

Security Protocols

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

May 22, 2012

Contents

1	Theory of Agents and Messages for Security Protocols	13
1.0.1	Inductive Definition of All Parts” of a Message	14
1.0.2	Inverse of keys	14
1.1	keysFor operator	14
1.2	Inductive relation ”parts”	15
1.2.1	Unions	16
1.2.2	Idempotence and transitivity	17
1.2.3	Rewrite rules for pulling out atomic messages	17
1.3	Inductive relation ”analz”	18
1.3.1	General equational properties	19
1.3.2	Rewrite rules for pulling out atomic messages	19
1.3.3	Idempotence and transitivity	21
1.4	Inductive relation ”synth”	22
1.4.1	Unions	23
1.4.2	Idempotence and transitivity	23
1.4.3	Combinations of parts, analz and synth	24
1.4.4	For reasoning about the Fake rule in traces	25
1.5	HPair: a combination of Hash and MPair	26
1.5.1	Freeness	26
1.5.2	Specialized laws, proved in terms of those for Hash and MPair	26
1.6	The set of key-free messages	27
1.7	Tactics useful for many protocol proofs	28
2	Theory of Events for Security Protocols	30
2.1	Function <i>knows</i>	31
2.2	Knowledge of Agents	32
2.2.1	Useful for case analysis on whether a hash is a spoof or not	35
2.3	Asymmetric Keys	35
2.4	Basic properties of <i>pubK</i> and <i>priEK</i>	37
2.5	”Image” equations that hold for injective functions	37
2.6	Symmetric Keys	37
2.7	Initial States of Agents	39
2.8	Function <i>knows Spy</i>	40
2.9	Fresh Nonces	41
2.10	Supply fresh nonces for possibility theorems	42

2.11	Specialized Rewriting for Theorems About <i>analz</i> and Image . . .	42
2.12	Specialized Methods for Possibility Theorems	43
3	Needham-Schroeder Shared-Key Protocol and the Issues Property	43
3.1	Inductive proofs about <i>ns_shared</i>	45
3.1.1	Forwarding lemmas, to aid simplification	45
3.1.2	Lemmas concerning the form of items passed in messages	46
3.1.3	Session keys are not used to encrypt other session keys . .	47
3.1.4	The session key K uniquely identifies the message	47
3.1.5	Crucial secrecy property: Spy doesn't see the keys sent in NS2	47
3.2	Guarantees available at various stages of protocol	48
3.3	Lemmas for reasoning about predicate "Issues"	50
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.	51
4	The Kerberos Protocol, BAN Version	53
4.1	Lemmas for reasoning about predicate "Issues"	55
4.2	Lemmas concerning the form of items passed in messages	58
4.3	Non-temporal guarantees, explicitly relying on non-occurrence of oops events - refined below by temporal guarantees	60
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	63
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	65
5	The Kerberos Protocol, BAN Version, with Gets event	66
5.1	Lemmas concerning the form of items passed in messages	71
5.2	Non-temporal guarantees, explicitly relying on non-occurrence of oops events - refined below by temporal guarantees	74
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	77
5.4	Combined guarantees of key distribution and non-injective agreement on the session keys	79
6	The Kerberos Protocol, Version IV	80
6.1	Lemmas about lists, for reasoning about Issues	84
6.2	Lemmas about <i>authKeys</i>	84
6.3	Forwarding Lemmas	85
6.4	Lemmas for reasoning about predicate "before"	87
6.5	Regularity Lemmas	88
6.6	Authenticity theorems: confirm origin of sensitive messages . . .	91

6.7	Reliability: friendly agents send something if something else happened	94
6.8	Unicity Theorems	97
6.9	Lemmas About the Predicate <i>AKcryptSK</i>	98
6.10	Secrecy Theorems	101
6.11	Parties authentication: each party verifies "the identity of another party who generated some data" (quoted from Neuman and Ts'o).	107
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	109
7	The Kerberos Protocol, Version IV	114
7.1	Lemmas about reception event	118
7.2	Lemmas about <i>authKeys</i>	118
7.3	Forwarding Lemmas	119
7.4	Regularity Lemmas	120
7.5	Authenticity theorems: confirm origin of sensitive messages . . .	123
7.6	Reliability: friendly agents send something if something else happened	126
7.7	Unicity Theorems	129
7.8	Lemmas About the Predicate <i>AKcryptSK</i>	130
7.9	Secrecy Theorems	133
7.10	2. Parties' strong authentication: non-injective agreement on the session key. The same guarantees also express key distribution, hence their names	139
8	The Kerberos Protocol, Version V	143
8.1	Lemmas about lists, for reasoning about Issues	146
8.2	Lemmas about <i>authKeys</i>	147
8.3	Forwarding Lemmas	148
8.4	Regularity Lemmas	149
8.5	Authenticity theorems: confirm origin of sensitive messages . . .	150
8.6	Reliability: friendly agents send something if something else happened	153
8.7	Unicity Theorems	156
8.8	Lemmas About the Predicate <i>AKcryptSK</i>	156
8.9	Secrecy Theorems	159
8.10	Parties authentication: each party verifies "the identity of another party who generated some data" (quoted from Neuman and Ts'o).	164
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.	166

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.	169
9	The Original Otway-Rees Protocol	171
9.1	Towards Secrecy: Proofs Involving <i>analz</i>	173
9.2	Authenticity properties relating to NA	174
9.3	Authenticity properties relating to NB	177
10	The Otway-Rees Protocol as Modified by Abadi and Needham	179
10.1	Proofs involving <i>analz</i>	181
10.2	Authenticity properties relating to NA	182
10.3	Authenticity properties relating to NB	183
11	The Otway-Rees Protocol: The Faulty BAN Version	184
11.1	For reasoning about the encrypted portion of messages	185
11.2	Proofs involving <i>analz</i>	186
11.3	Attempting to prove stronger properties	188
12	Bella's version of the Otway-Rees protocol	188
12.1	Proofs involving <i>analz</i>	191
13	The Woo-Lam Protocol	196
14	The Otway-Bull Recursive Authentication Protocol	198
15	The Yahalom Protocol	207
15.1	Regularity Lemmas for Yahalom	208
15.2	Secrecy Theorems	210
15.2.1	Security Guarantee for A upon receiving YM3	211
15.2.2	Security Guarantees for B upon receiving YM4	211
15.2.3	Towards proving secrecy of Nonce NB	212
15.2.4	The Nonce NB uniquely identifies B's message.	214
15.2.5	A nonce value is never used both as NA and as NB	215
15.3	Authenticating B to A	217
15.4	Authenticating A to B using the certificate <i>Crypt K (Nonce NB)</i>	217
16	The Yahalom Protocol, Variant 2	218
16.1	Inductive Proofs	220
16.2	Crucial Secrecy Property: Spy Does Not See Key <i>KAB</i>	222
16.3	Security Guarantee for A upon receiving YM3	223
16.4	Security Guarantee for B upon receiving YM4	223
16.5	Authenticating B to A	224
16.6	Authenticating A to B	225

17 The Yahalom Protocol: A Flawed Version	226
17.1 Regularity Lemmas for Yahalom	227
17.2 For reasoning about the encrypted portion of messages	227
17.3 Secrecy Theorems	228
17.4 Session keys are not used to encrypt other session keys	228
17.5 Security Guarantee for A upon receiving YM3	229
17.6 Security Guarantees for B upon receiving YM4	230
17.7 The Flaw in the Model	230
17.8 Basic Lemmas	234
17.9 About NRO: Validity for B	235
17.10 About NRR: Validity for A	236
17.11 Proofs About <i>sub_K</i>	236
17.12 Proofs About <i>con_K</i>	237
17.13 Proving fairness	238
18 Conventional protocols: rely on conventional Message, Event and Public – Shared-key protocols	240
19 Verifying the Needham-Schroeder Public-Key Protocol	241
20 Verifying the Needham-Schroeder-Lowe Public-Key Protocol	245
20.1 Authenticity properties obtained from NS2	246
20.2 Authenticity properties obtained from NS2	247
20.3 Overall guarantee for B	248
21 Bijections between natural numbers and other types	248
21.1 Type <i>nat</i> × <i>nat</i>	248
21.2 Type <i>nat</i> + <i>nat</i>	250
21.3 Type <i>int</i>	251
21.4 Type <i>nat list</i>	252
21.5 Finite sets of naturals	253
21.5.1 Preliminaries	253
21.5.2 From sets to naturals	253
21.5.3 From naturals to sets	254
21.5.4 Proof of isomorphism	254
22 The TLS Protocol: Transport Layer Security	255
22.1 Protocol Proofs	260
22.2 Inductive proofs about <i>tls</i>	261
22.2.1 Properties of items found in Notes	261
22.2.2 Protocol goal: if B receives CertVerify, then A sent it	262
22.2.3 Unicity results for PMS, the pre-master-secret	263
22.3 Secrecy Theorems	264
22.3.1 Protocol goal: <i>serverK(Na,Nb,M)</i> and <i>clientK(Na,Nb,M)</i> remain secure	265
22.3.2 Weakening the Oops conditions for leakage of <i>clientK</i>	267
22.3.3 Weakening the Oops conditions for leakage of <i>serverK</i>	268

22.3.4	Protocol goal: if A receives <code>ServerFinished</code> , 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.	269
22.3.5	Protocol goal: if B receives any message encrypted with <code>clientK</code> then A has sent it	270
22.3.6	Protocol goal: if B receives <code>ClientFinished</code> , and if B is able to check a <code>CertVerify</code> from A, then A has used the quoted values PA, PB, etc. Even this one requires A to be uncompromised.	270
23	The Certified Electronic Mail Protocol by Abadi et al.	271
23.1	Proving Confidentiality Results	275
23.2	The Guarantees for Sender and Recipient	278
24	Conventional protocols: rely on conventional Message, Event and Public – Public-key protocols	280
25	Theory of Events for Security Protocols that use smartcards	280
25.1	Function <i>knows</i>	283
25.2	Knowledge of Agents	285
26	Theory of smartcards	289
26.1	Basic properties of <code>shrK</code>	290
26.2	Function "knows"	291
26.3	Fresh nonces	293
26.4	Supply fresh nonces for possibility theorems.	293
26.5	Specialized Rewriting for Theorems About <i>analz</i> and <i>Image</i>	294
26.6	Tactics for possibility theorems	295
27	Original Shoup-Rubin protocol	296
28	Bella's modification of the Shoup-Rubin protocol	320
29	Smartcard protocols: rely on conventional Message and on new EventSC and Smartcard	345
30	Extensions to Standard Theories	345
30.1	Extensions to Theory <i>Set</i>	345
30.2	Extensions to Theory <i>List</i>	345
30.2.1	"remove l x" erase the first element of "l" equal to "x"	345
30.3	Extensions to Theory <i>Message</i>	345
30.3.1	declarations for tactics	345
30.3.2	extract the agent number of an Agent message	346
30.3.3	messages that are pairs	346
30.3.4	well-foundedness of messages	347
30.3.5	lemmas on <i>keysFor</i>	347
30.3.6	lemmas on <i>parts</i>	347
30.3.7	lemmas on <i>synth</i>	348
30.3.8	lemmas on <i>analz</i>	348
30.3.9	lemmas on <i>parts</i> , <i>synth</i> and <i>analz</i>	348

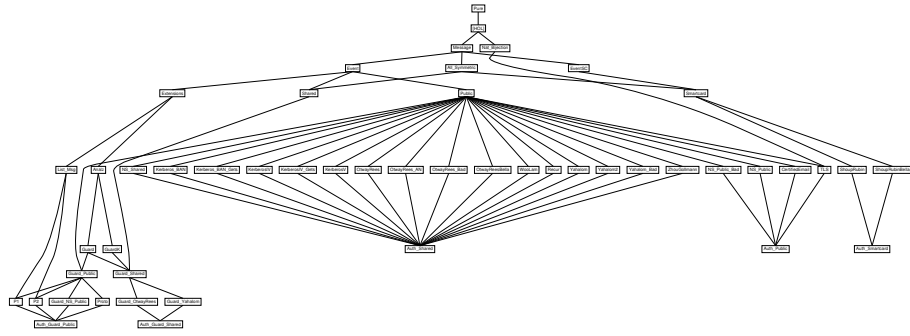
30.3.10	greatest nonce used in a message	349
30.3.11	sets of keys	349
30.3.12	keys a priori necessary for decrypting the messages of G .	349
30.3.13	only the keys necessary for G are useful in analz	350
30.4	Extensions to Theory <i>Event</i>	350
30.4.1	general protocol properties	350
30.4.2	lemma on knows	351
30.4.3	knows without initState	351
30.4.4	decomposition of knows into knows' and initState	351
30.4.5	knows' is finite	352
30.4.6	monotonicity of knows	352
30.4.7	maximum knowledge an agent can have includes messages sent to the agent	352
30.4.8	basic facts about knows_max	353
30.4.9	used without initState	354
30.4.10	monotonicity of used	354
30.4.11	lemmas on used and knows	355
30.4.12	a nonce or key in a message cannot equal a fresh nonce or key	356
30.4.13	message of an event	356
31	Decomposition of Analz into two parts	356
31.1	messages that do not contribute to analz	356
31.2	basic facts about pparts	357
31.3	facts about pparts and parts	358
31.4	facts about pparts and analz	358
31.5	messages that contribute to analz	358
31.6	basic facts about kparts	359
31.7	facts about kparts and parts	360
31.8	facts about kparts and analz	360
31.9	analz is $\text{pparts} + \text{analz}$ of kparts	361
32	Protocol-Independent Confidentiality Theorem on Nonces	361
32.1	basic facts about guard	362
32.2	guarded sets	362
32.3	basic facts about Guard	363
32.4	set obtained by decrypting a message	364
32.5	number of Crypt 's in a message	364
32.6	basic facts about crypt_nb	365
32.7	number of Crypt 's in a message list	365
32.8	basic facts about cnb	365
32.9	list of kparts	365
32.10	list corresponding to "decrypt"	366
32.11	basic facts about $\text{decrypt}'$	366
32.12	if the analyse of a finite guarded set gives n then it must also gives one of the keys of Ks	366
32.13	if the analyse of a finite guarded set and a (possibly infinite) set of keys gives n then it must also gives Ks	367

33 protocol-independent confidentiality theorem on keys	367
33.1 basic facts about <i>guardK</i>	367
33.2 guarded sets	368
33.3 basic facts about <i>GuardK</i>	368
33.4 set obtained by decrypting a message	370
33.5 number of Crypt's in a message	370
33.6 basic facts about <i>crypt_nb</i>	370
33.7 number of Crypt's in a message list	370
33.8 basic facts about <i>cnb</i>	370
33.9 list of kparts	371
33.10 list corresponding to "decrypt"	371
33.11 basic facts about <i>decrypt'</i>	371
33.12 Basic properties of <i>shrK</i>	373
33.13 Function "knows"	374
33.14 Fresh nonces	374
33.15 Supply fresh nonces for possibility theorems.	374
33.16 Specialized Rewriting for Theorems About <i>analz</i> and <i>Image</i>	375
33.17 Tactics for possibility theorems	376
34 lemmas on guarded messages for protocols with symmetric keys	377
34.1 Extensions to Theory <i>Shared</i>	377
34.1.1 a little abbreviation	377
34.1.2 agent associated to a key	377
34.1.3 basic facts about <i>initState</i>	377
34.1.4 sets of symmetric keys	378
34.1.5 sets of good keys	378
34.2 Proofs About Guarded Messages	378
34.2.1 small hack	378
34.2.2 guardedness results on nonces	379
34.2.3 guardedness results on keys	379
34.2.4 regular protocols	380
35 Otway-Rees Protocol	380
35.1 messages used in the protocol	380
35.2 definition of the protocol	381
35.3 declarations for tactics	382
35.4 general properties of <i>or</i>	382
35.5 <i>or</i> is regular	382
35.6 guardedness of <i>KAB</i>	382
35.7 guardedness of <i>NB</i>	383
36 Yahalom Protocol	384
36.1 messages used in the protocol	384
36.2 definition of the protocol	385
36.3 declarations for tactics	385
36.4 general properties of <i>ya</i>	385
36.5 guardedness of <i>KAB</i>	385
36.6 session keys are not symmetric keys	386
36.7 <i>ya2'</i> implies <i>ya1'</i>	386
36.8 uniqueness of <i>NB</i>	386

36.9	ya3' implies ya2'	387
36.10	ya3' implies ya3	387
36.11	guardedness of NB	387
37	Blanqui's "guard" concept: protocol-independent secrecy	388
37.1	Extensions to Theory <i>Public</i>	389
37.1.1	signature	389
37.1.2	agent associated to a key	389
37.1.3	basic facts about <i>initState</i>	389
37.1.4	sets of private keys	389
37.1.5	sets of good keys	390
37.1.6	greatest nonce used in a trace, 0 if there is no nonce	390
37.1.7	function giving a new nonce	390
37.2	Proofs About Guarded Messages	390
37.2.1	small hack necessary because <i>priK</i> is defined as the inverse of <i>pubK</i>	390
37.2.2	guardedness results	391
37.2.3	regular protocols	391
38	Lists of Messages and Lists of Agents	392
38.1	Implementation of Lists by Messages	392
38.1.1	<i>nil</i> is represented by any message which is not a pair	392
38.1.2	induction principle	392
38.1.3	<i>head</i>	392
38.1.4	<i>tail</i>	392
38.1.5	<i>length</i>	392
38.1.6	<i>membership</i>	392
38.1.7	<i>delete an element</i>	392
38.1.8	<i>concatenation</i>	393
38.1.9	<i>replacement</i>	393
38.1.10	<i>ith element</i>	393
38.1.11	<i>insertion</i>	393
38.1.12	<i>truncation</i>	393
38.2	Agent Lists	393
38.2.1	set of well-formed agent-list messages	393
38.2.2	basic facts about agent lists	394
39	Protocol P1	394
39.1	Protocol Definition	394
39.1.1	offer chaining: B chains his offer for A with the head offer of L for sending it to C	394
39.1.2	agent whose key is used to sign an offer	395
39.1.3	nonce used in an offer	395
39.1.4	next shop	395
39.1.5	anchor of the offer list	395
39.1.6	request event	396
39.1.7	propose event	396
39.1.8	protocol	396
39.1.9	Composition of Traces	397
39.1.10	Valid Offer Lists	397

39.1.11	basic properties of valid	397
39.1.12	offers of an offer list	397
39.1.13	the originator can get the offers	397
39.1.14	list of offers	397
39.1.15	list of agents whose keys are used to sign a list of offers	398
39.1.16	builds a trace from an itinerary	398
39.1.17	there is a trace in which the originator receives a valid answer	398
39.2	properties of protocol P1	398
39.2.1	strong forward integrity: except the last one, no offer can be modified	399
39.2.2	insertion resilience: except at the beginning, no offer can be inserted	399
39.2.3	truncation resilience: only shop i can truncate at offer i	399
39.2.4	declarations for tactics	400
39.2.5	get components of a message	400
39.2.6	general properties of $p1$	400
39.2.7	private keys are safe	401
39.2.8	general guardedness properties	401
39.2.9	guardedness of messages	401
39.2.10	Nonce uniqueness	402
39.2.11	requests are guarded	402
39.2.12	propositions are guarded	403
39.2.13	data confidentiality: no one other than the originator can decrypt the offers	404
39.2.14	non repudiability: an offer signed by B has been sent by B	404
40	Protocol P2	406
40.1	Protocol Definition	406
40.1.1	offer chaining: B chains his offer for A with the head offer of L for sending it to C	406
40.1.2	agent whose key is used to sign an offer	406
40.1.3	nonce used in an offer	406
40.1.4	next shop	406
40.1.5	anchor of the offer list	407
40.1.6	request event	407
40.1.7	propose event	407
40.1.8	protocol	408
40.1.9	valid offer lists	408
40.1.10	basic properties of valid	408
40.1.11	list of offers	408
40.2	Properties of Protocol P2	408
40.3	strong forward integrity: except the last one, no offer can be modified	409
40.4	insertion resilience: except at the beginning, no offer can be inserted	409
40.5	truncation resilience: only shop i can truncate at offer i	409
40.6	declarations for tactics	410
40.7	get components of a message	410
40.8	general properties of $p2$	410
40.9	private keys are safe	411

40.10	general guardedness properties	411
40.11	guardedness of messages	412
40.12	Nonce uniqueness	412
40.13	requests are guarded	412
40.14	propositions are guarded	413
40.15	data confidentiality: no one other than the originator can decrypt the offers	414
40.16	forward privacy: only the originator can know the identity of the shops	414
40.17	non repudiability: an offer signed by B has been sent by B	414
41	Needham-Schroeder-Lowe Public-Key Protocol	416
41.1	messages used in the protocol	416
41.2	definition of the protocol	417
41.3	declarations for tactics	417
41.4	general properties of nsp	417
41.5	nonce are used only once	417
41.6	guardedness of NA	418
41.7	guardedness of NB	418
41.8	Agents' Authentication	419
42	Other Protocol-Independent Results	420
42.1	protocols	420
42.2	substitutions	420
42.3	nonces generated by a rule	421
42.4	traces generated by a protocol	421
42.5	general properties	422
42.6	types	422
42.7	introduction of a fresh guarded nonce	422
42.8	safe keys	424
42.9	guardedness preservation	424
42.10	monotonic keyfun	424
42.11	guardedness theorem	424
42.12	useful properties for guardedness	425
42.13	unicity	425
42.14	Needham-Schroeder-Lowe	426
42.15	general properties	427
42.16	guardedness for NSL	428
42.17	unicity for NSL	429
43	Blanqui's "guard" concept: protocol-independent secrecy	431



1 Theory of Agents and Messages for Security Protocols

```

theory Message
imports Main
begin

lemma [simp] : "A  $\cup$  (B  $\cup$  A) = B  $\cup$  A"
by blast

type_synonym
  key = nat

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

specification (invKey)
  invKey [simp]: "invKey (invKey K) = K"
  invKey_symmetric: "all_symmetric --> invKey = id"
  by (rule exI [of _ id], auto)

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

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

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

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

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

syntax
  "_MTuple"      :: "[ 'a, args ] => 'a * 'b"          ("(2{|_,/ _|})")

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

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

definition HPair :: "[msg,msg] => msg" ("(4Hash[_] /_)" [0, 1000]) where

```

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

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

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]:` $X \in H \implies X \in \text{parts } H$
`| Fst:` $\{|X,Y|\} \in \text{parts } H \implies X \in \text{parts } H$
`| Snd:` $\{|X,Y|\} \in \text{parts } H \implies Y \in \text{parts } H$
`| Body:` $\text{Crypt } K X \in \text{parts } H \implies X \in \text{parts } H$

Monotonicity

lemma `parts_mono`: " $G \subseteq H \implies \text{parts}(G) \subseteq \text{parts}(H)$ "
apply `auto`
apply `(erule parts.induct)`
apply `(blast dest: parts.Fst parts.Snd parts.Body)+`
done

Equations hold because constructors are injective.

lemma `Friend_image_eq [simp]`: " $(\text{Friend } x \in \text{Friend}'A) = (x:A)$ "
by `auto`

lemma `Key_image_eq [simp]`: " $(\text{Key } x \in \text{Key}'A) = (x \in A)$ "
by `auto`

lemma `Nonce_Key_image_eq [simp]`: " $(\text{Nonce } x \notin \text{Key}'A)$ "
by `auto`

1.0.2 Inverse of keys

lemma `invKey_eq [simp]`: " $(\text{invKey } K = \text{invKey } K') = (K=K')$ "
by `(metis invKey)`

1.1 keysFor operator

lemma `keysFor_empty [simp]`: " $\text{keysFor } \{\} = \{\}$ "
by `(unfold keysFor_def, blast)`

lemma `keysFor_Un [simp]`: " $\text{keysFor } (H \cup H') = \text{keysFor } H \cup \text{keysFor } H'$ "
by `(unfold keysFor_def, blast)`

lemma `keysFor_UN [simp]`: " $\text{keysFor } (\bigcup_{i \in A}. H i) = (\bigcup_{i \in A}. \text{keysFor } (H i))$ "
by `(unfold keysFor_def, blast)`

Monotonicity

lemma `keysFor_mono`: " $G \subseteq H \implies \text{keysFor}(G) \subseteq \text{keysFor}(H)$ "
by `(unfold keysFor_def, blast)`

```

lemma keysFor_insert_Agent [simp]: "keysFor (insert (Agent A) H) = keysFor
H"
by (unfold keysFor_def, auto)

lemma keysFor_insert_Nonce [simp]: "keysFor (insert (Nonce N) H) = keysFor
H"
by (unfold keysFor_def, auto)

lemma keysFor_insert_Number [simp]: "keysFor (insert (Number N) H) = keysFor
H"
by (unfold keysFor_def, auto)

lemma keysFor_insert_Key [simp]: "keysFor (insert (Key K) H) = keysFor H"
by (unfold keysFor_def, auto)

lemma keysFor_insert_Hash [simp]: "keysFor (insert (Hash X) H) = keysFor
H"
by (unfold keysFor_def, auto)

lemma keysFor_insert_MPair [simp]: "keysFor (insert {|X,Y|} H) = keysFor
H"
by (unfold keysFor_def, auto)

lemma keysFor_insert_Crypt [simp]:
  "keysFor (insert (Crypt K X) H) = insert (invKey K) (keysFor H)"
by (unfold keysFor_def, auto)

lemma keysFor_image_Key [simp]: "keysFor (Key `E) = {}"
by (unfold keysFor_def, auto)

lemma Crypt_imp_invKey_keysFor: "Crypt K X ∈ H ==> invKey K ∈ keysFor H"
by (unfold keysFor_def, blast)

```

1.2 Inductive relation "parts"

```

lemma MPair_parts:
  "[| {|X,Y|} ∈ parts H;
    [| X ∈ parts H; Y ∈ parts H |] ==> P |] ==> P"
by (blast dest: parts.Fst parts.Snd)

```

```

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)"
by blast

```

```

lemmas parts_insertI = subset_insertI [THEN parts_mono, THEN subsetD]

```

```

lemma parts_empty [simp]: "parts{} = {}"
apply safe

```

```

apply (erule parts.induct, blast+)
done

```

```

lemma parts_emptyE [elim!]: "X ∈ parts {} ==> P"
by simp

```

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

```

lemma parts_singleton: "X ∈ parts H ==> ∃ Y ∈ H. X ∈ parts {Y}"
by (erule parts.induct, fast+)

```

1.2.1 Unions

```

lemma parts_Un_subset1: "parts(G) ∪ parts(H) ⊆ parts(G ∪ H)"
by (intro Un_least parts_mono Un_upper1 Un_upper2)

```

```

lemma parts_Un_subset2: "parts(G ∪ H) ⊆ parts(G) ∪ parts(H)"
apply (rule subsetI)
apply (erule parts.induct, blast+)
done

```

```

lemma parts_Un [simp]: "parts(G ∪ H) = parts(G) ∪ parts(H)"
by (intro equalityI parts_Un_subset1 parts_Un_subset2)

```

```

lemma parts_insert: "parts (insert X H) = parts {X} ∪ parts H"
by (metis insert_is_Un parts_Un)

```

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

```

lemma parts_insert2:
  "parts (insert X (insert Y H)) = parts {X} ∪ parts {Y} ∪ parts H"
by (metis Un_commute Un_empty_right Un_insert_right insert_is_Un parts_Un)

```

```

lemma parts_UN_subset1: "(∪ x ∈ A. parts(H x)) ⊆ parts(∪ x ∈ A. H x)"
by (intro UN_least parts_mono UN_upper)

```

```

lemma parts_UN_subset2: "parts(∪ x ∈ A. H x) ⊆ (∪ x ∈ A. parts(H x))"
apply (rule subsetI)
apply (erule parts.induct, blast+)
done

```

```

lemma parts_UN [simp]: "parts(∪ x ∈ A. H x) = (∪ x ∈ A. parts(H x))"
by (intro equalityI parts_UN_subset1 parts_UN_subset2)

```

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

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

```

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

```

```

declare in_parts_UnE [elim!]

```

```

lemma parts_insert_subset: "insert X (parts H) ⊆ parts(insert X H)"
by (blast intro: parts_mono [THEN [2] rev_subsetD])

```

1.2.2 Idempotence and transitivity

```
lemma parts_partsD [dest!]: "X ∈ parts (parts H) ==> X ∈ parts H"
by (erule parts.induct, blast+)
```

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

```
lemma parts_subset_iff [simp]: "(parts G ⊆ parts H) = (G ⊆ parts H)"
by (metis parts_idem parts_increasing parts_mono subset_trans)
```

```
lemma parts_trans: "[| X ∈ parts G; G ⊆ parts H |] ==> X ∈ parts H"
by (metis parts_subset_iff set_mp)
```

Cut

```
lemma parts_cut:
  "[| Y ∈ parts (insert X G); X ∈ parts H |] ==> Y ∈ parts (G ∪ H)"
by (blast intro: parts_trans)
```

```
lemma parts_cut_eq [simp]: "X ∈ parts H ==> parts (insert X H) = parts H"
by (metis insert_absorb parts_idem parts_insert)
```

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)"
apply (rule parts_insert_eq_I)
apply (erule parts.induct, auto)
done
```

```
lemma parts_insert_Nonce [simp]:
  "parts (insert (Nonce N) H) = insert (Nonce N) (parts H)"
apply (rule parts_insert_eq_I)
apply (erule parts.induct, auto)
done
```

```
lemma parts_insert_Number [simp]:
  "parts (insert (Number N) H) = insert (Number N) (parts H)"
apply (rule parts_insert_eq_I)
apply (erule parts.induct, auto)
done
```

```
lemma parts_insert_Key [simp]:
  "parts (insert (Key K) H) = insert (Key K) (parts H)"
apply (rule parts_insert_eq_I)
apply (erule parts.induct, auto)
done
```

```
lemma parts_insert_Hash [simp]:
  "parts (insert (Hash X) H) = insert (Hash X) (parts H)"
apply (rule parts_insert_eq_I)
apply (erule parts.induct, auto)
```

done

```
lemma parts_insert_Crypt [simp]:
  "parts (insert (Crypt K X) H) = insert (Crypt K X) (parts (insert X H))"
apply (rule equalityI)
apply (rule subsetI)
apply (erule parts.induct, auto)
apply (blast intro: parts.Body)
done
```

```
lemma parts_insert_MPair [simp]:
  "parts (insert {|X,Y|} H) =
   insert {|X,Y|} (parts (insert X (insert Y H)))"
apply (rule equalityI)
apply (rule subsetI)
apply (erule parts.induct, auto)
apply (blast intro: parts.Fst parts.Snd)+
done
```

```
lemma parts_image_Key [simp]: "parts (Key'N) = Key'N"
apply auto
apply (erule parts.induct, auto)
done
```

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}"
apply (induct msg)
apply (simp_all (no_asm_simp) add: exI parts_insert2)
```

Nonce case

```
apply (metis Suc_n_not_le_n)
```

MPair case: metis works out the necessary sum itself!

```
apply (metis le_trans nat_le_linear)
done
```

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)"
apply auto
apply (erule analz.induct)
apply (auto dest: analz.Fst analz.Snd)
done

```

Making it safe speeds up proofs

```

lemma MPair_analz [elim!]:
  "[| {|X,Y|} ∈ analz H;
     [| X ∈ analz H; Y ∈ analz H |] ==> P
  |] ==> P"
by (blast dest: analz.Fst analz.Snd)

```

```

lemma analz_increasing: "H ⊆ analz(H)"
by blast

```

```

lemma analz_subset_parts: "analz H ⊆ parts H"
apply (rule subsetI)
apply (erule analz.induct, blast+)
done

```

```

lemmas analz_into_parts = analz_subset_parts [THEN subsetD]

```

```

lemmas not_parts_not_analz = analz_subset_parts [THEN contra_subsetD]

```

```

lemma parts_analz [simp]: "parts (analz H) = parts H"
by (metis analz_increasing analz_subset_parts equalityI parts_mono parts_subset_iff)

```

```

lemma analz_parts [simp]: "analz (parts H) = parts H"
apply auto
apply (erule analz.induct, auto)
done

```

```

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

```

1.3.1 General equational properties

```

lemma analz_empty [simp]: "analz{} = {}"
apply safe
apply (erule analz.induct, blast+)
done

```

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) ∪ analz(H) ⊆ analz(G ∪ H)"
by (intro Un_least analz_mono Un_upper1 Un_upper2)

```

```

lemma analz_insert: "insert X (analz H) ⊆ analz(insert X H)"
by (blast intro: analz_mono [THEN [2] rev_subsetD])

```

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)"
apply (rule analz_insert_eq_I)
apply (erule analz.induct, auto)
done

```

```

lemma analz_insert_Nonce [simp]:
  "analz (insert (Nonce N) H) = insert (Nonce N) (analz H)"
apply (rule analz_insert_eq_I)
apply (erule analz.induct, auto)
done

```

```

lemma analz_insert_Number [simp]:
  "analz (insert (Number N) H) = insert (Number N) (analz H)"
apply (rule analz_insert_eq_I)
apply (erule analz.induct, auto)
done

```

```

lemma analz_insert_Hash [simp]:
  "analz (insert (Hash X) H) = insert (Hash X) (analz H)"
apply (rule analz_insert_eq_I)
apply (erule analz.induct, auto)
done

```

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

```

lemma analz_insert_Key [simp]:
  "K ∉ keysFor (analz H) ==>
   analz (insert (Key K) H) = insert (Key K) (analz H)"
apply (unfold keysFor_def)
apply (rule analz_insert_eq_I)
apply (erule analz.induct, auto)
done

```

```

lemma analz_insert_MPair [simp]:
  "analz (insert {|X,Y|} H) =
   insert {|X,Y|} (analz (insert X (insert Y H)))"
apply (rule equalityI)
apply (rule subsetI)
apply (erule analz.induct, auto)
apply (erule analz.induct)
apply (blast intro: analz.Fst analz.Snd)+
done

```

Can pull out enCrypted message if the Key is not known

```

lemma analz_insert_Crypt:
  "Key (invKey K) ∉ analz H
   ==> analz (insert (Crypt K X) H) = insert (Crypt K X) (analz H)"
apply (rule analz_insert_eq_I)
apply (erule analz.induct, auto)

done

```

```

lemma lemma1: "Key (invKey K) ∈ analz H ==>

```

```

      analz (insert (Crypt K X) H)  $\subseteq$ 
      insert (Crypt K X) (analz (insert X H))"
apply (rule subsetI)
apply (erule_tac x = x in analz.induct, auto)
done

```

```

lemma lemma2: "Key (invKey K)  $\in$  analz H ==>
  insert (Crypt K X) (analz (insert X H))  $\subseteq$ 
  analz (insert (Crypt K X) H)"
apply auto
apply (erule_tac x = x in analz.induct, auto)
apply (blast intro: analz_insertI analz.Decrypt)
done

```

```

lemma analz_insert_Decrypt:
  "Key (invKey K)  $\in$  analz H ==>
  analz (insert (Crypt K X) H) =
  insert (Crypt K X) (analz (insert X H))"
by (intro equalityI lemma1 lemma2)

```

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

```

lemma analz_Crypt_if [simp]:
  "analz (insert (Crypt K X) H) =
  (if (Key (invKey K)  $\in$  analz H)
  then insert (Crypt K X) (analz (insert X H))
  else insert (Crypt K X) (analz H))"
by (simp add: analz_insert_Crypt analz_insert_Decrypt)

```

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

```

lemma analz_insert_Crypt_subset:
  "analz (insert (Crypt K X) H)  $\subseteq$ 
  insert (Crypt K X) (analz (insert X H))"
apply (rule subsetI)
apply (erule analz.induct, auto)
done

```

```

lemma analz_image_Key [simp]: "analz (Key'N) = Key'N"
apply auto
apply (erule analz.induct, auto)
done

```

1.3.3 Idempotence and transitivity

```

lemma analz_analzD [dest!]: "X  $\in$  analz (analz H) ==> X  $\in$  analz H"
by (erule analz.induct, blast+)

```

```

lemma analz_idem [simp]: "analz (analz H) = analz H"
by blast

```

```

lemma analz_subset_iff [simp]: "(analz G  $\subseteq$  analz H) = (G  $\subseteq$  analz H)"
by (metis analz_idem analz_increasing analz_mono subset_trans)

```

```
lemma analz_trans: "[| X ∈ analz G; G ⊆ analz H |] ==> X ∈ analz H"
by (drule analz_mono, blast)
```

Cut; Lemma 2 of Lowe

```
lemma analz_cut: "[| Y ∈ analz (insert X H); X ∈ analz H |] ==> Y ∈ analz H"
by (erule analz_trans, blast)
```

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"
by (metis analz_cut analz_insert_eq_I insert_absorb)
```

A congruence rule for "analz"

```
lemma analz_subset_cong:
  "[| analz G ⊆ analz G'; analz H ⊆ analz H' |]
   ==> analz (G ∪ H) ⊆ analz (G' ∪ H')"
by (metis Un_mono analz_Un analz_subset_iff subset_trans)
```

```
lemma analz_cong:
  "[| analz G = analz G'; analz H = analz H' |]
   ==> analz (G ∪ H) = analz (G' ∪ H')"
by (intro equalityI analz_subset_cong, simp_all)
```

```
lemma analz_insert_cong:
  "analz H = analz H' ==> analz (insert X H) = analz (insert X H')"
by (force simp only: insert_def intro!: analz_cong)
```

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"
apply safe
apply (erule analz.induct, blast+)
done
```

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)"
apply (erule analz.induct)
apply (blast intro: analz_mono [THEN [2] rev_subsetD])
done
```

```
lemma analz_UN_analz [simp]: "analz (⋃ i ∈ A. analz (H i)) = analz (⋃ i ∈ A. H i)"
by (blast intro: analz_UN_analz_lemma analz_mono [THEN [2] rev_subsetD])
```

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)"
  by (auto, erule synth.induct, auto)

```

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

```

inductive_simps synth_simps [iff]:
  "Nonce n ∈ synth H"
  "Key K ∈ synth H"
  "Hash X ∈ synth H"
  "{|X,Y|} ∈ synth H"
  "Crypt K X ∈ synth H"

```

```

lemma synth_increasing: "H ⊆ synth(H)"
  by blast

```

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)"
  by (intro Un_least synth_mono Un_upper1 Un_upper2)

```

```

lemma synth_insert: "insert X (synth H) ⊆ synth(insert X H)"
  by (blast intro: synth_mono [THEN [2] rev_subsetD])

```

1.4.2 Idempotence and transitivity

```

lemma synth_synthD [dest!]: "X ∈ synth (synth H) ==> X ∈ synth H"
  by (erule synth.induct, auto)

```

```

lemma synth_idem: "synth (synth H) = synth H"
  by blast

```

```

lemma synth_subset_iff [simp]: "(synth G ⊆ synth H) = (G ⊆ synth H)"
  by (metis subset_trans synth_idem synth_increasing synth_mono)

```

```
lemma synth_trans: "[| X ∈ synth G; G ⊆ synth H |] ==> X ∈ synth H"
by (drule synth_mono, blast)
```

Cut; Lemma 2 of Lowe

```
lemma synth_cut: "[| Y ∈ synth (insert X H); X ∈ synth H |] ==> Y ∈ synth H"
by (erule synth_trans, blast)
```

```
lemma Agent_synth [simp]: "Agent A ∈ synth H"
by blast
```

```
lemma Number_synth [simp]: "Number n ∈ synth H"
by blast
```

```
lemma Nonce_synth_eq [simp]: "(Nonce N ∈ synth H) = (Nonce N ∈ H)"
by blast
```

```
lemma Key_synth_eq [simp]: "(Key K ∈ synth H) = (Key K ∈ H)"
by blast
```

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

```
lemma keysFor_synth [simp]:
  "keysFor (synth H) = keysFor H ∪ invKey' {K. Key K ∈ H}"
by (unfold keysFor_def, blast)
```

1.4.3 Combinations of parts, analz and synth

```
lemma parts_synth [simp]: "parts (synth H) = parts H ∪ synth H"
apply (rule equalityI)
apply (rule subsetI)
apply (erule parts.induct)
apply (blast intro: synth_increasing [THEN parts_mono, THEN subsetD]
  parts.Fst parts.Snd parts.Body)+
done
```

```
lemma analz_analz_Un [simp]: "analz (analz G ∪ H) = analz (G ∪ H)"
apply (intro equalityI analz_subset_cong)+
apply simp_all
done
```

```
lemma analz_synth_Un [simp]: "analz (synth G ∪ H) = analz (G ∪ H) ∪ synth G"
apply (rule equalityI)
apply (rule subsetI)
apply (erule analz.induct)
prefer 5 apply (blast intro: analz_mono [THEN [2] rev_subsetD])
apply (blast intro: analz.Fst analz.Snd analz.Decrypt)+
done
```

```
lemma analz_synth [simp]: "analz (synth H) = analz H ∪ synth H"
```

by (metis Un_empty_right analz_synth_Un)

1.4.4 For reasoning about the Fake rule in traces

lemma parts_insert_subset_Un: " $X \in G \implies \text{parts}(\text{insert } X \ H) \subseteq \text{parts } G \cup \text{parts } H$ "

by (metis UnCI Un_upper2 insert_subset parts_Un parts_mono)

More specifically for Fake. See also Fake_parts_sing below

lemma Fake_parts_insert:

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

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

by (metis Un_commute analz_increasing insert_subset parts_analz parts_mono

parts_synth synth_mono synth_subset_iff)

lemma Fake_parts_insert_in_Un:

" $[Z \in \text{parts } (\text{insert } X \ H); X: \text{synth } (\text{analz } H)]$

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

by (metis Fake_parts_insert set_mp)

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

lemma Fake_analz_insert:

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

$\text{analz } (\text{insert } X \ H) \subseteq \text{synth } (\text{analz } G) \cup \text{analz } (G \cup H)$ "

apply (rule subsetI)

apply (subgoal_tac " $x \in \text{analz } (\text{synth } (\text{analz } G) \cup H)$ ", force)

apply (blast intro: analz_mono [THEN [2] rev_subsetD] analz_mono [THEN synth_mono, THEN [2] rev_subsetD])

done

lemma analz_conj_parts [simp]:

" $(X \in \text{analz } H \ \& \ X \in \text{parts } H) = (X \in \text{analz } H)$ "

by (blast intro: analz_subset_parts [THEN subsetD])

lemma analz_disj_parts [simp]:

" $(X \in \text{analz } H \ | \ X \in \text{parts } H) = (X \in \text{parts } H)$ "

by (blast intro: analz_subset_parts [THEN subsetD])

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

lemma MPair_synth_analz [iff]:

" $(\{X, Y\} \in \text{synth } (\text{analz } H)) =$

$(X \in \text{synth } (\text{analz } H) \ \& \ Y \in \text{synth } (\text{analz } H))$ "

by blast

lemma Crypt_synth_analz:

" $[| \text{Key } K \in \text{analz } H; \text{Key } (\text{invKey } K) \in \text{analz } H |]$

$\implies (\text{Crypt } K \ X \in \text{synth } (\text{analz } H)) = (X \in \text{synth } (\text{analz } H))$ "

by blast

lemma Hash_synth_analz [simp]:

```

"X ∉ synth (analz H)
  ==> (Hash{|X,Y|} ∈ synth (analz H)) = (Hash{|X,Y|} ∈ analz H)"
by blast

```

1.5 HPair: a combination of Hash and MPair

1.5.1 Freeness

```

lemma Agent_neq_HPair: "Agent A ~ = Hash[X] Y"
by (unfold HPair_def, simp)

```

```

lemma Nonce_neq_HPair: "Nonce N ~ = Hash[X] Y"
by (unfold HPair_def, simp)

```

```

lemma Number_neq_HPair: "Number N ~ = Hash[X] Y"
by (unfold HPair_def, simp)

```

```

lemma Key_neq_HPair: "Key K ~ = Hash[X] Y"
by (unfold HPair_def, simp)

```

```

lemma Hash_neq_HPair: "Hash Z ~ = Hash[X] Y"
by (unfold HPair_def, simp)

```

```

lemma Crypt_neq_HPair: "Crypt K X' ~ = Hash[X] Y"
by (unfold HPair_def, simp)

```

```

lemmas HPair_neqs = Agent_neq_HPair Nonce_neq_HPair Number_neq_HPair
  Key_neq_HPair Hash_neq_HPair Crypt_neq_HPair

```

```

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

```

```

lemma HPair_eq [iff]: "(Hash[X'] Y' = Hash[X] Y) = (X' = X & Y'=Y)"
by (simp add: HPair_def)

```

```

lemma MPair_eq_HPair [iff]:
  "({|X',Y'|} = Hash[X] Y) = (X' = Hash{|X,Y|} & Y'=Y)"
by (simp add: HPair_def)

```

```

lemma HPair_eq_MPair [iff]:
  "(Hash[X] Y = {|X',Y'|}) = (X' = Hash{|X,Y|} & Y'=Y)"
by (auto simp add: HPair_def)

```

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"
by (simp add: HPair_def)

```

```

lemma parts_insert_HPair [simp]:
  "parts (insert (Hash[X] Y) H) =
  insert (Hash[X] Y) (insert (Hash{|X,Y|}) (parts (insert Y H)))"
by (simp add: HPair_def)

```

```

lemma analz_insert_HPair [simp]:

```

```
"analz (insert (Hash[X] Y) H) =
  insert (Hash[X] Y) (insert (Hash{|X,Y|}) (analz (insert Y H)))"
by (simp add: HPair_def)
```

```
lemma HPair_synth_analz [simp]:
  "X ∉ synth (analz H)
  ==> (Hash[X] Y ∈ synth (analz H)) =
    (Hash {|X, Y|} ∈ analz H & Y ∈ synth (analz H))"
by (auto simp add: HPair_def)
```

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

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

Rewrites to push in Key and Crypt messages, so that other messages can be pulled out using the `analz_insert` rules

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

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

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

```
lemmas pushes = pushKeys pushCrypts
```

1.6 The set of key-free messages

```
inductive_set
```

```
keyfree :: "msg set"
```

```
where
```

```
  Agent: "Agent A ∈ keyfree"
  | Number: "Number N ∈ keyfree"
  | Nonce: "Nonce N ∈ keyfree"
  | Hash: "Hash X ∈ keyfree"
  | MPair: "[|X ∈ keyfree; Y ∈ keyfree|] ==> {|X,Y|} ∈ keyfree"
  | Crypt: "[|X ∈ keyfree|] ==> Crypt K X ∈ keyfree"
```

```
declare keyfree.intros [intro]
```

```
inductive_cases keyfree_KeyE: "Key K ∈ keyfree"
```

```
inductive_cases keyfree_MPairE: "{|X,Y|} ∈ keyfree"
```

```
inductive_cases keyfree_CryptE: "Crypt K X ∈ keyfree"
```

```
lemma parts_keyfree: "parts (keyfree) ⊆ keyfree"
```

```
by (clarify, erule parts.induct, auto elim!: keyfree_KeyE keyfree_MPairE
keyfree_CryptE)
```

```
lemma analz_keyfree_into_Un: "[X ∈ analz (G ∪ H); G ⊆ keyfree] ⇒ X ∈
parts G ∪ analz H"
```

```
apply (erule analz.induct, auto)
```

```
apply (blast dest:parts.Body)
```

```
apply (blast dest:parts.Body)
```

```
apply (metis Un_absorb2 keyfree_KeyE parts_Un parts_keyfree UnI2)
```

```
done
```

1.7 Tactics useful for many protocol proofs

ML

```
{*
```

```
(*Analysis of Fake cases. Also works for messages that forward unknown parts,
but this application is no longer necessary if analz_insert_eq is used.
Abstraction over i is ESSENTIAL: it delays the dereferencing of claset
DEPENDS UPON "X" REFERRING TO THE FRADULENT MESSAGE *)
```

```
fun impOfSubs th = th RSN (2, @{thm rev_subsetD})
```

```
(*Apply rules to break down assumptions of the form
Y ∈ parts(insert X H) and Y ∈ analz(insert X H)
*)
```

```
val Fake_insert_tac =
  dresolve_tac [impOfSubs @{thm Fake_analz_insert},
               impOfSubs @{thm Fake_parts_insert}] THEN'
  eresolve_tac [asm_rl, @{thm synth.Inj}];
```

```
fun Fake_insert_simp_tac ss i =
  REPEAT (Fake_insert_tac i) THEN asm_full_simp_tac ss i;
```

```
fun atomic_spy_analz_tac ctxt =
  SELECT_GOAL
  (Fake_insert_simp_tac (simpset_of ctxt) 1 THEN
  IF_UNSOLVED
  (Blast.depth_tac
   (ctxt addIs [ @{thm analz_insertI}, impOfSubs @{thm analz_subset_parts} ]))
  4 1));
```

```
fun spy_analz_tac ctxt i =
  DETERM
  (SELECT_GOAL
   (EVERY
    [ (*push in occurrences of X...*)
      (REPEAT o CHANGED)
      (res_inst_tac ctxt [ ("x", 1), "X" ] (insert_commute RS ssubst)
      1),
    (*...allowing further simplifications*)
```

```

    simp_tac (simpset_of ctxt) 1,
    REPEAT (FIRSTGOAL (resolve_tac [allI,impI,notI,conjI,iffI])),
    DEPTH_SOLVE (atomic_spy_analz_tac ctxt 1)) i);
*}

```

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  $\notin$  Key ' A"
by auto
```

```
lemma Hash_notin_image_Key [simp] : "Hash X  $\notin$  Key ' A"
by auto
```

```
lemma synth_analz_mono: "G  $\subseteq$  H ==> synth (analz(G))  $\subseteq$  synth (analz(H))"
by (iprover intro: synth_mono analz_mono)
```

```
lemma Fake_analz_eq [simp]:
  "X  $\in$  synth(analz H) ==> synth (analz (insert X H)) = synth (analz H)"
by (metis Fake_analz_insert Un_absorb Un_absorb1 Un_commute
  subset_insertI synth_analz_mono synth_increasing synth_subset_iff)
```

Two generalizations of `analz_insert_eq`

```
lemma gen_analz_insert_eq [rule_format]:
  "X  $\in$  analz H ==> ALL G. H  $\subseteq$  G --> analz (insert X G) = analz G"
by (blast intro: analz_cut analz_insertI analz_mono [THEN [2] rev_subsetD])
```

```
lemma synth_analz_insert_eq [rule_format]:
  "X  $\in$  synth (analz H)
  ==> ALL G. H  $\subseteq$  G --> (Key K  $\in$  analz (insert X G)) = (Key K  $\in$  analz
  G)"
apply (erule synth.induct)
apply (simp_all add: gen_analz_insert_eq subset_trans [OF _ subset_insertI])
```

```
done
```

```
lemma Fake_parts_sing:
  "X  $\in$  synth (analz H) ==> parts{X}  $\subseteq$  synth (analz H)  $\cup$  parts H"
by (metis Fake_parts_insert empty_subsetI insert_mono parts_mono subset_trans)
```

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

```
method_setup spy_analz = {*
  Scan.succeed (SIMPLE_METHOD' o spy_analz_tac) *}
  "for proving the Fake case when analz is involved"
```

```
method_setup atomic_spy_analz = {*
  Scan.succeed (SIMPLE_METHOD' o atomic_spy_analz_tac) *}
  "for debugging spy_analz"
```

```
method_setup Fake_insert_simp = {*
```

```

Scan.succeed (SIMPLE_METHOD' o Fake_insert_simp_tac o simpset_of) *}
"for debugging spy_analz"

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"          — compromised agents
```

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"
  by (rule exI [of _ "{Spy}"], simp)
```

```
primrec knows :: "agent => event list => msg set"
```

```
where
```

```
  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))"
```

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

```
abbreviation (input)
```

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

```

primrec used :: "event list => msg set"
where
  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"
apply (induct_tac evs)
apply (auto split: event.split)
done

lemma Says_imp_used [rule_format]: "Says A B X ∈ set evs --> X ∈ used evs"
apply (induct_tac evs)
apply (auto split: event.split)
done

```

2.1 Function knows

lemmas parts_insert_knows_A = parts_insert [of _ "knows A evs"] for A evs

```

lemma knows_Spy_Says [simp]:
  "knows Spy (Says A B X # evs) = insert X (knows Spy evs)"
by simp

```

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)"
by simp

```

```

lemma knows_Spy_Gets [simp]: "knows Spy (Gets A X # evs) = knows Spy evs"
by simp

```

```

lemma knows_Spy_subset_knows_Spy_Says:
  "knows Spy evs ⊆ knows Spy (Says A B X # evs)"
by (simp add: subset_insertI)

```

```

lemma knows_Spy_subset_knows_Spy_Notes:
  "knows Spy evs ⊆ knows Spy (Notes A X # evs)"
by force

```

```

lemma knows_Spy_subset_knows_Spy_Gets:
  "knows Spy evs ⊆ knows Spy (Gets A X # evs)"
by (simp add: subset_insertI)

```

Spy sees what is sent on the traffic

```
lemma Says_imp_knows_Spy [rule_format]:
  "Says A B X ∈ set evs --> X ∈ knows Spy evs"
apply (induct_tac "evs")
apply (simp_all (no_asm_simp) split add: event.split)
done
```

```
lemma Notes_imp_knows_Spy [rule_format]:
  "Notes A X ∈ set evs --> A: bad --> X ∈ knows Spy evs"
apply (induct_tac "evs")
apply (simp_all (no_asm_simp) split add: event.split)
done
```

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

```
lemmas Says_imp_parts_knows_Spy =
  Says_imp_knows_Spy [THEN parts.Inj, elim_format]
```

```
lemmas knows_Spy_partsEs =
  Says_imp_parts_knows_Spy parts.Body [elim_format]
```

```
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)"
by simp
```

```
lemma knows_Notes: "knows A (Notes A X # evs) = insert X (knows A evs)"
by simp
```

```
lemma knows_Gets:
  "A ≠ Spy --> knows A (Gets A X # evs) = insert X (knows A evs)"
by simp
```

```
lemma knows_subset_knows_Says: "knows A evs ⊆ knows A (Says A' B X # evs)"
by (simp add: subset_insertI)
```

```
lemma knows_subset_knows_Notes: "knows A evs ⊆ knows A (Notes A' X # evs)"
by (simp add: subset_insertI)
```

```
lemma knows_subset_knows_Gets: "knows A evs ⊆ knows A (Gets A' X # evs)"
by (simp add: subset_insertI)
```

Agents know what they say

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

```

apply (induct_tac "evs")
apply (simp_all (no_asm_simp) split add: event.split)
apply blast
done

```

Agents know what they note

```

lemma Notes_imp_knows [rule_format]: "Notes A X ∈ set evs --> X ∈ knows
A evs"
apply (induct_tac "evs")
apply (simp_all (no_asm_simp) split add: event.split)
apply blast
done

```

Agents know what they receive

```

lemma Gets_imp_knows_agents [rule_format]:
  "A ≠ Spy --> Gets A X ∈ set evs --> X ∈ knows A evs"
apply (induct_tac "evs")
apply (simp_all (no_asm_simp) split add: event.split)
done

```

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

```

lemma knows_imp_Says_Gets_Notes_initState [rule_format]:
  "[| X ∈ knows A evs; A ≠ Spy |] ==> EX B.
  Says A B X ∈ set evs | Gets A X ∈ set evs | Notes A X ∈ set evs | X ∈
  initState A"
apply (erule rev_mp)
apply (induct_tac "evs")
apply (simp_all (no_asm_simp) split add: event.split)
apply blast
done

```

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

```

lemma knows_Spy_imp_Says_Notes_initState [rule_format]:
  "[| X ∈ knows Spy evs |] ==> EX A B.
  Says A B X ∈ set evs | Notes A X ∈ set evs | X ∈ initState Spy"
apply (erule rev_mp)
apply (induct_tac "evs")
apply (simp_all (no_asm_simp) split add: event.split)
apply blast
done

```

```

lemma parts_knows_Spy_subset_used: "parts (knows Spy evs) ⊆ used evs"
apply (induct_tac "evs", force)
apply (simp add: parts_insert_knows_A knows_Cons add: event.split, blast)

```

done

```

lemmas usedI = parts_knows_Spy_subset_used [THEN subsetD, intro]

```

```

lemma initState_into_used: "X ∈ parts (initState B) ==> X ∈ used evs"
apply (induct_tac "evs")

```

```

apply (simp_all add: parts_insert_knows_A split add: event.split, blast)
done

```

```

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

```

```

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

```

```

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

```

```

lemma used_nil_subset: "used [] ⊆ used evs"
apply simp
apply (blast intro: initState_into_used)
done

```

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

```

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

```

```

lemma knows_subset_knows_Cons: "knows A evs ⊆ knows A (e # evs)"
by (induct e, auto simp: knows_Cons)

```

```

lemma initState_subset_knows: "initState A ⊆ knows A evs"
apply (induct_tac evs, simp)
apply (blast intro: knows_subset_knows_Cons [THEN subsetD])
done

```

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"
by (force
  dest!: parts_insert_subset_Un [THEN keysFor_mono, THEN [2] rev_subsetD]
  analz_subset_parts [THEN keysFor_mono, THEN [2] rev_subsetD]
  intro: analz_subset_parts [THEN subsetD] parts_mono [THEN [2] rev_subsetD])

```

```

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

```

```

ML
{*

```

```

val analz_mono_contra_tac =
  rtac @{thm analz_impI} THEN'
  REPEAT1 o (dresolve_tac @{thms analz_mono_contra})
  THEN' mp_tac
*}

method_setup analz_mono_contra = {*
  Scan.succeed (K (SIMPLE_METHOD (REPEAT_FIRST analz_mono_contra_tac))) *}
  "for proving theorems of the form  $X \notin \text{analz} (\text{knows Spy evs}) \rightarrow P$ "

```

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

```

lemmas syan_impI = impI [where P = " $Y \notin \text{synth} (\text{analz} (\text{knows Spy evs}))$ "]
for Y evs

```

ML

```

{*
val synth_analz_mono_contra_tac =
  rtac @{thm syan_impI} THEN'
  REPEAT1 o
    (dresolve_tac
     [ @{thm knows_Spy_subset_knows_Spy_Says} RS @{thm synth_analz_mono} RS
       @{thm contra_subsetD},
       @{thm knows_Spy_subset_knows_Spy_Notes} RS @{thm synth_analz_mono} RS
       @{thm contra_subsetD},
       @{thm knows_Spy_subset_knows_Spy_Gets} RS @{thm synth_analz_mono} RS
       @{thm contra_subsetD} ])
  THEN'
  mp_tac
*}

method_setup synth_analz_mono_contra = {*
  Scan.succeed (K (SIMPLE_METHOD (REPEAT_FIRST synth_analz_mono_contra_tac)))
*}
  "for proving theorems of the form  $X \notin \text{synth} (\text{analz} (\text{knows Spy evs})) \rightarrow P$ "

end

```

```

theory Public
imports Event
begin

```

```

lemma invKey_K: " $K \in \text{symKeys} \implies \text{invKey } K = K$ "
by (simp add: symKeys_def)

```

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'"
  apply (rule exI [of _
    "%b A. 2 * agent_case 0 ( $\lambda n. n + 2$ ) 1 A + keymode_case 0 1 b"])
  apply (auto simp add: inj_on_def split: agent.split keymode.split)
  apply presburger
  apply presburger
  done
```

axiomatization where

```
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 priEK

lemma publicKey_inject [iff]: " $(\text{publicKey } b \ A = \text{publicKey } c \ A') = (b=c \ \& \ A=A')$ "

by (blast dest!: injective_publicKey)

lemma not_symKeys_pubK [iff]: " $\text{publicKey } b \ A \notin \text{symKeys}$ "

by (simp add: symKeys_def)

lemma not_symKeys_priK [iff]: " $\text{privateKey } b \ A \notin \text{symKeys}$ "

by (simp add: symKeys_def)

lemma symKey_neq_priEK : " $K \in \text{symKeys} \implies K \neq \text{priEK } A$ "

by auto

lemma $\text{symKeys_neq_imp_neq}$: " $(K \in \text{symKeys}) \neq (K' \in \text{symKeys}) \implies K \neq K'$ "

by blast

lemma $\text{symKeys_invKey_iff}$ [iff]: " $(\text{invKey } K \in \text{symKeys}) = (K \in \text{symKeys})$ "

by (unfold symKeys_def, auto)

lemma $\text{analz_symKeys_Decrypt}$:

"[| Crypt K $X \in \text{analz } H$; $K \in \text{symKeys}$; $\text{Key } K \in \text{analz } H$ |]
 $\implies X \in \text{analz } H$ "

by (auto simp add: symKeys_def)

2.5 "Image" equations that hold for injective functions

lemma invKey_image_eq [simp]: " $(\text{invKey } x \in \text{invKey } A) = (x \in A)$ "

by auto

lemma $\text{publicKey_image_eq}$ [simp]:

" $(\text{publicKey } b \ x \in \text{publicKey } c \ A) = (b=c \ \& \ x \in A)$ "

by auto

lemma $\text{privateKey_notin_image_publicKey}$ [simp]: " $\text{privateKey } b \ x \notin \text{publicKey } c \ A$ "

by auto

lemma $\text{privateKey_image_eq}$ [simp]:

" $(\text{privateKey } b \ A \in \text{invKey } c \ A) = (b=c \ \& \ A \in A)$ "

by auto

lemma $\text{publicKey_notin_image_privateKey}$ [simp]: " $\text{publicKey } b \ A \notin \text{invKey } c \ A$ "

by auto

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

$\text{shrK} \quad :: \text{"agent } \Rightarrow \text{key"}$ — long-term shared keys

specification (*shrK*)

```
inj_shrK: "inj shrK"
— No two agents have the same long-term key
  apply (rule exI [of _ "agent_case 0 (λn. n + 2) 1"])
  apply (simp add: inj_on_def split: agent.split)
done
```

axiomatization where

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

Injectiveness: Agents' long-term keys are distinct.

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

```
lemma invKey_shrK [simp]: "invKey (shrK A) = shrK A"
by (simp add: invKey_K)
```

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

```
lemma analz_Decrypt':
  "[| Crypt K X ∈ analz H; K ∈ symKeys; Key K ∈ analz H |] ==> X ∈ analz
H"
by (auto simp add: invKey_K)
```

```
lemma priK_neq_shrK [iff]: "shrK A ≠ privateKey b C"
by (simp add: symKeys_neq_imp_neq)
```

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

```
lemma pubK_neq_shrK [iff]: "shrK A ≠ publicKey b C"
by (simp add: symKeys_neq_imp_neq)
```

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

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

```
lemma publicKey_notin_image_shrK [simp]: "publicKey b x ∉ shrK ' AA"
by auto
```

```
lemma privateKey_notin_image_shrK [simp]: "privateKey b x ∉ shrK ' AA"
by auto
```

```
lemma shrK_notin_image_publicKey [simp]: "shrK x ∉ publicKey b ' AA"
by auto
```

```
lemma shrK_notin_image_privateKey [simp]: "shrK x ∉ invKey ' publicKey b
' AA"
```

by auto

```
lemma shrK_image_eq [simp]: "(shrK x ∈ shrK ` AA) = (x ∈ AA)"
by auto
```

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.

overloading

```
initState ≡ initState
```

begin

primrec initState where

```
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)"
```

end

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]:
  "∀X ∈ used evs. parts {X} ⊆ used evs"
apply (induct evs)
prefer 2
apply (simp add: used_Cons split: event.split)
apply (metis Un_iff empty_subsetI insert_subset le_supI1 le_supI2 parts_subset_iff)
```

Base case

```
apply (auto dest!: parts_cut simp add: used_Nil)
done
```

```
lemma MPair_used_D: "{|X,Y|} ∈ used H ==> X ∈ used H & Y ∈ used H"
```

by (drule used_parts_subset_parts, simp, blast)

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

```
lemma MPair_used [elim!]:
  "[| {|X,Y|} ∈ used H;
    [| X ∈ used H; Y ∈ used H |] ==> P |]
  ==> P"
by (blast dest: MPair_used_D)
```

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)) = {}"
apply (unfold keysFor_def)
apply (induct_tac "C")
apply (auto intro: range_eqI)
done
```

```
lemma Crypt_notin_initState: "Crypt K X ∉ parts (initState B)"
by (induct B, auto)
```

```
lemma Crypt_notin_used_empty [simp]: "Crypt K X ∉ used []"
by (simp add: Crypt_notin_initState used_Nil)
```

```
lemma shrK_in_initState [iff]: "Key (shrK A) ∈ initState A"
by (induct_tac "A", auto)
```

```
lemma shrK_in_knows [iff]: "Key (shrK A) ∈ knows A evs"
by (simp add: initState_subset_knows [THEN subsetD])
```

```
lemma shrK_in_used [iff]: "Key (shrK A) ∈ used evs"
by (rule initState_into_used, blast)
```

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

```
lemma shrK_neq: "Key K ∉ used evs ==> shrK B ≠ K"
by blast
```

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

2.8 Function `knows Spy`

```
lemma not_SignatureE [elim!]: "b ≠ Signature ==> b = Encryption"
  by (cases b, auto)
```

Agents see their own private keys!

```
lemma priK_in_initState [iff]: "Key (privateKey b A) ∈ initState A"
  by (cases A, auto)
```

Agents see all public keys!

```
lemma publicKey_in_initState [iff]: "Key (publicKey b A) ∈ initState B"
  by (cases B, auto)
```

All public keys are visible

```
lemma spies_pubK [iff]: "Key (publicKey b A) ∈ spies evs"
apply (induct_tac "evs")
apply (auto simp add: imageI knows_Cons split add: event.split)
done
```

```
lemmas analz_spies_pubK = spies_pubK [THEN analz.Inj]
declare analz_spies_pubK [iff]
```

Spy sees private keys of bad agents!

```
lemma Spy_spies_bad_privateKey [intro!]:
  "A ∈ bad ==> Key (privateKey b A) ∈ spies evs"
apply (induct_tac "evs")
apply (auto simp add: imageI knows_Cons split add: event.split)
done
```

Spy sees long-term shared keys of bad agents!

```
lemma Spy_spies_bad_shrK [intro!]:
  "A ∈ bad ==> Key (shrK A) ∈ spies evs"
apply (induct_tac "evs")
apply (simp_all add: imageI knows_Cons split add: event.split)
done
```

```
lemma publicKey_into_used [iff] : "Key (publicKey b A) ∈ used evs"
apply (rule initState_into_used)
apply (rule publicKey_in_initState [THEN parts.Inj])
done
```

```
lemma privateKey_into_used [iff]: "Key (privateKey b A) ∈ used evs"
apply (rule initState_into_used)
apply (rule priK_in_initState [THEN parts.Inj])
done
```

```
lemma Crypt_Spy_analz_bad:
  "[| Crypt (shrK A) X ∈ analz (knows Spy evs); A ∈ bad |]
  ==> X ∈ analz (knows Spy evs)"
by force
```

2.9 Fresh Nonces

```
lemma Nonce_notin_initState [iff]: "Nonce N ∉ parts (initState B)"
by (induct_tac "B", auto)
```

```
lemma Nonce_notin_used_empty [simp]: "Nonce N ∉ used []"
by (simp add: used_Nil)
```

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"
apply (induct_tac "evs")
apply (rule_tac x = 0 in exI)
apply (simp_all (no_asm_simp) add: used_Cons split add: event.split)
apply safe
apply (rule msg_Nonce_supply [THEN exE], blast elim!: add_leE)+
done
```

```
lemma Nonce_supply1: "EX N. Nonce N ∉ used evs"
by (rule Nonce_supply_lemma [THEN exE], blast)
```

```
lemma Nonce_supply: "Nonce (@ N. Nonce N ∉ used evs) ∉ used evs"
apply (rule Nonce_supply_lemma [THEN exE])
apply (rule someI, fast)
done
```

2.11 Specialized Rewriting for Theorems About *analz* and Image

```
lemma insert_Key_singleton: "insert (Key K) H = Key ` {K} Un H"
by blast
```

```
lemma insert_Key_image: "insert (Key K) (Key ` KK ∪ C) = Key ` (insert K KK)
∪ C"
by blast
```

```
lemma Crypt_imp_keysFor : "[|Crypt K X ∈ H; K ∈ symKeys|] ==> K ∈ keysFor
H"
by (drule Crypt_imp_invKey_keysFor, simp)
```

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)"
by (blast intro: analz_mono [THEN [2] rev_subsetD])
```

```
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 {*
structure Public =
struct
```

```

val analz_image_freshK_ss = @{simpset} delsimps [image_insert, image_Un]
  delsimps [@{thm imp_disjL}] (*reduces blow-up*)
  addsimps @{thms analz_image_freshK_simps}

(*Tactic for possibility theorems*)
fun possibility_tac ctxt =
  REPEAT (*omit used_Says so that Nonces start from different traces!*)
    (ALLGOALS (simp_tac (simpset_of ctxt delsimps [@{thm used_Says}])))
  THEN
  REPEAT_FIRST (eq_assume_tac ORELSE'
    resolve_tac [refl, conjI, @{thm Nonce_supply}]))

(*For harder protocols (such as Recur) where we have to set up some
nonces and keys initially*)
fun basic_possibility_tac ctxt =
  REPEAT
    (ALLGOALS (asm_simp_tac (simpset_of ctxt setSolver safe_solver)))
  THEN
  REPEAT_FIRST (resolve_tac [refl, conjI]))

end
*}

method_setup analz_freshK = {*
  Scan.succeed (fn ctxt =>
    (SIMPLE_METHOD
      (EVERY [REPEAT_FIRST (resolve_tac [allI, ballI, impI]),
        REPEAT_FIRST (rtac @{thm analz_image_freshK_lemma}),
        ALLGOALS (asm_simp_tac (Simplifier.context ctxt Public.analz_image_freshK_ss))])))
*}
"for proving the Session Key Compromise theorem"

```

2.12 Specialized Methods for Possibility Theorems

```

method_setup possibility = {*
  Scan.succeed (SIMPLE_METHOD o Public.possibility_tac) *}
"for proving possibility theorems"

method_setup basic_possibility = {*
  Scan.succeed (SIMPLE_METHOD o Public.basic_possibility_tac) *}
"for proving possibility theorems"

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

definition

```

Issues :: "[agent, agent, msg, event list] => bool"
        ("_ Issues _ with _ on _") where
  "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"

  / Oops: "[[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"
apply (intro exI bexI)
apply (rule_tac [2] ns_shared.Nil
  [THEN ns_shared.NS1, THEN ns_shared.NS2, THEN ns_shared.NS3,
  THEN ns_shared.NS4, THEN ns_shared.NS5])
apply (possibility, simp add: used_Cons)
done

```

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)"
by blast

```

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)"
by blast

```

Theorems of the form $X \notin \text{parts}(\text{knows Spy evs})$ 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)"
apply (erule ns_shared.induct, force, drule_tac [4] NS3_msg_in_parts_spies,
  simp_all, blast+)
done

```

```

lemma Spy_analz_shrK [simp]:
  "evs ∈ ns_shared ==> (Key (shrK A) ∈ analz (spies evs)) = (A ∈ bad)"
by auto

```

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))"
apply (erule rev_mp)
apply (erule ns_shared.induct, force, drule_tac [4] NS3_msg_in_parts_spies,
  simp_all)

```

Fake, NS2, NS4, NS5

```
apply (force dest!: keysFor_parts_insert, blast+)
done
```

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"
by (erule rev_mp, erule ns_shared.induct, auto)
```

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"
apply (erule rev_mp)
apply (erule ns_shared.induct, force, drule_tac [4] NS3_msg_in_parts_spies,
auto)
done
```

```
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})"
by (blast dest!: A_trusts_NS2 Says_Server_message_form)
```

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)"
by (blast dest: Says_imp_knows_Spy analz_shrK_Decrypt cert_A_form analz.Inj)
```

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)"
apply (erule ns_shared.induct, force, drule_tac [4] NS3_msg_in_parts_spies,
simp_all)
```

Fake

```
apply (blast dest: parts_insert_subset_Un)
```

Base, NS4 and NS5

```
apply auto
done
```

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))"
apply (erule ns_shared.induct)
apply (drule_tac [8] Says_Server_message_form)
apply (erule_tac [5] Says_S_message_form [THEN disjE], analz_freshK, spy_analz)
```

NS2, NS3

```
apply blast+
done
```

```
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))"
by (simp only: analz_image_freshK analz_image_freshK_simps)
```

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'"
by (erule rev_mp, erule rev_mp, erule ns_shared.induct, simp_all, blast+)
```

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)"
apply (erule rev_mp)
apply (erule ns_shared.induct, force)
apply (frule_tac [7] Says_Server_message_form)
apply (frule_tac [4] Says_S_message_form)
apply (erule_tac [5] disjE)
```

```

apply (simp_all add: analz_insert_eq analz_insert_freshK pushes split_ifs,
spy_analz)

```

NS2

```

apply blast

```

NS3

```

apply (blast dest!: Crypt_Spy_analz_bad A_trusts_NS2
dest: Says_imp_knows_Spy analz.Inj unique_session_keys)

```

Oops

```

apply (blast dest: unique_session_keys)
done

```

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)"
by (blast dest: Says_Server_message_form secrecy_lemma)

```

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"
apply (erule rev_mp)
apply (erule ns_shared.induct, force, drule_tac [4] NS3_msg_in_parts_spies,
auto)
done

```

```

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"
apply (erule ns_shared.induct, force, drule_tac [4] NS3_msg_in_parts_spies)
apply (analz_mono_contra, simp_all, blast)

```

NS2: contradiction from the assumptions $\text{Key } K \notin \text{used evs2}$ and $\text{Crypt } K (\text{Nonce } NB) \in \text{parts (knows Spy evs2)}$

```

apply (force dest!: Crypt_imp_keysFor)

```

NS4

```

apply (metis B_trusts_NS3 Crypt_Spy_analz_bad Says_imp_analz_Spy Says_imp_parts_knows_Spy
analz.Fst unique_session_keys)
done

```

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"

```

```

by (blast intro: A_trusts_NS4_lemma
      dest: A_trusts_NS2 Spy_not_see_encrypted_key)

```

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)"

```

```

apply (erule ns_shared.induct, force)
apply (drule_tac [4] NS3_msg_in_parts_spies)
apply analz_mono_contra
apply (simp_all add: ex_disj_distrib, blast)

```

NS2

```

apply (blast dest!: new_keys_not_used Crypt_imp_keysFor)

```

NS4

```

apply (metis B_trusts_NS3 Crypt_Spy_analz_bad Says_imp_analz_Spy Says_imp_parts_knows_Spy
analz.Fst unique_session_keys)
done

```

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"

```

```

apply (erule ns_shared.induct, force)
apply (drule_tac [4] NS3_msg_in_parts_spies)
apply (analz_mono_contra, simp_all, blast)

```

NS2

```

apply (blast dest!: new_keys_not_used Crypt_imp_keysFor)

```

NS5

```

apply (blast dest!: A_trusts_NS2
        dest: Says_imp_knows_Spy [THEN analz.Inj]
        unique_session_keys Crypt_Spy_analz_bad)
done

```

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"
by (blast intro: B_trusts_NS5_lemma
     dest: B_trusts_NS3 Spy_not_see_encrypted_key)

```

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)"
apply (induct_tac "evs")
apply (induct_tac [2] "a", auto)
done

```

```

lemma spies_Gets_rev: "spies (evs @ [Gets A X]) = spies evs"
apply (induct_tac "evs")
apply (induct_tac [2] "a", auto)
done

```

```

lemma spies_Notes_rev: "spies (evs @ [Notes A X]) =
  (if A:bad then insert X (spies evs) else spies evs)"
apply (induct_tac "evs")
apply (induct_tac [2] "a", auto)
done

```

```

lemma spies_ev_rev: "spies evs = spies (rev evs)"
apply (induct_tac "evs")
apply (induct_tac [2] "a")
apply (simp_all (no_asm_simp) add: spies_Says_rev spies_Gets_rev spies_Notes_rev)
done

```

```

lemmas parts_spies_ev_revD2 = spies_ev_rev [THEN equalityD2, THEN parts_mono]

```

```

lemma spies_takeWhile: "spies (takeWhile P evs) <= spies evs"
apply (induct_tac "evs")
apply (induct_tac [2] "a", auto)

```

Resembles `used_subset_append` in theory `Event`.

```

done

```

```

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

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"
apply (simp (no_asm) add: Issues_def)
apply (rule exI)
apply (rule conjI, assumption)
apply (simp (no_asm))
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule ns_shared.induct, analz_mono_contra)
apply (simp_all)

fake

apply blast
apply (simp_all add: takeWhile_tail)

NS3 remains by pure coincidence!

apply (force dest!: A_trusts_NS2 Says_Server_message_form)

NS4 would be the non-trivial case can be solved by Nb being used

apply (blast dest: parts_spies_takeWhile_mono [THEN subsetD]
        parts_spies_ev_revD2 [THEN subsetD])
done

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"
by (blast intro: A_trusts_NS4_lemma B_Issues_A dest: A_trusts_NS2)

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"
apply (erule rev_mp)

```

```

apply (erule rev_mp)
apply (erule rev_mp)
apply (erule ns_shared.induct, analz_mono_contra)
apply (simp_all)

```

Fake

```
apply blast
```

NS2

```
apply (force dest!: Crypt_imp_keysFor)
```

NS3

```
apply (metis NS3_msg_in_parts_spies parts_cut_eq)
```

NS5, the most important case, can be solved by unicity

```

apply (metis A_trusts_NS2 Crypt_Spy_analz_bad Says_imp_analz_Spy Says_imp_parts_knows_Spy
  analz.Fst analz.Snd unique_session_keys)
done

```

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"

```

```

apply (simp (no_asm) add: Issues_def)
apply (rule exI)
apply (rule conjI, assumption)
apply (simp (no_asm))
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule ns_shared.induct, analz_mono_contra)
apply (simp_all)

```

fake

```

apply blast
apply (simp_all add: takeWhile_tail)

```

NS3 remains by pure coincidence!

```
apply (force dest!: A_trusts_NS2 Says_Server_message_form)
```

NS5 is the non-trivial case and cannot be solved as in *B_Issues_A*! because NB is not fresh. We need *A_trusts_NS5*, proved for this very purpose

```

apply (blast dest: A_trusts_NS5 parts_spies_takeWhile_mono [THEN subsetD]
  parts_spies_evs_revD2 [THEN subsetD])
done

```

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"
by (blast intro: A_Issues_B B_trusts_NS5_lemma dest: B_trusts_NS3)

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"
    by blast

```

The authenticator only for one journey

specification (*authlife*)

```

    authlife_LB [iff]: "authlife ≠ 0"
    by blast

```

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"

```

definition

```

    Issues :: "[agent, agent, msg, event list] => bool"
             ("_ Issues _ with _ on _") where

```

```
"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))))"
```

definition

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

definition

```
Unique :: "[event, event list] => bool" ("Unique _ on _")
where "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"
```

```

/ Ops: "[ 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"
apply (cut_tac sesKlife_LB)
apply (intro exI bexI)
apply (rule_tac [2]
  bankerberos.Nil [THEN bankerberos.BK1, THEN bankerberos.BK2,
    THEN bankerberos.BK3, THEN bankerberos.BK4])
apply (possibility, simp_all (no_asm_simp) add: used_Cons)
done

```

4.1 Lemmas for reasoning about predicate "Issues"

```

lemma spies_Says_rev: "spies (evs @ [Says A B X]) = insert X (spies evs)"
apply (induct_tac "evs")
apply (induct_tac [2] "a", auto)
done

```

```

lemma spies_Gets_rev: "spies (evs @ [Gets A X]) = spies evs"
apply (induct_tac "evs")
apply (induct_tac [2] "a", auto)
done

```

```

lemma spies_Notes_rev: "spies (evs @ [Notes A X]) =
  (if A:bad then insert X (spies evs) else spies evs)"
apply (induct_tac "evs")
apply (induct_tac [2] "a", auto)
done

```

```

lemma spies_evs_rev: "spies evs = spies (rev evs)"
apply (induct_tac "evs")
apply (induct_tac [2] "a")
apply (simp_all (no_asm_simp) add: spies_Says_rev spies_Gets_rev spies_Notes_rev)
done

```

```

lemmas parts_spies_evs_revD2 = spies_evs_rev [THEN equalityD2, THEN parts_mono]

```

```

lemma spies_takeWhile: "spies (takeWhile P evs) <= spies evs"
apply (induct_tac "evs")

```

```
apply (induct_tac [2] "a", auto)
```

Resembles `used_subset_append` in theory `Event`.

```
done
```

```
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)"
```

```
apply (induct_tac "evs")
```

```
apply simp
```

```
apply (induct_tac "a")
```

```
apply auto
```

```
done
```

```
lemma used_Notes_rev: "used (evs @ [Notes A X]) = parts {X} ∪ (used evs)"
```

```
apply (induct_tac "evs")
```

```
apply simp
```

```
apply (induct_tac "a")
```

```
apply auto
```

```
done
```

```
lemma used_Gets_rev: "used (evs @ [Gets B X]) = used evs"
```

```
apply (induct_tac "evs")
```

```
apply simp
```

```
apply (induct_tac "a")
```

```
apply auto
```

```
done
```

```
lemma used_evns_rev: "used evs = used (rev evs)"
```

```
apply (induct_tac "evs")
```

```
apply simp
```

```
apply (induct_tac "a")
```

```
apply (simp add: used_Says_rev)
```

```
apply (simp add: used_Gets_rev)
```

```
apply (simp add: used_Notes_rev)
```

```
done
```

```
lemma used_takeWhile_used [rule_format]:
```

```
"x : used (takeWhile P X) --> x : used X"
```

```
apply (induct_tac "X")
```

```
apply simp
```

```
apply (induct_tac "a")
```

```
apply (simp_all add: used_Nil)
```

```
apply (blast dest!: initState_into_used)+
```

```
done
```

```
lemma set_evns_rev: "set evs = set (rev evs)"
```

```
apply auto
```

```
done
```

```
lemma takeWhile_void [rule_format]:
```

```
"x ∉ set evs → takeWhile (λz. z ≠ x) evs = evs"
```

```
apply auto
```

done

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)"
apply blast
done
```

```
lemma Oops_parts_spies:
  "Says Server A (Crypt (shrK A) {Timestamp, B, K, X}) ∈ set evs
   ⇒ K ∈ parts (spies evs)"
apply blast
done
```

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

```
lemma Spy_see_shrK [simp]:
  "evs ∈ bankerberos ⇒ (Key (shrK A) ∈ parts (spies evs)) = (A ∈ bad)"
apply (erule bankerberos.induct)
apply (frule_tac [7] Oops_parts_spies)
apply (frule_tac [5] BK3_msg_in_parts_spies, simp_all, blast+)
done
```

```
lemma Spy_analz_shrK [simp]:
  "evs ∈ bankerberos ⇒ (Key (shrK A) ∈ analz (spies evs)) = (A ∈ bad)"
apply auto
done
```

```
lemma Spy_see_shrK_D [dest!]:
  "[[ Key (shrK A) ∈ parts (spies evs);
     evs ∈ bankerberos ]] ⇒ A:bad"
apply (blast dest: Spy_see_shrK)
done
```

```
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]]
   ⇒ K ∉ keysFor (parts (spies evs))"
apply (erule rev_mp)
apply (erule bankerberos.induct)
apply (frule_tac [7] Oops_parts_spies)
apply (frule_tac [5] BK3_msg_in_parts_spies, simp_all)
```

Fake

```
apply (force dest!: keysFor_parts_insert)
```

BK2, BK3, BK4

```
apply (force dest!: analz_shrK_Decrypt)+
done
```

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 ]]
  ⇒ 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)"
apply (unfold before_def)
apply (erule rev_mp)
apply (erule bankerberos.induct, simp_all add: takeWhile_tail)
apply auto
apply (metis length_rev set_rev takeWhile_void used_evs_rev)+
done

```

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"
apply (erule rev_mp)
apply (erule bankerberos.induct)
apply (frule_tac [7] Oops_parts_spies)
apply (frule_tac [5] BK3_msg_in_parts_spies, simp_all, blast)
done

```

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"
apply (erule rev_mp)
apply (erule bankerberos.induct)
apply (frule_tac [7] Oops_parts_spies)
apply (frule_tac [5] BK3_msg_in_parts_spies, simp_all, blast)
done

```

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)"
apply (case_tac "A ∈ bad")
apply (force dest!: Says_imp_spies [THEN analz.Inj])
apply (frule Says_imp_spies [THEN parts.Inj])
apply (blast dest!: Kab_authentic Says_Server_message_form)
done

```

Session keys are not used to encrypt other session keys

```

lemma analz_image_freshK [rule_format (no_asm)]:
  "evs ∈ bankerberos ⇒
  ∀K KK. KK ⊆ - (range shrK) →
  (Key K ∈ analz (Key'KK Un (spies evs))) =
  (K ∈ KK | Key K ∈ analz (spies evs))"
apply (erule bankerberos.induct)
apply (drule_tac [7] Says_Server_message_form)
apply (erule_tac [5] Says_S_message_form [THEN disjE], analz_freshK, spy_analz,
  auto)
done

```

```

lemma analz_insert_freshK:
  "[ evs ∈ bankerberos; KAB ∉ range shrK ] ⇒
  (Key K ∈ analz (insert (Key KAB) (spies evs))) =
  (K = KAB | Key K ∈ analz (spies evs))"
apply (simp only: analz_image_freshK analz_image_freshK_simps)
done

```

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'"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule bankerberos.induct)
apply (frule_tac [7] Oops_parts_spies)
apply (frule_tac [5] BK3_msg_in_parts_spies, simp_all)

```

BK2: it can't be a new key

```

apply blast
done

```

```

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"
  apply (erule rev_mp, erule bankerberos.induct, simp_all add: Unique_def)
  apply blast
  done

```

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"
  apply (erule bankerberos.induct)
  apply (frule_tac [7] Says_Server_message_form)
  apply (frule_tac [5] Says_S_message_form [THEN disjE])
  apply (simp_all (no_asm_simp) add: analz_insert_eq analz_insert_freshK pushes)

  Fake

  apply spy_analz

  BK2

  apply (blast intro: parts_insertI)

  BK3

  apply (case_tac "Aa ∈ bad")
  prefer 2 apply (blast dest: Kab_authentic unique_session_keys)
  apply (blast dest: Says_imp_spies [THEN analz.Inj] Crypt_Spy_analz_bad elim!:
  MPair_analz)

  Oops

  apply (blast dest: unique_session_keys)
  done

```

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)"

```

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

```

apply (frule Says_Server_message_form, assumption)
apply (blast intro: lemma_conf)
done

```

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)"
apply (blast dest!: Kab_authentic Confidentiality_S)
done

```

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)"
apply (blast dest!: ticket_authentic Confidentiality_S)
done

```

Non temporal treatment of authentication

Lemmas *lemma_A* and *lemma_B* in fact are common to both temporal and non-temporal treatments

```

lemma lemma_A [rule_format]:
  "[ A ∉ bad; B ∉ bad; evs ∈ bankerberos ]
  ⇒
  Key K ∉ analz (spies evs) →
  Says Server A (Crypt (shrK A) {Number Tk, Agent B, Key K, X})
  ∈ set evs →
  Crypt K {Agent A, Number Ta} ∈ parts (spies evs) →
  Says A B {X, Crypt K {Agent A, Number Ta}}
  ∈ set evs"
apply (erule bankerberos.induct)
apply (frule_tac [7] Oops_parts_spies)
apply (frule_tac [5] Says_S_message_form)
apply (frule_tac [6] BK3_msg_in_parts_spies, analz_mono_contra)
apply (simp_all (no_asm_simp) add: all_conj_distrib)

```

Fake

```

apply blast

```

BK2

```

apply (force dest: Crypt_imp_invKey_keysFor)

```

BK3

```

apply (blast dest: Kab_authentic unique_session_keys)
done

```

```

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"
apply (erule bankerberos.induct)
apply (frule_tac [7] Oops_parts_spies)
apply (frule_tac [5] Says_S_message_form)
apply (drule_tac [6] BK3_msg_in_parts_spies, analz_mono_contra)
apply (simp_all (no_asm_simp) add: all_conj_distrib)

Fake

apply blast

BK2

apply (force dest: Crypt_imp_invKey_keysFor)

BK4

apply (blast dest: ticket_authentic unique_session_keys
          Says_imp_spies [THEN analz.Inj] Crypt_Spy_analz_bad)
done

```

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"
apply (blast dest!: ticket_authentic
  intro!: lemma_A
  elim!: Confidentiality_S [THEN [2] rev_notE])
done

```

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"
apply (blast dest!: Kab_authentic
  intro!: lemma_B elim!: Confidentiality_S [THEN [2] rev_notE])
done

```

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);

```

4.4 Temporal guarantees, relying on a temporal check that insures that no oops event occurred. These are available

```

    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"
apply (blast dest!: ticket_authentic intro!: lemma_A)
done

```

```

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"
apply (blast dest!: Kab_authentic intro!: lemma_B)
done

```

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"
apply (erule bankerberos.induct)
apply (frule_tac [7] Says_Server_message_form)
apply (frule_tac [5] Says_S_message_form [THEN disjE])
apply (simp_all (no_asm_simp) add: less_SucI analz_insert_eq analz_insert_freshK
  pushes)

```

Fake

```
apply spy_analz
```

BK2

```
apply (blast intro: parts_insertI less_SucI)
```

BK3

```
apply (metis Crypt_Spy_analz_bad Kab_authentic Says_imp_analz_Spy
  Says_imp_parts_knows_Spy analz.Snd less_SucI unique_session_keys)
```

Oops: PROOF FAILS if unsafe intro below

```
apply (blast dest: unique_session_keys intro!: less_SucI)
done
```

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)"
apply (frule Says_Server_message_form, assumption)
apply (blast intro: lemma_conf_temporal)
done

```

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)"
apply (blast dest!: Kab_authentic Confidentiality_S_temporal)
done

```

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)"
apply (blast dest!: ticket_authentic Confidentiality_S_temporal)
done

```

Temporal treatment of authentication

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"
apply (blast dest!: ticket_authentic
  intro!: lemma_A
  elim!: Confidentiality_S_temporal [THEN [2] rev_notE])
done

```

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"

```

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

```

apply (blast dest!: Kab_authentic
        intro!: lemma_B elim!: Confidentiality_S_temporal [THEN [2] rev_notE])
done

```

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"
apply (simp (no_asm) add: Issues_def)
apply (rule exI)
apply (rule conjI, assumption)
apply (simp (no_asm))
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule bankerberos.induct, analz_mono_contra)
apply (simp_all (no_asm_simp))

fake

apply blast

```

K4 obviously is the non-trivial case

```

apply (simp add: takeWhile_tail)
apply (blast dest: ticket_authentic parts_spies_takeWhile_mono [THEN subsetD]
        parts_spies_evsv2 [THEN subsetD] intro: A_authenticates_B_temporal)
done

```

```

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"
apply (blast dest!: A_authenticates_B B_Issues_A)
done

```

```

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"
apply (simp (no_asm) add: Issues_def)
apply (rule exI)

```

```

apply (rule conjI, assumption)
apply (simp (no_asm))
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule bankerberos.induct, analz_mono_contra)
apply (simp_all (no_asm_simp))

fake

apply blast

K3 is the non trivial case

apply (simp add: takeWhile_tail)
apply auto
apply (blast dest: Kab_authentic Says_Server_message_form parts_spies_takeWhile_mono
[THEN subsetD] parts_spies_evs_revD2 [THEN subsetD]
intro!: B_authenticates_A)
done

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"
apply (blast dest: B_authenticates_A A_Issues_B)
done

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]: " $4 \leq \text{sesKlife}$ "
  by blast

```

The authenticator only for one journey

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

```

specification (authlife)
  authlife_LB [iff]: " $2 \leq \text{authlife}$ "
  by blast

```

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"

```

definition

```

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

```

definition

```

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

```

inductive_set bankerb_gets :: "event list set"

where

```

  Nil: "[] \in bankerb_gets"

  / Fake: "[[ evsf \in bankerb_gets; X \in synth (analz (knows Spy evsf)) ] ]
    \implies Says Spy B X # evsf \in bankerb_gets"

  / Reception: "[[ evsr \in bankerb_gets; Says A B X \in set evsr ] ]
    \implies Gets B X # evsr \in bankerb_gets"

  / BK1: "[[ evs1 \in bankerb_gets ] ]
    \implies Says A Server {Agent A, Agent B} # evs1
      \in bankerb_gets"

  / BK2: "[[ evs2 \in bankerb_gets; Key K \notin used evs2; K \in 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"

| Ops: "[ 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"
apply (cut_tac sesKlife_LB)
apply (cut_tac authlife_LB)
apply (intro exI bexI)
apply (rule_tac [2]
  bankerb_gets.Nil [THEN bankerb_gets.BK1, THEN bankerb_gets.Reception,
    THEN bankerb_gets.BK2, THEN bankerb_gets.Reception,
    THEN bankerb_gets.BK3, THEN bankerb_gets.Reception,
    THEN bankerb_gets.BK4])
apply (possibility, simp_all (no_asm_simp) add: used_Cons)
done

```

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"
  apply (erule rev_mp)
  apply (erule bankerb_gets.induct)
  apply auto
  done
```

```
lemma Gets_imp_knows_Spy:
  "[ Gets B X ∈ set evs; evs ∈ bankerb_gets ] ⇒ X ∈ knows Spy evs"
  apply (blast dest!: Gets_imp_Says Says_imp_knows_Spy)
  done
```

```
lemma Gets_imp_knows_Spy_parts[dest]:
  "[ Gets B X ∈ set evs; evs ∈ bankerb_gets ] ⇒ X ∈ parts (knows Spy
  evs)"
  apply (blast dest: Gets_imp_knows_Spy [THEN parts.Inj])
  done
```

```
lemma Gets_imp_knows:
  "[ Gets B X ∈ set evs; evs ∈ bankerb_gets ] ⇒ X ∈ knows B evs"
  by (metis Gets_imp_knows_Spy Gets_imp_knows_agents)
```

```
lemma Gets_imp_knows_analz:
  "[ Gets B X ∈ set evs; evs ∈ bankerb_gets ] ⇒ X ∈ analz (knows B evs)"
  apply (blast dest: Gets_imp_knows [THEN analz.Inj])
  done
```

Lemmas for reasoning about predicate "before"

```
lemma used_Says_rev: "used (evs @ [Says A B X]) = parts {X} ∪ (used evs)"
  apply (induct_tac "evs")
  apply simp
  apply (induct_tac "a")
  apply auto
  done
```

```
lemma used_Notes_rev: "used (evs @ [Notes A X]) = parts {X} ∪ (used evs)"
  apply (induct_tac "evs")
  apply simp
  apply (induct_tac "a")
  apply auto
  done
```

```
lemma used_Gets_rev: "used (evs @ [Gets B X]) = used evs"
  apply (induct_tac "evs")
  apply simp
  apply (induct_tac "a")
  apply auto
  done
```

```
lemma used_evs_rev: "used evs = used (rev evs)"
  apply (induct_tac "evs")
  apply simp
```

```

apply (induct_tac "a")
apply (simp add: used_Says_rev)
apply (simp add: used_Gets_rev)
apply (simp add: used_Notes_rev)
done

```

```

lemma used_takeWhile_used [rule_format]:
  "x : used (takeWhile P X) --> x : used X"
apply (induct_tac "X")
apply simp
apply (induct_tac "a")
apply (simp_all add: used_Nil)
apply (blast dest!: initState_into_used)+
done

```

```

lemma set_evs_rev: "set evs = set (rev evs)"
apply auto
done

```

```

lemma takeWhile_void [rule_format]:
  "x ∉ set evs → takeWhile (λz. z ≠ x) evs = evs"
apply auto
done

```

Forwarding Lemma for reasoning about the encrypted portion of message BK3

```

lemma BK3_msg_in_parts_knows_Spy:
  "[[Gets A (Crypt KA {Timestamp, B, K, X})] ∈ set evs; evs ∈ bankerb_gets
  ]]
  ⇒ X ∈ parts (knows Spy evs)"
apply blast
done

```

```

lemma Ops_parts_knows_Spy:
  "Says Server A (Crypt (shrK A) {Timestamp, B, K, X}) ∈ set evs
  ⇒ K ∈ parts (knows Spy evs)"
apply blast
done

```

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

```

lemma Spy_see_shrK [simp]:
  "evs ∈ bankerb_gets ⇒ (Key (shrK A) ∈ parts (knows Spy evs)) = (A
  ∈ bad)"
apply (erule bankerb_gets.induct)
apply (frule_tac [8] Ops_parts_knows_Spy)
apply (frule_tac [6] BK3_msg_in_parts_knows_Spy, simp_all, blast+)
done

```

```

lemma Spy_analz_shrK [simp]:
  "evs ∈ bankerb_gets ⇒ (Key (shrK A) ∈ analz (knows Spy evs)) = (A
  ∈ bad)"
by auto

```

```

lemma Spy_see_shrK_D [dest!]:

```

```

    "[ Key (shrK A) ∈ parts (knows Spy evs);
      evs ∈ bankerb_gets ] ⇒ A:bad"
  by (blast dest: Spy_see_shrK)

```

```

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))"
apply (erule rev_mp)
apply (erule bankerb_gets.induct)
apply (frule_tac [8] Oops_parts_knows_Spy)
apply (frule_tac [6] BK3_msg_in_parts_knows_Spy, simp_all)

```

Fake

```

apply (force dest!: keysFor_parts_insert)

```

BK2, BK3, BK4

```

apply (force dest!: analz_shrK_Decrypt)+
done

```

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)"
apply (unfold before_def)
apply (erule rev_mp)
apply (erule bankerb_gets.induct, simp_all)

```

We need this simplification only for Message 2

```

apply (simp (no_asm) add: takeWhile_tail)
apply auto

```

Two subcases of Message 2. Subcase: used before

```

apply (blast dest: used_evs_rev [THEN equalityD2, THEN contra_subsetD]
      used_takeWhile_used)

```

subcase: CT before

```

apply (fastforce dest!: set_evs_rev [THEN equalityD2, THEN contra_subsetD,
THEN takeWhile_void])
done

```

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"
apply (erule rev_mp)
apply (erule bankerb_gets.induct)
apply (frule_tac [8] Oops_parts_knows_Spy)
apply (frule_tac [6] BK3_msg_in_parts_knows_Spy, simp_all, blast)
done

```

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"
apply (erule rev_mp)
apply (erule bankerb_gets.induct)
apply (frule_tac [8] Oops_parts_knows_Spy)
apply (frule_tac [6] BK3_msg_in_parts_knows_Spy, simp_all, blast)
done

```

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 ∈ analz (knows Spy evs)"
apply (case_tac "A ∈ bad")
apply (force dest!: Gets_imp_knows_Spy [THEN analz.Inj])
apply (blast dest!: Kab_authentic Says_Server_message_form)
done

```

Reliability guarantees: honest agents act as we expect

```

lemma BK3_imp_Gets:
  "[ Says A B {Ticket, Crypt K {Agent A, Number Ta}} ∈ set evs;
    A ∉ bad; evs ∈ bankerb_gets ]
  ⇒ ∃ Tk. Gets A (Crypt (shrK A) {Number Tk, Agent B, Key K, Ticket})

```

```

    ∈ set evs"
  apply (erule rev_mp)
  apply (erule bankerb_gets.induct)
  apply auto
done

```

```

lemma BK4_imp_Gets:
  "[ Says B A (Crypt K (Number Ta)) ∈ set evs;
    B ∉ bad; evs ∈ bankerb_gets ]
  ⇒ ∃ Tk. Gets B {Crypt (shrK B) {Number Tk, Agent A, Key K}},
    Crypt K {Agent A, Number Ta}} ∈ set evs"
  apply (erule rev_mp)
  apply (erule bankerb_gets.induct)
  apply auto
done

```

```

lemma Gets_A_knows_K:
  "[ Gets A (Crypt (shrK A) {Number Tk, Agent B, Key K, Ticket}) ∈ set evs;
    evs ∈ bankerb_gets ]
  ⇒ Key K ∈ analz (knows A evs)"
  apply (force dest: Gets_imp_knows_analz)
done

```

```

lemma Gets_B_knows_K:
  "[ Gets B {Crypt (shrK B) {Number Tk, Agent A, Key K}},
    Crypt K {Agent A, Number Ta}} ∈ set evs;
    evs ∈ bankerb_gets ]
  ⇒ Key K ∈ analz (knows B evs)"
  apply (force dest: Gets_imp_knows_analz)
done

```

Session keys are not used to encrypt other session keys

```

lemma analz_image_freshK [rule_format (no_asm)]:
  "evs ∈ bankerb_gets ⇒
  ∀K KK. KK ⊆ - (range shrK) →
    (Key K ∈ analz (Key'KK Un (knows Spy evs))) =
    (K ∈ KK | Key K ∈ analz (knows Spy evs))"
  apply (erule bankerb_gets.induct)
  apply (drule_tac [8] Says_Server_message_form)
  apply (erule_tac [6] Gets_Server_message_form [THEN disjE], analz_freshK,
    spy_analz, auto)
done

```

```

lemma analz_insert_freshK:
  "[ evs ∈ bankerb_gets; KAB ∉ range shrK ] ⇒
  (Key K ∈ analz (insert (Key KAB) (knows Spy evs))) =
  (K = KAB | Key K ∈ analz (knows Spy evs))"
  by (simp only: analz_image_freshK analz_image_freshK_simps)

```

The session key K uniquely identifies the message

```

lemma unique_session_keys:
  "[ Says Server A
    (Crypt (shrK A) {Number Tk, Agent B, Key K, X}) ∈ set evs;

```

```

    Says Server A'
    (Crypt (shrK A') {Number Tk', Agent B', Key K, X'}) ∈ set evs;
    evs ∈ bankerb_gets ] ⇒ A=A' & Tk=Tk' & B=B' & X = X'"
  apply (erule rev_mp)
  apply (erule rev_mp)
  apply (erule bankerb_gets.induct)
  apply (frule_tac [8] Oops_parts_knows_Spy)
  apply (frule_tac [6] BK3_msg_in_parts_knows_Spy, simp_all)

```

BK2: it can't be a new key

```

  apply blast
  done

```

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'"
  apply (blast dest!: Kab_authentic unique_session_keys)
  done

```

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"
  apply (erule rev_mp, erule bankerb_gets.induct, simp_all add: Unique_def)
  apply blast
  done

```

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"
  apply (erule bankerb_gets.induct)
  apply (frule_tac [8] Says_Server_message_form)
  apply (frule_tac [6] Gets_Server_message_form [THEN disjE])
  apply (simp_all (no_asm_simp) add: analz_insert_eq analz_insert_freshK pushes)

```

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

Fake

```
apply spy_analz
```

BK2

```
apply (blast intro: parts_insertI)
```

BK3

```
apply (case_tac "Aa ∈ bad")
  prefer 2 apply (blast dest: Kab_authentic unique_session_keys)
apply (blast dest: Gets_imp_knows_Spy [THEN analz.Inj] Crypt_Spy_analz_bad
  elim!: MPair_analz)
```

Oops

```
apply (blast dest: unique_session_keys)
done
```

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)"
apply (frule Says_Server_message_form, assumption)
apply (blast intro: lemma_conf)
done
```

Confidentiality for Alice

```
lemma Confidentiality_A:
  "[ Crypt (shrK A) {Number Tk, Agent B, Key K, X} ∈ parts (knows Spy evs);
    Notes Spy {Number Tk, Key K} ∉ set evs;
    A ∉ bad; B ∉ bad; evs ∈ bankerB_gets
  ] ⇒ Key K ∉ analz (knows Spy evs)"
by (blast dest!: Kab_authentic Confidentiality_S)
```

Confidentiality for Bob

```
lemma Confidentiality_B:
  "[ Crypt (shrK B) {Number Tk, Agent A, Key K}
    ∈ parts (knows Spy evs);
    Notes Spy {Number Tk, Key K} ∉ set evs;
    A ∉ bad; B ∉ bad; evs ∈ bankerB_gets
  ] ⇒ Key K ∉ analz (knows Spy evs)"
by (blast dest!: ticket_authentic Confidentiality_S)
```

Non temporal treatment of authentication

Lemmas *lemma_A* and *lemma_B* in fact are common to both temporal and non-temporal treatments

```
lemma lemma_A [rule_format]:
  "[ A ∉ bad; B ∉ bad; evs ∈ bankerB_gets ]
  ⇒
```

```

    Key K ∉ analz (knows Spy evs) →
    Says Server A (Crypt (shrK A) {Number Tk, Agent B, Key K, X})
    ∈ set evs →
    Crypt K {Agent A, Number Ta} ∈ parts (knows Spy evs) →
    Says A B {X, Crypt K {Agent A, Number Ta}}
    ∈ set evs"
apply (erule bankerb_gets.induct)
apply (frule_tac [8] Oops_parts_knows_Spy)
apply (frule_tac [6] Gets_Server_message_form)
apply (frule_tac [7] BK3_msg_in_parts_knows_Spy, analz_mono_contra)
apply (simp_all (no_asm_simp) add: all_conj_distrib)

Fake

apply blast

BK2

apply (force dest: Crypt_imp_invKey_keysFor)

BK3

apply (blast dest: Kab_authentic_unique_session_keys)
done
lemma lemma_B [rule_format]:
  "[ B ∉ bad; evs ∈ bankerb_gets ]
  ⇒ Key K ∉ analz (knows Spy evs) →
    Says Server A (Crypt (shrK A) {Number Tk, Agent B, Key K, X})
    ∈ set evs →
    Crypt K (Number Ta) ∈ parts (knows Spy evs) →
    Says B A (Crypt K (Number Ta)) ∈ set evs"
apply (erule bankerb_gets.induct)
apply (frule_tac [8] Oops_parts_knows_Spy)
apply (frule_tac [6] Gets_Server_message_form)
apply (drule_tac [7] BK3_msg_in_parts_knows_Spy, analz_mono_contra)
apply (simp_all (no_asm_simp) add: all_conj_distrib)

Fake

apply blast

BK2

apply (force dest: Crypt_imp_invKey_keysFor)

BK4

apply (blast dest: ticket_authentic_unique_session_keys
    Gets_imp_knows_Spy [THEN analz.Inj] Crypt_Spy_analz_bad)
done

```

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

Authentication of A to B

```

lemma B_authenticates_A_r:
  "[ Crypt K {Agent A, Number Ta} ∈ parts (knows Spy evs);
    Crypt (shrK B) {Number Tk, Agent A, Key K} ∈ parts (knows Spy evs);

```

5.3 Temporal guarantees, relying on a temporal check that insures that no oops event occurred. These are available

```

Notes Spy {Number Tk, Key K} ∉ set evs;
A ∉ bad; B ∉ bad; evs ∈ bankerb_gets ]
⇒ Says A B {Crypt (shrK B) {Number Tk, Agent A, Key K},
             Crypt K {Agent A, Number Ta}} ∈ set evs"
by (blast dest!: ticket_authentic
    intro!: lemma_A
    elim!: Confidentiality_S [THEN [2] rev_notE])

```

Authentication of B to A

```

lemma A_authenticates_B_r:
  "[ Crypt K (Number Ta) ∈ parts (knows Spy evs);
    Crypt (shrK A) {Number Tk, Agent B, Key K, X} ∈ parts (knows Spy evs);
    Notes Spy {Number Tk, Key K} ∉ set evs;
    A ∉ bad; B ∉ bad; evs ∈ bankerb_gets ]
  ⇒ Says B A (Crypt K (Number Ta)) ∈ set evs"
by (blast dest!: Kab_authentic
    intro!: lemma_B elim!: Confidentiality_S [THEN [2] rev_notE])

```

```

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

```

```

apply (blast dest!: ticket_authentic intro!: lemma_A)
done

```

```

lemma A_authenticates_B:
  "[ Crypt K (Number Ta) ∈ parts (spies evs);
    Crypt (shrK A) {Number Tk, Agent B, Key K, X} ∈ parts (spies evs);
    Key K ∉ analz (spies evs);
    A ∉ bad; B ∉ bad; evs ∈ bankerb_gets ]
  ⇒ Says B A (Crypt K (Number Ta)) ∈ set evs"
apply (blast dest!: Kab_authentic intro!: lemma_B)
done

```

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

Temporal treatment of confidentiality

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

```

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

```

```

apply (frule_tac [8] Says_Server_message_form)
apply (frule_tac [6] Gets_Server_message_form [THEN disjE])
apply (simp_all (no_asm_simp) add: less_SucI analz_insert_eq analz_insert_freshK
pushes)

```

Fake

```
apply spy_analz
```

BK2

```
apply (blast intro: parts_insertI less_SucI)
```

BK3

```

apply (case_tac "Aa ∈ bad")
  prefer 2 apply (blast dest: Kab_authentic unique_session_keys)
apply (blast dest: Gets_imp_knows_Spy [THEN analz.Inj] Crypt_Spy_analz_bad
elim!: MPair_analz intro: less_SucI)

```

Oops: PROOF FAILS if unsafe intro below

```

apply (blast dest: unique_session_keys intro!: less_SucI)
done

```

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 ∈ bankerb_gets
  ] ⇒ Key K ∉ analz (knows Spy evs)"
apply (frule Says_Server_message_form, assumption)
apply (blast intro: lemma_conf_temporal)
done

```

Confidentiality for Alice

```

lemma Confidentiality_A_temporal:
  "[[ Crypt (shrK A) {Number T, Agent B, Key K, X} ∈ parts (knows Spy evs);
    ¬ expiredK T evs;
    A ∉ bad; B ∉ bad; evs ∈ bankerb_gets
  ] ⇒ Key K ∉ analz (knows Spy evs)"
by (blast dest!: Kab_authentic Confidentiality_S_temporal)

```

Confidentiality for Bob

```

lemma Confidentiality_B_temporal:
  "[[ Crypt (shrK B) {Number Tk, Agent A, Key K}
    ∈ parts (knows Spy evs);
    ¬ expiredK Tk evs;
    A ∉ bad; B ∉ bad; evs ∈ bankerb_gets
  ] ⇒ Key K ∉ analz (knows Spy evs)"
by (blast dest!: ticket_authentic Confidentiality_S_temporal)

```

Temporal treatment of authentication

Authentication of A to B

5.4 Combined guarantees of key distribution and non-injective agreement on the session keys79

```

lemma B_authenticates_A_temporal:
  "[[ Crypt K {Agent A, Number Ta} ∈ parts (knows Spy evs);
    Crypt (shrK B) {Number Tk, Agent A, Key K}
    ∈ parts (knows Spy evs);
    ¬ expiredK Tk evs;
    A ∉ bad; B ∉ bad; evs ∈ bankerb_gets ]]
  ⇒ Says A B {Crypt (shrK B) {Number Tk, Agent A, Key K},
    Crypt K {Agent A, Number Ta}} ∈ set evs"
by (blast dest!: ticket_authentic
  intro!: lemma_A
  elim!: Confidentiality_S_temporal [THEN [2] rev_notE])

```

Authentication of B to A

```

lemma A_authenticates_B_temporal:
  "[[ Crypt K (Number Ta) ∈ parts (knows Spy evs);
    Crypt (shrK A) {Number Tk, Agent B, Key K, X}
    ∈ parts (knows Spy evs);
    ¬ expiredK Tk evs;
    A ∉ bad; B ∉ bad; evs ∈ bankerb_gets ]]
  ⇒ Says B A (Crypt K (Number Ta)) ∈ set evs"
by (blast dest!: Kab_authentic
  intro!: lemma_B elim!: Confidentiality_S_temporal [THEN [2] rev_notE])

```

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

```

lemma B_authenticates_and_keydist_to_A:
  "[[ Gets B {Crypt (shrK B) {Number Tk, Agent A, Key K}},
    Crypt K {Agent A, Number Ta}} ∈ set evs;
    Key K ∉ analz (spies evs);
    A ∉ bad; B ∉ bad; evs ∈ bankerb_gets ]]
  ⇒ Says A B {Crypt (shrK B) {Number Tk, Agent A, Key K},
    Crypt K {Agent A, Number Ta}} ∈ set evs
  ∧ Key K ∈ analz (knows A evs)"
apply (blast dest: B_authenticates_A BK3_imp_Gets Gets_A_knows_K)
done

```

```

lemma A_authenticates_and_keydist_to_B:
  "[[ Gets A (Crypt (shrK A) {Number Tk, Agent B, Key K, Ticket}) ∈ set
  evs;
    Gets A (Crypt K (Number Ta)) ∈ set evs;
    Key K ∉ analz (spies evs);
    A ∉ bad; B ∉ bad; evs ∈ bankerb_gets ]]
  ⇒ Says B A (Crypt K (Number Ta)) ∈ set evs
  ∧ Key K ∈ analz (knows B evs)"
apply (blast dest: A_authenticates_B BK4_imp_Gets Gets_B_knows_K)
done

```

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*"

axiomatization where

Tgs_not_bad [iff]: "*Tgs* \notin *bad*"
 — *Tgs* is secure — we already know that *Kas* is secure

definition

authKeys :: "event list => key set" where
 "*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*}"

definition

Issues :: "[agent, agent, msg, event list] => bool"
 ("_ *Issues* _ with _ on _" [50, 0, 0, 50] 50) where
 "(A *Issues* B with X on *evs*) =
 ($\exists Y$. Says A B Y \in set *evs* & X \in parts {Y} &
 X \notin parts (spies (takeWhile (% z. z \neq Says A B Y) (rev *evs*))))"

definition

before :: "[event, event list] => event list" ("before _ on _" [0, 50] 50)
 where "(before *ev* on *evs*) = takeWhile (% z. z \sim *ev*) (rev *evs*)"

definition

Unique :: "[event, event list] => bool" ("Unique _ on _" [0, 50] 50)
 where "(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 ≤ authKlife"
  by blast

specification (servKlife)
  servKlife_LB [iff]: "2 + authKlife ≤ servKlife"
  by blast

specification (authlife)
  authlife_LB [iff]: "Suc 0 ≤ authlife"
  by blast

specification (replylife)
  replylife_LB [iff]: "Suc 0 ≤ replylife"
  by blast

abbreviation

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

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

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

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

abbreviation
  valid :: "[nat, nat] => bool" ("valid _ wrt _" [0, 50] 50) where
    "valid T1 wrt T2 == T1 <= replylife + T2"

definition AKcryptSK :: "[key, key, event list] => bool" where
  "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 :: "event list set"

where

Nil: "[] ∈ kerbIV"

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

| K1: "[evs1 ∈ kerbIV]
 ⇒ Says A Kas {Agent A, Agent Tgs, Number (CT evs1)} # evs1
 ∈ kerbIV"

| K2: "[evs2 ∈ kerbIV; 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 ∈ kerbIV"

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

| K4: "[evs4 ∈ kerbIV; Key servK ∉ used evs4; servK ∈ symKeys;
 B ≠ Tgs; authK ∈ symKeys;
 Says A' Tgs {
 (Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK,
 Number Ta}),
 (Crypt authK {Agent A, Number T2}), Agent B}
 ∈ set evs4;
 ¬ expiredAK Ta evs4;
 ¬ expiredA T2 evs4;

```

servKlife + (CT evs4) <= authKlife + Ta
]
=> Says Tgs A
  (Crypt authK {Key servK, Agent B, Number (CT evs4),
               Crypt (shrK B) {Agent A, Agent B, Key servK,
                               Number (CT evs4)} })
  # evs4 ∈ kerbIV"

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

/ K6: "[ evs6 ∈ kerbIV;
  Says A' B {
    (Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}),
    (Crypt servK {Agent A, Number T3})}
  ∈ set evs6;
  ¬ expiredSK Ts evs6;
  ¬ expiredA T3 evs6
]
=> Says B A (Crypt servK (Number T3))
  # evs6 ∈ kerbIV"

/ Ops1: "[ 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)"
apply (induct_tac "evs")
apply (induct_tac [2] "a", auto)
done

```

```

lemma spies_Gets_rev: "spies (evs @ [Gets A X]) = spies evs"
apply (induct_tac "evs")
apply (induct_tac [2] "a", auto)
done

```

```

lemma spies_Notes_rev: "spies (evs @ [Notes A X]) =
  (if A:bad then insert X (spies evs) else spies evs)"
apply (induct_tac "evs")
apply (induct_tac [2] "a", auto)
done

```

```

lemma spies_evts_rev: "spies evs = spies (rev evs)"
apply (induct_tac "evs")
apply (induct_tac [2] "a")
apply (simp_all (no_asm_simp) add: spies_Says_rev spies_Gets_rev spies_Notes_rev)
done

```

```

lemmas parts_spies_evts_revD2 = spies_evts_rev [THEN equalityD2, THEN parts_mono]

```

```

lemma spies_takeWhile: "spies (takeWhile P evs) ≤ spies evs"
apply (induct_tac "evs")
apply (induct_tac [2] "a", auto)

```

Resembles `used_subset_append` in theory `Event`.

```
done
```

```
lemmas parts_spies_takeWhile_mono = spies_takeWhile [THEN parts_mono]
```

6.2 Lemmas about `authKeys`

```
lemma authKeys_empty: "authKeys [] = {}"
```

```

apply (unfold authKeys_def)
apply (simp (no_asm))
done

```

```

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"
by (unfold authKeys_def, auto)

```

```

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)"
by (unfold authKeys_def, auto)

```

```

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"
by (unfold authKeys_def, auto)

```

```

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"
by (unfold authKeys_def, auto)

```

```

lemma authKeys_used: "K  $\in$  authKeys evs  $\implies$  Key K  $\in$  used evs"
by (simp add: authKeys_def, blast)

```

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)"
by blast

```

```

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"
apply (erule rev_mp)
apply (erule kerbIV.induct, auto)
done

```

–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})

```

```

    ∈ set evs ⇒ servTicket ∈ parts (spies evs)"
  by blast

lemma Ops_range_spies2:
  "[[ Says Tgs A (Crypt authK {Key servK, Agent B, Ts, servTicket})
    ∈ set evs ;
    evs ∈ kerbIV ]] ⇒ servK ∉ range shrK & servK ∈ symKeys"
  apply (erule rev_mp)
  apply (erule kerbIV.induct, auto)
  done

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

lemma Spy_see_shrK [simp]:
  "evs ∈ kerbIV ⇒ (Key (shrK A) ∈ parts (spies evs)) = (A ∈ bad)"
  apply (erule kerbIV.induct)
  apply (frule_tac [7] K5_msg_in_parts_spies)
  apply (frule_tac [5] K3_msg_in_parts_spies, simp_all)
  apply (blast+)
  done

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

lemma Spy_see_shrK_D [dest!]:
  "[[ Key (shrK A) ∈ parts (spies evs); evs ∈ kerbIV ]] ⇒ A:bad"
  by (blast dest: Spy_see_shrK)

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))"
  apply (erule rev_mp)
  apply (erule kerbIV.induct)
  apply (frule_tac [7] K5_msg_in_parts_spies)
  apply (frule_tac [5] K3_msg_in_parts_spies, simp_all)

Fake

apply (force dest!: keysFor_parts_insert)

Others

apply (force dest!: analz_shrK_Decrypt)+
done

lemma new_keys_not_analzD:

```

```

"[[evs ∈ kerbIV; K ∈ symKeys; Key K ∉ used evs]]
  ⇒ K ∉ keysFor (analz (spies evs))"
by (blast dest: new_keys_not_used intro: keysFor_mono [THEN subsetD])

```

6.4 Lemmas for reasoning about predicate "before"

```

lemma used_Says_rev: "used (evs @ [Says A B X]) = parts {X} ∪ (used evs)"
apply (induct_tac "evs")
apply simp
apply (induct_tac "a")
apply auto
done

```

```

lemma used_Notes_rev: "used (evs @ [Notes A X]) = parts {X} ∪ (used evs)"
apply (induct_tac "evs")
apply simp
apply (induct_tac "a")
apply auto
done

```

```

lemma used_Gets_rev: "used (evs @ [Gets B X]) = used evs"
apply (induct_tac "evs")
apply simp
apply (induct_tac "a")
apply auto
done

```

```

lemma used_evns_rev: "used evs = used (rev evs)"
apply (induct_tac "evs")
apply simp
apply (induct_tac "a")
apply (simp add: used_Says_rev)
apply (simp add: used_Gets_rev)
apply (simp add: used_Notes_rev)
done

```

```

lemma used_takeWhile_used [rule_format]:
  "x : used (takeWhile P X) --> x : used X"
apply (induct_tac "X")
apply simp
apply (induct_tac "a")
apply (simp_all add: used_Nil)
apply (blast dest!: initState_into_used)+
done

```

```

lemma set_evns_rev: "set evs = set (rev evs)"
by auto

```

```

lemma takeWhile_void [rule_format]:
  "x ∉ set evs → takeWhile (λz. z ≠ x) evs = evs"
by auto

```

6.5 Regularity Lemmas

These concern the form of items passed in messages

Describes the form of all components sent by Kas

lemma *Says_Kas_message_form*:

```

  "[ Says Kas A (Crypt K {Key authK, Agent Peer, Number Ta, authTicket})
    ∈ set evs;
    evs ∈ kerbIV ] ⇒
  K = shrK A & Peer = Tgs &
  authK ∉ range shrK & authK ∈ authKeys evs & authK ∈ symKeys &
  authTicket = (Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta})
&
  Key authK ∉ used(before
    Says Kas A (Crypt K {Key authK, Agent Peer, Number Ta, authTicket})
    on evs) &
  Ta = CT (before
    Says Kas A (Crypt K {Key authK, Agent Peer, Number Ta, authTicket})
    on evs)"
apply (unfold before_def)
apply (erule rev_mp)
apply (erule kerbIV.induct)
apply (simp_all (no_asm) add: authKeys_def authKeys_insert, blast, blast)

```

K2

```

apply (simp (no_asm) add: takeWhile_tail)
apply (rule conjI)
apply (metis Key_not_used authKeys_used length_rev set_rev takeWhile_void
  used_evs_rev)
apply blast+
done

```

lemma *SesKey_is_session_key*:

```

  "[ Crypt (shrK Tgs_B) {Agent A, Agent Tgs_B, Key SesKey, Number T}
    ∈ parts (spies evs); Tgs_B ∉ bad;
    evs ∈ kerbIV ]
  ⇒ SesKey ∉ range shrK"
apply (erule rev_mp)
apply (erule kerbIV.induct)
apply (frule_tac [7] K5_msg_in_parts_spies)
apply (frule_tac [5] K3_msg_in_parts_spies, simp_all, blast)
done

```

lemma *authTicket_authentic*:

```

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

```

```

apply (erule kerbIV.induct)
apply (frule_tac [7] K5_msg_in_parts_spies)
apply (frule_tac [5] K3_msg_in_parts_spies, simp_all)

```

Fake, K4

```

apply (blast+)
done

```

```

lemma authTicket_crypt_authK:
  "[ Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}
    ∈ parts (spies evs);
    evs ∈ kerbIV ]
  ⇒ authK ∈ authKeys evs"
apply (frule authTicket_authentic, assumption)
apply (simp (no_asm) add: authKeys_def)
apply blast
done

```

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) "
apply (unfold before_def)
apply (erule rev_mp)
apply (erule kerbIV.induct)
apply (simp_all add: authKeys_insert authKeys_not_insert authKeys_empty authKeys_simp,
  blast)

```

We need this simplification only for Message 4

```

apply (simp (no_asm) add: takeWhile_tail)
apply auto

```

Five subcases of Message 4

```

apply (blast dest!: SesKey_is_session_key)
apply (blast dest: authTicket_crypt_authK)
apply (blast dest!: authKeys_used Says_Kas_message_form)

```

subcase: used before

```

apply (metis used_evts_rev used_takeWhile_used)

```

subcase: CT before

```

apply (metis length_rev set_evs_rev takeWhile_void)
done

```

```

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}"
apply (erule rev_mp)
apply (erule kerbIV.induct)
apply (frule_tac [7] K5_msg_in_parts_spies)
apply (frule_tac [5] K3_msg_in_parts_spies, simp_all)
apply (blast+)
done

```

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})"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbIV.induct, analz_mono_contra)
apply (frule_tac [7] K5_msg_in_parts_spies)
apply (frule_tac [5] K3_msg_in_parts_spies, simp_all, blast)
done

```

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)"
by (blast dest: analz_shrK_Decrypt authTicket_form
  Says_imp_spies [THEN analz.Inj])

```

```

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)"
by (metis Says_imp_analz_Spy Says_imp_parts_knows_Spy analz.Decrypt analz.Snd
  invKey_K servTicket_form)

```

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"
apply (erule rev_mp)
apply (erule kerbIV.induct)
apply (frule_tac [7] K5_msg_in_parts_spies)
apply (frule_tac [5] K3_msg_in_parts_spies, simp_all)

Fake

apply blast

K4

apply (blast dest!: authTicket_authentic [THEN Says_Kas_message_form])
done

```

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"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbIV.induct, analz_mono_contra)
apply (frule_tac [7] K5_msg_in_parts_spies)
apply (frule_tac [5] K3_msg_in_parts_spies, simp_all)

Fake

apply blast

K2

apply blast

K4

apply auto
done

```

```

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"
  apply (erule rev_mp)
  apply (erule rev_mp)
  apply (erule kerbIV.induct, analz_mono_contra)
  apply (frule_tac [7] K5_msg_in_parts_spies)
  apply (frule_tac [5] K3_msg_in_parts_spies, simp_all)

```

Fake

```
apply blast
```

K4

```
apply blast
```

```
done
```

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"
```

```

  apply (erule rev_mp)
  apply (erule rev_mp)
  apply (erule kerbIV.induct)
  apply (frule_tac [7] K5_msg_in_parts_spies)
  apply (frule_tac [5] K3_msg_in_parts_spies, simp_all)
  apply blast+
  done

```

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"
```

```

  apply (erule rev_mp)
  apply (erule kerbIV.induct)
  apply (frule_tac [7] K5_msg_in_parts_spies)
  apply (frule_tac [5] K3_msg_in_parts_spies, simp_all, auto)
  apply (blast dest!: Says_imp_spies [THEN parts.Inj, THEN parts.Fst, THEN authTicket_authentic])
  done

```

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)"
apply (erule rev_mp)
apply (erule kerbIV.induct)
apply (frule_tac [7] K5_msg_in_parts_spies)
apply (frule_tac [5] K3_msg_in_parts_spies, simp_all, auto)
apply (blast dest!: Says_imp_spies [THEN parts.Inj, THEN parts.Fst, THEN authTicket_authentic])
done

```

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"
by (blast dest!: servTicket_authentic_Tgs K4_imp_K2)

```

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"
by (blast dest!: servTicket_authentic_Tgs u_K4_imp_K2)

```

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"
by (blast dest: servTicket_authentic_Tgs K4_imp_K2)

```

lemma *u_servTicket_authentic*:

```

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

```

```

          Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}})
    ∈ set evs
    & servKlife + Ts <= authKlife + Ta)"
by (blast dest: servTicket_authentic_Tgs u_K4_imp_K2)

```

```

lemma u_NotexpiredSK_NotexpiredAK:
  "[[ ¬ expiredSK Ts evs; servKlife + Ts <= authKlife + Ta ]]
  ⇒ ¬ expiredAK Ta evs"
by (metis le_less_trans)

```

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"
apply (erule rev_mp)
apply (erule kerbIV.induct)
apply (frule_tac [7] K5_msg_in_parts_spies)
apply (frule_tac [5] K3_msg_in_parts_spies, simp_all, blast, blast)
apply (blast dest: Says_imp_spies [THEN parts.Inj, THEN authK_authentic])
done

```

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'"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbIV.induct, analz_mono_contra)
apply (frule_tac [7] K5_msg_in_parts_spies)
apply (frule_tac [5] K3_msg_in_parts_spies, simp_all)

```

Fake, K2, K4

```

apply (blast+)
done

```

```

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 ]]

```

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

```

⇒ ∃ 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"
apply (drule authTicket_authentic, assumption, rotate_tac 4)
apply (erule rev_mp, erule rev_mp, erule rev_mp)
apply (erule kerbIV.induct, analz_mono_contra)
apply (frule_tac [5] Says_ticket_parts)
apply (frule_tac [7] Says_ticket_parts)
apply (simp_all (no_asm_simp) add: all_conj_distrib)

Fake

apply blast

K2

apply (force dest!: Crypt_imp_keysFor)

K3

apply (blast dest: Key_unique_SesKey)

K5

apply (metis K3_imp_K2 Key_unique_SesKey Spy_see_shrK parts.Body parts.Fst
  Says_imp_knows_Spy [THEN parts.Inj])
done

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"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbIV.induct, analz_mono_contra)
apply (frule_tac [5] Says_ticket_parts)
apply (frule_tac [7] Says_ticket_parts)
apply (simp_all (no_asm_simp) add: all_conj_distrib)
apply blast

K3

apply (blast dest: authK_authentic Says_Kas_message_form Says_Tgs_message_form)

K4

apply (force dest!: Crypt_imp_keysFor)

K5

apply (blast dest: Key_unique_SesKey)
done

```

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'"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbIV.induct, analz_mono_contra)
apply (frule_tac [7] K5_msg_in_parts_spies)
apply (frule_tac [5] K3_msg_in_parts_spies, simp_all)

Fake, K2, K4

apply (blast+)
done

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"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbIV.induct, analz_mono_contra)
apply (frule_tac [5] Says_ticket_parts)
apply (frule_tac [7] Says_ticket_parts)
apply (simp_all (no_asm_simp))
apply blast
apply (metis Crypt_imp_invKey_keysFor invKey_K new_keys_not_used)
apply (clarify)
apply (frule Says_Tgs_message_form, assumption)
apply (metis K3_msg_in_parts_spies parts.Fst Says_imp_knows_Spy [THEN parts.Inj]

      unique_CryptKey)
done

```

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"
apply (frule Says_Kas_message_form, assumption)
apply (erule rev_mp)

```

```

apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbIV.induct, analz_mono_contra)
apply (frule_tac [7] K5_msg_in_parts_spies)
apply (frule_tac [5] K3_msg_in_parts_spies, simp_all, blast)

K2

apply (blast dest!: servK_authentic Says_Tgs_message_form authKeys_used)

K4 remain

apply (blast dest!: unique_CryptKey)
done

```

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'"

apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbIV.induct)
apply (frule_tac [7] K5_msg_in_parts_spies)
apply (frule_tac [5] K3_msg_in_parts_spies, simp_all)

K2

apply blast
done

```

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'"

apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbIV.induct)
apply (frule_tac [7] K5_msg_in_parts_spies)
apply (frule_tac [5] K3_msg_in_parts_spies, simp_all)

K4

apply blast
done

```

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"
  apply (erule rev_mp, erule kerbIV.induct, simp_all add: Unique_def)
  apply blast
  done

```

```

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"
  apply (erule rev_mp, erule kerbIV.induct, simp_all add: Unique_def)
  apply blast
  done

```

6.9 Lemmas About the Predicate $AKcryptSK$

```

lemma not_AKcryptSK_Nil [iff]: "¬ AKcryptSK authK servK []"
  by (simp add: AKcryptSK_def)

```

```

lemma AKcryptSKI:
  "[ Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, X }) ∈ set evs;
    evs ∈ kerbIV ] ⇒ AKcryptSK authK servK evs"
  apply (unfold AKcryptSK_def)
  apply (blast dest: Says_Tgs_message_form)
  done

```

```

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)"
  by (auto simp add: AKcryptSK_def)

```

```

lemma Auth_fresh_not_AKcryptSK:
  "[ Key authK ∉ used evs; evs ∈ kerbIV ]
  ⇒ ¬ AKcryptSK authK servK evs"
  apply (unfold AKcryptSK_def)
  apply (erule rev_mp)
  apply (erule kerbIV.induct)
  apply (frule_tac [7] K5_msg_in_parts_spies)
  apply (frule_tac [5] K3_msg_in_parts_spies, simp_all, blast)
  done

```

```

lemma Serv_fresh_not_AKcryptSK:
  "Key servK  $\notin$  used evs  $\implies$   $\neg$  AKcryptSK authK servK evs"
by (unfold AKcryptSK_def, blast)

```

```

lemma authK_not_AKcryptSK:
  "[[ Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, tk}
     $\in$  parts (spies evs); evs  $\in$  kerbIV ]]
     $\implies$   $\neg$  AKcryptSK K authK evs"
apply (erule rev_mp)
apply (erule kerbIV.induct)
apply (frule_tac [7] K5_msg_in_parts_spies)
apply (frule_tac [5] K3_msg_in_parts_spies, simp_all)

```

Fake

```
apply blast
```

K2: by freshness

```
apply (simp add: AKcryptSK_def)
```

K4

```
apply (blast+)
done
```

A secure serverkey cannot have been used to encrypt others

```

lemma servK_not_AKcryptSK:
  "[[ Crypt (shrK B) {Agent A, Agent B, Key SK, Number Ts}  $\in$  parts (spies evs);
    Key SK  $\notin$  analz (spies evs); SK  $\in$  symKeys;
    B  $\neq$  Tgs; evs  $\in$  kerbIV ]]
     $\implies$   $\neg$  AKcryptSK SK K evs"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbIV.induct, analz_mono_contra)
apply (frule_tac [7] K5_msg_in_parts_spies)
apply (frule_tac [5] K3_msg_in_parts_spies, simp_all, blast)

```

K4

```
apply (metis Auth_fresh_not_AKcryptSK Crypt_imp_keysFor new_keys_not_used
parts.Fst parts.Snd Says_imp_knows_Spy [THEN parts.Inj] unique_CryptKey)
done
```

Long term keys are not issued as *servKeys*

```

lemma shrK_not_AKcryptSK:
  "evs  $\in$  kerbIV  $\implies$   $\neg$  AKcryptSK K (shrK A) evs"
apply (unfold AKcryptSK_def)
apply (erule kerbIV.induct)
apply (frule_tac [7] K5_msg_in_parts_spies)
apply (frule_tac [5] K3_msg_in_parts_spies, auto)
done

```

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"
  apply (unfold AKcryptSK_def)
  apply (blast dest: unique_servKeys)
  done

```

Equivalently

```

lemma not_different_AKcryptSK:
  "[ AKcryptSK authK servK evs;
    authK' ≠ authK; evs ∈ kerbIV ]
  ⇒ ¬ AKcryptSK authK' servK evs ∧ servK ∈ symKeys"
  apply (simp add: AKcryptSK_def)
  apply (blast dest: unique_servKeys Says_Tgs_message_form)
  done

```

```

lemma AKcryptSK_not_AKcryptSK:
  "[ AKcryptSK authK servK evs; evs ∈ kerbIV ]
  ⇒ ¬ AKcryptSK servK K evs"
  apply (erule rev_mp)
  apply (erule kerbIV.induct)
  apply (frule_tac [7] K5_msg_in_parts_spies)
  apply (frule_tac [5] K3_msg_in_parts_spies, simp_all)
  apply (metis Auth_fresh_not_AKcryptSK Says_imp_spies authK_not_AKcryptSK
    authKeys_used authTicket_crypt_authK parts.Fst parts.Inj)
  done

```

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)"
  by (blast intro: analz_mono [THEN subsetD])

```

```

lemma AKcryptSK_analz_insert:
  "[ AKcryptSK K K' evs; K ∈ symKeys; evs ∈ kerbIV ]
  ⇒ Key K' ∈ analz (insert (Key K) (spies evs))"
  apply (simp add: AKcryptSK_def, clarify)
  apply (drule Says_imp_spies [THEN analz.Inj, THEN analz_insertI], auto)
  done

```

```

lemma authKeys_are_not_AKcryptSK:
  "[ K ∈ authKeys evs Un range shrK; evs ∈ kerbIV ]
  ⇒ ∀SK. ¬ AKcryptSK SK K evs ∧ K ∈ symKeys"
  apply (simp add: authKeys_def AKcryptSK_def)
  apply (blast dest: Says_Kas_message_form Says_Tgs_message_form)
  done

```

```

lemma not_authKeys_not_AKcryptSK:
  "[[ K ∉ authKeys evs;
    K ∉ range shrK; evs ∈ kerbIV ]]
  ⇒ ∀ SK. ¬ AKcryptSK K SK evs"
apply (simp add: AKcryptSK_def)
apply (blast dest: Says_Tgs_message_form)
done

```

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"
by (blast dest: AKcryptSKI AKcryptSK_not_AKcryptSK)

```

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)))"
apply (erule kerbIV.induct)
apply (frule_tac [10] Oops_range_spies2)
apply (frule_tac [9] Oops_range_spies1)
apply (frule_tac [7] Says_tgs_message_form)
apply (frule_tac [5] Says_kas_message_form)
apply (safe del: impI intro!: Key_analz_image_Key_lemma [THEN impI])

```

Case-splits for Oops1 and message 5: the negated case simplifies using the induction hypothesis

```

apply (case_tac [11] "AKcryptSK authK SK evs01")
apply (case_tac [8] "AKcryptSK servK SK evs5")
apply (simp_all del: image_insert
  add: analz_image_freshK_simps AKcryptSK_Says shrK_not_AKcryptSK
  Oops2_not_AKcryptSK Auth_fresh_not_AKcryptSK
  Serv_fresh_not_AKcryptSK Says_Tgs_AKcryptSK Spy_analz_shrK)

```

Fake

```
apply spy_analz
```

K2

```
apply blast
```

K3

apply blast

K4

apply (blast dest!: authK_not_AKcryptSK)

K5

apply (case_tac "Key servK \in analz (spies evs5) ")

If servK is compromised then the result follows directly...

apply (simp (no_asm_simp) add: analz_insert_eq Un_upper2 [THEN analz_mono, THEN subsetD])

...therefore servK is uncompromised.

The AKcryptSK servK SK evs5 case leads to a contradiction.

apply (blast elim!: servK_not_AKcryptSK [THEN [2] rev_notE] del: allE ballE)

Another K5 case

apply blast

Oops1

apply simp

apply (blast dest!: AKcryptSK_analz_insert)

done

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

lemma analz_insert_freshK1:

"[[evs \in kerbIV; K \in authKeys evs Un range shrK;
SesKey \notin range shrK]]
 \implies (Key K \in analz (insert (Key SesKey) (spies evs))) =
(K = SesKey | Key K \in analz (spies evs))"

apply (frule authKeys_are_not_AKcryptSK, assumption)

apply (simp del: image_insert
add: analz_image_freshK_simps add: Key_analz_image_Key)

done

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

lemma analz_insert_freshK2:

"[[evs \in kerbIV; servK \notin (authKeys evs); servK \notin range shrK;
K \in symKeys]]
 \implies (Key K \in analz (insert (Key servK) (spies evs))) =
(K = servK | Key K \in analz (spies evs))"

apply (frule not_authKeys_not_AKcryptSK, assumption, assumption)

apply (simp del: image_insert
add: analz_image_freshK_simps add: Key_analz_image_Key)

done

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))"
apply (drule_tac authK' = authK' in not_different_AKcryptSK, blast, assumption)
apply (simp del: image_insert
        add: analz_image_freshK_simps add: Key_analz_image_Key)
done

```

```

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))"
apply (frule AKcryptSKI, assumption)
apply (simp add: analz_insert_freshK3)
done

```

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)"
by (metis SAYS_imp_analz_Spy analz.Fst analz_Decrypt')

```

```

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"
apply (erule rev_mp)
apply (erule rev_mp)
apply (simp add: authKeys_def)
apply (erule kerbIV.induct, analz_mono_contra)
apply (frule_tac [7] K5_msg_in_parts_spies)
apply (frule_tac [5] K3_msg_in_parts_spies, simp_all)
apply (blast+)
done

```

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"

```

```

apply (erule kerbIV.induct)
apply (frule_tac [10] Oops_range_spies2)
apply (frule_tac [9] Oops_range_spies1)
apply (frule_tac [7] Says_tgs_message_form)
apply (frule_tac [5] Says_kas_message_form)
apply (safe del: impI conjI impCE)
apply (simp_all (no_asm_simp) add: Says_Kas_message_form less_SucI analz_insert_eq
not_parts_not_analz analz_insert_freshK1 pushes)

```

Fake

```
apply spy_analz
```

K2

```
apply blast
```

K4

```
apply blast
```

Level 8: K5

```
apply (blast dest: servK_notin_authKeysD Says_Kas_message_form intro: less_SucI)
```

Oops1

```
apply (blast dest!: unique_authKeys intro: less_SucI)
```

Oops2

```
apply (blast dest: Says_Tgs_message_form Says_Kas_message_form)
done
```

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)"

```

by (blast dest: Says_Kas_message_form Confidentiality_Kas_lemma)

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"

```

```
apply (erule rev_mp)
```

```
apply (erule rev_mp)
```

```
apply (erule kerbIV.induct)
```

apply (rule_tac [9] impI)+

— The Oops1 case is unusual: must simplify $\text{Authkey} \notin \text{analz} (\text{knows Spy } (ev \# \text{evs}))$, not letting analz_mono_contra weaken it to $\text{Authkey} \notin \text{analz} (\text{knows Spy } \text{evs})$, for we then conclude $\text{authK} \neq \text{authKa}$.

apply *analz_mono_contra*

apply (frule_tac [10] *Oops_range_spies2*)

apply (frule_tac [9] *Oops_range_spies1*)

apply (frule_tac [7] *Says_tgs_message_form*)

apply (frule_tac [5] *Says_kas_message_form*)

apply (safe del: *impI conjI impCE*)

apply (simp_all add: *less_SucI new_keys_not_analz_d Says_Kas_message_form Says_Tgs_message_form*

analz_insert_eq not_parts_not_analz analz_insert_freshK1 analz_insert_freshK2

analz_insert_freshK3_bis pushes)

Fake

apply *spy_analz*

K2

apply (blast intro: *parts_insertI less_SucI*)

K4

apply (blast dest: *authTicket_authentic Confidentiality_Kas*)

K5

apply (*metis Says_imp_spies Says_ticket_parts Tgs_not_bad analz_insert_freshK2*

less_SucI parts.Inj servK_notin_authKeysD unique_CryptKey)

Oops1

apply (blast dest: *Says_Kas_message_form Says_Tgs_message_form intro: less_SucI*)

Oops2

apply (blast dest: *Says_imp_spies [THEN parts.Inj] Key_unique_SesKey intro: less_SucI*)

done

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}*)

\in set *evs*;

Key authK \notin *analz (spies evs)*;

\neg *expiredSK Ts evs*;

A \notin *bad*; *B* \notin *bad*; *evs* \in *kerbIV*]

\implies *Key servK* \notin *analz (spies evs)*"

by (blast dest: *Says_Tgs_message_form Confidentiality_lemma*)

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}*)

\in 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)"
by (blast dest!: Confidentiality_Kas Confidentiality_Tgs)

```

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"
by (blast dest: authK_authentic Confidentiality_Auth_A servK_authentic_ter)

```

```

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)"
apply (drule authK_authentic, assumption, assumption)
apply (blast dest: Confidentiality_Kas Says_Kas_message_form servK_authentic_ter
Confidentiality_Tgs_bis)
done

```

```

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)"
apply (frule authK_authentic)
apply (frule_tac [3] Confidentiality_Kas)
apply (frule_tac [6] servTicket_authentic, auto)
apply (blast dest!: Confidentiality_Tgs_bis dest: Says_Kas_message_form servK_authentic
unique_servKeys unique_authKeys)
done

```

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

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)"
```

by (blast dest: u_servTicket_authentic u_NotexpiredSK_NotexpiredAK Confidentiality_Tgs_bis)

6.11 Parties authentication: each party verifies "the identity of another party who generated some data" (quoted from Neuman and Ts'o).

These guarantees don't assess whether two parties agree on the same session key: sending a message containing a key doesn't a priori state knowledge of the key.

Tgs_authenticates_A can be found above

lemma *A_authenticates_Tgs*:

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

apply (frule Says_Kas_message_form, assumption)

apply (erule rev_mp)

apply (erule rev_mp)

apply (erule rev_mp)

apply (erule kerbIV.induct, analz_mono_contra)

apply (frule_tac [7] K5_msg_in_parts_spies)

apply (frule_tac [5] K3_msg_in_parts_spies, simp_all, blast)

K2

apply (blast dest!: servK_authentic Says_Tgs_message_form authKeys_used)

K4

apply (blast dest!: unique_CryptKey)

done

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}},
```

$\text{Crypt servK } \{ \text{Agent } A, \text{ Number } T3 \} \in \text{set evs}$
 by (blast dest: servTicket_authentic_Tgs intro: Says_K5)

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"

by (blast intro: Says_K5 dest: Confidentiality_B servTicket_authentic_Tgs)

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"

by (blast intro: Says_K5 dest: u_Confidentiality_B servTicket_authentic_Tgs)

lemma *A_authenticates_B*:

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

by (blast dest: authK_authentic servK_authentic Says_Kas_message_form Key_unique_SesKey
 K4_imp_K2 intro: Says_K6)

lemma *A_authenticates_B_r*:

"[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);
 ¬ expiredAK Ta evs; ¬ expiredSK Ts evs;
 A ∉ bad; B ∉ bad; evs ∈ kerbIV]
 ⇒ Says B A (Crypt servK (Number T3)) ∈ set evs"

apply (frule authK_authentic)

apply (frule_tac [3] Says_Kas_message_form)

apply (frule_tac [4] Confidentiality_Kas)

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

```

apply (frule_tac [7] servK_authentic)
prefer 8 apply blast
apply (erule_tac [9] exE)
apply (frule_tac [9] K4_imp_K2)
apply assumption+
apply (blast dest: Key_unique_SesKey intro!: Says_K6 dest: Confidentiality_Tgs
)
done

```

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

```

```

    on evs"
apply (simp (no_asm) add: Issues_def)
apply (rule exI)
apply (rule conjI, assumption)
apply (simp (no_asm))
apply (erule rev_mp)
apply (erule kerbIV.induct)
apply (frule_tac [5] Says_ticket_parts)
apply (frule_tac [7] Says_ticket_parts)
apply (simp_all (no_asm_simp) add: all_conj_distrib)

```

K2

```

apply (simp add: takeWhile_tail)
apply (blast dest: authK_authentic parts_spies_takeWhile_mono [THEN subsetD]
parts_spies_ev_revD2 [THEN subsetD])
done

```

```

lemma A_authenticates_and_keydist_to_Kas:
  "[[ Crypt (shrK A) {Key authK, Peer, Ta, authTicket} ∈ parts (spies evs);
  A ∉ bad; evs ∈ kerbIV ]]
  ⇒ Kas Issues A with (Crypt (shrK A) {Key authK, Peer, Ta, authTicket})

```

```

    on evs"
by (blast dest: authK_authentic Kas_Issues_A)

```

```

lemma honest_never_says_newer_timestamp_in_auth:
  "[[ (CT evs) ≤ T; A ∉ bad; Number T ∈ parts {X}; evs ∈ kerbIV ]]
  ⇒ ∀ B Y. Says A B {Y, X} ∉ set evs"
apply (erule rev_mp)
apply (erule kerbIV.induct)
apply force+
done

```

lemma honest_never_says_current_timestamp_in_auth:

```
"[[ (CT evs) = T; Number T ∈ parts {X}; evs ∈ kerbIV ]]
  ⇒ ∀ A B Y. A ∉ bad → Says A B {Y, X} ∉ set evs"
  by (metis eq_imp_le honest_never_says_newer_timestamp_in_auth)
```

lemma A_trusts_secure_authenticator:

```
"[[ Crypt K {Agent A, Number T} ∈ parts (spies evs);
  Key K ∉ analz (spies evs); evs ∈ kerbIV ]]
  ⇒ ∃ B X. Says A Tgs {X, Crypt K {Agent A, Number T}, Agent B} ∈ set evs
  ∨
    Says A B {X, Crypt K {Agent A, Number T}} ∈ set evs"
  apply (erule rev_mp)
  apply (erule rev_mp)
  apply (erule kerbIV.induct, analz_mono_contra)
  apply (frule_tac [5] Says_ticket_parts)
  apply (frule_tac [7] Says_ticket_parts)
  apply (simp_all add: all_conj_distrib)
  apply blast+
  done
```

lemma A_Issues_Tgs:

```
"[[ Says A Tgs {authTicket, Crypt authK {Agent A, Number T2}, Agent B}
  ∈ set evs;
  Key authK ∉ analz (spies evs);
  A ∉ bad; evs ∈ kerbIV ]]
  ⇒ A Issues Tgs with (Crypt authK {Agent A, Number T2}) on evs"
  apply (simp (no_asm) add: Issues_def)
  apply (rule exI)
  apply (rule conjI, assumption)
  apply (simp (no_asm))
  apply (erule rev_mp)
  apply (erule rev_mp)
  apply (erule kerbIV.induct, analz_mono_contra)
  apply (frule_tac [5] Says_ticket_parts)
  apply (frule_tac [7] Says_ticket_parts)
  apply (simp_all (no_asm_simp) add: all_conj_distrib)
```

fake

apply blast

K3

```
apply (simp add: takeWhile_tail)
apply auto
apply (force dest!: authK_authentic Says_Kas_message_form)
apply (drule parts_spies_takeWhile_mono [THEN subsetD, THEN parts_spies_evs_revD2
  [THEN subsetD]])
apply (drule A_trusts_secure_authenticator, assumption, assumption)
apply (simp add: honest_never_says_current_timestamp_in_auth)
done
```

lemma Tgs_authenticates_and_keydist_to_A:

```
"[[ Crypt authK {Agent A, Number T2} ∈ parts (spies evs);
  Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}
  ∈ parts (spies evs);
```

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

```

    Key authK ∉ analz (spies evs);
    A ∉ bad; evs ∈ kerbIV ]
  ⇒ A Issues Tgs with (Crypt authK {Agent A, Number T2}) on evs"
by (blast dest: A_Issues_Tgs_authenticates_A)

lemma Tgs_Issues_A:
  "[ Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, servTicket
  })
    ∈ set evs;
    Key authK ∉ analz (spies evs); evs ∈ kerbIV ]
  ⇒ Tgs Issues A with
    (Crypt authK {Key servK, Agent B, Number Ts, servTicket }) on evs"
apply (simp (no_asm) add: Issues_def)
apply (rule exI)
apply (rule conjI, assumption)
apply (simp (no_asm))
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbIV.induct, analz_mono_contra)
apply (frule_tac [5] Says_ticket_parts)
apply (frule_tac [7] Says_ticket_parts)
apply (simp_all (no_asm_simp) add: all_conj_distrib)

K4

apply (simp add: takeWhile_tail)

apply (metis knows_Spy_partsEs(2) parts.Fst usedI used_evms_rev used_takeWhile_used)
done

lemma A_authenticates_and_keydist_to_Tgs:
  "[Crypt authK {Key servK, Agent B, Number Ts, servTicket} ∈ parts (spies evs);
  Key authK ∉ analz (spies evs); B ≠ Tgs; evs ∈ kerbIV ]
  ⇒ ∃A. Tgs Issues A with
    (Crypt authK {Key servK, Agent B, Number Ts, servTicket }) on evs"
by (blast dest: Tgs_Issues_A servK_authentic_bis)

lemma B_Issues_A:
  "[ Says B A (Crypt servK (Number T3)) ∈ set evs;
    Key servK ∉ analz (spies evs);
    A ∉ bad; B ∉ bad; B ≠ Tgs; evs ∈ kerbIV ]
  ⇒ B Issues A with (Crypt servK (Number T3)) on evs"
apply (simp (no_asm) add: Issues_def)
apply (rule exI)
apply (rule conjI, assumption)
apply (simp (no_asm))
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbIV.induct, analz_mono_contra)
apply (frule_tac [5] Says_ticket_parts)
apply (frule_tac [7] Says_ticket_parts)
apply (simp_all (no_asm_simp) add: all_conj_distrib)
apply blast

```

K6 requires numerous lemmas

```

apply (simp add: takeWhile_tail)
apply (blast dest: servTicket_authentic parts_spies_takeWhile_mono [THEN subsetD]
parts_spies_evs_revD2 [THEN subsetD] intro: Says_K6)
done

```

lemma *B_Issues_A_r*:

```

"[[ Says B A (Crypt servK (Number T3)) ∈ set 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;
  A ∉ bad; B ∉ bad; B ≠ Tgs; evs ∈ kerbIV ]]
⇒ B Issues A with (Crypt servK (Number T3)) on evs"
by (blast dest!: Confidentiality_B B_Issues_A)

```

lemma *u_B_Issues_A_r*:

```

"[[ Says B A (Crypt servK (Number T3)) ∈ set evs;
  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 ]]
⇒ B Issues A with (Crypt servK (Number T3)) on evs"
by (blast dest!: u_Confidentiality_B B_Issues_A)

```

lemma *A_authenticates_and_keydist_to_B*:

```

"[[ Crypt servK (Number T3) ∈ parts (spies evs);
  Crypt authK {Key servK, Agent B, Number Ts, servTicket}
  ∈ parts (spies evs);
  Crypt (shrK A) {Key authK, Agent Tgs, Number Ta, authTicket}
  ∈ parts (spies evs);
  Key authK ∉ analz (spies evs); Key servK ∉ analz (spies evs);
  A ∉ bad; B ∉ bad; B ≠ Tgs; evs ∈ kerbIV ]]
⇒ B Issues A with (Crypt servK (Number T3)) on evs"
by (blast dest!: A_authenticates_B B_Issues_A)

```

lemma *A_authenticates_and_keydist_to_B_r*:

```

"[[ 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);
  ¬ expiredAK Ta evs; ¬ expiredSK Ts evs;
  A ∉ bad; B ∉ bad; B ≠ Tgs; evs ∈ kerbIV ]]
⇒ B Issues A with (Crypt servK (Number T3)) on evs"
by (blast dest!: A_authenticates_B_r Confidentiality_Serv_A B_Issues_A)

```

lemma *A_Issues_B*:

```

"[[ Says A B {servTicket, Crypt servK {Agent A, Number T3}}
  ∈ set evs;

```

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

```

    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"
apply (simp (no_asm) add: Issues_def)
apply (rule exI)
apply (rule conjI, assumption)
apply (simp (no_asm))
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbIV.induct, analz_mono_contra)
apply (frule_tac [5] Says_ticket_parts)
apply (frule_tac [7] Says_ticket_parts)
apply (simp_all (no_asm_simp))
apply clarify

```

K5

```

apply auto
apply (simp add: takeWhile_tail)

```

Level 15: case analysis necessary because the assumption doesn't state the form of servTicket. The guarantee becomes stronger.

```

apply (blast dest: Says_imp_spies [THEN analz.Inj, THEN analz.Decrypt']
      K3_imp_K2 servK_authentic_ter
      parts_spies_takeWhile_mono [THEN subsetD]
      parts_spies_evs_revD2 [THEN subsetD]
      intro: Says_K5)
apply (simp add: takeWhile_tail)
done

```

lemma A_Issues_B_r:

```

  "[ Says A B {servTicket, Crypt servK {Agent A, Number T3}}
    ∈ set evs;
    Crypt (shrK A) {Key authK, Agent Tgs, Number Ta, authTicket}
    ∈ parts (spies evs);
    Crypt authK {Key servK, Agent B, Number Ts, servTicket}
    ∈ parts (spies evs);
    ¬ expiredAK Ta evs; ¬ expiredSK Ts evs;
    B ≠ Tgs; A ∉ bad; B ∉ bad; evs ∈ kerbIV ]
  ⇒ A Issues B with (Crypt servK {Agent A, Number T3}) on evs"
by (blast dest!: Confidentiality_Serv_A A_Issues_B)

```

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"
by (blast dest: B_authenticates_A A_Issues_B)

```

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"
by (blast dest: B_authenticates_A Confidentiality_B A_Issues_B)

```

`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"
by (blast dest: u_B_authenticates_A_r u_Confidentiality_B A_Issues_B)

```

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"
```

axiomatization where

```
Tgs_not_bad [iff]: "Tgs ∉ bad"
— Tgs is secure — we already know that Kas is secure
```

definition

```

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

```

definition

```
Unique :: "[event, event list] => bool" ("Unique _ on _" [0, 50] 50)
```

```

where "(Unique ev on evs) = (ev ∉ set (tl (dropWhile (% z. z ≠ ev) evs)))"

consts

  authKlife    :: nat

  servKlife    :: nat

  authlife     :: nat

  replylife    :: nat

specification (authKlife)
  authKlife_LB [iff]: "2 ≤ authKlife"
  by blast

specification (servKlife)
  servKlife_LB [iff]: "2 + authKlife ≤ servKlife"
  by blast

specification (authlife)
  authlife_LB [iff]: "Suc 0 ≤ authlife"
  by blast

specification (replylife)
  replylife_LB [iff]: "Suc 0 ≤ replylife"
  by blast

abbreviation

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

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

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

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

abbreviation
  valid :: "[nat, nat] => bool" ("valid _ wrt _" [0, 50] 50) where
    "valid T1 wrt T2 == T1 <= replylife + T2"

```

```

definition AKcryptSK :: "[key, key, event list] => bool" where
  "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_gets" :: "event list set"
where

  Nil: "[ ]  $\in$  kerbIV_gets"

  / Fake: "[ evsf  $\in$  kerbIV_gets; X  $\in$  synth (analz (spies evsf)) ]
     $\implies$  Says Spy B X # evsf  $\in$  kerbIV_gets"

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

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

  / K2: "[ evs2  $\in$  kerbIV_gets; Key authK  $\notin$  used evs2; authK  $\in$  symKeys;
    Gets 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_gets"

  / K3: "[ evs3  $\in$  kerbIV_gets;
    Says A Kas {Agent A, Agent Tgs, Number T1}  $\in$  set evs3;
    Gets 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_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"
apply (erule rev_mp)
apply (erule kerbIV_gets.induct)
apply auto
done

```

```

lemma Gets_imp_knows_Spy:
  "[ Gets B X ∈ set evs; evs ∈ kerbIV_gets ] ⇒ X ∈ knows Spy evs"
by (blast dest!: Gets_imp_Says Says_imp_knows_Spy)

```

```

declare Gets_imp_knows_Spy [THEN parts.Inj, dest]

```

```

lemma Gets_imp_knows:
  "[ Gets B X ∈ set evs; evs ∈ kerbIV_gets ] ⇒ X ∈ knows B evs"
by (metis Gets_imp_knows_Spy Gets_imp_knows_agents)

```

7.2 Lemmas about authKeys

```

lemma authKeys_empty: "authKeys [] = {}"

```

by (simp add: authKeys_def)

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"
```

by (unfold authKeys_def, auto)

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)"
```

by (unfold authKeys_def, auto)

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"
```

by (unfold authKeys_def, auto)

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"
```

by (unfold authKeys_def, auto)

lemma authKeys_used: "K ∈ authKeys evs ⇒ Key K ∈ used evs"

by (simp add: authKeys_def, blast)

7.3 Forwarding Lemmas

lemma Says_ticket_parts:

```
"Says S A (Crypt K {SesKey, B, TimeStamp, Ticket}) ∈ set evs
  ⇒ Ticket ∈ parts (spies evs)"
```

by blast

lemma Gets_ticket_parts:

```
"[[Gets A (Crypt K {SesKey, Peer, Ta, Ticket}) ∈ set evs; evs ∈ kerbIV_gets
]]
  ⇒ Ticket ∈ parts (spies evs)"
```

by (blast dest: Gets_imp_knows_Spy [THEN parts.Inj])

lemma Ops_range_spies1:

```
"[[ Says Kas A (Crypt KeyA {Key authK, Peer, Ta, authTicket})
  ∈ set evs ;
  evs ∈ kerbIV_gets ]] ⇒ authK ∉ range shrK & authK ∈ symkeys"
```

apply (erule rev_mp)

apply (erule kerbIV_gets.induct, auto)

done

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"
  apply (erule rev_mp)
  apply (erule kerbIV_gets.induct, auto)
  done

```

```

lemma Spy_see_shrK [simp]:
  "evs ∈ kerbIV_gets ⇒ (Key (shrK A) ∈ parts (spies evs)) = (A ∈ bad)"
  apply (erule kerbIV_gets.induct)
  apply (frule_tac [8] Gets_ticket_parts)
  apply (frule_tac [6] Gets_ticket_parts, simp_all)
  apply (blast+)
  done

```

```

lemma Spy_analz_shrK [simp]:
  "evs ∈ kerbIV_gets ⇒ (Key (shrK A) ∈ analz (spies evs)) = (A ∈ bad)"
  by auto

```

```

lemma Spy_see_shrK_D [dest!]:
  "[[ Key (shrK A) ∈ parts (spies evs); evs ∈ kerbIV_gets ]] ⇒ A:bad"
  by (blast dest: Spy_see_shrK)
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))"
  apply (erule rev_mp)
  apply (erule kerbIV_gets.induct)
  apply (frule_tac [8] Gets_ticket_parts)
  apply (frule_tac [6] Gets_ticket_parts, simp_all)

```

Fake

```

  apply (force dest!: keysFor_parts_insert)

```

Others

```

  apply (force dest!: analz_shrK_Decrypt)+
  done

```

```

lemma new_keys_not_analzD:
  "[[evs ∈ kerbIV_gets; K ∈ symKeys; Key K ∉ used evs]]
  ⇒ K ∉ keysFor (analz (spies evs))"
  by (blast dest: new_keys_not_used intro: keysFor_mono [THEN subsetD])

```

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})"
apply (erule rev_mp)
apply (erule kerbIV_gets.induct)
apply (simp_all (no_asm) add: authKeys_def authKeys_insert)
apply blast+
done

```

```

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"
apply (erule rev_mp)
apply (erule kerbIV_gets.induct)
apply (frule_tac [8] Gets_ticket_parts)
apply (frule_tac [6] Gets_ticket_parts, simp_all, blast)
done

```

```

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"
apply (erule rev_mp)
apply (erule kerbIV_gets.induct)
apply (frule_tac [8] Gets_ticket_parts)
apply (frule_tac [6] Gets_ticket_parts, simp_all)

```

Fake, K4

```

apply (blast+)
done

```

```

lemma authTicket_crypt_authK:
  "[[ Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}
    ∈ parts (spies evs);
    evs ∈ kerbIV_gets ] ]
  ==> authK ∈ authKeys evs"
apply (frule authTicket_authentic, assumption)
apply (simp (no_asm) add: authKeys_def)
apply blast
done

```

```

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})"
  apply (erule rev_mp)
  apply (erule kerbIV_gets.induct)
  apply (simp_all add: authKeys_insert authKeys_not_insert authKeys_empty authKeys_simp,
    blast, auto)

```

Three subcases of Message 4

```

  apply (blast dest!: SesKey_is_session_key)
  apply (blast dest: authTicket_crypt_authK)
  apply (blast dest!: authKeys_used Says_Kas_message_form)
done

```

```

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}]"
  apply (erule rev_mp)
  apply (erule kerbIV_gets.induct)
  apply (frule_tac [8] Gets_ticket_parts)
  apply (frule_tac [6] Gets_ticket_parts, simp_all)
  apply blast+
done

```

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})]"
  apply (erule rev_mp)
  apply (erule rev_mp)
  apply (erule kerbIV_gets.induct, analz_mono_contra)
  apply (frule_tac [8] Gets_ticket_parts)
  apply (frule_tac [6] Gets_ticket_parts, simp_all, blast)
done

```

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)"
by (blast dest: analz_shrK_Decrypt authTicket_form
    Gets_imp_knows_Spy [THEN analz.Inj])

```

```

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)"
apply (frule Gets_imp_knows_Spy [THEN analz.Inj], auto)
apply (force dest!: servTicket_form)
apply (frule analz_into_parts)
apply (frule servTicket_form, auto)
done

```

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"
apply (erule rev_mp)
apply (erule kerbIV_gets.induct)
apply (frule_tac [8] Gets_ticket_parts)
apply (frule_tac [6] Gets_ticket_parts, simp_all)

Fake

apply blast

K4

apply (blast dest!: authTicket_authentic [THEN Says_Kas_message_form])
done

```

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"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbIV_gets.induct, analz_mono_contra)
apply (frule_tac [8] Gets_ticket_parts)

```

apply (frule_tac [6] Gets_ticket_parts, simp_all)

Fake

apply blast

K2

apply blast

K4

apply auto

done

lemma servK_authentic_bis:

"[[Crypt authK {Key servK, Agent B, Number Ts, servTicket}
 ∈ parts (spies evs);
 Key authK ∉ analz (spies evs);
 B ≠ Tgs;
 evs ∈ kerbIV_gets]]

⇒ ∃ A. Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
 ∈ set evs"

apply (erule rev_mp)

apply (erule rev_mp)

apply (erule kerbIV_gets.induct, analz_mono_contra)

apply (frule_tac [8] Gets_ticket_parts)

apply (frule_tac [6] Gets_ticket_parts, simp_all)

Fake

apply blast

K4

apply blast

done

Authenticity of servK for B

lemma servTicket_authentic_Tgs:

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

⇒ ∃ authK.

Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts,
 Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}})
 ∈ set evs"

apply (erule rev_mp)

apply (erule rev_mp)

apply (erule kerbIV_gets.induct)

apply (frule_tac [8] Gets_ticket_parts)

apply (frule_tac [6] Gets_ticket_parts, simp_all)

apply blast+

done

Anticipated here from next subsection

lemma K4_imp_K2:

```

"[[ Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
  ∈ set evs; evs ∈ kerbIV_gets]]
⇒ ∃ Ta. Says Kas A
  (Crypt (shrK A)
   {Key authK, Agent Tgs, Number Ta,
    Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}})
  ∈ set evs"
apply (erule rev_mp)
apply (erule kerbIV_gets.induct)
apply (frule_tac [8] Gets_ticket_parts)
apply (frule_tac [6] Gets_ticket_parts, simp_all, auto)
apply (blast dest!: Gets_imp_knows_Spy [THEN parts.Inj, THEN parts.Fst, THEN
authTicket_authentic])
done

```

Anticipated here from next subsection

lemma *u_K4_imp_K2*:

```

"[[ Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
  ∈ set evs; evs ∈ kerbIV_gets]]
⇒ ∃ Ta. (Says Kas A (Crypt (shrK A) {Key authK, Agent Tgs, Number Ta,
  Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}})
  ∈ set evs
  & servKlife + Ts ≤ authKlife + Ta)"
apply (erule rev_mp)
apply (erule kerbIV_gets.induct)
apply (frule_tac [8] Gets_ticket_parts)
apply (frule_tac [6] Gets_ticket_parts, simp_all, auto)
apply (blast dest!: Gets_imp_knows_Spy [THEN parts.Inj, THEN parts.Fst, THEN
authTicket_authentic])
done

```

lemma *servTicket_authentic_Kas*:

```

"[[ Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}
  ∈ parts (spies evs); B ≠ Tgs; B ∉ bad;
  evs ∈ kerbIV_gets ]]
⇒ ∃ authK Ta.
  Says Kas A
  (Crypt (shrK A) {Key authK, Agent Tgs, Number Ta,
  Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}})
  ∈ set evs"
by (blast dest!: servTicket_authentic_Tgs K4_imp_K2)

```

lemma *u_servTicket_authentic_Kas*:

```

"[[ Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}
  ∈ parts (spies evs); B ≠ Tgs; B ∉ bad;
  evs ∈ kerbIV_gets ]]
⇒ ∃ authK Ta. Says Kas A (Crypt (shrK A) {Key authK, Agent Tgs, Number Ta,
  Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}})
  ∈ set evs
  & servKlife + Ts ≤ authKlife + Ta"
by (blast dest!: servTicket_authentic_Tgs u_K4_imp_K2)

```

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"
  by (blast dest: servTicket_authentic_Tgs K4_imp_K2)

```

```

lemma u_servTicket_authentic:
  "[[ Crypt (shrK B) {Agent B, 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]"
  by (blast dest: servTicket_authentic_Tgs u_K4_imp_K2)

```

```

lemma u_NotexpiredSK_NotexpiredAK:
  "[[ ¬ expiredSK Ts evs; servKlife + Ts ≤ authKlife + Ta ]
  ⇒ ¬ expiredAK Ta evs"
  by (blast dest: leI le_trans dest: leD)

```

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"
  apply (erule rev_mp)
  apply (erule kerbIV_gets.induct)
  apply (frule_tac [8] Gets_ticket_parts)
  apply (frule_tac [6] Gets_ticket_parts, simp_all, blast, blast)
  apply (blast dest: Gets_imp_knows_Spy [THEN parts.Inj, THEN authK_authentic])
  done

```

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}

```

7.6 Reliability: friendly agents send something if something else happened127

```

    ∈ 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'"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbIV_gets.induct, analz_mono_contra)
apply (frule_tac [8] Gets_ticket_parts)
apply (frule_tac [6] Gets_ticket_parts, simp_all)

Fake, K2, K4

apply (blast+)
done

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"
apply (drule authTicket_authentic, assumption, rotate_tac 4)
apply (erule rev_mp, erule rev_mp, erule rev_mp)
apply (erule kerbIV_gets.induct, analz_mono_contra)
apply (frule_tac [6] Gets_ticket_parts)
apply (frule_tac [9] Gets_ticket_parts)
apply (simp_all (no_asm_simp) add: all_conj_distrib)

Fake

apply blast

K2

apply (force dest!: Crypt_imp_keysFor)

K3

apply (blast dest: Key_unique_SesKey)

K5

If authKa were compromised, so would be authK

apply (case_tac "Key authKa ∈ analz (spies evs5)")
apply (force dest!: Gets_imp_knows_Spy [THEN analz.Inj, THEN analz.Decrypt,
  THEN analz.Fst])

Besides, since authKa originated with Kas anyway...

apply (clarify, drule K3_imp_K2, assumption, assumption)
apply (clarify, drule Says_Kas_message_form, assumption)

...it cannot be a shared key*. Therefore servK_authentic applies. Contradiction: Tgs
used authK as a servkey, while Kas used it as an authkey

```

```

apply (blast dest: servK_authentic Says_Tgs_message_form)
done

```

```

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"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbIV_gets.induct, analz_mono_contra)
apply (frule_tac [6] Gets_ticket_parts)
apply (frule_tac [9] Gets_ticket_parts)
apply (simp_all (no_asm_simp) add: all_conj_distrib)
apply blast

```

```

K3

```

```

apply (blast dest: authK_authentic Says_Kas_message_form Says_Tgs_message_form)

```

```

K4

```

```

apply (force dest!: Crypt_imp_keysFor)

```

```

K5

```

```

apply (blast dest: Key_unique_SesKey)
done

```

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'"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbIV_gets.induct, analz_mono_contra)
apply (frule_tac [8] Gets_ticket_parts)
apply (frule_tac [6] Gets_ticket_parts, simp_all)

```

```

Fake, K2, K4

```

```

apply (blast+)
done

```

```

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"
  apply (erule rev_mp)
  apply (erule rev_mp)
  apply (erule rev_mp)
  apply (erule kerbIV_gets.induct, analz_mono_contra)
  apply (frule_tac [8] Gets_ticket_parts)
  apply (frule_tac [6] Gets_ticket_parts)
  apply (simp_all (no_asm_simp))
  apply blast
  apply (force dest!: Crypt_imp_keysFor, clarify)
  apply (frule Says_Tgs_message_form, assumption, clarify)
  apply (blast dest: unique_CryptKey)
  done

```

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"
  apply (frule Says_Kas_message_form, assumption)
  apply (erule rev_mp)
  apply (erule rev_mp)
  apply (erule rev_mp)
  apply (erule kerbIV_gets.induct, analz_mono_contra)
  apply (frule_tac [8] Gets_ticket_parts)
  apply (frule_tac [6] Gets_ticket_parts, simp_all, blast)

```

K2 and K4 remain

```

prefer 2 apply (blast dest!: unique_CryptKey)
apply (blast dest!: servK_authentic Says_Tgs_message_form authKeys_used)
done

```

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'"
  apply (erule rev_mp)
  apply (erule rev_mp)
  apply (erule kerbIV_gets.induct)
  apply (frule_tac [8] Gets_ticket_parts)

```

```
apply (frule_tac [6] Gets_ticket_parts, simp_all)
```

K2

```
apply blast
done
```

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'"
```

```
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbIV_gets.induct)
apply (frule_tac [8] Gets_ticket_parts)
apply (frule_tac [6] Gets_ticket_parts, simp_all)
```

K4

```
apply blast
done
```

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"
```

```
apply (erule rev_mp, erule kerbIV_gets.induct, simp_all add: Unique_def)
apply blast
done
```

```
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"
```

```
apply (erule rev_mp, erule kerbIV_gets.induct, simp_all add: Unique_def)
apply blast
done
```

7.8 Lemmas About the Predicate $AKcryptSK$

```
lemma not_AKcryptSK_Nil [iff]: "¬ AKcryptSK authK servK []"
by (simp add: AKcryptSK_def)
```

```
lemma AKcryptSKI:
```

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

```
apply (unfold AKcryptSK_def)
```

```

apply (blast dest: Says_Tgs_message_form)
done

```

```

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"
by (auto simp add: AKcryptSK_def)

```

```

lemma Auth_fresh_not_AKcryptSK:
  "[ Key authK ∉ used evs; evs ∈ kerbIV_gets ]
  ⇒ ¬ AKcryptSK authK servK evs"
apply (unfold AKcryptSK_def)
apply (erule rev_mp)
apply (erule kerbIV_gets.induct)
apply (frule_tac [8] Gets_ticket_parts)
apply (frule_tac [6] Gets_ticket_parts, simp_all, blast)
done

```

```

lemma Serv_fresh_not_AKcryptSK:
  "Key servK ∉ used evs ⇒ ¬ AKcryptSK authK servK evs"
by (unfold AKcryptSK_def, blast)

```

```

lemma authK_not_AKcryptSK:
  "[ Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, tk}
    ∈ parts (spies evs); evs ∈ kerbIV_gets ]
  ⇒ ¬ AKcryptSK K authK evs"
apply (erule rev_mp)
apply (erule kerbIV_gets.induct)
apply (frule_tac [8] Gets_ticket_parts)
apply (frule_tac [6] Gets_ticket_parts, simp_all)

```

Fake

```
apply blast
```

Reception

```
apply (simp add: AKcryptSK_def)
```

K2: by freshness

```
apply (simp add: AKcryptSK_def)
```

K4

```
by (blast+)
```

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  $\notin$  analz (spies evs); SK  $\in$  symKeys;
      B  $\neq$  Tgs; evs  $\in$  kerbIV_gets ]
     $\implies \neg$  AKcryptSK SK K evs"
  apply (erule rev_mp)
  apply (erule rev_mp)
  apply (erule kerbIV_gets.induct, analz_mono_contra)
  apply (frule_tac [8] Gets_ticket_parts)
  apply (frule_tac [6] Gets_ticket_parts, simp_all, blast)

```

Reception

```

  apply (simp add: AKcryptSK_def)

```

K4 splits into distinct subcases

```

  apply auto

```

servK can't have been enclosed in two certificates

```

  prefer 2 apply (blast dest: unique_CryptKey)

```

servK is fresh and so could not have been used, by `new_keys_not_used`

```

  by (force dest!: Crypt_imp_invKey_keysFor simp add: AKcryptSK_def)

```

Long term keys are not issued as servKeys

```

lemma shrK_not_AKcryptSK:
  "evs  $\in$  kerbIV_gets  $\implies \neg$  AKcryptSK K (shrK A) evs"
  apply (unfold AKcryptSK_def)
  apply (erule kerbIV_gets.induct)
  apply (frule_tac [8] Gets_ticket_parts)
  by (frule_tac [6] Gets_ticket_parts, auto)

```

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

```

lemma Says_Tgs_AKcryptSK:
  "[[ Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, X })
     $\in$  set evs;
    authK'  $\neq$  authK; evs  $\in$  kerbIV_gets ]
   $\implies \neg$  AKcryptSK authK' servK evs"
  apply (unfold AKcryptSK_def)
  by (blast dest: unique_servKeys)

```

Equivalently

```

lemma not_different_AKcryptSK:
  "[[ AKcryptSK authK servK evs;
    authK'  $\neq$  authK; evs  $\in$  kerbIV_gets ]
   $\implies \neg$  AKcryptSK authK' servK evs  $\wedge$  servK  $\in$  symKeys"
  apply (simp add: AKcryptSK_def)
  by (blast dest: unique_servKeys Says_Tgs_message_form)

```

```

lemma AKcryptSK_not_AKcryptSK:
  "[[ AKcryptSK authK servK evs; evs  $\in$  kerbIV_gets ]
   $\implies \neg$  AKcryptSK servK K evs"
  apply (erule rev_mp)
  apply (erule kerbIV_gets.induct)

```

```

apply (frule_tac [8] Gets_ticket_parts)
apply (frule_tac [6] Gets_ticket_parts)

```

Reception

```

prefer 3 apply (simp add: AKcryptSK_def)
apply (simp_all, safe)

```

K4 splits into subcases

```

prefer 4 apply (blast dest!: authK_not_AKcryptSK)

```

servK is fresh and so could not have been used, by *new_keys_not_used*

```

prefer 2
apply (force dest!: Crypt_imp_invKey_keysFor simp add: AKcryptSK_def)

```

Others by freshness

```

by (blast+)

```

The only session keys that can be found with the help of session keys are those sent by Tgs in step K4.

We take some pains to express the property as a logical equivalence so that the simplifier can apply it.

```

lemma Key_analz_image_Key_lemma:
  "P  $\longrightarrow$  (Key K  $\in$  analz (Key'KK Un H))  $\longrightarrow$  (K:KK | Key K  $\in$  analz H)
   $\implies$ 
  P  $\longrightarrow$  (Key K  $\in$  analz (Key'KK Un H)) = (K:KK | Key K  $\in$  analz H)"
by (blast intro: analz_mono [THEN subsetD])

```

```

lemma AKcryptSK_analz_insert:
  "[[ AKcryptSK K K' evs; K  $\in$  symKeys; evs  $\in$  kerbIV_gets ]
   $\implies$  Key K'  $\in$  analz (insert (Key K) (spies evs))"
apply (simp add: AKcryptSK_def, clarify)
by (drule Says_imp_spies [THEN analz.Inj, THEN analz_insertI], auto)

```

```

lemma authKeys_are_not_AKcryptSK:
  "[[ K  $\in$  authKeys evs Un range shrK; evs  $\in$  kerbIV_gets ]
   $\implies$   $\forall$ SK.  $\neg$  AKcryptSK SK K evs  $\wedge$  K  $\in$  symKeys"
apply (simp add: authKeys_def AKcryptSK_def)
by (blast dest: Says_Kas_message_form Says_Tgs_message_form)

```

```

lemma not_authKeys_not_AKcryptSK:
  "[[ K  $\notin$  authKeys evs;
  K  $\notin$  range shrK; evs  $\in$  kerbIV_gets ]
   $\implies$   $\forall$ SK.  $\neg$  AKcryptSK K SK evs"
apply (simp add: AKcryptSK_def)
by (blast dest: Says_Tgs_message_form)

```

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"
by (blast dest: AKcryptSKI AKcryptSK_not_AKcryptSK)

```

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)))"
apply (erule kerbIV_gets.induct)
apply (frule_tac [11] Oops_range_spies2)
apply (frule_tac [10] Oops_range_spies1)
apply (frule_tac [8] Says_tgs_message_form)
apply (frule_tac [6] Says_kas_message_form)
apply (safe del: impI intro!: Key_analz_image_Key_lemma [THEN impI])

```

Case-splits for `Oops1` and message 5: the negated case simplifies using the induction hypothesis

```

apply (case_tac [12] "AKcryptSK authK SK evs01")
apply (case_tac [9] "AKcryptSK servK SK evs5")
apply (simp_all del: image_insert
  add: analz_image_freshK_simps AKcryptSK_Says shrK_not_AKcryptSK
       Oops2_not_AKcryptSK Auth_fresh_not_AKcryptSK
       Serv_fresh_not_AKcryptSK Says_Tgs_AKcryptSK Spy_analz_shrK)
  — 18 seconds on a 1.8GHz machine??

```

Fake

```
apply spy_analz
```

Reception

```
apply (simp add: AKcryptSK_def)
```

K2

```
apply blast
```

K3

```
apply blast
```

K4

```
apply (blast dest!: authK_not_AKcryptSK)
```

K5

```
apply (case_tac "Key servK ∈ analz (spies evs5) ")

```

If servK is compromised then the result follows directly...

```
apply (simp (no_asm_simp) add: analz_insert_eq Un_upper2 [THEN analz_mono,
THEN subsetD])
```

...therefore servK is uncompromised.

The AKcryptSK servK SK evs5 case leads to a contradiction.

```
apply (blast elim!: servK_not_AKcryptSK [THEN [2] rev_notE] del: allE ballE)
```

Another K5 case

```
apply blast
```

Oops1

```
apply simp
by (blast dest!: AKcryptSK_analz_insert)
```

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))"
apply (frule authKeys_are_not_AKcryptSK, assumption)
apply (simp del: image_insert
  add: analz_image_freshK_simps add: Key_analz_image_Key)
done
```

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))"
apply (frule not_authKeys_not_AKcryptSK, assumption, assumption)
apply (simp del: image_insert
  add: analz_image_freshK_simps add: Key_analz_image_Key)
done
```

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))"
apply (drule_tac authK' = authK' in not_different_AKcryptSK, blast, assumption)
apply (simp del: image_insert
  add: analz_image_freshK_simps add: Key_analz_image_Key)
done
```

```
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))"
apply (frule AKcryptSKI, assumption)
by (simp add: analz_insert_freshK3)

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)"
by (force dest: Says_imp_spies [THEN analz.Inj, THEN analz.Decrypt, THEN analz.Fst])

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"
apply (erule rev_mp)
apply (erule rev_mp)
apply (simp add: authKeys_def)
apply (erule kerbIV_gets.induct, analz_mono_contra)
apply (frule_tac [8] Gets_ticket_parts)
apply (frule_tac [6] Gets_ticket_parts, simp_all)
by (blast+)

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"
apply (erule kerbIV_gets.induct)
apply (frule_tac [11] Ops_range_spies2)
apply (frule_tac [10] Ops_range_spies1)
apply (frule_tac [8] Says_tgs_message_form)
apply (frule_tac [6] Says_kas_message_form)
apply (safe del: impI conjI impCE)
apply (simp_all (no_asm_simp) add: Says_Kas_message_form less_SucI analz_insert_eq
not_parts_not_analz analz_insert_freshK1 pushes)

Fake

apply spy_analz

```

K2

apply blast

K4

apply blast

Level 8: K5

apply (blast dest: servK_notin_authKeysD Says_Kas_message_form intro: less_SucI)

Oops1

apply (blast dest!: unique_authKeys intro: less_SucI)

Oops2

by (blast dest: Says_Tgs_message_form Says_Kas_message_form)

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)"
```

by (blast dest: Says_Kas_message_form Confidentiality_Kas_lemma)

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"
```

apply (erule rev_mp)

apply (erule rev_mp)

apply (erule kerbIV_gets.induct)

apply (rule_tac [10] impI)+

— The Oops1 case is unusual: must simplify $\text{Authkey} \notin \text{analz} (\text{knows Spy } (ev \# \text{ evs}))$, not letting `analz_mono_contra` weaken it to $\text{Authkey} \notin \text{analz} (\text{knows Spy } \text{ evs})$, for we then conclude $\text{authK} \neq \text{authKa}$.

apply analz_mono_contra

apply (frule_tac [11] Oops_range_spies2)

apply (frule_tac [10] Oops_range_spies1)

apply (frule_tac [8] Says_tgs_message_form)

apply (frule_tac [6] Says_kas_message_form)

apply (safe del: impI conjI impCE)

apply (simp_all add: less_SucI new_keys_not_analz_d Says_Kas_message_form Says_Tgs_message_form

analz_insert_eq not_parts_not_analz analz_insert_freshK1 analz_insert_freshK2

analz_insert_freshK3_bis pushes)

Fake

apply *spy_analz*

K2

apply (*blast intro: parts_insertI less_SucI*)

K4

apply (*blast dest: authTicket_authentic Confidentiality_Kas*)

Oops2

prefer 3

apply (*blast dest: Says_imp_spies [THEN parts.Inj] Key_unique_SesKey intro: less_SucI*)

Oops1

prefer 2

apply (*blast dest: Says_Kas_message_form Says_Tgs_message_form intro: less_SucI*)

K5. Not clear how this step could be integrated with the main simplification step.

Done in KerberosV.thy

apply *clarify*

apply (*erule_tac V = "Says Aa Tgs ?X ∈ set ?evs" in thin_rl*)

apply (*frule Gets_imp_knows_Spy [THEN parts.Inj, THEN servK_notin_authKeysD]*)

apply (*assumption, assumption, blast, assumption*)

apply (*simp add: analz_insert_freshK2*)

by (*blast dest: Says_imp_spies [THEN parts.Inj] Key_unique_SesKey intro: less_SucI*)

In the real world Tgs can't check wheter authK is secure!

lemma *Confidentiality_Tgs:*

```
"[[ Says Tgs A
  (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
  ∈ set evs;
  Key authK ∉ analz (spies evs);
  ¬ expiredSK Ts evs;
  A ∉ bad; B ∉ bad; evs ∈ kerbIV_gets ]
  ⇒ Key servK ∉ analz (spies evs)"
```

by (*blast dest: Says_Tgs_message_form Confidentiality_lemma*)

In the real world Tgs CAN check what Kas sends!

lemma *Confidentiality_Tgs_bis:*

```
"[[ Says Kas A
  (Crypt Ka {Key authK, Agent Tgs, Number Ta, authTicket})
  ∈ set evs;
  Says Tgs A
  (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
  ∈ set evs;
  ¬ expiredAK Ta evs; ¬ expiredSK Ts evs;
  A ∉ bad; B ∉ bad; evs ∈ kerbIV_gets ]
  ⇒ Key servK ∉ analz (spies evs)"
```

by (*blast dest!: Confidentiality_Kas Confidentiality_Tgs*)

Most general form

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

lemmas Confidentiality_Tgs_ter = authTicket_authentic [THEN Confidentiality_Tgs_bis]

lemmas Confidentiality_Auth_A = authK_authentic [THEN Confidentiality_Kas]

Needs a confidentiality guarantee, hence moved here. Authenticity of servK for A

lemma servK_authentic_bis_r:

```
"[ Crypt (shrK A) {Key authK, Agent Tgs, Number Ta, authTicket}
  ∈ parts (spies evs);
  Crypt authK {Key servK, Agent B, Number Ts, servTicket}
  ∈ parts (spies evs);
  ¬ expiredAK Ta evs; A ∉ bad; evs ∈ kerbIV_gets ]
⇒ Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
  ∈ set evs"
```

by (blast dest: authK_authentic Confidentiality_Auth_A servK_authentic_ter)

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_gets ]
⇒ Key servK ∉ analz (spies evs)"
```

by (metis Confidentiality_Auth_A Confidentiality_Tgs K4_imp_K2 authK_authentic authTicket_form servK_authentic unique_authKeys)

lemma u_Confidentiality_B:

```
"[ Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}
  ∈ parts (spies evs);
  ¬ expiredSK Ts evs;
  A ∉ bad; B ∉ bad; B ≠ Tgs; evs ∈ kerbIV_gets ]
⇒ Key servK ∉ analz (spies evs)"
```

by (blast dest: u_servTicket_authentic u_NotexpiredSK_NotexpiredAK Confidentiality_Tgs_bis)

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

Authentication here still is weak agreement - of B with A

lemma A_authenticates_B:

```
"[ Crypt servK (Number T3) ∈ parts (spies evs);
  Crypt authK {Key servK, Agent B, Number Ts, servTicket}
  ∈ parts (spies evs);
  Crypt (shrK A) {Key authK, Agent Tgs, Number Ta, authTicket}
  ∈ parts (spies evs);
  Key authK ∉ analz (spies evs); Key servK ∉ analz (spies evs);
  A ∉ bad; B ∉ bad; evs ∈ kerbIV_gets ]
⇒ Says B A (Crypt servK (Number T3)) ∈ set evs"
```

by (blast dest: authK_authentic servK_authentic Says_Kas_message_form Key_unique_SesKey K4_imp_K2 intro: Says_K6)

lemma *shrK_in_initState_Server*[iff]: "Key (shrK A) \in initState Kas"
by (induct_tac "A", auto)

lemma *shrK_in_knows_Server* [iff]: "Key (shrK A) \in knows Kas evs"
by (simp add: initState_subset_knows [THEN subsetD])

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 Says Kas A (Crypt (shrK A) {Key authK, Peer, Ta, authTicket}) \in set
 evs
 \wedge Key authK \in analz(knows Kas evs)"
by (force dest!: authK_authentic Says_imp_knows [THEN analz.Inj, THEN analz.Decrypt,
 THEN analz.Fst])

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 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"
apply (erule rev_mp)
apply (erule kerbIV_gets.induct)
apply auto
apply (blast dest: authTicket_form)
done

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 } \in set evs;
 Key authK \notin analz (spies evs); A \notin bad; evs \in kerbIV_gets]
 $\implies \exists B$. 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
 \wedge Key authK \in analz (knows A evs)"
apply (frule Gets_imp_knows_Spy [THEN parts.Inj, THEN parts.Fst], assumption)
apply (drule Gets_imp_knows_Spy [THEN parts.Inj, THEN parts.Snd, THEN parts.Fst],
 assumption)
apply (drule Tgs_authenticates_A, assumption+, simp)
apply (force dest!: K3_imp_Gets_evs Gets_imp_knows [THEN analz.Inj, THEN analz.Decrypt,
 THEN analz.Fst])
done

lemma *K4_imp_Gets*:
 "[Says Tgs A (Crypt authK {Key servK, Agent B, Number Ts, servTicket})
 \in set evs; evs \in kerbIV_gets]
 $\implies \exists Ta$ X.

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

```

    Gets Tgs {Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}},
X}
    ∈ set evs"
apply (erule rev_mp)
apply (erule kerbIV_gets.induct)
apply auto
done

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)"
apply (drule Gets_imp_knows_Spy [THEN parts.Inj], assumption)
apply (drule Gets_imp_knows_Spy [THEN parts.Inj], assumption)
apply (frule authK_authentic, assumption+)
apply (drule servK_authentic_ter, assumption+)
apply (frule K4_imp_Gets, assumption, erule exE, erule exE)
apply (drule Gets_imp_knows [THEN analz.Inj, THEN analz.Fst, THEN analz.Decrypt,
  THEN analz.Snd, THEN analz.Snd, THEN analz.Fst], assumption, force)
apply (metis Says_imp_knows analz.Fst analz.Inj analz_symKeys_Decrypt authTicket_form)
done

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"
apply (erule rev_mp)
apply (erule kerbIV_gets.induct)
apply auto
done

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"
apply (erule rev_mp)
apply (erule kerbIV_gets.induct)
apply auto
done

lemma B_authenticates_and_keydist_to_A:

```

```

    "[ Gets B {Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}},
      Crypt servK {Agent A, Number T3}} ∈ set evs;
      Key servK ∉ analz (spies evs);
      A ∉ bad; B ∉ bad; B ≠ Tgs; evs ∈ kerbIV_gets ]
  ⇒ Says A B {Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}},
      Crypt servK {Agent A, Number T3}} ∈ set evs
  ∧ Key servK ∈ analz (knows A evs)"
  apply (frule Gets_imp_knows_Spy [THEN parts.Inj, THEN parts.Fst, THEN servTicket_authentic_Tgs
  assumption+])
  apply (drule Gets_imp_knows_Spy [THEN parts.Inj, THEN parts.Snd], assumption)
  apply (erule exE, drule Says_K5, assumption+)
  apply (frule K5_imp_Gets, assumption+)
  apply clarify
  apply (drule K3_imp_Gets, assumption+)
  apply (erule exE)
  apply (frule Gets_imp_knows_Spy [THEN parts.Inj, THEN authK_authentic, THEN
  Says_Kas_message_form], assumption+, clarify)
  apply (force dest!: Gets_imp_knows [THEN analz.Inj, THEN analz.Decrypt, THEN
  analz.Fst])
done

```

lemma K6_imp_Gets:

```

    "[ Says B A (Crypt servK (Number T3)) ∈ set evs;
      B ∉ bad; evs ∈ kerbIV_gets ]
  ⇒ ∃ Ts X. Gets B {Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}}, X}
  ∈ set evs"
  apply (erule rev_mp)
  apply (erule kerbIV_gets.induct)
  apply auto
done

```

lemma A_authenticates_and_keydist_to_B:

```

    "[ Gets A {Crypt authK {Key servK, Agent B, Number Ts, servTicket}},
      Crypt servK (Number T3)} ∈ set evs;
      Gets A (Crypt (shrK A) {Key authK, Agent Tgs, Number Ta, authTicket})
      ∈ set evs;
      Key authK ∉ analz (spies evs); Key servK ∉ analz (spies evs);
      A ∉ bad; B ∉ bad; evs ∈ kerbIV_gets ]
  ⇒ Says B A (Crypt servK (Number T3)) ∈ set evs
  ∧ Key servK ∈ analz (knows B evs)"
  apply (frule Gets_imp_knows_Spy [THEN parts.Inj, THEN parts.Fst], assumption)
  apply (drule Gets_imp_knows_Spy [THEN parts.Inj, THEN parts.Snd], assumption)
  apply (drule Gets_imp_knows_Spy [THEN parts.Inj], assumption)
  apply (drule A_authenticates_B, assumption+)
  apply (force dest!: K6_imp_Gets Gets_imp_knows [THEN analz.Inj, THEN analz.Fst,
  THEN analz.Decrypt, THEN analz.Snd, THEN analz.Snd, THEN analz.Fst])
done

```

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"
```

axiomatization **where**

```
Tgs_not_bad [iff]: "Tgs ∉ bad"
  — Tgs is secure — we already know that Kas is secure
```

definition

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

definition

```
Issues :: "[agent, agent, msg, event list] => bool"
  ("_ Issues _ with _ on _") where
  "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))))"
```

consts

```
authKlife  :: nat
```

```
servKlife  :: nat
```

```
authlife   :: nat
```

```
replylife  :: nat
```

specification (*authKlife*)

```
authKlife_LB [iff]: "2 ≤ authKlife"
  by blast
```

specification (servKlife)

```
servKlife_LB [iff]: "2 + authKlife ≤ servKlife"
  by blast
```

specification (authlife)

```
authlife_LB [iff]: "Suc 0 ≤ authlife"
  by blast
```

specification (replylife)

```
replylife_LB [iff]: "Suc 0 ≤ replylife"
  by blast
```

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"
```

definition AKcryptSK :: "[key, key, event list] => bool" where

```
"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"

| Oops2: "[[ 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)"
apply (induct_tac "evs")
apply (induct_tac [2] "a", auto)
done

lemma spies_Gets_rev: "spies (evs @ [Gets A X]) = spies evs"
apply (induct_tac "evs")
apply (induct_tac [2] "a", auto)
done

lemma spies_Notes_rev: "spies (evs @ [Notes A X]) =
  (if A:bad then insert X (spies evs) else spies evs)"
apply (induct_tac "evs")
apply (induct_tac [2] "a", auto)

```

done

```
lemma spies_evs_rev: "spies evs = spies (rev evs)"
apply (induct_tac "evs")
apply (induct_tac [2] "a")
apply (simp_all (no_asm_simp) add: spies_Says_rev spies_Gets_rev spies_Notes_rev)
done
```

```
lemmas parts_spies_evs_revD2 = spies_evs_rev [THEN equalityD2, THEN parts_mono]
```

```
lemma spies_takeWhile: "spies (takeWhile P evs) <= spies evs"
apply (induct_tac "evs")
apply (induct_tac [2] "a", auto)
```

Resembles `used_subset_append` in theory `Event`.

done

```
lemmas parts_spies_takeWhile_mono = spies_takeWhile [THEN parts_mono]
```

8.2 Lemmas about authKeys

```
lemma authKeys_empty: "authKeys [] = {}"
by (simp add: authKeys_def)
```

```
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"
by (auto simp add: authKeys_def)
```

```
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)"
by (auto simp add: authKeys_def)
```

```
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"
by (auto simp add: authKeys_def)
```

```
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"
by (auto simp add: authKeys_def)
```

```
lemma authKeys_used: "K ∈ authKeys evs ⇒ Key K ∈ used evs"
by (auto simp add: authKeys_def)
```

8.3 Forwarding Lemmas

lemma *Says_ticket_parts*:

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

by blast

lemma *Says_ticket_analz*:

"Says S A {Crypt K {SesKey, B, TimeStamp}, Ticket}
 \in set evs \implies Ticket \in analz (spies evs)"

by (blast dest: Says_imp_knows_Spy [THEN analz.Inj, THEN analz.Snd])

lemma *Ops_range_spies1*:

"[[Says Kas A {Crypt KeyA {Key authK, Peer, Ta}, authTicket}
 \in set evs ;
 evs \in kerbV]] \implies authK \notin range shrK & authK \in symKeys"

apply (erule rev_mp)

apply (erule kerbV.induct, auto)

done

lemma *Ops_range_spies2*:

"[[Says Tgs A {Crypt authK {Key servK, Agent B, Ts}, servTicket}
 \in set evs ;
 evs \in kerbV]] \implies servK \notin range shrK \wedge servK \in symKeys"

apply (erule rev_mp)

apply (erule kerbV.induct, auto)

done

lemma *Spy_see_shrK [simp]*:

"evs \in kerbV \implies (Key (shrK A) \in parts (spies evs)) = (A \in bad)"

apply (erule kerbV.induct)

apply (frule_tac [7] Says_ticket_parts)

apply (frule_tac [5] Says_ticket_parts, simp_all)

apply (blast+)

done

lemma *Spy_analz_shrK [simp]*:

"evs \in kerbV \implies (Key (shrK A) \in analz (spies evs)) = (A \in bad)"

by auto

lemma *Spy_see_shrK_D [dest!]*:

"[[Key (shrK A) \in parts (spies evs); evs \in kerbV]] \implies A:bad"

by (blast dest: Spy_see_shrK)

lemmas *Spy_analz_shrK_D* = analz_subset_parts [THEN subsetD, THEN Spy_see_shrK_D,
 dest!]

Nobody can have used non-existent keys!

lemma *new_keys_not_used [simp]*:

"[[Key K \notin used evs; K \in symKeys; evs \in kerbV]]
 \implies K \notin keysFor (parts (spies evs))"

apply (erule rev_mp)

```

apply (erule kerbV.induct)
apply (frule_tac [7] Says_ticket_parts)
apply (frule_tac [5] Says_ticket_parts, simp_all)

```

Fake

```

apply (force dest!: keysFor_parts_insert)

```

Others

```

apply (force dest!: analz_shrK_Decrypt)+
done

```

lemma new_keys_not_analzd:

```

"[[evs ∈ kerbV; K ∈ symKeys; Key K ∉ used evs]
 ⇒ K ∉ keysFor (analz (spies evs))"

```

```

by (blast dest: new_keys_not_used intro: keysFor_mono [THEN subsetD])

```

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"

```

```

apply (erule rev_mp)
apply (erule kerbV.induct)
apply (simp_all (no_asm) add: authKeys_def authKeys_insert)
apply blast+
done

```

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"

```

```

apply (erule rev_mp)
apply (erule kerbV.induct)
apply (frule_tac [7] Says_ticket_parts)
apply (frule_tac [5] Says_ticket_parts, simp_all, blast)
done

```

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"
  apply (erule rev_mp)
  apply (erule kerbV.induct)
  apply (frule_tac [7] Says_ticket_parts)
  apply (frule_tac [5] Says_ticket_parts, simp_all)

```

Fake, K4

```

  apply (blast+)
  done

```

```

lemma authTicket_crypt_authK:
  "[[ Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, Number Ta}
    ∈ parts (spies evs);
    evs ∈ kerbV ]
  ⇒ authK ∈ authKeys evs"
by (metis authKeysI authTicket_authentic)

```

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"
  apply (erule rev_mp)
  apply (erule kerbV.induct)
  apply (simp_all add: authKeys_insert authKeys_not_insert authKeys_empty authKeys_simp,
    blast, auto)

```

Three subcases of Message 4

```

  apply (blast dest!: authKeys_used Says_Kas_message_form)
  apply (blast dest!: SesKey_is_session_key)
  apply (blast dest: authTicket_crypt_authK)
  done

```

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"
  apply (erule rev_mp)
  apply (erule kerbV.induct)
  apply (frule_tac [7] Says_ticket_parts)
  apply (frule_tac [5] Says_ticket_parts, simp_all)

```

```

apply blast+
done

```

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"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbV.induct, analz_mono_contra)
apply (frule_tac [7] Says_ticket_parts)
apply (frule_tac [5] Says_ticket_parts, simp_all)
apply blast+
done

```

```

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"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbV.induct, analz_mono_contra)
apply (frule_tac [7] Says_ticket_parts)
apply (frule_tac [5] Says_ticket_parts, simp_all, blast+)
done

```

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"
apply (erule rev_mp)
apply (erule kerbV.induct)
apply (frule_tac [7] Says_ticket_parts)
apply (frule_tac [5] Says_ticket_parts, simp_all, blast+)
done

```

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"
  apply (erule rev_mp)
  apply (erule kerbV.induct)
  apply (frule_tac [7] Says_ticket_parts)
  apply (frule_tac [5] Says_ticket_parts, simp_all, auto)
  apply (metis MPair_analz Says_imp_analz_Spy analz_conj_parts authTicket_authentic)
  done

```

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"

```

```

  apply (erule rev_mp)
  apply (erule kerbV.induct)
  apply (frule_tac [7] Says_ticket_parts)
  apply (frule_tac [5] Says_ticket_parts, simp_all, auto)
  apply (blast dest!: Says_imp_spies [THEN parts.Inj, THEN parts.Fst, THEN authTicket_authentic])
  done

```

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"
  by (metis K4_imp_K2 servTicket_authentic_Tgs)

```

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"
  by (metis servTicket_authentic_Tgs u_K4_imp_K2)

```

lemma servTicket_authentic:

```

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

```

8.6 Reliability: friendly agents send something if something else happened 153

```

⇒ ∃ 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"
by (metis K4_imp_K2 servTicket_authentic_Tgs)

```

```

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"
by (metis servTicket_authentic_Tgs u_K4_imp_K2)

```

```

lemma u_NotexpiredSK_NotexpiredAK:
  "[ ¬ expiredSK Ts evs; servKlife + Ts ≤ authKlife + Ta ]
  ⇒ ¬ expiredAK Ta evs"
by (metis order_1e_less_trans)

```

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"

apply (erule rev_mp)
apply (erule kerbV.induct)
apply (frule_tac [7] Says_ticket_parts)
apply (frule_tac [5] Says_ticket_parts, simp_all, blast, blast)
apply (blast dest: Says_imp_spies [THEN parts.Inj, THEN parts.Fst, THEN authK_authentic])
done

```

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 ]

```

```

     $\implies K=K' \wedge B=B' \wedge T=T''$ 
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbV.induct, analz_mono_contra)
apply (frule_tac [7] Says_ticket_parts)
apply (frule_tac [5] Says_ticket_parts, simp_all)

```

Fake, K2, K4

```

apply (blast+)
done

```

This inevitably has an existential form in version V

lemma Says_K5:

```

    "[[ Crypt servK {Agent A, Number T3}  $\in$  parts (spies evs);
      Says Tgs A {Crypt authK {Key servK, Agent B, Number Ts},
                  servTicket}  $\in$  set evs;
      Key servK  $\notin$  analz (spies evs);
      A  $\notin$  bad; B  $\notin$  bad; evs  $\in$  kerbV ]]
     $\implies \exists ST. \text{Says } A \ B \ \{ST, \text{Crypt servK } \{Agent \ A, \ Number \ T3\}\} \in \text{set evs}$ "
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbV.induct, analz_mono_contra)
apply (frule_tac [5] Says_ticket_parts)
apply (frule_tac [7] Says_ticket_parts)
apply (simp_all (no_asm_simp) add: all_conj_distrib)
apply blast

```

K3

```

apply (blast dest: authK_authentic Says_Kas_message_form Says_Tgs_message_form)

```

K4

```

apply (force dest!: Crypt_imp_keysFor)

```

K5

```

apply (blast dest: Key_unique_SesKey)
done

```

Anticipated here from next subsection

lemma unique_CryptKey:

```

    "[[ Crypt (shrK B) {Agent A, Agent B, Key SesKey, T}
       $\in$  parts (spies evs);
      Crypt (shrK B') {Agent A', Agent B', Key SesKey, T'}
       $\in$  parts (spies evs); Key SesKey  $\notin$  analz (spies evs);
      evs  $\in$  kerbV ]]
     $\implies A=A' \ \& \ B=B' \ \& \ T=T''$ "
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbV.induct, analz_mono_contra)
apply (frule_tac [7] Says_ticket_parts)
apply (frule_tac [5] Says_ticket_parts, simp_all)

```

8.6 Reliability: friendly agents send something if something else happened 155

Fake, K2, K4

```
apply (blast+)
done
```

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"
apply (frule Says_Tgs_message_form, assumption, clarify)
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbV.induct, analz_mono_contra)
apply (frule_tac [7] Says_ticket_parts)
apply (frule_tac [5] Says_ticket_parts)
apply simp_all
```

fake

```
apply blast
```

K4

```
apply (force dest!: Crypt_imp_keysFor)
```

K6

```
apply (metis MPair_parts Says_imp_parts_knows_Spy unique_CryptKey)
done
```

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"
apply (frule Says_Kas_message_form, assumption)
apply clarify
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbV.induct, analz_mono_contra)
apply (frule_tac [7] Says_ticket_parts)
apply (frule_tac [5] Says_ticket_parts, simp_all, blast)
```

K2 and K4 remain

```
apply (blast dest!: servK_authentic Says_Tgs_message_form authKeys_used)
apply (blast dest!: unique_CryptKey)
done
```

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'"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbV.induct)
apply (frule_tac [7] Says_ticket_parts)
apply (frule_tac [5] Says_ticket_parts, simp_all)
apply blast+
done

```

`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'"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbV.induct)
apply (frule_tac [7] Says_ticket_parts)
apply (frule_tac [5] Says_ticket_parts, simp_all)
apply blast+
done

```

8.8 Lemmas About the Predicate `AKcryptSK`

```

lemma not_AKcryptSK_Nil [iff]: "¬ AKcryptSK authK servK []"
apply (simp add: AKcryptSK_def)
done

```

```

lemma AKcryptSKI:
  "[[ Says Tgs A {Crypt authK {Key servK, Agent B, tt}}, X } ∈ set evs;
    evs ∈ kerbV ] ⇒ AKcryptSK authK servK evs"
by (metis AKcryptSK_def Says_Tgs_message_form)

```

```

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)"
by (auto simp add: AKcryptSK_def)

```

```

lemma AKcryptSK_Notes [simp]:
  "AKcryptSK authK servK (Notes A X # evs) =

```

```

    AKcryptSK authK servK evs"
  by (auto simp add: AKcryptSK_def)

```

```

lemma Auth_fresh_not_AKcryptSK:
  "[[ Key authK ∉ used evs; evs ∈ kerbV ]]
  ⇒ ¬ AKcryptSK authK servK evs"
apply (unfold AKcryptSK_def)
apply (erule rev_mp)
apply (erule kerbV.induct)
apply (frule_tac [7] Says_ticket_parts)
apply (frule_tac [5] Says_ticket_parts, simp_all, blast)
done

```

```

lemma Serv_fresh_not_AKcryptSK:
  "Key servK ∉ used evs ⇒ ¬ AKcryptSK authK servK evs"
by (auto simp add: AKcryptSK_def)

```

```

lemma authK_not_AKcryptSK:
  "[[ Crypt (shrK Tgs) {Agent A, Agent Tgs, Key authK, tk}
  ∈ parts (spies evs); evs ∈ kerbV ]]
  ⇒ ¬ AKcryptSK K authK evs"
apply (erule rev_mp)
apply (erule kerbV.induct)
apply (frule_tac [7] Says_ticket_parts)
apply (frule_tac [5] Says_ticket_parts, simp_all)

```

Fake,K2,K4

```

apply (auto simp add: AKcryptSK_def)
done

```

A secure serverkey cannot have been used to encrypt others

```

lemma servK_not_AKcryptSK:
  "[[ Crypt (shrK B) {Agent A, Agent B, Key SK, tt} ∈ parts (spies evs);
  Key SK ∉ analz (spies evs); SK ∈ symKeys;
  B ≠ Tgs; evs ∈ kerbV ]]
  ⇒ ¬ AKcryptSK SK K evs"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbV.induct, analz_mono_contra)
apply (frule_tac [7] Says_ticket_parts)
apply (frule_tac [5] Says_ticket_parts, simp_all, blast)

```

K4

```

apply (metis Auth_fresh_not_AKcryptSK MPair_parts Says_imp_parts_knows_Spy
  authKeys_used authTicket_crypt_authK unique_CryptKey)
done

```

Long term keys are not issued as servKeys

```

lemma shrK_not_AKcryptSK:
  "evs ∈ kerbV ⇒ ¬ AKcryptSK K (shrK A) evs"
apply (unfold AKcryptSK_def)

```

```

apply (erule kerbV.induct)
apply (frule_tac [7] Says_ticket_parts)
apply (frule_tac [5] Says_ticket_parts, auto)
done

```

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

```

lemma Says_Tgs_AKcryptSK:
  "[[ Says Tgs A {Crypt authK {Key servK, Agent B, tt}}, X }
    ∈ set evs;
    authK' ≠ authK; evs ∈ kerbV ]
  ⇒ ¬ AKcryptSK authK' servK evs"
by (metis AKcryptSK_def unique_servKeys)

```

```

lemma AKcryptSK_not_AKcryptSK:
  "[[ AKcryptSK authK servK evs; evs ∈ kerbV ]
  ⇒ ¬ AKcryptSK servK K evs"
apply (erule rev_mp)
apply (erule kerbV.induct)
apply (frule_tac [7] Says_ticket_parts)
apply (frule_tac [5] Says_ticket_parts)
apply (simp_all, safe)

```

K4 splits into subcases

```

prefer 4 apply (blast dest!: authK_not_AKcryptSK)

```

servK is fresh and so could not have been used, by *new_keys_not_used*

```

prefer 2
apply (force dest!: Crypt_imp_invKey_keysFor simp add: AKcryptSK_def)

```

Others by freshness

```

apply (blast+)
done

```

```

lemma not_different_AKcryptSK:
  "[[ AKcryptSK authK servK evs;
    authK' ≠ authK; evs ∈ kerbV ]
  ⇒ ¬ AKcryptSK authK' servK evs ∧ servK ∈ symKeys"
apply (simp add: AKcryptSK_def)
apply (blast dest: unique_servKeys Says_Tgs_message_form)
done

```

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)"
by (blast intro: analz_mono [THEN subsetD])

```

```

lemma AKcryptSK_analz_insert:
  "[[ AKcryptSK K K' evs; K ∈ symKeys; evs ∈ kerbV ]]
  ⇒ Key K' ∈ analz (insert (Key K) (spies evs))"
apply (simp add: AKcryptSK_def, clarify)
apply (drule Says_imp_spies [THEN analz.Inj, THEN analz_insertI], auto)
done

```

```

lemma authKeys_are_not_AKcryptSK:
  "[[ K ∈ authKeys evs Un range shrK; evs ∈ kerbV ]]
  ⇒ ∀SK. ¬ AKcryptSK SK K evs ∧ K ∈ symKeys"
apply (simp add: authKeys_def AKcryptSK_def)
apply (blast dest: Says_Kas_message_form Says_Tgs_message_form)
done

```

```

lemma not_authKeys_not_AKcryptSK:
  "[[ K ∉ authKeys evs;
    K ∉ range shrK; evs ∈ kerbV ]]
  ⇒ ∀SK. ¬ AKcryptSK K SK evs"
apply (simp add: AKcryptSK_def)
apply (blast dest: Says_Tgs_message_form)
done

```

8.9 Secrecy Theorems

For the Oops2 case of the next theorem

```

lemma Oops2_not_AKcryptSK:
  "[[ evs ∈ kerbV;
    Says Tgs A {Crypt authK
      {Key servK, Agent B, Number Ts}, servTicket}
    ∈ set evs ]]
  ⇒ ¬ AKcryptSK servK SK evs"
by (blast dest: AKcryptSKI AKcryptSK_not_AKcryptSK)

```

Big simplification law for keys SK that are not crypted by keys in KK It helps prove three, otherwise hard, facts about keys. These facts are exploited as simplification laws for analz, and also "limit the damage" in case of loss of a key to the spy. See ESORICS98.

```

lemma Key_analz_image_Key [rule_format (no_asm)]:
  "evs ∈ kerbV ⇒
  (∀SK KK. SK ∈ symKeys & KK ≤ -(range shrK) →
  (∀K ∈ KK. ¬ AKcryptSK K SK evs) →
  (Key SK ∈ analz (Key'KK Un (spies evs))) =
  (SK ∈ KK | Key SK ∈ analz (spies evs)))"
apply (erule kerbV.induct)
apply (frule_tac [10] Oops_range_spies2)
apply (frule_tac [9] Oops_range_spies1)

apply (drule_tac [7] Says_ticket_analz)

apply (drule_tac [5] Says_ticket_analz)
apply (safe del: impI intro!: Key_analz_image_Key_lemma [THEN impI])

```

Case-splits for Oops1 and message 5: the negated case simplifies using the induction hypothesis

```

apply (case_tac [9] "AKcryptSK authK SK evs01")
apply (case_tac [7] "AKcryptSK servK SK evs5")
apply (simp_all del: image_insert
  add: analz_image_freshK_simps AKcryptSK_Says shrK_not_AKcryptSK
  Oops2_not_AKcryptSK Auth_fresh_not_AKcryptSK
  Serv_fresh_not_AKcryptSK Says_Tgs_AKcryptSK Spy_analz_shrK)

```

Fake

```
apply spy_analz
```

K2

```
apply blast
```

Cases K3 and K5 solved by the simplifier thanks to the ticket being in analz - this strategy is new wrt version IV

K4

```
apply (blast dest!: authK_not_AKcryptSK)
```

Oops1

```
apply (metis AKcryptSK_analz_insert insert_Key_singleton)
done
```

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

```

lemma analz_insert_freshK1:
  "[[ evs ∈ kerbV; K ∈ authKeys evs Un range shrK;
    SesKey ∉ range shrK ]]"
  ⇒ (Key K ∈ analz (insert (Key SesKey) (spies evs))) =
    (K = SesKey | Key K ∈ analz (spies evs))"
apply (frule authKeys_are_not_AKcryptSK, assumption)
apply (simp del: image_insert
  add: analz_image_freshK_simps add: Key_analz_image_Key)
done

```

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

```

lemma analz_insert_freshK2:
  "[[ evs ∈ kerbV; servK ∉ (authKeys evs); servK ∉ range shrK;
    K ∈ symKeys ]]"
  ⇒ (Key K ∈ analz (insert (Key servK) (spies evs))) =
    (K = servK | Key K ∈ analz (spies evs))"
apply (frule not_authKeys_not_AKcryptSK, assumption, assumption)
apply (simp del: image_insert
  add: analz_image_freshK_simps add: Key_analz_image_Key)
done

```

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 ∈ kerbV ]
  ⇒ (Key servK ∈ analz (insert (Key authK') (spies evs))) =
    (servK = authK' | Key servK ∈ analz (spies evs))"
apply (drule_tac authK' = authK' in not_different_AKcryptSK, blast, assumption)
apply (simp del: image_insert
        add: analz_image_freshK_simps add: Key_analz_image_Key)
done

```

```

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))"
apply (frule AKcryptSKI, assumption)
apply (simp add: analz_insert_freshK3)
done

```

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)"
by (metis Says_imp_analz_Spy analz.Fst analz.Decrypt')

```

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"
apply (erule kerbV.induct)
apply (frule_tac [10] Oops_range_spies2)
apply (frule_tac [9] Oops_range_spies1)
apply (frule_tac [7] Says_ticket_analz)
apply (frule_tac [5] Says_ticket_analz)
apply (safe del: impI conjI impCE)
apply (simp_all (no_asm_simp) add: Says_Kas_message_form less_SucI analz_insert_eq
  not_parts_not_analz analz_insert_freshK1 pushes)

```

Fake

```
apply spy_analz
```

K2

```
apply blast
```

K4

```

apply blast
Oops1
apply (blast dest!: unique_authKeys intro: less_SucI)
Oops2
apply (blast dest: Says_Tgs_message_form Says_Kas_message_form)
done

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)"
apply (blast dest: Says_Kas_message_form Confidentiality_Kas_lemma)
done

```

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"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbV.induct)
apply (rule_tac [9] impI)+
  — The Oops1 case is unusual: must simplify Authkey ∉ analz (knows Spy (ev
  # evs)), not letting analz_mono_contra weaken it to Authkey ∉ analz (knows Spy
  evs), for we then conclude authK ≠ authKa.
apply analz_mono_contra
apply (frule_tac [10] Oops_range_spies2)
apply (frule_tac [9] Oops_range_spies1)
apply (frule_tac [7] Says_ticket_analz)
apply (frule_tac [5] Says_ticket_analz)
apply (safe del: impI conjI impCE)
apply (simp_all add: less_SucI new_keys_not_analzd Says_Kas_message_form Says_Tgs_message_form
  analz_insert_eq not_parts_not_analz analz_insert_freshK1 analz_insert_freshK2
  analz_insert_freshK3_bis pushes)

Fake

  apply spy_analz

K2

  apply (blast intro: parts_insertI less_SucI)

K4

```

```

apply (blast dest: authTicket_authentic Confidentiality_Kas)

Oops1

apply (blast dest: Says_Kas_message_form Says_Tgs_message_form intro: less_SucI)

Oops2

apply (metis Suc_le_eq linorder_linear linorder_not_le msg.simps(2) unique_servKeys)
done

```

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)"
by (blast dest: Says_Tgs_message_form Confidentiality_lemma)

```

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)"
by (blast dest!: Confidentiality_Kas Confidentiality_Tgs)

```

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"
by (metis Confidentiality_Kas authK_authentic servK_authentic_ter)

```

```

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)"
apply (drule authK_authentic, assumption, assumption)
apply (blast dest: Confidentiality_Kas Says_Kas_message_form servK_authentic_ter
Confidentiality_Tgs_bis)
done

lemma Confidentiality_B:
"[[ Crypt (shrK B) {Agent A, Agent B, Key servK, Number Ts}
  ∈ parts (spies evs);
  Crypt authK {Key servK, Agent B, Number Ts}
  ∈ parts (spies evs);
  Crypt (shrK A) {Key authK, Agent Tgs, Number Ta}
  ∈ parts (spies evs);
  ¬ expiredSK Ts evs; ¬ expiredAK Ta evs;
  A ∉ bad; B ∉ bad; B ≠ Tgs; evs ∈ kerbV ]]
⇒ Key servK ∉ analz (spies evs)"
apply (frule authK_authentic)
apply (erule_tac [3] exE)
apply (frule_tac [3] Confidentiality_Kas)
apply (frule_tac [6] servTicket_authentic, auto)
apply (blast dest!: Confidentiality_Tgs_bis dest: Says_Kas_message_form servK_authentic
unique_servKeys unique_authKeys)
done

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)"
by (blast dest: u_servTicket_authentic u_NotexpiredSK_NotexpiredAK Confidentiality_Tgs_bis)

```

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 ]]

```

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

```

  ⇒ ∃ ST. Says A B {ST, Crypt servK {Agent A, Number T3}} ∈ set evs"
by (blast dest: servTicket_authentic_Tgs intro: Says_K5)

```

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"
by (blast intro: Says_K5 dest: Confidentiality_B servTicket_authentic_Tgs)

```

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"
by (blast intro: Says_K5 dest: u_Confidentiality_B servTicket_authentic_Tgs)

```

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"
by (metis authK_authentic Oops_range_spies1 Says_K6 servK_authentic u_K4_imp_K2
unique_authKeys)

```

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"
apply (frule authK_authentic)
apply (erule_tac [3] exE)
apply (frule_tac [3] Says_Kas_message_form)
apply (frule_tac [4] Confidentiality_Kas)
apply (frule_tac [7] servK_authentic)

```

```

apply auto
apply (metis Confidentiality_Tgs K4_imp_K2 Says_K6 unique_authKeys)
done

```

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"
apply (simp (no_asm) add: Issues_def)
apply (rule exI)
apply (rule conjI, assumption)
apply (simp (no_asm))
apply (erule rev_mp)
apply (erule kerbV.induct)
apply (frule_tac [5] Says_ticket_parts)
apply (frule_tac [7] Says_ticket_parts)
apply (simp_all (no_asm_simp) add: all_conj_distrib)

K2

apply (simp add: takeWhile_tail)
apply (metis MPair_parts parts.Body parts_idem parts_spies_takeWhile_mono
parts_trans spies_evs_rev usedI)
done

```

```

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"
by (blast dest!: authK_authentic Kas_Issues_A)

```

```

lemma Tgs_Issues_A:
  "[ Says Tgs A {Crypt authK {Key servK, Agent B, Number Ts}}, servTicket}
  ∈ set evs;
  Key authK ∉ analz (spies evs); evs ∈ kerbV ]
  ⇒ Tgs Issues A with
  (Crypt authK {Key servK, Agent B, Number Ts}) on evs"
apply (simp (no_asm) add: Issues_def)
apply (rule exI)
apply (rule conjI, assumption)
apply (simp (no_asm))
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbV.induct, analz_mono_contra)
apply (frule_tac [5] Says_ticket_parts)

```

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

```

apply (frule_tac [7] Says_ticket_parts)
apply (simp_all (no_asm_simp) add: all_conj_distrib)
apply (simp add: takeWhile_tail)

apply (blast dest: servK_authentic parts_spies_takeWhile_mono [THEN subsetD]
  parts_spies_evs_revD2 [THEN subsetD] authTicket_authentic
  Says_Kas_message_form)
done

```

```

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"
by (blast dest: Tgs_Issues_A servK_authentic_bis)

```

```

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"
apply (simp (no_asm) add: Issues_def)
apply (rule exI)
apply (rule conjI, assumption)
apply (simp (no_asm))
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbV.induct, analz_mono_contra)
apply (simp_all (no_asm_simp) add: all_conj_distrib)
apply blast

```

K6 requires numerous lemmas

```

apply (simp add: takeWhile_tail)
apply (blast intro: Says_K6 dest: servTicket_authentic
  parts_spies_takeWhile_mono [THEN subsetD]
  parts_spies_evs_revD2 [THEN subsetD])
done

```

```

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"
by (blast dest!: A_authenticates_B B_Issues_A)

```

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"

```

```

apply (erule rev_mp)
apply (erule kerbV.induct)
apply auto
done

```

```

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"
by (metis honest_never_says_newer_timestamp_in_auth le_refl)

```

```

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"
apply (simp (no_asm) add: Issues_def)
apply (rule exI)
apply (rule conjI, assumption)
apply (simp (no_asm))
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule kerbV.induct, analz_mono_contra)
apply (frule_tac [7] Says_ticket_parts)
apply (frule_tac [5] Says_ticket_parts)
apply (simp_all (no_asm_simp))

```

K5

```

apply auto
apply (simp add: takeWhile_tail)

```

Level 15: case study necessary because the assumption doesn't state the form of servTicket. The guarantee becomes stronger.

```

prefer 2 apply (simp add: takeWhile_tail)

```

```

apply (frule K3_imp_K2, assumption, assumption, erule exE, erule exE)
apply (case_tac "Key authK ∈ analz (spies evs5)")
  apply (metis Says_imp_analz_Spy analz.Fst analz.Decrypt')
apply (frule K3_imp_K2, assumption, assumption, erule exE, erule exE)
apply (drule Says_imp_knows_Spy [THEN parts.Inj, THEN parts.Fst])
apply (frule servK_authentic_ter, blast, assumption+)
apply (drule parts_spies_takeWhile_mono [THEN subsetD])
apply (drule parts_spies_evs_revD2 [THEN subsetD])

```

Says_K5 closes the proof in version IV because it is clear which servTicket an authenticator appears with in msg 5. In version V an authenticator can appear with any item that the spy could replace the servTicket with

```

apply (frule Says_K5, blast)

```

We need to state that an honest agent wouldn't send the wrong timestamp within an authenticator, whatever it is paired with

```

apply (auto simp add: honest_never_says_current_timestamp_in_auth)
done

```

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

```

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"
by (blast dest: B_authenticates_A A_Issues_B)

```

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'"
apply (erule rev_mp, erule rev_mp)
apply (erule kerbV.induct)
apply (auto simp add: honest_never_says_current_timestamp_in_auth)
done

```

```

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'"
apply (erule rev_mp, erule rev_mp)
apply (erule kerbV.induct)
apply (auto simp add: honest_never_says_current_timestamp_in_auth)
done

```

```

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'"
apply (erule rev_mp, erule rev_mp)
apply (erule kerbV.induct)
apply (auto simp add: honest_never_says_current_timestamp_in_auth)
done

```

The second part of the message is treated as an authenticator by the last sim-

plification 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'"
apply (erule rev_mp, erule rev_mp)
apply (erule kerbV.induct)
apply (auto simp add: honest_never_says_current_timestamp_in_auth)
done

```

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'"
apply (erule rev_mp, erule rev_mp)
apply (erule kerbV.induct)
apply (auto simp add: honest_never_says_current_timestamp_in_auth)
done

```

```

lemma Kas_never_says_newer_timestamp:
  "[[ (CT evs) ≤ T; Number T ∈ parts {X}; evs ∈ kerbV ]
  ⇒ ∀ A. Says Kas A X ∉ set evs"
apply (erule rev_mp)
apply (erule kerbV.induct, auto)
done

```

```

lemma Kas_never_says_current_timestamp:
  "[[ (CT evs) = T; Number T ∈ parts {X}; evs ∈ kerbV ]
  ⇒ ∀ A. Says Kas A X ∉ set evs"
by (metis Kas_never_says_newer_timestamp eq_imp_le)

```

```

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'"
apply (erule rev_mp, erule rev_mp)
apply (erule kerbV.induct)
apply (auto simp add: Kas_never_says_current_timestamp)
done

```

```

lemma Tgs_never_says_newer_timestamp:
  "[[ (CT evs) ≤ T; Number T ∈ parts {X}; evs ∈ kerbV ]

```

```

    ==> ∀ A. Says Tgs A X ∉ set evs"
  apply (erule rev_mp)
  apply (erule kerbV.induct, auto)
  done

lemma Tgs_never_says_current_timestamp:
  "[ (CT evs) = T; Number T ∈ parts {X}; evs ∈ kerbV ]
  ==> ∀ A. Says Tgs A X ∉ set evs"
by (metis Tgs_never_says_newer_timestamp eq_imp_le)

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'"
  apply (erule rev_mp, erule rev_mp)
  apply (erule kerbV.induct)
  apply (auto simp add: Tgs_never_says_current_timestamp)
  done

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"

| Oops: "[| 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"
apply (intro exI bexI)
apply (rule_tac [2] otway.Nil
    [THEN otway.OR1, THEN otway.Reception,
     THEN otway.OR2, THEN otway.Reception,
     THEN otway.OR3, THEN otway.Reception, THEN otway.OR4])

apply (possibility, simp add: used_Cons)
done

```

```

lemma Gets_imp_Says [dest!]:
  "[| Gets B X ∈ set evs; evs ∈ otway |] ==> ∃A. Says A B X ∈ set evs"
apply (erule rev_mp)
apply (erule otway.induct, auto)
done

```

```

lemma OR2_analz_knows_Spy:
  "[| Gets B {|N, Agent A, Agent B, X|} ∈ set evs; evs ∈ otway |]
  ==> X ∈ analz (knows Spy evs)"
by blast

```

```

lemma OR4_analz_knows_Spy:
  "[| Gets B {|N, X, Crypt (shrK B) X'|} ∈ set evs; evs ∈ otway |]
  ==> X ∈ analz (knows Spy evs)"
by blast

```

```

lemmas OR2_parts_knows_Spy =
  OR2_analz_knows_Spy [THEN analz_into_parts]

```

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)"
by (erule otway.induct, force,
    drule_tac [4] OR2_parts_knows_Spy, simp_all, blast+)

```

```

lemma Spy_analz_shrK [simp]:
  "evs ∈ otway ==> (Key (shrK A) ∈ analz (knows Spy evs)) = (A ∈ bad)"
by auto

```

```

lemma Spy_see_shrK_D [dest!]:
  "[|Key (shrK A) ∈ parts (knows Spy evs); evs ∈ otway|] ==> A ∈ bad"
by (blast dest: Spy_see_shrK)

```

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)"
by (erule rev_mp, erule otway.induct, simp_all)

```

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))"
apply (erule otway.induct)
apply (frule_tac [8] Says_Server_message_form)
apply (drule_tac [7] OR4_analz_knows_Spy)
apply (drule_tac [5] OR2_analz_knows_Spy, analz_freshK, spy_analz, auto)
done

```

```

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))"
by (simp only: analz_image_freshK analz_image_freshK_simps)

```

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'"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule otway.induct, simp_all)
apply blast+ — OR3 and OR4
done

```

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"
by (erule otway.induct, force,
  drule_tac [4] OR2_parts_knows_Spy, simp_all, blast+)

```

```

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"
by (blast dest: Crypt_imp_OR1)

```

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"

```

```

apply (erule rev_mp, erule rev_mp)
apply (erule otway.induct, force,
  drule_tac [4] OR2_parts_knows_Spy, simp_all, blast+)
done

```

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)"
apply (erule rev_mp)
apply (erule otway.induct, force,
  drule_tac [4] OR2_parts_knows_Spy, simp_all, blast+)
done

```

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)"
apply (erule otway.induct, force,
  drule_tac [4] OR2_parts_knows_Spy, simp_all, blast)
apply blast — OR1: by freshness
apply (blast dest!: no_nonce_OR1_OR2 intro: unique_NA) — OR3
apply (blast intro!: Crypt_imp_OR1) — OR4
done

```

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"
by (blast intro!: NA_Crypt_imp_Server_msg)

```

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  $\notin$  bad; B  $\notin$  bad; evs  $\in$  otway |]
  ==> Says Server B
    {|NA, Crypt (shrK A) {|NA, Key K|},
     Crypt (shrK B) {|NB, Key K|}|}  $\in$  set evs -->
    Notes Spy {|NA, NB, Key K|}  $\notin$  set evs -->
    Key K  $\notin$  analz (knows Spy evs)"
apply (erule otway.induct, force)
apply (frule_tac [7] Says_Server_message_form)
apply (drule_tac [6] OR4_analz_knows_Spy)
apply (drule_tac [4] OR2_analz_knows_Spy)
apply (simp_all add: analz_insert_eq analz_insert_freshK pushes)
apply spy_analz — Fake
apply (blast dest: unique_session_keys)+ — OR3, OR4, Oops
done

```

```

theorem Spy_not_see_encrypted_key:
  "[| Says Server B
    {|NA, Crypt (shrK A) {|NA, Key K|},
     Crypt (shrK B) {|NB, Key K|}|}  $\in$  set evs;
    Notes Spy {|NA, NB, Key K|}  $\notin$  set evs;
    A  $\notin$  bad; B  $\notin$  bad; evs  $\in$  otway |]
  ==> Key K  $\notin$  analz (knows Spy evs)"
by (blast dest: Says_Server_message_form secrecy_lemma)

```

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 `Key K \notin knows Spy evs` appears not to be inductive: all the cases other than Fake are trivial, while Fake requires `Key K \notin 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|}|}  $\in$  set evs;
    Notes Spy {|NA, NB, Key K|}  $\notin$  set evs;
    A  $\notin$  bad; B  $\notin$  bad; evs  $\in$  otway |]
  ==> Key K  $\notin$  knows Spy evs"
by (blast dest: Spy_not_see_encrypted_key)

```

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|}|}  $\in$  set evs;
    Says B' A {|NA, Crypt (shrK A) {|NA, Key K|}|}  $\in$  set evs;
     $\forall$  NB. Notes Spy {|NA, NB, Key K|}  $\notin$  set evs;
    A  $\notin$  bad; B  $\notin$  bad; evs  $\in$  otway |]
  ==> Key K  $\notin$  analz (knows Spy evs)"
by (blast dest!: A_trusts_OR4 Spy_not_see_encrypted_key)

```

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"
apply (erule rev_mp)
apply (erule otway.induct, force,
        drule_tac [4] OR2_parts_knows_Spy, simp_all, blast+)
done

```

The Nonce NB uniquely identifies B's message

```

lemma unique_NB:
  "[| Crypt (shrK B) {|NA, NB, Agent A, Agent B|} ∈ parts(knows Spy evs);
    Crypt (shrK B) {|NC, NB, Agent C, Agent B|} ∈ parts(knows Spy evs);
    evs ∈ otway; B ∉ bad |]
  ==> NC = NA & C = A"
apply (erule rev_mp, erule rev_mp)
apply (erule otway.induct, force,
        drule_tac [4] OR2_parts_knows_Spy, simp_all)
apply blast+ — Fake, OR2
done

```

If the encrypted message appears, and B has used Nonce NB, then it originated with the Server! Quite messy proof.

```

lemma NB_Crypt_imp_Server_msg [rule_format]:
  "[| B ∉ bad; evs ∈ otway |]
  ==> Crypt (shrK B) {|NB, Key K|} ∈ parts (knows Spy evs)
  --> (∃X'. Says B Server
        {|NA, Agent A, Agent B, X',
          Crypt (shrK B) {|NA, NB, Agent A, Agent B|}|}
        ∈ set evs
  --> Says Server B
        {|NA, Crypt (shrK A) {|NA, Key K|},
          Crypt (shrK B) {|NB, Key K|}|}
        ∈ set evs)"
apply simp
apply (erule otway.induct, force,
        drule_tac [4] OR2_parts_knows_Spy, simp_all)
apply blast — Fake
apply blast — OR2
apply (blast dest: unique_NB dest!: no_nonce_OR1_OR2) — OR3
apply (blast dest!: Crypt_imp_OR2) — OR4
done

```

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"
by (blast intro!: NB_Crypt_imp_Server_msg)

```

The obvious combination of *B_trusts_OR3* with *Spy_not_see_encrypted_key*

```

lemma B_gets_good_key:
  "[| Says B Server {|NA, Agent A, Agent B, X',
                  Crypt (shrK B) {|NA, NB, Agent A, Agent B|} |}
    ∈ set evs;
    Gets B {|NA, X, Crypt (shrK B) {|NB, Key K|}|} ∈ set evs;
    Notes Spy {|NA, NB, Key K|} ∉ set evs;
    A ∉ bad; B ∉ bad; evs ∈ otway |]
  ==> Key K ∉ analz (knows Spy evs)"
by (blast dest!: B_trusts_OR3 Spy_not_see_encrypted_key)

```

```

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"
apply (erule rev_mp)
apply (erule otway.induct, simp_all)
apply (blast dest!: Crypt_imp_OR2)+
done

```

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"
by (blast dest!: A_trusts_OR4 OR3_imp_OR2)

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 (shr*K* *A*) {|Nonce *NA*, Agent *A*, Agent *B*, Key *KAB*|},
Crypt (shr*K* *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 (shr*K* *B*){|Nonce *NB*, Agent *A*, Agent *B*, Key *K*|}|}

```

    ∈ set evs4 []
    ==> Says B A X # evs4 ∈ otway"

```

I 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"
apply (intro exI bexI)
apply (rule_tac [2] otway.Nil
  [THEN otway.OR1, THEN otway.Reception,
   THEN otway.OR2, THEN otway.Reception,
   THEN otway.OR3, THEN otway.Reception, THEN otway.OR4])
apply (possibility, simp add: used_Cons)
done

```

```

lemma Gets_imp_Says [dest!]:
  "[| Gets B X ∈ set evs; evs ∈ otway |] ==> ∃ A. Says A B X ∈ set evs"
by (erule rev_mp, erule otway.induct, auto)

```

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)"
by blast

```

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)"
by (erule otway.induct, simp_all, blast+)

```

```

lemma Spy_analz_shrK [simp]:
  "evs ∈ otway ==> (Key (shrK A) ∈ analz (knows Spy evs)) = (A ∈ bad)"
by auto

```

```

lemma Spy_see_shrK_D [dest!]:
  "[|Key (shrK A) ∈ parts (knows Spy evs); evs ∈ otway|] ==> A ∈ bad"
by (blast dest: Spy_see_shrK)

```

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)"
apply (erule rev_mp)
apply (erule otway.induct, auto)
done

```

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))"
apply (erule otway.induct)
apply (frule_tac [8] Says_Server_message_form)
apply (drule_tac [7] OR4_analz_knows_Spy, analz_freshK, spy_analz, auto)
done

```

```

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))"
by (simp only: analz_image_freshK analz_image_freshK_simps)

```

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'"
apply (erule rev_mp, erule rev_mp, erule otway.induct, simp_all)
apply blast+ — OR3 and OR4
done

```

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  $\notin$  bad; A  $\neq$  B; evs  $\in$  otway |]
  ==> Crypt (shrK A) {|NA, Agent A, Agent B, Key K|}  $\in$  parts (knows Spy
  evs)
  --> ( $\exists$  NB. Says Server B
        {|Crypt (shrK A) {|NA, Agent A, Agent B, Key K|},
         Crypt (shrK B) {|NB, Agent A, Agent B, Key K|}|}
         $\in$  set evs)"
apply (erule otway.induct, force)
apply (simp_all add: ex_disj_distrib)
apply blast+ — Fake, OR3
done

```

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|})  $\in$  set
  evs;
   A  $\notin$  bad; A  $\neq$  B; evs  $\in$  otway |]
  ==>  $\exists$  NB. Says Server B
        {|Crypt (shrK A) {|NA, Agent A, Agent B, Key K|},
         Crypt (shrK B) {|NB, Agent A, Agent B, Key K|}|}
         $\in$  set evs"
by (blast intro!: NA_Crypt_imp_Server_msg)

```

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  $\notin$  bad; B  $\notin$  bad; evs  $\in$  otway |]
  ==> Says Server B
        {|Crypt (shrK A) {|NA, Agent A, Agent B, Key K|},
         Crypt (shrK B) {|NB, Agent A, Agent B, Key K|}|}
         $\in$  set evs -->
        Notes Spy {|NA, NB, Key K|}  $\notin$  set evs -->
        Key K  $\notin$  analz (knows Spy evs)"
apply (erule otway.induct, force)
apply (frule_tac [7] Says_Server_message_form)
apply (drule_tac [6] OR4_analz_knows_Spy)
apply (simp_all add: analz_insert_eq analz_insert_freshK pushes)
apply spy_analz — Fake
apply (blast dest: unique_session_keys)+ — OR3, OR4, Oops
done

```

```

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|}|}
         $\in$  set evs;

```

```

Notes Spy {|NA, NB, Key K|}  $\notin$  set evs;
A  $\notin$  bad; B  $\notin$  bad; evs  $\in$  otway []
==> Key K  $\notin$  analz (knows Spy evs)"
by (metis secrecy_lemma)

```

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|})  $\in$  set
evs;
   $\forall$  NB. Notes Spy {|NA, NB, Key K|}  $\notin$  set evs;
  A  $\notin$  bad; B  $\notin$  bad; A  $\neq$  B; evs  $\in$  otway []
  ==> Key K  $\notin$  analz (knows Spy evs)"
by (metis A_trusts_OR4 secrecy_lemma)

```

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  $\notin$  bad; A  $\neq$  B; evs  $\in$  otway []
  ==> Crypt (shrK B) {|NB, Agent A, Agent B, Key K|}  $\in$  parts (knows Spy evs)
  --> ( $\exists$  NA. Says Server B
      {|Crypt (shrK A) {|NA, Agent A, Agent B, Key K|},
       Crypt (shrK B) {|NB, Agent A, Agent B, Key K|}|}
       $\in$  set evs)"
apply (erule otway.induct, force, simp_all add: ex_disj_distrib)
apply blast+ — Fake, OR3
done

```

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|}|}
   $\in$  set evs;
  B  $\notin$  bad; A  $\neq$  B; evs  $\in$  otway []
  ==>  $\exists$  NA. Says Server B
      {|Crypt (shrK A) {|NA, Agent A, Agent B, Key K|},
       Crypt (shrK B) {|NB, Agent A, Agent B, Key K|}|}
       $\in$  set evs"
by (blast intro!: NB_Crypt_imp_Server_msg)

```

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|}|}
   $\in$  set evs;
   $\forall$  NA. Notes Spy {|NA, NB, Key K|}  $\notin$  set evs;
  A  $\notin$  bad; B  $\notin$  bad; A  $\neq$  B; evs  $\in$  otway []
  ==> Key K  $\notin$  analz (knows Spy evs)"
by (blast dest: B_trusts_OR3 Spy_not_see_encrypted_key)

```

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"
apply (intro exI bexI)
apply (rule_tac [2] otway.Nil
      [THEN otway.OR1, THEN otway.Reception,
       THEN otway.OR2, THEN otway.Reception,
       THEN otway.OR3, THEN otway.Reception, THEN otway.OR4])
apply (possibility, simp add: used_Cons)
done

lemma Gets_imp_Says [dest!]:
  "[| Gets B X ∈ set evs; evs ∈ otway |] ==> ∃A. Says A B X ∈ set evs"
apply (erule rev_mp)
apply (erule otway.induct, auto)
done

```

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)"

```

by blast

```
lemma OR4_analz_knows_Spy:
  "[| Gets B {N, X, Crypt (shrK B) X'|} ∈ set evs; evs ∈ otway |]
   ==> X ∈ analz (knows Spy evs)"
```

by blast

```
lemma Oops_parts_knows_Spy:
  "Says Server B {NA, X, Crypt K' {NB,K}|} ∈ set evs
   ==> K ∈ parts (knows Spy evs)"
```

by blast

Forwarding lemma: see comments in OtwayRees.thy

```
lemmas OR2_parts_knows_Spy =
  OR2_analz_knows_Spy [THEN analz_into_parts]
```

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)"
by (erule otway.induct, force,
    drule_tac [4] OR2_parts_knows_Spy, simp_all, blast+)
```

```
lemma Spy_analz_shrK [simp]:
  "evs ∈ otway ==> (Key (shrK A) ∈ analz (knows Spy evs)) = (A ∈ bad)"
by auto
```

```
lemma Spy_see_shrK_D [dest!]:
  "[|Key (shrK A) ∈ parts (knows Spy evs); evs ∈ otway|] ==> A ∈ bad"
by (blast dest: Spy_see_shrK)
```

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)"
apply (erule rev_mp)
apply (erule otway.induct, simp_all)
done
```

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))"
apply (erule otway.induct)
apply (frule_tac [8] Says_Server_message_form)
apply (drule_tac [7] OR4_analz_knows_Spy)
apply (drule_tac [5] OR2_analz_knows_Spy, analz_freshK, spy_analz, auto)
done

```

```

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))"
by (simp only: analz_image_freshK analz_image_freshK_simps)

```

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'"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule otway.induct, simp_all)
apply blast+ — OR3 and OR4
done

```

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)"
apply (erule otway.induct, force)
apply (frule_tac [7] Says_Server_message_form)
apply (drule_tac [6] OR4_analz_knows_Spy)
apply (drule_tac [4] OR2_analz_knows_Spy)
apply (simp_all add: analz_insert_eq analz_insert_freshK pushes)
apply spy_analz — Fake
apply (blast dest: unique_session_keys)+ — OR3, OR4, Oops
done

```

```

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)"
by (blast dest: Says_Server_message_form secrecy_lemma)

```

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"
by (erule otway.induct, force,
    drule_tac [4] OR2_parts_knows_Spy, simp_all, blast+)
```

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)"
apply (erule otway.induct, force,
       drule_tac [4] OR2_parts_knows_Spy, simp_all)
apply blast — Fake
apply blast — OR1: it cannot be a new Nonce, contradiction.
```

OR3 and OR4

```
apply (simp_all add: ex_disj_distrib)
prefer 2 apply (blast intro!: Crypt_imp_OR1) — OR4
```

OR3

```
apply clarify
```

oops

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"
apply (intro exI bexI)
apply (rule_tac [2] orb.Nil
      [THEN orb.OR1, THEN orb.Reception,
       THEN orb.OR2, THEN orb.Reception,
       THEN orb.OR3, THEN orb.Reception, THEN orb.OR4])
apply (possibility, simp add: used_Cons)
done

```

```

lemma Gets_imp_Says :
  "[[Gets B X ∈ set evs; evs ∈ orb]] ⇒ ∃ A. Says A B X ∈ set evs"
apply (erule rev_mp)
apply (erule orb.induct)
apply auto
done

```

```

lemma Gets_imp_knows_Spy:
  "[[Gets B X ∈ set evs; evs ∈ orb]] ⇒ X ∈ knows Spy evs"
by (blast dest!: Gets_imp_Says Says_imp_knows_Spy)

```

```

declare Gets_imp_knows_Spy [THEN parts.Inj, dest]

```

```

lemma Gets_imp_knows:
  "[[Gets B X ∈ set evs; evs ∈ orb]] ⇒ X ∈ knows B evs"
by (metis Gets_imp_knows_Spy Gets_imp_knows_agents)

```

```

lemma OR2_analz_knows_Spy:
  "[[Gets B {Nonce M, Agent A, Agent B, X} ∈ set evs; evs ∈ orb]]
  ⇒ X ∈ analz (knows Spy evs)"
by (blast dest!: Gets_imp_knows_Spy [THEN analz.Inj])

```

```

lemma OR4_parts_knows_Spy:
  "[[Gets B {Nonce M, Crypt (shrK B) {X, Nonce Nb, Key Kab}} ∈ set evs;
  evs ∈ orb]] ⇒ X ∈ parts (knows Spy evs)"

```

by blast

lemma *Oops_parts_knows_Spy*:

"Says Server B {Nonce M, Crypt K' {X, Nonce Nb, K}} ∈ set evs
 ⇒ K ∈ parts (knows Spy evs)"

by blast

lemmas *OR2_parts_knows_Spy* =

OR2_analz_knows_Spy [THEN *analz_into_parts*]

ML

{*

fun *parts_explicit_tac* i =
 forward_tac [@[thm *Oops_parts_knows_Spy*]] (i+7) THEN
 forward_tac [@[thm *OR4_parts_knows_Spy*]] (i+6) THEN
 forward_tac [@[thm *OR2_parts_knows_Spy*]] (i+4)

*}

method_setup *parts_explicit* = {*

Scan.succeed (K (SIMPLE_METHOD' *parts_explicit_tac*))*}

"to explicitly state that some message components belong to parts knows Spy"

lemma *Spy_see_shrK* [*simp*]:

"evs ∈ orb ⇒ (Key (shrK A) ∈ parts (knows Spy evs)) = (A ∈ bad)"

by (erule orb.induct, *parts_explicit*, *simp_all*, blast+)

lemma *Spy_analz_shrK* [*simp*]:

"evs ∈ orb ⇒ (Key (shrK A) ∈ analz (knows Spy evs)) = (A ∈ bad)"

by auto

lemma *Spy_see_shrK_D* [*dest!*]:

"[/Key (shrK A) ∈ parts (knows Spy evs); evs ∈ orb/] ⇒ A ∈ bad"

by (blast *dest*: *Spy_see_shrK*)

lemma *new_keys_not_used* [*simp*]:

"[/Key K ∉ used evs; K ∈ symKeys; evs ∈ orb] ⇒ K ∉ keysFor (parts (knows Spy evs))"

apply (erule *rev_mp*)

apply (erule orb.induct, *parts_explicit*, *simp_all*)

apply (force *dest!*: *keysFor_parts_insert*)

apply (blast+)

done

12.1 Proofs involving analz

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

lemma *Says_Server_message_form*:

"[/Says Server B {Nonce M, Crypt (shrK B) {X, Nonce Nb, Key K}} ∈ set evs;

evs ∈ orb]

⇒ K ∉ range shrK & (∃ A Na. X=(Crypt (shrK A) {Nonce Na, Key K}))"

by (erule rev_mp, erule orb.induct, simp_all)

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"

by (erule rev_mp, erule orb.induct, simp_all)

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"
apply (erule rev_mp, erule orb.induct, parts_explicit, simp_all)
apply (blast)
done

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)"
apply (erule rev_mp, erule orb.induct, parts_explicit, simp_all)
apply (blast+)
done

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"
apply (erule rev_mp, erule orb.induct, parts_explicit, simp_all)
apply (blast+)
done

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))"

by (metis B_trusts_OR3 Crypt_Spy_analz_bad Gets_imp_Says MPair_analz MPair_parts
Says_Server_message_form Says_imp_analz_Spy Says_imp_parts_knows_Spy)

```

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'"
by (erule rev_mp, erule rev_mp, erule orb.induct, simp_all, blast+)

```

```

lemma unique_Nb: "[[Says B Server {Nonce M, Agent A, Agent B, X, Crypt (shrK B)
B} {Nonce Nb, Nonce M, Nonce M, Agent A, Agent B}}] ∈ set evs;
  Says B Server {Nonce M', Agent A', Agent B, X', Crypt (shrK B) {Nonce
Nb, Nonce M', Nonce M', Agent A', Agent B}}] ∈ set evs;
  B ∉ bad; evs ∈ orb]] ⇒ M=M' & A=A' & X=X'"
by (erule rev_mp, erule rev_mp, erule orb.induct, simp_all, blast+)

```

```

lemma analz_image_freshCryptK_lemma:
"(Crypt K X ∈ analz (Key'nE ∪ H)) → (Crypt K X ∈ analz H) ⇒
  (Crypt K X ∈ analz (Key'nE ∪ H)) = (Crypt K X ∈ analz H)"
by (blast intro: analz_mono [THEN [2] rev_subsetD])

```

ML

```

{*
structure OtwayReesBella =
struct

val analz_image_freshK_ss =
  @{simpset} delsimps [image_insert, image_Un]
  delsimps @{{thm imp_disjL}} (*reduces blow-up*)
  addsimps @{thms analz_image_freshK_simps}

end
*}

method_setup analz_freshCryptK = {*
  Scan.succeed (fn ctxt =>
    (SIMPLE_METHOD
      (EVERY [REPEAT_FIRST (resolve_tac [allI, ballI, impI]),
        REPEAT_FIRST (rtac @{{thm analz_image_freshCryptK_lemma}},
          ALLGOALS (asm_simp_tac
            (Simplifier.context ctxt OtwayReesBella.analz_image_freshK_ss))))))
  *)
  "for proving useful rewrite rule"

method_setup disentangle = {*
  Scan.succeed
    (K (SIMPLE_METHOD
      (REPEAT_FIRST (eresolve_tac [asm_rl, conjE, disjE]
        ORELSE' hyp_subst_tac)))) *)
  "for eliminating conjunctions, disjunctions and the like"

```

```

lemma analz_image_freshCryptK [rule_format]:

```

```

"evs ∈ orb ⇒
  Key K ∉ analz (knows Spy evs) ⇒
    (∀ KK. KK ⊆ - (range shrK) ⇒
      (Crypt K X ∈ analz (Key'KK ∪ (knows Spy evs))) =
        (Crypt K X ∈ analz (knows Spy evs)))"
apply (erule orb.induct)
apply (analz_mono_contra)
apply (frule_tac [7] Gets_Server_message_form)
apply (frule_tac [9] Says_Server_message_form)
apply disentangle
apply (drule_tac [5] Gets_imp_knows_Spy [THEN analz.Inj, THEN analz.Snd, THEN
analz.Snd, THEN analz.Snd])
prefer 8 apply clarify
apply (analz_freshCryptK, spy_analz, fastforce)
done

```

```

lemma analz_insert_freshCryptK:
"[[evs ∈ orb; Key K ∉ analz (knows Spy evs);
  Seskey ∉ range shrK]] ⇒
  (Crypt K X ∈ analz (insert (Key Seskey) (knows Spy evs))) =
  (Crypt K X ∈ analz (knows Spy evs))"
by (simp only: analz_image_freshCryptK analz_image_freshK_simps)

```

```

lemma analz_hard:
"[[Says A B {Nonce M, Agent A, Agent B,
  Crypt (shrK A) {Nonce Na, Nonce M, Agent A, Agent B}}] ∈ set evs;

  Crypt (shrK A) {Nonce Na, Key K} ∈ analz (knows Spy evs);
  A ∉ bad; B ∉ bad; evs ∈ orb]]
⇒ Says B A {Nonce M, Crypt (shrK A) {Nonce Na, Key K}} ∈ set evs"
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule orb.induct)
apply (frule_tac [7] Gets_Server_message_form)
apply (frule_tac [9] Says_Server_message_form)
apply disentangle

```

letting the simplifier solve OR2

```

apply (drule_tac [5] Gets_imp_knows_Spy [THEN analz.Inj, THEN analz.Snd, THEN
analz.Snd, THEN analz.Snd])
apply (simp_all (no_asm_simp) add: analz_insert_eq pushes_split_ifs)
apply (spy_analz)

```

OR1

apply blast

Oops

prefer 4 **apply** (blast dest: analz_insert_freshCryptK)

OR4 - ii

prefer 3 **apply** (blast)

OR3

```
apply (blast dest:
  A_trusts_OR1 unique_Na Key_not_used analz_insert_freshCryptK)
```

OR4 - i

```
apply clarify
apply (simp add: pushes_split_ifs)
apply (case_tac "Aaa∈bad")
apply (blast dest: analz_insert_freshCryptK)
apply clarify
apply simp
apply (case_tac "Ba∈bad")
apply (frule Gets_imp_knows_Spy [THEN analz.Inj, THEN analz.Snd, THEN analz.Decrypt,
  THEN analz.Fst] , assumption)
apply (simp (no_asm_simp))
apply clarify
apply (frule Gets_imp_knows_Spy
  [THEN parts.Inj, THEN parts.Snd, THEN B_trusts_OR3],
  assumption, assumption, erule exE)
apply (frule Says_Server_imp_Gets
  [THEN Gets_imp_knows_Spy, THEN parts.Inj, THEN parts.Snd,
  THEN parts.Snd, THEN parts.Snd, THEN parts.Fst, THEN A_trusts_OR1],
  assumption, assumption, assumption, assumption)
apply (blast dest: Says_Server_imp_Gets B_trusts_OR2 unique_Na unique_Nb)
done
```

lemma Gets_Server_message_form':

```
"[[Gets B {Nonce M, Crypt (shrK B) {X, Nonce Nb, Key K}}] ∈ set evs;
  B ∉ bad; evs ∈ orb]
  ⇒ K ∉ range shrK & (∃ A Na. X = (Crypt (shrK A) {Nonce Na, Key K}))"
by (blast dest!: B_trusts_OR3 Says_Server_message_form)
```

lemma OR4_imp_Gets:

```
"[[Says B A {Nonce M, Crypt (shrK A) {Nonce Na, Key K}}] ∈ set evs;
  B ∉ bad; evs ∈ orb]
  ⇒ (∃ Nb. Gets B {Nonce M, Crypt (shrK B) {Crypt (shrK A) {Nonce Na, Key
  K},
  Nonce Nb, Key K}}] ∈ set evs)"
apply (erule rev_mp, erule orb.induct, parts_explicit, simp_all)
prefer 3 apply (blast dest: Gets_Server_message_form')
apply blast+
done
```

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)"
apply (drule Gets_imp_knows_Spy [THEN analz.Inj, THEN analz.Snd], assumption)
```

```

apply (drule analz_hard, assumption, assumption, assumption, assumption)
apply (drule OR4_imp_Gets, assumption, assumption)
apply (fastforce dest!: Gets_imp_knows [THEN analz.Inj] analz.Decrypt)
done

```

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"
apply (intro exI bexI)
apply (rule_tac [2] woolam.Nil
      [THEN woolam.WL1, THEN woolam.WL2, THEN woolam.WL3,
      THEN woolam.WL4, THEN woolam.WL5], possibility)
done

```

```

lemma Spy_see_shrK [simp]:
  "evs ∈ woolam ==> (Key (shrK A) ∈ parts (spies evs)) = (A ∈ bad)"
by (erule woolam.induct, force, simp_all, blast+)

```

```

lemma Spy_analz_shrK [simp]:
  "evs ∈ woolam ==> (Key (shrK A) ∈ analz (spies evs)) = (A ∈ bad)"
by auto

```

```

lemma Spy_see_shrK_D [dest!]:
  "[|Key (shrK A) ∈ parts (knows Spy evs); evs ∈ woolam|] ==> A ∈ bad"
by (blast dest: Spy_see_shrK)

```

```

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"
by (erule rev_mp, erule woolam.induct, force, simp_all, blast+)

```

```

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"
  by (blast intro!: NB_Crypt_imp_Alice_msg)

```

```

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"
  by (erule rev_mp, erule woolam.induct, force, simp_all, blast+)

```

```

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"
  by (erule rev_mp, erule woolam.induct, force, simp_all, blast+)

```

```

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"
  by (blast dest!: NB_Crypt_imp_Server_msg)

```

```

lemma B_said_WL2:
  "[| Says B A (Nonce NB) ∈ set evs; B ≠ Spy; evs ∈ woolam |]
  ==> ∃A'. Says A' B (Agent A) ∈ set evs"
  by (erule rev_mp, erule woolam.induct, force, simp_all, blast+)

```

```

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"
apply (erule rev_mp, erule woolam.induct, force, simp_all, blast, auto)
oops

```

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  $\in$  recur; Nonce NA  $\notin$  used evs1 |]
          ==> Says A B (Hash[Key(shrK A)] {|Agent A, Agent B, Nonce NA, END|})
            # evs1  $\in$  recur"
```

20014 THE OTWAY-BULL RECURSIVE AUTHENTICATION PROTOCOL

```
| 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"
apply (intro exI bexI)
apply (rule_tac [2] recur.Nil [THEN recur.RA1,
                              THEN recur.RA3 [OF _ _ respond.One]])
apply (possibility, simp add: used_Cons)
done
```

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"
apply (intro exI bexI)
apply (rule_tac [2]
      recur.Nil
      [THEN recur.RA1 [of _ NA],
       THEN recur.RA2 [of _ NB],
       THEN recur.RA3 [OF _ _ respond.One
                      [THEN respond.Cons [of _ _ K _ K']]],
       THEN recur.RA4], possibility)
apply (auto simp add: used_Cons)
```

done

```

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"

apply (intro exI bexI)
apply (rule_tac [2]
        recur.Nil [THEN recur.RA1,
                  THEN recur.RA2, THEN recur.RA2,
                  THEN recur.RA3
                  [OF _ _ respond.One
                   [THEN respond.Cons, THEN respond.Cons]],
                  THEN recur.RA4, THEN recur.RA4])
apply basic_possibility
apply (tactic "DEPTH_SOLVE (swap_res_tac [refl, conjI, disjCI] 1)")
done

```

```

lemma respond_imp_not_used: "(PA,RB,KAB) ∈ respond evs ==> Key KAB ∉ used
evs"
by (erule respond.induct, simp_all)

```

```

lemma Key_in_parts_respond [rule_format]:
  "[| Key K ∈ parts {RB}; (PB,RB,K') ∈ respond evs |] ==> Key K ∉ used
evs"
apply (erule rev_mp, erule respond.induct)
apply (auto dest: Key_not_used respond_imp_not_used)
done

```

Simple inductive reasoning about responses

```

lemma respond_imp_responses:
  "(PA,RB,KAB) ∈ respond evs ==> RB ∈ responses evs"
apply (erule respond.induct)
apply (blast intro!: respond_imp_not_used responses.intros)+
done

```

```

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)"
by blast

```

```

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)"
apply (erule recur.induct, auto)
```

RA3. It's ugly to call auto twice, but it seems necessary.

```
apply (auto dest: Key_in_parts_respond simp add: parts_insert_spies)
done
```

```
lemma Spy_analz_shrK [simp]:
  "evs ∈ recur ==> (Key (shrK A) ∈ analz (spies evs)) = (A ∈ bad)"
by auto
```

```
lemma Spy_see_shrK_D [dest!]:
  "[|Key (shrK A) ∈ parts (knows Spy evs); evs ∈ recur|] ==> A ∈ bad"
by (blast dest: Spy_see_shrK)
```

```
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))"
apply (erule responses.induct)
apply (simp_all del: image_insert
  add: analz_image_freshK_simps, auto)
done
```

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))"
apply (erule recur.induct)
apply (drule_tac [4] RA2_analz_spies,
  drule_tac [5] respond_imp_responses,
  drule_tac [6] RA4_analz_spies, analz_freshK, spy_analz)
```

RA3

```

apply (simp_all add: resp_analz_image_freshK_lemma)
done

```

```

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))"
by (simp del: image_insert
      add: analz_image_freshK_simps raw_analz_image_freshK)

```

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)"
apply (erule rev_mp)
apply (erule recur.induct,
  drule_tac [6] RA4_parts_spies,
  drule_tac [5] respond_imp_responses,
  drule_tac [4] RA2_parts_spies)

```

RA3 requires a further induction

```

apply (erule_tac [5] responses.induct, simp_all)

```

Fake

```

apply (blast intro: parts_insertI)
done

```

```

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'"
apply (erule rev_mp, erule rev_mp)
apply (erule recur.induct,
  drule_tac [5] respond_imp_responses)
apply (force, simp_all)

```

Fake

```

apply blast
apply (erule_tac [3] responses.induct)

```

RA1,2: creation of new Nonce

```

apply simp_all
apply (blast dest!: Hash_imp_body)+
done

```

```

lemma shrK_in_analz_respond [simp]:
  "[| RB ∈ responses evs; evs ∈ recur |]
  ==> (Key (shrK B) ∈ analz (insert RB (spies evs))) = (B:bad)"
apply (erule responses.induct)
apply (simp_all del: image_insert
          add: analz_image_freshK_simps resp_analz_image_freshK, auto)

```

done

```

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)"
apply (erule rev_mp, erule responses.induct)
apply (simp_all del: image_insert
          add: analz_image_freshK_simps resp_analz_image_freshK_lemma)

```

Simplification using two distinct treatments of "image"

```

apply (simp add: parts_insert2, blast)
done

```

```

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}"
apply (ind_cases "(Hash[Key (shrK A)] {|Agent A, B, NA, P|}, RA, K) ∈ respond
  evs")
apply simp_all
done

```

```

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))"
apply (erule respond.induct)
apply (simp_all add: all_conj_distrib)
apply (blast dest: respond_certificate)
done

```

```

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)"
  by (rule unique_lemma, auto)

```

```

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))"
apply (erule respond.induct)
apply (frule_tac [2] respond_imp_responses)
apply (frule_tac [2] respond_imp_not_used)
apply (simp_all del: image_insert
  add: analz_image_freshK_simps split_ifs shrK_in_analz_respond
  resp_analz_image_freshK parts_insert2)

```

Base case of respond

```
apply blast
```

Inductive step of respond

```
apply (intro allI conjI impI, simp_all)
```

by unicity, either $B = Aa$ or $B = A'$, a contradiction if $B \in \text{bad}$

```

apply (blast dest: unique_session_keys respond_certificate)
apply (blast dest!: respond_certificate)
apply (blast dest!: resp_analz_insert)
done

```

```

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)"
apply (erule rev_mp)
apply (erule recur.induct)
apply (drule_tac [4] RA2_analz_spies,
  frule_tac [5] respond_imp_responses,
  drule_tac [6] RA4_analz_spies,
  simp_all add: split_ifs analz_insert_eq analz_insert_freshK)

```

Fake

```
apply spy_analz
```

RA2

```
apply blast
```

RA3

```

apply (simp add: parts_insert_spies)
apply (metis Key_in_parts_respond parts.Body parts.Fst resp_analz_insert)

```

respond_Spy_not_see_session_key usedI)

RA4

apply *blast*
done

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"
apply (erule *rev_mp*)
apply (erule *respond_imp_responses [THEN responses.induct]*, auto)
done

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"
apply (unfold *HPair_def*)
apply (erule *rev_mp*)
apply (erule *recur.induct*,
 drule_tac [6] *RA4_parts_spies*,
 drule_tac [4] *RA2_parts_spies*,
 simp_all)

Fake, RA3

apply (*blast dest: Hash_in_parts_respond*)+
done

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}"
apply (erule *rev_mp*, erule *recur.induct*, simp_all)

Fake

apply *blast*

RA1

apply *blast*

RA2: it cannot be a new Nonce, contradiction.

apply *blast*

RA3. Pity that the proof is so brittle: this step requires the rewriting, which however would break all other steps.

```

apply (simp add: parts_insert_spies, blast)

RA4

apply blast
done

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: " $[] \in \text{yahalom}$ "

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

Reception: " $[| \text{evsr} \in \text{yahalom}; \text{Says } A \ B \ X \in \text{set } \text{evsr} |]$
 $\implies \text{Gets } B \ X \ \# \ \text{evsr} \in \text{yahalom}$ "

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

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

YM3: " $[| \text{evs3} \in \text{yahalom}; \text{Key } KAB \notin \text{used } \text{evs3}; \text{KAB} \in \text{symKeys};$
 $\text{Gets } \text{Server}$
 $\{|Agent\ B, \text{Crypt} (\text{shrK } B) \ \{|Agent\ A, \text{Nonce } NA, \text{Nonce } NB|\}|\}$
 $\in \text{set } \text{evs3} |]$
 $\implies \text{Says } \text{Server } A$
 $\{|Crypt (\text{shrK } A) \ \{|Agent\ B, \text{Key } KAB, \text{Nonce } NA, \text{Nonce } NB|\},$
 $\text{Crypt} (\text{shrK } B) \ \{|Agent\ A, \text{Key } KAB|\}|\}$
 $\# \ \text{evs3} \in \text{yahalom}$ "

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"

definition KeyWithNonce :: "[key, nat, event list] => bool" where
  "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"
apply (intro exI bexI)
apply (rule_tac [2] yahalom.Nil
  [THEN yahalom.YM1, THEN yahalom.Reception,
  THEN yahalom.YM2, THEN yahalom.Reception,
  THEN yahalom.YM3, THEN yahalom.Reception,
  THEN yahalom.YM4])
apply (possibility, simp add: used_Cons)
done

```

15.1 Regularity Lemmas for Yahalom

```

lemma Gets_imp_Says:
  "[| Gets B X ∈ set evs; evs ∈ yahalom |] ==> ∃ A. Says A B X ∈ set evs"
by (erule rev_mp, erule yahalom.induct, auto)

```

Must be proved separately for each protocol

```

lemma Gets_imp_knows_Spy:
  "[| Gets B X ∈ set evs; evs ∈ yahalom |] ==> X ∈ knows Spy evs"
by (blast dest!: Gets_imp_Says Says_imp_knows_Spy)

```

```

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)"
by blast
```

```
lemmas YM4_parts_knows_Spy =
  YM4_analz_knows_Spy [THEN analz_into_parts]
```

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)"
by (metis parts.Body parts.Fst parts.Snd Says_imp_knows_Spy parts.Inj)
```

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)"
by (erule yahalom.induct, force,
    drule_tac [6] YM4_parts_knows_Spy, simp_all, blast+)
```

```
lemma Spy_analz_shrK [simp]:
  "evs ∈ yahalom ==> (Key (shrK A) ∈ analz (knows Spy evs)) = (A ∈ bad)"
by auto
```

```
lemma Spy_see_shrK_D [dest!]:
  "[|Key (shrK A) ∈ parts (knows Spy evs); evs ∈ yahalom|] ==> A ∈ bad"
by (blast dest: Spy_see_shrK)
```

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))"
apply (erule rev_mp)
apply (erule yahalom.induct, force,
    frule_tac [6] YM4_parts_knows_Spy, simp_all)
```

Fake

```
apply (force dest!: keysFor_parts_insert, auto)
done
```

Earlier, all protocol proofs declared this theorem. But only a few proofs need it, e.g. Yahalom and Kerberos IV.

```
lemma new_keys_not_analzD:
  "[|K ∈ symKeys; evs ∈ yahalom; Key K ∉ used evs|]
   ==> K ∉ keysFor (analz (knows Spy evs))"
by (blast dest: new_keys_not_used intro: keysFor_mono [THEN subsetD])
```

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"
by (erule rev_mp, erule yahalom.induct, simp_all)
```

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))"
apply (erule yahalom.induct,
  drule_tac [7] YM4_analz_knows_Spy, analz_freshK, spy_analz, blast)
apply (simp only: Says_Server_not_range analz_image_freshK_simps)
apply safe
done
```

```
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))"
by (simp only: analz_image_freshK analz_image_freshK_simps)
```

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'"
apply (erule rev_mp, erule rev_mp)
apply (erule yahalom.induct, simp_all)
```

YM3, by freshness, and YM4

```
apply blast+
done
```

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)"
```

```

apply (erule yahalom.induct, force,
        drule_tac [6] YM4_analz_knows_Spy)
apply (simp_all add: pushes_analz_insert_eq analz_insert_freshK, spy_analz)
— Fake
apply (blast dest: unique_session_keys)+ — YM3, Oops
done

```

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)"
by (blast dest: secrecy_lemma)

```

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"
apply (erule rev_mp)
apply (erule yahalom.induct, force,
        frule_tac [6] YM4_parts_knows_Spy, simp_all)

```

Fake, YM3

```

apply blast+
done

```

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)"
by (metis A_trusts_YM3 secrecy_lemma)

```

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"
apply (erule rev_mp)
apply (erule yahalom.induct, force,
        frule_tac [6] YM4_parts_knows_Spy, simp_all)

```

Fake, YM3

```

apply blast+
done

```

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"
apply (erule rev_mp, erule rev_mp)
apply (erule yahalom.induct, force,
        frule_tac [6] YM4_parts_knows_Spy)
apply (analz_mono_contra, simp_all)

```

Fake, YM3

```

apply blast
apply blast

```

YM4. A is uncompromised because NB is secure A's certificate guarantees the existence of the Server message

```

apply (blast dest!: Gets_imp_Says Crypt_Spy_analz_bad
             dest: Says_imp_spies
             parts.Inj [THEN parts.Fst, THEN A_trusts_YM3])
done

```

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"
by (unfold KeyWithNonce_def, blast)

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)"
  by (simp add: KeyWithNonce_def, blast)

```

```

lemma KeyWithNonce_Notes [simp]:
  "KeyWithNonce K NB (Notes A X # evs) = KeyWithNonce K NB evs"
  by (simp add: KeyWithNonce_def)

```

```

lemma KeyWithNonce_Gets [simp]:
  "KeyWithNonce K NB (Gets A X # evs) = KeyWithNonce K NB evs"
  by (simp add: KeyWithNonce_def)

```

A fresh key cannot be associated with any nonce (with respect to a given trace).

```

lemma fresh_not_KeyWithNonce:
  "Key K  $\notin$  used evs ==>  $\sim$  KeyWithNonce K NB evs"
  by (unfold KeyWithNonce_def, blast)

```

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|}
     $\in$  set evs;
    NB  $\neq$  NB'; evs  $\in$  yahalom |]
    ==>  $\sim$  KeyWithNonce K NB evs"
  by (unfold KeyWithNonce_def, blast dest: unique_session_keys)

```

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  $\in$  analz (G Un H)) --> (X  $\in$  analz H) ==>
  P --> (X  $\in$  analz (G Un H)) = (X  $\in$  analz H)"
  by (blast intro: analz_mono [THEN subsetD])

```

```

lemma Nonce_secrecy:
  "evs  $\in$  yahalom ==>
  ( $\forall$  KK. KK  $\leq$  - (range shrK) -->
    ( $\forall$  K  $\in$  KK. K  $\in$  symKeys -->  $\sim$  KeyWithNonce K NB evs) -->
    (Nonce NB  $\in$  analz (Key'KK Un (knows Spy evs))) =
    (Nonce NB  $\in$  analz (knows Spy evs)))"
  apply (erule yahalom.induct,
    frule_tac [7] YM4_analz_knows_Spy)
  apply (safe del: allI impI intro!: Nonce_secrecy_lemma [THEN impI, THEN allI])
  apply (simp_all del: image_insert image_Un
    add: analz_image_freshK_simps split_ifs
    all_conj_distrib ball_conj_distrib
    analz_image_freshK fresh_not_KeyWithNonce
    imp_disj_not1
    Says_Server_KeyWithNonce)

```

For Oops, simplification proves $NBa \neq NB$. By *Says_Server_KeyWithNonce*, we get \neg *KeyWithNonce* $K NB$ *evs*; then simplification can apply the induction hypothesis with $KK = \{K\}$.

Fake

apply *spy_analz*

YM2

apply *blast*

YM3

apply *blast*

YM4

apply (*erule_tac* $V = "\forall KK. ?P KK"$ in *thin_rl*, *clarify*)

If $A \in \text{bad}$ then NBa is known, therefore $NBa \neq NB$. Previous two steps make the next step faster.

apply (*metis* *A_trusts_YM3* *Gets_imp_analz_Spy* *Gets_imp_knows_Spy* *KeyWithNonce_def* *Spy_analz_shrK* *analz.Fst* *analz.Snd* *analz_shrK_Decrypt* *parts.Fst* *parts.Inj*)
done

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))"
```

```
by (simp_all del: image_insert image_Un imp_disjL
    add: analz_image_freshK_simps split_ifs
        Nonce_secretcy Says_Server_KeyWithNonce)
```

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"
```

apply (*erule* *rev_mp*, *erule* *rev_mp*)

apply (*erule* *yahalom.induct*, *force*,
frule_tac [6] *YM4_parts_knows_Spy*, *simp_all*)

Fake, and YM2 by freshness

apply *blast+*

done

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"
by (blast dest!: Gets_imp_Says Crypt_Spy_analz_bad
      dest: Says_imp_spies unique_NB parts.Inj analz.Inj)

```

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)"
apply (erule rev_mp, erule rev_mp)
apply (erule yahalom.induct, force,
        frule_tac [6] YM4_parts_knows_Spy)
apply (analz_mono_contra, simp_all)

```

Fake, YM2

```

apply blast+
done

```

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"
by (erule rev_mp, erule yahalom.induct, auto)

```

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)"
apply (erule rev_mp, erule rev_mp)
apply (erule yahalom.induct, force,
        frule_tac [6] YM4_analz_knows_Spy)
apply (simp_all add: split_ifs pushes_new_keys_not_analz insert_eq
        analz_insert_freshK)

```

Fake

```

apply spy_analz

```

YM1: NB=NA is impossible anyway, but NA is secret because it is fresh!

```

apply blast

```

YM2

apply *blast*

Prove YM3 by showing that no NB can also be an NA

apply (*blast dest!: no_nonce_YM1_YM2 dest: Gets_imp_Says Says_unique_NB*)

LEVEL 7: YM4 and Oops remain

apply (*clarify, simp add: all_conj_distrib*)

YM4: key K is visible to Spy, contradicting session key secrecy theorem

Case analysis on Aa:bad; PROOF FAILED problems use *Says_unique_NB* to identify message components: Aa = A, Ba = B

apply (*blast dest!: Says_unique_NB analz_shrK_Decrypt
parts.Inj [THEN parts.Fst, THEN A_trusts_YM3]
dest: Gets_imp_Says Says_imp_spies Says_Server_imp_YM2
Spy_not_see_encrypted_key*)

Oops case: if the nonce is betrayed now, show that the Oops event is covered by the quantified Oops assumption.

apply (*clarify, simp add: all_conj_distrib*)
apply (*frule Says_Server_imp_YM2, assumption*)
apply (*metis Gets_imp_Says Says_Server_not_range Says_unique_NB no_nonce_YM1_YM2
parts.Snd single_Nonce_secrecy spies_partsEs(1)*)
done

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"
```

by (*blast dest: Spy_not_see_NB Says_unique_NB
Says_Server_imp_YM2 B_trusts_YM4_newK*)

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)"
  by (metis B_trusts_YM4 Spy_not_see_encrypted_key)

```

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"
apply (erule rev_mp, erule yahalom.induct, force,
  frule_tac [6] YM4_parts_knows_Spy, simp_all)

```

Fake

```

apply blast
done

```

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"
apply (erule rev_mp, erule yahalom.induct, simp_all)

```

YM3, YM4

```

apply (blast dest!: B_Said_YM2)+
done

```

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"
  by (metis A_trusts_YM3 Gets_imp_analz_Spy YM3_auth_B_to_A_lemma analz.Fst
    not_parts_not_analz)

```

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)"
apply (erule yahalom.induct, force,
  frule_tac [6] YM4_parts_knows_Spy)
apply (analz_mono_contra, simp_all)

Fake

apply blast

YM3: by new_keys_not_used, the message Crypt K (Nonce NB) could not exist
apply (force dest!: Crypt_imp_keysFor)

YM4: was Crypt K (Nonce NB) the very last message? If not, use the induction hypothesis
apply (simp add: ex_disj_distrib)

yes: apply unicity of session keys

apply (blast dest!: Gets_imp_Says A_trusts_YM3 B_trusts_YM4_shrK
  Crypt_Spy_analz_bad
  dest: Says_imp_knows_Spy [THEN parts.Inj] unique_session_keys)
done

```

If B receives YM4 then A has used nonce NB (and therefore is alive). Moreover, A associates K with NB (thus is talking about the same run). Other premises guarantee secrecy of K.

```

lemma YM4_imp_A_Said_YM3 [rule_format]:
  "[| Gets B {|Crypt (shrK B) {|Agent A, Key K|},
      Crypt K (Nonce NB)|} ∈ set evs;
   Says B Server
      {|Agent B, Crypt (shrK B) {|Agent A, Nonce NA, Nonce NB|}|}
   ∈ set evs;
   (∀NA k. Notes Spy {|Nonce NA, Nonce NB, k|} ∉ set evs);
   A ∉ bad; B ∉ bad; evs ∈ yahalom |]
  ==> ∃X. Says A B {|X, Crypt K (Nonce NB)|} ∈ set evs"
by (metis A_Said_YM3_lemma B_gets_good_key Gets_imp_analz_Spy YM4_parts_knows_Spy
  analz.Fst not_parts_not_analz)
end

```

16 The Yahalom Protocol, Variant 2

```

theory Yahalom2 imports Public begin

```

This version trades encryption of NB for additional explicitness in YM3. Also in YM3, care is taken to make the two certificates distinct.

From page 259 of Burrows, Abadi and Needham (1989). A Logic of Authentication. Proc. Royal Soc. 426

This theory has the prototypical example of a secrecy relation, KeyCryptNonce.

```

inductive_set yahalom :: "event list set"
  where

    Nil: "[ ] ∈ yahalom"

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

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

  | YM1: "[| evs1 ∈ yahalom; Nonce NA ∉ used evs1 |]
    ==> Says A B {|Agent A, Nonce NA|} # evs1 ∈ yahalom"

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

  | YM3: "[| evs3 ∈ yahalom; Key KAB ∉ used evs3;
    Gets Server {|Agent B, Nonce NB,
    Crypt (shrK B) {|Agent A, Nonce NA|}|}
    ∈ set evs3 |]
    ==> Says Server A
    {|Nonce NB,
    Crypt (shrK A) {|Agent B, Key KAB, Nonce NA|},
    Crypt (shrK B) {|Agent A, Agent B, Key KAB, Nonce NB|}|}
    # evs3 ∈ yahalom"

  | YM4: "[| evs4 ∈ yahalom;
    Gets A {|Nonce NB, Crypt (shrK A) {|Agent B, Key K, Nonce NA|},
    X|} ∈ set evs4;
    Says A B {|Agent A, Nonce NA|} ∈ set evs4 |]
    ==> Says A B {|X, Crypt K (Nonce NB)|} # evs4 ∈ yahalom"

  | 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  $\notin$  used []
  ==>  $\exists X NB. \exists evs \in yahalom.
    Says A B \{|X, Crypt K (Nonce NB)|\} \in set evs"$ 
apply (intro exI bexI)
apply (rule_tac [2] yahalom.Nil
  [THEN yahalom.YM1, THEN yahalom.Reception,
  THEN yahalom.YM2, THEN yahalom.Reception,
  THEN yahalom.YM3, THEN yahalom.Reception,
  THEN yahalom.YM4])
apply (possibility, simp add: used_Cons)
done

```

```

lemma Gets_imp_Says:
  "[| Gets B X  $\in set evs$ ;  $evs \in yahalom$  |] ==>  $\exists A. Says A B X \in set evs"$ 
by (erule rev_mp, erule yahalom.induct, auto)

```

Must be proved separately for each protocol

```

lemma Gets_imp_knows_Spy:
  "[| Gets B X  $\in set evs$ ;  $evs \in yahalom$  |] ==>  $X \in knows Spy evs"$ 
by (blast dest!: Gets_imp_Says Says_imp_knows_Spy)

```

```

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|\}  $\in set evs$ ;  $evs \in yahalom$  |]
  ==>  $X \in analz (knows Spy evs)"$ 
by blast

```

```

lemmas YM4_parts_knows_Spy =
  YM4_analz_knows_Spy [THEN analz_into_parts]

```

Spy never sees a good agent's shared key!

```

lemma Spy_see_shrK [simp]:
  " $evs \in yahalom ==> (Key (shrK A) \in parts (knows Spy evs)) = (A \in bad)"$ 
by (erule yahalom.induct, force,
  drule_tac [6] YM4_parts_knows_Spy, simp_all, blast+)

```

```

lemma Spy_analz_shrK [simp]:
  " $evs \in yahalom ==> (Key (shrK A) \in analz (knows Spy evs)) = (A \in bad)"$ 
by auto

```

```

lemma Spy_see_shrK_D [dest!]:
  "[| Key (shrK A)  $\in parts (knows Spy evs)$ ;  $evs \in yahalom$  |] ==>  $A \in bad"$ 
by (blast dest: Spy_see_shrK)

```

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))"
apply (erule rev_mp)
apply (erule yahalom.induct, force,
       frule_tac [6] YM4_parts_knows_Spy, simp_all)

Fake

apply (force dest!: keysFor_parts_insert)

YM3

apply blast

YM4

apply auto
apply (blast dest!: Gets_imp_knows_Spy [THEN parts.Inj])
done

```

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"
by (erule rev_mp, erule yahalom.induct, simp_all)

```

```

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))"
apply (erule yahalom.induct)
apply (frule_tac [8] Says_Server_message_form)
apply (drule_tac [7] YM4_analz_knows_Spy, analz_freshK, spy_analz, blast)
done

```

```

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))"
by (simp only: analz_image_freshK analz_image_freshK_simps)

```

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'"
  apply (erule rev_mp, erule rev_mp)
  apply (erule yahalom.induct, simp_all)

```

YM3, by freshness

```

  apply blast
done

```

16.2 Crucial Secrecy Property: Spy Does Not See Key K_{AB}

```

lemma secrecy_lemma:
  "[| A ∉ bad; B ∉ bad; evs ∈ yahalom |]
  ==> Says Server A
    {|nb, Crypt (shrK A) {|Agent B, Key K, na|},
     Crypt (shrK B) {|Agent A, Agent B, Key K, nb|}|}
  ∈ set evs -->
  Notes Spy {|na, nb, Key K|} ∉ set evs -->
  Key K ∉ analz (knows Spy evs)"
  apply (erule yahalom.induct, force, frule_tac [7] Says_Server_message_form,
    drule_tac [6] YM4_analz_knows_Spy)
  apply (simp_all add: pushes_analz_insert_eq analz_insert_freshK, spy_analz)
  apply (blast dest: unique_session_keys)+
done

```

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)"
  by (blast dest: secrecy_lemma Says_Server_message_form)

```

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"
  by (blast dest: Spy_not_see_encrypted_key)

```

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"
```

apply (*erule rev_mp*)

apply (*erule yahalom.induct, force,*
frule_tac [6] YM4_parts_knows_Spy, simp_all)

Fake, YM3

apply *blast+*
done

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)"
```

by (*blast dest!: A_trusts_YM3 Spy_not_see_encrypted_key*)

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"
```

apply (*erule rev_mp*)

apply (*erule yahalom.induct, force,*
frule_tac [6] YM4_parts_knows_Spy, simp_all)

Fake, YM3

apply *blast+*
done

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"
by (blast dest!: B_trusts_YM4_shrK)

```

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)"
by (blast dest!: B_trusts_YM4 Spy_not_see_encrypted_key)

```

16.5 Authenticating B to A

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

```

lemma B_Said_YM2:
  "[| Crypt (shrK B) {|Agent A, Nonce NA|} ∈ parts (knows Spy evs);
    B ∉ bad; evs ∈ yahalom |]
  ==> ∃NB. Says B Server {|Agent B, Nonce NB,
    Crypt (shrK B) {|Agent A, Nonce NA|}|}
    ∈ set evs"
apply (erule rev_mp)
apply (erule yahalom.induct, force,
  frule_tac [6] YM4_parts_knows_Spy, simp_all)

```

Fake, YM2

```

apply blast+
done

```

If the server sends YM3 then B sent YM2, perhaps with a different NB

```

lemma YM3_auth_B_to_A_lemma:
  "[| Says Server A {|nb, Crypt (shrK A) {|Agent B, Key K, Nonce NA|}, X|}
    ∈ set evs;
    B ∉ bad; evs ∈ yahalom |]
  ==> ∃nb'. Says B Server {|Agent B, nb',
    Crypt (shrK B) {|Agent A, Nonce NA|}|}
    ∈ set evs"
apply (erule rev_mp)
apply (erule yahalom.induct, simp_all)

```

Fake, YM2, YM3

```

apply (blast dest!: B_Said_YM2)+
done

```

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

```

theorem YM3_auth_B_to_A:
  "[| Gets A {nb, Crypt (shrK A) {|Agent B, Key K, Nonce NA|}, X|}
    ∈ set evs;
    A ∉ bad; B ∉ bad; evs ∈ yahalom |]
  ==> ∃nb'. Says B Server
    {|Agent B, nb', Crypt (shrK B) {|Agent A, Nonce NA|}|}
    ∈ set evs"
by (blast dest!: A_trusts_YM3 YM3_auth_B_to_A_lemma)

```

16.6 Authenticating A to B

using the certificate *Crypt K (Nonce NB)*

Assuming the session key is secure, if both certificates are present then A has said NB. We can't be sure about the rest of A's message, but only NB matters for freshness. Note that *Key K ∉ analz (knows Spy evs)* must be the FIRST antecedent of the induction formula.

This lemma allows a use of *unique_session_keys* in the next proof, which otherwise is extremely slow.

```

lemma secure_unique_session_keys:
  "[| Crypt (shrK A) {|Agent B, Key K, na|} ∈ analz (spies evs);
    Crypt (shrK A') {|Agent B', Key K, na'|} ∈ analz (spies evs);
    Key K ∉ analz (knows Spy evs); evs ∈ yahalom |]
  ==> A=A' & B=B'"
by (blast dest!: A_trusts_YM3 dest: unique_session_keys Crypt_Spy_analz_bad)

```

```

lemma Auth_A_to_B_lemma [rule_format]:
  "evs ∈ yahalom
  ==> Key K ∉ analz (knows Spy evs) -->
    K ∈ symKeys -->
    Crypt K (Nonce NB) ∈ parts (knows Spy evs) -->
    Crypt (shrK B) {|Agent A, Agent B, Key K, Nonce NB|}
      ∈ parts (knows Spy evs) -->
    B ∉ bad -->
    (∃X. Says A B {X, Crypt K (Nonce NB)|} ∈ set evs)"
apply (erule yahalom.induct, force,
  frule_tac [6] YM4_parts_knows_Spy)
apply (analz_mono_contra, simp_all)

```

Fake

apply blast

YM3: by *new_keys_not_used*, the message *Crypt K (Nonce NB)* could not exist

apply (force dest!: *Crypt_imp_keysFor*)

YM4: was *Crypt K (Nonce NB)* the very last message? If so, apply unicity of session keys; if not, use the induction hypothesis

apply (blast dest!: *B_trusts_YM4_shrK* dest: *secure_unique_session_keys*)
done

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"
by (blast intro: Auth_A_to_B_lemma
      dest: Spy_not_see_encrypted_key B_trusts_YM4_shrK)
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"
apply (intro exI bexI)
apply (rule_tac [2] yahalom.Nil
      [THEN yahalom.YM1, THEN yahalom.Reception,
       THEN yahalom.YM2, THEN yahalom.Reception,
       THEN yahalom.YM3, THEN yahalom.Reception,
       THEN yahalom.YM4])
apply (possibility, simp add: used_Cons)
done

```

17.1 Regularity Lemmas for Yahalom

```

lemma Gets_imp_Says:
  "[| Gets B X ∈ set evs; evs ∈ yahalom |] ==> ∃A. Says A B X ∈ set evs"
by (erule rev_mp, erule yahalom.induct, auto)

```

```

lemma Gets_imp_knows_Spy:
  "[| Gets B X ∈ set evs; evs ∈ yahalom |] ==> X ∈ knows Spy evs"
by (blast dest!: Gets_imp_Says Says_imp_knows_Spy)

```

```

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)"
by blast

```

```
lemmas YM4_parts_knows_Spy =
  YM4_analz_knows_Spy [THEN analz_into_parts]
```

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)"
apply (erule yahalom.induct, force,
  drule_tac [6] YM4_parts_knows_Spy, simp_all, blast+)
done
```

```
lemma Spy_analz_shrK [simp]:
  "evs ∈ yahalom ==> (Key (shrK A) ∈ analz (knows Spy evs)) = (A ∈ bad)"
by auto
```

```
lemma Spy_see_shrK_D [dest!]:
  "[|Key (shrK A) ∈ parts (knows Spy evs); evs ∈ yahalom|] ==> A ∈ bad"
by (blast dest: Spy_see_shrK)
```

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))"
apply (erule rev_mp)
apply (erule yahalom.induct, force,
  frule_tac [6] YM4_parts_knows_Spy, simp_all)
```

Fake

```
apply (force dest!: keysFor_parts_insert, auto)
done
```

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))"
by (erule yahalom.induct,
  drule_tac [7] YM4_analz_knows_Spy, analz_freshK, spy_analz, blast)
```

```
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))"
by (simp only: analz_image_freshK analz_image_freshK_simps)
```

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'"
apply (erule rev_mp, erule rev_mp)
apply (erule yahalom.induct, simp_all)

```

YM3, by freshness, and YM4

```

apply blast+
done

```

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)"
apply (erule yahalom.induct, force, drule_tac [6] YM4_analz_knows_Spy)
apply (simp_all add: pushes_analz_insert_eq analz_insert_freshK, spy_analz)

apply (blast dest: unique_session_keys)
done

```

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)"
by (blast dest: secrecy_lemma)

```

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"
apply (erule rev_mp)
apply (erule yahalom.induct, force,
  frule_tac [6] YM4_parts_knows_Spy, simp_all)

```

Fake, YM3

```

apply blast+
done

```

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)"
by (blast dest!: A_trusts_YM3 Spy_not_see_encrypted_key)

```

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"
apply (erule rev_mp)
apply (erule yahalom.induct, force,
        frule_tac [6] YM4_parts_knows_Spy, simp_all)

```

Fake, YM3

```

apply blast+
done

```

17.7 The Flaw in the Model

Up to now, the reasoning is similar to standard Yahalom. Now the doubtful reasoning occurs. We should not be assuming that an unknown key is secure, but the model allows us to: there is no Oops rule to let session keys become compromised.

B knows, by the second part of A's message, that the Server distributed the key quoting nonce NB. This part says nothing about agent names. Secrecy of K is assumed; the valid Yahalom proof uses (and later proves) the secrecy of NB.

```

lemma B_trusts_YM4_newK [rule_format]:
  "[|Key K ∉ analz (knows Spy evs); evs ∈ yahalom|]
   => Crypt K (Nonce NB) ∈ parts (knows Spy evs) -->
      (∃ A B NA. Says Server A
        {|Crypt (shrK A) {|Agent B, Key K,
                          Nonce NA, Nonce NB|},
          Crypt (shrK B) {|Agent A, Key K|}|}
        ∈ set evs)"
apply (erule rev_mp)
apply (erule yahalom.induct, force,
        frule_tac [6] YM4_parts_knows_Spy)
apply (analz_mono_contra, simp_all)

```

Fake

apply blast

YM3

apply blast

A is uncompromised because NB is secure A's certificate guarantees the existence of the Server message

```
apply (blast dest!: Gets_imp_Says Crypt_Spy_analz_bad
        dest: Says_imp_spies
        parts.Inj [THEN parts.Fst, THEN A_trusts_YM3])
done
```

B's session key guarantee from YM4. The two certificates contribute to a single conclusion about the Server's message.

lemma B_trusts_YM4:

```
"[| Gets B {|Crypt (shrK B) {|Agent A, Key K|},
      Crypt K (Nonce NB)|} ∈ set evs;
  Says B Server
    {|Agent B, Nonce NB, Crypt (shrK B) {|Agent A, Nonce NA|}|}
  ∈ set evs;
  A ∉ bad; B ∉ bad; evs ∈ yahalom |]
==> ∃na nb. Says Server A
    {|Crypt (shrK A) {|Agent B, Key K, na, nb|},
     Crypt (shrK B) {|Agent A, Key K|}|}
  ∈ set evs"
```

```
by (blast dest: B_trusts_YM4_newK B_trusts_YM4_shrK Spy_not_see_encrypted_key
      unique_session_keys)
```

The obvious combination of B_trusts_YM4 with Spy_not_see_encrypted_key

lemma B_gets_good_key:

```
"[| Gets B {|Crypt (shrK B) {|Agent A, Key K|},
      Crypt K (Nonce NB)|} ∈ set evs;
  Says B Server
    {|Agent B, Nonce NB, Crypt (shrK B) {|Agent A, Nonce NA|}|}
  ∈ set evs;
  A ∉ bad; B ∉ bad; evs ∈ yahalom |]
==> Key K ∉ analz (knows Spy evs)"
```

```
by (blast dest!: B_trusts_YM4 Spy_not_see_encrypted_key)
```

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)"
```

```
apply (erule yahalom.induct, force,
        frule_tac [6] YM4_parts_knows_Spy)
```

```

apply (analz_mono_contra, simp_all)

Fake

apply blast

YM3: by new_keys_not_used, the message Crypt K (Nonce NB) could not exist

apply (force dest!: Crypt_imp_keysFor)

YM4: was Crypt K (Nonce NB) the very last message? If not, use the induction hypothesis

apply (simp add: ex_disj_distrib)

yes: apply unicity of session keys

apply (blast dest!: Gets_imp_Says A_trusts_YM3 B_trusts_YM4_shrK
      Crypt_Spy_analz_bad
      dest: Says_imp_knows_Spy [THEN parts.Inj] unique_session_keys)

done

```

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"
by (blast intro!: A_Said_YM3_lemma
      dest: Spy_not_see_encrypted_key B_trusts_YM4 Gets_imp_Says)

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"

```

```

definition broken :: "agent set" where

```

— 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"
apply (intro exI bexI)
apply (rule_tac [2] zg.Nil
  [THEN zg.ZG1, THEN zg.Reception [of _ A B],
  THEN zg.ZG2, THEN zg.Reception [of _ B A],
  THEN zg.ZG3, THEN zg.Reception [of _ A TTP],
  THEN zg.ZG4])
apply (possibility, auto)
done

```

17.8 Basic Lemmas

```

lemma Gets_imp_Says:
  "[| Gets B X ∈ set evs; evs ∈ zg |] ==> ∃A. Says A B X ∈ set evs"
apply (erule rev_mp)
apply (erule zg.induct, auto)
done

```

```

lemma Gets_imp_knows_Spy:
  "[| Gets B X ∈ set evs; evs ∈ zg |] ==> X ∈ spies evs"
by (blast dest!: Gets_imp_Says Says_imp_knows_Spy)

```

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)"
apply (erule rev_mp)
apply (erule zg.induct)
apply (simp_all add: parts_insert_knows_A)
done

```

```

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"
apply (erule rev_mp)
apply (erule zg.induct, auto)
done

```

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)"
by (blast dest: Gets_imp_Says)

```

```

lemma Spy_see_priK [simp]:
  "evs ∈ zg ==> (Key (priK A) ∈ parts (spies evs)) = (A ∈ bad)"
apply (erule zg.induct)
apply (frule_tac [5] ZG2_msg_in_parts_spies, auto)
done

```

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)"
by auto

```

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"
apply clarify
apply (erule rev_mp)
apply (erule zg.induct)
apply (frule_tac [5] ZG2_msg_in_parts_spies, auto)
done

```

```

lemma NRO_sender:
  "[|Says A' B {|n, b, l, C, Crypt (priK A) X|} ∈ set evs; evs ∈ zg|]
  ==> A' ∈ {A, Spy}"
apply (erule rev_mp)
apply (erule zg.induct, simp_all)
done

```

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"
apply (drule Gets_imp_Says, assumption)
apply clarify
apply (frule NRO_sender, auto)

```

We are left with the case where the sender is *Spy* and not equal to *A*, because *A* ∉ *bad*. Thus theorem *NRO_validity_good* applies.

```

apply (blast dest: NRO_validity_good [OF refl])
done

```

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"
apply clarify
apply (erule rev_mp)
apply (erule zg.induct)
apply (frule_tac [5] ZG2_msg_in_parts_spies, auto)
done

```

```

lemma NRR_sender:
  "[|Says B' A {|n, a, l, Crypt (priK B) X|} ∈ set evs; evs ∈ zg|]
  ==> B' ∈ {B, Spy}"
apply (erule rev_mp)
apply (erule zg.induct, simp_all)
done

```

Holds also for $B = \text{Spy}$!

```

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"
apply clarify
apply (frule NRR_sender, auto)

```

We are left with the case where $B' = \text{Spy}$ and $B' \neq B$, i.e. $B \notin \text{bad}$, when we can apply *NRR_validity_good*.

```

apply (blast dest: NRR_validity_good [OF refl])
done

```

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"
apply clarify
apply (erule rev_mp)
apply (erule zg.induct)

```

```
apply (frule_tac [5] ZG2_msg_in_parts_spies, simp_all)
```

Fake

```
apply (blast dest!: Fake_parts_sing_imp_Un)
done
```

lemma *sub_K_sender*:

```
"[|Says A' TTP {|n, b, l, k, Crypt (priK A) X|} ∈ set evs; evs ∈ zg|]
==> A' ∈ {A, Spy}"
apply (erule rev_mp)
apply (erule zg.induct, simp_all)
done
```

Holds also for $A = \text{Spy}$!

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"
apply (drule Gets_imp_Says, assumption)
apply clarify
apply (frule sub_K_sender, auto)
```

We are left with the case where the sender is *Spy* and not equal to *A*, because $A \notin \text{broken}$. Thus theorem *sub_K_validity_good* applies.

```
apply (blast dest: sub_K_validity_good [OF refl])
done
```

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"
apply clarify
apply (erule rev_mp)
apply (erule zg.induct)
apply (frule_tac [5] ZG2_msg_in_parts_spies, simp_all)
```

Fake

```
apply (blast dest!: Fake_parts_sing_imp_Un)
```

ZG2

```
apply (blast dest: parts_cut)
done
```

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"
apply clarify
apply (erule rev_mp)
apply (erule zg.induct)
apply (frule_tac [5] ZG2_msg_in_parts_spies, simp_all)

ZG4

apply clarify
apply (rule sub_K_validity, auto)
done

```

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"
by (blast dest: con_K_validity Notes_TTP_imp_Says_A)

```

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"
apply clarify
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule zg.induct)
apply (frule_tac [5] ZG2_msg_in_parts_spies, auto)

ZG1: freshness

apply (blast dest: parts.Body)

```

done

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"
```

apply clarify

apply (erule rev_mp)

apply (erule rev_mp)

apply (erule zg.induct)

apply (frule_tac [5] ZG2_msg_in_parts_spies, simp_all)

Fake

apply blast

ZG1: freshness

apply (blast dest: parts.Body)

ZG3

apply (blast dest: A_unicity [OF refl])

done

lemma *Crypt_used_imp_L_used*:

```
"[| Crypt (priK TTP) {|F, A, B, L, K|} ∈ used evs; evs ∈ zg |]
==> L ∈ used evs"
```

apply (erule rev_mp)

apply (erule zg.induct, auto)

Fake

apply (blast dest!: Fake_parts_sing_imp_Un)

ZG2: freshness

apply (blast dest: parts.Body)

done

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"
```

```

apply clarify
apply (erule rev_mp)
apply (erule rev_mp)
apply (erule zg.induct)
apply (frule_tac [5] ZG2_msg_in_parts_spies, simp_all)

```

Fake

```

apply (simp add: parts_insert_knows_A)
apply (blast dest: Fake_parts_sing_imp_Un)

```

ZG1

```

apply (blast dest: Crypt_used_imp_L_used)

```

ZG2

```

apply (blast dest: parts_cut)

```

ZG4

```

apply (blast intro: sub_K_implies_NRR [OF refl]
        dest: Gets_imp_knows_Spy [THEN parts.Inj])

```

done

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"

```

```

apply clarify
apply (erule rev_mp)
apply (erule zg.induct)
apply (frule_tac [5] ZG2_msg_in_parts_spies, simp_all)

```

Fake

```

apply (blast dest!: Fake_parts_sing_imp_Un)

```

ZG2

```

apply (blast dest: parts_cut)

```

done

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 Conventional protocols: rely on conventional Message, Event and Public – Shared-key protocols

theory $Auth_Shared$

```

imports
  "NS_Shared"
  "Kerberos_BAN"
  "Kerberos_BAN_Gets"
  "KerberosIV"
  "KerberosIV_Gets"
  "KerberosV"
  "OtwayRees"
  "OtwayRees_AN"
  "OtwayRees_Bad"
  "OtwayReesBella"
  "WooLam"
  "Recur"
  "Yahalom"
  "Yahalom2"
  "Yahalom_Bad"
  "ZhouGollmann"
begin
end

```

19 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"
apply (intro exI bexI)
apply (rule_tac [2] ns_public.Nil [THEN ns_public.NS1, THEN ns_public.NS2,
THEN ns_public.NS3])
by possibility

lemma Spy_see_priEK [simp]:
  "evs ∈ ns_public ⇒ (Key (priEK A) ∈ parts (spies evs)) = (A ∈ bad)"
by (erule ns_public.induct, auto)

lemma Spy_analz_priEK [simp]:
  "evs ∈ ns_public ⇒ (Key (priEK A) ∈ analz (spies evs)) = (A ∈ bad)"
by auto

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)"
apply (erule ns_public.induct, simp_all)
apply (blast intro: analz_insertI)+
done

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'"
apply (erule rev_mp, erule rev_mp, erule rev_mp)
apply (erule ns_public.induct, simp_all)

apply (blast intro!: analz_insertI)+
done

```

```

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)"
apply (erule rev_mp)
apply (erule ns_public.induct, simp_all, spy_analz)
apply (blast dest: unique_NA intro: no_nonce_NS1_NS2)+
done

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"
apply (erule ns_public.induct)
apply (auto dest: Spy_not_see_NA unique_NA)
done

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"
by (blast intro: A_trusts_NS2_lemma)

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"
apply (erule ns_public.induct, simp_all)

apply (blast intro!: analz_insertI)
done

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'"
apply (erule rev_mp, erule rev_mp, erule rev_mp)
apply (erule ns_public.induct, simp_all)

```

```

apply (blast intro!: analz_insertI)+
done

```

```

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)"
apply (erule rev_mp, erule rev_mp)
apply (erule ns_public.induct, simp_all, spy_analz)
apply (simp_all add: all_conj_distrib)
apply (blast intro: no_nonce_NS1_NS2)+
done

```

```

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)"
apply (erule ns_public.induct, auto)
by (blast intro: no_nonce_NS1_NS2)+

```

```

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"
by (blast intro: B_trusts_NS3_lemma)

```

```

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)"
apply (erule ns_public.induct, simp_all, spy_analz)

```

```

apply blast

```

```

apply (blast intro: no_nonce_NS1_NS2)

```

```

apply clarify
apply (frule_tac A' = A in
  Says_imp_knows_Spy [THEN parts.Inj, THEN unique_NB], auto)
apply (rename_tac evs3 B' C)

```

This is the attack!

1. $\bigwedge \text{evs3 } B' C.$

$$[[A \notin \text{bad}; B \notin \text{bad}; \text{evs3} \in \text{ns_public};$$

$$\text{Says } A B' (\text{Crypt } (\text{pubK } B') \{ \text{Nonce } NA, \text{Agent } A \})$$

```

    ∈ set evs3;
    Says C A (Crypt (pubK A) {Nonce NA, Nonce NB})
    ∈ set evs3;
    B' ∈ bad;
    Says B A (Crypt (pubK A) {Nonce NA, Nonce NB})
    ∈ set evs3;
    Nonce NB ∉ analz (knows Spy evs3)]
    ⇒ False

```

oops

end

20 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"
apply (intro exI bexI)
apply (rule_tac [2] ns_public.Nil [THEN ns_public.NS1, THEN ns_public.NS2,
  THEN ns_public.NS3], possibility)
done

```

```

lemma Spy_see_priEK [simp]:
  "evs ∈ ns_public ⇒ (Key (priEK A) ∈ parts (spies evs)) = (A ∈ bad)"
by (erule ns_public.induct, auto)

```

```

lemma Spy_analz_priEK [simp]:
  "evs ∈ ns_public ⇒ (Key (priEK A) ∈ analz (spies evs)) = (A ∈ bad)"
by auto

```

20.1 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)"
apply (erule ns_public.induct, simp_all)
apply (blast intro: analz_insertI)+
done

```

```

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'"
apply (erule rev_mp, erule rev_mp, erule rev_mp)
apply (erule ns_public.induct, simp_all)

```

```

apply (blast intro: analz_insertI)+
done

```

```

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)"
apply (erule rev_mp)
apply (erule ns_public.induct, simp_all, spy_analz)
apply (blast dest: unique_NA intro: no_nonce_NS1_NS2)+
done

```

```

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"
apply (erule ns_public.induct, simp_all)

apply (blast dest: Spy_not_see_NA)+
done

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"
by (blast intro: A_trusts_NS2_lemma)

```

```

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"
apply (erule ns_public.induct, simp_all)

apply (blast intro!: analz_insertI)
done

```

20.2 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'"
apply (erule rev_mp, erule rev_mp, erule rev_mp)
apply (erule ns_public.induct, simp_all)

apply (blast intro: analz_insertI)+
done

```

```

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)"
apply (erule rev_mp)
apply (erule ns_public.induct, simp_all, spy_analz)

```

```

apply (blast intro: no_nonce_NS1_NS2)+
done

```

```

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"
by (erule ns_public.induct, auto)

```

```

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"
by (blast intro: B_trusts_NS3_lemma)

```

20.3 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"
by (erule ns_public.induct, auto)

end

```

21 Bijections between natural numbers and other types

```

theory Nat_Bijection
imports Main Parity
begin

```

21.1 Type $\text{nat} \times \text{nat}$

Triangle numbers: 0, 1, 3, 6, 10, 15, ...

```

definition
  triangle :: "nat ⇒ nat"
where
  "triangle n = n * Suc n div 2"

```

```

lemma triangle_0 [simp]: "triangle 0 = 0"
unfolding triangle_def by simp

```

```

lemma triangle_Suc [simp]: "triangle (Suc n) = triangle n + Suc n"

```

unfolding `triangle_def` by `simp`

definition

```
prod_encode :: "nat × nat ⇒ nat"
where
  "prod_encode = (λ(m, n). triangle (m + n) + m)"
```

In this auxiliary function, `triangle k + m` is an invariant.

fun

```
prod_decode_aux :: "nat ⇒ nat ⇒ nat × nat"
where
  "prod_decode_aux k m =
    (if m ≤ k then (m, k - m) else prod_decode_aux (Suc k) (m - Suc k))"
```

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

definition

```
prod_decode :: "nat ⇒ nat × nat"
where
  "prod_decode = prod_decode_aux 0"
```

lemma `prod_encode_prod_decode_aux`:

```
"prod_encode (prod_decode_aux k m) = triangle k + m"
apply (induct k m rule: prod_decode_aux.induct)
apply (subst prod_decode_aux.simps)
apply (simp add: prod_encode_def)
done
```

lemma `prod_decode_inverse` [`simp`]: "`prod_encode (prod_decode n) = n`"
 unfolding `prod_decode_def` by (`simp add: prod_encode_prod_decode_aux`)

lemma `prod_decode_triangle_add`:

```
"prod_decode (triangle k + m) = prod_decode_aux k m"
apply (induct k arbitrary: m)
apply (simp add: prod_decode_def)
apply (simp only: triangle_Suc add_assoc)
apply (subst prod_decode_aux.simps, simp)
done
```

lemma `prod_encode_inverse` [`simp`]: "`prod_decode (prod_encode x) = x`"

```
unfolding prod_encode_def
apply (induct x)
apply (simp add: prod_decode_triangle_add)
apply (subst prod_decode_aux.simps, simp)
done
```

lemma `inj_prod_encode`: "`inj_on prod_encode A`"
 by (`rule inj_on_inverseI, rule prod_encode_inverse`)

lemma `inj_prod_decode`: "`inj_on prod_decode A`"
 by (`rule inj_on_inverseI, rule prod_decode_inverse`)

lemma `surj_prod_encode`: "`surj prod_encode`"
 by (`rule surjI, rule prod_decode_inverse`)

```
lemma surj_prod_decode: "surj prod_decode"
```

```
by (rule surjI, rule prod_encode_inverse)
```

```
lemma bij_prod_encode: "bij prod_encode"
```

```
by (rule bijI [OF inj_prod_encode surj_prod_encode])
```

```
lemma bij_prod_decode: "bij prod_decode"
```

```
by (rule bijI [OF inj_prod_decode surj_prod_decode])
```

```
lemma prod_encode_eq: "prod_encode x = prod_encode y  $\longleftrightarrow$  x = y"
```

```
by (rule inj_prod_encode [THEN inj_eq])
```

```
lemma prod_decode_eq: "prod_decode x = prod_decode y  $\longleftrightarrow$  x = y"
```

```
by (rule inj_prod_decode [THEN inj_eq])
```

Ordering properties

```
lemma le_prod_encode_1: "a  $\leq$  prod_encode (a, b)"
```

```
unfolding prod_encode_def by simp
```

```
lemma le_prod_encode_2: "b  $\leq$  prod_encode (a, b)"
```

```
unfolding prod_encode_def by (induct b, simp_all)
```

21.2 Type $\text{nat} + \text{nat}$

definition

```
sum_encode :: "nat + nat  $\Rightarrow$  nat"
```

where

```
"sum_encode x = (case x of Inl a  $\Rightarrow$  2 * a | Inr b  $\Rightarrow$  Suc (2 * b))"
```

definition

```
sum_decode :: "nat  $\Rightarrow$  nat + nat"
```

where

```
"sum_decode n = (if even n then Inl (n div 2) else Inr (n div 2))"
```

```
lemma sum_encode_inverse [simp]: "sum_decode (sum_encode x) = x"
```

```
unfolding sum_decode_def sum_encode_def
```

```
by (induct x) simp_all
```

```
lemma sum_decode_inverse [simp]: "sum_encode (sum_decode n) = n"
```

```
unfolding sum_decode_def sum_encode_def numeral_2_eq_2
```

```
by (simp add: even_nat_div_two_times_two odd_nat_div_two_times_two_plus_one  
del: mult_Suc)
```

```
lemma inj_sum_encode: "inj_on sum_encode A"
```

```
by (rule inj_on_inverseI, rule sum_encode_inverse)
```

```
lemma inj_sum_decode: "inj_on sum_decode A"
```

```
by (rule inj_on_inverseI, rule sum_decode_inverse)
```

```
lemma surj_sum_encode: "surj sum_encode"
```

```
by (rule surjI, rule sum_decode_inverse)
```

```
lemma surj_sum_decode: "surj sum_decode"
```

by (rule surjI, rule sum_encode_inverse)

lemma bij_sum_encode: "bij sum_encode"
by (rule bijI [OF inj_sum_encode surj_sum_encode])

lemma bij_sum_decode: "bij sum_decode"
by (rule bijI [OF inj_sum_decode surj_sum_decode])

lemma sum_encode_eq: "sum_encode x = sum_encode y \longleftrightarrow x = y"
by (rule inj_sum_encode [THEN inj_eq])

lemma sum_decode_eq: "sum_decode x = sum_decode y \longleftrightarrow x = y"
by (rule inj_sum_decode [THEN inj_eq])

21.3 Type int

definition

int_encode :: "int \Rightarrow nat"

where

"int_encode i = sum_encode (if $0 \leq i$ then Inl (nat i) else Inr (nat (- i - 1)))"

definition

int_decode :: "nat \Rightarrow int"

where

"int_decode n = (case sum_decode n of Inl a \Rightarrow int a | Inr b \Rightarrow - int b - 1)"

lemma int_encode_inverse [simp]: "int_decode (int_encode x) = x"
unfolding int_decode_def int_encode_def by simp

lemma int_decode_inverse [simp]: "int_encode (int_decode n) = n"
unfolding int_decode_def int_encode_def using sum_decode_inverse [of n]
by (cases "sum_decode n", simp_all)

lemma inj_int_encode: "inj_on int_encode A"
by (rule inj_on_inverseI, rule int_encode_inverse)

lemma inj_int_decode: "inj_on int_decode A"
by (rule inj_on_inverseI, rule int_decode_inverse)

lemma surj_int_encode: "surj int_encode"
by (rule surjI, rule int_decode_inverse)

lemma surj_int_decode: "surj int_decode"
by (rule surjI, rule int_encode_inverse)

lemma bij_int_encode: "bij int_encode"
by (rule bijI [OF inj_int_encode surj_int_encode])

lemma bij_int_decode: "bij int_decode"
by (rule bijI [OF inj_int_decode surj_int_decode])

lemma int_encode_eq: "int_encode x = int_encode y \longleftrightarrow x = y"

```
by (rule inj_int_encode [THEN inj_eq])
```

```
lemma int_decode_eq: "int_decode x = int_decode y  $\longleftrightarrow$  x = y"
by (rule inj_int_decode [THEN inj_eq])
```

21.4 Type nat list

```
fun
```

```
  list_encode :: "nat list  $\Rightarrow$  nat"
```

```
where
```

```
  "list_encode [] = 0"
```

```
| "list_encode (x # xs) = Suc (prod_encode (x, list_encode xs))"
```

```
function
```

```
  list_decode :: "nat  $\Rightarrow$  nat list"
```

```
where
```

```
  "list_decode 0 = []"
```

```
| "list_decode (Suc n) = (case prod_decode n of (x, y)  $\Rightarrow$  x # list_decode y)"
```

```
by pat_completeness auto
```

```
termination list_decode
```

```
apply (relation "measure id", simp_all)
```

```
apply (drule arg_cong [where f="prod_encode"])
```

```
apply (drule sym)
```

```
apply (simp add: le_imp_less_Suc le_prod_encode_2)
```

```
done
```

```
lemma list_encode_inverse [simp]: "list_decode (list_encode x) = x"
```

```
by (induct x rule: list_encode.induct) simp_all
```

```
lemma list_decode_inverse [simp]: "list_encode (list_decode n) = n"
```

```
apply (induct n rule: list_decode.induct, simp)
```

```
apply (simp split: prod.split)
```

```
apply (simp add: prod_decode_eq [symmetric])
```

```
done
```

```
lemma inj_list_encode: "inj_on list_encode A"
```

```
by (rule inj_on_inverseI, rule list_encode_inverse)
```

```
lemma inj_list_decode: "inj_on list_decode A"
```

```
by (rule inj_on_inverseI, rule list_decode_inverse)
```

```
lemma surj_list_encode: "surj list_encode"
```

```
by (rule surjI, rule list_decode_inverse)
```

```
lemma surj_list_decode: "surj list_decode"
```

```
by (rule surjI, rule list_encode_inverse)
```

```
lemma bij_list_encode: "bij list_encode"
```

```
by (rule bijI [OF inj_list_encode surj_list_encode])
```

```
lemma bij_list_decode: "bij list_decode"
```

```
by (rule bijI [OF inj_list_decode surj_list_decode])
```

```
lemma list_encode_eq: "list_encode x = list_encode y  $\longleftrightarrow$  x = y"
by (rule inj_list_encode [THEN inj_eq])
```

```
lemma list_decode_eq: "list_decode x = list_decode y  $\longleftrightarrow$  x = y"
by (rule inj_list_decode [THEN inj_eq])
```

21.5 Finite sets of naturals

21.5.1 Preliminaries

```
lemma finite_vimage_Suc_iff: "finite (Suc -' F)  $\longleftrightarrow$  finite F"
apply (safe intro!: finite_vimageI inj_Suc)
apply (rule finite_subset [where B="insert 0 (Suc ' Suc -' F)"])
apply (rule subsetI, case_tac x, simp, simp)
apply (rule finite_insert [THEN iffD2])
apply (erule finite_imageI)
done
```

```
lemma vimage_Suc_insert_0: "Suc -' insert 0 A = Suc -' A"
by auto
```

```
lemma vimage_Suc_insert_Suc:
  "Suc -' insert (Suc n) A = insert n (Suc -' A)"
by auto
```

```
lemma even_nat_Suc_div_2: "even x  $\implies$  Suc x div 2 = x div 2"
by (simp only: numeral_2_eq_2 even_nat_plus_one_div_two)
```

```
lemma div2_even_ext_nat:
  "[x div 2 = y div 2; even x = even y]  $\implies$  (x::nat) = y"
apply (rule mod_div_equality [of x 2, THEN subst])
apply (rule mod_div_equality [of y 2, THEN subst])
apply (case_tac "even x")
apply (simp add: numeral_2_eq_2 even_nat_equiv_def)
apply (simp add: numeral_2_eq_2 odd_nat_equiv_def)
done
```

21.5.2 From sets to naturals

definition

```
set_encode :: "nat set  $\Rightarrow$  nat"
where
  "set_encode = setsum (op ^ 2)"
```

```
lemma set_encode_empty [simp]: "set_encode {} = 0"
by (simp add: set_encode_def)
```

```
lemma set_encode_insert [simp]:
  "[finite A; n  $\notin$  A]  $\implies$  set_encode (insert n A) = 2^n + set_encode A"
by (simp add: set_encode_def)
```

```
lemma even_set_encode_iff: "finite A  $\implies$  even (set_encode A)  $\longleftrightarrow$  0  $\notin$  A"
unfolding set_encode_def by (induct set: finite, auto)
```

```

lemma set_encode_vimage_Suc: "set_encode (Suc -' A) = set_encode A div 2"
apply (cases "finite A")
apply (erule finite_induct, simp)
apply (case_tac x)
apply (simp add: even_nat_Suc_div_2 even_set_encode_iff vimage_Suc_insert_0)
apply (simp add: finite_vimageI add_commute vimage_Suc_insert_Suc)
apply (simp add: set_encode_def finite_vimage_Suc_iff)
done

```

```

lemmas set_encode_div_2 = set_encode_vimage_Suc [symmetric]

```

21.5.3 From naturals to sets

definition

```

set_decode :: "nat ⇒ nat set"
where
  "set_decode x = {n. odd (x div 2 ^ n)}"

```

```

lemma set_decode_0 [simp]: "0 ∈ set_decode x ↔ odd x"
by (simp add: set_decode_def)

```

```

lemma set_decode_Suc [simp]:
  "Suc n ∈ set_decode x ↔ n ∈ set_decode (x div 2)"
by (simp add: set_decode_def div_mult2_eq)

```

```

lemma set_decode_zero [simp]: "set_decode 0 = {}"
by (simp add: set_decode_def)

```

```

lemma set_decode_div_2: "set_decode (x div 2) = Suc -' set_decode x"
by auto

```

```

lemma set_decode_plus_power_2:
  "n ∉ set_decode z ⇒ set_decode (2 ^ n + z) = insert n (set_decode z)"
apply (induct n arbitrary: z, simp_all)
apply (rule set_eqI, induct_tac x, simp, simp add: even_nat_Suc_div_2)
apply (rule set_eqI, induct_tac x, simp, simp add: add_commute)
done

```

```

lemma finite_set_decode [simp]: "finite (set_decode n)"
apply (induct n rule: nat_less_induct)
apply (case_tac "n = 0", simp)
apply (drule_tac x="n div 2" in spec, simp)
apply (simp add: set_decode_div_2)
apply (simp add: finite_vimage_Suc_iff)
done

```

21.5.4 Proof of isomorphism

```

lemma set_decode_inverse [simp]: "set_encode (set_decode n) = n"
apply (induct n rule: nat_less_induct)
apply (case_tac "n = 0", simp)
apply (drule_tac x="n div 2" in spec, simp)
apply (simp add: set_decode_div_2 set_encode_vimage_Suc)
apply (erule div2_even_ext_nat)
apply (simp add: even_set_encode_iff)

```

```

done

lemma set_encode_inverse [simp]: "finite A  $\implies$  set_decode (set_encode A)
= A"
apply (erule finite_induct, simp_all)
apply (simp add: set_decode_plus_power_2)
done

lemma inj_on_set_encode: "inj_on set_encode (Collect finite)"
by (rule inj_on_inverseI [where g="set_decode"], simp)

lemma set_encode_eq:
  "[[finite A; finite B]  $\implies$  set_encode A = set_encode B  $\longleftrightarrow$  A = B"
by (rule iffI, simp add: inj_onD [OF inj_on_set_encode], simp)

end

```

22 The TLS Protocol: Transport Layer Security

```

theory TLS imports Public "~/src/HOL/Library/Nat_Bijection" begin

```

```

definition certificate :: "[agent,key] => msg" where
  "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

```

  apply (rule exI [of _ "%(x,y,z). prod_encode(x, prod_encode(y,z))"])
  apply (simp add: inj_on_def prod_encode_eq)
  done

```

```

specification (sessionK)

```

```

inj_sessionK: "inj sessionK"
— sessionK is collision-free; also, no clientK clashes with any serverK.
apply (rule exI [of _
      "%((x,y,z), r). prod_encode(role_case 0 1 r,
                                prod_encode(x, prod_encode(y,z)))"])
apply (simp add: inj_on_def prod_encode_eq split: role.split)
done

```

axiomatization where

— sessionK makes symmetric keys
isSym_sessionK: "sessionK nonces \in symKeys" and

— sessionK never clashes with a long-term symmetric key (they don't exist in TLS anyway)

```

sessionK_neq_shrK [iff]: "sessionK nonces  $\neq$  shrK A"

```

inductive_set *tls* :: "event list set"

where

Nil: — The initial, empty trace
 "[] \in *tls*"

| *Fake*: — The Spy may say anything he can say. The sender field is correct, but agents don't use that information.

```

"[| evsf  $\in$  tls; X  $\in$  synth (analz (spies evsf)) |]"
==> Says Spy B X # evsf  $\in$  tls"

```

| *SpyKeys*: — The spy may apply *PRF* and *sessionK* to available nonces

```

"[| evsSK  $\in$  tls;
   {Nonce NA, Nonce NB, Nonce M}  $\leq$  analz (spies evsSK) |]"
==> Notes Spy {| Nonce (PRF(M,NA,NB)),
                Key (sessionK((NA,NB,M),role)) |} # evsSK  $\in$  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 range *PRF* because CLIENT RANDOM is 28 bytes while MASTER SECRET is 48 bytes

```

"[| evsCH  $\in$  tls; Nonce NA  $\notin$  used evsCH; NA  $\notin$  range PRF |]"
==> Says A B {|Agent A, Nonce NA, Number SID, Number PA|}
      # evsCH  $\in$  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  $\in$  tls; Nonce NB  $\notin$  used evsSH; NB  $\notin$  range PRF;
   Says A' B {|Agent A, Nonce NA, Number SID, Number PA|}
    $\in$  set evsSH |]"
==> Says B A {|Nonce NB, Number SID, Number PB|} # evsSH  $\in$  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 $\text{PRF}(PMS, NA, NB)$. Either party may send its message first.

/ ClientFinished:

— The occurrence of Notes A —Agent B, Nonce PMS— stops the rule's applying when the Spy has satisfied the "Says A B" by repaying messages sent by the true client; in that case, the Spy does not know PMS and could not send ClientFinished. One could simply put $A \neq \text{Spy}$ into the rule, but one should not expect the spy to be well-behaved.

```
"[| evsCF ∈ tls;
  Says A B {|Agent A, Nonce NA, Number SID, Number PA|}
  ∈ set evsCF;
  Says B' A {|Nonce NB, Number SID, Number PB|} ∈ set evsCF;
  Notes A {|Agent B, Nonce PMS|} ∈ set evsCF;
  M = PRF(PMS, NA, NB) |]
==> Says A B (Crypt (clientK(NA, NB, M))
  (Hash{|Number SID, Nonce M,
        Nonce NA, Number PA, Agent A,
        Nonce NB, Number PB, Agent B|}))
  # evsCF ∈ tls"
```

/ ServerFinished:

— Keeping A' and A" distinct means B cannot even check that the two messages originate from the same source.

```
"[| evsSF ∈ tls;
  Says A' B {|Agent A, Nonce NA, Number SID, Number PA|}
  ∈ set evsSF;
```

```

Says B A {|Nonce NB, Number SID, Number PB|} ∈ set evsSF;
Says A'' B (Crypt (pubK B) (Nonce PMS)) ∈ set evsSF;
M = PRF(PMS,NA,NB) []
==> Says B A (Crypt (serverK(NA,NB,M))
              (Hash{|Number SID, Nonce M,
                    Nonce NA, Number PA, Agent A,
                    Nonce NB, Number PB, Agent B|}))
# evsSF ∈ tls"

| ClientAccepts:
— Having transmitted ClientFinished and received an identical message encrypted with serverK, the client stores the parameters needed to resume this session. The "Notes A ..." premise is used to prove Notes_master_imp_Crypt_PMS.
"[| evsCA ∈ tls;
   Notes A {|Agent B, Nonce PMS|} ∈ set evsCA;
   M = PRF(PMS,NA,NB);
   X = Hash{|Number SID, Nonce M,
            Nonce NA, Number PA, Agent A,
            Nonce NB, Number PB, Agent B|};
   Says A B (Crypt (clientK(NA,NB,M)) X) ∈ set evsCA;
   Says B' A (Crypt (serverK(NA,NB,M)) X) ∈ set evsCA []
==>
   Notes A {|Number SID, Agent A, Agent B, Nonce M|} # evsCA ∈
tls"

| ServerAccepts:
— Having transmitted ServerFinished and received an identical message 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 ≠ sessionK arg"
by (simp add: symKeys_neq_imp_neq)

declare pubK_neq_sessionK [THEN not_sym, iff]

lemma priK_neq_sessionK [iff]: "invKey (publicKey b A) ≠ sessionK arg"
by (simp add: symKeys_neq_imp_neq)

declare priK_neq_sessionK [THEN not_sym, iff]

lemmas keys_distinct = pubK_neq_sessionK priK_neq_sessionK

```

22.1 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"
apply (intro exI bexI)
apply (rule_tac [2] tls.Nil
      [THEN tls.ClientHello, THEN tls.ServerHello,
       THEN tls.Certificate, THEN tls.ClientKeyExch,
       THEN tls.ClientFinished, THEN tls.ServerFinished,
       THEN tls.ClientAccepts], possibility, blast+)
done
```

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"
apply (intro exI bexI)
apply (rule_tac [2] tls.Nil
      [THEN tls.ClientHello, THEN tls.ServerHello,
       THEN tls.Certificate, THEN tls.ClientKeyExch,
       THEN tls.ServerFinished, THEN tls.ClientFinished,
       THEN tls.ServerAccepts], possibility, blast+)
done
```

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"
apply (intro exI bexI)
apply (rule_tac [2] tls.Nil
      [THEN tls.ClientHello, THEN tls.ServerHello,
       THEN tls.Certificate, THEN tls.ClientKeyExch,
       THEN tls.CertVerify], possibility, blast+)
done
```

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 |]"
  ==>  $\exists$  NA PA NB PB X.  $\exists$  evs  $\in$  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"
apply (intro exI bexI)
apply (rule_tac [2] tls.ClientHello
      [THEN tls.ServerHello,
      THEN tls.ServerResume, THEN tls.ClientResume], possibility,
blast+)
done

```

22.2 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)"
by (erule tls.induct, force, simp_all, blast)

```

```

lemma Spy_analz_priK [simp]:
  "evs ∈ tls ==> (Key (privateKey b A) ∈ analz (spies evs)) = (A ∈ bad)"
by auto

```

```

lemma Spy_see_priK_D [dest!]:
  "[|Key (privateKey b A) ∈ parts (knows Spy evs); evs ∈ tls|] ==> A ∈
bad"
by (blast dest: Spy_see_priK)

```

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"
apply (erule rev_mp)
apply (erule tls.induct, force, simp_all, blast)
done

```

```

lemmas CX_KB_is_pubKB = Says_imp_spies [THEN parts.Inj, THEN certificate_valid]

```

22.2.1 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)"
apply (erule rev_mp)
apply (erule tls.induct,
      frule_tac [7] CX_KB_is_pubKB, force, simp_all)
apply (blast intro: parts_insertI)
done

```

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)"

```

```

apply (erule rev_mp)
apply (erule tls.induct, force, simp_all)

```

Fake

```

apply (blast intro: parts_insertI)

```

Client, Server Accept

```

apply (blast dest!: Notes_Crypt_parts_spies)+
done

```

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"
apply (erule rev_mp)
apply (erule tls.induct, force, simp_all)

```

ServerAccepts

```

apply blast
done

```

22.2.2 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"
apply (erule rev_mp, erule ssubst)
apply (erule tls.induct, force, simp_all, blast)
done

```

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"
by (blast dest!: certificate_valid intro!: TrustCertVerify_lemma)

```

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"
apply (erule rev_mp)
apply (erule tls.induct, force, simp_all, blast)
done

```

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"
by (blast dest!: certificate_valid intro!: UseCertVerify_lemma)

```

```

lemma no_Notes_A_PRF [simp]:
  "evs ∈ tls ==> Notes A {|Agent B, Nonce (PRF x)!} ∉ set evs"
apply (erule tls.induct, force, simp_all)

```

ClientKeyExch: PMS is assumed to differ from any PRF.

```

apply blast
done

```

```

lemma MS_imp_PMS [dest!]:
  "[| Nonce (PRF (PMS,NA,NB)) ∈ parts (spies evs); evs ∈ tls |]
  ==> Nonce PMS ∈ parts (spies evs)"
apply (erule rev_mp)
apply (erule tls.induct, force, simp_all)

```

Fake

```

apply (blast intro: parts_insertI)

```

Easy, e.g. by freshness

```

apply (blast dest: Notes_Crypt_parts_spies)+
done

```

22.2.3 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'"
apply (erule rev_mp, erule rev_mp, erule rev_mp)
apply (erule tls.induct, analz_mono_contra, force, simp_all (no_asm_simp))

```

Fake, ClientKeyExch

```

apply blast+
done

```

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'"
  apply (erule rev_mp, erule rev_mp)
  apply (erule tls.induct, force, simp_all)

ClientKeyExch

  apply (blast dest!: Notes_Crypt_parts_spies)
done

```

22.3 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)"
  apply (erule tls.induct)
  apply (simp_all (no_asm_simp)
    del: image_insert
    add: image_Un [THEN sym]
    insert_Key_image Un_assoc [THEN sym])

Fake

  apply spy_analz
done

```

slightly speeds up the big simplification below

```

lemma range_sessionkeys_not_priK:
  "KK <= range sessionK ==> priK B ∉ KK"
  by blast

```

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)"
  by (blast intro: analz_mono [THEN subsetD])

```

```

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))"
  apply (erule tls.induct, frule_tac [7] CX_KB_is_pubKB)
  apply (safe del: iffI)
  apply (safe del: impI iffI intro!: analz_image_keys_lemma)
  apply (simp_all (no_asm_simp)
    del: image_insert imp_disjL
    add: image_Un [THEN sym] Un_assoc [THEN sym]
    insert_Key_singleton

```

```

                                range_sessionkeys_not_priK analz_image_priK)
apply (simp_all add: insert_absorb)

Fake

apply spy_analz
done

```

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))"
by (simp del: image_insert
      add: insert_Key_singleton analz_image_keys)

```

22.3.1 Protocol goal: serverK(Na,Nb,M) and clientK(Na,Nb,M) remain secure

Lemma: session keys are never used if PMS is fresh. Nonces don't have to agree, allowing session resumption. Converse doesn't hold; revealing PMS doesn't force the keys to be sent. THEY ARE NOT SUITABLE AS SAFE ELIM RULES.

```

lemma PMS_lemma:
  "[| Nonce PMS ∉ parts (spies evs);
    K = sessionK((Na, Nb, PRF(PMS,NA,NB)), role);
    evs ∈ tls |]
  ==> Key K ∉ parts (spies evs) & (∀Y. Crypt K Y ∉ parts (spies evs))"
apply (erule rev_mp, erule ssubst)
apply (erule tls.induct, frule_tac [7] CX_KB_is_pubKB)
apply (force, simp_all (no_asm_simp))

```

Fake

```

apply (blast intro: parts_insertI)

```

SpyKeys

```

apply blast

```

Many others

```

apply (force dest!: Notes_Crypt_parts_spies Notes_master_imp_Crypt_PMS)+
done

```

```

lemma PMS_sessionK_not_spied:
  "[| Key (sessionK((Na, Nb, PRF(PMS,NA,NB)), role)) ∈ parts (spies evs);
    evs ∈ tls |]
  ==> Nonce PMS ∈ parts (spies evs)"
by (blast dest: PMS_lemma)

```

```

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)"
by (blast dest: PMS_lemma)

```

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:
  "[|  $\forall A.$  Says A Spy (Key (sessionK((NA,NB,M),role)))  $\notin$  set evs;
    Nonce M  $\notin$  analz (spies evs); evs  $\in$  tls |]
  ==> Key (sessionK((NA,NB,M),role))  $\notin$  parts (spies evs)"
apply (erule rev_mp, erule rev_mp)
apply (erule tls.induct, analz_mono_contra)
apply (force, simp_all (no_asm_simp))
```

Fake, SpyKeys

```
apply blast+
done
```

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|}  $\in$  set evs;
    evs  $\in$  tls; A  $\notin$  bad; B  $\notin$  bad |]
  ==> Nonce PMS  $\notin$  analz (spies evs)"
apply (erule rev_mp, erule tls.induct, frule_tac [7] CX_KB_is_pubKB)
apply (force, simp_all (no_asm_simp))
```

Fake

```
apply spy_analz
```

SpyKeys

```
apply force
apply (simp_all add: insert_absorb)
```

ClientHello, ServerHello, ClientKeyExch: mostly freshness reasoning

```
apply (blast dest: Notes_Crypt_parts_spies)
apply (blast dest: Notes_Crypt_parts_spies)
apply (blast dest: Notes_Crypt_parts_spies)
```

ClientAccepts and ServerAccepts: because $PMS \notin \text{range PRF}$

```
apply force+
done
```

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|}  $\in$  set evs;
    evs  $\in$  tls; A  $\notin$  bad; B  $\notin$  bad |]
  ==> Nonce (PRF(PMS,NA,NB))  $\notin$  analz (spies evs)"
apply (erule rev_mp, erule tls.induct, frule_tac [7] CX_KB_is_pubKB)
apply (force, simp_all (no_asm_simp))
```

Fake

```
apply spy_analz
```

SpyKeys: by secrecy of the PMS, Spy cannot make the MS

```
apply (blast dest!: Spy_not_see_PMS)
apply (simp_all add: insert_absorb)
```

ClientAccepts and ServerAccepts: because PMS was already visible; others, freshness etc.

```
apply (blast dest: Notes_Crypt_parts_spies Spy_not_see_PMS
        Notes_imp_knows_Spy [THEN analz.Inj])+
done
```

22.3.2 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'"
apply (erule rev_mp, erule rev_mp)
apply (erule tls.induct, frule_tac [7] CX_KB_is_pubKB)
apply (force, simp_all)
```

ClientKeyExch

```
apply (blast dest!: PMS_Crypt_sessionK_not_spied)
```

ClientFinished, ClientResume: by unicity of PMS

```
apply (blast dest!: Notes_master_imp_Notes_PMS
        intro: Notes_unique_PMS [THEN conjunct1])+
done
```

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)"
apply (erule rev_mp, erule rev_mp)
apply (erule tls.induct, frule_tac [7] CX_KB_is_pubKB)
apply (force, simp_all (no_asm_simp))
```

ClientKeyExch

```
apply blast
```

SpyKeys

```
apply (blast dest!: Spy_not_see_MS)
```

ClientKeyExch

```
apply (blast dest!: PMS_sessionK_not_spied)
```

Oops

```
apply (blast intro: Says_clientK_unique)
done
```

22.3.3 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'"
apply (erule rev_mp, erule rev_mp)
apply (erule tls.induct, frule_tac [7] CX_KB_is_pubKB)
apply (force, simp_all)
```

ClientKeyExch

```
apply (blast dest!: PMS_Crypt_sessionK_not_spied)
```

ServerResume, ServerFinished: by unicity of PMS

```
apply (blast dest!: Notes_master_imp_Crypt_PMS
  dest: Spy_not_see_PMS Notes_Crypt_parts_spies Crypt_unique_PMS)+
done
```

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)"
apply (erule rev_mp, erule rev_mp)
apply (erule tls.induct, frule_tac [7] CX_KB_is_pubKB)
apply (force, simp_all (no_asm_simp))
```

Fake

```
apply blast
```

SpyKeys

```
apply (blast dest!: Spy_not_see_MS)
```

ClientKeyExch

```
apply (blast dest!: PMS_sessionK_not_spied)
```

Oops

```
apply (blast intro: Says_serverK_unique)
done
```

22.3.4 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"
apply (erule ssubst)+
apply (erule tls.induct, frule_tac [7] CX_KB_is_pubKB)
apply (force, simp_all (no_asm_simp))

```

Fake: the Spy doesn't have the critical session key!

```
apply (blast dest: serverK_not_spied)
```

ClientKeyExch

```
apply (blast dest!: PMS_Crypt_sessionK_not_spied)
done
```

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)"
apply (erule ssubst)
apply (erule tls.induct, frule_tac [7] CX_KB_is_pubKB)
apply (force, simp_all (no_asm_simp) add: ex_disj_distrib)

```

Fake: the Spy doesn't have the critical session key!

```
apply (blast dest: serverK_not_spied)
```

ClientKeyExch

```
apply (clarify, blast dest!: PMS_Crypt_sessionK_not_spied)
```

ServerResume, ServerFinished: by unicity of PMS

```
apply (blast dest!: Notes_master_imp_Crypt_PMS
  dest: Spy_not_see_PMS Notes_Crypt_parts_spies Crypt_unique_PMS)+
done
```

22.3.5 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"
apply (erule ssubst)
apply (erule tls.induct, frule_tac [7] CX_KB_is_pubKB)
apply (force, simp_all (no_asm_simp))
```

Fake: the Spy doesn't have the critical session key!

```
apply (blast dest: clientK_not_spied)
```

ClientKeyExch

```
apply (blast dest!: PMS_Crypt_sessionK_not_spied)
```

ClientFinished, ClientResume: by unicity of PMS

```
apply (blast dest!: Notes_master_imp_Notes_PMS dest: Notes_unique_PMS)+
done
```

22.3.6 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"
by (blast intro!: TrustClientMsg UseCertVerify)
```

end

23 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.

definition "*response*" :: "agent => agent => nat => msg" where
 "*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"
apply (intro exI bexI)
apply (rule_tac [2] certified_mail.Nil
  [THEN certified_mail.CM1, THEN certified_mail.Reception,
  THEN certified_mail.CM2,
  THEN certified_mail.CM3])
apply (possibility, auto)
done

lemma Gets_imp_Says:
  "[| Gets B X ∈ set evs; evs ∈ certified_mail |] ==> ∃ A. Says A B X ∈ set
  evs"

```

```

apply (erule rev_mp)
apply (erule certified_mail.induct, auto)
done

```

```

lemma Gets_imp_parts_knows_Spy:
  "[|Gets A X ∈ set evs; evs ∈ certified_mail|] ==> X ∈ parts(spies evs)"
apply (drule Gets_imp_Says, simp)
apply (blast dest: Says_imp_knows_Spy parts.Inj)
done

```

```

lemma CM2_S2TTP_analz_knows_Spy:
  "[|Gets R {|Agent A, Agent B, em, Number A0, Number cleartext,
    Nonce q, S2TTP|} ∈ set evs;
    evs ∈ certified_mail|]
  ==> S2TTP ∈ analz(spies evs)"
apply (drule Gets_imp_Says, simp)
apply (blast dest: Says_imp_knows_Spy analz.Inj)
done

```

```

lemmas CM2_S2TTP_parts_knows_Spy =
  CM2_S2TTP_analz_knows_Spy [THEN analz_subset_parts [THEN subsetD]]

```

```

lemma hr_form_lemma [rule_format]:
  "evs ∈ certified_mail
  ==> hr ∉ synth (analz (spies evs)) -->
  (∀S2TTP. Notes TTP {|Agent R, Agent TTP, S2TTP, pwd, hr|}
  ∈ set evs -->
  (∃clt q S em. hr = Hash {|Number clt, Nonce q, response S R q, em|}))"
apply (erule certified_mail.induct)
apply (synth_analz_mono_contra, simp_all, blast+)
done

```

Cannot strengthen the first disjunct to $R \neq \text{Spy}$ because the fakessl rule allows Spy to spoof the sender's name. Maybe can strengthen the second disjunct with $R \neq \text{Spy}$.

```

lemma hr_form:
  "[|Notes TTP {|Agent R, Agent TTP, S2TTP, pwd, hr|} ∈ set evs;
    evs ∈ certified_mail|]
  ==> hr ∈ synth (analz (spies evs)) |
  (∃clt q S em. hr = Hash {|Number clt, Nonce q, response S R q, em|})"
by (blast intro: hr_form_lemma)

```

```

lemma Spy_dont_know_private_keys [dest!]:
  "[|Key (privateKey b A) ∈ parts (spies evs); evs ∈ certified_mail|]
  ==> A ∈ bad"
apply (erule rev_mp)
apply (erule certified_mail.induct, simp_all)

```

Fake

```

apply (blast dest: Fake_parts_insert_in_Un)

```

Message 1

```

apply blast

```

Message 3

```

apply (frule_tac hr_form, assumption)
apply (elim disjE exE)
apply (simp_all add: parts_insert2)
  apply (force dest!: parts_insert_subset_Un [THEN [2] rev_subsetD]
        analz_subset_parts [THEN subsetD], blast)
done

lemma Spy_know_private_keys_iff [simp]:
  "evs ∈ certified_mail
   ==> (Key (privateKey b A) ∈ parts (spies evs)) = (A ∈ bad)"
by blast

lemma Spy_dont_know_TTPKey_parts [simp]:
  "evs ∈ certified_mail ==> Key (privateKey b TTP) ∉ parts(spies evs)"

by simp

lemma Spy_dont_know_TTPKey_analz [simp]:
  "evs ∈ certified_mail ==> Key (privateKey b TTP) ∉ analz(spies evs)"
by auto

```

Thus, prove any goal that assumes that *Spy* knows a private key belonging to *TTP*

```
declare Spy_dont_know_TTPKey_parts [THEN [2] rev_notE, elim!]
```

```

lemma CM3_k_parts_knows_Spy:
  "[| evs ∈ certified_mail;
     Notes TTP {|Agent A, Agent TTP,
                Crypt (pubEK TTP) {|Agent S, Number A0, Key K,
                Agent R, hs|}, Key (RPwd R), hs|} ∈ set evs|]
   ==> Key K ∈ parts(spies evs)"
apply (rotate_tac 1)
apply (erule rev_mp)
apply (erule certified_mail.induct, simp_all)
  apply (blast intro:parts_insertI)

```

Fake SSL

```
apply (blast dest: parts.Body)
```

Message 2

```
apply (blast dest!: Gets_imp_Says elim!: knows_Spy_partsEs)
```

Message 3

```

apply (metis parts_insertI)
done

```

```

lemma Spy_dont_know_RPwd [rule_format]:
  "evs ∈ certified_mail ==> Key (RPwd A) ∈ parts(spies evs) --> A ∈ bad"
apply (erule certified_mail.induct, simp_all)

```

Fake

```

apply (blast dest: Fake_parts_insert_in_Un)

Message 1

apply blast

Message 3

apply (frule CM3_k_parts_knows_Spy, assumption)
apply (frule_tac hr_form, assumption)
apply (elim disjE exE)
apply (simp_all add: parts_insert2)
apply (force dest!: parts_insert_subset_Un [THEN [2] rev_subsetD]
        analz_subset_parts [THEN subsetD])
done

lemma Spy_know_RPw_iff [simp]:
  "evs ∈ certified_mail ==> (Key (RPw A) ∈ parts(spies evs)) = (A∈bad)"
by (auto simp add: Spy_dont_know_RPw)

lemma Spy_analz_RPw_iff [simp]:
  "evs ∈ certified_mail ==> (Key (RPw A) ∈ analz(spies evs)) = (A∈bad)"
by (metis Spy_know_RPw_iff Spy_spies_bad_shrK analz.Inj analz_into_parts)

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"
apply (erule rev_mp)
apply (erule certified_mail.induct, simp_all)

Fake

apply (blast dest: Spy_dont_know_private_keys Fake_parts_insert_in_Un)

Message 1

apply blast

Message 3

apply (frule_tac hr_form, assumption)
apply (elim disjE exE)
apply (simp_all add: parts_insert2 parts_insert_knows_A)
apply (blast dest!: Fake_parts_sing_imp_Un, blast)
done

```

23.1 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))"
apply (erule certified_mail.induct)
apply (drule_tac [6] A=TTP in symKey_neq_priEK)
apply (erule_tac [6] disjE [OF hr_form])

```

```

apply (drule_tac [5] CM2_S2TTP_analz_knows_Spy)
prefer 9
apply (elim exE)
apply (simp_all add: synth_analz_insert_eq
           subset_trans [OF _ subset_insertI]
           subset_trans [OF _ Un_upper2]
           del: image_insert image_Un add: analz_image_freshK_simps)
done

```

```

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))"
by (simp only: analz_image_freshK analz_image_freshK_simps)

```

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"
by (blast dest!: Notes_imp_used)

```

```

lemma S2TTP_sender_lemma [rule_format]:
  "evs ∈ certified_mail ==>
   Key K ∉ analz (spies evs) -->
   (∃ A0. Crypt (pubEK TTP)
    {|Agent S, Number A0, 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 A0,
     Number ctxt, Nonce q,
     Crypt (pubEK TTP)
     {|Agent S, Number A0, Key K, Agent R, hs |}|} ∈ set evs))"
apply (erule certified_mail.induct, analz_mono_contra)
apply (drule_tac [5] CM2_S2TTP_parts_knows_Spy, simp)
apply (simp add: used_Nil Crypt_notin_initState, simp_all)

```

Fake

```

apply (blast dest: Fake_parts_sing [THEN subsetD]
         dest!: analz_subset_parts [THEN subsetD])

```

Fake SSL

```

apply (blast dest: Fake_parts_sing [THEN subsetD]
         dest: analz_subset_parts [THEN subsetD])

```

Message 1

```

apply (clarsimp, blast)

```

Message 2

```

apply (simp add: parts_insert2, clarify)
apply (metis parts_cut Un_empty_left usedI)

```

Message 3

```

apply (blast dest: Notes_SSL_imp_used used_parts_subset_parts)
done

```

lemma S2TTP_sender:

```

"[/Crypt (pubEK TTP) {/Agent S, Number A0, 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 A0,
   Number ctxt, Nonce q,
   Crypt (pubEK TTP)
   {/Agent S, Number A0, Key K, Agent R, hs /}]} ∈ set evs"
by (blast intro: S2TTP_sender_lemma)

```

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))"
apply (erule rev_mp)
apply (erule certified_mail.induct, simp_all)

```

Fake

```

apply (force dest!: keysFor_parts_insert)

```

Message 1

```

apply blast

```

Message 3

```

apply (frule CM3_k_parts_knows_Spy, assumption)
apply (frule_tac hr_form, assumption)
apply (force dest!: keysFor_parts_insert)
done

```

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 A0,
    Number cleartext, Nonce q,
    Crypt (pubEK TTP) {/Agent S, Number A0, Key K, Agent R, hs /}]}
   ∈ set evs -->
  (∃ m' cleartext' q' hs'.
   Says S' R'

```

```

      {|Agent S', Agent TTP, Crypt K (Number m'), Number AO',
        Number cleartext', Nonce q',
        Crypt (pubEK TTP) {|Agent S', Number AO', Key K, Agent R', hs'|}|}
    ∈ set evs --> R' = R & S' = S & AO' = AO & hs' = hs)"
apply (erule certified_mail.induct, analz_mono_contra, simp_all)
prefer 2

```

Message 1

```

apply (blast dest!: Says_imp_knows_Spy [THEN parts.Inj] new_keys_not_used
Crypt_imp_keysFor)

```

Fake

```

apply (auto dest!: usedI S2TTP_sender analz_subset_parts [THEN subsetD])
done

```

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"
by (rule Key_unique_lemma, assumption+)

```

23.2 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"
apply (erule rev_mp)
apply (erule ssubst)
apply (erule rev_mp)
apply (erule certified_mail.induct, simp_all)

```

Fake

```

apply spy_analz

```

Fake SSL

apply *spy_analz*

Message 3

```
apply (frule_tac hr_form, assumption)
apply (elim disjE exE)
apply (simp_all add: synth_analz_insert_eq
             subset_trans [OF _ subset_insertI]
             subset_trans [OF _ Un_upper2]
             del: image_insert image_Un add: analz_image_freshK_simps)
apply (simp_all add: symKey_neq_priEK analz_insert_freshK)
apply (blast dest: Notes_SSL_imp_used S2TTP_sender Key_unique)+
done
```

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)"
```

by (blast dest: *S_fairness_bad_R*)

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"
```

apply (erule rev_mp)

apply (erule ssubst)

apply (erule rev_mp)

apply (erule certified_mail.induct, simp_all)

Message 1

apply (blast dest: *Notes_imp_used*)

Message 3

apply (blast dest: *Notes_SSL_imp_used S2TTP_sender Key_unique S_fairness_bad_R*)

done

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"
  apply (erule rev_mp)
  apply (erule ssubst)
  apply (erule ssubst)
  apply (erule certified_mail.induct, simp_all)

Fake

  apply (blast dest: Fake_parts_sing [THEN subsetD]
        dest!: analz_subset_parts [THEN subsetD])

Fake SSL

  apply (blast dest: Fake_parts_sing [THEN subsetD]
        dest!: analz_subset_parts [THEN subsetD])

Message 2

  apply (drule CM2_S2TTP_parts_knows_Spy, assumption)
  apply (force dest: parts_cut)

Message 3

  apply (frule_tac hr_form, assumption)
  apply (elim disjE exE, simp_all)
  apply (blast dest: Fake_parts_sing [THEN subsetD]
        dest!: analz_subset_parts [THEN subsetD])
done
end

```

24 Conventional protocols: rely on conventional Message, Event and Public – Public-key protocols

```

theory Auth_Public
imports
  "NS_Public_Bad"
  "NS_Public"
  "TLS"
  "CertifiedEmail"
begin

end

```

25 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"
  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"
  apply (rule exI [of _ "{Spy}"], simp) done

```

```

specification (stolen)

  Card_Server_not_stolen [iff]: "Card Server ∉ stolen"
  Card_Spy_not_stolen [iff]: "Card Spy ∉ stolen"
  apply blast done

```

```

specification (cloned)

  Card_Server_not_cloned [iff]: "Card Server ∉ cloned"
  Card_Spy_not_cloned [iff]: "Card Spy ∉ cloned"
  apply blast done

```

```

primrec
  knows :: "agent => event list => msg set" where
  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)"

```

primrec

```

used :: "event list => msg set" where
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"
apply (induct_tac evs)
apply (auto split: event.split)
done

```

```

lemma Says_imp_used [rule_format]: "Says A B X ∈ set evs → X ∈ used evs"
apply (induct_tac evs)
apply (auto split: event.split)
done

```

```

lemma MPair_used [rule_format]:
    "MPair X Y ∈ used evs → X ∈ used evs & Y ∈ used evs"
apply (induct_tac evs)
apply (auto split: event.split)
done

```

25.1 Function knows

lemmas parts_insert_knows_A = parts_insert [of _ "knows A evs"] for A evs

```
lemma knows_Spy_Says [simp]:
  "knows Spy (Says A B X # evs) = insert X (knows Spy evs)"
by simp
```

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)"
by simp
```

```
lemma knows_Spy_Gets [simp]: "knows Spy (Gets A X # evs) = knows Spy evs"
by simp
```

```
lemma knows_Spy_Inputs_secureM [simp]:
  "secureM  $\implies$  knows Spy (Inputs A C X # evs) =
    (if A=Spy then insert X (knows Spy evs) else knows Spy evs)"
by simp
```

```
lemma knows_Spy_Inputs_insecureM [simp]:
  "insecureM  $\implies$  knows Spy (Inputs A C X # evs) = insert X (knows Spy evs)"
by simp
```

```
lemma knows_Spy_C_Gets [simp]: "knows Spy (C_Gets C X # evs) = knows Spy
evs"
by simp
```

```
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)"
by simp
```

```
lemma knows_Spy_Outpts_insecureM [simp]:
  "insecureM  $\implies$  knows Spy (Outpts C A X # evs) = insert X (knows Spy
evs)"
by simp
```

```
lemma knows_Spy_A_Gets [simp]: "knows Spy (A_Gets A X # evs) = knows Spy
evs"
by simp
```

```
lemma knows_Spy_subset_knows_Spy_Says:
  "knows Spy evs  $\subseteq$  knows Spy (Says A B X # evs)"
by (simp add: subset_insertI)
```

```
lemma knows_Spy_subset_knows_Spy_Notes:
  "knows Spy evs  $\subseteq$  knows Spy (Notes A X # evs)"
```

by force

lemma knows_Spy_subset_knows_Spy_Gets:
 "knows Spy evs \subseteq knows Spy (Gets A X # evs)"
 by (simp add: subset_insertI)

lemma knows_Spy_subset_knows_Spy_Inputs:
 "knows Spy evs \subseteq knows Spy (Inputs A C X # evs)"
 by auto

lemma knows_Spy_equals_knows_Spy_Gets:
 "knows Spy evs = knows Spy (C_Gets C X # evs)"
 by (simp add: subset_insertI)

lemma knows_Spy_subset_knows_Spy_Outpts: "knows Spy evs \subseteq knows Spy (Outpts
 C A X # evs)"
 by auto

lemma knows_Spy_subset_knows_Spy_A_Gets: "knows Spy evs \subseteq knows Spy (A_Gets
 A X # evs)"
 by (simp add: subset_insertI)

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"
 apply (induct_tac "evs")
 apply (simp_all (no_asm_simp) split add: event.split)
 done

lemma Notes_imp_knows_Spy [rule_format]:
 "Notes A X \in set evs \longrightarrow A \in bad \longrightarrow X \in knows Spy evs"
 apply (induct_tac "evs")
 apply (simp_all (no_asm_simp) split add: event.split)
 done

lemma Inputs_imp_knows_Spy_secureM [rule_format (no_asm)]:
 "Inputs Spy C X \in set evs \longrightarrow secureM \longrightarrow X \in knows Spy evs"
 apply (induct_tac "evs")
 apply (simp_all (no_asm_simp) split add: event.split)
 done

lemma Inputs_imp_knows_Spy_insecureM [rule_format (no_asm)]:
 "Inputs A C X \in set evs \longrightarrow insecureM \longrightarrow X \in knows Spy evs"
 apply (induct_tac "evs")
 apply (simp_all (no_asm_simp) split add: event.split)
 done

lemma Outpts_imp_knows_Spy_secureM [rule_format (no_asm)]:
 "Outpts C Spy X \in set evs \longrightarrow secureM \longrightarrow X \in knows Spy evs"
 apply (induct_tac "evs")

```

apply (simp_all (no_asm_simp) split add: event.split)
done

```

```

lemma Outpts_imp_knows_Spy_insecureM [rule_format (no_asm)]:
  "Outpts C A X ∈ set evs → insecureM → X ∈ knows Spy evs"
apply (induct_tac "evs")
apply (simp_all (no_asm_simp) split add: event.split)
done

```

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, elim_format]
  parts.Body [elim_format]

```

25.2 Knowledge of Agents

```

lemma knows_Says: "knows A (Says A B X # evs) = insert X (knows A evs)"
by simp

```

```

lemma knows_Notes: "knows A (Notes A X # evs) = insert X (knows A evs)"
by simp

```

```

lemma knows_Gets:
  "A ≠ Spy → knows A (Gets A X # evs) = insert X (knows A evs)"
by simp

```

```

lemma knows_Inputs: "knows A (Inputs A C X # evs) = insert X (knows A evs)"
by simp

```

```

lemma knows_C_Gets: "knows A (C_Gets C X # evs) = knows A evs"
by simp

```

```

lemma knows_Outpts_secureM:
  "secureM → knows A (Outpts C A X # evs) = insert X (knows A evs)"
by simp

```

```

lemma knows_Outpts_insecureM:
  "insecureM → knows Spy (Outpts C A X # evs) = insert X (knows Spy
  evs)"
by simp

```

```

lemma knows_subset_knows_Says: "knows A evs ⊆ knows A (Says A' B X # evs)"
by (simp add: subset_insertI)

```

```

lemma knows_subset_knows_Notes: "knows A evs ⊆ knows A (Notes A' X # evs)"
by (simp add: subset_insertI)

```

```

lemma knows_subset_knows_Gets: "knows A evs ⊆ knows A (Gets A' X # evs)"
by (simp add: subset_insertI)

```

```

lemma knows_subset_knows_Inputs: "knows A evs  $\subseteq$  knows A (Inputs A' C X #
evs)"
by (simp add: subset_insertI)

```

```

lemma knows_subset_knows_C_Gets: "knows A evs  $\subseteq$  knows A (C_Gets C X # evs)"
by (simp add: subset_insertI)

```

```

lemma knows_subset_knows_Outpts: "knows A evs  $\subseteq$  knows A (Outpts C A' X #
evs)"
by (simp add: subset_insertI)

```

```

lemma knows_subset_knows_A_Gets: "knows A evs  $\subseteq$  knows A (A_Gets A' X # evs)"
by (simp add: subset_insertI)

```

Agents know what they say

```

lemma Says_imp_knows [rule_format]: "Says A B X  $\in$  set evs  $\longrightarrow$  X  $\in$  knows
A evs"
apply (induct_tac "evs")
apply (simp_all (no_asm_simp) split add: event.split)
apply blast
done

```

Agents know what they note

```

lemma Notes_imp_knows [rule_format]: "Notes A X  $\in$  set evs  $\longrightarrow$  X  $\in$  knows
A evs"
apply (induct_tac "evs")
apply (simp_all (no_asm_simp) split add: event.split)
apply blast
done

```

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"
apply (induct_tac "evs")
apply (simp_all (no_asm_simp) split add: event.split)
done

```

```

lemma Inputs_imp_knows_agents [rule_format (no_asm)]:
  "Inputs A (Card A) X  $\in$  set evs  $\longrightarrow$  X  $\in$  knows A evs"
apply (induct_tac "evs")
apply (simp_all (no_asm_simp) split add: event.split)
apply blast
done

```

```

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"
apply (induct_tac "evs")
apply (simp_all (no_asm_simp) split add: event.split)

```

done

```
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"
apply (induct_tac "evs")
apply (simp_all (no_asm_simp) split add: event.split)
done
```

```
lemma parts_knows_Spy_subset_used: "parts (knows Spy evs)  $\subseteq$  used evs"
apply (induct_tac "evs", force)
apply (simp add: parts_insert_knows_A knows_Cons add: event.split, blast)
```

done

```
lemmas usedI = parts_knows_Spy_subset_used [THEN subsetD, intro]
```

```
lemma initState_into_used: "X  $\in$  parts (initState B)  $\implies$  X  $\in$  used evs"
apply (induct_tac "evs")
apply (simp_all add: parts_insert_knows_A split add: event.split, blast)
done
```

```
lemma used_Says [simp]: "used (Says A B X # evs) = parts{X}  $\cup$  used evs"
by simp
```

```
lemma used_Notes [simp]: "used (Notes A X # evs) = parts{X}  $\cup$  used evs"
by simp
```

```
lemma used_Gets [simp]: "used (Gets A X # evs) = used evs"
by simp
```

```
lemma used_Inputs [simp]: "used (Inputs A C X # evs) = parts{X}  $\cup$  used evs"
by simp
```

```
lemma used_C_Gets [simp]: "used (C_Gets C X # evs) = used evs"
by simp
```

```
lemma used_Outpts [simp]: "used (Outpts C A X # evs) = parts{X}  $\cup$  used evs"
by simp
```

```
lemma used_A_Gets [simp]: "used (A_Gets A X # evs) = used evs"
by simp
```

```
lemma used_nil_subset: "used []  $\subseteq$  used evs"
apply simp
apply (blast intro: initState_into_used)
done
```

```

lemma Says_parts_used [rule_format (no_asm)]:
  "Says A B X ∈ set evs → (parts {X}) ⊆ used evs"
apply (induct_tac "evs")
apply (simp_all (no_asm_simp) split add: event.split)
apply blast
done

```

```

lemma Notes_parts_used [rule_format (no_asm)]:
  "Notes A X ∈ set evs → (parts {X}) ⊆ used evs"
apply (induct_tac "evs")
apply (simp_all (no_asm_simp) split add: event.split)
apply blast
done

```

```

lemma Outpts_parts_used [rule_format (no_asm)]:
  "Outpts C A X ∈ set evs → (parts {X}) ⊆ used evs"
apply (induct_tac "evs")
apply (simp_all (no_asm_simp) split add: event.split)
apply blast
done

```

```

lemma Inputs_parts_used [rule_format (no_asm)]:
  "Inputs A C X ∈ set evs → (parts {X}) ⊆ used evs"
apply (induct_tac "evs")
apply (simp_all (no_asm_simp) split add: event.split)
apply blast
done

```

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)"
by (induct e, auto simp: knows_Cons)

```

```

lemma initState_subset_knows: "initState A ⊆ knows A evs"
apply (induct_tac evs, simp)
apply (blast intro: knows_subset_knows_Cons [THEN subsetD])
done

```

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"
by (force
  dest!: parts_insert_subset_Un [THEN keysFor_mono, THEN [2] rev_subsetD]
  analz_subset_parts [THEN keysFor_mono, THEN [2] rev_subsetD]
  intro: analz_subset_parts [THEN subsetD] parts_mono [THEN [2] rev_subsetD])
end

```

```

theory All_Symmetric

```

```

imports Message
begin

All keys are symmetric

defs all_symmetric_def: "all_symmetric  $\equiv$  True"

lemma isSym_keys: "K  $\in$  symKeys"
  by (simp add: symKeys_def all_symmetric_def invKey_symmetric)

end

```

26 Theory of smartcards

```

theory Smartcard
imports EventSC "../All_Symmetric"
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.

axiomatization

```

  shrK    :: "agent => key" and
  crdK    :: "card  => key" and
  pin     :: "agent => key" and

  Pairkey :: "agent * agent => nat" and
  pairK   :: "agent * agent => key"

where
  inj_shrK: "inj shrK" and — No two smartcards store the same key
  inj_crdK: "inj crdK" and — Nor do two cards
  inj_pin  : "inj pin" and  — Nor do two agents have the same pin

  inj_pairK [iff]: "(pairK(A,B) = pairK(A',B')) = (A = A' & B = B')" and
  comm_Pairkey [iff]: "Pairkey(A,B) = Pairkey(B,A)" and

  pairK_disj_crdK [iff]: "pairK(A,B)  $\neq$  crdK C" and
  pairK_disj_shrK [iff]: "pairK(A,B)  $\neq$  shrK P" and
  pairK_disj_pin [iff]: "pairK(A,B)  $\neq$  pin P" and
  shrK_disj_crdK [iff]: "shrK P  $\neq$  crdK C" and
  shrK_disj_pin [iff]: "shrK P  $\neq$  pin Q" and
  crdK_disj_pin [iff]: "crdK C  $\neq$  pin P"

definition legalUse :: "card => bool" ("legalUse (_)") where
  "legalUse C == C  $\notin$  stolen"

primrec illegalUse :: "card => bool" where

```

```
illegalUse_def: "illegalUse (Card A) = ( (Card A ∈ stolen ∧ A ∈ bad) ∨
Card A ∈ cloned )"

```

initState must be defined with care

overloading

```
initState ≡ initState

```

begin

primrec initState where

```
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}))"

```

end

Still relying on axioms

axiomatization where

```
Key_supply_ax: "finite KK ⇒ ∃ K. K ∉ KK & Key K ∉ used evs" and

```

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

```

26.1 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"
apply (insert isSym_keys)
apply (simp add: symKeys_def)
done

```

```
lemma analz_Decrypt' [dest]:
  "[ Crypt K X ∈ analz H; Key K ∈ analz H ] ⇒ X ∈ analz H"
by auto

```

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"
apply auto
apply (erule parts.induct)
apply auto
done
```

```
lemma keysFor_parts_initState [simp]: "keysFor (parts (initState C)) = {}"
apply (unfold keysFor_def)
apply (induct_tac "C", auto)
done
```

```
lemma keysFor_parts_insert:
  "[[ K ∈ keysFor (parts (insert X G)); X ∈ synth (analz H) ]]
  ⇒ K ∈ keysFor (parts (G ∪ H)) | Key K ∈ parts H"
by (force dest: EventSC.keysFor_parts_insert)
```

```
lemma Crypt_imp_keysFor: "Crypt K X ∈ H ⇒ K ∈ keysFor H"
by (drule Crypt_imp_invKey_keysFor, simp)
```

26.2 Function "knows"

```
lemma Spy_knows_bad [intro!]: "A ∈ bad ⇒ Key (pin A) ∈ knows Spy evs"
apply (induct_tac "evs")
apply (simp_all (no_asm_simp) add: imageI knows_Cons split add: event.split)
done
```

```
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)"
apply (induct_tac "evs")
apply (simp_all (no_asm_simp) add: imageI knows_Cons split add: event.split)
done
```

```
lemma Spy_knows_cloned1 [intro!]: "C ∈ cloned ⇒ Key (crdK C) ∈ knows Spy
evs"
apply (induct_tac "evs")
apply (simp_all (no_asm_simp) add: imageI knows_Cons split add: event.split)
done
```

```
lemma Spy_knows_cloned2 [intro!]: "[[ Card A ∈ cloned; Card B ∈ cloned ]]
  ⇒ Nonce (Pairkey(A,B)) ∈ knows Spy evs"
apply (induct_tac "evs")
apply (simp_all (no_asm_simp) add: imageI knows_Cons split add: event.split)
done
```

```

lemma Spy_knows_Spy_bad [intro!]: "A ∈ bad ⇒ Key (pin A) ∈ knows Spy evs"
apply (induct_tac "evs")
apply (simp_all (no_asm_simp) add: imageI knows_Cons split add: event.split)
done

```

```

lemma Crypt_Spy_analz_bad:
  "[ Crypt (pin A) X ∈ analz (knows Spy evs); A ∈ bad ]
  ⇒ X ∈ analz (knows Spy evs)"
apply (force dest!: analz.Decrypt)
done

```

```

lemma shrK_in_initState [iff]: "Key (shrK A) ∈ initState Server"
apply (induct_tac "A")
apply auto
done

```

```

lemma shrK_in_used [iff]: "Key (shrK A) ∈ used evs"
apply (rule initState_into_used)
apply blast
done

```

```

lemma crdK_in_initState [iff]: "Key (crdK A) ∈ initState Server"
apply (induct_tac "A")
apply auto
done

```

```

lemma crdK_in_used [iff]: "Key (crdK A) ∈ used evs"
apply (rule initState_into_used)
apply blast
done

```

```

lemma pin_in_initState [iff]: "Key (pin A) ∈ initState A"
apply (induct_tac "A")
apply auto
done

```

```

lemma pin_in_used [iff]: "Key (pin A) ∈ used evs"
apply (rule initState_into_used)
apply blast
done

```

```

lemma pairK_in_initState [iff]: "Key (pairK X) ∈ initState Server"
apply (induct_tac "X")
apply auto
done

```

```

lemma pairK_in_used [iff]: "Key (pairK X) ∈ used evs"
apply (rule initState_into_used)
apply blast

```

done

```
lemma Key_not_used [simp]: "Key K ∉ used evs ⇒ K ∉ range shrK"
by blast
```

```
lemma shrK_neq [simp]: "Key K ∉ used evs ⇒ shrK B ≠ K"
by blast
```

```
lemma crdK_not_used [simp]: "Key K ∉ used evs ⇒ K ∉ range crdK"
apply clarify
done
```

```
lemma crdK_neq [simp]: "Key K ∉ used evs ⇒ crdK C ≠ K"
apply clarify
done
```

```
lemma pin_not_used [simp]: "Key K ∉ used evs ⇒ K ∉ range pin"
apply clarify
done
```

```
lemma pin_neq [simp]: "Key K ∉ used evs ⇒ pin A ≠ K"
apply clarify
done
```

```
lemma pairK_not_used [simp]: "Key K ∉ used evs ⇒ K ∉ range pairK"
apply clarify
done
```

```
lemma pairK_neq [simp]: "Key K ∉ used evs ⇒ pairK(A,B) ≠ K"
apply clarify
done
```

```
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]
```

26.3 Fresh nonces

```
lemma Nonce_notin_initState [iff]: "Nonce N ∉ parts (initState (Friend i))"
by auto
```

26.4 Supply fresh nonces for possibility theorems.

```
lemma Nonce_supply1: "∃N. Nonce N ∉ used evs"
apply (rule finite.emptyI [THEN Nonce_supply_ax, THEN exE], blast)
done
```

```
lemma Nonce_supply2:
  "∃N N'. Nonce N ∉ used evs & Nonce N' ∉ used evs' & N ≠ N'"
apply (cut_tac evs = evs in finite.emptyI [THEN Nonce_supply_ax])
```

```

apply (erule exE)
apply (cut_tac evs = evs' in finite.emptyI [THEN finite.insertI, THEN Nonce_supply_ax])

```

```

apply auto
done

```

```

lemma Nonce_supply3: " $\exists N N' N''$ . Nonce  $N \notin$  used evs & Nonce  $N' \notin$  used evs'
&

```

```

      Nonce  $N'' \notin$  used evs'' &  $N \neq N'$  &  $N' \neq N''$  &  $N \neq N''$ "

```

```

apply (cut_tac evs = evs in finite.emptyI [THEN Nonce_supply_ax])
apply (erule exE)
apply (cut_tac evs = evs' and a1 = N in finite.emptyI [THEN finite.insertI,
THEN Nonce_supply_ax])
apply (erule exE)
apply (cut_tac evs = evs'' and a1 = Na and a2 = N in finite.emptyI [THEN
finite.insertI, THEN finite.insertI, THEN Nonce_supply_ax])
apply blast
done

```

```

lemma Nonce_supply: "Nonce ( $\exists N$ . Nonce  $N \notin$  used evs)  $\notin$  used evs"

```

```

apply (rule finite.emptyI [THEN Nonce_supply_ax, THEN exE])
apply (rule someI, blast)
done

```

Unlike the corresponding property of nonces, we cannot prove $\text{finite } KK \implies \exists K. K \notin KK \wedge \text{Key } K \notin \text{used evs}$. We have infinitely many agents and there is nothing to stop their long-term keys from exhausting all the natural numbers. Instead, possibility theorems must assume the existence of a few keys.

26.5 Specialized Rewriting for Theorems About *analz* and Image

```

lemma subset_Compl_range_shrK: " $A \subseteq -$  (range shrK)  $\implies$  shrK  $x \notin A$ "
by blast

```

```

lemma subset_Compl_range_crdK: " $A \subseteq -$  (range crdK)  $\implies$  crdK  $x \notin A$ "
apply blast
done

```

```

lemma subset_Compl_range_pin: " $A \subseteq -$  (range pin)  $\implies$  pin  $x \notin A$ "
apply blast
done

```

```

lemma subset_Compl_range_pairK: " $A \subseteq -$  (range pairK)  $\implies$  pairK  $x \notin A$ "
apply blast
done

```

```

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

```

```

lemma insert_Key_image: "insert (Key K) (Key'KK  $\cup$  C) = Key'(insert K KK)
 $\cup$  C"
by blast

```

```

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:
  "(Key K ∈ analz (Key'nE ∪ H)) → (K ∈ nE | Key K ∈ analz H) ⇒
   (Key K ∈ analz (Key'nE ∪ H)) = (K ∈ nE | Key K ∈ analz H)"
by (blast intro: analz_mono [THEN [2] rev_subsetD])

```

26.6 Tactics for possibility theorems

ML

```

{*
structure Smartcard =
struct

(*Omitting used_Says makes the tactic much faster: it leaves expressions
such as Nonce ?N ∉ used evs that match Nonce_supply*)
fun possibility_tac ctxt =
  (REPEAT
   (ALLGOALS (simp_tac (simpset_of ctxt
    delsimps [@{thm used_Says}, @{thm used_Notes}, @{thm used_Gets},
    @{thm used_Inputs}, @{thm used_C_Gets}, @{thm used_Outpts}, @{thm used_A_Gets}]

    setSolver safe_solver))
   THEN
   REPEAT_FIRST (eq_assume_tac ORELSE'
    resolve_tac [refl, conjI, @{thm Nonce_supply}])))

(*For harder protocols (such as Recur) where we have to set up some
nonces and keys initially*)
fun basic_possibility_tac ctxt =
  REPEAT
  (ALLGOALS (asm_simp_tac (simpset_of ctxt setSolver safe_solver))
   THEN
   REPEAT_FIRST (resolve_tac [refl, conjI]))

val analz_image_freshK_ss =
  @{simpset} delsimps [image_insert, image_Un]
  delsimps [@{thm imp_disjL}] (*reduces blow-up*)
  addsimps @{thms analz_image_freshK_simps}

end
*}

```

```

lemma invKey_shrK_iff [iff]:
  "(Key (invKey K) ∈ X) = (Key K ∈ X)"
by auto

method_setup analz_freshK = {*
  Scan.succeed (fn ctxt =>
    (SIMPLE_METHOD
      (EVERY [REPEAT_FIRST (resolve_tac [allI, ballI, impI]),
        REPEAT_FIRST (rtac @{thm analz_image_freshK_lemma}),
        ALLGOALS (asm_simp_tac (Simplifier.context ctxt Smartcard.analz_image_freshK_ss))]))
  *)
  "for proving the Session Key Compromise theorem"

method_setup possibility = {*
  Scan.succeed (fn ctxt =>
    SIMPLE_METHOD (Smartcard.possibility_tac ctxt)) *)
  "for proving possibility theorems"

method_setup basic_possibility = {*
  Scan.succeed (fn ctxt =>
    SIMPLE_METHOD (Smartcard.basic_possibility_tac ctxt)) *)
  "for proving possibility theorems"

lemma knows_subset_knows_Cons: "knows A evs ⊆ knows A (e # evs)"
by (induct e) (auto simp: knows_Cons)

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

```

27 Original Shoup-Rubin protocol

```

theory ShoupRubin imports Smartcard begin

axiomatization sesK :: "nat*key => key"
where

  inj_sesK [iff]: "(sesK(m,k) = sesK(m',k')) = (m = m' ∧ k = k')" and

  shrK_disj_sesK [iff]: "shrK A ≠ sesK(m,pk)" and
  crdK_disj_sesK [iff]: "crdK C ≠ sesK(m,pk)" and

```

```
pin_disj_sesK [iff]: "pin P ≠ sesK(m,pk)" and
pairK_disj_sesK[iff]:"pairK(A,B) ≠ sesK(m,pk)" and
```

```
Atomic_distrib [iff]: "Atomic'(KEY'K ∪ NONCE'N) =
Atomic'(KEY'K) ∪ Atomic'(NONCE'N)" and
```

```
shouprubin_assumes_securemeans [iff]: "evs ∈ sr ⇒ secureM"
```

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

```
inductive_set sr :: "event list set"
where
```

```
  Nil: "[] ∈ sr"
```

```
| Fake: "[ evsF ∈ sr; X ∈ synth (analz (knows Spy evsF));
  illegalUse(Card B) ]
  ⇒ Says Spy A X #
  Inputs Spy (Card B) X # evsF ∈ sr"
```

```
| Forge:
  "[ evsFo ∈ sr; Nonce Nb ∈ analz (knows Spy evsFo);
  Key (pairK(A,B)) ∈ knows Spy evsFo ]
  ⇒ Notes Spy (Key (sesK(Nb,pairK(A,B)))) # evsFo ∈ sr"
```

```
| Reception: "[ evsR ∈ sr; Says A B X ∈ set evsR ]
  ⇒ Gets B X # evsR ∈ sr"
```

```
| SR1: "[ evs1 ∈ sr; A ≠ Server ]
  ⇒ Says A Server {Agent A, Agent B}
  # evs1 ∈ sr"
```

```
| SR2: "[ evs2 ∈ sr;
  Gets Server {Agent A, Agent B} ∈ set evs2 ]
  ⇒ Says Server A {Nonce (Pairkey(A,B)),
  Crypt (shrK A) {Nonce (Pairkey(A,B)), Agent B}
  }
  # evs2 ∈ sr"
```

```

/ SR3: "[ evs3 ∈ sr; legalUse(Card A);
        Says A Server {Agent A, Agent B} ∈ set evs3;
        Gets A {Nonce Pk, Certificate} ∈ set evs3 ]
⇒ Inputs A (Card A) (Agent A)
   # evs3 ∈ sr"

/ SR4: "[ evs4 ∈ sr; A ≠ Server;
        Nonce Na ∉ used evs4; legalUse(Card A);
        Inputs A (Card A) (Agent A) ∈ set evs4 ]
⇒ Outputs (Card A) A {Nonce Na, Crypt (crdK (Card A)) (Nonce Na)}
   # evs4 ∈ sr"

/ SR4Fake: "[ evs4F ∈ sr; Nonce Na ∉ used evs4F;
            illegalUse(Card A);
            Inputs Spy (Card A) (Agent A) ∈ set evs4F ]
⇒ Outputs (Card A) Spy {Nonce Na, Crypt (crdK (Card A)) (Nonce Na)}
   # evs4F ∈ sr"

/ SR5: "[ evs5 ∈ sr;
        Outputs (Card A) A {Nonce Na, Certificate} ∈ set evs5;
        ∀ p q. Certificate ≠ {p, q} ]
⇒ Says A B {Agent A, Nonce Na} # evs5 ∈ sr"

/ SR6: "[ evs6 ∈ sr; legalUse(Card B);
        Gets B {Agent A, Nonce Na} ∈ set evs6 ]
⇒ Inputs B (Card B) {Agent A, Nonce Na}
   # evs6 ∈ sr"

/ SR7: "[ evs7 ∈ sr;
        Nonce Nb ∉ used evs7; legalUse(Card B); B ≠ Server;
        K = sesK(Nb, pairK(A,B));
        Key K ∉ used evs7;
        Inputs B (Card B) {Agent A, Nonce Na} ∈ set evs7 ]
⇒ Outputs (Card B) B {Nonce Nb, Key K,
                    Crypt (pairK(A,B)) {Nonce Na, Nonce Nb},
                    Crypt (pairK(A,B)) (Nonce Nb)}
   # evs7 ∈ sr"

```

```

/ SR7Fake: "[ evs7F ∈ sr; Nonce Nb ∉ used evs7F;
            illegalUse(Card B);
            K = sesK(Nb, pairK(A,B));
            Key K ∉ used evs7F;
            Inputs Spy (Card B) {Agent A, Nonce Na} ∈ set evs7F ]
⇒ Outputs (Card B) Spy {Nonce Nb, Key K,
                       Crypt (pairK(A,B)) {Nonce Na, Nonce Nb},
                       Crypt (pairK(A,B)) (Nonce Nb)}
   # evs7F ∈ sr"

/ SR8: "[ evs8 ∈ sr;
          Inputs B (Card B) {Agent A, Nonce Na} ∈ set evs8;
          Outputs (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;
          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 ∈ 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 ]
⇒ Outputs (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 ]
⇒ Outputs (Card A) Spy {Key K, Crypt (pairK(A,B)) (Nonce Nb)}
  # evs10F ∈ sr"

```

```

/ SR11: "[ evs11 ∈ sr;
  Says A Server {Agent A, Agent B} ∈ set evs11;
  Outpts (Card A) A {Key K, Certificate} ∈ set evs11 ]
⇒ Says A B (Certificate)
  # evs11 ∈ sr"

```

```

/ Oops1:
  "[ evs01 ∈ sr;
    Outpts (Card B) B {Nonce Nb, Key K, Certificate,
      Crypt (pairK(A,B)) (Nonce Nb)} ∈ set evs01 ]
⇒ Notes Spy {Key K, Nonce Nb, Agent A, Agent B} # evs01 ∈ sr"

```

```

/ Oops2:
  "[ evs02 ∈ sr;
    Outpts (Card A) A {Key K, Crypt (pairK(A,B)) (Nonce Nb)}
    ∈ set evs02 ]
⇒ Notes Spy {Key K, Nonce Nb, Agent A, Agent B} # evs02 ∈ sr"

```

```

declare Fake_parts_insert_in_Un [dest]
declare analz_into_parts [dest]

```

```

lemma Gets_imp_Says:
  "[[ Gets B X ∈ set evs; evs ∈ sr ]] ⇒ ∃ A. Says A B X ∈ set evs"
apply (erule rev_mp, erule sr.induct)
apply auto
done

lemma Gets_imp_knows_Spy:
  "[[ Gets B X ∈ set evs; evs ∈ sr ]] ⇒ X ∈ knows Spy evs"
apply (blast dest!: Gets_imp_Says Says_imp_knows_Spy)
done

lemma Gets_imp_knows_Spy_parts_Snd:
  "[[ Gets B {X, Y} ∈ set evs; evs ∈ sr ]] ⇒ Y ∈ parts (knows Spy evs)"
apply (blast dest!: Gets_imp_Says Says_imp_knows_Spy parts.Inj parts.Snd)
done

lemma Gets_imp_knows_Spy_analz_Snd:
  "[[ Gets B {X, Y} ∈ set evs; evs ∈ sr ]] ⇒ Y ∈ analz (knows Spy evs)"
apply (blast dest!: Gets_imp_Says Says_imp_knows_Spy analz.Inj analz.Snd)
done

lemma Inputs_imp_knows_Spy_secureM_sr:
  "[[ Inputs Spy C X ∈ set evs; evs ∈ sr ]] ⇒ X ∈ knows Spy evs"
apply (simp (no_asm_simp) add: Inputs_imp_knows_Spy_secureM)
done

lemma knows_Spy_Inputs_secureM_sr_Spy:
  "evs ∈ sr ⇒ knows Spy (Inputs Spy C X # evs) = insert X (knows Spy
evs)"
apply (simp (no_asm_simp))
done

lemma knows_Spy_Inputs_secureM_sr:
  "[[ A ≠ Spy; evs ∈ sr ]] ⇒ knows Spy (Inputs A C X # evs) = knows Spy
evs"
apply (simp (no_asm_simp))
done

lemma knows_Spy_Outpts_secureM_sr_Spy:
  "evs ∈ sr ⇒ knows Spy (Outpts C Spy X # evs) = insert X (knows Spy
evs)"
apply (simp (no_asm_simp))
done

lemma knows_Spy_Outpts_secureM_sr:
  "[[ A ≠ Spy; evs ∈ sr ]] ⇒ knows Spy (Outpts C A X # evs) = knows Spy
evs"

```

```

apply (simp (no_asm_simp))
done

```

```

lemma Inputs_A_Card_3:
  "[[ Inputs A C (Agent A) ∈ set evs; A ≠ Spy; evs ∈ sr ]]
  ⇒ legalUse(C) ∧ C = (Card A) ∧
  (∃ Pk Certificate. Gets A {Pk, Certificate} ∈ set evs)"
apply (erule rev_mp, erule sr.induct)
apply auto
done

```

```

lemma Inputs_B_Card_6:
  "[[ Inputs B C {Agent A, Nonce Na} ∈ set evs; B ≠ Spy; evs ∈ sr ]]
  ⇒ legalUse(C) ∧ C = (Card B) ∧ Gets B {Agent A, Nonce Na} ∈ set
  evs"
apply (erule rev_mp, erule sr.induct)
apply auto
done

```

```

lemma Inputs_A_Card_9:
  "[[ Inputs A C {Agent B, Nonce Na, Nonce Nb, Nonce Pk,
  Cert1, Cert2, Cert3} ∈ set evs;

  A ≠ Spy; evs ∈ sr ]]
  ⇒ 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"
apply (erule rev_mp, erule sr.induct)
apply auto
done

```

```

lemma Outpts_A_Card_4:
  "[[ Outpts C A {Nonce Na, (Crypt (crdK (Card A)) (Nonce Na))} ∈ set evs;

  evs ∈ sr ]]
  ⇒ legalUse(C) ∧ C = (Card A) ∧
  Inputs A (Card A) (Agent A) ∈ set evs"
apply (erule rev_mp, erule sr.induct)
apply auto
done

```

```

lemma Outpts_B_Card_7:
  "[[ Outpts C B {Nonce Nb, Key K,
    Crypt (pairK(A,B)) {Nonce Na, Nonce Nb},
    Cert2} ∈ set evs;
    evs ∈ sr ]]
  ⇒ legalUse(C) ∧ C = (Card B) ∧
    Inputs B (Card B) {Agent A, Nonce Na} ∈ set evs"
apply (erule rev_mp, erule sr.induct)
apply auto
done

lemma Outpts_A_Card_10:
  "[[ Outpts C A {Key K, (Crypt (pairK(A,B)) (Nonce Nb))} ∈ set evs;
    evs ∈ sr ]]
  ⇒ legalUse(C) ∧ C = (Card A) ∧
    (∃ Na Ver1 Ver2 Ver3.
      Inputs A (Card A) {Agent B, Nonce Na, Nonce Nb, Nonce (Pairkey(A,B)),
        Ver1, Ver2, Ver3} ∈ set evs)"
apply (erule rev_mp, erule sr.induct)
apply auto
done

lemma Outpts_A_Card_10_imp_Inputs:
  "[[ Outpts (Card A) A {Key K, Certificate} ∈ set evs; evs ∈ sr ]]
  ⇒ (∃ B Na Nb Ver1 Ver2 Ver3.
    Inputs A (Card A) {Agent B, Nonce Na, Nonce Nb, Nonce (Pairkey(A,B)),
      Ver1, Ver2, Ver3} ∈ set evs)"
apply (erule rev_mp, erule sr.induct)
apply simp_all
apply blast+
done

lemma Outpts_honest_A_Card_4:
  "[[ Outpts C A {Nonce Na, Crypt K X} ∈ set evs;
    A ≠ Spy; evs ∈ sr ]]
  ⇒ legalUse(C) ∧ C = (Card A) ∧
    Inputs A (Card A) (Agent A) ∈ set evs"
apply (erule rev_mp, erule sr.induct)
apply auto
done

lemma Outpts_honest_B_Card_7:

```

```

    "[[ Outputs C B {Nonce Nb, Key K, Cert1, Cert2} ∈ set evs;
      B ≠ Spy; evs ∈ sr ]]
    ⇒ legalUse(C) ∧ C = (Card B) ∧
      (∃ A Na. Inputs B (Card B) {Agent A, Nonce Na} ∈ set evs)"
  apply (erule rev_mp, erule sr.induct)
  apply auto
  done

```

```

lemma Outpts_honest_A_Card_10:
  "[[ Outputs C A {Key K, Certificate} ∈ set evs;
    A ≠ Spy; evs ∈ sr ]]
  ⇒ legalUse (C) ∧ C = (Card A) ∧
    (∃ B Na Nb Pk Ver1 Ver2 Ver3.
      Inputs A (Card A) {Agent B, Nonce Na, Nonce Nb, Pk,
        Ver1, Ver2, Ver3} ∈ set evs)"
  apply (erule rev_mp, erule sr.induct)
  apply simp_all
  apply blast+
  done

```

```

lemma Outpts_which_Card_4:
  "[[ Outputs (Card A) A {Nonce Na, Crypt K X} ∈ set evs; evs ∈ sr ]]
  ⇒ Inputs A (Card A) (Agent A) ∈ set evs"
  apply (erule rev_mp, erule sr.induct)
  apply (simp_all (no_asm_simp))
  apply clarify
  done

```

```

lemma Outpts_which_Card_7:
  "[[ Outputs (Card B) B {Nonce Nb, Key K, Cert1, Cert2} ∈ set evs;
    evs ∈ sr ]]
  ⇒ ∃ A Na. Inputs B (Card B) {Agent A, Nonce Na} ∈ set evs"
  apply (erule rev_mp, erule sr.induct)
  apply auto
  done

```

```

lemma Outpts_which_Card_10:
  "[[ Outputs (Card A) A {Key (sesK(Nb, pairK(A,B))),
    Crypt (pairK(A,B)) (Nonce Nb) } ∈ set evs;
    evs ∈ sr ]]
  ⇒ ∃ 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"
  apply (erule rev_mp, erule sr.induct)
  apply auto
  done

```

```

lemma Outpts_A_Card_form_4:
  "[[ Outpts (Card A) A {Nonce Na, Certificate} ∈ set evs;
    ∃ p q. Certificate ≠ {p, q}; evs ∈ sr ]]
    ⇒ Certificate = (Crypt (crdK (Card A)) (Nonce Na))"
apply (erule rev_mp, erule sr.induct)
apply (simp_all (no_asm_simp))
done

lemma Outpts_B_Card_form_7:
  "[[ Outpts (Card B) B {Nonce Nb, Key K, Cert1, Cert2} ∈ set evs;
    evs ∈ sr ]]
    ⇒ ∃ A Na.
      K = sesK(Nb, pairK(A,B)) ∧
      Cert1 = (Crypt (pairK(A,B)) {Nonce Na, Nonce Nb}) ∧
      Cert2 = (Crypt (pairK(A,B)) (Nonce Nb))"
apply (erule rev_mp, erule sr.induct)
apply auto
done

lemma Outpts_A_Card_form_10:
  "[[ Outpts (Card A) A {Key K, Certificate} ∈ set evs; evs ∈ sr ]]
    ⇒ ∃ B Nb.
      K = sesK(Nb, pairK(A,B)) ∧
      Certificate = (Crypt (pairK(A,B)) (Nonce Nb))"
apply (erule rev_mp, erule sr.induct)
apply (simp_all (no_asm_simp))
done

lemma Outpts_A_Card_form_bis:
  "[[ Outpts (Card A') A' {Key (sesK(Nb, pairK(A,B))), Certificate} ∈ set evs;
    evs ∈ sr ]]
    ⇒ A' = A ∧
      Certificate = (Crypt (pairK(A,B)) (Nonce Nb))"
apply (erule rev_mp, erule sr.induct)
apply (simp_all (no_asm_simp))
done

lemma Inputs_A_Card_form_9:
  "[[ Inputs A (Card A) {Agent B, Nonce Na, Nonce Nb, Nonce Pk,
    Cert1, Cert2, Cert3} ∈ set evs;
    evs ∈ sr ]]
    ⇒ Cert3 = Crypt (crdK (Card A)) (Nonce Na)"
apply (erule rev_mp)
apply (erule sr.induct)
apply (simp_all (no_asm_simp))

apply force

```

```

apply (blast dest!: Outpts_A_Card_form_4)
done

```

```

lemma Inputs_Card_legalUse:
  "[[ Inputs A (Card A) X ∈ set evs; evs ∈ sr ]] ⇒ legalUse(Card A)"
apply (erule rev_mp, erule sr.induct)
apply auto
done

```

```

lemma Outpts_Card_legalUse:
  "[[ Outpts (Card A) A X ∈ set evs; evs ∈ sr ]] ⇒ legalUse(Card A)"
apply (erule rev_mp, erule sr.induct)
apply auto
done

```

```

lemma Inputs_Card: "[[ Inputs A C X ∈ set evs; A ≠ Spy; evs ∈ sr ]]
  ⇒ C = (Card A) ∧ legalUse(C)"
apply (erule rev_mp, erule sr.induct)
apply auto
done

```

```

lemma Outpts_Card: "[[ Outpts C A X ∈ set evs; A ≠ Spy; evs ∈ sr ]]
  ⇒ C = (Card A) ∧ legalUse(C)"
apply (erule rev_mp, erule sr.induct)
apply auto
done

```

```

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)"
apply (blast dest: Inputs_Card Outpts_Card)
done

```

```

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))"
apply (erule rev_mp, erule sr.induct)
apply auto
done

```

```

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'"
apply (erule rev_mp, erule rev_mp, erule sr.induct, simp_all)
apply (fastforce dest: Outpts_parts_used)
apply blast
done

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'"
apply (erule rev_mp, erule rev_mp, erule sr.induct, simp_all)
apply (fastforce dest: Outpts_parts_used)
apply blast
done

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'"
apply (erule rev_mp, erule rev_mp, erule sr.induct, simp_all)
apply (fastforce dest: Outpts_parts_used)
apply blast
done

lemma Outpts_A_Card_unique_key: "[ Outpts (Card A) A {Key K, V} ∈ set evs;

    Outpts (Card A') A' {Key K, V'} ∈ set evs;
    evs ∈ sr ] ⇒ A=A' ∧ V=V'"
apply (erule rev_mp, erule rev_mp, erule sr.induct, simp_all)
apply (blast dest: Outpts_A_Card_form_bis)
apply blast

```

done

lemma *Outpts_A_Card_Unique*:

```
"[[ Outpts (Card A) A {Nonce Na, rest} ∈ set evs; evs ∈ sr ]
  ⇒ Unique (Outpts (Card A) A {Nonce Na, rest}) on evs"
apply (erule rev_mp, erule sr.induct, simp_all add: Unique_def)
apply (fastforce dest: Outpts_parts_used)
apply blast
apply (fastforce dest: Outpts_parts_used)
apply blast
done
```

lemma *Spy_knows_Na*:

```
"[[ Says A B {Agent A, Nonce Na} ∈ set evs; evs ∈ sr ]
  ⇒ Nonce Na ∈ analz (knows Spy evs)"
apply (blast dest!: Says_imp_knows_Spy [THEN analz.Inj, THEN analz.Snd])
done
```

lemma *Spy_knows_Nb*:

```
"[[ Says B A {Nonce Nb, Certificate} ∈ set evs; evs ∈ sr ]
  ⇒ Nonce Nb ∈ analz (knows Spy evs)"
apply (blast dest!: Says_imp_knows_Spy [THEN analz.Inj, THEN analz.Fst])
done
```

lemma *Pairkey_Gets_analz_knows_Spy*:

```
"[[ Gets A {Nonce (Pairkey(A,B)), Certificate} ∈ set evs; evs ∈ sr ]
  ⇒ Nonce (Pairkey(A,B)) ∈ analz (knows Spy evs)"
apply (blast dest!: Gets_imp_knows_Spy [THEN analz.Inj])
done
```

lemma *Pairkey_Inputs_imp_Gets*:

```
"[[ Inputs A (Card A)
  {Agent B, Nonce Na, Nonce Nb, Nonce (Pairkey(A,B)),
   Cert1, Cert3, Cert2} ∈ set evs;
  A ≠ Spy; evs ∈ sr ]
  ⇒ Gets A {Nonce (Pairkey(A,B)), Cert1} ∈ set evs"
apply (erule rev_mp, erule sr.induct)
apply (simp_all (no_asm_simp))
```

```

apply force
done

```

```

lemma Pairkey_Inputs_analz_knows_Spy:
  "[ Inputs A (Card A)
    {Agent B, Nonce Na, Nonce Nb, Nonce (Pairkey(A,B)),
     Cert1, Cert3, Cert2} ∈ set evs;
    evs ∈ sr ]
  ⇒ Nonce (Pairkey(A,B)) ∈ analz (knows Spy evs)"
apply (case_tac "A = Spy")
apply (fastforce dest!: Inputs_imp_knows_Spy_secureM [THEN analz.Inj])
apply (blast dest!: Pairkey_Inputs_imp_Gets [THEN Pairkey_Gets_analz_knows_Spy])
done

```

```

declare shrK_disj_sesK [THEN not_sym, iff]
declare pin_disj_sesK [THEN not_sym, iff]
declare crdK_disj_sesK [THEN not_sym, iff]
declare pairK_disj_sesK [THEN not_sym, iff]

```

ML

```

{*
structure ShoupRubin =
struct

val prepare_tac =
  (*SR8*) forward_tac [@{thm Outpts_B_Card_form_7}] 14 THEN
    eresolve_tac [exE] 15 THEN eresolve_tac [exE] 15 THEN
  (*SR9*) forward_tac [@{thm Outpts_A_Card_form_4}] 16 THEN
  (*SR11*) forward_tac [@{thm Outpts_A_Card_form_10}] 21 THEN
    eresolve_tac [exE] 22 THEN eresolve_tac [exE] 22

fun parts_prepare_tac ctxt =
  prepare_tac THEN
  (*SR9*) dresolve_tac [@{thm Gets_imp_knows_Spy_parts_Snd}] 18 THEN
  (*SR9*) dresolve_tac [@{thm Gets_imp_knows_Spy_parts_Snd}] 19 THEN
  (*Ops1*) dresolve_tac [@{thm Outpts_B_Card_form_7}] 25 THEN
  (*Ops2*) dresolve_tac [@{thm Outpts_A_Card_form_10}] 27 THEN
  (*Base*) (force_tac ctxt) 1

val analz_prepare_tac =
  prepare_tac THEN
  dtac @{thm Gets_imp_knows_Spy_analz_Snd} 18 THEN
  (*SR9*) dtac @{thm Gets_imp_knows_Spy_analz_Snd} 19 THEN
  REPEAT_FIRST (eresolve_tac [asm_rl, conjE] ORELSE' hyp_subst_tac)

```

```

end
*}

method_setup prepare = {*
  Scan.succeed (K (SIMPLE_METHOD ShoupRubin.prepare_tac)) *}
  "to launch a few simple facts that will help the simplifier"

method_setup parts_prepare = {*
  Scan.succeed (fn ctxt => SIMPLE_METHOD (ShoupRubin.parts_prepare_tac ctxt))
  *}
  "additional facts to reason about parts"

method_setup analz_prepare = {*
  Scan.succeed (K (SIMPLE_METHOD ShoupRubin.analz_prepare_tac)) *}
  "additional facts to reason about analz"

lemma Spy_parts_keys [simp]: "evs ∈ sr ⇒
  (Key (shrK P) ∈ parts (knows Spy evs)) = (Card P ∈ cloned) ∧
  (Key (pin P) ∈ parts (knows Spy evs)) = (P ∈ bad ∨ Card P ∈ cloned) ∧

  (Key (crdK C) ∈ parts (knows Spy evs)) = (C ∈ cloned) ∧
  (Key (pairK(A,B)) ∈ parts (knows Spy evs)) = (Card B ∈ cloned)"
apply (erule sr.induct)
apply parts_prepare
apply simp_all
apply (blast intro: parts_insertI)
done

lemma Spy_analz_shrK [simp]: "evs ∈ sr ⇒
  (Key (shrK P) ∈ analz (knows Spy evs)) = (Card P ∈ cloned)"
apply (auto dest!: Spy_knows_cloned)
done

lemma Spy_analz_crdK [simp]: "evs ∈ sr ⇒
  (Key (crdK C) ∈ analz (knows Spy evs)) = (C ∈ cloned)"
apply (auto dest!: Spy_knows_cloned)
done

lemma Spy_analz_pairK [simp]: "evs ∈ sr ⇒
  (Key (pairK(A,B)) ∈ analz (knows Spy evs)) = (Card B ∈ cloned)"
apply (auto dest!: Spy_knows_cloned)
done

lemma analz_image_Key_Un_Nonce: "analz (Key'K ∪ Nonce'N) = Key'K ∪ Nonce'N"

```

```

apply auto
done

method_setup sc_analz_freshK = {*
  Scan.succeed (fn ctxt =>
    (SIMPLE_METHOD
      (EVERY [REPEAT_FIRST
        (resolve_tac [allI, ballI, impI]),
        REPEAT_FIRST (rtac @{thm analz_image_freshK_lemma}),
        ALLGOALS (asm_simp_tac (Simplifier.context ctxt Smartcard.analz_image_freshK_ss
          addsimps [{thm knows_Spy_Inputs_secureM_sr_Spy},
            @{thm knows_Spy_Outpts_secureM_sr_Spy},
            @{thm shouprubin_assumes_securemeans},
            @{thm analz_image_Key_Un_Nonce}]))) *}
    "for proving the Session Key Compromise theorem for smartcard protocols"

lemma analz_image_freshK [rule_format]:
  "evs ∈ sr ⇒      ∀ K KK.
    (Key K ∈ analz (Key'KK ∪ (knows Spy evs))) =
    (K ∈ KK ∨ Key K ∈ analz (knows Spy evs))"
apply (erule sr.induct)
apply analz_prepare
apply sc_analz_freshK
apply spy_analz
done

lemma analz_insert_freshK: "evs ∈ sr ⇒
  Key K ∈ analz (insert (Key K') (knows Spy evs)) =
  (K = K' ∨ Key K ∈ analz (knows Spy evs))"
apply (simp only: analz_image_freshK_simps analz_image_freshK)
done

lemma Na_Nb_certificate_authentic:
  "[ Crypt (pairK(A,B)) {Nonce Na, Nonce Nb} ∈ parts (knows Spy evs);
    ¬illegalUse(Card B);
    evs ∈ sr ]
  ⇒ Outputs (Card B) B {Nonce Nb, Key (sesK(Nb,pairK(A,B))),
    Crypt (pairK(A,B)) {Nonce Na, Nonce Nb},
    Crypt (pairK(A,B)) (Nonce Nb)} ∈ set evs"
apply (erule rev_mp, erule sr.induct)
apply parts_prepare
apply simp_all

```

apply spy_analz

apply clarify
done

lemma Nb_certificate_authentic:

"[[Crypt (pairK(A,B)) (Nonce Nb) ∈ parts (knows Spy evs);
B ≠ Spy; ¬illegalUse(Card A); ¬illegalUse(Card B);
evs ∈ sr]]

⇒ Outputs (Card A) A {Key (sesK(Nb,pairK(A,B))),
Crypt (pairK(A,B)) (Nonce Nb)} ∈ set evs"

apply (erule rev_mp, erule sr.induct)

apply parts_prepare

apply (case_tac [17] "Aa = Spy")

apply simp_all

apply spy_analz

apply clarify+
done

lemma Outputs_A_Card_imp_pairK_parts:

"[[Outputs (Card A) A
{Key K, Crypt (pairK(A,B)) (Nonce Nb)} ∈ set evs;
evs ∈ sr]]

⇒ ∃ Na. Crypt (pairK(A,B)) {Nonce Na, Nonce Nb} ∈ parts (knows Spy evs)"

apply (erule rev_mp, erule sr.induct)

apply parts_prepare

apply simp_all

apply (blast dest: parts_insertI)

apply force

apply force

apply blast

apply (blast dest: Inputs_imp_knows_Spy_secureM_sr parts.Inj Inputs_A_Card_9
Gets_imp_knows_Spy elim: knows_Spy_partsEs)

apply (blast dest: Inputs_imp_knows_Spy_secureM_sr [THEN parts.Inj]
Inputs_A_Card_9 Gets_imp_knows_Spy
elim: knows_Spy_partsEs)

done

```

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"
apply (erule rev_mp, erule sr.induct)
apply parts_prepare
apply (simp_all (no_asm_simp))

apply spy_analz

apply blast

apply blast

apply (blast dest: Na_Nb_certificate_authentic Inputs_imp_knows_Spy_secureM_sr
[THEN parts.Inj] elim: knows_Spy_partsEs)

apply (blast dest: Na_Nb_certificate_authentic Inputs_imp_knows_Spy_secureM_sr
[THEN parts.Inj] elim: knows_Spy_partsEs)

apply (blast dest: Na_Nb_certificate_authentic Outpts_A_Card_imp_pairK_parts)
done

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"
apply (erule rev_mp, erule sr.induct)
apply parts_prepare
apply (simp_all (no_asm_simp))

apply spy_analz
done

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"
apply (erule rev_mp, erule sr.induct)
apply parts_prepare
apply (simp_all (no_asm_simp))

```

apply spy_analz

apply (fastforce dest: analz.Inj)

apply clarify

apply clarify

apply simp_all

done

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)"

apply (blast intro: sesK_authentic)

done

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)"

apply (erule rev_mp, erule rev_mp, erule sr.induct)

apply analz_prepare

apply (simp_all add: analz_insert_eq analz_insert_freshK pushes_split_ifs)

apply spy_analz

apply (rotate_tac 7)

apply (drule parts.Inj)

apply (fastforce dest: Outpts_B_Card_form_7)

apply (blast dest!: Outpts_B_Card_form_7)

apply clarify

apply (drule Outpts_parts_used)

apply simp

```

apply (fastforce dest: Outpts_B_Card_form_7)

apply clarify
apply (drule Outpts_B_Card_form_7, assumption)
apply simp

apply (blast dest!: Outpts_B_Card_form_7)

apply (blast dest!: Outpts_B_Card_form_7 Outpts_A_Card_form_10)
done

```

```

lemma A_authenticates_B:
  "[ Outpts (Card A) A {Key K, Crypt (pairK(A,B)) (Nonce Nb)} ∈ set evs;
    ¬illegalUse(Card B);
    evs ∈ sr ]
  ⇒ ∃ Na.
    Outpts (Card B) B {Nonce Nb, Key K,
      Crypt (pairK(A,B)) {Nonce Na, Nonce Nb},
      Crypt (pairK(A,B)) (Nonce Nb)} ∈ set evs"
apply (blast dest: Na_Nb_certificate_authentic Outpts_A_Card_form_10 Outpts_A_Card_imp_pairK_parts)
done

```

```

lemma A_authenticates_B_Gets:
  "[ Gets A {Nonce Nb, Crypt (pairK(A,B)) {Nonce Na, Nonce Nb}}
    ∈ set evs;
    ¬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"
apply (blast dest: Gets_imp_knows_Spy [THEN parts.Inj, THEN parts.Snd, THEN
  Na_Nb_certificate_authentic])
done

```

```

lemma B_authenticates_A:
  "[ Gets B (Crypt (pairK(A,B)) (Nonce Nb)) ∈ set evs;
    B ≠ Spy; ¬illegalUse(Card A); ¬illegalUse(Card B);
    evs ∈ sr ]
  ⇒ Outpts (Card A) A
    {Key (sesK(Nb,pairK(A,B))), Crypt (pairK(A,B)) (Nonce Nb)} ∈ set evs"

```

```

apply (erule rev_mp)
apply (erule sr.induct)
apply (simp_all (no_asm_simp))
apply (blast dest: Says_imp_knows_Spy [THEN parts.Inj] Nb_certificate_authentic)
done

```

```

lemma Confidentiality_A: "[ Outpts (Card A) A
  {Key K, Crypt (pairK(A,B)) (Nonce Nb)} ∈ set evs;
  Notes Spy {Key K, Nonce Nb, Agent A, Agent B} ∉ set evs;
  A ≠ Spy; B ≠ Spy; ¬illegalUse(Card A); ¬illegalUse(Card B);
  evs ∈ sr ]
  ⇒ Key K ∉ analz (knows Spy evs)"
apply (drule A_authenticates_B)
prefer 3
apply (erule exE)
apply (drule Confidentiality_B)
apply auto
done

```

```

lemma Outpts_imp_knows_agents_secureM_sr:
  "[ Outpts (Card A) A X ∈ set evs; evs ∈ sr ] ⇒ X ∈ knows A evs"
apply (simp (no_asm_simp) add: Outpts_imp_knows_agents_secureM)
done

```

```

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)"
apply (drule A_authenticates_B)
prefer 3
apply (erule exE)
apply (rule Outpts_imp_knows_agents_secureM_sr [THEN analz.Inj, THEN analz.Snd,
  THEN analz.Fst])
apply assumption+
done

```

```

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)"
  apply (frule B_authenticates_A)
  apply (drule_tac [5] Outpts_B_Card_form_7)
  apply (rule_tac [6] Outpts_imp_knows_agents_secureM_sr [THEN analz.Inj, THEN
analz.Fst])
  prefer 6 apply force
  apply assumption+
  done

```

```

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"
apply (blast dest: Gets_imp_knows_Spy [THEN parts.Inj, THEN Nb_certificate_authentic_bis])
done

```

```

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 "
apply (blast dest: Inputs_A_Card_9 Gets_imp_knows_Spy [THEN parts.Inj, THEN
parts.Snd] Pairkey_certificate_authentic)
done

```

```

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 "
apply (blast dest: Inputs_A_Card_9 Gets_imp_knows_Spy [THEN parts.Inj, THEN
parts.Snd, THEN Na_Nb_certificate_authentic])
done

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"
apply (blast dest: Inputs_A_Card_9)
done

lemma Inputs_A_Card_9_authentic:
  "[[ Inputs A (Card A)
     {Agent B, Nonce Na, Nonce Nb, Nonce Pk,
      Crypt (shrK A) {Nonce Pk, Agent B},
      Crypt (pairK(A,B)) {Nonce Na, Nonce Nb}}, Cert3} ∈ set evs;

   A ≠ Spy; Card A ∉ cloned; ¬illegalUse(Card B); evs ∈ sr ]]
  ⇒ Says Server A {Nonce Pk, Crypt (shrK A) {Nonce Pk, Agent B}}
     ∈ set evs ∧
     Outpts (Card B) B {Nonce Nb, Key (sesK(Nb, pairK (A, B))),
       Crypt (pairK(A,B)) {Nonce Na, Nonce Nb}},
       Crypt (pairK(A,B)) (Nonce Nb)}
     ∈ set evs ∧
     Outpts (Card A) A {Nonce Na, Cert3}
     ∈ set evs"
apply (blast dest: Inputs_A_Card_9 Na_Nb_certificate_authentic Gets_imp_knows_Spy
[THEN parts.Inj, THEN parts.Snd] Pairkey_certificate_authentic)
done

```

```

lemma SR4_imp:
  "[[ Outputs (Card A) A {Nonce Na, Crypt (crdK (Card A)) (Nonce Na)}
    ∈ set evs;
    A ≠ Spy; evs ∈ sr ]]"
  ⇒ ∃ Pk V. Gets A {Pk, V} ∈ set evs"
apply (blast dest: Outpts_A_Card_4 Inputs_A_Card_3)
done

```

```

lemma SR7_imp:
  "[[ Outputs (Card B) B {Nonce Nb, Key K,
    Crypt (pairK(A,B)) {Nonce Na, Nonce Nb},
    Cert2} ∈ set evs;
    B ≠ Spy; evs ∈ sr ]]"
  ⇒ Gets B {Agent A, Nonce Na} ∈ set evs"
apply (blast dest: Outpts_B_Card_7 Inputs_B_Card_6)
done

```

```

lemma SR10_imp:
  "[[ Outputs (Card A) A {Key K, Crypt (pairK(A,B)) (Nonce Nb)}
    ∈ set evs;
    A ≠ Spy; evs ∈ sr ]]"
  ⇒ ∃ Cert1 Cert2.
    Gets A {Nonce (Pairkey (A, B)), Cert1} ∈ set evs ∧
    Gets A {Nonce Nb, Cert2} ∈ set evs"
apply (blast dest: Outpts_A_Card_10 Inputs_A_Card_9)
done

```

```

lemma Outpts_Server_not_evs: "evs ∈ sr ⇒ Outpts (Card Server) P X ∉ set
evs"
apply (erule sr.induct)
apply auto
done

```

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}"

```

```

apply (erule rev_mp)
apply (erule sr.induct)
apply auto
apply (blast dest!: Outpts_Server_not_evs)+
done

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"
apply (erule rev_mp)
apply (erule sr.induct)
prefer 18 apply (fastforce dest: Outpts_A_Card_form_10)
apply auto
done

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"
apply (erule rev_mp)
apply (erule sr.induct)
apply auto
done

end

```

28 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.

axiomatization sesK :: "nat*key => key"
where

inj_sesK [iff]: "(sesK(m,k) = sesK(m',k')) = (m = m' ∧ k = k')" **and**

```
shrK_disj_sesK [iff]: "shrK A ≠ sesK(m,pk)" and
crdK_disj_sesK [iff]: "crdK C ≠ sesK(m,pk)" and
pin_disj_sesK [iff]: "pin P ≠ sesK(m,pk)" and
pairK_disj_sesK[iff]: "pairK(A,B) ≠ sesK(m,pk)" and
```

```
Atomic_distrib [iff]: "Atomic'(KEY'K ∪ NONCE'N) =
Atomic'(KEY'K) ∪ Atomic'(NONCE'N)" and
```

```
shouprubin_assumes_securemeans [iff]: "evs ∈ srb ⇒ secureM"
```

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

```
inductive_set srb :: "event list set"
where
```

```
Nil: "[ ] ∈ srb"
```

```
/ Fake: "[ evsF ∈ srb; X ∈ synth (analz (knows Spy evsF));
illegalUse(Card B) ]
⇒ Says Spy A X #
Inputs Spy (Card B) X # evsF ∈ srb"
```

```
/ Forge:
"[ evsFo ∈ srb; Nonce Nb ∈ analz (knows Spy evsFo);
Key (pairK(A,B)) ∈ knows Spy evsFo ]
⇒ Notes Spy (Key (sesK(Nb,pairK(A,B)))) # evsFo ∈ srb"
```

```
/ Reception: "[ evsrb ∈ srb; Says A B X ∈ set evsrb ]
⇒ Gets B X # evsrb ∈ srb"
```

```
/ SR_U1: "[ evs1 ∈ srb; A ≠ Server ]
⇒ Says A Server {Agent A, Agent B}
# evs1 ∈ srb"
```

```
/ SR_U2: "[ evs2 ∈ srb;
Gets Server {Agent A, Agent B} ∈ set evs2 ]
⇒ Says Server A {Nonce (Pairkey(A,B)),
Crypt (shrK A) {Nonce (Pairkey(A,B)), Agent B}
}
# evs2 ∈ srb"
```

```

/ SR_U3: "[ evs3 ∈ srb; legalUse(Card A);
          Says A Server {Agent A, Agent B} ∈ set evs3;
          Gets A {Nonce Pk, Certificate} ∈ set evs3 ]
⇒ Inputs A (Card A) (Agent A)
   # evs3 ∈ srb"

/ SR_U4: "[ evs4 ∈ srb;
          Nonce Na ∉ used evs4; legalUse(Card A); A ≠ Server;
          Inputs A (Card A) (Agent A) ∈ set evs4 ]
⇒ Outputs (Card A) A {Nonce Na, Crypt (crdK (Card A)) (Nonce Na)}
   # evs4 ∈ srb"

/ SR_U4Fake: "[ evs4F ∈ srb; Nonce Na ∉ used evs4F;
              illegalUse(Card A);
              Inputs Spy (Card A) (Agent A) ∈ set evs4F ]
⇒ Outputs (Card A) Spy {Nonce Na, Crypt (crdK (Card A)) (Nonce Na)}
   # evs4F ∈ srb"

/ SR_U5: "[ evs5 ∈ srb;
          Outputs (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 ]
⇒ Outputs (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;
  Outputs (Card A) A {Agent B, Nonce Nb, Key K, Certificate}
  ∈ set evs11 ]
  ⇒ Says A B (Certificate)
  # evs11 ∈ srb"

/ Oops1:
  "[ evs01 ∈ srb;
    Outputs (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"

/ Oops2:
  "[ evs02 ∈ srb;
    Outputs (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"
apply (erule rev_mp, erule srb.induct)
apply auto
done

lemma Gets_imp_knows_Spy:
  "[[ Gets B X ∈ set evs; evs ∈ srb ]] ⇒ X ∈ knows Spy evs"
apply (blast dest!: Gets_imp_Says Says_imp_knows_Spy)
done

lemma Gets_imp_knows_Spy_parts_Snd:
  "[[ Gets B {X, Y} ∈ set evs; evs ∈ srb ]] ⇒ Y ∈ parts (knows Spy evs)"
apply (blast dest!: Gets_imp_Says Says_imp_knows_Spy parts.Inj parts.Snd)
done

lemma Gets_imp_knows_Spy_analz_Snd:
  "[[ Gets B {X, Y} ∈ set evs; evs ∈ srb ]] ⇒ Y ∈ analz (knows Spy evs)"
apply (blast dest!: Gets_imp_Says Says_imp_knows_Spy analz.Inj analz.Snd)
done

lemma Inputs_imp_knows_Spy_secureM_srb:
  "[[ Inputs Spy C X ∈ set evs; evs ∈ srb ]] ⇒ X ∈ knows Spy evs"
apply (simp (no_asm_simp) add: Inputs_imp_knows_Spy_secureM)
done

lemma knows_Spy_Inputs_secureM_srb_Spy:
  "evs ∈ srb ⇒ knows Spy (Inputs Spy C X # evs) = insert X (knows Spy
evs)"
apply (simp (no_asm_simp))
done

lemma knows_Spy_Inputs_secureM_srb:
  "[[ A ≠ Spy; evs ∈ srb ]] ⇒ knows Spy (Inputs A C X # evs) = knows Spy
evs"
apply (simp (no_asm_simp))
done

lemma knows_Spy_Outpts_secureM_srb_Spy:
  "evs ∈ srb ⇒ knows Spy (Outpts C Spy X # evs) = insert X (knows Spy
evs)"
apply (simp (no_asm_simp))
done

lemma knows_Spy_Outpts_secureM_srb:
  "[[ A ≠ Spy; evs ∈ srb ]] ⇒ knows Spy (Outpts C A X # evs) = knows Spy
evs"

```

```

apply (simp (no_asm_simp))
done

```

```

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)"
apply (erule rev_mp, erule srb.induct)
apply auto
done

```

```

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"
apply (erule rev_mp, erule srb.induct)
apply auto
done

```

```

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"
apply (erule rev_mp, erule srb.induct)
apply auto
done

```

```

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"
apply (erule rev_mp, erule srb.induct)
apply auto
done

```

```

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"
apply (erule rev_mp, erule srb.induct)
apply auto
done

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)),
      Ver1, Ver2, Ver3} ∈ set evs)"
apply (erule rev_mp, erule srb.induct)
apply auto
done

lemma Outpts_A_Card_10_imp_Inputs:
  "[[ Outpts (Card A) A {Agent B, Nonce Nb, Key K, Certificate}
    ∈ set evs; evs ∈ srb ] ]
  ⇒ (∃ Na Ver1 Ver2 Ver3.
    Inputs A (Card A) {Agent B, Nonce Na, Nonce Nb, Nonce (Pairkey(A,B)),
      Ver1, Ver2, Ver3} ∈ set evs)"
apply (erule rev_mp, erule srb.induct)
apply simp_all
apply blast+
done

lemma Outpts_honest_A_Card_4:
  "[[ Outpts C A {Nonce Na, Crypt K X} ∈ set evs;
    A ≠ Spy; evs ∈ srb ] ]
  ⇒ legalUse(C) ∧ C = (Card A) ∧
    Inputs A (Card A) (Agent A) ∈ set evs"
apply (erule rev_mp, erule srb.induct)
apply auto
done

```

lemma *Outpts_honest_B_Card_7:*

```
"[[ Outpts C B {Nonce Nb, Agent A, Key K, Cert1, Cert2} ∈ set evs;
   B ≠ Spy; evs ∈ srb ]]
⇒ legalUse(C) ∧ C = (Card B) ∧
   (∃ Na. Inputs B (Card B) {Agent A, Nonce Na} ∈ set evs)"
apply (erule rev_mp, erule srb.induct)
apply auto
done
```

lemma *Outpts_honest_A_Card_10:*

```
"[[ Outpts C A {Agent B, Nonce Nb, Key K, Certificate} ∈ set evs;
   A ≠ Spy; evs ∈ srb ]]
⇒ legalUse(C) ∧ C = (Card A) ∧
   (∃ Na Pk Ver1 Ver2 Ver3.
    Inputs A (Card A) {Agent B, Nonce Na, Nonce Nb, Pk,
                      Ver1, Ver2, Ver3} ∈ set evs)"
apply (erule rev_mp, erule srb.induct)
apply simp_all
apply blast+
done
```

lemma *Outpts_which_Card_4:*

```
"[[ Outpts (Card A) A {Nonce Na, Crypt K X} ∈ set evs; evs ∈ srb ]]
⇒ Inputs A (Card A) (Agent A) ∈ set evs"
apply (erule rev_mp, erule srb.induct)
apply (simp_all (no_asm_simp))
apply clarify
done
```

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"
apply (erule rev_mp, erule srb.induct)
apply auto
done
```

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"
apply (erule rev_mp, erule srb.induct)
apply auto
done
```

```

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))"
apply (erule rev_mp, erule srb.induct)
apply (simp_all (no_asm_simp))
done

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))"
apply (erule rev_mp, erule srb.induct)
apply auto
done

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))"
apply (erule rev_mp, erule srb.induct)
apply (simp_all (no_asm_simp))
done

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))"
apply (erule rev_mp, erule srb.induct)
apply (simp_all (no_asm_simp))
done

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)"
apply (erule rev_mp)

```

```

apply (erule srb.induct)
apply (simp_all (no_asm_simp))

apply force

apply (blast dest!: Outpts_A_Card_form_4)
done

```

```

lemma Inputs_Card_legalUse:
  "[[ Inputs A (Card A) X ∈ set evs; evs ∈ srb ]] ⇒ legalUse(Card A)"
apply (erule rev_mp, erule srb.induct)
apply auto
done

```

```

lemma Outpts_Card_legalUse:
  "[[ Outpts (Card A) A X ∈ set evs; evs ∈ srb ]] ⇒ legalUse(Card A)"
apply (erule rev_mp, erule srb.induct)
apply auto
done

```

```

lemma Inputs_Card: "[[ Inputs A C X ∈ set evs; A ≠ Spy; evs ∈ srb ]]
  ⇒ C = (Card A) ∧ legalUse(C)"
apply (erule rev_mp, erule srb.induct)
apply auto
done

```

```

lemma Outpts_Card: "[[ Outpts C A X ∈ set evs; A ≠ Spy; evs ∈ srb ]]
  ⇒ C = (Card A) ∧ legalUse(C)"
apply (erule rev_mp, erule srb.induct)
apply auto
done

```

```

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)"
apply (blast dest: Inputs_Card Outpts_Card)
done

```

```

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))"
apply (erule rev_mp, erule srb.induct)
apply auto
done

```

```

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'"
apply (erule rev_mp, erule srb.induct, simp_all)
apply (fastforce dest: Outpts_parts_used)
apply blast
done

```

```

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'"
apply (erule rev_mp, erule rev_mp, erule srb.induct, simp_all)
apply (fastforce dest: Outpts_parts_used)
apply blast
done

```

```

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'"
apply (erule rev_mp, erule rev_mp, erule srb.induct, simp_all)
apply (fastforce dest: Outpts_parts_used)
apply blast
done

```

```

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'"
apply (erule rev_mp, erule rev_mp, erule srb.induct, simp_all)
apply (blast dest: Outpts_A_Card_form_bis)
apply blast
done

```

```

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"
apply (erule rev_mp, erule srb.induct, simp_all add: Unique_def)
apply (fastforce dest: Outpts_parts_used)
apply blast
apply (fastforce dest: Outpts_parts_used)
apply blast
done

```

```

lemma Spy_knows_Na:
  "[[ Says A B {Agent A, Nonce Na} ∈ set evs; evs ∈ srb ]
  ⇒ Nonce Na ∈ analz (knows Spy evs)"
apply (blast dest!: Says_imp_knows_Spy [THEN analz.Inj, THEN analz.Snd])
done

```

```

lemma Spy_knows_Nb:
  "[[ Says B A {Nonce Nb, Certificate} ∈ set evs; evs ∈ srb ]
  ⇒ Nonce Nb ∈ analz (knows Spy evs)"
apply (blast dest!: Says_imp_knows_Spy [THEN analz.Inj, THEN analz.Fst])
done

```

```

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)"
apply (blast dest!: Gets_imp_knows_Spy [THEN analz.Inj])
done

```

```

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"
apply (erule rev_mp, erule srb.induct)
apply (simp_all (no_asm_simp))
apply force
done

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)"
apply (case_tac "A = Spy")
apply (fastforce dest!: Inputs_imp_knows_Spy_secureM [THEN analz.Inj])
apply (blast dest!: Pairkey_Inputs_imp_Gets [THEN Pairkey_Gets_analz_knows_Spy])
done

declare shrK_disj_sesK [THEN not_sym, iff]
declare pin_disj_sesK [THEN not_sym, iff]
declare crdK_disj_sesK [THEN not_sym, iff]
declare pairK_disj_sesK [THEN not_sym, iff]

ML
{*
structure ShoupRubinBella =
struct

fun prepare_tac ctxt =
  (*SR_U8*) forward_tac [ @{thm Outpts_B_Card_form_7} ] 14 THEN
  (*SR_U8*) clarify_tac ctxt 15 THEN
  (*SR_U9*) forward_tac [ @{thm Outpts_A_Card_form_4} ] 16 THEN
  (*SR_U11*) forward_tac [ @{thm Outpts_A_Card_form_10} ] 21

fun parts_prepare_tac ctxt =
  prepare_tac ctxt THEN
  (*SR_U9*) dresolve_tac [ @{thm Gets_imp_knows_Spy_parts_Snd} ] 18 THEN
  (*SR_U9*) dresolve_tac [ @{thm Gets_imp_knows_Spy_parts_Snd} ] 19 THEN
  (*Oops1*) dresolve_tac [ @{thm Outpts_B_Card_form_7} ] 25 THEN
  (*Oops2*) dresolve_tac [ @{thm Outpts_A_Card_form_10} ] 27 THEN
  (*Base*) (force_tac ctxt) 1
}

```

```

fun analz_prepare_tac ctxt =
  prepare_tac ctxt THEN
  dtac (@{thm Gets_imp_knows_Spy_analz_Snd}) 18 THEN
  (*SR_U9*) dtac (@{thm Gets_imp_knows_Spy_analz_Snd}) 19 THEN
  REPEAT_FIRST (eresolve_tac [asm_rl, conjE] ORELSE' hyp_subst_tac)

end
*}

method_setup prepare = {*
  Scan.succeed (fn ctxt => SIMPLE_METHOD (ShoupRubinBella.prepare_tac ctxt))
*}
  "to launch a few simple facts that will help the simplifier"

method_setup parts_prepare = {*
  Scan.succeed (fn ctxt => SIMPLE_METHOD (ShoupRubinBella.parts_prepare_tac
  ctxt)) *}
  "additional facts to reason about parts"

method_setup analz_prepare = {*
  Scan.succeed (fn ctxt => SIMPLE_METHOD (ShoupRubinBella.analz_prepare_tac
  ctxt)) *}
  "additional facts to reason about analz"

lemma Spy_parts_keys [simp]: "evs ∈ srb ⇒
  (Key (shrK P) ∈ parts (knows Spy evs)) = (Card P ∈ cloned) ∧
  (Key (pin P) ∈ parts (knows Spy evs)) = (P ∈ bad ∨ Card P ∈ cloned) ∧

  (Key (crdK C) ∈ parts (knows Spy evs)) = (C ∈ cloned) ∧
  (Key (pairK(A,B)) ∈ parts (knows Spy evs)) = (Card B ∈ cloned)"
apply (erule srb.induct)
apply parts_prepare
apply simp_all
apply (blast intro: parts_insertI)
done

lemma Spy_analz_shrK[simp]: "evs ∈ srb ⇒
  (Key (shrK P) ∈ analz (knows Spy evs)) = (Card P ∈ cloned)"
apply (auto dest!: Spy_knows_cloned)
done

lemma Spy_analz_crdK[simp]: "evs ∈ srb ⇒
  (Key (crdK C) ∈ analz (knows Spy evs)) = (C ∈ cloned)"
apply (auto dest!: Spy_knows_cloned)
done

```

```

lemma Spy_analz_pairK[simp]: "evs ∈ srb ⇒
  (Key (pairK(A,B)) ∈ analz (knows Spy evs)) = (Card B ∈ cloned)"
apply (auto dest!: Spy_knows_cloned)
done

```

```

lemma analz_image_Key_Un_Nonce: "analz (Key'K ∪ Nonce'N) = Key'K ∪ Nonce'N"
apply auto
done

```

```

method_setup sc_analz_freshK = {*
  Scan.succeed (fn ctxt =>
    (SIMPLE_METHOD
      (EVERY [REPEAT_FIRST (resolve_tac [allI, ballI, impI]),
        REPEAT_FIRST (rtac @{thm analz_image_freshK_lemma}),
        ALLGOALS (asm_simp_tac (Simplifier.context ctxt Smartcard.analz_image_freshK_ss
          addsimps [{thm knows_Spy_Inputs_secureM_srb_Spy},
            @{thm knows_Spy_Outpts_secureM_srb_Spy},
            @{thm shouprubin_assumes_securemeans},
            @{thm analz_image_Key_Un_Nonce}])))))) *}
"for proving the Session Key Compromise theorem for smartcard protocols"

```

```

lemma analz_image_freshK [rule_format]:
  "evs ∈ srb ⇒      ∃ K KK.
    (Key K ∈ analz (Key'KK ∪ (knows Spy evs))) =
    (K ∈ KK ∨ Key K ∈ analz (knows Spy evs))"
apply (erule srb.induct)
apply analz_prepare
apply sc_analz_freshK
apply spy_analz
done

```

```

lemma analz_insert_freshK: "evs ∈ srb ⇒
  Key K ∈ analz (insert (Key K') (knows Spy evs)) =
  (K = K' ∨ Key K ∈ analz (knows Spy evs))"
apply (simp only: analz_image_freshK_simps analz_image_freshK)
done

```

```

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"
  apply (erule rev_mp, erule srb.induct)
  apply parts_prepare
  apply simp_all

```

```

  apply spy_analz

```

```

  apply clarify

```

```

  apply clarify
  done

```

```

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"

```

```

  apply (erule rev_mp, erule srb.induct)

```

```

  apply parts_prepare

```

```

  apply (case_tac [17] "Aa = Spy")

```

```

  apply simp_all

```

```

  apply spy_analz

```

```

  apply clarify+

```

```

  done

```

```

lemma Outpts_A_Card_imp_pairK_parts:

```

```

  "[ Outpts (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)"

```

```

  apply (erule rev_mp, erule srb.induct)

```

```

  apply parts_prepare

```

```

  apply simp_all

```

```

  apply (blast dest: parts_insertI)

```

```

  apply force

```

```

  apply force

```

```

  apply blast

```

```

  apply (blast dest: Inputs_imp_knows_Spy_secureM_srb parts.Inj Inputs_A_Card_9

```

```

Gets_imp_knows_Spy elim: knows_Spy_partsEs)

apply (blast dest: Inputs_imp_knows_Spy_secureM_srb [THEN parts.Inj]
        Inputs_A_Card_9 Gets_imp_knows_Spy
        elim: knows_Spy_partsEs)
done

lemma Nb_certificate_authentic_bis:
  "[ Crypt (pairK(A,B)) (Nonce Nb) ∈ parts (knows Spy 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"
apply (erule rev_mp, erule srb.induct)
apply parts_prepare
apply (simp_all (no_asm_simp))

apply spy_analz

apply blast

apply blast

apply force

apply (blast dest: Na_Nb_certificate_authentic Inputs_imp_knows_Spy_secureM_srb
[THEN parts.Inj] elim: knows_Spy_partsEs)

apply (blast dest: Na_Nb_certificate_authentic Inputs_imp_knows_Spy_secureM_srb
[THEN parts.Inj] elim: knows_Spy_partsEs)

apply (blast dest: Na_Nb_certificate_authentic Outpts_A_Card_imp_pairK_parts)
done

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"
apply (erule rev_mp, erule srb.induct)
apply parts_prepare
apply (simp_all (no_asm_simp))

apply spy_analz

apply force
done

```

```

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"
apply (erule rev_mp, erule srb.induct)
apply parts_prepare
apply (simp_all)

apply spy_analz

apply (fastforce dest: analz.Inj)

apply clarify

apply clarify
done

```

```

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)"
apply (blast intro: sesK_authentic)
done

```

```

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)"
apply (erule rev_mp, erule rev_mp, erule srb.induct)
apply analz_prepare
apply (simp_all add: analz_insert_eq analz_insert_freshK pushes split_ifs)

apply spy_analz

apply (rotate_tac 7)

```

```

apply (drule parts.Inj)
apply (fastforce dest: Outpts_B_Card_form_7)

apply (blast dest!: Outpts_B_Card_form_7)

apply clarify
apply (drule Outpts_parts_used)
apply simp

apply (fastforce dest: Outpts_B_Card_form_7)

apply clarify
apply (drule Outpts_B_Card_form_7, assumption)
apply simp

apply (blast dest!: Outpts_B_Card_form_7)

apply (blast dest!: Outpts_B_Card_form_7 Outpts_A_Card_form_10)
done

```

```

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"
apply (blast dest: Na_Nb_certificate_authentic Outpts_A_Card_form_10 Outpts_A_Card_imp_pairK_parts)
done

```

```

lemma A_authenticates_B_Gets:
  "[ Gets A {Nonce Nb, Crypt (pairK(A,B)) {Nonce Na, Nonce Nb}}
    ∈ set evs;
    ¬illegalUse(Card B);
    evs ∈ srb ]
  ⇒ Outpts (Card B) B {Nonce Nb, Agent A, Key (sesK(Nb, pairK (A, B))),
    Crypt (pairK(A,B)) {Nonce Na, Nonce Nb},
    Crypt (pairK(A,B)) (Nonce Nb)} ∈ set evs"
apply (blast dest: Gets_imp_knows_Spy [THEN parts.Inj, THEN parts.Snd, THEN
  Na_Nb_certificate_authentic])
done

```

```

lemma A_authenticates_B_bis:
  "[ Outpts (Card A) A {Agent B, Nonce Nb, Key K, Cert2} ∈ set evs;
    ¬illegalUse(Card B);

```

```

    evs ∈ srb ]]
  ⇒ ∃ Cert1. Outpts (Card B) B {Nonce Nb, Agent A, Key K, Cert1, Cert2}
    ∈ set evs"
apply (blast dest: Na_Nb_certificate_authentic Outpts_A_Card_form_10 Outpts_A_Card_imp_pairK_p
done

```

```

lemma B_authenticates_A:
  "[[ Gets B (Crypt (pairK(A,B)) (Nonce Nb)) ∈ set evs;
    B ≠ Spy; ¬illegalUse(Card A); ¬illegalUse(Card B);
    evs ∈ srb ]]
  ⇒ Outpts (Card A) A {Agent B, Nonce Nb,
    Key (sesK(Nb,pairK(A,B))), Crypt (pairK(A,B)) (Nonce Nb)} ∈ set evs"
apply (erule rev_mp)
apply (erule srb.induct)
apply (simp_all (no_asm_simp))
apply (blast dest: Says_imp_knows_Spy [THEN parts.Inj] Nb_certificate_authentic)
done

```

```

lemma B_authenticates_A_bis:
  "[[ Outpts (Card B) B {Nonce Nb, Agent A, Key K, Cert1, Cert2} ∈ set evs;
    Gets B (Cert2) ∈ set evs;
    B ≠ Spy; ¬illegalUse(Card A); ¬illegalUse(Card B);
    evs ∈ srb ]]
  ⇒ Outpts (Card A) A {Agent B, Nonce Nb, Key K, Cert2} ∈ set evs"
apply (blast dest: Outpts_B_Card_form_7 B_authenticates_A)
done

```

```

lemma Confidentiality_A:
  "[[ Outpts (Card A) A {Agent B, Nonce Nb,
    Key K, Certificate} ∈ set evs;
    Notes Spy {Key K, Nonce Nb, Agent A, Agent B} ∉ set evs;
    A ≠ Spy; B ≠ Spy; ¬illegalUse(Card A); ¬illegalUse(Card B);
    evs ∈ srb ]]
  ⇒ Key K ∉ analz (knows Spy evs)"
apply (drule A_authenticates_B)
prefer 3
apply (erule exE)
apply (drule Confidentiality_B)
apply auto
done

```

```

lemma Outpts_imp_knows_agents_secureM_srb:

```

```

"[[ Outpts (Card A) A X ∈ set evs; evs ∈ srb ]] ⇒ X ∈ knows A evs"
apply (simp (no_asm_simp) add: Outpts_imp_knows_agents_secureM)
done

```

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)"
apply (drule A_authenticates_B)
prefer 3
apply (erule exE)
apply (rule Outpts_imp_knows_agents_secureM_srb [THEN analz.Inj, THEN analz.Snd,
THEN analz.Snd, THEN analz.Fst])
apply assumption+
done

```

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)"
apply (frule Outpts_B_Card_form_7)
apply assumption apply simp
apply (frule B_authenticates_A)
apply (rule_tac [5] Outpts_imp_knows_agents_secureM_srb [THEN analz.Inj, THEN
analz.Snd, THEN analz.Snd, THEN analz.Fst])
apply simp+
done

```

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"
apply (blast dest: Gets_imp_knows_Spy [THEN parts.Inj, THEN Nb_certificate_authentic_bis])
done

```

```

lemma Pairkey_certificate_authentic_A_Card:
  "[[ Inputs A (Card A)
    {Agent B, Nonce Na, Nonce Nb, Nonce Pk,
     Crypt (shrK A) {Nonce Pk, Agent B},
     Cert2, Cert3} ∈ set evs;
    A ≠ Spy; Card A ∉ cloned; evs ∈ srb ]
  ⇒ Pk = Pairkey(A,B) ∧
    Says Server A {Nonce (Pairkey(A,B)),
                  Crypt (shrK A) {Nonce (Pairkey(A,B)), Agent B}}
    ∈ set evs "
apply (blast dest: Inputs_A_Card_9 Gets_imp_knows_Spy [THEN parts.Inj, THEN
parts.Snd] Pairkey_certificate_authentic)
done

```

```

lemma Na_Nb_certificate_authentic_A_Card:
  "[[ Inputs A (Card A)
    {Agent B, Nonce Na, Nonce Nb, Nonce Pk,
     Cert1, Crypt (pairK(A,B)) {Nonce Na, Nonce Nb}, Cert3} ∈ set evs;

    A ≠ Spy; ¬illegalUse(Card B); evs ∈ srb ]
  ⇒ 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 "
apply (frule Inputs_A_Card_9)
apply assumption+
apply (blast dest: Inputs_A_Card_9 Gets_imp_knows_Spy [THEN parts.Inj, THEN
parts.Snd, THEN Na_Nb_certificate_authentic])
done

```

```

lemma Na_authentic_A_Card:
  "[[ Inputs A (Card A)
    {Agent B, Nonce Na, Nonce Nb, Nonce Pk,
     Cert1, Cert2, Cert3} ∈ set evs;
    A ≠ Spy; evs ∈ srb ]
  ⇒ Outputs (Card A) A {Nonce Na, Cert3}
    ∈ set evs"
apply (blast dest: Inputs_A_Card_9)
done

```

```

lemma Inputs_A_Card_9_authentic:
  "[[ Inputs A (Card A)
    {Agent B, Nonce Na, Nonce Nb, Nonce Pk,

```

```

    Crypt (shrK A) {Nonce Pk, Agent B},
    Crypt (pairK(A,B)) {Nonce Na, Nonce Nb}, Cert3} ∈ set evs;

A ≠ Spy; Card A ∉ cloned; ¬illegalUse(Card B); evs ∈ srb ]
⇒ Says Server A {Nonce Pk, Crypt (shrK A) {Nonce Pk, Agent B}}
  ∈ set evs ∧
  Outpts (Card B) B {Nonce Nb, Agent A, Key (sesK(Nb, pairK (A, B))),

                                Crypt (pairK(A,B)) {Nonce Na, Nonce Nb},
                                Crypt (pairK(A,B)) (Nonce Nb)}
  ∈ set evs ∧
  Outpts (Card A) A {Nonce Na, Cert3}
  ∈ set evs"
apply (blast dest: Inputs_A_Card_9 Na_Nb_certificate_authentic Gets_imp_knows_Spy
[THEN parts.Inj, THEN parts.Snd] Pairkey_certificate_authentic)
done

```

```

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"
apply (blast dest: Outpts_A_Card_4 Inputs_A_Card_3)
done

```

```

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"
apply (blast dest: Outpts_B_Card_7 Inputs_B_Card_6)
done

```

```

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"
apply (blast dest: Outpts_A_Card_10 Inputs_A_Card_9)
done

```

```

lemma Outpts_Server_not_evs:
  "evs ∈ srb ⇒ Outpts (Card Server) P X ∉ set evs"
apply (erule srb.induct)
apply auto
done

```

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}"
apply (erule rev_mp)
apply (erule srb.induct)
apply auto
apply (blast dest!: Outpts_Server_not_evs)+
done

```

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"
apply (erule rev_mp)
apply (erule srb.induct)
prefer 18 apply (fastforce dest: Outpts_A_Card_form_10)
apply auto
done

```

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"
apply (erule rev_mp)

```

```

apply (erule srb.induct)
apply auto
done

end

```

29 Smartcard protocols: rely on conventional Message and on new EventSC and Smartcard

```

theory Auth_Smartcard
imports
  "ShoupRubin"
  "ShoupRubinBella"
begin

end

```

30 Extensions to Standard Theories

```

theory Extensions
imports "../Event"
begin

```

30.1 Extensions to Theory Set

```

lemma eq: "[| !!x. x:A ==> x:B; !!x. x:B ==> x:A |] ==> A=B"
by auto

```

```

lemma insert_Un: "P ({x} Un A) ==> P (insert x A)"
by simp

```

```

lemma in_sub: "x:A ==> {x}<=A"
by auto

```

30.2 Extensions to Theory List

30.2.1 "remove l x" erase the first element of "l" equal to "x"

```

primrec remove :: "'a list => 'a => 'a list" where
  "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"
by (induct l, auto)

```

30.3 Extensions to Theory Message

30.3.1 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]

```

30.3.2 extract the agent number of an Agent message

```

primrec agt_nb :: "msg => agent" where
  "agt_nb (Agent A) = A"

```

30.3.3 messages that are pairs

```

definition is_MPair :: "msg => bool" where
  "is_MPair X == EX Y Z. X = {|Y,Z|}"

```

```

declare is_MPair_def [simp]

```

```

lemma MPair_is_MPair [iff]: "is_MPair {|X,Y|}"
by simp

```

```

lemma Agent_isnt_MPair [iff]: "~ is_MPair (Agent A)"
by simp

```

```

lemma Number_isnt_MPair [iff]: "~ is_MPair (Number n)"
by simp

```

```

lemma Key_isnt_MPair [iff]: "~ is_MPair (Key K)"
by simp

```

```

lemma Nonce_isnt_MPair [iff]: "~ is_MPair (Nonce n)"
by simp

```

```

lemma Hash_isnt_MPair [iff]: "~ is_MPair (Hash X)"
by simp

```

```

lemma Crypt_isnt_MPair [iff]: "~ is_MPair (Crypt K X)"
by simp

```

abbreviation

```

not_MPair :: "msg => bool" where
  "not_MPair X == ~ is_MPair X"

```

```

lemma is_MPairE: "[| is_MPair X ==> P; not_MPair X ==> P |] ==> P"
by auto

```

```

declare is_MPair_def [simp del]

```

```

definition has_no_pair :: "msg set => bool" where
  "has_no_pair H == ALL X Y. {|X,Y|} ~:H"

```

```

declare has_no_pair_def [simp]

```

30.3.4 well-foundedness of messages

```
lemma wf_Crypt1 [iff]: "Crypt K X ~ = X"
by (induct X, auto)
```

```
lemma wf_Crypt2 [iff]: "X ~ = Crypt K X"
by (induct X, auto)
```

```
lemma parts_size: "X:parts {Y} ==> X=Y | size X < size Y"
by (erule parts.induct, auto)
```

```
lemma wf_Crypt_parts [iff]: "Crypt K X ~:parts {X}"
by (auto dest: parts_size)
```

30.3.5 lemmas on keysFor

```
definition usekeys :: "msg set => key set" where
"usekeys G == {K. EX Y. Crypt K Y:G}"
```

```
lemma finite_keysFor [intro]: "finite G ==> finite (keysFor G)"
apply (simp add: keysFor_def)
apply (rule finite_UN_I, auto)
apply (erule finite_induct, auto)
apply (case_tac "EX K X. x = Crypt K X", clarsimp)
apply (subgoal_tac "{Ka. EX Xa. (Ka=K & Xa=X) | Crypt Ka Xa:F}
= insert K (usekeys F)", auto simp: usekeys_def)
by (subgoal_tac "{K. EX X. Crypt K X = x | Crypt K X:F} = usekeys F",
auto simp: usekeys_def)
```

30.3.6 lemmas on parts

```
lemma parts_sub: "[| X:parts G; G<=H |] ==> X:parts H"
by (auto dest: parts_mono)
```

```
lemma parts_Diff [dest]: "X:parts (G - H) ==> X:parts G"
by (erule parts_sub, auto)
```

```
lemma parts_Diff_notin: "[| Y ~:H; Nonce n ~:parts (H - {Y}) |]
==> Nonce n ~:parts H"
by simp
```

```
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})"
by (induct X, auto)
```

```
lemma finite_parts [intro]: "finite H ==> finite (parts H)"
apply (erule finite_induct, simp)
by (rule parts_insert_substI, simp)
```

```
lemma parts_parts: "[| X:parts {Y}; Y:parts G |] ==> X:parts G"
by (frule parts_cut, auto)
```

```
lemma parts_parts_parts: "[| X:parts {Y}; Y:parts {Z}; Z:parts G |] ==> X:parts G"

```

```
by (auto dest: parts_parts)

```

```
lemma parts_parts_Crypt: "[| Crypt K X:parts G; Nonce n:parts {X} |] ==> Nonce n:parts G"

```

```
by (blast intro: parts.Body dest: parts_parts)

```

30.3.7 lemmas on synth

```
lemma synth_sub: "[| X:synth G; G<=H |] ==> X:synth H"

```

```
by (auto dest: synth_mono)

```

```
lemma Crypt_synth [rule_format]: "[| X:synth G; Key K ~:G |] ==> Crypt K Y:parts {X} --> Crypt K Y:parts G"

```

```
by (erule synth.induct, auto dest: parts_sub)

```

30.3.8 lemmas on analz

```
lemma analz_UnI1 [intro]: "X:analz G ==> X:analz (G Un H)"

```

```
by (subgoal_tac "G <= G Un H") (blast dest: analz_mono)+

```

```
lemma analz_sub: "[| X:analz G; G <= H |] ==> X:analz H"

```

```
by (auto dest: analz_mono)

```

```
lemma analz_Diff [dest]: "X:analz (G - H) ==> X:analz G"

```

```
by (erule analz.induct, auto)

```

```
lemmas in_analz_subset_cong = analz_subset_cong [THEN subsetD]

```

```
lemma analz_eq: "A=A' ==> analz A = analz A'"

```

```
by auto

```

```
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"

```

```
by (blast intro: analz.Decrypt analz_insert_eq)

```

```
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)"

```

```
by (erule analz.induct, auto)

```

```
by (erule analz.induct, auto)

```

```
lemma analz_needs_only_finite: "X:analz H ==> EX G. G <= H & finite G"

```

```
by (erule analz.induct, auto)

```

```
lemma notin_analz_insert: "X ~:analz (insert Y G) ==> X ~:analz G"

```

```
by auto

```

30.3.9 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"

```

```
by (erule parts.induct, auto dest: parts.Fst parts.Snd parts.Body)

```

```
lemma in_analz: "Y:analz H ==> EX X. X:H & Y:parts {X}"
by (erule analz.induct, auto intro: parts.Fst parts.Snd parts.Body)
```

```
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"
apply (drule parts_insert_substD, clarify)
apply (frule in_sub)
apply (frule parts_mono)
apply auto
done
```

30.3.10 greatest nonce used in a message

```
fun greatest_msg :: "msg => nat"
where
  "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"
by (induct X, auto)
```

30.3.11 sets of keys

```
definition keyset :: "msg set => bool" where
"keyset G == ALL X. X:G --> (EX K. X = Key K)"
```

```
lemma keyset_in [dest]: "[| keyset G; X:G |] ==> EX K. X = Key K"
by (auto simp: keyset_def)
```

```
lemma MPair_notin_keyset [simp]: "keyset G ==> {|X,Y|} ~:G"
by auto
```

```
lemma Crypt_notin_keyset [simp]: "keyset G ==> Crypt K X ~:G"
by auto
```

```
lemma Nonce_notin_keyset [simp]: "keyset G ==> Nonce n ~:G"
by auto
```

```
lemma parts_keyset [simp]: "keyset G ==> parts G = G"
by (auto, erule parts.induct, auto)
```

30.3.12 keys a priori necessary for decrypting the messages of G

```
definition keysfor :: "msg set => msg set" where
"keysfor G == Key ` keysFor (parts G)"
```

```
lemma keyset_keysfor [iff]: "keyset (keysfor G)"
by (simp add: keyset_def keysfor_def, blast)
```

```
lemma keyset_Diff_keysfor [simp]: "keyset H ==> keyset (H - keysfor G)"
```

by (auto simp: keyset_def)

lemma keysfor_Crypt: "Crypt K X:parts G ==> Key (invKey K):keysfor G"
by (auto simp: keysfor_def Crypt_imp_invKey_keysFor)

lemma no_key_no_Crypt: "Key K ~:keysfor G ==> Crypt (invKey K) X ~:parts G"
by (auto dest: keysfor_Crypt)

lemma finite_keysfor [intro]: "finite G ==> finite (keysfor G)"
by (auto simp: keysfor_def intro: finite_UN_I)

30.3.13 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)))"
apply (rule eq)
apply (erule analz.induct, blast)
apply (simp, blast)
apply (simp, blast)
apply (case_tac "Key (invKey K):H - keysfor G", clarsimp)
apply (drule_tac X=X in no_key_no_Crypt)
by (auto intro: analz_sub)

lemmas analz_keyset_substD = analz_keyset [THEN sym, THEN ssubst]

30.4 Extensions to Theory Event

30.4.1 general protocol properties

definition is_Says :: "event => bool" where
"is_Says ev == (EX A B X. ev = Says A B X)"

lemma is_Says_Says [iff]: "is_Says (Says A B X)"
by (simp add: is_Says_def)

definition Gets_correct :: "event list set => bool" where
"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"
apply (simp add: Gets_correct_def)
by (drule_tac x="Gets B X # evs" in spec, auto)

definition one_step :: "event list set => bool" where
"one_step p == ALL evs ev. ev#evs:p --> evs:p"

lemma one_step_Cons [dest]: "[| one_step p; ev#evs:p |] ==> evs:p"
by (unfold one_step_def, blast)

lemma one_step_app: "[| evs@evs':p; one_step p; []:p |] ==> evs':p"
by (induct evs, auto)

```
lemma trunc: "[| evs @ evs':p; one_step p |] ==> evs':p"
by (induct evs, auto)
```

```
definition has_only_Says :: "event list set => bool" where
"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"
by (unfold has_only_Says_def, blast)
```

```
lemma in_has_only_Says [dest]: "[| has_only_Says p; evs:p; ev:set evs |]
==> EX A B X. ev = Says A B X"
by (auto simp: has_only_Says_def)
```

```
lemma has_only_Says_imp_Gets_correct [simp]: "has_only_Says p
==> Gets_correct p"
by (auto simp: has_only_Says_def Gets_correct_def)
```

30.4.2 lemma on knows

```
lemma Says_imp_spies2: "Says A B {|X,Y|}:set evs ==> Y:parts (spies evs)"
by (drule Says_imp_spies, drule parts.Inj, drule parts.Snd, simp)
```

```
lemma Says_not_parts: "[| Says A B X:set evs; Y ~:parts (spies evs) |]
==> Y ~:parts {X}"
by (auto dest: Says_imp_spies parts_parts)
```

30.4.3 knows without initState

```
primrec knows' :: "agent => event list => msg set" where
  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"
```

30.4.4 decomposition of knows into knows' and initState

```
lemma knows_decomp: "knows A evs = knows' A evs Un (initState A)"
by (induct evs, auto split: event.split simp: knows.simps)
```

```

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"
by (auto simp: knows_decomp)

lemma knows'_Cons: "knows' A (ev#evs) = knows' A [ev] Un knows' A evs"
by (induct ev, auto)

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"
apply (simp only: knows_decomp)
apply (rule_tac s="(knows' A [ev] Un knows' A evs) Un initState A" in trans)
apply (simp only: knows'_Cons [of A ev evs] Un_ac)
apply blast
done

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"
by (induct evs, auto split: event.splits)

```

30.4.5 knows' is finite

```

lemma finite_knows' [iff]: "finite (knows' A evs)"
by (induct evs, auto split: event.split simp: knows.simps)

```

30.4.6 monotonicity of knows

```

lemma knows_sub_Cons: "knows A evs <= knows A (ev#evs)"
by (cases A, induct evs, auto simp: knows.simps split:event.split)

```

```

lemma knows_ConsI: "X:knows A evs ==> X:knows A (ev#evs)"
by (auto dest: knows_sub_Cons [THEN subsetD])

```

```

lemma knows_sub_app: "knows A evs <= knows A (evs @ evs')"
apply (induct evs, auto)
apply (simp add: knows_decomp)
by (case_tac a, auto simp: knows.simps)

```

30.4.7 maximum knowledge an agent can have includes messages sent to the agent

```

primrec knows_max' :: "agent => event list => msg set" where
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
  ))"

```

definition `knows_max` :: "agent => event list => msg set" where
"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"

```

30.4.8 basic facts about `knows_max`

lemma `spies_max_spies` [iff]: "spies_max evs = spies evs"
by (induct evs, auto simp: knows_max_def split: event.splits)

lemma `knows_max'_Cons`: "knows_max' A (ev#evs)
= knows_max' A [ev] Un knows_max' A evs"
by (auto split: event.splits)

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"
apply (simp add: knows_max_def del: knows_max'_def_Cons)
apply (rule_tac evs=evs in knows_max'_Cons_substI)
by blast

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)"
by (induct evs, auto split: event.split)

lemma `knows_max'_sub_spies'`: "[| evs:p; has_only_Says p; one_step p |]
==> knows_max' A evs <= spies' evs"
by (induct evs, auto split: event.splits)

lemma `knows_max'_in_spies'` [dest]: "[| evs:p; X:knows_max' A evs;
has_only_Says p; one_step p |] ==> X:spies' evs"
by (rule knows_max'_sub_spies' [THEN `subsetD`], auto)

lemma `knows_max'_app`: "knows_max' A (evs @ evs')
= knows_max' A evs Un knows_max' A evs'"

by (induct evs, auto split: event.splits)

lemma Says_to_knows_max': "Says A B X:set evs ==> X:knows_max' B evs"
by (simp add: in_set_conv_decomp, clarify, simp add: knows_max'_app)

lemma Says_from_knows_max': "Says A B X:set evs ==> X:knows_max' A evs"
by (simp add: in_set_conv_decomp, clarify, simp add: knows_max'_app)

30.4.9 used without initState

primrec used' :: "event list => msg set" where
"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
)"

definition init :: "msg set" where
"init == used []"

lemma used_decomp: "used evs = init Un used' evs"
by (induct evs, auto simp: init_def split: event.split)

lemma used'_sub_app: "used' evs <= used' (evs@evs')"
by (induct evs, auto split: event.split)

lemma used'_parts [rule_format]: "X:used' evs ==> Y:parts {X} --> Y:used' evs"
apply (induct evs, simp)
apply (case_tac a, simp_all)
apply (blast dest: parts_trans)+
done

30.4.10 monotonicity of used

lemma used_sub_Cons: "used evs <= used (ev#evs)"
by (induct evs, (induct ev, auto)+)

lemma used_ConsI: "X:used evs ==> X:used (ev#evs)"
by (auto dest: used_sub_Cons [THEN subsetD])

lemma notin_used_ConsD: "X ~:used (ev#evs) ==> X ~:used evs"
by (auto dest: used_sub_Cons [THEN subsetD])

lemma used_appD [dest]: "X:used (evs @ evs') ==> X:used evs | X:used evs'"
by (induct evs, auto, case_tac a, auto)

lemma used_ConsD: "X:used (ev#evs) ==> X:used [ev] | X:used evs"
by (case_tac ev, auto)

lemma used_sub_app: "used evs <= used (evs@evs')"
by (auto simp: used_decomp dest: used'_sub_app [THEN subsetD])

```
lemma used_appIL: "X:used evs ==> X:used (evs' @ evs)"
by (induct evs', auto intro: used_ConsI)
```

```
lemma used_appIR: "X:used evs ==> X:used (evs @ evs')"
by (erule used_sub_app [THEN subsetD])
```

```
lemma used_parts: "[| X:parts {Y}; Y:used evs |] ==> X:used evs"
apply (auto simp: used_decomp dest: used'_parts)
by (auto simp: init_def used_Nil dest: parts_trans)
```

```
lemma parts_Says_used: "[| Says A B X:set evs; Y:parts {X} |] ==> Y:used
evs"
by (induct evs, simp_all, safe, auto intro: used_ConsI)
```

```
lemma parts_used_app: "X:parts {Y} ==> X:used (evs @ Says A B Y # evs')"
apply (drule_tac evs="[Says A B Y]" in used_parts, simp, blast)
apply (drule_tac evs'=evs' in used_appIR)
apply (drule_tac evs'=evs in used_appIL)
by simp
```

30.4.11 lemmas on used and knows

```
lemma initState_used: "X:parts (initState A) ==> X:used evs"
by (induct evs, auto simp: used.simps split: event.split)
```

```
lemma Says_imp_used: "Says A B X:set evs ==> parts {X} <= used evs"
by (induct evs, auto intro: used_ConsI)
```

```
lemma not_used_not_spied: "X ~:used evs ==> X ~:parts (spies evs)"
by (induct evs, auto simp: used_Nil)
```

```
lemma not_used_not_parts: "[| Y ~:used evs; Says A B X:set evs |]
==> Y ~:parts {X}"
by (induct evs, auto intro: used_ConsI)
```

```
lemma not_used_parts_false: "[| X ~:used evs; Y:parts (spies evs) |]
==> X ~:parts {Y}"
by (auto dest: parts_parts)
```

```
lemma known_used [rule_format]: "[| evs:p; Gets_correct p; one_step p |]
==> X:parts (knows A evs) --> X:used evs"
apply (case_tac "A=Spy", blast)
apply (induct evs)
apply (simp add: used.simps, blast)
apply (frule_tac ev=a and evs=evs in one_step_Cons, simp, clarify)
apply (drule_tac P="%G. X:parts G" in knows_Cons_substD, safe)
apply (erule initState_used)
apply (case_tac a, auto)
apply (drule_tac B=A and X=msg and evs=evs in Gets_correct_Says)
by (auto dest: Says_imp_used intro: used_ConsI)
```

```
lemma known_max_used [rule_format]: "[| evs:p; Gets_correct p; one_step p
|]
==> X:parts (knows_max A evs) --> X:used evs"
```

```

apply (case_tac "A=Spy")
apply force
apply (induct evs)
apply (simp add: knows_max_def used.simps, blast)
apply (frule_tac ev=a and evs=evs in one_step_Cons, simp, clarify)
apply (drule_tac P="%G. X:parts G" in knows_max_Cons_substD, safe)
apply (case_tac a, auto)
apply (drule_tac B=A and X=msg and evs=evs in Gets_correct_Says)
by (auto simp: knows_max'_Cons dest: Says_imp_used intro: used_ConsI)

```

```

lemma not_used_not_known: "[| evs:p; X ~:used evs;
Gets_correct p; one_step p |] ==> X ~:parts (knows A evs)"
by (case_tac "A=Spy", auto dest: not_used_not_spied known_used)

```

```

lemma not_used_not_known_max: "[| evs:p; X ~:used evs;
Gets_correct p; one_step p |] ==> X ~:parts (knows_max A evs)"
by (case_tac "A=Spy", auto dest: not_used_not_spied known_max_used)

```

30.4.12 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'"
by (drule not_used_not_spied, auto dest: Says_imp_knows_Spy parts_sub)

```

```

lemma Key_neq [dest]: "[| Key n' ~:used evs;
Says A B X:set evs; Key n:parts {X} |] ==> n ~= n'"
by (drule not_used_not_spied, auto dest: Says_imp_knows_Spy parts_sub)

```

30.4.13 message of an event

```

primrec msg :: "event => msg"

```

```

where

```

```

  "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"

```

```

by (induct ev, auto)

```

```

end

```

31 Decomposition of Analz into two parts

```

theory Analz imports Extensions begin

```

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

31.1 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"

```

31.2 basic facts about pparts

```

lemma pparts_is_MPair [dest]: "X:pparts H ==> is_MPair X"
by (erule pparts.induct, auto)

```

```

lemma Crypt_notin_pparts [iff]: "Crypt K X ~:pparts H"
by auto

```

```

lemma Key_notin_pparts [iff]: "Key K ~:pparts H"
by auto

```

```

lemma Nonce_notin_pparts [iff]: "Nonce n ~:pparts H"
by auto

```

```

lemma Number_notin_pparts [iff]: "Number n ~:pparts H"
by auto

```

```

lemma Agent_notin_pparts [iff]: "Agent A ~:pparts H"
by auto

```

```

lemma pparts_empty [iff]: "pparts {} = {}"
by (auto, erule pparts.induct, auto)

```

```

lemma pparts_insertI [intro]: "X:pparts H ==> X:pparts (insert Y H)"
by (erule pparts.induct, auto)

```

```

lemma pparts_sub: "[| X:pparts G; G<=H |] ==> X:pparts H"
by (erule pparts.induct, auto)

```

```

lemma pparts_insert2 [iff]: "pparts (insert X (insert Y H))
= pparts {X} Un pparts {Y} Un pparts H"
by (rule eq, (erule pparts.induct, auto)+)

```

```

lemma pparts_insert_MPair [iff]: "pparts (insert {|X,Y|} H)
= insert {|X,Y|} (pparts ({X,Y} Un H))"
apply (rule eq, (erule pparts.induct, auto)+)
apply (rule_tac Y=Y in pparts.Fst, auto)
apply (erule pparts.induct, auto)
by (rule_tac X=X in pparts.Snd, auto)

```

```

lemma pparts_insert_Nonce [iff]: "pparts (insert (Nonce n) H) = pparts H"
by (rule eq, erule pparts.induct, auto)

```

```

lemma pparts_insert_Crypt [iff]: "pparts (insert (Crypt K X) H) = pparts
H"
by (rule eq, erule pparts.induct, auto)

```

```

lemma pparts_insert_Key [iff]: "pparts (insert (Key K) H) = pparts H"

```

```

by (rule eq, erule pparts.induct, auto)

lemma pparts_insert_Agent [iff]: "pparts (insert (Agent A) H) = pparts H"
by (rule eq, erule pparts.induct, auto)

lemma pparts_insert_Number [iff]: "pparts (insert (Number n) H) = pparts
H"
by (rule eq, erule pparts.induct, auto)

lemma pparts_insert_Hash [iff]: "pparts (insert (Hash X) H) = pparts H"
by (rule eq, erule pparts.induct, auto)

lemma pparts_insert: "X:pparts (insert Y H) ==> X:pparts {Y} Un pparts H"
by (erule pparts.induct, blast+)

lemma insert_pparts: "X:pparts {Y} Un pparts H ==> X:pparts (insert Y H)"
by (safe, erule pparts.induct, auto)

lemma pparts_Un [iff]: "pparts (G Un H) = pparts G Un pparts H"
by (rule eq, erule pparts.induct, auto dest: pparts_sub)

lemma pparts_pparts [iff]: "pparts (pparts H) = pparts H"
by (rule eq, erule pparts.induct, auto)

lemma pparts_insert_eq: "pparts (insert X H) = pparts {X} Un pparts H"
by (rule_tac A=H in insert_Un, rule pparts_Un)

lemmas pparts_insert_substI = pparts_insert_eq [THEN ssubst]

lemma in_pparts: "Y:pparts H ==> EX X. X:H & Y:pparts {X}"
by (erule pparts.induct, auto)

```

31.3 facts about *pparts* and *parts*

```

lemma pparts_no_Nonce [dest]: "[| X:pparts {Y}; Nonce n ~:parts {Y} |]
==> Nonce n ~:parts {X}"
by (erule pparts.induct, simp_all)

```

31.4 facts about *pparts* and *analz*

```

lemma pparts_analz: "X:pparts H ==> X:analz H"
by (erule pparts.induct, auto)

lemma pparts_analz_sub: "[| X:pparts G; G<=H |] ==> X:analz H"
by (auto dest: pparts_sub pparts_analz)

```

31.5 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"
```

31.6 basic facts about kparts

```
lemma kparts_not_MPair [dest]: "X:kparts H ==> not_MPair X"
by (erule kparts.induct, auto)
```

```
lemma kparts_empty [iff]: "kparts {} = {}"
by (rule eq, erule kparts.induct, auto)
```

```
lemma kparts_insertI [intro]: "X:kparts H ==> X:kparts (insert Y H)"
by (erule kparts.induct, auto dest: pparts_insertI)
```

```
lemma kparts_insert2 [iff]: "kparts (insert X (insert Y H))
= kparts {X} Un kparts {Y} Un kparts H"
by (rule eq, (erule kparts.induct, auto)+)
```

```
lemma kparts_insert_MPair [iff]: "kparts (insert {X,Y} H)
= kparts ({X,Y} Un H)"
by (rule eq, (erule kparts.induct, auto)+)
```

```
lemma kparts_insert_Nonce [iff]: "kparts (insert (Nonce n) H)
= insert (Nonce n) (kparts H)"
by (rule eq, erule kparts.induct, auto)
```

```
lemma kparts_insert_Crypt [iff]: "kparts (insert (Crypt K X) H)
= insert (Crypt K X) (kparts H)"
by (rule eq, erule kparts.induct, auto)
```

```
lemma kparts_insert_Key [iff]: "kparts (insert (Key K) H)
= insert (Key K) (kparts H)"
by (rule eq, erule kparts.induct, auto)
```

```
lemma kparts_insert_Agent [iff]: "kparts (insert (Agent A) H)
= insert (Agent A) (kparts H)"
by (rule eq, erule kparts.induct, auto)
```

```
lemma kparts_insert_Number [iff]: "kparts (insert (Number n) H)
= insert (Number n) (kparts H)"
by (rule eq, erule kparts.induct, auto)
```

```
lemma kparts_insert_Hash [iff]: "kparts (insert (Hash X) H)
= insert (Hash X) (kparts H)"
by (rule eq, erule kparts.induct, auto)
```

```
lemma kparts_insert: "X:kparts (insert X H) ==> X:kparts {X} Un kparts H"
by (erule kparts.induct, (blast dest: pparts_insert)+)
```

```
lemma kparts_insert_fst [rule_format,dest]: "X:kparts (insert Z H) ==>
X ~:kparts H --> X:kparts {Z}"
by (erule kparts.induct, (blast dest: pparts_insert)+)
```

```
lemma kparts_sub: "[| X:kparts G; G<=H |] ==> X:kparts H"
by (erule kparts.induct, auto dest: pparts_sub)
```

lemma *kparts_Un* [iff]: "kparts (G Un H) = kparts G Un kparts H"
 by (rule eq, erule *kparts.induct*, auto dest: *kparts_sub*)

lemma *pparts_kparts* [iff]: "pparts (kparts H) = {}"
 by (rule eq, erule *pparts.induct*, auto)

lemma *kparts_kparts* [iff]: "kparts (kparts H) = kparts H"
 by (rule eq, erule *kparts.induct*, auto)

lemma *kparts_insert_eq*: "kparts (insert X H) = kparts {X} Un kparts H"
 by (rule_tac A=H in *insert_Un*, rule *kparts_Un*)

lemmas *kparts_insert_substI* = *kparts_insert_eq* [THEN *ssubst*]

lemma *in_kparts*: "Y:kparts H ==> EX X. X:H & Y:kparts {X}"
 by (erule *kparts.induct*, auto dest: *in_pparts*)

lemma *kparts_has_no_pair* [iff]: "has_no_pair (kparts H)"
 by auto

31.7 facts about *kparts* and *parts*

lemma *kparts_no_Nonce* [dest]: "[| X:kparts {Y}; Nonce n ~:parts {Y} |]
 ==> Nonce n ~:parts {X}"
 by (erule *kparts.induct*, auto)

lemma *kparts_parts*: "X:kparts H ==> X:parts H"
 by (erule *kparts.induct*, auto dest: *pparts_analz*)

lemma *parts_kparts*: "X:parts (kparts H) ==> X:parts H"
 by (erule *parts.induct*, auto dest: *kparts_parts*
 intro: *parts.Fst* *parts.Snd* *parts.Body*)

lemma *Crypt_kparts_Nonce_parts* [dest]: "[| Crypt K Y:kparts {Z};
 Nonce n:parts {Y} |] ==> Nonce n:parts {Z}"
 by auto

31.8 facts about *kparts* and *analz*

lemma *kparts_analz*: "X:kparts H ==> X:analz H"
 by (erule *kparts.induct*, auto dest: *pparts_analz*)

lemma *kparts_analz_sub*: "[| X:kparts G; G<=H |] ==> X:analz H"
 by (erule *kparts.induct*, auto dest: *pparts_analz_sub*)

lemma *analz_kparts* [rule_format,dest]: "X:analz H ==>
 Y:kparts {X} --> Y:analz H"
 by (erule *analz.induct*, auto dest: *kparts_analz_sub*)

lemma *analz_kparts_analz*: "X:analz (kparts H) ==> X:analz H"
 by (erule *analz.induct*, auto dest: *kparts_analz*)

lemma *analz_kparts_insert*: "X:analz (kparts (insert Z H)) ==> X:analz (kparts

```

{Z} Un kparts H)"
by (rule analz_sub, auto)

lemma Nonce_kparts_synth [rule_format]: "Y:synth (analz G)
==> Nonce n:kparts {Y} --> Nonce n:analz G"
by (erule synth.induct, auto)

lemma kparts_insert_synth: "[| Y:parts (insert X G); X:synth (analz G);
Nonce n:kparts {Y}; Nonce n ~:analz G |] ==> Y:parts G"
apply (drule parts_insert_substD, clarify)
apply (drule in_sub, drule_tac X=Y in parts_sub, simp)
apply (auto dest: Nonce_kparts_synth)
done

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"
by (metis Fake_parts_insert_in_Un Nonce_kparts_synth UnE analz_conj_parts
synth_simps(5))

```

31.9 *analz* is *pparts* + *analz* of *kparts*

```

lemma analz_pparts_kparts: "X:analz H ==> X:pparts H | X:analz (kparts H)"
by (erule analz.induct, auto)

lemma analz_pparts_kparts_eq: "analz H = pparts H Un analz (kparts H)"
by (rule eq, auto dest: analz_pparts_kparts pparts_analz analz_kparts_analz)

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

```

32 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"

```

32.1 basic facts about *guard*

lemma *Key_is_guard* [iff]: "Key K :guard n Ks "
by auto

lemma *Agent_is_guard* [iff]: "Agent A :guard n Ks "
by auto

lemma *Number_is_guard* [iff]: "Number r :guard n Ks "
by auto

lemma *Nonce_notin_guard*: " X :guard n Ks \implies $X \not\sim$ Nonce n "
by (erule guard.induct, auto)

lemma *Nonce_notin_guard_iff* [iff]: "Nonce $n \sim$:guard n Ks "
by (auto dest: Nonce_notin_guard)

lemma *guard_has_Crypt* [rule_format]: " X :guard n Ks \implies Nonce n :parts $\{X\}$
 \longrightarrow (EX K Y . Crypt K Y :kparts $\{X\}$ & Nonce n :parts $\{Y\}$)"
by (erule guard.induct, auto)

lemma *Nonce_notin_kparts_msg*: " X :guard n Ks \implies Nonce $n \sim$:kparts $\{X\}$ "
by (erule guard.induct, auto)

lemma *Nonce_in_kparts_imp_no_guard*: "Nonce n :kparts H
 \implies EX X . $X:H$ & $X \sim$:guard n Ks "
apply (drule in_kparts, clarify)
apply (rule_tac $x=X$ in exI, clarify)
by (auto dest: Nonce_notin_kparts_msg)

lemma *guard_kparts* [rule_format]: " X :guard n Ks \implies
 Y :kparts $\{X\}$ \longrightarrow Y :guard n Ks "
by (erule guard.induct, auto)

lemma *guard_Crypt*: "[| Crypt K Y :guard n Ks ; $K \sim$:invKey' Ks |] \implies Y :guard
 n Ks "
by (ind_cases "Crypt K Y :guard n Ks ", auto)

lemma *guard_MPair* [iff]: " $\{X, Y\}$:guard n Ks = (X :guard n Ks & Y :guard
 n Ks)"
by (auto, (ind_cases " $\{X, Y\}$:guard n Ks ", auto)+)

lemma *guard_not_guard* [rule_format]: " X :guard n Ks \implies
Crypt K Y :kparts $\{X\}$ \longrightarrow Nonce n :kparts $\{Y\}$ \longrightarrow $Y \sim$:guard n Ks "
by (erule guard.induct, auto dest: guard_kparts)

lemma *guard_extand*: "[| X :guard n Ks ; $Ks \leq Ks'$ |] \implies X :guard n Ks' "
by (erule guard.induct, auto)

32.2 guarded sets

definition *Guard* :: "nat \Rightarrow key set \Rightarrow msg set \Rightarrow bool" where
"Guard n Ks H == ALL X . $X:H$ \longrightarrow X :guard n Ks "

32.3 basic facts about Guard

```

lemma Guard_empty [iff]: "Guard n Ks {}"
by (simp add: Guard_def)

lemma notin_parts_Guard [intro]: "Nonce n ~:parts G ==> Guard n Ks G"
apply (unfold Guard_def, clarify)
apply (subgoal_tac "Nonce n ~:parts {X}")
by (auto dest: parts_sub)

lemma Nonce_notin_kparts [simplified]: "Guard n Ks H ==> Nonce n ~:kparts
H"
by (auto simp: Guard_def dest: in_kparts Nonce_notin_kparts_msg)

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"
apply (drule_tac P="%G. Nonce n:G" in analz_pparts_kparts_substD, simp)
by (drule must_decrypt, auto dest: Nonce_notin_kparts)

lemma Guard_kparts [intro]: "Guard n Ks H ==> Guard n Ks (kparts H)"
by (auto simp: Guard_def dest: in_kparts guard_kparts)

lemma Guard_mono: "[| Guard n Ks H; G <= H |] ==> Guard n Ks G"
by (auto simp: Guard_def)

lemma Guard_insert [iff]: "Guard n Ks (insert X H)
= (Guard n Ks H & X:guard n Ks)"
by (auto simp: Guard_def)

lemma Guard_Un [iff]: "Guard n Ks (G Un H) = (Guard n Ks G & Guard n Ks H)"
by (auto simp: Guard_def)

lemma Guard_synth [intro]: "Guard n Ks G ==> Guard n Ks (synth G)"
by (auto simp: Guard_def, erule synth.induct, auto)

lemma Guard_analz [intro]: "[| Guard n Ks G; ALL K. K:Ks --> Key K ~:analz
G |]
==> Guard n Ks (analz G)"
apply (auto simp: Guard_def)
apply (erule analz.induct, auto)
by (ind_cases "Crypt K Xa:guard n Ks" for K Xa, auto)

lemma in_Guard [dest]: "[| X:G; Guard n Ks G |] ==> X:guard n Ks"
by (auto simp: Guard_def)

lemma in_synth_Guard: "[| X:synth G; Guard n Ks G |] ==> X:guard n Ks"
by (drule Guard_synth, auto)

lemma in_analz_Guard: "[| X:analz G; Guard n Ks G;
ALL K. K:Ks --> Key K ~:analz G |] ==> X:guard n Ks"
by (drule Guard_analz, auto)

lemma Guard_keyset [simp]: "keyset G ==> Guard n Ks G"
by (auto simp: Guard_def)

```

```
lemma Guard_Un_keyset: "[| Guard n Ks G; keyset H |] ==> Guard n Ks (G Un
H)"
by auto
```

```
lemma in_Guard_kparts: "[| X:G; Guard n Ks G; Y:kparts {X} |] ==> Y:guard
n Ks"
by blast
```

```
lemma in_Guard_kparts_neq: "[| X:G; Guard n Ks G; Nonce n':kparts {X} |]
==> n ~= n'"
by (blast dest: in_Guard_kparts)
```

```
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"
apply (drule in_Guard, simp)
apply (frule guard_not_guard, simp+)
apply (drule guard_kparts, simp)
by (ind_cases "Crypt K Y:guard n Ks", auto)
```

```
lemma Guard_extand: "[| Guard n Ks G; Ks <= Ks' |] ==> Guard n Ks' G"
by (auto simp: Guard_def dest: guard_extand)
```

```
lemma guard_invKey [rule_format]: "[| X:guard n Ks; Nonce n:kparts {Y} |]
==>
Crypt K Y:kparts {X} --> invKey K:Ks"
by (erule guard.induct, auto)
```

```
lemma Crypt_guard_invKey [rule_format]: "[| Crypt K Y:guard n Ks;
Nonce n:kparts {Y} |] ==> invKey K:Ks"
by (auto dest: guard_invKey)
```

32.4 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)"
apply (drule_tac P="%H. Nonce n:analz H" in ssubst [OF insert_Diff])
apply assumption
apply (simp only: analz_Crypt_if, simp)
done
```

```
lemma parts_decrypt: "[| Crypt K Y:H; X:parts (decrypt H K Y) |] ==> X:parts
H"
by (erule parts.induct, auto intro: parts.Fst parts.Snd parts.Body)
```

32.5 number of Crypt's in a message

```
fun crypt_nb :: "msg => nat"
where
  "crypt_nb (Crypt K X) = Suc (crypt_nb X)"
```

```
| "crypt_nb {|X,Y|} = crypt_nb X + crypt_nb Y"
| "crypt_nb X = 0"
```

32.6 basic facts about `crypt_nb`

```
lemma non_empty_crypt_msg: "Crypt K Y:parts {X} ==> crypt_nb X ≠ 0"
by (induct X, simp_all, safe, simp_all)
```

32.7 number of Crypt's in a message list

```
primrec cnb :: "msg list => nat"
where
  "cnb [] = 0"
| "cnb (X#l) = crypt_nb X + cnb l"
```

32.8 basic facts about `cnb`

```
lemma cnb_app [simp]: "cnb (l @ l') = cnb l + cnb l'"
by (induct l, auto)
```

```
lemma mem_cnb_minus: "x ∈ set l ==> cnb l = crypt_nb x + (cnb l - crypt_nb x)"
by (induct l) auto
```

```
lemmas mem_cnb_minus_substI = mem_cnb_minus [THEN ssubst]
```

```
lemma cnb_minus [simp]: "x ∈ set l ==> cnb (remove l x) = cnb l - crypt_nb x"
apply (induct l, auto)
apply (erule_tac l=l and x=x in mem_cnb_minus_substI)
apply simp
done
```

```
lemma parts_cnb: "Z:parts (set l) ==>
cnb l = (cnb l - crypt_nb Z) + crypt_nb Z"
by (erule parts.induct, auto simp: in_set_conv_decomp)
```

```
lemma non_empty_crypt: "Crypt K Y:parts (set l) ==> cnb l ≠ 0"
by (induct l, auto dest: non_empty_crypt_msg parts_insert_substD)
```

32.9 list of kparts

```
lemma kparts_msg_set: "EX l. kparts {X} = set l & cnb l = crypt_nb X"
apply (induct X, simp_all)
apply (rule_tac x="[Agent agent]" in exI, simp)
apply (rule_tac x="[Number nat]" in exI, simp)
apply (rule_tac x="[Nonce nat]" in exI, simp)
apply (rule_tac x="[Key nat]" in exI, simp)
apply (rule_tac x="[Hash X]" in exI, simp)
apply (clarify, rule_tac x="l@la" in exI, simp)
by (clarify, rule_tac x="[Crypt nat X]" in exI, simp)
```

```
lemma kparts_set: "EX l'. kparts (set l) = set l' & cnb l' = cnb l"
apply (induct l)
```

```

apply (rule_tac x="[]" in exI, simp, clarsimp)
apply (subgoal_tac "EX l''. kparts {a} = set l'' & cnb l'' = crypt_nb a",
clarify)
apply (rule_tac x="l''@l'" in exI, simp)
apply (rule kparts_insert_substI, simp)
by (rule kparts_msg_set)

```

32.10 list corresponding to "decrypt"

```

definition decrypt' :: "msg list => key => msg => msg list" where
"decrypt' l K Y == Y # remove l (Crypt K Y)"

```

```

declare decrypt'_def [simp]

```

32.11 basic facts about decrypt'

```

lemma decrypt_minus: "decrypt (set l) K Y <= set (decrypt' l K Y)"
by (induct l, auto)

```

32.12 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))"
apply (induct p)

```

```

apply (clarify, drule Guard_must_decrypt, simp, clarify)
apply (drule kparts_parts, drule non_empty_crypt, simp)

```

```

apply (clarify, frule Guard_must_decrypt, simp, clarify)
apply (drule_tac P="%G. Nonce n:G" in analz_pparts_kparts_substD, simp)
apply (frule analz_decrypt, simp_all)
apply (subgoal_tac "EX l'. kparts (set l) = set l' & cnb l' = cnb l", clarsimp)
apply (drule_tac G="insert Y (set l' - {Crypt K Y})"
and H="set (decrypt' l' K Y)" in analz_sub, rule decrypt_minus)
apply (rule_tac analz_pparts_kparts_substI, simp)
apply (case_tac "K:invKey'Ks")

```

```

apply (clarsimp, blast)

```

```

apply (subgoal_tac "Guard n Ks (set (decrypt' l' K Y))")
apply (drule_tac x="decrypt' l' K Y" in spec, simp)
apply (subgoal_tac "Crypt K Y:parts (set l)")
apply (drule parts_cnb, rotate_tac -1, simp)
apply (clarify, drule_tac X="Key Ka" and H="insert Y (set l')" in analz_sub)
apply (rule insert_mono, rule set_remove)
apply (simp add: analz_insertD, blast)

```

```

apply (blast dest: kparts_parts)

```

```

apply (rule_tac H="insert Y (set l')" in Guard_mono)
apply (subgoal_tac "Guard n Ks (set l')", simp)
apply (rule_tac K=K in guard_Crypt, simp add: Guard_def, simp)

```

32.13 if the analyse of a finite guarded set and a (possibly infinite) set of keys gives n then it must also gives Ks 367

```

apply (drule_tac t="set l'" in sym, simp)
apply (rule Guard_kparts, simp, simp)
apply (rule_tac B="set l'" in subset_trans, rule set_remove, blast)
by (rule kparts_set)

lemma Guard_invKey_finite: "[| Nonce n:analz G; Guard n Ks G; finite G |]
==> EX K. K:Ks & Key K:analz G"
apply (drule finite_list, clarify)
by (rule Guard_invKey_by_list, auto)

lemma Guard_invKey: "[| Nonce n:analz G; Guard n Ks G |]
==> EX K. K:Ks & Key K:analz G"
by (auto dest: analz_needs_only_finite Guard_invKey_finite)

```

32.13 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)"
apply (frule_tac P="%G. Nonce n:G" and G=G in analz_keyset_substD, simp_all)
apply (drule_tac G="G Un (H Int keysfor G)" in Guard_invKey_finite)
by (auto simp: Guard_def intro: analz_sub)

```

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"
by auto

```

```

lemma Agent_is_guardK [iff]: "Agent A:guardK n Ks"
by auto

```

```
lemma Number_is_guardK [iff]: "Number r:guardK n Ks"
by auto
```

```
lemma Key_notin_guardK: "X:guardK n Ks ==> X ~ = Key n"
by (erule guardK.induct, auto)
```

```
lemma Key_notin_guardK_iff [iff]: "Key n ~:guardK n Ks"
by (auto dest: Key_notin_guardK)
```

```
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})"
by (erule guardK.induct, auto)
```

```
lemma Key_notin_kparts_msg: "X:guardK n Ks ==> Key n ~:kparts {X}"
by (erule guardK.induct, auto dest: kparts_parts)
```

```
lemma Key_in_kparts_imp_no_guardK: "Key n:kparts H
==> EX X. X:H & X ~:guardK n Ks"
apply (drule in_kparts, clarify)
apply (rule_tac x=X in exI, clarify)
by (auto dest: Key_notin_kparts_msg)
```

```
lemma guardK_kparts [rule_format]: "X:guardK n Ks ==>
Y:kparts {X} --> Y:guardK n Ks"
by (erule guardK.induct, auto dest: kparts_parts parts_sub)
```

```
lemma guardK_Crypt: "[| Crypt K Y:guardK n Ks; K ~:invKey'Ks |] ==> Y:guardK
n Ks"
by (ind_cases "Crypt K Y:guardK n Ks", auto)
```

```
lemma guardK_MPair [iff]: "({|X,Y|}:guardK n Ks)
= (X:guardK n Ks & Y:guardK n Ks)"
by (auto, (ind_cases "{|X,Y|}:guardK n Ks", auto)+)
```

```
lemma guardK_not_guardK [rule_format]: "X:guardK n Ks ==>
Crypt K Y:kparts {X} --> Key n:kparts {Y} --> Y ~:guardK n Ks"
by (erule guardK.induct, auto dest: guardK_kparts)
```

```
lemma guardK_extand: "[| X:guardK n Ks; Ks <= Ks';
[| K:Ks'; K ~:Ks |] ==> Key K ~:parts {X} |] ==> X:guardK n Ks'"
by (erule guardK.induct, auto)
```

33.2 guarded sets

```
definition GuardK :: "nat => key set => msg set => bool" where
"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 {}"
by (simp add: GuardK_def)
```

```
lemma Key_notin_kparts [simplified]: "GuardK n Ks H ==> Key n ~:kparts H"
```

```

by (auto simp: GuardK_def dest: in_kparts Key_notin_kparts_msg)

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"
apply (drule_tac P="%G. Key n:G" in analz_pparts_kparts_substD, simp)
by (drule must_decrypt, auto dest: Key_notin_kparts)

lemma GuardK_kparts [intro]: "GuardK n Ks H ==> GuardK n Ks (kparts H)"
by (auto simp: GuardK_def dest: in_kparts guardK_kparts)

lemma GuardK_mono: "[| GuardK n Ks H; G <= H |] ==> GuardK n Ks G"
by (auto simp: GuardK_def)

lemma GuardK_insert [iff]: "GuardK n Ks (insert X H)
= (GuardK n Ks H & X:guardK n Ks)"
by (auto simp: GuardK_def)

lemma GuardK_Un [iff]: "GuardK n Ks (G Un H) = (GuardK n Ks G & GuardK n
Ks H)"
by (auto simp: GuardK_def)

lemma GuardK_synth [intro]: "GuardK n Ks G ==> GuardK n Ks (synth G)"
by (auto simp: GuardK_def, erule synth.induct, auto)

lemma GuardK_analz [intro]: "[| GuardK n Ks G; ALL K. K:Ks --> Key K ~:analz
G |]
==> GuardK n Ks (analz G)"
apply (auto simp: GuardK_def)
apply (erule analz.induct, auto)
by (ind_cases "Crypt K Xa:guardK n Ks" for K Xa, auto)

lemma in_GuardK [dest]: "[| X:G; GuardK n Ks G |] ==> X:guardK n Ks"
by (auto simp: GuardK_def)

lemma in_synth_GuardK: "[| X:synth G; GuardK n Ks G |] ==> X:guardK n Ks"
by (drule GuardK_synth, auto)

lemma in_analz_GuardK: "[| X:analz G; GuardK n Ks G;
ALL K. K:Ks --> Key K ~:analz G |] ==> X:guardK n Ks"
by (drule GuardK_analz, auto)

lemma GuardK_keyset [simp]: "[| keyset G; Key n ~:G |] ==> GuardK n Ks G"
by (simp only: GuardK_def, clarify, drule keyset_in, auto)

lemma GuardK_Un_keyset: "[| GuardK n Ks G; keyset H; Key n ~:H |]
==> GuardK n Ks (G Un H)"
by auto

lemma in_GuardK_kparts: "[| X:G; GuardK n Ks G; Y:kparts {X} |] ==> Y:guardK
n Ks"
by blast

lemma in_GuardK_kparts_neq: "[| X:G; GuardK n Ks G; Key n':kparts {X} |]
==> n ~ = n'"

```

by (blast dest: in_GuardK_kparts)

```
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"
apply (drule in_GuardK, simp)
apply (frule guardK_not_guardK, simp+)
apply (drule guardK_kparts, simp)
by (ind_cases "Crypt K Y:guardK n Ks", auto)
```

```
lemma GuardK_extand: "[| GuardK n Ks G; Ks <= Ks';
| K:Ks'; K ~:Ks |] ==> Key K ~:parts G |] ==> GuardK n Ks' G"
by (auto simp: GuardK_def dest: guardK_extand parts_sub)
```

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)"
apply (drule_tac P="%H. Key n:analz H" in ssubst [OF insert_Diff])
apply assumption
apply (simp only: analz_Crypt_if, simp)
done
```

```
lemma parts_decrypt: "[| Crypt K Y:H; X:parts (decrypt H K Y) |] ==> X:parts
H"
by (erule parts.induct, auto intro: parts.Fst parts.Snd parts.Body)
```

33.5 number of Crypt's in a message

```
fun crypt_nb :: "msg => nat" where
"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"
by (induct X, simp_all, safe, simp_all)
```

33.7 number of Crypt's in a message list

```
primrec cnb :: "msg list => nat" where
"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'"
by (induct l, auto)
```

```
lemma mem_cnb_minus: "x ∈ set l ==> cnb l = crypt_nb x + (cnb l - crypt_nb
x)"
```

by (induct l, auto)

lemmas mem_cnb_minus_substI = mem_cnb_minus [THEN ssubst]

lemma cnb_minus [simp]: "x ∈ set l ==> cnb (remove l x) = cnb l - crypt_nb x"

apply (induct l, auto)

by (erule_tac l=l and x=x in mem_cnb_minus_substI, simp)

lemma parts_cnb: "Z:parts (set l) ==>

cnb l = (cnb l - crypt_nb Z) + crypt_nb Z"

by (erule parts.induct, auto simp: in_set_conv_decomp)

lemma non_empty_crypt: "Crypt K Y:parts (set l) ==> cnb l ≠ 0"

by (induct l, auto dest: non_empty_crypt_msg parts_insert_substD)

33.9 list of kparts

lemma kparts_msg_set: "EX l. kparts {X} = set l & cnb l = crypt_nb X"

apply (induct X, simp_all)

apply (rule_tac x="[Agent agent]" in exI, simp)

apply (rule_tac x="[Number nat]" in exI, simp)

apply (rule_tac x="[Nonce nat]" in exI, simp)

apply (rule_tac x="[Key nat]" in exI, simp)

apply (rule_tac x="[Hash X]" in exI, simp)

apply (clarify, rule_tac x="l@la" in exI, simp)

by (clarify, rule_tac x="[Crypt nat X]" in exI, simp)

lemma kparts_set: "EX l'. kparts (set l) = set l' & cnb l' = cnb l"

apply (induct l)

apply (rule_tac x="[]" in exI, simp, clarsimp)

apply (subgoal_tac "EX l''. kparts {a} = set l'' & cnb l'' = crypt_nb a",
clarify)

apply (rule_tac x="l''@l'" in exI, simp)

apply (rule kparts_insert_substI, simp)

by (rule kparts_msg_set)

33.10 list corresponding to "decrypt"

definition decrypt' :: "msg list => key => msg => msg list" where
"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)"

by (induct l, auto)

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))"

```

apply (induct p)

apply (clarify, drule GuardK_must_decrypt, simp, clarify)
apply (drule kparts_parts, drule non_empty_crypt, simp)

apply (clarify, frule GuardK_must_decrypt, simp, clarify)
apply (drule_tac P="%G. Key n:G" in analz_pparts_kparts_substD, simp)
apply (frule analz_decrypt, simp_all)
apply (subgoal_tac "EX l'. kparts (set l) = set l' & cnb l' = cnb l", clarsimp)
apply (drule_tac G="insert Y (set l' - {Crypt K Y})"
and H="set (decrypt' l' K Y)" in analz_sub, rule decrypt_minus)
apply (rule_tac analz_pparts_kparts_substI, simp)
apply (case_tac "K:invKey'Ks")

apply (clarsimp, blast)

apply (subgoal_tac "GuardK n Ks (set (decrypt' l' K Y))")
apply (drule_tac x="decrypt' l' K Y" in spec, simp)
apply (subgoal_tac "Crypt K Y:parts (set l)")
apply (drule parts_cnb, rotate_tac -1, simp)
apply (clarify, drule_tac X="Key Ka" and H="insert Y (set l')" in analz_sub)
apply (rule insert_mono, rule set_remove)
apply (simp add: analz_insertD, blast)

apply (blast dest: kparts_parts)

apply (rule_tac H="insert Y (set l')" in GuardK_mono)
apply (subgoal_tac "GuardK n Ks (set l')", simp)
apply (rule_tac K=K in guardK_Crypt, simp add: GuardK_def, simp)
apply (drule_tac t="set l'" in sym, simp)
apply (rule GuardK_kparts, simp, simp)
apply (rule_tac B="set l'" in subset_trans, rule set_remove, blast)
by (rule kparts_set)

lemma GuardK_invKey_finite: "[| Key n:analz G; GuardK n Ks G; finite G |]
==> EX K. K:Ks & Key K:analz G"
apply (drule finite_list, clarify)
by (rule GuardK_invKey_by_list, auto)

lemma GuardK_invKey: "[| Key n:analz G; GuardK n Ks G |]
==> EX K. K:Ks & Key K:analz G"
by (auto dest: analz_needs_only_finite GuardK_invKey_finite)

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)"
apply (frule_tac P="%G. Key n:G" and G=G in analz_keyset_substD, simp_all)
apply (drule_tac G="G Un (H Int keysfor G)" in GuardK_invKey_finite)
apply (auto simp: GuardK_def intro: analz_sub)
by (drule keyset_in, auto)

end

```

```

theory Shared
imports Event All_Symmetric
begin

consts
  shrK    :: "agent => key"

specification (shrK)
  inj_shrK: "inj shrK"
  — No two agents have the same long-term key
  apply (rule exI [of _ "agent_case 0 ( $\lambda n. n + 2$ ) 1"])
  apply (simp add: inj_on_def split: agent.split)
  done

```

Server knows all long-term keys; other agents know only their own

```

overloading
  initState  $\equiv$  initState
begin

primrec initState where
  initState_Server: "initState Server = Key ' range shrK"
| initState_Friend: "initState (Friend i) = {Key (shrK (Friend i))}"
| initState_Spy:    "initState Spy = Key 'shrK'bad"

end

```

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"
apply (insert isSym_keys)
apply (simp add: symKeys_def)
done

```

```

lemma analz_Decrypt' [dest]:
  "[| Crypt K X  $\in$  analz H; Key K  $\in$  analz H |] ==> X  $\in$  analz H"
by auto

```

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)) = {}"
apply (unfold keysFor_def)
apply (induct_tac "C", auto)
done

```

```

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

```

```

lemma Crypt_imp_keysFor: "Crypt K X ∈ H ==> K ∈ keysFor H"
by (metis Crypt_imp_invKey_keysFor invKey_K)

```

33.13 Function "knows"

```

lemma Spy_knows_Spy_bad [intro!]: "A: bad ==> Key (shrK A) ∈ knows Spy evs"
apply (induct_tac "evs")
apply (simp_all (no_asm_simp) add: imageI knows_Cons split add: event.split)
done

```

```

lemma Crypt_Spy_analz_bad: "[| Crypt (shrK A) X ∈ analz (knows Spy evs);
A: bad |]
 ==> X ∈ analz (knows Spy evs)"
by (metis Spy_knows_Spy_bad analz.Inj analz_Decrypt')

```

```

lemma shrK_in_initState [iff]: "Key (shrK A) ∈ initState A"
by (induct_tac "A", auto)

```

```

lemma shrK_in_used [iff]: "Key (shrK A) ∈ used evs"
by (rule initState_into_used, blast)

```

```

lemma Key_not_used [simp]: "Key K ∉ used evs ==> K ∉ range shrK"
by blast

```

```

lemma shrK_neq [simp]: "Key K ∉ used evs ==> shrK B ≠ K"
by blast

```

```

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)"
by (induct_tac "B", auto)

```

```

lemma Nonce_notin_used_empty [simp]: "Nonce N ∉ used []"
by (simp add: used_Nil)

```

33.15 Supply fresh nonces for possibility theorems.

```

lemma Nonce_supply_lemma: "∃N. ALL n. N<=n --> Nonce n ∉ used evs"
apply (induct_tac "evs")

```

```

apply (rule_tac x = 0 in exI)
apply (simp_all (no_asm_simp) add: used_Cons split add: event.split)
apply (metis le_sup_iff msg_Nonce_supply)
done

lemma Nonce_supply1: " $\exists N. \text{Nonce } N \notin \text{used evs}$ "
by (metis Nonce_supply_lemma order_eq_iff)

lemma Nonce_supply2: " $\exists N N'. \text{Nonce } N \notin \text{used evs} \ \& \ \text{Nonce } N' \notin \text{used evs}'$ 
 $\ \& \ N \neq N'$ "
apply (cut_tac evs = evs in Nonce_supply_lemma)
apply (cut_tac evs = "evs'" in Nonce_supply_lemma, clarify)
apply (metis Suc_n_not_le_n nat_le_linear)
done

lemma Nonce_supply3: " $\exists N N' N''. \text{Nonce } N \notin \text{used evs} \ \& \ \text{Nonce } N' \notin \text{used evs}'$ 
 $\ \&$ 
 $\ \text{Nonce } N'' \notin \text{used evs}'' \ \& \ N \neq N' \ \& \ N' \neq N'' \ \& \ N \neq N''$ "
apply (cut_tac evs = evs in Nonce_supply_lemma)
apply (cut_tac evs = "evs'" in Nonce_supply_lemma)
apply (cut_tac evs = "evs''" in Nonce_supply_lemma, clarify)
apply (rule_tac x = N in exI)
apply (rule_tac x = "Suc (N+Na)" in exI)
apply (rule_tac x = "Suc (Suc (N+Na+Nb))" in exI)
apply (simp (no_asm_simp) add: less_not_refl3 le_add1 le_add2 less_Suc_eq_le)
done

lemma Nonce_supply: "Nonce (@ N. Nonce N  $\notin$  used evs)  $\notin$  used evs"
apply (rule Nonce_supply_lemma [THEN exE])
apply (rule someI, blast)
done

```

Unlike the corresponding property of nonces, we cannot prove $\text{finite } KK \implies \exists K. K \notin KK \wedge \text{Key } K \notin \text{used evs}$. We have infinitely many agents and there is nothing to stop their long-term keys from exhausting all the natural numbers. Instead, possibility theorems must assume the existence of a few keys.

33.16 Specialized Rewriting for Theorems About `analz` and Image

```

lemma subset_CompL_range: "A <= - (range shrK) ==> shrK x  $\notin$  A"
by blast

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

lemma insert_Key_image: "insert (Key K) (Key'KK  $\cup$  C) = Key'(insert K KK)
 $\cup$  C"
by blast

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_Cmpl_range
Key_not_used insert_Key_image Un_assoc [THEN sym]

```

```

lemma analz_image_freshK_lemma:
  "(Key K ∈ analz (Key'nE ∪ H)) --> (K ∈ nE | Key K ∈ analz H) ==>
   (Key K ∈ analz (Key'nE ∪ H)) = (K ∈ nE | Key K ∈ analz H)"
by (blast intro: analz_mono [THEN [2] rev_subsetD])

```

33.17 Tactics for possibility theorems

```

ML
{*
structure Shared =
struct

(*Omitting used_Says makes the tactic much faster: it leaves expressions
  such as Nonce ?N ∉ used evs that match Nonce_supply*)
fun possibility_tac ctxt =
  (REPEAT
   (ALLGOALS (simp_tac (simpset_of ctxt
    delsimps [@{thm used_Says}, @{thm used_Notes}, @{thm used_Gets}]

    setSolver safe_solver))
   THEN
   REPEAT_FIRST (eq_assume_tac ORELSE'
    resolve_tac [refl, conjI, @{thm Nonce_supply}])))

(*For harder protocols (such as Recur) where we have to set up some
  nonces and keys initially*)
fun basic_possibility_tac ctxt =
  REPEAT
  (ALLGOALS (asm_simp_tac (simpset_of ctxt setSolver safe_solver))
   THEN
   REPEAT_FIRST (resolve_tac [refl, conjI]))

val analz_image_freshK_ss =
  @{simpset} delsimps [image_insert, image_Un]
  delsimps [@{thm imp_disjL}] (*reduces blow-up*)
  addsimps @{thms analz_image_freshK_simps}

end
*}

```

```

lemma invKey_shrK_iff [iff]:
  "(Key (invKey K) ∈ X) = (Key K ∈ X)"

```

by auto

```

method_setup analz_freshK = {*
  Scan.succeed (fn ctxt =>
    (SIMPLE_METHOD
      (EVERY [REPEAT_FIRST (resolve_tac [allI, ballI, impI]),
        REPEAT_FIRST (rtac @{thm analz_image_freshK_lemma}),
        ALLGOALS (asm_simp_tac (Simplifier.context ctxt Shared.analz_image_freshK_ss))]))))
*}
  "for proving the Session Key Compromise theorem"

method_setup possibility = {*
  Scan.succeed (fn ctxt => SIMPLE_METHOD (Shared.possibility_tac ctxt)) *}
  "for proving possibility theorems"

method_setup basic_possibility = {*
  Scan.succeed (fn ctxt => SIMPLE_METHOD (Shared.basic_possibility_tac ctxt))
*}
  "for proving possibility theorems"

lemma knows_subset_knows_Cons: "knows A evs <= knows A (e # evs)"
by (induct e) (auto simp: knows_Cons)

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

```

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

```

```

lemma agt_shrK [simp]: "agt (shrK A) = A"
by (simp add: agt_def)

```

34.1.3 basic facts about initState

```

lemma no_Crypt_in_parts_init [simp]: "Crypt K X ~:parts (initState A)"

```

```
by (cases A, auto simp: initState.simps)
```

```
lemma no_Crypt_in_analz_init [simp]: "Crypt K X ~:analz (initState A)"
by auto
```

```
lemma no_shrK_in_analz_init [simp]: "A ~:bad
==> Key (shrK A) ~:analz (initState Spy)"
by (auto simp: initState.simps)
```

```
lemma shrK_notin_initState_Friend [simp]: "A ~ = Friend C
==> Key (shrK A) ~: parts (initState (Friend C))"
by (auto simp: initState.simps)
```

```
lemma keyset_init [iff]: "keyset (initState A)"
by (cases A, auto simp: keyset_def initState.simps)
```

34.1.4 sets of symmetric keys

```
definition shrK_set :: "key set => bool" where
"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"
by (simp add: shrK_set_def)
```

```
lemma shrK_set1 [iff]: "shrK_set {shrK A}"
by (simp add: shrK_set_def)
```

```
lemma shrK_set2 [iff]: "shrK_set {shrK A, shrK B}"
by (simp add: shrK_set_def)
```

34.1.5 sets of good keys

```
definition good :: "key set => bool" where
"good Ks == ALL K. K:Ks --> agt K ~:bad"
```

```
lemma in_good: "[| good Ks; K:Ks |] ==> agt K ~:bad"
by (simp add: good_def)
```

```
lemma good1 [simp]: "A ~:bad ==> good {shrK A}"
by (simp add: good_def)
```

```
lemma good2 [simp]: "[| A ~:bad; B ~:bad |] ==> good {shrK A, shrK B}"
by (simp add: good_def)
```

34.2 Proofs About Guarded Messages

34.2.1 small hack

```
lemma shrK_is_invKey_shrK: "shrK A = invKey (shrK A)"
by simp
```

```
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}"
apply (rule shrK_is_invKey_shrK_substI, rule invKey_invKey_substI)
by (rule Guard_Nonce, simp+)
```

34.2.2 guardedness results on nonces

```
lemma guard_ciph [simp]: "shrK A:Ks ==> Ciph A X:guard n Ks"
by (rule Guard_Nonce, simp)
```

```
lemma guardK_ciph [simp]: "shrK A:Ks ==> Ciph A X:guardK n Ks"
by (rule Guard_Key, simp)
```

```
lemma Guard_init [iff]: "Guard n Ks (initState B)"
by (induct B, auto simp: Guard_def initState.simps)
```

```
lemma Guard_knows_max': "Guard n Ks (knows_max' C evs)
==> Guard n Ks (knows_max C evs)"
by (simp add: knows_max_def)
```

```
lemma Nonce_not_used_Guard_spies [dest]: "Nonce n ~:used evs
==> Guard n Ks (spies evs)"
by (auto simp: Guard_def dest: not_used_not_known parts_sub)
```

```
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)"
by (auto simp: Guard_def dest: known_used parts_trans)
```

```
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)"
by (auto simp: Guard_def dest: known_max_used parts_trans)
```

```
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)"
apply (rule_tac H="knows_max (Friend C) evs" in Guard_mono)
by (auto simp: knows_max_def)
```

34.2.3 guardedness results on keys

```
lemma GuardK_init [simp]: "n ~:range shrK ==> GuardK n Ks (initState B)"
by (induct B, auto simp: GuardK_def initState.simps)
```

```
lemma GuardK_knows_max': "[| GuardK n A (knows_max' C evs); n ~:range shrK
|]
==> GuardK n A (knows_max C evs)"
by (simp add: knows_max_def)
```

```
lemma Key_not_used_GuardK_spies [dest]: "Key n ~:used evs
==> GuardK n A (spies evs)"
by (auto simp: GuardK_def dest: not_used_not_known parts_sub)
```

```
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)"
by (auto simp: GuardK_def dest: known_used parts_trans)
```

```
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)"
by (auto simp: GuardK_def dest: known_max_used parts_trans)
```

```
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)"
apply (rule_tac H="knows_max (Friend C) evs" in GuardK_mono)
by (auto simp: knows_max_def)
```

34.2.4 regular protocols

```
definition regular :: "event list set => bool" where
"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)"
by (auto simp: regular_def)
```

```
lemma shrK_analz_iff_bad [simp]: "[| evs:p; regular p |] ==>
(Key (shrK A):analz (spies evs)) = (A:bad)"
by auto
```

```
lemma Guard_Nonce_analz: "[| Guard n Ks (spies evs); evs:p;
shrK_set Ks; good Ks; regular p |] ==> Nonce n ~:analz (spies evs)"
apply (clarify, simp only: knows_decomp)
apply (drule Guard_invKey_keyset, simp+, safe)
apply (drule in_good, simp)
apply (drule in_shrK_set, simp+, clarify)
apply (frule_tac A=A in shrK_analz_iff_bad)
by (simp add: knows_decomp)+
```

```
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)"
apply (clarify, simp only: knows_decomp)
apply (drule GuardK_invKey_keyset, clarify, simp+, simp add: initState.simps)
apply clarify
apply (drule in_good, simp)
apply (drule in_shrK_set, simp+, clarify)
apply (frule_tac A=A in shrK_analz_iff_bad)
by (simp add: knows_decomp)+
```

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"
by (erule or.induct, auto)

```

```

lemma or_is_Gets_correct [iff]: "Gets_correct or"
by (auto simp: Gets_correct_def dest: or_has_no_Gets)

```

```

lemma or_is_one_step [iff]: "one_step or"
by (unfold one_step_def, clarify, ind_cases "ev#evs:or" for ev evs, auto)

```

```

lemma or_has_only_Says' [rule_format]: "evs:or ==>
ev:set evs --> (EX A B X. ev=Says A B X)"
by (erule or.induct, auto)

```

```

lemma or_has_only_Says [iff]: "has_only_Says or"
by (auto simp: has_only_Says_def dest: or_has_only_Says')

```

35.5 or is regular

```

lemma or1'_parts_spies [dest]: "or1' A' A B NA X:set evs
==> X:parts (spies evs)"
by blast

```

```

lemma or2_parts_spies [dest]: "or2 A B NA NB X:set evs
==> X:parts (spies evs)"
by blast

```

```

lemma or3_parts_spies [dest]: "Says S B {|NA, Y, Ciph B {|NB, K|}|}:set evs
==> K:parts (spies evs)"
by blast

```

```

lemma or_is_regular [iff]: "regular or"
apply (simp only: regular_def, clarify)
apply (erule or.induct, simp_all add: initState.simps knows.simps)
by (auto dest: parts_sub)

```

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)"
apply (erule or.induct)

apply simp_all

apply (clarify, erule in_synth_GuardK, erule GuardK_analz, simp)

apply blast

apply safe
apply (blast dest: Says_imp_spies, blast)

apply blast
apply (drule_tac A=Server in Key_neq, simp+, rule No_Key, simp)
apply (drule_tac A=Server in Key_neq, simp+, rule No_Key, simp)

by (blast dest: Says_imp_spies in_GuardK_kparts)

```

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)"
apply (erule or.induct)

apply simp_all

apply safe
apply (erule in_synth_Guard, erule Guard_analz, simp)

apply (drule_tac n=NB in Nonce_neq, simp+, rule No_Nonce, simp)
apply (drule_tac n=NB in Nonce_neq, simp+, rule No_Nonce, simp)

apply blast
apply (drule_tac n=NA in Nonce_neq, simp+, rule No_Nonce, simp)
apply (blast intro!: No_Nonce dest: used_parts)
apply (drule_tac n=NA in Nonce_neq, simp+, rule No_Nonce, simp)
apply (blast intro!: No_Nonce dest: used_parts)
apply (blast dest: Says_imp_spies)
apply (blast dest: Says_imp_spies)
apply (case_tac "Ba=B", clarsimp)
apply (drule_tac n=NB and A=B in Nonce_neq, simp+)
apply (drule Says_imp_spies)
apply (drule_tac n'=NAa in in_Guard_kparts_neq, simp+, rule No_Nonce, simp)

apply (drule Says_imp_spies)
apply (frule_tac n'=NAa in in_Guard_kparts_neq, simp+, rule No_Nonce, simp)
apply (case_tac "Aa=B", clarsimp)
apply (case_tac "NAa=NB", clarsimp)
apply (drule Says_imp_spies)
apply (drule_tac Y="{|Nonce NB, Agent Aa, Agent Ba|}"
and K="shrK Aa" in in_Guard_kparts_Crypt, simp+)

apply (simp add: No_Nonce)
apply (case_tac "Ba=B", clarsimp)
apply (case_tac "NBa=NB", clarify)

```

```

apply (drule Says_imp_spies)
apply (drule_tac Y="{|Nonce NAa, Nonce NB, Agent Aa, Agent Ba|}"
      and K="shrK Ba" in in_Guard_kparts_Crypt, simp+)
apply (simp add: No_Nonce)

by (blast dest: Says_imp_spies)+

end

```

36 Yahalom Protocol

```
theory Guard_Yahalom imports "../Shared" 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"
by (erule ya.induct, auto)
```

```
lemma ya_is_Gets_correct [iff]: "Gets_correct ya"
by (auto simp: Gets_correct_def dest: ya_has_no_Gets)
```

```
lemma ya_is_one_step [iff]: "one_step ya"
by (unfold one_step_def, clarify, ind_cases "ev#evs:ya" for ev evs, auto)
```

```
lemma ya_has_only_Says' [rule_format]: "evs:ya ==>
ev:set evs --> (EX A B X. ev=Says A B X)"
by (erule ya.induct, auto)
```

```
lemma ya_has_only_Says [iff]: "has_only_Says ya"
by (auto simp: has_only_Says_def dest: ya_has_only_Says')
```

```
lemma ya_is_regular [iff]: "regular ya"
apply (simp only: regular_def, clarify)
apply (erule ya.induct, simp_all add: initState.simps knows.simps)
by (auto dest: parts_sub)
```

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)"
apply (erule ya.induct)
```

```

apply simp_all

apply (clarify, erule in_synth_GuardK, erule GuardK_analz, simp)

apply safe
apply (blast dest: Says_imp_spies)

apply blast
apply (drule_tac A=Server in Key_neq, simp+, rule No_Key, simp)
apply (drule_tac A=Server in Key_neq, simp+, rule No_Key, simp)

apply (blast dest: Says_imp_spies in_GuardK_kparts)
by blast

```

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"
by (erule ya.induct, auto)

lemma ya3_shrK: "evs:ya ==> ya3 A B NA NB (shrK C) ~:set evs"
by (blast dest: KAB_isnt_shrK)

```

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"
by (erule ya.induct, auto dest: Says_imp_spies intro: parts_parts)

lemma ya2'_imp_ya1'_parts: "[| ya2' B' A B NA NB:set evs; evs:ya; B ~:bad
|]
==> {|Agent A, Nonce NA|}:spies evs"
by (blast dest: Says_imp_spies ya2'_parts_imp_ya1'_parts)

```

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'"
apply (erule ya.induct, simp_all, clarify)
apply (drule Crypt_synth_insert, simp+)
apply (drule Crypt_synth_insert, simp+, safe)
apply (drule not_used_parts_false, simp+)+
by (drule Says_not_parts, simp+)+

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'"
by (drule NB_is_uniq_in_ya2'_parts, auto dest: Says_imp_spies)

```

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)"
apply (erule ya.induct, simp_all)
apply (clarify, drule Crypt_synth_insert, simp+)
apply (blast intro: parts_sub, blast)
by (auto dest: Says_imp_spies parts_parts)
```

```
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)"
apply (erule ya.induct, simp_all, safe)
apply (drule Crypt_synth_insert, simp+)
apply (drule Crypt_synth_insert, simp+, blast)
apply blast
apply blast
by (auto dest: Says_imp_spies2 parts_parts)
```

```
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)"
by (drule ya3'_parts_imp_ya2', auto dest: Says_imp_spies)
```

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"
apply (erule ya.induct, simp_all, safe)
apply (drule Crypt_synth_insert, simp+)
by (blast dest: Says_imp_spies2 parts_parts)
```

```
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"
by (blast dest: Says_imp_spies ya3'_parts_imp_ya3)
```

36.11 guardedness of NB

```
definition ya_keys :: "agent => agent => nat => nat => event list => key set"
where
"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)"
apply (erule ya.induct)
```

```
apply (simp_all add: ya_keys_def)
```

```
apply safe
apply (erule in_synth_Guard, erule Guard_analz, simp, clarify)
apply (frule_tac B=B in Guard_KAB, simp+)
apply (drule_tac p=ya in GuardK_Key_analz, simp+)
apply (blast dest: KAB_isnt_shrK, simp)
```

```

apply (drule_tac n=NB in Nonce_neq, simp+, rule No_Nonce, simp)

apply blast
apply (drule Says_imp_spies)
apply (drule_tac n=NB in Nonce_neq, simp+)
apply (drule_tac n'=NAa in in_Guard_kparts_neq, simp+)
apply (rule No_Nonce, simp)

apply (rule Guard_extand, simp, blast)
apply (case_tac "NAa=NB", clarify)
apply (frule Says_imp_spies)
apply (frule in_Guard_kparts_Crypt, simp+, blast, simp+)
apply (frule_tac A=A and B=B and NA=NA and NB=NB and C=Ba in ya3_shrK,
simp)
apply (drule ya2'_imp_ya1'_parts, simp, blast, blast)
apply (case_tac "NBa=NB", clarify)
apply (frule Says_imp_spies)
apply (frule in_Guard_kparts_Crypt, simp+, blast, simp+)
apply (frule_tac A=A and B=B and NA=NA and NB=NB and C=Ba in ya3_shrK,
simp)
apply (drule NB_is_uniq_in_ya2', simp+, blast, simp+)
apply (simp add: No_Nonce, blast)

apply (blast dest: Says_imp_spies)
apply (case_tac "NBa=NB", clarify)
apply (frule_tac A=S in Says_imp_spies)
apply (frule in_Guard_kparts_Crypt, simp+)
apply (blast dest: Says_imp_spies)
apply (case_tac "NBa=NB", clarify)
apply (frule_tac A=S in Says_imp_spies)
apply (frule in_Guard_kparts_Crypt, simp+, blast, simp+)
apply (frule_tac A=A and B=B and NA=NA and NB=NB and C=Aa in ya3_shrK,
simp)
apply (frule ya3'_imp_ya2', simp+, blast, clarify)
apply (frule_tac A=B' in Says_imp_spies)
apply (rotate_tac -1, frule in_Guard_kparts_Crypt, simp+, blast, simp+)
apply (frule_tac A=A and B=B and NA=NA and NB=NB and C=Ba in ya3_shrK,
simp)
apply (drule NB_is_uniq_in_ya2', simp+, blast, clarify)
apply (drule ya3'_imp_ya3, simp+)
apply (simp add: Guard_Nonce)
apply (simp add: No_Nonce)
done

end

```

37 Blanqui's "guard" concept: protocol-independent secrecy

```

theory Auth_Guard_Shared
imports
  "Guard_OtwayRees"

```

```
"Guard_Yahalom"
begin

end
```

```
theory Guard_Public imports Guard Public Extensions begin
```

37.1 Extensions to Theory Public

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

37.1.1 signature

```
definition sign :: "agent => msg => msg" where
"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')"
by (auto simp: sign_def)
```

37.1.2 agent associated to a key

```
definition agt :: "key => agent" where
"agt K == @A. K = priK A | K = pubK A"
```

```
lemma agt_priK [simp]: "agt (priK A) = A"
by (simp add: agt_def)
```

```
lemma agt_pubK [simp]: "agt (pubK A) = A"
by (simp add: agt_def)
```

37.1.3 basic facts about initState

```
lemma no_Crypt_in_parts_init [simp]: "Crypt K X ~:parts (initState A)"
by (cases A, auto simp: initState.simps)
```

```
lemma no_Crypt_in_analz_init [simp]: "Crypt K X ~:analz (initState A)"
by auto
```

```
lemma no_priK_in_analz_init [simp]: "A ~:bad
==> Key (priK A) ~:analz (initState Spy)"
by (auto simp: initState.simps)
```

```
lemma priK_notin_initState_Friend [simp]: "A ~= Friend C
==> Key (priK A) ~:parts (initState (Friend C))"
by (auto simp: initState.simps)
```

```
lemma keyset_init [iff]: "keyset (initState A)"
by (cases A, auto simp: keyset_def initState.simps)
```

37.1.4 sets of private keys

```
definition priK_set :: "key set => bool" where
"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"
by (simp add: priK_set_def)
```

```
lemma priK_set1 [iff]: "priK_set {priK A}"
by (simp add: priK_set_def)
```

```
lemma priK_set2 [iff]: "priK_set {priK A, priK B}"
by (simp add: priK_set_def)
```

37.1.5 sets of good keys

```
definition good :: "key set => bool" where
"good Ks == ALL K. K:Ks --> agt K ~:bad"
```

```
lemma in_good: "[| good Ks; K:Ks |] ==> agt K ~:bad"
by (simp add: good_def)
```

```
lemma good1 [simp]: "A ~:bad ==> good {priK A}"
by (simp add: good_def)
```

```
lemma good2 [simp]: "[| A ~:bad; B ~:bad |] ==> good {priK A, priK B}"
by (simp add: good_def)
```

37.1.6 greatest nonce used in a trace, 0 if there is no nonce

```
primrec greatest :: "event list => nat"
where
  "greatest [] = 0"
| "greatest (ev # evs) = max (greatest_msg (msg ev)) (greatest evs)"
```

```
lemma greatest_is_greatest: "Nonce n:used evs ==> n <= greatest evs"
apply (induct evs, auto simp: initState.simps)
apply (drule used_sub_parts_used, safe)
apply (drule greatest_msg_is_greatest, arith)
by simp
```

37.1.7 function giving a new nonce

```
definition new :: "event list => nat" where
"new evs == Suc (greatest evs)"
```

```
lemma new_isnt_used [iff]: "Nonce (new evs) ~:used evs"
by (clarify, drule greatest_is_greatest, auto simp: new_def)
```

37.2 Proofs About Guarded Messages

37.2.1 small hack necessary because priK is defined as the inverse of pubK

```
lemma pubK_is_invKey_priK: "pubK A = invKey (priK A)"
by simp
```

```
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}"
apply (rule pubK_is_invKey_priK_substI, rule invKey_invKey_substI)
by (rule Guard_Nonce, simp+)
```

37.2.2 guardedness results

```
lemma sign_guard [intro]: "X:guard n Ks ==> sign A X:guard n Ks"
by (auto simp: sign_def)
```

```
lemma Guard_init [iff]: "Guard n Ks (initState B)"
by (induct B, auto simp: Guard_def initState.simps)
```

```
lemma Guard_knows_max': "Guard n Ks (knows_max' C evs)
==> Guard n Ks (knows_max C evs)"
by (simp add: knows_max_def)
```

```
lemma Nonce_not_used_Guard_spies [dest]: "Nonce n ~:used evs
==> Guard n Ks (spies evs)"
by (auto simp: Guard_def dest: not_used_not_known parts_sub)
```

```
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)"
by (auto simp: Guard_def dest: known_used parts_trans)
```

```
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)"
by (auto simp: Guard_def dest: known_max_used parts_trans)
```

```
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)"
apply (rule_tac H="knows_max (Friend C) evs" in Guard_mono)
by (auto simp: knows_max_def)
```

37.2.3 regular protocols

```
definition regular :: "event list set => bool" where
"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)"
by (auto simp: regular_def)
```

```
lemma priK_analz_iff_bad [simp]: "[| evs:p; regular p |] ==>
(Key (priK A):analz (spies evs)) = (A:bad)"
by auto
```

```
lemma Guard_Nonce_analz: "[| Guard n Ks (spies evs); evs:p;
priK_set Ks; good Ks; regular p |] ==> Nonce n ~:analz (spies evs)"
apply (clarify, simp only: knows_decomp)
apply (drule Guard_invKey_keyset, simp+, safe)
apply (drule in_good, simp)
apply (drule in_priK_set, simp+, clarify)
apply (frule_tac A=A in priK_analz_iff_bad)
```

```

by (simp add: knows_decomp)+
end

```

38 Lists of Messages and Lists of Agents

```

theory List_Msg imports Extensions begin

```

38.1 Implementation of Lists by Messages

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

```

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

```

38.1.2 induction principle

```

lemma lmsg_induct: "[| !!x. not_MPair x ==> P x; !!x l. P l ==> P (cons x
l) |]
==> P l"
by (induct l) auto

```

38.1.3 head

```

primrec head :: "msg => msg" where
"head (cons x l) = x"

```

38.1.4 tail

```

primrec tail :: "msg => msg" where
"tail (cons x l) = l"

```

38.1.5 length

```

fun len :: "msg => nat" where
"len (cons x l) = Suc (len l)" |
"len other = 0"

```

```

lemma len_not_empty: "n < len l ==> EX x l'. l = cons x l'"
by (cases l) auto

```

38.1.6 membership

```

fun isin :: "msg * msg => bool" where
"isin (x, cons y l) = (x=y | isin (x,l))" |
"isin (x, other) = False"

```

38.1.7 delete an element

```

fun del :: "msg * msg => msg" where
"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"
by (induct l) auto
```

```
lemma isin_del [rule_format]: "isin (y, del (x,l)) --> isin (y,l)"
by (induct l) auto
```

38.1.8 concatenation

```
fun app :: "msg * msg => msg" where
"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'))"
by (induct l) auto
```

38.1.9 replacement

```
fun repl :: "msg * nat * msg => msg" where
"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"
```

38.1.10 ith element

```
fun ith :: "msg * nat => msg" where
"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"
by (cases l) auto
```

38.1.11 insertion

```
fun ins :: "msg * nat * msg => msg" where
"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"
by (cases l) auto
```

38.1.12 truncation

```
fun trunc :: "msg * nat => msg" where
"trunc (l,0) = l" |
"trunc (cons x l, Suc i) = trunc (l,i)"
```

```
lemma trunc_zero [simp]: "trunc (l,0) = l"
by (cases l) auto
```

38.2 Agent Lists

38.2.1 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"

38.2.2 basic facts about agent lists

lemma del_in_agl [intro]: "I:agl ==> del (a,I):agl"
by (erule agl.induct, auto)

lemma app_in_agl [intro]: "[| I:agl; J:agl |] ==> app (I,J):agl"
by (erule agl.induct, auto)

lemma no_Key_in_agl: "I:agl ==> Key K ~:parts {I}"
by (erule agl.induct, auto)

lemma no_Nonce_in_agl: "I:agl ==> Nonce n ~:parts {I}"
by (erule agl.induct, auto)

lemma no_Key_in_appdel: "[| I:agl; J:agl |] ==>
Key K ~:parts {app (J, del (Agent B, I))}"
by (rule no_Key_in_agl, auto)

lemma no_Nonce_in_appdel: "[| I:agl; J:agl |] ==>
Nonce n ~:parts {app (J, del (Agent B, I))}"
by (rule no_Nonce_in_agl, auto)

lemma no_Crypt_in_agl: "I:agl ==> Crypt K X ~:parts {I}"
by (erule agl.induct, auto)

lemma no_Crypt_in_appdel: "[| I:agl; J:agl |] ==>
Crypt K X ~:parts {app (J, del (Agent B,I))}"
by (rule no_Crypt_in_agl, auto)

end

```

39 Protocol P1

```
theory P1 imports "../Public" Guard_Public List_Msg begin
```

39.1 Protocol Definition

39.1.1 offer chaining: B chains his offer for A with the head offer of L for sending it to C

```

definition chain :: "agent => nat => agent => msg => agent => msg" where
"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')"
by (auto simp: chain_def Let_def)
```

```
lemma Nonce_in_chain [iff]: "Nonce ofr:parts {chain B ofr A L C}"
by (auto simp: chain_def sign_def)
```

39.1.2 agent whose key is used to sign an offer

```
fun shop :: "msg => msg" where
"shop {|B,X,Crypt K H|} = Agent (agt K)"
```

```
lemma shop_chain [simp]: "shop (chain B ofr A L C) = Agent B"
by (simp add: chain_def sign_def)
```

39.1.3 nonce used in an offer

```
fun nonce :: "msg => msg" where
"nonce {|B,{|Crypt K ofr,m2|},CryptH|} = ofr"
```

```
lemma nonce_chain [simp]: "nonce (chain B ofr A L C) = Nonce ofr"
by (simp add: chain_def sign_def)
```

39.1.4 next shop

```
fun next_shop :: "msg => agent" where
"next_shop {|B,{|m1,Hash{|headL,Agent C|}|},CryptH|} = C"
```

```
lemma next_shop_chain [iff]: "next_shop (chain B ofr A L C) = C"
by (simp add: chain_def sign_def)
```

39.1.5 anchor of the offer list

```
definition anchor :: "agent => nat => agent => msg" where
"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')"
by (auto simp: anchor_def)
```

```
lemma Nonce_in_anchor [iff]: "Nonce n:parts {anchor A n B}"
by (auto simp: anchor_def)
```

```
lemma shop_anchor [simp]: "shop (anchor A n B) = Agent A"
by (simp add: anchor_def)
```

```
lemma nonce_anchor [simp]: "nonce (anchor A n B) = Nonce n"
by (simp add: anchor_def)
```

```
lemma next_shop_anchor [iff]: "next_shop (anchor A n B) = B"
by (simp add: anchor_def)
```

39.1.6 request event

definition *reqm* :: "agent => nat => nat => msg => agent => msg" where
 "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')"
 by (auto simp: reqm_def)

lemma *Nonce_in_reqm* [iff]: "Nonce n:parts {reqm A r n I B}"
 by (auto simp: reqm_def)

definition *req* :: "agent => nat => nat => msg => agent => event" where
 "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')"
 by (auto simp: req_def)

39.1.7 propose event

definition *prom* :: "agent => nat => agent => nat => msg => msg =>
 msg => agent => msg" where
 "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'"
 by (auto simp: prom_def)

lemma *Nonce_in_prom* [iff]: "Nonce ofr:parts {prom B ofr A r I L J C}"
 by (auto simp: prom_def)

definition *pro* :: "agent => nat => agent => nat => msg => msg =>
 msg => agent => event" where
 "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'"
 by (auto simp: pro_def dest: prom_inj)

39.1.8 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"

```

39.1.9 Composition of Traces

```

lemma "evs':p1 ==>
  evs:p1 & (ALL n. Nonce n:used evs' --> Nonce n ~:used evs) -->
  evs'@evs : p1"
apply (erule p1.induct, safe)
apply (simp_all add: used_ConsI)
apply (erule p1.Fake, erule synth_sub, rule analz_mono, rule knows_sub_app)
apply (erule p1.Request, safe, simp_all add: req_def, force)
apply (erule_tac A'=A' in p1.Propose, simp_all)
apply (drule_tac x=ofr in spec, simp add: pro_def, blast)
apply (erule_tac A'=A' in p1.Propose, auto simp: pro_def)
done

```

39.1.10 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"

```

39.1.11 basic properties of valid

```

lemma valid_not_empty: "L:valid A n B ==> EX M L'. L = cons M L'"
by (erule valid.cases, auto)

```

```

lemma valid_pos_len: "L:valid A n B ==> 0 < len L"
by (erule valid.induct, auto)

```

39.1.12 offers of an offer list

```

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

```

39.1.13 the originator can get the offers

```

lemma "L:valid A n B ==> offer_nonces L <= analz (insert L (initState A))"
by (erule valid.induct, auto simp: anchor_def chain_def sign_def
offer_nonces_def initState.simps)

```

39.1.14 list of offers

```

fun offers :: "msg => msg" where
"offers (cons M L) = cons {|shop M, nonce M|} (offers L)" |
"offers other = nil"

```

39.1.15 list of agents whose keys are used to sign a list of offers

```

fun shops :: "msg => msg" where
  "shops (cons M L) = cons (shop M) (shops L)" |
  "shops other = other"

```

```

lemma shops_in_agl: "L:valid A n B ==> shops L:agl"
by (erule valid.induct, auto simp: anchor_def chain_def sign_def)

```

39.1.16 builds a trace from an itinerary

```

fun offer_list :: "agent * nat * agent * msg * nat => msg" where
  "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"
by (erule agl.induct, auto)

```

```

fun trace :: "agent * nat * agent * nat * msg * msg * msg
=> event list" where
  "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))"

```

```

definition trace' :: "agent => nat => nat => msg => agent => nat => event list"
where
  "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]

```

39.1.17 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)"
oops

```

39.2 properties of protocol P1

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

39.2.1 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)"
apply (induct i)
```

```
apply clarify
apply (frule len_not_empty, clarsimp)
apply (frule len_not_empty, clarsimp)
apply (ind_cases "{|x,xa,l'a|}:valid A n B" for x xa l'a)
apply (ind_cases "{|x,M,l'a|}:valid A n B" for x l'a)
apply (simp add: chain_def)
```

```
apply clarify
apply (frule len_not_empty, clarsimp)
apply (ind_cases "{|x,repl(l',Suc na,M)|}:valid A n B" for x l' na)
apply (frule len_not_empty, clarsimp)
apply (ind_cases "{|x,l'|}:valid A n B" for x l')
by (drule_tac x=l' in spec, simp, blast)
```

39.2.2 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"
by (erule valid.induct, auto simp: chain_def sign_def anchor_def)
```

```
lemma insertion_resilience: "ALL L. L:valid A n B --> Suc i < len L
--> ins (L,Suc i,M) ~:valid A n B"
apply (induct i)
```

```
apply clarify
apply (frule len_not_empty, clarsimp)
apply (ind_cases "{|x,l'|}:valid A n B" for x l', simp)
apply (ind_cases "{|x,M,l'|}:valid A n B" for x l', clarsimp)
apply (ind_cases "{|head l',l'|}:valid A n B" for l', simp, simp)
```

```
apply clarify
apply (frule len_not_empty, clarsimp)
apply (ind_cases "{|x,l'|}:valid A n B" for x l')
apply (frule len_not_empty, clarsimp)
apply (ind_cases "{|x,ins(l',Suc na,M)|}:valid A n B" for x l' na)
apply (frule len_not_empty, clarsimp)
by (drule_tac x=l' in spec, clarsimp)
```

39.2.3 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))"
apply (induct i)
```

```
apply clarify
apply (frule len_not_empty, clarsimp)
apply (ind_cases "{|x,l'|}:valid A n B" for x l')
```

```

apply (frule len_not_empty, clarsimp)
apply (ind_cases "{|M,l'|}:valid A n B" for l')
apply (frule len_not_empty, clarsimp, simp)

```

```

apply clarify
apply (frule len_not_empty, clarsimp)
apply (ind_cases "{|x,l'|}:valid A n B" for x l')
apply (frule len_not_empty, clarsimp)
by (drule_tac x=l' in spec, clarsimp)

```

39.2.4 declarations for tactics

```

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

```

39.2.5 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)"
by blast

```

39.2.6 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"
by (auto simp: reqm_def prom_def)

```

```

lemma prom_neq_reqm [iff]:
"prom B' ofr A' r' I' (cons M L) J C ~= reqm A r n I B"
by (auto simp: reqm_def prom_def)

```

```

lemma req_neq_pro [iff]: "req A r n I B ~= pro B' ofr A' r' I' (cons M L)
J C"
by (auto simp: req_def pro_def)

```

```

lemma pro_neq_req [iff]: "pro B' ofr A' r' I' (cons M L) J C ~= req A r n
I B"
by (auto simp: req_def pro_def)

```

```

lemma p1_has_no_Gets: "evs:p1 ==> ALL A X. Gets A X ~:set evs"
by (erule p1.induct, auto simp: req_def pro_def)

```

```

lemma p1_is_Gets_correct [iff]: "Gets_correct p1"
by (auto simp: Gets_correct_def dest: p1_has_no_Gets)

```

```

lemma p1_is_one_step [iff]: "one_step p1"
by (unfold one_step_def, clarify, ind_cases "ev#evs:p1" for ev evs, auto)

```

```

lemma p1_has_only_Says' [rule_format]: "evs:p1 ==>
ev:set evs --> (EX A B X. ev=Says A B X)"
by (erule p1.induct, auto simp: req_def pro_def)

```

```

lemma p1_has_only_Says [iff]: "has_only_Says p1"
by (auto simp: has_only_Says_def dest: p1_has_only_Says')

```

```

lemma p1_is_regular [iff]: "regular p1"
apply (simp only: regular_def, clarify)
apply (erule_tac p1.induct)
apply (simp_all add: initState.simps knows.simps pro_def prom_def
    req_def reqm_def anchor_def chain_def sign_def)
by (auto dest: no_Key_in_agl no_Key_in_appdel parts_trans)

```

39.2.7 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)"
apply (erule p1.induct)
apply (simp_all add: initState.simps knows.simps pro_def prom_def
    req_def reqm_def anchor_def chain_def sign_def)
apply (blast dest: no_Key_in_agl)
apply (auto del: parts_invKey disjE dest: parts_trans
    simp add: no_Key_in_appdel)
done

```

```

lemma priK_analz_Friend_imp_bad [rule_format,dest]:
  "[| evs:p1; Friend B ~= A |]
  ==> (Key (priK A):analz (knows (Friend B) evs)) --> (A:bad)"
by auto

```

```

lemma priK_notin_knows_max_Friend: "[| evs:p1; A ~:bad; A ~= Friend C |]
  ==> Key (priK A) ~:analz (knows_max (Friend C) evs)"
apply (rule not_parts_not_analz, simp add: knows_max_def, safe)
apply (drule_tac H="spies' evs" in parts_sub)
apply (rule_tac p=p1 in knows_max'_sub_spies', simp+)
apply (drule_tac H="spies evs" in parts_sub)
by (auto dest: knows'_sub_knows [THEN subsetD] priK_notin_initState_Friend)

```

39.2.8 general guardedness properties

```

lemma agl_guard [intro]: "I:agl ==> I:guard n Ks"
by (erule agl.induct, auto)

```

```

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"
by (auto dest: Says_to_knows_max')

```

```

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"
by (auto dest: Says_from_knows_max')

```

```

lemma Says_Nonce_not_used_guard: "[| Says A' B {|A'',r,I,L|}:set evs;
  Nonce n ~:used evs |] ==> L:guard n Ks"
by (drule not_used_not_parts, auto)

```

39.2.9 guardedness of messages

```

lemma chain_guard [iff]: "chain B ofr A L C:guard n {priK A}"
by (case_tac "ofr=n", auto simp: chain_def sign_def)

```

```
lemma chain_guard_Nonce_neq [intro]: "n ~= ofr
==> chain B ofr A' L C:guard n {priK A}"
by (auto simp: chain_def sign_def)
```

```
lemma anchor_guard [iff]: "anchor A n' B:guard n {priK A}"
by (case_tac "n'=n", auto simp: anchor_def)
```

```
lemma anchor_guard_Nonce_neq [intro]: "n ~= n'
==> anchor A' n' B:guard n {priK A}"
by (auto simp: anchor_def)
```

```
lemma reqm_guard [intro]: "I:agl ==> reqm A r n' I B:guard n {priK A}"
by (case_tac "n'=n", auto simp: reqm_def)
```

```
lemma reqm_guard_Nonce_neq [intro]: "[| n ~= n'; I:agl |]
==> reqm A' r n' I B:guard n {priK A}"
by (auto simp: reqm_def)
```

```
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}"
by (auto simp: prom_def)
```

```
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}"
by (auto simp: prom_def)
```

39.2.10 Nonce uniqueness

```
lemma uniq_Nonce_in_chain [dest]: "Nonce k:parts {chain B ofr A L C} ==>
k=ofr"
by (auto simp: chain_def sign_def)
```

```
lemma uniq_Nonce_in_anchor [dest]: "Nonce k:parts {anchor A n B} ==> k=n"
by (auto simp: anchor_def chain_def sign_def)
```

```
lemma uniq_Nonce_in_reqm [dest]: "[| Nonce k:parts {reqm A r n I B};
I:agl |] ==> k=n"
by (auto simp: reqm_def dest: no_Nonce_in_agl)
```

```
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"
by (auto simp: prom_def dest: no_Nonce_in_agl no_Nonce_in_appdel)
```

39.2.11 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)"
apply (erule p1.induct, simp)
apply (simp add: req_def knows.simps, safe)
apply (erule in_synth_Guard, erule Guard_analz, simp)
by (auto simp: req_def pro_def dest: Says_imp_knows_Spy)
```

```
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)"
apply (rule Guard_knows_max')
apply (rule_tac H="spies evs" in Guard_mono)
apply (rule req_imp_Guard, simp+)
apply (rule_tac B="spies' evs" in subset_trans)
apply (rule_tac p=p1 in knows_max'_sub_spies', simp+)
by (rule knows'_sub_knows)

```

39.2.12 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)"
apply (erule p1.induct)

apply simp

apply (simp add: pro_def, safe)

apply (erule in_synth_Guard, drule Guard_analz, simp, simp)

apply simp

apply (simp, simp add: req_def pro_def, blast)

apply (simp add: pro_def)
apply (blast dest: prom_inj Says_Nonce_not_used_guard Nonce_not_used_Guard)

apply simp
apply safe
apply (simp add: pro_def)
apply (blast dest: prom_inj Says_Nonce_not_used_guard)

apply (simp add: pro_def)
apply (blast dest: Says_imp_knows_Spy)

apply (simp add: pro_def)
apply (blast dest: prom_inj Says_Nonce_not_used_guard Nonce_not_used_Guard)

apply simp
apply safe

apply (simp add: pro_def)
apply (blast dest: prom_inj Says_Nonce_not_used_guard)

apply (simp add: pro_def)
by (blast dest: Says_imp_knows_Spy)

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)"
apply (rule Guard_knows_max')
apply (rule_tac H="spies evs" in Guard_mono)
apply (rule pro_imp_Guard, simp+)
apply (rule_tac B="spies' evs" in subset_trans)

```

```

apply (rule_tac p=p1 in knows_max'_sub_spies', simp+)
by (rule knows'_sub_knows)

```

39.2.13 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)"
by (frule req_imp_Guard, simp+, erule Guard_Nonce_analz, simp+)

```

```

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)"
apply (clarify, frule_tac C=C in req_imp_Guard_Friend, simp+)
apply (simp add: knows_max_def, drule Guard_invKey_keyset, simp+)
by (drule priK_notin_knows_max_Friend, auto simp: knows_max_def)

```

```

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)"
by (frule pro_imp_Guard, simp+, erule Guard_Nonce_analz, simp+)

```

```

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)"
apply (clarify, frule_tac A=A in pro_imp_Guard_Friend, simp+)
apply (simp add: knows_max_def, drule Guard_invKey_keyset, simp+)
by (drule priK_notin_knows_max_Friend, auto simp: knows_max_def)

```

39.2.14 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'"
by (auto simp: reqm_def anchor_def chain_def sign_def dest: no_Crypt_in_agl)

```

```

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}"
apply (simp add: prom_def anchor_def chain_def sign_def)
by (blast dest: no_Crypt_in_agl no_Crypt_in_appdel)

```

```

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})"
apply (erule p1.induct)

```

```

apply simp

```

```

apply clarsimp
apply (drule_tac P="%G. Crypt (priK A) X:G" in parts_insert_substD, simp)
apply (erule disjE)
apply (drule_tac K="priK A" in Crypt_synth, simp+, blast, blast)

```

```

apply (simp add: req_def, clarify)
apply (drule_tac P="%G. Crypt (priK A) X:G" in parts_insert_substD, simp)
apply (erule disjE)
apply (frule Crypt_reqm, simp, clarify)

```

```

apply (rule_tac x=B in exI, rule_tac x="reqm A r n I B" in exI, simp, blast)

apply (simp add: pro_def, clarify)
apply (drule_tac P="%G. Crypt (priK A) X:G" in parts_insert_substD, simp)
apply (rotate_tac -1, erule disjE)
apply (frule Crypt_prom, simp, simp)
apply (rotate_tac -1, erule disjE)
apply (rule_tac x=C in exI)
apply (rule_tac x="prom B ofr Aa r I (cons M L) J C" in exI, blast)
apply (subgoal_tac "cons M L:parts (spies evsp)")
apply (drule_tac G="{cons M L}" and H="spies evsp" in parts_trans, blast,
blast)
apply (drule Says_imp_spies, rotate_tac -1, drule parts.Inj)
apply (drule parts.Snd, drule parts.Snd, drule parts.Snd)
by auto

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})"
apply (erule p1.induct)

apply simp

apply clarsimp
apply (drule_tac P="%G. Crypt (priK A) (Hash X):G" in parts_insert_substD)
apply simp
apply (erule disjE)
apply (drule_tac K="priK A" in Crypt_synth, simp+, blast, blast)

apply (simp add: req_def, clarify)
apply (drule_tac P="%G. Crypt (priK A) (Hash X):G" in parts_insert_substD)
apply simp
apply (erule disjE)
apply (frule Crypt_reqm, simp+)
apply (rule_tac x=B in exI, rule_tac x="reqm Aa r n I B" in exI)
apply (simp add: reqm_def sign_def anchor_def no_Crypt_in_agl)
apply (simp add: chain_def sign_def, blast)

apply (simp add: pro_def, clarify)
apply (drule_tac P="%G. Crypt (priK A) (Hash X):G" in parts_insert_substD)
apply simp
apply (rotate_tac -1, erule disjE)
apply (simp add: prom_def sign_def no_Crypt_in_agl no_Crypt_in_appdel)
apply (simp add: chain_def sign_def)
apply (rotate_tac -1, erule disjE)
apply (rule_tac x=C in exI)
apply (rule_tac x="prom B ofr Aa r I (cons M L) J C" in exI)
apply (simp add: prom_def chain_def sign_def)
apply (erule impE)
apply (blast dest: get_ML parts_sub)
apply (blast del: MPair_parts)+
done

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})"
apply (clarify, simp add: sign_def, frule parts.Snd)
apply (blast dest: Crypt_Hash_imp_sign [unfolded sign_def])
done

end

```

40 Protocol P2

theory P2 imports Guard_Public List_Msg begin

40.1 Protocol Definition

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

40.1.1 offer chaining: B chains his offer for A with the head offer of L for sending it to C

```

definition chain :: "agent => nat => agent => msg => agent => msg" where
"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')"
by (auto simp: chain_def Let_def)

```

```

lemma Nonce_in_chain [iff]: "Nonce ofr:parts {chain B ofr A L C}"
by (auto simp: chain_def sign_def)

```

40.1.2 agent whose key is used to sign an offer

```

fun shop :: "msg => msg" where
"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"
by (simp add: chain_def sign_def)

```

40.1.3 nonce used in an offer

```

fun nonce :: "msg => msg" where
"nonce {|Crypt K {|B,ofr,CryptH|},m2|} = ofr"

```

```

lemma nonce_chain [simp]: "nonce (chain B ofr A L C) = Nonce ofr"
by (simp add: chain_def sign_def)

```

40.1.4 next shop

```

fun next_shop :: "msg => agent" where
"next_shop {|m1,Hash {|headL,Agent C|}|} = C"

```

```
lemma "next_shop (chain B ofr A L C) = C"
by (simp add: chain_def sign_def)
```

40.1.5 anchor of the offer list

```
definition anchor :: "agent => nat => agent => msg" where
"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')"
by (auto simp: anchor_def)
```

```
lemma Nonce_in_anchor [iff]: "Nonce n:parts {anchor A n B}"
by (auto simp: anchor_def)
```

```
lemma shop_anchor [simp]: "shop (anchor A n B) = Agent A"
by (simp add: anchor_def)
```

40.1.6 request event

```
definition reqm :: "agent => nat => nat => msg => agent => msg" where
"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')"
by (auto simp: reqm_def)
```

```
lemma Nonce_in_reqm [iff]: "Nonce n:parts {reqm A r n I B}"
by (auto simp: reqm_def)
```

```
definition req :: "agent => nat => nat => msg => agent => event" where
"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')"
by (auto simp: req_def)
```

40.1.7 propose event

```
definition prom :: "agent => nat => agent => nat => msg => msg =>
msg => agent => msg" where
"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'"
by (auto simp: prom_def)
```

```
lemma Nonce_in_prom [iff]: "Nonce ofr:parts {prom B ofr A r I L J C}"
by (auto simp: prom_def)
```

```
definition pro :: "agent => nat => agent => nat => msg => msg =>
msg => agent => event" where
```

```

"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'"
by (auto simp: pro_def dest: prom_inj)

```

40.1.8 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"

```

40.1.9 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"

```

40.1.10 basic properties of valid

```

lemma valid_not_empty: "L:valid A n B ==> EX M L'. L = cons M L'"
by (erule valid.cases, auto)

```

```

lemma valid_pos_len: "L:valid A n B ==> 0 < len L"
by (erule valid.induct, auto)

```

40.1.11 list of offers

```

fun offers :: "msg => msg"
where
  "offers (cons M L) = cons {|shop M, nonce M|} (offers L)"
  / "offers other = nil"

```

40.2 Properties of Protocol P2

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

40.3 strong forward integrity: except the last one, no offer can be modified 409

40.3 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)"
apply (induct i)
```

```
apply clarify
apply (frule len_not_empty, clarsimp)
apply (frule len_not_empty, clarsimp)
apply (ind_cases "{|x,xa,l'a|}:valid A n B" for x xa l'a)
apply (ind_cases "{|x,M,l'a|}:valid A n B" for x l'a)
apply (simp add: chain_def)
```

```
apply clarify
apply (frule len_not_empty, clarsimp)
apply (ind_cases "{|x,repl(l',Suc na,M)|}:valid A n B" for x l' na)
apply (frule len_not_empty, clarsimp)
apply (ind_cases "{|x,l'|}:valid A n B" for x l')
by (drule_tac x=l' in spec, simp, blast)
```

40.4 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"
by (erule valid.induct, auto simp: chain_def sign_def anchor_def)
```

```
lemma insertion_resilience: "ALL L. L:valid A n B --> Suc i < len L
--> ins (L,Suc i,M) ~:valid A n B"
apply (induct i)
```

```
apply clarify
apply (frule len_not_empty, clarsimp)
apply (ind_cases "{|x,l'|}:valid A n B" for x l', simp)
apply (ind_cases "{|x,M,l'|}:valid A n B" for x l', clarsimp)
apply (ind_cases "{|head l',l'|}:valid A n B" for l', simp, simp)
```

```
apply clarify
apply (frule len_not_empty, clarsimp)
apply (ind_cases "{|x,l'|}:valid A n B" for x l')
apply (frule len_not_empty, clarsimp)
apply (ind_cases "{|x,ins(l',Suc na,M)|}:valid A n B" for x l' na)
apply (frule len_not_empty, clarsimp)
by (drule_tac x=l' in spec, clarsimp)
```

40.5 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))"
apply (induct i)
```

```
apply clarify
```

```

apply (frule len_not_empty, clarsimp)
apply (ind_cases "{|x,l'|}:valid A n B" for x l')
apply (frule len_not_empty, clarsimp)
apply (ind_cases "{|M,l'|}:valid A n B" for l')
apply (frule len_not_empty, clarsimp, simp)

apply clarify
apply (frule len_not_empty, clarsimp)
apply (ind_cases "{|x,l'|}:valid A n B" for x l')
apply (frule len_not_empty, clarsimp)
by (drule_tac x=l' in spec, clarsimp)

```

40.6 declarations for tactics

```

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

```

40.7 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)"
by blast

```

40.8 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"
by (auto simp: reqm_def prom_def)

lemma prom_neq_reqm [iff]:
"prom B' ofr A' r' I' (cons M L) J C ~= reqm A r n I B"
by (auto simp: reqm_def prom_def)

lemma req_neq_pro [iff]: "req A r n I B ~= pro B' ofr A' r' I' (cons M L)
J C"
by (auto simp: req_def pro_def)

lemma pro_neq_req [iff]: "pro B' ofr A' r' I' (cons M L) J C ~= req A r n
I B"
by (auto simp: req_def pro_def)

lemma p2_has_no_Gets: "evs:p2 ==> ALL A X. Gets A X ~:set evs"
by (erule p2.induct, auto simp: req_def pro_def)

lemma p2_is_Gets_correct [iff]: "Gets_correct p2"
by (auto simp: Gets_correct_def dest: p2_has_no_Gets)

lemma p2_is_one_step [iff]: "one_step p2"
by (unfold one_step_def, clarify, ind_cases "ev#evs:p2" for ev evs, auto)

lemma p2_has_only_Says' [rule_format]: "evs:p2 ==>
ev:set evs --> (EX A B X. ev=Says A B X)"
by (erule p2.induct, auto simp: req_def pro_def)

```

```
lemma p2_has_only_Says [iff]: "has_only_Says p2"
by (auto simp: has_only_Says_def dest: p2_has_only_Says')
```

```
lemma p2_is_regular [iff]: "regular p2"
apply (simp only: regular_def, clarify)
apply (erule_tac p2.induct)
apply (simp_all add: initState.simps knows.simps pro_def prom_def
req_def reqm_def anchor_def chain_def sign_def)
by (auto dest: no_Key_in_agl no_Key_in_appdel parts_trans)
```

40.9 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)"
apply (erule p2.induct)
apply (simp_all add: initState.simps knows.simps pro_def prom_def
  req_def reqm_def anchor_def chain_def sign_def)
apply (blast dest: no_Key_in_agl)
apply (auto del: parts_invKey disjE dest: parts_trans
  simp add: no_Key_in_appdel)
done
```

```
lemma priK_analz_Friend_imp_bad [rule_format,dest]:
  "[| evs:p2; Friend B ~= A |]
  ==> (Key (priK A):analz (knows (Friend B) evs)) --> (A:bad)"
by auto
```

```
lemma priK_notin_knows_max_Friend:
  "[| evs:p2; A ~:bad; A ~= Friend C |]
  ==> Key (priK A) ~:analz (knows_max (Friend C) evs)"
apply (rule not_parts_not_analz, simp add: knows_max_def, safe)
apply (drule_tac H="spies' evs" in parts_sub)
apply (rule_tac p=p2 in knows_max'_sub_spies', simp+)
apply (drule_tac H="spies evs" in parts_sub)
by (auto dest: knows'_sub_knows [THEN subsetD] priK_notin_initState_Friend)
```

40.10 general guardedness properties

```
lemma agl_guard [intro]: "I:agl ==> I:guard n Ks"
by (erule agl.induct, auto)
```

```
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"
by (auto dest: Says_to_knows_max')
```

```
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"
by (auto dest: Says_from_knows_max')
```

```
lemma Says_Nonce_not_used_guard: "[| Says A' B {|A'',r,I,L|}:set evs;
Nonce n ~:used evs |] ==> L:guard n Ks"
by (drule not_used_not_parts, auto)
```

40.11 guardedness of messages

lemma chain_guard [iff]: "chain B ofr A L C:guard n {priK A}"
by (case_tac "ofr=n", auto simp: chain_def sign_def)

lemma chain_guard_Nonce_neq [intro]: "n ~= ofr
==> chain B ofr A' L C:guard n {priK A}"
by (auto simp: chain_def sign_def)

lemma anchor_guard [iff]: "anchor A n' B:guard n {priK A}"
by (case_tac "n'=n", auto simp: anchor_def)

lemma anchor_guard_Nonce_neq [intro]: "n ~= n'
==> anchor A' n' B:guard n {priK A}"
by (auto simp: anchor_def)

lemma reqm_guard [intro]: "I:agl ==> reqm A r n' I B:guard n {priK A}"
by (case_tac "n'=n", auto simp: reqm_def)

lemma reqm_guard_Nonce_neq [intro]: "[| n ~= n'; I:agl |]
==> reqm A' r n' I B:guard n {priK A}"
by (auto simp: reqm_def)

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}"
by (auto simp: prom_def)

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}"
by (auto simp: prom_def)

40.12 Nonce uniqueness

lemma uniq_Nonce_in_chain [dest]: "Nonce k:parts {chain B ofr A L C} ==>
k=ofr"
by (auto simp: chain_def sign_def)

lemma uniq_Nonce_in_anchor [dest]: "Nonce k:parts {anchor A n B} ==> k=n"
by (auto simp: anchor_def chain_def sign_def)

lemma uniq_Nonce_in_reqm [dest]: "[| Nonce k:parts {reqm A r n I B};
I:agl |] ==> k=n"
by (auto simp: reqm_def dest: no_Nonce_in_agl)

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"
by (auto simp: prom_def dest: no_Nonce_in_agl no_Nonce_in_appdel)

40.13 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)"
apply (erule p2.induct, simp)
apply (simp add: req_def knows.simps, safe)

```

apply (erule in_synth_Guard, erule Guard_analz, simp)
by (auto simp: req_def pro_def dest: Says_imp_knows_Spy)

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)"
apply (rule Guard_knows_max')
apply (rule_tac H="spies evs" in Guard_mono)
apply (rule req_imp_Guard, simp+)
apply (rule_tac B="spies' evs" in subset_trans)
apply (rule_tac p=p2 in knows_max'_sub_spies', simp+)
by (rule knows'_sub_knows)

```

40.14 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)"
apply (erule p2.induct)

apply simp

apply (simp add: pro_def, safe)

apply (erule in_synth_Guard, drule Guard_analz, simp, simp)

apply simp

apply (simp, simp add: req_def pro_def, blast)

apply (simp add: pro_def)
apply (blast dest: prom_inj Says_Nonce_not_used_guard Nonce_not_used_Guard)

apply simp
apply safe
apply (simp add: pro_def)
apply (blast dest: prom_inj Says_Nonce_not_used_guard)

apply (simp add: pro_def)
apply (blast dest: Says_imp_knows_Spy)

apply (simp add: pro_def)
apply (blast dest: prom_inj Says_Nonce_not_used_guard Nonce_not_used_Guard)

apply simp
apply safe

apply (simp add: pro_def)
apply (blast dest: prom_inj Says_Nonce_not_used_guard)

apply (simp add: pro_def)
by (blast dest: Says_imp_knows_Spy)

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)"

```

```

apply (rule Guard_knows_max')
apply (rule_tac H="spies evs" in Guard_mono)
apply (rule pro_imp_Guard, simp+)
apply (rule_tac B="spies' evs" in subset_trans)
apply (rule_tac p=p2 in knows_max'_sub_spies', simp+)
by (rule knows'_sub_knows)

```

40.15 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)"
by (frule req_imp_Guard, simp+, erule Guard_Nonce_analz, simp+)

```

```

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)"
apply (clarify, frule_tac C=C in req_imp_Guard_Friend, simp+)
apply (simp add: knows_max_def, drule Guard_invKey_keyset, simp+)
by (drule priK_notin_knows_max_Friend, auto simp: knows_max_def)

```

```

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)"
by (frule pro_imp_Guard, simp+, erule Guard_Nonce_analz, simp+)

```

```

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)"
apply (clarify, frule_tac A=A in pro_imp_Guard_Friend, simp+)
apply (simp add: knows_max_def, drule Guard_invKey_keyset, simp+)
by (drule priK_notin_knows_max_Friend, auto simp: knows_max_def)

```

40.16 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)"
by (auto simp:sign_def dest: Nonce_pro_notin_spies)

```

```

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)"
by (auto simp:sign_def dest:Nonce_pro_notin_knows_max_Friend )

```

40.17 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'"
by (auto simp: reqm_def anchor_def chain_def sign_def dest: no_Crypt_in_agl)

```

```

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}"

```

```

apply (simp add: prom_def anchor_def chain_def sign_def)
by (blast dest: no_Crypt_in_agl no_Crypt_in_appdel)

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})"
apply (erule p2.induct)

apply simp

apply clarsimp
apply (drule_tac P="%G. Crypt (priK A) X:G" in parts_insert_substD, simp)
apply (erule disjE)
apply (drule_tac K="priK A" in Crypt_synth, simp+, blast, blast)

apply (simp add: req_def, clarify)
apply (drule_tac P="%G. Crypt (priK A) X:G" in parts_insert_substD, simp)
apply (erule disjE)
apply (frule Crypt_reqm, simp, clarify)
apply (rule_tac x=B in exI, rule_tac x="reqm A r n I B" in exI, simp, blast)

apply (simp add: pro_def, clarify)
apply (drule_tac P="%G. Crypt (priK A) X:G" in parts_insert_substD, simp)
apply (rotate_tac -1, erule disjE)
apply (frule Crypt_prom, simp, simp)
apply (rotate_tac -1, erule disjE)
apply (rule_tac x=C in exI)
apply (rule_tac x="prom B ofr Aa r I (cons M L) J C" in exI, blast)
apply (subgoal_tac "cons M L:parts (spies evsp)")
apply (drule_tac G="{cons M L}" and H="spies evsp" in parts_trans, blast,
blast)
apply (drule Says_imp_spies, rotate_tac -1, drule parts.Inj)
apply (drule parts.Snd, drule parts.Snd, drule parts.Snd)
by auto

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})"
apply (erule p2.induct)

apply simp

apply clarsimp
apply (drule_tac P="%G. Crypt (priK A) (Hash X):G" in parts_insert_substD)
apply simp
apply (erule disjE)
apply (drule_tac K="priK A" in Crypt_synth, simp+, blast, blast)

apply (simp add: req_def, clarify)
apply (drule_tac P="%G. Crypt (priK A) (Hash X):G" in parts_insert_substD)
apply simp
apply (erule disjE)
apply (frule Crypt_reqm, simp+)
apply (rule_tac x=B in exI, rule_tac x="reqm Aa r n I B" in exI)

```

```

apply (simp add: reqm_def sign_def anchor_def no_Crypt_in_agl)
apply (simp add: chain_def sign_def, blast)

apply (simp add: pro_def, clarify)
apply (drule_tac P="%G. Crypt (priK A) (Hash X):G" in parts_insert_substD)
apply simp
apply (rotate_tac -1, erule disjE)
apply (simp add: prom_def sign_def no_Crypt_in_agl no_Crypt_in_appdel)
apply (simp add: chain_def sign_def)
apply (rotate_tac -1, erule disjE)
apply (rule_tac x=C in exI)
apply (rule_tac x="prom B ofr Aa r I (cons M L) J C" in exI)
apply (simp add: prom_def chain_def sign_def)
apply (erule impE)
apply (blast dest: get_ML parts_sub)
apply (blast del: MPair_parts)+
done

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})"
apply (clarify, simp add: sign_def, frule parts.Snd)
apply (blast dest: Crypt_Hash_imp_sign [unfolded sign_def])
done

end

```

41 Needham-Schroeder-Lowe Public-Key Protocol

```

theory Guard_NS_Public imports Guard_Public begin

```

41.1 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))"

```

41.2 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"

```

41.3 declarations for tactics

```

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

```

41.4 general properties of nsp

```

lemma nsp_has_no_Gets: "evs:nsp ==> ALL A X. Gets A X ~:set evs"
by (erule nsp.induct, auto)

lemma nsp_is_Gets_correct [iff]: "Gets_correct nsp"
by (auto simp: Gets_correct_def dest: nsp_has_no_Gets)

lemma nsp_is_one_step [iff]: "one_step nsp"
by (unfold one_step_def, clarify, ind_cases "ev#evs:nsp" for ev evs, auto)

lemma nsp_has_only_Says' [rule_format]: "evs:nsp ==>
ev:set evs --> (EX A B X. ev=Says A B X)"
by (erule nsp.induct, auto)

lemma nsp_has_only_Says [iff]: "has_only_Says nsp"
by (auto simp: has_only_Says_def dest: nsp_has_only_Says')

lemma nsp_is_regular [iff]: "regular nsp"
apply (simp only: regular_def, clarify)
by (erule nsp.induct, auto simp: initState.simps knows.simps)



## 41.5 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'"
apply (erule nsp.induct, simp_all)
by (blast intro: analz_insertI)+

```

```

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)"
apply (erule nsp.induct, simp_all)
by (blast intro: analz_insertI)+

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)"
by (rule no_Nonce_NS1_NS2, auto)

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'"
apply (erule nsp.induct, simp_all)
by (blast intro: analz_insertI)+

```

41.6 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)"
apply (erule nsp.induct)

apply simp_all

apply safe
apply (erule in_synth_Guard, erule Guard_analz, simp)

apply blast
apply blast
apply blast
apply (drule Nonce_neq, simp+, rule No_Nonce, simp)

apply (frule_tac A=A in Nonce_neq, simp+)
apply (case_tac "NAa=NA")
apply (drule Guard_Nonce_analz, simp+)
apply (drule Says_imp_knows_Spy)+
apply (drule_tac B=B and A'=Aa in NA_is_uniq, auto)

apply (case_tac "NB=NA", clarify)
apply (drule Guard_Nonce_analz, simp+)
apply (drule Says_imp_knows_Spy)+
by (drule no_Nonce_NS1_NS2, auto)

```

41.7 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)"
apply (erule nsp.induct)

```

```

apply simp_all

apply safe
apply (erule in_synth_Guard, erule Guard_analz, simp)

apply (frule Nonce_neq, simp+, blast, rule No_Nonce, simp)

apply blast
apply blast
apply blast
apply (frule_tac A=B and n=NB in Nonce_neq, simp+)
apply (case_tac "NAa=NB")
apply (drule Guard_Nonce_analz, simp+)
apply (drule Says_imp_knows_Spy)+
apply (drule no_Nonce_NS1_NS2, auto)

apply (case_tac "NBa=NB", clarify)
apply (drule Guard_Nonce_analz, simp+)
apply (drule Says_imp_knows_Spy)+
by (drule_tac A=Aa and A'=A in NB_is_uniq, auto)

```

41.8 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"
apply (erule nsp.induct, simp_all)
by (blast intro: analz_insertI)+

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"
apply (erule nsp.induct, simp_all, safe)
apply (frule_tac B=B in ns1_imp_Guard, simp+)
apply (drule Guard_Nonce_analz, simp+, blast)
apply (frule_tac B=B in ns1_imp_Guard, simp+)
apply (drule Guard_Nonce_analz, simp+, blast)
apply (frule_tac B=B in ns1_imp_Guard, simp+)
by (drule Guard_Nonce_analz, simp+, blast+)

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"
apply (erule nsp.induct, simp_all, safe)
apply (frule_tac B=B in ns2_imp_Guard, simp+)
apply (drule Guard_Nonce_analz, simp+, blast)
apply (frule_tac B=B in ns2_imp_Guard, simp+)
apply (drule Guard_Nonce_analz, simp+, blast)
apply (frule_tac B=B in ns2_imp_Guard, simp+)
apply (drule Guard_Nonce_analz, simp+, blast, blast)
apply (frule_tac B=B in ns2_imp_Guard, simp+)
by (drule Guard_Nonce_analz, auto dest: Says_imp_knows_Spy NB_is_uniq)

end

```

42 Other Protocol-Independent Results

theory *Proto* imports *Guard_Public* begin

42.1 protocols

type_synonym *rule* = "event set * event"

abbreviation

```
msg' :: "rule => msg" where
  "msg' R == msg (snd R)"
```

type_synonym *proto* = "rule set"

definition *wdef* :: "proto => bool" where

```
"wdef p == ALL R k. R:p --> Number k:parts {msg' R}
--> Number k:parts (msg'(fst R))"
```

42.2 substitutions

record *subs* =

```
agent  :: "agent => agent"
nonce  :: "nat => nat"
nb     :: "nat => msg"
key    :: "key => key"
```

primrec *apm* :: "subs => msg => msg" where

```
"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}"

apply (erule parts.induct, simp_all, blast)

apply (erule parts.Fst)

apply (erule parts.Snd)

by (erule parts.Body)+

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)"
```

by (induct X, simp_all, blast)

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)"
```

```

apply (erule Nonce_apm, unfold wdef_def)
apply (drule_tac x=R in spec, drule_tac x=k in spec, clarsimp)
apply (drule_tac x=x in bspec, simp)
apply (drule_tac Y="msg x" and s=s in apm_parts, simp)
by (blast dest: parts_parts)

```

```

primrec ap :: "subs => event => event" where
  "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)"

```

42.3 nonces generated by a rule

```

definition newn :: "rule => nat set" where
  "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}"
by (auto simp: newn_def dest: apm_parts)

```

42.4 traces generated by a protocol

```

definition ok :: "event list => rule => subs => bool" where
  "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"

```

42.5 general properties

```
lemma one_step_tr [iff]: "one_step (tr p)"
apply (unfold one_step_def, clarify)
by (ind_cases "ev # evs:tr p" for ev evs, auto)
```

```
definition has_only_Says' :: "proto => bool" where
"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)"
by (unfold has_only_Says'_def is_Says_def, blast)
```

```
lemma has_only_Says_tr [simp]: "has_only_Says' p ==> has_only_Says (tr p)"
apply (unfold has_only_Says_def)
apply (rule allI, rule allI, rule impI)
apply (erule tr.induct)
apply (auto simp: has_only_Says'_def ok_def)
by (drule_tac x=a in spec, auto simp: is_Says_def)
```

```
lemma has_only_Says'_in_trD: "[| has_only_Says' p; list @ ev # evs1 ∈ tr
p |]
==> (EX A B X. ev = Says A B X)"
by (drule has_only_Says_tr, auto)
```

```
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)))"
apply (unfold ok_def, clarsimp)
apply (drule_tac x=x in spec, drule_tac x=x in spec)
by (auto simp: is_Says_def dest: Says_imp_spies not_used_not_spied parts_parts)
```

```
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"
apply (unfold ok_def is_Says_def, clarify)
apply (drule_tac x=x in spec, simp)
apply (subgoal_tac "one_step (tr p)")
apply (drule trunc, simp, drule one_step_Cons, simp)
apply (drule has_only_SaysD, simp+)
by (clarify, case_tac x, auto)
```

42.6 types

```
type_synonym keyfun = "rule => subs => nat => event list => key set"
```

```
type_synonym secfun = "rule => nat => subs => key set => msg"
```

42.7 introduction of a fresh guarded nonce

```
definition fresh :: "proto => rule => subs => nat => key set => event list
=> bool" where
"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)"
by (unfold fresh_def, blast)

```

```

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)"
by (unfold fresh_def, blast)

```

```

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)"
by (drule freshI, simp+)

```

```

lemma fresh_used: "[| fresh p R' s' n Ks evs; has_only_Says' p |]
==> Nonce n:used evs"
apply (unfold fresh_def, clarify)
apply (drule has_only_Says'D)
by (auto intro: parts_used_app)

```

```

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"
apply (drule wdef_Nonce, simp+)
apply (frule ok_not_used, simp+)
apply (clarify, erule ok_is_Says, simp+)
apply (clarify, rule_tac x=k in exI, simp add: newn_def)
apply (clarify, drule_tac Y="msg x" and s=s in apm_parts)
apply (drule ok_not_used, simp+)
by (clarify, erule ok_is_Says, simp+)

```

```

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"
apply (drule trunc, simp, ind_cases "ev # evs:tr p", simp)
by (drule_tac x=X in in_sub, drule parts_sub, simp, simp, blast+)

```

```

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"
apply (frule fresh_used, simp)
apply (unfold fresh_def, clarify)
apply (drule_tac x=R' in spec)
apply (drule fresh_newn, simp+, clarify)
apply (drule_tac x=k in spec)
apply (drule_tac x=s' in spec)
apply (subgoal_tac "apm' s' R':parts (spies (evs2 @ ap' s' R' # evs1))")
apply (case_tac R', drule has_only_Says'D, simp, clarsimp)
apply (case_tac R', drule has_only_Says'D, simp, clarsimp)
apply (rule_tac Y="apm s' X" in parts_parts, blast)
by (rule parts.Inj, rule Says_imp_spies, simp, blast)

```

42.8 safe keys

definition `safe` :: "key set => msg set => bool" where
`"safe Ks G == ALL K. K:Ks --> Key K ~:analz G"`

lemma `safeD [dest]`: "`[| safe Ks G; K:Ks |]` ==> `Key K ~:analz G`"
 by (unfold `safe_def`, blast)

lemma `safe_insert`: "`safe Ks (insert X G) ==> safe Ks G`"
 by (unfold `safe_def`, blast)

lemma `Guard_safe`: "`[| Guard n Ks G; safe Ks G |]` ==> `Nonce n ~:analz G`"
 by (blast dest: `Guard_invKey`)

42.9 guardedness preservation

definition `preserv` :: "proto => keyfun => nat => key set => bool" where
`"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`"
 by (unfold `preserv_def`, blast)

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`"
 by (drule `preservD`, simp+)

42.10 monotonic keyfun

definition `monoton` :: "proto => keyfun => bool" where
`"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`"
 by (unfold `monoton_def`, blast)

42.11 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)"`
 apply (erule `tr.induct`)

apply simp

apply (clarify, drule `freshD`, clarsimp)
 apply (case_tac `evs2`)

apply (frule `has_only_Says'D`, simp)

```

apply (clarsimp, blast)

apply (clarsimp, rule conjI)
apply (blast dest: safe_insert)

apply (rule in_synth_Guard, simp, rule Guard_analz)
apply (blast dest: safe_insert)
apply (drule safe_insert, simp add: safe_def)

apply (clarify, drule freshD, clarify)
apply (case_tac evs2)

apply (frule has_only_Says'D, simp)
apply (frule_tac R=R' in has_only_Says'D, simp)
apply (case_tac R', clarsimp, blast)

apply (frule has_only_Says'D, simp)
apply (clarsimp, rule conjI)
apply (drule Proto, simp+, blast dest: safe_insert)

apply (frule Proto, simp+)
apply (erule preservD', simp+)
apply (blast dest: safe_insert)
apply (blast dest: safe_insert)
by (blast, simp, simp, blast)

```

42.12 useful properties for guardedness

```

lemma newn_neq_used: "[| Nonce n:used evs; ok evs R s; k:newn R |]
==> n ~= nonce s k"
by (auto simp: ok_def)

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"
apply (unfold ok_def is_Says_def, clarify)
apply (drule_tac x="Says A B X" in spec, simp)
by (drule Says_imp_spies, auto intro: parts_parts)

lemma ok_parts_not_new: "[| Y:parts (spies evs); Nonce (nonce s n):parts
{Y};
ok evs R s |] ==> n ~:newn R"
by (auto simp: ok_def dest: not_used_not_spied parts_parts)

```

42.13 unicity

```

definition uniq :: "proto => secfun => bool" where
"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"
by (unfold uniq_def, blast)

definition ord :: "proto => (rule => rule => bool) => bool" where
"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"
by (unfold ord_def, blast)

definition uniq' :: "proto => (rule => rule => bool) => secfun => bool" where
"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"
by (unfold uniq'_def, blast)

lemma uniq'_imp_uniq: "[| uniq' p inff secret; ord p inff |] ==> uniq p secret"
apply (unfold uniq_def)
apply (rule allI)+
apply (case_tac "inff R R'")
apply (blast dest: uniq'D)
by (auto dest: ordD uniq'D intro: sym)

```

42.14 Needham-Schroeder-Lowe

```

definition a :: agent where "a == Friend 0"
definition b :: agent where "b == Friend 1"
definition a' :: agent where "a' == Friend 2"
definition b' :: agent where "b' == Friend 3"
definition Na :: nat where "Na == 0"
definition Nb :: nat where "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|})"

definition keys :: "keyfun" where
  "keys R' s' n evs == {priK' s' a, priK' s' b}"

lemma "monoton ns keys"
by (simp add: keys_def monoton_def)

definition secret :: "secfun" where
  "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)"

definition inf :: "rule => rule => bool" where
  "inf R R' == (R=ns1 | (R=ns2 & R'~ns1) | (R=ns3 & R'=ns3))"

lemma inf_is_ord [iff]: "ord ns inf"
apply (unfold ord_def inf_def)
apply (rule allI)+
apply (rule impI)
apply (simp add: split_paired_all)
by (rule impI, erule ns.cases, simp_all)+

42.15 general properties

lemma ns_has_only_Says' [iff]: "has_only_Says' ns"
apply (unfold has_only_Says'_def)
apply (rule allI, rule impI)

```

```

apply (simp add: split_paired_all)
by (erule ns.cases, auto)

```

```

lemma newn_ns1 [iff]: "newn ns1 = {Na}"
by (simp add: newn_def)

```

```

lemma newn_ns2 [iff]: "newn ns2 = {Nb}"
by (auto simp: newn_def Na_def Nb_def)

```

```

lemma newn_ns3 [iff]: "newn ns3 = {}"
by (auto simp: newn_def)

```

```

lemma ns_wdef [iff]: "wdef ns"
by (auto simp: wdef_def elim: ns.cases)

```

42.16 guardedness for NSL

```

lemma "uniq ns secret ==> preserv ns keys n Ks"
apply (unfold preserv_def)
apply (rule allI)+
apply (rule impI, rule impI, rule impI, rule impI, rule impI)
apply (erule fresh_ruleD, simp, simp, simp, simp)
apply (rule allI)+
apply (rule impI, rule impI, rule impI)
apply (simp add: split_paired_all)
apply (erule ns.cases)

apply (rule impI, rule impI, rule impI, rule impI, rule impI, rule impI)
apply (erule ns.cases)

apply clarsimp
apply (frule newn_neq_used, simp, simp)
apply (rule No_Nonce, simp)

apply clarsimp
apply (frule newn_neq_used, simp, simp)
apply (case_tac "nonce sa Na = nonce s Na")
apply (frule Guard_safe, simp)
apply (frule Crypt_guard_invKey, simp)
apply (frule ok_Guard, simp, simp, simp, clarsimp)
apply (frule_tac K="pubK' s b" in Crypt_guard_invKey, simp)
apply (frule_tac R=ns1 and R'=ns1 and Ks=Ks and s=sa and s'=s in uniqD,
simp+)
apply (simp add: secret_def, simp add: secret_def, force, force)
apply (simp add: secret_def keys_def, blast)
apply (rule No_Nonce, simp)

apply clarsimp
apply (case_tac "nonce sa Na = nonce s Nb")
apply (frule Guard_safe, simp)
apply (frule Crypt_guard_invKey, simp)
apply (frule_tac x=ns3b in ok_Guard, simp, simp, simp, clarsimp)
apply (frule_tac K="pubK' s a" in Crypt_guard_invKey, simp)
apply (frule_tac R=ns1 and R'=ns2 and Ks=Ks and s=sa and s'=s in uniqD,

```

```

simp+)
apply (simp add: secret_def, simp add: secret_def, force, force)
apply (simp add: secret_def, rule No_Nonce, simp)

apply (rule impI, rule impI, rule impI, rule impI, rule impI, rule impI)
apply (erule ns.cases)

apply clarsimp
apply (frule newn_neq_used, simp, simp)
apply (rule No_Nonce, simp)

apply clarsimp
apply (frule newn_neq_used, simp, simp)
apply (case_tac "nonce sa Nb = nonce s Na")
apply (frule Guard_safe, simp)
apply (frule Crypt_guard_invKey, simp)
apply (frule ok_Guard, simp, simp, simp, clarsimp)
apply (frule_tac K="pubK' s b" in Crypt_guard_invKey, simp)
apply (frule_tac R=ns2 and R'=ns1 and Ks=Ks and s=sa and s'=s in uniqD,
simp+)
apply (simp add: secret_def, simp add: secret_def, force, force)
apply (simp add: secret_def, rule No_Nonce, simp)

apply clarsimp
apply (case_tac "nonce sa Nb = nonce s Nb")
apply (frule Guard_safe, simp)
apply (frule Crypt_guard_invKey, simp)
apply (frule_tac x=ns3b in ok_Guard, simp, simp, simp, clarsimp)
apply (frule_tac K="pubK' s a" in Crypt_guard_invKey, simp)
apply (frule_tac R=ns2 and R'=ns2 and Ks=Ks and s=sa and s'=s in uniqD,
simp+)
apply (simp add: secret_def, simp add: secret_def, force, force)
apply (simp add: secret_def keys_def, blast)
apply (rule No_Nonce, simp)

by simp

```

42.17 unicity for NSL

```

lemma "uniq' ns inf secret"
apply (unfold uniq'_def)
apply (rule allI)+
apply (simp add: split_paired_all)
apply (rule impI, erule ns.cases)

apply (rule impI, erule ns.cases)

apply (rule impI, rule impI, rule impI, rule impI)
apply (rule impI, rule impI, rule impI, rule impI)
apply (rule impI, erule tr.induct)

apply (simp add: secret_def)

apply (clarify, simp add: secret_def)

```

```

apply (drule notin_analz_insert)
apply (drule Crypt_insert_synth, simp, simp, simp)
apply (drule Crypt_insert_synth, simp, simp, simp, simp)

apply (erule_tac P="ok evsa R sa" in rev_mp)
apply (simp add: split_paired_all)
apply (erule ns.cases)

apply (clarify, simp add: secret_def)
apply (erule disjE, erule disjE, clarsimp)
apply (drule ok_parts_not_new, simp, simp, simp)
apply (clarify, drule ok_parts_not_new, simp, simp, simp)

apply (simp add: secret_def)

apply (simp add: secret_def)

apply (rule impI, rule impI, rule impI, rule impI)
apply (rule impI, rule impI, rule impI, rule impI)
apply (rule impI, erule tr.induct)

apply (simp add: secret_def)

apply (clarify, simp add: secret_def)
apply (drule notin_analz_insert)
apply (drule Crypt_insert_synth, simp, simp, simp)
apply (drule_tac n="nonce s' Nb" in Crypt_insert_synth, simp, simp, simp,
simp)

apply (erule_tac P="ok evsa R sa" in rev_mp)
apply (simp add: split_paired_all)
apply (erule ns.cases)

apply (clarify, simp add: secret_def)
apply (drule_tac s=sa and n=Na in ok_parts_not_new, simp, simp, simp)

apply (clarify, simp add: secret_def)
apply (drule_tac s=sa and n=Nb in ok_parts_not_new, simp, simp, simp)

apply (simp add: secret_def)

apply simp

apply (rule impI, erule ns.cases)

apply (simp only: inf_def, blast)

apply (rule impI, rule impI, rule impI, rule impI)
apply (rule impI, rule impI, rule impI, rule impI)
apply (rule impI, erule tr.induct)

apply (simp add: secret_def)

apply (clarify, simp add: secret_def)

```

```

apply (drule notin_analz_insert)
apply (drule_tac n="nonce s' Nb" in Crypt_insert_synth, simp, simp, simp)
apply (drule_tac n="nonce s' Nb" in Crypt_insert_synth, simp, simp, simp,
simp)

apply (erule_tac P="ok evsa R sa" in rev_mp)
apply (simp add: split_paired_all)
apply (erule ns.cases)

apply (simp add: secret_def)

apply (clarify, simp add: secret_def)
apply (erule disjE, erule disjE, clarsimp, clarsimp)
apply (drule_tac s=sa and n=Nb in ok_parts_not_new, simp, simp, simp)
apply (erule disjE, clarsimp)
apply (drule_tac s=sa and n=Nb in ok_parts_not_new, simp, simp, simp)
by (simp_all add: secret_def)

end

```

43 Blanqui's "guard" concept: protocol-independent secrecy

```

theory Auth_Guard_Public
imports
  "P1"
  "P2"
  "Guard_NS_Public"
  "Proto"
begin

end

```