imp_K2:
  "[ Says A Tgs
    {authTicket, Crypt authK {Agent A, Number T2}, Agent B}
    ∈ set evs;
    A ∉ bad; evs ∈ kerbIV ]
  ⇒ ∃ Ta. Says Kas A (Crypt (shrK A)
    {Key authK, Agent Tgs, Number Ta, authTicket})
    ∈ set evs"
⟨proof⟩

```

Anticipated here from next subsection. An authK is encrypted by one and only one Shared key. A servK is encrypted by one and only one authK.

```

lemma Key_unique_SesKey:
  "[ Crypt K {Key SesKey, Agent B, T, Ticket}
    ∈ parts (spies evs);
    Crypt K' {Key SesKey, Agent B', T', Ticket'}
    ∈ parts (spies evs); Key SesKey ∉ analz (spies evs);
    evs ∈ kerbIV ]
  ⇒ K=K' & B=B' & T=T' & Ticket=Ticket'"
⟨proof⟩

```

```

lemma Tgs_authenticates_A:

```

```

"[[ Crypt authK {Agent A, Number T2} ∈ parts (spies evs);
   Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}
   ∈ parts (spies evs);
   Key authK ∉ analz (spies evs); A ∉ bad; evs ∈ kerbIV ]]
⇒ ∃ B. Says A Tgs {
   Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta},
   Crypt authK {Agent A, Number T2}, Agent B } ∈ set evs"
⟨proof⟩

```

**lemma Says\_K5:**

```

"[[ Crypt servK {Agent A, Number T3} ∈ parts (spies evs);
   Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts,
   servTicket}) ∈ set evs;
   Key servK ∉ analz (spies evs);
   A ∉ bad; B ∉ bad; evs ∈ kerbIV ]]
⇒ Says A B {servTicket, Crypt servK {Agent A, Number T3}} ∈ set evs"
⟨proof⟩

```

Anticipated here from next subsection

**lemma unique\_CryptKey:**

```

"[[ Crypt (shrK B) {Agent A, Agent B, Key SesKey, T}
   ∈ parts (spies evs);
   Crypt (shrK B') {Agent A', Agent B', Key SesKey, T'}
   ∈ parts (spies evs); Key SesKey ∉ analz (spies evs);
   evs ∈ kerbIV ]]
⇒ A=A' & B=B' & T=T'"
⟨proof⟩

```

**lemma Says\_K6:**

```

"[[ Crypt servK (Number T3) ∈ parts (spies evs);
   Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts,
   servTicket}) ∈ set evs;
   Key servK ∉ analz (spies evs);
   A ∉ bad; B ∉ bad; evs ∈ kerbIV ]]
⇒ Says B A (Crypt servK (Number T3)) ∈ set evs"
⟨proof⟩

```

Needs a unicity theorem, hence moved here

**lemma servK\_authentic\_ter:**

```

"[[ Says Kas A
   (Crypt (shrK A) {Key authK, Agent Tgs, Number Ta, authTicket}) ∈ set
   evs;
   Crypt authK {Key servK, Agent B, Number Ts, servTicket}
   ∈ parts (spies evs);
   Key authK ∉ analz (spies evs);
   evs ∈ kerbIV ]]
⇒ Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
   ∈ set evs"
⟨proof⟩

```

## 6.8 Unicity Theorems

The session key, if secure, uniquely identifies the Ticket whether authTicket or servTicket. As a matter of fact, one can read also Tgs in the place of B.

**lemma** *unique\_authKeys*:

```
"[ Says Kas A
  (Crypt Ka {Key authK, Agent Tgs, Ta, X}) ∈ set evs;
  Says Kas A'
  (Crypt Ka' {Key authK, Agent Tgs, Ta', X'}) ∈ set evs;
  evs ∈ kerbIV ] ⇒ A=A' & Ka=Ka' & Ta=Ta' & X=X'"
⟨proof⟩
```

*servK* uniquely identifies the message from *Tgs*

**lemma** *unique\_servKeys*:

```
"[ Says Tgs A
  (Crypt K {Key servK, Agent B, Ts, X}) ∈ set evs;
  Says Tgs A'
  (Crypt K' {Key servK, Agent B', Ts', X'}) ∈ set evs;
  evs ∈ kerbIV ] ⇒ A=A' & B=B' & K=K' & Ts=Ts' & X=X'"
⟨proof⟩
```

Revised unicity theorems

**lemma** *Kas\_Unique*:

```
"[ Says Kas A
  (Crypt Ka {Key authK, Agent Tgs, Ta, authTicket}) ∈ set evs;
  evs ∈ kerbIV ] ⇒
  Unique (Says Kas A (Crypt Ka {Key authK, Agent Tgs, Ta, authTicket}))
  on evs"
⟨proof⟩
```

**lemma** *Tgs\_Unique*:

```
"[ Says Tgs A
  (Crypt authK {Key servK, Agent B, Ts, servTicket}) ∈ set evs;
  evs ∈ kerbIV ] ⇒
  Unique (Says Tgs A (Crypt authK {Key servK, Agent B, Ts, servTicket}))
  on evs"
⟨proof⟩
```

## 6.9 Lemmas About the Predicate $AKcryptSK$

**lemma** *not\_AKcryptSK\_Nil* [iff]: " $\neg AKcryptSK\ authK\ servK\ []$ "

⟨proof⟩

**lemma** *AKcryptSKI*:

```
"[ Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, X}) ∈ set evs;
  evs ∈ kerbIV ] ⇒ AKcryptSK authK servK evs"
⟨proof⟩
```

**lemma** *AKcryptSK\_Says* [simp]:

```
"AKcryptSK authK servK (Says S A X # evs) =
  (Tgs = S &
   (∃ B Ts. X = Crypt authK
     {Key servK, Agent B, Number Ts,
      Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}})
  )
  / AKcryptSK authK servK evs"
⟨proof⟩
```

**lemma** *Auth\_fresh\_not\_AKcryptSK*:  
 "[[ Key authK  $\notin$  used evs; evs  $\in$  kerbIV ]]  
 $\implies \neg$  AKcryptSK authK servK evs"  
 <proof>

**lemma** *Serv\_fresh\_not\_AKcryptSK*:  
 "Key servK  $\notin$  used evs  $\implies \neg$  AKcryptSK authK servK evs"  
 <proof>

**lemma** *authK\_not\_AKcryptSK*:  
 "[[ Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, tk}  
 $\in$  parts (spies evs); evs  $\in$  kerbIV ]]  
 $\implies \neg$  AKcryptSK K authK evs"  
 <proof>

A secure serverkey cannot have been used to encrypt others

**lemma** *servK\_not\_AKcryptSK*:  
 "[[ Crypt (shrK B) {Agent A, Agent B, Key SK, Number Ts}  $\in$  parts (spies evs);  
 Key SK  $\notin$  analz (spies evs); SK  $\in$  symKeys;  
 B  $\neq$  Tgs; evs  $\in$  kerbIV ]]  
 $\implies \neg$  AKcryptSK SK K evs"  
 <proof>

Long term keys are not issued as servKeys

**lemma** *shrK\_not\_AKcryptSK*:  
 "evs  $\in$  kerbIV  $\implies \neg$  AKcryptSK K (shrK A) evs"  
 <proof>

The Tgs message associates servK with authK and therefore not with any other key authK.

**lemma** *Says\_Tgs\_AKcryptSK*:  
 "[[ Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, X})  
 $\in$  set evs;  
 authK'  $\neq$  authK; evs  $\in$  kerbIV ]]  
 $\implies \neg$  AKcryptSK authK' servK evs"  
 <proof>

Equivalently

**lemma** *not\_different\_AKcryptSK*:  
 "[[ AKcryptSK authK servK evs;  
 authK'  $\neq$  authK; evs  $\in$  kerbIV ]]  
 $\implies \neg$  AKcryptSK authK' servK evs  $\wedge$  servK  $\in$  symKeys"  
 <proof>

**lemma** *AKcryptSK\_not\_AKcryptSK*:  
 "[[ AKcryptSK authK servK evs; evs  $\in$  kerbIV ]]  
 $\implies \neg$  AKcryptSK servK K evs"  
 <proof>

The only session keys that can be found with the help of session keys are those sent by Tgs in step K4.

We take some pains to express the property as a logical equivalence so that the simplifier can apply it.

```

lemma Key_analz_image_Key_lemma:
  "P  $\longrightarrow$  (Key K  $\in$  analz (Key'KK Un H))  $\longrightarrow$  (K:KK | Key K  $\in$  analz H)
   $\implies$ 
  P  $\longrightarrow$  (Key K  $\in$  analz (Key'KK Un H)) = (K:KK | Key K  $\in$  analz H)"
<proof>

```

```

lemma AKcryptSK_analz_insert:
  "[[ AKcryptSK K K' evs; K  $\in$  symKeys; evs  $\in$  kerbIV ]]
   $\implies$  Key K'  $\in$  analz (insert (Key K) (spies evs))"
<proof>

```

```

lemma authKeys_are_not_AKcryptSK:
  "[[ K  $\in$  authKeys evs Un range shrK; evs  $\in$  kerbIV ]]
   $\implies$   $\forall$  SK.  $\neg$  AKcryptSK SK K evs  $\wedge$  K  $\in$  symKeys"
<proof>

```

```

lemma not_authKeys_not_AKcryptSK:
  "[[ K  $\notin$  authKeys evs;
    K  $\notin$  range shrK; evs  $\in$  kerbIV ]]
   $\implies$   $\forall$  SK.  $\neg$  AKcryptSK K SK evs"
<proof>

```

## 6.10 Secrecy Theorems

For the Oops2 case of the next theorem

```

lemma Oops2_not_AKcryptSK:
  "[[ evs  $\in$  kerbIV;
    Says Tgs A (Crypt authK
      {Key servK, Agent B, Number Ts, servTicket})
       $\in$  set evs ]]
   $\implies$   $\neg$  AKcryptSK servK SK evs"
<proof>

```

Big simplification law for keys SK that are not crypted by keys in KK It helps prove three, otherwise hard, facts about keys. These facts are exploited as simplification laws for analz, and also "limit the damage" in case of loss of a key to the spy. See ESORICS98. [simplified by LCP]

```

lemma Key_analz_image_Key [rule_format (no_asm)]:
  "evs  $\in$  kerbIV  $\implies$ 
  ( $\forall$  SK KK. SK  $\in$  symKeys & KK  $\leq$  -(range shrK)  $\longrightarrow$ 
  ( $\forall$  K  $\in$  KK.  $\neg$  AKcryptSK K SK evs)  $\longrightarrow$ 
  (Key SK  $\in$  analz (Key'KK Un (spies evs))) =
  (SK  $\in$  KK | Key SK  $\in$  analz (spies evs)))"
<proof>

```

First simplification law for analz: no session keys encrypt authentication keys or shared keys.

```

lemma analz_insert_freshK1:
  "[[ evs  $\in$  kerbIV; K  $\in$  authKeys evs Un range shrK;

```

$$\text{SesKey} \notin \text{range shrK} \parallel$$

$$\implies (\text{Key } K \in \text{analz} (\text{insert} (\text{Key } \text{SesKey}) (\text{spies evs}))) =$$

$$(K = \text{SesKey} \mid \text{Key } K \in \text{analz} (\text{spies evs}))"$$

$$\langle \text{proof} \rangle$$

Second simplification law for analz: no service keys encrypt any other keys.

**lemma** *analz\_insert\_freshK2*:  

$$" \parallel \text{evs} \in \text{kerbIV}; \text{servK} \notin (\text{authKeys evs}); \text{servK} \notin \text{range shrK};$$

$$K \in \text{symKeys} \parallel$$

$$\implies (\text{Key } K \in \text{analz} (\text{insert} (\text{Key } \text{servK}) (\text{spies evs}))) =$$

$$(K = \text{servK} \mid \text{Key } K \in \text{analz} (\text{spies evs}))"$$

$$\langle \text{proof} \rangle$$

Third simplification law for analz: only one authentication key encrypts a certain service key.

**lemma** *analz\_insert\_freshK3*:  

$$" \parallel \text{AKcryptSK } \text{authK } \text{servK } \text{evs};$$

$$\text{authK}' \neq \text{authK}; \text{authK}' \notin \text{range shrK}; \text{evs} \in \text{kerbIV} \parallel$$

$$\implies (\text{Key } \text{servK} \in \text{analz} (\text{insert} (\text{Key } \text{authK}') (\text{spies evs}))) =$$

$$(\text{servK} = \text{authK}' \mid \text{Key } \text{servK} \in \text{analz} (\text{spies evs}))"$$

$$\langle \text{proof} \rangle$$

**lemma** *analz\_insert\_freshK3\_bis*:  

$$" \parallel \text{Says Tgs A}$$

$$(\text{Crypt } \text{authK } \{\text{Key } \text{servK}, \text{Agent B}, \text{Number Ts}, \text{servTicket}\})$$

$$\in \text{set evs};$$

$$\text{authK} \neq \text{authK}'; \text{authK}' \notin \text{range shrK}; \text{evs} \in \text{kerbIV} \parallel$$

$$\implies (\text{Key } \text{servK} \in \text{analz} (\text{insert} (\text{Key } \text{authK}') (\text{spies evs}))) =$$

$$(\text{servK} = \text{authK}' \mid \text{Key } \text{servK} \in \text{analz} (\text{spies evs}))"$$

$$\langle \text{proof} \rangle$$

a weakness of the protocol

**lemma** *authK\_compromises\_servK*:  

$$" \parallel \text{Says Tgs A}$$

$$(\text{Crypt } \text{authK } \{\text{Key } \text{servK}, \text{Agent B}, \text{Number Ts}, \text{servTicket}\})$$

$$\in \text{set evs}; \text{authK} \in \text{symKeys};$$

$$\text{Key } \text{authK} \in \text{analz} (\text{spies evs}); \text{evs} \in \text{kerbIV} \parallel$$

$$\implies \text{Key } \text{servK} \in \text{analz} (\text{spies evs})"$$

$$\langle \text{proof} \rangle$$

**lemma** *servK\_notin\_authKeysD*:  

$$" \parallel \text{Crypt } \text{authK } \{\text{Key } \text{servK}, \text{Agent B}, \text{Ts},$$

$$\text{Crypt } (\text{shrK B}) \{\text{Agent A}, \text{Agent B}, \text{Key } \text{servK}, \text{Ts}\}\}$$

$$\in \text{parts} (\text{spies evs});$$

$$\text{Key } \text{servK} \notin \text{analz} (\text{spies evs});$$

$$B \neq \text{Tgs}; \text{evs} \in \text{kerbIV} \parallel$$

$$\implies \text{servK} \notin \text{authKeys evs}"$$

$$\langle \text{proof} \rangle$$

If Spy sees the Authentication Key sent in msg K2, then the Key has expired.

**lemma** *Confidentiality\_Kas\_lemma* [rule\_format]:  

$$" \parallel \text{authK} \in \text{symKeys}; A \notin \text{bad}; \text{evs} \in \text{kerbIV} \parallel$$

$$\implies \text{Says Kas A}$$

```

      (Crypt (shrK A)
        {Key authK, Agent Tgs, Number Ta,
         Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}}})
    ∈ set evs ⟶
    Key authK ∈ analz (spies evs) ⟶
    expiredAK Ta evs"
⟨proof⟩

```

**lemma Confidentiality\_Kas:**

```

"[[ Says Kas A
   (Crypt Ka {Key authK, Agent Tgs, Number Ta, authTicket})
   ∈ set evs;
   ¬ expiredAK Ta evs;
   A ∉ bad; evs ∈ kerbIV ]]
⇒ Key authK ∉ analz (spies evs)"
⟨proof⟩

```

If Spy sees the Service Key sent in msg K4, then the Key has expired.

**lemma Confidentiality\_lemma [rule\_format]:**

```

"[[ Says Tgs A
   (Crypt authK
    {Key servK, Agent B, Number Ts,
     Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}}})
   ∈ set evs;
   Key authK ∉ analz (spies evs);
   servK ∈ symKeys;
   A ∉ bad; B ∉ bad; evs ∈ kerbIV ]]
⇒ Key servK ∈ analz (spies evs) ⟶
   expiredSK Ts evs"
⟨proof⟩

```

In the real world Tgs can't check wheter authK is secure!

**lemma Confidentiality\_Tgs:**

```

"[[ Says Tgs A
   (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
   ∈ set evs;
   Key authK ∉ analz (spies evs);
   ¬ expiredSK Ts evs;
   A ∉ bad; B ∉ bad; evs ∈ kerbIV ]]
⇒ Key servK ∉ analz (spies evs)"
⟨proof⟩

```

In the real world Tgs CAN check what Kas sends!

**lemma Confidentiality\_Tgs\_bis:**

```

"[[ Says Kas A
   (Crypt Ka {Key authK, Agent Tgs, Number Ta, authTicket})
   ∈ set evs;
   Says Tgs A
   (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
   ∈ set evs;
   ¬ expiredAK Ta evs; ¬ expiredSK Ts evs;
   A ∉ bad; B ∉ bad; evs ∈ kerbIV ]]
⇒ Key servK ∉ analz (spies evs)"

```

*<proof>*

Most general form

**lemmas** Confidentiality\_Tgs\_ter = authTicket\_authentic [THEN Confidentiality\_Tgs\_bis]

**lemmas** Confidentiality\_Auth\_A = authK\_authentic [THEN Confidentiality\_Kas]

Needs a confidentiality guarantee, hence moved here. Authenticity of servK for A

**lemma** servK\_authentic\_bis\_r:  
 "[ Crypt (shrK A) {Key authK, Agent Tgs, Number Ta, authTicket}  
   ∈ parts (spies evs);  
   Crypt authK {Key servK, Agent B, Number Ts, servTicket}  
   ∈ parts (spies evs);  
   ¬ expiredAK Ta evs; A ∉ bad; evs ∈ kerbIV ]  
 ⇒ Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, servTicket})  
   ∈ set evs"  
*<proof>*

**lemma** Confidentiality\_Serv\_A:  
 "[ Crypt (shrK A) {Key authK, Agent Tgs, Number Ta, authTicket}  
   ∈ parts (spies evs);  
   Crypt authK {Key servK, Agent B, Number Ts, servTicket}  
   ∈ parts (spies evs);  
   ¬ expiredAK Ta evs; ¬ expiredSK Ts evs;  
   A ∉ bad; B ∉ bad; evs ∈ kerbIV ]  
 ⇒ Key servK ∉ analz (spies evs)"  
*<proof>*

**lemma** Confidentiality\_B:  
 "[ Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}  
   ∈ parts (spies evs);  
   Crypt authK {Key servK, Agent B, Number Ts, servTicket}  
   ∈ parts (spies evs);  
   Crypt (shrK A) {Key authK, Agent Tgs, Number Ta, authTicket}  
   ∈ parts (spies evs);  
   ¬ expiredSK Ts evs; ¬ expiredAK Ta evs;  
   A ∉ bad; B ∉ bad; B ≠ Tgs; evs ∈ kerbIV ]  
 ⇒ Key servK ∉ analz (spies evs)"  
*<proof>*

**lemma** u\_Confidentiality\_B:  
 "[ Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}  
   ∈ parts (spies evs);  
   ¬ expiredSK Ts evs;  
   A ∉ bad; B ∉ bad; B ≠ Tgs; evs ∈ kerbIV ]  
 ⇒ Key servK ∉ analz (spies evs)"  
*<proof>*



6.11 Parties authentication: each party verifies "the identity of another party who generated some data" (quoted from

### 6.11 Parties authentication: each party verifies "the identity of another party who generated some data" (quoted from Neuman and Ts'o).

These guarantees don't assess whether two parties agree on the same session key: sending a message containing a key doesn't a priori state knowledge of the key.

*Tgs\_authenticates\_A* can be found above

**lemma** *A\_authenticates\_Tgs*:

```
"[[ Says Kas A
  (Crypt (shrK A) {Key authK, Agent Tgs, Number Ta, authTicket}) ∈ set
  evs;
  Crypt authK {Key servK, Agent B, Number Ts, servTicket}
    ∈ parts (spies evs);
  Key authK ∉ analz (spies evs);
  evs ∈ kerbIV ]]
⇒ Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
  ∈ set evs"
⟨proof⟩
```

**lemma** *B\_authenticates\_A*:

```
"[[ Crypt servK {Agent A, Number T3} ∈ parts (spies evs);
  Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}
    ∈ parts (spies evs);
  Key servK ∉ analz (spies evs);
  A ∉ bad; B ∉ bad; B ≠ Tgs; evs ∈ kerbIV ]]
⇒ Says A B {Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts},
  Crypt servK {Agent A, Number T3}} ∈ set evs"
⟨proof⟩
```

The second assumption tells B what kind of key servK is.

**lemma** *B\_authenticates\_A\_r*:

```
"[[ Crypt servK {Agent A, Number T3} ∈ parts (spies evs);
  Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}
    ∈ parts (spies evs);
  Crypt authK {Key servK, Agent B, Number Ts, servTicket}
    ∈ parts (spies evs);
  Crypt (shrK A) {Key authK, Agent Tgs, Number Ta, authTicket}
    ∈ parts (spies evs);
  ¬ expiredSK Ts evs; ¬ expiredAK Ta evs;
  B ≠ Tgs; A ∉ bad; B ∉ bad; evs ∈ kerbIV ]]
⇒ Says A B {Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts},
  Crypt servK {Agent A, Number T3}} ∈ set evs"
⟨proof⟩
```

*u\_B\_authenticates\_A* would be the same as *B\_authenticates\_A* because the servK confidentiality assumption is yet unrelaxed

**lemma** *u\_B\_authenticates\_A\_r*:

```
"[[ Crypt servK {Agent A, Number T3} ∈ parts (spies evs);
  Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}
    ∈ parts (spies evs);
```

$\neg \text{expiredSK } Ts \text{ evs};$   
 $B \neq Tgs; A \notin \text{bad}; B \notin \text{bad}; \text{ evs} \in \text{kerbIV} \parallel$   
 $\implies \text{Says } A \ B \ \{\{\text{Crypt } (\text{shrK } B) \ \{\text{Agent } A, \text{Agent } B, \text{Key } \text{servK}, \text{Number } Ts\}\},$   
 $\text{Crypt } \text{servK} \ \{\text{Agent } A, \text{Number } T3\}\} \in \text{set evs}"$   
 $\langle \text{proof} \rangle$

**lemma** *A\_authenticates\_B*:  
 $\parallel \text{Crypt } \text{servK} \ (\text{Number } T3) \in \text{parts } (\text{spies evs});$   
 $\text{Crypt } \text{authK} \ \{\{\text{Key } \text{servK}, \text{Agent } B, \text{Number } Ts, \text{servTicket}\}$   
 $\in \text{parts } (\text{spies evs});$   
 $\text{Crypt } (\text{shrK } A) \ \{\{\text{Key } \text{authK}, \text{Agent } Tgs, \text{Number } Ta, \text{authTicket}\}$   
 $\in \text{parts } (\text{spies evs});$   
 $\text{Key } \text{authK} \notin \text{analz } (\text{spies evs}); \text{Key } \text{servK} \notin \text{analz } (\text{spies evs});$   
 $A \notin \text{bad}; B \notin \text{bad}; \text{ evs} \in \text{kerbIV} \parallel$   
 $\implies \text{Says } B \ A \ (\text{Crypt } \text{servK} \ (\text{Number } T3)) \in \text{set evs}"$   
 $\langle \text{proof} \rangle$

**lemma** *A\_authenticates\_B\_r*:  
 $\parallel \text{Crypt } \text{servK} \ (\text{Number } T3) \in \text{parts } (\text{spies evs});$   
 $\text{Crypt } \text{authK} \ \{\{\text{Key } \text{servK}, \text{Agent } B, \text{Number } Ts, \text{servTicket}\}$   
 $\in \text{parts } (\text{spies evs});$   
 $\text{Crypt } (\text{shrK } A) \ \{\{\text{Key } \text{authK}, \text{Agent } Tgs, \text{Number } Ta, \text{authTicket}\}$   
 $\in \text{parts } (\text{spies evs});$   
 $\neg \text{expiredAK } Ta \text{ evs}; \neg \text{expiredSK } Ts \text{ evs};$   
 $A \notin \text{bad}; B \notin \text{bad}; \text{ evs} \in \text{kerbIV} \parallel$   
 $\implies \text{Says } B \ A \ (\text{Crypt } \text{servK} \ (\text{Number } T3)) \in \text{set evs}"$   
 $\langle \text{proof} \rangle$

### 6.12 Key distribution guarantees An agent knows a session key if he used it to issue a cipher. These guarantees also convey a stronger form of authentication - non-injective agreement on the session key

**lemma** *Kas\_Issues\_A*:  
 $\parallel \text{Says } Kas \ A \ (\text{Crypt } (\text{shrK } A) \ \{\{\text{Key } \text{authK}, \text{Peer}, Ta, \text{authTicket}\}) \in \text{set evs};$   
 $\text{ evs} \in \text{kerbIV} \parallel$   
 $\implies \text{Kas Issues } A \text{ with } (\text{Crypt } (\text{shrK } A) \ \{\{\text{Key } \text{authK}, \text{Peer}, Ta, \text{authTicket}\})$   
 $\text{ on evs}"$   
 $\langle \text{proof} \rangle$

**lemma** *A\_authenticates\_and\_keydist\_to\_Kas*:  
 $\parallel \text{Crypt } (\text{shrK } A) \ \{\{\text{Key } \text{authK}, \text{Peer}, Ta, \text{authTicket}\} \in \text{parts } (\text{spies evs});$   
 $A \notin \text{bad}; \text{ evs} \in \text{kerbIV} \parallel$   
 $\implies \text{Kas Issues } A \text{ with } (\text{Crypt } (\text{shrK } A) \ \{\{\text{Key } \text{authK}, \text{Peer}, Ta, \text{authTicket}\})$   
 $\text{ on evs}"$   
 $\langle \text{proof} \rangle$

**lemma** *honest\_never\_says\_newer\_timestamp\_in\_auth*:  
 $\parallel (CT \text{ evs}) \leq T; A \notin \text{bad}; \text{Number } T \in \text{parts } \{X\}; \text{ evs} \in \text{kerbIV} \parallel$   
 $\implies \forall B \ Y. \ \text{Says } A \ B \ \{Y, X\} \notin \text{set evs}"$

6.12 Key distribution guarantees An agent knows a session key if he used it to issue a cipher. These guarantees also

*<proof>*

**lemma** *honest\_never\_says\_current\_timestamp\_in\_auth:*  
 "[ (CT evs) = T; Number T ∈ parts {X}; evs ∈ kerbIV ]  
 ⇒ ∀ A B Y. A ∉ bad → Says A B {Y, X} ∉ set evs"  
*<proof>*

**lemma** *A\_trusts\_secure\_authenticator:*  
 "[ Crypt K {Agent A, Number T} ∈ parts (spies evs);  
 Key K ∉ analz (spies evs); evs ∈ kerbIV ]  
 ⇒ ∃ B X. Says A Tgs {X, Crypt K {Agent A, Number T}}, Agent B ∈ set evs  
 ∨  
 Says A B {X, Crypt K {Agent A, Number T}} ∈ set evs"  
*<proof>*

**lemma** *A\_Issues\_Tgs:*  
 "[ Says A Tgs {authTicket, Crypt authK {Agent A, Number T2}}, Agent B}  
 ∈ set evs;  
 Key authK ∉ analz (spies evs);  
 A ∉ bad; evs ∈ kerbIV ]  
 ⇒ A Issues Tgs with (Crypt authK {Agent A, Number T2}) on evs"  
*<proof>*

**lemma** *Tgs\_authenticates\_and\_keydist\_to\_A:*  
 "[ Crypt authK {Agent A, Number T2} ∈ parts (spies evs);  
 Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}  
 ∈ parts (spies evs);  
 Key authK ∉ analz (spies evs);  
 A ∉ bad; evs ∈ kerbIV ]  
 ⇒ A Issues Tgs with (Crypt authK {Agent A, Number T2}) on evs"  
*<proof>*

**lemma** *Tgs\_Issues\_A:*  
 "[ Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, servTicket  
 })  
 ∈ set evs;  
 Key authK ∉ analz (spies evs); evs ∈ kerbIV ]  
 ⇒ Tgs Issues A with  
 (Crypt authK {Key servK, Agent B, Number Ts, servTicket }) on evs"  
*<proof>*

**lemma** *A\_authenticates\_and\_keydist\_to\_Tgs:*  
 "[Crypt authK {Key servK, Agent B, Number Ts, servTicket} ∈ parts (spies evs);  
 Key authK ∉ analz (spies evs); B ≠ Tgs; evs ∈ kerbIV ]  
 ⇒ ∃ A. Tgs Issues A with  
 (Crypt authK {Key servK, Agent B, Number Ts, servTicket }) on evs"  
*<proof>*

**lemma** *B\_Issues\_A:*  
 "[ Says B A (Crypt servK (Number T3)) ∈ set evs;  
 Key servK ∉ analz (spies evs);  
 A ∉ bad; B ∉ bad; B ≠ Tgs; evs ∈ kerbIV ]

$\Rightarrow B \text{ Issues } A \text{ with } (\text{Crypt servK } (\text{Number } T3)) \text{ on evs''}$   
 $\langle \text{proof} \rangle$

**lemma** *B\_Issues\_A\_r*:

"[ Says B A (Crypt servK (Number T3))  $\in$  set evs;  
 Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}  
 $\in$  parts (spies evs);  
 Crypt authK {Key servK, Agent B, Number Ts, servTicket}  
 $\in$  parts (spies evs);  
 Crypt (shrK A) {Key authK, Agent Tgs, Number Ta, authTicket}  
 $\in$  parts (spies evs);  
 $\neg$  expiredSK Ts evs;  $\neg$  expiredAK Ta evs;  
 A  $\notin$  bad; B  $\notin$  bad; B  $\neq$  Tgs; evs  $\in$  kerbIV ]  
 $\Rightarrow B \text{ Issues } A \text{ with } (\text{Crypt servK } (\text{Number } T3)) \text{ on evs''}$   
 $\langle \text{proof} \rangle$

**lemma** *u\_B\_Issues\_A\_r*:

"[ Says B A (Crypt servK (Number T3))  $\in$  set evs;  
 Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}  
 $\in$  parts (spies evs);  
 $\neg$  expiredSK Ts evs;  
 A  $\notin$  bad; B  $\notin$  bad; B  $\neq$  Tgs; evs  $\in$  kerbIV ]  
 $\Rightarrow B \text{ Issues } A \text{ with } (\text{Crypt servK } (\text{Number } T3)) \text{ on evs''}$   
 $\langle \text{proof} \rangle$

**lemma** *A\_authenticates\_and\_keydist\_to\_B*:

"[ Crypt servK (Number T3)  $\in$  parts (spies evs);  
 Crypt authK {Key servK, Agent B, Number Ts, servTicket}  
 $\in$  parts (spies evs);  
 Crypt (shrK A) {Key authK, Agent Tgs, Number Ta, authTicket}  
 $\in$  parts (spies evs);  
 Key authK  $\notin$  analz (spies evs); Key servK  $\notin$  analz (spies evs);  
 A  $\notin$  bad; B  $\notin$  bad; B  $\neq$  Tgs; evs  $\in$  kerbIV ]  
 $\Rightarrow B \text{ Issues } A \text{ with } (\text{Crypt servK } (\text{Number } T3)) \text{ on evs''}$   
 $\langle \text{proof} \rangle$

**lemma** *A\_authenticates\_and\_keydist\_to\_B\_r*:

"[ Crypt servK (Number T3)  $\in$  parts (spies evs);  
 Crypt authK {Key servK, Agent B, Number Ts, servTicket}  
 $\in$  parts (spies evs);  
 Crypt (shrK A) {Key authK, Agent Tgs, Number Ta, authTicket}  
 $\in$  parts (spies evs);  
 $\neg$  expiredAK Ta evs;  $\neg$  expiredSK Ts evs;  
 A  $\notin$  bad; B  $\notin$  bad; B  $\neq$  Tgs; evs  $\in$  kerbIV ]  
 $\Rightarrow B \text{ Issues } A \text{ with } (\text{Crypt servK } (\text{Number } T3)) \text{ on evs''}$   
 $\langle \text{proof} \rangle$

**lemma** *A\_Issues\_B*:

"[ Says A B {servTicket, Crypt servK {Agent A, Number T3}}  
 $\in$  set evs;  
 Key servK  $\notin$  analz (spies evs);  
 B  $\neq$  Tgs; A  $\notin$  bad; B  $\notin$  bad; evs  $\in$  kerbIV ]  
 $\Rightarrow A \text{ Issues } B \text{ with } (\text{Crypt servK } \{ \text{Agent } A, \text{Number } T3 \}) \text{ on evs''}$

6.12 Key distribution guarantees An agent knows a session key if he used it to issue a cipher. These guarantees also

*<proof>*

**lemma** *A\_Issues\_B\_r:*

```
"[[ Says A B {servTicket, Crypt servK {Agent A, Number T3}}
    ∈ set evs;
    Crypt (shrK A) {Key authK, Agent Tgs, Number Ta, authTicket}
    ∈ parts (spies evs);
    Crypt authK {Key servK, Agent B, Number Ts, servTicket}
    ∈ parts (spies evs);
    ¬ expiredAK Ta evs; ¬ expiredSK Ts evs;
    B ≠ Tgs; A ∉ bad; B ∉ bad; evs ∈ kerbIV ]]
⇒ A Issues B with (Crypt servK {Agent A, Number T3}) on evs"
<proof>
```

**lemma** *B\_authenticates\_and\_keydist\_to\_A:*

```
"[[ Crypt servK {Agent A, Number T3} ∈ parts (spies evs);
    Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}
    ∈ parts (spies evs);
    Key servK ∉ analz (spies evs);
    B ≠ Tgs; A ∉ bad; B ∉ bad; evs ∈ kerbIV ]]
⇒ A Issues B with (Crypt servK {Agent A, Number T3}) on evs"
<proof>
```

**lemma** *B\_authenticates\_and\_keydist\_to\_A\_r:*

```
"[[ Crypt servK {Agent A, Number T3} ∈ parts (spies evs);
    Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}
    ∈ parts (spies evs);
    Crypt authK {Key servK, Agent B, Number Ts, servTicket}
    ∈ parts (spies evs);
    Crypt (shrK A) {Key authK, Agent Tgs, Number Ta, authTicket}
    ∈ parts (spies evs);
    ¬ expiredSK Ts evs; ¬ expiredAK Ta evs;
    B ≠ Tgs; A ∉ bad; B ∉ bad; evs ∈ kerbIV ]]
⇒ A Issues B with (Crypt servK {Agent A, Number T3}) on evs"
<proof>
```

*u\_B\_authenticates\_and\_keydist\_to\_A* would be the same as *B\_authenticates\_and\_keydist\_to\_A* because the *servK* confidentiality assumption is yet unrelaxed

**lemma** *u\_B\_authenticates\_and\_keydist\_to\_A\_r:*

```
"[[ Crypt servK {Agent A, Number T3} ∈ parts (spies evs);
    Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}
    ∈ parts (spies evs);
    ¬ expiredSK Ts evs;
    B ≠ Tgs; A ∉ bad; B ∉ bad; evs ∈ kerbIV ]]
⇒ A Issues B with (Crypt servK {Agent A, Number T3}) on evs"
<proof>
```

**end**

## 7 The Kerberos Protocol, Version IV

**theory** *KerberosIV\_Gets* **imports** *Public* **begin**

The "u" prefix indicates theorems referring to an updated version of the protocol. The "r" suffix indicates theorems where the confidentiality assumptions are relaxed by the corresponding arguments.

**abbreviation**

*Kas* :: *agent* **where** "*Kas* == *Server*"

**abbreviation**

*Tgs* :: *agent* **where** "*Tgs* == *Friend 0*"

**axioms**

*Tgs\_not\_bad* [iff]: "*Tgs*  $\notin$  *bad*"

— *Tgs* is secure — we already know that *Kas* is secure

**constdefs**

*authKeys* :: "*event list* => *key set*"

"*authKeys evs* == {*authK*.  $\exists A$  *Peer Ta*. *Says Kas A*

(*Crypt* (*shrK A*)  $\{\text{Key } \text{authK}, \text{Agent Peer}, \text{Number Ta},$

(*Crypt* (*shrK Peer*)  $\{\text{Agent A}, \text{Agent Peer}, \text{Key authK}, \text{Number$

*Ta*  $\}\}$ )

$\}\} \in \text{set } \text{evs}\}$ "

*Unique* :: "[*event*, *event list*] => *bool*" ("*Unique \_ on \_*")

"*Unique ev on evs* ==

*ev*  $\notin$  *set* (*tl* (*dropWhile* ( $\% z. z \neq ev$ ) *evs*))"

**consts**

*authKlife* :: *nat*

*servKlife* :: *nat*

*authlife* :: *nat*

*replylife* :: *nat*

**specification** (*authKlife*)

*authKlife\_LB* [iff]: "*2*  $\leq$  *authKlife*"

$\langle \text{proof} \rangle$

**specification** (*servKlife*)

*servKlife\_LB* [iff]: "*2* + *authKlife*  $\leq$  *servKlife*"

$\langle \text{proof} \rangle$

**specification (authlife)**

```
authlife_LB [iff]: "Suc 0 ≤ authlife"
  ⟨proof⟩
```

**specification (replylife)**

```
replylife_LB [iff]: "Suc 0 ≤ replylife"
  ⟨proof⟩
```

**abbreviation**

```
CT :: "event list=>nat" where
  "CT == length"
```

**abbreviation**

```
expiredAK :: "[nat, event list] => bool" where
  "expiredAK Ta evs == authKlife + Ta < CT evs"
```

**abbreviation**

```
expiredSK :: "[nat, event list] => bool" where
  "expiredSK Ts evs == servKlife + Ts < CT evs"
```

**abbreviation**

```
expiredA :: "[nat, event list] => bool" where
  "expiredA T evs == authlife + T < CT evs"
```

**abbreviation**

```
valid :: "[nat, nat] => bool" ("valid _ wrt _") where
  "valid T1 wrt T2 == T1 ≤ replylife + T2"
```

**constdefs**

```
AKcryptSK :: "[key, key, event list] => bool"
  "AKcryptSK authK servK evs ==
    ∃ A B Ts.
      Says Tgs A (Crypt authK
        {Key servK, Agent B, Number Ts,
         Crypt (shrK B) {Agent A, Agent B, Key servK, Number
Ts}})
    ∈ set evs"
```

**inductive\_set "kerbIV\_gets" :: "event list set"**

**where**

```
Nil: "[] ∈ kerbIV_gets"
```

```
/ Fake: "[ evsf ∈ kerbIV_gets; X ∈ synth (analz (spies evsf)) ]
  ⇒ Says Spy B X # evsf ∈ kerbIV_gets"
```

```
/ Reception: "[ evsr ∈ kerbIV_gets; Says A B X ∈ set evsr ]
  ⇒ Gets B X # evsr ∈ kerbIV_gets"
```

```

/ K1: "[ evs1 ∈ kerbIV_gets ]
      ⇒ Says A Kas {Agent A, Agent Tgs, Number (CT evs1)} # evs1
      ∈ kerbIV_gets"

/ K2: "[ evs2 ∈ kerbIV_gets; Key authK ∉ used evs2; authK ∈ symKeys;
      Gets Kas {Agent A, Agent Tgs, Number T1} ∈ set evs2 ]
      ⇒ Says Kas A
      (Crypt (shrK A) {Key authK, Agent Tgs, Number (CT evs2),
      (Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK,
      Number (CT evs2)} )} ) # evs2 ∈ kerbIV_gets"

/ K3: "[ evs3 ∈ kerbIV_gets;
      Says A Kas {Agent A, Agent Tgs, Number T1} ∈ set evs3;
      Gets A (Crypt (shrK A) {Key authK, Agent Tgs, Number Ta,
      authTicket}) ∈ set evs3;
      valid Ta wrt T1
      ]
      ⇒ Says A Tgs {authTicket,
      (Crypt authK {Agent A, Number (CT evs3)} ),
      Agent B} # evs3 ∈ kerbIV_gets"

/ K4: "[ evs4 ∈ kerbIV_gets; Key servK ∉ used evs4; servK ∈ symKeys;
      B ≠ Tgs; authK ∈ symKeys;
      Gets Tgs {
      (Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK,
      Number Ta} ),
      (Crypt authK {Agent A, Number T2} ), Agent B}
      ∈ set evs4;
      ¬ expiredAK Ta evs4;
      ¬ expiredA T2 evs4;
      servKlife + (CT evs4) ≤ authKlife + Ta
      ]
      ⇒ Says Tgs A
      (Crypt authK {Key servK, Agent B, Number (CT evs4),
      Crypt (shrK B) {Agent A, Agent B, Key servK,
      Number (CT evs4)} } )
      # evs4 ∈ kerbIV_gets"

```



```

/ K5: "[ evs5 ∈ kerbIV_gets; authK ∈ symKeys; servK ∈ symKeys;
      Says A Tgs
        {authTicket, Crypt authK {Agent A, Number T2},
         Agent B}
      ∈ set evs5;
      Gets A
        (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
      ∈ set evs5;
      valid Ts wrt T2 ]
⇒ Says A B {servTicket,
            Crypt servK {Agent A, Number (CT evs5)} }
  # evs5 ∈ kerbIV_gets"

/ K6: "[ evs6 ∈ kerbIV_gets;
      Gets B {
        (Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}),
        (Crypt servK {Agent A, Number T3}) }
      ∈ set evs6;
      ¬ expiredSK Ts evs6;
      ¬ expiredA T3 evs6
    ]
⇒ Says B A (Crypt servK (Number T3))
  # evs6 ∈ kerbIV_gets"

/ Ops1: "[ evs01 ∈ kerbIV_gets; A ≠ Spy;
      Says Kas A
        (Crypt (shrK A) {Key authK, Agent Tgs, Number Ta,
                          authTicket}) ∈ set evs01;
      expiredAK Ta evs01 ]
⇒ Says A Spy {Agent A, Agent Tgs, Number Ta, Key authK}
  # evs01 ∈ kerbIV_gets"

/ Ops2: "[ evs02 ∈ kerbIV_gets; A ≠ Spy;
      Says Tgs A
        (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
      ∈ set evs02;
      expiredSK Ts evs02 ]
⇒ Says A Spy {Agent A, Agent B, Number Ts, Key servK}
  # evs02 ∈ kerbIV_gets"

```

```

declare Says_imp_knows_Spy [THEN parts.Inj, dest]
declare parts.Body [dest]
declare analz_into_parts [dest]
declare Fake_parts_insert_in_Un [dest]

```

### 7.1 Lemmas about reception event

```

lemma Gets_imp_Says :
  "[[ Gets B X ∈ set evs; evs ∈ kerbIV_gets ]] ⇒ ∃ A. Says A B X ∈ set evs"
⟨proof⟩

```

```

lemma Gets_imp_knows_Spy:
  "[[ Gets B X ∈ set evs; evs ∈ kerbIV_gets ]] ⇒ X ∈ knows Spy evs"
⟨proof⟩

```

```

declare Gets_imp_knows_Spy [THEN parts.Inj, dest]

```

```

lemma Gets_imp_knows:
  "[[ Gets B X ∈ set evs; evs ∈ kerbIV_gets ]] ⇒ X ∈ knows B evs"
⟨proof⟩

```

### 7.2 Lemmas about authKeys

```

lemma authKeys_empty: "authKeys [] = {}"
⟨proof⟩

```

```

lemma authKeys_not_insert:
  "(∀ A Ta akey Peer.
    ev ≠ Says Kas A (Crypt (shrK A) {akey, Agent Peer, Ta,
      (Crypt (shrK Peer) {Agent A, Agent Peer, akey, Ta})}))
    ⇒ authKeys (ev # evs) = authKeys evs"
⟨proof⟩

```

```

lemma authKeys_insert:
  "authKeys
    (Says Kas A (Crypt (shrK A) {Key K, Agent Peer, Number Ta,
      (Crypt (shrK Peer) {Agent A, Agent Peer, Key K, Number Ta})})) # evs)
    = insert K (authKeys evs)"
⟨proof⟩

```

```

lemma authKeys_simp:
  "K ∈ authKeys
    (Says Kas A (Crypt (shrK A) {Key K', Agent Peer, Number Ta,
      (Crypt (shrK Peer) {Agent A, Agent Peer, Key K', Number Ta})})) # evs)
    ⇒ K = K' | K ∈ authKeys evs"
⟨proof⟩

```

```

lemma authKeysI:
  "Says Kas A (Crypt (shrK A) {Key K, Agent Tgs, Number Ta,
    (Crypt (shrK Tgs) {Agent A, Agent Tgs, Key K, Number Ta})}) ∈ set evs

```

$\implies K \in \text{authKeys evs}$   
 $\langle \text{proof} \rangle$

**lemma** *authKeys\_used*: " $K \in \text{authKeys evs} \implies \text{Key } K \in \text{used evs}$ "  
 $\langle \text{proof} \rangle$

### 7.3 Forwarding Lemmas

**lemma** *Says\_ticket\_parts*:  
 " $\text{Says } S \ A \ (\text{Crypt } K \ \{\text{SesKey}, B, \text{TimeStamp}, \text{Ticket}\}) \in \text{set evs}$   
 $\implies \text{Ticket} \in \text{parts (spies evs)}$ "  
 $\langle \text{proof} \rangle$

**lemma** *Gets\_ticket\_parts*:  
 " $\text{Gets } A \ (\text{Crypt } K \ \{\text{SesKey}, \text{Peer}, Ta, \text{Ticket}\}) \in \text{set evs}; \text{ evs} \in \text{kerbIV\_gets}$   
 $\implies \text{Ticket} \in \text{parts (spies evs)}$ "  
 $\langle \text{proof} \rangle$

**lemma** *Ops\_range\_spies1*:  
 " $\text{Says } Kas \ A \ (\text{Crypt } \text{KeyA} \ \{\text{Key authK}, \text{Peer}, Ta, \text{authTicket}\})$   
 $\in \text{set evs};$   
 $\text{ evs} \in \text{kerbIV\_gets} \implies \text{authK} \notin \text{range shrK} \ \& \ \text{authK} \in \text{symKeys}$ "  
 $\langle \text{proof} \rangle$

**lemma** *Ops\_range\_spies2*:  
 " $\text{Says } Tgs \ A \ (\text{Crypt } \text{authK} \ \{\text{Key servK}, \text{Agent } B, Ts, \text{servTicket}\})$   
 $\in \text{set evs};$   
 $\text{ evs} \in \text{kerbIV\_gets} \implies \text{servK} \notin \text{range shrK} \ \& \ \text{servK} \in \text{symKeys}$ "  
 $\langle \text{proof} \rangle$

**lemma** *Spy\_see\_shrK [simp]*:  
 " $\text{ evs} \in \text{kerbIV\_gets} \implies (\text{Key (shrK } A) \in \text{parts (spies evs)}) = (A \in \text{bad})$ "  
 $\langle \text{proof} \rangle$

**lemma** *Spy\_analz\_shrK [simp]*:  
 " $\text{ evs} \in \text{kerbIV\_gets} \implies (\text{Key (shrK } A) \in \text{analz (spies evs)}) = (A \in \text{bad})$ "  
 $\langle \text{proof} \rangle$

**lemma** *Spy\_see\_shrK\_D [dest!]*:  
 " $\text{Key (shrK } A) \in \text{parts (spies evs)}; \text{ evs} \in \text{kerbIV\_gets} \implies A:\text{bad}$ "  
 $\langle \text{proof} \rangle$   
**lemmas** *Spy\_analz\_shrK\_D* = *analz\_subset\_parts [THEN subsetD, THEN Spy\_see\_shrK\_D, dest!]*

Nobody can have used non-existent keys!

**lemma** *new\_keys\_not\_used [simp]*:  
 " $\text{Key } K \notin \text{used evs}; K \in \text{symKeys}; \text{ evs} \in \text{kerbIV\_gets}$   
 $\implies K \notin \text{keysFor (parts (spies evs))}$ "  
 $\langle \text{proof} \rangle$

**lemma** *new\_keys\_not\_analz*:  
 "[[*evs* ∈ *kerbIV\_gets*; *K* ∈ *symKeys*; Key *K* ∉ *used evs*]]  
 ⇒ *K* ∉ *keysFor (analz (spies evs))*"  
 <proof>

## 7.4 Regularity Lemmas

These concern the form of items passed in messages

Describes the form of all components sent by Kas

**lemma** *Says\_Kas\_message\_form*:  
 "[[ *Says Kas A (Crypt K {Key authK, Agent Peer, Number Ta, authTicket})*  
   ∈ *set evs*;  
   *evs* ∈ *kerbIV\_gets* ] ] ⇒  
*K* = *shrK A* & *Peer* = *Tgs* &  
*authK* ∉ *range shrK* & *authK* ∈ *authKeys evs* & *authK* ∈ *symKeys* &  
*authTicket* = (*Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}*) ] ]"  
 <proof>

**lemma** *SesKey\_is\_session\_key*:  
 "[[ *Crypt (shrK Tgs\_B) {Agent A, Agent Tgs\_B, Key SesKey, Number T}*  
   ∈ *parts (spies evs)*; *Tgs\_B* ∉ *bad*;  
   *evs* ∈ *kerbIV\_gets* ] ]  
 ⇒ *SesKey* ∉ *range shrK*"  
 <proof>

**lemma** *authTicket\_authentic*:  
 "[[ *Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}*  
   ∈ *parts (spies evs)*;  
   *evs* ∈ *kerbIV\_gets* ] ]  
 ⇒ *Says Kas A (Crypt (shrK A) {Key authK, Agent Tgs, Number Ta,  
   Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}})*  
   ∈ *set evs*"  
 <proof>

**lemma** *authTicket\_crypt\_authK*:  
 "[[ *Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}*  
   ∈ *parts (spies evs)*;  
   *evs* ∈ *kerbIV\_gets* ] ]  
 ⇒ *authK* ∈ *authKeys evs*"  
 <proof>

**lemma** *Says\_Tgs\_message\_form*:  
 "[[ *Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, servTicket})*  
   ∈ *set evs*;  
   *evs* ∈ *kerbIV\_gets* ] ]  
 ⇒ *B* ≠ *Tgs* &  
*authK* ∉ *range shrK* & *authK* ∈ *authKeys evs* & *authK* ∈ *symKeys* &  
*servK* ∉ *range shrK* & *servK* ∉ *authKeys evs* & *servK* ∈ *symKeys* &  
*servTicket* = (*Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}*) ] ]"  
 <proof>

**lemma** *authTicket\_form*:

```
"[[ Crypt (shrK A) {Key authK, Agent Tgs, Ta, authTicket}
  ∈ parts (spies evs);
  A ∉ bad;
  evs ∈ kerbIV_gets ]]
⇒ authK ∉ range shrK & authK ∈ symKeys &
  authTicket = Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Ta}"
⟨proof⟩
```

This form holds also over an *authTicket*, but is not needed below.

**lemma** *servTicket\_form*:

```
"[[ Crypt authK {Key servK, Agent B, Ts, servTicket}
  ∈ parts (spies evs);
  Key authK ∉ analz (spies evs);
  evs ∈ kerbIV_gets ]]
⇒ servK ∉ range shrK & servK ∈ symKeys &
  (∃ A. servTicket = Crypt (shrK B) {Agent A, Agent B, Key servK, Ts})"
⟨proof⟩
```

Essentially the same as *authTicket\_form*

**lemma** *Says\_kas\_message\_form*:

```
"[[ Gets A (Crypt (shrK A)
  {Key authK, Agent Tgs, Ta, authTicket}) ∈ set evs;
  evs ∈ kerbIV_gets ]]
⇒ authK ∉ range shrK & authK ∈ symKeys &
  authTicket =
    Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Ta}
  / authTicket ∈ analz (spies evs)"
⟨proof⟩
```

**lemma** *Says\_tgs\_message\_form*:

```
"[[ Gets A (Crypt authK {Key servK, Agent B, Ts, servTicket})
  ∈ set evs; authK ∈ symKeys;
  evs ∈ kerbIV_gets ]]
⇒ servK ∉ range shrK &
  (∃ A. servTicket =
    Crypt (shrK B) {Agent A, Agent B, Key servK, Ts})
  / servTicket ∈ analz (spies evs)"
⟨proof⟩
```

## 7.5 Authenticity theorems: confirm origin of sensitive messages

**lemma** *authK\_authentic*:

```
"[[ Crypt (shrK A) {Key authK, Peer, Ta, authTicket}
  ∈ parts (spies evs);
  A ∉ bad; evs ∈ kerbIV_gets ]]
⇒ Says Kas A (Crypt (shrK A) {Key authK, Peer, Ta, authTicket})
  ∈ set evs"
⟨proof⟩
```

If a certain encrypted message appears then it originated with Tgs

**lemma** *servK\_authentic*:

```

    "[ Crypt authK {Key servK, Agent B, Number Ts, servTicket}
      ∈ parts (spies evs);
      Key authK ∉ analz (spies evs);
      authK ∉ range shrK;
      evs ∈ kerbIV_gets ]
  ⇒ ∃ A. Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
    ∈ set evs"
<proof>

```

```

lemma servK_authentic_bis:
  "[ Crypt authK {Key servK, Agent B, Number Ts, servTicket}
    ∈ parts (spies evs);
    Key authK ∉ analz (spies evs);
    B ≠ Tgs;
    evs ∈ kerbIV_gets ]
  ⇒ ∃ A. Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
    ∈ set evs"
<proof>

```

Authenticity of servK for B

```

lemma servTicket_authentic_Tgs:
  "[ Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}
    ∈ parts (spies evs); B ≠ Tgs; B ∉ bad;
    evs ∈ kerbIV_gets ]
  ⇒ ∃ authK.
    Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts,
      Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}})
    ∈ set evs"
<proof>

```

Anticipated here from next subsection

```

lemma K4_imp_K2:
  "[ Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
    ∈ set evs; evs ∈ kerbIV_gets ]
  ⇒ ∃ Ta. Says Kas A
    (Crypt (shrK A)
      {Key authK, Agent Tgs, Number Ta,
      Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}})
    ∈ set evs"
<proof>

```

Anticipated here from next subsection

```

lemma u_K4_imp_K2:
  "[ Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
    ∈ set evs; evs ∈ kerbIV_gets ]
  ⇒ ∃ Ta. (Says Kas A (Crypt (shrK A) {Key authK, Agent Tgs, Number Ta,
    Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}})
    ∈ set evs
    & servKlife + Ts ≤ authKlife + Ta)"
<proof>

```

```

lemma servTicket_authentic_Kas:
  "[ Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}

```

```

      ∈ parts (spies evs); B ≠ Tgs; B ∉ bad;
      evs ∈ kerbIV_gets ]
⇒ ∃ authK Ta.
  Says Kas A
    (Crypt (shrK A) {Key authK, Agent Tgs, Number Ta,
      Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}})
    ∈ set evs"
⟨proof⟩

```

**lemma** u\_servTicket\_authentic\_Kas:

```

  "[ Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}
    ∈ parts (spies evs); B ≠ Tgs; B ∉ bad;
    evs ∈ kerbIV_gets ]
⇒ ∃ authK Ta. Says Kas A (Crypt (shrK A) {Key authK, Agent Tgs, Number Ta,
  Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}})
  ∈ set evs
  & servKlife + Ts ≤ authKlife + Ta"
⟨proof⟩

```

**lemma** servTicket\_authentic:

```

  "[ Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}
    ∈ parts (spies evs); B ≠ Tgs; B ∉ bad;
    evs ∈ kerbIV_gets ]
⇒ ∃ Ta authK.
  Says Kas A (Crypt (shrK A) {Key authK, Agent Tgs, Number Ta,
    Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number
Ta}})
    ∈ set evs
    & Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts,
      Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}})
    ∈ set evs"
⟨proof⟩

```

**lemma** u\_servTicket\_authentic:

```

  "[ Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}
    ∈ parts (spies evs); B ≠ Tgs; B ∉ bad;
    evs ∈ kerbIV_gets ]
⇒ ∃ Ta authK.
  (Says Kas A (Crypt (shrK A) {Key authK, Agent Tgs, Number Ta,
    Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number
Ta}})
    ∈ set evs
    & Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts,
      Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}})
    ∈ set evs
    & servKlife + Ts ≤ authKlife + Ta)"
⟨proof⟩

```

**lemma** u\_NotexpiredSK\_NotexpiredAK:

```

  "[ ¬ expiredSK Ts evs; servKlife + Ts ≤ authKlife + Ta ]
⇒ ¬ expiredAK Ta evs"
⟨proof⟩

```

## 7.6 Reliability: friendly agents send something if something else happened

**lemma** *K3\_imp\_K2*:

```
"[[ Says A Tgs
    {authTicket, Crypt authK {Agent A, Number T2}}, Agent B}
   ∈ set evs;
   A ∉ bad; evs ∈ kerbIV_gets ]
⇒ ∃ Ta. Says Kas A (Crypt (shrK A)
    {Key authK, Agent Tgs, Number Ta, authTicket})
   ∈ set evs"
```

*<proof>*

Anticipated here from next subsection. An authK is encrypted by one and only one Shared key. A servK is encrypted by one and only one authK.

**lemma** *Key\_unique\_SesKey*:

```
"[[ Crypt K {Key SesKey, Agent B, T, Ticket}
   ∈ parts (spies evs);
   Crypt K' {Key SesKey, Agent B', T', Ticket'}
   ∈ parts (spies evs); Key SesKey ∉ analz (spies evs);
   evs ∈ kerbIV_gets ]
⇒ K=K' & B=B' & T=T' & Ticket=Ticket'"
```

*<proof>*

**lemma** *Tgs\_authenticates\_A*:

```
"[[ Crypt authK {Agent A, Number T2} ∈ parts (spies evs);
   Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}
   ∈ parts (spies evs);
   Key authK ∉ analz (spies evs); A ∉ bad; evs ∈ kerbIV_gets ]
⇒ ∃ B. Says A Tgs {
   Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta},
   Crypt authK {Agent A, Number T2}, Agent B } ∈ set evs"
```

*<proof>*

**lemma** *Says\_K5*:

```
"[[ Crypt servK {Agent A, Number T3} ∈ parts (spies evs);
   Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts,
   servTicket}) ∈ set evs;
   Key servK ∉ analz (spies evs);
   A ∉ bad; B ∉ bad; evs ∈ kerbIV_gets ]
⇒ Says A B {servTicket, Crypt servK {Agent A, Number T3}} ∈ set evs"
```

*<proof>*

Anticipated here from next subsection

**lemma** *unique\_CryptKey*:

```
"[[ Crypt (shrK B) {Agent A, Agent B, Key SesKey, T}
   ∈ parts (spies evs);
   Crypt (shrK B') {Agent A', Agent B', Key SesKey, T'}
   ∈ parts (spies evs); Key SesKey ∉ analz (spies evs);
   evs ∈ kerbIV_gets ]
⇒ A=A' & B=B' & T=T'"
```

*<proof>*

**lemma** *Says\_K6*:



```

"[[ Crypt servK (Number T3) ∈ parts (spies evs);
   Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts,
                           servTicket}) ∈ set evs;
   Key servK ∉ analz (spies evs);
   A ∉ bad; B ∉ bad; evs ∈ kerbIV_gets ]]
⇒ Says B A (Crypt servK (Number T3)) ∈ set evs"
<proof>

```

Needs a unicity theorem, hence moved here

**lemma** *servK\_authentic\_ter*:

```

"[[ Says Kas A
   (Crypt (shrK A) {Key authK, Agent Tgs, Number Ta, authTicket}) ∈ set
evs;
   Crypt authK {Key servK, Agent B, Number Ts, servTicket}
   ∈ parts (spies evs);
   Key authK ∉ analz (spies evs);
   evs ∈ kerbIV_gets ]]
⇒ Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
   ∈ set evs"
<proof>

```

## 7.7 Unicity Theorems

The session key, if secure, uniquely identifies the Ticket whether authTicket or servTicket. As a matter of fact, one can read also Tgs in the place of B.

**lemma** *unique\_authKeys*:

```

"[[ Says Kas A
   (Crypt Ka {Key authK, Agent Tgs, Ta, X}) ∈ set evs;
   Says Kas A'
   (Crypt Ka' {Key authK, Agent Tgs, Ta', X'}) ∈ set evs;
   evs ∈ kerbIV_gets ]] ⇒ A=A' & Ka=Ka' & Ta=Ta' & X=X'"
<proof>

```

servK uniquely identifies the message from Tgs

**lemma** *unique\_servKeys*:

```

"[[ Says Tgs A
   (Crypt K {Key servK, Agent B, Ts, X}) ∈ set evs;
   Says Tgs A'
   (Crypt K' {Key servK, Agent B', Ts', X'}) ∈ set evs;
   evs ∈ kerbIV_gets ]] ⇒ A=A' & B=B' & K=K' & Ts=Ts' & X=X'"
<proof>

```

Revised unicity theorems

**lemma** *Kas\_Unique*:

```

"[[ Says Kas A
   (Crypt Ka {Key authK, Agent Tgs, Ta, authTicket}) ∈ set evs;
   evs ∈ kerbIV_gets ]] ⇒
   Unique (Says Kas A (Crypt Ka {Key authK, Agent Tgs, Ta, authTicket}))
   on evs"
<proof>

```

**lemma** *Tgs\_Unique*:

```

"[[ Says Tgs A
  (Crypt authK {Key servK, Agent B, Ts, servTicket}) ∈ set evs;
  evs ∈ kerbIV_gets ]] ⇒
Unique (Says Tgs A (Crypt authK {Key servK, Agent B, Ts, servTicket}))
on evs"
⟨proof⟩

```

## 7.8 Lemmas About the Predicate *AKcryptSK*

```

lemma not_AKcryptSK_Nil [iff]: "¬ AKcryptSK authK servK []"
⟨proof⟩

```

```

lemma AKcryptSKI:
"[[ Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, X }) ∈ set evs;
  evs ∈ kerbIV_gets ]] ⇒ AKcryptSK authK servK evs"
⟨proof⟩

```

```

lemma AKcryptSK_Says [simp]:
  "AKcryptSK authK servK (Says S A X # evs) =
    (Tgs = S &
     (∃ B Ts. X = Crypt authK
       {Key servK, Agent B, Number Ts,
        Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}}
     )
    / AKcryptSK authK servK evs)"
⟨proof⟩

```

```

lemma Auth_fresh_not_AKcryptSK:
  "[[ Key authK ∉ used evs; evs ∈ kerbIV_gets ]]
   ⇒ ¬ AKcryptSK authK servK evs"
⟨proof⟩

```

```

lemma Serv_fresh_not_AKcryptSK:
  "Key servK ∉ used evs ⇒ ¬ AKcryptSK authK servK evs"
⟨proof⟩

```

```

lemma authK_not_AKcryptSK:
  "[[ Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, tk}
    ∈ parts (spies evs); evs ∈ kerbIV_gets ]]
   ⇒ ¬ AKcryptSK K authK evs"
⟨proof⟩

```

A secure serverkey cannot have been used to encrypt others

```

lemma servK_not_AKcryptSK:
  "[[ Crypt (shrK B) {Agent A, Agent B, Key SK, Number Ts} ∈ parts (spies evs);
    Key SK ∉ analz (spies evs); SK ∈ symKeys;
    B ≠ Tgs; evs ∈ kerbIV_gets ]]
   ⇒ ¬ AKcryptSK SK K evs"
⟨proof⟩

```

Long term keys are not issued as servKeys

```

lemma shrK_not_AKcryptSK:

```

"*evs* ∈ *kerbIV\_gets* ⇒ ¬ *AKcryptSK* *K* (*shrK* *A*) *evs*"  
 ⟨*proof*⟩

The Tgs message associates *servK* with *authK* and therefore not with any other key *authK*.

**lemma** *Says\_Tgs\_AKcryptSK*:  
 "[ *Says Tgs A* (*Crypt* *authK* {*Key servK*, *Agent B*, *Number Ts*, *X*})  
   ∈ *set evs*;  
   *authK'* ≠ *authK*; *evs* ∈ *kerbIV\_gets* ]  
 ⇒ ¬ *AKcryptSK* *authK'* *servK* *evs*"  
 ⟨*proof*⟩

Equivalently

**lemma** *not\_different\_AKcryptSK*:  
 "[ *AKcryptSK* *authK* *servK* *evs*;  
   *authK'* ≠ *authK*; *evs* ∈ *kerbIV\_gets* ]  
 ⇒ ¬ *AKcryptSK* *authK'* *servK* *evs* ∧ *servK* ∈ *symKeys*"  
 ⟨*proof*⟩

**lemma** *AKcryptSK\_not\_AKcryptSK*:  
 "[ *AKcryptSK* *authK* *servK* *evs*; *evs* ∈ *kerbIV\_gets* ]  
 ⇒ ¬ *AKcryptSK* *servK* *K* *evs*"  
 ⟨*proof*⟩

The only session keys that can be found with the help of session keys are those sent by Tgs in step K4.

We take some pains to express the property as a logical equivalence so that the simplifier can apply it.

**lemma** *Key\_analz\_image\_Key\_lemma*:  
 "*P* → (*Key* *K* ∈ *analz* (*Key*'*KK* *Un* *H*)) → (*K*:*KK* | *Key* *K* ∈ *analz* *H*)  
 ⇒  
 "*P* → (*Key* *K* ∈ *analz* (*Key*'*KK* *Un* *H*)) = (*K*:*KK* | *Key* *K* ∈ *analz* *H*)"

⟨*proof*⟩

**lemma** *AKcryptSK\_analz\_insert*:  
 "[ *AKcryptSK* *K* *K'* *evs*; *K* ∈ *symKeys*; *evs* ∈ *kerbIV\_gets* ]  
 ⇒ *Key* *K'* ∈ *analz* (*insert* (*Key* *K*) (*spies* *evs*))"

⟨*proof*⟩

**lemma** *authKeys\_are\_not\_AKcryptSK*:  
 "[ *K* ∈ *authKeys* *evs* *Un* *range shrK*; *evs* ∈ *kerbIV\_gets* ]  
 ⇒ ∀ *SK*. ¬ *AKcryptSK* *SK* *K* *evs* ∧ *K* ∈ *symKeys*"

⟨*proof*⟩

**lemma** *not\_authKeys\_not\_AKcryptSK*:  
 "[ *K* ∉ *authKeys* *evs*;  
   *K* ∉ *range shrK*; *evs* ∈ *kerbIV\_gets* ]  
 ⇒ ∀ *SK*. ¬ *AKcryptSK* *K* *SK* *evs*"

⟨*proof*⟩

## 7.9 Secrecy Theorems

For the Oops2 case of the next theorem

**lemma** *Oops2\_not\_AKcryptSK*:  
 "[[ *evs* ∈ *kerbIV\_gets*;  
   *Says Tgs A* (*Crypt authK*  
     {Key *servK*, *Agent B*, *Number Ts*, *servTicket*})  
   ∈ *set evs* ]]  
 ⇒ ¬ *AKcryptSK servK SK evs*"  
*<proof>*

Big simplification law for keys SK that are not crypted by keys in KK It helps prove three, otherwise hard, facts about keys. These facts are exploited as simplification laws for *analz*, and also "limit the damage" in case of loss of a key to the spy. See ESORICS98.

**lemma** *Key\_analz\_image\_Key* [*rule\_format* (*no\_asm*)]:  
 "*evs* ∈ *kerbIV\_gets* ⇒  
 (∀ *SK KK*. *SK* ∈ *symKeys* & *KK* ≤ -(*range shrK*) →  
 (∀ *K* ∈ *KK*. ¬ *AKcryptSK K SK evs*) →  
 (*Key SK* ∈ *analz* (*Key'KK Un* (*spies evs*)))) =  
 (*SK* ∈ *KK* | *Key SK* ∈ *analz* (*spies evs*)))"  
*<proof>*

First simplification law for *analz*: no session keys encrypt authentication keys or shared keys.

**lemma** *analz\_insert\_freshK1*:  
 "[[ *evs* ∈ *kerbIV\_gets*; *K* ∈ *authKeys evs Un range shrK*;  
   *SesKey* ∉ *range shrK* ]]  
 ⇒ (*Key K* ∈ *analz* (*insert* (*Key SesKey*) (*spies evs*))) =  
 (*K* = *SesKey* | *Key K* ∈ *analz* (*spies evs*)))"  
*<proof>*

Second simplification law for *analz*: no service keys encrypt any other keys.

**lemma** *analz\_insert\_freshK2*:  
 "[[ *evs* ∈ *kerbIV\_gets*; *servK* ∉ (*authKeys evs*); *servK* ∉ *range shrK*;  
   *K* ∈ *symKeys* ]]  
 ⇒ (*Key K* ∈ *analz* (*insert* (*Key servK*) (*spies evs*))) =  
 (*K* = *servK* | *Key K* ∈ *analz* (*spies evs*)))"  
*<proof>*

Third simplification law for *analz*: only one authentication key encrypts a certain service key.

**lemma** *analz\_insert\_freshK3*:  
 "[[ *AKcryptSK authK servK evs*;  
   *authK'* ≠ *authK*; *authK'* ∉ *range shrK*; *evs* ∈ *kerbIV\_gets* ]]  
 ⇒ (*Key servK* ∈ *analz* (*insert* (*Key authK'*) (*spies evs*)))) =  
 (*servK* = *authK'* | *Key servK* ∈ *analz* (*spies evs*)))"  
*<proof>*

**lemma** *analz\_insert\_freshK3\_bis*:  
 "[[ *Says Tgs A*  
   (*Crypt authK* {Key *servK*, *Agent B*, *Number Ts*, *servTicket*})

```

    ∈ set evs;
    authK ≠ authK'; authK' ∉ range shrK; evs ∈ kerbIV_gets ]
    ⇒ (Key servK ∈ analz (insert (Key authK') (spies evs))) =
        (servK = authK' | Key servK ∈ analz (spies evs))"
  <proof>

```

a weakness of the protocol

**lemma** *authK\_compromises\_servK*:

```

  "[ Says Tgs A
    (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
    ∈ set evs; authK ∈ symKeys;
    Key authK ∈ analz (spies evs); evs ∈ kerbIV_gets ]
  ⇒ Key servK ∈ analz (spies evs)"
  <proof>

```

**lemma** *servK\_notin\_authKeysD*:

```

  "[ Crypt authK {Key servK, Agent B, Ts,
    Crypt (shrK B) {Agent A, Agent B, Key servK, Ts}}
    ∈ parts (spies evs);
    Key servK ∉ analz (spies evs);
    B ≠ Tgs; evs ∈ kerbIV_gets ]
  ⇒ servK ∉ authKeys evs"
  <proof>

```

If Spy sees the Authentication Key sent in msg K2, then the Key has expired.

**lemma** *Confidentiality\_Kas\_lemma* [rule\_format]:

```

  "[ authK ∈ symKeys; A ∉ bad; evs ∈ kerbIV_gets ]
  ⇒ Says Kas A
    (Crypt (shrK A)
      {Key authK, Agent Tgs, Number Ta,
      Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}})
    ∈ set evs →
    Key authK ∈ analz (spies evs) →
    expiredAK Ta evs"
  <proof>

```

**lemma** *Confidentiality\_Kas*:

```

  "[ Says Kas A
    (Crypt Ka {Key authK, Agent Tgs, Number Ta, authTicket})
    ∈ set evs;
    ¬ expiredAK Ta evs;
    A ∉ bad; evs ∈ kerbIV_gets ]
  ⇒ Key authK ∉ analz (spies evs)"
  <proof>

```

If Spy sees the Service Key sent in msg K4, then the Key has expired.

**lemma** *Confidentiality\_lemma* [rule\_format]:

```

  "[ Says Tgs A
    (Crypt authK
      {Key servK, Agent B, Number Ts,
      Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}})
    ∈ set evs;
    Key authK ∉ analz (spies evs);

```

```

servK ∈ symKeys;
A ∉ bad; B ∉ bad; evs ∈ kerbIV_gets ]
⇒ Key servK ∈ analz (spies evs) →
  expiredSK Ts evs"
⟨proof⟩

```

In the real world Tgs can't check wheter authK is secure!

**lemma Confidentiality\_Tgs:**

```

"[[ Says Tgs A
  (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
  ∈ set evs;
  Key authK ∉ analz (spies evs);
  ¬ expiredSK Ts evs;
  A ∉ bad; B ∉ bad; evs ∈ kerbIV_gets ]
⇒ Key servK ∉ analz (spies evs)"
⟨proof⟩

```

In the real world Tgs CAN check what Kas sends!

**lemma Confidentiality\_Tgs\_bis:**

```

"[[ Says Kas A
  (Crypt Ka {Key authK, Agent Tgs, Number Ta, authTicket})
  ∈ set evs;
  Says Tgs A
  (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
  ∈ set evs;
  ¬ expiredAK Ta evs; ¬ expiredSK Ts evs;
  A ∉ bad; B ∉ bad; evs ∈ kerbIV_gets ]
⇒ Key servK ∉ analz (spies evs)"
⟨proof⟩

```

Most general form

**lemmas Confidentiality\_Tgs\_ter = authTicket\_authentic [THEN Confidentiality\_Tgs\_bis]**

**lemmas Confidentiality\_Auth\_A = authK\_authentic [THEN Confidentiality\_Kas]**

Needs a confidentiality guarantee, hence moved here. Authenticity of servK for A

**lemma servK\_authentic\_bis\_r:**

```

"[[ Crypt (shrK A) {Key authK, Agent Tgs, Number Ta, authTicket}
  ∈ parts (spies evs);
  Crypt authK {Key servK, Agent B, Number Ts, servTicket}
  ∈ parts (spies evs);
  ¬ expiredAK Ta evs; A ∉ bad; evs ∈ kerbIV_gets ]
⇒ Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
  ∈ set evs"
⟨proof⟩

```

**lemma Confidentiality\_Serv\_A:**

```

"[[ Crypt (shrK A) {Key authK, Agent Tgs, Number Ta, authTicket}
  ∈ parts (spies evs);
  Crypt authK {Key servK, Agent B, Number Ts, servTicket}
  ∈ parts (spies evs);
  ¬ expiredAK Ta evs; ¬ expiredSK Ts evs;

```

7.10 2. Parties' strong authentication: non-injective agreement on the session key. The same guarantees also express

```

    A ∉ bad; B ∉ bad; evs ∈ kerbIV_gets ]
    ⇒ Key servK ∉ analz (spies evs)"
  <proof>

```

**lemma Confidentiality\_B:**

```

  "[ Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}
    ∈ parts (spies evs);
    Crypt authK {Key servK, Agent B, Number Ts, servTicket}
    ∈ parts (spies evs);
    Crypt (shrK A) {Key authK, Agent Tgs, Number Ta, authTicket}
    ∈ parts (spies evs);
    ¬ expiredSK Ts evs; ¬ expiredAK Ta evs;
    A ∉ bad; B ∉ bad; B ≠ Tgs; evs ∈ kerbIV_gets ]
    ⇒ Key servK ∉ analz (spies evs)"
  <proof>

```

**lemma u\_Confidentiality\_B:**

```

  "[ Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}
    ∈ parts (spies evs);
    ¬ expiredSK Ts evs;
    A ∉ bad; B ∉ bad; B ≠ Tgs; evs ∈ kerbIV_gets ]
    ⇒ Key servK ∉ analz (spies evs)"
  <proof>

```

## 7.10 2. Parties' strong authentication: non-injective agreement on the session key. The same guarantees also express key distribution, hence their names

Authentication here still is weak agreement - of B with A

**lemma A\_authenticates\_B:**

```

  "[ Crypt servK (Number T3) ∈ parts (spies evs);
    Crypt authK {Key servK, Agent B, Number Ts, servTicket}
    ∈ parts (spies evs);
    Crypt (shrK A) {Key authK, Agent Tgs, Number Ta, authTicket}
    ∈ parts (spies evs);
    Key authK ∉ analz (spies evs); Key servK ∉ analz (spies evs);
    A ∉ bad; B ∉ bad; evs ∈ kerbIV_gets ]
    ⇒ Says B A (Crypt servK (Number T3)) ∈ set evs"
  <proof>

```

**lemma shrK\_in\_initState\_Server [iff]:** "Key (shrK A) ∈ initState Kas"

<proof>

**lemma shrK\_in\_knows\_Server [iff]:** "Key (shrK A) ∈ knows Kas evs"

<proof>

**lemma A\_authenticates\_and\_keydist\_to\_Kas:**

```

  "[ Gets A (Crypt (shrK A) {Key authK, Peer, Ta, authTicket}) ∈ set evs;
    A ∉ bad; evs ∈ kerbIV_gets ]
    ⇒ Says Kas A (Crypt (shrK A) {Key authK, Peer, Ta, authTicket}) ∈ set
    evs

```

$\wedge \text{Key authK} \in \text{analz}(\text{knows Kas evs})"$   
 $\langle \text{proof} \rangle$

**lemma** *K3\_imp\_Gets\_evs*:

" $\llbracket \text{Says A Tgs } \{\text{Crypt (shrK Tgs)} \{\text{Agent A, Agent Tgs, Key authK, Number Ta}\},$   
 $\text{Crypt authK } \{\text{Agent A, Number T2}\}, \text{Agent B}\}$   
 $\in \text{set evs}; \text{A} \notin \text{bad}; \text{evs} \in \text{kerbIV\_gets} \rrbracket$   
 $\implies \text{Gets A (Crypt (shrK A)} \{\text{Key authK, Agent Tgs, Number Ta,}$   
 $\text{Crypt (shrK Tgs)} \{\text{Agent A, Agent Tgs, Key authK, Number Ta}\}\}$   
 $\in \text{set evs}"$   
 $\langle \text{proof} \rangle$

**lemma** *Tgs\_authenticates\_and\_keydist\_to\_A*:

" $\llbracket \text{Gets Tgs } \{\text{Crypt (shrK Tgs)} \{\text{Agent A, Agent Tgs, Key authK, Number Ta}\},$   
 $\text{Crypt authK } \{\text{Agent A, Number T2}\}, \text{Agent B}\} \in \text{set evs};$   
 $\text{Key authK} \notin \text{analz}(\text{spies evs}); \text{A} \notin \text{bad}; \text{evs} \in \text{kerbIV\_gets} \rrbracket$   
 $\implies \exists \text{B. Says A Tgs } \{\text{Crypt (shrK Tgs)} \{\text{Agent A, Agent Tgs, Key authK, Number Ta}\},$   
 $\text{Crypt authK } \{\text{Agent A, Number T2}\}, \text{Agent B}\} \in \text{set evs}$   
 $\wedge \text{Key authK} \in \text{analz}(\text{knows A evs})"$   
 $\langle \text{proof} \rangle$

**lemma** *K4\_imp\_Gets*:

" $\llbracket \text{Says Tgs A (Crypt authK } \{\text{Key servK, Agent B, Number Ts, servTicket}\})$   
 $\in \text{set evs}; \text{evs} \in \text{kerbIV\_gets} \rrbracket$   
 $\implies \exists \text{Ta X.}$   
 $\text{Gets Tgs } \{\text{Crypt (shrK Tgs)} \{\text{Agent A, Agent Tgs, Key authK, Number Ta}\},$   
 $\text{X}\}$   
 $\in \text{set evs}"$   
 $\langle \text{proof} \rangle$

**lemma** *A\_authenticates\_and\_keydist\_to\_Tgs*:

" $\llbracket \text{Gets A (Crypt (shrK A)} \{\text{Key authK, Agent Tgs, Number Ta, authTicket}\})$   
 $\in \text{set evs};$   
 $\text{Gets A (Crypt authK } \{\text{Key servK, Agent B, Number Ts, servTicket}\})$   
 $\in \text{set evs};$   
 $\text{Key authK} \notin \text{analz}(\text{spies evs}); \text{A} \notin \text{bad};$   
 $\text{evs} \in \text{kerbIV\_gets} \rrbracket$   
 $\implies \text{Says Tgs A (Crypt authK } \{\text{Key servK, Agent B, Number Ts, servTicket}\})$   
 $\in \text{set evs}$   
 $\wedge \text{Key authK} \in \text{analz}(\text{knows Tgs evs})$   
 $\wedge \text{Key servK} \in \text{analz}(\text{knows Tgs evs})"$   
 $\langle \text{proof} \rangle$

**lemma** *K5\_imp\_Gets*:

" $\llbracket \text{Says A B } \{\text{servTicket, Crypt servK } \{\text{Agent A, Number T3}\}\} \in \text{set evs};$   
 $\text{A} \notin \text{bad}; \text{evs} \in \text{kerbIV\_gets} \rrbracket$   
 $\implies \exists \text{authK Ts authTicket T2.}$   
 $\text{Gets A (Crypt authK } \{\text{Key servK, Agent B, Number Ts, servTicket}\}) \in \text{set}$   
 $\text{evs}$   
 $\wedge \text{Says A Tgs } \{\text{authTicket, Crypt authK } \{\text{Agent A, Number T2}\}, \text{Agent B}\} \in$   
 $\text{set evs}"$



*<proof>*

**lemma** *K3\_imp\_Gets:*

"[[ Says A Tgs {authTicket, Crypt authK {Agent A, Number T2}}, Agent B}  
   ∈ set evs;  
   A ∉ bad; evs ∈ kerbIV\_gets ]]  
 ⇒ ∃ Ta. Gets A (Crypt (shrK A) {Key authK, Agent Tgs, Number Ta, authTicket})  
 ∈ set evs"  
*<proof>*

**lemma** *B\_authenticates\_and\_keydist\_to\_A:*

"[[ Gets B {Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}},  
   Crypt servK {Agent A, Number T3}}] ∈ set evs;  
   Key servK ∉ analz (spies evs);  
   A ∉ bad; B ∉ bad; B ≠ Tgs; evs ∈ kerbIV\_gets ]]  
 ⇒ Says A B {Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}},  
   Crypt servK {Agent A, Number T3}}] ∈ set evs  
 ∧ Key servK ∈ analz (knows A evs)"  
*<proof>*

**lemma** *K6\_imp\_Gets:*

"[[ Says B A (Crypt servK (Number T3)) ∈ set evs;  
   B ∉ bad; evs ∈ kerbIV\_gets ]]  
 ⇒ ∃ Ts X. Gets B {Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}}, X  
 ∈ set evs"  
*<proof>*

**lemma** *A\_authenticates\_and\_keydist\_to\_B:*

"[[ Gets A {Crypt authK {Key servK, Agent B, Number Ts, servTicket}},  
   Crypt servK (Number T3)}] ∈ set evs;  
   Gets A (Crypt (shrK A) {Key authK, Agent Tgs, Number Ta, authTicket})  
   ∈ set evs;  
   Key authK ∉ analz (spies evs); Key servK ∉ analz (spies evs);  
   A ∉ bad; B ∉ bad; evs ∈ kerbIV\_gets ]]  
 ⇒ Says B A (Crypt servK (Number T3)) ∈ set evs  
 ∧ Key servK ∈ analz (knows B evs)"  
*<proof>*

**end**

## 8 The Kerberos Protocol, Version V

**theory** *KerberosV* imports *Public* begin

The "u" prefix indicates theorems referring to an updated version of the protocol. The "r" suffix indicates theorems where the confidentiality assumptions are relaxed by the corresponding arguments.

**abbreviation**

```
Kas :: agent where
  "Kas == Server"
```

**abbreviation**

```
Tgs :: agent where
  "Tgs == Friend 0"
```

**axioms**

```
Tgs_not_bad [iff]: "Tgs  $\notin$  bad"
  — Tgs is secure — we already know that Kas is secure
```

**constdefs**

```
authKeys :: "event list => key set"
"authKeys evs == {authK.  $\exists$  A Peer Ta.
  Says Kas A {Crypt (shrK A) {Key authK, Agent Peer, Ta}},
  Crypt (shrK Peer) {Agent A, Agent Peer, Key authK, Ta}}
   $\in$  set evs}"
```

```
Issues :: "[agent, agent, msg, event list] => bool"
  ("_ Issues _ with _ on _")
"A Issues B with X on evs ==
   $\exists$  Y. Says A B Y  $\in$  set evs  $\wedge$  X  $\in$  parts {Y}  $\wedge$ 
  X  $\notin$  parts (spies (takeWhile (% z. z  $\neq$  Says A B Y) (rev evs)))"
```

**consts**

```
authKlife  :: nat
```

```
servKlife  :: nat
```

```
authlife   :: nat
```

```
replylife  :: nat
```

**specification (authKlife)**

```
authKlife_LB [iff]: "2  $\leq$  authKlife"
  <proof>
```

**specification (servKlife)**

```
servKlife_LB [iff]: "2 + authKlife  $\leq$  servKlife"
  <proof>
```

**specification (authlife)**

```
authlife_LB [iff]: "Suc 0  $\leq$  authlife"
  <proof>
```

```

specification (replylife)
  replylife_LB [iff]: "Suc 0 ≤ replylife"
    ⟨proof⟩

```

#### abbreviation

```

CT :: "event list=>nat" where
  "CT == length"

```

#### abbreviation

```

expiredAK :: "[nat, event list] => bool" where
  "expiredAK T evs == authKlife + T < CT evs"

```

#### abbreviation

```

expiredSK :: "[nat, event list] => bool" where
  "expiredSK T evs == servKlife + T < CT evs"

```

#### abbreviation

```

expiredA :: "[nat, event list] => bool" where
  "expiredA T evs == authlife + T < CT evs"

```

#### abbreviation

```

valid :: "[nat, nat] => bool" ("valid _ wrt _") where
  "valid T1 wrt T2 == T1 <= replylife + T2"

```

#### constdefs

```

AKcryptSK :: "[key, key, event list] => bool"
  "AKcryptSK authK servK evs ==
    ∃ A B tt.
      Says Tgs A {Crypt authK {Key servK, Agent B, tt}},
      Crypt (shrK B) {Agent A, Agent B, Key servK, tt}}
    ∈ set evs"

```

#### inductive\_set

**kerbV** :: "event list set"

**where**

```

Nil: "[ ] ∈ kerbV"

/ Fake: "[ evsf ∈ kerbV; X ∈ synth (analz (spies evsf)) ]
  ⇒ Says Spy B X # evsf ∈ kerbV"

/ KV1: "[ evs1 ∈ kerbV ]
  ⇒ Says A Kas {Agent A, Agent Tgs, Number (CT evs1)} # evs1
  ∈ kerbV"

/ KV2: "[ evs2 ∈ kerbV; Key authK ∉ used evs2; authK ∈ symKeys;
  Says A' Kas {Agent A, Agent Tgs, Number T1} ∈ set evs2 ]
  ⇒ Says Kas A {

```

```

Crypt (shrK A) {Key authK, Agent Tgs, Number (CT evs2)},
Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number (CT evs2)}

} # evs2 ∈ kerbV"

/ KV3: "[ evs3 ∈ kerbV; A ≠ Kas; A ≠ Tgs;
  Says A Kas {Agent A, Agent Tgs, Number T1} ∈ set evs3;
  Says Kas' A {Crypt (shrK A) {Key authK, Agent Tgs, Number Ta},
    authTicket} ∈ set evs3;
  valid Ta wrt T1
]
⇒ Says A Tgs {authTicket,
  (Crypt authK {Agent A, Number (CT evs3)}),
  Agent B} # evs3 ∈ kerbV"

/ KV4: "[ evs4 ∈ kerbV; Key servK ∉ used evs4; servK ∈ symKeys;
  B ≠ Tgs; authK ∈ symKeys;
  Says A' Tgs {
    (Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK,
      Number Ta}),
    (Crypt authK {Agent A, Number T2}), Agent B}
  ∈ set evs4;
  ¬ expiredAK Ta evs4;
  ¬ expiredA T2 evs4;
  servKlife + (CT evs4) ≤ authKlife + Ta
]
⇒ Says Tgs A {
  Crypt authK {Key servK, Agent B, Number (CT evs4)},
  Crypt (shrK B) {Agent A, Agent B, Key servK, Number (CT evs4)}
} # evs4 ∈ kerbV"

/ KV5: "[ evs5 ∈ kerbV; authK ∈ symKeys; servK ∈ symKeys;
  A ≠ Kas; A ≠ Tgs;
  Says A Tgs
    {authTicket, Crypt authK {Agent A, Number T2},
     Agent B}
  ∈ set evs5;
  Says Tgs' A {Crypt authK {Key servK, Agent B, Number Ts},
    servTicket}
  ∈ set evs5;
  valid Ts wrt T2 ]
⇒ Says A B {servTicket,
  Crypt servK {Agent A, Number (CT evs5)} }
  # evs5 ∈ kerbV"

/ KV6: "[ evs6 ∈ kerbV; B ≠ Kas; B ≠ Tgs;
  Says A' B {
    (Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}),
    (Crypt servK {Agent A, Number T3})}
  ∈ set evs6;

```

```

      ¬ expiredSK Ts evs6;
      ¬ expiredA T3 evs6
    ]
  ⇒ Says B A (Crypt servK (Number Ta2))
    # evs6 ∈ kerbV"

/ Ops1: "[ evs01 ∈ kerbV; A ≠ Spy;
  Says Kas A {Crypt (shrK A) {Key authK, Agent Tgs, Number Ta}},
            authTicket} ∈ set evs01;
  expiredAK Ta evs01 ]
⇒ Notes Spy {Agent A, Agent Tgs, Number Ta, Key authK}
  # evs01 ∈ kerbV"

/ Ops2: "[ evs02 ∈ kerbV; A ≠ Spy;
  Says Tgs A {Crypt authK {Key servK, Agent B, Number Ts}},
            servTicket} ∈ set evs02;
  expiredSK Ts evs02 ]
⇒ Notes Spy {Agent A, Agent B, Number Ts, Key servK}
  # evs02 ∈ kerbV"

declare Says_imp_knows_Spy [THEN parts.Inj, dest]
declare parts.Body [dest]
declare analz_into_parts [dest]
declare Fake_parts_insert_in_Un [dest]

```

## 8.1 Lemmas about lists, for reasoning about Issues

**lemma** spies\_Says\_rev: "spies (evs @ [Says A B X]) = insert X (spies evs)"  
 <proof>

**lemma** spies\_Gets\_rev: "spies (evs @ [Gets A X]) = spies evs"  
 <proof>

**lemma** spies\_Notes\_rev: "spies (evs @ [Notes A X]) =  
 (if A:bad then insert X (spies evs) else spies evs)"  
 <proof>

**lemma** spies\_evs\_rev: "spies evs = spies (rev evs)"  
 <proof>

**lemmas** parts\_spies\_evs\_revD2 = spies\_evs\_rev [THEN equalityD2, THEN parts\_mono]

**lemma** spies\_takeWhile: "spies (takeWhile P evs) ≤ spies evs"  
 <proof>

**lemmas** parts\_spies\_takeWhile\_mono = spies\_takeWhile [THEN parts\_mono]

## 8.2 Lemmas about `authKeys`

**lemma** `authKeys_empty`: "`authKeys [] = {}`"  
 $\langle proof \rangle$

**lemma** `authKeys_not_insert`:  
 $(\forall A \text{ Ta akey Peer.}$   
 $\text{ev} \neq \text{Says Kas A } \{\{\text{Crypt (shrK A) } \{\{\text{akey, Agent Peer, Ta}\},$   
 $\text{Crypt (shrK Peer) } \{\{\text{Agent A, Agent Peer, akey, Ta}\}\}\})$   
 $\implies \text{authKeys (ev \# evs) = authKeys evs}$ "  
 $\langle proof \rangle$

**lemma** `authKeys_insert`:  
 $\text{"authKeys}$   
 $(\text{Says Kas A } \{\{\text{Crypt (shrK A) } \{\{\text{Key K, Agent Peer, Number Ta}\},$   
 $\text{Crypt (shrK Peer) } \{\{\text{Agent A, Agent Peer, Key K, Number Ta}\}\}\} \# \text{ evs})$   
 $= \text{insert K (authKeys evs)}$ "  
 $\langle proof \rangle$

**lemma** `authKeys_simp`:  
 $\text{"K} \in \text{authKeys}$   
 $(\text{Says Kas A } \{\{\text{Crypt (shrK A) } \{\{\text{Key K', Agent Peer, Number Ta}\},$   
 $\text{Crypt (shrK Peer) } \{\{\text{Agent A, Agent Peer, Key K', Number Ta}\}\}\} \# \text{ evs})$   
 $\implies K = K' \mid K \in \text{authKeys evs}$ "  
 $\langle proof \rangle$

**lemma** `authKeysI`:  
 $\text{"Says Kas A } \{\{\text{Crypt (shrK A) } \{\{\text{Key K, Agent Tgs, Number Ta}\},$   
 $\text{Crypt (shrK Tgs) } \{\{\text{Agent A, Agent Tgs, Key K, Number Ta}\}\}\} \in \text{set evs}$   
 $\implies K \in \text{authKeys evs}$ "  
 $\langle proof \rangle$

**lemma** `authKeys_used`: " $K \in \text{authKeys evs} \implies \text{Key K} \in \text{used evs}$ "  
 $\langle proof \rangle$

## 8.3 Forwarding Lemmas

**lemma** `Says_ticket_parts`:  
 $\text{"Says S A } \{\{\text{Crypt K } \{\{\text{SesKey, B, TimeStamp}\}, \text{Ticket}\}$   
 $\in \text{set evs} \implies \text{Ticket} \in \text{parts (spies evs)}$ "  
 $\langle proof \rangle$

**lemma** `Says_ticket_analz`:  
 $\text{"Says S A } \{\{\text{Crypt K } \{\{\text{SesKey, B, TimeStamp}\}, \text{Ticket}\}$   
 $\in \text{set evs} \implies \text{Ticket} \in \text{analz (spies evs)}$ "  
 $\langle proof \rangle$

**lemma** `Ops_range_spies1`:  
 $\text{"[ Says Kas A } \{\{\text{Crypt KeyA } \{\{\text{Key authK, Peer, Ta}\}, \text{authTicket}\}$   
 $\in \text{set evs} ;$   
 $\text{evs} \in \text{kerbV} \text{ ]} \implies \text{authK} \notin \text{range shrK} \ \& \ \text{authK} \in \text{symKeys}$ "  
 $\langle proof \rangle$

**lemma** `Ops_range_spies2`:  
 $\text{"[ Says Tgs A } \{\{\text{Crypt authK } \{\{\text{Key servK, Agent B, Ts}\}, \text{servTicket}\}$

```

    ∈ set evs ;
    evs ∈ kerbV ] ⇒ servK ∉ range shrK ∧ servK ∈ symKeys"
⟨proof⟩

```

```

lemma Spy_see_shrK [simp]:
  "evs ∈ kerbV ⇒ (Key (shrK A) ∈ parts (spies evs)) = (A ∈ bad)"
⟨proof⟩

```

```

lemma Spy_analz_shrK [simp]:
  "evs ∈ kerbV ⇒ (Key (shrK A) ∈ analz (spies evs)) = (A ∈ bad)"
⟨proof⟩

```

```

lemma Spy_see_shrK_D [dest!]:
  "[ Key (shrK A) ∈ parts (spies evs); evs ∈ kerbV ] ⇒ A:bad"
⟨proof⟩

```

```

lemmas Spy_analz_shrK_D = analz_subset_parts [THEN subsetD, THEN Spy_see_shrK_D,
dest!]

```

Nobody can have used non-existent keys!

```

lemma new_keys_not_used [simp]:
  "[Key K ∉ used evs; K ∈ symKeys; evs ∈ kerbV]
  ⇒ K ∉ keysFor (parts (spies evs))"
⟨proof⟩

```

```

lemma new_keys_not_analzD:
  "[evs ∈ kerbV; K ∈ symKeys; Key K ∉ used evs]
  ⇒ K ∉ keysFor (analz (spies evs))"
⟨proof⟩

```

## 8.4 Regularity Lemmas

These concern the form of items passed in messages

Describes the form of all components sent by Kas

```

lemma Says_Kas_message_form:
  "[ Says Kas A {Crypt K {Key authK, Agent Peer, Ta}}, authTicket}
    ∈ set evs;
    evs ∈ kerbV ]
  ⇒ authK ∉ range shrK ∧ authK ∈ authKeys evs ∧ authK ∈ symKeys ∧

    authTicket = (Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Ta}) ∧
    K = shrK A ∧ Peer = Tgs"
⟨proof⟩

```

```

lemma SesKey_is_session_key:
  "[ Crypt (shrK Tgs_B) {Agent A, Agent Tgs_B, Key SesKey, Number T}
    ∈ parts (spies evs); Tgs_B ∉ bad;

```

$$\begin{aligned} & \text{evs} \in \text{kerbV} \ ] \\ \implies & \text{SesKey} \notin \text{range shrK}'' \\ \langle \text{proof} \rangle & \end{aligned}$$

**lemma** *authTicket\_authentic*:  

$$\begin{aligned} & "[ \text{Crypt (shrK Tgs)} \ \{ \text{Agent A, Agent Tgs, Key authK, Ta} \} \\ & \quad \in \text{parts (spies evs)}; \\ & \quad \text{evs} \in \text{kerbV} \ ] \\ \implies & \text{Says Kas A} \ \{ \text{Crypt (shrK A)} \ \{ \text{Key authK, Agent Tgs, Ta} \}, \\ & \quad \text{Crypt (shrK Tgs)} \ \{ \text{Agent A, Agent Tgs, Key authK, Ta} \} \} \\ & \quad \in \text{set evs}'' \\ \langle \text{proof} \rangle & \end{aligned}$$

**lemma** *authTicket\_crypt\_authK*:  

$$\begin{aligned} & "[ \text{Crypt (shrK Tgs)} \ \{ \text{Agent A, Agent Tgs, Key authK, Number Ta} \} \\ & \quad \in \text{parts (spies evs)}; \\ & \quad \text{evs} \in \text{kerbV} \ ] \\ \implies & \text{authK} \in \text{authKeys evs}'' \\ \langle \text{proof} \rangle & \end{aligned}$$

Describes the form of *servK*, *servTicket* and *authK* sent by Tgs

**lemma** *Says\_Tgs\_message\_form*:  

$$\begin{aligned} & "[ \text{Says Tgs A} \ \{ \text{Crypt authK} \ \{ \text{Key servK, Agent B, Ts} \}, \text{servTicket} \} \\ & \quad \in \text{set evs}; \\ & \quad \text{evs} \in \text{kerbV} \ ] \\ \implies & B \neq \text{Tgs} \wedge \\ & \text{servK} \notin \text{range shrK} \wedge \text{servK} \notin \text{authKeys evs} \wedge \text{servK} \in \text{symKeys} \wedge \\ & \text{servTicket} = (\text{Crypt (shrK B)} \ \{ \text{Agent A, Agent B, Key servK, Ts} \}) \wedge \\ & \text{authK} \notin \text{range shrK} \wedge \text{authK} \in \text{authKeys evs} \wedge \text{authK} \in \text{symKeys}'' \\ \langle \text{proof} \rangle & \end{aligned}$$

## 8.5 Authenticity theorems: confirm origin of sensitive messages

**lemma** *authK\_authentic*:  

$$\begin{aligned} & "[ \text{Crypt (shrK A)} \ \{ \text{Key authK, Peer, Ta} \} \\ & \quad \in \text{parts (spies evs)}; \\ & \quad A \notin \text{bad}; \text{ evs} \in \text{kerbV} \ ] \\ \implies & \exists \text{AT. Says Kas A} \ \{ \text{Crypt (shrK A)} \ \{ \text{Key authK, Peer, Ta} \}, \text{AT} \} \\ & \quad \in \text{set evs}'' \\ \langle \text{proof} \rangle & \end{aligned}$$

If a certain encrypted message appears then it originated with Tgs

**lemma** *servK\_authentic*:  

$$\begin{aligned} & "[ \text{Crypt authK} \ \{ \text{Key servK, Agent B, Ts} \} \\ & \quad \in \text{parts (spies evs)}; \\ & \quad \text{Key authK} \notin \text{analz (spies evs)}; \\ & \quad \text{authK} \notin \text{range shrK}; \\ & \quad \text{evs} \in \text{kerbV} \ ] \\ \implies & \exists A \text{ ST. Says Tgs A} \ \{ \text{Crypt authK} \ \{ \text{Key servK, Agent B, Ts} \}, \text{ST} \} \\ & \quad \in \text{set evs}'' \\ \langle \text{proof} \rangle & \end{aligned}$$



**lemma** *servK\_authentic\_bis*:

```
"[ Crypt authK {Key servK, Agent B, Ts}
  ∈ parts (spies evs);
  Key authK ∉ analz (spies evs);
  B ≠ Tgs;
  evs ∈ kerbV ]
⇒ ∃ A ST. Says Tgs A {Crypt authK {Key servK, Agent B, Ts}, ST}
  ∈ set evs"
⟨proof⟩
```

Authenticity of servK for B

**lemma** *servTicket\_authentic\_Tgs*:

```
"[ Crypt (shrK B) {Agent A, Agent B, Key servK, Ts}
  ∈ parts (spies evs); B ≠ Tgs; B ∉ bad;
  evs ∈ kerbV ]
⇒ ∃ authK.
  Says Tgs A {Crypt authK {Key servK, Agent B, Ts},
    Crypt (shrK B) {Agent A, Agent B, Key servK, Ts}}
  ∈ set evs"
⟨proof⟩
```

Anticipated here from next subsection

**lemma** *K4\_imp\_K2*:

```
"[ Says Tgs A {Crypt authK {Key servK, Agent B, Number Ts}, servTicket}
  ∈ set evs; evs ∈ kerbV ]
⇒ ∃ Ta. Says Kas A
  {Crypt (shrK A) {Key authK, Agent Tgs, Number Ta},
   Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}}
  ∈ set evs"
⟨proof⟩
```

Anticipated here from next subsection

**lemma** *u\_K4\_imp\_K2*:

```
"[ Says Tgs A {Crypt authK {Key servK, Agent B, Number Ts}, servTicket} ∈
set evs; evs ∈ kerbV ]
⇒ ∃ Ta. Says Kas A {Crypt (shrK A) {Key authK, Agent Tgs, Number Ta},
  Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}}
  ∈ set evs
  ∧ servKlife + Ts ≤ authKlife + Ta"
⟨proof⟩
```

**lemma** *servTicket\_authentic\_Kas*:

```
"[ Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}
  ∈ parts (spies evs); B ≠ Tgs; B ∉ bad;
  evs ∈ kerbV ]
⇒ ∃ authK Ta.
  Says Kas A
  {Crypt (shrK A) {Key authK, Agent Tgs, Number Ta},
   Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}}
  ∈ set evs"
⟨proof⟩
```

**lemma** *u\_servTicket\_authentic\_Kas*:

$$\begin{aligned}
& \text{"[ Crypt (shrK B) \{Agent A, Agent B, Key servK, Number Ts\} } \\
& \quad \in \text{parts (spies evs)}; B \neq Tgs; B \notin \text{bad}; \\
& \quad \text{evs} \in \text{kerbV} ] \\
& \implies \exists \text{authK Ta.} \\
& \quad \text{Says Kas A} \\
& \quad \quad \{ \text{Crypt (shrK A) \{Key authK, Agent Tgs, Number Ta\}}, \\
& \quad \quad \text{Crypt (shrK Tgs) \{Agent A, Agent Tgs, Key authK, Number Ta\}} \} \\
& \quad \in \text{set evs} \wedge \\
& \quad \text{servKlife} + Ts \leq \text{authKlife} + Ta" \\
& \langle \text{proof} \rangle
\end{aligned}$$

**lemma** *servTicket\_authentic*:  

$$\begin{aligned}
& \text{"[ Crypt (shrK B) \{Agent A, Agent B, Key servK, Number Ts\} } \\
& \quad \in \text{parts (spies evs)}; B \neq Tgs; B \notin \text{bad}; \\
& \quad \text{evs} \in \text{kerbV} ] \\
& \implies \exists Ta \text{authK.} \\
& \quad \text{Says Kas A \{Crypt (shrK A) \{Key authK, Agent Tgs, Number Ta\},} \\
& \quad \quad \text{Crypt (shrK Tgs) \{Agent A, Agent Tgs, Key authK, Number} } \\
& \quad \text{Ta\}} \} \in \text{set evs} \\
& \quad \wedge \text{Says Tgs A \{Crypt authK \{Key servK, Agent B, Number Ts\},} \\
& \quad \quad \text{Crypt (shrK B) \{Agent A, Agent B, Key servK, Number Ts\}} \} \\
& \quad \in \text{set evs}" \\
& \langle \text{proof} \rangle
\end{aligned}$$

**lemma** *u\_servTicket\_authentic*:  

$$\begin{aligned}
& \text{"[ Crypt (shrK B) \{Agent A, Agent B, Key servK, Number Ts\} } \\
& \quad \in \text{parts (spies evs)}; B \neq Tgs; B \notin \text{bad}; \\
& \quad \text{evs} \in \text{kerbV} ] \\
& \implies \exists Ta \text{authK.} \\
& \quad \text{Says Kas A \{Crypt (shrK A) \{Key authK, Agent Tgs, Number Ta\},} \\
& \quad \quad \text{Crypt (shrK Tgs) \{Agent A, Agent Tgs, Key authK, Number} } \\
& \quad \text{Ta\}} \} \in \text{set evs} \\
& \quad \wedge \text{Says Tgs A \{Crypt authK \{Key servK, Agent B, Number Ts\},} \\
& \quad \quad \text{Crypt (shrK B) \{Agent A, Agent B, Key servK, Number Ts\}} \} \\
& \quad \in \text{set evs} \\
& \quad \wedge \text{servKlife} + Ts \leq \text{authKlife} + Ta" \\
& \langle \text{proof} \rangle
\end{aligned}$$

**lemma** *u\_NotexpiredSK\_NotexpiredAK*:  

$$\begin{aligned}
& \text{"[ } \neg \text{expiredSK Ts evs; servKlife} + Ts \leq \text{authKlife} + Ta ] \\
& \implies \neg \text{expiredAK Ta evs}" \\
& \langle \text{proof} \rangle
\end{aligned}$$

## 8.6 Reliability: friendly agents send something if something else happened

**lemma** *K3\_imp\_K2*:  

$$\begin{aligned}
& \text{"[ Says A Tgs} \\
& \quad \{ \text{authTicket, Crypt authK \{Agent A, Number T2\}, Agent B} \} \\
& \quad \in \text{set evs}; \\
& \quad A \notin \text{bad}; \text{ evs} \in \text{kerbV} ] \\
& \implies \exists Ta AT. \text{Says Kas A \{Crypt (shrK A) \{Key authK, Agent Tgs, Ta\},} \\
& \quad \quad \quad AT\} \in \text{set evs}"
\end{aligned}$$

*<proof>*

Anticipated here from next subsection. An authK is encrypted by one and only one Shared key. A servK is encrypted by one and only one authK.

**lemma** *Key\_unique\_SesKey:*

```
"[[ Crypt K  {Key SesKey,  Agent B, T}
    ∈ parts (spies evs);
    Crypt K' {Key SesKey,  Agent B', T'}
    ∈ parts (spies evs); Key SesKey ∉ analz (spies evs);
    evs ∈ kerbV ]]
⇒ K=K' ∧ B=B' ∧ T=T'"
```

*<proof>*

This inevitably has an existential form in version V

**lemma** *Says\_K5:*

```
"[[ Crypt servK {Agent A, Number T3} ∈ parts (spies evs);
    Says Tgs A {Crypt authK {Key servK, Agent B, Number Ts},
                servTicket} ∈ set evs;
    Key servK ∉ analz (spies evs);
    A ∉ bad; B ∉ bad; evs ∈ kerbV ]]
⇒ ∃ ST. Says A B {ST, Crypt servK {Agent A, Number T3}} ∈ set evs"
```

*<proof>*

Anticipated here from next subsection

**lemma** *unique\_CryptKey:*

```
"[[ Crypt (shrK B) {Agent A, Agent B, Key SesKey, T}
    ∈ parts (spies evs);
    Crypt (shrK B') {Agent A', Agent B', Key SesKey, T'}
    ∈ parts (spies evs); Key SesKey ∉ analz (spies evs);
    evs ∈ kerbV ]]
⇒ A=A' & B=B' & T=T'"
```

*<proof>*

**lemma** *Says\_K6:*

```
"[[ Crypt servK (Number T3) ∈ parts (spies evs);
    Says Tgs A {Crypt authK {Key servK, Agent B, Number Ts},
                servTicket} ∈ set evs;
    Key servK ∉ analz (spies evs);
    A ∉ bad; B ∉ bad; evs ∈ kerbV ]]
⇒ Says B A (Crypt servK (Number T3)) ∈ set evs"
```

*<proof>*

Needs a unicity theorem, hence moved here

**lemma** *servK\_authentic\_ter:*

```
"[[ Says Kas A
    {Crypt (shrK A) {Key authK, Agent Tgs, Ta}, authTicket} ∈ set evs;
    Crypt authK {Key servK, Agent B, Ts}
    ∈ parts (spies evs);
    Key authK ∉ analz (spies evs);
    evs ∈ kerbV ]]
⇒ Says Tgs A {Crypt authK {Key servK, Agent B, Ts},
    Crypt (shrK B) {Agent A, Agent B, Key servK, Ts}}
    ∈ set evs"
```

*<proof>*

## 8.7 Unicity Theorems

The session key, if secure, uniquely identifies the Ticket whether `authTicket` or `servTicket`. As a matter of fact, one can read also `Tgs` in the place of `B`.

**lemma** *unique\_authKeys*:

```
"[[ Says Kas A
    {Crypt Ka {Key authK, Agent Tgs, Ta}}, X} ∈ set evs;
   Says Kas A'
    {Crypt Ka' {Key authK, Agent Tgs, Ta'}}, X'} ∈ set evs;
   evs ∈ kerbV ]] ⇒ A=A' ∧ Ka=Ka' ∧ Ta=Ta' ∧ X=X'"
⟨proof⟩
```

`servK` uniquely identifies the message from `Tgs`

**lemma** *unique\_servKeys*:

```
"[[ Says Tgs A
    {Crypt K {Key servK, Agent B, Ts}}, X} ∈ set evs;
   Says Tgs A'
    {Crypt K' {Key servK, Agent B', Ts'}}, X'} ∈ set evs;
   evs ∈ kerbV ]] ⇒ A=A' ∧ B=B' ∧ K=K' ∧ Ts=Ts' ∧ X=X'"
⟨proof⟩
```

## 8.8 Lemmas About the Predicate *AKcryptSK*

**lemma** *not\_AKcryptSK\_Nil* [iff]: " $\neg$  *AKcryptSK* `authK` `servK` []"

⟨proof⟩

**lemma** *AKcryptSKI*:

```
"[[ Says Tgs A {Crypt authK {Key servK, Agent B, tt}}, X} ∈ set evs;
   evs ∈ kerbV ]] ⇒ AKcryptSK authK servK evs"
⟨proof⟩
```

**lemma** *AKcryptSK\_Says* [simp]:

```
"AKcryptSK authK servK (Says S A X # evs) =
  (S = Tgs ∧
   (∃ B tt. X = {Crypt authK {Key servK, Agent B, tt}},
                Crypt (shrK B) {Agent A, Agent B, Key servK, tt}})
  / AKcryptSK authK servK evs)"
⟨proof⟩
```

**lemma** *AKcryptSK\_Notes* [simp]:

```
"AKcryptSK authK servK (Notes A X # evs) =
  AKcryptSK authK servK evs"
⟨proof⟩
```

**lemma** *Auth\_fresh\_not\_AKcryptSK*:

```
"[[ Key authK ∉ used evs; evs ∈ kerbV ]]
 ⇒ ¬ AKcryptSK authK servK evs"
⟨proof⟩
```

**lemma** *Serv\_fresh\_not\_AKcryptSK*:

```
"Key servK ∉ used evs ⇒ ¬ AKcryptSK authK servK evs"
⟨proof⟩
```

**lemma** *authK\_not\_AKcryptSK*:  
 "[ Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, tk}  
   ∈ parts (spies evs); evs ∈ kerbV ]  
 ⇒ ¬ AKcryptSK K authK evs"  
 <proof>

A secure serverkey cannot have been used to encrypt others

**lemma** *servK\_not\_AKcryptSK*:  
 "[ Crypt (shrK B) {Agent A, Agent B, Key SK, tt} ∈ parts (spies evs);  
   Key SK ∉ analz (spies evs); SK ∈ symKeys;  
   B ≠ Tgs; evs ∈ kerbV ]  
 ⇒ ¬ AKcryptSK SK K evs"  
 <proof>

Long term keys are not issued as servKeys

**lemma** *shrK\_not\_AKcryptSK*:  
 "evs ∈ kerbV ⇒ ¬ AKcryptSK K (shrK A) evs"  
 <proof>

The Tgs message associates servK with authK and therefore not with any other key authK.

**lemma** *Says\_Tgs\_AKcryptSK*:  
 "[ Says Tgs A {Crypt authK {Key servK, Agent B, tt}, X}  
   ∈ set evs;  
   authK' ≠ authK; evs ∈ kerbV ]  
 ⇒ ¬ AKcryptSK authK' servK evs"  
 <proof>

**lemma** *AKcryptSK\_not\_AKcryptSK*:  
 "[ AKcryptSK authK servK evs; evs ∈ kerbV ]  
 ⇒ ¬ AKcryptSK servK K evs"  
 <proof>

**lemma** *not\_different\_AKcryptSK*:  
 "[ AKcryptSK authK servK evs;  
   authK' ≠ authK; evs ∈ kerbV ]  
 ⇒ ¬ AKcryptSK authK' servK evs ∧ servK ∈ symKeys"  
 <proof>

The only session keys that can be found with the help of session keys are those sent by Tgs in step K4.

We take some pains to express the property as a logical equivalence so that the simplifier can apply it.

**lemma** *Key\_analz\_image\_Key\_lemma*:  
 "P → (Key K ∈ analz (Key'KK Un H)) → (K:KK | Key K ∈ analz H)  
 ⇒  
 P → (Key K ∈ analz (Key'KK Un H)) = (K:KK | Key K ∈ analz H)"  
 <proof>

**lemma** *AKcryptSK\_analz\_insert*:

"[ AKcryptSK K K' evs; K ∈ symKeys; evs ∈ kerbV ]  
 ⇒ Key K' ∈ analz (insert (Key K) (spies evs))"  
 <proof>

**lemma** authKeys\_are\_not\_AKcryptSK:  
 "[ K ∈ authKeys evs Un range shrK; evs ∈ kerbV ]  
 ⇒ ∀ SK. ¬ AKcryptSK SK K evs ∧ K ∈ symKeys"  
 <proof>

**lemma** not\_authKeys\_not\_AKcryptSK:  
 "[ K ∉ authKeys evs;  
 K ∉ range shrK; evs ∈ kerbV ]  
 ⇒ ∀ SK. ¬ AKcryptSK K SK evs"  
 <proof>

## 8.9 Secrecy Theorems

For the Oops2 case of the next theorem

**lemma** Oops2\_not\_AKcryptSK:  
 "[ evs ∈ kerbV;  
 Says Tgs A {Crypt authK  
 {Key servK, Agent B, Number Ts}, servTicket}  
 ∈ set evs ]  
 ⇒ ¬ AKcryptSK servK SK evs"  
 <proof>

Big simplification law for keys SK that are not crypted by keys in KK It helps prove three, otherwise hard, facts about keys. These facts are exploited as simplification laws for analz, and also "limit the damage" in case of loss of a key to the spy. See ESORICS98.

**lemma** Key\_analz\_image\_Key [rule\_format (no\_asm)]:  
 "evs ∈ kerbV ⇒  
 (∀ SK KK. SK ∈ symKeys & KK ≤ -(range shrK) →  
 (∀ K ∈ KK. ¬ AKcryptSK K SK evs) →  
 (Key SK ∈ analz (Key'KK Un (spies evs))) =  
 (SK ∈ KK | Key SK ∈ analz (spies evs)))"  
 <proof>

First simplification law for analz: no session keys encrypt authentication keys or shared keys.

**lemma** analz\_insert\_freshK1:  
 "[ evs ∈ kerbV; K ∈ authKeys evs Un range shrK;  
 SesKey ∉ range shrK ]  
 ⇒ (Key K ∈ analz (insert (Key SesKey) (spies evs))) =  
 (K = SesKey | Key K ∈ analz (spies evs))"  
 <proof>

Second simplification law for analz: no service keys encrypt any other keys.

**lemma** analz\_insert\_freshK2:  
 "[ evs ∈ kerbV; servK ∉ (authKeys evs); servK ∉ range shrK;  
 K ∈ symKeys ]  
 ⇒ (Key K ∈ analz (insert (Key servK) (spies evs))) =

( $K = \text{servK} \mid \text{Key } K \in \text{analz } (\text{spies evs})$ )"  
 <proof>

Third simplification law for analz: only one authentication key encrypts a certain service key.

**lemma** *analz\_insert\_freshK3*:

"[ AKcryptSK authK servK evs;  
 authK'  $\neq$  authK; authK'  $\notin$  range shrK; evs  $\in$  kerbV ]  
 $\implies$  ( $\text{Key servK} \in \text{analz } (\text{insert } (\text{Key authK}') (\text{spies evs}))$ ) =  
 ( $\text{servK} = \text{authK}' \mid \text{Key servK} \in \text{analz } (\text{spies evs})$ )"

<proof>

**lemma** *analz\_insert\_freshK3\_bis*:

"[ Says Tgs A {Crypt authK {Key servK, Agent B, Number Ts}}, servTicket}  
 $\in$  set evs;  
 authK  $\neq$  authK'; authK'  $\notin$  range shrK; evs  $\in$  kerbV ]  
 $\implies$  ( $\text{Key servK} \in \text{analz } (\text{insert } (\text{Key authK}') (\text{spies evs}))$ ) =  
 ( $\text{servK} = \text{authK}' \mid \text{Key servK} \in \text{analz } (\text{spies evs})$ )"

<proof>

a weakness of the protocol

**lemma** *authK\_compromises\_servK*:

"[ Says Tgs A {Crypt authK {Key servK, Agent B, Number Ts}}, servTicket}  
 $\in$  set evs; authK  $\in$  symKeys;  
 Key authK  $\in$  analz (spies evs); evs  $\in$  kerbV ]  
 $\implies$  Key servK  $\in$  analz (spies evs)"

<proof>

lemma *servK\_notin\_authKeysD* not needed in version V

If Spy sees the Authentication Key sent in msg K2, then the Key has expired.

**lemma** *Confidentiality\_Kas\_lemma* [rule\_format]:

"[ authK  $\in$  symKeys; A  $\notin$  bad; evs  $\in$  kerbV ]  
 $\implies$  Says Kas A  
 {Crypt (shrK A) {Key authK, Agent Tgs, Number Ta}},  
 Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}}  
 $\in$  set evs  $\longrightarrow$   
 Key authK  $\in$  analz (spies evs)  $\longrightarrow$   
 expiredAK Ta evs"

<proof>

**lemma** *Confidentiality\_Kas*:

"[ Says Kas A  
 {Crypt Ka {Key authK, Agent Tgs, Number Ta}}, authTicket}  
 $\in$  set evs;  
 $\neg$  expiredAK Ta evs;  
 A  $\notin$  bad; evs  $\in$  kerbV ]  
 $\implies$  Key authK  $\notin$  analz (spies evs)"

<proof>

If Spy sees the Service Key sent in msg K4, then the Key has expired.

**lemma** *Confidentiality\_lemma* [rule\_format]:

"[ Says Tgs A

```

    {Crypt authK {Key servK, Agent B, Number Ts}},
    Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}}
  ∈ set evs;
  Key authK ∉ analz (spies evs);
  servK ∈ symKeys;
  A ∉ bad; B ∉ bad; evs ∈ kerbV ]
⇒ Key servK ∈ analz (spies evs) →
  expiredSK Ts evs"
⟨proof⟩

```

In the real world Tgs can't check wheter authK is secure!

**lemma Confidentiality\_Tgs:**

```

"[[ Says Tgs A
    {Crypt authK {Key servK, Agent B, Number Ts}}, servTicket}
  ∈ set evs;
  Key authK ∉ analz (spies evs);
  ¬ expiredSK Ts evs;
  A ∉ bad; B ∉ bad; evs ∈ kerbV ]
⇒ Key servK ∉ analz (spies evs)"
⟨proof⟩

```

In the real world Tgs CAN check what Kas sends!

**lemma Confidentiality\_Tgs\_bis:**

```

"[[ Says Kas A
    {Crypt Ka {Key authK, Agent Tgs, Number Ta}}, authTicket}
  ∈ set evs;
  Says Tgs A
    {Crypt authK {Key servK, Agent B, Number Ts}}, servTicket}
  ∈ set evs;
  ¬ expiredAK Ta evs; ¬ expiredSK Ts evs;
  A ∉ bad; B ∉ bad; evs ∈ kerbV ]
⇒ Key servK ∉ analz (spies evs)"
⟨proof⟩

```

Most general form

**lemmas Confidentiality\_Tgs\_ter = authTicket\_authentic [THEN Confidentiality\_Tgs\_bis]**

**lemmas Confidentiality\_Auth\_A = authK\_authentic [THEN exE, THEN Confidentiality\_Kas]**

Needs a confidentiality guarantee, hence moved here. Authenticity of servK for A

**lemma servK\_authentic\_bis\_r:**

```

"[[ Crypt (shrK A) {Key authK, Agent Tgs, Number Ta}
  ∈ parts (spies evs);
  Crypt authK {Key servK, Agent B, Number Ts}
  ∈ parts (spies evs);
  ¬ expiredAK Ta evs; A ∉ bad; evs ∈ kerbV ]
⇒ Says Tgs A {Crypt authK {Key servK, Agent B, Number Ts}},
    Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}}
  ∈ set evs"
⟨proof⟩

```

**lemma Confidentiality\_Serv\_A:**



8.10 Parties authentication: each party verifies "the identity of another party who generated some data" (quoted from

```
"[ Crypt (shrK A) {Key authK, Agent Tgs, Number Ta}
  ∈ parts (spies evs);
  Crypt authK {Key servK, Agent B, Number Ts}
  ∈ parts (spies evs);
  ¬ expiredAK Ta evs; ¬ expiredSK Ts evs;
  A ∉ bad; B ∉ bad; B ≠ Tgs; evs ∈ kerbV ]
⇒ Key servK ∉ analz (spies evs)"
⟨proof⟩
```

**lemma Confidentiality\_B:**

```
"[ Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}
  ∈ parts (spies evs);
  Crypt authK {Key servK, Agent B, Number Ts}
  ∈ parts (spies evs);
  Crypt (shrK A) {Key authK, Agent Tgs, Number Ta}
  ∈ parts (spies evs);
  ¬ expiredSK Ts evs; ¬ expiredAK Ta evs;
  A ∉ bad; B ∉ bad; B ≠ Tgs; evs ∈ kerbV ]
⇒ Key servK ∉ analz (spies evs)"
⟨proof⟩
```

**lemma u\_Confidentiality\_B:**

```
"[ Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}
  ∈ parts (spies evs);
  ¬ expiredSK Ts evs;
  A ∉ bad; B ∉ bad; B ≠ Tgs; evs ∈ kerbV ]
⇒ Key servK ∉ analz (spies evs)"
⟨proof⟩
```

## 8.10 Parties authentication: each party verifies "the identity of another party who generated some data" (quoted from Neuman and Ts'o).

These guarantees don't assess whether two parties agree on the same session key: sending a message containing a key doesn't a priori state knowledge of the key.

These didn't have existential form in version IV

**lemma B\_authenticates\_A:**

```
"[ Crypt servK {Agent A, Number T3} ∈ parts (spies evs);
  Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}
  ∈ parts (spies evs);
  Key servK ∉ analz (spies evs);
  A ∉ bad; B ∉ bad; B ≠ Tgs; evs ∈ kerbV ]
⇒ ∃ ST. Says A B {ST, Crypt servK {Agent A, Number T3}} ∈ set evs"
⟨proof⟩
```

The second assumption tells B what kind of key servK is.

**lemma B\_authenticates\_A\_r:**

```
"[ Crypt servK {Agent A, Number T3} ∈ parts (spies evs);
  Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}
  ∈ parts (spies evs);
  Crypt authK {Key servK, Agent B, Number Ts}
```

$\in \text{parts}(\text{spies evs});$   
 $\text{Crypt}(\text{shrK } A) \{ \text{Key authK, Agent Tgs, Number Ta} \}$   
 $\in \text{parts}(\text{spies evs});$   
 $\neg \text{expiredSK } Ts \text{ evs}; \neg \text{expiredAK } Ta \text{ evs};$   
 $B \neq Tgs; A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{kerbV} \]$   
 $\Rightarrow \exists ST. \text{Says } A \ B \{ ST, \text{Crypt servK } \{ \text{Agent } A, \text{Number } T3 \} \} \in \text{set evs}$   
 $\langle \text{proof} \rangle$

$u\_B\_authenticates\_A$  would be the same as  $B\_authenticates\_A$  because the servK confidentiality assumption is yet unrelaxed

**lemma**  $u\_B\_authenticates\_A\_r$ :

$"[ \text{Crypt servK } \{ \text{Agent } A, \text{Number } T3 \} \in \text{parts}(\text{spies evs});$   
 $\text{Crypt}(\text{shrK } B) \{ \text{Agent } A, \text{Agent } B, \text{Key servK, Number } Ts \}$   
 $\in \text{parts}(\text{spies evs});$   
 $\neg \text{expiredSK } Ts \text{ evs};$   
 $B \neq Tgs; A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{kerbV} \]$   
 $\Rightarrow \exists ST. \text{Says } A \ B \{ ST, \text{Crypt servK } \{ \text{Agent } A, \text{Number } T3 \} \} \in \text{set evs}$   
 $\langle \text{proof} \rangle$

**lemma**  $A\_authenticates\_B$ :

$"[ \text{Crypt servK } (\text{Number } T3) \in \text{parts}(\text{spies evs});$   
 $\text{Crypt authK } \{ \text{Key servK, Agent } B, \text{Number } Ts \}$   
 $\in \text{parts}(\text{spies evs});$   
 $\text{Crypt}(\text{shrK } A) \{ \text{Key authK, Agent Tgs, Number Ta} \}$   
 $\in \text{parts}(\text{spies evs});$   
 $\text{Key authK} \notin \text{analz}(\text{spies evs}); \text{Key servK} \notin \text{analz}(\text{spies evs});$   
 $A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{kerbV} \]$   
 $\Rightarrow \text{Says } B \ A \ (\text{Crypt servK } (\text{Number } T3)) \in \text{set evs}$   
 $\langle \text{proof} \rangle$

**lemma**  $A\_authenticates\_B\_r$ :

$"[ \text{Crypt servK } (\text{Number } T3) \in \text{parts}(\text{spies evs});$   
 $\text{Crypt authK } \{ \text{Key servK, Agent } B, \text{Number } Ts \}$   
 $\in \text{parts}(\text{spies evs});$   
 $\text{Crypt}(\text{shrK } A) \{ \text{Key authK, Agent Tgs, Number Ta} \}$   
 $\in \text{parts}(\text{spies evs});$   
 $\neg \text{expiredAK } Ta \text{ evs}; \neg \text{expiredSK } Ts \text{ evs};$   
 $A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{kerbV} \]$   
 $\Rightarrow \text{Says } B \ A \ (\text{Crypt servK } (\text{Number } T3)) \in \text{set evs}$   
 $\langle \text{proof} \rangle$

**8.11 Parties' knowledge of session keys.** An agent knows a session key if he used it to issue a cipher. These guarantees can be interpreted both in terms of key distribution and of non-injective agreement on the session key.

**lemma**  $Kas\_Issues\_A$ :

$"[ \text{Says } Kas \ A \{ \text{Crypt}(\text{shrK } A) \{ \text{Key authK, Peer, Ta} \}, \text{authTicket} \} \in \text{set evs};$   
 $\text{evs} \in \text{kerbV} \]$   
 $\Rightarrow Kas \text{ Issues } A \text{ with } (\text{Crypt}(\text{shrK } A) \{ \text{Key authK, Peer, Ta} \})$   
 $\text{on evs}$

8.11 Parties' knowledge of session keys. An agent knows a session key if he used it to issue a cipher. These guarantees

*<proof>*

**lemma** *A\_authenticates\_and\_keydist\_to\_Kas:*

"[ Crypt (shrK A) {Key authK, Peer, Ta} ∈ parts (spies evs);  
 A ∉ bad; evs ∈ kerbV ]  
 ⇒ Kas Issues A with (Crypt (shrK A) {Key authK, Peer, Ta})  
 on evs"

*<proof>*

**lemma** *Tgs\_Issues\_A:*

"[ Says Tgs A {Crypt authK {Key servK, Agent B, Number Ts}}, servTicket}  
 ∈ set evs;  
 Key authK ∉ analz (spies evs); evs ∈ kerbV ]  
 ⇒ Tgs Issues A with  
 (Crypt authK {Key servK, Agent B, Number Ts}) on evs"

*<proof>*

**lemma** *A\_authenticates\_and\_keydist\_to\_Tgs:*

"[ Crypt authK {Key servK, Agent B, Number Ts}  
 ∈ parts (spies evs);  
 Key authK ∉ analz (spies evs); B ≠ Tgs; evs ∈ kerbV ]  
 ⇒ ∃A. Tgs Issues A with  
 (Crypt authK {Key servK, Agent B, Number Ts}) on evs"

*<proof>*

**lemma** *B\_Issues\_A:*

"[ Says B A (Crypt servK (Number T3)) ∈ set evs;  
 Key servK ∉ analz (spies evs);  
 A ∉ bad; B ∉ bad; B ≠ Tgs; evs ∈ kerbV ]  
 ⇒ B Issues A with (Crypt servK (Number T3)) on evs"

*<proof>*

**lemma** *A\_authenticates\_and\_keydist\_to\_B:*

"[ Crypt servK (Number T3) ∈ parts (spies evs);  
 Crypt authK {Key servK, Agent B, Number Ts}  
 ∈ parts (spies evs);  
 Crypt (shrK A) {Key authK, Agent Tgs, Number Ta}  
 ∈ parts (spies evs);  
 Key authK ∉ analz (spies evs); Key servK ∉ analz (spies evs);  
 A ∉ bad; B ∉ bad; B ≠ Tgs; evs ∈ kerbV ]  
 ⇒ B Issues A with (Crypt servK (Number T3)) on evs"

*<proof>*

But can prove a less general fact concerning only authenticators!

**lemma** *honest\_never\_says\_newer\_timestamp\_in\_auth:*

"[ (CT evs) ≤ T; Number T ∈ parts {X}; A ∉ bad; evs ∈ kerbV ]  
 ⇒ Says A B {Y, X} ∉ set evs"

*<proof>*

**lemma** *honest\_never\_says\_current\_timestamp\_in\_auth:*

"[ (CT evs) = T; Number T ∈ parts {X}; A ∉ bad; evs ∈ kerbV ]  
 ⇒ Says A B {Y, X} ∉ set evs"

*<proof>*

**lemma** *A\_Issues\_B*:  
 "[[ Says A B {ST, Crypt servK {Agent A, Number T3}} ∈ set evs;  
 Key servK ∉ analz (spies evs);  
 B ≠ Tgs; A ∉ bad; B ∉ bad; evs ∈ kerbV ]]  
 ⇒ A Issues B with (Crypt servK {Agent A, Number T3}) on evs"  
 <proof>

**lemma** *B\_authenticates\_and\_keydist\_to\_A*:  
 "[[ Crypt servK {Agent A, Number T3} ∈ parts (spies evs);  
 Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}  
 ∈ parts (spies evs);  
 Key servK ∉ analz (spies evs);  
 B ≠ Tgs; A ∉ bad; B ∉ bad; evs ∈ kerbV ]]  
 ⇒ A Issues B with (Crypt servK {Agent A, Number T3}) on evs"  
 <proof>

**8.12 Novel guarantees, never studied before.** Because honest agents always say the right timestamp in authenticators, we can prove unicity guarantees based exactly on timestamps. Classical unicity guarantees are based on nonces. Of course assuming the agent to be different from the Spy, rather than not in bad, would suffice below. Similar guarantees must also hold of Kerberos IV.

Notice that an honest agent can send the same timestamp on two different traces of the same length, but not on the same trace!

**lemma** *unique\_timestamp\_authenticator1*:  
 "[[ Says A Kas {Agent A, Agent Tgs, Number T1} ∈ set evs;  
 Says A Kas' {Agent A, Agent Tgs', Number T1} ∈ set evs;  
 A ∉ bad; evs ∈ kerbV ]]  
 ⇒ Kas=Kas' ∧ Tgs=Tgs' "  
 <proof>

**lemma** *unique\_timestamp\_authenticator2*:  
 "[[ Says A Tgs {AT, Crypt AK {Agent A, Number T2}, Agent B} ∈ set evs;  
 Says A Tgs' {AT', Crypt AK' {Agent A, Number T2}, Agent B'} ∈ set evs;  
 A ∉ bad; evs ∈ kerbV ]]  
 ⇒ Tgs=Tgs' ∧ AT=AT' ∧ AK=AK' ∧ B=B' "  
 <proof>

**lemma** *unique\_timestamp\_authenticator3*:  
 "[[ Says A B {ST, Crypt SK {Agent A, Number T}} ∈ set evs;  
 Says A B' {ST', Crypt SK' {Agent A, Number T}} ∈ set evs;  
 A ∉ bad; evs ∈ kerbV ]]  
 ⇒ B=B' ∧ ST=ST' ∧ SK=SK' "  
 <proof>

The second part of the message is treated as an authenticator by the last simplification step, even if it is not an authenticator!

8.12 Novel guarantees, never studied before. Because honest agents always say the right timestamp in authenticator

**lemma** *unique\_timestamp\_authticket:*

```
"[ Says Kas A {X, Crypt (shrK Tgs) {Agent A, Agent Tgs, Key AK, T}} ] ∈
set evs;
  Says Kas A' {X', Crypt (shrK Tgs') {Agent A', Agent Tgs', Key AK',
T}} ] ∈ set evs;
    evs ∈ kerbV ]
⇒ A=A' ∧ X=X' ∧ Tgs=Tgs' ∧ AK=AK'"
⟨proof⟩
```

The second part of the message is treated as an authenticator by the last simplification step, even if it is not an authenticator!

**lemma** *unique\_timestamp\_servticket:*

```
"[ Says Tgs A {X, Crypt (shrK B) {Agent A, Agent B, Key SK, T}} ] ∈ set
evs;
  Says Tgs A' {X', Crypt (shrK B') {Agent A', Agent B', Key SK', T}}
∈ set evs;
    evs ∈ kerbV ]
⇒ A=A' ∧ X=X' ∧ B=B' ∧ SK=SK'"
⟨proof⟩
```

**lemma** *Kas\_never\_says\_newer\_timestamp:*

```
"[ (CT evs) ≤ T; Number T ∈ parts {X}; evs ∈ kerbV ]
⇒ ∀ A. Says Kas A X ∉ set evs"
⟨proof⟩
```

**lemma** *Kas\_never\_says\_current\_timestamp:*

```
"[ (CT evs) = T; Number T ∈ parts {X}; evs ∈ kerbV ]
⇒ ∀ A. Says Kas A X ∉ set evs"
⟨proof⟩
```

**lemma** *unique\_timestamp\_msg2:*

```
"[ Says Kas A {Crypt (shrK A) {Key AK, Agent Tgs, T}, AT} ] ∈ set evs;
  Says Kas A' {Crypt (shrK A') {Key AK', Agent Tgs', T}, AT'} ] ∈ set evs;
    evs ∈ kerbV ]
⇒ A=A' ∧ AK=AK' ∧ Tgs=Tgs' ∧ AT=AT'"
⟨proof⟩
```

**lemma** *Tgs\_never\_says\_newer\_timestamp:*

```
"[ (CT evs) ≤ T; Number T ∈ parts {X}; evs ∈ kerbV ]
⇒ ∀ A. Says Tgs A X ∉ set evs"
⟨proof⟩
```

**lemma** *Tgs\_never\_says\_current\_timestamp:*

```
"[ (CT evs) = T; Number T ∈ parts {X}; evs ∈ kerbV ]
⇒ ∀ A. Says Tgs A X ∉ set evs"
⟨proof⟩
```

**lemma** *unique\_timestamp\_msg4:*

```
"[ Says Tgs A {Crypt (shrK A) {Key SK, Agent B, T}, ST} ] ∈ set evs;
  Says Tgs A' {Crypt (shrK A') {Key SK', Agent B', T}, ST'} ] ∈ set evs;
    evs ∈ kerbV ]
```

$\implies A=A' \wedge SK=SK' \wedge B=B' \wedge ST=ST'$   
 $\langle proof \rangle$

end

## 9 The Original Otway-Rees Protocol

theory OtwayRees imports Public begin

From page 244 of Burrows, Abadi and Needham (1989). A Logic of Authentication. Proc. Royal Soc. 426

This is the original version, which encrypts Nonce NB.

inductive\_set otway :: "event list set"  
 where

Nil: " $[] \in \text{otway}$ "

| Fake: " $[| \text{evsf} \in \text{otway}; X \in \text{synth} (\text{analz} (\text{knows Spy evsf})) |]$   
 $\implies \text{Says Spy B X} \# \text{evsf} \in \text{otway}$ "

| Reception: " $[| \text{evsr} \in \text{otway}; \text{Says A B X} \in \text{set evsr} |]$   
 $\implies \text{Gets B X} \# \text{evsr} \in \text{otway}$ "

| OR1: " $[| \text{evs1} \in \text{otway}; \text{Nonce NA} \notin \text{used evs1} |]$   
 $\implies \text{Says A B} \{ | \text{Nonce NA, Agent A, Agent B,}$   
 $\quad \text{Crypt (shrK A)} \{ | \text{Nonce NA, Agent A, Agent B} | \} | \}$   
 $\# \text{evs1} : \text{otway}$ "

| OR2: " $[| \text{evs2} \in \text{otway}; \text{Nonce NB} \notin \text{used evs2};$   
 $\text{Gets B} \{ | \text{Nonce NA, Agent A, Agent B, X} | \} : \text{set evs2} |]$   
 $\implies \text{Says B Server}$   
 $\{ | \text{Nonce NA, Agent A, Agent B, X,}$   
 $\quad \text{Crypt (shrK B)}$   
 $\quad \{ | \text{Nonce NA, Nonce NB, Agent A, Agent B} | \} | \}$   
 $\# \text{evs2} : \text{otway}$ "

| OR3: " $[| \text{evs3} \in \text{otway}; \text{Key KAB} \notin \text{used evs3};$   
 $\text{Gets Server}$   
 $\{ | \text{Nonce NA, Agent A, Agent B,}$   
 $\quad \text{Crypt (shrK A)} \{ | \text{Nonce NA, Agent A, Agent B} | \},$   
 $\quad \text{Crypt (shrK B)} \{ | \text{Nonce NA, Nonce NB, Agent A, Agent B} | \} | \}$   
 $: \text{set evs3} |]$   
 $\implies \text{Says Server B}$   
 $\{ | \text{Nonce NA,}$   
 $\quad \text{Crypt (shrK A)} \{ | \text{Nonce NA, Key KAB} | \},$   
 $\quad \text{Crypt (shrK B)} \{ | \text{Nonce NB, Key KAB} | \} | \}$   
 $\# \text{evs3} : \text{otway}$ "

```

| OR4: "[| evs4 ∈ otway; B ≠ Server;
      Says B Server {|Nonce NA, Agent A, Agent B, X',
                    Crypt (shrK B)
                      {|Nonce NA, Nonce NB, Agent A, Agent B|}|}
      : set evs4;
      Gets B {|Nonce NA, X, Crypt (shrK B) {|Nonce NB, Key K|}|}
      : set evs4 |]
==> Says B A {|Nonce NA, X|} # evs4 : otway"

```

```

| Ops: "[| evso ∈ otway;
      Says Server B {|Nonce NA, X, Crypt (shrK B) {|Nonce NB, Key K|}|}
      : set evso |]
==> Notes Spy {|Nonce NA, Nonce NB, Key K|} # evso : otway"

```

```

declare Says_imp_analz_Spy [dest]
declare parts.Body [dest]
declare analz_into_parts [dest]
declare Fake_parts_insert_in_Un [dest]

```

A "possibility property": there are traces that reach the end

```

lemma "[| B ≠ Server; Key K ∉ used [] |]
==> ∃ evs ∈ otway.
      Says B A {|Nonce NA, Crypt (shrK A) {|Nonce NA, Key K|}|}
      ∈ set evs"

```

⟨proof⟩

```

lemma Gets_imp_Says [dest!]:
  "[| Gets B X ∈ set evs; evs ∈ otway |] ==> ∃ A. Says A B X ∈ set evs"
⟨proof⟩

```

```

lemma OR2_analz_knows_Spy:
  "[| Gets B {|N, Agent A, Agent B, X|} ∈ set evs; evs ∈ otway |]
  ==> X ∈ analz (knows Spy evs)"
⟨proof⟩

```

```

lemma OR4_analz_knows_Spy:
  "[| Gets B {|N, X, Crypt (shrK B) X'|} ∈ set evs; evs ∈ otway |]
  ==> X ∈ analz (knows Spy evs)"
⟨proof⟩

```

```

lemmas OR2_parts_knows_Spy =
  OR2_analz_knows_Spy [THEN analz_into_parts, standard]

```

Theorems of the form  $X \notin \text{parts } (\text{knows Spy evs})$  imply that NOBODY sends messages containing X!

Spy never sees a good agent's shared key!

```

lemma Spy_see_shrK [simp]:
  "evs ∈ otway ==> (Key (shrK A) ∈ parts (knows Spy evs)) = (A ∈ bad)"
  <proof>

```

```

lemma Spy_analz_shrK [simp]:
  "evs ∈ otway ==> (Key (shrK A) ∈ analz (knows Spy evs)) = (A ∈ bad)"
  <proof>

```

```

lemma Spy_see_shrK_D [dest!]:
  "[| Key (shrK A) ∈ parts (knows Spy evs); evs ∈ otway |] ==> A ∈ bad"
  <proof>

```

### 9.1 Towards Secrecy: Proofs Involving *analz*

```

lemma Says_Server_message_form:
  "[| Says Server B {|NA, X, Crypt (shrK B) {|NB, Key K|}|} ∈ set evs;
    evs ∈ otway |]
    ==> K ∉ range shrK & (∃ i. NA = Nonce i) & (∃ j. NB = Nonce j)"
  <proof>

```

Session keys are not used to encrypt other session keys

The equality makes the induction hypothesis easier to apply

```

lemma analz_image_freshK [rule_format]:
  "evs ∈ otway ==>
    ∀ K KK. KK ≤ -(range shrK) -->
      (Key K ∈ analz (Key'KK Un (knows Spy evs))) =
      (K ∈ KK | Key K ∈ analz (knows Spy evs))"
  <proof>

```

```

lemma analz_insert_freshK:
  "[| evs ∈ otway; KAB ∉ range shrK |] ==>
    (Key K ∈ analz (insert (Key KAB) (knows Spy evs))) =
    (K = KAB | Key K ∈ analz (knows Spy evs))"
  <proof>

```

The Key K uniquely identifies the Server's message.

```

lemma unique_session_keys:
  "[| Says Server B {|NA, X, Crypt (shrK B) {|NB, K|}|} ∈ set evs;
    Says Server B' {|NA', X', Crypt (shrK B') {|NB', K|}|} ∈ set evs;
    evs ∈ otway |] ==> X=X' & B=B' & NA=NA' & NB=NB'"
  <proof>

```

### 9.2 Authenticity properties relating to NA

Only OR1 can have caused such a part of a message to appear.

```

lemma Crypt_imp_OR1 [rule_format]:
  "[| A ∉ bad; evs ∈ otway |]
    ==> Crypt (shrK A) {|NA, Agent A, Agent B|} ∈ parts (knows Spy evs) -->
      Says A B {|NA, Agent A, Agent B,
        Crypt (shrK A) {|NA, Agent A, Agent B|}|}
        ∈ set evs"

```



*<proof>*

**lemma** *Crypt\_imp\_OR1\_Gets*:

```
"[| Gets B {|NA, Agent A, Agent B,
          Crypt (shrK A) {|NA, Agent A, Agent B|}|} ∈ set evs;
  A ∉ bad; evs ∈ otway |]
==> Says A B {|NA, Agent A, Agent B,
              Crypt (shrK A) {|NA, Agent A, Agent B|}|}
   ∈ set evs"
```

*<proof>*

The Nonce NA uniquely identifies A's message

**lemma** *unique\_NA*:

```
"[| Crypt (shrK A) {|NA, Agent A, Agent B|} ∈ parts (knows Spy evs);
  Crypt (shrK A) {|NA, Agent A, Agent C|} ∈ parts (knows Spy evs);
  evs ∈ otway; A ∉ bad |]
==> B = C"
```

*<proof>*

It is impossible to re-use a nonce in both OR1 and OR2. This holds because OR2 encrypts Nonce NB. It prevents the attack that can occur in the over-simplified version of this protocol: see *OtwayRees\_Bad*.

**lemma** *no\_nonce\_OR1\_OR2*:

```
"[| Crypt (shrK A) {|NA, Agent A, Agent B|} ∈ parts (knows Spy evs);
  A ∉ bad; evs ∈ otway |]
==> Crypt (shrK A) {|NA', NA, Agent A', Agent A|} ∉ parts (knows Spy evs)"
```

*<proof>*

Crucial property: If the encrypted message appears, and A has used NA to start a run, then it originated with the Server!

**lemma** *NA\_Crypt\_imp\_Server\_msg* [rule\_format]:

```
"[| A ∉ bad; evs ∈ otway |]
==> Says A B {|NA, Agent A, Agent B,
              Crypt (shrK A) {|NA, Agent A, Agent B|}|} ∈ set evs -->
  Crypt (shrK A) {|NA, Key K|} ∈ parts (knows Spy evs)
  --> (∃NB. Says Server B
      {|NA,
        Crypt (shrK A) {|NA, Key K|},
        Crypt (shrK B) {|NB, Key K|}|} ∈ set evs)"
```

*<proof>*

Corollary: if A receives B's OR4 message and the nonce NA agrees then the key really did come from the Server! CANNOT prove this of the bad form of this protocol, even though we can prove *Spy\_not\_see\_encrypted\_key*

**lemma** *A\_trusts\_OR4*:

```
"[| Says A B {|NA, Agent A, Agent B,
              Crypt (shrK A) {|NA, Agent A, Agent B|}|} ∈ set evs;
  Says B' A {|NA, Crypt (shrK A) {|NA, Key K|}|} ∈ set evs;
  A ∉ bad; evs ∈ otway |]
==> ∃NB. Says Server B
  {|NA,
    Crypt (shrK A) {|NA, Key K|},
    Crypt (shrK B) {|NB, Key K|}|}
```

$\in \text{set evs}$ "

*<proof>*

Crucial secrecy property: Spy does not see the keys sent in msg OR3 Does not in itself guarantee security: an attack could violate the premises, e.g. by having  $A = \text{Spy}$

**lemma** *secrecy\_lemma*:

```
"[| A ∉ bad; B ∉ bad; evs ∈ otway |]
  ==> Says Server B
    {|NA, Crypt (shrK A) {|NA, Key K|},
      Crypt (shrK B) {|NB, Key K|}|} ∈ set evs -->
    Notes Spy {|NA, NB, Key K|} ∉ set evs -->
    Key K ∉ analz (knows Spy evs)"
```

*<proof>*

**theorem** *Spy\_not\_see\_encrypted\_key*:

```
"[| Says Server B
    {|NA, Crypt (shrK A) {|NA, Key K|},
      Crypt (shrK B) {|NB, Key K|}|} ∈ set evs;
    Notes Spy {|NA, NB, Key K|} ∉ set evs;
    A ∉ bad; B ∉ bad; evs ∈ otway |]
  ==> Key K ∉ analz (knows Spy evs)"
```

*<proof>*

This form is an immediate consequence of the previous result. It is similar to the assertions established by other methods. It is equivalent to the previous result in that the Spy already has *analz* and *synth* at his disposal. However, the conclusion  $\text{Key } K \notin \text{knows Spy evs}$  appears not to be inductive: all the cases other than Fake are trivial, while Fake requires  $\text{Key } K \notin \text{analz (knows Spy evs)}$ .

**lemma** *Spy\_not\_know\_encrypted\_key*:

```
"[| Says Server B
    {|NA, Crypt (shrK A) {|NA, Key K|},
      Crypt (shrK B) {|NB, Key K|}|} ∈ set evs;
    Notes Spy {|NA, NB, Key K|} ∉ set evs;
    A ∉ bad; B ∉ bad; evs ∈ otway |]
  ==> Key K ∉ knows Spy evs"
```

*<proof>*

A's guarantee. The Oops premise quantifies over NB because A cannot know what it is.

**lemma** *A\_gets\_good\_key*:

```
"[| Says A B {|NA, Agent A, Agent B,
    Crypt (shrK A) {|NA, Agent A, Agent B|}|} ∈ set evs;
    Says B' A {|NA, Crypt (shrK A) {|NA, Key K|}|} ∈ set evs;
    ∀NB. Notes Spy {|NA, NB, Key K|} ∉ set evs;
    A ∉ bad; B ∉ bad; evs ∈ otway |]
  ==> Key K ∉ analz (knows Spy evs)"
```

*<proof>*

### 9.3 Authenticity properties relating to NB

Only OR2 can have caused such a part of a message to appear. We do not know anything about X: it does NOT have to have the right form.

**lemma** *Crypt\_imp\_OR2*:

```
"[| Crypt (shrK B) {|NA, NB, Agent A, Agent B|} ∈ parts (knows Spy evs);
  B ∉ bad; evs ∈ otway |]
==> ∃X. Says B Server
      {|NA, Agent A, Agent B, X,
       Crypt (shrK B) {|NA, NB, Agent A, Agent B|}|}
      ∈ set evs"
```

*<proof>*

The Nonce NB uniquely identifies B's message

**lemma** *unique\_NB*:

```
"[| Crypt (shrK B) {|NA, NB, Agent A, Agent B|} ∈ parts(knows Spy evs);
  Crypt (shrK B) {|NC, NB, Agent C, Agent B|} ∈ parts(knows Spy evs);
  evs ∈ otway; B ∉ bad |]
==> NC = NA & C = A"
```

*<proof>*

If the encrypted message appears, and B has used Nonce NB, then it originated with the Server! Quite messy proof.

**lemma** *NB\_Crypt\_imp\_Server\_msg [rule\_format]*:

```
"[| B ∉ bad; evs ∈ otway |]
==> Crypt (shrK B) {|NB, Key K|} ∈ parts (knows Spy evs)
--> (∃X'. Says B Server
      {|NA, Agent A, Agent B, X',
       Crypt (shrK B) {|NA, NB, Agent A, Agent B|}|}
      ∈ set evs
--> Says Server B
      {|NA, Crypt (shrK A) {|NA, Key K|},
       Crypt (shrK B) {|NB, Key K|}|}
      ∈ set evs)"
```

*<proof>*

Guarantee for B: if it gets a message with matching NB then the Server has sent the correct message.

**theorem** *B\_trusts\_OR3*:

```
"[| Says B Server {|NA, Agent A, Agent B, X',
                  Crypt (shrK B) {|NA, NB, Agent A, Agent B|} |}
  ∈ set evs;
  Gets B {|NA, X, Crypt (shrK B) {|NB, Key K|}|} ∈ set evs;
  B ∉ bad; evs ∈ otway |]
==> Says Server B
      {|NA,
       Crypt (shrK A) {|NA, Key K|},
       Crypt (shrK B) {|NB, Key K|}|}
      ∈ set evs"
```

*<proof>*

The obvious combination of *B\_trusts\_OR3* with *Spy\_not\_see\_encrypted\_key*

**lemma** *B\_gets\_good\_key*:

```
"[| Says B Server {|NA, Agent A, Agent B, X',
                  Crypt (shrK B) {|NA, NB, Agent A, Agent B|} |}
  ∈ set evs;
  Gets B {|NA, X, Crypt (shrK B) {|NB, Key K|}|} ∈ set evs;
```

```

Notes Spy {|NA, NB, Key K|} ∉ set evs;
A ∉ bad; B ∉ bad; evs ∈ otway []
==> Key K ∉ analz (knows Spy evs)"
⟨proof⟩

```

```

lemma OR3_imp_OR2:
  "[| Says Server B
    {|NA, Crypt (shrK A) {|NA, Key K|}|},
    Crypt (shrK B) {|NB, Key K|}|} ∈ set evs;
    B ∉ bad; evs ∈ otway []
  ==> ∃X. Says B Server {|NA, Agent A, Agent B, X,
    Crypt (shrK B) {|NA, NB, Agent A, Agent B|}|} |}
    ∈ set evs"
⟨proof⟩

```

After getting and checking OR4, agent A can trust that B has been active. We could probably prove that X has the expected form, but that is not strictly necessary for authentication.

```

theorem A_auths_B:
  "[| Says B' A {|NA, Crypt (shrK A) {|NA, Key K|}|} ∈ set evs;
    Says A B {|NA, Agent A, Agent B,
    Crypt (shrK A) {|NA, Agent A, Agent B|}|} ∈ set evs;
    A ∉ bad; B ∉ bad; evs ∈ otway []
  ==> ∃NB X. Says B Server {|NA, Agent A, Agent B, X,
    Crypt (shrK B) {|NA, NB, Agent A, Agent B|}|}
    |}
    ∈ set evs"
⟨proof⟩

end

```

## 10 The Otway-Rees Protocol as Modified by Abadi and Needham

**theory** *OtwayRees\_AN* **imports** *Public* **begin**

This simplified version has minimal encryption and explicit messages.

Note that the formalization does not even assume that nonces are fresh. This is because the protocol does not rely on uniqueness of nonces for security, only for freshness, and the proof script does not prove freshness properties.

From page 11 of Abadi and Needham (1996). Prudent Engineering Practice for Cryptographic Protocols. IEEE Trans. SE 22 (1)

```

inductive_set otway :: "event list set"
  where
    Nil: — The empty trace
          "[|] ∈ otway"

```

/ **Fake:** — The Spy may say anything he can say. The sender field is correct, but agents don't use that information.

```

"[| evsf ∈ otway; X ∈ synth (analz (knows Spy evsf)) |]
==> Says Spy B X # evsf ∈ otway"

/ Reception: — A message that has been sent can be received by the intended
recipient.
"[| evsr ∈ otway; Says A B X ∈ set evsr |]
==> Gets B X # evsr ∈ otway"

/ OR1: — Alice initiates a protocol run
"evs1 ∈ otway
==> Says A B {|Agent A, Agent B, Nonce NA|} # evs1 ∈ otway"

/ OR2: — Bob's response to Alice's message.
"[| evs2 ∈ otway;
  Gets B {|Agent A, Agent B, Nonce NA|} ∈ set evs2 |]
==> Says B Server {|Agent A, Agent B, Nonce NA, Nonce NB|}
  # evs2 ∈ otway"

/ OR3: — The Server receives Bob's message. Then he sends a new session key to
Bob with a packet for forwarding to Alice.
"[| evs3 ∈ otway; Key KAB ∉ used evs3;
  Gets Server {|Agent A, Agent B, Nonce NA, Nonce NB|}
  ∈ set evs3 |]
==> Says Server B
  {|Crypt (shrK A) {|Nonce NA, Agent A, Agent B, Key KAB|},
   Crypt (shrK B) {|Nonce NB, Agent A, Agent B, Key KAB|}|}
  # evs3 ∈ otway"

/ OR4: — Bob receives the Server's (?) message and compares the Nonces with
those in the message he previously sent the Server. Need B ≠ Server because we
allow messages to self.
"[| evs4 ∈ otway; B ≠ Server;
  Says B Server {|Agent A, Agent B, Nonce NA, Nonce NB|} ∈ set evs4;
  Gets B {|X, Crypt (shrK B) {|Nonce NB, Agent A, Agent B, Key K|}|}
  ∈ set evs4 |]
==> Says B A X # evs4 ∈ otway"

/ Oops: — This message models possible leaks of session keys. The nonces identify
the protocol run.
"[| evso ∈ otway;
  Says Server B
    {|Crypt (shrK A) {|Nonce NA, Agent A, Agent B, Key K|},
     Crypt (shrK B) {|Nonce NB, Agent A, Agent B, Key K|}|}
  ∈ set evso |]
==> Notes Spy {|Nonce NA, Nonce NB, Key K|} # evso ∈ otway"

declare Says_imp_knows_Spy [THEN analz.Inj, dest]
declare parts.Body [dest]
declare analz_into_parts [dest]
declare Fake_parts_insert_in_Un [dest]

```

A "possibility property": there are traces that reach the end

```

lemma "[| B ≠ Server; Key K ∉ used [] |]
  ==> ∃ evs ∈ otway.
    Says B A (Crypt (shrK A) {|Nonce NA, Agent A, Agent B, Key K|})
      ∈ set evs"
<proof>

```

```

lemma Gets_imp_Says [dest!]:
  "[| Gets B X ∈ set evs; evs ∈ otway |] ==> ∃ A. Says A B X ∈ set evs"
<proof>

```

For reasoning about the encrypted portion of messages

```

lemma OR4_analz_knows_Spy:
  "[| Gets B {|X, Crypt(shrK B) X'|} ∈ set evs; evs ∈ otway |]
  ==> X ∈ analz (knows Spy evs)"
<proof>

```

Theorems of the form  $X \notin \text{parts}(\text{knows Spy evs})$  imply that NOBODY sends messages containing X!

Spy never sees a good agent's shared key!

```

lemma Spy_see_shrK [simp]:
  "evs ∈ otway ==> (Key (shrK A) ∈ parts (knows Spy evs)) = (A ∈ bad)"
<proof>

```

```

lemma Spy_analz_shrK [simp]:
  "evs ∈ otway ==> (Key (shrK A) ∈ analz (knows Spy evs)) = (A ∈ bad)"
<proof>

```

```

lemma Spy_see_shrK_D [dest!]:
  "[| Key (shrK A) ∈ parts (knows Spy evs); evs ∈ otway |] ==> A ∈ bad"
<proof>

```

## 10.1 Proofs involving analz

Describes the form of K and NA when the Server sends this message.

```

lemma Says_Server_message_form:
  "[| Says Server B
    {|Crypt (shrK A) {|NA, Agent A, Agent B, Key K|},
      Crypt (shrK B) {|NB, Agent A, Agent B, Key K|}|}
    ∈ set evs;
    evs ∈ otway |]
  ==> K ∉ range shrK & (∃ i. NA = Nonce i) & (∃ j. NB = Nonce j)"
<proof>

```

Session keys are not used to encrypt other session keys

The equality makes the induction hypothesis easier to apply

```

lemma analz_image_freshK [rule_format]:
  "evs ∈ otway ==>
    ∀ K KK. KK ≤ -(range shrK) -->
      (Key K ∈ analz (Key'KK Un (knows Spy evs))) =
      (K ∈ KK | Key K ∈ analz (knows Spy evs))"
<proof>

```

```

lemma analz_insert_freshK:
  "[| evs ∈ otway; KAB ∉ range shrK |] ==>
    (Key K ∈ analz (insert (Key KAB) (knows Spy evs))) =
      (K = KAB | Key K ∈ analz (knows Spy evs))"
  <proof>

```

The Key K uniquely identifies the Server's message.

```

lemma unique_session_keys:
  "[| Says Server B
      {|Crypt (shrK A) {|NA, Agent A, Agent B, K|},
       Crypt (shrK B) {|NB, Agent A, Agent B, K|}|}
   ∈ set evs;
   Says Server B'
      {|Crypt (shrK A') {|NA', Agent A', Agent B', K|},
       Crypt (shrK B') {|NB', Agent A', Agent B', K|}|}
   ∈ set evs;
   evs ∈ otway |]
  ==> A=A' & B=B' & NA=NA' & NB=NB'"
  <proof>

```

## 10.2 Authenticity properties relating to NA

If the encrypted message appears then it originated with the Server!

```

lemma NA_Crypt_imp_Server_msg [rule_format]:
  "[| A ∉ bad; A ≠ B; evs ∈ otway |]
  ==> Crypt (shrK A) {|NA, Agent A, Agent B, Key K|} ∈ parts (knows Spy
    evs)
    --> (∃ NB. Says Server B
          {|Crypt (shrK A) {|NA, Agent A, Agent B, Key K|},
           Crypt (shrK B) {|NB, Agent A, Agent B, Key K|}|}
        ∈ set evs)"
  <proof>

```

Corollary: if A receives B's OR4 message then it originated with the Server. Freshness may be inferred from nonce NA.

```

lemma A_trusts_OR4:
  "[| Says B' A (Crypt (shrK A) {|NA, Agent A, Agent B, Key K|}) ∈ set
    evs;
   A ∉ bad; A ≠ B; evs ∈ otway |]
  ==> ∃ NB. Says Server B
      {|Crypt (shrK A) {|NA, Agent A, Agent B, Key K|},
       Crypt (shrK B) {|NB, Agent A, Agent B, Key K|}|}
    ∈ set evs"
  <proof>

```

Crucial secrecy property: Spy does not see the keys sent in msg OR3 Does not in itself guarantee security: an attack could violate the premises, e.g. by having  $A = \text{Spy}$

```

lemma secrecy_lemma:
  "[| A ∉ bad; B ∉ bad; evs ∈ otway |]
  ==> Says Server B

```

```

      {|Crypt (shrK A) {|NA, Agent A, Agent B, Key K|},
       Crypt (shrK B) {|NB, Agent A, Agent B, Key K|}|}
    ∈ set evs -->
    Notes Spy {|NA, NB, Key K|} ∉ set evs -->
    Key K ∉ analz (knows Spy evs)"
⟨proof⟩

```

```

lemma Spy_not_see_encrypted_key:
  "[| Says Server B
    {|Crypt (shrK A) {|NA, Agent A, Agent B, Key K|},
     Crypt (shrK B) {|NB, Agent A, Agent B, Key K|}|}
   ∈ set evs;
   Notes Spy {|NA, NB, Key K|} ∉ set evs;
   A ∉ bad; B ∉ bad; evs ∈ otway |]
  ==> Key K ∉ analz (knows Spy evs)"
⟨proof⟩

```

A's guarantee. The Oops premise quantifies over NB because A cannot know what it is.

```

lemma A_gets_good_key:
  "[| Says B' A (Crypt (shrK A) {|NA, Agent A, Agent B, Key K|}) ∈ set
   evs;
   ∀NB. Notes Spy {|NA, NB, Key K|} ∉ set evs;
   A ∉ bad; B ∉ bad; A ≠ B; evs ∈ otway |]
  ==> Key K ∉ analz (knows Spy evs)"
⟨proof⟩

```

### 10.3 Authenticity properties relating to NB

If the encrypted message appears then it originated with the Server!

```

lemma NB_Crypt_imp_Server_msg [rule_format]:
  "[| B ∉ bad; A ≠ B; evs ∈ otway |]
  ==> Crypt (shrK B) {|NB, Agent A, Agent B, Key K|} ∈ parts (knows Spy evs)
  --> (∃NA. Says Server B
    {|Crypt (shrK A) {|NA, Agent A, Agent B, Key K|},
     Crypt (shrK B) {|NB, Agent A, Agent B, Key K|}|}
    ∈ set evs)"
⟨proof⟩

```

Guarantee for B: if it gets a well-formed certificate then the Server has sent the correct message in round 3.

```

lemma B_trusts_OR3:
  "[| Says S B {|X, Crypt (shrK B) {|NB, Agent A, Agent B, Key K|}|}
   ∈ set evs;
   B ∉ bad; A ≠ B; evs ∈ otway |]
  ==> ∃NA. Says Server B
    {|Crypt (shrK A) {|NA, Agent A, Agent B, Key K|},
     Crypt (shrK B) {|NB, Agent A, Agent B, Key K|}|}
    ∈ set evs"
⟨proof⟩

```

The obvious combination of *B\_trusts\_OR3* with *Spy\_not\_see\_encrypted\_key*



```

lemma B_gets_good_key:
  "[| Gets B {|X, Crypt (shrK B) {|NB, Agent A, Agent B, Key K|}|}
    ∈ set evs;
    ∀ NA. Notes Spy {|NA, NB, Key K|} ∉ set evs;
    A ∉ bad; B ∉ bad; A ≠ B; evs ∈ otway |]
    ==> Key K ∉ analz (knows Spy evs)"
  <proof>

end

```

## 11 The Otway-Rees Protocol: The Faulty BAN Version

**theory** OtwayRees\_Bad **imports** Public **begin**

The FAULTY version omitting encryption of Nonce NB, as suggested on page 247 of Burrows, Abadi and Needham (1988). A Logic of Authentication. Proc. Royal Soc. 426

This file illustrates the consequences of such errors. We can still prove impressive-looking properties such as *Spy\_not\_see\_encrypted\_key*, yet the protocol is open to a middleperson attack. Attempting to prove some key lemmas indicates the possibility of this attack.

```

inductive_set otway :: "event list set"
  where
    Nil: — The empty trace
      "[| ∈ otway"

    / Fake: — The Spy may say anything he can say. The sender field is correct, but
    agents don't use that information.
      "[| evsf ∈ otway; X ∈ synth (analz (knows Spy evsf)) |]
        ==> Says Spy B X # evsf ∈ otway"

    / Reception: — A message that has been sent can be received by the intended
    recipient.
      "[| evsr ∈ otway; Says A B X ∈ set evsr |]
        ==> Gets B X # evsr ∈ otway"

    / OR1: — Alice initiates a protocol run
      "[| evs1 ∈ otway; Nonce NA ∉ used evs1 |]
        ==> Says A B {|Nonce NA, Agent A, Agent B,
          Crypt (shrK A) {|Nonce NA, Agent A, Agent B|}|}
          # evs1 ∈ otway"

    / OR2: — Bob's response to Alice's message. This variant of the protocol does
    NOT encrypt NB.
      "[| evs2 ∈ otway; Nonce NB ∉ used evs2;
        Gets B {|Nonce NA, Agent A, Agent B, X|} ∈ set evs2 |]
        ==> Says B Server
          {|Nonce NA, Agent A, Agent B, X, Nonce NB,
            Crypt (shrK B) {|Nonce NA, Agent A, Agent B|}|}

```

```

# evs2 ∈ otway"

/ OR3: — The Server receives Bob's message and checks that the three NAs match.
Then he sends a new session key to Bob with a packet for forwarding to Alice.
"[| evs3 ∈ otway; Key KAB ∉ used evs3;
  Gets Server
    {|Nonce NA, Agent A, Agent B,
      Crypt (shrK A) {|Nonce NA, Agent A, Agent B|},
      Nonce NB,
      Crypt (shrK B) {|Nonce NA, Agent A, Agent B|}|}
  ∈ set evs3 |]
==> Says Server B
    {|Nonce NA,
      Crypt (shrK A) {|Nonce NA, Key KAB|},
      Crypt (shrK B) {|Nonce NB, Key KAB|}|}
# evs3 ∈ otway"

/ OR4: — Bob receives the Server's (?) message and compares the Nonces with
those in the message he previously sent the Server. Need B ≠ Server because we
allow messages to self.
"[| evs4 ∈ otway; B ≠ Server;
  Says B Server {|Nonce NA, Agent A, Agent B, X', Nonce NB,
    Crypt (shrK B) {|Nonce NA, Agent A, Agent B|}|}
  ∈ set evs4;
  Gets B {|Nonce NA, X, Crypt (shrK B) {|Nonce NB, Key K|}|}
  ∈ set evs4 |]
==> Says B A {|Nonce NA, X|} # evs4 ∈ otway"

/ Oops: — This message models possible leaks of session keys. The nonces identify
the protocol run.
"[| evso ∈ otway;
  Says Server B {|Nonce NA, X, Crypt (shrK B) {|Nonce NB, Key K|}|}
  ∈ set evso |]
==> Notes Spy {|Nonce NA, Nonce NB, Key K|} # evso ∈ otway"

declare Says_imp_knows_Spy [THEN analz.Inj, dest]
declare parts.Body [dest]
declare analz_into_parts [dest]
declare Fake_parts_insert_in_Un [dest]

```

A "possibility property": there are traces that reach the end

```

lemma "[| B ≠ Server; Key K ∉ used [] |]
==> ∃ NA. ∃ evs ∈ otway.
  Says B A {|Nonce NA, Crypt (shrK A) {|Nonce NA, Key K|}|}
  ∈ set evs"
<proof>

lemma Gets_imp_Says [dest!]:
  "[| Gets B X ∈ set evs; evs ∈ otway |] ==> ∃ A. Says A B X ∈ set evs"
<proof>

```

## 11.1 For reasoning about the encrypted portion of messages

```
lemma OR2_analz_knows_Spy:
  "[| Gets B {|N, Agent A, Agent B, X|} ∈ set evs; evs ∈ otway |]
   ==> X ∈ analz (knows Spy evs)"
⟨proof⟩
```

```
lemma OR4_analz_knows_Spy:
  "[| Gets B {|N, X, Crypt (shrK B) X'|} ∈ set evs; evs ∈ otway |]
   ==> X ∈ analz (knows Spy evs)"
⟨proof⟩
```

```
lemma Oops_parts_knows_Spy:
  "Says Server B {|NA, X, Crypt K' {|NB,K|}|} ∈ set evs
   ==> K ∈ parts (knows Spy evs)"
⟨proof⟩
```

Forwarding lemma: see comments in OtwayRees.thy

```
lemmas OR2_parts_knows_Spy =
  OR2_analz_knows_Spy [THEN analz_into_parts, standard]
```

Theorems of the form  $X \notin \text{parts } (\text{knows Spy evs})$  imply that NOBODY sends messages containing X!

Spy never sees a good agent's shared key!

```
lemma Spy_see_shrK [simp]:
  "evs ∈ otway ==> (Key (shrK A) ∈ parts (knows Spy evs)) = (A ∈ bad)"
⟨proof⟩
```

```
lemma Spy_analz_shrK [simp]:
  "evs ∈ otway ==> (Key (shrK A) ∈ analz (knows Spy evs)) = (A ∈ bad)"
⟨proof⟩
```

```
lemma Spy_see_shrK_D [dest!]:
  "[| Key (shrK A) ∈ parts (knows Spy evs); evs ∈ otway |] ==> A ∈ bad"
⟨proof⟩
```

## 11.2 Proofs involving analz

Describes the form of K and NA when the Server sends this message. Also for Oops case.

```
lemma Says_Server_message_form:
  "[| Says Server B {|NA, X, Crypt (shrK B) {|NB, Key K|}|} ∈ set evs;
    evs ∈ otway |]
   ==> K ∉ range shrK & (∃ i. NA = Nonce i) & (∃ j. NB = Nonce j)"
⟨proof⟩
```

Session keys are not used to encrypt other session keys

The equality makes the induction hypothesis easier to apply

```
lemma analz_image_freshK [rule_format]:
```

```

"evs ∈ otway ==>
  ∀K KK. KK ≤ -(range shrK) -->
    (Key K ∈ analz (Key'KK Un (knows Spy evs))) =
    (K ∈ KK | Key K ∈ analz (knows Spy evs))"
<proof>

```

```

lemma analz_insert_freshK:
  "[| evs ∈ otway; KAB ∉ range shrK |] ==>
    (Key K ∈ analz (insert (Key KAB) (knows Spy evs))) =
    (K = KAB | Key K ∈ analz (knows Spy evs))"
<proof>

```

The Key K uniquely identifies the Server's message.

```

lemma unique_session_keys:
  "[| Says Server B {|NA, X, Crypt (shrK B) {|NB, K|}|} ∈ set evs;
    Says Server B' {|NA', X', Crypt (shrK B') {|NB', K'|}|} ∈ set evs;
    evs ∈ otway |] ==> X=X' & B=B' & NA=NA' & NB=NB'"
<proof>

```

Crucial secrecy property: Spy does not see the keys sent in msg OR3 Does not in itself guarantee security: an attack could violate the premises, e.g. by having  $A = \text{Spy}$

```

lemma secrecy_lemma:
  "[| A ∉ bad; B ∉ bad; evs ∈ otway |]
  ==> Says Server B
    {|NA, Crypt (shrK A) {|NA, Key K|},
     Crypt (shrK B) {|NB, Key K|}|} ∈ set evs -->
    Notes Spy {|NA, NB, Key K|} ∉ set evs -->
    Key K ∉ analz (knows Spy evs)"
<proof>

```

```

lemma Spy_not_see_encrypted_key:
  "[| Says Server B
    {|NA, Crypt (shrK A) {|NA, Key K|},
     Crypt (shrK B) {|NB, Key K|}|} ∈ set evs;
    Notes Spy {|NA, NB, Key K|} ∉ set evs;
    A ∉ bad; B ∉ bad; evs ∈ otway |]
  ==> Key K ∉ analz (knows Spy evs)"
<proof>

```

### 11.3 Attempting to prove stronger properties

Only OR1 can have caused such a part of a message to appear. The premise  $A \neq B$  prevents OR2's similar-looking cryptogram from being picked up. Original Otway-Rees doesn't need it.

```

lemma Crypt_imp_OR1 [rule_format]:
  "[| A ∉ bad; A ≠ B; evs ∈ otway |]
  ==> Crypt (shrK A) {|NA, Agent A, Agent B|} ∈ parts (knows Spy evs)
  -->
    Says A B {|NA, Agent A, Agent B,
               Crypt (shrK A) {|NA, Agent A, Agent B|}|} ∈ set evs"

```

*<proof>*

Crucial property: If the encrypted message appears, and A has used NA to start a run, then it originated with the Server! The premise  $A \neq B$  allows use of *Crypt\_imp\_OR1*

Only it is FALSE. Somebody could make a fake message to Server substituting some other nonce NA' for NB.

```

lemma "[| A  $\notin$  bad; A  $\neq$  B; evs  $\in$  otway |]
  ==> Crypt (shrK A) {|NA, Key K|}  $\in$  parts (knows Spy evs) -->
    Says A B {|NA, Agent A, Agent B,
      Crypt (shrK A) {|NA, Agent A, Agent B|}|}
     $\in$  set evs -->
    ( $\exists$  B NB. Says Server B
      {|NA,
        Crypt (shrK A) {|NA, Key K|},
        Crypt (shrK B) {|NB, Key K|}|}  $\in$  set evs)"

```

*<proof>*

**end**

## 12 Bella's version of the Otway-Rees protocol

**theory** *OtwayReesBella* **imports** *Public* **begin**

Bella's modifications to a version of the Otway-Rees protocol taken from the BAN paper only concern message 7. The updated protocol makes the goal of key distribution of the session key available to A. Investigating the principle of Goal Availability undermines the BAN claim about the original protocol, that "this protocol does not make use of Kab as an encryption key, so neither principal can know whether the key is known to the other". The updated protocol makes no use of the session key to encrypt but informs A that B knows it.

**inductive\_set** *orb* :: "event list set"  
**where**

```

  Nil: "[|]  $\in$  orb"

  / Fake: "[| evsa  $\in$  orb; X  $\in$  synth (analz (knows Spy evsa)) |]
    ==> Says Spy B X # evsa  $\in$  orb"

  / Reception: "[| evsr  $\in$  orb; Says A B X  $\in$  set evsr |]
    ==> Gets B X # evsr  $\in$  orb"

  / OR1: "[| evs1  $\in$  orb; Nonce NA  $\notin$  used evs1 |]
    ==> Says A B {|Nonce M, Agent A, Agent B,
      Crypt (shrK A) {|Nonce NA, Nonce M, Agent A, Agent B|}|}
      # evs1  $\in$  orb"

  / OR2: "[| evs2  $\in$  orb; Nonce NB  $\notin$  used evs2;
    Gets B {|Nonce M, Agent A, Agent B, X|}  $\in$  set evs2 |]
    ==> Says B Server

```

```

    {Nonce M, Agent A, Agent B, X,
    Crypt (shrK B) {Nonce NB, Nonce M, Nonce M, Agent A, Agent B}}
    # evs2 ∈ orb"

/ OR3: "[evs3 ∈ orb; Key KAB ∉ used evs3;
    Gets Server
    {Nonce M, Agent A, Agent B,
    Crypt (shrK A) {Nonce NA, Nonce M, Agent A, Agent B},
    Crypt (shrK B) {Nonce NB, Nonce M, Nonce M, Agent A, Agent
B}}
    ∈ set evs3]
⇒ Says Server B {Nonce M,
    Crypt (shrK B) {Crypt (shrK A) {Nonce NA, Key KAB},
    Nonce NB, Key KAB}}
    # evs3 ∈ orb"

/ OR4: "[evs4 ∈ orb; B ≠ Server; ∀ p q. X ≠ {p, q};
    Says B Server {Nonce M, Agent A, Agent B, X',
    Crypt (shrK B) {Nonce NB, Nonce M, Nonce M, Agent A, Agent
B}}
    ∈ set evs4;
    Gets B {Nonce M, Crypt (shrK B) {X, Nonce NB, Key KAB}}
    ∈ set evs4]
⇒ Says B A {Nonce M, X} # evs4 ∈ orb"

/ Ops: "[evso ∈ orb;
    Says Server B {Nonce M,
    Crypt (shrK B) {Crypt (shrK A) {Nonce NA, Key KAB},
    Nonce NB, Key KAB}}
    ∈ set evso]
⇒ Notes Spy {Agent A, Agent B, Nonce NA, Nonce NB, Key KAB} # evso
    ∈ orb"

```

```

declare knows_Spy_partsEs [elim]
declare analz_into_parts [dest]
declare Fake_parts_insert_in_Un [dest]

```

Fragile proof, with backtracking in the possibility call.

```

lemma possibility_thm: "[A ≠ Server; B ≠ Server; Key K ∉ used[]]
    ⇒ ∃ evs ∈ orb.
    Says B A {Nonce M, Crypt (shrK A) {Nonce Na, Key K}} ∈ set evs"
<proof>

```

```

lemma Gets_imp_Says :
    "[Gets B X ∈ set evs; evs ∈ orb] ⇒ ∃ A. Says A B X ∈ set evs"
<proof>

```

```

lemma Gets_imp_knows_Spy:

```

```
"[[Gets B X ∈ set evs; evs ∈ orb]] ⇒ X ∈ knows Spy evs"
⟨proof⟩
```

```
declare Gets_imp_knows_Spy [THEN parts.Inj, dest]
```

```
lemma Gets_imp_knows:
```

```
"[[Gets B X ∈ set evs; evs ∈ orb]] ⇒ X ∈ knows B evs"
⟨proof⟩
```

```
lemma OR2_analz_knows_Spy:
```

```
"[[Gets B {Nonce M, Agent A, Agent B, X} ∈ set evs; evs ∈ orb]]
⇒ X ∈ analz (knows Spy evs)"
⟨proof⟩
```

```
lemma OR4_parts_knows_Spy:
```

```
"[[Gets B {Nonce M, Crypt (shrK B) {X, Nonce Nb, Key Kab}}] ∈ set evs;
evs ∈ orb]] ⇒ X ∈ parts (knows Spy evs)"
⟨proof⟩
```

```
lemma Oops_parts_knows_Spy:
```

```
"Says Server B {Nonce M, Crypt K' {X, Nonce Nb, K}} ∈ set evs
⇒ K ∈ parts (knows Spy evs)"
⟨proof⟩
```

```
lemmas OR2_parts_knows_Spy =
```

```
OR2_analz_knows_Spy [THEN analz_into_parts, standard]
```

```
⟨ML⟩
```

```
lemma Spy_see_shrK [simp]:
```

```
"evs ∈ orb ⇒ (Key (shrK A) ∈ parts (knows Spy evs)) = (A ∈ bad)"
⟨proof⟩
```

```
lemma Spy_analz_shrK [simp]:
```

```
"evs ∈ orb ⇒ (Key (shrK A) ∈ analz (knows Spy evs)) = (A ∈ bad)"
⟨proof⟩
```

```
lemma Spy_see_shrK_D [dest!]:
```

```
"[[Key (shrK A) ∈ parts (knows Spy evs); evs ∈ orb]] ⇒ A ∈ bad"
⟨proof⟩
```

```
lemma new_keys_not_used [simp]:
```

```
"[[Key K ∉ used evs; K ∈ symKeys; evs ∈ orb]] ⇒ K ∉ keysFor (parts (knows
Spy evs))"
⟨proof⟩
```

## 12.1 Proofs involving analz

Describes the form of K and NA when the Server sends this message. Also for Oops case.

**lemma** Says\_Server\_message\_form:

"[[Says Server B {Nonce M, Crypt (shrK B) {X, Nonce Nb, Key K}}] ∈ set evs;

evs ∈ orb]]  
 ⇒ K ∉ range shrK & (∃ A Na. X = (Crypt (shrK A) {Nonce Na, Key K}))"  
 <proof>

**lemma** Says\_Server\_imp\_Gets:

"[[Says Server B {Nonce M, Crypt (shrK B) {Crypt (shrK A) {Nonce Na, Key K},  
 Nonce Nb, Key K}}] ∈ set evs;

evs ∈ orb]]  
 ⇒ Gets Server {Nonce M, Agent A, Agent B,  
 Crypt (shrK A) {Nonce Na, Nonce M, Agent A, Agent B},  
 Crypt (shrK B) {Nonce Nb, Nonce M, Nonce M, Agent A, Agent  
 B}}  
 ∈ set evs"  
 <proof>

**lemma** A\_trusts\_OR1:

"[[Crypt (shrK A) {Nonce Na, Nonce M, Agent A, Agent B}] ∈ parts (knows Spy  
 evs);

A ∉ bad; evs ∈ orb]]  
 ⇒ Says A B {Nonce M, Agent A, Agent B, Crypt (shrK A) {Nonce Na, Nonce  
 M, Agent A, Agent B}} ∈ set evs"  
 <proof>

**lemma** B\_trusts\_OR2:

"[[Crypt (shrK B) {Nonce Nb, Nonce M, Nonce M, Agent A, Agent B}]  
 ∈ parts (knows Spy evs); B ∉ bad; evs ∈ orb]]  
 ⇒ (∃ X. Says B Server {Nonce M, Agent A, Agent B, X,  
 Crypt (shrK B) {Nonce Nb, Nonce M, Nonce M, Agent A, Agent B}}  
 ∈ set evs)"  
 <proof>

**lemma** B\_trusts\_OR3:

"[[Crypt (shrK B) {X, Nonce Nb, Key K}] ∈ parts (knows Spy evs);  
 B ∉ bad; evs ∈ orb]]  
 ⇒ ∃ M. Says Server B {Nonce M, Crypt (shrK B) {X, Nonce Nb, Key K}}  
 ∈ set evs"  
 <proof>

**lemma** Gets\_Server\_message\_form:

"[[Gets B {Nonce M, Crypt (shrK B) {X, Nonce Nb, Key K}}] ∈ set evs;  
 evs ∈ orb]]  
 ⇒ (K ∉ range shrK & (∃ A Na. X = (Crypt (shrK A) {Nonce Na, Key K}))  
 / X ∈ analz (knows Spy evs))"  
 <proof>

**lemma** unique\_Na: "[[Says A B {Nonce M, Agent A, Agent B, Crypt (shrK A) {Nonce



Na, Nonce M, Agent A, Agent B}} ∈ set evs;  
 Says A B' {Nonce M', Agent A, Agent B', Crypt (shrK A) {Nonce Na,  
 Nonce M', Agent A, Agent B'}} ∈ set evs;  
 A ∉ bad; evs ∈ orb ⟹ B=B' & M=M'"  
 <proof>

**lemma** unique\_Nb: "[Says B Server {Nonce M, Agent A, Agent B, X, Crypt (shrK B) {Nonce Nb, Nonce M, Nonce M, Agent A, Agent B}} ∈ set evs;  
 Says B Server {Nonce M', Agent A', Agent B, X', Crypt (shrK B) {Nonce Nb, Nonce M', Nonce M', Agent A', Agent B}} ∈ set evs;  
 B ∉ bad; evs ∈ orb ⟹ M=M' & A=A' & X=X'"  
 <proof>

**lemma** analz\_image\_freshCryptK\_lemma:  
 "(Crypt K X ∈ analz (Key'nE ∪ H)) ⟹ (Crypt K X ∈ analz H) ⟹  
 (Crypt K X ∈ analz (Key'nE ∪ H)) = (Crypt K X ∈ analz H)"  
 <proof>  
 <ML>

**lemma** analz\_image\_freshCryptK [rule\_format]:  
 "evs ∈ orb ⟹  
 Key K ∉ analz (knows Spy evs) ⟹  
 (∀ KK. KK ⊆ - (range shrK) ⟹  
 (Crypt K X ∈ analz (Key'KK ∪ (knows Spy evs))) =  
 (Crypt K X ∈ analz (knows Spy evs)))"  
 <proof>

**lemma** analz\_insert\_freshCryptK:  
 "[evs ∈ orb; Key K ∉ analz (knows Spy evs);  
 Seskey ∉ range shrK] ⟹  
 (Crypt K X ∈ analz (insert (Key Seskey) (knows Spy evs))) =  
 (Crypt K X ∈ analz (knows Spy evs))"  
 <proof>

**lemma** analz\_hard:  
 "[Says A B {Nonce M, Agent A, Agent B,  
 Crypt (shrK A) {Nonce Na, Nonce M, Agent A, Agent B}} ∈ set evs;  
 Crypt (shrK A) {Nonce Na, Key K} ∈ analz (knows Spy evs);  
 A ∉ bad; B ∉ bad; evs ∈ orb]  
 ⟹ Says B A {Nonce M, Crypt (shrK A) {Nonce Na, Key K}} ∈ set evs"  
 <proof>

**lemma** Gets\_Server\_message\_form':  
 "[Gets B {Nonce M, Crypt (shrK B) {X, Nonce Nb, Key K}} ∈ set evs;  
 B ∉ bad; evs ∈ orb]  
 ⟹ K ∉ range shrK & (∃ A Na. X = (Crypt (shrK A) {Nonce Na, Key K}))"

*<proof>*

**lemma** *OR4\_imp\_Gets*:

```
"[[Says B A {Nonce M, Crypt (shrK A) {Nonce Na, Key K}}] ∈ set evs;
  B ∉ bad; evs ∈ orb]]
⇒ (∃ Nb. Gets B {Nonce M, Crypt (shrK B) {Crypt (shrK A) {Nonce Na, Key
K}},
                                     Nonce Nb, Key K}} ∈ set evs)"
```

*<proof>*

**lemma** *A\_keydist\_to\_B*:

```
"[[Says A B {Nonce M, Agent A, Agent B,
  Crypt (shrK A) {Nonce Na, Nonce M, Agent A, Agent B}}] ∈ set evs;

  Gets A {Nonce M, Crypt (shrK A) {Nonce Na, Key K}} ∈ set evs;
  A ∉ bad; B ∉ bad; evs ∈ orb]]
⇒ Key K ∈ analz (knows B evs)"
```

*<proof>*

Other properties as for the original protocol

**end**

## 13 The Woo-Lam Protocol

**theory** *Woolam* **imports** *Public* **begin**

Simplified version from page 11 of Abadi and Needham (1996). Prudent Engineering Practice for Cryptographic Protocols. IEEE Trans. S.E. 22(1), pages 6-15.

Note: this differs from the Woo-Lam protocol discussed by Lowe (1996): Some New Attacks upon Security Protocols. Computer Security Foundations Workshop

**inductive\_set** *woolam* :: "event list set"

**where**

*Nil*: "*[]* ∈ *woolam*"

*/ Fake*: "*[| evsf* ∈ *woolam*; *X* ∈ *synth* (*analz* (*spies evsf*)) |]  
*==>* *Says Spy B X # evsf* ∈ *woolam*"

*/ WL1*: "*evs1* ∈ *woolam* ==> *Says A B* (*Agent A*) # *evs1* ∈ *woolam*"

*/ WL2*: "*[| evs2* ∈ *woolam*; *Says A' B* (*Agent A*) ∈ *set evs2* |]  
*==>* *Says B A* (*Nonce NB*) # *evs2* ∈ *woolam*"

```

| WL3: "[| evs3 ∈ woolam;
        Says A B (Agent A) ∈ set evs3;
        Says B' A (Nonce NB) ∈ set evs3 |]
    ==> Says A B (Crypt (shrK A) (Nonce NB)) # evs3 ∈ woolam"

| WL4: "[| evs4 ∈ woolam;
        Says A' B X ∈ set evs4;
        Says A'' B (Agent A) ∈ set evs4 |]
    ==> Says B Server {|Agent A, Agent B, X|} # evs4 ∈ woolam"

| WL5: "[| evs5 ∈ woolam;
        Says B' Server {|Agent A, Agent B, Crypt (shrK A) (Nonce NB)|}
        ∈ set evs5 |]
    ==> Says Server B (Crypt (shrK B) {|Agent A, Nonce NB|})
        # evs5 ∈ woolam"

declare Says_imp_knows_Spy [THEN analz.Inj, dest]
declare parts.Body [dest]
declare analz_into_parts [dest]
declare Fake_parts_insert_in_Un [dest]

lemma "∃NB. ∃ evs ∈ woolam.
        Says Server B (Crypt (shrK B) {|Agent A, Nonce NB|}) ∈ set evs"
<proof>

lemma Spy_see_shrK [simp]:
  "evs ∈ woolam ==> (Key (shrK A) ∈ parts (spies evs)) = (A ∈ bad)"
<proof>

lemma Spy_analz_shrK [simp]:
  "evs ∈ woolam ==> (Key (shrK A) ∈ analz (spies evs)) = (A ∈ bad)"
<proof>

lemma Spy_see_shrK_D [dest!]:
  "[|Key (shrK A) ∈ parts (knows Spy evs); evs ∈ woolam|] ==> A ∈ bad"
<proof>

```

```

lemma NB_Crypt_imp_Alice_msg:
  "[| Crypt (shrK A) (Nonce NB) ∈ parts (spies evs);
    A ∉ bad; evs ∈ woolam |]
  ==> ∃B. Says A B (Crypt (shrK A) (Nonce NB)) ∈ set evs"
  <proof>

```

```

lemma Server_trusts_WL4 [dest]:
  "[| Says B' Server {|Agent A, Agent B, Crypt (shrK A) (Nonce NB)|}
    ∈ set evs;
    A ∉ bad; evs ∈ woolam |]
  ==> ∃B. Says A B (Crypt (shrK A) (Nonce NB)) ∈ set evs"
  <proof>

```

```

lemma Server_sent_WL5 [dest]:
  "[| Says Server B (Crypt (shrK B) {|Agent A, NB|}) ∈ set evs;
    evs ∈ woolam |]
  ==> ∃B'. Says B' Server {|Agent A, Agent B, Crypt (shrK A) NB|}
    ∈ set evs"
  <proof>

```

```

lemma NB_Crypt_imp_Server_msg [rule_format]:
  "[| Crypt (shrK B) {|Agent A, NB|} ∈ parts (spies evs);
    B ∉ bad; evs ∈ woolam |]
  ==> Says Server B (Crypt (shrK B) {|Agent A, NB|}) ∈ set evs"
  <proof>

```

```

lemma B_trusts_WL5:
  "[| Says S B (Crypt (shrK B) {|Agent A, Nonce NB|}): set evs;
    A ∉ bad; B ∉ bad; evs ∈ woolam |]
  ==> ∃B. Says A B (Crypt (shrK A) (Nonce NB)) ∈ set evs"
  <proof>

```

```

lemma B_said_WL2:
  "[| Says B A (Nonce NB) ∈ set evs; B ≠ Spy; evs ∈ woolam |]
  ==> ∃A'. Says A' B (Agent A) ∈ set evs"
  <proof>

```

```

lemma "[| A ∉ bad; B ≠ Spy; evs ∈ woolam |]
  ==> Crypt (shrK A) (Nonce NB) ∈ parts (spies evs) &
  Says B A (Nonce NB) ∈ set evs"

```

```

--> Says A B (Crypt (shrK A) (Nonce NB)) ∈ set evs"
⟨proof⟩

end

```

## 14 The Otway-Bull Recursive Authentication Protocol

```
theory Recur imports Public begin
```

End marker for message bundles

abbreviation

```

END :: "msg" where
"END == Number 0"

```

inductive\_set

```

respond :: "event list => (msg*msg*key)set"
for evs :: "event list"
where
  One: "Key KAB ∉ used evs
    ==> (Hash[Key(shrK A)] {|Agent A, Agent B, Nonce NA, END|},
        {|Crypt (shrK A) {|Key KAB, Agent B, Nonce NA|}, END|},
        KAB) ∈ respond evs"

```

```

| Cons: "[| (PA, RA, KAB) ∈ respond evs;
    Key KBC ∉ used evs; Key KBC ∉ parts {RA};
    PA = Hash[Key(shrK A)] {|Agent A, Agent B, Nonce NA, P|} |]
    ==> (Hash[Key(shrK B)] {|Agent B, Agent C, Nonce NB, PA|},
        {|Crypt (shrK B) {|Key KBC, Agent C, Nonce NB|},
         Crypt (shrK B) {|Key KAB, Agent A, Nonce NB|},
         RA|},
        KBC)
    ∈ respond evs"

```

inductive\_set

```

responses :: "event list => msg set"
for evs :: "event list"
where

```

```

  Nil: "END ∈ responses evs"

```

```

| Cons: "[| RA ∈ responses evs; Key KAB ∉ used evs |]
    ==> {|Crypt (shrK B) {|Key KAB, Agent A, Nonce NB|},
        RA|} ∈ responses evs"

```

inductive\_set recur :: "event list set"

where

```

Nil: "[] ∈ recur"

| Fake: "[| evsf ∈ recur; X ∈ synth (analz (knows Spy evsf)) |]
    ==> Says Spy B X # evsf ∈ recur"

| RA1: "[| evs1 ∈ recur; Nonce NA ∉ used evs1 |]
    ==> Says A B (Hash[Key(shrK A)] {|Agent A, Agent B, Nonce NA, END|})
    # evs1 ∈ recur"

| RA2: "[| evs2 ∈ recur; Nonce NB ∉ used evs2;
    Says A' B PA ∈ set evs2 |]
    ==> Says B C (Hash[Key(shrK B)] {|Agent B, Agent C, Nonce NB, PA|})
    # evs2 ∈ recur"

| RA3: "[| evs3 ∈ recur; Says B' Server PB ∈ set evs3;
    (PB,RB,K) ∈ respond evs3 |]
    ==> Says Server B RB # evs3 ∈ recur"

| RA4: "[| evs4 ∈ recur;
    Says B C {|XH, Agent B, Agent C, Nonce NB,
              XA, Agent A, Agent B, Nonce NA, P|} ∈ set evs4;
    Says C' B {|Crypt (shrK B) {|Key KBC, Agent C, Nonce NB|},
              Crypt (shrK B) {|Key KAB, Agent A, Nonce NB|},
              RA|} ∈ set evs4 |]
    ==> Says B A RA # evs4 ∈ recur"

```

```

declare Says_imp_knows_Spy [THEN analz.Inj, dest]
declare parts.Body [dest]
declare analz_into_parts [dest]
declare Fake_parts_insert_in_Un [dest]

```

Simplest case: Alice goes directly to the server

```

lemma "Key K ∉ used []
    ==> ∃ NA. ∃ evs ∈ recur.
        Says Server A {|Crypt (shrK A) {|Key K, Agent Server, Nonce NA|},
                        END|} ∈ set evs"
<proof>

```

Case two: Alice, Bob and the server

```

lemma "[| Key K ∉ used []; Key K' ∉ used []; K ≠ K';
    Nonce NA ∉ used []; Nonce NB ∉ used []; NA < NB |]
    ==> ∃ NA. ∃ evs ∈ recur.
        Says B A {|Crypt (shrK A) {|Key K, Agent B, Nonce NA|},
                  END|} ∈ set evs"
<proof>

```

```

lemma "[| Key K  $\notin$  used []; Key K'  $\notin$  used [];
        Key K''  $\notin$  used []; K  $\neq$  K'; K'  $\neq$  K''; K  $\neq$  K'';
        Nonce NA  $\notin$  used []; Nonce NB  $\notin$  used []; Nonce NC  $\notin$  used [];
        NA < NB; NB < NC |]
  ==>  $\exists$ K.  $\exists$ NA.  $\exists$ evs  $\in$  recur.
        Says B A {|Crypt (shrK A) {|Key K, Agent B, Nonce NA|},
        END|}  $\in$  set evs"
<proof>

```

```

lemma respond_imp_not_used: "(PA,RB,KAB)  $\in$  respond evs ==> Key KAB  $\notin$  used
evs"
<proof>

```

```

lemma Key_in_parts_respond [rule_format]:
  "[| Key K  $\in$  parts {RB}; (PB,RB,K')  $\in$  respond evs |] ==> Key K  $\notin$  used
evs"
<proof>

```

Simple inductive reasoning about responses

```

lemma respond_imp_responses:
  "(PA,RB,KAB)  $\in$  respond evs ==> RB  $\in$  responses evs"
<proof>

```

```

lemmas RA2_analz_spies = Says_imp_spies [THEN analz.Inj]

```

```

lemma RA4_analz_spies:
  "Says C' B {|Crypt K X, X', RA|}  $\in$  set evs ==> RA  $\in$  analz (spies evs)"
<proof>

```

```

lemmas RA2_parts_spies = RA2_analz_spies [THEN analz_into_parts]
lemmas RA4_parts_spies = RA4_analz_spies [THEN analz_into_parts]

```

```

lemma Spy_see_shrK [simp]:
  "evs  $\in$  recur ==> (Key (shrK A)  $\in$  parts (spies evs)) = (A  $\in$  bad)"
<proof>

```

```

lemma Spy_analz_shrK [simp]:
  "evs  $\in$  recur ==> (Key (shrK A)  $\in$  analz (spies evs)) = (A  $\in$  bad)"
<proof>

```

```

lemma Spy_see_shrK_D [dest!]:
  "[| Key (shrK A) ∈ parts (knows Spy evs);  evs ∈ recur |] ==> A ∈ bad"
  <proof>

```

```

lemma resp_analz_image_freshK_lemma:
  "[| RB ∈ responses evs;
    ∀ K KK. KK ⊆ - (range shrK) -->
      (Key K ∈ analz (Key'KK Un H)) =
      (K ∈ KK | Key K ∈ analz H) |]
  ==> ∀ K KK. KK ⊆ - (range shrK) -->
      (Key K ∈ analz (insert RB (Key'KK Un H))) =
      (K ∈ KK | Key K ∈ analz (insert RB H))"
  <proof>

```

Version for the protocol. Proof is easy, thanks to the lemma.

```

lemma raw_analz_image_freshK:
  "evs ∈ recur ==>
    ∀ K KK. KK ⊆ - (range shrK) -->
      (Key K ∈ analz (Key'KK Un (spies evs))) =
      (K ∈ KK | Key K ∈ analz (spies evs))"
  <proof>

```

```

lemmas resp_analz_image_freshK =
  resp_analz_image_freshK_lemma [OF _ raw_analz_image_freshK]

```

```

lemma analz_insert_freshK:
  "[| evs ∈ recur;  KAB ∉ range shrK |]
  ==> (Key K ∈ analz (insert (Key KAB) (spies evs))) =
      (K = KAB | Key K ∈ analz (spies evs))"
  <proof>

```

Everything that's hashed is already in past traffic.

```

lemma Hash_imp_body:
  "[| Hash {|Key(shrK A), X|} ∈ parts (spies evs);
    evs ∈ recur;  A ∉ bad |] ==> X ∈ parts (spies evs)"
  <proof>

```

```

lemma unique_NA:
  "[| Hash {|Key(shrK A), Agent A, B, NA, P|} ∈ parts (spies evs);
    Hash {|Key(shrK A), Agent A, B', NA, P'|} ∈ parts (spies evs);
    evs ∈ recur;  A ∉ bad |]
  ==> B=B' & P=P'"
  <proof>

```



```

lemma shrK_in_analz_respond [simp]:
  "[| RB ∈ responses evs; evs ∈ recur |]
  ==> (Key (shrK B) ∈ analz (insert RB (spies evs))) = (B:bad)"
<proof>

```

```

lemma resp_analz_insert_lemma:
  "[| Key K ∈ analz (insert RB H);
    ∀K KK. KK ⊆ - (range shrK) -->
      (Key K ∈ analz (Key'KK Un H)) =
      (K ∈ KK | Key K ∈ analz H);
    RB ∈ responses evs |]
  ==> (Key K ∈ parts{RB} | Key K ∈ analz H)"
<proof>

```

```

lemmas resp_analz_insert =
  resp_analz_insert_lemma [OF _ raw_analz_image_freshK]

```

The last key returned by respond indeed appears in a certificate

```

lemma respond_certificate:
  "(Hash[Key(shrK A)] {|Agent A, B, NA, P|}, RA, K) ∈ respond evs
  ==> Crypt (shrK A) {|Key K, B, NA|} ∈ parts {RA}"
<proof>

```

```

lemma unique_lemma [rule_format]:
  "(PB, RB, KXY) ∈ respond evs ==>
  ∀A B N. Crypt (shrK A) {|Key K, Agent B, N|} ∈ parts {RB} -->
  (∀A' B' N'. Crypt (shrK A') {|Key K, Agent B', N'|} ∈ parts {RB} -->
  (A'=A & B'=B) | (A'=B & B'=A))"
<proof>

```

```

lemma unique_session_keys:
  "[| Crypt (shrK A) {|Key K, Agent B, N|} ∈ parts {RB};
    Crypt (shrK A') {|Key K, Agent B', N'|} ∈ parts {RB};
    (PB, RB, KXY) ∈ respond evs |]
  ==> (A'=A & B'=B) | (A'=B & B'=A)"
<proof>

```

```

lemma respond_Spy_not_see_session_key [rule_format]:
  "[| (PB, RB, KAB) ∈ respond evs; evs ∈ recur |]
  ==> ∀A A' N. A ∉ bad & A' ∉ bad -->
    Crypt (shrK A) {|Key K, Agent A', N|} ∈ parts{RB} -->
    Key K ∉ analz (insert RB (spies evs))"
<proof>

```

```

lemma Spy_not_see_session_key:
  "[| Crypt (shrK A) {|Key K, Agent A', N|} ∈ parts (spies evs);
    A ∉ bad; A' ∉ bad; evs ∈ recur |]
    ==> Key K ∉ analz (spies evs)"
  <proof>

```

The response never contains Hashes

```

lemma Hash_in_parts_respond:
  "[| Hash {|Key (shrK B), M|} ∈ parts (insert RB H);
    (PB, RB, K) ∈ respond evs |]
    ==> Hash {|Key (shrK B), M|} ∈ parts H"
  <proof>

```

Only RA1 or RA2 can have caused such a part of a message to appear. This result is of no use to B, who cannot verify the Hash. Moreover, it can say nothing about how recent A's message is. It might later be used to prove B's presence to A at the run's conclusion.

```

lemma Hash_auth_sender [rule_format]:
  "[| Hash {|Key (shrK A), Agent A, Agent B, NA, P|} ∈ parts (spies evs);
    A ∉ bad; evs ∈ recur |]
    ==> Says A B (Hash [Key (shrK A)] {|Agent A, Agent B, NA, P|}) ∈ set evs"
  <proof>

```

Certificates can only originate with the Server.

```

lemma Cert_imp_Server_msg:
  "[| Crypt (shrK A) Y ∈ parts (spies evs);
    A ∉ bad; evs ∈ recur |]
    ==> ∃ C RC. Says Server C RC ∈ set evs &
      Crypt (shrK A) Y ∈ parts {RC}"
  <proof>

```

**end**

## 15 The Yahalom Protocol

**theory** Yahalom **imports** Public **begin**

From page 257 of Burrows, Abadi and Needham (1989). A Logic of Authentication. Proc. Royal Soc. 426

This theory has the prototypical example of a secrecy relation, KeyCryptNonce.

```

inductive_set yahalom :: "event list set"
  where

```

```

  Nil: "[ ] ∈ yahalom"

```

```

  | Fake: "[| evsf ∈ yahalom; X ∈ synth (analz (knows Spy evsf)) |]
    ==> Says Spy B X # evsf ∈ yahalom"

```

```

  | Reception: "[| evsr ∈ yahalom; Says A B X ∈ set evsr |]

```

```

==> Gets B X # evsr ∈ yahalom"

/ YM1: "[| evs1 ∈ yahalom; Nonce NA ∉ used evs1 |]
==> Says A B {|Agent A, Nonce NA|} # evs1 ∈ yahalom"

/ YM2: "[| evs2 ∈ yahalom; Nonce NB ∉ used evs2;
Gets B {|Agent A, Nonce NA|} ∈ set evs2 |]
==> Says B Server
{|Agent B, Crypt (shrK B) {|Agent A, Nonce NA, Nonce NB|}|}
# evs2 ∈ yahalom"

/ YM3: "[| evs3 ∈ yahalom; Key KAB ∉ used evs3; KAB ∈ symKeys;
Gets Server
{|Agent B, Crypt (shrK B) {|Agent A, Nonce NA, Nonce NB|}|}
∈ set evs3 |]
==> Says Server A
{|Crypt (shrK A) {|Agent B, Key KAB, Nonce NA, Nonce NB|},
Crypt (shrK B) {|Agent A, Key KAB|}|}
# evs3 ∈ yahalom"

/ YM4:
— Alice receives the Server's (?) message, checks her Nonce, and uses the
new session key to send Bob his Nonce. The premise  $A \neq \text{Server}$  is needed for
Says_Server_not_range. Alice can check that K is symmetric by its length.
"[| evs4 ∈ yahalom; A ≠ Server; K ∈ symKeys;
Gets A {|Crypt(shrK A) {|Agent B, Key K, Nonce NA, Nonce NB|},
X|}
X|}
∈ set evs4;
Says A B {|Agent A, Nonce NA|} ∈ set evs4 |]
==> Says A B {|X, Crypt K (Nonce NB)|} # evs4 ∈ yahalom"

/ Ops: "[| evso ∈ yahalom;
Says Server A {|Crypt (shrK A)
{|Agent B, Key K, Nonce NA, Nonce NB|},
X|} ∈ set evso |]
==> Notes Spy {|Nonce NA, Nonce NB, Key K|} # evso ∈ yahalom"

constdefs
KeyWithNonce :: "[key, nat, event list] => bool"
"KeyWithNonce K NB evs ==
∃ A B na X.
Says Server A {|Crypt (shrK A) {|Agent B, Key K, na, Nonce NB|}, X|}
∈ set evs"

declare Says_imp_analz_Spy [dest]
declare parts.Body [dest]
declare Fake_parts_insert_in_Un [dest]

```

**declare** *analz\_into\_parts* [*dest*]

A "possibility property": there are traces that reach the end

**lemma** "[| A ≠ Server; K ∈ symKeys; Key K ∉ used [] |]  
 ==> ∃ X NB. ∃ evs ∈ yahalom.  
       Says A B {|X, Crypt K (Nonce NB)|} ∈ set evs"  
 <proof>

## 15.1 Regularity Lemmas for Yahalom

**lemma** *Gets\_imp\_Says*:  
 "[| Gets B X ∈ set evs; evs ∈ yahalom |] ==> ∃ A. Says A B X ∈ set evs"  
 <proof>

Must be proved separately for each protocol

**lemma** *Gets\_imp\_knows\_Spy*:  
 "[| Gets B X ∈ set evs; evs ∈ yahalom |] ==> X ∈ knows Spy evs"  
 <proof>

**lemmas** *Gets\_imp\_analz\_Spy* = *Gets\_imp\_knows\_Spy* [THEN *analz.Inj*]  
**declare** *Gets\_imp\_analz\_Spy* [*dest*]

Lets us treat YM4 using a similar argument as for the Fake case.

**lemma** *YM4\_analz\_knows\_Spy*:  
 "[| Gets A {|Crypt (shrK A) Y, X|} ∈ set evs; evs ∈ yahalom |]  
 ==> X ∈ analz (knows Spy evs)"  
 <proof>

**lemmas** *YM4\_parts\_knows\_Spy* =  
*YM4\_analz\_knows\_Spy* [THEN *analz\_into\_parts*, *standard*]

For Oops

**lemma** *YM4\_Key\_parts\_knows\_Spy*:  
 "Says Server A {|Crypt (shrK A) {|B,K,NA,NB|}, X|} ∈ set evs  
 ==> K ∈ parts (knows Spy evs)"  
 <proof>

Theorems of the form  $X \notin \text{parts (knows Spy evs)}$  imply that NOBODY sends messages containing X!

Spy never sees a good agent's shared key!

**lemma** *Spy\_see\_shrK* [*simp*]:  
 "evs ∈ yahalom ==> (Key (shrK A) ∈ parts (knows Spy evs)) = (A ∈ bad)"  
 <proof>

**lemma** *Spy\_analz\_shrK* [*simp*]:  
 "evs ∈ yahalom ==> (Key (shrK A) ∈ analz (knows Spy evs)) = (A ∈ bad)"  
 <proof>

**lemma** *Spy\_see\_shrK\_D* [*dest!*]:  
 "[| Key (shrK A) ∈ parts (knows Spy evs); evs ∈ yahalom |] ==> A ∈ bad"  
 <proof>

Nobody can have used non-existent keys! Needed to apply `analz_insert_Key`

```
lemma new_keys_not_used [simp]:
  "[|Key K ∉ used evs; K ∈ symKeys; evs ∈ yahalom|]
   ==> K ∉ keysFor (parts (spies evs))"
<proof>
```

Earlier, all protocol proofs declared this theorem. But only a few proofs need it, e.g. Yahalom and Kerberos IV.

```
lemma new_keys_not_analz:
  "[|K ∈ symKeys; evs ∈ yahalom; Key K ∉ used evs|]
   ==> K ∉ keysFor (analz (knows Spy evs))"
<proof>
```

Describes the form of K when the Server sends this message. Useful for Oops as well as main secrecy property.

```
lemma Says_Server_not_range [simp]:
  "[|Says Server A {|Crypt (shrK A) {|Agent B, Key K, na, nb|}, X|}
   ∈ set evs; evs ∈ yahalom |]
   ==> K ∉ range shrK"
<proof>
```

## 15.2 Secrecy Theorems

Session keys are not used to encrypt other session keys

```
lemma analz_image_freshK [rule_format]:
  "evs ∈ yahalom ==>
   ∀K KK. KK ≤ - (range shrK) -->
     (Key K ∈ analz (Key'KK Un (knows Spy evs))) =
     (K ∈ KK | Key K ∈ analz (knows Spy evs))"
<proof>
```

```
lemma analz_insert_freshK:
  "[| evs ∈ yahalom; KAB ∉ range shrK |] ==>
   (Key K ∈ analz (insert (Key KAB) (knows Spy evs))) =
   (K = KAB | Key K ∈ analz (knows Spy evs))"
<proof>
```

The Key K uniquely identifies the Server's message.

```
lemma unique_session_keys:
  "[|Says Server A
     {|Crypt (shrK A) {|Agent B, Key K, na, nb|}, X|} ∈ set evs;
   Says Server A'
     {|Crypt (shrK A') {|Agent B', Key K, na', nb'|}, X'|} ∈ set evs;
   evs ∈ yahalom |]
   ==> A=A' & B=B' & na=na' & nb=nb'"
<proof>
```

Crucial secrecy property: Spy does not see the keys sent in msg YM3

```
lemma secrecy_lemma:
  "[|A ∉ bad; B ∉ bad; evs ∈ yahalom |]
   ==> Says Server A
```

```

      {|Crypt (shrK A) {|Agent B, Key K, na, nb|},
       Crypt (shrK B) {|Agent A, Key K|}|}
    ∈ set evs -->
    Notes Spy {|na, nb, Key K|} ∉ set evs -->
    Key K ∉ analz (knows Spy evs)"
  <proof>

```

Final version

```

lemma Spy_not_see_encrypted_key:
  "[| Says Server A
     {|Crypt (shrK A) {|Agent B, Key K, na, nb|},
      Crypt (shrK B) {|Agent A, Key K|}|}
   ∈ set evs;
   Notes Spy {|na, nb, Key K|} ∉ set evs;
   A ∉ bad; B ∉ bad; evs ∈ yahalom |]
  ==> Key K ∉ analz (knows Spy evs)"
  <proof>

```

### 15.2.1 Security Guarantee for A upon receiving YM3

If the encrypted message appears then it originated with the Server

```

lemma A_trusts_YM3:
  "[| Crypt (shrK A) {|Agent B, Key K, na, nb|} ∈ parts (knows Spy evs);
   A ∉ bad; evs ∈ yahalom |]
  ==> Says Server A
     {|Crypt (shrK A) {|Agent B, Key K, na, nb|},
      Crypt (shrK B) {|Agent A, Key K|}|}
   ∈ set evs"
  <proof>

```

The obvious combination of *A\_trusts\_YM3* with *Spy\_not\_see\_encrypted\_key*

```

lemma A_gets_good_key:
  "[| Crypt (shrK A) {|Agent B, Key K, na, nb|} ∈ parts (knows Spy evs);
   Notes Spy {|na, nb, Key K|} ∉ set evs;
   A ∉ bad; B ∉ bad; evs ∈ yahalom |]
  ==> Key K ∉ analz (knows Spy evs)"
  <proof>

```

### 15.2.2 Security Guarantees for B upon receiving YM4

B knows, by the first part of A's message, that the Server distributed the key for A and B. But this part says nothing about nonces.

```

lemma B_trusts_YM4_shrK:
  "[| Crypt (shrK B) {|Agent A, Key K|} ∈ parts (knows Spy evs);
   B ∉ bad; evs ∈ yahalom |]
  ==> ∃ NA NB. Says Server A
     {|Crypt (shrK A) {|Agent B, Key K,
                      Nonce NA, Nonce NB|},
      Crypt (shrK B) {|Agent A, Key K|}|}
   ∈ set evs"
  <proof>

```

B knows, by the second part of A's message, that the Server distributed the key quoting nonce NB. This part says nothing about agent names. Secrecy of NB is crucial. Note that  $\text{Nonce NB} \notin \text{analz}(\text{knows Spy evs})$  must be the FIRST antecedent of the induction formula.

```

lemma B_trusts_YM4_newK [rule_format]:
  "[|Crypt K (Nonce NB) ∈ parts (knows Spy evs);
    Nonce NB ∉ analz (knows Spy evs); evs ∈ yahalom|]
  ==> ∃ A B NA. Says Server A
      {|Crypt (shrK A) {|Agent B, Key K, Nonce NA, Nonce NB|},
        Crypt (shrK B) {|Agent A, Key K|}|}
      ∈ set evs"
  <proof>

```

### 15.2.3 Towards proving secrecy of Nonce NB

Lemmas about the predicate KeyWithNonce

```

lemma KeyWithNonceI:
  "Says Server A
    {|Crypt (shrK A) {|Agent B, Key K, na, Nonce NB|}, X|}
    ∈ set evs ==> KeyWithNonce K NB evs"
  <proof>

```

```

lemma KeyWithNonce_Says [simp]:
  "KeyWithNonce K NB (Says S A X # evs) =
    (Server = S &
     (∃ B n X'. X = {|Crypt (shrK A) {|Agent B, Key K, n, Nonce NB|}, X'|}
      | KeyWithNonce K NB evs))"
  <proof>

```

```

lemma KeyWithNonce_Notes [simp]:
  "KeyWithNonce K NB (Notes A X # evs) = KeyWithNonce K NB evs"
  <proof>

```

```

lemma KeyWithNonce_Gets [simp]:
  "KeyWithNonce K NB (Gets A X # evs) = KeyWithNonce K NB evs"
  <proof>

```

A fresh key cannot be associated with any nonce (with respect to a given trace).

```

lemma fresh_not_KeyWithNonce:
  "Key K ∉ used evs ==> ~ KeyWithNonce K NB evs"
  <proof>

```

The Server message associates K with NB' and therefore not with any other nonce NB.

```

lemma Says_Server_KeyWithNonce:
  "[| Says Server A {|Crypt (shrK A) {|Agent B, Key K, na, Nonce NB'|}, X|}
    ∈ set evs;
    NB ≠ NB'; evs ∈ yahalom |]
  ==> ~ KeyWithNonce K NB evs"
  <proof>

```

The only nonces that can be found with the help of session keys are those distributed as nonce NB by the Server. The form of the theorem recalls *analz\_image\_freshK*, but it is much more complicated.

As with *analz\_image\_freshK*, we take some pains to express the property as a logical equivalence so that the simplifier can apply it.

**lemma** *Nonce\_secretcy\_lemma*:

```
"P --> (X ∈ analz (G Un H)) --> (X ∈ analz H) ==>
  P --> (X ∈ analz (G Un H)) = (X ∈ analz H)"
⟨proof⟩
```

**lemma** *Nonce\_secretcy*:

```
"evs ∈ yahalom ==>
  (∀ KK. KK ≤ - (range shrK) -->
    (∀ K ∈ KK. K ∈ symKeys --> ~ KeyWithNonce K NB evs) -->
    (Nonce NB ∈ analz (Key'KK Un (knows Spy evs))) =
    (Nonce NB ∈ analz (knows Spy evs)))"
⟨proof⟩
```

Version required below: if NB can be decrypted using a session key then it was distributed with that key. The more general form above is required for the induction to carry through.

**lemma** *single\_Nonce\_secretcy*:

```
"[| Says Server A
  {|Crypt (shrK A) {|Agent B, Key KAB, na, Nonce NB'|}, X|}
  ∈ set evs;
  NB ≠ NB'; KAB ∉ range shrK; evs ∈ yahalom |]
==> (Nonce NB ∈ analz (insert (Key KAB) (knows Spy evs))) =
  (Nonce NB ∈ analz (knows Spy evs))"
⟨proof⟩
```

#### 15.2.4 The Nonce NB uniquely identifies B's message.

**lemma** *unique\_NB*:

```
"[| Crypt (shrK B) {|Agent A, Nonce NA, nb|} ∈ parts (knows Spy evs);
  Crypt (shrK B') {|Agent A', Nonce NA', nb|} ∈ parts (knows Spy evs);
  evs ∈ yahalom; B ∉ bad; B' ∉ bad |]
==> NA' = NA & A' = A & B' = B"
⟨proof⟩
```

Variant useful for proving secrecy of NB. Because nb is assumed to be secret, we no longer must assume B, B' not bad.

**lemma** *Says\_unique\_NB*:

```
"[| Says C S {|X, Crypt (shrK B) {|Agent A, Nonce NA, nb|}|}
  ∈ set evs;
  Gets S' {|X', Crypt (shrK B') {|Agent A', Nonce NA', nb|}|}
  ∈ set evs;
  nb ∉ analz (knows Spy evs); evs ∈ yahalom |]
==> NA' = NA & A' = A & B' = B"
⟨proof⟩
```

#### 15.2.5 A nonce value is never used both as NA and as NB

**lemma** *no\_nonce\_YM1\_YM2*:



```

"[/Crypt (shrK B') {|Agent A', Nonce NB, nb'|} ∈ parts(knows Spy evs);
  Nonce NB ∉ analz (knows Spy evs); evs ∈ yahalom/]
==> Crypt (shrK B) {|Agent A, na, Nonce NB|} ∉ parts(knows Spy evs)"
⟨proof⟩

```

The Server sends YM3 only in response to YM2.

**lemma** Says\_Server\_imp\_YM2:

```

"[/ Says Server A {|Crypt (shrK A) {|Agent B, k, na, nb|}, X|} ∈ set
evs;
  evs ∈ yahalom /]
==> Gets Server {| Agent B, Crypt (shrK B) {|Agent A, na, nb|} |}
  ∈ set evs"
⟨proof⟩

```

A vital theorem for B, that nonce NB remains secure from the Spy.

**lemma** Spy\_not\_see\_NB :

```

"[/ Says B Server
  {|Agent B, Crypt (shrK B) {|Agent A, Nonce NA, Nonce NB|}|}
  ∈ set evs;
  (∀ k. Notes Spy {|Nonce NA, Nonce NB, k|} ∉ set evs);
  A ∉ bad; B ∉ bad; evs ∈ yahalom /]
==> Nonce NB ∉ analz (knows Spy evs)"
⟨proof⟩

```

B's session key guarantee from YM4. The two certificates contribute to a single conclusion about the Server's message. Note that the "Notes Spy" assumption must quantify over  $\forall$  POSSIBLE keys instead of our particular K. If this run is broken and the spy substitutes a certificate containing an old key, B has no means of telling.

**lemma** B\_trusts\_YM4:

```

"[/ Gets B {|Crypt (shrK B) {|Agent A, Key K|},
  Crypt K (Nonce NB)|} ∈ set evs;
  Says B Server
  {|Agent B, Crypt (shrK B) {|Agent A, Nonce NA, Nonce NB|}|}
  ∈ set evs;
  ∀ k. Notes Spy {|Nonce NA, Nonce NB, k|} ∉ set evs;
  A ∉ bad; B ∉ bad; evs ∈ yahalom /]
==> Says Server A
  {|Crypt (shrK A) {|Agent B, Key K,
    Nonce NA, Nonce NB|},
  Crypt (shrK B) {|Agent A, Key K|}|}
  ∈ set evs"
⟨proof⟩

```

The obvious combination of B\_trusts\_YM4 with Spy\_not\_see\_encrypted\_key

**lemma** B\_gets\_good\_key:

```

"[/ Gets B {|Crypt (shrK B) {|Agent A, Key K|},
  Crypt K (Nonce NB)|} ∈ set evs;
  Says B Server
  {|Agent B, Crypt (shrK B) {|Agent A, Nonce NA, Nonce NB|}|}
  ∈ set evs;
  ∀ k. Notes Spy {|Nonce NA, Nonce NB, k|} ∉ set evs;
  A ∉ bad; B ∉ bad; evs ∈ yahalom /]

```

$\Rightarrow \text{Key } K \notin \text{analz } (\text{knows Spy evs})$   
 $\langle \text{proof} \rangle$

### 15.3 Authenticating B to A

The encryption in message YM2 tells us it cannot be faked.

**lemma** *B\_Said\_YM2 [rule\_format]:*  
 $"[| \text{Crypt } (\text{shrK } B) \{ | \text{Agent } A, \text{Nonce } NA, nb | \} \in \text{parts } (\text{knows Spy evs});$   
 $\text{evs} \in \text{yahalom} |]$   
 $\Rightarrow B \notin \text{bad} \rightarrow$   
 $\text{Says } B \text{ Server } \{ | \text{Agent } B, \text{Crypt } (\text{shrK } B) \{ | \text{Agent } A, \text{Nonce } NA, nb | \} | \}$   
 $\in \text{set evs}"$   
 $\langle \text{proof} \rangle$

If the server sends YM3 then B sent YM2

**lemma** *YM3\_auth\_B\_to\_A\_lemma:*  
 $"[| \text{Says Server } A \{ | \text{Crypt } (\text{shrK } A) \{ | \text{Agent } B, \text{Key } K, \text{Nonce } NA, nb | \}, X | \}$   
 $\in \text{set evs}; \text{evs} \in \text{yahalom} |]$   
 $\Rightarrow B \notin \text{bad} \rightarrow$   
 $\text{Says } B \text{ Server } \{ | \text{Agent } B, \text{Crypt } (\text{shrK } B) \{ | \text{Agent } A, \text{Nonce } NA, nb | \} | \}$   
 $\in \text{set evs}"$   
 $\langle \text{proof} \rangle$

If A receives YM3 then B has used nonce NA (and therefore is alive)

**lemma** *YM3\_auth\_B\_to\_A:*  
 $"[| \text{Gets } A \{ | \text{Crypt } (\text{shrK } A) \{ | \text{Agent } B, \text{Key } K, \text{Nonce } NA, nb | \}, X | \}$   
 $\in \text{set evs};$   
 $A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{yahalom} |]$   
 $\Rightarrow \text{Says } B \text{ Server } \{ | \text{Agent } B, \text{Crypt } (\text{shrK } B) \{ | \text{Agent } A, \text{Nonce } NA, nb | \} | \}$   
 $\in \text{set evs}"$   
 $\langle \text{proof} \rangle$

### 15.4 Authenticating A to B using the certificate $\text{Crypt } K$ $(\text{Nonce } NB)$

Assuming the session key is secure, if both certificates are present then A has said NB. We can't be sure about the rest of A's message, but only NB matters for freshness.

**lemma** *A\_Said\_YM3\_lemma [rule\_format]:*  
 $"\text{evs} \in \text{yahalom}$   
 $\Rightarrow \text{Key } K \notin \text{analz } (\text{knows Spy evs}) \rightarrow$   
 $\text{Crypt } K (\text{Nonce } NB) \in \text{parts } (\text{knows Spy evs}) \rightarrow$   
 $\text{Crypt } (\text{shrK } B) \{ | \text{Agent } A, \text{Key } K | \} \in \text{parts } (\text{knows Spy evs}) \rightarrow$   
 $B \notin \text{bad} \rightarrow$   
 $(\exists X. \text{Says } A \text{ B } \{ | X, \text{Crypt } K (\text{Nonce } NB) | \} \in \text{set evs})"$   
 $\langle \text{proof} \rangle$

If B receives YM4 then A has used nonce NB (and therefore is alive). Moreover, A associates K with NB (thus is talking about the same run). Other premises guarantee secrecy of K.

**lemma** *YM4\_imp\_A\_Said\_YM3 [rule\_format]:*

```

"[] Gets B {|Crypt (shrK B) {|Agent A, Key K|},
           Crypt K (Nonce NB)|} ∈ set evs;
Says B Server
  {|Agent B, Crypt (shrK B) {|Agent A, Nonce NA, Nonce NB|}|}
  ∈ set evs;
(∀ NA k. Notes Spy {|Nonce NA, Nonce NB, k|} ∉ set evs);
A ∉ bad; B ∉ bad; evs ∈ yahalom []
==> ∃ X. Says A B {|X, Crypt K (Nonce NB)|} ∈ set evs"
⟨proof⟩

end

```

## 16 The Yahalom Protocol, Variant 2

**theory** *Yahalom2* **imports** *Public* **begin**

This version trades encryption of NB for additional explicitness in YM3. Also in YM3, care is taken to make the two certificates distinct.

From page 259 of Burrows, Abadi and Needham (1989). A Logic of Authentication. Proc. Royal Soc. 426

This theory has the prototypical example of a secrecy relation, KeyCryptNonce.

**inductive\_set** *yahalom* :: "event list set"

**where**

```

Nil: "[] ∈ yahalom"

| Fake: "[| evsf ∈ yahalom; X ∈ synth (analz (knows Spy evsf)) |]
  ==> Says Spy B X # evsf ∈ yahalom"

| Reception: "[| evsr ∈ yahalom; Says A B X ∈ set evsr |]
  ==> Gets B X # evsr ∈ yahalom"

| YM1: "[| evs1 ∈ yahalom; Nonce NA ∉ used evs1 |]
  ==> Says A B {|Agent A, Nonce NA|} # evs1 ∈ yahalom"

| YM2: "[| evs2 ∈ yahalom; Nonce NB ∉ used evs2;
  Gets B {|Agent A, Nonce NA|} ∈ set evs2 |]
  ==> Says B Server
    {|Agent B, Nonce NB, Crypt (shrK B) {|Agent A, Nonce NA|}|}
    # evs2 ∈ yahalom"

| YM3: "[| evs3 ∈ yahalom; Key KAB ∉ used evs3;
  Gets Server {|Agent B, Nonce NB,
    Crypt (shrK B) {|Agent A, Nonce NA|}|}
  ∈ set evs3 |]
  ==> Says Server A

```

```

    {/Nonce NB,
      Crypt (shrK A) {|Agent B, Key KAB, Nonce NA|},
      Crypt (shrK B) {|Agent A, Agent B, Key KAB, Nonce NB|}|}
    # evs3 ∈ yahalom"

| YM4: "[| evs4 ∈ yahalom;
  Gets A {|Nonce NB, Crypt (shrK A) {|Agent B, Key K, Nonce NA|},
    X|} ∈ set evs4;
  Says A B {|Agent A, Nonce NA|} ∈ set evs4 |]
==> Says A B {|X, Crypt K (Nonce NB)|} # evs4 ∈ yahalom"

| Ops: "[| evso ∈ yahalom;
  Says Server A {|Nonce NB,
    Crypt (shrK A) {|Agent B, Key K, Nonce NA|},
    X|} ∈ set evso |]
==> Notes Spy {|Nonce NA, Nonce NB, Key K|} # evso ∈ yahalom"

```

```

declare Says_imp_knows_Spy [THEN analz.Inj, dest]
declare parts.Body [dest]
declare Fake_parts_insert_in_Un [dest]
declare analz_into_parts [dest]

```

A "possibility property": there are traces that reach the end

```

lemma "Key K ∉ used []
==> ∃ X NB. ∃ evs ∈ yahalom.
  Says A B {|X, Crypt K (Nonce NB)|} ∈ set evs"
<proof>

```

```

lemma Gets_imp_Says:
  "[| Gets B X ∈ set evs; evs ∈ yahalom |] ==> ∃ A. Says A B X ∈ set evs"
<proof>

```

Must be proved separately for each protocol

```

lemma Gets_imp_knows_Spy:
  "[| Gets B X ∈ set evs; evs ∈ yahalom |] ==> X ∈ knows Spy evs"
<proof>

```

```

declare Gets_imp_knows_Spy [THEN analz.Inj, dest]

```

## 16.1 Inductive Proofs

Result for reasoning about the encrypted portion of messages. Lets us treat YM4 using a similar argument as for the Fake case.

```

lemma YM4_analz_knows_Spy:
  "[| Gets A {|NB, Crypt (shrK A) Y, X|} ∈ set evs; evs ∈ yahalom |]
==> X ∈ analz (knows Spy evs)"
<proof>

```

```

lemmas YM4_parts_knows_Spy =
  YM4_analz_knows_Spy [THEN analz_into_parts, standard]

```

Spy never sees a good agent's shared key!

```
lemma Spy_see_shrK [simp]:
  "evs ∈ yahalom ==> (Key (shrK A) ∈ parts (knows Spy evs)) = (A ∈ bad)"
<proof>
```

```
lemma Spy_analz_shrK [simp]:
  "evs ∈ yahalom ==> (Key (shrK A) ∈ analz (knows Spy evs)) = (A ∈ bad)"
<proof>
```

```
lemma Spy_see_shrK_D [dest!]:
  "[|Key (shrK A) ∈ parts (knows Spy evs); evs ∈ yahalom|] ==> A ∈ bad"
<proof>
```

Nobody can have used non-existent keys! Needed to apply `analz_insert_Key`

```
lemma new_keys_not_used [simp]:
  "[|Key K ∉ used evs; K ∈ symKeys; evs ∈ yahalom|]
  ==> K ∉ keysFor (parts (spies evs))"
<proof>
```

Describes the form of K when the Server sends this message. Useful for Oops as well as main secrecy property.

```
lemma Says_Server_message_form:
  "[|Says Server A {|nb', Crypt (shrK A) {|Agent B, Key K, na|}, X|}
   ∈ set evs; evs ∈ yahalom |]
  ==> K ∉ range shrK"
<proof>
```

```
lemma analz_image_freshK [rule_format]:
  "evs ∈ yahalom ==>
  ∀K KK. KK ≤ - (range shrK) -->
    (Key K ∈ analz (Key'KK Un (knows Spy evs))) =
    (K ∈ KK | Key K ∈ analz (knows Spy evs))"
<proof>
```

```
lemma analz_insert_freshK:
  "[| evs ∈ yahalom; KAB ∉ range shrK |] ==>
  (Key K ∈ analz (insert (Key KAB) (knows Spy evs))) =
  (K = KAB | Key K ∈ analz (knows Spy evs))"
<proof>
```

The Key K uniquely identifies the Server's message

```
lemma unique_session_keys:
  "[|Says Server A
    {|nb, Crypt (shrK A) {|Agent B, Key K, na|}, X|} ∈ set evs;
   Says Server A'
    {|nb', Crypt (shrK A') {|Agent B', Key K, na'|}, X'|} ∈ set evs;
   evs ∈ yahalom |]
  ==> A=A' & B=B' & na=na' & nb=nb'"

```

*<proof>*

## 16.2 Crucial Secrecy Property: Spy Does Not See Key $K_{AB}$

```
lemma secrecy_lemma:
  "[| A ∉ bad; B ∉ bad; evs ∈ yahalom |]
   ==> Says Server A
      {|nb, Crypt (shrK A) {|Agent B, Key K, na|},
        Crypt (shrK B) {|Agent A, Agent B, Key K, nb|}|}
      ∈ set evs -->
      Notes Spy {|na, nb, Key K|} ∉ set evs -->
      Key K ∉ analz (knows Spy evs)"
<proof>
```

Final version

```
lemma Spy_not_see_encrypted_key:
  "[| Says Server A
      {|nb, Crypt (shrK A) {|Agent B, Key K, na|},
        Crypt (shrK B) {|Agent A, Agent B, Key K, nb|}|}
      ∈ set evs;
      Notes Spy {|na, nb, Key K|} ∉ set evs;
      A ∉ bad; B ∉ bad; evs ∈ yahalom |]
   ==> Key K ∉ analz (knows Spy evs)"
<proof>
```

This form is an immediate consequence of the previous result. It is similar to the assertions established by other methods. It is equivalent to the previous result in that the Spy already has *analz* and *synth* at his disposal. However, the conclusion  $\text{Key } K \notin \text{knows Spy evs}$  appears not to be inductive: all the cases other than Fake are trivial, while Fake requires  $\text{Key } K \notin \text{analz (knows Spy evs)}$ .

```
lemma Spy_not_know_encrypted_key:
  "[| Says Server A
      {|nb, Crypt (shrK A) {|Agent B, Key K, na|},
        Crypt (shrK B) {|Agent A, Agent B, Key K, nb|}|}
      ∈ set evs;
      Notes Spy {|na, nb, Key K|} ∉ set evs;
      A ∉ bad; B ∉ bad; evs ∈ yahalom |]
   ==> Key K ∉ knows Spy evs"
<proof>
```

## 16.3 Security Guarantee for A upon receiving YM3

If the encrypted message appears then it originated with the Server. May now apply *Spy\_not\_see\_encrypted\_key*, subject to its conditions.

```
lemma A_trusts_YM3:
  "[| Crypt (shrK A) {|Agent B, Key K, na|} ∈ parts (knows Spy evs);
      A ∉ bad; evs ∈ yahalom |]
   ==> ∃nb. Says Server A
      {|nb, Crypt (shrK A) {|Agent B, Key K, na|},
        Crypt (shrK B) {|Agent A, Agent B, Key K, nb|}|}
      ∈ set evs"
<proof>
```

The obvious combination of *A\_trusts\_YM3* with *Spy\_not\_see\_encrypted\_key*

**theorem** *A\_gets\_good\_key*:

```
"[| Crypt (shrK A) {|Agent B, Key K, na|} ∈ parts (knows Spy evs);
  ∀nb. Notes Spy {|na, nb, Key K|} ∉ set evs;
  A ∉ bad; B ∉ bad; evs ∈ yahalom |]
==> Key K ∉ analz (knows Spy evs)"
```

*<proof>*

## 16.4 Security Guarantee for B upon receiving YM4

B knows, by the first part of A's message, that the Server distributed the key for A and B, and has associated it with NB.

**lemma** *B\_trusts\_YM4\_shrK*:

```
"[| Crypt (shrK B) {|Agent A, Agent B, Key K, Nonce NB|}
  ∈ parts (knows Spy evs);
  B ∉ bad; evs ∈ yahalom |]
==> ∃NA. Says Server A
    {|Nonce NB,
     Crypt (shrK A) {|Agent B, Key K, Nonce NA|},
     Crypt (shrK B) {|Agent A, Agent B, Key K, Nonce NB|}|}
  ∈ set evs"
```

*<proof>*

With this protocol variant, we don't need the 2nd part of YM4 at all: Nonce NB is available in the first part.

What can B deduce from receipt of YM4? Stronger and simpler than Yahalom because we do not have to show that NB is secret.

**lemma** *B\_trusts\_YM4*:

```
"[| Gets B {|Crypt (shrK B) {|Agent A, Agent B, Key K, Nonce NB|}, X|}
  ∈ set evs;
  A ∉ bad; B ∉ bad; evs ∈ yahalom |]
==> ∃NA. Says Server A
    {|Nonce NB,
     Crypt (shrK A) {|Agent B, Key K, Nonce NA|},
     Crypt (shrK B) {|Agent A, Agent B, Key K, Nonce NB|}|}
  ∈ set evs"
```

*<proof>*

The obvious combination of *B\_trusts\_YM4* with *Spy\_not\_see\_encrypted\_key*

**theorem** *B\_gets\_good\_key*:

```
"[| Gets B {|Crypt (shrK B) {|Agent A, Agent B, Key K, Nonce NB|}, X|}
  ∈ set evs;
  ∀na. Notes Spy {|na, Nonce NB, Key K|} ∉ set evs;
  A ∉ bad; B ∉ bad; evs ∈ yahalom |]
==> Key K ∉ analz (knows Spy evs)"
```

*<proof>*

## 16.5 Authenticating B to A

The encryption in message YM2 tells us it cannot be faked.

**lemma** *B\_Said\_YM2*:

```

"[| Crypt (shrK B) {|Agent A, Nonce NA|} ∈ parts (knows Spy evs);
  B ∉ bad; evs ∈ yahalom |]
==> ∃ NB. Says B Server {|Agent B, Nonce NB,
                        Crypt (shrK B) {|Agent A, Nonce NA|}|}
    ∈ set evs"
<proof>

```

If the server sends YM3 then B sent YM2, perhaps with a different NB

```

lemma YM3_auth_B_to_A_lemma:
  "[| Says Server A {|nb, Crypt (shrK A) {|Agent B, Key K, Nonce NA|}, X|}
    ∈ set evs;
    B ∉ bad; evs ∈ yahalom |]
  ==> ∃ nb'. Says B Server {|Agent B, nb',
                          Crypt (shrK B) {|Agent A, Nonce NA|}|}
    ∈ set evs"
<proof>

```

If A receives YM3 then B has used nonce NA (and therefore is alive)

```

theorem YM3_auth_B_to_A:
  "[| Gets A {|nb, Crypt (shrK A) {|Agent B, Key K, Nonce NA|}, X|}
    ∈ set evs;
    A ∉ bad; B ∉ bad; evs ∈ yahalom |]
  ==> ∃ nb'. Says B Server
    {|Agent B, nb', Crypt (shrK B) {|Agent A, Nonce NA|}|}
    ∈ set evs"
<proof>

```

## 16.6 Authenticating A to B

using the certificate *Crypt K (Nonce NB)*

Assuming the session key is secure, if both certificates are present then A has said NB. We can't be sure about the rest of A's message, but only NB matters for freshness. Note that *Key K ∉ analz (knows Spy evs)* must be the FIRST antecedent of the induction formula.

This lemma allows a use of *unique\_session\_keys* in the next proof, which otherwise is extremely slow.

```

lemma secure_unique_session_keys:
  "[| Crypt (shrK A) {|Agent B, Key K, na|} ∈ analz (spies evs);
    Crypt (shrK A') {|Agent B', Key K, na'|} ∈ analz (spies evs);
    Key K ∉ analz (knows Spy evs); evs ∈ yahalom |]
  ==> A=A' & B=B'"
<proof>

```

```

lemma Auth_A_to_B_lemma [rule_format]:
  "evs ∈ yahalom
  ==> Key K ∉ analz (knows Spy evs) -->
    K ∈ symKeys -->
    Crypt K (Nonce NB) ∈ parts (knows Spy evs) -->
    Crypt (shrK B) {|Agent A, Agent B, Key K, Nonce NB|}
      ∈ parts (knows Spy evs) -->

```



```

      B ∉ bad -->
      (∃X. Says A B {|X, Crypt K (Nonce NB)|} ∈ set evs)"
⟨proof⟩

```

If B receives YM4 then A has used nonce NB (and therefore is alive). Moreover, A associates K with NB (thus is talking about the same run). Other premises guarantee secrecy of K.

```

theorem YM4_imp_A_Said_YM3 [rule_format]:
  "[| Gets B {|Crypt (shrK B) {|Agent A, Agent B, Key K, Nonce NB|},
      Crypt K (Nonce NB)|} ∈ set evs;
    (∀NA. Notes Spy {|Nonce NA, Nonce NB, Key K|} ∉ set evs);
    K ∈ symKeys; A ∉ bad; B ∉ bad; evs ∈ yahalom |]
  ==> ∃X. Says A B {|X, Crypt K (Nonce NB)|} ∈ set evs"
⟨proof⟩

end

```

## 17 The Yahalom Protocol: A Flawed Version

```

theory Yahalom_Bad imports Public begin

```

Demonstrates of why Oops is necessary. This protocol can be attacked because it doesn't keep NB secret, but without Oops it can be "verified" anyway. The issues are discussed in lcp's LICS 2000 invited lecture.

```

inductive_set yahalom :: "event list set"
  where

    Nil: "[ ] ∈ yahalom"

    / Fake: "[| evsf ∈ yahalom; X ∈ synth (analz (knows Spy evsf)) |]
      ==> Says Spy B X # evsf ∈ yahalom"

    / Reception: "[| evsr ∈ yahalom; Says A B X ∈ set evsr |]
      ==> Gets B X # evsr ∈ yahalom"

    / YM1: "[| evs1 ∈ yahalom; Nonce NA ∉ used evs1 |]
      ==> Says A B {|Agent A, Nonce NA|} # evs1 ∈ yahalom"

    / YM2: "[| evs2 ∈ yahalom; Nonce NB ∉ used evs2;
      Gets B {|Agent A, Nonce NA|} ∈ set evs2 |]
      ==> Says B Server
        {|Agent B, Nonce NB, Crypt (shrK B) {|Agent A, Nonce NA|}|}
        # evs2 ∈ yahalom"

    / YM3: "[| evs3 ∈ yahalom; Key KAB ∉ used evs3; KAB ∈ symKeys;
      Gets Server
        {|Agent B, Nonce NB, Crypt (shrK B) {|Agent A, Nonce NA|}|}

```

```

      ∈ set evs3 []
==> Says Server A
      {|Crypt (shrK A) {|Agent B, Key KAB, Nonce NA, Nonce NB|},
       Crypt (shrK B) {|Agent A, Key KAB|}|}
      # evs3 ∈ yahalom"

| YM4: "[| evs4 ∈ yahalom; A ≠ Server; K ∈ symKeys;
      Gets A {|Crypt(shrK A) {|Agent B, Key K, Nonce NA, Nonce NB|},
X|}]
      ∈ set evs4;
      Says A B {|Agent A, Nonce NA|} ∈ set evs4 []
==> Says A B {|X, Crypt K (Nonce NB)|} # evs4 ∈ yahalom"

declare Says_imp_knows_Spy [THEN analz.Inj, dest]
declare parts.Body [dest]
declare Fake_parts_insert_in_Un [dest]
declare analz_into_parts [dest]

A "possibility property": there are traces that reach the end
lemma "[| A ≠ Server; Key K ∉ used []; K ∈ symKeys |]
==> ∃ X NB. ∃ evs ∈ yahalom.
      Says A B {|X, Crypt K (Nonce NB)|} ∈ set evs"
⟨proof⟩

```

### 17.1 Regularity Lemmas for Yahalom

```

lemma Gets_imp_Says:
  "[| Gets B X ∈ set evs; evs ∈ yahalom |] ==> ∃ A. Says A B X ∈ set evs"
⟨proof⟩

lemma Gets_imp_knows_Spy:
  "[| Gets B X ∈ set evs; evs ∈ yahalom |] ==> X ∈ knows Spy evs"
⟨proof⟩

declare Gets_imp_knows_Spy [THEN analz.Inj, dest]

```

### 17.2 For reasoning about the encrypted portion of messages

Lets us treat YM4 using a similar argument as for the Fake case.

```

lemma YM4_analz_knows_Spy:
  "[| Gets A {|Crypt (shrK A) Y, X|} ∈ set evs; evs ∈ yahalom |]
  ==> X ∈ analz (knows Spy evs)"
⟨proof⟩

lemmas YM4_parts_knows_Spy =
  YM4_analz_knows_Spy [THEN analz_into_parts, standard]

```

Theorems of the form  $X \notin \text{parts } (\text{knows Spy evs})$  imply that NOBODY sends messages containing X!

Spy never sees a good agent's shared key!

```
lemma Spy_see_shrK [simp]:
  "evs ∈ yahalom ==> (Key (shrK A) ∈ parts (knows Spy evs)) = (A ∈ bad)"
  <proof>
```

```
lemma Spy_analz_shrK [simp]:
  "evs ∈ yahalom ==> (Key (shrK A) ∈ analz (knows Spy evs)) = (A ∈ bad)"
  <proof>
```

```
lemma Spy_see_shrK_D [dest!]:
  "[|Key (shrK A) ∈ parts (knows Spy evs); evs ∈ yahalom|] ==> A ∈ bad"
  <proof>
```

Nobody can have used non-existent keys! Needed to apply `analz_insert_Key`

```
lemma new_keys_not_used [simp]:
  "[|Key K ∉ used evs; K ∈ symKeys; evs ∈ yahalom|]
   ==> K ∉ keysFor (parts (spies evs))"
  <proof>
```

## 17.3 Secrecy Theorems

### 17.4 Session keys are not used to encrypt other session keys

```
lemma analz_image_freshK [rule_format]:
  "evs ∈ yahalom ==>
   ∀ K KK. KK ≤ - (range shrK) -->
     (Key K ∈ analz (Key KK Un (knows Spy evs))) =
     (K ∈ KK | Key K ∈ analz (knows Spy evs))"
  <proof>
```

```
lemma analz_insert_freshK:
  "[| evs ∈ yahalom; KAB ∉ range shrK |] ==>
   (Key K ∈ analz (insert (Key KAB) (knows Spy evs))) =
   (K = KAB | Key K ∈ analz (knows Spy evs))"
  <proof>
```

The Key K uniquely identifies the Server's message.

```
lemma unique_session_keys:
  "[| Says Server A
     {|Crypt (shrK A) {|Agent B, Key K, na, nb|}, X|} ∈ set evs;
    Says Server A'
     {|Crypt (shrK A') {|Agent B', Key K, na', nb'|}, X'|} ∈ set evs;
     evs ∈ yahalom |]
   ==> A=A' & B=B' & na=na' & nb=nb'"
  <proof>
```

Crucial secrecy property: Spy does not see the keys sent in msg YM3

```
lemma secrecy_lemma:
  "[| A ∉ bad; B ∉ bad; evs ∈ yahalom |]
   ==> Says Server A
     {|Crypt (shrK A) {|Agent B, Key K, na, nb|},
      Crypt (shrK B) {|Agent A, Key K|}|}
```

```

      ∈ set evs -->
      Key K ∉ analz (knows Spy evs)"
⟨proof⟩

```

Final version

```

lemma Spy_not_see_encrypted_key:
  "[| Says Server A
      {|Crypt (shrK A) {|Agent B, Key K, na, nb|},
       Crypt (shrK B) {|Agent A, Key K|}|}
   ∈ set evs;
   A ∉ bad; B ∉ bad; evs ∈ yahalom |]
  ==> Key K ∉ analz (knows Spy evs)"
⟨proof⟩

```

### 17.5 Security Guarantee for A upon receiving YM3

If the encrypted message appears then it originated with the Server

```

lemma A_trusts_YM3:
  "[| Crypt (shrK A) {|Agent B, Key K, na, nb|} ∈ parts (knows Spy evs);
   A ∉ bad; evs ∈ yahalom |]
  ==> Says Server A
      {|Crypt (shrK A) {|Agent B, Key K, na, nb|},
       Crypt (shrK B) {|Agent A, Key K|}|}
   ∈ set evs"
⟨proof⟩

```

The obvious combination of *A\_trusts\_YM3* with *Spy\_not\_see\_encrypted\_key*

```

lemma A_gets_good_key:
  "[| Crypt (shrK A) {|Agent B, Key K, na, nb|} ∈ parts (knows Spy evs);
   A ∉ bad; B ∉ bad; evs ∈ yahalom |]
  ==> Key K ∉ analz (knows Spy evs)"
⟨proof⟩

```

### 17.6 Security Guarantees for B upon receiving YM4

B knows, by the first part of A's message, that the Server distributed the key for A and B. But this part says nothing about nonces.

```

lemma B_trusts_YM4_shrK:
  "[| Crypt (shrK B) {|Agent A, Key K|} ∈ parts (knows Spy evs);
   B ∉ bad; evs ∈ yahalom |]
  ==> ∃ NA NB. Says Server A
      {|Crypt (shrK A) {|Agent B, Key K, Nonce NA, Nonce NB|},
       Crypt (shrK B) {|Agent A, Key K|}|}
   ∈ set evs"
⟨proof⟩

```

### 17.7 The Flaw in the Model

Up to now, the reasoning is similar to standard Yahalom. Now the doubtful reasoning occurs. We should not be assuming that an unknown key is secure, but the model allows us to: there is no Oops rule to let session keys become compromised.

B knows, by the second part of A's message, that the Server distributed the key quoting nonce NB. This part says nothing about agent names. Secrecy of K is assumed; the valid Yahalom proof uses (and later proves) the secrecy of NB.

```

lemma B_trusts_YM4_newK [rule_format]:
  "[|Key K ∉ analz (knows Spy evs); evs ∈ yahalom|]
  ==> Crypt K (Nonce NB) ∈ parts (knows Spy evs) -->
    (∃ A B NA. Says Server A
      {|Crypt (shrK A) {|Agent B, Key K,
        Nonce NA, Nonce NB|},
        Crypt (shrK B) {|Agent A, Key K|}|}
      ∈ set evs)"
  <proof>

```

B's session key guarantee from YM4. The two certificates contribute to a single conclusion about the Server's message.

```

lemma B_trusts_YM4:
  "[|Gets B {|Crypt (shrK B) {|Agent A, Key K|},
    Crypt K (Nonce NB)|} ∈ set evs;
    Says B Server
      {|Agent B, Nonce NB, Crypt (shrK B) {|Agent A, Nonce NA|}|}
    ∈ set evs;
    A ∉ bad; B ∉ bad; evs ∈ yahalom |]
  ==> ∃ na nb. Says Server A
    {|Crypt (shrK A) {|Agent B, Key K, na, nb|},
     Crypt (shrK B) {|Agent A, Key K|}|}
    ∈ set evs"
  <proof>

```

The obvious combination of *B\_trusts\_YM4* with *Spy\_not\_see\_encrypted\_key*

```

lemma B_gets_good_key:
  "[|Gets B {|Crypt (shrK B) {|Agent A, Key K|},
    Crypt K (Nonce NB)|} ∈ set evs;
    Says B Server
      {|Agent B, Nonce NB, Crypt (shrK B) {|Agent A, Nonce NA|}|}
    ∈ set evs;
    A ∉ bad; B ∉ bad; evs ∈ yahalom |]
  ==> Key K ∉ analz (knows Spy evs)"
  <proof>

```

Assuming the session key is secure, if both certificates are present then A has said NB. We can't be sure about the rest of A's message, but only NB matters for freshness.

```

lemma A_Said_YM3_lemma [rule_format]:
  "evs ∈ yahalom
  ==> Key K ∉ analz (knows Spy evs) -->
    Crypt K (Nonce NB) ∈ parts (knows Spy evs) -->
    Crypt (shrK B) {|Agent A, Key K|} ∈ parts (knows Spy evs) -->
    B ∉ bad -->
    (∃ X. Says A B {|X, Crypt K (Nonce NB)|} ∈ set evs)"
  <proof>

```

If B receives YM4 then A has used nonce NB (and therefore is alive). Moreover,

A associates K with NB (thus is talking about the same run). Other premises guarantee secrecy of K.

```

lemma YM4_imp_A_Said_YM3 [rule_format]:
  "[| Gets B {|Crypt (shrK B) {|Agent A, Key K|},
    Crypt K (Nonce NB)|} ∈ set evs;
    Says B Server
    {|Agent B, Nonce NB, Crypt (shrK B) {|Agent A, Nonce NA|}|}
    ∈ set evs;
    A ∉ bad; B ∉ bad; evs ∈ yahalom |]
  ==> ∃X. Says A B {|X, Crypt K (Nonce NB)|} ∈ set evs"
  <proof>

end

```

```

theory ZhouGollmann imports Public begin

```

```

abbreviation

```

```

  TTP :: agent where "TTP == Server"

```

```

abbreviation f_sub :: nat where "f_sub == 5"

```

```

abbreviation f_nro :: nat where "f_nro == 2"

```

```

abbreviation f_nrr :: nat where "f_nrr == 3"

```

```

abbreviation f_con :: nat where "f_con == 4"

```

```

constdefs

```

```

  broken :: "agent set"

```

— the compromised honest agents; TTP is included as it's not allowed to use the protocol

```

  "broken == bad - {Spy}"

```

```

declare broken_def [simp]

```

```

inductive_set zg :: "event list set"

```

```

  where

```

```

  Nil: "[ ] ∈ zg"

```

```

  / Fake: "[| evsf ∈ zg; X ∈ synth (analz (spies evsf)) |]
    ==> Says Spy B X # evsf ∈ zg"

```

```

  / Reception: "[| evsr ∈ zg; Says A B X ∈ set evsr |] ==> Gets B X # evsr
    ∈ zg"

```

```

  / ZG1: "[| evs1 ∈ zg; Nonce L ∉ used evs1; C = Crypt K (Number m);
    K ∈ symKeys;
    NRO = Crypt (priK A) {|Number f_nro, Agent B, Nonce L, C|}|]
    ==> Says A B {|Number f_nro, Agent B, Nonce L, C, NRO|} # evs1 ∈ zg"

```

```

  / ZG2: "[| evs2 ∈ zg;

```

```

    Gets B {|Number f_nro, Agent B, Nonce L, C, NRO|} ∈ set evs2;
    NRO = Crypt (priK A) {|Number f_nro, Agent B, Nonce L, C|};
    NRR = Crypt (priK B) {|Number f_nrr, Agent A, Nonce L, C|};
    ==> Says B A {|Number f_nrr, Agent A, Nonce L, NRR|} # evs2 ∈ zg"

| ZG3: "[| evs3 ∈ zg; C = Crypt K M; K ∈ symKeys;
    Says A B {|Number f_nro, Agent B, Nonce L, C, NRO|} ∈ set evs3;
    Gets A {|Number f_nrr, Agent A, Nonce L, NRR|} ∈ set evs3;
    NRR = Crypt (priK B) {|Number f_nrr, Agent A, Nonce L, C|};
    sub_K = Crypt (priK A) {|Number f_sub, Agent B, Nonce L, Key K|};
    ==> Says A TTP {|Number f_sub, Agent B, Nonce L, Key K, sub_K|}
        # evs3 ∈ zg"

| ZG4: "[| evs4 ∈ zg; K ∈ symKeys;
    Gets TTP {|Number f_sub, Agent B, Nonce L, Key K, sub_K|}
        ∈ set evs4;
    sub_K = Crypt (priK A) {|Number f_sub, Agent B, Nonce L, Key K|};
    con_K = Crypt (priK TTP) {|Number f_con, Agent A, Agent B,
        Nonce L, Key K|};
    ==> Says TTP Spy con_K
        #
        Notes TTP {|Number f_con, Agent A, Agent B, Nonce L, Key K, con_K|}
        # evs4 ∈ zg"

declare Says_imp_knows_Spy [THEN analz.Inj, dest]
declare Fake_parts_insert_in_Un [dest]
declare analz_into_parts [dest]

declare symKey_neq_priEK [simp]
declare symKey_neq_priEK [THEN not_sym, simp]

A "possibility property": there are traces that reach the end
lemma "[|A ≠ B; TTP ≠ A; TTP ≠ B; K ∈ symKeys|] ==>
  ∃L. ∃evs ∈ zg.
    Notes TTP {|Number f_con, Agent A, Agent B, Nonce L, Key K,
      Crypt (priK TTP) {|Number f_con, Agent A, Agent B, Nonce L,
Key K|} |}
      ∈ set evs"
<proof>

```

## 17.8 Basic Lemmas

```

lemma Gets_imp_Says:
  "[| Gets B X ∈ set evs; evs ∈ zg |] ==> ∃A. Says A B X ∈ set evs"
<proof>

lemma Gets_imp_knows_Spy:
  "[| Gets B X ∈ set evs; evs ∈ zg |] ==> X ∈ spies evs"
<proof>

```

Lets us replace proofs about *used evs* by simpler proofs about *parts* (*knows*

*Spy evs*).

**lemma** *Crypt\_used\_imp\_spies*:

"[| Crypt K X ∈ used evs; evs ∈ zg |]  
 ==> Crypt K X ∈ parts (spies evs)"

⟨proof⟩

**lemma** *Notes\_TTP\_imp\_Gets*:

"[|Notes TTP {|Number f\_con, Agent A, Agent B, Nonce L, Key K, con\_K |}  
 ∈ set evs;  
 sub\_K = Crypt (priK A) {|Number f\_sub, Agent B, Nonce L, Key K|};  
 evs ∈ zg|]  
 ==> Gets TTP {|Number f\_sub, Agent B, Nonce L, Key K, sub\_K|} ∈ set evs"

⟨proof⟩

For reasoning about C, which is encrypted in message ZG2

**lemma** *ZG2\_msg\_in\_parts\_spies*:

"[|Gets B {|F, B', L, C, X|} ∈ set evs; evs ∈ zg|]  
 ==> C ∈ parts (spies evs)"

⟨proof⟩

**lemma** *Spy\_see\_priK [simp]*:

"evs ∈ zg ==> (Key (priK A) ∈ parts (spies evs)) = (A ∈ bad)"

⟨proof⟩

So that blast can use it too

**declare** *Spy\_see\_priK [THEN [2] rev\_iffD1, dest!]*

**lemma** *Spy\_analz\_priK [simp]*:

"evs ∈ zg ==> (Key (priK A) ∈ analz (spies evs)) = (A ∈ bad)"

⟨proof⟩

## 17.9 About NRO: Validity for B

Below we prove that if *NRO* exists then A definitely sent it, provided A is not broken.

Strong conclusion for a good agent

**lemma** *NRO\_validity\_good*:

"[|NRO = Crypt (priK A) {|Number f\_nro, Agent B, Nonce L, C|};  
 NRO ∈ parts (spies evs);  
 A ∉ bad; evs ∈ zg |]  
 ==> Says A B {|Number f\_nro, Agent B, Nonce L, C, NRO|} ∈ set evs"

⟨proof⟩

**lemma** *NRO\_sender*:

"[|Says A' B {|n, b, l, C, Crypt (priK A) X|} ∈ set evs; evs ∈ zg|]  
 ==> A' ∈ {A, Spy}"

⟨proof⟩

Holds also for A = *Spy*!

**theorem** *NRO\_validity*:



```

"[/Gets B {|Number f_nro, Agent B, Nonce L, C, NRO|} ∈ set evs;
  NRO = Crypt (priK A) {|Number f_nro, Agent B, Nonce L, C|};
  A ∉ broken; evs ∈ zg |]
==> Says A B {|Number f_nro, Agent B, Nonce L, C, NRO|} ∈ set evs"
<proof>

```

## 17.10 About NRR: Validity for A

Below we prove that if *NRR* exists then *B* definitely sent it, provided *B* is not broken.

Strong conclusion for a good agent

**lemma** *NRR\_validity\_good*:

```

"[/NRR = Crypt (priK B) {|Number f_nrr, Agent A, Nonce L, C|};
  NRR ∈ parts (spies evs);
  B ∉ bad; evs ∈ zg |]
==> Says B A {|Number f_nrr, Agent A, Nonce L, NRR|} ∈ set evs"
<proof>

```

**lemma** *NRR\_sender*:

```

"[/Says B' A {|n, a, l, Crypt (priK B) X|} ∈ set evs; evs ∈ zg|]
==> B' ∈ {B, Spy}"
<proof>

```

Holds also for *B* = *Spy*!

**theorem** *NRR\_validity*:

```

"[/Says B' A {|Number f_nrr, Agent A, Nonce L, NRR|} ∈ set evs;
  NRR = Crypt (priK B) {|Number f_nrr, Agent A, Nonce L, C|};
  B ∉ broken; evs ∈ zg|]
==> Says B A {|Number f_nrr, Agent A, Nonce L, NRR|} ∈ set evs"
<proof>

```

## 17.11 Proofs About *sub\_K*

Below we prove that if *sub\_K* exists then *A* definitely sent it, provided *A* is not broken.

Strong conclusion for a good agent

**lemma** *sub\_K\_validity\_good*:

```

"[/sub_K = Crypt (priK A) {|Number f_sub, Agent B, Nonce L, Key K|};
  sub_K ∈ parts (spies evs);
  A ∉ bad; evs ∈ zg |]
==> Says A TTP {|Number f_sub, Agent B, Nonce L, Key K, sub_K|} ∈ set evs"
<proof>

```

**lemma** *sub\_K\_sender*:

```

"[/Says A' TTP {|n, b, l, k, Crypt (priK A) X|} ∈ set evs; evs ∈ zg|]
==> A' ∈ {A, Spy}"
<proof>

```

Holds also for *A* = *Spy*!

**theorem** *sub\_K\_validity*:

```
"[/Gets TTP {|Number f_sub, Agent B, Nonce L, Key K, sub_K|} ∈ set evs;
  sub_K = Crypt (priK A) {|Number f_sub, Agent B, Nonce L, Key K|};
  A ∉ broken; evs ∈ zg |]
==> Says A TTP {|Number f_sub, Agent B, Nonce L, Key K, sub_K|} ∈ set
evs"
⟨proof⟩
```

### 17.12 Proofs About *con\_K*

Below we prove that if *con\_K* exists, then *TTP* has it, and therefore *A* and *B*) can get it too. Moreover, we know that *A* sent *sub\_K*

**lemma** *con\_K\_validity*:

```
"[/con_K ∈ used evs;
  con_K = Crypt (priK TTP)
    {|Number f_con, Agent A, Agent B, Nonce L, Key K|};
  evs ∈ zg |]
==> Notes TTP {|Number f_con, Agent A, Agent B, Nonce L, Key K, con_K|}
  ∈ set evs"
⟨proof⟩
```

If *TTP* holds *con\_K* then *A* sent *sub\_K*. We assume that *A* is not broken. Importantly, nothing needs to be assumed about the form of *con\_K*!

**lemma** *Notes\_TTP\_imp\_Says\_A*:

```
"[/Notes TTP {|Number f_con, Agent A, Agent B, Nonce L, Key K, con_K|}
  ∈ set evs;
  sub_K = Crypt (priK A) {|Number f_sub, Agent B, Nonce L, Key K|};
  A ∉ broken; evs ∈ zg |]
==> Says A TTP {|Number f_sub, Agent B, Nonce L, Key K, sub_K|} ∈ set
evs"
⟨proof⟩
```

If *con\_K* exists, then *A* sent *sub\_K*. We again assume that *A* is not broken.

**theorem** *B\_sub\_K\_validity*:

```
"[/con_K ∈ used evs;
  con_K = Crypt (priK TTP) {|Number f_con, Agent A, Agent B,
    Nonce L, Key K|};
  sub_K = Crypt (priK A) {|Number f_sub, Agent B, Nonce L, Key K|};
  A ∉ broken; evs ∈ zg |]
==> Says A TTP {|Number f_sub, Agent B, Nonce L, Key K, sub_K|} ∈ set
evs"
⟨proof⟩
```

### 17.13 Proving fairness

Cannot prove that, if *B* has *NRO*, then *A* has her *NRR*. It would appear that *B* has a small advantage, though it is useless to win disputes: *B* needs to present *con\_K* as well.

Strange: unicity of the label protects *A*?

**lemma** *A\_unicity*:

```
"[/NRO = Crypt (priK A) {|Number f_nro, Agent B, Nonce L, Crypt K M|};
```

```

      NRO ∈ parts (spies evs);
      Says A B {|Number f_nro, Agent B, Nonce L, Crypt K M', NRO'|}
        ∈ set evs;
      A ∉ bad; evs ∈ zg []
    ==> M'=M"
  <proof>

```

Fairness lemma: if *sub\_K* exists, then *A* holds *NRR*. Relies on unicity of labels.

```

lemma sub_K_implies_NRR:
  "[| NRO = Crypt (priK A) {|Number f_nro, Agent B, Nonce L, Crypt K M|};
    NRR = Crypt (priK B) {|Number f_nrr, Agent A, Nonce L, Crypt K M|};
    sub_K ∈ parts (spies evs);
    NRO ∈ parts (spies evs);
    sub_K = Crypt (priK A) {|Number f_sub, Agent B, Nonce L, Key K|};
    A ∉ bad; evs ∈ zg []
  ==> Gets A {|Number f_nrr, Agent A, Nonce L, NRR|} ∈ set evs"
  <proof>

```

```

lemma Crypt_used_imp_L_used:
  "[| Crypt (priK TTP) {|F, A, B, L, K|} ∈ used evs; evs ∈ zg []
  ==> L ∈ used evs"
  <proof>

```

Fairness for *A*: if *con\_K* and *NRO* exist, then *A* holds *NRR*. *A* must be uncompromised, but there is no assumption about *B*.

```

theorem A_fairness_NRO:
  "[|con_K ∈ used evs;
    NRO ∈ parts (spies evs);
    con_K = Crypt (priK TTP)
      {|Number f_con, Agent A, Agent B, Nonce L, Key K|};
    NRO = Crypt (priK A) {|Number f_nro, Agent B, Nonce L, Crypt K M|};
    NRR = Crypt (priK B) {|Number f_nrr, Agent A, Nonce L, Crypt K M|};
    A ∉ bad; evs ∈ zg []
  ==> Gets A {|Number f_nrr, Agent A, Nonce L, NRR|} ∈ set evs"
  <proof>

```

Fairness for *B*: *NRR* exists at all, then *B* holds *NRO*. *B* must be uncompromised, but there is no assumption about *A*.

```

theorem B_fairness_NRR:
  "[|NRR ∈ used evs;
    NRR = Crypt (priK B) {|Number f_nrr, Agent A, Nonce L, C|};
    NRO = Crypt (priK A) {|Number f_nro, Agent B, Nonce L, C|};
    B ∉ bad; evs ∈ zg []
  ==> Gets B {|Number f_nro, Agent B, Nonce L, C, NRO|} ∈ set evs"
  <proof>

```

If *con\_K* exists at all, then *B* can get it, by *con\_K\_validity*. Cannot conclude that also *NRO* is available to *B*, because if *A* were unfair, *A* could build message 3 without building message 1, which contains *NRO*.

**end**

## 18 Verifying the Needham-Schroeder Public-Key Protocol

```
theory NS_Public_Bad imports Public begin
```

```
inductive_set ns_public :: "event list set"
  where
```

```
  Nil: "[] ∈ ns_public"
```

```
  / Fake: "[[evsf ∈ ns_public; X ∈ synth (analz (spies evsf))]]
    ⇒ Says Spy B X # evsf ∈ ns_public"
```

```
  / NS1: "[[evs1 ∈ ns_public; Nonce NA ∉ used evs1]]
    ⇒ Says A B (Crypt (pubEK B) {Nonce NA, Agent A})
      # evs1 ∈ ns_public"
```

```
  / NS2: "[[evs2 ∈ ns_public; Nonce NB ∉ used evs2;
    Says A' B (Crypt (pubEK B) {Nonce NA, Agent A}) ∈ set evs2]]
    ⇒ Says B A (Crypt (pubEK A) {Nonce NA, Nonce NB})
      # evs2 ∈ ns_public"
```

```
  / NS3: "[[evs3 ∈ ns_public;
    Says A B (Crypt (pubEK B) {Nonce NA, Agent A}) ∈ set evs3;
    Says B' A (Crypt (pubEK A) {Nonce NA, Nonce NB}) ∈ set evs3]]
    ⇒ Says A B (Crypt (pubEK B) (Nonce NB)) # evs3 ∈ ns_public"
```

```
declare knows_Spy_partsEs [elim]
declare analz_into_parts [dest]
declare Fake_parts_insert_in_Un [dest]
declare image_eq_UN [simp]
```

```
lemma "∃ NB. ∃ evs ∈ ns_public. Says A B (Crypt (pubEK B) (Nonce NB)) ∈ set
  evs"
  <proof>
```

```
lemma Spy_see_priEK [simp]:
  "evs ∈ ns_public ⇒ (Key (priEK A) ∈ parts (spies evs)) = (A ∈ bad)"
  <proof>
```

```
lemma Spy_analz_priEK [simp]:
  "evs ∈ ns_public ⇒ (Key (priEK A) ∈ analz (spies evs)) = (A ∈ bad)"
  <proof>
```

```

lemma no_nonce_NS1_NS2 [rule_format]:
  "evs ∈ ns_public
   ⇒ Crypt (pubEK C) {NA', Nonce NA} ∈ parts (spies evs) →
     Crypt (pubEK B) {Nonce NA, Agent A} ∈ parts (spies evs) →
     Nonce NA ∈ analz (spies evs)"

```

⟨proof⟩

```

lemma unique_NA:
  "[Crypt(pubEK B) {Nonce NA, Agent A} ∈ parts(spies evs);
   Crypt(pubEK B') {Nonce NA, Agent A'} ∈ parts(spies evs);
   Nonce NA ∉ analz (spies evs); evs ∈ ns_public]
   ⇒ A=A' ∧ B=B'"

```

⟨proof⟩

```

theorem Spy_not_see_NA:
  "[Says A B (Crypt(pubEK B) {Nonce NA, Agent A}) ∈ set evs;
   A ∉ bad; B ∉ bad; evs ∈ ns_public]
   ⇒ Nonce NA ∉ analz (spies evs)"

```

⟨proof⟩

```

lemma A_trusts_NS2_lemma [rule_format]:
  "[A ∉ bad; B ∉ bad; evs ∈ ns_public]
   ⇒ Crypt (pubEK A) {Nonce NA, Nonce NB} ∈ parts (spies evs) →
     Says A B (Crypt(pubEK B) {Nonce NA, Agent A}) ∈ set evs →
     Says B A (Crypt(pubEK A) {Nonce NA, Nonce NB}) ∈ set evs"

```

⟨proof⟩

```

theorem A_trusts_NS2:
  "[Says A B (Crypt(pubEK B) {Nonce NA, Agent A}) ∈ set evs;
   Says B' A (Crypt(pubEK A) {Nonce NA, Nonce NB}) ∈ set evs;
   A ∉ bad; B ∉ bad; evs ∈ ns_public]
   ⇒ Says B A (Crypt(pubEK A) {Nonce NA, Nonce NB}) ∈ set evs"

```

⟨proof⟩

```

lemma B_trusts_NS1 [rule_format]:
  "evs ∈ ns_public
   ⇒ Crypt (pubEK B) {Nonce NA, Agent A} ∈ parts (spies evs) →
     Nonce NA ∉ analz (spies evs) →
     Says A B (Crypt (pubEK B) {Nonce NA, Agent A}) ∈ set evs"

```

⟨proof⟩

**lemma** unique\_NB [dest]:

"[[Crypt(pubEK A) {Nonce NA, Nonce NB} ∈ parts(spies evs);  
 Crypt(pubEK A') {Nonce NA', Nonce NB} ∈ parts(spies evs);  
 Nonce NB ∉ analz (spies evs); evs ∈ ns\_public]]  
 ⇒ A=A' ∧ NA=NA'"

⟨proof⟩

**theorem** Spy\_not\_see\_NB [dest]:

"[[Says B A (Crypt (pubEK A) {Nonce NA, Nonce NB}) ∈ set evs;  
 ∀ C. Says A C (Crypt (pubEK C) (Nonce NB)) ∉ set evs;  
 A ∉ bad; B ∉ bad; evs ∈ ns\_public]]  
 ⇒ Nonce NB ∉ analz (spies evs)"

⟨proof⟩

**lemma** B\_trusts\_NS3\_lemma [rule\_format]:

"[[A ∉ bad; B ∉ bad; evs ∈ ns\_public]]  
 ⇒ Crypt (pubEK B) (Nonce NB) ∈ parts (spies evs) →  
 Says B A (Crypt (pubEK A) {Nonce NA, Nonce NB}) ∈ set evs →  
 (∃ C. Says A C (Crypt (pubEK C) (Nonce NB)) ∈ set evs)"

⟨proof⟩

**theorem** B\_trusts\_NS3:

"[[Says B A (Crypt (pubEK A) {Nonce NA, Nonce NB}) ∈ set evs;  
 Says A' B (Crypt (pubEK B) (Nonce NB)) ∈ set evs;  
 A ∉ bad; B ∉ bad; evs ∈ ns\_public]]  
 ⇒ ∃ C. Says A C (Crypt (pubEK C) (Nonce NB)) ∈ set evs"

⟨proof⟩

**lemma** "[A ∉ bad; B ∉ bad; evs ∈ ns\_public]"

⇒ Says B A (Crypt (pubEK A) {Nonce NA, Nonce NB}) ∈ set evs  
 → Nonce NB ∉ analz (spies evs)"

⟨proof⟩

**end**

## 19 Verifying the Needham-Schroeder-Lowe Public-Key Protocol

```

theory NS_Public imports Public begin

inductive_set ns_public :: "event list set"
  where

    Nil: "[] ∈ ns_public"

    / Fake: "[[evsf ∈ ns_public; X ∈ synth (analz (spies evsf))]]
      ⇒ Says Spy B X # evsf ∈ ns_public"

    / NS1: "[[evs1 ∈ ns_public; Nonce NA ∉ used evs1]
      ⇒ Says A B (Crypt (pubEK B) {Nonce NA, Agent A})
      # evs1 ∈ ns_public"

    / NS2: "[[evs2 ∈ ns_public; Nonce NB ∉ used evs2;
      Says A' B (Crypt (pubEK B) {Nonce NA, Agent A}) ∈ set evs2]
      ⇒ Says B A (Crypt (pubEK A) {Nonce NA, Nonce NB, Agent B})
      # evs2 ∈ ns_public"

    / NS3: "[[evs3 ∈ ns_public;
      Says A B (Crypt (pubEK B) {Nonce NA, Agent A}) ∈ set evs3;
      Says B' A (Crypt (pubEK A) {Nonce NA, Nonce NB, Agent B})
      ∈ set evs3]
      ⇒ Says A B (Crypt (pubEK B) (Nonce NB)) # evs3 ∈ ns_public"

declare knows_Spy_partsEs [elim]
declare knows_Spy_partsEs [elim]
declare analz_into_parts [dest]
declare Fake_parts_insert_in_Un [dest]
declare image_eq_UN [simp]

lemma "∃ NB. ∃ evs ∈ ns_public. Says A B (Crypt (pubEK B) (Nonce NB)) ∈ set
  evs"
  <proof>

lemma Spy_see_priEK [simp]:
  "evs ∈ ns_public ⇒ (Key (priEK A) ∈ parts (spies evs)) = (A ∈ bad)"
  <proof>

lemma Spy_analz_priEK [simp]:
  "evs ∈ ns_public ⇒ (Key (priEK A) ∈ analz (spies evs)) = (A ∈ bad)"
  <proof>

```

## 19.1 Authenticity properties obtained from NS2

**lemma** *no\_nonce\_NS1\_NS2* [rule\_format]:  
 "evs ∈ ns\_public  
 $\implies \text{Crypt}(\text{pubEK } C) \{NA', \text{Nonce } NA, \text{Agent } D\} \in \text{parts}(\text{spies evs}) \longrightarrow$   
 $\text{Crypt}(\text{pubEK } B) \{ \text{Nonce } NA, \text{Agent } A \} \in \text{parts}(\text{spies evs}) \longrightarrow$   
 $\text{Nonce } NA \in \text{analz}(\text{spies evs})"$   
 <proof>

**lemma** *unique\_NA*:  
 "[Crypt(pubEK B) {Nonce NA, Agent A} ∈ parts(spies evs);  
 Crypt(pubEK B') {Nonce NA, Agent A'} ∈ parts(spies evs);  
 Nonce NA ∉ analz(spies evs); evs ∈ ns\_public]  
 $\implies A=A' \wedge B=B'"$   
 <proof>

**theorem** *Spy\_not\_see\_NA*:  
 "[Says A B (Crypt(pubEK B) {Nonce NA, Agent A}) ∈ set evs;  
 A ∉ bad; B ∉ bad; evs ∈ ns\_public]  
 $\implies \text{Nonce } NA \notin \text{analz}(\text{spies evs})"$   
 <proof>

**lemma** *A\_trusts\_NS2\_lemma* [rule\_format]:  
 "[A ∉ bad; B ∉ bad; evs ∈ ns\_public]  
 $\implies \text{Crypt}(\text{pubEK } A) \{ \text{Nonce } NA, \text{Nonce } NB, \text{Agent } B \} \in \text{parts}(\text{spies evs})$   
 $\longrightarrow$   
 $\text{Says } A \ B \ (\text{Crypt}(\text{pubEK } B) \{ \text{Nonce } NA, \text{Agent } A \}) \in \text{set evs} \longrightarrow$   
 $\text{Says } B \ A \ (\text{Crypt}(\text{pubEK } A) \{ \text{Nonce } NA, \text{Nonce } NB, \text{Agent } B \}) \in \text{set evs}"$   
 <proof>

**theorem** *A\_trusts\_NS2*:  
 "[Says A B (Crypt(pubEK B) {Nonce NA, Agent A}) ∈ set evs;  
 Says B' A (Crypt(pubEK A) {Nonce NA, Nonce NB, Agent B}) ∈ set evs;  
 A ∉ bad; B ∉ bad; evs ∈ ns\_public]  
 $\implies \text{Says } B \ A \ (\text{Crypt}(\text{pubEK } A) \{ \text{Nonce } NA, \text{Nonce } NB, \text{Agent } B \}) \in \text{set evs}"$   
 <proof>

**lemma** *B\_trusts\_NS1* [rule\_format]:  
 "evs ∈ ns\_public  
 $\implies \text{Crypt}(\text{pubEK } B) \{ \text{Nonce } NA, \text{Agent } A \} \in \text{parts}(\text{spies evs}) \longrightarrow$   
 $\text{Nonce } NA \notin \text{analz}(\text{spies evs}) \longrightarrow$   
 $\text{Says } A \ B \ (\text{Crypt}(\text{pubEK } B) \{ \text{Nonce } NA, \text{Agent } A \}) \in \text{set evs}"$   
 <proof>

## 19.2 Authenticity properties obtained from NS2

**lemma** *unique\_NB* [dest]:  
 "[Crypt(pubEK A) {Nonce NA, Nonce NB, Agent B} ∈ parts(spies evs);



```

    Crypt(pubEK A') {Nonce NA', Nonce NB, Agent B'} ∈ parts(spies evs);
    Nonce NB ∉ analz (spies evs); evs ∈ ns_public
  ⇒ A=A' ∧ NA=NA' ∧ B=B'"
⟨proof⟩

```

```

theorem Spy_not_see_NB [dest]:
  "([Says B A (Crypt (pubEK A) {Nonce NA, Nonce NB, Agent B}) ∈ set evs;
    A ∉ bad; B ∉ bad; evs ∈ ns_public]
  ⇒ Nonce NB ∉ analz (spies evs))"
⟨proof⟩

```

```

lemma B_trusts_NS3_lemma [rule_format]:
  "[A ∉ bad; B ∉ bad; evs ∈ ns_public] ⇒
    Crypt (pubEK B) (Nonce NB) ∈ parts (spies evs) →
    Says B A (Crypt (pubEK A) {Nonce NA, Nonce NB, Agent B}) ∈ set evs
  →
    Says A B (Crypt (pubEK B) (Nonce NB)) ∈ set evs"
⟨proof⟩

```

```

theorem B_trusts_NS3:
  "([Says B A (Crypt (pubEK A) {Nonce NA, Nonce NB, Agent B}) ∈ set evs;
    Says A' B (Crypt (pubEK B) (Nonce NB)) ∈ set evs;
    A ∉ bad; B ∉ bad; evs ∈ ns_public]
  ⇒ Says A B (Crypt (pubEK B) (Nonce NB)) ∈ set evs)"
⟨proof⟩

```

## 19.3 Overall guarantee for B

```

theorem B_trusts_protocol:
  "[A ∉ bad; B ∉ bad; evs ∈ ns_public] ⇒
    Crypt (pubEK B) (Nonce NB) ∈ parts (spies evs) →
    Says B A (Crypt (pubEK A) {Nonce NA, Nonce NB, Agent B}) ∈ set evs
  →
    Says A B (Crypt (pubEK B) {Nonce NA, Agent A}) ∈ set evs"
⟨proof⟩

end

```

## 20 Bijections $\mathbb{N} \rightarrow \mathbb{N}^2$ and $\mathbb{N} \rightarrow \mathbb{Z}$

```

theory Nat_Int_Bij
imports Main
begin

```

### 20.1 A bijection between $\mathbb{N}$ and $\mathbb{N}^2$

Definition and proofs are from [?, page 85].

```

definition nat2_to_nat:: "(nat * nat) ⇒ nat" where

```

```

"nat2_to_nat pair = (let (n,m) = pair in (n+m) * Suc (n+m) div 2 + n)"
definition nat_to_nat2:: "nat  $\Rightarrow$  (nat * nat)" where
"nat_to_nat2 = inv nat2_to_nat"

```

```

lemma dvd2_a_x_suc_a: "2 dvd a * (Suc a)"
<proof>

```

```

lemma
  assumes eq: "nat2_to_nat (u,v) = nat2_to_nat (x,y)"
  shows nat2_to_nat_help: "u+v  $\leq$  x+y"
<proof>

```

```

theorem nat2_to_nat_inj: "inj nat2_to_nat"
<proof>

```

```

lemma nat_to_nat2_surj: "surj nat_to_nat2"
<proof>

```

```

lemma gauss_sum_nat_upto: "2 * ( $\sum i \leq n :: \text{nat}. i$ ) = n * (n + 1)"
<proof>

```

```

lemma nat2_to_nat_surj: "surj nat2_to_nat"
<proof>

```

## 20.2 A bijection between $\mathbb{N}$ and $\mathbb{Z}$

```

definition nat_to_int_bij :: "nat  $\Rightarrow$  int" where
"nat_to_int_bij n = (if 2 dvd n then int(n div 2) else -int(Suc n div 2))"

```

```

definition int_to_nat_bij :: "int  $\Rightarrow$  nat" where
"int_to_nat_bij i = (if 0  $\leq$  i then 2*nat(i) else 2*nat(-i) - 1)"

```

```

lemma i2n_n2i_id: "int_to_nat_bij (nat_to_int_bij n) = n"
<proof>

```

```

lemma n2i_i2n_id: "nat_to_int_bij(int_to_nat_bij i) = i"
<proof>

```

```

lemma inv_nat_to_int_bij: "inv nat_to_int_bij = int_to_nat_bij"
<proof>

```

```

lemma inv_int_to_nat_bij: "inv int_to_nat_bij = nat_to_int_bij"
<proof>

```

```

lemma surj_nat_to_int_bij: "surj nat_to_int_bij"
<proof>

```

```

lemma surj_int_to_nat_bij: "surj int_to_nat_bij"
<proof>

```

```

lemma inj_nat_to_int_bij: "inj nat_to_int_bij"
<proof>

```

```
lemma inj_int_to_nat_bij: "inj int_to_nat_bij"
  <proof>
```

```
end
```

## 21 The TLS Protocol: Transport Layer Security

```
theory TLS imports Public Nat_Int_Bij begin
```

```
constdefs
```

```
  certificate      :: "[agent,key] => msg"
    "certificate A KA == Crypt (priSK Server) {|Agent A, Key KA|}"
```

TLS apparently does not require separate keypairs for encryption and signature. Therefore, we formalize signature as encryption using the private encryption key.

```
datatype role = ClientRole | ServerRole
```

```
consts
```

```
  PRF  :: "nat*nat*nat => nat"
```

```
  sessionK :: "(nat*nat*nat) * role => key"
```

```
abbreviation
```

```
  clientK :: "nat*nat*nat => key" where
    "clientK X == sessionK(X, ClientRole)"
```

```
abbreviation
```

```
  serverK :: "nat*nat*nat => key" where
    "serverK X == sessionK(X, ServerRole)"
```

```
specification (PRF)
```

```
  inj_PRF: "inj PRF"
  — the pseudo-random function is collision-free
  <proof>
```

```
specification (sessionK)
```

```
  inj_sessionK: "inj sessionK"
  — sessionK is collision-free; also, no clientK clashes with any serverK.
  <proof>
```

```
axioms
```

```
  — sessionK makes symmetric keys
  isSym_sessionK: "sessionK nonces ∈ symKeys"
```

— sessionK never clashes with a long-term symmetric key (they don't exist in TLS anyway)

```
  sessionK_neq_shrK [iff]: "sessionK nonces ≠ shrK A"
```

**inductive\_set** *tls* :: "event list set"

**where**

*Nil*: — The initial, empty trace  
 "[ ] ∈ *tls*"

| *Fake*: — The Spy may say anything he can say. The sender field is correct, but agents don't use that information.

"[| *evsf* ∈ *tls*; *X* ∈ synth (analz (spies *evsf*)) |]  
 ==> Says Spy *B X* # *evsf* ∈ *tls*"

| *SpyKeys*: — The spy may apply *PRF* and *sessionK* to available nonces

"[| *evsSK* ∈ *tls*;  
 {Nonce *NA*, Nonce *NB*, Nonce *M*} <= analz (spies *evsSK*) |]  
 ==> Notes Spy {| Nonce (PRF(*M*,*NA*,*NB*)),  
 Key (sessionK((*NA*,*NB*,*M*),role)) |} # *evsSK* ∈ *tls*"

| *ClientHello*:

— (7.4.1.2) PA represents *CLIENT\_VERSION*, *CIPHER\_SUITES* and *COMPRESSION\_METHODS*.

It is uninterpreted but will be confirmed in the FINISHED messages. *NA* is CLIENT RANDOM, while *SID* is *SESSION\_ID*. UNIX TIME is omitted because the protocol doesn't use it. May assume *NA* ∉ range *PRF* because CLIENT RANDOM is 28 bytes while MASTER SECRET is 48 bytes

"[| *evsCH* ∈ *tls*; Nonce *NA* ∉ used *evsCH*; *NA* ∉ range *PRF* |]  
 ==> Says A *B* {|Agent *A*, Nonce *NA*, Number *SID*, Number *PA*|}  
 # *evsCH* ∈ *tls*"

| *ServerHello*:

— 7.4.1.3 of the TLS Internet-Draft PB represents *CLIENT\_VERSION*, *CIPHER\_SUITE* and *COMPRESSION\_METHOD*. SERVER CERTIFICATE (7.4.2) is always present. *CERTIFICATE\_REQUEST* (7.4.4) is implied.

"[| *evsSH* ∈ *tls*; Nonce *NB* ∉ used *evsSH*; *NB* ∉ range *PRF*;  
 Says A' *B* {|Agent *A*, Nonce *NA*, Number *SID*, Number *PA*|}  
 ∈ set *evsSH* |]  
 ==> Says B *A* {|Nonce *NB*, Number *SID*, Number *PB*|} # *evsSH* ∈ *tls*"

| *Certificate*:

— SERVER (7.4.2) or CLIENT (7.4.6) CERTIFICATE.

"*evsC* ∈ *tls* ==> Says B *A* (certificate *B* (pubK *B*)) # *evsC* ∈ *tls*"

| *ClientKeyExch*:

— CLIENT KEY EXCHANGE (7.4.7). The client, *A*, chooses *PMS*, the PREMASTER SECRET. She encrypts *PMS* using the supplied *KB*, which ought to be pubK *B*. We assume *PMS* ∉ range *PRF* because a clash between the *PMS* and another MASTER SECRET is highly unlikely (even though both items have the same length, 48 bytes). The Note event records in the trace that she knows *PMS* (see REMARK at top).

"[| *evsCX* ∈ *tls*; Nonce *PMS* ∉ used *evsCX*; *PMS* ∉ range *PRF*;  
 Says B' *A* (certificate *B* *KB*) ∈ set *evsCX* |]  
 ==> Says A *B* (Crypt *KB* (Nonce *PMS*))  
 # Notes A {|Agent *B*, Nonce *PMS*|}  
 # *evsCX* ∈ *tls*"

| *CertVerify*:

— The optional Certificate Verify (7.4.8) message contains the specific components listed in the security analysis, F.1.1.2. It adds the pre-master-secret, which is also essential! Checking the signature, which is the only use of A's certificate, assures B of A's presence

```
"[| evsCV ∈ tls;
  Says B' A {|Nonce NB, Number SID, Number PB|} ∈ set evsCV;
  Notes A {|Agent B, Nonce PMS|} ∈ set evsCV |]
==> Says A B (Crypt (priK A) (Hash{|Nonce NB, Agent B, Nonce PMS|}))
# evsCV ∈ tls"
```

— Finally come the FINISHED messages (7.4.8), confirming PA and PB among other things. The master-secret is PRF(PMS,NA,NB). Either party may send its message first.

*/ ClientFinished:*

— The occurrence of Notes A —Agent B, Nonce PMS— stops the rule's applying when the Spy has satisfied the "Says A B" by repaying messages sent by the true client; in that case, the Spy does not know PMS and could not send ClientFinished. One could simply put  $A \neq \text{Spy}$  into the rule, but one should not expect the spy to be well-behaved.

```
"[| evsCF ∈ tls;
  Says A B {|Agent A, Nonce NA, Number SID, Number PA|}
    ∈ set evsCF;
  Says B' A {|Nonce NB, Number SID, Number PB|} ∈ set evsCF;
  Notes A {|Agent B, Nonce PMS|} ∈ set evsCF;
  M = PRF(PMS,NA,NB) |]
==> Says A B (Crypt (clientK(NA,NB,M))
  (Hash{|Number SID, Nonce M,
        Nonce NA, Number PA, Agent A,
        Nonce NB, Number PB, Agent B|}))
# evsCF ∈ tls"
```

*/ ServerFinished:*

— Keeping A' and A" distinct means B cannot even check that the two messages originate from the same source.

```
"[| evsSF ∈ tls;
  Says A' B {|Agent A, Nonce NA, Number SID, Number PA|}
    ∈ set evsSF;
  Says B A {|Nonce NB, Number SID, Number PB|} ∈ set evsSF;
  Says A'' B (Crypt (pubK B) (Nonce PMS)) ∈ set evsSF;
  M = PRF(PMS,NA,NB) |]
==> Says B A (Crypt (serverK(NA,NB,M))
  (Hash{|Number SID, Nonce M,
        Nonce NA, Number PA, Agent A,
        Nonce NB, Number PB, Agent B|}))
# evsSF ∈ tls"
```

*/ ClientAccepts:*

— Having transmitted ClientFinished and received an identical message encrypted with serverK, the client stores the parameters needed to resume this session. The "Notes A ..." premise is used to prove *Notes\_master\_imp\_Crypt\_PMS*.

```
"[| evsCA ∈ tls;
  Notes A {|Agent B, Nonce PMS|} ∈ set evsCA;
  M = PRF(PMS,NA,NB);
```

```

X = Hash{|Number SID, Nonce M,
          Nonce NA, Number PA, Agent A,
          Nonce NB, Number PB, Agent B|};
Says A B (Crypt (clientK(NA,NB,M)) X) ∈ set evsCA;
Says B' A (Crypt (serverK(NA,NB,M)) X) ∈ set evsCA []
==>
Notes A {|Number SID, Agent A, Agent B, Nonce M|} # evsCA ∈
tls"

/ ServerAccepts:
— Having transmitted ServerFinished and received an identical message en-
cryptd with clientK, the server stores the parameters needed to resume this session.
The "Says A" B ..." premise is used to prove Notes_master_imp_Crypt_PMS.
"/ evsSA ∈ tls;
A ≠ B;
Says A' B (Crypt (pubK B) (Nonce PMS)) ∈ set evsSA;
M = PRF(PMS,NA,NB);
X = Hash{|Number SID, Nonce M,
          Nonce NA, Number PA, Agent A,
          Nonce NB, Number PB, Agent B|};
Says B A (Crypt (serverK(NA,NB,M)) X) ∈ set evsSA;
Says A' B (Crypt (clientK(NA,NB,M)) X) ∈ set evsSA []
==>
Notes B {|Number SID, Agent A, Agent B, Nonce M|} # evsSA ∈
tls"

/ ClientResume:
— If A recalls the SESSION_ID, then she sends a FINISHED message using
the new nonces and stored MASTER SECRET.
"/ evsCR ∈ tls;
Says A B {|Agent A, Nonce NA, Number SID, Number PA|}: set evsCR;
Says B' A {|Nonce NB, Number SID, Number PB|} ∈ set evsCR;
Notes A {|Number SID, Agent A, Agent B, Nonce M|} ∈ set evsCR
[]
==> Says A B (Crypt (clientK(NA,NB,M))
              (Hash{|Number SID, Nonce M,
                    Nonce NA, Number PA, Agent A,
                    Nonce NB, Number PB, Agent B|}))
              # evsCR ∈ tls"

/ ServerResume:
— Resumption (7.3): If B finds the SESSION_ID then he can send a FIN-
ISHED message using the recovered MASTER SECRET
"/ evsSR ∈ tls;
Says A' B {|Agent A, Nonce NA, Number SID, Number PA|}: set evsSR;
Says B A {|Nonce NB, Number SID, Number PB|} ∈ set evsSR;
Notes B {|Number SID, Agent A, Agent B, Nonce M|} ∈ set evsSR
[]
==> Says B A (Crypt (serverK(NA,NB,M))
              (Hash{|Number SID, Nonce M,
                    Nonce NA, Number PA, Agent A,
                    Nonce NB, Number PB, Agent B|})) # evsSR
              ∈ tls"

```

*/ Oops:*

— The most plausible compromise is of an old session key. Losing the MASTER SECRET or PREMASTER SECRET is more serious but rather unlikely. The assumption  $A \neq \text{Spy}$  is essential: otherwise the Spy could learn session keys merely by replaying messages!

```
"[| evso ∈ tls; A ≠ Spy;
  Says A B (Crypt (sessionK((NA,NB,M),role)) X) ∈ set evso |]
==> Says A Spy (Key (sessionK((NA,NB,M),role))) # evso ∈ tls"
```

```
declare Says_imp_knows_Spy [THEN analz.Inj, dest]
declare parts.Body [dest]
declare analz_into_parts [dest]
declare Fake_parts_insert_in_Un [dest]
```

Automatically unfold the definition of "certificate"

```
declare certificate_def [simp]
```

Injectiveness of key-generating functions

```
declare inj_PRF [THEN inj_eq, iff]
declare inj_sessionK [THEN inj_eq, iff]
declare isSym_sessionK [simp]
```

```
lemma pubK_neq_sessionK [iff]: "publicKey b A ≠ sessionK arg"
<proof>
```

```
declare pubK_neq_sessionK [THEN not_sym, iff]
```

```
lemma priK_neq_sessionK [iff]: "invKey (publicKey b A) ≠ sessionK arg"
<proof>
```

```
declare priK_neq_sessionK [THEN not_sym, iff]
```

```
lemmas keys_distinct = pubK_neq_sessionK priK_neq_sessionK
```

## 21.1 Protocol Proofs

Possibility properties state that some traces run the protocol to the end. Four paths and 12 rules are considered.

Possibility property ending with ClientAccepts.

```
lemma "[| ∀ evs. (@ N. Nonce N ∉ used evs) ∉ range PRF; A ≠ B |]
==> ∃ SID M. ∃ evs ∈ tls.
  Notes A {|Number SID, Agent A, Agent B, Nonce M|} ∈ set evs"
<proof>
```

And one for ServerAccepts. Either FINISHED message may come first.

```
lemma "[| ∀ evs. (@ N. Nonce N ∉ used evs) ∉ range PRF; A ≠ B |]
==> ∃ SID NA PA NB PB M. ∃ evs ∈ tls."
```

*Notes B {/Number SID, Agent A, Agent B, Nonce M/} ∈ set evs"*  
 ⟨proof⟩

Another one, for CertVerify (which is optional)

**lemma** "[/ ∀ evs. (@ N. Nonce N ∉ used evs) ∉ range PRF; A ≠ B /]  
 ==> ∃ NB PMS. ∃ evs ∈ tls.  
 Says A B (Crypt (priK A) (Hash{/Nonce NB, Agent B, Nonce PMS/}))  
 ∈ set evs"  
 ⟨proof⟩

Another one, for session resumption (both ServerResume and ClientResume).  
 NO tls.Nil here: we refer to a previous session, not the empty trace.

**lemma** "[/ evs0 ∈ tls;  
 Notes A {/Number SID, Agent A, Agent B, Nonce M/} ∈ set evs0;  
 Notes B {/Number SID, Agent A, Agent B, Nonce M/} ∈ set evs0;  
 ∀ evs. (@ N. Nonce N ∉ used evs) ∉ range PRF;  
 A ≠ B /]  
 ==> ∃ NA PA NB PB X. ∃ evs ∈ tls.  
 X = Hash{/Number SID, Nonce M,  
 Nonce NA, Number PA, Agent A,  
 Nonce NB, Number PB, Agent B/} &  
 Says A B (Crypt (clientK(NA,NB,M)) X) ∈ set evs &  
 Says B A (Crypt (serverK(NA,NB,M)) X) ∈ set evs"  
 ⟨proof⟩

## 21.2 Inductive proofs about tls

Spy never sees a good agent's private key!

**lemma** *Spy\_see\_priK [simp]:*  
 "evs ∈ tls ==> (Key (privateKey b A) ∈ parts (spies evs)) = (A ∈ bad)"  
 ⟨proof⟩

**lemma** *Spy\_analz\_priK [simp]:*  
 "evs ∈ tls ==> (Key (privateKey b A) ∈ analz (spies evs)) = (A ∈ bad)"  
 ⟨proof⟩

**lemma** *Spy\_see\_priK\_D [dest!]:*  
 "[/Key (privateKey b A) ∈ parts (knows Spy evs); evs ∈ tls/] ==> A ∈ bad"  
 ⟨proof⟩

This lemma says that no false certificates exist. One might extend the model to include bogus certificates for the agents, but there seems little point in doing so: the loss of their private keys is a worse breach of security.

**lemma** *certificate\_valid:*  
 "[/ certificate B KB ∈ parts (spies evs); evs ∈ tls /] ==> KB = pubK B"  
 ⟨proof⟩

**lemmas** *CX\_KB\_is\_pubKB = Says\_imp\_spies [THEN parts.Inj, THEN certificate\_valid]*



### 21.2.1 Properties of items found in Notes

**lemma** *Notes\_Crypt\_parts\_spies*:  
 "[| Notes A {|Agent B, X|} ∈ set evs; evs ∈ tls |]  
 ==> Crypt (pubK B) X ∈ parts (spies evs)"  
 <proof>

C may be either A or B

**lemma** *Notes\_master\_imp\_Crypt\_PMS*:  
 "[| Notes C {|s, Agent A, Agent B, Nonce(PRF(PMS,NA,NB))|} ∈ set evs;  
 evs ∈ tls |]  
 ==> Crypt (pubK B) (Nonce PMS) ∈ parts (spies evs)"  
 <proof>

Compared with the theorem above, both premise and conclusion are stronger

**lemma** *Notes\_master\_imp\_Notes\_PMS*:  
 "[| Notes A {|s, Agent A, Agent B, Nonce(PRF(PMS,NA,NB))|} ∈ set evs;  
 evs ∈ tls |]  
 ==> Notes A {|Agent B, Nonce PMS|} ∈ set evs"  
 <proof>

### 21.2.2 Protocol goal: if B receives CertVerify, then A sent it

B can check A's signature if he has received A's certificate.

**lemma** *TrustCertVerify\_lemma*:  
 "[| X ∈ parts (spies evs);  
 X = Crypt (priK A) (Hash{|nb, Agent B, pms|});  
 evs ∈ tls; A ∉ bad |]  
 ==> Says A B X ∈ set evs"  
 <proof>

Final version: B checks X using the distributed KA instead of priK A

**lemma** *TrustCertVerify*:  
 "[| X ∈ parts (spies evs);  
 X = Crypt (invKey KA) (Hash{|nb, Agent B, pms|});  
 certificate A KA ∈ parts (spies evs);  
 evs ∈ tls; A ∉ bad |]  
 ==> Says A B X ∈ set evs"  
 <proof>

If CertVerify is present then A has chosen PMS.

**lemma** *UseCertVerify\_lemma*:  
 "[| Crypt (priK A) (Hash{|nb, Agent B, Nonce PMS|}) ∈ parts (spies evs);  
 evs ∈ tls; A ∉ bad |]  
 ==> Notes A {|Agent B, Nonce PMS|} ∈ set evs"  
 <proof>

Final version using the distributed KA instead of priK A

**lemma** *UseCertVerify*:  
 "[| Crypt (invKey KA) (Hash{|nb, Agent B, Nonce PMS|})  
 ∈ parts (spies evs);  
 certificate A KA ∈ parts (spies evs);

```

    evs ∈ tls; A ∉ bad []
  ==> Notes A {|Agent B, Nonce PMS|} ∈ set evs"
<proof>

```

```

lemma no_Notes_A_PRF [simp]:
  "evs ∈ tls ==> Notes A {|Agent B, Nonce (PRF x)|} ∉ set evs"
<proof>

```

```

lemma MS_imp_PMS [dest!]:
  "[| Nonce (PRF (PMS,NA,NB)) ∈ parts (spies evs); evs ∈ tls |]
  ==> Nonce PMS ∈ parts (spies evs)"
<proof>

```

### 21.2.3 Unicity results for PMS, the pre-master-secret

PMS determines B.

```

lemma Crypt_unique_PMS:
  "[| Crypt(pubK B) (Nonce PMS) ∈ parts (spies evs);
    Crypt(pubK B') (Nonce PMS) ∈ parts (spies evs);
    Nonce PMS ∉ analz (spies evs);
    evs ∈ tls |]
  ==> B=B'"
<proof>

```

In A's internal Note, PMS determines A and B.

```

lemma Notes_unique_PMS:
  "[| Notes A {|Agent B, Nonce PMS|} ∈ set evs;
    Notes A' {|Agent B', Nonce PMS|} ∈ set evs;
    evs ∈ tls |]
  ==> A=A' & B=B'"
<proof>

```

## 21.3 Secrecy Theorems

Key compromise lemma needed to prove *analz\_image\_keys*. No collection of keys can help the spy get new private keys.

```

lemma analz_image_priK [rule_format]:
  "evs ∈ tls
  ==> ∀ KK. (Key(priK B) ∈ analz (Key'KK Un (spies evs))) =
    (priK B ∈ KK | B ∈ bad)"
<proof>

```

slightly speeds up the big simplification below

```

lemma range_sessionkeys_not_priK:
  "KK <= range sessionK ==> priK B ∉ KK"
<proof>

```

Lemma for the trivial direction of the if-and-only-if

```

lemma analz_image_keys_lemma:
  "(X ∈ analz (G Un H)) --> (X ∈ analz H) ==>

```

```

(X ∈ analz (G Un H)) = (X ∈ analz H)"
⟨proof⟩

```

```

lemma analz_image_keys [rule_format]:
  "evs ∈ tls ==>
  ∀ KK. KK ≤ range sessionK -->
    (Nonce N ∈ analz (Key'KK Un (spies evs))) =
    (Nonce N ∈ analz (spies evs))"
⟨proof⟩

```

Knowing some session keys is no help in getting new nonces

```

lemma analz_insert_key [simp]:
  "evs ∈ tls ==>
  (Nonce N ∈ analz (insert (Key (sessionK z)) (spies evs))) =
  (Nonce N ∈ analz (spies evs))"
⟨proof⟩

```

### 21.3.1 Protocol goal: serverK(Na,Nb,M) and clientK(Na,Nb,M) remain secure

Lemma: session keys are never used if PMS is fresh. Nonces don't have to agree, allowing session resumption. Converse doesn't hold; revealing PMS doesn't force the keys to be sent. THEY ARE NOT SUITABLE AS SAFE ELIM RULES.

```

lemma PMS_lemma:
  "[| Nonce PMS ∉ parts (spies evs);
    K = sessionK((Na, Nb, PRF(PMS,NA,NB)), role);
    evs ∈ tls |]
  ==> Key K ∉ parts (spies evs) & (∀ Y. Crypt K Y ∉ parts (spies evs))"
⟨proof⟩

```

```

lemma PMS_sessionK_not_spied:
  "[| Key (sessionK((Na, Nb, PRF(PMS,NA,NB)), role)) ∈ parts (spies evs);
    evs ∈ tls |]
  ==> Nonce PMS ∈ parts (spies evs)"
⟨proof⟩

```

```

lemma PMS_Crypt_sessionK_not_spied:
  "[| Crypt (sessionK((Na, Nb, PRF(PMS,NA,NB)), role)) Y
    ∈ parts (spies evs); evs ∈ tls |]
  ==> Nonce PMS ∈ parts (spies evs)"
⟨proof⟩

```

Write keys are never sent if M (MASTER SECRET) is secure. Converse fails; betraying M doesn't force the keys to be sent! The strong Oops condition can be weakened later by unicity reasoning, with some effort. NO LONGER USED: see `clientK_not_spied` and `serverK_not_spied`

```

lemma sessionK_not_spied:
  "[| ∀ A. Says A Spy (Key (sessionK((NA,NB,M),role))) ∉ set evs;
    Nonce M ∉ analz (spies evs); evs ∈ tls |]
  ==> Key (sessionK((NA,NB,M),role)) ∉ parts (spies evs)"

```

*<proof>*

If A sends ClientKeyExch to an honest B, then the PMS will stay secret.

**lemma** *Spy\_not\_see\_PMS*:  
 "[| Notes A {|Agent B, Nonce PMS|} ∈ set evs;  
   evs ∈ tls; A ∉ bad; B ∉ bad |]  
 ==> Nonce PMS ∉ analz (spies evs)"  
*<proof>*

If A sends ClientKeyExch to an honest B, then the MASTER SECRET will stay secret.

**lemma** *Spy\_not\_see\_MS*:  
 "[| Notes A {|Agent B, Nonce PMS|} ∈ set evs;  
   evs ∈ tls; A ∉ bad; B ∉ bad |]  
 ==> Nonce (PRF(PMS,NA,NB)) ∉ analz (spies evs)"  
*<proof>*

### 21.3.2 Weakening the Oops conditions for leakage of clientK

If A created PMS then nobody else (except the Spy in replays) would send a message using a clientK generated from that PMS.

**lemma** *Says\_clientK\_unique*:  
 "[| Says A' B' (Crypt (clientK(Na,Nb,PRF(PMS,NA,NB))) Y) ∈ set evs;  
   Notes A {|Agent B, Nonce PMS|} ∈ set evs;  
   evs ∈ tls; A' ≠ Spy |]  
 ==> A = A'"  
*<proof>*

If A created PMS and has not leaked her clientK to the Spy, then it is completely secure: not even in parts!

**lemma** *clientK\_not\_spied*:  
 "[| Notes A {|Agent B, Nonce PMS|} ∈ set evs;  
   Says A Spy (Key (clientK(Na,Nb,PRF(PMS,NA,NB)))) ∉ set evs;  
   A ∉ bad; B ∉ bad;  
   evs ∈ tls |]  
 ==> Key (clientK(Na,Nb,PRF(PMS,NA,NB))) ∉ parts (spies evs)"  
*<proof>*

### 21.3.3 Weakening the Oops conditions for leakage of serverK

If A created PMS for B, then nobody other than B or the Spy would send a message using a serverK generated from that PMS.

**lemma** *Says\_serverK\_unique*:  
 "[| Says B' A' (Crypt (serverK(Na,Nb,PRF(PMS,NA,NB))) Y) ∈ set evs;  
   Notes A {|Agent B, Nonce PMS|} ∈ set evs;  
   evs ∈ tls; A ∉ bad; B ∉ bad; B' ≠ Spy |]  
 ==> B = B'"  
*<proof>*

If A created PMS for B, and B has not leaked his serverK to the Spy, then it is completely secure: not even in parts!

**lemma** *serverK\_not\_spied*:

```
"[| Notes A {|Agent B, Nonce PMS|} ∈ set evs;
  Says B Spy (Key(serverK(Na,Nb,PRF(PMS,NA,NB)))) ∉ set evs;
  A ∉ bad; B ∉ bad; evs ∈ tls |]
==> Key (serverK(Na,Nb,PRF(PMS,NA,NB))) ∉ parts (spies evs)"
⟨proof⟩
```

#### 21.3.4 Protocol goals: if A receives ServerFinished, then B is present and has used the quoted values PA, PB, etc. Note that it is up to A to compare PA with what she originally sent.

The mention of her name (A) in X assures A that B knows who she is.

**lemma** *TrustServerFinished [rule\_format]*:

```
"[| X = Crypt (serverK(Na,Nb,M))
  (Hash{|Number SID, Nonce M,
        Nonce Na, Number PA, Agent A,
        Nonce Nb, Number PB, Agent B|});
  M = PRF(PMS,NA,NB);
  evs ∈ tls; A ∉ bad; B ∉ bad |]
==> Says B Spy (Key(serverK(Na,Nb,M))) ∉ set evs -->
  Notes A {|Agent B, Nonce PMS|} ∈ set evs -->
  X ∈ parts (spies evs) --> Says B A X ∈ set evs"
⟨proof⟩
```

This version refers not to ServerFinished but to any message from B. We don't assume B has received CertVerify, and an intruder could have changed A's identity in all other messages, so we can't be sure that B sends his message to A. If CLIENT KEY EXCHANGE were augmented to bind A's identity with PMS, then we could replace A' by A below.

**lemma** *TrustServerMsg [rule\_format]*:

```
"[| M = PRF(PMS,NA,NB); evs ∈ tls; A ∉ bad; B ∉ bad |]
==> Says B Spy (Key(serverK(Na,Nb,M))) ∉ set evs -->
  Notes A {|Agent B, Nonce PMS|} ∈ set evs -->
  Crypt (serverK(Na,Nb,M)) Y ∈ parts (spies evs) -->
  (∃ A'. Says B A' (Crypt (serverK(Na,Nb,M)) Y) ∈ set evs)"
⟨proof⟩
```

#### 21.3.5 Protocol goal: if B receives any message encrypted with clientK then A has sent it

ASSUMING that A chose PMS. Authentication is assumed here; B cannot verify it. But if the message is ClientFinished, then B can then check the quoted values PA, PB, etc.

**lemma** *TrustClientMsg [rule\_format]*:

```
"[| M = PRF(PMS,NA,NB); evs ∈ tls; A ∉ bad; B ∉ bad |]
==> Says A Spy (Key(clientK(Na,Nb,M))) ∉ set evs -->
  Notes A {|Agent B, Nonce PMS|} ∈ set evs -->
  Crypt (clientK(Na,Nb,M)) Y ∈ parts (spies evs) -->
  Says A B (Crypt (clientK(Na,Nb,M)) Y) ∈ set evs"
⟨proof⟩
```

**21.3.6 Protocol goal:** if B receives `ClientFinished`, and if B is able to check a `CertVerify` from A, then A has used the quoted values PA, PB, etc. Even this one requires A to be uncompromised.

```
lemma AuthClientFinished:
  "[| M = PRF(PMS,NA,NB);
    Says A Spy (Key(clientK(Na,Nb,M))) ∉ set evs;
    Says A' B (Crypt (clientK(Na,Nb,M)) Y) ∈ set evs;
    certificate A KA ∈ parts (spies evs);
    Says A'' B (Crypt (invKey KA) (Hash{|nb, Agent B, Nonce PMS|}))
      ∈ set evs;
    evs ∈ tls; A ∉ bad; B ∉ bad |]
  ==> Says A B (Crypt (clientK(Na,Nb,M)) Y) ∈ set evs"
⟨proof⟩
```

end

## 22 The Certified Electronic Mail Protocol by Abadi et al.

```
theory CertifiedEmail imports Public begin
```

```
abbreviation
```

```
TTP :: agent where
  "TTP == Server"
```

```
abbreviation
```

```
RPwd :: "agent => key" where
  "RPwd == shrK"
```

```
consts
```

```
NoAuth    :: nat
TTPAuth    :: nat
SAuth      :: nat
BothAuth   :: nat
```

We formalize a fixed way of computing responses. Could be better.

```
constdefs
```

```

"response"      :: "agent => agent => nat => msg"
"response S R q == Hash {|Agent S, Key (shrK R), Nonce q|}"

inductive_set certified_mail :: "event list set"
where

  Nil: — The empty trace
    "[] ∈ certified_mail"

  / Fake: — The Spy may say anything he can say. The sender field is correct, but
  agents don't use that information.
    "[| evsf ∈ certified_mail; X ∈ synth(analz(spies evsf))|]
    ==> Says Spy B X # evsf ∈ certified_mail"

  / FakeSSL: — The Spy may open SSL sessions with TTP, who is the only agent
  equipped with the necessary credentials to serve as an SSL server.
    "[| evsfssl ∈ certified_mail; X ∈ synth(analz(spies evsfssl))|]
    ==> Notes TTP {|Agent Spy, Agent TTP, X|} # evsfssl ∈ certified_mail"

  / CM1: — The sender approaches the recipient. The message is a number.
    "[|evs1 ∈ certified_mail;
      Key K ∉ used evs1;
      K ∈ symKeys;
      Nonce q ∉ used evs1;
      hs = Hash{|Number cleartext, Nonce q, response S R q, Crypt K (Number m)|};
      S2TTP = Crypt(pubEK TTP) {|Agent S, Number BothAuth, Key K, Agent R, hs|}|]
    ==> Says S R {|Agent S, Agent TTP, Crypt K (Number m), Number BothAuth,
      Number cleartext, Nonce q, S2TTP|} # evs1
      ∈ certified_mail"

  / CM2: — The recipient records S2TTP while transmitting it and her password to TTP
  over an SSL channel.
    "[|evs2 ∈ certified_mail;
      Gets R {|Agent S, Agent TTP, em, Number BothAuth, Number cleartext,
      Nonce q, S2TTP|} ∈ set evs2;
      TTP ≠ R;
      hr = Hash {|Number cleartext, Nonce q, response S R q, em|} |]
    ==>
      Notes TTP {|Agent R, Agent TTP, S2TTP, Key(RPw R), hr|} # evs2
      ∈ certified_mail"

  / CM3: — TTP simultaneously reveals the key to the recipient and gives a receipt to
  the sender. The SSL channel does not authenticate the client (R), but TTP accepts the
  message only if the given password is that of the claimed sender, R. He replies over
  the established SSL channel.
    "[|evs3 ∈ certified_mail;
      Notes TTP {|Agent R, Agent TTP, S2TTP, Key(RPw R), hr|} ∈ set evs3;
      S2TTP = Crypt (pubEK TTP)
        {|Agent S, Number BothAuth, Key k, Agent R, hs|};
      TTP ≠ R; hs = hr; k ∈ symKeys|]
    ==>
      Notes R {|Agent TTP, Agent R, Key k, hr|} #
      Gets S (Crypt (priSK TTP) S2TTP) #

```

```

    Says TTP S (Crypt (priSK TTP) S2TTP) # evs3 ∈ certified_mail"

/ Reception:
"[|evsr ∈ certified_mail; Says A B X ∈ set evsr|]
==> Gets B X#evsr ∈ certified_mail"

declare Says_imp_knows_Spy [THEN analz.Inj, dest]
declare analz_into_parts [dest]

lemma "[| Key K ∉ used []; K ∈ symKeys |] ==>
  ∃ S2TTP. ∃ evs ∈ certified_mail.
    Says TTP S (Crypt (priSK TTP) S2TTP) ∈ set evs"
<proof>

lemma Gets_imp_Says:
  "[| Gets B X ∈ set evs; evs ∈ certified_mail |] ==> ∃ A. Says A B X ∈ set
  evs"
<proof>

lemma Gets_imp_parts_knows_Spy:
  "[|Gets A X ∈ set evs; evs ∈ certified_mail|] ==> X ∈ parts(spies evs)"
<proof>

lemma CM2_S2TTP_analz_knows_Spy:
  "[|Gets R {|Agent A, Agent B, em, Number AO, Number cleartext,
    Nonce q, S2TTP|} ∈ set evs;
    evs ∈ certified_mail|]
  ==> S2TTP ∈ analz(spies evs)"
<proof>

lemmas CM2_S2TTP_parts_knows_Spy =
  CM2_S2TTP_analz_knows_Spy [THEN analz_subset_parts [THEN subsetD]]

lemma hr_form_lemma [rule_format]:
  "evs ∈ certified_mail
  ==> hr ∉ synth (analz (spies evs)) -->
    (∀ S2TTP. Notes TTP {|Agent R, Agent TTP, S2TTP, pwd, hr|}
      ∈ set evs -->
      (∃ clt q S em. hr = Hash {|Number clt, Nonce q, response S R q, em|}))"
<proof>

Cannot strengthen the first disjunct to  $R \neq \text{Spy}$  because the fakessl rule allows
Spy to spoof the sender's name. Maybe can strengthen the second disjunct with
 $R \neq \text{Spy}$ .

lemma hr_form:
  "[|Notes TTP {|Agent R, Agent TTP, S2TTP, pwd, hr|} ∈ set evs;
    evs ∈ certified_mail|]
  ==> hr ∈ synth (analz (spies evs)) |
    (∃ clt q S em. hr = Hash {|Number clt, Nonce q, response S R q, em|}))"
<proof>

```



```

lemma Spy_dont_know_private_keys [dest!]:
  "[|Key (privateKey b A) ∈ parts (spies evs); evs ∈ certified_mail|]
    ==> A ∈ bad"
  <proof>

```

```

lemma Spy_know_private_keys_iff [simp]:
  "evs ∈ certified_mail
    ==> (Key (privateKey b A) ∈ parts (spies evs)) = (A ∈ bad)"
  <proof>

```

```

lemma Spy_dont_know_TTPKey_parts [simp]:
  "evs ∈ certified_mail ==> Key (privateKey b TTP) ∉ parts(spies evs)"
  <proof>

```

```

lemma Spy_dont_know_TTPKey_analz [simp]:
  "evs ∈ certified_mail ==> Key (privateKey b TTP) ∉ analz(spies evs)"
  <proof>

```

Thus, prove any goal that assumes that *Spy* knows a private key belonging to *TTP*

```

declare Spy_dont_know_TTPKey_parts [THEN [2] rev_notE, elim!]

```

```

lemma CM3_k_parts_knows_Spy:
  "[| evs ∈ certified_mail;
    Notes TTP {|Agent A, Agent TTP,
      Crypt (pubEK TTP) {|Agent S, Number A0, Key K,
        Agent R, hs|}, Key (RPwd R), hs|} ∈ set evs|]
    ==> Key K ∈ parts(spies evs)"
  <proof>

```

```

lemma Spy_dont_know_RPwd [rule_format]:
  "evs ∈ certified_mail ==> Key (RPwd A) ∈ parts(spies evs) --> A ∈ bad"
  <proof>

```

```

lemma Spy_know_RPwd_iff [simp]:
  "evs ∈ certified_mail ==> (Key (RPwd A) ∈ parts(spies evs)) = (A ∈ bad)"
  <proof>

```

```

lemma Spy_analz_RPwd_iff [simp]:
  "evs ∈ certified_mail ==> (Key (RPwd A) ∈ analz(spies evs)) = (A ∈ bad)"
  <proof>

```

Unused, but a guarantee of sorts

```

theorem CertAutenticity:
  "[|Crypt (priSK TTP) X ∈ parts (spies evs); evs ∈ certified_mail|]
    ==> ∃A. Says TTP A (Crypt (priSK TTP) X) ∈ set evs"
  <proof>

```

## 22.1 Proving Confidentiality Results

```

lemma analz_image_freshK [rule_format]:
  "evs ∈ certified_mail ==>
    ∀K KK. invKey (pubEK TTP) ∉ KK -->
      (Key K ∈ analz (Key'KK Un (spies evs))) =
      (K ∈ KK | Key K ∈ analz (spies evs))"
  <proof>

```

```

lemma analz_insert_freshK:
  "[| evs ∈ certified_mail; KAB ≠ invKey (pubEK TTP) |] ==>
    (Key K ∈ analz (insert (Key KAB) (spies evs))) =
    (K = KAB | Key K ∈ analz (spies evs))"
  <proof>

```

S2TTP must have originated from a valid sender provided  $K$  is secure. Proof is surprisingly hard.

```

lemma Notes_SSL_imp_used:
  "[|Notes B {|Agent A, Agent B, X|} ∈ set evs|] ==> X ∈ used evs"
  <proof>

```

```

lemma S2TTP_sender_lemma [rule_format]:
  "evs ∈ certified_mail ==>
    Key K ∉ analz (spies evs) -->
    (∀AO. Crypt (pubEK TTP)
      {|Agent S, Number AO, Key K, Agent R, hs|} ∈ used evs -->
    (∃m ctxt q.
      hs = Hash{|Number ctxt, Nonce q, response S R q, Crypt K (Number m)|}
    &
      Says S R
      {|Agent S, Agent TTP, Crypt K (Number m), Number AO,
        Number ctxt, Nonce q,
        Crypt (pubEK TTP)
        {|Agent S, Number AO, Key K, Agent R, hs |}|} ∈ set evs)))"
  <proof>

```

```

lemma S2TTP_sender:
  "[|Crypt (pubEK TTP) {|Agent S, Number AO, Key K, Agent R, hs|} ∈ used evs;
    Key K ∉ analz (spies evs);
    evs ∈ certified_mail|]
  ==> ∃m ctxt q.
    hs = Hash{|Number ctxt, Nonce q, response S R q, Crypt K (Number m)|}
  &
    Says S R
    {|Agent S, Agent TTP, Crypt K (Number m), Number AO,
      Number ctxt, Nonce q,
      Crypt (pubEK TTP)
      {|Agent S, Number AO, Key K, Agent R, hs |}|} ∈ set evs"
  <proof>

```

Nobody can have used non-existent keys!

```

lemma new_keys_not_used [simp]:
  "[|Key K ∉ used evs; K ∈ symKeys; evs ∈ certified_mail|]
   ==> K ∉ keysFor (parts (spies evs))"
<proof>

```

Less easy to prove  $m' = m$ . Maybe needs a separate unicity theorem for ciphertexts of the form  $\text{Crypt } K \text{ (Number } m)$ , where  $K$  is secure.

```

lemma Key_unique_lemma [rule_format]:
  "evs ∈ certified_mail ==>
   Key K ∉ analz (spies evs) -->
   (∀m cleartext q hs.
    Says S R
      {|Agent S, Agent TTP, Crypt K (Number m), Number AO,
       Number cleartext, Nonce q,
       Crypt (pubEK TTP) {|Agent S, Number AO, Key K, Agent R, hs|}|}
     ∈ set evs -->
    (∀m' cleartext' q' hs'.
     Says S' R'
       {|Agent S', Agent TTP, Crypt K (Number m'), Number AO',
        Number cleartext', Nonce q',
        Crypt (pubEK TTP) {|Agent S', Number AO', Key K, Agent R', hs'|}|}
      ∈ set evs --> R' = R & S' = S & AO' = AO & hs' = hs))"
<proof>

```

The key determines the sender, recipient and protocol options.

```

lemma Key_unique:
  "[|Says S R
    {|Agent S, Agent TTP, Crypt K (Number m), Number AO,
     Number cleartext, Nonce q,
     Crypt (pubEK TTP) {|Agent S, Number AO, Key K, Agent R, hs|}|}
   ∈ set evs;
   Says S' R'
    {|Agent S', Agent TTP, Crypt K (Number m'), Number AO',
     Number cleartext', Nonce q',
     Crypt (pubEK TTP) {|Agent S', Number AO', Key K, Agent R', hs'|}|}
   ∈ set evs;
   Key K ∉ analz (spies evs);
   evs ∈ certified_mail|]
   ==> R' = R & S' = S & AO' = AO & hs' = hs"
<proof>

```

## 22.2 The Guarantees for Sender and Recipient

A Sender's guarantee: If Spy gets the key then  $R$  is bad and  $S$  moreover gets his return receipt (and therefore has no grounds for complaint).

```

theorem S_fairness_bad_R:
  "[|Says S R {|Agent S, Agent TTP, Crypt K (Number m), Number AO,
    Number cleartext, Nonce q, S2TTP|} ∈ set evs;
    S2TTP = Crypt (pubEK TTP) {|Agent S, Number AO, Key K, Agent R, hs|};
    Key K ∈ analz (spies evs);
    evs ∈ certified_mail;
    S ≠ Spy|]
   ==> R ∈ bad & Gets S (Crypt (priSK TTP) S2TTP) ∈ set evs"

```

*<proof>*

Confidentially for the symmetric key

**theorem** *Spy\_not\_see\_encrypted\_key:*

```
"[/Says S R {|Agent S, Agent TTP, Crypt K (Number m), Number AO,
              Number cleartext, Nonce q, S2TTP|} ∈ set evs;
  S2TTP = Crypt (pubEK TTP) {|Agent S, Number AO, Key K, Agent R, hs|};
  evs ∈ certified_mail;
  S ≠ Spy; R ∉ bad|]
==> Key K ∉ analz(spies evs)"
```

*<proof>*

Agent R, who may be the Spy, doesn't receive the key until S has access to the return receipt.

**theorem** *S\_guarantee:*

```
"[/Says S R {|Agent S, Agent TTP, Crypt K (Number m), Number AO,
              Number cleartext, Nonce q, S2TTP|} ∈ set evs;
  S2TTP = Crypt (pubEK TTP) {|Agent S, Number AO, Key K, Agent R, hs|};
  Notes R {|Agent TTP, Agent R, Key K, hs|} ∈ set evs;
  S ≠ Spy; evs ∈ certified_mail|]
==> Gets S (Crypt (priSK TTP) S2TTP) ∈ set evs"
```

*<proof>*

If R sends message 2, and a delivery certificate exists, then R receives the necessary key. This result is also important to S, as it confirms the validity of the return receipt.

**theorem** *RR\_validity:*

```
"[/Crypt (priSK TTP) S2TTP ∈ used evs;
  S2TTP = Crypt (pubEK TTP)
    {|Agent S, Number AO, Key K, Agent R,
      Hash {|Number cleartext, Nonce q, r, em|}|};
  hr = Hash {|Number cleartext, Nonce q, r, em|};
  R ≠ Spy; evs ∈ certified_mail|]
==> Notes R {|Agent TTP, Agent R, Key K, hr|} ∈ set evs"
```

*<proof>*

end

## 23 Theory of Events for Security Protocols that use smartcards

**theory** *EventSC* imports *"../Message"* begin

**consts**

```
initState :: "agent => msg set"
```

**datatype** *card* = *Card agent*

Four new events express the traffic between an agent and his card

**datatype**

```

event = Says agent agent msg
      | Notes agent      msg
      | Gets agent      msg
      | Inputs agent card msg
      | C_Gets card      msg
      | Outpts card agent msg
      | A_Gets agent      msg

consts
  bad      :: "agent set"
  knows    :: "agent => event list => msg set"
  stolen    :: "card set"
  cloned    :: "card set"
  secureM  :: "bool"

abbreviation
  insecureM :: bool where
    "insecureM == ¬secureM"

Spy has access to his own key for spoof messages, but Server is secure

specification (bad)
  Spy_in_bad      [iff]: "Spy ∈ bad"
  Server_not_bad [iff]: "Server ∉ bad"
  ⟨proof⟩

specification (stolen)

  Card_Server_not_stolen [iff]: "Card Server ∉ stolen"
  Card_Spy_not_stolen   [iff]: "Card Spy ∉ stolen"
  ⟨proof⟩

specification (cloned)

  Card_Server_not_cloned [iff]: "Card Server ∉ cloned"
  Card_Spy_not_cloned   [iff]: "Card Spy ∉ cloned"
  ⟨proof⟩

primrec
  knows_Nil: "knows A [] = initState A"
  knows_Cons: "knows A (ev # evs) =
    (case ev of
      Says A' B X =>
        if (A=A' | A=Spy) then insert X (knows A evs) else knows A
evs
      | Notes A' X =>
        if (A=A' | (A=Spy & A'∈bad)) then insert X (knows A evs)
        else knows A evs
      | Gets A' X =>
        if (A=A' & A ≠ Spy) then insert X (knows A evs)
        else knows A evs
      | Inputs A' C X =>
        if secureM then
          if A=A' then insert X (knows A evs) else knows A evs
    )
  
```

```

      else
        if (A=A' | A=Spy) then insert X (knows A evs) else knows A
    evs
  | C_Gets C X    => knows A evs
  | Outpts C A' X =>
    if secureM then
      if A=A' then insert X (knows A evs) else knows A evs
    else
      if A=Spy then insert X (knows A evs) else knows A evs
  | A_Gets A' X    =>
    if (A=A' & A ≠ Spy) then insert X (knows A evs)
      else knows A evs)"

```

**consts**

```
used :: "event list => msg set"
```

**primrec**

```

used_Nil:  "used []          = (UN B. parts (initState B))"
used_Cons: "used (ev # evs) =
  (case ev of
    Says A B X => parts {X} ∪ (used evs)
  | Notes A X  => parts {X} ∪ (used evs)
  | Gets A X    => used evs
  | Inputs A C X => parts{X} ∪ (used evs)
  | C_Gets C X   => used evs
  | Outpts C A X => parts{X} ∪ (used evs)
  | A_Gets A X   => used evs)"

```

— *Gets* always follows *Says* in real protocols. Likewise, *C\_Gets* will always have to follow *Inputs* and *A\_Gets* will always have to follow *Outpts*

**lemma** *Notes\_imp\_used* [rule\_format]: "Notes A X ∈ set evs ⟶ X ∈ used evs"  
 <proof>

**lemma** *Says\_imp\_used* [rule\_format]: "Says A B X ∈ set evs ⟶ X ∈ used evs"  
 <proof>

**lemma** *MPair\_used* [rule\_format]:  
 "MPair X Y ∈ used evs ⟶ X ∈ used evs & Y ∈ used evs"  
 <proof>

### 23.1 Function *knows*

**lemmas** *parts\_insert\_knows\_A* = *parts\_insert* [of \_ "knows A evs", standard]

**lemma** *knows\_Spy\_Says* [simp]:  
 "knows Spy (Says A B X # evs) = insert X (knows Spy evs)"  
 <proof>

Letting the Spy see "bad" agents' notes avoids redundant case-splits on whether  $A = \text{Spy}$  and whether  $A \in \text{bad}$

```

lemma knows_Spy_Notes [simp]:
  "knows Spy (Notes A X # evs) =
    (if A ∈ bad then insert X (knows Spy evs) else knows Spy evs)"
<proof>

lemma knows_Spy_Gets [simp]: "knows Spy (Gets A X # evs) = knows Spy evs"
<proof>

lemma knows_Spy_Inputs_secureM [simp]:
  "secureM  $\implies$  knows Spy (Inputs A C X # evs) =
    (if A=Spy then insert X (knows Spy evs) else knows Spy evs)"
<proof>

lemma knows_Spy_Inputs_insecureM [simp]:
  "insecureM  $\implies$  knows Spy (Inputs A C X # evs) = insert X (knows Spy evs)"
<proof>

lemma knows_Spy_C_Gets [simp]: "knows Spy (C_Gets C X # evs) = knows Spy
  evs"
<proof>

lemma knows_Spy_Outpts_secureM [simp]:
  "secureM  $\implies$  knows Spy (Outpts C A X # evs) =
    (if A=Spy then insert X (knows Spy evs) else knows Spy evs)"
<proof>

lemma knows_Spy_Outpts_insecureM [simp]:
  "insecureM  $\implies$  knows Spy (Outpts C A X # evs) = insert X (knows Spy
  evs)"
<proof>

lemma knows_Spy_A_Gets [simp]: "knows Spy (A_Gets A X # evs) = knows Spy
  evs"
<proof>

lemma knows_Spy_subset_knows_Spy_Says:
  "knows Spy evs  $\subseteq$  knows Spy (Says A B X # evs)"
<proof>

lemma knows_Spy_subset_knows_Spy_Notes:
  "knows Spy evs  $\subseteq$  knows Spy (Notes A X # evs)"
<proof>

lemma knows_Spy_subset_knows_Spy_Gets:
  "knows Spy evs  $\subseteq$  knows Spy (Gets A X # evs)"
<proof>

lemma knows_Spy_subset_knows_Spy_Inputs:
  "knows Spy evs  $\subseteq$  knows Spy (Inputs A C X # evs)"
<proof>

```

```

lemma knows_Spy_equals_knows_Spy_Gets:
  "knows Spy evs = knows Spy (C_Gets C X # evs)"
  <proof>

```

```

lemma knows_Spy_subset_knows_Spy_Outpts: "knows Spy evs  $\subseteq$  knows Spy (Outpts
  C A X # evs)"
  <proof>

```

```

lemma knows_Spy_subset_knows_Spy_A_Gets: "knows Spy evs  $\subseteq$  knows Spy (A_Gets
  A X # evs)"
  <proof>

```

Spy sees what is sent on the traffic

```

lemma Says_imp_knows_Spy [rule_format]:
  "Says A B X  $\in$  set evs  $\longrightarrow$  X  $\in$  knows Spy evs"
  <proof>

```

```

lemma Notes_imp_knows_Spy [rule_format]:
  "Notes A X  $\in$  set evs  $\longrightarrow$  A  $\in$  bad  $\longrightarrow$  X  $\in$  knows Spy evs"
  <proof>

```

```

lemma Inputs_imp_knows_Spy_secureM [rule_format (no_asm)]:
  "Inputs Spy C X  $\in$  set evs  $\longrightarrow$  secureM  $\longrightarrow$  X  $\in$  knows Spy evs"
  <proof>

```

```

lemma Inputs_imp_knows_Spy_insecureM [rule_format (no_asm)]:
  "Inputs A C X  $\in$  set evs  $\longrightarrow$  insecureM  $\longrightarrow$  X  $\in$  knows Spy evs"
  <proof>

```

```

lemma Outpts_imp_knows_Spy_secureM [rule_format (no_asm)]:
  "Outpts C Spy X  $\in$  set evs  $\longrightarrow$  secureM  $\longrightarrow$  X  $\in$  knows Spy evs"
  <proof>

```

```

lemma Outpts_imp_knows_Spy_insecureM [rule_format (no_asm)]:
  "Outpts C A X  $\in$  set evs  $\longrightarrow$  insecureM  $\longrightarrow$  X  $\in$  knows Spy evs"
  <proof>

```

Elimination rules: derive contradictions from old Says events containing items known to be fresh

```

lemmas knows_Spy_partsEs =
  Says_imp_knows_Spy [THEN parts.Inj, THEN revcut_rl, standard]
  parts.Body [THEN revcut_rl, standard]

```

## 23.2 Knowledge of Agents

```

lemma knows_Says: "knows A (Says A B X # evs) = insert X (knows A evs)"
  <proof>

```

```

lemma knows_Notes: "knows A (Notes A X # evs) = insert X (knows A evs)"
  <proof>

```



**lemma** *knows\_Gets*:

" $A \neq \text{Spy} \longrightarrow \text{knows } A (\text{Gets } A \ X \ \# \ \text{evs}) = \text{insert } X (\text{knows } A \ \text{evs})$ "  
 <proof>

**lemma** *knows\_Inputs*: " $\text{knows } A (\text{Inputs } A \ C \ X \ \# \ \text{evs}) = \text{insert } X (\text{knows } A \ \text{evs})$ "  
 <proof>

**lemma** *knows\_C\_Gets*: " $\text{knows } A (\text{C_Gets } C \ X \ \# \ \text{evs}) = \text{knows } A \ \text{evs}$ "  
 <proof>

**lemma** *knows\_Outpts\_secureM*:  
 " $\text{secureM} \longrightarrow \text{knows } A (\text{Outpts } C \ A \ X \ \# \ \text{evs}) = \text{insert } X (\text{knows } A \ \text{evs})$ "  
 <proof>

**lemma** *knows\_Outpts\_insecureM*:  
 " $\text{insecureM} \longrightarrow \text{knows } \text{Spy} (\text{Outpts } C \ A \ X \ \# \ \text{evs}) = \text{insert } X (\text{knows } \text{Spy} \ \text{evs})$ "  
 <proof>

**lemma** *knows\_subset\_knows\_Says*: " $\text{knows } A \ \text{evs} \subseteq \text{knows } A (\text{Says } A' \ B \ X \ \# \ \text{evs})$ "  
 <proof>

**lemma** *knows\_subset\_knows\_Notes*: " $\text{knows } A \ \text{evs} \subseteq \text{knows } A (\text{Notes } A' \ X \ \# \ \text{evs})$ "  
 <proof>

**lemma** *knows\_subset\_knows\_Gets*: " $\text{knows } A \ \text{evs} \subseteq \text{knows } A (\text{Gets } A' \ X \ \# \ \text{evs})$ "  
 <proof>

**lemma** *knows\_subset\_knows\_Inputs*: " $\text{knows } A \ \text{evs} \subseteq \text{knows } A (\text{Inputs } A' \ C \ X \ \# \ \text{evs})$ "  
 <proof>

**lemma** *knows\_subset\_knows\_C\_Gets*: " $\text{knows } A \ \text{evs} \subseteq \text{knows } A (\text{C_Gets } C \ X \ \# \ \text{evs})$ "  
 <proof>

**lemma** *knows\_subset\_knows\_Outpts*: " $\text{knows } A \ \text{evs} \subseteq \text{knows } A (\text{Outpts } C \ A' \ X \ \# \ \text{evs})$ "  
 <proof>

**lemma** *knows\_subset\_knows\_A\_Gets*: " $\text{knows } A \ \text{evs} \subseteq \text{knows } A (\text{A_Gets } A' \ X \ \# \ \text{evs})$ "  
 <proof>

Agents know what they say

**lemma** *Says\_imp\_knows* [rule\_format]: " $\text{Says } A \ B \ X \in \text{set } \text{evs} \longrightarrow X \in \text{knows } A \ \text{evs}$ "  
 <proof>

Agents know what they note

**lemma** *Notes\_imp\_knows* [rule\_format]: " $\text{Notes } A \ X \in \text{set } \text{evs} \longrightarrow X \in \text{knows } A \ \text{evs}$ "

*A evs"*

*<proof>*

Agents know what they receive

**lemma** *Gets\_imp\_knows\_agents [rule\_format]:*

*"A  $\neq$  Spy  $\longrightarrow$  Gets A X  $\in$  set evs  $\longrightarrow$  X  $\in$  knows A evs"*

*<proof>*

**lemma** *Inputs\_imp\_knows\_agents [rule\_format (no\_asm)]:*

*"Inputs A (Card A) X  $\in$  set evs  $\longrightarrow$  X  $\in$  knows A evs"*

*<proof>*

**lemma** *Outpts\_imp\_knows\_agents\_secureM [rule\_format (no\_asm)]:*

*"secureM  $\longrightarrow$  Outpts (Card A) A X  $\in$  set evs  $\longrightarrow$  X  $\in$  knows A evs"*

*<proof>*

**lemma** *Outpts\_imp\_knows\_agents\_insecureM [rule\_format (no\_asm)]:*

*"insecureM  $\longrightarrow$  Outpts (Card A) A X  $\in$  set evs  $\longrightarrow$  X  $\in$  knows Spy evs"*

*<proof>*

**lemma** *parts\_knows\_Spy\_subset\_used: "parts (knows Spy evs)  $\subseteq$  used evs"*

*<proof>*

**lemmas** *usedI = parts\_knows\_Spy\_subset\_used [THEN subsetD, intro]*

**lemma** *initState\_into\_used: "X  $\in$  parts (initState B)  $\implies$  X  $\in$  used evs"*

*<proof>*

**lemma** *used\_Says [simp]: "used (Says A B X # evs) = parts{X}  $\cup$  used evs"*

*<proof>*

**lemma** *used\_Notes [simp]: "used (Notes A X # evs) = parts{X}  $\cup$  used evs"*

*<proof>*

**lemma** *used\_Gets [simp]: "used (Gets A X # evs) = used evs"*

*<proof>*

**lemma** *used\_Inputs [simp]: "used (Inputs A C X # evs) = parts{X}  $\cup$  used evs"*

*<proof>*

**lemma** *used\_C\_Gets [simp]: "used (C\_Gets C X # evs) = used evs"*

*<proof>*

**lemma** *used\_Outpts [simp]: "used (Outpts C A X # evs) = parts{X}  $\cup$  used evs"*

*<proof>*

```
lemma used_A_Gets [simp]: "used (A_Gets A X # evs) = used evs"
<proof>
```

```
lemma used_nil_subset: "used []  $\subseteq$  used evs"
<proof>
```

```
lemma Says_parts_used [rule_format (no_asm)]:
  "Says A B X  $\in$  set evs  $\longrightarrow$  (parts {X})  $\subseteq$  used evs"
<proof>
```

```
lemma Notes_parts_used [rule_format (no_asm)]:
  "Notes A X  $\in$  set evs  $\longrightarrow$  (parts {X})  $\subseteq$  used evs"
<proof>
```

```
lemma Outpts_parts_used [rule_format (no_asm)]:
  "Outpts C A X  $\in$  set evs  $\longrightarrow$  (parts {X})  $\subseteq$  used evs"
<proof>
```

```
lemma Inputs_parts_used [rule_format (no_asm)]:
  "Inputs A C X  $\in$  set evs  $\longrightarrow$  (parts {X})  $\subseteq$  used evs"
<proof>
```

NOTE REMOVAL—laws above are cleaner, as they don't involve "case"

```
declare knows_Cons [simp del]
      used_Nil [simp del] used_Cons [simp del]
```

```
lemma knows_subset_knows_Cons: "knows A evs  $\subseteq$  knows A (e # evs)"
<proof>
```

```
lemma initState_subset_knows: "initState A  $\subseteq$  knows A evs"
<proof>
```

For proving *new\_keys\_not\_used*

```
lemma keysFor_parts_insert:
  "[[ K  $\in$  keysFor (parts (insert X G)); X  $\in$  synth (analz H) ]
 $\implies$  K  $\in$  keysFor (parts (G  $\cup$  H))  $\vee$  Key (invKey K)  $\in$  parts H]"
<proof>
```

end

## 24 Theory of smartcards

```
theory Smartcard imports EventSC begin
```

As smartcards handle long-term (symmetric) keys, this theory extends and supersedes theory *Private.thy*

An agent is bad if she reveals her PIN to the spy, not the shared key that is embedded in her card. An agent's being bad implies nothing about her smartcard, which independently may be stolen or cloned.

#### consts

```
shrK    :: "agent => key"
crdK    :: "card  => key"
pin     :: "agent => key"
```

```
Pairkey :: "agent * agent => nat"
pairK   :: "agent * agent => key"
```

#### axioms

```
inj_shrK: "inj shrK"  — No two smartcards store the same key
inj_crdK: "inj crdK"  — Nor do two cards
inj_pin  : "inj pin"   — Nor do two agents have the same pin
```

```
inj_pairK [iff]: "(pairK(A,B) = pairK(A',B')) = (A = A' & B = B')"
```

```
comm_Pairkey [iff]: "Pairkey(A,B) = Pairkey(B,A)"
```

```
pairK_disj_crdK [iff]: "pairK(A,B) ≠ crdK C"
pairK_disj_shrK [iff]: "pairK(A,B) ≠ shrK P"
pairK_disj_pin [iff]:  "pairK(A,B) ≠ pin P"
shrK_disj_crdK [iff]:  "shrK P ≠ crdK C"
shrK_disj_pin [iff]:   "shrK P ≠ pin Q"
crdK_disj_pin [iff]:   "crdK C ≠ pin P"
```

All keys are symmetric

```
defs all_symmetric_def: "all_symmetric == True"
```

```
lemma isSym_keys: "K ∈ symKeys"
```

*<proof>*

#### constdefs

```
legalUse :: "card => bool" ("legalUse (_)" )
"legalUse C == C ∉ stolen"
```

#### consts

```
illegalUse :: "card => bool"
```

#### primrec

```
illegalUse_def:
```

```
"illegalUse (Card A) = ( (Card A ∈ stolen ∧ A ∈ bad) ∨ Card A ∈ cloned )"
)"
```

initState must be defined with care

#### primrec

```
initState_Server: "initState Server =
  (Key'(range shrK ∪ range crdK ∪ range pin ∪ range pairK)) ∪
  (Nonce'(range Pairkey))"
```

```

initState_Friend: "initState (Friend i) = {Key (pin (Friend i))}"

initState_Spy: "initState Spy =
  (Key'((pin'bad) ∪ (pin '{A. Card A ∈ cloned}) ∪
    (shrK'{A. Card A ∈ cloned}) ∪
    (crdK'cloned) ∪
    (pairK'{(X,A). Card A ∈ cloned})))
  ∪ (Nonce'(Pairkey'{(A,B). Card A ∈ cloned & Card B ∈ cloned}))"

```

Still relying on axioms

**axioms**

```

Key_supply_ax: "finite KK  $\implies \exists K. K \notin KK \ \& \ Key \ K \notin used \ evs"$ 

```

```

Nonce_supply_ax: "finite NN  $\implies \exists N. N \notin NN \ \& \ Nonce \ N \notin used \ evs"$ 

```

## 24.1 Basic properties of shrK

```

declare inj_shrK [THEN inj_eq, iff]
declare inj_crdK [THEN inj_eq, iff]
declare inj_pin [THEN inj_eq, iff]

```

```

lemma invKey_K [simp]: "invKey K = K"
<proof>

```

```

lemma analz_Decrypt' [dest]:
  "[ Crypt K X ∈ analz H; Key K ∈ analz H ]  $\implies X \in analz H$ "
<proof>

```

Now cancel the *dest* attribute given to *analz.Decrypt* in its declaration.

```

declare analz.Decrypt [rule del]

```

Rewrites should not refer to *initState (Friend i)* because that expression is not in normal form.

Added to extend *initstate* with set of nonces

```

lemma parts_image_Nonce [simp]: "parts (Nonce'N) = Nonce'N"
<proof>

```

```

lemma keysFor_parts_initState [simp]: "keysFor (parts (initState C)) = {}"
<proof>

```

```

lemma keysFor_parts_insert:
  "[ K ∈ keysFor (parts (insert X G)); X ∈ synth (analz H) ]
 $\implies K \in keysFor (parts (G \cup H)) \mid Key \ K \in parts \ H$ "
<proof>

```

```

lemma Crypt_imp_keysFor: "Crypt K X ∈ H  $\implies K \in keysFor \ H$ "
<proof>

```

## 24.2 Function "knows"

**lemma** *Spy\_knows\_bad* [intro!]: "A ∈ bad ⇒ Key (pin A) ∈ knows Spy evs"  
 <proof>

**lemma** *Spy\_knows\_cloned* [intro!]:  
 "Card A ∈ cloned ⇒ Key (crdK (Card A)) ∈ knows Spy evs &  
 Key (shrK A) ∈ knows Spy evs &  
 Key (pin A) ∈ knows Spy evs &  
 (∀ B. Key (pairK(B,A)) ∈ knows Spy evs)"  
 <proof>

**lemma** *Spy\_knows\_cloned1* [intro!]: "C ∈ cloned ⇒ Key (crdK C) ∈ knows Spy evs"  
 <proof>

**lemma** *Spy\_knows\_cloned2* [intro!]: "[| Card A ∈ cloned; Card B ∈ cloned |]  
 ⇒ Nonce (Pairkey(A,B)) ∈ knows Spy evs"  
 <proof>

**lemma** *Spy\_knows\_Spy\_bad* [intro!]: "A ∈ bad ⇒ Key (pin A) ∈ knows Spy evs"  
 <proof>

**lemma** *Crypt\_Spy\_analz\_bad*:  
 "[| Crypt (pin A) X ∈ analz (knows Spy evs); A ∈ bad |]  
 ⇒ X ∈ analz (knows Spy evs)"  
 <proof>

**lemma** *shrK\_in\_initState* [iff]: "Key (shrK A) ∈ initState Server"  
 <proof>

**lemma** *shrK\_in\_used* [iff]: "Key (shrK A) ∈ used evs"  
 <proof>

**lemma** *crdK\_in\_initState* [iff]: "Key (crdK A) ∈ initState Server"  
 <proof>

**lemma** *crdK\_in\_used* [iff]: "Key (crdK A) ∈ used evs"  
 <proof>

**lemma** *pin\_in\_initState* [iff]: "Key (pin A) ∈ initState A"  
 <proof>

**lemma** *pin\_in\_used* [iff]: "Key (pin A) ∈ used evs"  
 <proof>

**lemma** *pairK\_in\_initState* [iff]: "Key (pairK X) ∈ initState Server"

*<proof>*

**lemma** *pairK\_in\_used* [iff]: "Key (pairK X) ∈ used evs"  
*<proof>*

**lemma** *Key\_not\_used* [simp]: "Key K ∉ used evs ⇒ K ∉ range shrK"  
*<proof>*

**lemma** *shrK\_neq* [simp]: "Key K ∉ used evs ⇒ shrK B ≠ K"  
*<proof>*

**lemma** *crdK\_not\_used* [simp]: "Key K ∉ used evs ⇒ K ∉ range crdK"  
*<proof>*

**lemma** *crdK\_neq* [simp]: "Key K ∉ used evs ⇒ crdK C ≠ K"  
*<proof>*

**lemma** *pin\_not\_used* [simp]: "Key K ∉ used evs ⇒ K ∉ range pin"  
*<proof>*

**lemma** *pin\_neq* [simp]: "Key K ∉ used evs ⇒ pin A ≠ K"  
*<proof>*

**lemma** *pairK\_not\_used* [simp]: "Key K ∉ used evs ⇒ K ∉ range pairK"  
*<proof>*

**lemma** *pairK\_neq* [simp]: "Key K ∉ used evs ⇒ pairK(A,B) ≠ K"  
*<proof>*

**declare** *shrK\_neq* [THEN not\_sym, simp]  
**declare** *crdK\_neq* [THEN not\_sym, simp]  
**declare** *pin\_neq* [THEN not\_sym, simp]  
**declare** *pairK\_neq* [THEN not\_sym, simp]

## 24.3 Fresh nonces

**lemma** *Nonce\_notin\_initState* [iff]: "Nonce N ∉ parts (initState (Friend i))"  
*<proof>*

## 24.4 Supply fresh nonces for possibility theorems.

**lemma** *Nonce\_supply1*: "∃N. Nonce N ∉ used evs"  
*<proof>*

**lemma** *Nonce\_supply2*:  
 "∃N N'. Nonce N ∉ used evs & Nonce N' ∉ used evs' & N ≠ N'"  
*<proof>*

**lemma** *Nonce\_supply3*: "∃N N' N''. Nonce N ∉ used evs & Nonce N' ∉ used evs' &  
 &

$\text{Nonce } N'' \notin \text{used evs}'' \ \& \ N \neq N' \ \& \ N' \neq N'' \ \& \ N \neq N''''$

$\langle \text{proof} \rangle$

**lemma** *Nonce\_supply*: " $\text{Nonce } (@ \ N. \ \text{Nonce } N \notin \text{used evs}) \notin \text{used evs}$ "

$\langle \text{proof} \rangle$

Unlike the corresponding property of nonces, we cannot prove  $\text{finite } KK \implies \exists K. K \notin KK \wedge \text{Key } K \notin \text{used evs}$ . We have infinitely many agents and there is nothing to stop their long-term keys from exhausting all the natural numbers. Instead, possibility theorems must assume the existence of a few keys.

## 24.5 Specialized Rewriting for Theorems About *analz* and Image

**lemma** *subset\_Compl\_range\_shrK*: " $A \subseteq - (\text{range shrK}) \implies \text{shrK } x \notin A$ "

$\langle \text{proof} \rangle$

**lemma** *subset\_Compl\_range\_crdK*: " $A \subseteq - (\text{range crdK}) \implies \text{crdK } x \notin A$ "

$\langle \text{proof} \rangle$

**lemma** *subset\_Compl\_range\_pin*: " $A \subseteq - (\text{range pin}) \implies \text{pin } x \notin A$ "

$\langle \text{proof} \rangle$

**lemma** *subset\_Compl\_range\_pairK*: " $A \subseteq - (\text{range pairK}) \implies \text{pairK } x \notin A$ "

$\langle \text{proof} \rangle$

**lemma** *insert\_Key\_singleton*: " $\text{insert } (\text{Key } K) \ H = \text{Key } ' \{K\} \cup H$ "

$\langle \text{proof} \rangle$

**lemma** *insert\_Key\_image*: " $\text{insert } (\text{Key } K) \ (\text{Key}'KK \cup C) = \text{Key}'(\text{insert } K \ KK) \cup C$ "

$\langle \text{proof} \rangle$

**lemmas** *analz\_image\_freshK\_simps* =

*simp\_thms mem\_simps* — these two allow its use with only:

*disj\_comms*

*image\_insert [THEN sym] image\_Un [THEN sym] empty\_subsetI insert\_subset*

*analz\_insert\_eq Un\_upper2 [THEN analz\_mono, THEN [2] rev\_subsetD]*

*insert\_Key\_singleton subset\_Compl\_range\_shrK subset\_Compl\_range\_crdK*

*subset\_Compl\_range\_pin subset\_Compl\_range\_pairK*

*Key\_not\_used insert\_Key\_image Un\_assoc [THEN sym]*

**lemma** *analz\_image\_freshK\_lemma*:

$(\text{Key } K \in \text{analz } (\text{Key}'nE \cup H)) \implies (K \in nE \mid \text{Key } K \in \text{analz } H) \implies$

$(\text{Key } K \in \text{analz } (\text{Key}'nE \cup H)) = (K \in nE \mid \text{Key } K \in \text{analz } H)$

$\langle \text{proof} \rangle$

## 24.6 Tactics for possibility theorems

$\langle ML \rangle$



```

lemma invKey_shrK_iff [iff]:
  "(Key (invKey K) ∈ X) = (Key K ∈ X)"
  <proof>

  <ML>

lemma knows_subset_knows_Cons: "knows A evs ⊆ knows A (e # evs)"
  <proof>

  declare shrK_disj_crdK[THEN not_sym, iff]
  declare shrK_disj_pin[THEN not_sym, iff]
  declare pairK_disj_shrK[THEN not_sym, iff]
  declare pairK_disj_crdK[THEN not_sym, iff]
  declare pairK_disj_pin[THEN not_sym, iff]
  declare crdK_disj_pin[THEN not_sym, iff]

  declare legalUse_def [iff] illegalUse_def [iff]

  end

```

## 25 Original Shoup-Rubin protocol

```

theory ShoupRubin imports Smartcard begin

consts

  sesK :: "nat*key => key"

axioms

  inj_sesK [iff]: "(sesK(m,k) = sesK(m',k')) = (m = m' ∧ k = k')"

  shrK_disj_sesK [iff]: "shrK A ≠ sesK(m,pk)"
  crdK_disj_sesK [iff]: "crdK C ≠ sesK(m,pk)"
  pin_disj_sesK [iff]: "pin P ≠ sesK(m,pk)"
  pairK_disj_sesK [iff]: "pairK(A,B) ≠ sesK(m,pk)"

  Atomic_distrib [iff]: "Atomic'(KEY'K ∪ NONCE'N) =
    Atomic'(KEY'K) ∪ Atomic'(NONCE'N)"

  shouprubin_assumes_securemeans [iff]: "evs ∈ sr ⟹ secureM"

constdefs

```

```

Unique :: "[event, event list] => bool" ("Unique _ on _")
"Unique ev on evs ==
  ev ∉ set (tl (dropWhile (% z. z ≠ ev) evs))"

```

```

inductive_set sr :: "event list set"
where

```

```

  Nil: "[] ∈ sr"

```

```

/ Fake: "[[ evsF ∈ sr; X ∈ synth (analz (knows Spy evsF));
  illegalUse(Card B) ] ]
  ⇒ Says Spy A X #
    Inputs Spy (Card B) X # evsF ∈ sr"

```

```

/ Forge:
  "[[ evsFo ∈ sr; Nonce Nb ∈ analz (knows Spy evsFo);
    Key (pairK(A,B)) ∈ knows Spy evsFo ] ]
  ⇒ Notes Spy (Key (sesK(Nb,pairK(A,B)))) # evsFo ∈ sr"

```

```

/ Reception: "[[ evsR ∈ sr; Says A B X ∈ set evsR ] ]
  ⇒ Gets B X # evsR ∈ sr"

```

```

/ SR1: "[[ evs1 ∈ sr; A ≠ Server ] ]
  ⇒ Says A Server {Agent A, Agent B}
    # evs1 ∈ sr"

```

```

/ SR2: "[[ evs2 ∈ sr;
  Gets Server {Agent A, Agent B} ∈ set evs2 ] ]
  ⇒ Says Server A {Nonce (Pairkey(A,B)),
    Crypt (shrK A) {Nonce (Pairkey(A,B)), Agent B}
    }
  # evs2 ∈ sr"

```

```

/ SR3: "[[ evs3 ∈ sr; legalUse(Card A);
  Says A Server {Agent A, Agent B} ∈ set evs3;
  Gets A {Nonce Pk, Certificate} ∈ set evs3 ] ]
  ⇒ Inputs A (Card A) (Agent A)
    # evs3 ∈ sr"

```

```

/ SR4: "[ evs4 ∈ sr; A ≠ Server;
        Nonce Na ∉ used evs4; legalUse(Card A);
        Inputs A (Card A) (Agent A) ∈ set evs4 ]
⇒ Outputs (Card A) A {Nonce Na, Crypt (crdK (Card A)) (Nonce Na)}
   # evs4 ∈ sr"

/ SR4Fake: "[ evs4F ∈ sr; Nonce Na ∉ used evs4F;
              illegalUse(Card A);
              Inputs Spy (Card A) (Agent A) ∈ set evs4F ]
⇒ Outputs (Card A) Spy {Nonce Na, Crypt (crdK (Card A)) (Nonce Na)}
   # evs4F ∈ sr"

/ SR5: "[ evs5 ∈ sr;
          Outputs (Card A) A {Nonce Na, Certificate} ∈ set evs5;
          ∀ p q. Certificate ≠ {p, q} ]
⇒ Says A B {Agent A, Nonce Na} # evs5 ∈ sr"

/ SR6: "[ evs6 ∈ sr; legalUse(Card B);
          Gets B {Agent A, Nonce Na} ∈ set evs6 ]
⇒ Inputs B (Card B) {Agent A, Nonce Na}
   # evs6 ∈ sr"

/ SR7: "[ evs7 ∈ sr;
          Nonce Nb ∉ used evs7; legalUse(Card B); B ≠ Server;
          K = sesK(Nb, pairK(A, B));
          Key K ∉ used evs7;
          Inputs B (Card B) {Agent A, Nonce Na} ∈ set evs7 ]
⇒ Outputs (Card B) B {Nonce Nb, Key K,
                      Crypt (pairK(A, B)) {Nonce Na, Nonce Nb},
                      Crypt (pairK(A, B)) (Nonce Nb)}
   # evs7 ∈ sr"

/ SR7Fake: "[ evs7F ∈ sr; Nonce Nb ∉ used evs7F;
              illegalUse(Card B);
              K = sesK(Nb, pairK(A, B));
              Key K ∉ used evs7F;
              Inputs Spy (Card B) {Agent A, Nonce Na} ∈ set evs7F ]
⇒ Outputs (Card B) Spy {Nonce Nb, Key K,
                        Crypt (pairK(A, B)) {Nonce Na, Nonce Nb},

```



```

Nb}},
                                Crypt (crdK (Card A)) (Nonce Na)}
                                ∈ set evs10F ]]
⇒ Outpts (Card A) Spy {Key K, Crypt (pairK(A,B)) (Nonce Nb)}
    # evs10F ∈ sr"

```

```

/ SR11: "[ evs11 ∈ sr;
    Says A Server {Agent A, Agent B} ∈ set evs11;
    Outpts (Card A) A {Key K, Certificate} ∈ set evs11 ]]
⇒ Says A B (Certificate)
    # evs11 ∈ sr"

```

```

/ Oops1:
  "[ evs01 ∈ sr;
    Outpts (Card B) B {Nonce Nb, Key K, Certificate,
                      Crypt (pairK(A,B)) (Nonce Nb)} ∈ set evs01 ]]
⇒ Notes Spy {Key K, Nonce Nb, Agent A, Agent B} # evs01 ∈ sr"

```

```

/ Oops2:
  "[ evs02 ∈ sr;
    Outpts (Card A) A {Key K, Crypt (pairK(A,B)) (Nonce Nb)}
    ∈ set evs02 ]]
⇒ Notes Spy {Key K, Nonce Nb, Agent A, Agent B} # evs02 ∈ sr"

```

```

declare Fake_parts_insert_in_Un [dest]
declare analz_into_parts [dest]

```

```

lemma Gets_imp_Says:
  "[ Gets B X ∈ set evs; evs ∈ sr ] ⇒ ∃ A. Says A B X ∈ set evs"
⟨proof⟩

```

```

lemma Gets_imp_knows_Spy:
  "[ Gets B X ∈ set evs; evs ∈ sr ] ⇒ X ∈ knows Spy evs"
⟨proof⟩

```

```

lemma Gets_imp_knows_Spy_parts_Snd:

```

"[[ Gets B {X, Y} ∈ set evs; evs ∈ sr ]] ⇒ Y ∈ parts (knows Spy evs)"  
 <proof>

**lemma** Gets\_imp\_knows\_Spy\_analz\_Snd:

"[[ Gets B {X, Y} ∈ set evs; evs ∈ sr ]] ⇒ Y ∈ analz (knows Spy evs)"  
 <proof>

**lemma** Inputs\_imp\_knows\_Spy\_secureM\_sr:

"[[ Inputs Spy C X ∈ set evs; evs ∈ sr ]] ⇒ X ∈ knows Spy evs"  
 <proof>

**lemma** knows\_Spy\_Inputs\_secureM\_sr\_Spy:

"evs ∈ sr ⇒ knows Spy (Inputs Spy C X # evs) = insert X (knows Spy evs)"  
 <proof>

**lemma** knows\_Spy\_Inputs\_secureM\_sr:

"[[ A ≠ Spy; evs ∈ sr ]] ⇒ knows Spy (Inputs A C X # evs) = knows Spy evs"  
 <proof>

**lemma** knows\_Spy\_Outpts\_secureM\_sr\_Spy:

"evs ∈ sr ⇒ knows Spy (Outpts C Spy X # evs) = insert X (knows Spy evs)"  
 <proof>

**lemma** knows\_Spy\_Outpts\_secureM\_sr:

"[[ A ≠ Spy; evs ∈ sr ]] ⇒ knows Spy (Outpts C A X # evs) = knows Spy evs"  
 <proof>

**lemma** Inputs\_A\_Card\_3:

"[[ Inputs A C (Agent A) ∈ set evs; A ≠ Spy; evs ∈ sr ]]  
 ⇒ legalUse(C) ∧ C = (Card A) ∧  
 (∃ Pk Certificate. Gets A {Pk, Certificate} ∈ set evs)"  
 <proof>

**lemma** Inputs\_B\_Card\_6:

"[[ Inputs B C {Agent A, Nonce Na} ∈ set evs; B ≠ Spy; evs ∈ sr ]]

$\implies \text{legalUse}(C) \wedge C = (\text{Card } B) \wedge \text{Gets } B \{ \text{Agent } A, \text{Nonce } Na \} \in \text{set evs}$ "  
 $\langle \text{proof} \rangle$

**lemma** *Inputs\_A\_Card\_9:*

" $\llbracket \text{Inputs } A \ C \{ \text{Agent } B, \text{Nonce } Na, \text{Nonce } Nb, \text{Nonce } Pk, \text{Cert1}, \text{Cert2}, \text{Cert3} \} \in \text{set evs};$

$A \neq \text{Spy}; \text{evs} \in \text{sr} \rrbracket$   
 $\implies \text{legalUse}(C) \wedge C = (\text{Card } A) \wedge$   
 $\text{Gets } A \{ \text{Nonce } Pk, \text{Cert1} \} \in \text{set evs} \quad \wedge$   
 $\text{Outpts } (\text{Card } A) \ A \{ \text{Nonce } Na, \text{Cert3} \} \in \text{set evs} \quad \wedge$   
 $\text{Gets } A \{ \text{Nonce } Nb, \text{Cert2} \} \in \text{set evs}"$

$\langle \text{proof} \rangle$

**lemma** *Outpts\_A\_Card\_4:*

" $\llbracket \text{Outpts } C \ A \{ \text{Nonce } Na, (\text{Crypt } (\text{crdK } (\text{Card } A)) (\text{Nonce } Na)) \} \in \text{set evs};$

$\text{evs} \in \text{sr} \rrbracket$   
 $\implies \text{legalUse}(C) \wedge C = (\text{Card } A) \wedge$   
 $\text{Inputs } A \ (\text{Card } A) \ (\text{Agent } A) \in \text{set evs}"$

$\langle \text{proof} \rangle$

**lemma** *Outpts\_B\_Card\_7:*

" $\llbracket \text{Outpts } C \ B \{ \text{Nonce } Nb, \text{Key } K, \text{Crypt } (\text{pairK}(A,B)) \{ \text{Nonce } Na, \text{Nonce } Nb \}, \text{Cert2} \} \in \text{set evs};$

$\text{evs} \in \text{sr} \rrbracket$   
 $\implies \text{legalUse}(C) \wedge C = (\text{Card } B) \wedge$   
 $\text{Inputs } B \ (\text{Card } B) \{ \text{Agent } A, \text{Nonce } Na \} \in \text{set evs}"$

$\langle \text{proof} \rangle$

**lemma** *Outpts\_A\_Card\_10:*

" $\llbracket \text{Outpts } C \ A \{ \text{Key } K, (\text{Crypt } (\text{pairK}(A,B)) (\text{Nonce } Nb)) \} \in \text{set evs};$   
 $\text{evs} \in \text{sr} \rrbracket$

$\implies \text{legalUse}(C) \wedge C = (\text{Card } A) \wedge$   
 $(\exists \ Na \ Ver1 \ Ver2 \ Ver3.$   
 $\text{Inputs } A \ (\text{Card } A) \{ \text{Agent } B, \text{Nonce } Na, \text{Nonce } Nb, \text{Nonce } (\text{Pairkey}(A,B)),$

$\text{Ver1}, \text{Ver2}, \text{Ver3} \} \in \text{set evs})"$

$\langle \text{proof} \rangle$

**lemma** *Outpts\_A\_Card\_10\_imp\_Inputs:*

" $\llbracket \text{Outpts } (\text{Card } A) \ A \{ \text{Key } K, \text{Certificate} \} \in \text{set evs}; \text{evs} \in \text{sr} \rrbracket$

$\implies (\exists \ B \ Na \ Nb \ Ver1 \ Ver2 \ Ver3.$   
 $\text{Inputs } A \ (\text{Card } A) \{ \text{Agent } B, \text{Nonce } Na, \text{Nonce } Nb, \text{Nonce } (\text{Pairkey}(A,B)),$

$Ver1, Ver2, Ver3\} \in \text{set evs})"$

$\langle \text{proof} \rangle$

**lemma** *Outpts\_honest\_A\_Card\_4:*  
 "[[ Outpts C A {Nonce Na, Crypt K X} ∈ set evs;  
 A ≠ Spy; evs ∈ sr ]]  
 $\implies$  legalUse(C) ∧ C = (Card A) ∧  
 Inputs A (Card A) (Agent A) ∈ set evs"  
 $\langle \text{proof} \rangle$

**lemma** *Outpts\_honest\_B\_Card\_7:*  
 "[[ Outpts C B {Nonce Nb, Key K, Cert1, Cert2} ∈ set evs;  
 B ≠ Spy; evs ∈ sr ]]  
 $\implies$  legalUse(C) ∧ C = (Card B) ∧  
 (∃ A Na. Inputs B (Card B) {Agent A, Nonce Na} ∈ set evs)"  
 $\langle \text{proof} \rangle$

**lemma** *Outpts\_honest\_A\_Card\_10:*  
 "[[ Outpts C A {Key K, Certificate} ∈ set evs;  
 A ≠ Spy; evs ∈ sr ]]  
 $\implies$  legalUse (C) ∧ C = (Card A) ∧  
 (∃ B Na Nb Pk Ver1 Ver2 Ver3.  
 Inputs A (Card A) {Agent B, Nonce Na, Nonce Nb, Pk,  
 Ver1, Ver2, Ver3} ∈ set evs)"  
 $\langle \text{proof} \rangle$

**lemma** *Outpts\_which\_Card\_4:*  
 "[[ Outpts (Card A) A {Nonce Na, Crypt K X} ∈ set evs; evs ∈ sr ]]  
 $\implies$  Inputs A (Card A) (Agent A) ∈ set evs"  
 $\langle \text{proof} \rangle$

**lemma** *Outpts\_which\_Card\_7:*  
 "[[ Outpts (Card B) B {Nonce Nb, Key K, Cert1, Cert2} ∈ set evs;  
 evs ∈ sr ]]  
 $\implies$  ∃ A Na. Inputs B (Card B) {Agent A, Nonce Na} ∈ set evs"  
 $\langle \text{proof} \rangle$

**lemma** *Outpts\_which\_Card\_10:*  
 "[[ Outpts (Card A) A {Key (sesK(Nb, pairK(A,B))),  
 Crypt (pairK(A,B)) (Nonce Nb) } ∈ set evs;  
 evs ∈ sr ]]  
 $\implies$  ∃ Na. Inputs A (Card A) {Agent B, Nonce Na, Nonce Nb, Nonce (Pairkey(A,B)),  
 Crypt (shrK A) {Nonce (Pairkey(A,B)), Agent B},



Crypt (pairK(A,B)) {Nonce Na, Nonce Nb},  
Crypt (crdK (Card A)) (Nonce Na) } ∈ set evs"

⟨proof⟩

**lemma** Outpts\_A\_Card\_form\_4:  
 "[[ Outpts (Card A) A {Nonce Na, Certificate} ∈ set evs;  
   ∀ p q. Certificate ≠ {p, q}; evs ∈ sr ]]  
 ⇒ Certificate = (Crypt (crdK (Card A)) (Nonce Na))"  
 ⟨proof⟩

**lemma** Outpts\_B\_Card\_form\_7:  
 "[[ Outpts (Card B) B {Nonce Nb, Key K, Cert1, Cert2} ∈ set evs;  
   evs ∈ sr ]]  
 ⇒ ∃ A Na.  
   K = sesK(Nb, pairK(A,B)) ∧  
   Cert1 = (Crypt (pairK(A,B)) {Nonce Na, Nonce Nb}) ∧  
   Cert2 = (Crypt (pairK(A,B)) (Nonce Nb))"  
 ⟨proof⟩

**lemma** Outpts\_A\_Card\_form\_10:  
 "[[ Outpts (Card A) A {Key K, Certificate} ∈ set evs; evs ∈ sr ]]  
 ⇒ ∃ B Nb.  
   K = sesK(Nb, pairK(A,B)) ∧  
   Certificate = (Crypt (pairK(A,B)) (Nonce Nb))"  
 ⟨proof⟩

**lemma** Outpts\_A\_Card\_form\_bis:  
 "[[ Outpts (Card A') A' {Key (sesK(Nb, pairK(A,B))), Certificate} ∈ set evs;  
   evs ∈ sr ]]  
 ⇒ A' = A ∧  
   Certificate = (Crypt (pairK(A,B)) (Nonce Nb))"  
 ⟨proof⟩

**lemma** Inputs\_A\_Card\_form\_9:  
 "[[ Inputs A (Card A) {Agent B, Nonce Na, Nonce Nb, Nonce Pk,  
   Cert1, Cert2, Cert3} ∈ set evs;  
   evs ∈ sr ]]  
 ⇒ Cert3 = Crypt (crdK (Card A)) (Nonce Na)"  
 ⟨proof⟩

**lemma** *Inputs\_Card\_legalUse*:

"[[ Inputs A (Card A) X ∈ set evs; evs ∈ sr ]] ⇒ legalUse(Card A)"  
 ⟨proof⟩

**lemma** *Outpts\_Card\_legalUse*:

"[[ Outpts (Card A) A X ∈ set evs; evs ∈ sr ]] ⇒ legalUse(Card A)"  
 ⟨proof⟩

**lemma** *Inputs\_Card*: "[[ Inputs A C X ∈ set evs; A ≠ Spy; evs ∈ sr ]]  
 ⇒ C = (Card A) ∧ legalUse(C)"  
 ⟨proof⟩

**lemma** *Outpts\_Card*: "[[ Outpts C A X ∈ set evs; A ≠ Spy; evs ∈ sr ]]  
 ⇒ C = (Card A) ∧ legalUse(C)"  
 ⟨proof⟩

**lemma** *Inputs\_Outpts\_Card*:

"[[ Inputs A C X ∈ set evs ∨ Outpts C A Y ∈ set evs;  
 A ≠ Spy; evs ∈ sr ]]  
 ⇒ C = (Card A) ∧ legalUse(Card A)"  
 ⟨proof⟩

**lemma** *Inputs\_Card\_Spy*:

"[[ Inputs Spy C X ∈ set evs ∨ Outpts C Spy X ∈ set evs; evs ∈ sr ]]  
 ⇒ C = (Card Spy) ∧ legalUse(Card Spy) ∨  
 (∃ A. C = (Card A) ∧ illegalUse(Card A))"  
 ⟨proof⟩

**lemma** *Outpts\_A\_Card\_unique\_nonce*:

"[[ Outpts (Card A) A {Nonce Na, Crypt (crdK (Card A)) (Nonce Na)}  
 ∈ set evs;  
 Outpts (Card A') A' {Nonce Na, Crypt (crdK (Card A')) (Nonce Na)}  
 ∈ set evs;  
 evs ∈ sr ]] ⇒ A=A'"  
 ⟨proof⟩

**lemma** *Outpts\_B\_Card\_unique\_nonce:*

"[[ Outpts (Card B) B {Nonce Nb, Key SK, Cert1, Cert2} ∈ set evs;  
Outpts (Card B') B' {Nonce Nb, Key SK', Cert1', Cert2'} ∈ set evs;

evs ∈ sr ] ⇒ B=B' ∧ SK=SK' ∧ Cert1=Cert1' ∧ Cert2=Cert2'"]

⟨proof⟩

**lemma** *Outpts\_B\_Card\_unique\_key:*

"[[ Outpts (Card B) B {Nonce Nb, Key SK, Cert1, Cert2} ∈ set evs;  
Outpts (Card B') B' {Nonce Nb', Key SK, Cert1', Cert2'} ∈ set evs;

evs ∈ sr ] ⇒ B=B' ∧ Nb=Nb' ∧ Cert1=Cert1' ∧ Cert2=Cert2'"]

⟨proof⟩

**lemma** *Outpts\_A\_Card\_unique\_key:* "[ Outpts (Card A) A {Key K, V} ∈ set evs;

Outpts (Card A') A' {Key K, V'} ∈ set evs;

evs ∈ sr ] ⇒ A=A' ∧ V=V'"]

⟨proof⟩

**lemma** *Outpts\_A\_Card\_Unique:*

"[[ Outpts (Card A) A {Nonce Na, rest} ∈ set evs; evs ∈ sr ]  
⇒ Unique (Outpts (Card A) A {Nonce Na, rest}) on evs"]

⟨proof⟩

**lemma** *Spy\_knows\_Na:*

"[[ Says A B {Agent A, Nonce Na} ∈ set evs; evs ∈ sr ]

⇒ Nonce Na ∈ analz (knows Spy evs)"]

⟨proof⟩

**lemma** *Spy\_knows\_Nb:*

"[[ Says B A {Nonce Nb, Certificate} ∈ set evs; evs ∈ sr ]

⇒ Nonce Nb ∈ analz (knows Spy evs)"]

⟨proof⟩

**lemma** *Pairkey\_Gets\_analz\_knows\_Spy*:  
 "[[ Gets A {Nonce (Pairkey(A,B)), Certificate} ∈ set evs; evs ∈ sr ]]  
 ⇒ Nonce (Pairkey(A,B)) ∈ analz (knows Spy evs)"  
 <proof>

**lemma** *Pairkey\_Inputs\_imp\_Gets*:  
 "[[ Inputs A (Card A)  
   {Agent B, Nonce Na, Nonce Nb, Nonce (Pairkey(A,B)),  
   Cert1, Cert3, Cert2} ∈ set evs;  
   A ≠ Spy; evs ∈ sr ]]  
 ⇒ Gets A {Nonce (Pairkey(A,B)), Cert1} ∈ set evs"  
 <proof>

**lemma** *Pairkey\_Inputs\_analz\_knows\_Spy*:  
 "[[ Inputs A (Card A)  
   {Agent B, Nonce Na, Nonce Nb, Nonce (Pairkey(A,B)),  
   Cert1, Cert3, Cert2} ∈ set evs;  
   evs ∈ sr ]]  
 ⇒ Nonce (Pairkey(A,B)) ∈ analz (knows Spy evs)"  
 <proof>

**declare** *shrK\_disj\_sesK* [THEN not\_sym, iff]  
**declare** *pin\_disj\_sesK* [THEN not\_sym, iff]  
**declare** *crdK\_disj\_sesK* [THEN not\_sym, iff]  
**declare** *pairK\_disj\_sesK* [THEN not\_sym, iff]

<ML>

**lemma** *Spy\_parts\_keys [simp]*: "evs ∈ sr ⇒  
 (Key (shrK P) ∈ parts (knows Spy evs)) = (Card P ∈ cloned) ∧  
 (Key (pin P) ∈ parts (knows Spy evs)) = (P ∈ bad ∨ Card P ∈ cloned) ∧  
 (Key (crdK C) ∈ parts (knows Spy evs)) = (C ∈ cloned) ∧  
 (Key (pairK(A,B)) ∈ parts (knows Spy evs)) = (Card B ∈ cloned)"  
 <proof>

**lemma** *Spy\_analz\_shrK[simp]*: "evs ∈ sr ⇒

(Key (shrK P) ∈ analz (knows Spy evs)) = (Card P ∈ cloned)"  
 <proof>

**lemma** Spy\_analz\_crdK[simp]: "evs ∈ sr ⇒  
 (Key (crdK C) ∈ analz (knows Spy evs)) = (C ∈ cloned)"  
 <proof>

**lemma** Spy\_analz\_pairK[simp]: "evs ∈ sr ⇒  
 (Key (pairK(A,B)) ∈ analz (knows Spy evs)) = (Card B ∈ cloned)"  
 <proof>

**lemma** analz\_image\_Key\_Un\_Nonce: "analz (Key'K ∪ Nonce'N) = Key'K ∪ Nonce'N"  
 <proof>

<ML>

**lemma** analz\_image\_freshK [rule\_format]:  
 "evs ∈ sr ⇒ ∀ K KK.  
 (Key K ∈ analz (Key'KK ∪ (knows Spy evs))) =  
 (K ∈ KK ∨ Key K ∈ analz (knows Spy evs))"  
 <proof>

**lemma** analz\_insert\_freshK: "evs ∈ sr ⇒  
 Key K ∈ analz (insert (Key K') (knows Spy evs)) =  
 (K = K' ∨ Key K ∈ analz (knows Spy evs))"  
 <proof>

**lemma** Na\_Nb\_certificate\_authentic:  
 "[ Crypt (pairK(A,B)) {Nonce Na, Nonce Nb} ∈ parts (knows Spy evs);  
 ¬illegalUse(Card B);  
 evs ∈ sr ]  
 ⇒ Outputs (Card B) B {Nonce Nb, Key (sesK(Nb,pairK(A,B))),  
 Crypt (pairK(A,B)) {Nonce Na, Nonce Nb},  
 Crypt (pairK(A,B)) (Nonce Nb)} ∈ set evs"  
 <proof>

**lemma** Nb\_certificate\_authentic:  
 "[ Crypt (pairK(A,B)) (Nonce Nb) ∈ parts (knows Spy evs);  
 B ≠ Spy; ¬illegalUse(Card A); ¬illegalUse(Card B);

$$\begin{aligned} & \text{evs} \in \text{sr} \parallel \\ \implies & \text{Outpts (Card A) A } \{ \text{Key (sesK(Nb, pairK(A, B))),} \\ & \text{Crypt (pairK(A, B)) (Nonce Nb)} \} \in \text{set evs}'' \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *Outpts\_A\_Card\_imp\_pairK\_parts:*  

$$\begin{aligned} & " \parallel \text{Outpts (Card A) A} \\ & \quad \{ \text{Key K, Crypt (pairK(A, B)) (Nonce Nb)} \} \in \text{set evs}; \\ & \quad \text{evs} \in \text{sr} \parallel \\ \implies & \exists \text{ Na. Crypt (pairK(A, B)) } \{ \text{Nonce Na, Nonce Nb} \} \in \text{parts (knows Spy} \\ & \text{evs)}'' \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *Nb\_certificate\_authentic\_bis:*  

$$\begin{aligned} & " \parallel \text{Crypt (pairK(A, B)) (Nonce Nb)} \in \text{parts (knows Spy evs)}; \\ & \quad B \neq \text{Spy}; \neg \text{illegalUse(Card B)}; \\ & \quad \text{evs} \in \text{sr} \parallel \\ \implies & \exists \text{ Na. Outpts (Card B) B } \{ \text{Nonce Nb, Key (sesK(Nb, pairK(A, B))),} \\ & \quad \text{Crypt (pairK(A, B)) } \{ \text{Nonce Na, Nonce Nb} \}, \\ & \quad \text{Crypt (pairK(A, B)) (Nonce Nb)} \} \in \text{set evs}'' \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *Pairkey\_certificate\_authentic:*  

$$\begin{aligned} & " \parallel \text{Crypt (shrK A) } \{ \text{Nonce Pk, Agent B} \} \in \text{parts (knows Spy evs)}; \\ & \quad \text{Card A} \notin \text{cloned}; \text{evs} \in \text{sr} \parallel \\ \implies & \text{Pk} = \text{Pairkey(A, B)} \wedge \\ & \quad \text{Says Server A } \{ \text{Nonce Pk,} \\ & \quad \quad \text{Crypt (shrK A) } \{ \text{Nonce Pk, Agent B} \} \} \\ & \quad \in \text{set evs}'' \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *sesK\_authentic:*  

$$\begin{aligned} & " \parallel \text{Key (sesK(Nb, pairK(A, B)))} \in \text{parts (knows Spy evs)}; \\ & \quad A \neq \text{Spy}; B \neq \text{Spy}; \neg \text{illegalUse(Card A)}; \neg \text{illegalUse(Card B)}; \\ & \quad \text{evs} \in \text{sr} \parallel \\ \implies & \text{Notes Spy } \{ \text{Key (sesK(Nb, pairK(A, B))), Nonce Nb, Agent A, Agent B} \} \\ & \quad \in \text{set evs}'' \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma Confidentiality:**

```
"[[ Notes Spy {Key (sesK(Nb,pairK(A,B))), Nonce Nb, Agent A, Agent B}
    {
      not (set evs);
      A ≠ Spy; B ≠ Spy; not illegalUse(Card A); not illegalUse(Card B);
      evs ∈ sr
    }
  ] => Key (sesK(Nb,pairK(A,B))) not analz (knows Spy evs)"
⟨proof⟩
```

**lemma Confidentiality\_B:**

```
"[[ Outpts (Card B) B {Nonce Nb, Key K, Certificate,
    Crypt (pairK(A,B)) (Nonce Nb)} ∈ set evs;
    Notes Spy {Key K, Nonce Nb, Agent A, Agent B} not (set evs);
    A ≠ Spy; B ≠ Spy; not illegalUse(Card A); Card B not cloned;
    evs ∈ sr
  ] => Key K not analz (knows Spy evs)"
⟨proof⟩
```

**lemma A\_authenticates\_B:**

```
"[[ Outpts (Card A) A {Key K, Crypt (pairK(A,B)) (Nonce Nb)} ∈ set evs;
    not illegalUse(Card B);
    evs ∈ sr
  ] => ∃ Na.
    Outpts (Card B) B {Nonce Nb, Key K,
      Crypt (pairK(A,B)) {Nonce Na, Nonce Nb},
      Crypt (pairK(A,B)) (Nonce Nb)} ∈ set evs"
⟨proof⟩
```

**lemma A\_authenticates\_B\_Gets:**

```
"[[ Gets A {Nonce Nb, Crypt (pairK(A,B)) {Nonce Na, Nonce Nb}}
    ∈ set evs;
    not illegalUse(Card B);
    evs ∈ sr
  ] => Outpts (Card B) B {Nonce Nb, Key (sesK(Nb, pairK (A, B))),
    Crypt (pairK(A,B)) {Nonce Na, Nonce Nb},
    Crypt (pairK(A,B)) (Nonce Nb)} ∈ set evs"
⟨proof⟩
```

**lemma** *B\_authenticates\_A*:  
 "[[ Gets B (Crypt (pairK(A,B)) (Nonce Nb)) ∈ set evs;  
   B ≠ Spy; ¬illegalUse(Card A); ¬illegalUse(Card B);  
   evs ∈ sr ]]  
 ⇒ Outpts (Card A) A  
   {Key (sesK(Nb,pairK(A,B))), Crypt (pairK(A,B)) (Nonce Nb)} ∈ set evs"  
 <proof>

**lemma** *Confidentiality\_A*: "[[ Outpts (Card A) A  
   {Key K, Crypt (pairK(A,B)) (Nonce Nb)} ∈ set evs;  
   Notes Spy {Key K, Nonce Nb, Agent A, Agent B} ∉ set evs;  
   A ≠ Spy; B ≠ Spy; ¬illegalUse(Card A); ¬illegalUse(Card B);  
   evs ∈ sr ]]  
 ⇒ Key K ∉ analz (knows Spy evs)"  
 <proof>

**lemma** *Outpts\_imp\_knows\_agents\_secureM\_sr*:  
 "[[ Outpts (Card A) A X ∈ set evs; evs ∈ sr ]] ⇒ X ∈ knows A evs"  
 <proof>

**lemma** *A\_keydist\_to\_B*:  
 "[[ Outpts (Card A) A  
   {Key K, Crypt (pairK(A,B)) (Nonce Nb)} ∈ set evs;  
   ¬illegalUse(Card B);  
   evs ∈ sr ]]  
 ⇒ Key K ∈ analz (knows B evs)"  
 <proof>

**lemma** *B\_keydist\_to\_A*:  
 "[[ Outpts (Card B) B {Nonce Nb, Key K, Certificate,  
                           (Crypt (pairK(A,B)) (Nonce Nb))} ∈ set evs;  
   Gets B (Crypt (pairK(A,B)) (Nonce Nb)) ∈ set evs;  
   B ≠ Spy; ¬illegalUse(Card A); ¬illegalUse(Card B);  
   evs ∈ sr ]]  
 ⇒ Key K ∈ analz (knows A evs)"  
 <proof>



**lemma** *Nb\_certificate\_authentic\_B:*

```
"[ Gets B (Crypt (pairK(A,B)) (Nonce Nb)) ∈ set evs;
  B ≠ Spy; ¬illegalUse(Card B);
  evs ∈ sr ]
⇒ ∃ Na.
  Outpts (Card B) B {Nonce Nb, Key (sesK(Nb,pairK(A,B))),
    Crypt (pairK(A,B)) {Nonce Na, Nonce Nb},
    Crypt (pairK(A,B)) (Nonce Nb)} ∈ set evs"
```

*<proof>*

**lemma** *Pairkey\_certificate\_authentic\_A\_Card:*

```
"[ Inputs A (Card A)
  {Agent B, Nonce Na, Nonce Nb, Nonce Pk,
   Crypt (shrK A) {Nonce Pk, Agent B},
   Cert2, Cert3} ∈ set evs;
  A ≠ Spy; Card A ∉ cloned; evs ∈ sr ]
⇒ Pk = Pairkey(A,B) ∧
  Says Server A {Nonce (Pairkey(A,B)),
    Crypt (shrK A) {Nonce (Pairkey(A,B)), Agent B}}
  ∈ set evs "
```

*<proof>*

**lemma** *Na\_Nb\_certificate\_authentic\_A\_Card:*

```
"[ Inputs A (Card A)
  {Agent B, Nonce Na, Nonce Nb, Nonce Pk,
   Cert1,
   Crypt (pairK(A,B)) {Nonce Na, Nonce Nb}, Cert3} ∈ set evs;

  A ≠ Spy; ¬illegalUse(Card B); evs ∈ sr ]
⇒ Outpts (Card B) B {Nonce Nb, Key (sesK(Nb, pairK (A, B))),
  Crypt (pairK(A,B)) {Nonce Na, Nonce Nb},
  Crypt (pairK(A,B)) (Nonce Nb)}
  ∈ set evs "
```

*<proof>*

**lemma** *Na\_authentic\_A\_Card:*

```
"[ Inputs A (Card A)
  {Agent B, Nonce Na, Nonce Nb, Nonce Pk,
   Cert1, Cert2, Cert3} ∈ set evs;
  A ≠ Spy; evs ∈ sr ]
⇒ Outpts (Card A) A {Nonce Na, Cert3}
  ∈ set evs "
```

$\langle proof \rangle$

**lemma** *Inputs\_A\_Card\_9\_authentic:*

"[[ Inputs A (Card A)  
 {Agent B, Nonce Na, Nonce Nb, Nonce Pk,  
 Crypt (shrK A) {Nonce Pk, Agent B},  
 Crypt (pairK(A,B)) {Nonce Na, Nonce Nb}, Cert3} ∈ set evs;

A ≠ Spy; Card A ∉ cloned; ¬illegalUse(Card B); evs ∈ sr ]]  
 ⇒ Says Server A {Nonce Pk, Crypt (shrK A) {Nonce Pk, Agent B}}  
 ∈ set evs ∧  
 Outpts (Card B) B {Nonce Nb, Key (sesK(Nb, pairK (A, B))),  
 Crypt (pairK(A,B)) {Nonce Na, Nonce Nb},  
 Crypt (pairK(A,B)) (Nonce Nb)}  
 ∈ set evs ∧  
 Outpts (Card A) A {Nonce Na, Cert3}  
 ∈ set evs"

$\langle proof \rangle$

**lemma** *SR4\_imp:*

"[[ Outpts (Card A) A {Nonce Na, Crypt (crdK (Card A)) (Nonce Na)}  
 ∈ set evs;  
 A ≠ Spy; evs ∈ sr ]]  
 ⇒ ∃ Pk V. Gets A {Pk, V} ∈ set evs"

$\langle proof \rangle$

**lemma** *SR7\_imp:*

"[[ Outpts (Card B) B {Nonce Nb, Key K,  
 Crypt (pairK(A,B)) {Nonce Na, Nonce Nb},  
 Cert2} ∈ set evs;  
 B ≠ Spy; evs ∈ sr ]]  
 ⇒ Gets B {Agent A, Nonce Na} ∈ set evs"

$\langle proof \rangle$

**lemma** *SR10\_imp*:  
 "[ Outpts (Card A) A {Key K, Crypt (pairK(A,B)) (Nonce Nb)}  
   ∈ set evs;  
   A ≠ Spy; evs ∈ sr ]  
 ⇒ ∃ Cert1 Cert2.  
     Gets A {Nonce (Pairkey (A, B)), Cert1} ∈ set evs ∧  
     Gets A {Nonce Nb, Cert2} ∈ set evs"  
 <proof>

**lemma** *Outpts\_Server\_not\_evs*: "evs ∈ sr ⇒ Outpts (Card Server) P X ∉ set evs"  
 <proof>

*step2\_integrity* also is a reliability theorem

**lemma** *Says\_Server\_message\_form*:  
 "[ Says Server A {Pk, Certificate} ∈ set evs;  
   evs ∈ sr ]  
 ⇒ ∃ B. Pk = Nonce (Pairkey(A,B)) ∧  
     Certificate = Crypt (shrK A) {Nonce (Pairkey(A,B)), Agent B}"  
 <proof>

*step4integrity* is *Outpts\_A\_Card\_form\_4*

*step7integrity* is *Outpts\_B\_Card\_form\_7*

**lemma** *step8\_integrity*:  
 "[ Says B A {Nonce Nb, Certificate} ∈ set evs;  
   B ≠ Server; B ≠ Spy; evs ∈ sr ]  
 ⇒ ∃ Cert2 K.  
     Outpts (Card B) B {Nonce Nb, Key K, Certificate, Cert2} ∈ set evs"  
 <proof>

*step9integrity* is *Inputs\_A\_Card\_form\_9*

*step10integrity* is *Outpts\_A\_Card\_form\_10*.

**lemma** *step11\_integrity*:  
 "[ Says A B (Certificate) ∈ set evs;  
   ∀ p q. Certificate ≠ {p, q};  
   A ≠ Spy; evs ∈ sr ]  
 ⇒ ∃ K.  
     Outpts (Card A) A {Key K, Certificate} ∈ set evs"  
 <proof>

**end**

## 26 Bella's modification of the Shoup-Rubin protocol

**theory** ShoupRubinBella **imports** Smartcard **begin**

The modifications are that message 7 now mentions A, while message 10 now mentions Nb and B. The lack of explicitness of the original version was discovered by investigating adherence to the principle of Goal Availability. Only the updated version makes the goals of confidentiality, authentication and key distribution available to both peers.

**consts**

*sesK* :: "nat\*key => key"

**axioms**

*inj\_sesK* [iff]: "(*sesK*(*m*,*k*) = *sesK*(*m'*,*k'*)) = (*m* = *m'* ∧ *k* = *k'*)"

*shrK\_disj\_sesK* [iff]: "*shrK* A ≠ *sesK*(*m*,*pk*)"

*crdK\_disj\_sesK* [iff]: "*crdK* C ≠ *sesK*(*m*,*pk*)"

*pin\_disj\_sesK* [iff]: "*pin* P ≠ *sesK*(*m*,*pk*)"

*pairK\_disj\_sesK*[iff]: "*pairK*(A,B) ≠ *sesK*(*m*,*pk*)"

*Atomic\_distrib* [iff]: "*Atomic'*(*KEY'*K ∪ *NONCE'*N) =  
                          *Atomic'*(*KEY'*K) ∪ *Atomic'*(*NONCE'*N)"

*shouprubin\_assumes\_securemeans* [iff]: "*evs* ∈ *srb* ⇒ *secureM*"

**constdefs**

*Unique* :: "[event, event list] => bool" ("Unique \_ on \_")  
*Unique* ev on *evs* ==  
 ev ∉ set (tl (dropWhile (% z. z ≠ ev) *evs*))"

**inductive\_set** *srb* :: "event list set"

**where**

*Nil*: "[ ] ∈ *srb*"

| *Fake*: "[[ *evsF* ∈ *srb*; *X* ∈ synth (analz (knows Spy *evsF*));  
           illegalUse(Card B) ]]  
       ⇒ Says Spy A *X* #  
           Inputs Spy (Card B) *X* # *evsF* ∈ *srb*"

| *Forge*:

```

"[[ evsFo ∈ srb; Nonce Nb ∈ analz (knows Spy evsFo);
   Key (pairK(A,B)) ∈ knows Spy evsFo ]]
⇒ Notes Spy (Key (sesK(Nb,pairK(A,B)))) # evsFo ∈ srb"

/ Reception: "[[ evsrb ∈ srb; Says A B X ∈ set evsrb ]]
              ⇒ Gets B X # evsrb ∈ srb"

/ SR_U1: "[[ evs1 ∈ srb; A ≠ Server ]]
          ⇒ Says A Server {Agent A, Agent B}
              # evs1 ∈ srb"

/ SR_U2: "[[ evs2 ∈ srb;
             Gets Server {Agent A, Agent B} ∈ set evs2 ]]
          ⇒ Says Server A {Nonce (Pairkey(A,B)),
                           Crypt (shrK A) {Nonce (Pairkey(A,B)), Agent B}}
              # evs2 ∈ srb"

/ SR_U3: "[[ evs3 ∈ srb; legalUse(Card A);
             Says A Server {Agent A, Agent B} ∈ set evs3;
             Gets A {Nonce Pk, Certificate} ∈ set evs3 ]]
          ⇒ Inputs A (Card A) (Agent A)
              # evs3 ∈ srb"

/ SR_U4: "[[ evs4 ∈ srb;
             Nonce Na ∉ used evs4; legalUse(Card A); A ≠ Server;
             Inputs A (Card A) (Agent A) ∈ set evs4 ]]
          ⇒ Outputs (Card A) A {Nonce Na, Crypt (crdK (Card A)) (Nonce Na)}
              # evs4 ∈ srb"

/ SR_U4Fake: "[[ evs4F ∈ srb; Nonce Na ∉ used evs4F;
                 illegalUse(Card A);
                 Inputs Spy (Card A) (Agent A) ∈ set evs4F ]]
              ⇒ Outputs (Card A) Spy {Nonce Na, Crypt (crdK (Card A)) (Nonce Na)}
                  # evs4F ∈ srb"

/ SR_U5: "[[ evs5 ∈ srb;

```

$\text{Outpts (Card A) } A \{ \text{Nonce Na, Certificate} \} \in \text{set evs5};$   
 $\forall p q. \text{Certificate} \neq \{p, q\}$   
 $\implies \text{Says A B } \{ \text{Agent A, Nonce Na} \} \# \text{ evs5} \in \text{srb}"$

$/ \text{ SR\_U6: } "[ \text{ evs6} \in \text{srb}; \text{ legalUse(Card B)};$   
 $\text{ Gets B } \{ \text{Agent A, Nonce Na} \} \in \text{set evs6} ]$   
 $\implies \text{ Inputs B (Card B) } \{ \text{Agent A, Nonce Na} \}$   
 $\# \text{ evs6} \in \text{srb}"$

$/ \text{ SR\_U7: } "[ \text{ evs7} \in \text{srb};$   
 $\text{ Nonce Nb} \notin \text{used evs7}; \text{ legalUse(Card B)}; B \neq \text{Server};$   
 $K = \text{sesK(Nb, pairK(A,B))};$   
 $\text{ Key } K \notin \text{used evs7};$   
 $\text{ Inputs B (Card B) } \{ \text{Agent A, Nonce Na} \} \in \text{set evs7}]$   
 $\implies \text{ Outpts (Card B) } B \{ \text{Nonce Nb, Agent A, Key K,}$   
 $\text{ Crypt (pairK(A,B)) } \{ \text{Nonce Na, Nonce Nb} \},$   
 $\text{ Crypt (pairK(A,B)) (Nonce Nb)} \}$   
 $\# \text{ evs7} \in \text{srb}"$

$/ \text{ SR\_U7Fake: } "[ \text{ evs7F} \in \text{srb}; \text{ Nonce Nb} \notin \text{used evs7F};$   
 $\text{ illegalUse(Card B)};$   
 $K = \text{sesK(Nb, pairK(A,B))};$   
 $\text{ Key } K \notin \text{used evs7F};$   
 $\text{ Inputs Spy (Card B) } \{ \text{Agent A, Nonce Na} \} \in \text{set evs7F} ]$   
 $\implies \text{ Outpts (Card B) } \text{Spy} \{ \text{Nonce Nb, Agent A, Key K,}$   
 $\text{ Crypt (pairK(A,B)) } \{ \text{Nonce Na, Nonce Nb} \},$   
 $\text{ Crypt (pairK(A,B)) (Nonce Nb)} \}$   
 $\# \text{ evs7F} \in \text{srb}"$

$/ \text{ SR\_U8: } "[ \text{ evs8} \in \text{srb};$   
 $\text{ Inputs B (Card B) } \{ \text{Agent A, Nonce Na} \} \in \text{set evs8};$   
 $\text{ Outpts (Card B) } B \{ \text{Nonce Nb, Agent A, Key K,}$   
 $\text{ Cert1, Cert2} \} \in \text{set evs8} ]$   
 $\implies \text{ Says B A } \{ \text{Nonce Nb, Cert1} \} \# \text{ evs8} \in \text{srb}"$

$/ \text{ SR\_U9: } "[ \text{ evs9} \in \text{srb}; \text{ legalUse(Card A)};$   
 $\text{ Gets A } \{ \text{Nonce Pk, Cert1} \} \in \text{set evs9};$   
 $\text{ Outpts (Card A) } A \{ \text{Nonce Na, Cert2} \} \in \text{set evs9};$

```

    Gets A {Nonce Nb, Cert3} ∈ set evs9;
    ∀ p q. Cert2 ≠ {p, q}
  ⇒ Inputs A (Card A)
    {Agent B, Nonce Na, Nonce Nb, Nonce Pk,
     Cert1, Cert3, Cert2}
    # evs9 ∈ srb"

/ SR_U10: "[ evs10 ∈ srb; legalUse(Card A); A ≠ Server;
  K = sesK(Nb, pairK(A,B));
  Inputs A (Card A) {Agent B, Nonce Na, Nonce Nb,
    Nonce (Pairkey(A,B)),
    Crypt (shrK A) {Nonce (Pairkey(A,B)),
      Agent B},
    Crypt (pairK(A,B)) {Nonce Na, Nonce Nb},

    Crypt (crdK (Card A)) (Nonce Na)}
  ∈ set evs10 ]
  ⇒ Outputs (Card A) A {Agent B, Nonce Nb,
    Key K, Crypt (pairK(A,B)) (Nonce Nb)}
    # evs10 ∈ srb"

/ SR_U10Fake: "[ evs10F ∈ srb;
  illegalUse(Card A);
  K = sesK(Nb, pairK(A,B));
  Inputs Spy (Card A) {Agent B, Nonce Na, Nonce Nb,
    Nonce (Pairkey(A,B)),
    Crypt (shrK A) {Nonce (Pairkey(A,B)),
      Agent B},
    Crypt (pairK(A,B)) {Nonce Na, Nonce Nb},

    Crypt (crdK (Card A)) (Nonce Na)}
  ∈ set evs10F ]
  ⇒ Outputs (Card A) Spy {Agent B, Nonce Nb,
    Key K, Crypt (pairK(A,B)) (Nonce Nb)}
    # evs10F ∈ srb"

/ SR_U11: "[ evs11 ∈ srb;
  Says A Server {Agent A, Agent B} ∈ set evs11;
  Outputs (Card A) A {Agent B, Nonce Nb, Key K, Certificate}
  ∈ set evs11 ]
  ⇒ Says A B (Certificate)
    # evs11 ∈ srb"

/ Ops1:
  "[ evs01 ∈ srb;

```

```

      Outpts (Card B) B {Nonce Nb, Agent A, Key K, Cert1, Cert2}
      ∈ set evs01 ]
    ⇒ Notes Spy {Key K, Nonce Nb, Agent A, Agent B} # evs01 ∈ srb"

```

```

/ Ops2:

```

```

  "[ evs02 ∈ srb;
    Outpts (Card A) A {Agent B, Nonce Nb, Key K, Certificate}
    ∈ set evs02 ]
  ⇒ Notes Spy {Key K, Nonce Nb, Agent A, Agent B} # evs02 ∈ srb"

```

```

declare Fake_parts_insert_in_Un [dest]
declare analz_into_parts [dest]

```

```

lemma Gets_imp_Says:

```

```

  "[ Gets B X ∈ set evs; evs ∈ srb ] ⇒ ∃ A. Says A B X ∈ set evs"
<proof>

```

```

lemma Gets_imp_knows_Spy:

```

```

  "[ Gets B X ∈ set evs; evs ∈ srb ] ⇒ X ∈ knows Spy evs"
<proof>

```

```

lemma Gets_imp_knows_Spy_parts_Snd:

```

```

  "[ Gets B {X, Y} ∈ set evs; evs ∈ srb ] ⇒ Y ∈ parts (knows Spy evs)"
<proof>

```

```

lemma Gets_imp_knows_Spy_analz_Snd:

```

```

  "[ Gets B {X, Y} ∈ set evs; evs ∈ srb ] ⇒ Y ∈ analz (knows Spy evs)"
<proof>

```

```

lemma Inputs_imp_knows_Spy_secureM_srb:

```

```

  "[ Inputs Spy C X ∈ set evs; evs ∈ srb ] ⇒ X ∈ knows Spy evs"
<proof>

```

```

lemma knows_Spy_Inputs_secureM_srb_Spy:

```

```

  "evs ∈ srb ⇒ knows Spy (Inputs Spy C X # evs) = insert X (knows Spy
  evs)"
<proof>

```



**lemma** *knows\_Spy\_Inputs\_secureM\_srb:*

"[[  $A \neq \text{Spy}$ ;  $\text{evs} \in \text{srb}$  ]]  $\implies \text{knows Spy (Inputs } A \ C \ X \ \# \ \text{evs}) = \text{knows Spy evs}$ "  
 <proof>

**lemma** *knows\_Spy\_Outpts\_secureM\_srb\_Spy:*

" $\text{evs} \in \text{srb} \implies \text{knows Spy (Outpts } C \ \text{Spy } X \ \# \ \text{evs}) = \text{insert } X \ (\text{knows Spy evs})$ "  
 <proof>

**lemma** *knows\_Spy\_Outpts\_secureM\_srb:*

"[[  $A \neq \text{Spy}$ ;  $\text{evs} \in \text{srb}$  ]]  $\implies \text{knows Spy (Outpts } C \ A \ X \ \# \ \text{evs}) = \text{knows Spy evs}$ "  
 <proof>

**lemma** *Inputs\_A\_Card\_3:*

"[[  $\text{Inputs } A \ C \ (\text{Agent } A) \in \text{set evs}$ ;  $A \neq \text{Spy}$ ;  $\text{evs} \in \text{srb}$  ]]  
 $\implies \text{legalUse}(C) \wedge C = (\text{Card } A) \wedge$   
 $(\exists \text{ Pk Certificate. Gets } A \ \{\text{Pk, Certificate}\} \in \text{set evs})$ "  
 <proof>

**lemma** *Inputs\_B\_Card\_6:*

"[[  $\text{Inputs } B \ C \ \{\text{Agent } A, \text{Nonce Na}\} \in \text{set evs}$ ;  $B \neq \text{Spy}$ ;  $\text{evs} \in \text{srb}$  ]]  
 $\implies \text{legalUse}(C) \wedge C = (\text{Card } B) \wedge \text{Gets } B \ \{\text{Agent } A, \text{Nonce Na}\} \in \text{set evs}$ "  
 <proof>

**lemma** *Inputs\_A\_Card\_9:*

"[[  $\text{Inputs } A \ C \ \{\text{Agent } B, \text{Nonce Na}, \text{Nonce Nb}, \text{Nonce Pk},$   
 $\text{Cert1}, \text{Cert2}, \text{Cert3}\} \in \text{set evs}$ ;  
 $A \neq \text{Spy}$ ;  $\text{evs} \in \text{srb}$  ]]  
 $\implies \text{legalUse}(C) \wedge C = (\text{Card } A) \wedge$   
 $\text{Gets } A \ \{\text{Nonce Pk}, \text{Cert1}\} \in \text{set evs} \quad \wedge$   
 $\text{Outpts } (\text{Card } A) \ A \ \{\text{Nonce Na}, \text{Cert3}\} \in \text{set evs} \quad \wedge$   
 $\text{Gets } A \ \{\text{Nonce Nb}, \text{Cert2}\} \in \text{set evs}$ "  
 <proof>

**lemma** *Outpts\_A\_Card\_4:*

"[[  $\text{Outpts } C \ A \ \{\text{Nonce Na}, (\text{Crypt } (\text{crdK } (\text{Card } A)) \ (\text{Nonce Na}))\} \in \text{set evs}$ ;  
 $\text{evs} \in \text{srb}$  ]]  
 $\implies \text{legalUse}(C) \wedge C = (\text{Card } A) \wedge$

```

lemma Outpts_honest_B_Card_7:
  "[ Outpts C B {Nonce Nb, Agent A, Key K, Cert1, Cert2} ∈ set evs;
    B ≠ Spy; evs ∈ srb ]

```

$\implies \text{legalUse}(C) \wedge C = (\text{Card } B) \wedge$   
 $(\exists \text{ Na. Inputs } B (\text{Card } B) \{ \text{Agent } A, \text{Nonce Na} \} \in \text{set evs})"$   
 <proof>

**lemma** *Outpts\_honest\_A\_Card\_10:*  
 $"[ \text{Outpts } C \ A \ \{ \text{Agent } B, \text{Nonce Nb, Key } K, \text{Certificate} \} \in \text{set evs};$   
 $A \neq \text{Spy}; \text{evs} \in \text{srb} ]$   
 $\implies \text{legalUse } (C) \wedge C = (\text{Card } A) \wedge$   
 $(\exists \text{ Na Pk Ver1 Ver2 Ver3.}$   
 $\text{Inputs } A (\text{Card } A) \{ \text{Agent } B, \text{Nonce Na, Nonce Nb, Pk,}$   
 $\text{Ver1, Ver2, Ver3} \} \in \text{set evs})"$   
 <proof>

**lemma** *Outpts\_which\_Card\_4:*  
 $"[ \text{Outpts } (\text{Card } A) \ A \ \{ \text{Nonce Na, Crypt } K \ X \} \in \text{set evs}; \text{evs} \in \text{srb} ]$   
 $\implies \text{Inputs } A (\text{Card } A) (\text{Agent } A) \in \text{set evs}"$   
 <proof>

**lemma** *Outpts\_which\_Card\_7:*  
 $"[ \text{Outpts } (\text{Card } B) \ B \ \{ \text{Nonce Nb, Agent } A, \text{Key } K, \text{Cert1, Cert2} \}$   
 $\in \text{set evs}; \text{evs} \in \text{srb} ]$   
 $\implies \exists \text{ Na. Inputs } B (\text{Card } B) \{ \text{Agent } A, \text{Nonce Na} \} \in \text{set evs}"$   
 <proof>

**lemma** *Outpts\_which\_Card\_10:*  
 $"[ \text{Outpts } (\text{Card } A) \ A \ \{ \text{Agent } B, \text{Nonce Nb, Key } K, \text{Certificate} \} \in \text{set evs};$   
 $\text{evs} \in \text{srb} ]$   
 $\implies \exists \text{ Na. Inputs } A (\text{Card } A) \{ \text{Agent } B, \text{Nonce Na, Nonce Nb, Nonce } (\text{Pairkey}(A,B)),$   
 $\text{Crypt } (\text{shrK } A) \ \{ \text{Nonce } (\text{Pairkey}(A,B)), \text{Agent } B \},$   
 $\text{Crypt } (\text{pairK}(A,B)) \ \{ \text{Nonce Na, Nonce Nb} \},$   
 $\text{Crypt } (\text{crdK } (\text{Card } A)) (\text{Nonce Na}) \} \in \text{set evs}"$   
 <proof>

**lemma** *Outpts\_A\_Card\_form\_4:*  
 $"[ \text{Outpts } (\text{Card } A) \ A \ \{ \text{Nonce Na, Certificate} \} \in \text{set evs};$   
 $\forall p \ q. \text{Certificate} \neq \{ p, q \}; \text{evs} \in \text{srb} ]$   
 $\implies \text{Certificate} = (\text{Crypt } (\text{crdK } (\text{Card } A)) (\text{Nonce Na}))"$   
 <proof>

**lemma** *Outpts\_B\_Card\_form\_7:*  
 $"[ \text{Outpts } (\text{Card } B) \ B \ \{ \text{Nonce Nb, Agent } A, \text{Key } K, \text{Cert1, Cert2} \}$   
 $\in \text{set evs}; \text{evs} \in \text{srb} ]$   
 $\implies \exists \text{ Na.}$

$K = \text{sesK}(\text{Nb}, \text{pairK}(A, B)) \wedge$   
 $\text{Cert1} = (\text{Crypt}(\text{pairK}(A, B)) \{ \text{Nonce Na}, \text{Nonce Nb} \}) \wedge$   
 $\text{Cert2} = (\text{Crypt}(\text{pairK}(A, B)) (\text{Nonce Nb}))$   
 $\langle \text{proof} \rangle$

**lemma** *Outpts\_A\_Card\_form\_10*:  
 $"[ \text{Outpts}(\text{Card } A) \ A \ \{ \text{Agent } B, \text{Nonce Nb}, \text{Key } K, \text{Certificate} \}$   
 $\in \text{set evs}; \text{evs} \in \text{srb} ]$   
 $\implies K = \text{sesK}(\text{Nb}, \text{pairK}(A, B)) \wedge$   
 $\text{Certificate} = (\text{Crypt}(\text{pairK}(A, B)) (\text{Nonce Nb}))"$   
 $\langle \text{proof} \rangle$

**lemma** *Outpts\_A\_Card\_form\_bis*:  
 $"[ \text{Outpts}(\text{Card } A') \ A' \ \{ \text{Agent } B', \text{Nonce Nb}', \text{Key}(\text{sesK}(\text{Nb}, \text{pairK}(A, B)))$   
 $\text{Certificate} \} \in \text{set evs};$   
 $\text{evs} \in \text{srb} ]$   
 $\implies A' = A \wedge B' = B \wedge \text{Nb} = \text{Nb}' \wedge$   
 $\text{Certificate} = (\text{Crypt}(\text{pairK}(A, B)) (\text{Nonce Nb}))"$   
 $\langle \text{proof} \rangle$

**lemma** *Inputs\_A\_Card\_form\_9*:  
 $"[ \text{Inputs } A(\text{Card } A) \ \{ \text{Agent } B, \text{Nonce Na}, \text{Nonce Nb}, \text{Nonce Pk},$   
 $\text{Cert1}, \text{Cert2}, \text{Cert3} \} \in \text{set evs};$   
 $\text{evs} \in \text{srb} ]$   
 $\implies \text{Cert3} = \text{Crypt}(\text{crdK}(\text{Card } A)) (\text{Nonce Na})"$   
 $\langle \text{proof} \rangle$

**lemma** *Inputs\_Card\_legalUse*:  
 $"[ \text{Inputs } A(\text{Card } A) \ X \in \text{set evs}; \text{evs} \in \text{srb} ] \implies \text{legalUse}(\text{Card } A)"$   
 $\langle \text{proof} \rangle$

**lemma** *Outpts\_Card\_legalUse*:  
 $"[ \text{Outpts}(\text{Card } A) \ A \ X \in \text{set evs}; \text{evs} \in \text{srb} ] \implies \text{legalUse}(\text{Card } A)"$   
 $\langle \text{proof} \rangle$

**lemma** *Inputs\_Card*:  $"[ \text{Inputs } A \ C \ X \in \text{set evs}; A \neq \text{Spy}; \text{evs} \in \text{srb} ]$   
 $\implies C = (\text{Card } A) \wedge \text{legalUse}(C)"$   
 $\langle \text{proof} \rangle$

**lemma** *Outpts\_Card*:  $"[ \text{Outpts } C \ A \ X \in \text{set evs}; A \neq \text{Spy}; \text{evs} \in \text{srb} ]$

$\implies C = (\text{Card } A) \wedge \text{legalUse}(C)$   
 $\langle \text{proof} \rangle$

**lemma** *Inputs\_Outpts\_Card:*  
 $"[ \text{Inputs } A \ C \ X \in \text{set evs} \vee \text{Outpts } C \ A \ Y \in \text{set evs};$   
 $A \neq \text{Spy}; \text{evs} \in \text{srb} ]$   
 $\implies C = (\text{Card } A) \wedge \text{legalUse}(\text{Card } A)"$   
 $\langle \text{proof} \rangle$

**lemma** *Inputs\_Card\_Spy:*  
 $"[ \text{Inputs Spy } C \ X \in \text{set evs} \vee \text{Outpts } C \ \text{Spy } X \in \text{set evs}; \text{evs} \in \text{srb} ]$   
 $\implies C = (\text{Card Spy}) \wedge \text{legalUse}(\text{Card Spy}) \vee$   
 $(\exists A. C = (\text{Card } A) \wedge \text{illegalUse}(\text{Card } A))"$   
 $\langle \text{proof} \rangle$

**lemma** *Outpts\_A\_Card\_unique\_nonce:*  
 $"[ \text{Outpts } (\text{Card } A) \ A \ \{ \text{Nonce Na}, \text{Crypt } (\text{crdK } (\text{Card } A)) \ (\text{Nonce Na}) \}$   
 $\in \text{set evs};$   
 $\text{Outpts } (\text{Card } A') \ A' \ \{ \text{Nonce Na}, \text{Crypt } (\text{crdK } (\text{Card } A')) \ (\text{Nonce Na}) \}$   
 $\in \text{set evs};$   
 $\text{evs} \in \text{srb} ] \implies A=A'"$   
 $\langle \text{proof} \rangle$

**lemma** *Outpts\_B\_Card\_unique\_nonce:*  
 $"[ \text{Outpts } (\text{Card } B) \ B \ \{ \text{Nonce Nb}, \text{Agent } A, \text{Key } SK, \text{Cert1}, \text{Cert2} \} \in \text{set}$   
 $\text{evs};$   
 $\text{Outpts } (\text{Card } B') \ B' \ \{ \text{Nonce Nb}, \text{Agent } A', \text{Key } SK', \text{Cert1}', \text{Cert2}' \} \in$   
 $\text{set evs};$   
 $\text{evs} \in \text{srb} ] \implies B=B' \wedge A=A' \wedge SK=SK' \wedge \text{Cert1}=\text{Cert1}' \wedge \text{Cert2}=\text{Cert2}'"$   
 $\langle \text{proof} \rangle$

**lemma** *Outpts\_B\_Card\_unique\_key:*  
 $"[ \text{Outpts } (\text{Card } B) \ B \ \{ \text{Nonce Nb}, \text{Agent } A, \text{Key } SK, \text{Cert1}, \text{Cert2} \} \in \text{set}$   
 $\text{evs};$   
 $\text{Outpts } (\text{Card } B') \ B' \ \{ \text{Nonce Nb}', \text{Agent } A', \text{Key } SK, \text{Cert1}', \text{Cert2}' \} \in$

set evs;  
 evs ∈ srb ] ⇒ B=B' ∧ A=A' ∧ Nb=Nb' ∧ Cert1=Cert1' ∧ Cert2=Cert2'"  
 ⟨proof⟩

**lemma** Outpts\_A\_Card\_unique\_key:  
 "[ Outpts (Card A) A {Agent B, Nonce Nb, Key K, V} ∈ set evs;  
 Outpts (Card A') A' {Agent B', Nonce Nb', Key K, V'} ∈ set evs;  
 evs ∈ srb ] ⇒ A=A' ∧ B=B' ∧ Nb=Nb' ∧ V=V'"  
 ⟨proof⟩

**lemma** Outpts\_A\_Card\_Unique:  
 "[ Outpts (Card A) A {Nonce Na, rest} ∈ set evs; evs ∈ srb ]  
 ⇒ Unique (Outpts (Card A) A {Nonce Na, rest}) on evs"  
 ⟨proof⟩

**lemma** Spy\_knows\_Na:  
 "[ Says A B {Agent A, Nonce Na} ∈ set evs; evs ∈ srb ]  
 ⇒ Nonce Na ∈ analz (knows Spy evs)"  
 ⟨proof⟩

**lemma** Spy\_knows\_Nb:  
 "[ Says B A {Nonce Nb, Certificate} ∈ set evs; evs ∈ srb ]  
 ⇒ Nonce Nb ∈ analz (knows Spy evs)"  
 ⟨proof⟩

**lemma** Pairkey\_Gets\_analz\_knows\_Spy:  
 "[ Gets A {Nonce (Pairkey(A,B)), Certificate} ∈ set evs; evs ∈ srb  
 ]  
 ⇒ Nonce (Pairkey(A,B)) ∈ analz (knows Spy evs)"  
 ⟨proof⟩

**lemma** Pairkey\_Inputs\_imp\_Gets:  
 "[ Inputs A (Card A)  
 {Agent B, Nonce Na, Nonce Nb, Nonce (Pairkey(A,B)),  
 Cert1, Cert3, Cert2} ∈ set evs;  
 A ≠ Spy; evs ∈ srb ]  
 ⇒ Gets A {Nonce (Pairkey(A,B)), Cert1} ∈ set evs"  
 ⟨proof⟩

```

lemma Pairkey_Inputs_analz_knows_Spy:
  "[[ Inputs A (Card A)
    {Agent B, Nonce Na, Nonce Nb, Nonce (Pairkey(A,B)),
      Cert1, Cert3, Cert2} ∈ set evs;
    evs ∈ srb ]]"
  ⇒ Nonce (Pairkey(A,B)) ∈ analz (knows Spy evs)"
⟨proof⟩

```

```

declare shrK_disj_sesK [THEN not_sym, iff]
declare pin_disj_sesK [THEN not_sym, iff]
declare crdK_disj_sesK [THEN not_sym, iff]
declare pairK_disj_sesK [THEN not_sym, iff]

```

⟨ML⟩

```

lemma Spy_parts_keys [simp]: "evs ∈ srb ⇒
  (Key (shrK P) ∈ parts (knows Spy evs)) = (Card P ∈ cloned) ∧
  (Key (pin P) ∈ parts (knows Spy evs)) = (P ∈ bad ∨ Card P ∈ cloned) ∧

  (Key (crdK C) ∈ parts (knows Spy evs)) = (C ∈ cloned) ∧
  (Key (pairK(A,B)) ∈ parts (knows Spy evs)) = (Card B ∈ cloned)"
⟨proof⟩

```

```

lemma Spy_analz_shrK[simp]: "evs ∈ srb ⇒
  (Key (shrK P) ∈ analz (knows Spy evs)) = (Card P ∈ cloned)"
⟨proof⟩

```

```

lemma Spy_analz_crdK[simp]: "evs ∈ srb ⇒
  (Key (crdK C) ∈ analz (knows Spy evs)) = (C ∈ cloned)"
⟨proof⟩

```

```

lemma Spy_analz_pairK[simp]: "evs ∈ srb ⇒
  (Key (pairK(A,B)) ∈ analz (knows Spy evs)) = (Card B ∈ cloned)"
⟨proof⟩

```

**lemma** *analz\_image\_Key\_Un\_Nonce*: "analz (Key'K  $\cup$  Nonce'N) = Key'K  $\cup$  Nonce'N"  
 <proof>

<ML>

**lemma** *analz\_image\_freshK* [rule\_format]:  
 "evs  $\in$  srb  $\implies \forall K KK.$   
 (Key K  $\in$  analz (Key'KK  $\cup$  (knows Spy evs))) =  
 (K  $\in$  KK  $\vee$  Key K  $\in$  analz (knows Spy evs))"  
 <proof>

**lemma** *analz\_insert\_freshK*: "evs  $\in$  srb  $\implies$   
 Key K  $\in$  analz (insert (Key K') (knows Spy evs)) =  
 (K = K'  $\vee$  Key K  $\in$  analz (knows Spy evs))"  
 <proof>

**lemma** *Na\_Nb\_certificate\_authentic*:  
 "[ Crypt (pairK(A,B)) {Nonce Na, Nonce Nb}  $\in$  parts (knows Spy evs);  
 $\neg$ illegalUse(Card B);  
 evs  $\in$  srb ]  
 $\implies$  Outpts (Card B) B {Nonce Nb, Agent A, Key (sesK(Nb,pairK(A,B))),  
 Crypt (pairK(A,B)) {Nonce Na, Nonce Nb},  
 Crypt (pairK(A,B)) (Nonce Nb)}  $\in$  set evs"  
 <proof>

**lemma** *Nb\_certificate\_authentic*:  
 "[ Crypt (pairK(A,B)) (Nonce Nb)  $\in$  parts (knows Spy evs);  
 B  $\neq$  Spy;  $\neg$ illegalUse(Card A);  $\neg$ illegalUse(Card B);  
 evs  $\in$  srb ]  
 $\implies$  Outpts (Card A) A {Agent B, Nonce Nb, Key (sesK(Nb,pairK(A,B))),  
 Crypt (pairK(A,B)) (Nonce Nb)}  $\in$  set evs"  
 <proof>

**lemma** *Outpts\_A\_Card\_imp\_pairK\_parts*:  
 "[ Outpts (Card A) A {Agent B, Nonce Nb,  
 Key K, Certificate}  $\in$  set evs;  
 evs  $\in$  srb ]  
 $\implies \exists Na.$  Crypt (pairK(A,B)) {Nonce Na, Nonce Nb}  $\in$  parts (knows Spy evs)"



$\langle proof \rangle$

**lemma** *Nb\_certificate\_authentic\_bis:*

"[ Crypt (pairK(A,B)) (Nonce Nb)  $\in$  parts (knows Spy evs);  
      $B \neq \text{Spy}$ ;  $\neg \text{illegalUse}(\text{Card } B)$ ;  
      $\text{evs} \in \text{srb}$  ]  
 $\implies \exists \text{Na. Outputs } (\text{Card } B) \ B \ \{\text{Nonce Nb, Agent A, Key } (\text{sesK}(\text{Nb}, \text{pairK}(\text{A}, \text{B})))$ ,  
     Crypt (pairK(A,B))  $\{\text{Nonce Na, Nonce Nb}\}$ ,  
     Crypt (pairK(A,B)) (Nonce Nb) $\} \in \text{set evs}$ "

$\langle proof \rangle$

**lemma** *Pairkey\_certificate\_authentic:*

"[ Crypt (shrK A)  $\{\text{Nonce Pk, Agent B}\} \in$  parts (knows Spy evs);  
     Card A  $\notin$  cloned;  $\text{evs} \in \text{srb}$  ]  
 $\implies \text{Pk} = \text{Pairkey}(\text{A}, \text{B}) \wedge$   
     Says Server A  $\{\text{Nonce Pk,}$   
         Crypt (shrK A)  $\{\text{Nonce Pk, Agent B}\}\}$   
      $\in \text{set evs}$ "

$\langle proof \rangle$

**lemma** *sesK\_authentic:*

"[ Key (sesK(Nb, pairK(A,B)))  $\in$  parts (knows Spy evs);  
      $A \neq \text{Spy}$ ;  $B \neq \text{Spy}$ ;  $\neg \text{illegalUse}(\text{Card } A)$ ;  $\neg \text{illegalUse}(\text{Card } B)$ ;  
      $\text{evs} \in \text{srb}$  ]  
 $\implies \text{Notes Spy } \{\text{Key } (\text{sesK}(\text{Nb}, \text{pairK}(\text{A}, \text{B}))), \text{Nonce Nb, Agent A, Agent B}\}$   
      $\in \text{set evs}$ "

$\langle proof \rangle$

**lemma** *Confidentiality:*

"[ Notes Spy  $\{\text{Key } (\text{sesK}(\text{Nb}, \text{pairK}(\text{A}, \text{B}))), \text{Nonce Nb, Agent A, Agent B}\}$   
      $\notin \text{set evs}$ ;  
      $A \neq \text{Spy}$ ;  $B \neq \text{Spy}$ ;  $\neg \text{illegalUse}(\text{Card } A)$ ;  $\neg \text{illegalUse}(\text{Card } B)$ ;  
      $\text{evs} \in \text{srb}$  ]  
 $\implies \text{Key } (\text{sesK}(\text{Nb}, \text{pairK}(\text{A}, \text{B}))) \notin \text{analz } (\text{knows Spy evs})"$

$\langle proof \rangle$

**lemma** *Confidentiality\_B:*

"[ Outputs (Card B) B  $\{\text{Nonce Nb, Agent A, Key K, Cert1, Cert2}\}$   
      $\in \text{set evs}$ ;  
     Notes Spy  $\{\text{Key K, Nonce Nb, Agent A, Agent B}\} \notin \text{set evs}$ ;  
      $A \neq \text{Spy}$ ;  $B \neq \text{Spy}$ ;  $\neg \text{illegalUse}(\text{Card } A)$ ; Card B  $\notin$  cloned;

$$\begin{aligned} & \text{evs} \in \text{srb} \text{ ]} \\ & \implies \text{Key } K \notin \text{analz } (\text{knows Spy evs})" \\ & \langle \text{proof} \rangle
\end{aligned}$$

**lemma** *A\_authenticates\_B*:  

$$\begin{aligned} & "[ \text{Outpts (Card A) A } \{ \text{Agent B, Nonce Nb, Key K, Certificate} \} \in \text{set evs}; \\ & \quad \neg \text{illegalUse (Card B)}; \\ & \quad \text{evs} \in \text{srb} \text{ ]} \\ & \implies \exists \text{ Na. Outpts (Card B) B } \{ \text{Nonce Nb, Agent A, Key K,} \\ & \quad \text{Crypt (pairK(A,B)) } \{ \text{Nonce Na, Nonce Nb} \}, \\ & \quad \text{Crypt (pairK(A,B)) (Nonce Nb)} \} \in \text{set evs}" \\ & \langle \text{proof} \rangle
\end{aligned}$$

**lemma** *A\_authenticates\_B\_Gets*:  

$$\begin{aligned} & "[ \text{Gets A } \{ \text{Nonce Nb, Crypt (pairK(A,B)) } \{ \text{Nonce Na, Nonce Nb} \} \} \\ & \quad \in \text{set evs}; \\ & \quad \neg \text{illegalUse (Card B)}; \\ & \quad \text{evs} \in \text{srb} \text{ ]} \\ & \implies \text{Outpts (Card B) B } \{ \text{Nonce Nb, Agent A, Key (sesK(Nb, pairK (A, B))),} \\ & \quad \text{Crypt (pairK(A,B)) } \{ \text{Nonce Na, Nonce Nb} \}, \\ & \quad \text{Crypt (pairK(A,B)) (Nonce Nb)} \} \in \text{set evs}" \\ & \langle \text{proof} \rangle
\end{aligned}$$

**lemma** *A\_authenticates\_B\_bis*:  

$$\begin{aligned} & "[ \text{Outpts (Card A) A } \{ \text{Agent B, Nonce Nb, Key K, Cert2} \} \in \text{set evs}; \\ & \quad \neg \text{illegalUse (Card B)}; \\ & \quad \text{evs} \in \text{srb} \text{ ]} \\ & \implies \exists \text{ Cert1. Outpts (Card B) B } \{ \text{Nonce Nb, Agent A, Key K, Cert1, Cert2} \} \\ & \quad \in \text{set evs}" \\ & \langle \text{proof} \rangle
\end{aligned}$$

**lemma** *B\_authenticates\_A*:  

$$\begin{aligned} & "[ \text{Gets B (Crypt (pairK(A,B)) (Nonce Nb))} \in \text{set evs}; \\ & \quad \text{B} \neq \text{Spy}; \neg \text{illegalUse (Card A)}; \neg \text{illegalUse (Card B)}; \\ & \quad \text{evs} \in \text{srb} \text{ ]} \\ & \implies \text{Outpts (Card A) A } \{ \text{Agent B, Nonce Nb,} \\ & \quad \text{Key (sesK(Nb, pairK(A,B))), Crypt (pairK(A,B)) (Nonce Nb)} \} \in \text{set evs}" \\ & \langle \text{proof} \rangle
\end{aligned}$$

**lemma** *B\_authenticates\_A\_bis*:  
 "[[ Outpts (Card B) B {Nonce Nb, Agent A, Key K, Cert1, Cert2} ∈ set evs;  
 Gets B (Cert2) ∈ set evs;  
 B ≠ Spy; ¬illegalUse(Card A); ¬illegalUse(Card B);  
 evs ∈ srb ]]  
 ⇒ Outpts (Card A) A {Agent B, Nonce Nb, Key K, Cert2} ∈ set evs"  
 <proof>

**lemma** *Confidentiality\_A*:  
 "[[ Outpts (Card A) A {Agent B, Nonce Nb,  
 Key K, Certificate} ∈ set evs;  
 Notes Spy {Key K, Nonce Nb, Agent A, Agent B} ∉ set evs;  
 A ≠ Spy; B ≠ Spy; ¬illegalUse(Card A); ¬illegalUse(Card B);  
 evs ∈ srb ]]  
 ⇒ Key K ∉ analz (knows Spy evs)"  
 <proof>

**lemma** *Outpts\_imp\_knows\_agents\_secureM\_srb*:  
 "[[ Outpts (Card A) A X ∈ set evs; evs ∈ srb ]] ⇒ X ∈ knows A evs"  
 <proof>

**lemma** *A\_keydist\_to\_B*:  
 "[[ Outpts (Card A) A {Agent B, Nonce Nb, Key K, Certificate} ∈ set evs;  
 ¬illegalUse(Card B);  
 evs ∈ srb ]]  
 ⇒ Key K ∈ analz (knows B evs)"  
 <proof>

**lemma** *B\_keydist\_to\_A*:  
 "[[ Outpts (Card B) B {Nonce Nb, Agent A, Key K, Cert1, Cert2} ∈ set evs;  
 Gets B (Cert2) ∈ set evs;  
 B ≠ Spy; ¬illegalUse(Card A); ¬illegalUse(Card B);  
 evs ∈ srb ]]  
 ⇒ Key K ∈ analz (knows A evs)"  
 <proof>

**lemma** *Nb\_certificate\_authentic\_B*:

"[ Gets B (Crypt (pairK(A,B)) (Nonce Nb)) ∈ set evs;  
 B ≠ Spy; ¬illegalUse(Card B);  
 evs ∈ srb ]  
 ⇒ ∃ Na.  
 Outpts (Card B) B {Nonce Nb, Agent A, Key (sesK(Nb,pairK(A,B))),  
 Crypt (pairK(A,B)) {Nonce Na, Nonce Nb},  
 Crypt (pairK(A,B)) (Nonce Nb)} ∈ set evs"  
 <proof>

**lemma** Pairkey\_certificate\_authentic\_A\_Card:  
 "[ Inputs A (Card A)  
 {Agent B, Nonce Na, Nonce Nb, Nonce Pk,  
 Crypt (shrK A) {Nonce Pk, Agent B},  
 Cert2, Cert3} ∈ set evs;  
 A ≠ Spy; Card A ∉ cloned; evs ∈ srb ]  
 ⇒ Pk = Pairkey(A,B) ∧  
 Says Server A {Nonce (Pairkey(A,B)),  
 Crypt (shrK A) {Nonce (Pairkey(A,B)), Agent B}}  
 ∈ set evs "  
 <proof>

**lemma** Na\_Nb\_certificate\_authentic\_A\_Card:  
 "[ Inputs A (Card A)  
 {Agent B, Nonce Na, Nonce Nb, Nonce Pk,  
 Cert1, Crypt (pairK(A,B)) {Nonce Na, Nonce Nb}, Cert3} ∈ set evs;  
 A ≠ Spy; ¬illegalUse(Card B); evs ∈ srb ]  
 ⇒ Outpts (Card B) B {Nonce Nb, Agent A, Key (sesK(Nb, pairK (A, B))),  
 Crypt (pairK(A,B)) {Nonce Na, Nonce Nb},  
 Crypt (pairK(A,B)) (Nonce Nb)}  
 ∈ set evs "  
 <proof>

**lemma** Na\_authentic\_A\_Card:  
 "[ Inputs A (Card A)  
 {Agent B, Nonce Na, Nonce Nb, Nonce Pk,  
 Cert1, Cert2, Cert3} ∈ set evs;  
 A ≠ Spy; evs ∈ srb ]  
 ⇒ Outpts (Card A) A {Nonce Na, Cert3}  
 ∈ set evs "  
 <proof>

**lemma** Inputs\_A\_Card\_9\_authentic:

```

"[[ Inputs A (Card A)
    {Agent B, Nonce Na, Nonce Nb, Nonce Pk,
     Crypt (shrK A) {Nonce Pk, Agent B},
     Crypt (pairK(A,B)) {Nonce Na, Nonce Nb}, Cert3} ∈ set evs;

A ≠ Spy; Card A ∉ cloned; ¬illegalUse(Card B); evs ∈ srb ]
⇒ Says Server A {Nonce Pk, Crypt (shrK A) {Nonce Pk, Agent B}}
    ∈ set evs ∧
   Outpts (Card B) B {Nonce Nb, Agent A, Key (sesK(Nb, pairK (A, B))),

                                Crypt (pairK(A,B)) {Nonce Na, Nonce Nb},
                                Crypt (pairK(A,B)) (Nonce Nb)}
    ∈ set evs ∧
   Outpts (Card A) A {Nonce Na, Cert3}
    ∈ set evs"
⟨proof⟩

```

```

lemma SR_U4_imp:
  "[[ Outpts (Card A) A {Nonce Na, Crypt (crdK (Card A)) (Nonce Na)}
    ∈ set evs;
   A ≠ Spy; evs ∈ srb ]
  ⇒ ∃ Pk V. Gets A {Pk, V} ∈ set evs"
⟨proof⟩

```

```

lemma SR_U7_imp:
  "[[ Outpts (Card B) B {Nonce Nb, Agent A, Key K,
    Crypt (pairK(A,B)) {Nonce Na, Nonce Nb},
    Cert2} ∈ set evs;
   B ≠ Spy; evs ∈ srb ]
  ⇒ Gets B {Agent A, Nonce Na} ∈ set evs"
⟨proof⟩

```

```

lemma SR_U10_imp:
  "[[ Outpts (Card A) A {Agent B, Nonce Nb,
    Key K, Crypt (pairK(A,B)) (Nonce Nb)}
    ∈ set evs;
   A ≠ Spy; evs ∈ srb ]
  ⇒ ∃ Cert1 Cert2.

```

$$\begin{aligned} & \text{Gets } A \{ \text{Nonce } (\text{Pairkey } (A, B)), \text{Cert1} \} \in \text{set evs} \wedge \\ & \text{Gets } A \{ \text{Nonce } Nb, \text{Cert2} \} \in \text{set evs}'' \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *Outpts\_Server\_not\_evs*:  

$$" \text{evs} \in \text{srb} \implies \text{Outpts } (\text{Card Server}) P X \notin \text{set evs} "$$

$$\langle \text{proof} \rangle$$

*step2\_integrity* also is a reliability theorem

**lemma** *Says\_Server\_message\_form*:  

$$\begin{aligned} & "[ \text{Says Server } A \{ \text{Pk}, \text{Certificate} \} \in \text{set evs}; \\ & \quad \text{evs} \in \text{srb} ] \\ & \implies \exists B. \text{Pk} = \text{Nonce } (\text{Pairkey}(A, B)) \wedge \\ & \quad \text{Certificate} = \text{Crypt } (\text{shrK } A) \{ \text{Nonce } (\text{Pairkey}(A, B)), \text{Agent } B \} " \\ & \langle \text{proof} \rangle \end{aligned}$$

*step4\_integrity* is *Outpts\_A\_Card\_form\_4*

*step7\_integrity* is *Outpts\_B\_Card\_form\_7*

**lemma** *step8\_integrity*:  

$$\begin{aligned} & "[ \text{Says } B A \{ \text{Nonce } Nb, \text{Certificate} \} \in \text{set evs}; \\ & \quad B \neq \text{Server}; B \neq \text{Spy}; \text{evs} \in \text{srb} ] \\ & \implies \exists \text{Cert2 } K. \\ & \quad \text{Outpts } (\text{Card } B) B \{ \text{Nonce } Nb, \text{Agent } A, \text{Key } K, \text{Certificate}, \text{Cert2} \} \in \text{set} \\ & \quad \text{evs} " \\ & \langle \text{proof} \rangle \end{aligned}$$

*step9\_integrity* is *Inputs\_A\_Card\_form\_9* *step10\_integrity* is *Outpts\_A\_Card\_form\_10*.

**lemma** *step11\_integrity*:  

$$\begin{aligned} & "[ \text{Says } A B (\text{Certificate}) \in \text{set evs}; \\ & \quad \forall p \ q. \text{Certificate} \neq \{ p, q \}; \\ & \quad A \neq \text{Spy}; \text{evs} \in \text{srb} ] \\ & \implies \exists K \ Nb. \\ & \quad \text{Outpts } (\text{Card } A) A \{ \text{Agent } B, \text{Nonce } Nb, \text{Key } K, \text{Certificate} \} \in \text{set evs} " \\ & \langle \text{proof} \rangle \end{aligned}$$

**end**

## 27 Extensions to Standard Theories

**theory** *Extensions* imports "../Event" begin

### 27.1 Extensions to Theory *Set*

**lemma** *eq*: " $[ \mid \neg \exists x. x:A \implies x:B; \neg \exists x. x:B \implies x:A \mid ] \implies A=B$ "

*<proof>*

**lemma** insert\_Un: "P ({x} Un A) ==> P (insert x A)"

*<proof>*

**lemma** in\_sub: "x:A ==> {x}<=A"

*<proof>*

## 27.2 Extensions to Theory List

### 27.2.1 "remove l x" erase the first element of "l" equal to "x"

**consts** remove :: "'a list => 'a => 'a list"

**primrec**

"remove [] y = []"

"remove (x#xs) y = (if x=y then xs else x # remove xs y)"

**lemma** set\_remove: "set (remove l x) <= set l"

*<proof>*

## 27.3 Extensions to Theory Message

### 27.3.1 declarations for tactics

**declare** analz\_subset\_parts [THEN subsetD, dest]

**declare** image\_eq\_UN [simp]

**declare** parts\_insert2 [simp]

**declare** analz\_cut [dest]

**declare** split\_if\_asm [split]

**declare** analz\_insertI [intro]

**declare** Un\_Diff [simp]

### 27.3.2 extract the agent number of an Agent message

**consts** agt\_nb :: "msg => agent"

**recdef** agt\_nb "measure size"

"agt\_nb (Agent A) = A"

### 27.3.3 messages that are pairs

**constdefs** is\_MPair :: "msg => bool"

"is\_MPair X == EX Y Z. X = {|Y,Z|}"

**declare** is\_MPair\_def [simp]

**lemma** MPair\_is\_MPair [iff]: "is\_MPair {|X,Y|}"

*<proof>*

**lemma** Agent\_isnt\_MPair [iff]: "~ is\_MPair (Agent A)"

*<proof>*

**lemma** Number\_isnt\_MPair [iff]: "~ is\_MPair (Number n)"

*<proof>*

**lemma** *Key\_isnt\_MPair* [iff]: " $\sim$  is\_MPair (Key K)"  
 <proof>

**lemma** *Nonce\_isnt\_MPair* [iff]: " $\sim$  is\_MPair (Nonce n)"  
 <proof>

**lemma** *Hash\_isnt\_MPair* [iff]: " $\sim$  is\_MPair (Hash X)"  
 <proof>

**lemma** *Crypt\_isnt\_MPair* [iff]: " $\sim$  is\_MPair (Crypt K X)"  
 <proof>

**abbreviation**

*not\_MPair* :: "msg => bool" where  
 "not\_MPair X ==  $\sim$  is\_MPair X"

**lemma** *is\_MPairE*: "[| is\_MPair X ==> P; not\_MPair X ==> P |] ==> P"  
 <proof>

**declare** *is\_MPair\_def* [simp del]

**constdefs** *has\_no\_pair* :: "msg set => bool"  
 "has\_no\_pair H == ALL X Y. {X,Y}  $\sim$ :H"

**declare** *has\_no\_pair\_def* [simp]

### 27.3.4 well-foundedness of messages

**lemma** *wf\_Crypt1* [iff]: "Crypt K X  $\sim$ = X"  
 <proof>

**lemma** *wf\_Crypt2* [iff]: "X  $\sim$ = Crypt K X"  
 <proof>

**lemma** *parts\_size*: "X:parts {Y} ==> X=Y | size X < size Y"  
 <proof>

**lemma** *wf\_Crypt\_parts* [iff]: "Crypt K X  $\sim$ :parts {X}"  
 <proof>

### 27.3.5 lemmas on keysFor

**constdefs** *usekeys* :: "msg set => key set"  
 "usekeys G == {K. EX Y. Crypt K Y:G}"

**lemma** *finite\_keysFor* [intro]: "finite G ==> finite (keysFor G)"  
 <proof>

### 27.3.6 lemmas on parts

**lemma** *parts\_sub*: "[| X:parts G; G<=H |] ==> X:parts H"  
 <proof>

**lemma** *parts\_Diff* [dest]: "X:parts (G - H) ==> X:parts G"



*<proof>*

**lemma** parts\_Diff\_notin: "[| Y ~:H; Nonce n ~:parts (H - {Y}) |]  
==> Nonce n ~:parts H"

*<proof>*

**lemmas** parts\_insert\_substI = parts\_insert [THEN ssubst]  
**lemmas** parts\_insert\_substD = parts\_insert [THEN sym, THEN ssubst]

**lemma** finite\_parts\_msg [iff]: "finite (parts {X})"

*<proof>*

**lemma** finite\_parts [intro]: "finite H ==> finite (parts H)"

*<proof>*

**lemma** parts\_parts: "[| X:parts {Y}; Y:parts G |] ==> X:parts G"

*<proof>*

**lemma** parts\_parts\_parts: "[| X:parts {Y}; Y:parts {Z}; Z:parts G |] ==> X:parts G"

*<proof>*

**lemma** parts\_parts\_Crypt: "[| Crypt K X:parts G; Nonce n:parts {X} |]  
==> Nonce n:parts G"

*<proof>*

### 27.3.7 lemmas on synth

**lemma** synth\_sub: "[| X:synth G; G<=H |] ==> X:synth H"

*<proof>*

**lemma** Crypt\_synth [rule\_format]: "[| X:synth G; Key K ~:G |] ==>  
Crypt K Y:parts {X} --> Crypt K Y:parts G"

*<proof>*

### 27.3.8 lemmas on analz

**lemma** analz\_UnI1 [intro]: "X:analz G ==> X:analz (G Un H)"

*<proof>*

**lemma** analz\_sub: "[| X:analz G; G <= H |] ==> X:analz H"

*<proof>*

**lemma** analz\_Diff [dest]: "X:analz (G - H) ==> X:analz G"

*<proof>*

**lemmas** in\_analz\_subset\_cong = analz\_subset\_cong [THEN subsetD]

**lemma** analz\_eq: "A=A' ==> analz A = analz A'"

*<proof>*

**lemmas** insert\_commute\_substI = insert\_commute [THEN ssubst]

**lemma** analz\_insertD:

```
"[| Crypt K Y:H; Key (invKey K):H |] ==> analz (insert Y H) = analz H"
⟨proof⟩
```

```
lemma must_decrypt [rule_format,dest]: "[| X:analz H; has_no_pair H |] ==>
X ~:H --> (EX K Y. Crypt K Y:H & Key (invKey K):H)"
⟨proof⟩
```

```
lemma analz_needs_only_finite: "X:analz H ==> EX G. G <= H & finite G"
⟨proof⟩
```

```
lemma notin_analz_insert: "X ~:analz (insert Y G) ==> X ~:analz G"
⟨proof⟩
```

### 27.3.9 lemmas on parts, synth and analz

```
lemma parts_invKey [rule_format,dest]: "X:parts {Y} ==>
X:analz (insert (Crypt K Y) H) --> X ~:analz H --> Key (invKey K):analz H"
⟨proof⟩
```

```
lemma in_analz: "Y:analz H ==> EX X. X:H & Y:parts {X}"
⟨proof⟩
```

```
lemmas in_analz_subset_parts = analz_subset_parts [THEN subsetD]
```

```
lemma Crypt_synth_insert: "[| Crypt K X:parts (insert Y H);
Y:synth (analz H); Key K ~:analz H |] ==> Crypt K X:parts H"
⟨proof⟩
```

### 27.3.10 greatest nonce used in a message

```
consts greatest_msg :: "msg => nat"
```

```
recdef greatest_msg "measure size"
"greatest_msg (Nonce n) = n"
"greatest_msg {|X,Y|} = max (greatest_msg X) (greatest_msg Y)"
"greatest_msg (Crypt K X) = greatest_msg X"
"greatest_msg other = 0"
```

```
lemma greatest_msg_is_greatest: "Nonce n:parts {X} ==> n <= greatest_msg
X"
⟨proof⟩
```

### 27.3.11 sets of keys

```
constdefs keyset :: "msg set => bool"
"keyset G == ALL X. X:G --> (EX K. X = Key K)"
```

```
lemma keyset_in [dest]: "[| keyset G; X:G |] ==> EX K. X = Key K"
⟨proof⟩
```

```
lemma MPair_notin_keyset [simp]: "keyset G ==> {|X,Y|} ~:G"
⟨proof⟩
```

```
lemma Crypt_notin_keyset [simp]: "keyset G ==> Crypt K X ~:G"
⟨proof⟩
```

```
lemma Nonce_notin_keyset [simp]: "keyset G ==> Nonce n ~:G"
<proof>
```

```
lemma parts_keyset [simp]: "keyset G ==> parts G = G"
<proof>
```

### 27.3.12 keys a priori necessary for decrypting the messages of G

```
constdefs keysfor :: "msg set => msg set"
"keysfor G == Key ` keysFor (parts G)"
```

```
lemma keyset_keysfor [iff]: "keyset (keysfor G)"
<proof>
```

```
lemma keyset_Diff_keysfor [simp]: "keyset H ==> keyset (H - keysfor G)"
<proof>
```

```
lemma keysfor_Crypt: "Crypt K X:parts G ==> Key (invKey K):keysfor G"
<proof>
```

```
lemma no_key_no_Crypt: "Key K ~:keysfor G ==> Crypt (invKey K) X ~:parts G"
<proof>
```

```
lemma finite_keysfor [intro]: "finite G ==> finite (keysfor G)"
<proof>
```

### 27.3.13 only the keys necessary for G are useful in analz

```
lemma analz_keyset: "keyset H ==>
analz (G Un H) = H - keysfor G Un (analz (G Un (H Int keysfor G)))"
<proof>
```

```
lemmas analz_keyset_substD = analz_keyset [THEN sym, THEN ssubst]
```

## 27.4 Extensions to Theory Event

### 27.4.1 general protocol properties

```
constdefs is_Says :: "event => bool"
"is_Says ev == (EX A B X. ev = Says A B X)"
```

```
lemma is_Says_Says [iff]: "is_Says (Says A B X)"
<proof>
```

```
constdefs Gets_correct :: "event list set => bool"
"Gets_correct p == ALL evs B X. evs:p --> Gets B X:set evs
--> (EX A. Says A B X:set evs)"
```

```
lemma Gets_correct_Says: "[| Gets_correct p; Gets B X # evs:p |]
==> EX A. Says A B X:set evs"
<proof>
```

```
constdefs one_step :: "event list set => bool"
"one_step p == ALL evs ev. ev#evs:p --> evs:p"
```

```
lemma one_step_Cons [dest]: "[| one_step p; ev#evs:p |] ==> evs:p"
<proof>
```

```
lemma one_step_app: "[| evs@evs':p; one_step p; []:p |] ==> evs':p"
<proof>
```

```
lemma trunc: "[| evs @ evs':p; one_step p |] ==> evs':p"
<proof>
```

```
constdefs has_only_Says :: "event list set => bool"
"has_only_Says p == ALL evs ev. evs:p --> ev:set evs
--> (EX A B X. ev = Says A B X)"
```

```
lemma has_only_SaysD: "[| ev:set evs; evs:p; has_only_Says p |]
==> EX A B X. ev = Says A B X"
<proof>
```

```
lemma in_has_only_Says [dest]: "[| has_only_Says p; evs:p; ev:set evs |]
==> EX A B X. ev = Says A B X"
<proof>
```

```
lemma has_only_Says_imp_Gets_correct [simp]: "has_only_Says p
==> Gets_correct p"
<proof>
```

#### 27.4.2 lemma on knows

```
lemma Says_imp_spies2: "Says A B {|X,Y|}:set evs ==> Y:parts (spies evs)"
<proof>
```

```
lemma Says_not_parts: "[| Says A B X:set evs; Y ~:parts (spies evs) |]
==> Y ~:parts {X}"
<proof>
```

#### 27.4.3 knows without initState

```
consts knows' :: "agent => event list => msg set"
```

```
primrec
```

```
knows'_Nil:
"knows' A [] = {}"
```

```
knows'_Cons0:
"knows' A (ev # evs) = (
  if A = Spy then (
    case ev of
      Says A' B X => insert X (knows' A evs)
    | Gets A' X => knows' A evs
    | Notes A' X => if A':bad then insert X (knows' A evs) else knows' A evs
  ) else (
    case ev of
      Says A' B X => if A=A' then insert X (knows' A evs) else knows' A evs
```

```

    | Gets A' X => if A=A' then insert X (knows' A evs) else knows' A evs
    | Notes A' X => if A=A' then insert X (knows' A evs) else knows' A evs
  ))"

```

**abbreviation**

```

spies' :: "event list => msg set" where
  "spies' == knows' Spy"

```

**27.4.4 decomposition of knows into knows' and initState**

```

lemma knows_decomp: "knows A evs = knows' A evs Un (initState A)"
<proof>

```

```

lemmas knows_decomp_substI = knows_decomp [THEN ssubst]
lemmas knows_decomp_substD = knows_decomp [THEN sym, THEN ssubst]

```

```

lemma knows'_sub_knows: "knows' A evs <= knows A evs"
<proof>

```

```

lemma knows'_Cons: "knows' A (ev#evs) = knows' A [ev] Un knows' A evs"
<proof>

```

```

lemmas knows'_Cons_substI = knows'_Cons [THEN ssubst]
lemmas knows'_Cons_substD = knows'_Cons [THEN sym, THEN ssubst]

```

```

lemma knows_Cons: "knows A (ev#evs) = initState A Un knows' A [ev]
Un knows A evs"
<proof>

```

```

lemmas knows_Cons_substI = knows_Cons [THEN ssubst]
lemmas knows_Cons_substD = knows_Cons [THEN sym, THEN ssubst]

```

```

lemma knows'_sub_spies': "[| evs:p; has_only_Says p; one_step p |]
==> knows' A evs <= spies' evs"
<proof>

```

**27.4.5 knows' is finite**

```

lemma finite_knows' [iff]: "finite (knows' A evs)"
<proof>

```

**27.4.6 monotonicity of knows**

```

lemma knows_sub_Cons: "knows A evs <= knows A (ev#evs)"
<proof>

```

```

lemma knows_ConsI: "X:knows A evs ==> X:knows A (ev#evs)"
<proof>

```

```

lemma knows_sub_app: "knows A evs <= knows A (evs @ evs')"
<proof>

```

### 27.4.7 maximum knowledge an agent can have includes messages sent to the agent

```
consts knows_max' :: "agent => event list => msg set"
```

```
primrec
```

```
knows_max'_def_Nil: "knows_max' A [] = {}"
```

```
knows_max'_def_Cons: "knows_max' A (ev # evs) = (
```

```
  if A=Spy then (
```

```
    case ev of
```

```
      Says A' B X => insert X (knows_max' A evs)
```

```
    | Gets A' X => knows_max' A evs
```

```
    | Notes A' X =>
```

```
      if A':bad then insert X (knows_max' A evs) else knows_max' A evs
```

```
  ) else (
```

```
    case ev of
```

```
      Says A' B X =>
```

```
      if A=A' | A=B then insert X (knows_max' A evs) else knows_max' A evs
```

```
    | Gets A' X =>
```

```
      if A=A' then insert X (knows_max' A evs) else knows_max' A evs
```

```
    | Notes A' X =>
```

```
      if A=A' then insert X (knows_max' A evs) else knows_max' A evs
```

```
  ))"
```

```
constdefs knows_max :: "agent => event list => msg set"
```

```
"knows_max A evs == knows_max' A evs Un initState A"
```

```
abbreviation
```

```
spies_max :: "event list => msg set" where
```

```
"spies_max evs == knows_max Spy evs"
```

### 27.4.8 basic facts about knows\_max

```
lemma spies_max_spies [iff]: "spies_max evs = spies evs"
```

```
<proof>
```

```
lemma knows_max'_Cons: "knows_max' A (ev#evs)
```

```
= knows_max' A [ev] Un knows_max' A evs"
```

```
<proof>
```

```
lemmas knows_max'_Cons_substI = knows_max'_Cons [THEN ssubst]
```

```
lemmas knows_max'_Cons_substD = knows_max'_Cons [THEN sym, THEN ssubst]
```

```
lemma knows_max_Cons: "knows_max A (ev#evs)
```

```
= knows_max' A [ev] Un knows_max A evs"
```

```
<proof>
```

```
lemmas knows_max_Cons_substI = knows_max_Cons [THEN ssubst]
```

```
lemmas knows_max_Cons_substD = knows_max_Cons [THEN sym, THEN ssubst]
```

```
lemma finite_knows_max' [iff]: "finite (knows_max' A evs)"
```

```
<proof>
```

```
lemma knows_max'_sub_spies': "[| evs:p; has_only_Says p; one_step p |]
```

```
==> knows_max' A evs <= spies' evs"
```

*<proof>*

**lemma** knows\_max'\_in\_spies' [dest]: "[| evs:p; X:knows\_max' A evs;  
has\_only\_Says p; one\_step p |] ==> X:spies' evs"

*<proof>*

**lemma** knows\_max'\_app: "knows\_max' A (evs @ evs')  
= knows\_max' A evs Un knows\_max' A evs'"

*<proof>*

**lemma** Says\_to\_knows\_max': "Says A B X:set evs ==> X:knows\_max' B evs"

*<proof>*

**lemma** Says\_from\_knows\_max': "Says A B X:set evs ==> X:knows\_max' A evs"

*<proof>*

#### 27.4.9 used without initState

**consts** used' :: "event list => msg set"

**primrec**

"used' [] = {}"

"used' (ev # evs) = (  
  case ev of  
    Says A B X => parts {X} Un used' evs  
  | Gets A X => used' evs  
  | Notes A X => parts {X} Un used' evs  
)"

**constdefs** init :: "msg set"

"init == used []"

**lemma** used\_decomp: "used evs = init Un used' evs"

*<proof>*

**lemma** used'\_sub\_app: "used' evs <= used' (evs@evs')"

*<proof>*

**lemma** used'\_parts [rule\_format]: "X:used' evs ==> Y:parts {X} --> Y:used' evs"

*<proof>*

#### 27.4.10 monotonicity of used

**lemma** used\_sub\_Cons: "used evs <= used (ev#evs)"

*<proof>*

**lemma** used\_ConsI: "X:used evs ==> X:used (ev#evs)"

*<proof>*

**lemma** notin\_used\_ConsD: "X ~:used (ev#evs) ==> X ~:used evs"

*<proof>*

**lemma** used\_appD [dest]: "X:used (evs @ evs') ==> X:used evs | X:used evs'"

*<proof>*

**lemma** *used\_ConsD*: "X:used (ev#evs) ==> X:used [ev] | X:used evs"  
 <proof>

**lemma** *used\_sub\_app*: "used evs <= used (evs@evs')"  
 <proof>

**lemma** *used\_appIL*: "X:used evs ==> X:used (evs' @ evs)"  
 <proof>

**lemma** *used\_appIR*: "X:used evs ==> X:used (evs @ evs')"  
 <proof>

**lemma** *used\_parts*: "[| X:parts {Y}; Y:used evs |] ==> X:used evs"  
 <proof>

**lemma** *parts\_Says\_used*: "[| Says A B X:set evs; Y:parts {X} |] ==> Y:used evs"  
 <proof>

**lemma** *parts\_used\_app*: "X:parts {Y} ==> X:used (evs @ Says A B Y # evs')"  
 <proof>

#### 27.4.11 lemmas on used and knows

**lemma** *initState\_used*: "X:parts (initState A) ==> X:used evs"  
 <proof>

**lemma** *Says\_imp\_used*: "Says A B X:set evs ==> parts {X} <= used evs"  
 <proof>

**lemma** *not\_used\_not\_spied*: "X ~:used evs ==> X ~:parts (spies evs)"  
 <proof>

**lemma** *not\_used\_not\_parts*: "[| Y ~:used evs; Says A B X:set evs |]  
 ==> Y ~:parts {X}"  
 <proof>

**lemma** *not\_used\_parts\_false*: "[| X ~:used evs; Y:parts (spies evs) |]  
 ==> X ~:parts {Y}"  
 <proof>

**lemma** *known\_used [rule\_format]*: "[| evs:p; Gets\_correct p; one\_step p |]  
 ==> X:parts (knows A evs) --> X:used evs"  
 <proof>

**lemma** *known\_max\_used [rule\_format]*: "[| evs:p; Gets\_correct p; one\_step p  
 |]  
 ==> X:parts (knows\_max A evs) --> X:used evs"  
 <proof>

**lemma** *not\_used\_not\_known*: "[| evs:p; X ~:used evs;  
 Gets\_correct p; one\_step p |] ==> X ~:parts (knows A evs)"  
 <proof>



```

lemma not_used_not_known_max: "[| evs:p; X ~:used evs;
Gets_correct p; one_step p |] ==> X ~:parts (knows_max A evs)"
<proof>

```

#### 27.4.12 a nonce or key in a message cannot equal a fresh nonce or key

```

lemma Nonce_neq [dest]: "[| Nonce n' ~:used evs;
Says A B X:set evs; Nonce n:parts {X} |] ==> n ~= n'"
<proof>

```

```

lemma Key_neq [dest]: "[| Key n' ~:used evs;
Says A B X:set evs; Key n:parts {X} |] ==> n ~= n'"
<proof>

```

#### 27.4.13 message of an event

```

consts msg :: "event => msg"

```

```

recdef msg "measure size"
"msg (Says A B X) = X"
"msg (Gets A X) = X"
"msg (Notes A X) = X"

```

```

lemma used_sub_parts_used: "X:used (ev # evs) ==> X:parts {msg ev} Un used
evs"
<proof>

```

```

end

```

## 28 Decomposition of Analz into two parts

```

theory Analz imports Extensions begin

```

decomposition of analz into two parts: *pparts* (for pairs) and analz of *kparts*

### 28.1 messages that do not contribute to analz

```

inductive_set
  pparts :: "msg set => msg set"
  for H :: "msg set"
where
  Inj [intro]: "[| X:H; is_MPair X |] ==> X:pparts H"
| Fst [dest]: "[| {|X,Y|}:pparts H; is_MPair X |] ==> X:pparts H"
| Snd [dest]: "[| {|X,Y|}:pparts H; is_MPair Y |] ==> Y:pparts H"

```

### 28.2 basic facts about *pparts*

```

lemma pparts_is_MPair [dest]: "X:pparts H ==> is_MPair X"
<proof>

```

**lemma** *Crypt\_notin\_pparts* [iff]: "Crypt K X ~:pparts H"  
 <proof>

**lemma** *Key\_notin\_pparts* [iff]: "Key K ~:pparts H"  
 <proof>

**lemma** *Nonce\_notin\_pparts* [iff]: "Nonce n ~:pparts H"  
 <proof>

**lemma** *Number\_notin\_pparts* [iff]: "Number n ~:pparts H"  
 <proof>

**lemma** *Agent\_notin\_pparts* [iff]: "Agent A ~:pparts H"  
 <proof>

**lemma** *pparts\_empty* [iff]: "pparts {} = {}"  
 <proof>

**lemma** *pparts\_insertI* [intro]: "X:pparts H ==> X:pparts (insert Y H)"  
 <proof>

**lemma** *pparts\_sub*: "[| X:pparts G; G<=H |] ==> X:pparts H"  
 <proof>

**lemma** *pparts\_insert2* [iff]: "pparts (insert X (insert Y H))  
 = pparts {X} Un pparts {Y} Un pparts H"  
 <proof>

**lemma** *pparts\_insert\_MPair* [iff]: "pparts (insert {|X,Y|} H)  
 = insert {|X,Y|} (pparts ({X,Y} Un H))"  
 <proof>

**lemma** *pparts\_insert\_Nonce* [iff]: "pparts (insert (Nonce n) H) = pparts H"  
 <proof>

**lemma** *pparts\_insert\_Crypt* [iff]: "pparts (insert (Crypt K X) H) = pparts H"  
 <proof>

**lemma** *pparts\_insert\_Key* [iff]: "pparts (insert (Key K) H) = pparts H"  
 <proof>

**lemma** *pparts\_insert\_Agent* [iff]: "pparts (insert (Agent A) H) = pparts H"  
 <proof>

**lemma** *pparts\_insert\_Number* [iff]: "pparts (insert (Number n) H) = pparts H"  
 <proof>

**lemma** *pparts\_insert\_Hash* [iff]: "pparts (insert (Hash X) H) = pparts H"  
 <proof>

**lemma** *pparts\_insert*: "X:pparts (insert Y H) ==> X:pparts {Y} Un pparts H"

*<proof>*

**lemma** insert\_pparts: "X:pparts {Y} Un pparts H ==> X:pparts (insert Y H)"  
*<proof>*

**lemma** pparts\_Un [iff]: "pparts (G Un H) = pparts G Un pparts H"  
*<proof>*

**lemma** pparts\_pparts [iff]: "pparts (pparts H) = pparts H"  
*<proof>*

**lemma** pparts\_insert\_eq: "pparts (insert X H) = pparts {X} Un pparts H"  
*<proof>*

**lemmas** pparts\_insert\_substI = pparts\_insert\_eq [THEN ssubst]

**lemma** in\_pparts: "Y:pparts H ==> EX X. X:H & Y:pparts {X}"  
*<proof>*

### 28.3 facts about pparts and parts

**lemma** pparts\_no\_Nonce [dest]: "[| X:pparts {Y}; Nonce n ~:parts {Y} |]  
 ==> Nonce n ~:parts {X}"  
*<proof>*

### 28.4 facts about pparts and analz

**lemma** pparts\_analz: "X:pparts H ==> X:analz H"  
*<proof>*

**lemma** pparts\_analz\_sub: "[| X:pparts G; G<=H |] ==> X:analz H"  
*<proof>*

### 28.5 messages that contribute to analz

**inductive\_set**

  kparts :: "msg set => msg set"

  for H :: "msg set"

**where**

  Inj [intro]: "[| X:H; not\_MPair X |] ==> X:kparts H"

| Fst [intro]: "[| {|X,Y|}:pparts H; not\_MPair X |] ==> X:kparts H"

| Snd [intro]: "[| {|X,Y|}:pparts H; not\_MPair Y |] ==> Y:kparts H"

### 28.6 basic facts about kparts

**lemma** kparts\_not\_MPair [dest]: "X:kparts H ==> not\_MPair X"  
*<proof>*

**lemma** kparts\_empty [iff]: "kparts {} = {}"  
*<proof>*

**lemma** kparts\_insertI [intro]: "X:kparts H ==> X:kparts (insert Y H)"  
*<proof>*

**lemma** *kparts\_insert2* [iff]: "kparts (insert X (insert Y H))  
= kparts {X} Un kparts {Y} Un kparts H"  
<proof>

**lemma** *kparts\_insert\_MPair* [iff]: "kparts (insert {|X,Y|} H)  
= kparts ({X,Y} Un H)"  
<proof>

**lemma** *kparts\_insert\_Nonce* [iff]: "kparts (insert (Nonce n) H)  
= insert (Nonce n) (kparts H)"  
<proof>

**lemma** *kparts\_insert\_Crypt* [iff]: "kparts (insert (Crypt K X) H)  
= insert (Crypt K X) (kparts H)"  
<proof>

**lemma** *kparts\_insert\_Key* [iff]: "kparts (insert (Key K) H)  
= insert (Key K) (kparts H)"  
<proof>

**lemma** *kparts\_insert\_Agent* [iff]: "kparts (insert (Agent A) H)  
= insert (Agent A) (kparts H)"  
<proof>

**lemma** *kparts\_insert\_Number* [iff]: "kparts (insert (Number n) H)  
= insert (Number n) (kparts H)"  
<proof>

**lemma** *kparts\_insert\_Hash* [iff]: "kparts (insert (Hash X) H)  
= insert (Hash X) (kparts H)"  
<proof>

**lemma** *kparts\_insert*: "X:kparts (insert X H) ==> X:kparts {X} Un kparts H"  
<proof>

**lemma** *kparts\_insert\_fst* [rule\_format,dest]: "X:kparts (insert Z H) ==>  
X ~:kparts H --> X:kparts {Z}"  
<proof>

**lemma** *kparts\_sub*: "[| X:kparts G; G<=H |] ==> X:kparts H"  
<proof>

**lemma** *kparts\_Un* [iff]: "kparts (G Un H) = kparts G Un kparts H"  
<proof>

**lemma** *pparts\_kparts* [iff]: "pparts (kparts H) = {}"  
<proof>

**lemma** *kparts\_kparts* [iff]: "kparts (kparts H) = kparts H"  
<proof>

**lemma** *kparts\_insert\_eq*: "kparts (insert X H) = kparts {X} Un kparts H"  
<proof>

**lemmas** kparts\_insert\_substI = kparts\_insert\_eq [THEN ssubst]

**lemma** in\_kparts: "Y:kparts H ==> EX X. X:H & Y:kparts {X}"  
 <proof>

**lemma** kparts\_has\_no\_pair [iff]: "has\_no\_pair (kparts H)"  
 <proof>

## 28.7 facts about kparts and parts

**lemma** kparts\_no\_Nonce [dest]: "[| X:kparts {Y}; Nonce n ~:parts {Y} |]  
 ==> Nonce n ~:parts {X}"  
 <proof>

**lemma** kparts\_parts: "X:kparts H ==> X:parts H"  
 <proof>

**lemma** parts\_kparts: "X:parts (kparts H) ==> X:parts H"  
 <proof>

**lemma** Crypt\_kparts\_Nonce\_parts [dest]: "[| Crypt K Y:kparts {Z};  
 Nonce n:parts {Y} |] ==> Nonce n:parts {Z}"  
 <proof>

## 28.8 facts about kparts and analz

**lemma** kparts\_analz: "X:kparts H ==> X:analz H"  
 <proof>

**lemma** kparts\_analz\_sub: "[| X:kparts G; G<=H |] ==> X:analz H"  
 <proof>

**lemma** analz\_kparts [rule\_format,dest]: "X:analz H ==>  
 Y:kparts {X} --> Y:analz H"  
 <proof>

**lemma** analz\_kparts\_analz: "X:analz (kparts H) ==> X:analz H"  
 <proof>

**lemma** analz\_kparts\_insert: "X:analz (kparts (insert Z H)) ==>  
 X:analz (kparts {Z} Un kparts H)"  
 <proof>

**lemma** Nonce\_kparts\_synth [rule\_format]: "Y:synth (analz G)  
 ==> Nonce n:kparts {Y} --> Nonce n:analz G"  
 <proof>

**lemma** kparts\_insert\_synth: "[| Y:parts (insert X G); X:synth (analz G);  
 Nonce n:kparts {Y}; Nonce n ~:analz G |] ==> Y:parts G"  
 <proof>

**lemma** Crypt\_insert\_synth: "[| Crypt K Y:parts (insert X G); X:synth (analz  
 G);  
 Nonce n:kparts {Y}; Nonce n ~:analz G |] ==> Crypt K Y:parts G"

*<proof>*

## 28.9 analz is pparts + analz of kparts

**lemma** *analz\_pparts\_kparts*: "X:analz H ==> X:pparts H | X:analz (kparts H)"  
*<proof>*

**lemma** *analz\_pparts\_kparts\_eq*: "analz H = pparts H Un analz (kparts H)"  
*<proof>*

**lemmas** *analz\_pparts\_kparts\_substI* = *analz\_pparts\_kparts\_eq* [THEN *ssubst*]  
**lemmas** *analz\_pparts\_kparts\_substD*  
 = *analz\_pparts\_kparts\_eq* [THEN *sym*, THEN *ssubst*]

end

## 29 Protocol-Independent Confidentiality Theorem on Nonces

**theory** *Guard* imports *Analz Extensions* begin

**inductive\_set**

*guard* :: "nat => key set => msg set"  
 for *n* :: nat and *Ks* :: "key set"

**where**

*No\_Nonce* [intro]: "Nonce *n* ~:parts {X} ==> X:guard *n* *Ks*"  
 | *Guard\_Nonce* [intro]: "invKey *K*:*Ks* ==> Crypt *K* X:guard *n* *Ks*"  
 | *Crypt* [intro]: "X:guard *n* *Ks* ==> Crypt *K* X:guard *n* *Ks*"  
 | *Pair* [intro]: "[| X:guard *n* *Ks*; Y:guard *n* *Ks* |] ==> {|X,Y|}:guard *n* *Ks*"

### 29.1 basic facts about *guard*

**lemma** *Key\_is\_guard* [iff]: "Key *K*:guard *n* *Ks*"  
*<proof>*

**lemma** *Agent\_is\_guard* [iff]: "Agent *A*:guard *n* *Ks*"  
*<proof>*

**lemma** *Number\_is\_guard* [iff]: "Number *r*:guard *n* *Ks*"  
*<proof>*

**lemma** *Nonce\_notin\_guard*: "X:guard *n* *Ks* ==> X ~= Nonce *n*"  
*<proof>*

**lemma** *Nonce\_notin\_guard\_iff* [iff]: "Nonce *n* ~:guard *n* *Ks*"  
*<proof>*

**lemma** *guard\_has\_Crypt* [rule\_format]: "X:guard *n* *Ks* ==> Nonce *n*:parts {X}  
 --> (EX *K* *Y*. Crypt *K* *Y*:kparts {X} & Nonce *n*:parts {Y})"  
*<proof>*

**lemma** *Nonce\_notin\_kparts\_msg*: "X:guard n Ks ==> Nonce n ~:kparts {X}"  
 <proof>

**lemma** *Nonce\_in\_kparts\_imp\_no\_guard*: "Nonce n:kparts H  
 ==> EX X. X:H & X ~:guard n Ks"  
 <proof>

**lemma** *guard\_kparts [rule\_format]*: "X:guard n Ks ==>  
 Y:kparts {X} --> Y:guard n Ks"  
 <proof>

**lemma** *guard\_Crypt*: "[| Crypt K Y:guard n Ks; K ~:invKey Ks |] ==> Y:guard  
 n Ks"  
 <proof>

**lemma** *guard\_MPair [iff]*: "({|X,Y|}:guard n Ks) = (X:guard n Ks & Y:guard  
 n Ks)"  
 <proof>

**lemma** *guard\_not\_guard [rule\_format]*: "X:guard n Ks ==>  
 Crypt K Y:kparts {X} --> Nonce n:kparts {Y} --> Y ~:guard n Ks"  
 <proof>

**lemma** *guard\_extand*: "[| X:guard n Ks; Ks <= Ks' |] ==> X:guard n Ks'"  
 <proof>

## 29.2 guarded sets

**constdefs** *Guard* :: "nat => key set => msg set => bool"  
 "Guard n Ks H == ALL X. X:H --> X:guard n Ks"

## 29.3 basic facts about *Guard*

**lemma** *Guard\_empty [iff]*: "Guard n Ks {}"  
 <proof>

**lemma** *notin\_parts\_Guard [intro]*: "Nonce n ~:parts G ==> Guard n Ks G"  
 <proof>

**lemma** *Nonce\_notin\_kparts [simplified]*: "Guard n Ks H ==> Nonce n ~:kparts  
 H"  
 <proof>

**lemma** *Guard\_must\_decrypt*: "[| Guard n Ks H; Nonce n:analz H |] ==>  
 EX K Y. Crypt K Y:kparts H & Key (invKey K):kparts H"  
 <proof>

**lemma** *Guard\_kparts [intro]*: "Guard n Ks H ==> Guard n Ks (kparts H)"  
 <proof>

**lemma** *Guard\_mono*: "[| Guard n Ks H; G <= H |] ==> Guard n Ks G"  
 <proof>

**lemma** Guard\_insert [iff]: "Guard n Ks (insert X H)  
= (Guard n Ks H & X:guard n Ks)"  
⟨proof⟩

**lemma** Guard\_Un [iff]: "Guard n Ks (G Un H) = (Guard n Ks G & Guard n Ks H)"  
⟨proof⟩

**lemma** Guard\_synth [intro]: "Guard n Ks G ==> Guard n Ks (synth G)"  
⟨proof⟩

**lemma** Guard\_analz [intro]: "[| Guard n Ks G; ALL K. K:Ks --> Key K ~:analz  
G |]  
==> Guard n Ks (analz G)"  
⟨proof⟩

**lemma** in\_Guard [dest]: "[| X:G; Guard n Ks G |] ==> X:guard n Ks"  
⟨proof⟩

**lemma** in\_synth\_Guard: "[| X:synth G; Guard n Ks G |] ==> X:guard n Ks"  
⟨proof⟩

**lemma** in\_analz\_Guard: "[| X:analz G; Guard n Ks G;  
ALL K. K:Ks --> Key K ~:analz G |] ==> X:guard n Ks"  
⟨proof⟩

**lemma** Guard\_keyset [simp]: "keyset G ==> Guard n Ks G"  
⟨proof⟩

**lemma** Guard\_Un\_keyset: "[| Guard n Ks G; keyset H |] ==> Guard n Ks (G Un  
H)"  
⟨proof⟩

**lemma** in\_Guard\_kparts: "[| X:G; Guard n Ks G; Y:kparts {X} |] ==> Y:guard  
n Ks"  
⟨proof⟩

**lemma** in\_Guard\_kparts\_neq: "[| X:G; Guard n Ks G; Nonce n':kparts {X} |]  
==> n ~ n'"  
⟨proof⟩

**lemma** in\_Guard\_kparts\_Crypt: "[| X:G; Guard n Ks G; is\_MPair X;  
Crypt K Y:kparts {X}; Nonce n:kparts {Y} |] ==> invKey K:Ks"  
⟨proof⟩

**lemma** Guard\_extand: "[| Guard n Ks G; Ks <= Ks' |] ==> Guard n Ks' G"  
⟨proof⟩

**lemma** guard\_invKey [rule\_format]: "[| X:guard n Ks; Nonce n:kparts {Y} |]  
==>  
Crypt K Y:kparts {Y} --> invKey K:Ks"  
⟨proof⟩

**lemma** Crypt\_guard\_invKey [rule\_format]: "[| Crypt K Y:guard n Ks;  
Nonce n:kparts {Y} |] ==> invKey K:Ks"



*<proof>*

## 29.4 set obtained by decrypting a message

**abbreviation** (input)

```
decrypt :: "msg set => key => msg => msg set" where
  "decrypt H K Y == insert Y (H - {Crypt K Y})"
```

```
lemma analz_decrypt: "[| Crypt K Y:H; Key (invKey K):H; Nonce n:analz H |]
==> Nonce n:analz (decrypt H K Y)"
```

*<proof>*

```
lemma parts_decrypt: "[| Crypt K Y:H; X:parts (decrypt H K Y) |] ==> X:parts
H"
```

*<proof>*

## 29.5 number of Crypt's in a message

```
consts crypt_nb :: "msg => nat"
```

```
recdef crypt_nb "measure size"
  "crypt_nb (Crypt K X) = Suc (crypt_nb X)"
  "crypt_nb {|X,Y|} = crypt_nb X + crypt_nb Y"
  "crypt_nb X = 0"
```

## 29.6 basic facts about crypt\_nb

```
lemma non_empty_crypt_msg: "Crypt K Y:parts {X} ==> crypt_nb X ≠ 0"
<proof>
```

## 29.7 number of Crypt's in a message list

```
consts cnb :: "msg list => nat"
```

```
recdef cnb "measure size"
  "cnb [] = 0"
  "cnb (X#l) = crypt_nb X + cnb l"
```

## 29.8 basic facts about cnb

```
lemma cnb_app [simp]: "cnb (l @ l') = cnb l + cnb l'"
<proof>
```

```
lemma mem_cnb_minus: "x mem l ==> cnb l = crypt_nb x + (cnb l - crypt_nb
x)"
<proof>
```

```
lemmas mem_cnb_minus_substI = mem_cnb_minus [THEN ssubst]
```

```
lemma cnb_minus [simp]: "x mem l ==> cnb (remove l x) = cnb l - crypt_nb
x"
<proof>
```

```
lemma parts_cnb: "Z:parts (set l) ==>
```

```
cnb l = (cnb l - crypt_nb Z) + crypt_nb Z"
⟨proof⟩
```

```
lemma non_empty_crypt: "Crypt K Y:parts (set l) ==> cnb l ≠ 0"
⟨proof⟩
```

## 29.9 list of kparts

```
lemma kparts_msg_set: "EX l. kparts {X} = set l & cnb l = crypt_nb X"
⟨proof⟩
```

```
lemma kparts_set: "EX l'. kparts (set l) = set l' & cnb l' = cnb l"
⟨proof⟩
```

## 29.10 list corresponding to "decrypt"

```
constdefs decrypt' :: "msg list => key => msg => msg list"
"decrypt' l K Y == Y # remove l (Crypt K Y)"
```

```
declare decrypt'_def [simp]
```

## 29.11 basic facts about decrypt'

```
lemma decrypt_minus: "decrypt (set l) K Y <= set (decrypt' l K Y)"
⟨proof⟩
```

## 29.12 if the analyse of a finite guarded set gives n then it must also gives one of the keys of Ks

```
lemma Guard_invKey_by_list [rule_format]: "ALL l. cnb l = p
--> Guard n Ks (set l) --> Nonce n:analz (set l)
--> (EX K. K:Ks & Key K:analz (set l))"
⟨proof⟩
```

```
lemma Guard_invKey_finite: "[| Nonce n:analz G; Guard n Ks G; finite G |]
==> EX K. K:Ks & Key K:analz G"
⟨proof⟩
```

```
lemma Guard_invKey: "[| Nonce n:analz G; Guard n Ks G |]
==> EX K. K:Ks & Key K:analz G"
⟨proof⟩
```

## 29.13 if the analyse of a finite guarded set and a (possibly infinite) set of keys gives n then it must also gives Ks

```
lemma Guard_invKey_keyset: "[| Nonce n:analz (G Un H); Guard n Ks G; finite
G;
keyset H |] ==> EX K. K:Ks & Key K:analz (G Un H)"
⟨proof⟩
```

```
end
```

```
theory Guard_Public imports Guard Public Extensions begin
```

## 29.14 Extensions to Theory Public

```
declare initState.simps [simp del]
```

### 29.14.1 signature

```
constdefs sign :: "agent => msg => msg"
"sign A X == {|Agent A, X, Crypt (priK A) (Hash X)|}"
```

```
lemma sign_inj [iff]: "(sign A X = sign A' X') = (A=A' & X=X'"
<proof>
```

### 29.14.2 agent associated to a key

```
constdefs agt :: "key => agent"
"agt K == @A. K = priK A | K = pubK A"
```

```
lemma agt_priK [simp]: "agt (priK A) = A"
<proof>
```

```
lemma agt_pubK [simp]: "agt (pubK A) = A"
<proof>
```

### 29.14.3 basic facts about initState

```
lemma no_Crypt_in_parts_init [simp]: "Crypt K X ~:parts (initState A)"
<proof>
```

```
lemma no_Crypt_in_analz_init [simp]: "Crypt K X ~:analz (initState A)"
<proof>
```

```
lemma no_priK_in_analz_init [simp]: "A ~:bad
==> Key (priK A) ~:analz (initState Spy)"
<proof>
```

```
lemma priK_notin_initState_Friend [simp]: "A ~= Friend C
==> Key (priK A) ~: parts (initState (Friend C))"
<proof>
```

```
lemma keyset_init [iff]: "keyset (initState A)"
<proof>
```

### 29.14.4 sets of private keys

```
constdefs priK_set :: "key set => bool"
"priK_set Ks == ALL K. K:Ks --> (EX A. K = priK A)"
```

```
lemma in_priK_set: "[| priK_set Ks; K:Ks |] ==> EX A. K = priK A"
<proof>
```

```
lemma priK_set1 [iff]: "priK_set {priK A}"
<proof>
```

```
lemma priK_set2 [iff]: "priK_set {priK A, priK B}"
  <proof>
```

#### 29.14.5 sets of good keys

```
constdefs good :: "key set => bool"
"good Ks == ALL K. K:Ks --> agt K ~:bad"
```

```
lemma in_good: "[| good Ks; K:Ks |] ==> agt K ~:bad"
  <proof>
```

```
lemma good1 [simp]: "A ~:bad ==> good {priK A}"
  <proof>
```

```
lemma good2 [simp]: "[| A ~:bad; B ~:bad |] ==> good {priK A, priK B}"
  <proof>
```

#### 29.14.6 greatest nonce used in a trace, 0 if there is no nonce

```
consts greatest :: "event list => nat"
```

```
recdef greatest "measure size"
"greatest [] = 0"
"greatest (ev # evs) = max (greatest_msg (msg ev)) (greatest evs)"
```

```
lemma greatest_is_greatest: "Nonce n:used evs ==> n <= greatest evs"
  <proof>
```

#### 29.14.7 function giving a new nonce

```
constdefs new :: "event list => nat"
"new evs == Suc (greatest evs)"
```

```
lemma new_isnt_used [iff]: "Nonce (new evs) ~:used evs"
  <proof>
```

### 29.15 Proofs About Guarded Messages

#### 29.15.1 small hack necessary because priK is defined as the inverse of pubK

```
lemma pubK_is_invKey_priK: "pubK A = invKey (priK A)"
  <proof>
```

```
lemmas pubK_is_invKey_priK_substI = pubK_is_invKey_priK [THEN ssubst]
```

```
lemmas invKey_invKey_substI = invKey [THEN ssubst]
```

```
lemma "Nonce n:parts {X} ==> Crypt (pubK A) X:guard n {priK A}"
  <proof>
```

#### 29.15.2 guardedness results

```
lemma sign_guard [intro]: "X:guard n Ks ==> sign A X:guard n Ks"
  <proof>
```

```

lemma Guard_init [iff]: "Guard n Ks (initState B)"
  <proof>

lemma Guard_knows_max': "Guard n Ks (knows_max' C evs)
  ==> Guard n Ks (knows_max C evs)"
  <proof>

lemma Nonce_not_used_Guard_spies [dest]: "Nonce n ~:used evs
  ==> Guard n Ks (spies evs)"
  <proof>

lemma Nonce_not_used_Guard [dest]: "[| evs:p; Nonce n ~:used evs;
  Gets_correct p; one_step p |] ==> Guard n Ks (knows (Friend C) evs)"
  <proof>

lemma Nonce_not_used_Guard_max [dest]: "[| evs:p; Nonce n ~:used evs;
  Gets_correct p; one_step p |] ==> Guard n Ks (knows_max (Friend C) evs)"
  <proof>

lemma Nonce_not_used_Guard_max' [dest]: "[| evs:p; Nonce n ~:used evs;
  Gets_correct p; one_step p |] ==> Guard n Ks (knows_max' (Friend C) evs)"
  <proof>

29.15.3 regular protocols

constdefs regular :: "event list set => bool"
  "regular p == ALL evs A. evs:p --> (Key (priK A):parts (spies evs)) = (A:bad)"

lemma priK_parts_iff_bad [simp]: "[| evs:p; regular p |] ==>
  (Key (priK A):parts (spies evs)) = (A:bad)"
  <proof>

lemma priK_analz_iff_bad [simp]: "[| evs:p; regular p |] ==>
  (Key (priK A):analz (spies evs)) = (A:bad)"
  <proof>

lemma Guard_Nonce_analz: "[| Guard n Ks (spies evs); evs:p;
  priK_set Ks; good Ks; regular p |] ==> Nonce n ~:analz (spies evs)"
  <proof>

end

```

## 30 Lists of Messages and Lists of Agents

**theory** List\_Msg imports Extensions **begin**

### 30.1 Implementation of Lists by Messages

#### 30.1.1 nil is represented by any message which is not a pair

**abbreviation** (input)

**cons** :: "msg => msg => msg" **where**

```
"cons x l == {|x,l|}"
```

### 30.1.2 induction principle

```
lemma lmsg_induct: "[| !!x. not_MPair x ==> P x; !!x l. P l ==> P (cons x
l) |]
==> P l"
<proof>
```

### 30.1.3 head

```
consts head :: "msg => msg"
```

```
recdef head "measure size"
"head (cons x l) = x"
```

### 30.1.4 tail

```
consts tail :: "msg => msg"
```

```
recdef tail "measure size"
"tail (cons x l) = l"
```

### 30.1.5 length

```
consts len :: "msg => nat"
```

```
recdef len "measure size"
"len (cons x l) = Suc (len l)"
"len other = 0"
```

```
lemma len_not_empty: "n < len l ==> EX x l'. l = cons x l'"
<proof>
```

### 30.1.6 membership

```
consts isin :: "msg * msg => bool"
```

```
recdef isin "measure (%(x,l). size l)"
"isin (x, cons y l) = (x=y | isin (x,l))"
"isin (x, other) = False"
```

### 30.1.7 delete an element

```
consts del :: "msg * msg => msg"
```

```
recdef del "measure (%(x,l). size l)"
"del (x, cons y l) = (if x=y then l else cons y (del (x,l)))"
"del (x, other) = other"
```

```
lemma notin_del [simp]: "~ isin (x,l) ==> del (x,l) = l"
<proof>
```

```
lemma isin_del [rule_format]: "isin (y, del (x,l)) --> isin (y,l)"
<proof>
```

**30.1.8 concatenation**

```

consts app :: "msg * msg => msg"

recdef app "measure (%(l,l'). size l)"
  "app (cons x l, l') = cons x (app (l,l'))"
  "app (other, l') = l'"

lemma isin_app [iff]: "isin (x, app(l,l')) = (isin (x,l) | isin (x,l'))"
  <proof>

```

**30.1.9 replacement**

```

consts repl :: "msg * nat * msg => msg"

recdef repl "measure (%(l,i,x'). i)"
  "repl (cons x l, Suc i, x') = cons x (repl (l,i,x'))"
  "repl (cons x l, 0, x') = cons x' l"
  "repl (other, i, M') = other"

```

**30.1.10 ith element**

```

consts ith :: "msg * nat => msg"

recdef ith "measure (%(l,i). i)"
  "ith (cons x l, Suc i) = ith (l,i)"
  "ith (cons x l, 0) = x"
  "ith (other, i) = other"

lemma ith_head: "0 < len l ==> ith (l,0) = head l"
  <proof>

```

**30.1.11 insertion**

```

consts ins :: "msg * nat * msg => msg"

recdef ins "measure (%(l,i,x). i)"
  "ins (cons x l, Suc i, y) = cons x (ins (l,i,y))"
  "ins (l, 0, y) = cons y l"

lemma ins_head [simp]: "ins (l,0,y) = cons y l"
  <proof>

```

**30.1.12 truncation**

```

consts trunc :: "msg * nat => msg"

recdef trunc "measure (%(l,i). i)"
  "trunc (l,0) = l"
  "trunc (cons x l, Suc i) = trunc (l,i)"

lemma trunc_zero [simp]: "trunc (l,0) = l"
  <proof>

```

## 30.2 Agent Lists

### 30.2.1 set of well-formed agent-list messages

abbreviation

```
nil :: msg where
  "nil == Number 0"
```

inductive\_set agl :: "msg set"

where

```
Nil[intro]: "nil:agl"
| Cons[intro]: "[| A:agent; I:agl |] ==> cons (Agent A) I :agl"
```

### 30.2.2 basic facts about agent lists

lemma del\_in\_agl [intro]: "I:agl ==> del (a,I):agl"  
 <proof>

lemma app\_in\_agl [intro]: "[| I:agl; J:agl |] ==> app (I,J):agl"  
 <proof>

lemma no\_Key\_in\_agl: "I:agl ==> Key K ~:parts {I}"  
 <proof>

lemma no\_Nonce\_in\_agl: "I:agl ==> Nonce n ~:parts {I}"  
 <proof>

lemma no\_Key\_in\_appdel: "[| I:agl; J:agl |] ==>  
 Key K ~:parts {app (J, del (Agent B, I))}"  
 <proof>

lemma no\_Nonce\_in\_appdel: "[| I:agl; J:agl |] ==>  
 Nonce n ~:parts {app (J, del (Agent B, I))}"  
 <proof>

lemma no\_Crypt\_in\_agl: "I:agl ==> Crypt K X ~:parts {I}"  
 <proof>

lemma no\_Crypt\_in\_appdel: "[| I:agl; J:agl |] ==>  
 Crypt K X ~:parts {app (J, del (Agent B, I))}"  
 <proof>

end

## 31 Protocol P1

theory P1 imports Guard\_Public List\_Msg begin

### 31.1 Protocol Definition

31.1.1 offer chaining: B chains his offer for A with the head offer of  
 L for sending it to C

constdefs chain :: "agent => nat => agent => msg => agent => msg"



```

"chain B ofr A L C ==
let m1= Crypt (pubK A) (Nonce ofr) in
let m2= Hash {|head L, Agent C|} in
sign B {|m1,m2|}"

```

```

declare Let_def [simp]

```

```

lemma chain_inj [iff]: "(chain B ofr A L C = chain B' ofr' A' L' C')
= (B=B' & ofr=ofr' & A=A' & head L = head L' & C=C')"
<proof>

```

```

lemma Nonce_in_chain [iff]: "Nonce ofr:parts {chain B ofr A L C}"
<proof>

```

### 31.1.2 agent whose key is used to sign an offer

```

consts shop :: "msg => msg"

```

```

recdef shop "measure size"
"shop {|B,X,Crypt K H|} = Agent (agt K)"

```

```

lemma shop_chain [simp]: "shop (chain B ofr A L C) = Agent B"
<proof>

```

### 31.1.3 nonce used in an offer

```

consts nonce :: "msg => msg"

```

```

recdef nonce "measure size"
"nonce {|B,{|Crypt K ofr,m2|},CryptH|} = ofr"

```

```

lemma nonce_chain [simp]: "nonce (chain B ofr A L C) = Nonce ofr"
<proof>

```

### 31.1.4 next shop

```

consts next_shop :: "msg => agent"

```

```

recdef next_shop "measure size"
"next_shop {|B,{|m1,Hash{|headL,Agent C|}|},CryptH|} = C"

```

```

lemma next_shop_chain [iff]: "next_shop (chain B ofr A L C) = C"
<proof>

```

### 31.1.5 anchor of the offer list

```

constdefs anchor :: "agent => nat => agent => msg"
"anchor A n B == chain A n A (cons nil nil) B"

```

```

lemma anchor_inj [iff]: "(anchor A n B = anchor A' n' B')
= (A=A' & n=n' & B=B')"
<proof>

```

```

lemma Nonce_in_anchor [iff]: "Nonce n:parts {anchor A n B}"
<proof>

```

**lemma** *shop\_anchor [simp]*: "shop (anchor A n B) = Agent A"  
 <proof>

**lemma** *nonce\_anchor [simp]*: "nonce (anchor A n B) = Nonce n"  
 <proof>

**lemma** *next\_shop\_anchor [iff]*: "next\_shop (anchor A n B) = B"  
 <proof>

### 31.1.6 request event

**constdefs** *reqm* :: "agent => nat => nat => msg => agent => msg"  
 "reqm A r n I B == {|Agent A, Number r, cons (Agent A) (cons (Agent B) I),  
 cons (anchor A n B) nil|}"

**lemma** *reqm\_inj [iff]*: "(reqm A r n I B = reqm A' r' n' I' B')  
 = (A=A' & r=r' & n=n' & I=I' & B=B')"  
 <proof>

**lemma** *Nonce\_in\_reqm [iff]*: "Nonce n:parts {reqm A r n I B}"  
 <proof>

**constdefs** *req* :: "agent => nat => nat => msg => agent => event"  
 "req A r n I B == Says A B (reqm A r n I B)"

**lemma** *req\_inj [iff]*: "(req A r n I B = req A' r' n' I' B')  
 = (A=A' & r=r' & n=n' & I=I' & B=B')"  
 <proof>

### 31.1.7 propose event

**constdefs** *prom* :: "agent => nat => agent => nat => msg => msg =>  
 msg => agent => msg"  
 "prom B ofr A r I L J C == {|Agent A, Number r,  
 app (J, del (Agent B, I)), cons (chain B ofr A L C) L|}"

**lemma** *prom\_inj [dest]*: "prom B ofr A r I L J C  
 = prom B' ofr' A' r' I' L' J' C'  
 ==> B=B' & ofr=ofr' & A=A' & r=r' & L=L' & C=C'"  
 <proof>

**lemma** *Nonce\_in\_prom [iff]*: "Nonce ofr:parts {prom B ofr A r I L J C}"  
 <proof>

**constdefs** *pro* :: "agent => nat => agent => nat => msg => msg =>  
 msg => agent => event"  
 "pro B ofr A r I L J C == Says B C (prom B ofr A r I L J C)"

**lemma** *pro\_inj [dest]*: "pro B ofr A r I L J C = pro B' ofr' A' r' I' L' J'  
 C'  
 ==> B=B' & ofr=ofr' & A=A' & r=r' & L=L' & C=C'"  
 <proof>

### 31.1.8 protocol

**inductive\_set** *p1* :: "event list set"

**where**

*Nil*: "[ ]:p1"

| *Fake*: "[| evsf:p1; X:synth (analz (spies evsf)) |] ==> Says Spy B X # evsf : p1"

| *Request*: "[| evsr:p1; Nonce n ~:used evsr; I:agl |] ==> req A r n I B # evsr : p1"

| *Propose*: "[| evsp:p1; Says A' B {|Agent A,Number r,I,cons M L|}:set evsp; I:agl; J:agl; isin (Agent C, app (J, del (Agent B, I))); Nonce ofr ~:used evsp |] ==> pro B ofr A r I (cons M L) J C # evsp : p1"

### 31.1.9 Composition of Traces

**lemma** "evs':p1 ==>

evs:p1 & (ALL n. Nonce n:used evs' --> Nonce n ~:used evs) -->

evs'@evs : p1"

*<proof>*

### 31.1.10 Valid Offer Lists

**inductive\_set**

*valid* :: "agent => nat => agent => msg set"

**for** *A* :: agent **and** *n* :: nat **and** *B* :: agent

**where**

*Request* [intro]: "cons (anchor A n B) nil:valid A n B"

| *Propose* [intro]: "L:valid A n B

==> cons (chain (next\_shop (head L)) ofr A L C) L:valid A n B"

### 31.1.11 basic properties of valid

**lemma** *valid\_not\_empty*: "L:valid A n B ==> EX M L'. L = cons M L'"

*<proof>*

**lemma** *valid\_pos\_len*: "L:valid A n B ==> 0 < len L"

*<proof>*

### 31.1.12 offers of an offer list

**constdefs** *offer\_nonces* :: "msg => msg set"

"offer\_nonces L == {X. X:parts {L} & (EX n. X = Nonce n)}"

### 31.1.13 the originator can get the offers

**lemma** "L:valid A n B ==> offer\_nonces L <= analz (insert L (initState A))"

*<proof>*

### 31.1.14 list of offers

**consts** *offers* :: "msg => msg"

```

recdef offers "measure size"
"offers (cons M L) = cons {|shop M, nonce M|} (offers L)"
"offers other = nil"

```

### 31.1.15 list of agents whose keys are used to sign a list of offers

```

consts shops :: "msg => msg"

recdef shops "measure size"
"shops (cons M L) = cons (shop M) (shops L)"
"shops other = other"

lemma shops_in_agl: "L:valid A n B ==> shops L:agl"
<proof>

```

### 31.1.16 builds a trace from an itinerary

```

consts offer_list :: "agent * nat * agent * msg * nat => msg"

recdef offer_list "measure (%(A,n,B,I,ofr). size I)"
"offer_list (A,n,B,nil,ofr) = cons (anchor A n B) nil"
"offer_list (A,n,B,cons (Agent C) I,ofr) = (
  let L = offer_list (A,n,B,I,Suc ofr) in
  cons (chain (next_shop (head L)) ofr A L C) L)"

lemma "I:agl ==> ALL ofr. offer_list (A,n,B,I,ofr):valid A n B"
<proof>

consts trace :: "agent * nat * agent * nat * msg * msg * msg
=> event list"

recdef trace "measure (%(B,ofr,A,r,I,L,K). size K)"
"trace (B,ofr,A,r,I,L,nil) = []"
"trace (B,ofr,A,r,I,L,cons (Agent D) K) = (
  let C = (if K=nil then B else agt_nb (head K)) in
  let I' = (if K=nil then cons (Agent A) (cons (Agent B) I)
    else cons (Agent A) (app (I, cons (head K) nil))) in
  let I'' = app (I, cons (head K) nil) in
  pro C (Suc ofr) A r I' L nil D
  # trace (B,Suc ofr,A,r,I'',tail L,K))"

constdefs trace' :: "agent => nat => nat => msg => agent => nat => event list"
"trace' A r n I B ofr == (
  let AI = cons (Agent A) I in
  let L = offer_list (A,n,B,AI,ofr) in
  trace (B,ofr,A,r,nil,L,AI))"

declare trace'_def [simp]

```

### 31.1.17 there is a trace in which the originator receives a valid answer

```

lemma p1_not_empty: "evs:p1 ==> req A r n I B:set evs -->
(EX evs'. evs'@evs:p1 & pro B' ofr A r I' L J A:set evs' & L:valid A n B)"

```

*<proof>*

## 31.2 properties of protocol P1

publicly verifiable forward integrity: anyone can verify the validity of an offer list

### 31.2.1 strong forward integrity: except the last one, no offer can be modified

**lemma** *strong\_forward\_integrity*: "ALL L. Suc i < len L  
 --> L:valid A n B & repl (L,Suc i,M):valid A n B --> M = ith (L,Suc i)"  
*<proof>*

### 31.2.2 insertion resilience: except at the beginning, no offer can be inserted

**lemma** *chain\_isnt\_head [simp]*: "L:valid A n B ==>  
 head L ~= chain (next\_shop (head L)) ofr A L C"  
*<proof>*

**lemma** *insertion\_resilience*: "ALL L. L:valid A n B --> Suc i < len L  
 --> ins (L,Suc i,M) ~:valid A n B"  
*<proof>*

### 31.2.3 truncation resilience: only shop i can truncate at offer i

**lemma** *truncation\_resilience*: "ALL L. L:valid A n B --> Suc i < len L  
 --> cons M (trunc (L,Suc i)):valid A n B --> shop M = shop (ith (L,i))"  
*<proof>*

### 31.2.4 declarations for tactics

**declare** *knows\_Spy\_partsEs [elim]*  
**declare** *Fake\_parts\_insert [THEN subsetD, dest]*  
**declare** *initState.simps [simp del]*

### 31.2.5 get components of a message

**lemma** *get\_ML [dest]*: "Says A' B {|A,r,I,M,L|}:set evs ==>  
 M:parts (spies evs) & L:parts (spies evs)"  
*<proof>*

### 31.2.6 general properties of p1

**lemma** *reqm\_neq\_prom [iff]*:  
 "reqm A r n I B ~= prom B' ofr A' r' I' (cons M L) J C"  
*<proof>*

**lemma** *prom\_neq\_reqm [iff]*:  
 "prom B' ofr A' r' I' (cons M L) J C ~= reqm A r n I B"  
*<proof>*

**lemma** *req\_neq\_pro [iff]*: "req A r n I B ~= pro B' ofr A' r' I' (cons M L)  
 J C"

*<proof>*

**lemma** *pro\_neq\_req [iff]: "pro B' ofr A' r' I' (cons M L) J C ~= req A r n I B"*

*<proof>*

**lemma** *p1\_has\_no\_Gets: "evs:p1 ==> ALL A X. Gets A X ~:set evs"*

*<proof>*

**lemma** *p1\_is\_Gets\_correct [iff]: "Gets\_correct p1"*

*<proof>*

**lemma** *p1\_is\_one\_step [iff]: "one\_step p1"*

*<proof>*

**lemma** *p1\_has\_only\_Says' [rule\_format]: "evs:p1 ==> ev:set evs --> (EX A B X. ev=Says A B X)"*

*<proof>*

**lemma** *p1\_has\_only\_Says [iff]: "has\_only\_Says p1"*

*<proof>*

**lemma** *p1\_is\_regular [iff]: "regular p1"*

*<proof>*

### 31.2.7 private keys are safe

**lemma** *priK\_parts\_Friend\_imp\_bad [rule\_format,dest]:*

*"[| evs:p1; Friend B ~= A |]*

*==> (Key (priK A):parts (knows (Friend B) evs)) --> (A:bad)"*

*<proof>*

**lemma** *priK\_analz\_Friend\_imp\_bad [rule\_format,dest]:*

*"[| evs:p1; Friend B ~= A |]*

*==> (Key (priK A):analz (knows (Friend B) evs)) --> (A:bad)"*

*<proof>*

**lemma** *priK\_notin\_knows\_max\_Friend: "[| evs:p1; A ~:bad; A ~= Friend C |]*

*==> Key (priK A) ~:analz (knows\_max (Friend C) evs)"*

*<proof>*

### 31.2.8 general guardedness properties

**lemma** *agl\_guard [intro]: "I:agl ==> I:guard n Ks"*

*<proof>*

**lemma** *Says\_to\_knows\_max'\_guard: "[| Says A' C {|A'',r,I,L|}:set evs;*

*Guard n Ks (knows\_max' C evs) |] ==> L:guard n Ks"*

*<proof>*

**lemma** *Says\_from\_knows\_max'\_guard: "[| Says C A' {|A'',r,I,L|}:set evs;*

*Guard n Ks (knows\_max' C evs) |] ==> L:guard n Ks"*

*<proof>*

**lemma** *Says\_Nonce\_not\_used\_guard: "[| Says A' B {|A'',r,I,L|}:set evs;*

Nonce n ~:used evs [] ==> L:guard n Ks"  
 <proof>

### 31.2.9 guardedness of messages

lemma chain\_guard [iff]: "chain B ofr A L C:guard n {priK A}"  
 <proof>

lemma chain\_guard\_Nonce\_neq [intro]: "n ~= ofr  
 ==> chain B ofr A' L C:guard n {priK A}"  
 <proof>

lemma anchor\_guard [iff]: "anchor A n' B:guard n {priK A}"  
 <proof>

lemma anchor\_guard\_Nonce\_neq [intro]: "n ~= n'  
 ==> anchor A' n' B:guard n {priK A}"  
 <proof>

lemma reqm\_guard [intro]: "I:agl ==> reqm A r n' I B:guard n {priK A}"  
 <proof>

lemma reqm\_guard\_Nonce\_neq [intro]: "[| n ~= n'; I:agl |]  
 ==> reqm A' r n' I B:guard n {priK A}"  
 <proof>

lemma prom\_guard [intro]: "[| I:agl; J:agl; L:guard n {priK A} |]  
 ==> prom B ofr A r I L J C:guard n {priK A}"  
 <proof>

lemma prom\_guard\_Nonce\_neq [intro]: "[| n ~= ofr; I:agl; J:agl;  
 L:guard n {priK A} |] ==> prom B ofr A' r I L J C:guard n {priK A}"  
 <proof>

### 31.2.10 Nonce uniqueness

lemma uniq\_Nonce\_in\_chain [dest]: "Nonce k:parts {chain B ofr A L C} ==>  
 k=ofr"  
 <proof>

lemma uniq\_Nonce\_in\_anchor [dest]: "Nonce k:parts {anchor A n B} ==> k=n"  
 <proof>

lemma uniq\_Nonce\_in\_reqm [dest]: "[| Nonce k:parts {reqm A r n I B};  
 I:agl |] ==> k=n"  
 <proof>

lemma uniq\_Nonce\_in\_prom [dest]: "[| Nonce k:parts {prom B ofr A r I L J  
 C};  
 I:agl; J:agl; Nonce k ~:parts {L} |] ==> k=ofr"  
 <proof>

### 31.2.11 requests are guarded

lemma req\_imp\_Guard [rule\_format]: "[| evs:p1; A ~:bad |] ==>

```
req A r n I B:set evs --> Guard n {priK A} (spies evs)"
⟨proof⟩
```

```
lemma req_imp_Guard_Friend: "[| evs:p1; A ~:bad; req A r n I B:set evs |]
==> Guard n {priK A} (knows_max (Friend C) evs)"
⟨proof⟩
```

### 31.2.12 propositions are guarded

```
lemma pro_imp_Guard [rule_format]: "[| evs:p1; B ~:bad; A ~:bad |] ==>
pro B ofr A r I (cons M L) J C:set evs --> Guard ofr {priK A} (spies evs)"
⟨proof⟩
```

```
lemma pro_imp_Guard_Friend: "[| evs:p1; B ~:bad; A ~:bad;
pro B ofr A r I (cons M L) J C:set evs |]
==> Guard ofr {priK A} (knows_max (Friend D) evs)"
⟨proof⟩
```

### 31.2.13 data confidentiality: no one other than the originator can decrypt the offers

```
lemma Nonce_req_notin_spies: "[| evs:p1; req A r n I B:set evs; A ~:bad |]
==> Nonce n ~:analz (spies evs)"
⟨proof⟩
```

```
lemma Nonce_req_notin_knows_max_Friend: "[| evs:p1; req A r n I B:set evs;
A ~:bad; A ~= Friend C |] ==> Nonce n ~:analz (knows_max (Friend C) evs)"
⟨proof⟩
```

```
lemma Nonce_pro_notin_spies: "[| evs:p1; B ~:bad; A ~:bad;
pro B ofr A r I (cons M L) J C:set evs |] ==> Nonce ofr ~:analz (spies evs)"
⟨proof⟩
```

```
lemma Nonce_pro_notin_knows_max_Friend: "[| evs:p1; B ~:bad; A ~:bad;
A ~= Friend D; pro B ofr A r I (cons M L) J C:set evs |]
==> Nonce ofr ~:analz (knows_max (Friend D) evs)"
⟨proof⟩
```

### 31.2.14 non repudiability: an offer signed by B has been sent by B

```
lemma Crypt_reqm: "[| Crypt (priK A) X:parts {reqm A' r n I B}; I:agl |]
==> A=A'"
⟨proof⟩
```

```
lemma Crypt_prom: "[| Crypt (priK A) X:parts {prom B ofr A' r I L J C};
I:agl; J:agl |] ==> A=B | Crypt (priK A) X:parts {L}"
⟨proof⟩
```

```
lemma Crypt_safeness: "[| evs:p1; A ~:bad |] ==> Crypt (priK A) X:parts (spies
evs)
--> (EX B Y. Says A B Y:set evs & Crypt (priK A) X:parts {Y})"
⟨proof⟩
```

```
lemma Crypt_Hash_imp_sign: "[| evs:p1; A ~:bad |] ==>
Crypt (priK A) (Hash X):parts (spies evs)"
```



```
--> (EX B Y. Says A B Y:set evs & sign A X:parts {Y})"
<proof>
```

```
lemma sign_safeness: "[| evs:p1; A ~:bad |] ==> sign A X:parts (spies evs)
--> (EX B Y. Says A B Y:set evs & sign A X:parts {Y})"
<proof>
```

```
end
```

## 32 Protocol P2

```
theory P2 imports Guard_Public List_Msg begin
```

### 32.1 Protocol Definition

Like P1 except the definitions of *chain*, *shop*, *next\_shop* and *nonce*

#### 32.1.1 offer chaining: B chains his offer for A with the head offer of L for sending it to C

```
constdefs chain :: "agent => nat => agent => msg => agent => msg"
"chain B ofr A L C ==
let m1= sign B (Nonce ofr) in
let m2= Hash {|head L, Agent C|} in
{|Crypt (pubK A) m1, m2|}"
```

```
declare Let_def [simp]
```

```
lemma chain_inj [iff]: "(chain B ofr A L C = chain B' ofr' A' L' C')
= (B=B' & ofr=ofr' & A=A' & head L = head L' & C=C')"
<proof>
```

```
lemma Nonce_in_chain [iff]: "Nonce ofr:parts {chain B ofr A L C}"
<proof>
```

#### 32.1.2 agent whose key is used to sign an offer

```
consts shop :: "msg => msg"
```

```
recdef shop "measure size"
"shop {|Crypt K {|B,ofr,Crypt K' H|},m2|} = Agent (agt K')"
```

```
lemma shop_chain [simp]: "shop (chain B ofr A L C) = Agent B"
<proof>
```

#### 32.1.3 nonce used in an offer

```
consts nonce :: "msg => msg"
```

```
recdef nonce "measure size"
"nonce {|Crypt K {|B,ofr,Crypt H|},m2|} = ofr"
```

```
lemma nonce_chain [simp]: "nonce (chain B ofr A L C) = Nonce ofr"
<proof>
```

### 32.1.4 next shop

```

consts next_shop :: "msg => agent"

recdef next_shop "measure size"
  "next_shop {|m1,Hash {|headL,Agent C|}|} = C"

lemma "next_shop (chain B ofr A L C) = C"
  <proof>

```

### 32.1.5 anchor of the offer list

```

constdefs anchor :: "agent => nat => agent => msg"
  "anchor A n B == chain A n A (cons nil nil) B"

lemma anchor_inj [iff]:
  "(anchor A n B = anchor A' n' B') = (A=A' & n=n' & B=B' )"
  <proof>

lemma Nonce_in_anchor [iff]: "Nonce n:parts {anchor A n B}"
  <proof>

lemma shop_anchor [simp]: "shop (anchor A n B) = Agent A"
  <proof>

```

### 32.1.6 request event

```

constdefs reqm :: "agent => nat => nat => msg => agent => msg"
  "reqm A r n I B == {|Agent A, Number r, cons (Agent A) (cons (Agent B) I),
  cons (anchor A n B) nil|}"

lemma reqm_inj [iff]: "(reqm A r n I B = reqm A' r' n' I' B')
  = (A=A' & r=r' & n=n' & I=I' & B=B' )"
  <proof>

lemma Nonce_in_reqm [iff]: "Nonce n:parts {reqm A r n I B}"
  <proof>

constdefs req :: "agent => nat => nat => msg => agent => event"
  "req A r n I B == Says A B (reqm A r n I B)"

lemma req_inj [iff]: "(req A r n I B = req A' r' n' I' B')
  = (A=A' & r=r' & n=n' & I=I' & B=B' )"
  <proof>

```

### 32.1.7 propose event

```

constdefs prom :: "agent => nat => agent => nat => msg => msg =>
  msg => agent => msg"
  "prom B ofr A r I L J C == {|Agent A, Number r,
  app (J, del (Agent B, I)), cons (chain B ofr A L C) L|}"

lemma prom_inj [dest]: "prom B ofr A r I L J C = prom B' ofr' A' r' I' L'
  J' C'
  ==> B=B' & ofr=ofr' & A=A' & r=r' & L=L' & C=C'"

```

*<proof>*

**lemma** *Nonce\_in\_prom* [iff]: "Nonce ofr:parts {prom B ofr A r I L J C}"

*<proof>*

**constdefs** *pro* :: "agent => nat => agent => nat => msg => msg =>  
                   msg => agent => event"  
*"pro B ofr A r I L J C == Says B C (prom B ofr A r I L J C)"*

**lemma** *pro\_inj* [dest]: "pro B ofr A r I L J C = pro B' ofr' A' r' I' L' J'  
 C'

==> B=B' & ofr=ofr' & A=A' & r=r' & L=L' & C=C'"

*<proof>*

### 32.1.8 protocol

**inductive\_set** *p2* :: "event list set"

**where**

*Nil*: "[ ]:p2"

| *Fake*: "[| evsf:p2; X:synth (analz (spies evsf)) |] ==> Says Spy B X # evsf  
 : p2"

| *Request*: "[| evsr:p2; Nonce n ~:used evsr; I:agl |] ==> req A r n I B # evsr  
 : p2"

| *Propose*: "[| evsp:p2; Says A' B {|Agent A,Number r,I,cons M L|}:set evsp;  
 I:agl; J:agl; isin (Agent C, app (J, del (Agent B, I)));  
 Nonce ofr ~:used evsp |] ==> pro B ofr A r I (cons M L) J C # evsp : p2"

### 32.1.9 valid offer lists

**inductive\_set**

*valid* :: "agent => nat => agent => msg set"

**for** *A* :: agent **and** *n* :: nat **and** *B* :: agent

**where**

*Request* [intro]: "cons (anchor A n B) nil:valid A n B"

| *Propose* [intro]: "L:valid A n B  
 ==> cons (chain (next\_shop (head L)) ofr A L C) L:valid A n B"

### 32.1.10 basic properties of valid

**lemma** *valid\_not\_empty*: "L:valid A n B ==> EX M L'. L = cons M L'"

*<proof>*

**lemma** *valid\_pos\_len*: "L:valid A n B ==> 0 < len L"

*<proof>*

### 32.1.11 list of offers

**consts** *offers* :: "msg => msg"

**recdef** *offers* "measure size"

```
"offers (cons M L) = cons {/shop M, nonce M/} (offers L)"
"offers other = nil"
```

### 32.2 Properties of Protocol P2

same as *P1\_Prop* except that publicly verifiable forward integrity is replaced by forward privacy

#### 32.3 strong forward integrity: except the last one, no offer can be modified

```
lemma strong_forward_integrity: "ALL L. Suc i < len L
--> L:valid A n B --> repl (L,Suc i,M):valid A n B --> M = ith (L,Suc i)"
<proof>
```

#### 32.4 insertion resilience: except at the beginning, no offer can be inserted

```
lemma chain_isnt_head [simp]: "L:valid A n B ==>
head L ~= chain (next_shop (head L)) ofr A L C"
<proof>
```

```
lemma insertion_resilience: "ALL L. L:valid A n B --> Suc i < len L
--> ins (L,Suc i,M) ~:valid A n B"
<proof>
```

#### 32.5 truncation resilience: only shop i can truncate at offer i

```
lemma truncation_resilience: "ALL L. L:valid A n B --> Suc i < len L
--> cons M (trunc (L,Suc i)):valid A n B --> shop M = shop (ith (L,i))"
<proof>
```

### 32.6 declarations for tactics

```
declare knows_Spy_partsEs [elim]
declare Fake_parts_insert [THEN subsetD, dest]
declare initState.simps [simp del]
```

### 32.7 get components of a message

```
lemma get_ML [dest]: "Says A' B {/A,R,I,M,L/}:set evs ==>
M:parts (spies evs) & L:parts (spies evs)"
<proof>
```

### 32.8 general properties of p2

```
lemma reqm_neq_prom [iff]:
"reqm A r n I B ~= prom B' ofr A' r' I' (cons M L) J C"
<proof>
```

```
lemma prom_neq_reqm [iff]:
"prom B' ofr A' r' I' (cons M L) J C ~= reqm A r n I B"
```

*<proof>*

**lemma** req\_neq\_pro [iff]: "req A r n I B ~ = pro B' ofr A' r' I' (cons M L) J C"

*<proof>*

**lemma** pro\_neq\_req [iff]: "pro B' ofr A' r' I' (cons M L) J C ~ = req A r n I B"

*<proof>*

**lemma** p2\_has\_no\_Gets: "evs:p2 ==> ALL A X. Gets A X ~:set evs"

*<proof>*

**lemma** p2\_is\_Gets\_correct [iff]: "Gets\_correct p2"

*<proof>*

**lemma** p2\_is\_one\_step [iff]: "one\_step p2"

*<proof>*

**lemma** p2\_has\_only\_Says' [rule\_format]: "evs:p2 ==> ev:set evs --> (EX A B X. ev=Says A B X)"

*<proof>*

**lemma** p2\_has\_only\_Says [iff]: "has\_only\_Says p2"

*<proof>*

**lemma** p2\_is\_regular [iff]: "regular p2"

*<proof>*

## 32.9 private keys are safe

**lemma** priK\_parts\_Friend\_imp\_bad [rule\_format,dest]:  
 "[| evs:p2; Friend B ~ = A |]  
 ==> (Key (priK A):parts (knows (Friend B) evs)) --> (A:bad)"

*<proof>*

**lemma** priK\_analz\_Friend\_imp\_bad [rule\_format,dest]:  
 "[| evs:p2; Friend B ~ = A |]  
 ==> (Key (priK A):analz (knows (Friend B) evs)) --> (A:bad)"

*<proof>*

**lemma** priK\_notin\_knows\_max\_Friend:  
 "[| evs:p2; A ~:bad; A ~ = Friend C |]  
 ==> Key (priK A) ~:analz (knows\_max (Friend C) evs)"

*<proof>*

## 32.10 general guardedness properties

**lemma** agl\_guard [intro]: "I:agl ==> I:guard n Ks"

*<proof>*

**lemma** Says\_to\_knows\_max'\_guard: "[| Says A' C {|A'',r,I,L|}:set evs;  
 Guard n Ks (knows\_max' C evs) |] ==> L:guard n Ks"

*<proof>*

**lemma** Says\_from\_knows\_max'\_guard: "[| Says C A' {|A'',r,I,L|}:set evs;  
Guard n Ks (knows\_max' C evs) |] ==> L:guard n Ks"  
<proof>

**lemma** Says\_Nonce\_not\_used\_guard: "[| Says A' B {|A'',r,I,L|}:set evs;  
Nonce n ~:used evs |] ==> L:guard n Ks"  
<proof>

### 32.11 guardedness of messages

**lemma** chain\_guard [iff]: "chain B ofr A L C:guard n {priK A}"  
<proof>

**lemma** chain\_guard\_Nonce\_neq [intro]: "n ~= ofr  
==> chain B ofr A' L C:guard n {priK A}"  
<proof>

**lemma** anchor\_guard [iff]: "anchor A n' B:guard n {priK A}"  
<proof>

**lemma** anchor\_guard\_Nonce\_neq [intro]: "n ~= n'  
==> anchor A' n' B:guard n {priK A}"  
<proof>

**lemma** reqm\_guard [intro]: "I:agl ==> reqm A r n' I B:guard n {priK A}"  
<proof>

**lemma** reqm\_guard\_Nonce\_neq [intro]: "[| n ~= n'; I:agl |]  
==> reqm A' r n' I B:guard n {priK A}"  
<proof>

**lemma** prom\_guard [intro]: "[| I:agl; J:agl; L:guard n {priK A} |]  
==> prom B ofr A r I L J C:guard n {priK A}"  
<proof>

**lemma** prom\_guard\_Nonce\_neq [intro]: "[| n ~= ofr; I:agl; J:agl;  
L:guard n {priK A} |] ==> prom B ofr A' r I L J C:guard n {priK A}"  
<proof>

### 32.12 Nonce uniqueness

**lemma** uniq\_Nonce\_in\_chain [dest]: "Nonce k:parts {chain B ofr A L C} ==>  
k=ofr"  
<proof>

**lemma** uniq\_Nonce\_in\_anchor [dest]: "Nonce k:parts {anchor A n B} ==> k=n"  
<proof>

**lemma** uniq\_Nonce\_in\_reqm [dest]: "[| Nonce k:parts {reqm A r n I B};  
I:agl |] ==> k=n"  
<proof>

**lemma** uniq\_Nonce\_in\_prom [dest]: "[| Nonce k:parts {prom B ofr A r I L J

```

C};
I:agl; J:agl; Nonce k ~:parts {L} [] ==> k=ofr"
<proof>

```

### 32.13 requests are guarded

```

lemma req_imp_Guard [rule_format]: "[| evs:p2; A ~:bad |] ==>
req A r n I B:set evs --> Guard n {priK A} (spies evs)"
<proof>

```

```

lemma req_imp_Guard_Friend: "[| evs:p2; A ~:bad; req A r n I B:set evs |]
==> Guard n {priK A} (knows_max (Friend C) evs)"
<proof>

```

### 32.14 propositions are guarded

```

lemma pro_imp_Guard [rule_format]: "[| evs:p2; B ~:bad; A ~:bad |] ==>
pro B ofr A r I (cons M L) J C:set evs --> Guard ofr {priK A} (spies evs)"
<proof>

```

```

lemma pro_imp_Guard_Friend: "[| evs:p2; B ~:bad; A ~:bad;
pro B ofr A r I (cons M L) J C:set evs |]
==> Guard ofr {priK A} (knows_max (Friend D) evs)"
<proof>

```

### 32.15 data confidentiality: no one other than the originator can decrypt the offers

```

lemma Nonce_req_notin_spies: "[| evs:p2; req A r n I B:set evs; A ~:bad |]
==> Nonce n ~:analz (spies evs)"
<proof>

```

```

lemma Nonce_req_notin_knows_max_Friend: "[| evs:p2; req A r n I B:set evs;
A ~:bad; A ~= Friend C |] ==> Nonce n ~:analz (knows_max (Friend C) evs)"
<proof>

```

```

lemma Nonce_pro_notin_spies: "[| evs:p2; B ~:bad; A ~:bad;
pro B ofr A r I (cons M L) J C:set evs |] ==> Nonce ofr ~:analz (spies evs)"
<proof>

```

```

lemma Nonce_pro_notin_knows_max_Friend: "[| evs:p2; B ~:bad; A ~:bad;
A ~= Friend D; pro B ofr A r I (cons M L) J C:set evs |]
==> Nonce ofr ~:analz (knows_max (Friend D) evs)"
<proof>

```

### 32.16 forward privacy: only the originator can know the identity of the shops

```

lemma forward_privacy_Spy: "[| evs:p2; B ~:bad; A ~:bad;
pro B ofr A r I (cons M L) J C:set evs |]
==> sign B (Nonce ofr) ~:analz (spies evs)"
<proof>

```

```

lemma forward_privacy_Friend: "[| evs:p2; B ~:bad; A ~:bad; A ~= Friend D;

```

```

pro B ofr A r I (cons M L) J C:set evs []
==> sign B (Nonce ofr) ~:analz (knows_max (Friend D) evs)"
<proof>

```

### 32.17 non repudiability: an offer signed by B has been sent by B

```

lemma Crypt_reqm: "[| Crypt (priK A) X:parts {reqm A' r n I B}; I:agl |]
==> A=A'"
<proof>

```

```

lemma Crypt_prom: "[| Crypt (priK A) X:parts {prom B ofr A' r I L J C};
I:agl; J:agl |] ==> A=B | Crypt (priK A) X:parts {L}"
<proof>

```

```

lemma Crypt_safeness: "[| evs:p2; A ~:bad |] ==> Crypt (priK A) X:parts (spies
evs)
--> (EX B Y. Says A B Y:set evs & Crypt (priK A) X:parts {Y})"
<proof>

```

```

lemma Crypt_Hash_imp_sign: "[| evs:p2; A ~:bad |] ==>
Crypt (priK A) (Hash X):parts (spies evs)
--> (EX B Y. Says A B Y:set evs & sign A X:parts {Y})"
<proof>

```

```

lemma sign_safeness: "[| evs:p2; A ~:bad |] ==> sign A X:parts (spies evs)
--> (EX B Y. Says A B Y:set evs & sign A X:parts {Y})"
<proof>

```

end

## 33 Needham-Schroeder-Lowe Public-Key Protocol

```

theory Guard_NS_Public imports Guard_Public begin

```

### 33.1 messages used in the protocol

abbreviation (input)

```

ns1 :: "agent => agent => nat => event" where
"ns1 A B NA == Says A B (Crypt (pubK B) {|Nonce NA, Agent A|})"

```

abbreviation (input)

```

ns1' :: "agent => agent => agent => nat => event" where
"ns1' A' A B NA == Says A' B (Crypt (pubK B) {|Nonce NA, Agent A|})"

```

abbreviation (input)

```

ns2 :: "agent => agent => nat => nat => event" where
"ns2 B A NA NB == Says B A (Crypt (pubK A) {|Nonce NA, Nonce NB, Agent B|})"

```

abbreviation (input)

```

ns2' :: "agent => agent => agent => nat => nat => event" where

```



```
"ns2' B' B A NA NB == Says B' A (Crypt (pubK A) {|Nonce NA, Nonce NB, Agent
B|})"
```

```
abbreviation (input)
```

```
  ns3 :: "agent => agent => nat => event" where
  "ns3 A B NB == Says A B (Crypt (pubK B) (Nonce NB))"
```

### 33.2 definition of the protocol

```
inductive_set nsp :: "event list set"
```

```
where
```

```
  Nil: "[]:nsp"
```

```
| Fake: "[| evs:nsp; X:synth (analz (spies evs)) |] ==> Says Spy B X # evs
: nsp"
```

```
| NS1: "[| evs1:nsp; Nonce NA ~:used evs1 |] ==> ns1 A B NA # evs1 : nsp"
```

```
| NS2: "[| evs2:nsp; Nonce NB ~:used evs2; ns1' A' A B NA:set evs2 |] ==>
ns2 B A NA NB # evs2:nsp"
```

```
| NS3: "!!A B B' NA NB evs3. [| evs3:nsp; ns1 A B NA:set evs3; ns2' B' B A
NA NB:set evs3 |] ==>
  ns3 A B NB # evs3:nsp"
```

### 33.3 declarations for tactics

```
declare knows_Spy_partsEs [elim]
declare Fake_parts_insert [THEN subsetD, dest]
declare initState.simps [simp del]
```

### 33.4 general properties of nsp

```
lemma nsp_has_no_Gets: "evs:nsp ==> ALL A X. Gets A X ~:set evs"
<proof>
```

```
lemma nsp_is_Gets_correct [iff]: "Gets_correct nsp"
<proof>
```

```
lemma nsp_is_one_step [iff]: "one_step nsp"
<proof>
```

```
lemma nsp_has_only_Says' [rule_format]: "evs:nsp ==>
ev:set evs --> (EX A B X. ev=Says A B X)"
<proof>
```

```
lemma nsp_has_only_Says [iff]: "has_only_Says nsp"
<proof>
```

```
lemma nsp_is_regular [iff]: "regular nsp"
<proof>
```

### 33.5 nonce are used only once

```
lemma NA_is_uniq [rule_format]: "evs:nsp ==>
Crypt (pubK B) {|Nonce NA, Agent A|}:parts (spies evs)
--> Crypt (pubK B') {|Nonce NA, Agent A'|}:parts (spies evs)
--> Nonce NA ~:analz (spies evs) --> A=A' & B=B'"
<proof>
```

```
lemma no_Nonce_NS1_NS2 [rule_format]: "evs:nsp ==>
Crypt (pubK B') {|Nonce NA', Nonce NA, Agent A'|}:parts (spies evs)
--> Crypt (pubK B) {|Nonce NA, Agent A|}:parts (spies evs)
--> Nonce NA:analz (spies evs)"
<proof>
```

```
lemma no_Nonce_NS1_NS2' [rule_format]:
"[| Crypt (pubK B') {|Nonce NA', Nonce NA, Agent A'|}:parts (spies evs);
Crypt (pubK B) {|Nonce NA, Agent A|}:parts (spies evs); evs:nsp |]
==> Nonce NA:analz (spies evs)"
<proof>
```

```
lemma NB_is_uniq [rule_format]: "evs:nsp ==>
Crypt (pubK A) {|Nonce NA, Nonce NB, Agent B|}:parts (spies evs)
--> Crypt (pubK A') {|Nonce NA', Nonce NB, Agent B'|}:parts (spies evs)
--> Nonce NB ~:analz (spies evs) --> A=A' & B=B' & NA=NA'"
<proof>
```

### 33.6 guardedness of NA

```
lemma ns1_imp_Guard [rule_format]: "[| evs:nsp; A ~:bad; B ~:bad |] ==>
ns1 A B NA:set evs --> Guard NA {priK A,priK B} (spies evs)"
<proof>
```

### 33.7 guardedness of NB

```
lemma ns2_imp_Guard [rule_format]: "[| evs:nsp; A ~:bad; B ~:bad |] ==>
ns2 B A NA NB:set evs --> Guard NB {priK A,priK B} (spies evs)"
<proof>
```

### 33.8 Agents' Authentication

```
lemma B_trusts_NS1: "[| evs:nsp; A ~:bad; B ~:bad |] ==>
Crypt (pubK B) {|Nonce NA, Agent A|}:parts (spies evs)
--> Nonce NA ~:analz (spies evs) --> ns1 A B NA:set evs"
<proof>
```

```
lemma A_trusts_NS2: "[| evs:nsp; A ~:bad; B ~:bad |] ==> ns1 A B NA:set evs
--> Crypt (pubK A) {|Nonce NA, Nonce NB, Agent B|}:parts (spies evs)
--> ns2 B A NA NB:set evs"
<proof>
```

```
lemma B_trusts_NS3: "[| evs:nsp; A ~:bad; B ~:bad |] ==> ns2 B A NA NB:set
evs
--> Crypt (pubK B) (Nonce NB):parts (spies evs) --> ns3 A B NB:set evs"
<proof>
```

end

## 34 protocol-independent confidentiality theorem on keys

```
theory GuardK
imports Analz Extensions
begin
```

```
inductive_set
  guardK :: "nat => key set => msg set"
  for n :: nat and Ks :: "key set"
where
  No_Key [intro]: "Key n ~:parts {X} ==> X:guardK n Ks"
| Guard_Key [intro]: "invKey K:Ks ==> Crypt K X:guardK n Ks"
| Crypt [intro]: "X:guardK n Ks ==> Crypt K X:guardK n Ks"
| Pair [intro]: "[| X:guardK n Ks; Y:guardK n Ks |] ==> {|X,Y|}:guardK n Ks"
```

### 34.1 basic facts about *guardK*

```
lemma Nonce_is_guardK [iff]: "Nonce p:guardK n Ks"
<proof>
```

```
lemma Agent_is_guardK [iff]: "Agent A:guardK n Ks"
<proof>
```

```
lemma Number_is_guardK [iff]: "Number r:guardK n Ks"
<proof>
```

```
lemma Key_notin_guardK: "X:guardK n Ks ==> X ~ = Key n"
<proof>
```

```
lemma Key_notin_guardK_iff [iff]: "Key n ~:guardK n Ks"
<proof>
```

```
lemma guardK_has_Crypt [rule_format]: "X:guardK n Ks ==> Key n:parts {X}
--> (EX K Y. Crypt K Y:kparts {X} & Key n:parts {Y})"
<proof>
```

```
lemma Key_notin_kparts_msg: "X:guardK n Ks ==> Key n ~:kparts {X}"
<proof>
```

```
lemma Key_in_kparts_imp_no_guardK: "Key n:kparts H
==> EX X. X:H & X ~:guardK n Ks"
<proof>
```

```
lemma guardK_kparts [rule_format]: "X:guardK n Ks ==>
Y:kparts {X} --> Y:guardK n Ks"
<proof>
```

```
lemma guardK_Crypt: "[| Crypt K Y:guardK n Ks; K ~:invKey'Ks |] ==> Y:guardK
n Ks"
<proof>
```

```
lemma guardK_MPair [iff]: "({|X,Y|}:guardK n Ks)
= (X:guardK n Ks & Y:guardK n Ks)"
<proof>
```

```
lemma guardK_not_guardK [rule_format]: "X:guardK n Ks ==>
Crypt K Y:kparts {X} --> Key n:kparts {Y} --> Y ~:guardK n Ks"
<proof>
```

```
lemma guardK_extend: "[| X:guardK n Ks; Ks <= Ks';
[| K:Ks'; K ~:Ks |] ==> Key K ~:kparts {X} |] ==> X:guardK n Ks'"
<proof>
```

## 34.2 guarded sets

```
constdefs GuardK :: "nat => key set => msg set => bool"
"GuardK n Ks H == ALL X. X:H --> X:guardK n Ks"
```

## 34.3 basic facts about GuardK

```
lemma GuardK_empty [iff]: "GuardK n Ks {}"
<proof>
```

```
lemma Key_notin_kparts [simplified]: "GuardK n Ks H ==> Key n ~:kparts H"
<proof>
```

```
lemma GuardK_must_decrypt: "[| GuardK n Ks H; Key n:analz H |] ==>
EX K Y. Crypt K Y:kparts H & Key (invKey K):kparts H"
<proof>
```

```
lemma GuardK_kparts [intro]: "GuardK n Ks H ==> GuardK n Ks (kparts H)"
<proof>
```

```
lemma GuardK_mono: "[| GuardK n Ks H; G <= H |] ==> GuardK n Ks G"
<proof>
```

```
lemma GuardK_insert [iff]: "GuardK n Ks (insert X H)
= (GuardK n Ks H & X:guardK n Ks)"
<proof>
```

```
lemma GuardK_Un [iff]: "GuardK n Ks (G Un H) = (GuardK n Ks G & GuardK n
Ks H)"
<proof>
```

```
lemma GuardK_synth [intro]: "GuardK n Ks G ==> GuardK n Ks (synth G)"
<proof>
```

```
lemma GuardK_analz [intro]: "[| GuardK n Ks G; ALL K. K:Ks --> Key K ~:analz
G |]
==> GuardK n Ks (analz G)"
<proof>
```

**lemma** in\_GuardK [dest]: "[| X:G; GuardK n Ks G |] ==> X:guardK n Ks"  
 <proof>

**lemma** in\_synth\_GuardK: "[| X:synth G; GuardK n Ks G |] ==> X:guardK n Ks"  
 <proof>

**lemma** in\_analz\_GuardK: "[| X:analz G; GuardK n Ks G;  
 ALL K. K:Ks --> Key K ~:analz G |] ==> X:guardK n Ks"  
 <proof>

**lemma** GuardK\_keyset [simp]: "[| keyset G; Key n ~:G |] ==> GuardK n Ks G"  
 <proof>

**lemma** GuardK\_Un\_keyset: "[| GuardK n Ks G; keyset H; Key n ~:H |]  
 ==> GuardK n Ks (G Un H)"  
 <proof>

**lemma** in\_GuardK\_kparts: "[| X:G; GuardK n Ks G; Y:kparts {X} |] ==> Y:guardK  
 n Ks"  
 <proof>

**lemma** in\_GuardK\_kparts\_neq: "[| X:G; GuardK n Ks G; Key n':kparts {X} |]  
 ==> n ~ = n'"  
 <proof>

**lemma** in\_GuardK\_kparts\_Crypt: "[| X:G; GuardK n Ks G; is\_MPair X;  
 Crypt K Y:kparts {X}; Key n:kparts {Y} |] ==> invKey K:Ks"  
 <proof>

**lemma** GuardK\_extand: "[| GuardK n Ks G; Ks <= Ks';  
 [| K:Ks'; K ~:Ks |] ==> Key K ~:parts G |] ==> GuardK n Ks' G"  
 <proof>

## 34.4 set obtained by decrypting a message

**abbreviation** (input)

decrypt :: "msg set => key => msg => msg set" where  
 "decrypt H K Y == insert Y (H - {Crypt K Y})"

**lemma** analz\_decrypt: "[| Crypt K Y:H; Key (invKey K):H; Key n:analz H |]  
 ==> Key n:analz (decrypt H K Y)"  
 <proof>

**lemma** parts\_decrypt: "[| Crypt K Y:H; X:parts (decrypt H K Y) |] ==> X:parts  
 H"  
 <proof>

## 34.5 number of Crypt's in a message

**consts** crypt\_nb :: "msg => nat"

**recdef** crypt\_nb "measure size"  
 "crypt\_nb (Crypt K X) = Suc (crypt\_nb X)"

```
"crypt_nb {|X,Y|} = crypt_nb X + crypt_nb Y"
"crypt_nb X = 0"
```

### 34.6 basic facts about `crypt_nb`

```
lemma non_empty_crypt_msg: "Crypt K Y:parts {X} ==> crypt_nb X ≠ 0"
⟨proof⟩
```

### 34.7 number of Crypt's in a message list

```
consts cnb :: "msg list => nat"
```

```
recdef cnb "measure size"
"cnb [] = 0"
"cnb (X#l) = crypt_nb X + cnb l"
```

### 34.8 basic facts about `cnb`

```
lemma cnb_app [simp]: "cnb (l @ l') = cnb l + cnb l'"
⟨proof⟩
```

```
lemma mem_cnb_minus: "x mem l ==> cnb l = crypt_nb x + (cnb l - crypt_nb x)"
⟨proof⟩
```

```
lemmas mem_cnb_minus_substI = mem_cnb_minus [THEN ssubst]
```

```
lemma cnb_minus [simp]: "x mem l ==> cnb (remove l x) = cnb l - crypt_nb x"
⟨proof⟩
```

```
lemma parts_cnb: "Z:parts (set l) ==>
cnb l = (cnb l - crypt_nb Z) + crypt_nb Z"
⟨proof⟩
```

```
lemma non_empty_crypt: "Crypt K Y:parts (set l) ==> cnb l ≠ 0"
⟨proof⟩
```

### 34.9 list of kparts

```
lemma kparts_msg_set: "EX l. kparts {X} = set l & cnb l = crypt_nb X"
⟨proof⟩
```

```
lemma kparts_set: "EX l'. kparts (set l) = set l' & cnb l' = cnb l"
⟨proof⟩
```

### 34.10 list corresponding to "decrypt"

```
constdefs decrypt' :: "msg list => key => msg => msg list"
"decrypt' l K Y == Y # remove l (Crypt K Y)"
```

```
declare decrypt'_def [simp]
```

**34.11 basic facts about *decrypt*'**

**lemma** *decrypt\_minus*: "*decrypt* (set l) K Y <= set (*decrypt*' l K Y)"  
 <proof>

if the analysis of a finite guarded set gives n then it must also give one of the keys of Ks

**lemma** *GuardK\_invKey\_by\_list* [rule\_format]: "ALL l. cnb l = p  
 --> GuardK n Ks (set l) --> Key n:analz (set l)  
 --> (EX K. K:Ks & Key K:analz (set l))"  
 <proof>

**lemma** *GuardK\_invKey\_finite*: "[| Key n:analz G; GuardK n Ks G; finite G |]  
 ==> EX K. K:Ks & Key K:analz G"  
 <proof>

**lemma** *GuardK\_invKey*: "[| Key n:analz G; GuardK n Ks G |]  
 ==> EX K. K:Ks & Key K:analz G"  
 <proof>

if the analyse of a finite guarded set and a (possibly infinite) set of keys gives n then it must also gives Ks

**lemma** *GuardK\_invKey\_keyset*: "[| Key n:analz (G Un H); GuardK n Ks G; finite G;  
 keyset H; Key n ~:H |] ==> EX K. K:Ks & Key K:analz (G Un H)"  
 <proof>

**end**

**theory** *Shared* imports *Event* begin

**consts**

*shrK* :: "agent => key"

**specification** (*shrK*)

*inj\_shrK*: "inj *shrK*"

— No two agents have the same long-term key

<proof>

All keys are symmetric

**defs** *all\_symmetric\_def*: "all\_symmetric == True"

**lemma** *isSym\_keys*: "K ∈ symKeys"

<proof>

Server knows all long-term keys; other agents know only their own

**primrec**

*initState\_Server*: "initState Server = Key ' range *shrK*"

*initState\_Friend*: "initState (Friend i) = {Key (*shrK* (Friend i))}"

*initState\_Spy*: "initState Spy = Key '*shrK*'bad"

### 34.12 Basic properties of shrK

```
lemmas shrK_injective = inj_shrK [THEN inj_eq]
declare shrK_injective [iff]
```

```
lemma invKey_K [simp]: "invKey K = K"
<proof>
```

```
lemma analz_Decrypt' [dest]:
  "[| Crypt K X ∈ analz H; Key K ∈ analz H |] ==> X ∈ analz H"
<proof>
```

Now cancel the *dest* attribute given to *analz.Decrypt* in its declaration.

```
declare analz.Decrypt [rule del]
```

Rewrites should not refer to *initState (Friend i)* because that expression is not in normal form.

```
lemma keysFor_parts_initState [simp]: "keysFor (parts (initState C)) = {}"
<proof>
```

```
lemma keysFor_parts_insert:
  "[| K ∈ keysFor (parts (insert X G)); X ∈ synth (analz H) |]
   ==> K ∈ keysFor (parts (G ∪ H)) | Key K ∈ parts H"
<proof>
```

```
lemma Crypt_imp_keysFor: "Crypt K X ∈ H ==> K ∈ keysFor H"
<proof>
```

### 34.13 Function "knows"

```
lemma Spy_knows_Spy_bad [intro!]: "A: bad ==> Key (shrK A) ∈ knows Spy evs"
<proof>
```

```
lemma Crypt_Spy_analz_bad: "[| Crypt (shrK A) X ∈ analz (knows Spy evs);
A: bad |]
  ==> X ∈ analz (knows Spy evs)"
<proof>
```

```
lemma shrK_in_initState [iff]: "Key (shrK A) ∈ initState A"
<proof>
```

```
lemma shrK_in_used [iff]: "Key (shrK A) ∈ used evs"
<proof>
```

```
lemma Key_not_used [simp]: "Key K ∉ used evs ==> K ∉ range shrK"
<proof>
```



**lemma** *shrK\_neq [simp]: "Key K  $\notin$  used evs  $\implies$  shrK B  $\neq$  K"*  
*<proof>*

**lemmas** *shrK\_sym\_neq = shrK\_neq [THEN not\_sym]*  
**declare** *shrK\_sym\_neq [simp]*

### 34.14 Fresh nonces

**lemma** *Nonce\_notin\_initState [iff]: "Nonce N  $\notin$  parts (initState B)"*  
*<proof>*

**lemma** *Nonce\_notin\_used\_empty [simp]: "Nonce N  $\notin$  used []"*  
*<proof>*

### 34.15 Supply fresh nonces for possibility theorems.

**lemma** *Nonce\_supply\_lemma: " $\exists N$ . ALL n. N  $\leq$  n  $\implies$  Nonce n  $\notin$  used evs"*  
*<proof>*

**lemma** *Nonce\_supply1: " $\exists N$ . Nonce N  $\notin$  used evs"*  
*<proof>*

**lemma** *Nonce\_supply2: " $\exists N N'$ . Nonce N  $\notin$  used evs & Nonce N'  $\notin$  used evs' & N  $\neq$  N'"*  
*<proof>*

**lemma** *Nonce\_supply3: " $\exists N N' N''$ . Nonce N  $\notin$  used evs & Nonce N'  $\notin$  used evs' & Nonce N''  $\notin$  used evs'' & N  $\neq$  N' & N'  $\neq$  N'' & N  $\neq$  N''"*  
*<proof>*

**lemma** *Nonce\_supply: "Nonce (@ N. Nonce N  $\notin$  used evs)  $\notin$  used evs"*  
*<proof>*

Unlike the corresponding property of nonces, we cannot prove *finite KK  $\implies$   $\exists K$ . K  $\notin$  KK  $\wedge$  Key K  $\notin$  used evs*. We have infinitely many agents and there is nothing to stop their long-term keys from exhausting all the natural numbers. Instead, possibility theorems must assume the existence of a few keys.

### 34.16 Specialized Rewriting for Theorems About *analz* and Image

**lemma** *subset\_Compl\_range: "A  $\leq$  - (range shrK)  $\implies$  shrK x  $\notin$  A"*  
*<proof>*

**lemma** *insert\_Key\_singleton: "insert (Key K) H = Key ' {K}  $\cup$  H"*  
*<proof>*

**lemma** *insert\_Key\_image: "insert (Key K) (Key'KK  $\cup$  C) = Key'(insert K KK)  $\cup$  C"*  
*<proof>*

```

lemmas analz_image_freshK_simps =
  simp_thms mem_simps — these two allow its use with only:
  disj_comms
  image_insert [THEN sym] image_Un [THEN sym] empty_subsetI insert_subset
  analz_insert_eq Un_upper2 [THEN analz_mono, THEN [2] rev_subsetD]
  insert_Key_singleton subset_Compl_range
  Key_not_used insert_Key_image Un_assoc [THEN sym]

```

```

lemma analz_image_freshK_lemma:
  "(Key K ∈ analz (Key'nE ∪ H)) --> (K ∈ nE | Key K ∈ analz H) ==>
   (Key K ∈ analz (Key'nE ∪ H)) = (K ∈ nE | Key K ∈ analz H)"
<proof>

```

### 34.17 Tactics for possibility theorems

<ML>

```

lemma invKey_shrK_iff [iff]:
  "(Key (invKey K) ∈ X) = (Key K ∈ X)"
<proof>

```

<ML>

```

lemma knows_subset_knows_Cons: "knows A evs <= knows A (e # evs)"
<proof>

```

**end**

## 35 lemmas on guarded messages for protocols with symmetric keys

```

theory Guard_Shared imports Guard GuardK Shared begin

```

### 35.1 Extensions to Theory *Shared*

```

declare initState_simps [simp del]

```

#### 35.1.1 a little abbreviation

```

abbreviation
  Ciph :: "agent => msg => msg" where
    "Ciph A X == Crypt (shrK A) X"

```

### 35.1.2 agent associated to a key

```
constdefs agt :: "key => agent"
"agt K == @A. K = shrK A"
```

```
lemma agt_shrK [simp]: "agt (shrK A) = A"
<proof>
```

### 35.1.3 basic facts about initState

```
lemma no_Crypt_in_parts_init [simp]: "Crypt K X ~:parts (initState A)"
<proof>
```

```
lemma no_Crypt_in_analz_init [simp]: "Crypt K X ~:analz (initState A)"
<proof>
```

```
lemma no_shrK_in_analz_init [simp]: "A ~:bad
==> Key (shrK A) ~:analz (initState Spy)"
<proof>
```

```
lemma shrK_notin_initState_Friend [simp]: "A ~= Friend C
==> Key (shrK A) ~: parts (initState (Friend C))"
<proof>
```

```
lemma keyset_init [iff]: "keyset (initState A)"
<proof>
```

### 35.1.4 sets of symmetric keys

```
constdefs shrK_set :: "key set => bool"
"shrK_set Ks == ALL K. K:Ks --> (EX A. K = shrK A)"
```

```
lemma in_shrK_set: "[| shrK_set Ks; K:Ks |] ==> EX A. K = shrK A"
<proof>
```

```
lemma shrK_set1 [iff]: "shrK_set {shrK A}"
<proof>
```

```
lemma shrK_set2 [iff]: "shrK_set {shrK A, shrK B}"
<proof>
```

### 35.1.5 sets of good keys

```
constdefs good :: "key set => bool"
"good Ks == ALL K. K:Ks --> agt K ~:bad"
```

```
lemma in_good: "[| good Ks; K:Ks |] ==> agt K ~:bad"
<proof>
```

```
lemma good1 [simp]: "A ~:bad ==> good {shrK A}"
<proof>
```

```
lemma good2 [simp]: "[| A ~:bad; B ~:bad |] ==> good {shrK A, shrK B}"
<proof>
```

## 35.2 Proofs About Guarded Messages

### 35.2.1 small hack

**lemma** *shrK\_is\_invKey\_shrK*: "shrK A = invKey (shrK A)"  
 <proof>

**lemmas** *shrK\_is\_invKey\_shrK\_substI* = *shrK\_is\_invKey\_shrK* [THEN *ssubst*]

**lemmas** *invKey\_invKey\_substI* = *invKey* [THEN *ssubst*]

**lemma** "Nonce *n*:parts {*X*} ==> Crypt (shrK A) *X*:guard *n* {shrK A}"  
 <proof>

### 35.2.2 guardedness results on nonces

**lemma** *guard\_ciph [simp]*: "shrK A:Ks ==> Ciph A *X*:guard *n* Ks"  
 <proof>

**lemma** *guardK\_ciph [simp]*: "shrK A:Ks ==> Ciph A *X*:guardK *n* Ks"  
 <proof>

**lemma** *Guard\_init [iff]*: "Guard *n* Ks (initState B)"  
 <proof>

**lemma** *Guard\_knows\_max'*: "Guard *n* Ks (knows\_max' C evs)  
 ==> Guard *n* Ks (knows\_max C evs)"  
 <proof>

**lemma** *Nonce\_not\_used\_Guard\_spies [dest]*: "Nonce *n* ~:used evs  
 ==> Guard *n* Ks (spies evs)"  
 <proof>

**lemma** *Nonce\_not\_used\_Guard [dest]*: "[| evs:p; Nonce *n* ~:used evs;  
 Gets\_correct p; one\_step p |] ==> Guard *n* Ks (knows (Friend C) evs)"  
 <proof>

**lemma** *Nonce\_not\_used\_Guard\_max [dest]*: "[| evs:p; Nonce *n* ~:used evs;  
 Gets\_correct p; one\_step p |] ==> Guard *n* Ks (knows\_max (Friend C) evs)"  
 <proof>

**lemma** *Nonce\_not\_used\_Guard\_max' [dest]*: "[| evs:p; Nonce *n* ~:used evs;  
 Gets\_correct p; one\_step p |] ==> Guard *n* Ks (knows\_max' (Friend C) evs)"  
 <proof>

### 35.2.3 guardedness results on keys

**lemma** *GuardK\_init [simp]*: "*n* ~:range shrK ==> GuardK *n* Ks (initState B)"  
 <proof>

**lemma** *GuardK\_knows\_max'*: "[| GuardK *n* A (knows\_max' C evs); *n* ~:range shrK  
 |]  
 ==> GuardK *n* A (knows\_max C evs)"  
 <proof>

```
lemma Key_not_used_GuardK_spies [dest]: "Key n ~:used evs
==> GuardK n A (spies evs)"
<proof>
```

```
lemma Key_not_used_GuardK [dest]: "[| evs:p; Key n ~:used evs;
Gets_correct p; one_step p |] ==> GuardK n A (knows (Friend C) evs)"
<proof>
```

```
lemma Key_not_used_GuardK_max [dest]: "[| evs:p; Key n ~:used evs;
Gets_correct p; one_step p |] ==> GuardK n A (knows_max (Friend C) evs)"
<proof>
```

```
lemma Key_not_used_GuardK_max' [dest]: "[| evs:p; Key n ~:used evs;
Gets_correct p; one_step p |] ==> GuardK n A (knows_max' (Friend C) evs)"
<proof>
```

### 35.2.4 regular protocols

```
constdefs regular :: "event list set => bool"
"regular p == ALL evs A. evs:p --> (Key (shrK A):parts (spies evs)) = (A:bad)"
```

```
lemma shrK_parts_iff_bad [simp]: "[| evs:p; regular p |] ==>
(Key (shrK A):parts (spies evs)) = (A:bad)"
<proof>
```

```
lemma shrK_analz_iff_bad [simp]: "[| evs:p; regular p |] ==>
(Key (shrK A):analz (spies evs)) = (A:bad)"
<proof>
```

```
lemma Guard_Nonce_analz: "[| Guard n Ks (spies evs); evs:p;
shrK_set Ks; good Ks; regular p |] ==> Nonce n ~:analz (spies evs)"
<proof>
```

```
lemma GuardK_Key_analz: "[| GuardK n Ks (spies evs); evs:p;
shrK_set Ks; good Ks; regular p; n ~:range shrK |] ==> Key n ~:analz (spies
evs)"
<proof>
```

end

## 36 Otway-Rees Protocol

```
theory Guard_OtwayRees imports Guard_Shared begin
```

### 36.1 messages used in the protocol

abbreviation

```
nil :: "msg" where
"nil == Number 0"
```

abbreviation

```
or1 :: "agent => agent => nat => event" where
"or1 A B NA ==
```

Says A B {|Nonce NA, Agent A, Agent B, Ciph A {|Nonce NA, Agent A, Agent B|}|}

**abbreviation**

or1' :: "agent => agent => agent => nat => msg => event" where  
 "or1' A' A B NA X == Says A' B {|Nonce NA, Agent A, Agent B, X|}"

**abbreviation**

or2 :: "agent => agent => nat => nat => msg => event" where  
 "or2 A B NA NB X ==  
 Says B Server {|Nonce NA, Agent A, Agent B, X,  
 Ciph B {|Nonce NA, Nonce NB, Agent A, Agent B|}|}"

**abbreviation**

or2' :: "agent => agent => agent => nat => nat => event" where  
 "or2' B' A B NA NB ==  
 Says B' Server {|Nonce NA, Agent A, Agent B,  
 Ciph A {|Nonce NA, Agent A, Agent B|},  
 Ciph B {|Nonce NA, Nonce NB, Agent A, Agent B|}|}"

**abbreviation**

or3 :: "agent => agent => nat => nat => key => event" where  
 "or3 A B NA NB K ==  
 Says Server B {|Nonce NA, Ciph A {|Nonce NA, Key K|},  
 Ciph B {|Nonce NB, Key K|}|}"

**abbreviation**

or3' :: "agent => msg => agent => agent => nat => nat => key => event" where  
 "or3' S Y A B NA NB K ==  
 Says S B {|Nonce NA, Y, Ciph B {|Nonce NB, Key K|}|}"

**abbreviation**

or4 :: "agent => agent => nat => msg => event" where  
 "or4 A B NA X == Says B A {|Nonce NA, X, nil|}"

**abbreviation**

or4' :: "agent => agent => nat => key => event" where  
 "or4' B' A NA K == Says B' A {|Nonce NA, Ciph A {|Nonce NA, Key K|}, nil|}"

## 36.2 definition of the protocol

inductive\_set or :: "event list set"

where

Nil: "[]:or"

| Fake: "[| evs:or; X:synth (analz (spies evs)) |] ==> Says Spy B X # evs:or"

| OR1: "[| evs1:or; Nonce NA ~:used evs1 |] ==> or1 A B NA # evs1:or"

| OR2: "[| evs2:or; or1' A' A B NA X:set evs2; Nonce NB ~:used evs2 |]  
 ==> or2 A B NA NB X # evs2:or"

| OR3: "[| evs3:or; or2' B' A B NA NB:set evs3; Key K ~:used evs3 |]"

```

==> or3 A B NA NB K # evs3:or"

| OR4: "[| evs4:or; or2 A B NA NB X:set evs4; or3' S Y A B NA NB K:set evs4
|]
==> or4 A B NA X # evs4:or"

```

### 36.3 declarations for tactics

```

declare knows_Spy_partsEs [elim]
declare Fake_parts_insert [THEN subsetD, dest]
declare initState.simps [simp del]

```

### 36.4 general properties of or

```

lemma or_has_no_Gets: "evs:or ==> ALL A X. Gets A X ~:set evs"
<proof>

```

```

lemma or_is_Gets_correct [iff]: "Gets_correct or"
<proof>

```

```

lemma or_is_one_step [iff]: "one_step or"
<proof>

```

```

lemma or_has_only_Says' [rule_format]: "evs:or ==>
ev:set evs --> (EX A B X. ev=Says A B X)"
<proof>

```

```

lemma or_has_only_Says [iff]: "has_only_Says or"
<proof>

```

### 36.5 or is regular

```

lemma or1'_parts_spies [dest]: "or1' A' A B NA X:set evs
==> X:parts (spies evs)"
<proof>

```

```

lemma or2_parts_spies [dest]: "or2 A B NA NB X:set evs
==> X:parts (spies evs)"
<proof>

```

```

lemma or3_parts_spies [dest]: "Says S B {|NA, Y, Ciph B {|NB, K|}|}:set evs
==> K:parts (spies evs)"
<proof>

```

```

lemma or_is_regular [iff]: "regular or"
<proof>

```

### 36.6 guardedness of KAB

```

lemma Guard_KAB [rule_format]: "[| evs:or; A ~:bad; B ~:bad |] ==>
or3 A B NA NB K:set evs --> GuardK K {shrK A, shrK B} (spies evs)"
<proof>

```

### 36.7 guardedness of NB

```
lemma Guard_NB [rule_format]: "[| evs:or; B ~:bad |] ==>
or2 A B NA NB X:set evs --> Guard NB {shrK B} (spies evs)"
<proof>

end
```

## 37 Yahalom Protocol

```
theory Guard_Yahalom imports Guard_Shared begin
```

### 37.1 messages used in the protocol

```
abbreviation (input)
```

```
ya1 :: "agent => agent => nat => event" where
"ya1 A B NA == Says A B {|Agent A, Nonce NA|}"
```

```
abbreviation (input)
```

```
ya1' :: "agent => agent => agent => nat => event" where
"ya1' A' A B NA == Says A' B {|Agent A, Nonce NA|}"
```

```
abbreviation (input)
```

```
ya2 :: "agent => agent => nat => nat => event" where
"ya2 A B NA NB == Says B Server {|Agent B, Ciph B {|Agent A, Nonce NA, Nonce
NB|}|}"
```

```
abbreviation (input)
```

```
ya2' :: "agent => agent => agent => nat => nat => event" where
"ya2' B' A B NA NB == Says B' Server {|Agent B, Ciph B {|Agent A, Nonce NA,
Nonce NB|}|}"
```

```
abbreviation (input)
```

```
ya3 :: "agent => agent => nat => nat => key => event" where
"ya3 A B NA NB K ==
Says Server A {|Ciph A {|Agent B, Key K, Nonce NA, Nonce NB|},
Ciph B {|Agent A, Key K|}|}"
```

```
abbreviation (input)
```

```
ya3' :: "agent => msg => agent => agent => nat => nat => key => event" where
"ya3' S Y A B NA NB K ==
Says S A {|Ciph A {|Agent B, Key K, Nonce NA, Nonce NB|}, Y|}"
```

```
abbreviation (input)
```

```
ya4 :: "agent => agent => nat => nat => msg => event" where
"ya4 A B K NB Y == Says A B {|Y, Crypt K (Nonce NB)|}"
```

```
abbreviation (input)
```

```
ya4' :: "agent => agent => nat => nat => msg => event" where
"ya4' A' B K NB Y == Says A' B {|Y, Crypt K (Nonce NB)|}"
```



## 37.2 definition of the protocol

**inductive\_set** *ya* :: "event list set"  
**where**

```

  Nil: "[]:ya"

| Fake: "[| evs:ya; X:synth (analz (spies evs)) |] ==> Says Spy B X # evs:ya"

| YA1: "[| evs1:ya; Nonce NA ~:used evs1 |] ==> ya1 A B NA # evs1:ya"

| YA2: "[| evs2:ya; ya1' A' A B NA:set evs2; Nonce NB ~:used evs2 |]
  ==> ya2 A B NA NB # evs2:ya"

| YA3: "[| evs3:ya; ya2' B' A B NA NB:set evs3; Key K ~:used evs3 |]
  ==> ya3 A B NA NB K # evs3:ya"

| YA4: "[| evs4:ya; ya1 A B NA:set evs4; ya3' S Y A B NA NB K:set evs4 |]
  ==> ya4 A B K NB Y # evs4:ya"

```

## 37.3 declarations for tactics

**declare** *knows\_Spy\_partsEs* [elim]  
**declare** *Fake\_parts\_insert* [THEN subsetD, dest]  
**declare** *initState.simps* [simp del]

## 37.4 general properties of ya

**lemma** *ya\_has\_no\_Gets*: "evs:ya ==> ALL A X. Gets A X ~:set evs"  
 <proof>

**lemma** *ya\_is\_Gets\_correct* [iff]: "Gets\_correct ya"  
 <proof>

**lemma** *ya\_is\_one\_step* [iff]: "one\_step ya"  
 <proof>

**lemma** *ya\_has\_only\_Says'* [rule\_format]: "evs:ya ==>  
 ev:set evs --> (EX A B X. ev=Says A B X)"  
 <proof>

**lemma** *ya\_has\_only\_Says* [iff]: "has\_only\_Says ya"  
 <proof>

**lemma** *ya\_is\_regular* [iff]: "regular ya"  
 <proof>

## 37.5 guardedness of KAB

**lemma** *Guard\_KAB* [rule\_format]: "[| evs:ya; A ~:bad; B ~:bad |] ==>  
 ya3 A B NA NB K:set evs --> GuardK K {shrK A, shrK B} (spies evs)"  
 <proof>

### 37.6 session keys are not symmetric keys

```
lemma KAB_isnt_shrK [rule_format]: "evs:ya ==>
ya3 A B NA NB K:set evs --> K ~:range shrK"
<proof>
```

```
lemma ya3_shrK: "evs:ya ==> ya3 A B NA NB (shrK C) ~:set evs"
<proof>
```

### 37.7 ya2' implies ya1'

```
lemma ya2'_parts_imp_ya1'_parts [rule_format]:
  "[| evs:ya; B ~:bad |] ==>
    Ciph B {|Agent A, Nonce NA, Nonce NB|}:parts (spies evs) -->
      {|Agent A, Nonce NA|}:spies evs"
<proof>
```

```
lemma ya2'_imp_ya1'_parts: "[| ya2' B' A B NA NB:set evs; evs:ya; B ~:bad
|]
==> {|Agent A, Nonce NA|}:spies evs"
<proof>
```

### 37.8 uniqueness of NB

```
lemma NB_is_uniq_in_ya2'_parts [rule_format]: "[| evs:ya; B ~:bad; B' ~:bad
|] ==>
  Ciph B {|Agent A, Nonce NA, Nonce NB|}:parts (spies evs) -->
  Ciph B' {|Agent A', Nonce NA', Nonce NB|}:parts (spies evs) -->
  A=A' & B=B' & NA=NA'"
<proof>
```

```
lemma NB_is_uniq_in_ya2': "[| ya2' C A B NA NB:set evs;
ya2' C' A' B' NA' NB:set evs; evs:ya; B ~:bad; B' ~:bad |]
==> A=A' & B=B' & NA=NA'"
<proof>
```

### 37.9 ya3' implies ya2'

```
lemma ya3'_parts_imp_ya2'_parts [rule_format]: "[| evs:ya; A ~:bad |] ==>
  Ciph A {|Agent B, Key K, Nonce NA, Nonce NB|}:parts (spies evs)
--> Ciph B {|Agent A, Nonce NA, Nonce NB|}:parts (spies evs)"
<proof>
```

```
lemma ya3'_parts_imp_ya2' [rule_format]: "[| evs:ya; A ~:bad |] ==>
  Ciph A {|Agent B, Key K, Nonce NA, Nonce NB|}:parts (spies evs)
--> (EX B'. ya2' B' A B NA NB:set evs)"
<proof>
```

```
lemma ya3'_imp_ya2': "[| ya3' S Y A B NA NB K:set evs; evs:ya; A ~:bad |]
==> (EX B'. ya2' B' A B NA NB:set evs)"
<proof>
```

### 37.10 ya3' implies ya3

```
lemma ya3'_parts_imp_ya3 [rule_format]: "[| evs:ya; A ~:bad |] ==>
```

```
Ciph A {|Agent B, Key K, Nonce NA, Nonce NB|}:parts(spies evs)
--> ya3 A B NA NB K:set evs"
⟨proof⟩
```

```
lemma ya3'_imp_ya3: "[| ya3' S Y A B NA NB K:set evs; evs:ya; A ~:bad |]
==> ya3 A B NA NB K:set evs"
⟨proof⟩
```

### 37.11 guardedness of NB

```
constdefs ya_keys :: "agent => agent => nat => nat => event list => key set"
"ya_keys A B NA NB evs == {shrK A, shrK B} Un {K. ya3 A B NA NB K:set evs}"

lemma Guard_NB [rule_format]: "[| evs:ya; A ~:bad; B ~:bad |] ==>
ya2 A B NA NB:set evs --> Guard NB (ya_keys A B NA NB evs) (spies evs)"
⟨proof⟩

end
```

## 38 Other Protocol-Independent Results

```
theory Proto imports Guard_Public begin
```

### 38.1 protocols

```
types rule = "event set * event"

abbreviation
  msg' :: "rule => msg" where
    "msg' R == msg (snd R)"

types proto = "rule set"

constdefs wdef :: "proto => bool"
"wdef p == ALL R k. R:p --> Number k:parts {msg' R}
--> Number k:parts (msg'(fst R))"
```

### 38.2 substitutions

```
record subs =
  agent    :: "agent => agent"
  nonce    :: "nat => nat"
  nb       :: "nat => msg"
  key      :: "key => key"

consts apm :: "subs => msg => msg"

primrec
  "apm s (Agent A) = Agent (agent s A)"
  "apm s (Nonce n) = Nonce (nonce s n)"
  "apm s (Number n) = nb s n"
  "apm s (Key K) = Key (key s K)"
  "apm s (Hash X) = Hash (apm s X)"
```

```

"apm s (Crypt K X) = (
  if (EX A. K = pubK A) then Crypt (pubK (agent s (agt K))) (apm s X)
  else if (EX A. K = priK A) then Crypt (priK (agent s (agt K))) (apm s X)
  else Crypt (key s K) (apm s X))"
"apm s {|X,Y|} = {|apm s X, apm s Y|}"

```

```

lemma apm_parts: "X:parts {Y} ==> apm s X:parts {apm s Y}"
<proof>

```

```

lemma Nonce_apm [rule_format]: "Nonce n:parts {apm s X} ==>
  (ALL k. Number k:parts {X} --> Nonce n ~:parts {nb s k}) -->
  (EX k. Nonce k:parts {X} & nonce s k = n)"
<proof>

```

```

lemma wdef_Nonce: "[| Nonce n:parts {apm s X}; R:p; msg' R = X; wdef p;
  Nonce n ~:parts (apm s '(msg' (fst R))) |] ==>
  (EX k. Nonce k:parts {X} & nonce s k = n)"
<proof>

```

```

consts ap :: "subs => event => event"

```

**primrec**

```

"ap s (Says A B X) = Says (agent s A) (agent s B) (apm s X)"
"ap s (Gets A X) = Gets (agent s A) (apm s X)"
"ap s (Notes A X) = Notes (agent s A) (apm s X)"

```

**abbreviation**

```

ap' :: "subs => rule => event" where
  "ap' s R == ap s (snd R)"

```

**abbreviation**

```

apm' :: "subs => rule => msg" where
  "apm' s R == apm s (msg' R)"

```

**abbreviation**

```

priK' :: "subs => agent => key" where
  "priK' s A == priK (agent s A)"

```

**abbreviation**

```

pubK' :: "subs => agent => key" where
  "pubK' s A == pubK (agent s A)"

```

### 38.3 nonces generated by a rule

```

constdefs newn :: "rule => nat set"
"newn R == {n. Nonce n:parts {msg' (snd R)} & Nonce n ~:parts (msg' (fst R))}"

```

```

lemma newn_parts: "n:newn R ==> Nonce (nonce s n):parts {apm' s R}"
<proof>

```

### 38.4 traces generated by a protocol

```

constdefs ok :: "event list => rule => subs => bool"
"ok evs R s == ((ALL x. x:fst R --> ap s x:set evs)

```

```

& (ALL n. n:newn R --> Nonce (nonce s n) ~:used evs))"

inductive_set
  tr :: "proto => event list set"
  for p :: proto
where

  Nil [intro]: "[]:tr p"

  | Fake [intro]: "[| evsf:tr p; X:synth (analz (spies evsf)) |]
    ==> Says Spy B X # evsf:tr p"

  | Proto [intro]: "[| evs:tr p; R:p; ok evs R s |] ==> ap' s R # evs:tr p"

```

### 38.5 general properties

```

lemma one_step_tr [iff]: "one_step (tr p)"
<proof>

constdefs has_only_Says' :: "proto => bool"
"has_only_Says' p == ALL R. R:p --> is_Says (snd R)"

lemma has_only_Says'D: "[| R:p; has_only_Says' p |]
==> (EX A B X. snd R = Says A B X)"
<proof>

lemma has_only_Says_tr [simp]: "has_only_Says' p ==> has_only_Says (tr p)"
<proof>

lemma has_only_Says'_in_trD: "[| has_only_Says' p; list @ ev # evs1 ∈ tr
p |]
==> (EX A B X. ev = Says A B X)"
<proof>

lemma ok_not_used: "[| Nonce n ~:used evs; ok evs R s;
ALL x. x:fst R --> is_Says x |] ==> Nonce n ~:parts (apm s '(msg '(fst R)))"
<proof>

lemma ok_is_Says: "[| evs' @ ev # evs:tr p; ok evs R s; has_only_Says' p;
R:p; x:fst R |] ==> is_Says x"
<proof>

```

### 38.6 types

```

types keyfun = "rule => subs => nat => event list => key set"

types secfun = "rule => nat => subs => key set => msg"

```

### 38.7 introduction of a fresh guarded nonce

```

constdefs fresh :: "proto => rule => subs => nat => key set => event list
=> bool"
"fresh p R s n Ks evs == (EX evs1 evs2. evs = evs2 @ ap' s R # evs1
& Nonce n ~:used evs1 & R:p & ok evs1 R s & Nonce n:parts {apm' s R})

```

& apm' s R:guard n Ks)"

**lemma** freshD: "fresh p R s n Ks evs ==> (EX evs1 evs2.  
evs = evs2 @ ap' s R # evs1 & Nonce n ~:used evs1 & R:p & ok evs1 R s  
& Nonce n:parts {apm' s R} & apm' s R:guard n Ks)"  
<proof>

**lemma** freshI [intro]: "[| Nonce n ~:used evs1; R:p; Nonce n:parts {apm' s  
R};  
ok evs1 R s; apm' s R:guard n Ks |]  
==> fresh p R s n Ks (list @ ap' s R # evs1)"  
<proof>

**lemma** freshI': "[| Nonce n ~:used evs1; (l,r):p;  
Nonce n:parts {apm s (msg r)}; ok evs1 (l,r) s; apm s (msg r):guard n Ks |]  
==> fresh p (l,r) s n Ks (evs2 @ ap s r # evs1)"  
<proof>

**lemma** fresh\_used: "[| fresh p R' s' n Ks evs; has\_only\_Says' p |]  
==> Nonce n:used evs"  
<proof>

**lemma** fresh\_newn: "[| evs' @ ap' s R # evs:tr p; wdef p; has\_only\_Says' p;  
Nonce n ~:used evs; R:p; ok evs R s; Nonce n:parts {apm' s R} |]  
==> EX k. k:newn R & nonce s k = n"  
<proof>

**lemma** fresh\_rule: "[| evs' @ ev # evs:tr p; wdef p; Nonce n ~:used evs;  
Nonce n:parts {msg ev} |] ==> EX R s. R:p & ap' s R = ev"  
<proof>

**lemma** fresh\_ruleD: "[| fresh p R' s' n Ks evs; keys R' s' n evs <= Ks; wdef  
p;  
has\_only\_Says' p; evs:tr p; ALL R k s. nonce s k = n --> Nonce n:used evs -->  
R:p --> k:newn R --> Nonce n:parts {apm' s R} --> apm' s R:guard n Ks -->  
apm' s R:parts (spies evs) --> keys R s n evs <= Ks --> P |] ==> P"  
<proof>

### 38.8 safe keys

**consts** safe :: "key set => msg set => bool"  
"safe Ks G == ALL K. K:Ks --> Key K ~:analz G"

**lemma** safeD [dest]: "[| safe Ks G; K:Ks |] ==> Key K ~:analz G"  
<proof>

**lemma** safe\_insert: "safe Ks (insert X G) ==> safe Ks G"  
<proof>

**lemma** Guard\_safe: "[| Guard n Ks G; safe Ks G |] ==> Nonce n ~:analz G"  
<proof>

### 38.9 guardedness preservation

```
constdefs preserv :: "proto => keyfun => nat => key set => bool"
"preserv p keys n Ks == (ALL evs R' s' R s. evs:tr p -->
Guard n Ks (spies evs) --> safe Ks (spies evs) --> fresh p R' s' n Ks evs -->
keys R' s' n evs <= Ks --> R:p --> ok evs R s --> apm' s R:guard n Ks)"
```

```
lemma preservD: "[| preserv p keys n Ks; evs:tr p; Guard n Ks (spies evs);
safe Ks (spies evs); fresh p R' s' n Ks evs; R:p; ok evs R s;
keys R' s' n evs <= Ks |] ==> apm' s R:guard n Ks"
<proof>
```

```
lemma preservD': "[| preserv p keys n Ks; evs:tr p; Guard n Ks (spies evs);
safe Ks (spies evs); fresh p R' s' n Ks evs; (l,Says A B X):p;
ok evs (l,Says A B X) s; keys R' s' n evs <= Ks |] ==> apm s X:guard n Ks"
<proof>
```

### 38.10 monotonic keyfun

```
constdefs monoton :: "proto => keyfun => bool"
"monoton p keys == ALL R' s' n ev evs. ev # evs:tr p -->
keys R' s' n evs <= keys R' s' n (ev # evs)"
```

```
lemma monotonD [dest]: "[| keys R' s' n (ev # evs) <= Ks; monoton p keys;
ev # evs:tr p |] ==> keys R' s' n evs <= Ks"
<proof>
```

### 38.11 guardedness theorem

```
lemma Guard_tr [rule_format]: "[| evs:tr p; has_only_Says' p;
preserv p keys n Ks; monoton p keys; Guard n Ks (initState Spy) |] ==>
safe Ks (spies evs) --> fresh p R' s' n Ks evs --> keys R' s' n evs <= Ks -->
Guard n Ks (spies evs)"
<proof>
```

### 38.12 useful properties for guardedness

```
lemma newn_neq_used: "[| Nonce n:used evs; ok evs R s; k:newn R |]
==> n ~= nonce s k"
<proof>
```

```
lemma ok_Guard: "[| ok evs R s; Guard n Ks (spies evs); x:fst R; is_Says
x |]
==> apm s (msg x):parts (spies evs) & apm s (msg x):guard n Ks"
<proof>
```

```
lemma ok_parts_not_new: "[| Y:parts (spies evs); Nonce (nonce s n):parts
{Y};
ok evs R s |] ==> n ~:newn R"
<proof>
```

### 38.13 unicity

```
constdefs uniq :: "proto => secfun => bool"
"uniq p secret == ALL evs R R' n n' Ks s s'. R:p --> R':p -->
```

```

n:newn R --> n':newn R' --> nonce s n = nonce s' n' -->
Nonce (nonce s n):parts {apm' s R} --> Nonce (nonce s n):parts {apm' s' R'}
-->
apm' s R:guard (nonce s n) Ks --> apm' s' R':guard (nonce s n) Ks -->
evs:tr p --> Nonce (nonce s n) ~:analz (spies evs) -->
secret R n s Ks:parts (spies evs) --> secret R' n' s' Ks:parts (spies evs)
-->
secret R n s Ks = secret R' n' s' Ks"

```

```

lemma uniqD: "[| uniq p secret; evs: tr p; R:p; R':p; n:newn R; n':newn R';
nonce s n = nonce s' n'; Nonce (nonce s n) ~:analz (spies evs);
Nonce (nonce s n):parts {apm' s R}; Nonce (nonce s n):parts {apm' s' R'};
secret R n s Ks:parts (spies evs); secret R' n' s' Ks:parts (spies evs);
apm' s R:guard (nonce s n) Ks; apm' s' R':guard (nonce s n) Ks |] ==>
secret R n s Ks = secret R' n' s' Ks"
<proof>

```

```

constdefs ord :: "proto => (rule => rule => bool) => bool"
"ord p inff == ALL R R'. R:p --> R':p --> ~ inff R R' --> inff R' R"

```

```

lemma ordD: "[| ord p inff; ~ inff R R'; R:p; R':p |] ==> inff R' R"
<proof>

```

```

constdefs uniq' :: "proto => (rule => rule => bool) => secfun => bool"
"uniq' p inff secret == ALL evs R R' n n' Ks s s'. R:p --> R':p -->
inff R R' --> n:newn R --> n':newn R' --> nonce s n = nonce s' n' -->
Nonce (nonce s n):parts {apm' s R} --> Nonce (nonce s n):parts {apm' s' R'}
-->
apm' s R:guard (nonce s n) Ks --> apm' s' R':guard (nonce s n) Ks -->
evs:tr p --> Nonce (nonce s n) ~:analz (spies evs) -->
secret R n s Ks:parts (spies evs) --> secret R' n' s' Ks:parts (spies evs)
-->
secret R n s Ks = secret R' n' s' Ks"

```

```

lemma uniq'D: "[| uniq' p inff secret; evs: tr p; inff R R'; R:p; R':p; n:newn
R;
n':newn R'; nonce s n = nonce s' n'; Nonce (nonce s n) ~:analz (spies evs);
Nonce (nonce s n):parts {apm' s R}; Nonce (nonce s n):parts {apm' s' R'};
secret R n s Ks:parts (spies evs); secret R' n' s' Ks:parts (spies evs);
apm' s R:guard (nonce s n) Ks; apm' s' R':guard (nonce s n) Ks |] ==>
secret R n s Ks = secret R' n' s' Ks"
<proof>

```

```

lemma uniq'_imp_uniq: "[| uniq' p inff secret; ord p inff |] ==> uniq p secret"
<proof>

```

### 38.14 Needham-Schroeder-Lowe

```

constdefs
a :: agent "a == Friend 0"
b :: agent "b == Friend 1"
a' :: agent "a' == Friend 2"
b' :: agent "b' == Friend 3"
Na :: nat "Na == 0"

```



```
Nb :: nat "Nb == 1"
```

**abbreviation**

```
ns1 :: rule where
  "ns1 == ({}, Says a b (Crypt (pubK b) {|Nonce Na, Agent a|}))"
```

**abbreviation**

```
ns2 :: rule where
  "ns2 == ({Says a' b (Crypt (pubK b) {|Nonce Na, Agent a|})},
    Says b a (Crypt (pubK a) {|Nonce Na, Nonce Nb, Agent b|}))"
```

**abbreviation**

```
ns3 :: rule where
  "ns3 == ({Says a b (Crypt (pubK b) {|Nonce Na, Agent a|}),
    Says b' a (Crypt (pubK a) {|Nonce Na, Nonce Nb, Agent b|})},
    Says a b (Crypt (pubK b) (Nonce Nb)))"
```

**inductive\_set ns :: proto where**

```
[iff]: "ns1:ns"
| [iff]: "ns2:ns"
| [iff]: "ns3:ns"
```

**abbreviation (input)**

```
ns3a :: event where
  "ns3a == Says a b (Crypt (pubK b) {|Nonce Na, Agent a|})"
```

**abbreviation (input)**

```
ns3b :: event where
  "ns3b == Says b' a (Crypt (pubK a) {|Nonce Na, Nonce Nb, Agent b|})"
```

**constdefs keys :: "keyfun"**

```
"keys R' s' n evs == {priK' s' a, priK' s' b}"
```

**lemma "monoton ns keys"**

*<proof>*

**constdefs secret :: "secfun"**

```
"secret R n s Ks ==
(if R=ns1 then apm s (Crypt (pubK b) {|Nonce Na, Agent a|})
else if R=ns2 then apm s (Crypt (pubK a) {|Nonce Na, Nonce Nb, Agent b|})
else Number 0)"
```

**constdefs inf :: "rule => rule => bool"**

```
"inf R R' == (R=ns1 | (R=ns2 & R'~ns1) | (R=ns3 & R'=ns3))"
```

**lemma inf\_is\_ord [iff]: "ord ns inf"**

*<proof>*

## 38.15 general properties

**lemma ns\_has\_only\_Says' [iff]: "has\_only\_Says' ns"**

*<proof>*

**lemma newn\_ns1 [iff]: "newn ns1 = {Na}"**

*<proof>*

**lemma** *newn\_ns2 [iff]: "newn ns2 = {Nb}"*  
*<proof>*

**lemma** *newn\_ns3 [iff]: "newn ns3 = {}"*  
*<proof>*

**lemma** *ns\_wdef [iff]: "wdef ns"*  
*<proof>*

### 38.16 guardedness for NSL

**lemma** *"uniq ns secret ==> preserv ns keys n Ks"*  
*<proof>*

### 38.17 unicity for NSL

**lemma** *"uniq' ns inf secret"*  
*<proof>*

**end**