

Security Protocols

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

June 8, 2008

Contents

1	Theory of Agents and Messages for Security Protocols	12
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	24
2	Theory of Events for Security Protocols	25
2.1	Function <i>knows</i>	26
2.2	Knowledge of Agents	27
2.2.1	Useful for case analysis on whether a hash is a spoof or not	29
2.3	Asymmetric Keys	29
2.4	Basic properties of <i>pubK</i> and $\lambda A. invKey (pubK A)$	30
2.5	”Image” equations that hold for injective functions	31
2.6	Symmetric Keys	31
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 . . .	35
2.12	Specialized Methods for Possibility Theorems	36

3	Needham-Schroeder Shared-Key Protocol and the Issues Property	36
3.1	Inductive proofs about <i>ns_shared</i>	37
3.1.1	Forwarding lemmas, to aid simplification	37
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	39
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.	41
4	The Kerberos Protocol, BAN Version	42
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
5	The Kerberos Protocol, BAN Version, with Gets event	52
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
6	The Kerberos Protocol, Version IV	62
6.1	Lemmas about lists, for reasoning about Issues	66
6.2	Lemmas about <i>authKeys</i>	66
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).	80
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
7	The Kerberos Protocol, Version IV	85
7.1	Lemmas about reception event	89
7.2	Lemmas about <i>authKeys</i>	90
7.3	Forwarding Lemmas	90
7.4	Regularity Lemmas	91
7.5	Authenticity theorems: confirm origin of sensitive messages . . .	93
7.6	Reliability: friendly agents send something if something else happened	95
7.7	Unicity Theorems	97
7.8	Lemmas About the Predicate <i>AKcryptSK</i>	97
7.9	Secrecy Theorems	99
7.10	2. Parties' strong authentication: non-injective agreement on the session key. The same guarantees also express key distribution, hence their names	103
8	The Kerberos Protocol, Version V	105
8.1	Lemmas about lists, for reasoning about Issues	109
8.2	Lemmas about <i>authKeys</i>	109
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	115
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

9	The Original Otway-Rees Protocol	126
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
10	The Otway-Rees Protocol as Modified by Abadi and Needham	132
10.1	Proofs involving <i>analz</i>	134
10.2	Authenticity properties relating to NA	135
10.3	Authenticity properties relating to NB	136
11	The Otway-Rees Protocol: The Faulty BAN Version	137
11.1	For reasoning about the encrypted portion of messages	138
11.2	Proofs involving <i>analz</i>	139
11.3	Attempting to prove stronger properties	140
12	Bella's version of the Otway-Rees protocol	141
12.1	Proofs involving <i>analz</i>	143
13	The Woo-Lam Protocol	146
14	The Otway-Bull Recursive Authentication Protocol	149
15	The Yahalom Protocol	154
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	161
15.4	Authenticating A to B using the certificate <i>Crypt K (Nonce NB)</i>	162
16	The Yahalom Protocol, Variant 2	163
16.1	Inductive Proofs	164
16.2	Crucial Secrecy Property: Spy Does Not See Key <i>KAB</i>	165
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
17	The Yahalom Protocol: A Flawed Version	169
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
18	Verifying the Needham-Schroeder Public-Key Protocol	180
19	Verifying the Needham-Schroeder-Lowe Public-Key Protocol	183
19.1	Authenticity properties obtained from NS2	184
19.2	Authenticity properties obtained from NS2	184
19.3	Overall guarantee for B	185
20	The TLS Protocol: Transport Layer Security	185
20.1	Protocol Proofs	190
20.2	Inductive proofs about tls	191
20.2.1	Properties of items found in Notes	191
20.2.2	Protocol goal: if B receives $CertVerify$, then A sent it	192
20.2.3	Unicity results for PMS, the pre-master-secret	192
20.3	Secrecy Theorems	193
20.3.1	Protocol goal: $serverK(Na, Nb, M)$ and $clientK(Na, Nb, M)$ remain secure	194
20.3.2	Weakening the Oops conditions for leakage of $clientK$	195
20.3.3	Weakening the Oops conditions for leakage of $serverK$	195
20.3.4	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.	195
20.3.5	Protocol goal: if B receives any message encrypted with $clientK$ then A has sent it	196
20.3.6	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.	196
21	The Certified Electronic Mail Protocol by Abadi et al.	197
21.1	Proving Confidentiality Results	200
21.2	The Guarantees for Sender and Recipient	202
22	Theory of Events for Security Protocols that use smartcards	203
22.1	Function <i>knows</i>	205
22.2	Knowledge of Agents	207
23	Theory of smartcards	210
23.1	Basic properties of $shrK$	212
23.2	Function "knows"	212
23.3	Fresh nonces	214
23.4	Supply fresh nonces for possibility theorems.	214
23.5	Specialized Rewriting for Theorems About <i>analz</i> and <i>Image</i>	214
23.6	Tactics for possibility theorems	215
24	Original Shoup-Rubin protocol	216

25 Bella's modification of the Shoup-Rubin protocol	234
26 Extensions to Standard Theories	253
26.1 Extensions to Theory <i>Set</i>	253
26.2 Extensions to Theory <i>List</i>	253
26.2.1 "remove l x" erase the first element of "l" equal to "x" . . .	253
26.3 Extensions to Theory <i>Message</i>	254
26.3.1 declarations for tactics	254
26.3.2 extract the agent number of an Agent message	254
26.3.3 messages that are pairs	254
26.3.4 well-foundedness of messages	255
26.3.5 lemmas on keysFor	255
26.3.6 lemmas on parts	255
26.3.7 lemmas on synth	256
26.3.8 lemmas on analz	256
26.3.9 lemmas on parts, synth and analz	256
26.3.10 greatest nonce used in a message	257
26.3.11 sets of keys	257
26.3.12 keys a priori necessary for decrypting the messages of G . . .	257
26.3.13 only the keys necessary for G are useful in analz	258
26.4 Extensions to Theory <i>Event</i>	258
26.4.1 general protocol properties	258
26.4.2 lemma on knows	259
26.4.3 knows without initState	259
26.4.4 decomposition of knows into knows' and initState	259
26.4.5 knows' is finite	260
26.4.6 monotonicity of knows	260
26.4.7 maximum knowledge an agent can have includes messages sent to the agent	260
26.4.8 basic facts about <i>knows_max</i>	261
26.4.9 used without initState	261
26.4.10 monotonicity of used	262
26.4.11 lemmas on used and knows	263
26.4.12 a nonce or key in a message cannot equal a fresh nonce or key	263
26.4.13 message of an event	263
27 Decomposition of Analz into two parts	264
27.1 messages that do not contribute to analz	264
27.2 basic facts about <i>pparts</i>	264
27.3 facts about <i>pparts</i> and <i>parts</i>	265
27.4 facts about <i>pparts</i> and <i>analz</i>	266
27.5 messages that contribute to analz	266
27.6 basic facts about <i>kparts</i>	266
27.7 facts about <i>kparts</i> and <i>parts</i>	267
27.8 facts about <i>kparts</i> and <i>analz</i>	268
27.9 analz is <i>pparts</i> + analz of <i>kparts</i>	268

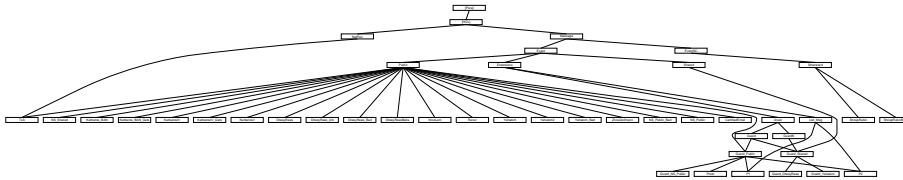
28 Protocol-Independent Confidentiality Theorem on Nonces	268
28.1 basic facts about <i>guard</i>	269
28.2 guarded sets	270
28.3 basic facts about <i>Guard</i>	270
28.4 set obtained by decrypting a message	271
28.5 number of Crypt's in a message	271
28.6 basic facts about <i>crypt_nb</i>	272
28.7 number of Crypt's in a message list	272
28.8 basic facts about <i>cnb</i>	272
28.9 list of kparts	272
28.10 list corresponding to "decrypt"	272
28.11 basic facts about <i>decrypt'</i>	272
28.12 if the analyse of a finite guarded set gives n then it must also gives one of the keys of Ks	273
28.13 if the analyse of a finite guarded set and a (possibly infinite) set of keys gives n then it must also gives Ks	273
28.14 Extensions to Theory <i>Public</i>	273
28.14.1 signature	273
28.14.2 agent associated to a key	273
28.14.3 basic facts about <i>initState</i>	274
28.14.4 sets of private keys	274
28.14.5 sets of good keys	274
28.14.6 greatest nonce used in a trace, 0 if there is no nonce	274
28.14.7 function giving a new nonce	275
28.15 Proofs About Guarded Messages	275
28.15.1 small hack necessary because priK is defined as the inverse of pubK	275
28.15.2 guardedness results	275
28.15.3 regular protocols	276
29 Lists of Messages and Lists of Agents	276
29.1 Implementation of Lists by Messages	276
29.1.1 nil is represented by any message which is not a pair	276
29.1.2 induction principle	276
29.1.3 head	276
29.1.4 tail	276
29.1.5 length	277
29.1.6 membership	277
29.1.7 delete an element	277
29.1.8 concatenation	277
29.1.9 replacement	277
29.1.10 ith element	277
29.1.11 insertion	278
29.1.12 truncation	278
29.2 Agent Lists	278
29.2.1 set of well-formed agent-list messages	278
29.2.2 basic facts about agent lists	278

30 Protocol P1	279
30.1 Protocol Definition	279
30.1.1 offer chaining: B chains his offer for A with the head offer of L for sending it to C	279
30.1.2 agent whose key is used to sign an offer	279
30.1.3 nonce used in an offer	280
30.1.4 next shop	280
30.1.5 anchor of the offer list	280
30.1.6 request event	280
30.1.7 propose event	281
30.1.8 protocol	281
30.1.9 Composition of Traces	281
30.1.10 Valid Offer Lists	282
30.1.11 basic properties of valid	282
30.1.12 offers of an offer list	282
30.1.13 the originator can get the offers	282
30.1.14 list of offers	282
30.1.15 list of agents whose keys are used to sign a list of offers	282
30.1.16 builds a trace from an itinerary	282
30.1.17 there is a trace in which the originator receives a valid answer	283
30.2 properties of protocol P1	283
30.2.1 strong forward integrity: except the last one, no offer can be modified	283
30.2.2 insertion resilience: except at the beginning, no offer can be inserted	283
30.2.3 truncation resilience: only shop i can truncate at offer i	284
30.2.4 declarations for tactics	284
30.2.5 get components of a message	284
30.2.6 general properties of p1	284
30.2.7 private keys are safe	285
30.2.8 general guardedness properties	285
30.2.9 guardedness of messages	285
30.2.10 Nonce uniqueness	286
30.2.11 requests are guarded	286
30.2.12 propositions are guarded	286
30.2.13 data confidentiality: no one other than the originator can decrypt the offers	286
30.2.14 non repudiability: an offer signed by B has been sent by B	287
31 Protocol P2	287
31.1 Protocol Definition	287
31.1.1 offer chaining: B chains his offer for A with the head offer of L for sending it to C	287
31.1.2 agent whose key is used to sign an offer	288
31.1.3 nonce used in an offer	288
31.1.4 next shop	288
31.1.5 anchor of the offer list	288
31.1.6 request event	289
31.1.7 propose event	289

31.1.8	protocol	289
31.1.9	valid offer lists	290
31.1.10	basic properties of valid	290
31.1.11	list of offers	290
31.2	Properties of Protocol P2	290
31.3	strong forward integrity: except the last one, no offer can be modified	290
31.4	insertion resilience: except at the beginning, no offer can be inserted	290
31.5	truncation resilience: only shop i can truncate at offer i	291
31.6	declarations for tactics	291
31.7	get components of a message	291
31.8	general properties of $p2$	291
31.9	private keys are safe	292
31.10	general guardedness properties	292
31.11	guardedness of messages	292
31.12	Nonce uniqueness	293
31.13	requests are guarded	293
31.14	propositions are guarded	293
31.15	data confidentiality: no one other than the originator can decrypt the offers	294
31.16	forward privacy: only the originator can know the identity of the shops	294
31.17	non repudiability: an offer signed by B has been sent by B	294
32	Needham-Schroeder-Lowe Public-Key Protocol	295
32.1	messages used in the protocol	295
32.2	definition of the protocol	295
32.3	declarations for tactics	296
32.4	general properties of nsp	296
32.5	nonce are used only once	296
32.6	guardedness of NA	297
32.7	guardedness of NB	297
32.8	Agents' Authentication	297
33	protocol-independent confidentiality theorem on keys	297
33.1	basic facts about $guardK$	298
33.2	guarded sets	298
33.3	basic facts about $GuardK$	299
33.4	set obtained by decrypting a message	300
33.5	number of $Crypt$'s in a message	300
33.6	basic facts about $crypt_nb$	300
33.7	number of $Crypt$'s in a message list	300
33.8	basic facts about cnb	300
33.9	list of $kparts$	301
33.10	list corresponding to "decrypt"	301
33.11	basic facts about $decrypt'$	301
33.12	Basic properties of $shrK$	302
33.13	Function "knows"	303
33.14	Fresh nonces	303
33.15	Supply fresh nonces for possibility theorems.	303

33.16	Specialized Rewriting for Theorems About <code>analz</code> and <code>Image</code> . . .	304
33.17	Tactics for possibility theorems	304
34	lemmas on guarded messages for protocols with symmetric keys	305
34.1	Extensions to Theory <code>Shared</code>	305
34.1.1	a little abbreviation	305
34.1.2	agent associated to a key	305
34.1.3	basic facts about <code>initState</code>	305
34.1.4	sets of symmetric keys	306
34.1.5	sets of good keys	306
34.2	Proofs About Guarded Messages	306
34.2.1	small hack	306
34.2.2	guardedness results on nonces	306
34.2.3	guardedness results on keys	307
34.2.4	regular protocols	307
35	Otway-Rees Protocol	308
35.1	messages used in the protocol	308
35.2	definition of the protocol	309
35.3	declarations for tactics	309
35.4	general properties of <code>or</code>	309
35.5	<code>or</code> is regular	310
35.6	guardedness of <code>KAB</code>	310
35.7	guardedness of <code>NB</code>	310
36	Yahalom Protocol	310
36.1	messages used in the protocol	310
36.2	definition of the protocol	311
36.3	declarations for tactics	311
36.4	general properties of <code>ya</code>	312
36.5	guardedness of <code>KAB</code>	312
36.6	session keys are not symmetric keys	312
36.7	<code>ya2'</code> implies <code>ya1'</code>	312
36.8	uniqueness of <code>NB</code>	312
36.9	<code>ya3'</code> implies <code>ya2'</code>	313
36.10	<code>ya3'</code> implies <code>ya3</code>	313
36.11	guardedness of <code>NB</code>	313
37	Other Protocol-Independent Results	313
37.1	protocols	313
37.2	substitutions	314
37.3	nonces generated by a rule	315
37.4	traces generated by a protocol	315
37.5	general properties	315
37.6	types	316
37.7	introduction of a fresh guarded nonce	316
37.8	safe keys	317
37.9	guardedness preservation	317
37.10	monotonic <code>keyfun</code>	317
37.11	guardedness theorem	318

37.12	useful properties for guardedness	318
37.13	unicity	318
37.14	Needham-Schroeder-Lowe	319
37.15	general properties	320
37.16	guardedness for NSL	320
37.17	unicity for NSL	320



1 Theory of Agents and Messages for Security Protocols

theory *Message* **imports** *Main* **begin**

lemma [*simp*] : " $A \cup (B \cup A) = B \cup A$ "
 $\langle proof \rangle$

types
 $key = nat$

consts
 $all_symmetric :: bool$ — true if all keys are symmetric
 $invKey :: "key \Rightarrow key"$ — inverse of a symmetric key

specification (*invKey*)
 $invKey$ [*simp*]: " $invKey (invKey K) = K$ "
 $invKey_symmetric: "all_symmetric \rightarrow invKey = id"$
 $\langle proof \rangle$

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
 $agent = Server \mid Friend\ nat \mid Spy$

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

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

syntax
 $"@MTuple" :: "['a, args] \Rightarrow 'a * 'b" \quad (" (2\{ _ , / _ \}) ")$

syntax (*xsymbols*)
 $"@MTuple" :: "['a, args] \Rightarrow 'a * 'b" \quad (" (2\{ _ , / _ \}) ")$

translations
 $"\{ | x, y, z | \}" == "\{ | x, \{ | y, z | \} | \}"$
 $"\{ | x, y | \}" == "MPair\ x\ y"$

constdefs
 $HPair :: "[msg,msg] \Rightarrow msg" \quad (" (4Hash\ _ / _)" \ [0, 1000])$

— 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. $\exists X. \text{Crypt } K \ X \in H$ }"`

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 \in H ==> X \in parts H"`
`| Fst: "{ |X,Y| } \in parts H ==> X \in parts H"`
`| Snd: "{ |X,Y| } \in parts H ==> Y \in parts H"`
`| Body: "Crypt K X \in parts H ==> X \in parts H"`

Monotonicity

lemma `parts_mono`: `"G \subseteq H ==> parts(G) \subseteq parts(H)"`
`<proof>`

Equations hold because constructors are injective.

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

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

lemma `Nonce_Key_image_eq [simp]`: `"(Nonce x \notin 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 \cup H') = keysFor H \cup keysFor H'"`
`<proof>`

lemma `keysFor_UN [simp]`: `"keysFor ($\bigcup_{i \in A} H \ i$) = ($\bigcup_{i \in A} \text{keysFor } (H \ i)$)"`
`<proof>`

Monotonicity

lemma `keysFor_mono`: `"G \subseteq H ==> keysFor(G) \subseteq 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*: " $X \in \text{parts } H \implies \exists Y \in H. X \in \text{parts } \{Y\}$ "
 <proof>

1.2.1 Unions

lemma *parts_Un_subset1*: " $\text{parts}(G) \cup \text{parts}(H) \subseteq \text{parts}(G \cup H)$ "
 <proof>

lemma *parts_Un_subset2*: " $\text{parts}(G \cup H) \subseteq \text{parts}(G) \cup \text{parts}(H)$ "
 <proof>

lemma *parts_Un [simp]*: " $\text{parts}(G \cup H) = \text{parts}(G) \cup \text{parts}(H)$ "
 <proof>

lemma *parts_insert*: " $\text{parts } (\text{insert } X \ H) = \text{parts } \{X\} \cup \text{parts } H$ "
 <proof>

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

lemma *parts_insert2*:
 " $\text{parts } (\text{insert } X \ (\text{insert } Y \ H)) = \text{parts } \{X\} \cup \text{parts } \{Y\} \cup \text{parts } H$ "
 <proof>

lemma *parts_UN_subset1*: " $(\bigcup_{x \in A. \text{parts}(H \ x)) \subseteq \text{parts}(\bigcup_{x \in A. H \ x})$ "
 <proof>

lemma *parts_UN_subset2*: " $\text{parts}(\bigcup_{x \in A. H \ x) \subseteq (\bigcup_{x \in A. \text{parts}(H \ x))$ "
 <proof>

lemma *parts_UN [simp]*: " $\text{parts}(\bigcup_{x \in A. H \ x) = (\bigcup_{x \in A. \text{parts}(H \ x))$ "
 <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*: " $\text{insert } X \ (\text{parts } H) \subseteq \text{parts}(\text{insert } X \ H)$ "
 <proof>

1.2.2 Idempotence and transitivity

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

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

lemma *parts_subset_iff [simp]*: " $(\text{parts } G \subseteq \text{parts } H) = (G \subseteq \text{parts } H)$ "

<proof>

lemma parts_trans: "[| X ∈ parts G; G ⊆ parts H |] ==> X ∈ parts H"
<proof>

Cut

lemma parts_cut:
 "[| Y ∈ parts (insert X G); X ∈ parts H |] ==> Y ∈ parts (G ∪ H)"
<proof>

lemma parts_cut_eq [simp]: "X ∈ 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) ∈ analz H ==>
  analz (insert (Crypt K X) H) =
    insert (Crypt K X) (analz (insert X H))"
<proof>

```

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

```

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

```

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

```

lemma analz_insert_Crypt_subset:
  "analz (insert (Crypt K X) H) ⊆
    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 ∈ analz (analz H) ==> X ∈ analz H"
<proof>

```

```

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

```

```

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

```

```

lemma analz_trans: "[| X ∈ analz G; G ⊆ analz H |] ==> X ∈ analz H"
<proof>

```

Cut; Lemma 2 of Lowe

```

lemma analz_cut: "[| Y ∈ analz (insert X H); X ∈ analz H |] ==> Y ∈ 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 ∈ 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 ⊆ 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: "synth(G) ∪ synth(H) ⊆ synth(G ∪ H)"
<proof>

```

```

lemma synth_insert: "insert X (synth H) ⊆ synth(insert X H)"
<proof>

```

1.4.2 Idempotence and transitivity

```

lemma synth_synthD [dest!]: "X ∈ synth (synth H) ==> X ∈ synth H"
<proof>

```

```

lemma synth_idem: "synth (synth H) = synth H"
<proof>

```

```

lemma synth_subset_iff [simp]: "(synth G ⊆ synth H) = (G ⊆ synth H)"
<proof>

```

```

lemma synth_trans: "[| X ∈ synth G; G ⊆ synth H |] ==> X ∈ synth H"
<proof>

```

Cut; Lemma 2 of Lowe

```

lemma synth_cut: "[| Y ∈ synth (insert X H); X ∈ synth H |] ==> Y ∈ synth H"
<proof>

```

```

lemma Agent_synth [simp]: "Agent A ∈ synth H"
<proof>

```

```

lemma Number_synth [simp]: "Number n ∈ synth H"
<proof>

```

```

lemma Nonce_synth_eq [simp]: "(Nonce N ∈ synth H) = (Nonce N ∈ H)"
<proof>

```

```

lemma Key_synth_eq [simp]: "(Key K ∈ synth H) = (Key K ∈ H)"
<proof>

```

```

lemma Crypt_synth_eq [simp]:
  "Key K ∉ H ==> (Crypt K X ∈ synth H) = (Crypt K X ∈ H)"

```

$\langle proof \rangle$

```
lemma keysFor_synth [simp]:
  "keysFor (synth H) = keysFor H  $\cup$  invKey '{K. Key K  $\in$  H}"
 $\langle proof \rangle$ 
```

1.4.3 Combinations of parts, analz and synth

```
lemma parts_synth [simp]: "parts (synth H) = parts H  $\cup$  synth H"
 $\langle proof \rangle$ 
```

```
lemma analz_analz_Un [simp]: "analz (analz G  $\cup$  H) = analz (G  $\cup$  H)"
 $\langle proof \rangle$ 
```

```
lemma analz_synth_Un [simp]: "analz (synth G  $\cup$  H) = analz (G  $\cup$  H)  $\cup$  synth G"
 $\langle proof \rangle$ 
```

```
lemma analz_synth [simp]: "analz (synth H) = analz H  $\cup$  synth H"
 $\langle proof \rangle$ 
```

1.4.4 For reasoning about the Fake rule in traces

```
lemma parts_insert_subset_Un: "X  $\in$  G  $\implies$  parts(insert X H)  $\subseteq$  parts G  $\cup$  parts H"
 $\langle proof \rangle$ 
```

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  $\in$  synth (analz H)  $\implies$ 
  parts (insert X H)  $\subseteq$  synth (analz H)  $\cup$  parts H"
 $\langle proof \rangle$ 
```

```
lemma Fake_parts_insert_in_Un:
  "[[Z  $\in$  parts (insert X H); X: synth (analz H)]]
 $\implies$  Z  $\in$  synth (analz H)  $\cup$  parts H"
 $\langle proof \rangle$ 
```

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

```
lemma Fake_analz_insert:
  "X  $\in$  synth (analz G)  $\implies$ 
  analz (insert X H)  $\subseteq$  synth (analz G)  $\cup$  analz (G  $\cup$  H)"
 $\langle proof \rangle$ 
```

```
lemma analz_conj_parts [simp]:
  "(X  $\in$  analz H  $\&$  X  $\in$  parts H) = (X  $\in$  analz H)"
 $\langle proof \rangle$ 
```

```
lemma analz_disj_parts [simp]:
  "(X  $\in$  analz H | X  $\in$  parts H) = (X  $\in$  parts H)"
 $\langle proof \rangle$ 
```

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_MPpair [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

⟨ML⟩

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 ∈ set evs --> X ∈ used evs"
 <proof>

lemma *Says_imp_used* [rule_format]: "Says A B X ∈ set evs --> X ∈ 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 $A = \text{Spy}$ 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_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 \neq Spy \rightarrow knows A (Gets A X # evs) = insert X (knows A evs)"
 <proof>

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

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

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

Agents know what they say

lemma *Says_imp_knows* [rule_format]: "Says A B X \in set evs \rightarrow X \in knows A evs"
 <proof>

Agents know what they note

lemma *Notes_imp_knows* [rule_format]: "Notes A X \in set evs \rightarrow X \in knows A evs"
 <proof>

Agents know what they receive

lemma *Gets_imp_knows_agents* [rule_format]:
 "A \neq Spy \rightarrow Gets A X \in set evs \rightarrow X \in 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 \in knows A evs; A \neq Spy |] \Rightarrow EX B.
 Says A B X \in set evs | Gets A X \in set evs | Notes A X \in set evs | X \in 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 \in knows Spy evs |] \Rightarrow EX A B.
 Says A B X \in set evs | Notes A X \in set evs | X \in initState Spy"
 <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) \Rightarrow 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_nil_subset: "used []  $\subseteq$  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]
```

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  $\subseteq$  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) |]
   ==> K  $\in$  keysFor (parts (G  $\cup$  H)) | Key (invKey K)  $\in$  parts H"
<proof>
```

<ML>

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

<ML>

end

```
theory Public imports Event begin
```

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

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

```

axioms

```

privateKey_neq_publicKey [iff]: "privateKey b A  $\neq$  publicKey c A'"

```

```

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

```

2.4 Basic properties of *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 \notin symKeys"
<proof>

lemma *not_symKeys_priK* [iff]: "privateKey b A \notin symKeys"
<proof>

lemma *symKey_neq_priEK*: "K \in symKeys \implies K \neq priEK A"
<proof>

lemma *symKeys_neq_imp_neq*: "(K \in symKeys) \neq (K' \in symKeys) \implies K \neq K'"
<proof>

lemma *symKeys_invKey_iff* [iff]: "(invKey K \in symKeys) = (K \in symKeys)"
<proof>

lemma *analz_symKeys_Decrypt*:
 "[| Crypt K X \in analz H; K \in symKeys; Key K \in analz H |]
 \implies X \in analz H"
<proof>

2.5 "Image" equations that hold for injective functions

lemma *invKey_image_eq* [simp]: "(invKey x \in invKey 'A) = (x \in A)"
<proof>

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

lemma *privateKey_notin_image_publicKey* [simp]: "privateKey b x \notin publicKey
 c ' AA"
<proof>

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

lemma *publicKey_notin_image_privateKey* [simp]: "publicKey b A \notin 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 \Rightarrow key" — long-term shared keys

specification (*shrK*)
inj_shrK: "inj *shrK*"

— No two agents have the same long-term key
 $\langle proof \rangle$

axioms

$sym_shrK \ [iff]: "shrK \ X \in \ symKeys"$ — 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 $invKey_shrK \ [simp]: "invKey \ (shrK \ A) = shrK \ A"$
 $\langle proof \rangle$

lemma $analz_shrK_Decrypt:$
 $"[| \ Crypt \ (shrK \ A) \ X \in \ analz \ H; \ Key(shrK \ A) \in \ analz \ H \ |] \implies X \in \ analz \ H"$
 $\langle proof \rangle$

lemma $analz_Decrypt':$
 $"[| \ Crypt \ K \ X \in \ analz \ H; \ K \in \ symKeys; \ Key \ K \in \ analz \ H \ |] \implies X \in \ analz \ H"$
 $\langle proof \rangle$

lemma $priK_neq_shrK \ [iff]: "shrK \ A \neq \ privateKey \ b \ C"$
 $\langle proof \rangle$

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

lemma $pubK_neq_shrK \ [iff]: "shrK \ A \neq \ publicKey \ b \ C"$
 $\langle proof \rangle$

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

lemma $priEK_noteq_shrK \ [simp]: "priEK \ A \neq \ shrK \ B"$
 $\langle proof \rangle$

lemma $publicKey_notin_image_shrK \ [simp]: "publicKey \ b \ x \notin \ shrK \ ' \ AA"$
 $\langle proof \rangle$

lemma $privateKey_notin_image_shrK \ [simp]: "privateKey \ b \ x \notin \ shrK \ ' \ AA"$
 $\langle proof \rangle$

lemma $shrK_notin_image_publicKey \ [simp]: "shrK \ x \notin \ publicKey \ b \ ' \ AA"$
 $\langle proof \rangle$

lemma $shrK_notin_image_privateKey \ [simp]: "shrK \ x \notin \ invKey \ ' \ publicKey \ b \ ' \ AA"$
 $\langle proof \rangle$

lemma $shrK_image_eq \ [simp]: "(shrK \ x \in \ shrK \ ' \ AA) = (x \in \ AA)"$
 $\langle proof \rangle$

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)} ∪
    (Key ' range pubEK) ∪ (Key ' range pubSK) ∪ (Key ' range shrK)"

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

initState_Spy:
  "initState Spy      =
    (Key ' invKey ' pubEK ' bad) ∪ (Key ' invKey ' pubSK ' bad) ∪
    (Key ' shrK ' bad) ∪
    (Key ' range pubEK) ∪ (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*]:

" $\forall X \in \text{used evs}. \text{parts } \{X\} \subseteq \text{used evs}$ "
 <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!*]:

" $[| \{|X, Y|\} \in \text{used } H;$
 $[| X \in \text{used } H; Y \in \text{used } H |] \implies P |]$
 $\implies P$ "
 <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* \notin *parts* (*initState B*)"

<proof>

lemma *Crypt_notin_used_empty* [simp]: "Crypt K X \notin used []"
 <proof>

lemma *shrK_in_initState* [iff]: "Key (shrK A) \in initState A"
 <proof>

lemma *shrK_in_knows* [iff]: "Key (shrK A) \in knows A evs"
 <proof>

lemma *shrK_in_used* [iff]: "Key (shrK A) \in used evs"
 <proof>

lemma *Key_not_used* [simp]: "Key K \notin used evs \implies K \notin range shrK"
 <proof>

lemma *shrK_neq*: "Key K \notin used evs \implies shrK B \neq K"
 <proof>

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

2.8 Function knows Spy

lemma *not_SignatureE* [elim!]: "b \neq Signature \implies b = Encryption"
 <proof>

Agents see their own private keys!

lemma *priK_in_initState* [iff]: "Key (privateKey b A) \in initState A"
 <proof>

Agents see all public keys!

lemma *publicKey_in_initState* [iff]: "Key (publicKey b A) \in initState B"
 <proof>

All public keys are visible

lemma *spies_pubK* [iff]: "Key (publicKey b A) \in 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 \in bad \implies Key (privateKey b A) \in 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 ∪ C) = Key ' (insert K KK)
 ∪ C"
 <proof>

lemma *Crypt_imp_keysFor :*"[|Crypt K X ∈ H; K ∈ symKeys|] ==> K ∈ 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 ∈ analz (Key'nE ∪ H)) --> (K ∈ nE | Key K ∈ analz H) ==>
   (Key K ∈ analz (Key'nE ∪ H)) = (K ∈ nE | Key K ∈ 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 ==
   ∃ Y. Says A B Y ∈ set evs & X ∈ parts {Y} &
   X ∉ parts (spies (takeWhile (% z. z ≠ Says A B Y) (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 $X \notin \text{parts (knows Spy evs)}$ 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 ∈ ns_shared]] ⇒ A=A' ∧ NA=NA' ∧ B=B' ∧ 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*:

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

<proof>

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

lemma *Spy_not_see_encrypted_key*:
 "Says Server A (Crypt K' {NA, Agent B, Key K, X}) ∈ set evs;
 ∀ NB. Notes Spy {NA, NB, Key K} ∉ set evs;
 A ∉ bad; B ∉ bad; evs ∈ ns_shared
 ⇒ Key K ∉ 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*:
 "Crypt (shrK B) {Key K, Agent A} ∈ parts (spies evs);
 B ∉ bad; evs ∈ ns_shared
 ⇒ ∃ NA. Says Server A
 (Crypt (shrK A) {NA, Agent B, Key K,
 Crypt (shrK B) {Key K, Agent A}})
 ∈ set evs"
<proof>

lemma *A_trusts_NS4_lemma [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) ⇒
 Says B A (Crypt K (Nonce NB)) ∈ set evs"
<proof>

This version no longer assumes that K is secure

lemma *A_trusts_NS4*:
 "Crypt K (Nonce NB) ∈ parts (spies evs);
 Crypt (shrK A) {NA, Agent B, Key K, X} ∈ parts (spies evs);
 ∀ NB. Notes Spy {NA, NB, Key K} ∉ set evs;
 A ∉ bad; B ∉ bad; evs ∈ ns_shared
 ⇒ Says B A (Crypt K (Nonce NB)) ∈ 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 ≤ sesKlife"
⟨proof⟩
```

The authenticator only for one journey

specification (authlife)

```
authlife_LB [iff]: "authlife ≠ 0"
⟨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

```
Issues :: "[agent, agent, msg, event list] => bool"
(" _ Issues _ with _ on _")
"A Issues B with X on evs ==
  ∃ Y. Says A B Y ∈ set evs & X ∈ parts {Y} &
  X ∉ parts (spies (takeWhile (% z. z ≠ 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 ]
        ⇒ Notes Spy {Number Tk, Key K} # evso ∈ 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 ∉ used []; K ∈ symKeys]
        ⇒ ∃ Timestamp. ∃ evs ∈ bankerberos.

```

```

      Says B A (Crypt K (Number Timestamp))
        ∈ 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} ∪ (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_spies:
  "Says S A (Crypt KA {Timestamp, B, K, X}) ∈ set evs
   ⇒ X ∈ parts (spies evs)"

```

<proof>

lemma *Ops_parts_spies*:

"Says Server A (Crypt (shrK A) {Timestamp, B, K, X}) ∈ set evs
 $\implies K \in \text{parts } (\text{spies evs})"$

<proof>

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

lemma *Spy_see_shrK [simp]*:

"evs ∈ bankerberos $\implies (\text{Key } (\text{shrK } A) \in \text{parts } (\text{spies evs})) = (A \in \text{bad})"$

<proof>

lemma *Spy_analz_shrK [simp]*:

"evs ∈ bankerberos $\implies (\text{Key } (\text{shrK } A) \in \text{analz } (\text{spies evs})) = (A \in \text{bad})"$

<proof>

lemma *Spy_see_shrK_D [dest!]*:

"[Key (shrK A) ∈ parts (spies evs);
 evs ∈ bankerberos] $\implies A:\text{bad}"$

<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 ∈ bankerberos]
 $\implies K \notin \text{keysFor } (\text{parts } (\text{spies evs}))"$

<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})
 ∈ set evs; evs ∈ bankerberos]
 $\implies K' = \text{shrK } A \ \& \ K \notin \text{range shrK} \ \&$
 Ticket = (Crypt (shrK B) {Number Tk, Agent A, Key K}) &
 Key K ∉ 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 (insert (Key KK) (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)"
  <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

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

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 ]]
  ⇒ 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 ∈ bankerberos ]]
  ⇒ 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 ∈ bankerberos ]]
⇒ Says B A (Crypt K (Number Ta)) ∈ set evs"
⟨proof⟩

```

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

```

Confidentiality for Alice

```

lemma Confidentiality_A_temporal:
  "[[ Crypt (shrK A) {Number T, Agent B, Key K, X} ∈ parts (spies evs);
    ¬ expiredK T evs;
    A ∉ bad; B ∉ bad; evs ∈ bankerberos
  ]] ⇒ Key K ∉ analz (spies evs)"
⟨proof⟩

```

Confidentiality for Bob

```

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

```

Temporal treatment of authentication

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

Authentication of A to B

lemma *B_authenticates_A_temporal*:

```
"[ Crypt K {Agent A, Number Ta} ∈ parts (spies evs);
  Crypt (shrK B) {Number Tk, Agent A, Key K}
  ∈ parts (spies evs);
  ¬ expiredK Tk 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_temporal*:

```
"[ Crypt K (Number Ta) ∈ parts (spies evs);
  Crypt (shrK A) {Number Tk, Agent B, Key K, X}
  ∈ parts (spies evs);
  ¬ expiredK Tk evs;
  A ∉ bad; B ∉ bad; evs ∈ bankerberos ]
⇒ Says B A (Crypt K (Number Ta)) ∈ 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)) ∈ set 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_authenticates_and_keydist_to_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 ]
⇒ 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} \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) \rightarrow x : used X"
 <proof>

lemma *set_evs_rev*: "set evs = set (rev evs)"
 <proof>

lemma *takeWhile_void [rule_format]*:
 "x \notin set evs \rightarrow 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_knows_Spy*:
 "[Gets A (Crypt KA {Timestamp, B, K, X}) \in set evs; evs \in bankerb_gets
]
 $\implies X \in$ parts (knows Spy evs)"
 <proof>

lemma *Ops_parts_knows_Spy*:
 "Says Server A (Crypt (shrK A) {Timestamp, B, K, X}) \in set evs
 $\implies K \in$ 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 \in bankerb_gets \implies (Key (shrK A) \in parts (knows Spy evs)) = (A
 \in bad)"
 <proof>

lemma *Spy_analz_shrK [simp]*:
 "evs \in bankerb_gets \implies (Key (shrK A) \in analz (knows Spy evs)) = (A
 \in bad)"
 <proof>

lemma *Spy_see_shrK_D [dest!]*:
 "[Key (shrK A) \in parts (knows Spy evs);
 evs \in bankerb_gets] $\implies A$: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 ∉ used evs; K ∈ symKeys; evs ∈ bankerb_gets]
  ⇒ K ∉ keysFor (parts (knows Spy evs))"
<proof>

```

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:
  "[ Says Server A (Crypt K' {Number Tk, Agent B, Key K, Ticket})
    ∈ set evs; evs ∈ bankerb_gets ]
  ⇒ K' = shrK A & K ∉ range shrK &
    Ticket = (Crypt (shrK B) {Number Tk, Agent A, Key K}) &
    Key K ∉ 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 (knows Spy evs);
    A ∉ bad; evs ∈ bankerb_gets ]
  ⇒ 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 (knows Spy evs);
    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"
<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})
    ∈ set evs;
    evs ∈ bankerb_gets ]
  ⇒ (K ∉ range shrK & X = (Crypt (shrK B) {Number Tk, Agent A, Key K}))

```


$| X \in \text{analz } (\text{knows Spy evs})"$

$\langle \text{proof} \rangle$

Reliability guarantees: honest agents act as we expect

lemma *BK3_imp_Gets*:

" $\llbracket \text{Says } A \ B \ \{\text{Ticket}, \text{Crypt } K \ \{\text{Agent } A, \text{Number } Ta\}\} \in \text{set evs};$
 $A \notin \text{bad}; \text{evs} \in \text{bankerb_gets} \rrbracket$
 $\implies \exists \text{Tk. Gets } A \ (\text{Crypt } (\text{shrK } A) \ \{\text{Number } Tk, \text{Agent } B, \text{Key } K, \text{Ticket}\})$
 $\in \text{set evs}"$

$\langle \text{proof} \rangle$

lemma *BK4_imp_Gets*:

" $\llbracket \text{Says } B \ A \ (\text{Crypt } K \ (\text{Number } Ta)) \in \text{set evs};$
 $B \notin \text{bad}; \text{evs} \in \text{bankerb_gets} \rrbracket$
 $\implies \exists \text{Tk. Gets } B \ \{\text{Crypt } (\text{shrK } B) \ \{\text{Number } Tk, \text{Agent } A, \text{Key } K\},$
 $\text{Crypt } K \ \{\text{Agent } A, \text{Number } Ta\}\} \in \text{set evs}"$

$\langle \text{proof} \rangle$

lemma *Gets_A_knows_K*:

" $\llbracket \text{Gets } A \ (\text{Crypt } (\text{shrK } A) \ \{\text{Number } Tk, \text{Agent } B, \text{Key } K, \text{Ticket}\}) \in \text{set evs};$
 $\text{evs} \in \text{bankerb_gets} \rrbracket$
 $\implies \text{Key } K \in \text{analz } (\text{knows } A \ \text{evs})"$

$\langle \text{proof} \rangle$

lemma *Gets_B_knows_K*:

" $\llbracket \text{Gets } B \ \{\text{Crypt } (\text{shrK } B) \ \{\text{Number } Tk, \text{Agent } A, \text{Key } K\},$
 $\text{Crypt } K \ \{\text{Agent } A, \text{Number } Ta\}\} \in \text{set evs};$
 $\text{evs} \in \text{bankerb_gets} \rrbracket$
 $\implies \text{Key } K \in \text{analz } (\text{knows } B \ \text{evs})"$

$\langle \text{proof} \rangle$

Session keys are not used to encrypt other session keys

lemma *analz_image_freshK* [*rule_format* (*no_asm*)]:

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

$\langle \text{proof} \rangle$

lemma *analz_insert_freshK*:

" $\llbracket \text{evs} \in \text{bankerb_gets}; \text{KAB} \notin \text{range shrK} \rrbracket \implies$
 $(\text{Key } K \in \text{analz } (\text{insert } (\text{Key } \text{KAB}) (\text{knows Spy evs}))) =$
 $(K = \text{KAB} \mid \text{Key } K \in \text{analz } (\text{knows Spy evs}))"$

$\langle \text{proof} \rangle$

The session key K uniquely identifies the message

lemma *unique_session_keys*:

" $\llbracket \text{Says Server } A$
 $(\text{Crypt } (\text{shrK } A) \ \{\text{Number } Tk, \text{Agent } B, \text{Key } K, X\}) \in \text{set evs};$
 $\text{Says Server } A'$
 $(\text{Crypt } (\text{shrK } A') \ \{\text{Number } Tk', \text{Agent } B', \text{Key } K, X'\}) \in \text{set evs};$
 $\text{evs} \in \text{bankerb_gets} \rrbracket \implies A=A' \ \& \ Tk=Tk' \ \& \ B=B' \ \& \ X = X'"$

<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*:

"[[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

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

]] \implies Key $K \notin \text{analz} (\text{knows Spy evs})$ "
 <proof>

Confidentiality for Bob

lemma Confidentiality_B:
 "[Crypt (shrK B) {Number Tk, Agent A, Key K}
 $\in \text{parts} (\text{knows Spy evs})$;
 Notes Spy {Number Tk, Key K} $\notin \text{set evs}$;
 $A \notin \text{bad}$; $B \notin \text{bad}$; $\text{evs} \in \text{bankerb_gets}$
] \implies Key $K \notin \text{analz} (\text{knows Spy evs})$ "
 <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]:
 "[$A \notin \text{bad}$; $B \notin \text{bad}$; $\text{evs} \in \text{bankerb_gets}$]
 \implies
 Key $K \notin \text{analz} (\text{knows Spy evs}) \longrightarrow$
 Says Server A (Crypt (shrK A) {Number Tk, Agent B, Key K, X})
 $\in \text{set evs} \longrightarrow$
 Crypt K {Agent A, Number Ta} $\in \text{parts} (\text{knows Spy evs}) \longrightarrow$
 Says A B {X, Crypt K {Agent A, Number Ta}}
 $\in \text{set evs}$ "
 <proof>
lemma lemma_B [rule_format]:
 "[$B \notin \text{bad}$; $\text{evs} \in \text{bankerb_gets}$]
 \implies Key $K \notin \text{analz} (\text{knows Spy evs}) \longrightarrow$
 Says Server A (Crypt (shrK A) {Number Tk, Agent B, Key K, X})
 $\in \text{set evs} \longrightarrow$
 Crypt K (Number Ta) $\in \text{parts} (\text{knows Spy evs}) \longrightarrow$
 Says B A (Crypt K (Number Ta)) $\in \text{set evs}$ "
 <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} $\in \text{parts} (\text{knows Spy evs})$;
 Crypt (shrK B) {Number Tk, Agent A, Key K} $\in \text{parts} (\text{knows Spy evs})$;
 Notes Spy {Number Tk, Key K} $\notin \text{set evs}$;
 $A \notin \text{bad}$; $B \notin \text{bad}$; $\text{evs} \in \text{bankerb_gets}$]
 \implies 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_r:
 "[Crypt K (Number Ta) $\in \text{parts} (\text{knows Spy evs})$;
 Crypt (shrK A) {Number Tk, Agent B, Key K, X} $\in \text{parts} (\text{knows Spy evs})$;
 Notes Spy {Number Tk, Key K} $\notin \text{set evs}$;

$A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{bankerb_gets} \]$
 $\Rightarrow \text{Says } B \ A \ (\text{Crypt } K \ (\text{Number } Ta)) \in \text{set evs}"$
 <proof>

lemma *B_authenticates_A*:
 $"[\text{Crypt } K \ \{\text{Agent } A, \text{Number } Ta\} \in \text{parts (spies evs)};$
 $\text{Crypt } (\text{shrK } B) \ \{\text{Number } Tk, \text{Agent } A, \text{Key } K\} \in \text{parts (spies evs)};$
 $\text{Key } K \notin \text{analz (spies evs)};$
 $A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{bankerb_gets} \]$
 $\Rightarrow \text{Says } A \ B \ \{\text{Crypt } (\text{shrK } B) \ \{\text{Number } Tk, \text{Agent } A, \text{Key } K\},$
 $\text{Crypt } K \ \{\text{Agent } A, \text{Number } Ta\}\} \in \text{set evs}"$
 <proof>

lemma *A_authenticates_B*:
 $"[\text{Crypt } K \ (\text{Number } Ta) \in \text{parts (spies evs)};$
 $\text{Crypt } (\text{shrK } A) \ \{\text{Number } Tk, \text{Agent } B, \text{Key } K, X\} \in \text{parts (spies evs)};$
 $\text{Key } K \notin \text{analz (spies evs)};$
 $A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{bankerb_gets} \]$
 $\Rightarrow \text{Says } B \ A \ (\text{Crypt } K \ (\text{Number } Ta)) \in \text{set evs}"$
 <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 \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{bankerb_gets} \]$
 $\Rightarrow \text{Says Server } A$
 $(\text{Crypt } (\text{shrK } A) \ \{\text{Number } Tk, \text{Agent } B, \text{Key } K,$
 $\text{Crypt } (\text{shrK } B) \ \{\text{Number } Tk, \text{Agent } A, \text{Key } K\}\})$
 $\in \text{set evs} \longrightarrow$
 $\text{Key } K \in \text{analz (knows Spy evs)} \longrightarrow \text{expiredK } Tk \ \text{evs}"$
 <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*:
 $"[\text{Says Server } A$
 $(\text{Crypt } K' \ \{\text{Number } T, \text{Agent } B, \text{Key } K, X\}) \in \text{set evs};$
 $\neg \text{expiredK } T \ \text{evs};$
 $A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{bankerb_gets}$
 $] \Rightarrow \text{Key } K \notin \text{analz (knows Spy evs)}"$
 <proof>

Confidentiality for Alice

lemma *Confidentiality_A_temporal*:
 $"[\text{Crypt } (\text{shrK } A) \ \{\text{Number } T, \text{Agent } B, \text{Key } K, X\} \in \text{parts (knows Spy evs)};$
 $\neg \text{expiredK } T \ \text{evs};$

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

$$A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{bankerb_gets}$$

$$\implies \text{Key } K \notin \text{analz } (\text{knows Spy evs})"$$

$$\langle \text{proof} \rangle$$

Confidentiality for Bob

lemma Confidentiality_B_temporal:

$$\begin{aligned}
 & \text{"[Crypt (shrK B) \{Number Tk, Agent A, Key K\}} \\
 & \quad \in \text{parts (knows Spy evs);} \\
 & \quad \neg \text{expiredK Tk evs;} \\
 & \quad A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{bankerb_gets} \\
 & \text{]} \implies \text{Key } K \notin \text{analz (knows Spy evs)}" \\
 & \langle \text{proof} \rangle
 \end{aligned}$$

Temporal treatment of authentication

Authentication of A to B

lemma B_authenticates_A_temporal:

$$\begin{aligned}
 & \text{"[Crypt K \{Agent A, Number Ta\} \in \text{parts (knows Spy evs);} \\
 & \quad \text{Crypt (shrK B) \{Number Tk, Agent A, Key K\}} \\
 & \quad \in \text{parts (knows Spy evs);} \\
 & \quad \neg \text{expiredK Tk evs;} \\
 & \quad A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{bankerb_gets} \text{]} \\
 & \implies \text{Says A B \{Crypt (shrK B) \{Number Tk, Agent A, Key K\},} \\
 & \quad \text{Crypt K \{Agent A, Number Ta\}\} \in \text{set evs}" \\
 & \langle \text{proof} \rangle
 \end{aligned}$$

Authentication of B to A

lemma A_authenticates_B_temporal:

$$\begin{aligned}
 & \text{"[Crypt K (Number Ta) \in \text{parts (knows Spy evs);} \\
 & \quad \text{Crypt (shrK A) \{Number Tk, Agent B, Key K, X\}} \\
 & \quad \in \text{parts (knows Spy evs);} \\
 & \quad \neg \text{expiredK Tk evs;} \\
 & \quad A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{bankerb_gets} \text{]} \\
 & \implies \text{Says B A (Crypt K (Number Ta)) \in \text{set evs}" \\
 & \langle \text{proof} \rangle
 \end{aligned}$$

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

lemma B_authenticates_and_keydist_to_A:

$$\begin{aligned}
 & \text{"[Gets B \{Crypt (shrK B) \{Number Tk, Agent A, Key K\},} \\
 & \quad \text{Crypt K \{Agent A, Number Ta\}\} \in \text{set evs;} \\
 & \quad \text{Key K} \notin \text{analz (spies evs);} \\
 & \quad A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{bankerb_gets} \text{]} \\
 & \implies \text{Says A B \{Crypt (shrK B) \{Number Tk, Agent A, Key K\},} \\
 & \quad \text{Crypt K \{Agent A, Number Ta\}\} \in \text{set evs} \\
 & \quad \wedge \text{Key K} \in \text{analz (knows A evs)}" \\
 & \langle \text{proof} \rangle
 \end{aligned}$$

lemma A_authenticates_and_keydist_to_B:

$$\begin{aligned}
 & \text{"[Gets A (Crypt (shrK A) \{Number Tk, Agent B, Key K, Ticket\}) \in \text{set} \\
 & \text{evs;} \\
 & \quad \text{Gets A (Crypt K (Number Ta)) \in \text{set evs;} \\
 & \quad \text{Key K} \in \text{analz (knows A evs)}" \\
 & \langle \text{proof} \rangle
 \end{aligned}$$

```

      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}))
                      })"

```

```

Issues :: "[agent, agent, msg, event list] => bool"
(" _ Issues _ with _ on _")
"A Issues B with X on evs ==
  ∃ Y. Says A B Y ∈ set evs & X ∈ parts {Y} &
  X ∉ parts (spies (takeWhile (% z. z ≠ 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  $\notin$  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  $\Rightarrow$  nat" where
    "CT == length"

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

abbreviation
  expiredSK :: "[nat, event list]  $\Rightarrow$  bool" where
    "expiredSK Ts evs == servKlife + Ts < CT evs"

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

abbreviation
  valid :: "[nat, nat]  $\Rightarrow$  bool" ("valid _ wrt _") where
    "valid T1 wrt T2 == T1  $\leq$  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: "[[ evsf  $\in$  kerbIV; X  $\in$  synth (analz(spies evsf)) ]
       $\implies$  Says Spy B X # evsf  $\in$  kerbIV"

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

    / K2: "[[ evs2  $\in$  kerbIV; Key authK  $\notin$  used evs2; authK  $\in$  symKeys;
      Says A' Kas {Agent A, Agent Tgs, Number T1}  $\in$  set evs2 ]
       $\implies$  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)}}) # evs2  $\in$  kerbIV"

    / K3: "[[ evs3  $\in$  kerbIV;
      Says A Kas {Agent A, Agent Tgs, Number T1}  $\in$  set evs3;
      Says Kas' A (Crypt (shrK A) {Key authK, Agent Tgs, Number Ta,
        authTicket})  $\in$  set evs3;
      valid Ta wrt T1
      ]
       $\implies$  Says A Tgs {authTicket,
        (Crypt authK {Agent A, Number (CT evs3)}),
        Agent B} # evs3  $\in$  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})
 \in set evs ;
 evs \in kerbIV] \implies authK \notin range shrK & authK \in symKeys"

<proof>

–For reasoning about the encrypted portion of message K5–

lemma *K5_msg_in_parts_spies*:

"Says Tgs' A (Crypt authK {servK, Agent B, Ts, servTicket})
 \in set evs \implies servTicket \in parts (spies evs)"

<proof>

lemma *Ops_range_spies2*:

```

    "[ Says Tgs A (Crypt authK {Key servK, Agent B, Ts, servTicket})
      ∈ set evs ;
      evs ∈ kerbIV ] ⇒ servK ∉ range shrK & servK ∈ symKeys"
  <proof>

```

```

lemma Says_ticket_parts:
  "Says S A (Crypt K {SesKey, B, TimeStamp, Ticket}) ∈ set evs
   ⇒ Ticket ∈ parts (spies evs)"
  <proof>

```

```

lemma Spy_see_shrK [simp]:
  "evs ∈ kerbIV ⇒ (Key (shrK A) ∈ parts (spies evs)) = (A ∈ bad)"
  <proof>

```

```

lemma Spy_analz_shrK [simp]:
  "evs ∈ kerbIV ⇒ (Key (shrK A) ∈ analz (spies evs)) = (A ∈ bad)"
  <proof>

```

```

lemma Spy_see_shrK_D [dest!]:
  "[ Key (shrK A) ∈ parts (spies evs); evs ∈ kerbIV ] ⇒ 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 ∈ kerbIV]
   ⇒ K ∉ keysFor (parts (spies evs))"
  <proof>

```

```

lemma new_keys_not_analz:
  "[evs ∈ kerbIV; K ∈ symKeys; Key K ∉ used evs]
   ⇒ K ∉ keysFor (analz (spies evs))"
  <proof>

```

6.4 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 \notin set evs \longrightarrow takeWhile ($\lambda z. z \neq x$) 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})
 \in set evs;
 evs \in kerbIV] \implies
 K = shrK A & Peer = Tgs &
 authK \notin range shrK & authK \in authKeys evs & authK \in symKeys &
 authTicket = (Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta})
 &
 Key authK \notin 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}
 \in parts (spies evs); Tgs_B \notin bad;
 evs \in kerbIV]
 \implies SesKey \notin range shrK"
 <proof>

lemma *authTicket_authentic*:
 "[Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}
 \in parts (spies evs);
 evs \in kerbIV]
 \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"
 <proof>

lemma *authTicket_crypt_authK*:
 "[Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}
 \in parts (spies evs);
 evs \in kerbIV]
 \implies authK \in authKeys evs"
 <proof>

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

lemma Says_Tgs_message_form:

```
"[[ Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
   ∈ set evs;
   evs ∈ kerbIV ]]
⇒ 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})
&
   Key servK ∉ used (before
     Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
       on evs) &
   Ts = CT(before
     Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
       on evs) "
```

⟨proof⟩

lemma authTicket_form:

```
"[[ Crypt (shrK A) {Key authK, Agent Tgs, Ta, authTicket}
   ∈ parts (spies evs);
   A ∉ bad;
   evs ∈ kerbIV ]]
⇒ 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 ]]
⇒ 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:

```
"[[ Says Kas' A (Crypt (shrK A)
   {Key authK, Agent Tgs, Ta, authTicket}) ∈ 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.
    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;

```


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

```

    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 ∉ used evs; evs ∈ kerbIV ]]
  ⇒ ¬ 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]
 ⇒ ¬ 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]
 ⇒ ¬ AKcryptSK SK K evs"
 <proof>

Long term keys are not issued as servKeys

lemma *shrK_not_AKcryptSK*:
 "evs ∈ kerbIV ⇒ ¬ 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]
 ⇒ ¬ AKcryptSK authK' servK evs"
 <proof>

Equivalently

lemma *not_different_AKcryptSK*:
 "[AKcryptSK authK servK evs;
 authK' ≠ authK; evs ∈ kerbIV]
 ⇒ ¬ AKcryptSK authK' servK evs ∧ servK ∈ symKeys"
 <proof>

lemma *AKcryptSK_not_AKcryptSK*:
 "[AKcryptSK authK servK evs; evs ∈ kerbIV]
 ⇒ ¬ 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]]
 ⇒ Key K' ∈ analz (insert (Key K) (spies evs))"
 <proof>

lemma *authKeys_are_not_AKcryptSK*:
 "[[K ∈ authKeys evs Un range shrK; evs ∈ kerbIV]]
 ⇒ ∀ SK. ¬ AKcryptSK SK K evs ∧ K ∈ symKeys"
 <proof>

lemma *not_authKeys_not_AKcryptSK*:
 "[[K ∉ authKeys evs;
 K ∉ range shrK; evs ∈ kerbIV]]
 ⇒ ∀ SK. ¬ AKcryptSK K SK evs"
 <proof>

6.10 Secrecy Theorems

For the Oops2 case of the next theorem

lemma *Oops2_not_AKcryptSK*:
 "[[evs ∈ kerbIV;
 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. [simplified by LCP]

lemma *Key_analz_image_Key [rule_format (no_asm)]*:
 "evs ∈ kerbIV ⇒
 (∀ 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; 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; 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 ]
  ⇒ (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 ]
  ⇒ (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 ]
  ⇒ 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 ]
  ⇒ 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 ]
  ⇒ 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 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

```


6.11 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, 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);
    ¬ expiredSK Ts 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>

```

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);

```

$\text{Key authK} \notin \text{analz}(\text{spies evs}); \text{Key servK} \notin \text{analz}(\text{spies evs});$
 $A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{kerbIV} \parallel$
 $\implies \text{Says } B \ A \ (\text{Crypt servK} (\text{Number } T3)) \in \text{set evs}"$
 $\langle \text{proof} \rangle$

lemma *A_authenticates_B_r:*

$\parallel \text{Crypt servK} (\text{Number } T3) \in \text{parts}(\text{spies evs});$
 $\text{Crypt authK} \{ \text{Key servK}, \text{Agent } B, \text{Number } Ts, \text{servTicket} \}$
 $\in \text{parts}(\text{spies evs});$
 $\text{Crypt} (\text{shrK } A) \{ \text{Key authK}, \text{Agent } Tgs, \text{Number } Ta, \text{authTicket} \}$
 $\in \text{parts}(\text{spies evs});$
 $\neg \text{expiredAK } Ta \text{ evs}; \neg \text{expiredSK } Ts \text{ evs};$
 $A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{kerbIV} \parallel$
 $\implies \text{Says } B \ A \ (\text{Crypt servK} (\text{Number } T3)) \in \text{set evs}"$
 $\langle \text{proof} \rangle$

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:*

$\parallel \text{Says Kas } A \ (\text{Crypt} (\text{shrK } A) \{ \text{Key authK}, \text{Peer}, Ta, \text{authTicket} \}) \in \text{set evs};$
 $\text{evs} \in \text{kerbIV} \parallel$
 $\implies \text{Kas Issues } A \text{ with } (\text{Crypt} (\text{shrK } A) \{ \text{Key authK}, \text{Peer}, Ta, \text{authTicket} \})$
 $\text{on evs}"$
 $\langle \text{proof} \rangle$

lemma *A_authenticates_and_keydist_to_Kas:*

$\parallel \text{Crypt} (\text{shrK } A) \{ \text{Key authK}, \text{Peer}, Ta, \text{authTicket} \} \in \text{parts}(\text{spies evs});$
 $A \notin \text{bad}; \text{evs} \in \text{kerbIV} \parallel$
 $\implies \text{Kas Issues } A \text{ with } (\text{Crypt} (\text{shrK } A) \{ \text{Key authK}, \text{Peer}, Ta, \text{authTicket} \})$
 $\text{on evs}"$
 $\langle \text{proof} \rangle$

lemma *honest_never_says_newer_timestamp_in_auth:*

$\parallel (CT \text{ evs}) \leq T; A \notin \text{bad}; \text{Number } T \in \text{parts} \{X\}; \text{evs} \in \text{kerbIV} \parallel$
 $\implies \forall B \ Y. \text{Says } A \ B \ \{Y, X\} \notin \text{set evs}"$
 $\langle \text{proof} \rangle$

lemma *honest_never_says_current_timestamp_in_auth:*

$\parallel (CT \text{ evs}) = T; \text{Number } T \in \text{parts} \{X\}; \text{evs} \in \text{kerbIV} \parallel$
 $\implies \forall A \ B \ Y. A \notin \text{bad} \longrightarrow \text{Says } A \ B \ \{Y, X\} \notin \text{set evs}"$
 $\langle \text{proof} \rangle$

lemma *A_trusts_secure_authenticator:*

$\parallel \text{Crypt } K \ \{ \text{Agent } A, \text{Number } T \} \in \text{parts}(\text{spies evs});$
 $\text{Key } K \notin \text{analz}(\text{spies evs}); \text{evs} \in \text{kerbIV} \parallel$
 $\implies \exists B \ X. \text{Says } A \ Tgs \ \{X, \text{Crypt } K \ \{ \text{Agent } A, \text{Number } T \}, \text{Agent } B \} \in \text{set evs}$
 \vee

6.12 Key distribution guarantees An agent knows a session key if he used it to issue a cipher. These guarantees also

$\langle \text{proof} \rangle$ Says A B $\{X, \text{Crypt } K \{ \text{Agent } A, \text{Number } T \} \} \in \text{set evs}$

lemma A_Issues_Tgs:

" \llbracket Says A Tgs $\{ \text{authTicket}, \text{Crypt authK } \{ \text{Agent } A, \text{Number } T2 \}, \text{Agent } B \}$
 $\in \text{set evs};$
 $\text{Key authK} \notin \text{analz} (\text{spies evs});$
 $A \notin \text{bad}; \text{evs} \in \text{kerbIV} \rrbracket$
 $\implies A \text{ Issues Tgs with } (\text{Crypt authK } \{ \text{Agent } A, \text{Number } T2 \}) \text{ on evs}$ "
 $\langle \text{proof} \rangle$

lemma Tgs_authenticates_and_keydist_to_A:

" \llbracket Crypt authK $\{ \text{Agent } A, \text{Number } T2 \} \in \text{parts} (\text{spies evs});$
 $\text{Crypt} (\text{shrK Tgs}) \{ \text{Agent } A, \text{Agent Tgs}, \text{Key authK}, \text{Number Ta} \}$
 $\in \text{parts} (\text{spies evs});$
 $\text{Key authK} \notin \text{analz} (\text{spies evs});$
 $A \notin \text{bad}; \text{evs} \in \text{kerbIV} \rrbracket$
 $\implies A \text{ Issues Tgs with } (\text{Crypt authK } \{ \text{Agent } A, \text{Number } T2 \}) \text{ on evs}$ "
 $\langle \text{proof} \rangle$

lemma Tgs_Issues_A:

" \llbracket Says Tgs A $(\text{Crypt authK } \{ \text{Key servK}, \text{Agent } B, \text{Number Ts}, \text{servTicket} \})$
 $\in \text{set evs};$
 $\text{Key authK} \notin \text{analz} (\text{spies evs}); \text{evs} \in \text{kerbIV} \rrbracket$
 $\implies \text{Tgs Issues A with}$
 $(\text{Crypt authK } \{ \text{Key servK}, \text{Agent } B, \text{Number Ts}, \text{servTicket} \}) \text{ on evs}$ "
 $\langle \text{proof} \rangle$

lemma A_authenticates_and_keydist_to_Tgs:

" $\llbracket \text{Crypt authK } \{ \text{Key servK}, \text{Agent } B, \text{Number Ts}, \text{servTicket} \} \in \text{parts} (\text{spies evs});$
 $\text{Key authK} \notin \text{analz} (\text{spies evs}); B \neq \text{Tgs}; \text{evs} \in \text{kerbIV} \rrbracket$
 $\implies \exists A. \text{Tgs Issues A with}$
 $(\text{Crypt authK } \{ \text{Key servK}, \text{Agent } B, \text{Number Ts}, \text{servTicket} \}) \text{ on evs}$ "
 $\langle \text{proof} \rangle$

lemma B_Issues_A:

" \llbracket Says B A $(\text{Crypt servK} (\text{Number T3})) \in \text{set evs};$
 $\text{Key servK} \notin \text{analz} (\text{spies evs});$
 $A \notin \text{bad}; B \notin \text{bad}; B \neq \text{Tgs}; \text{evs} \in \text{kerbIV} \rrbracket$
 $\implies B \text{ Issues A with } (\text{Crypt servK} (\text{Number T3})) \text{ on evs}$ "
 $\langle \text{proof} \rangle$

lemma B_Issues_A_r:

" \llbracket Says B A $(\text{Crypt servK} (\text{Number T3})) \in \text{set evs};$
 $\text{Crypt} (\text{shrK B}) \{ \text{Agent } A, \text{Agent } B, \text{Key servK}, \text{Number Ts} \}$
 $\in \text{parts} (\text{spies evs});$
 $\text{Crypt authK } \{ \text{Key servK}, \text{Agent } B, \text{Number Ts}, \text{servTicket} \}$
 $\in \text{parts} (\text{spies evs});$
 $\text{Crypt} (\text{shrK A}) \{ \text{Key authK}, \text{Agent Tgs}, \text{Number Ta}, \text{authTicket} \}$
 $\in \text{parts} (\text{spies evs});$
 $\neg \text{expiredSK Ts evs}; \neg \text{expiredAK Ta evs};$
 \rrbracket

$A \notin \text{bad}; B \notin \text{bad}; B \neq \text{Tgs}; \text{evs} \in \text{kerbIV} \parallel$
 $\implies B \text{ Issues } A \text{ with } (\text{Crypt servK (Number T3)}) \text{ on evs''}$
 $\langle \text{proof} \rangle$

lemma *u_B_Issues_A_r*:

$\parallel \text{ Says } B \ A \ (\text{Crypt servK (Number T3)}) \in \text{set evs};$
 $\text{Crypt (shrK B)} \ \{ \text{Agent A, Agent B, Key servK, Number Ts} \}$
 $\in \text{parts (spies evs)};$
 $\neg \text{ expiredSK Ts evs};$
 $A \notin \text{bad}; B \notin \text{bad}; B \neq \text{Tgs}; \text{evs} \in \text{kerbIV} \parallel$
 $\implies B \text{ Issues } A \text{ with } (\text{Crypt servK (Number T3)}) \text{ on evs''}$
 $\langle \text{proof} \rangle$

lemma *A_authenticates_and_keydist_to_B*:

$\parallel \text{ Crypt servK (Number T3)} \in \text{parts (spies evs)};$
 $\text{Crypt authK } \{ \text{Key servK, Agent B, Number Ts, servTicket} \}$
 $\in \text{parts (spies evs)};$
 $\text{Crypt (shrK A)} \ \{ \text{Key authK, Agent Tgs, Number Ta, authTicket} \}$
 $\in \text{parts (spies evs)};$
 $\text{Key authK} \notin \text{analz (spies evs)}; \text{Key servK} \notin \text{analz (spies evs)};$
 $A \notin \text{bad}; B \notin \text{bad}; B \neq \text{Tgs}; \text{evs} \in \text{kerbIV} \parallel$
 $\implies B \text{ Issues } A \text{ with } (\text{Crypt servK (Number T3)}) \text{ on evs''}$
 $\langle \text{proof} \rangle$

lemma *A_authenticates_and_keydist_to_B_r*:

$\parallel \text{ Crypt servK (Number T3)} \in \text{parts (spies evs)};$
 $\text{Crypt authK } \{ \text{Key servK, Agent B, Number Ts, servTicket} \}$
 $\in \text{parts (spies evs)};$
 $\text{Crypt (shrK A)} \ \{ \text{Key authK, Agent Tgs, Number Ta, authTicket} \}$
 $\in \text{parts (spies evs)};$
 $\neg \text{ expiredAK Ta evs}; \neg \text{ expiredSK Ts evs};$
 $A \notin \text{bad}; B \notin \text{bad}; B \neq \text{Tgs}; \text{evs} \in \text{kerbIV} \parallel$
 $\implies B \text{ Issues } A \text{ with } (\text{Crypt servK (Number T3)}) \text{ on evs''}$
 $\langle \text{proof} \rangle$

lemma *A_Issues_B*:

$\parallel \text{ Says } A \ B \ \{ \text{servTicket, Crypt servK } \{ \text{Agent A, Number T3} \} \}$
 $\in \text{set evs};$
 $\text{Key servK} \notin \text{analz (spies evs)};$
 $B \neq \text{Tgs}; A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{kerbIV} \parallel$
 $\implies A \text{ Issues } B \text{ with } (\text{Crypt servK } \{ \text{Agent A, Number T3} \}) \text{ on evs''}$
 $\langle \text{proof} \rangle$

lemma *A_Issues_B_r*:

$\parallel \text{ Says } A \ B \ \{ \text{servTicket, Crypt servK } \{ \text{Agent A, Number T3} \} \}$
 $\in \text{set evs};$
 $\text{Crypt (shrK A)} \ \{ \text{Key authK, Agent Tgs, Number Ta, authTicket} \}$
 $\in \text{parts (spies evs)};$
 $\text{Crypt authK } \{ \text{Key servK, Agent B, Number Ts, servTicket} \}$
 $\in \text{parts (spies evs)};$
 $\neg \text{ expiredAK Ta evs}; \neg \text{ expiredSK Ts evs};$
 $B \neq \text{Tgs}; A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{kerbIV} \parallel$
 $\implies A \text{ Issues } B \text{ with } (\text{Crypt servK } \{ \text{Agent A, Number T3} \}) \text{ on evs''}$

<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 \Rightarrow key set"
authKeys *evs* == {*authK*. \exists *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})
 }) \in set *evs*}"

Unique :: "[event, event list] \Rightarrow 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 \Rightarrow 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"

```

```

/ Oops1: "[ 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"

```

```

/ Oops2: "[ 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"

```

$\langle proof \rangle$

lemma Gets_imp_knows_Spy:

" $\llbracket \text{Gets } B \ X \in \text{set } \text{evs}; \text{evs} \in \text{kerbIV_gets} \rrbracket \implies X \in \text{knows Spy evs}$ "
 $\langle proof \rangle$

declare Gets_imp_knows_Spy [THEN parts.Inj, dest]

lemma Gets_imp_knows:

" $\llbracket \text{Gets } B \ X \in \text{set } \text{evs}; \text{evs} \in \text{kerbIV_gets} \rrbracket \implies X \in \text{knows } B \text{ evs}$ "
 $\langle proof \rangle$

7.2 Lemmas about authKeys

lemma authKeys_empty: "authKeys [] = {}"

$\langle proof \rangle$

lemma authKeys_not_insert:

" $(\forall A \ Ta \ akey \ Peer.$
 $ev \neq \text{Says Kas } A \ (\text{Crypt } (\text{shrK } A) \ \{\{akey, \text{Agent Peer}, Ta,$
 $(\text{Crypt } (\text{shrK } Peer) \ \{\{Agent A, \text{Agent Peer}, akey, Ta\}\})\})$
 $\implies \text{authKeys } (ev \ \# \ \text{evs}) = \text{authKeys evs}$ "
 $\langle proof \rangle$

lemma authKeys_insert:

"authKeys
 $(\text{Says Kas } A \ (\text{Crypt } (\text{shrK } A) \ \{\{Key K, \text{Agent Peer}, \text{Number } Ta,$
 $(\text{Crypt } (\text{shrK } Peer) \ \{\{Agent A, \text{Agent Peer}, Key K, \text{Number } Ta\}\})\}) \ \# \ \text{evs})$
 $= \text{insert } K \ (\text{authKeys evs})$ "
 $\langle proof \rangle$

lemma authKeys_simp:

" $K \in \text{authKeys}$
 $(\text{Says Kas } A \ (\text{Crypt } (\text{shrK } A) \ \{\{Key K', \text{Agent Peer}, \text{Number } Ta,$
 $(\text{Crypt } (\text{shrK } Peer) \ \{\{Agent A, \text{Agent Peer}, Key K', \text{Number } Ta\}\})\}) \ \# \ \text{evs})$
 $\implies K = K' \mid K \in \text{authKeys evs}$ "
 $\langle proof \rangle$

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
 $\implies K \in \text{authKeys evs}$ "
 $\langle proof \rangle$

lemma authKeys_used: " $K \in \text{authKeys evs} \implies \text{Key } K \in \text{used evs}$ "

$\langle proof \rangle$

7.3 Forwarding Lemmas

lemma Says_ticket_parts:

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

lemma *Gets_ticket_parts*:
 "[[Gets A (Crypt K {SesKey, Peer, Ta, Ticket}) ∈ set evs; evs ∈ kerbIV_gets
]]
 ⇒ Ticket ∈ parts (spies evs)"
 <proof>

lemma *Ops_range_spies1*:
 "[[Says Kas A (Crypt KeyA {Key authK, Peer, Ta, authTicket})
 ∈ set evs ;
 evs ∈ kerbIV_gets]] ⇒ authK ∉ range shrK & authK ∈ symKeys"
 <proof>

lemma *Ops_range_spies2*:
 "[[Says Tgs A (Crypt authK {Key servK, Agent B, Ts, servTicket})
 ∈ set evs ;
 evs ∈ kerbIV_gets]] ⇒ servK ∉ range shrK & servK ∈ symKeys"
 <proof>

lemma *Spy_see_shrK [simp]*:
 "evs ∈ kerbIV_gets ⇒ (Key (shrK A) ∈ parts (spies evs)) = (A ∈ bad)"
 <proof>

lemma *Spy_analz_shrK [simp]*:
 "evs ∈ kerbIV_gets ⇒ (Key (shrK A) ∈ analz (spies evs)) = (A ∈ bad)"
 <proof>

lemma *Spy_see_shrK_D [dest!]*:
 "[[Key (shrK A) ∈ parts (spies evs); evs ∈ kerbIV_gets]] ⇒ 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 ∈ kerbIV_gets]
 ⇒ K ∉ keysFor (parts (spies evs))"
 <proof>

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}

```

$\in \text{parts } (\text{spies evs});$
 $\text{Key authK} \notin \text{analz } (\text{spies evs});$
 $B \neq \text{Tgs};$
 $\text{evs} \in \text{kerbIV_gets} \parallel$
 $\implies \exists A. \text{ Says Tgs A } (\text{Crypt authK } \{\{\text{Key servK, Agent B, Number Ts, servTicket}\}\})$
 $\in \text{set evs}"$
 $\langle \text{proof} \rangle$

Authenticity of servK for B

lemma *servTicket_authentic_Tgs*:
 $"\parallel \text{Crypt } (\text{shrK B}) \{\{\text{Agent A, Agent B, Key servK, Number Ts}\}\}$
 $\in \text{parts } (\text{spies evs}); B \neq \text{Tgs}; B \notin \text{bad};$
 $\text{evs} \in \text{kerbIV_gets} \parallel$
 $\implies \exists \text{authK.}$
 $\text{ Says Tgs A } (\text{Crypt authK } \{\{\text{Key servK, Agent B, Number Ts,}$
 $\text{Crypt } (\text{shrK B}) \{\{\text{Agent A, Agent B, Key servK, Number Ts}\}\}\})$
 $\in \text{set evs}"$
 $\langle \text{proof} \rangle$

Anticipated here from next subsection

lemma *K4_imp_K2*:
 $"\parallel \text{ Says Tgs A } (\text{Crypt authK } \{\{\text{Key servK, Agent B, Number Ts, servTicket}\}\})$
 $\in \text{set evs}; \text{ evs} \in \text{kerbIV_gets} \parallel$
 $\implies \exists \text{Ta. Says Kas A}$
 $(\text{Crypt } (\text{shrK A})$
 $\{\{\text{Key authK, Agent Tgs, Number Ta,}$
 $\text{Crypt } (\text{shrK Tgs}) \{\{\text{Agent A, Agent Tgs, Key authK, Number Ta}\}\}\})$
 $\in \text{set evs}"$
 $\langle \text{proof} \rangle$

Anticipated here from next subsection

lemma *u_K4_imp_K2*:
 $"\parallel \text{ Says Tgs A } (\text{Crypt authK } \{\{\text{Key servK, Agent B, Number Ts, servTicket}\}\})$
 $\in \text{set evs}; \text{ evs} \in \text{kerbIV_gets} \parallel$
 $\implies \exists \text{Ta. (Says Kas A } (\text{Crypt } (\text{shrK A}) \{\{\text{Key authK, Agent Tgs, Number Ta,}$
 $\text{Crypt } (\text{shrK Tgs}) \{\{\text{Agent A, Agent Tgs, Key authK, Number Ta}\}\}\})$
 $\in \text{set evs}$
 $\& \text{ servKlife} + \text{Ts} \leq \text{authKlife} + \text{Ta})"$
 $\langle \text{proof} \rangle$

lemma *servTicket_authentic_Kas*:
 $"\parallel \text{Crypt } (\text{shrK B}) \{\{\text{Agent A, Agent B, Key servK, Number Ts}\}\}$
 $\in \text{parts } (\text{spies evs}); B \neq \text{Tgs}; B \notin \text{bad};$
 $\text{evs} \in \text{kerbIV_gets} \parallel$
 $\implies \exists \text{authK Ta.}$
 Says Kas A
 $(\text{Crypt } (\text{shrK A}) \{\{\text{Key authK, Agent Tgs, Number Ta,}$
 $\text{Crypt } (\text{shrK Tgs}) \{\{\text{Agent A, Agent Tgs, Key authK, Number Ta}\}\}\})$
 $\in \text{set evs}"$
 $\langle \text{proof} \rangle$

lemma *u_servTicket_authentic_Kas*:
 $"\parallel \text{Crypt } (\text{shrK B}) \{\{\text{Agent A, Agent B, Key servK, Number Ts}\}\}$

7.6 Reliability: friendly agents send something if something else happened 95

```

    ∈ 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;
```

$A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{kerbIV_gets} \parallel$
 $\implies \text{Key servK} \notin \text{analz} (\text{spies evs})"$
 $\langle \text{proof} \rangle$

In the real world Tgs CAN check what Kas sends!

lemma Confidentiality_Tgs_bis:

$"\parallel \text{Says Kas A}$
 $\quad (\text{Crypt Ka } \{\text{Key authK, Agent Tgs, Number Ta, authTicket}\})$
 $\quad \in \text{set evs};$
 Says Tgs A
 $\quad (\text{Crypt authK } \{\text{Key servK, Agent B, Number Ts, servTicket}\})$
 $\quad \in \text{set evs};$
 $\neg \text{expiredAK Ta evs}; \neg \text{expiredSK Ts evs};$
 $A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{kerbIV_gets} \parallel$
 $\implies \text{Key servK} \notin \text{analz} (\text{spies evs})"$
 $\langle \text{proof} \rangle$

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:

$"\parallel \text{Crypt (shrK A) } \{\text{Key authK, Agent Tgs, Number Ta, authTicket}\}$
 $\quad \in \text{parts (spies evs)};$
 $\text{Crypt authK } \{\text{Key servK, Agent B, Number Ts, servTicket}\}$
 $\quad \in \text{parts (spies evs)};$
 $\neg \text{expiredAK Ta evs}; A \notin \text{bad}; \text{evs} \in \text{kerbIV_gets} \parallel$
 $\implies \text{Says Tgs A (Crypt authK } \{\text{Key servK, Agent B, Number Ts, servTicket}\})$
 $\quad \in \text{set evs}"$
 $\langle \text{proof} \rangle$

lemma Confidentiality_Serv_A:

$"\parallel \text{Crypt (shrK A) } \{\text{Key authK, Agent Tgs, Number Ta, authTicket}\}$
 $\quad \in \text{parts (spies evs)};$
 $\text{Crypt authK } \{\text{Key servK, Agent B, Number Ts, servTicket}\}$
 $\quad \in \text{parts (spies evs)};$
 $\neg \text{expiredAK Ta evs}; \neg \text{expiredSK Ts evs};$
 $A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{kerbIV_gets} \parallel$
 $\implies \text{Key servK} \notin \text{analz} (\text{spies evs})"$
 $\langle \text{proof} \rangle$

lemma Confidentiality_B:

$"\parallel \text{Crypt (shrK B) } \{\text{Agent A, Agent B, Key servK, Number Ts}\}$
 $\quad \in \text{parts (spies evs)};$
 $\text{Crypt authK } \{\text{Key servK, Agent B, Number Ts, servTicket}\}$
 $\quad \in \text{parts (spies evs)};$
 $\text{Crypt (shrK A) } \{\text{Key authK, Agent Tgs, Number Ta, authTicket}\}$
 $\quad \in \text{parts (spies evs)};$
 $\neg \text{expiredSK Ts evs}; \neg \text{expiredAK Ta evs};$
 $A \notin \text{bad}; B \notin \text{bad}; B \neq \text{Tgs}; \text{evs} \in \text{kerbIV_gets} \parallel$

7.10 2. Parties' strong authentication: non-injective agreement on the session key. The same guarantees also express

$\implies \text{Key servK} \notin \text{analz}(\text{spies evs})$
 <proof>

lemma u_Confidentiality_B:
 "[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_gets]
 $\implies \text{Key servK} \notin \text{analz}(\text{spies evs})$ "
 <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) \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; evs \in kerbIV_gets]
 $\implies \text{Says B A (Crypt servK (Number T3))} \in \text{set evs}$ "
 <proof>

lemma shrK_in_initState_Server[iff]: "Key (shrK A) \in initState Kas"
 <proof>

lemma shrK_in_knows_Server [iff]: "Key (shrK A) \in knows Kas evs"
 <proof>

lemma A_authenticates_and_keydist_to_Kas:
 "[Gets A (Crypt (shrK A) {Key authK, Peer, Ta, authTicket}) \in set evs;
 A \notin bad; evs \in kerbIV_gets]
 $\implies \text{Says Kas A (Crypt (shrK A) {Key authK, Peer, Ta, authTicket})} \in \text{set evs}$
 $\wedge \text{Key authK} \in \text{analz}(\text{knows Kas evs})$ "
 <proof>

lemma K3_imp_Gets_evs:
 "[Says A Tgs {Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta},
 Crypt authK {Agent A, Number T2}, Agent B}
 \in set evs; A \notin bad; evs \in kerbIV_gets]
 $\implies \text{Gets A (Crypt (shrK A) {Key authK, Agent Tgs, Number Ta,$
 Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}})
 \in set evs"

<proof>

lemma *Tgs_authenticates_and_keydist_to_A:*

```
"[[ Gets Tgs {
  Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta},
  Crypt authK {Agent A, Number T2}, Agent B } ∈ set 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
  ∧ Key authK ∈ analz (knows A evs)"
⟨proof⟩
```

lemma *K4_imp_Gets:*

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

lemma *A_authenticates_and_keydist_to_Tgs:*

```
"[[ Gets A (Crypt (shrK A) {Key authK, Agent Tgs, Number Ta, authTicket})
  ∈ set evs;
  Gets A (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
  ∈ set evs;
  Key authK ∉ analz (spies evs); A ∉ bad;
  evs ∈ kerbIV_gets ]
⇒ Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
  ∈ set evs
  ∧ Key authK ∈ analz (knows Tgs evs)
  ∧ Key servK ∈ analz (knows Tgs evs)"
⟨proof⟩
```

lemma *K5_imp_Gets:*

```
"[[ Says A B {servTicket, Crypt servK {Agent A, Number T3}} ∈ set evs;
  A ∉ bad; evs ∈ kerbIV_gets ]
⇒ ∃ authK Ts authTicket T2.
  Gets A (Crypt authK {Key servK, Agent B, Number Ts, servTicket}) ∈ set
  evs
  ∧ Says A Tgs {authTicket, Crypt authK {Agent A, Number T2}, Agent B} ∈
  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  $\notin$  analz (spies evs);
    A  $\notin$  bad; B  $\notin$  bad; B  $\neq$  Tgs; evs  $\in$  kerbIV_gets ]
 $\implies$  Says A B {Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}},
        Crypt servK {Agent A, Number T3}}  $\in$  set evs
     $\wedge$  Key servK  $\in$  analz (knows A evs)"
<proof>

```

lemma K6_imp_Gets:

```

    "[ Says B A (Crypt servK (Number T3))  $\in$  set evs;
      B  $\notin$  bad; evs  $\in$  kerbIV_gets ]
 $\implies \exists$  Ts X. Gets B {Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}}, X}
     $\in$  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)}  $\in$  set evs;
      Gets A (Crypt (shrK A) {Key authK, Agent Tgs, Number Ta, authTicket})
       $\in$  set evs;
      Key authK  $\notin$  analz (spies evs); Key servK  $\notin$  analz (spies evs);
      A  $\notin$  bad; B  $\notin$  bad; evs  $\in$  kerbIV_gets ]
 $\implies$  Says B A (Crypt servK (Number T3))  $\in$  set evs
     $\wedge$  Key servK  $\in$  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  $\leq$  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"

/ Oops1: "[ 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 [] = {}"
<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
    ⇒ K ∈ authKeys evs"
  <proof>

```

```

lemma authKeys_used: "K ∈ authKeys evs ⇒ Key K ∈ used evs"
  <proof>

```

8.3 Forwarding Lemmas

```

lemma Says_ticket_parts:
  "Says S A {Crypt K {SesKey, B, TimeStamp}, Ticket}
    ∈ set evs ⇒ Ticket ∈ parts (spies evs)"
  <proof>

```

```

lemma Says_ticket_analz:
  "Says S A {Crypt K {SesKey, B, TimeStamp}, Ticket}
    ∈ set evs ⇒ Ticket ∈ analz (spies evs)"
  <proof>

```

```

lemma Oops_range_spies1:
  "[[ Says Kas A {Crypt KeyA {Key authK, Peer, Ta}}, authTicket}
    ∈ set evs ;
    evs ∈ kerbV ] ⇒ authK ∉ range shrK & authK ∈ symKeys"
  <proof>

```

```

lemma Oops_range_spies2:
  "[[ Says Tgs A {Crypt authK {Key servK, Agent B, Ts}}, 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_analz:*

"[[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;
evs ∈ kerbV]]
⇒ SesKey ∉ range shrK"

⟨proof⟩

lemma *authTicket_authentic:*

"[[Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Ta}
∈ parts (spies evs);
evs ∈ kerbV]]
⇒ Says Kas A {Crypt (shrK A) {Key authK, Agent Tgs, Ta},
Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Ta}}
∈ set evs"

⟨proof⟩

lemma *authTicket_crypt_authK*:
 "[Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}
 ∈ parts (spies evs);
 evs ∈ kerbV]
 ⇒ authK ∈ authKeys evs"
 <proof>

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

lemma *Says_Tgs_message_form*:
 "[Says Tgs A {Crypt authK {Key servK, Agent B, Ts}, servTicket}
 ∈ set evs;
 evs ∈ kerbV]
 ⇒ B ≠ Tgs ∧
 servK ∉ range shrK ∧ servK ∉ authKeys evs ∧ servK ∈ symKeys ∧
 servTicket = (Crypt (shrK B) {Agent A, Agent B, Key servK, Ts}) ∧
 authK ∉ range shrK ∧ authK ∈ authKeys evs ∧ authK ∈ symKeys"
 <proof>

8.5 Authenticity theorems: confirm origin of sensitive messages

lemma *authK_authentic*:
 "[Crypt (shrK A) {Key authK, Peer, Ta}
 ∈ parts (spies evs);
 A ∉ bad; evs ∈ kerbV]
 ⇒ ∃ AT. Says Kas A {Crypt (shrK A) {Key authK, Peer, Ta}, AT}
 ∈ set evs"
 <proof>

If a certain encrypted message appears then it originated with Tgs

lemma *servK_authentic*:
 "[Crypt authK {Key servK, Agent B, Ts}
 ∈ parts (spies evs);
 Key authK ∉ analz (spies evs);
 authK ∉ range shrK;
 evs ∈ kerbV]
 ⇒ ∃ A ST. Says Tgs A {Crypt authK {Key servK, Agent B, Ts}, ST}
 ∈ set evs"
 <proof>

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:
  "[ 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 ∧
    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 ∈ kerbV ]
  ⇒ ∃ 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 ∈ kerbV ]
  ⇒ ∃ 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>

```

8.6 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 ∈ kerbV ]
  ⇒ ∃ Ta AT. Says Kas A {Crypt (shrK A) {Key authK, Agent Tgs, Ta}},
    AT} ∈ 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}
    ∈ 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]: "¬ 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;

```

$$B \neq Tgs; \text{ evs} \in \text{kerbV} \quad]$$

$$\implies \neg \text{AKcryptSK } SK \ K \ \text{evs}"$$

$$\langle \text{proof} \rangle$$

Long term keys are not issued as *servKeys*

lemma *shrK_not_AKcryptSK*:

$$" \text{evs} \in \text{kerbV} \implies \neg \text{AKcryptSK } K \ (\text{shrK } A) \ \text{evs}"$$

$$\langle \text{proof} \rangle$$

The Tgs message associates *servK* with *authK* and therefore not with any other key *authK*.

lemma *Says_Tgs_AKcryptSK*:

$$" [\text{Says } Tgs \ A \ \{\text{Crypt } \text{authK} \ \{\text{Key } \text{servK}, \text{Agent } B, \text{tt}\}, X \} \}$$

$$\in \text{set evs};$$

$$\text{authK}' \neq \text{authK}; \text{ evs} \in \text{kerbV} \quad]$$

$$\implies \neg \text{AKcryptSK } \text{authK}' \ \text{servK} \ \text{evs}"$$

$$\langle \text{proof} \rangle$$

lemma *AKcryptSK_not_AKcryptSK*:

$$" [\text{AKcryptSK } \text{authK} \ \text{servK} \ \text{evs}; \text{ evs} \in \text{kerbV} \quad]$$

$$\implies \neg \text{AKcryptSK } \text{servK} \ K \ \text{evs}"$$

$$\langle \text{proof} \rangle$$

lemma *not_different_AKcryptSK*:

$$" [\text{AKcryptSK } \text{authK} \ \text{servK} \ \text{evs};$$

$$\text{authK}' \neq \text{authK}; \text{ evs} \in \text{kerbV} \quad]$$

$$\implies \neg \text{AKcryptSK } \text{authK}' \ \text{servK} \ \text{evs} \ \wedge \ \text{servK} \in \text{symKeys}"$$

$$\langle \text{proof} \rangle$$

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*:

$$" P \longrightarrow (\text{Key } K \in \text{analz } (\text{Key}'KK \ \text{Un } H)) \longrightarrow (K:KK \mid \text{Key } K \in \text{analz } H)$$

$$\implies$$

$$P \longrightarrow (\text{Key } K \in \text{analz } (\text{Key}'KK \ \text{Un } H)) = (K:KK \mid \text{Key } K \in \text{analz } H)"$$

$$\langle \text{proof} \rangle$$

lemma *AKcryptSK_analz_insert*:

$$" [\text{AKcryptSK } K \ K' \ \text{evs}; K \in \text{symKeys}; \text{ evs} \in \text{kerbV} \quad]$$

$$\implies \text{Key } K' \in \text{analz } (\text{insert } (\text{Key } K) \ (\text{spies } \text{evs}))"$$

$$\langle \text{proof} \rangle$$

lemma *authKeys_are_not_AKcryptSK*:

$$" [K \in \text{authKeys } \text{evs} \ \text{Un } \text{range } \text{shrK}; \text{ evs} \in \text{kerbV} \quad]$$

$$\implies \forall SK. \neg \text{AKcryptSK } SK \ K \ \text{evs} \ \wedge \ K \in \text{symKeys}"$$

$$\langle \text{proof} \rangle$$

lemma *not_authKeys_not_AKcryptSK*:

$$" [K \notin \text{authKeys } \text{evs};$$

$$\begin{aligned}
& K \notin \text{range shrK}; \text{ evs} \in \text{kerbV} \parallel \\
& \implies \forall SK. \neg \text{AKcryptSK } K \text{ } SK \text{ evs}'' \\
& \langle \text{proof} \rangle
\end{aligned}$$

8.9 Secrecy Theorems

For the Oops2 case of the next theorem

lemma *Oops2_not_AKcryptSK*:

$$\begin{aligned}
& "[\text{ evs} \in \text{kerbV}; \\
& \quad \text{Says Tgs A } \{\text{Crypt authK} \\
& \quad \quad \{\text{Key servK, Agent B, Number Ts}\}, \text{servTicket}\} \\
& \quad \in \text{set evs}] \\
& \implies \neg \text{AKcryptSK servK SK evs}'' \\
& \langle \text{proof} \rangle
\end{aligned}$$

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)]*:

$$\begin{aligned}
& "\text{evs} \in \text{kerbV} \implies \\
& (\forall SK \text{ KK. } SK \in \text{symKeys} \ \& \ \text{KK} \leq -(\text{range shrK}) \longrightarrow \\
& (\forall K \in \text{KK. } \neg \text{AKcryptSK } K \text{ } SK \text{ evs}) \longrightarrow \\
& (\text{Key } SK \in \text{analz } (\text{Key}'\text{KK Un } (\text{spies evs}))) = \\
& (SK \in \text{KK} \mid \text{Key } SK \in \text{analz } (\text{spies evs}))" \\
& \langle \text{proof} \rangle
\end{aligned}$$

First simplification law for *analz*: no session keys encrypt authentication keys or shared keys.

lemma *analz_insert_freshK1*:

$$\begin{aligned}
& "[\text{ evs} \in \text{kerbV}; \text{ K} \in \text{authKeys evs Un range shrK}; \\
& \quad \text{SesKey} \notin \text{range shrK}] \\
& \implies (\text{Key } K \in \text{analz } (\text{insert } (\text{Key SesKey}) (\text{spies evs}))) = \\
& \quad (K = \text{SesKey} \mid \text{Key } K \in \text{analz } (\text{spies evs}))'' \\
& \langle \text{proof} \rangle
\end{aligned}$$

Second simplification law for *analz*: no service keys encrypt any other keys.

lemma *analz_insert_freshK2*:

$$\begin{aligned}
& "[\text{ evs} \in \text{kerbV}; \text{ servK} \notin (\text{authKeys evs}); \text{ servK} \notin \text{range shrK}; \\
& \quad K \in \text{symKeys}] \\
& \implies (\text{Key } K \in \text{analz } (\text{insert } (\text{Key servK}) (\text{spies evs}))) = \\
& \quad (K = \text{servK} \mid \text{Key } K \in \text{analz } (\text{spies evs}))'' \\
& \langle \text{proof} \rangle
\end{aligned}$$

Third simplification law for *analz*: only one authentication key encrypts a certain service key.

lemma *analz_insert_freshK3*:

$$\begin{aligned}
& "[\text{ AKcryptSK authK servK evs}; \\
& \quad \text{authK}' \neq \text{authK}; \text{ authK}' \notin \text{range shrK}; \text{ evs} \in \text{kerbV}] \\
& \implies (\text{Key servK} \in \text{analz } (\text{insert } (\text{Key authK}') (\text{spies evs}))) = \\
& \quad (\text{servK} = \text{authK}' \mid \text{Key servK} \in \text{analz } (\text{spies evs}))'' \\
& \langle \text{proof} \rangle
\end{aligned}$$

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 ∈ kerbV]
 ⇒ (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 ∈ kerbV]
 ⇒ Key servK ∈ 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 ∈ symKeys; A ∉ bad; evs ∈ kerbV]
 ⇒ 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 ∈ kerbV]
 ⇒ 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 ∈ 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:

```
"[ 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}
```


8.10 Parties authentication: each party verifies "the identity of another party who generated some data" (quoted from

```

    ∈ 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}
    ∈ parts (spies evs);
    Crypt (shrK A) {Key authK, Agent Tgs, Number Ta}
    ∈ parts (spies evs);
    ¬ expiredSK Ts evs; ¬ expiredAK Ta evs;
    B ≠ Tgs; A ∉ bad; B ∉ bad; evs ∈ kerbV ]
  ⇒ ∃ ST. Says A B {ST, 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);
 ¬ expiredSK Ts evs;
 B ≠ Tgs; A ∉ bad; B ∉ bad; evs ∈ kerbV]
 ⇒ ∃ ST. Says A B {ST, Crypt servK {Agent A, Number T3}} ∈ set evs"
 <proof>

lemma *A_authenticates_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; evs ∈ kerbV]
 ⇒ Says B A (Crypt servK (Number T3)) ∈ set evs"
 <proof>

lemma *A_authenticates_B_r*:
 "[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);
 ¬ expiredAK Ta evs; ¬ expiredSK Ts evs;
 A ∉ bad; B ∉ bad; evs ∈ kerbV]
 ⇒ Says B A (Crypt servK (Number T3)) ∈ set evs"
 <proof>

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*:
 "[Says Kas A {Crypt (shrK A) {Key authK, Peer, Ta}, authTicket} ∈ set evs;
 evs ∈ kerbV]
 ⇒ Kas Issues A with (Crypt (shrK A) {Key authK, Peer, Ta})
 on evs"
 <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*:

8.11 Parties' knowledge of session keys. An agent knows a session key if he used it to issue a cipher. These guarantee

```
"[ 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!

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⟩

```

8.12 Novel guarantees, never studied before. Because honest agents always say the right timestamp in authenticator

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 ]
  ⇒ A=A' ∧ SK=SK' ∧ B=B' ∧ ST=ST'"
<proof>

```

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: "[] ∈ otway"

  / Fake: "[| evsf ∈ otway; X ∈ synth (analz (knows Spy evsf)) |]
    ==> Says Spy B X # evsf ∈ otway"

  / Reception: "[| evsr ∈ otway; Says A B X ∈ set evsr |]
    ==> Gets B X # evsr ∈ otway"

  / OR1: "[| 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: "[| 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,
         Crypt (shrK B)
           {|Nonce NA, Nonce NB, Agent A, Agent B|}|}
    # evs2 : otway"

  / OR3: "[| evs3 ∈ otway; Key KAB ∉ used evs3;
    Gets Server
        {|Nonce NA, Agent A, Agent B,
         Crypt (shrK A) {|Nonce NA, Agent A, Agent B|},
         Crypt (shrK B) {|Nonce NA, Nonce NB, 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: "[| 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|}|}
    ∈ 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(knowns Spy evs);
    Crypt (shrK B) {|NC, NB, Agent C, Agent B|}|} ∈ parts(knowns 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 (knowns 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 (knowns 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"

/ Ops: — 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 $B \neq \text{Server}$ 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 ∉ bad; A ≠ B; evs ∈ otway |]
  ==> Crypt (shrK A) {|NA, Key K|} ∈ parts (knows Spy evs) -->
    Says A B {|NA, Agent A, Agent B,
      Crypt (shrK A) {|NA, Agent A, Agent B|}|}
    ∈ set evs -->
    (∃ B NB. Says Server B
      {|NA,
        Crypt (shrK A) {|NA, Key K|},
        Crypt (shrK B) {|NB, Key K|}|} ∈ 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 K_{ab} 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: "[|] ∈ orb"

  / Fake: "[| evsa ∈ orb; X ∈ synth (analz (knows Spy evsa)) |]
    ==> Says Spy B X # evsa ∈ orb"

  / Reception: "[| evsr ∈ orb; Says A B X ∈ set evsr |]
    ==> Gets B X # evsr ∈ orb"

  / OR1: "[| evs1 ∈ orb; Nonce NA ∉ used evs1 |]
    ==> Says A B {|Nonce M, Agent A, Agent B,
      Crypt (shrK A) {|Nonce NA, Nonce M, Agent A, Agent B|}|}
    # evs1 ∈ orb"

  / OR2: "[| evs2 ∈ orb; Nonce NB ∉ used evs2;
    Gets B {|Nonce M, Agent A, Agent B, X|} ∈ 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]]
 $\implies X \in \text{analz } (\text{knows Spy evs})$ "
 <proof>

lemma *OR4_parts_knows_Spy*:

"[[Gets B {Nonce M, Crypt (shrK B) {X, Nonce Nb, Key Kab}}] ∈ set evs;
 evs ∈ orb]] $\implies X \in \text{parts } (\text{knows Spy evs})$ "
 <proof>

lemma *Ops_parts_knows_Spy*:

"Says Server B {Nonce M, Crypt K' {X, Nonce Nb, K}} ∈ set evs
 $\implies K \in \text{parts } (\text{knows Spy evs})$ "
 <proof>

lemmas *OR2_parts_knows_Spy* =

OR2_analz_knows_Spy [THEN analz_into_parts, standard]

<ML>

lemma *Spy_see_shrK [simp]*:

"evs ∈ orb $\implies (\text{Key } (\text{shrK } A) \in \text{parts } (\text{knows Spy evs})) = (A \in \text{bad})$ "
 <proof>

lemma *Spy_analz_shrK [simp]*:

"evs ∈ orb $\implies (\text{Key } (\text{shrK } A) \in \text{analz } (\text{knows Spy evs})) = (A \in \text{bad})$ "
 <proof>

lemma *Spy_see_shrK_D [dest!]*:

"[|Key (shrK A) ∈ parts (knows Spy evs); evs ∈ orb|] $\implies A \in \text{bad}$ "
 <proof>

lemma *new_keys_not_used [simp]*:

"[Key K ∉ used evs; K ∈ symKeys; evs ∈ orb] $\implies K \notin \text{keysFor } (\text{parts } (\text{knows Spy evs}))$ "
 <proof>

12.1 Proofs involving analz

Describes the form of K and NA when the Server sends this message. Also for Ops case.

lemma *Says_Server_message_form*:

"[[Says Server B {Nonce M, Crypt (shrK B) {X, Nonce Nb, Key K}}] ∈ set evs;
 evs ∈ orb]]
 $\implies K \notin \text{range shrK} \ \& \ (\exists A \text{ Na. } X = (\text{Crypt } (\text{shrK } A) \{ \text{Nonce Na, Key K} \}))$ "
 <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 $\{\{ \text{Nonce } M', \text{Agent } A', \text{Agent } B, X', \text{Crypt } (\text{shrK } B) \{ \text{Nonce } Nb, \text{Nonce } M', \text{Nonce } M', \text{Agent } A', \text{Agent } B \} \} \} \in \text{set evs};$

$B \notin \text{bad}; \text{evs} \in \text{orb} \implies M=M' \ \& \ A=A' \ \& \ X=X' "$
 $\langle \text{proof} \rangle$

lemma analz_image_freshCryptK_lemma:

"(Crypt K X \in analz (Key'nE \cup H)) \longrightarrow (Crypt K X \in analz H) \implies
 (Crypt K X \in analz (Key'nE \cup H)) = (Crypt K X \in analz H)"
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

lemma analz_image_freshCryptK [rule_format]:

"evs \in orb \implies
 Key K \notin analz (knows Spy evs) \longrightarrow
 (\forall KK. KK \subseteq - (range shrK) \longrightarrow
 (Crypt K X \in analz (Key'KK \cup (knows Spy evs))) =
 (Crypt K X \in analz (knows Spy evs)))"
 $\langle \text{proof} \rangle$

lemma analz_insert_freshCryptK:

"[evs \in orb; Key K \notin analz (knows Spy evs);
 Seskey \notin range shrK] \implies
 (Crypt K X \in analz (insert (Key Seskey) (knows Spy evs))) =
 (Crypt K X \in analz (knows Spy evs))"
 $\langle \text{proof} \rangle$

lemma analz_hard:

"[Says A B $\{\{ \text{Nonce } M, \text{Agent } A, \text{Agent } B,$
 Crypt (shrK A) $\{\{ \text{Nonce } Na, \text{Nonce } M, \text{Agent } A, \text{Agent } B \} \} \} \in \text{set evs};$
 Crypt (shrK A) $\{\{ \text{Nonce } Na, \text{Key } K \} \} \in \text{analz (knows Spy evs);}$
 A \notin bad; B \notin bad; evs \in orb]
 \implies Says B A $\{\{ \text{Nonce } M, \text{Crypt (shrK A) } \{\{ \text{Nonce } Na, \text{Key } K \} \} \} \} \in \text{set evs}"$
 $\langle \text{proof} \rangle$

lemma Gets_Server_message_form':

"[Gets B $\{\{ \text{Nonce } M, \text{Crypt (shrK B) } \{X, \text{Nonce } Nb, \text{Key } K \} \} \} \in \text{set evs};$
 B \notin bad; evs \in orb]
 $\implies K \notin \text{range shrK} \ \& \ (\exists A Na. X = (\text{Crypt (shrK A) } \{\{ \text{Nonce } Na, \text{Key } K \} \}))"$
 $\langle \text{proof} \rangle$

lemma OR4_imp_Gets:

"[Says B A $\{\{ \text{Nonce } M, \text{Crypt (shrK A) } \{ \text{Nonce } Na, \text{Key } K \} \} \} \in \text{set evs};$
 B \notin bad; evs \in orb]
 $\implies (\exists Nb. \text{Gets } B \{ \{ \text{Nonce } M, \text{Crypt (shrK B) } \{ \text{Crypt (shrK A) } \{ \text{Nonce } Na, \text{Key } K \} \},$

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  $\notin$  used evs
    ==> (Hash[Key(shrK A)] {|Agent A, Agent B, Nonce NA, END|},
      {|Crypt (shrK A) {|Key KAB, Agent B, Nonce NA|}, END|},
      KAB)  $\in$  respond evs"

  | Cons: "[| (PA, RA, KAB)  $\in$  respond evs;
    Key KBC  $\notin$  used evs; Key KBC  $\notin$  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)
       $\in$  respond evs"
```

inductive_set

```
responses :: "event list => msg set"
for evs :: "event list"
where
  Nil: "END  $\in$  responses evs"

  | Cons: "[| RA  $\in$  responses evs; Key KAB  $\notin$  used evs |]
    ==> {|Crypt (shrK B) {|Key KAB, Agent A, Nonce NB|},
      RA|}  $\in$  responses evs"
```

inductive_set recur :: "event list set"

```
where
  Nil: "[]  $\in$  recur"

  | Fake: "[| evsf  $\in$  recur; X  $\in$  synth (analz (knows Spy evsf)) |]
    ==> Says Spy B X # evsf  $\in$  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 ∉ used []; Key K' ∉ used [];
      Key K'' ∉ used []; K ≠ K'; K' ≠ K''; K ≠ K'';
      Nonce NA ∉ used []; Nonce NB ∉ used []; Nonce NC ∉ used [];
      NA < NB; NB < NC |]

```

```

==> ∃K. ∃NA. ∃evs ∈ recur.
      Says B A {|Crypt (shrK A) {|Key K, Agent B, Nonce NA|}},
      END|} ∈ set evs"
⟨proof⟩

```

```

lemma respond_imp_not_used: "(PA, RB, KAB) ∈ respond evs ==> Key KAB ∉ used
evs"
⟨proof⟩

```

```

lemma Key_in_parts_respond [rule_format]:
  "[| Key K ∈ parts {RB}; (PB, RB, K') ∈ respond evs |] ==> Key K ∉ used
evs"
⟨proof⟩

```

Simple inductive reasoning about responses

```

lemma respond_imp_responses:
  "(PA, RB, KAB) ∈ respond evs ==> RB ∈ 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|} ∈ set evs ==> RA ∈ 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 ∈ recur ==> (Key (shrK A) ∈ parts (spies evs)) = (A ∈ bad)"
⟨proof⟩

```

```

lemma Spy_analz_shrK [simp]:
  "evs ∈ recur ==> (Key (shrK A) ∈ analz (spies evs)) = (A ∈ 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|}
    ∈ 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_secrecy_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 []
==> Key K ∉ analz (knows Spy evs)"
<proof>

```

15.3 Authenticating B to A

The encryption in message YM2 tells us it cannot be faked.

lemma *B_Said_YM2* [rule_format]:
 "[|Crypt (shrK B) {|Agent A, Nonce NA, nb|} ∈ parts (knows Spy evs);
 evs ∈ yahalom|]
 ==> B ∉ bad -->
 Says B Server {|Agent B, Crypt (shrK B) {|Agent A, Nonce NA, nb|}|}
 ∈ set evs"
 <proof>

If the server sends YM3 then B sent YM2

lemma *YM3_auth_B_to_A_lemma*:
 "[|Says Server A {|Crypt (shrK A) {|Agent B, Key K, Nonce NA, nb|}, X|}
 ∈ set evs; evs ∈ yahalom|]
 ==> B ∉ bad -->
 Says B Server {|Agent B, Crypt (shrK B) {|Agent A, Nonce NA, nb|}|}
 ∈ set evs"
 <proof>

If A receives YM3 then B has used nonce NA (and therefore is alive)

lemma *YM3_auth_B_to_A*:
 "[|Gets A {|Crypt (shrK A) {|Agent B, Key K, Nonce NA, nb|}, X|}
 ∈ set evs;
 A ∉ bad; B ∉ bad; evs ∈ yahalom|]
 ==> Says B Server {|Agent B, Crypt (shrK B) {|Agent A, Nonce NA, nb|}|}
 ∈ set evs"
 <proof>

15.4 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.

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, 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|]

```

    ==>  $\exists X. \text{ Says } A \ B \ \{|X, \text{ Crypt } K \ (\text{Nonce } NB)|\} \in \text{ set evs}$ "
  <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: "[ ]  $\in$  yahalom"

    / Fake: "[ | evsf  $\in$  yahalom;  $X \in \text{ synth } (\text{analz } (\text{knows Spy evsf}))$  | ]
      ==> Says Spy B X # evsf  $\in$  yahalom"

    / Reception: "[ | evsr  $\in$  yahalom; Says A B X  $\in$  set evsr | ]
      ==> Gets B X # evsr  $\in$  yahalom"

    / YM1: "[ | evs1  $\in$  yahalom; Nonce NA  $\notin$  used evs1 | ]
      ==> Says A B {|Agent A, Nonce NA|} # evs1  $\in$  yahalom"

    / YM2: "[ | evs2  $\in$  yahalom; Nonce NB  $\notin$  used evs2;
      Gets B {|Agent A, Nonce NA|}  $\in$  set evs2 | ]
      ==> Says B Server
        {|Agent B, Nonce NB, Crypt (shrK B) {|Agent A, Nonce NA|}|}
        # evs2  $\in$  yahalom"

    / YM3: "[ | evs3  $\in$  yahalom; Key KAB  $\notin$  used evs3;
      Gets Server {|Agent B, Nonce NB,
        Crypt (shrK B) {|Agent A, Nonce NA|}|}
         $\in$  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  $\in$  yahalom"

    / YM4: "[ | evs4  $\in$  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"

```

```

| Oops: "[| 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 KAB

```

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|}|}

```

$\in \text{set evs}$ "

<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|}
 $\in \text{set evs}$;
 $B \notin \text{bad}$; $\text{evs} \in \text{yahalom}$ |]
 $\implies \exists \text{nb}'. \text{ Says B Server } \{|Agent B, \text{nb}',$
 $\text{Crypt (shrK B) {|Agent A, Nonce NA|}\}$
 $\in \text{set evs}$ "

<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|}
 $\in \text{set evs}$;
 $A \notin \text{bad}$; $B \notin \text{bad}$; $\text{evs} \in \text{yahalom}$ |]
 $\implies \exists \text{nb}'. \text{ Says B Server }$
 $\{|Agent B, \text{nb}', \text{Crypt (shrK B) {|Agent A, Nonce NA|}\}\}$
 $\in \text{set evs}$ "

<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 $\text{Key K} \notin \text{analz (knows Spy evs)}$ 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|} $\in \text{analz (spies evs)}$;
 Crypt (shrK A') {|Agent B', Key K, na'|} $\in \text{analz (spies evs)}$;
 $\text{Key K} \notin \text{analz (knows Spy evs)}$; $\text{evs} \in \text{yahalom}$ |]
 $\implies A=A' \ \& \ B=B'$ "

<proof>

lemma *Auth_A_to_B_lemma [rule_format]*:

" $\text{evs} \in \text{yahalom}$
 $\implies \text{Key K} \notin \text{analz (knows Spy evs)} \implies$
 $K \in \text{symKeys} \implies$
 $\text{Crypt K (Nonce NB)} \in \text{parts (knows Spy evs)} \implies$
 $\text{Crypt (shrK B) {|Agent A, Agent B, Key K, Nonce NB|}}$
 $\in \text{parts (knows Spy evs)} \implies$
 $B \notin \text{bad} \implies$
 $(\exists X. \text{ Says A B } \{|X, \text{Crypt K (Nonce NB)}|\} \in \text{set evs})"$

<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 \notin analz (knows Spy evs); evs \in yahalom|]
 \implies Crypt K (Nonce NB) \in parts (knows Spy evs) \longrightarrow
 (\exists A B NA. Says Server A
 {|Crypt (shrK A) {|Agent B, Key K,
 Nonce NA, Nonce NB|},
 Crypt (shrK B) {|Agent A, Key K|}|}
 \in 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)|} \in set evs;
 Says B Server
 {|Agent B, Nonce NB, Crypt (shrK B) {|Agent A, Nonce NA|}|}
 \in set evs;
 A \notin bad; B \notin bad; evs \in yahalom |]
 $\implies \exists$ na nb. Says Server A
 {|Crypt (shrK A) {|Agent B, Key K, na, nb|},
 Crypt (shrK B) {|Agent A, Key K|}|}
 \in 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)|} \in set evs;
 Says B Server
 {|Agent B, Nonce NB, Crypt (shrK B) {|Agent A, Nonce NA|}|}
 \in set evs;
 A \notin bad; B \notin bad; evs \in yahalom |]
 \implies Key K \notin 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 \in yahalom
 \implies Key K \notin analz (knows Spy evs) \longrightarrow
 Crypt K (Nonce NB) \in parts (knows Spy evs) \longrightarrow
 Crypt (shrK B) {|Agent A, Key K|} \in parts (knows Spy evs) \longrightarrow
 B \notin bad \longrightarrow
 (\exists X. Says A B {|X, Crypt K (Nonce NB)|} \in 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
  ⇒ Crypt (pubEK C) {NA', Nonce NA, Agent D} ∈ 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, Agent B} ∈ 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, Agent B}) ∈ 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]]
  ⇒ Says B A (Crypt (pubEK A) {Nonce NA, Nonce NB, Agent B}) ∈ 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⟩
```

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 The TLS Protocol: Transport Layer Security

theory TLS imports Public NatPair 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 $NA \notin \text{range PRF}$ 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 $PMS \notin \text{range PRF}$ 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 $\text{PRF}(\text{PMS}, \text{NA}, \text{NB})$. 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 ClientFin-

ished. 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 encrypted 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"

/ Ops:
  — 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  $\neq$  sessionK arg"
<proof>
```

```
declare pubK_neq_sessionK [THEN not_sym, iff]
```

```
lemma priK_neq_sessionK [iff]: "invKey (publicKey b A)  $\neq$  sessionK arg"
<proof>
```

```
declare priK_neq_sessionK [THEN not_sym, iff]
```

```
lemmas keys_distinct = pubK_neq_sessionK priK_neq_sessionK
```

20.1 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 "[|  $\forall$  evs. ( $\@$  N. Nonce N  $\notin$  used evs)  $\notin$  range PRF; A  $\neq$  B |]"
==>  $\exists$  SID M.  $\exists$  evs  $\in$  tls.
      Notes A {|Number SID, Agent A, Agent B, Nonce M|}  $\in$  set evs"
<proof>
```

And one for ServerAccepts. Either FINISHED message may come first.

```
lemma "[|  $\forall$  evs. ( $\@$  N. Nonce N  $\notin$  used evs)  $\notin$  range PRF; A  $\neq$  B |]"
==>  $\exists$  SID NA PA NB PB M.  $\exists$  evs  $\in$  tls.
      Notes B {|Number SID, Agent A, Agent B, Nonce M|}  $\in$  set evs"
<proof>
```

Another one, for CertVerify (which is optional)

```
lemma "[|  $\forall$  evs. ( $\@$  N. Nonce N  $\notin$  used evs)  $\notin$  range PRF; A  $\neq$  B |]"
==>  $\exists$  NB PMS.  $\exists$  evs  $\in$  tls.
      Says A B (Crypt (priK A) (Hash{|Nonce NB, Agent B, Nonce PMS|}))
       $\in$  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  $\in$  tls;
      Notes A {|Number SID, Agent A, Agent B, Nonce M|}  $\in$  set evs0;
      Notes B {|Number SID, Agent A, Agent B, Nonce M|}  $\in$  set evs0;
       $\forall$  evs. ( $\@$  N. Nonce N  $\notin$  used evs)  $\notin$  range PRF;
      A  $\neq$  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⟩

```

20.2 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]

```

20.2.1 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⟩

```

20.2.2 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>
```

20.2.3 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>

```

20.3 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>

```

20.3.1 Protocol goal: $\text{serverK}(\text{Na}, \text{Nb}, \text{M})$ and $\text{clientK}(\text{Na}, \text{Nb}, \text{M})$ 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⟩
```

20.3.2 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>

```

20.3.3 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>

```

20.3.4 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>

```

20.3.5 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>

```

20.3.6 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

21 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(RPwD 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(RPwD 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 \in set evs; evs \in certified_mail|] ==> X \in parts(spies evs)"

<proof>

lemma CM2_S2TTP_analz_knows_Spy:

"[|Gets R {|Agent A, Agent B, em, Number AO, Number cleartext,
Nonce q, S2TTP|} \in set evs;

evs \in certified_mail|]

==> S2TTP \in 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 \in certified_mail

==> hr \notin synth (analz (spies evs)) -->

(\forall S2TTP. Notes TTP {|Agent R, Agent TTP, S2TTP, pwd, hr|}

\in set evs -->

(\exists 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|} \in set evs;

evs \in certified_mail|]

==> hr \in synth (analz (spies evs)) |

(\exists 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) \in parts (spies evs); evs \in certified_mail|]

==> A \in bad"

<proof>

lemma Spy_know_private_keys_iff [simp]:

"evs \in certified_mail

==> (Key (privateKey b A) \in parts (spies evs)) = (A \in bad)"

<proof>

lemma Spy_dont_know_TTPKey_parts [simp]:

"evs \in certified_mail ==> Key (privateKey b TTP) \notin parts(spies evs)"

<proof>

lemma Spy_dont_know_TTPKey_analz [simp]:

"evs \in certified_mail ==> Key (privateKey b TTP) \notin 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 AO, 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>
```

21.1 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'|}|}

```

$\in \text{set evs} \rightarrow R' = R \ \& \ S' = S \ \& \ AO' = AO \ \& \ hs' = hs))"$
 $\langle \text{proof} \rangle$

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⟩
```

21.2 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

```

22 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⟩

```

22.1 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 ⟹ 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 ⟹ 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 ∈ set evs → secureM → X ∈ knows Spy evs"
 ⟨proof⟩

lemma *Inputs_imp_knows_Spy_insecureM* [rule_format (no_asm)]:
 "Inputs A C X ∈ set evs → insecureM → X ∈ knows Spy evs"
 ⟨proof⟩

lemma *Outpts_imp_knows_Spy_secureM* [rule_format (no_asm)]:
 "Outpts C Spy X ∈ set evs → secureM → X ∈ knows Spy evs"
 ⟨proof⟩

lemma *Outpts_imp_knows_Spy_insecureM* [rule_format (no_asm)]:
 "Outpts C A X ∈ set evs → insecureM → X ∈ 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]

22.2 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_Inputs*: "knows A (Inputs A C X # evs) = insert X (knows A evs)"
 ⟨proof⟩

lemma *knows_C_Gets*: "knows A (C_Gets C X # evs) = knows A evs"
 ⟨proof⟩

lemma *knows_Outpts_secureM*:
 "secureM → knows A (Outpts C A X # evs) = insert X (knows A evs)"
 ⟨proof⟩

lemma *knows_Outpts_insecureM*:
 "insecureM → knows Spy (Outpts C A X # evs) = insert X (knows Spy evs)"
 ⟨proof⟩

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

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

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

lemma *knows_subset_knows_Inputs*: "knows A evs \subseteq knows A (Inputs A' C X # evs)"
 <proof>

lemma *knows_subset_knows_C_Gets*: "knows A evs \subseteq knows A (C_Gets C X # evs)"
 <proof>

lemma *knows_subset_knows_Outpts*: "knows A evs \subseteq knows A (Outpts C A' X # evs)"
 <proof>

lemma *knows_subset_knows_A_Gets*: "knows A evs \subseteq knows A (A_Gets A' X # evs)"
 <proof>

Agents know what they say

lemma *Says_imp_knows* [rule_format]: "Says A B X \in set evs \longrightarrow X \in knows A evs"
 <proof>

Agents know what they note

lemma *Notes_imp_knows* [rule_format]: "Notes A X \in set evs \longrightarrow X \in knows 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 ∈ set evs ⟶ (parts {X}) ⊆ 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 ⊆ 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>

end
```

23 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 ⟹ ∃ K. K ∉ KK & Key K ∉ used evs"

```

```

Nonce_supply_ax: "finite NN ⟹ ∃ N. N ∉ NN & Nonce N ∉ used evs"

```

23.1 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 ] ⇒ 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.

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) ]
  ⇒ K ∈ keysFor (parts (G ∪ H)) | Key K ∈ parts H"
<proof>

```

```

lemma Crypt_imp_keysFor: "Crypt K X ∈ H ⇒ K ∈ keysFor H"
<proof>

```

23.2 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 ]

```

$\implies \text{Nonce } (\text{Pairkey}(A,B)) \in \text{knows Spy evs}$
 $\langle \text{proof} \rangle$

lemma *Spy_knows_Spy_bad* [intro!]: " $A \in \text{bad} \implies \text{Key } (\text{pin } A) \in \text{knows Spy evs}$ "
 $\langle \text{proof} \rangle$

lemma *Crypt_Spy_analz_bad*:
 "[Crypt (pin A) X \in analz (knows Spy evs); A \in bad]
 $\implies X \in \text{analz } (\text{knows Spy evs})$ "
 $\langle \text{proof} \rangle$

lemma *shrK_in_initState* [iff]: " $\text{Key } (\text{shrK } A) \in \text{initState Server}$ "
 $\langle \text{proof} \rangle$

lemma *shrK_in_used* [iff]: " $\text{Key } (\text{shrK } A) \in \text{used evs}$ "
 $\langle \text{proof} \rangle$

lemma *crdK_in_initState* [iff]: " $\text{Key } (\text{crdK } A) \in \text{initState Server}$ "
 $\langle \text{proof} \rangle$

lemma *crdK_in_used* [iff]: " $\text{Key } (\text{crdK } A) \in \text{used evs}$ "
 $\langle \text{proof} \rangle$

lemma *pin_in_initState* [iff]: " $\text{Key } (\text{pin } A) \in \text{initState } A$ "
 $\langle \text{proof} \rangle$

lemma *pin_in_used* [iff]: " $\text{Key } (\text{pin } A) \in \text{used evs}$ "
 $\langle \text{proof} \rangle$

lemma *pairK_in_initState* [iff]: " $\text{Key } (\text{pairK } X) \in \text{initState Server}$ "
 $\langle \text{proof} \rangle$

lemma *pairK_in_used* [iff]: " $\text{Key } (\text{pairK } X) \in \text{used evs}$ "
 $\langle \text{proof} \rangle$

lemma *Key_not_used* [simp]: " $\text{Key } K \notin \text{used evs} \implies K \notin \text{range shrK}$ "
 $\langle \text{proof} \rangle$

lemma *shrK_neq* [simp]: " $\text{Key } K \notin \text{used evs} \implies \text{shrK } B \neq K$ "
 $\langle \text{proof} \rangle$

lemma *crdK_not_used* [simp]: " $\text{Key } K \notin \text{used evs} \implies K \notin \text{range crdK}$ "
 $\langle \text{proof} \rangle$

lemma *crdK_neq* [simp]: " $\text{Key } K \notin \text{used evs} \implies \text{crdK } C \neq K$ "
 $\langle \text{proof} \rangle$

lemma `pin_not_used [simp]: "Key K \notin used evs \implies K \notin range pin"`
`<proof>`

lemma `pin_neq [simp]: "Key K \notin used evs \implies pin A \neq K"`
`<proof>`

lemma `pairK_not_used [simp]: "Key K \notin used evs \implies K \notin range pairK"`
`<proof>`

lemma `pairK_neq [simp]: "Key K \notin used evs \implies pairK(A,B) \neq 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]`

23.3 Fresh nonces

lemma `Nonce_notin_initState [iff]: "Nonce N \notin parts (initState (Friend i))"`
`<proof>`

23.4 Supply fresh nonces for possibility theorems.

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* $KK \implies \exists K. K \notin KK \wedge \text{Key } K \notin \text{used evs}$. 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.

23.5 Specialized Rewriting for Theorems About *analz* and Image

lemma `subset_Compl_range_shrK: "A \subseteq - (range shrK) \implies shrK x \notin A"`
`<proof>`

lemma `subset_Compl_range_crdK: "A \subseteq - (range crdK) \implies crdK x \notin A"`

<proof>

lemma subset_Compl_range_pin: " $A \subseteq - (\text{range pin}) \implies \text{pin } x \notin A$ "

<proof>

lemma subset_Compl_range_pairK: " $A \subseteq - (\text{range pairK}) \implies \text{pairK } x \notin A$ "

<proof>

lemma insert_Key_singleton: " $\text{insert } (\text{Key } K) H = \text{Key } ' \{K\} \cup H$ "

<proof>

lemma insert_Key_image: " $\text{insert } (\text{Key } K) (\text{Key } 'KK \cup C) = \text{Key } '(\text{insert } K KK) \cup C$ "

<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_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:

" $(\text{Key } K \in \text{analz } (\text{Key } 'nE \cup H)) \longrightarrow (K \in nE \mid \text{Key } K \in \text{analz } H) \implies$
 $(\text{Key } K \in \text{analz } (\text{Key } 'nE \cup H)) = (K \in nE \mid \text{Key } K \in \text{analz } H)$ "

<proof>

23.6 Tactics for possibility theorems

<ML>

lemma invKey_shrK_iff [iff]:

" $(\text{Key } (\text{invKey } K) \in X) = (\text{Key } K \in X)$ "

<proof>

<ML>

lemma knows_subset_knows_Cons: " $\text{knows } A \text{ evs} \subseteq \text{knows } A (e \# \text{ evs})$ "

<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

```

24 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;
        Outpts (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 ]
⇒ Outpts (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 ]
⇒ Outpts (Card B) Spy {Nonce Nb, Key K,
                      Crypt (pairK(A, B)) {Nonce Na, Nonce Nb},
                      Crypt (pairK(A, B)) (Nonce Nb)}
   # evs7F ∈ sr"

/ SR8: "[ evs8 ∈ sr;
        Inputs B (Card B) {Agent A, Nonce Na} ∈ set evs8;
        Outpts (Card B) B {Nonce Nb, Key K,
                          Cert1, Cert2} ∈ set evs8 ]
⇒ Says B A {Nonce Nb, Cert1} # evs8 ∈ sr"

```

```

/ SR9: "[ evs9 ∈ sr; legalUse(Card A);
  Gets A {Nonce Pk, Cert1} ∈ set evs9;
  Outpts (Card A) A {Nonce Na, Cert2} ∈ 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 ∈ sr"

/ SR10: "[ evs10 ∈ sr; 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 ]
⇒ Outpts (Card A) A {Key K, Crypt (pairK(A, B)) (Nonce Nb)}
  # evs10 ∈ sr"

/ SR10Fake: "[ evs10F ∈ sr;
  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 ]
⇒ 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"

```

```

/ Ops1:
  "[ 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"

/ Ops2:
  "[ 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:

```

" $evs \in sr \implies \text{knows Spy (Inputs Spy C X \# evs) = insert X (knows Spy evs)}$ "
 <proof>

lemma *knows_Spy_Inputs_secureM_sr*:
 " $\llbracket A \neq \text{Spy}; evs \in sr \rrbracket \implies \text{knows Spy (Inputs A C X \# evs) = knows Spy evs}$ "
 <proof>

lemma *knows_Spy_Outpts_secureM_sr_Spy*:
 " $evs \in sr \implies \text{knows Spy (Outpts C Spy X \# evs) = insert X (knows Spy evs)}$ "
 <proof>

lemma *knows_Spy_Outpts_secureM_sr*:
 " $\llbracket A \neq \text{Spy}; evs \in sr \rrbracket \implies \text{knows Spy (Outpts C A X \# evs) = knows Spy evs}$ "
 <proof>

lemma *Inputs_A_Card_3*:
 " $\llbracket \text{Inputs A C (Agent A) } \in \text{ set evs}; A \neq \text{Spy}; evs \in sr \rrbracket$
 $\implies \text{legalUse(C)} \wedge C = (\text{Card A}) \wedge$
 $(\exists \text{ Pk Certificate. Gets A } \llbracket \text{Pk, Certificate} \rrbracket \in \text{ set evs})$ "
 <proof>

lemma *Inputs_B_Card_6*:
 " $\llbracket \text{Inputs B C } \llbracket \text{Agent A, Nonce Na} \rrbracket \in \text{ set evs}; B \neq \text{Spy}; evs \in sr \rrbracket$
 $\implies \text{legalUse(C)} \wedge C = (\text{Card B}) \wedge \text{Gets B } \llbracket \text{Agent A, Nonce Na} \rrbracket \in \text{ set evs}$ "
 <proof>

lemma *Inputs_A_Card_9*:
 " $\llbracket \text{Inputs A C } \llbracket \text{Agent B, Nonce Na, Nonce Nb, Nonce Pk,}$
 $\text{Cert1, Cert2, Cert3} \rrbracket \in \text{ set evs};$
 $A \neq \text{Spy}; evs \in sr \rrbracket$
 $\implies \text{legalUse(C)} \wedge C = (\text{Card A}) \wedge$
 $\text{Gets A } \llbracket \text{Nonce Pk, Cert1} \rrbracket \in \text{ set evs} \quad \wedge$
 $\text{Outpts (Card A) A } \llbracket \text{Nonce Na, Cert3} \rrbracket \in \text{ set evs} \quad \wedge$
 $\text{Gets A } \llbracket \text{Nonce Nb, Cert2} \rrbracket \in \text{ set evs}$ "
 <proof>

lemma *Outpts_A_Card_4*:
 " $\llbracket \text{Outpts C A } \llbracket \text{Nonce Na, (Crypt (crdK (Card A)) (Nonce Na))} \rrbracket \in \text{ set evs};$

$$\begin{aligned} & \text{evs} \in \text{sr} \quad] \\ \implies & \text{legalUse}(C) \wedge C = (\text{Card } A) \wedge \\ & \text{Inputs } A \ (\text{Card } A) \ (\text{Agent } A) \in \text{set evs}'' \\ \langle \text{proof} \rangle & \end{aligned}$$

lemma *Outpts_B_Card_7:*

$$\begin{aligned} & "[\text{Outpts } C \ B \ \{\text{Nonce } Nb, \text{Key } K, \\ & \quad \text{Crypt } (\text{pairK}(A,B)) \ \{\text{Nonce } Na, \text{Nonce } Nb\}, \\ & \quad \text{Cert2}\} \in \text{set evs}; \\ & \text{evs} \in \text{sr} \quad] \\ \implies & \text{legalUse}(C) \wedge C = (\text{Card } B) \wedge \\ & \text{Inputs } B \ (\text{Card } B) \ \{\text{Agent } A, \text{Nonce } Na\} \in \text{set evs}'' \\ \langle \text{proof} \rangle & \end{aligned}$$

lemma *Outpts_A_Card_10:*

$$\begin{aligned} & "[\text{Outpts } C \ A \ \{\text{Key } K, (\text{Crypt } (\text{pairK}(A,B)) \ (\text{Nonce } Nb))\} \in \text{set evs}; \\ & \text{evs} \in \text{sr} \quad] \\ \implies & \text{legalUse}(C) \wedge C = (\text{Card } A) \wedge \\ & (\exists \ Na \ Ver1 \ Ver2 \ Ver3. \\ & \text{Inputs } A \ (\text{Card } A) \ \{\text{Agent } B, \text{Nonce } Na, \text{Nonce } Nb, \text{Nonce } (\text{Pairkey}(A,B)), \\ & \quad \text{Ver1}, \text{Ver2}, \text{Ver3}\} \in \text{set evs})'' \\ \langle \text{proof} \rangle & \end{aligned}$$

lemma *Outpts_A_Card_10_imp_Inputs:*

$$\begin{aligned} & "[\text{Outpts } (\text{Card } A) \ A \ \{\text{Key } K, \text{Certificate}\} \in \text{set evs}; \text{evs} \in \text{sr} \quad] \\ \implies & (\exists \ B \ Na \ Nb \ Ver1 \ Ver2 \ Ver3. \\ & \text{Inputs } A \ (\text{Card } A) \ \{\text{Agent } B, \text{Nonce } Na, \text{Nonce } Nb, \text{Nonce } (\text{Pairkey}(A,B)), \\ & \quad \text{Ver1}, \text{Ver2}, \text{Ver3}\} \in \text{set evs})'' \\ \langle \text{proof} \rangle & \end{aligned}$$

lemma *Outpts_honest_A_Card_4:*

$$\begin{aligned} & "[\text{Outpts } C \ A \ \{\text{Nonce } Na, \text{Crypt } K \ X\} \in \text{set evs}; \\ & \quad A \neq \text{Spy}; \text{evs} \in \text{sr} \quad] \\ \implies & \text{legalUse}(C) \wedge C = (\text{Card } A) \wedge \\ & \text{Inputs } A \ (\text{Card } A) \ (\text{Agent } A) \in \text{set evs}'' \\ \langle \text{proof} \rangle & \end{aligned}$$

lemma *Outpts_honest_B_Card_7:*

$$[\text{Outpts } C \ B \ \{\text{Nonce } Nb, \text{Key } K, \text{Cert1}, \text{Cert2}\} \in \text{set evs};$$

$$B \neq \text{Spy}; \text{evs} \in \text{sr} \parallel$$

$$\implies \text{legalUse}(C) \wedge C = (\text{Card } B) \wedge$$

$$(\exists A \text{ Na. Inputs } B (\text{Card } B) \{ \text{Agent } A, \text{Nonce Na} \} \in \text{set evs})"$$

$$\langle \text{proof} \rangle$$

lemma *Outpts_honest_A_Card_10:*

$$\parallel \text{Outpts } C \text{ A } \{ \text{Key } K, \text{Certificate} \} \in \text{set evs};$$

$$A \neq \text{Spy}; \text{evs} \in \text{sr} \parallel$$

$$\implies \text{legalUse } (C) \wedge C = (\text{Card } A) \wedge$$

$$(\exists B \text{ Na Nb Pk Ver1 Ver2 Ver3.}$$

$$\text{Inputs } A (\text{Card } A) \{ \text{Agent } B, \text{Nonce Na, Nonce Nb, Pk,}$$

$$\text{Ver1, Ver2, Ver3} \} \in \text{set evs})"$$

$$\langle \text{proof} \rangle$$

lemma *Outpts_which_Card_4:*

$$\parallel \text{Outpts } (\text{Card } A) \text{ A } \{ \text{Nonce Na, Crypt } K \text{ X} \} \in \text{set evs}; \text{evs} \in \text{sr} \parallel$$

$$\implies \text{Inputs } A (\text{Card } A) (\text{Agent } A) \in \text{set evs}"$$

$$\langle \text{proof} \rangle$$

lemma *Outpts_which_Card_7:*

$$\parallel \text{Outpts } (\text{Card } B) \text{ B } \{ \text{Nonce Nb, Key } K, \text{Cert1, Cert2} \} \in \text{set evs};$$

$$\text{evs} \in \text{sr} \parallel$$

$$\implies \exists A \text{ Na. Inputs } B (\text{Card } B) \{ \text{Agent } A, \text{Nonce Na} \} \in \text{set evs}"$$

$$\langle \text{proof} \rangle$$

lemma *Outpts_which_Card_10:*

$$\parallel \text{Outpts } (\text{Card } A) \text{ A } \{ \text{Key } (\text{sesK}(\text{Nb}, \text{pairK}(\text{A}, \text{B}))),$$

$$\text{Crypt } (\text{pairK}(\text{A}, \text{B})) (\text{Nonce Nb}) \} \in \text{set evs};$$

$$\text{evs} \in \text{sr} \parallel$$

$$\implies \exists \text{ Na. Inputs } A (\text{Card } A) \{ \text{Agent } B, \text{Nonce Na, Nonce Nb, Nonce } (\text{Pairkey}(\text{A}, \text{B})),$$

$$\text{Crypt } (\text{shrK } A) \{ \text{Nonce } (\text{Pairkey}(\text{A}, \text{B})), \text{Agent } B \},$$

$$\text{Crypt } (\text{pairK}(\text{A}, \text{B})) \{ \text{Nonce Na, Nonce Nb} \},$$

$$\text{Crypt } (\text{crdK } (\text{Card } A)) (\text{Nonce Na}) \} \in \text{set evs}"$$

$$\langle \text{proof} \rangle$$

lemma *Outpts_A_Card_form_4:*

$$\parallel \text{Outpts } (\text{Card } A) \text{ A } \{ \text{Nonce Na, Certificate} \} \in \text{set evs};$$

$$\forall p \text{ q. Certificate} \neq \{ p, q \}; \text{evs} \in \text{sr} \parallel$$

$$\implies \text{Certificate} = (\text{Crypt } (\text{crdK } (\text{Card } A)) (\text{Nonce Na}))"$$

$$\langle \text{proof} \rangle$$

lemma *Outpts_B_Card_form_7:*

$$\parallel \text{Outpts } (\text{Card } B) \text{ B } \{ \text{Nonce Nb, Key } K, \text{Cert1, Cert2} \} \in \text{set evs};$$

$$\text{evs} \in \text{sr} \parallel$$

$\implies \exists A \text{ Na.}$
 $K = \text{sesK}(\text{Nb}, \text{pairK}(A, B)) \wedge$
 $\text{Cert1} = (\text{Crypt}(\text{pairK}(A, B)) \{ \text{Nonce Na}, \text{Nonce Nb} \}) \wedge$
 $\text{Cert2} = (\text{Crypt}(\text{pairK}(A, B)) (\text{Nonce Nb}))"$
 $\langle \text{proof} \rangle$

lemma *Outpts_A_Card_form_10*:
 $"\llbracket \text{Outpts}(\text{Card } A) A \{ \text{Key } K, \text{Certificate} \} \in \text{set evs}; \text{evs} \in \text{sr} \rrbracket$
 $\implies \exists B \text{ Nb.}$
 $K = \text{sesK}(\text{Nb}, \text{pairK}(A, B)) \wedge$
 $\text{Certificate} = (\text{Crypt}(\text{pairK}(A, B)) (\text{Nonce Nb}))"$
 $\langle \text{proof} \rangle$

lemma *Outpts_A_Card_form_bis*:
 $"\llbracket \text{Outpts}(\text{Card } A') A' \{ \text{Key}(\text{sesK}(\text{Nb}, \text{pairK}(A, B))), \text{Certificate} \} \in \text{set evs};$
 $\text{evs} \in \text{sr} \rrbracket$
 $\implies A' = A \wedge$
 $\text{Certificate} = (\text{Crypt}(\text{pairK}(A, B)) (\text{Nonce Nb}))"$
 $\langle \text{proof} \rangle$

lemma *Inputs_A_Card_form_9*:
 $"\llbracket \text{Inputs } A (\text{Card } A) \{ \text{Agent } B, \text{Nonce Na}, \text{Nonce Nb}, \text{Nonce Pk},$
 $\text{Cert1}, \text{Cert2}, \text{Cert3} \} \in \text{set evs};$
 $\text{evs} \in \text{sr} \rrbracket$
 $\implies \text{Cert3} = \text{Crypt}(\text{crdK}(\text{Card } A)) (\text{Nonce Na})"$
 $\langle \text{proof} \rangle$

lemma *Inputs_Card_legalUse*:
 $"\llbracket \text{Inputs } A (\text{Card } A) X \in \text{set evs}; \text{evs} \in \text{sr} \rrbracket \implies \text{legalUse}(\text{Card } A)"$
 $\langle \text{proof} \rangle$

lemma *Outpts_Card_legalUse*:
 $"\llbracket \text{Outpts}(\text{Card } A) A X \in \text{set evs}; \text{evs} \in \text{sr} \rrbracket \implies \text{legalUse}(\text{Card } A)"$
 $\langle \text{proof} \rangle$

lemma *Inputs_Card*: $"\llbracket \text{Inputs } A C X \in \text{set evs}; A \neq \text{Spy}; \text{evs} \in \text{sr} \rrbracket$
 $\implies C = (\text{Card } A) \wedge \text{legalUse}(C)"$
 $\langle \text{proof} \rangle$

lemma *Outpts_Card*: $"\llbracket \text{Outpts } C A X \in \text{set evs}; A \neq \text{Spy}; \text{evs} \in \text{sr} \rrbracket$
 $\implies C = (\text{Card } A) \wedge \text{legalUse}(C)"$
 $\langle \text{proof} \rangle$

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;

$$\text{Outpts (Card } A') \ A' \ \{\text{Key } K, V'\} \in \text{set evs};$$

$$\text{evs} \in \text{sr} \implies A=A' \wedge V=V'$$

$$\langle \text{proof} \rangle$$

lemma *Outpts_A_Card_Unique*:

$$\llbracket \text{Outpts (Card } A) \ A \ \{\text{Nonce } Na, \text{rest}\} \in \text{set evs}; \text{evs} \in \text{sr} \rrbracket$$

$$\implies \text{Unique (Outpts (Card } A) \ A \ \{\text{Nonce } Na, \text{rest}\}) \text{ on evs}$$

$$\langle \text{proof} \rangle$$

lemma *Spy_knows_Na*:

$$\llbracket \text{Says } A \ B \ \{\text{Agent } A, \text{Nonce } Na\} \in \text{set evs}; \text{evs} \in \text{sr} \rrbracket$$

$$\implies \text{Nonce } Na \in \text{analz (knows Spy evs)}$$

$$\langle \text{proof} \rangle$$

lemma *Spy_knows_Nb*:

$$\llbracket \text{Says } B \ A \ \{\text{Nonce } Nb, \text{Certificate}\} \in \text{set evs}; \text{evs} \in \text{sr} \rrbracket$$

$$\implies \text{Nonce } Nb \in \text{analz (knows Spy evs)}$$

$$\langle \text{proof} \rangle$$

lemma *Pairkey_Gets_analz_knows_Spy*:

$$\llbracket \text{Gets } A \ \{\text{Nonce (Pairkey(A,B)), Certificate}\} \in \text{set evs}; \text{evs} \in \text{sr} \rrbracket$$

$$\implies \text{Nonce (Pairkey(A,B))} \in \text{analz (knows Spy evs)}$$

$$\langle \text{proof} \rangle$$

lemma *Pairkey_Inputs_imp_Gets*:

$$\llbracket \text{Inputs } A \ (\text{Card } A)$$

$$\quad \{\text{Agent } B, \text{Nonce } Na, \text{Nonce } Nb, \text{Nonce (Pairkey(A,B))},$$

$$\quad \text{Cert1, Cert3, Cert2}\} \in \text{set evs};$$

$$A \neq \text{Spy}; \text{evs} \in \text{sr} \rrbracket$$

$$\implies \text{Gets } A \ \{\text{Nonce (Pairkey(A,B)), Cert1}\} \in \text{set evs}$$

$$\langle \text{proof} \rangle$$

lemma *Pairkey_Inputs_analz_knows_Spy*:

$$\llbracket \text{Inputs } A \ (\text{Card } A)$$

$$\quad \{\text{Agent } B, \text{Nonce } Na, \text{Nonce } Nb, \text{Nonce (Pairkey(A,B))},$$

$$\quad \text{Cert1, Cert3, Cert2}\} \in \text{set evs};$$

$$\text{evs} \in \text{sr} \rrbracket$$

$\implies \text{Nonce } (\text{Pairkey}(A,B)) \in \text{analz } (\text{knows Spy evs})"$
 $\langle \text{proof} \rangle$

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]

$\langle ML \rangle$

lemma Spy_parts_keys [simp]: "evs \in sr \implies
 (Key (shrK P) \in parts (knows Spy evs)) = (Card P \in cloned) \wedge
 (Key (pin P) \in parts (knows Spy evs)) = (P \in bad \vee Card P \in cloned) \wedge
 (Key (crdK C) \in parts (knows Spy evs)) = (C \in cloned) \wedge
 (Key (pairK(A,B)) \in parts (knows Spy evs)) = (Card B \in cloned)"
 $\langle \text{proof} \rangle$

lemma Spy_analz_shrK[simp]: "evs \in sr \implies
 (Key (shrK P) \in analz (knows Spy evs)) = (Card P \in cloned)"
 $\langle \text{proof} \rangle$

lemma Spy_analz_crdK[simp]: "evs \in sr \implies
 (Key (crdK C) \in analz (knows Spy evs)) = (C \in cloned)"
 $\langle \text{proof} \rangle$

lemma Spy_analz_pairK[simp]: "evs \in sr \implies
 (Key (pairK(A,B)) \in analz (knows Spy evs)) = (Card B \in cloned)"
 $\langle \text{proof} \rangle$

lemma analz_image_Key_Un_Nonce: "analz (Key'K \cup Nonce'N) = Key'K \cup Nonce'N"
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

lemma *analz_image_freshK* [rule_format]:
 "evs \in sr $\implies \forall K KK.$
 (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 sr \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 sr]
 \implies 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)} \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 sr]
 \implies Outpts (Card A) A {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
 {Key K, Crypt (pairK(A,B)) (Nonce Nb)} \in set evs;
 evs \in sr]
 $\implies \exists Na.$ Crypt (pairK(A,B)) {Nonce Na, Nonce Nb} \in parts (knows Spy
 evs)"
 <proof>

lemma *Nb_certificate_authentic_bis:*

```
"[[ Crypt (pairK(A,B)) (Nonce Nb) ∈ parts (knows Spy 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:*

```
"[[ Crypt (shrK A) {Nonce Pk, Agent B} ∈ parts (knows Spy evs);
   Card A ∉ cloned; evs ∈ sr ]]
⇒ Pk = Pairkey(A,B) ∧
   Says Server A {Nonce Pk,
   Crypt (shrK A) {Nonce Pk, Agent B}}
   ∈ set evs"
```

<proof>

lemma *sesK_authentic:*

```
"[[ Key (sesK(Nb,pairK(A,B))) ∈ parts (knows Spy evs);
   A ≠ Spy; B ≠ Spy; ¬illegalUse(Card A); ¬illegalUse(Card B);
   evs ∈ sr ]]
⇒ Notes Spy {Key (sesK(Nb,pairK(A,B))), Nonce Nb, Agent A, Agent B}
   ∈ set evs"
```

<proof>

lemma *Confidentiality:*

```
"[[ Notes Spy {Key (sesK(Nb,pairK(A,B))), Nonce Nb, Agent A, Agent B}
   ∉ set evs;
   A ≠ Spy; B ≠ Spy; ¬illegalUse(Card A); ¬illegalUse(Card B);
   evs ∈ sr ]]
⇒ Key (sesK(Nb,pairK(A,B))) ∉ 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} ∉ set evs;
   A ≠ Spy; B ≠ Spy; ¬illegalUse(Card A); Card B ∉ cloned;
   evs ∈ sr ]]
⇒ Key K ∉ analz (knows Spy evs)"
```

$\langle proof \rangle$

lemma *A_authenticates_B*:

" $\llbracket \text{Outpts (Card A) A } \{ \text{Key K, Crypt (pairK(A,B)) (Nonce Nb)} \} \in \text{set evs};$

$\neg \text{illegalUse(Card B)};$

$\text{evs} \in \text{sr} \rrbracket$

$\implies \exists \text{ Na.}$

$\text{Outpts (Card B) B } \{ \text{Nonce Nb, Key K,}$
 $\text{Crypt (pairK(A,B)) } \{ \text{Nonce Na, Nonce Nb} \},$
 $\text{Crypt (pairK(A,B)) (Nonce Nb)} \} \in \text{set evs}"$

$\langle proof \rangle$

lemma *A_authenticates_B_Gets*:

" $\llbracket \text{Gets A } \{ \text{Nonce Nb, Crypt (pairK(A,B)) } \{ \text{Nonce Na, Nonce Nb} \} \}$

$\in \text{set evs};$

$\neg \text{illegalUse(Card B)};$

$\text{evs} \in \text{sr} \rrbracket$

$\implies \text{Outpts (Card B) B } \{ \text{Nonce Nb, Key (sesK(Nb, pairK (A, B))),}$
 $\text{Crypt (pairK(A,B)) } \{ \text{Nonce Na, Nonce Nb} \},$
 $\text{Crypt (pairK(A,B)) (Nonce Nb)} \} \in \text{set evs}"$

$\langle proof \rangle$

lemma *B_authenticates_A*:

" $\llbracket \text{Gets B (Crypt (pairK(A,B)) (Nonce Nb))} \in \text{set evs};$

$B \neq \text{Spy}; \neg \text{illegalUse(Card A)}; \neg \text{illegalUse(Card B)};$

$\text{evs} \in \text{sr} \rrbracket$

$\implies \text{Outpts (Card A) A}$

$\{ \text{Key (sesK(Nb, pairK(A,B))), Crypt (pairK(A,B)) (Nonce Nb)} \} \in \text{set evs}"$

$\langle proof \rangle$

lemma *Confidentiality_A*: " $\llbracket \text{Outpts (Card A) A}$

$\{ \text{Key K, Crypt (pairK(A,B)) (Nonce Nb)} \} \in \text{set evs};$

$\text{Notes Spy } \{ \text{Key K, Nonce Nb, Agent A, Agent B} \} \notin \text{set evs};$

$A \neq \text{Spy}; B \neq \text{Spy}; \neg \text{illegalUse(Card A)}; \neg \text{illegalUse(Card B)};$

$\text{evs} \in \text{sr} \rrbracket$

$\implies \text{Key K} \notin \text{analz (knows Spy evs)}"$

$\langle proof \rangle$

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 "
```

⟨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 ∈ 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 ∧
```


$$\text{Outpts (Card A) A } \{\text{Nonce Na, Cert3}\}$$

$$\in \text{set evs}"$$

$$\langle \text{proof} \rangle$$

lemma SR4_imp:

$$" [\text{Outpts (Card A) A } \{\text{Nonce Na, Crypt (crdK (Card A)) (Nonce Na)}\}$$

$$\in \text{set evs};$$

$$A \neq \text{Spy}; \text{ evs} \in \text{sr}]$$

$$\implies \exists \text{ Pk V. Gets A } \{\text{Pk, V}\} \in \text{set evs}"$$

$$\langle \text{proof} \rangle$$

lemma SR7_imp:

$$" [\text{Outpts (Card B) B } \{\text{Nonce Nb, Key K,}$$

$$\text{Crypt (pairK(A,B)) } \{\text{Nonce Na, Nonce Nb}\},$$

$$\text{Cert2}\} \in \text{set evs};$$

$$B \neq \text{Spy}; \text{ evs} \in \text{sr}]$$

$$\implies \text{Gets B } \{\text{Agent A, Nonce Na}\} \in \text{set evs}"$$

$$\langle \text{proof} \rangle$$

lemma SR10_imp:

$$" [\text{Outpts (Card A) A } \{\text{Key K, Crypt (pairK(A,B)) (Nonce Nb)}\}$$

$$\in \text{set evs};$$

$$A \neq \text{Spy}; \text{ evs} \in \text{sr}]$$

$$\implies \exists \text{ Cert1 Cert2.}$$

$$\text{Gets A } \{\text{Nonce (Pairkey (A, B)), Cert1}\} \in \text{set evs} \wedge$$

$$\text{Gets A } \{\text{Nonce Nb, Cert2}\} \in \text{set evs}"$$

$$\langle \text{proof} \rangle$$

lemma Outpts_Server_not_evs: $\text{"evs} \in \text{sr} \implies \text{Outpts (Card Server) P X} \notin \text{set evs}"$

$$\langle \text{proof} \rangle$$

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

25 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  $\neq$  sesK(m,pk)"
crdK_disj_sesK [iff]: "crdK C  $\neq$  sesK(m,pk)"
pin_disj_sesK [iff]: "pin P  $\neq$  sesK(m,pk)"
pairK_disj_sesK [iff]: "pairK(A,B)  $\neq$  sesK(m,pk)"

```

```

Atomic_distrib [iff]: "Atomic'(KEY'K  $\cup$  NONCE'N) =
    Atomic'(KEY'K)  $\cup$  Atomic'(NONCE'N)"

```

```

shouprubin_assumes_securemeans [iff]: "evs  $\in$  srb  $\implies$  secureM"

```

constdefs

```

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

```

inductive_set srb :: "event list set"
where

```

Nil: "[]  $\in$  srb"

```

```

/ Fake: "[ evsF  $\in$  srb; X  $\in$  synth (analz (knows Spy evsF));
    illegalUse(Card B) ]
 $\implies$  Says Spy A X #
    Inputs Spy (Card B) X # evsF  $\in$  srb"

```

```

/ Forge:
    "[ evsFo  $\in$  srb; Nonce Nb  $\in$  analz (knows Spy evsFo);
    Key (pairK(A,B))  $\in$  knows Spy evsFo ]
 $\implies$  Notes Spy (Key (sesK(Nb,pairK(A,B)))) # evsFo  $\in$  srb"

```

```

/ Reception: "[ evsrb  $\in$  srb; Says A B X  $\in$  set evsrb ]
 $\implies$  Gets B X # evsrb  $\in$  srb"

```

```

/ SR_U1: "[ evs1  $\in$  srb; A  $\neq$  Server ]
 $\implies$  Says A Server {Agent A, Agent B}
    # evs1  $\in$  srb"

```

```

/ SR_U2: "[ evs2  $\in$  srb;
    Gets Server {Agent A, Agent B}  $\in$  set evs2 ]
 $\implies$  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]
 ⇒ Outpts (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]
 ⇒ Outpts (Card A) Spy {Nonce Na, Crypt (crdK (Card A)) (Nonce Na)}
 # evs4F ∈ srb"

/ SR_U5: "[evs5 ∈ srb;
 Outpts (Card A) A {Nonce Na, Certificate} ∈ set evs5;
 ∀ p q. Certificate ≠ {p, q}]
 ⇒ Says A B {Agent A, Nonce Na} # evs5 ∈ srb"

/ SR_U6: "[evs6 ∈ srb; legalUse(Card B);
 Gets B {Agent A, Nonce Na} ∈ set evs6]
 ⇒ Inputs B (Card B) {Agent A, Nonce Na}
 # evs6 ∈ srb"

/ SR_U7: "[evs7 ∈ srb;
 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]
 ⇒ Outpts (Card B) B {Nonce Nb, Agent A, Key K,
 Crypt (pairK(A, B)) {Nonce Na, Nonce Nb}},

```

          Crypt (pairK(A,B)) (Nonce Nb) }
# evs7 ∈ srb"

/ SR_U7Fake: "[ evs7F ∈ srb; 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, Agent A, Key K,
  Crypt (pairK(A,B)) {Nonce Na, Nonce Nb},
  Crypt (pairK(A,B)) (Nonce Nb) }
# evs7F ∈ srb"

/ SR_U8: "[ evs8 ∈ srb;
  Inputs B (Card B) {Agent A, Nonce Na} ∈ set evs8;
  Outputs (Card B) B {Nonce Nb, Agent A, Key K,
  Cert1, Cert2} ∈ set evs8 ]
⇒ Says B A {Nonce Nb, Cert1} # evs8 ∈ srb"

/ SR_U9: "[ evs9 ∈ srb; legalUse(Card A);
  Gets A {Nonce Pk, Cert1} ∈ set evs9;
  Outputs (Card A) A {Nonce Na, Cert2} ∈ 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;
    Outpts (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 ≠ Spy; evs ∈ srb] ⇒ knows Spy (Inputs A C X # evs) = knows Spy evs"
 <proof>

lemma *knows_Spy_Outpts_secureM_srb_Spy*:
 "evs ∈ srb ⇒ knows Spy (Outpts C Spy X # evs) = insert X (knows Spy evs)"
 <proof>

lemma *knows_Spy_Outpts_secureM_srb*:
 "[A ≠ Spy; evs ∈ srb] ⇒ 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 ∈ srb]
 ⇒ 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 ∈ srb]
 ⇒ legalUse(C) ∧ C = (Card B) ∧ Gets B {Agent A, Nonce Na} ∈ set
 evs"
 <proof>

lemma *Inputs_A_Card_9:*

"[Inputs A C {Agent B, Nonce Na, Nonce Nb, Nonce Pk,
 Cert1, Cert2, Cert3} ∈ set evs;
 A ≠ Spy; evs ∈ srb]
 ⇒ legalUse(C) ∧ C = (Card A) ∧
 Gets A {Nonce Pk, Cert1} ∈ set evs ∧
 Outpts (Card A) A {Nonce Na, Cert3} ∈ set evs ∧
 Gets A {Nonce Nb, Cert2} ∈ set evs"
 <proof>

lemma *Outpts_A_Card_4:*

"[Outpts C A {Nonce Na, (Crypt (crdK (Card A)) (Nonce Na))} ∈ set evs;
 evs ∈ srb]
 ⇒ legalUse(C) ∧ C = (Card A) ∧
 Inputs A (Card A) (Agent A) ∈ set evs"
 <proof>

lemma *Outpts_B_Card_7:*

"[Outpts C B {Nonce Nb, Agent A, Key K,
 Crypt (pairK(A,B)) {Nonce Na, Nonce Nb},
 Cert2} ∈ set evs;
 evs ∈ srb]
 ⇒ legalUse(C) ∧ C = (Card B) ∧
 Inputs B (Card B) {Agent A, Nonce Na} ∈ set evs"
 <proof>

lemma *Outpts_A_Card_10:*

"[Outpts C A {Agent B, Nonce Nb,
 Key K, (Crypt (pairK(A,B)) (Nonce Nb))} ∈ set evs;
 evs ∈ srb]
 ⇒ legalUse(C) ∧ C = (Card A) ∧
 (∃ Na Ver1 Ver2 Ver3.
 Inputs A (Card A) {Agent B, Nonce Na, Nonce Nb, Nonce (Pairkey(A,B)),

$\langle proof \rangle$ $Ver1, Ver2, Ver3 \in set\ evs"$

lemma *Outpts_A_Card_10_imp_Inputs:*
 $"[Outpts\ (Card\ A)\ A\ \{Agent\ B,\ Nonce\ Nb,\ Key\ K,\ Certificate\}$
 $\in\ set\ evs;\ evs \in srb]$
 $\implies (\exists\ Na\ Ver1\ Ver2\ Ver3.$
 $Inputs\ A\ (Card\ A)\ \{Agent\ B,\ Nonce\ Na,\ Nonce\ Nb,\ Nonce\ (Pairkey(A,B)),$
 $Ver1, Ver2, Ver3 \in set\ evs)"$
 $\langle proof \rangle$

lemma *Outpts_honest_A_Card_4:*
 $"[Outpts\ C\ A\ \{Nonce\ Na,\ Crypt\ K\ X\} \in set\ evs;$
 $A \neq Spy;\ evs \in srb]$
 $\implies legalUse(C) \wedge C = (Card\ A) \wedge$
 $Inputs\ A\ (Card\ A)\ (Agent\ A) \in set\ evs"$
 $\langle proof \rangle$

lemma *Outpts_honest_B_Card_7:*
 $"[Outpts\ C\ B\ \{Nonce\ Nb,\ Agent\ A,\ Key\ K,\ Cert1,\ Cert2\} \in set\ evs;$
 $B \neq Spy;\ evs \in srb]$
 $\implies legalUse(C) \wedge C = (Card\ B) \wedge$
 $(\exists\ Na.\ Inputs\ B\ (Card\ B)\ \{Agent\ A,\ Nonce\ Na\} \in set\ evs)"$
 $\langle proof \rangle$

lemma *Outpts_honest_A_Card_10:*
 $"[Outpts\ C\ A\ \{Agent\ B,\ Nonce\ Nb,\ Key\ K,\ Certificate\} \in set\ evs;$
 $A \neq Spy;\ evs \in srb]$
 $\implies legalUse\ (C) \wedge C = (Card\ A) \wedge$
 $(\exists\ Na\ Pk\ Ver1\ Ver2\ Ver3.$
 $Inputs\ A\ (Card\ A)\ \{Agent\ B,\ Nonce\ Na,\ Nonce\ Nb,\ Pk,$
 $Ver1, Ver2, Ver3 \in set\ evs)"$
 $\langle proof \rangle$

lemma *Outpts_which_Card_4:*
 $"[Outpts\ (Card\ A)\ A\ \{Nonce\ Na,\ Crypt\ K\ X\} \in set\ evs;\ evs \in srb]$
 $\implies Inputs\ A\ (Card\ A)\ (Agent\ A) \in set\ evs"$
 $\langle proof \rangle$

lemma *Outpts_which_Card_7:*

"[[Outpts (Card B) B {Nonce Nb, Agent A, Key K, Cert1, Cert2}
 ∈ set evs; evs ∈ srb]]
 ⇒ ∃ Na. Inputs B (Card B) {Agent A, Nonce Na} ∈ set evs"
 ⟨proof⟩

lemma *Outpts_which_Card_10:*

"[[Outpts (Card A) A {Agent B, Nonce Nb, Key K, Certificate } ∈ set evs;
 evs ∈ srb]]
 ⇒ ∃ 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 ∈ srb]]
 ⇒ Certificate = (Crypt (crdK (Card A)) (Nonce Na))"
 ⟨proof⟩

lemma *Outpts_B_Card_form_7:*

"[[Outpts (Card B) B {Nonce Nb, Agent A, Key K, Cert1, Cert2}
 ∈ set evs; evs ∈ srb]]
 ⇒ ∃ 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 {Agent B, Nonce Nb, Key K, Certificate}
 ∈ set evs; evs ∈ srb]]
 ⇒ K = sesK(Nb, pairK(A,B)) ∧
 Certificate = (Crypt (pairK(A,B)) (Nonce Nb))"
 ⟨proof⟩

lemma *Outpts_A_Card_form_bis:*

"[[Outpts (Card A') A' {Agent B', Nonce Nb', Key (sesK(Nb, pairK(A,B))),
 Certificate} ∈ set evs;
 evs ∈ srb]]
 ⇒ A' = A ∧ B' = B ∧ Nb = Nb' ∧
 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 ∈ srb ] ]
⇒ Cert3 = Crypt (crdK (Card A)) (Nonce Na)"
⟨proof⟩
```

lemma *Inputs_Card_legalUse:*

```
"[[ Inputs A (Card A) X ∈ set evs; evs ∈ srb ] ] ⇒ legalUse(Card A)"
⟨proof⟩
```

lemma *Outpts_Card_legalUse:*

```
"[[ Outpts (Card A) A X ∈ set evs; evs ∈ srb ] ] ⇒ legalUse(Card A)"
⟨proof⟩
```

lemma *Inputs_Card:* "[[Inputs A C X ∈ set evs; A ≠ Spy; evs ∈ srb]]
 ⇒ C = (Card A) ∧ legalUse(C)"
 ⟨proof⟩

lemma *Outpts_Card:* "[[Outpts C A X ∈ set evs; A ≠ Spy; evs ∈ srb]]
 ⇒ 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 ∈ srb ] ]
⇒ C = (Card A) ∧ legalUse(Card A)"
⟨proof⟩
```

lemma *Inputs_Card_Spy:*

```
"[[ Inputs Spy C X ∈ set evs ∨ Outpts C Spy X ∈ set evs; evs ∈ srb ] ]
⇒ 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 ∈ srb] ⇒ A=A'"]

⟨proof⟩

lemma *Outpts_B_Card_unique_nonce:*

"[[Outpts (Card B) B {Nonce Nb, Agent A, Key SK, Cert1, Cert2} ∈ set
 evs;
 Outpts (Card B') B' {Nonce Nb, Agent A', Key SK', Cert1', Cert2'} ∈
 set evs;
 evs ∈ srb] ⇒ B=B' ∧ A=A' ∧ SK=SK' ∧ Cert1=Cert1' ∧ Cert2=Cert2'"]

⟨proof⟩

lemma *Outpts_B_Card_unique_key:*

"[[Outpts (Card B) B {Nonce Nb, Agent A, Key SK, Cert1, Cert2} ∈ set
 evs;
 Outpts (Card B') B' {Nonce Nb', Agent A', Key SK, Cert1', Cert2'} ∈
 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]

```

$\langle ML \rangle$

lemma *Spy_parts_keys [simp]*: "evs \in srb \implies
 (Key (shrK P) \in parts (knows Spy evs)) = (Card P \in cloned) \wedge
 (Key (pin P) \in parts (knows Spy evs)) = (P \in bad \vee Card P \in cloned) \wedge
 (Key (crdK C) \in parts (knows Spy evs)) = (C \in cloned) \wedge
 (Key (pairK(A,B)) \in parts (knows Spy evs)) = (Card B \in cloned)"
 $\langle proof \rangle$

lemma *Spy_analz_shrK [simp]*: "evs \in srb \implies
 (Key (shrK P) \in analz (knows Spy evs)) = (Card P \in cloned)"
 $\langle proof \rangle$

lemma *Spy_analz_crdK [simp]*: "evs \in srb \implies
 (Key (crdK C) \in analz (knows Spy evs)) = (C \in cloned)"
 $\langle proof \rangle$

lemma *Spy_analz_pairK [simp]*: "evs \in srb \implies
 (Key (pairK(A,B)) \in analz (knows Spy evs)) = (Card B \in cloned)"
 $\langle proof \rangle$

lemma *analz_image_Key_Un_Nonce*: "analz (Key'K \cup Nonce'N) = Key'K \cup Nonce'N"
 $\langle proof \rangle$

$\langle ML \rangle$

lemma *analz_image_freshK [rule_format]*:
 "evs \in srb $\implies \quad \forall K KK.$
 (Key K \in analz (Key'KK \cup (knows Spy evs))) =
 (K \in KK \vee Key K \in analz (knows Spy evs))"
 $\langle proof \rangle$

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))"
 $\langle proof \rangle$

lemma *Na_Nb_certificate_authentic:*

"[[Crypt (pairK(A,B)) {Nonce Na, Nonce Nb} ∈ parts (knows Spy evs);
 ¬illegalUse(Card B);
 evs ∈ srb]]
 ⇒ Outputs (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 *Nb_certificate_authentic:*

"[[Crypt (pairK(A,B)) (Nonce Nb) ∈ parts (knows Spy evs);
 B ≠ Spy; ¬illegalUse(Card A); ¬illegalUse(Card B);
 evs ∈ srb]]
 ⇒ Outputs (Card A) A {Agent B, Nonce Nb, Key (sesK(Nb,pairK(A,B)))},

Crypt (pairK(A,B)) (Nonce Nb) ∈ set evs"

⟨proof⟩

lemma *Outputs_A_Card_imp_pairK_parts:*

"[[Outputs (Card A) A {Agent B, Nonce Nb,
 Key K, Certificate} ∈ set evs;
 evs ∈ srb]]
 ⇒ ∃ Na. Crypt (pairK(A,B)) {Nonce Na, Nonce Nb} ∈ parts (knows Spy
 evs)"
 ⟨proof⟩

lemma *Nb_certificate_authentic_bis:*

"[[Crypt (pairK(A,B)) (Nonce Nb) ∈ parts (knows Spy evs);
 B ≠ Spy; ¬illegalUse(Card B);
 evs ∈ srb]]
 ⇒ ∃ Na. Outputs (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:*

"[[Crypt (shrK A) {Nonce Pk, Agent B} ∈ parts (knows Spy evs);
 Card A ∉ cloned; evs ∈ srb]]
 ⇒ Pk = Pairkey(A,B) ∧
 Says Server A {Nonce Pk,
 Crypt (shrK A) {Nonce Pk, Agent B}}
 ∈ set evs"

<proof>

lemma *sesK_authentic*:

```
"[[ Key (sesK(Nb,pairK(A,B))) ∈ parts (knows Spy evs);
   A ≠ Spy; B ≠ Spy; ¬illegalUse(Card A); ¬illegalUse(Card B);
   evs ∈ srb ]]
⇒ Notes Spy {Key (sesK(Nb,pairK(A,B))), Nonce Nb, Agent A, Agent B}

   ∈ set evs"
```

<proof>

lemma *Confidentiality*:

```
"[[ Notes Spy {Key (sesK(Nb,pairK(A,B))), Nonce Nb, Agent A, Agent B}

   ∉ set evs;
   A ≠ Spy; B ≠ Spy; ¬illegalUse(Card A); ¬illegalUse(Card B);
   evs ∈ srb ]]
⇒ Key (sesK(Nb,pairK(A,B))) ∉ analz (knows Spy evs)"
```

<proof>

lemma *Confidentiality_B*:

```
"[[ Outpts (Card B) B {Nonce Nb, Agent A, Key K, Cert1, Cert2}
   ∈ set evs;
   Notes Spy {Key K, Nonce Nb, Agent A, Agent B} ∉ set evs;
   A ≠ Spy; B ≠ Spy; ¬illegalUse(Card A); Card B ∉ cloned;
   evs ∈ srb ]]
⇒ Key K ∉ analz (knows Spy evs)"
```

<proof>

lemma *A_authenticates_B*:

```
"[[ Outpts (Card A) A {Agent B, Nonce Nb, Key K, Certificate} ∈ set evs;
   ¬illegalUse(Card B);
   evs ∈ srb ]]
⇒ ∃ Na. Outpts (Card B) B {Nonce Nb, Agent A, 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}}"
```


$\in \text{set evs};$
 $\neg \text{illegalUse}(\text{Card } B);$
 $\text{evs} \in \text{srb} \parallel$
 $\implies \text{Outpts}(\text{Card } B) B \{ \text{Nonce } Nb, \text{Agent } A, \text{Key}(\text{sesK}(Nb, \text{pairK}(A, B))),$
 $\text{Crypt}(\text{pairK}(A, B)) \{ \text{Nonce } Na, \text{Nonce } Nb \},$
 $\text{Crypt}(\text{pairK}(A, B))(\text{Nonce } Nb) \} \in \text{set evs}"$

<proof>

lemma *A_authenticates_B_bis:*

$" \parallel \text{Outpts}(\text{Card } A) A \{ \text{Agent } B, \text{Nonce } Nb, \text{Key } K, \text{Cert2} \} \in \text{set evs};$
 $\neg \text{illegalUse}(\text{Card } B);$
 $\text{evs} \in \text{srb} \parallel$
 $\implies \exists \text{Cert1}. \text{Outpts}(\text{Card } B) B \{ \text{Nonce } Nb, \text{Agent } A, \text{Key } K, \text{Cert1}, \text{Cert2} \}$

$\in \text{set evs}"$

<proof>

lemma *B_authenticates_A:*

$" \parallel \text{Gets } B(\text{Crypt}(\text{pairK}(A, B))(\text{Nonce } Nb)) \in \text{set evs};$
 $B \neq \text{Spy}; \neg \text{illegalUse}(\text{Card } A); \neg \text{illegalUse}(\text{Card } B);$
 $\text{evs} \in \text{srb} \parallel$
 $\implies \text{Outpts}(\text{Card } A) A \{ \text{Agent } B, \text{Nonce } Nb,$
 $\text{Key}(\text{sesK}(Nb, \text{pairK}(A, B))), \text{Crypt}(\text{pairK}(A, B))(\text{Nonce } Nb) \} \in \text{set evs}"$

<proof>

lemma *B_authenticates_A_bis:*

$" \parallel \text{Outpts}(\text{Card } B) B \{ \text{Nonce } Nb, \text{Agent } A, \text{Key } K, \text{Cert1}, \text{Cert2} \} \in \text{set evs};$
 $\text{Gets } B(\text{Cert2}) \in \text{set evs};$
 $B \neq \text{Spy}; \neg \text{illegalUse}(\text{Card } A); \neg \text{illegalUse}(\text{Card } B);$
 $\text{evs} \in \text{srb} \parallel$
 $\implies \text{Outpts}(\text{Card } A) A \{ \text{Agent } B, \text{Nonce } Nb, \text{Key } K, \text{Cert2} \} \in \text{set evs}"$

<proof>

lemma *Confidentiality_A:*

$" \parallel \text{Outpts}(\text{Card } A) A \{ \text{Agent } B, \text{Nonce } Nb,$
 $\text{Key } K, \text{Certificate} \} \in \text{set evs};$
 $\text{Notes Spy} \{ \text{Key } K, \text{Nonce } Nb, \text{Agent } A, \text{Agent } B \} \notin \text{set evs};$
 $A \neq \text{Spy}; B \neq \text{Spy}; \neg \text{illegalUse}(\text{Card } A); \neg \text{illegalUse}(\text{Card } B);$
 $\text{evs} \in \text{srb} \parallel$
 $\implies \text{Key } K \notin \text{analz}(\text{knows Spy evs})"$

<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]]

$\Rightarrow Pk = \text{Pairkey}(A,B) \wedge$
 $\text{Says Server } A \{ \text{Nonce } (\text{Pairkey}(A,B)),$
 $\text{Crypt } (\text{shrK } A) \{ \text{Nonce } (\text{Pairkey}(A,B)), \text{Agent } B \} \}$
 $\in \text{set evs "}$
 $\langle \text{proof} \rangle$

lemma *Na_Nb_certificate_authentic_A_Card:*
 $"[\text{Inputs } A \text{ (Card } A)$
 $\{ \text{Agent } B, \text{Nonce } Na, \text{Nonce } Nb, \text{Nonce } Pk,$
 $\text{Cert1, Crypt } (\text{pairK}(A,B)) \{ \text{Nonce } Na, \text{Nonce } Nb \}, \text{Cert3} \} \in \text{set evs};$
 $A \neq \text{Spy}; \neg \text{illegalUse}(\text{Card } B); \text{evs} \in \text{srb}]$
 $\Rightarrow \text{Outpts (Card } B) B \{ \text{Nonce } Nb, \text{Agent } A, \text{Key } (\text{sesK}(Nb, \text{pairK } (A, B))),$
 $\text{Crypt } (\text{pairK}(A,B)) \{ \text{Nonce } Na, \text{Nonce } Nb \},$
 $\text{Crypt } (\text{pairK}(A,B)) (\text{Nonce } Nb) \}$
 $\in \text{set evs "}$
 $\langle \text{proof} \rangle$

lemma *Na_authentic_A_Card:*
 $"[\text{Inputs } A \text{ (Card } A)$
 $\{ \text{Agent } B, \text{Nonce } Na, \text{Nonce } Nb, \text{Nonce } Pk,$
 $\text{Cert1, Cert2, Cert3} \} \in \text{set evs};$
 $A \neq \text{Spy}; \text{evs} \in \text{srb}]$
 $\Rightarrow \text{Outpts (Card } A) A \{ \text{Nonce } Na, \text{Cert3} \}$
 $\in \text{set evs "}$
 $\langle \text{proof} \rangle$

lemma *Inputs_A_Card_9_authentic:*
 $"[\text{Inputs } A \text{ (Card } A)$
 $\{ \text{Agent } B, \text{Nonce } Na, \text{Nonce } Nb, \text{Nonce } Pk,$
 $\text{Crypt } (\text{shrK } A) \{ \text{Nonce } Pk, \text{Agent } B \},$
 $\text{Crypt } (\text{pairK}(A,B)) \{ \text{Nonce } Na, \text{Nonce } Nb \}, \text{Cert3} \} \in \text{set evs};$
 $A \neq \text{Spy}; \text{Card } A \notin \text{cloned}; \neg \text{illegalUse}(\text{Card } B); \text{evs} \in \text{srb}]$
 $\Rightarrow \text{Says Server } A \{ \text{Nonce } Pk, \text{Crypt } (\text{shrK } A) \{ \text{Nonce } Pk, \text{Agent } B \} \}$
 $\in \text{set evs} \wedge$
 $\text{Outpts (Card } B) B \{ \text{Nonce } Nb, \text{Agent } A, \text{Key } (\text{sesK}(Nb, \text{pairK } (A, B))),$
 $\text{Crypt } (\text{pairK}(A,B)) \{ \text{Nonce } Na, \text{Nonce } Nb \},$
 $\text{Crypt } (\text{pairK}(A,B)) (\text{Nonce } Nb) \}$
 $\in \text{set evs} \wedge$
 $\text{Outpts (Card } A) A \{ \text{Nonce } Na, \text{Cert3} \}$
 $\in \text{set evs "}$
 $\langle \text{proof} \rangle$

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.
 Gets A {Nonce (Pairkey (A, B)), Cert1} ∈ set evs ∧
 Gets A {Nonce Nb, Cert2} ∈ set evs"
 <proof>

lemma *Outpts_Server_not_evs*:
 "evs ∈ srb ⇒ 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 ∈ srb]
 ⇒ ∃ 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 ∈ srb ]
⇒ ∃ Cert2 K.
  Outpts (Card B) B {Nonce Nb, Agent A, 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 ∈ srb ]
⇒ ∃ K Nb.
  Outpts (Card A) A {Agent B, Nonce Nb, Key K, Certificate} ∈ set evs"
⟨proof⟩
```

end

26 Extensions to Standard Theories

theory *Extensions* imports "../Event" begin

26.1 Extensions to Theory *Set*

lemma *eq*: "[| !!x. x:A ==> x:B; !!x. x:B ==> x:A |] ==> A=B"
 ⟨proof⟩

lemma *insert_Un*: "P ({x} Un A) ==> P (insert x A)"
 ⟨proof⟩

lemma *in_sub*: "x:A ==> {x}<=A"
 ⟨proof⟩

26.2 Extensions to Theory *List*

26.2.1 "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⟩

26.3 Extensions to Theory *Message*

26.3.1 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]

```

26.3.2 extract the agent number of an Agent message

```

consts agt_nb :: "msg => agent"

```

```

recdef agt_nb "measure size"
"agt_nb (Agent A) = A"

```

26.3.3 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]: "~ is_MPair (Key K)"
<proof>

```

```

lemma Nonce_isnt_MPair [iff]: "~ is_MPair (Nonce n)"
<proof>

```

```

lemma Hash_isnt_MPair [iff]: "~ is_MPair (Hash X)"
<proof>

```

```

lemma Crypt_isnt_MPair [iff]: "~ is_MPair (Crypt K X)"
<proof>

```

abbreviation

```

not_MPair :: "msg => bool" where
"not_MPair X == ~ 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} ~:H"
```

```
declare has_no_pair_def [simp]
```

26.3.4 well-foundedness of messages

```
lemma wf_Crypt1 [iff]: "Crypt K X ~= X"
<proof>
```

```
lemma wf_Crypt2 [iff]: "X ~= 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 ~:parts {X}"
<proof>
```

26.3.5 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>
```

26.3.6 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>

26.3.7 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>

26.3.8 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>

26.3.9 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>
```

26.3.10 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>
```

26.3.11 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>
```

26.3.12 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>
```

26.3.13 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]
```

26.4 Extensions to Theory *Event*

26.4.1 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>
```

26.4.2 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>
```

26.4.3 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"
```

26.4.4 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>

```

26.4.5 knows' is finite

```

lemma finite_knows' [iff]: "finite (knows' A evs)"
<proof>

```

26.4.6 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>

```

26.4.7 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"
```

26.4.8 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>
```

26.4.9 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>

26.4.10 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>

```

26.4.11 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>

26.4.12 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>

26.4.13 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
```

27 Decomposition of Analz into two parts

```
theory Analz imports Extensions begin
```

decomposition of *analz* into two parts: *pparts* (for pairs) and *analz* of *kparts*

27.1 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"
```

27.2 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>

27.3 facts about pparts and parts

lemma pparts_no_Nonce [dest]: "[| X:pparts {Y}; Nonce n ~:parts {Y} |]
==> Nonce n ~:parts {X}"

<proof>

27.4 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>

27.5 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*"

27.6 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>

27.7 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>

27.8 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>
```

27.9 *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
```

28 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"

```

28.1 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>
```

28.2 guarded sets

```
constdefs Guard :: "nat => key set => msg set => bool"
"Guard n Ks H == ALL X. X:H --> X:guard n Ks"
```

28.3 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 {X} --> invKey K:Ks"
 <proof>

lemma Crypt_guard_invKey [rule_format]: "[| Crypt K Y:guard n Ks; Nonce n:kparts {Y} |] ==> invKey K:Ks"
 <proof>

28.4 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>

28.5 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"

28.6 basic facts about crypt_nb

lemma $\text{non_empty_crypt_msg}$: " $\text{Crypt } K \ Y:\text{parts } \{X\} \implies \text{crypt_nb } X \neq 0$ "
 $\langle \text{proof} \rangle$

28.7 number of Crypt's in a message list

consts $\text{cnb} :: \text{"msg list} \Rightarrow \text{nat}"$

recdef cnb "measure size"
 $\text{"cnb } [] = 0"$
 $\text{"cnb } (X\#l) = \text{crypt_nb } X + \text{cnb } l"$

28.8 basic facts about cnb

lemma cnb_app [simp]: " $\text{cnb } (l @ l') = \text{cnb } l + \text{cnb } l'$ "
 $\langle \text{proof} \rangle$

lemma mem_cnb_minus : " $x \text{ mem } l \implies \text{cnb } l = \text{crypt_nb } x + (\text{cnb } l - \text{crypt_nb } x)$ "
 $\langle \text{proof} \rangle$

lemmas $\text{mem_cnb_minus_substI} = \text{mem_cnb_minus } [\text{THEN } \text{ssubst}]$

lemma cnb_minus [simp]: " $x \text{ mem } l \implies \text{cnb } (\text{remove } l \ x) = \text{cnb } l - \text{crypt_nb } x$ "
 $\langle \text{proof} \rangle$

lemma parts_cnb : " $Z:\text{parts } (\text{set } l) \implies$
 $\text{cnb } l = (\text{cnb } l - \text{crypt_nb } Z) + \text{crypt_nb } Z$ "
 $\langle \text{proof} \rangle$

lemma non_empty_crypt : " $\text{Crypt } K \ Y:\text{parts } (\text{set } l) \implies \text{cnb } l \neq 0$ "
 $\langle \text{proof} \rangle$

28.9 list of kparts

lemma kparts_msg_set : " $\text{EX } l. \text{kparts } \{X\} = \text{set } l \ \& \ \text{cnb } l = \text{crypt_nb } X$ "
 $\langle \text{proof} \rangle$

lemma kparts_set : " $\text{EX } l'. \text{kparts } (\text{set } l) = \text{set } l' \ \& \ \text{cnb } l' = \text{cnb } l$ "
 $\langle \text{proof} \rangle$

28.10 list corresponding to "decrypt"

constdefs $\text{decrypt}' :: \text{"msg list} \Rightarrow \text{key} \Rightarrow \text{msg} \Rightarrow \text{msg list}"$
 $\text{"decrypt}' } l \ K \ Y == Y \ \# \ \text{remove } l \ (\text{Crypt } K \ Y)"$

declare $\text{decrypt}'_def$ [simp]

28.11 basic facts about $\text{decrypt}'$

lemma decrypt_minus : " $\text{decrypt } (\text{set } l) \ K \ Y \leq \text{set } (\text{decrypt}' } l \ K \ Y)"$
 $\langle \text{proof} \rangle$

28.12 if the analyse of a finite guarded set gives n then it must also gives one of the keys of Ks 273

28.12 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>
```

28.13 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

28.14 Extensions to Theory Public

```
declare initState.simps [simp del]
```

28.14.1 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>
```

28.14.2 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>
```

28.14.3 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>

28.14.4 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>

28.14.5 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>

28.14.6 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>
```

28.14.7 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>
```

28.15 Proofs About Guarded Messages

28.15.1 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>
```

28.15.2 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>
```

28.15.3 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

```

29 Lists of Messages and Lists of Agents

```
theory List_Msg imports Extensions begin
```

29.1 Implementation of Lists by Messages

29.1.1 nil is represented by any message which is not a pair

```

abbreviation (input)
  cons :: "msg => msg => msg" where
  "cons x l == {|x,l|}"

```

29.1.2 induction principle

```

lemma lmsg_induct: "[| !!x. not_MPair x ==> P x; !!x l. P l ==> P (cons x
l) |]
==> P l"
<proof>

```

29.1.3 head

```

consts head :: "msg => msg"

recdef head "measure size"
"head (cons x l) = x"

```

29.1.4 tail

```

consts tail :: "msg => msg"

recdef tail "measure size"
"tail (cons x l) = l"

```

29.1.5 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>
```

29.1.6 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"
```

29.1.7 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>
```

29.1.8 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>
```

29.1.9 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"
```

29.1.10 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>

```

29.1.11 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>

```

29.1.12 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>

```

29.2 Agent Lists

29.2.1 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"

```

29.2.2 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

```

30 Protocol P1

```

theory P1 imports Guard_Public List_Msg begin

```

30.1 Protocol Definition

30.1.1 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>

```

30.1.2 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>

```

30.1.3 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>

30.1.4 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>

30.1.5 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>

30.1.6 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>
```

30.1.7 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>
```

30.1.8 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"
```

30.1.9 Composition of Traces

```
lemma "evs':p1 ==>
evs:p1 & (ALL n. Nonce n:used evs' --> Nonce n ~:used evs) -->
evs'@evs : p1"
<proof>
```

30.1.10 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"
```

30.1.11 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>

30.1.12 offers of an offer list

```

constdefs offer_nonces :: "msg => msg set"
"offer_nonces L == {X. X:parts {L} & (EX n. X = Nonce n)}"
```

30.1.13 the originator can get the offers

lemma "L:valid A n B ==> offer_nonces L <= analz (insert L (initState A))"
 <proof>

30.1.14 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"
```

30.1.15 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>
```

30.1.16 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]

```

30.1.17 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>

```

30.2 properties of protocol P1

publicly verifiable forward integrity: anyone can verify the validity of an offer list

30.2.1 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>

```

30.2.2 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>

30.2.3 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>
```

30.2.4 declarations for tactics

```
declare knows_Spy_partsEs [elim]
declare Fake_parts_insert [THEN subsetD, dest]
declare initState.simps [simp del]
```

30.2.5 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>
```

30.2.6 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>

30.2.7 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>

30.2.8 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>

30.2.9 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>
```

30.2.10 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>
```

30.2.11 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>
```

30.2.12 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>
```

30.2.13 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>

30.2.14 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

31 Protocol P2

theory *P2* imports *Guard_Public List_Msg* begin

31.1 Protocol Definition

Like P1 except the definitions of *chain*, *shop*, *next_shop* and *nonce*

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= 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>

```

31.1.2 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>

```

31.1.3 nonce used in an offer

```

consts nonce :: "msg => msg"

recdef nonce "measure size"
"nonce {|Crypt K {|B,ofr,CryptH|},m2|} = 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 {|m1,Hash {|headL,Agent C|}|} = C"

lemma "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>
```

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 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"

```

31.1.9 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"

```

31.1.10 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>

31.1.11 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"

```

31.2 Properties of Protocol P2

same as *P1_Prop* except that publicly verifiable forward integrity is replaced by forward privacy

31.3 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>

31.4 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>

31.5 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>

31.6 declarations for tactics

declare knows_Spy_partsEs [elim]
declare Fake_parts_insert [THEN subsetD, dest]
declare initState.simps [simp del]

31.7 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>

31.8 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>

31.9 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>

31.10 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>

31.11 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>
```

31.12 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>
```

31.13 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>
```

31.14 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>

```

31.15 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>

```

31.16 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>

```

31.17 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

```

32 Needham-Schroeder-Lowe Public-Key Protocol

```
theory Guard_NS_Public imports Guard_Public begin
```

32.1 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))"

```

32.2 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"
```

32.3 declarations for tactics

```
declare knows_Spy_partsEs [elim]
declare Fake_parts_insert [THEN subsetD, dest]
declare initState.simps [simp del]
```

32.4 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>
```

32.5 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⟩
```

32.6 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⟩
```

32.7 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⟩
```

32.8 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

33 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"
```

33.1 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*})"

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_extand*: "[| *X*:*guardK* *n* *Ks*; *Ks* <= *Ks'*;
 [| *K*:*Ks'*; *K* ~ :*Ks* |] ==> Key *K* ~ :parts {*X*} |] ==> *X*:*guardK* *n* *Ks'*"
 ⟨*proof*⟩

33.2 guarded sets

constdefs *GuardK* :: "nat => key set => msg set => bool"
 "*GuardK* *n* *Ks* *H* == ALL *X*. *X*:*H* --> *X*:*guardK* *n* *Ks*"

33.3 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>
```

33.4 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>
```

33.5 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"
```

33.6 basic facts about crypt_nb

```
lemma non_empty_crypt_msg: "Crypt K Y:parts {X} ==> crypt_nb X ≠ 0"
<proof>
```

33.7 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"
```

33.8 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⟩
```

33.9 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⟩
```

33.10 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]
```

33.11 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"

```

33.12 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>

```

33.13 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 ∉ used evs ==> shrK B ≠ K"
  <proof>

```

```

lemmas shrK_sym_neq = shrK_neq [THEN not_sym]
declare shrK_sym_neq [simp]

```

33.14 Fresh nonces

```

lemma Nonce_notin_initState [iff]: "Nonce N ∉ parts (initState B)"
  <proof>

```

```

lemma Nonce_notin_used_empty [simp]: "Nonce N ∉ used []"
  <proof>

```

33.15 Supply fresh nonces for possibility theorems.

```

lemma Nonce_supply_lemma: "∃ N. ALL n. N ≤ n --> Nonce n ∉ used evs"
  <proof>

```

```

lemma Nonce_supply1: "∃ N. Nonce N ∉ used evs"
  <proof>

```

lemma *Nonce_supply2*: " $\exists N N'. \text{Nonce } N \notin \text{used evs} \ \& \ \text{Nonce } N' \notin \text{used evs}'$
 $\& \ N \neq N'$ "
 <proof>

lemma *Nonce_supply3*: " $\exists N N' N''. \text{Nonce } N \notin \text{used evs} \ \& \ \text{Nonce } N' \notin \text{used evs}'$
 $\&$
 $\text{Nonce } N'' \notin \text{used evs}'' \ \& \ N \neq N' \ \& \ N' \neq N'' \ \& \ N \neq N''$ "
 <proof>

lemma *Nonce_supply*: " $\text{Nonce } (@ \ N. \text{Nonce } N \notin \text{used evs}) \notin \text{used evs}$ "
 <proof>

Unlike the corresponding property of nonces, we cannot prove *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.

33.16 Specialized Rewriting for Theorems About *analz* and Image

lemma *subset_Compl_range*: " $A \leq - \ (\text{range } \text{shrK}) \implies \text{shrK } x \notin A$ "
 <proof>

lemma *insert_Key_singleton*: " $\text{insert } (\text{Key } K) \ H = \text{Key } ' \ \{K\} \cup H$ "
 <proof>

lemma *insert_Key_image*: " $\text{insert } (\text{Key } K) \ (\text{Key}'KK \cup C) = \text{Key}'(\text{insert } K \ KK) \cup C$ "
 <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*:
 " $(\text{Key } K \in \text{analz } (\text{Key}'nE \cup H)) \implies (K \in nE \mid \text{Key } K \in \text{analz } H) \implies$
 $(\text{Key } K \in \text{analz } (\text{Key}'nE \cup H)) = (K \in nE \mid \text{Key } K \in \text{analz } H)$ "
 <proof>

33.17 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

34 lemmas on guarded messages for protocols with symmetric keys

theory *Guard_Shared* imports *Guard GuardK Shared* begin

34.1 Extensions to Theory *Shared*

```

declare initState.simps [simp del]

```

34.1.1 a little abbreviation

abbreviation

```

  Ciph :: "agent => msg => msg" where
  "Ciph A X == Crypt (shrK A) X"

```

34.1.2 agent associated to a key

```

constdefs agt :: "key => agent"
"agt K == @A. K = shrK A"

```

```

lemma agt_shrK [simp]: "agt (shrK A) = A"
⟨proof⟩

```

34.1.3 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>

34.1.4 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>
```

34.1.5 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>
```

34.2 Proofs About Guarded Messages

34.2.1 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>
```

34.2.2 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>
```

34.2.3 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>
```

34.2.4 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

```

35 Otway-Rees Protocol

theory Guard_OtwayRees **imports** Guard_Shared **begin**

35.1 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|}"
```

35.2 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"
```

35.3 declarations for tactics

```
declare knows_Spy_partsEs [elim]
declare Fake_parts_insert [THEN subsetD, dest]
declare initState.simps [simp del]
```

35.4 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>

35.5 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>

35.6 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>

35.7 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

36 Yahalom Protocol

theory *Guard_Yahalom* **imports** *Guard_Shared* **begin**

36.1 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)|}"
```

36.2 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"
```

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 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>

36.5 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>

36.6 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>

36.7 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>

36.8 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>

36.9 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>

36.10 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>

36.11 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

37 Other Protocol-Independent Results

theory Proto imports Guard_Public begin

37.1 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))"
```

37.2 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)"
```

37.3 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>
```

37.4 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"
```

37.5 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>

37.6 types

types *keyfun* = "rule => subs => nat => event list => key set"

types *secfun* = "rule => nat => subs => key set => msg"

37.7 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>
```

37.8 safe keys

```
constdefs 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>
```

37.9 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>
```

37.10 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>
```

37.11 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>
```

37.12 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>
```

37.13 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>

```

37.14 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>

```

37.15 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>

```

37.16 guardedness for NSL

```

lemma "uniq ns secret ==> preserv ns keys n Ks"
  <proof>

```

37.17 unicity for NSL

```

lemma "uniq' ns inf secret"
  <proof>

```

end