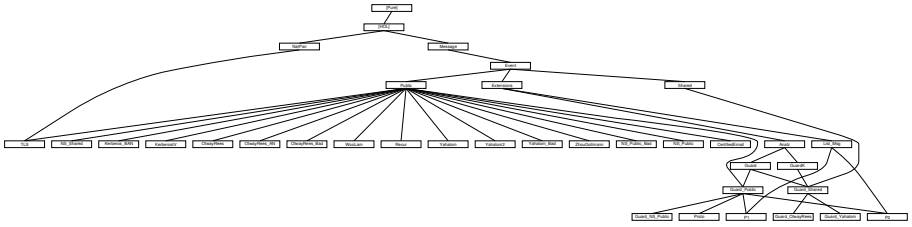
Security Protocols

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

October 1, 2005

Contents



1 Theory of Agents and Messages for Security Protocols

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

lemma *[simp] : "A ∪ (B ∪ A) = B ∪ A"*
⟨*proof*⟩

types
key = nat

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

specification *(invKey)*
invKey [simp]: "invKey (invKey K) = K"
invKey_symmetric: "all_symmetric --> invKey = id"
⟨*proof*⟩

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

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

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

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

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

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

syntax *(xsymbols)*
*"@MTuple" :: "['a, args] => 'a * 'b" ("(2⦃_,/ _⦄)")*

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

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

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

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

1.0.1 Inductive Definition of All Parts" of a Message

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

Monotonicity

lemma *parts_mono: "G ⊆ H ==> parts(G) ⊆ parts(H)"*
⟨*proof*⟩

Equations hold because constructors are injective.

lemma *Friend_image_eq [simp]: "(Friend x ∈ Friend‘A) = (x:A)"*
⟨*proof*⟩

lemma *Key_image_eq [simp]: "(Key x ∈ Key‘A) = (x∈A)"*
⟨*proof*⟩

lemma *Nonce_Key_image_eq [simp]: "(Nonce x ∉ Key‘A)"*
⟨*proof*⟩

1.0.2 Inverse of keys

lemma *invKey_eq [simp]: "(invKey K = invKey K') = (K=K')"*
⟨*proof*⟩

1.1 keysFor operator

lemma *keysFor_empty [simp]: "keysFor {} = {}"*
⟨*proof*⟩

lemma *keysFor_Un [simp]: "keysFor (H ∪ H') = keysFor H ∪ keysFor H'"*
⟨*proof*⟩

lemma *keysFor_UN [simp]: "keysFor (⋃i∈A. H i) = (⋃i∈A. keysFor (H i))"*
⟨*proof*⟩

Monotonicity

lemma *keysFor_mono: "G ⊆ H ==> keysFor(G) ⊆ keysFor(H)"*
⟨*proof*⟩

lemma *keysFor_insert_Agent [simp]: "keysFor (insert (Agent A) H) = keysFor H"*
⟨*proof*⟩

lemma *keysFor_insert_Nonce [simp]: "keysFor (insert (Nonce N) H) = keysFor H"*
⟨*proof*⟩

lemma *keysFor_insert_Number [simp]: "keysFor (insert (Number N) H) = keysFor H"*
⟨*proof*⟩

lemma *keysFor_insert_Key [simp]: "keysFor (insert (Key K) H) = keysFor H"*
⟨*proof*⟩

lemma *keysFor_insert_Hash [simp]: "keysFor (insert (Hash X) H) = keysFor H"*
⟨*proof*⟩

lemma *keysFor_insert_MPair [simp]: "keysFor (insert {|X,Y|} H) = keysFor H"*
⟨*proof*⟩

lemma *keysFor_insert_Crypt [simp]:*
"keysFor (insert (Crypt K X) H) = insert (invKey K) (keysFor H)"
⟨*proof*⟩

lemma *keysFor_image_Key [simp]: "keysFor (Key‘E) = {}"*
⟨*proof*⟩

lemma *Crypt_imp_invKey_keysFor: "Crypt K X ∈ H ==> invKey K ∈ keysFor H"*
⟨*proof*⟩

1.2 Inductive relation "parts"

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

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

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

lemma *parts_increasing: "H ⊆ parts(H)"*
⟨*proof*⟩

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

lemma *parts_empty [simp]: "parts{} = {}"*
⟨*proof*⟩

lemma *parts_emptyE [elim!]: "X∈ parts{} ==> P"*
⟨*proof*⟩

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

lemma *parts_singleton: "X∈ parts H ==> ∃Y∈H. X∈ parts {Y}"*
⟨*proof*⟩

1.2.1 Unions

lemma *parts_Un_subset1: "parts(G) ∪ parts(H) ⊆ parts(G ∪ H)"*
⟨*proof*⟩

lemma *parts_Un_subset2: "parts(G ∪ H) ⊆ parts(G) ∪ parts(H)"*
⟨*proof*⟩

lemma *parts_Un [simp]: "parts(G ∪ H) = parts(G) ∪ parts(H)"*
⟨*proof*⟩

lemma *parts_insert: "parts (insert X H) = parts {X} ∪ parts H"*
⟨*proof*⟩

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

lemma *parts_insert2:*
"parts (insert X (insert Y H)) = parts {X} ∪ parts {Y} ∪ parts H"
⟨*proof*⟩

lemma *parts_UN_subset1: "(⋃x∈A. parts(H x)) ⊆ parts(⋃x∈A. H x)"*
⟨*proof*⟩

lemma *parts_UN_subset2: "parts(⋃x∈A. H x) ⊆ (⋃x∈A. parts(H x))"*
⟨*proof*⟩

lemma *parts_UN [simp]: "parts(⋃x∈A. H x) = (⋃x∈A. parts(H x))"*
⟨*proof*⟩

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

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

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

declare *in_parts_UnE [elim!]*

lemma *parts_insert_subset: "insert X (parts H) ⊆ parts(insert X H)"*
⟨*proof*⟩

1.2.2 Idempotence and transitivity

lemma *parts_partsD [dest!]: "X∈ parts (parts H) ==> X∈ parts H"*
⟨*proof*⟩

lemma *parts_idem [simp]: "parts (parts H) = parts H"*
⟨*proof*⟩

lemma *parts_subset_iff [simp]: "(parts G ⊆ parts H) = (G ⊆ parts H)"*

⟨*proof*⟩

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

Cut

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

lemma *parts_cut_eq [simp]: "X∈ parts H ==> parts (insert X H) = parts H"*
⟨*proof*⟩

1.2.3 Rewrite rules for pulling out atomic messages

lemmas *parts_insert_eq_I = equalityI [OF subsetI parts_insert_subset]*

lemma *parts_insert_Agent [simp]:*
"parts (insert (Agent agt) H) = insert (Agent agt) (parts H)"
⟨*proof*⟩

lemma *parts_insert_Nonce [simp]:*
"parts (insert (Nonce N) H) = insert (Nonce N) (parts H)"
⟨*proof*⟩

lemma *parts_insert_Number [simp]:*
"parts (insert (Number N) H) = insert (Number N) (parts H)"
⟨*proof*⟩

lemma *parts_insert_Key [simp]:*
"parts (insert (Key K) H) = insert (Key K) (parts H)"
⟨*proof*⟩

lemma *parts_insert_Hash [simp]:*
"parts (insert (Hash X) H) = insert (Hash X) (parts H)"
⟨*proof*⟩

lemma *parts_insert_Crypt [simp]:*
"parts (insert (Crypt K X) H) = insert (Crypt K X) (parts (insert X H))"
⟨*proof*⟩

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

lemma *parts_image_Key [simp]: "parts (Key‘N) = Key‘N"*
⟨*proof*⟩

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

lemma *msg_Nonce_supply: "∃N. ∀n. N≤n --> Nonce n ∉ parts {msg}"*
⟨*proof*⟩

1.3 Inductive relation "analz"

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

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

Monotonicity; Lemma 1 of Lowe's paper

```
lemma analz_mono: "G⊆H ==> analz(G) ⊆ analz(H)"
```
⟨*proof*⟩

Making it safe speeds up proofs

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

```
lemma analz_increasing: "H ⊆ analz(H)"
```
⟨*proof*⟩

```
lemma analz_subset_parts: "analz H ⊆ parts H"
```
⟨*proof*⟩

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

lemmas not_parts_not_analz = analz_subset_parts [THEN contra_subsetD, standard]


lemma parts_analz [simp]: "parts (analz H) = parts H"
```
⟨*proof*⟩

```
lemma analz_parts [simp]: "analz (parts H) = parts H"
```
⟨*proof*⟩

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

1.3.1 General equational properties

```
lemma analz_empty [simp]: "analz{} = {}"
```
⟨*proof*⟩

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

lemma *analz_Un: "analz(G) ∪ analz(H) ⊆ analz(G ∪ H)"*
⟨*proof*⟩

lemma *analz_insert: "insert X (analz H) ⊆ analz(insert X H)"*
⟨*proof*⟩

1.3.2 Rewrite rules for pulling out atomic messages

lemmas *analz_insert_eq_I = equalityI [OF subsetI analz_insert]*

lemma *analz_insert_Agent [simp]:*
"analz (insert (Agent agt) H) = insert (Agent agt) (analz H)"
⟨*proof*⟩

lemma *analz_insert_Nonce [simp]:*
"analz (insert (Nonce N) H) = insert (Nonce N) (analz H)"
⟨*proof*⟩

lemma *analz_insert_Number [simp]:*
"analz (insert (Number N) H) = insert (Number N) (analz H)"
⟨*proof*⟩

lemma *analz_insert_Hash [simp]:*
"analz (insert (Hash X) H) = insert (Hash X) (analz H)"
⟨*proof*⟩

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

lemma *analz_insert_Key [simp]:*
"K ∉ keysFor (analz H) ==>
analz (insert (Key K) H) = insert (Key K) (analz H)"
⟨*proof*⟩

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

Can pull out enCrypted message if the Key is not known

lemma *analz_insert_Crypt:*
"Key (invKey K) ∉ analz H
==> analz (insert (Crypt K X) H) = insert (Crypt K X) (analz H)"
⟨*proof*⟩

lemma *lemma1: "Key (invKey K) ∈ analz H ==>*
analz (insert (Crypt K X) H) ⊆
insert (Crypt K X) (analz (insert X H))"
⟨*proof*⟩

lemma *lemma2: "Key (invKey K) ∈ analz H ==>*
insert (Crypt K X) (analz (insert X H)) ⊆
analz (insert (Crypt K X) H)"
⟨*proof*⟩

lemma *analz_insert_Decrypt:*

```
"Key (invKey K) ∈ analz H ==>
          analz (insert (Crypt K X) H) =
          insert (Crypt K X) (analz (insert X H))"
```
⟨*proof*⟩

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

lemma *analz_Crypt_if [simp]:*
```
"analz (insert (Crypt K X) H) =
     (if (Key (invKey K) ∈ analz H)
      then insert (Crypt K X) (analz (insert X H))
      else insert (Crypt K X) (analz H))"
```
⟨*proof*⟩

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

lemma *analz_insert_Crypt_subset:*
```
"analz (insert (Crypt K X) H) ⊆
      insert (Crypt K X) (analz (insert X H))"
```
⟨*proof*⟩

lemma *analz_image_Key [simp]: "analz (Key‘N) = Key‘N"*
⟨*proof*⟩

1.3.3 Idempotence and transitivity

lemma *analz_analzD [dest!]: "X∈ analz (analz H) ==> X∈ analz H"*
⟨*proof*⟩

lemma *analz_idem [simp]: "analz (analz H) = analz H"*
⟨*proof*⟩

lemma *analz_subset_iff [simp]: "(analz G ⊆ analz H) = (G ⊆ analz H)"*
⟨*proof*⟩

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

Cut; Lemma 2 of Lowe

lemma *analz_cut: "[| Y∈ analz (insert X H); X∈ analz H |] ==> Y∈ analz H"*
⟨*proof*⟩

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

lemma *analz_insert_eq: "X∈ analz H ==> analz (insert X H) = analz H"*
⟨*proof*⟩

A congruence rule for "analz"

lemma *analz_subset_cong:*

```
"[| analz G ⊆ analz G'; analz H ⊆ analz H' |]
 ==> analz (G ∪ H) ⊆ analz (G' ∪ H')"
```
⟨*proof*⟩

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

lemma *analz_insert_cong:*
```
"analz H = analz H' ==> analz(insert X H) = analz(insert X H')"
```
⟨*proof*⟩

If there are no pairs or encryptions then analz does nothing

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

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

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

lemma *analz_UN_analz [simp]: "analz (⋃i∈A. analz (H i)) = analz (⋃i∈A. H i)"*
⟨*proof*⟩

1.4 Inductive relation "synth"

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

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

Monotonicity

lemma *synth_mono: "G⊆H ==> synth(G) ⊆ synth(H)"*
⟨*proof*⟩

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

inductive_cases *Nonce_synth [elim!]: "Nonce n ∈ synth H"*

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

lemma *synth_increasing: "H ⊆ synth(H)"*
⟨*proof*⟩

1.4.1 Unions

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

lemma *synth_Un: "synth(G) ∪ synth(H) ⊆ synth(G ∪ H)"*
⟨*proof*⟩

lemma *synth_insert: "insert X (synth H) ⊆ synth(insert X H)"*
⟨*proof*⟩

1.4.2 Idempotence and transitivity

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

lemma *synth_idem: "synth (synth H) = synth H"*
⟨*proof*⟩

lemma *synth_subset_iff [simp]: "(synth G ⊆ synth H) = (G ⊆ synth H)"*
⟨*proof*⟩

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

Cut; Lemma 2 of Lowe

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

lemma *Agent_synth [simp]: "Agent A ∈ synth H"*
⟨*proof*⟩

lemma *Number_synth [simp]: "Number n ∈ synth H"*
⟨*proof*⟩

lemma *Nonce_synth_eq [simp]: "(Nonce N ∈ synth H) = (Nonce N ∈ H)"*
⟨*proof*⟩

lemma *Key_synth_eq [simp]: "(Key K ∈ synth H) = (Key K ∈ H)"*
⟨*proof*⟩

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

lemma *keysFor_synth [simp]:*
"keysFor (synth H) = keysFor H ∪ invKey‘{K. Key K ∈ H}"
⟨*proof*⟩

1.4.3 Combinations of parts, analz and synth

lemma *parts_synth [simp]: "parts (synth H) = parts H ∪ synth H"*
⟨*proof*⟩

lemma *analz_analz_Un [simp]: "analz (analz G ∪ H) = analz (G ∪ H)"*
⟨*proof*⟩

lemma *analz_synth_Un [simp]: "analz (synth G ∪ H) = analz (G ∪ H) ∪ synth G"*
⟨*proof*⟩

lemma *analz_synth [simp]: "analz (synth H) = analz H ∪ synth H"*
⟨*proof*⟩

1.4.4 For reasoning about the Fake rule in traces

lemma *parts_insert_subset_Un: "X∈ G ==> parts(insert X H) ⊆ parts G ∪ parts H"*
⟨*proof*⟩

More specifically for Fake. Very occasionally we could do with a version of the form *parts {X} ⊆ synth (analz H) ∪ parts H*

lemma *Fake_parts_insert:*
"X ∈ synth (analz H) ==>
parts (insert X H) ⊆ synth (analz H) ∪ parts H"
⟨*proof*⟩

lemma *Fake_parts_insert_in_Un:*
"[|Z ∈ parts (insert X H); X: synth (analz H)|]
==> Z ∈ synth (analz H) ∪ parts H"
⟨*proof*⟩

H is sometimes *Key ‘ KK ∪ spies evs*, so can’t put *G = H*.

lemma *Fake_analz_insert:*
"X∈ synth (analz G) ==>
analz (insert X H) ⊆ synth (analz G) ∪ analz (G ∪ H)"
⟨*proof*⟩

lemma *analz_conj_parts [simp]:*
"(X ∈ analz H & X ∈ parts H) = (X ∈ analz H)"
⟨*proof*⟩

lemma *analz_disj_parts [simp]:*
"(X ∈ analz H | X ∈ parts H) = (X ∈ parts H)"
⟨*proof*⟩

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

lemma *MPair_synth_analz [iff]:*
```
"({|X,Y|} ∈ synth (analz H)) =
 (X ∈ synth (analz H) & Y ∈ synth (analz H))"
```
⟨*proof*⟩

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

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

1.5 HPair: a combination of Hash and MPair

1.5.1 Freeness

lemma *Agent_neq_HPair: "Agent A ~= Hash[X] Y"*
⟨*proof*⟩

lemma *Nonce_neq_HPair: "Nonce N ~= Hash[X] Y"*
⟨*proof*⟩

lemma *Number_neq_HPair: "Number N ~= Hash[X] Y"*
⟨*proof*⟩

lemma *Key_neq_HPair: "Key K ~= Hash[X] Y"*
⟨*proof*⟩

lemma *Hash_neq_HPair: "Hash Z ~= Hash[X] Y"*
⟨*proof*⟩

lemma *Crypt_neq_HPair: "Crypt K X' ~= Hash[X] Y"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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

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

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

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

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

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

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

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

declare *parts.Body [rule del]*

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

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

lemmas *pushes = pushKeys pushCrypts*

1.6 Tactics useful for many protocol proofs

⟨*ML*⟩
By default only *o_apply* is built-in. But in the presence of eta-expansion this means that some terms displayed as *f* ∘ *g* will be rewritten, and others will not!

declare *o_def [simp]*

lemma *Crypt_notin_image_Key [simp]: "Crypt K X ∉ Key ‘ A"*
⟨*proof*⟩

lemma *Hash_notin_image_Key [simp] :"Hash X ∉ Key ‘ A"*
⟨*proof*⟩

lemma *synth_analz_mono: "G⊆H ==> synth (analz(G)) ⊆ synth (analz(H))"*
⟨*proof*⟩

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

Two generalizations of *analz_insert_eq*

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

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

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

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

⟨*ML*⟩

end

2 Theory of Events for Security Protocols

theory *Event* **imports** *Message* **begin**

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

datatype
```
  event = Says  agent agent msg
        | Gets  agent       msg
        | Notes agent       msg
```

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

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

syntax
```
  spies  :: "event list => msg set"
```

translations
```
  "spies"   => "knows Spy"
```

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

specification *(bad)*
```
  Spy_in_bad     [iff]: "Spy ∈ bad"
  Server_not_bad [iff]: "Server ∉ bad"
```
⟨*proof*⟩

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

consts
```
  used :: "event list => msg set"
```

primrec
```
  used_Nil:   "used []         = (UN B. parts (initState B))"
  used_Cons:  "used (ev # evs) =
                      (case ev of
                         Says A B X => parts {X} ∪ used evs
                       | Gets A X   => used evs
                       | Notes A X  => parts {X} ∪ used evs)"
```
— The case for *Gets* seems anomalous, but *Gets* always follows *Says* in real protocols. Seems difficult to change. See *Gets_correct* in theory *Guard/Extensions.thy*.

lemma *Notes_imp_used [rule_format]: "Notes A X ∈ set evs --> X ∈ used evs"*
⟨*proof*⟩

lemma *Says_imp_used [rule_format]: "Says A B X ∈ set evs --> X ∈ used evs"*
⟨*proof*⟩

lemma *MPair_used [rule_format]:*
"MPair X Y ∈ used evs --> X ∈ used evs & Y ∈ used evs"
⟨*proof*⟩

2.1 Function *knows*

lemmas *parts_insert_knows_A = parts_insert [of _ "knows A evs", standard]*

lemma *knows_Spy_Says [simp]:*
"knows Spy (Says A B X # evs) = insert X (knows Spy evs)"
⟨*proof*⟩

Letting the Spy see "bad" agents' notes avoids redundant case-splits on whether `A = Spy` and whether `A ∈ bad`

lemma *knows_Spy_Notes [simp]:*
```
"knows Spy (Notes A X # evs) =
     (if A:bad then insert X (knows Spy evs) else knows Spy evs)"
```
⟨*proof*⟩

lemma *knows_Spy_Gets [simp]: "knows Spy (Gets A X # evs) = knows Spy evs"*
⟨*proof*⟩

lemma *knows_Spy_subset_knows_Spy_Says:*
```
"knows Spy evs ⊆ knows Spy (Says A B X # evs)"
```
⟨*proof*⟩

lemma *knows_Spy_subset_knows_Spy_Notes:*
```
"knows Spy evs ⊆ knows Spy (Notes A X # evs)"
```
⟨*proof*⟩

lemma *knows_Spy_subset_knows_Spy_Gets:*
```
"knows Spy evs ⊆ knows Spy (Gets A X # evs)"
```
⟨*proof*⟩

Spy sees what is sent on the traffic

lemma *Says_imp_knows_Spy [rule_format]:*
```
"Says A B X ∈ set evs --> X ∈ knows Spy evs"
```
⟨*proof*⟩

lemma *Notes_imp_knows_Spy [rule_format]:*
```
"Notes A X ∈ set evs --> A: bad --> X ∈ knows Spy evs"
```
⟨*proof*⟩

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

lemmas *knows_Spy_partsEs =*
```
Says_imp_knows_Spy [THEN parts.Inj, THEN revcut_rl, standard]
parts.Body [THEN revcut_rl, standard]
```

Compatibility for the old "spies" function

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

2.2 Knowledge of Agents

lemma *knows_Says: "knows A (Says A B X # evs) = insert X (knows A evs)"*
⟨*proof*⟩

lemma *knows_Notes: "knows A (Notes A X # evs) = insert X (knows A evs)"*
⟨*proof*⟩

lemma *knows_Gets:*
```
"A ≠ Spy --> knows A (Gets A X # evs) = insert X (knows A evs)"
```
⟨*proof*⟩

lemma *knows_subset_knows_Says: "knows A evs ⊆ knows A (Says A' B X # evs)"*
⟨*proof*⟩

lemma *knows_subset_knows_Notes: "knows A evs ⊆ knows A (Notes A' X # evs)"*
⟨*proof*⟩

lemma *knows_subset_knows_Gets: "knows A evs ⊆ knows A (Gets A' X # evs)"*
⟨*proof*⟩

Agents know what they say

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

Agents know what they note

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

Agents know what they receive

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

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

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

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

lemma *knows_Spy_imp_Says_Notes_initState [rule_format]:*
"[| X ∈ knows Spy evs |] ==> EX A B.
Says A B X ∈ set evs | Notes A X ∈ set evs | X ∈ initState Spy"
⟨*proof*⟩

lemma *parts_knows_Spy_subset_used: "parts (knows Spy evs) ⊆ used evs"*
⟨*proof*⟩

lemmas *usedI = parts_knows_Spy_subset_used [THEN subsetD, intro]*

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

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

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

lemma *used_Gets [simp]: "used (Gets A X # evs) = used evs"*
⟨*proof*⟩

lemma *used_nil_subset: "used [] ⊆ used evs"*
⟨*proof*⟩

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

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

For proving theorems of the form *X ∉ analz (knows Spy evs) ⟶ 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]

⟨*ML*⟩

lemma *knows_subset_knows_Cons: "knows A evs ⊆ knows A (e # evs)"*
⟨*proof*⟩

lemma *initState_subset_knows: "initState A ⊆ knows A evs"*
⟨*proof*⟩

For proving *new_keys_not_used*

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

⟨*ML*⟩

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

⟨*ML*⟩

end

theory *Public* **imports** *Event* **begin**

lemma *invKey_K: "K ∈ symKeys ==> invKey K = K"*
⟨*proof*⟩

2.3 Asymmetric Keys

consts

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

syntax

```
pubEK :: "agent => key"
pubSK :: "agent => key"

privateKey :: "[bool,agent] => key"
priEK :: "agent => key"
priSK :: "agent => key"
```

translations

```
"pubEK"  == "publicKey False"
"pubSK"  == "publicKey True"


"privateKey b A" == "invKey (publicKey b A)"
"priEK A"  == "privateKey False A"
"priSK A"  == "privateKey True A"
```

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

syntax

```
pubK :: "agent => key"
priK :: "agent => key"
```

translations

```
"pubK A" == "pubEK A"
"priK A" == "invKey (pubEK A)"
```

By freeness of agents, no two agents have the same key. Since `True` $\neq$ `False`, no agent has identical signing and encryption keys

specification *(publicKey)*

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

⟨*proof*⟩

axioms

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

declare `privateKey_neq_publicKey [THEN not_sym, iff]`

2.4 Basic properties of `pubK` and `priK`

lemma `[iff]: "(publicKey b A = publicKey c A') = (b=c & A=A')"`
⟨*proof*⟩

lemma `not_symKeys_pubK [iff]: "publicKey b A ∉ symKeys"`
⟨*proof*⟩

lemma *not_symKeys_priK [iff]: "privateKey b A ∉ symKeys"*
⟨*proof*⟩

lemma *symKey_neq_priEK: "K ∈ symKeys ==> K ≠ priEK A"*
⟨*proof*⟩

lemma *symKeys_neq_imp_neq: "(K ∈ symKeys) ≠ (K' ∈ symKeys) ==> K ≠ K'"*
⟨*proof*⟩

lemma *symKeys_invKey_iff [iff]: "(invKey K ∈ symKeys) = (K ∈ symKeys)"*
⟨*proof*⟩

lemma *analz_symKeys_Decrypt:*
"[| Crypt K X ∈ analz H; K ∈ symKeys; Key K ∈ analz H |]
==> X ∈ analz H"
⟨*proof*⟩

2.5 "Image" equations that hold for injective functions

lemma *invKey_image_eq [simp]: "(invKey x ∈ invKey‘A) = (x ∈ A)"*
⟨*proof*⟩

lemma *publicKey_image_eq [simp]:*
"(publicKey b x ∈ publicKey c ‘ AA) = (b=c & x ∈ AA)"
⟨*proof*⟩

lemma *privateKey_notin_image_publicKey [simp]: "privateKey b x ∉ publicKey c ‘ AA"*
⟨*proof*⟩

lemma *privateKey_image_eq [simp]:*
"(privateKey b A ∈ invKey ‘ publicKey c ‘ AS) = (b=c & A∈AS)"
⟨*proof*⟩

lemma *publicKey_notin_image_privateKey [simp]: "publicKey b A ∉ invKey ‘ publicKey c ‘ AS"*
⟨*proof*⟩

2.6 Symmetric Keys

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

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

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

axioms

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

declare *inj_shrK [THEN inj_eq, iff]*

lemma *invKey_shrK [simp]: "invKey (shrK A) = shrK A"*
⟨*proof*⟩

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

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

lemma *priK_neq_shrK [iff]: "shrK A ≠ privateKey b C"*
⟨*proof*⟩

declare *priK_neq_shrK [THEN not_sym, simp]*

lemma *pubK_neq_shrK [iff]: "shrK A ≠ publicKey b C"*
⟨*proof*⟩

declare *pubK_neq_shrK [THEN not_sym, simp]*

lemma *priEK_noteq_shrK [simp]: "priEK A ≠ shrK B"*
⟨*proof*⟩

lemma *publicKey_notin_image_shrK [simp]: "publicKey b x ∉ shrK ‘ AA"*
⟨*proof*⟩

lemma *privateKey_notin_image_shrK [simp]: "privateKey b x ∉ shrK ‘ AA"*
⟨*proof*⟩

lemma *shrK_notin_image_publicKey [simp]: "shrK x ∉ publicKey b ‘ AA"*
⟨*proof*⟩

lemma *shrK_notin_image_privateKey [simp]: "shrK x ∉ invKey ‘ publicKey b ‘ AA"*
⟨*proof*⟩

lemma *shrK_image_eq [simp]: "(shrK x ∈ shrK ‘ AA) = (x ∈ AA)"*
⟨*proof*⟩

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

declare *invKey_K [simp]*

2.7 Initial States of Agents

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

primrec

```
  initState_Server:
    "initState Server     =
       {Key (priEK Server), Key (priSK Server)} ∪
       (Key ‘ range pubEK) ∪ (Key ‘ range pubSK) ∪ (Key ‘ range shrK)"

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

  initState_Spy:
    "initState Spy        =
       (Key ‘ invKey ‘ pubEK ‘ bad) ∪ (Key ‘ invKey ‘ pubSK ‘ bad) ∪
       (Key ‘ shrK ‘ bad) ∪
       (Key ‘ range pubEK) ∪ (Key ‘ range pubSK)"
```

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

lemma `used_parts_subset_parts [rule_format]:`
`"∀X ∈ used evs. parts {X} ⊆ used evs"`
⟨*proof*⟩

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

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

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

lemma `keysFor_parts_initState [simp]: "keysFor (parts (initState C)) = {}"`
⟨*proof*⟩

lemma `Crypt_notin_initState: "Crypt K X ∉ parts (initState B)"`
⟨*proof*⟩

lemma `Crypt_notin_used_empty [simp]: "Crypt K X ∉ used []"`
⟨*proof*⟩

lemma `shrK_in_initState [iff]: "Key (shrK A) ∈ initState A"`
⟨*proof*⟩

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

⟨*proof*⟩

lemma *shrK_in_used [iff]: "Key (shrK A) ∈ used evs"*
⟨*proof*⟩

lemma *Key_not_used [simp]: "Key K ∉ used evs ==> K ∉ range shrK"*
⟨*proof*⟩

lemma *shrK_neq: "Key K ∉ used evs ==> shrK B ≠ K"*
⟨*proof*⟩

declare *shrK_neq [THEN not_sym, simp]*

2.8 Function *knows Spy*

Agents see their own private keys!

lemma *priK_in_initState [iff]: "Key (privateKey b A) ∈ initState A"*
⟨*proof*⟩

Agents see all public keys!

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

All public keys are visible

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

declare *spies_pubK [THEN analz.Inj, iff]*

Spy sees private keys of bad agents!

lemma *Spy_spies_bad_privateKey [intro!]:*
"A ∈ bad ==> Key (privateKey b A) ∈ spies evs"
⟨*proof*⟩

Spy sees long-term shared keys of bad agents!

lemma *Spy_spies_bad_shrK [intro!]:*
"A ∈ bad ==> Key (shrK A) ∈ spies evs"
⟨*proof*⟩

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

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

lemma *Crypt_Spy_analz_bad:*
"[| Crypt (shrK A) X ∈ analz (knows Spy evs); A ∈ bad |]

```
     ==> X ∈ analz (knows Spy evs)"
```
⟨*proof*⟩

2.9 Fresh Nonces

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

lemma *Nonce_notin_used_empty [simp]: "Nonce N ∉ used []"*
⟨*proof*⟩

2.10 Supply fresh nonces for possibility theorems

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

lemma *Nonce_supply_lemma: "EX N. ALL n. N<=n --> Nonce n ∉ used evs"*
⟨*proof*⟩

lemma *Nonce_supply1: "EX N. Nonce N ∉ used evs"*
⟨*proof*⟩

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

2.11 Specialized Rewriting for Theorems About `analz` and Image

lemma *insert_Key_singleton: "insert (Key K) H = Key ‘ {K} Un H"*
⟨*proof*⟩

lemma *insert_Key_image: "insert (Key K) (Key‘KK ∪ C) = Key ‘ (insert K KK) ∪ C"*
⟨*proof*⟩

⟨*ML*⟩

lemma *Crypt_imp_keysFor :"[|Crypt K X ∈ H; K ∈ symKeys|] ==> K ∈ keysFor H"*
⟨*proof*⟩

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

lemma *analz_image_freshK_lemma:*
```
     "(Key K ∈ analz (Key‘nE ∪ H)) --> (K ∈ nE | Key K ∈ analz H)  ==>
         (Key K ∈ analz (Key‘nE ∪ H)) = (K ∈ nE | Key K ∈ analz H)"
```
⟨*proof*⟩

lemmas *analz_image_freshK_simps =*
simp_thms mem_simps — these two allow its use with *only:*
disj_comms
image_insert [THEN sym] image_Un [THEN sym] empty_subsetI insert_subset
analz_insert_eq Un_upper2 [THEN analz_mono, THEN subsetD]
insert_Key_singleton

```
Key_not_used insert_Key_image Un_assoc [THEN sym]
```

⟨*ML*⟩

2.12 Specialized Methods for Possibility Theorems

⟨*ML*⟩

end

3 The Needham-Schroeder Shared-Key Protocol

theory *NS_Shared* **imports** *Public* **begin**

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

```
consts  ns_shared   :: "event list set"
inductive "ns_shared"
 intros

  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"
⟨*proof*⟩

3.1 Inductive proofs about *ns_shared*

3.1.1 Forwarding lemmas, to aid simplification

For reasoning about the encrypted portion of message NS3

lemma *NS3_msg_in_parts_spies:*
"Says S A (Crypt KA {|N, B, K, X|}) ∈ set evs ⟹ X ∈ parts (spies evs)"
⟨*proof*⟩

For reasoning about the Oops message

lemma *Oops_parts_spies:*
"Says Server A (Crypt (shrK A) {|NA, B, K, X|}) ∈ set evs
⟹ K ∈ parts (spies evs)"
⟨*proof*⟩

Theorems of the form *X ∉ parts (knows Spy evs)* imply that NOBODY sends messages containing *X*

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

lemma *Spy_see_shrK [simp]:*
"evs ∈ ns_shared ⟹ (Key (shrK A) ∈ parts (spies evs)) = (A ∈ bad)"
⟨*proof*⟩

lemma *Spy_analz_shrK [simp]:*
"evs ∈ ns_shared ⟹ (Key (shrK A) ∈ analz (spies evs)) = (A ∈ bad)"
⟨*proof*⟩

Nobody can have used non-existent keys!

lemma *new_keys_not_used [simp]:*

```
    "[|Key K ∉ used evs; K ∈ symKeys; evs ∈ ns_shared|]
     ==> K ∉ keysFor (parts (spies evs))"
```
⟨*proof*⟩

3.1.2 Lemmas concerning the form of items passed in messages

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

lemma *Says_Server_message_form:*
```
     "⟦Says Server A (Crypt K' {|N, Agent B, Key K, X|}) ∈ set evs;
       evs ∈ ns_shared⟧
      ⟹ K ∉ range shrK ∧
          X = (Crypt (shrK B) {|Key K, Agent A|}) ∧
          K' = shrK A"
```
⟨*proof*⟩

If the encrypted message appears then it originated with the Server

lemma *A_trusts_NS2:*
```
     "⟦Crypt (shrK A) {|NA, Agent B, Key K, X|} ∈ parts (spies evs);
       A ∉ bad;  evs ∈ ns_shared⟧
      ⟹ Says Server A (Crypt (shrK A) {|NA, Agent B, Key K, X|}) ∈ set evs"
```
⟨*proof*⟩

lemma *cert_A_form:*
```
     "⟦Crypt (shrK A) {|NA, Agent B, Key K, X|} ∈ parts (spies evs);
       A ∉ bad;  evs ∈ ns_shared⟧
      ⟹ K ∉ range shrK ∧  X = (Crypt (shrK B) {|Key K, Agent A|})"
```
⟨*proof*⟩

EITHER describes the form of X when the following message is sent, OR reduces it to the Fake case. Use *Says_Server_message_form* if applicable.

lemma *Says_S_message_form:*
```
     "⟦Says S A (Crypt (shrK A) {|Nonce NA, Agent B, Key K, X|}) ∈ set evs;
       evs ∈ ns_shared⟧
      ⟹ (K ∉ range shrK ∧ X = (Crypt (shrK B) {|Key K, Agent A|}))
          ∨ X ∈ analz (spies evs)"
```
⟨*proof*⟩

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

lemma
```
  "⟦evs ∈ ns_shared;  Kab ∉ range shrK⟧ ⟹
   (Crypt KAB X) ∈ parts (spies evs) ∧
   Key K ∈ parts {X} ⟶ Key K ∈ parts (spies evs)"
```
⟨*proof*⟩

3.1.3 Session keys are not used to encrypt other session keys

The equality makes the induction hypothesis easier to apply

lemma *analz_image_freshK [rule_format]:*
```
 "evs ∈ ns_shared ⟹
   ∀K KK. KK ⊆ - (range shrK) ⟶
             (Key K ∈ analz (Key`KK ∪ (spies evs))) =
```

```
            (K ∈ KK ∨ Key K ∈ analz (spies evs))"
```
⟨*proof*⟩

lemma *analz_insert_freshK:*
```
     "⟦evs ∈ ns_shared;  KAB ∉ range shrK⟧ ⟹
       (Key K ∈ analz (insert (Key KAB) (spies evs))) =
       (K = KAB ∨ Key K ∈ analz (spies evs))"
```
⟨*proof*⟩

3.1.4 The session key K uniquely identifies the message

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

lemma *unique_session_keys:*
```
     "⟦Says Server A (Crypt (shrK A) {|NA, Agent B, Key K, X|}) ∈ set evs;
       Says Server A' (Crypt (shrK A') {|NA', Agent B', Key K, X'|}) ∈ set
evs;
       evs ∈ ns_shared⟧ ⟹ A=A' ∧ NA=NA' ∧ B=B' ∧ X = X'"
```
⟨*proof*⟩

3.1.5 Crucial secrecy property: Spy does not see the keys sent in msg NS2

Beware of *[rule_format]* and the universal quantifier!

lemma *secrecy_lemma:*
```
     "⟦Says Server A (Crypt (shrK A) {|NA, Agent B, Key K,
                                       Crypt (shrK B) {|Key K, Agent A|}|})
             ∈ set evs;
        A ∉ bad;  B ∉ bad;  evs ∈ ns_shared⟧
      ⟹ (∀NB. Notes Spy {|NA, NB, Key K|} ∉ set evs) ⟶
         Key K ∉ analz (spies evs)"
```
⟨*proof*⟩

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

lemma *Spy_not_see_encrypted_key:*
```
     "⟦Says Server A (Crypt K' {|NA, Agent B, Key K, X|}) ∈ set evs;
       ∀NB. Notes Spy {|NA, NB, Key K|} ∉ set evs;
       A ∉ bad;  B ∉ bad;  evs ∈ ns_shared⟧
      ⟹ Key K ∉ analz (spies evs)"
```
⟨*proof*⟩

3.2 Guarantees available at various stages of protocol

If the encrypted message appears then it originated with the Server

lemma *B_trusts_NS3:*
```
     "⟦Crypt (shrK B) {|Key K, Agent A|} ∈ parts (spies evs);
       B ∉ bad;  evs ∈ ns_shared⟧
      ⟹ ∃NA. Says Server A
               (Crypt (shrK A) {|NA, Agent B, Key K,
                                 Crypt (shrK B) {|Key K, Agent A|}|})
              ∈ set evs"
```
⟨*proof*⟩

lemma *A_trusts_NS4_lemma [rule_format]:*
```
"evs ∈ ns_shared ⟹
   Key K ∉ analz (spies evs) ⟶
   Says Server A (Crypt (shrK A) {|NA, Agent B, Key K, X|}) ∈ set evs ⟶
   Crypt K (Nonce NB) ∈ parts (spies evs) ⟶
   Says B A (Crypt K (Nonce NB)) ∈ set evs"
```
⟨*proof*⟩

This version no longer assumes that K is secure

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

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

theorem *NS4_implies_NS3 [rule_format]:*
```
"evs ∈ ns_shared ⟹
   Key K ∉ analz (spies evs) ⟶
   Says Server A (Crypt (shrK A) {|NA, Agent B, Key K, X|}) ∈ set evs ⟶
   Crypt K (Nonce NB) ∈ parts (spies evs) ⟶
   (∃A'. Says A' B X ∈ set evs)"
```
⟨*proof*⟩

lemma *B_trusts_NS5_lemma [rule_format]:*
```
"⟦B ∉ bad;  evs ∈ ns_shared⟧ ⟹
   Key K ∉ analz (spies evs) ⟶
   Says Server A
        (Crypt (shrK A) {|NA, Agent B, Key K,
                          Crypt (shrK B) {|Key K, Agent A|}|}) ∈ set evs ⟶
   Crypt K {|Nonce NB, Nonce NB|} ∈ parts (spies evs) ⟶
   Says A B (Crypt K {|Nonce NB, Nonce NB|}) ∈ set evs"
```
⟨*proof*⟩

Very strong Oops condition reveals protocol's weakness

lemma *B_trusts_NS5:*
```
"⟦Crypt K {|Nonce NB, Nonce NB|} ∈ parts (spies evs);
  Crypt (shrK B) {|Key K, Agent A|} ∈ parts (spies evs);
  ∀NA NB. Notes Spy {|NA, NB, Key K|} ∉ set evs;
  A ∉ bad;  B ∉ bad;  evs ∈ ns_shared⟧
 ⟹ Says A B (Crypt K {|Nonce NB, Nonce NB|}) ∈ set evs"
```
⟨*proof*⟩

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 rely on temporal checks: strong guarantees in a little abstracted - but very realistic - model.

syntax
```
    CT :: "event list=>nat"
    Expired :: "[nat, event list] => bool"
    RecentAuth :: "[nat, event list] => bool"
```

consts

```
    SesKeyLife   :: nat

    AutLife :: nat
```

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

specification *(SesKeyLife)*
```
  SesKeyLife_LB [iff]: "2 ≤ SesKeyLife"
```
⟨*proof*⟩

The authenticator only for one journey

specification *(AutLife)*
```
  AutLife_LB [iff]:    "Suc 0 ≤ AutLife"
```
⟨*proof*⟩

translations
```
   "CT" == "length "

   "Expired T evs" == "SesKeyLife + T < CT evs"

   "RecentAuth T evs" == "CT evs ≤ AutLife + T"
```

consts *kerberos_ban   :: "event list set"*
inductive *"kerberos_ban"*
intros

```
   Nil:  "[] ∈ kerberos_ban"

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

   Kb1:  "[| evs1 ∈ kerberos_ban |]
          ==> Says A Server {|Agent A, Agent B|} # evs1
                ∈  kerberos_ban"
```

```
Kb2:  "[| evs2 ∈ kerberos_ban;  Key KAB ∉ used evs2;  KAB ∈ symKeys;
          Says A' Server {|Agent A, Agent B|} ∈ set evs2 |]
       ==> Says Server A
             (Crypt (shrK A)
                {|Number (CT evs2), Agent B, Key KAB,
                  (Crypt (shrK B) {|Number (CT evs2), Agent A, Key KAB|})|})
             # evs2 ∈ kerberos_ban"


Kb3:  "[| evs3 ∈ kerberos_ban;
          Says S A (Crypt (shrK A) {|Number Ts, Agent B, Key K, X|})
            ∈ set evs3;
          Says A Server {|Agent A, Agent B|} ∈ set evs3;
          ~ Expired Ts evs3 |]
       ==> Says A B {|X, Crypt K {|Agent A, Number (CT evs3)|} |}
            # evs3 ∈ kerberos_ban"


Kb4:  "[| evs4 ∈ kerberos_ban;
          Says A' B {|(Crypt (shrK B) {|Number Ts, Agent A, Key K|}),
                       (Crypt K {|Agent A, Number Ta|}) |}: set evs4;
          ~ Expired Ts evs4;  RecentAuth Ta evs4 |]
       ==> Says B A (Crypt K (Number Ta)) # evs4
             ∈ kerberos_ban"


Oops: "[| evso ∈ kerberos_ban;
          Says Server A (Crypt (shrK A) {|Number Ts, Agent B, Key K, X|})
            ∈ set evso;
          Expired Ts evso |]
       ==> Notes Spy {|Number Ts, Key K|} # evso ∈ kerberos_ban"
```

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 ∈ kerberos_ban.
Says B A (Crypt K (Number Timestamp))
∈ set evs"

⟨*proof*⟩

Forwarding Lemma for reasoning about the encrypted portion of message Kb3

lemma *Kb3_msg_in_parts_spies:*
"Says S A (Crypt KA {|Timestamp, B, K, X|}) ∈ set evs
==> X ∈ parts (spies evs)"

⟨*proof*⟩

lemma *Oops_parts_spies:*
"Says Server A (Crypt (shrK A) {|Timestamp, B, K, X|}) ∈ set evs

```
     ==> K ∈ parts (spies evs)"
```
⟨*proof*⟩

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

lemma *Spy_see_shrK [simp]:*
```
    "evs ∈ kerberos_ban ==> (Key (shrK A) ∈ parts (spies evs)) = (A ∈ bad)"
```
⟨*proof*⟩

lemma *Spy_analz_shrK [simp]:*
```
    "evs ∈ kerberos_ban ==> (Key (shrK A) ∈ analz (spies evs)) = (A ∈ bad)"
```
⟨*proof*⟩

lemma *Spy_see_shrK_D [dest!]:*
```
    "[| Key (shrK A) ∈ parts (spies evs);
                evs ∈ kerberos_ban |] ==> A:bad"
```
⟨*proof*⟩

lemmas *Spy_analz_shrK_D = analz_subset_parts [THEN subsetD, THEN Spy_see_shrK_D, dest!]*

Nobody can have used non-existent keys!

lemma *new_keys_not_used [simp]:*
```
   "[|Key K ∉ used evs; K ∈ symKeys; evs ∈ kerberos_ban|]
    ==> K ∉ keysFor (parts (spies evs))"
```
⟨*proof*⟩

4.1 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 Ts, Agent B, Key K, X|})
        ∈ set evs; evs ∈ kerberos_ban |]
     ==> K ∉ range shrK &
         X = (Crypt (shrK B) {|Number Ts, Agent A, Key K|}) &
         K' = shrK A"
```
⟨*proof*⟩

If the encrypted message appears then it originated with the Server PROVIDED that A is NOT compromised!
This shows implicitly the FRESHNESS OF THE SESSION KEY to A

lemma *A_trusts_K_by_Kb2:*
```
    "[| Crypt (shrK A) {|Number Ts, Agent B, Key K, X|}
          ∈ parts (spies evs);
        A ∉ bad;  evs ∈ kerberos_ban |]
      ==> Says Server A (Crypt (shrK A) {|Number Ts, Agent B, Key K, X|})
            ∈ set evs"
```
⟨*proof*⟩

If the TICKET appears then it originated with the Server

FRESHNESS OF THE SESSION KEY to B

lemma *B_trusts_K_by_Kb3:*
```
"[| Crypt (shrK B) {|Number Ts, Agent A, Key K|} ∈ parts (spies evs);
    B ∉ bad;  evs ∈ kerberos_ban |]
  ==> Says Server A
        (Crypt (shrK A) {|Number Ts, Agent B, Key K,
                          Crypt (shrK B) {|Number Ts, Agent A, Key K|}|})
       ∈ set evs"
```
⟨*proof*⟩

EITHER describes the form of X when the following message is sent, OR reduces it to the Fake case. Use *Says_Server_message_form* if applicable.

lemma *Says_S_message_form:*
```
"[| Says S A (Crypt (shrK A) {|Number Ts, Agent B, Key K, X|})
         ∈ set evs;
      evs ∈ kerberos_ban |]
 ==> (K ∉ range shrK & X = (Crypt (shrK B) {|Number Ts, Agent A, Key K|}))
       | X ∈ analz (spies evs)"
```
⟨*proof*⟩

Session keys are not used to encrypt other session keys

lemma *analz_image_freshK [rule_format (no_asm)]:*
```
"evs ∈ kerberos_ban ==>
 ∀K KK. KK ⊆ - (range shrK) -->
        (Key K ∈ analz (Key‘KK Un (spies evs))) =
        (K ∈ KK | Key K ∈ analz (spies evs))"
```
⟨*proof*⟩

lemma *analz_insert_freshK:*
```
"[| evs ∈ kerberos_ban;  KAB ∉ range shrK |] ==>
 (Key K ∈ analz (insert (Key KAB) (spies evs))) =
 (K = KAB | Key K ∈ analz (spies evs))"
```
⟨*proof*⟩

The session key K uniquely identifies the message

lemma *unique_session_keys:*
```
"[| Says Server A
      (Crypt (shrK A) {|Number Ts, Agent B, Key K, X|}) ∈ set evs;
    Says Server A'
      (Crypt (shrK A') {|Number Ts', Agent B', Key K, X'|}) ∈ set evs;
    evs ∈ kerberos_ban |] ==> A=A' & Ts=Ts' & B=B' & X = X'"
```
⟨*proof*⟩

Lemma: the session key sent in msg Kb2 would be EXPIRED if the spy could see it!

lemma *lemma2 [rule_format (no_asm)]:*
```
"[| A ∉ bad;  B ∉ bad;  evs ∈ kerberos_ban |]
 ==> Says Server A
       (Crypt (shrK A) {|Number Ts, Agent B, Key K,
                         Crypt (shrK B) {|Number Ts, Agent A, Key K|}|})
      ∈ set evs -->
     Key K ∈ analz (spies evs) --> Expired Ts evs"
```
⟨*proof*⟩

Confidentiality for the Server: Spy does not see the keys sent in msg Kb2 as long as they have not expired.

lemma *Confidentiality_S:*
```
"[| Says Server A
     (Crypt K' {|Number T, Agent B, Key K, X|}) ∈ set evs;
    ~ Expired T evs;
    A ∉ bad;  B ∉ bad;  evs ∈ kerberos_ban
 |] ==> Key K ∉ analz (spies evs)"
```
⟨*proof*⟩

Confidentiality for Alice

lemma *Confidentiality_A:*
```
"[| Crypt (shrK A) {|Number T, Agent B, Key K, X|} ∈ parts (spies evs);
    ~ Expired T evs;
    A ∉ bad;  B ∉ bad;  evs ∈ kerberos_ban
 |] ==> Key K ∉ analz (spies evs)"
```
⟨*proof*⟩

Confidentiality for Bob

lemma *Confidentiality_B:*
```
"[| Crypt (shrK B) {|Number Tk, Agent A, Key K|}
     ∈ parts (spies evs);
   ~ Expired Tk evs;
  A ∉ bad;  B ∉ bad;  evs ∈ kerberos_ban
 |] ==> Key K ∉ analz (spies evs)"
```
⟨*proof*⟩

lemma *lemma_B [rule_format]:*
```
"[| B ∉ bad;  evs ∈ kerberos_ban |]
 ==> Key K ∉ analz (spies evs) -->
     Says Server A (Crypt (shrK A) {|Number Ts, Agent B, Key K, X|})
     ∈ set evs -->
     Crypt K (Number Ta) ∈ parts (spies evs) -->
     Says B A (Crypt K (Number Ta)) ∈ set evs"
```
⟨*proof*⟩

Authentication of B to A

lemma *Authentication_B:*
```
"[| Crypt K (Number Ta) ∈ parts (spies evs);
    Crypt (shrK A) {|Number Ts, Agent B, Key K, X|}
    ∈ parts (spies evs);
    ~ Expired Ts evs;
    A ∉ bad;  B ∉ bad;  evs ∈ kerberos_ban |]
 ==> Says B A (Crypt K (Number Ta)) ∈ set evs"
```
⟨*proof*⟩

lemma *lemma_A [rule_format]:*
```
"[| A ∉ bad; B ∉ bad; evs ∈ kerberos_ban |]
 ==>
    Key K ∉ analz (spies evs) -->
    Says Server A (Crypt (shrK A) {|Number Ts, Agent B, Key K, X|})
    ∈ set evs -->
```

```
         Crypt K {|Agent A, Number Ta|} ∈ parts (spies evs) -->
        Says A B {|X, Crypt K {|Agent A, Number Ta|}|}
           ∈ set evs"
```

⟨*proof*⟩

Authentication of A to B

lemma *Authentication_A:*

```
     "[| Crypt K {|Agent A, Number Ta|} ∈ parts (spies evs);
         Crypt (shrK B) {|Number Ts, Agent A, Key K|}
         ∈ parts (spies evs);
         ~ Expired Ts evs;
         A ∉ bad;  B ∉ bad;  evs ∈ kerberos_ban |]
      ==> Says A B {|Crypt (shrK B) {|Number Ts, Agent A, Key K|},
                     Crypt K {|Agent A, Number Ta|}|} ∈ set evs"
```

⟨*proof*⟩

end

5 The Kerberos Protocol, Version IV

theory *KerberosIV* **imports** *Public* **begin**

syntax

```
  Kas :: agent
  Tgs :: agent
```

— the two servers are translations...

translations

```
  "Kas"       == "Server "
  "Tgs"       == "Friend 0"
```

axioms

```
  Tgs_not_bad [iff]: "Tgs ∉ bad"
```

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

syntax

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

    ExpirAuth :: "[nat, event list] => bool"

    ExpirServ :: "[nat, event list] => bool"

    ExpirAutc :: "[nat, event list] => bool"

    RecentResp :: "[nat, nat] => bool"
```

constdefs

```
    AuthKeys :: "event list => key set"
```

```
    "AuthKeys evs == {AuthKey. ∃A Peer Tk. Says Kas A
                         (Crypt (shrK A) {|Key AuthKey, Agent Peer, Tk,
                      (Crypt (shrK Peer) {|Agent A, Agent Peer, Key AuthKey, Tk|})
                     |}) ∈ set evs}"


  Issues :: "[agent, agent, msg, event list] => bool"
            ("_ Issues _ with _ on _")
   "A Issues B with X on evs ==
      ∃Y. Says A B Y ∈ set evs & X ∈ parts {Y} &
      X ∉ parts (spies (takeWhile (% z. z  ≠ Says A B Y) (rev evs)))"
```

consts

```
    AuthLife   :: nat


    ServLife   :: nat


    AutcLife   :: nat


    RespLife   :: nat
```

specification *(AuthLife)*

```
  AuthLife_LB [iff]: "2 ≤ AuthLife"
```

⟨*proof*⟩

specification *(ServLife)*

```
  ServLife_LB [iff]: "2 ≤ ServLife"
```

⟨*proof*⟩

specification *(AutcLife)*

```
  AutcLife_LB [iff]: "Suc 0 ≤ AutcLife"
```

⟨*proof*⟩

specification *(RespLife)*

```
  RespLife_LB [iff]: "Suc 0 ≤ RespLife"
```

⟨*proof*⟩

translations

```
   "CT" == "length "

   "ExpirAuth T evs" == "AuthLife + T < CT evs"

   "ExpirServ T evs" == "ServLife + T < CT evs"

   "ExpirAutc T evs" == "AutcLife + T < CT evs"

   "RecentResp T1 T2" == "T1 <= RespLife + T2"
```

```
constdefs
  KeyCryptKey :: "[key, key, event list] => bool"
  "KeyCryptKey AuthKey ServKey evs ==
     ∃A B tt.
       Says Tgs A (Crypt AuthKey
                     {|Key ServKey, Agent B, tt,
                       Crypt (shrK B) {|Agent A, Agent B, Key ServKey, tt|}
|})
         ∈ set evs"

consts

kerberos   :: "event list set"
inductive "kerberos"
  intros

   Nil:  "[] ∈ kerberos"

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


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




   K2:  "[| evs2 ∈ kerberos; Key AuthKey ∉ used evs2; AuthKey ∈ symKeys;
             Says A' Kas {|Agent A, Agent Tgs, Number Ta|} ∈ set evs2 |]
           ==> Says Kas A
                 (Crypt (shrK A) {|Key AuthKey, Agent Tgs, Number (CT evs2),
                       (Crypt (shrK Tgs) {|Agent A, Agent Tgs, Key AuthKey,
                           Number (CT evs2)|})|}) # evs2 ∈ kerberos"




   K3:  "[| evs3 ∈ kerberos;
             Says A Kas {|Agent A, Agent Tgs, Number Ta|} ∈ set evs3;
             Says Kas' A (Crypt (shrK A) {|Key AuthKey, Agent Tgs, Number Tk,
               AuthTicket|}) ∈ set evs3;
             RecentResp Tk Ta
          |]
           ==> Says A Tgs {|AuthTicket,
                            (Crypt AuthKey {|Agent A, Number (CT evs3)|}),
                            Agent B|} # evs3 ∈ kerberos"
```

```
K4:  "[| evs4 ∈ kerberos; Key ServKey ∉ used evs4; ServKey ∈ symKeys;
         B ≠ Tgs;  AuthKey ∈ symKeys;
         Says A' Tgs {|
          (Crypt (shrK Tgs) {|Agent A, Agent Tgs, Key AuthKey,
                              Number Tk|}),
          (Crypt AuthKey {|Agent A, Number Ta1|}), Agent B|}
             ∈ set evs4;
         ~ ExpirAuth Tk evs4;
         ~ ExpirAutc Ta1 evs4;
         ServLife + (CT evs4) <= AuthLife + Tk
       |]
        ==> Says Tgs A
              (Crypt AuthKey {|Key ServKey, Agent B, Number (CT evs4),
                              Crypt (shrK B) {|Agent A, Agent B, Key ServKey,
                                               Number (CT evs4)|} |})
              # evs4 ∈ kerberos"
```

```
K5:  "[| evs5 ∈ kerberos; AuthKey ∈ symKeys; ServKey ∈ symKeys;
         Says A Tgs
             {|AuthTicket, Crypt AuthKey {|Agent A, Number Ta1|},
               Agent B|}
           ∈ set evs5;
         Says Tgs' A
          (Crypt AuthKey {|Key ServKey, Agent B, Number Tt, ServTicket|})
             ∈ set evs5;
         RecentResp Tt Ta1 |]
        ==> Says A B {|ServTicket,
                       Crypt ServKey {|Agent A, Number (CT evs5)|} |}
             # evs5 ∈ kerberos"
```

```
K6:  "[| evs6 ∈ kerberos;
         Says A' B {|
           (Crypt (shrK B) {|Agent A, Agent B, Key ServKey, Number Tt|}),
           (Crypt ServKey {|Agent A, Number Ta2|})|}
         ∈ set evs6;
         ~ ExpirServ Tt evs6;
         ~ ExpirAutc Ta2 evs6
       |]
        ==> Says B A (Crypt ServKey (Number Ta2))
             # evs6 ∈ kerberos"
```

```
Oops1: "[| evsO1 ∈ kerberos;  A ≠ Spy;
           Says Kas A
             (Crypt (shrK A) {|Key AuthKey, Agent Tgs, Number Tk,
                               AuthTicket|})  ∈ set evsO1;
           ExpirAuth Tk evsO1 |]
        ==> Says A Spy {|Agent A, Agent Tgs, Number Tk, Key AuthKey|}
             # evsO1 ∈ kerberos"
```

```
Oops2: "[| evsO2 ∈ kerberos;  A ≠ Spy;
           Says Tgs A
             (Crypt AuthKey {|Key ServKey, Agent B, Number Tt, ServTicket|})
                ∈ set evsO2;
           ExpirServ Tt evsO2 |]
        ==> Says A Spy {|Agent A, Agent B, Number Tt, Key ServKey|}
             # evsO2 ∈ kerberos"
```

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]*

5.1 Lemmas about Lists

lemma *spies_Says_rev: "spies (evs @ [Says A B X]) = insert X (spies evs)"*
⟨*proof*⟩

lemma *spies_Gets_rev: "spies (evs @ [Gets A X]) = spies evs"*
⟨*proof*⟩

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

lemma *spies_evs_rev: "spies evs = spies (rev evs)"*
⟨*proof*⟩

lemmas *parts_spies_evs_revD2 = spies_evs_rev [THEN equalityD2, THEN parts_mono]*

lemma *spies_takeWhile: "spies (takeWhile P evs) <= spies evs"*
⟨*proof*⟩

lemmas *parts_spies_takeWhile_mono = spies_takeWhile [THEN parts_mono]*

5.2 Lemmas about AuthKeys

lemma *AuthKeys_empty: "AuthKeys [] = {}"*
⟨*proof*⟩

lemma *AuthKeys_not_insert:*
"(∀A Tk akey Peer.
ev ≠ Says Kas A (Crypt (shrK A) {|akey, Agent Peer, Tk,
(Crypt (shrK Peer) {|Agent A, Agent Peer, akey, Tk|})|}))
==> AuthKeys (ev # evs) = AuthKeys evs"
⟨*proof*⟩

lemma *AuthKeys_insert:*
"AuthKeys
(Says Kas A (Crypt (shrK A) {|Key K, Agent Peer, Number Tk,
(Crypt (shrK Peer) {|Agent A, Agent Peer, Key K, Number Tk|})|}) # evs)
= insert K (AuthKeys evs)"
⟨*proof*⟩

lemma *AuthKeys_simp:*
"K ∈ AuthKeys
(Says Kas A (Crypt (shrK A) {|Key K', Agent Peer, Number Tk,
(Crypt (shrK Peer) {|Agent A, Agent Peer, Key K', Number Tk|})|}) # evs)
==> K = K' | K ∈ AuthKeys evs"
⟨*proof*⟩

lemma *AuthKeysI:*
"Says Kas A (Crypt (shrK A) {|Key K, Agent Tgs, Number Tk,
(Crypt (shrK Tgs) {|Agent A, Agent Tgs, Key K, Number Tk|})|}) ∈ set evs
==> K ∈ AuthKeys evs"
⟨*proof*⟩

lemma *AuthKeys_used: "K ∈ AuthKeys evs ==> Key K ∈ used evs"*
⟨*proof*⟩

5.3 Forwarding Lemmas

–For reasoning about the encrypted portion of message K3–

lemma *K3_msg_in_parts_spies:*
"Says Kas' A (Crypt KeyA {|AuthKey, Peer, Tk, AuthTicket|})
∈ set evs ==> AuthTicket ∈ parts (spies evs)"
⟨*proof*⟩

lemma *Oops_range_spies1:*
"[| Says Kas A (Crypt KeyA {|Key AuthKey, Peer, Tk, AuthTicket|})
∈ set evs ;
evs ∈ kerberos |] ==> AuthKey ∉ range shrK & AuthKey ∈ symKeys"
⟨*proof*⟩

–For reasoning about the encrypted portion of message K5–

lemma *K5_msg_in_parts_spies:*
"Says Tgs' A (Crypt AuthKey {|ServKey, Agent B, Tt, ServTicket|})
∈ set evs ==> ServTicket ∈ parts (spies evs)"

⟨*proof*⟩

lemma *Oops_range_spies2:*
```
"[| Says Tgs A (Crypt AuthKey {|Key ServKey, Agent B, Tt, ServTicket|})
      ∈ set evs ;
    evs ∈ kerberos |] ==> ServKey ∉ range shrK & ServKey ∈ symKeys"
```
⟨*proof*⟩

lemma *Says_ticket_in_parts_spies:*
```
"Says S A (Crypt K {|SesKey, B, TimeStamp, Ticket|}) ∈ set evs
 ==> Ticket ∈ parts (spies evs)"
```
⟨*proof*⟩

lemma *Spy_see_shrK [simp]:*
```
"evs ∈ kerberos ==> (Key (shrK A) ∈ parts (spies evs)) = (A ∈ bad)"
```
⟨*proof*⟩

lemma *Spy_analz_shrK [simp]:*
```
"evs ∈ kerberos ==> (Key (shrK A) ∈ analz (spies evs)) = (A ∈ bad)"
```
⟨*proof*⟩

lemma *Spy_see_shrK_D [dest!]:*
```
"[| Key (shrK A) ∈ parts (spies evs);  evs ∈ kerberos |] ==> A:bad"
```
⟨*proof*⟩
lemmas *Spy_analz_shrK_D = analz_subset_parts [THEN subsetD, THEN Spy_see_shrK_D, dest!]*

Nobody can have used non-existent keys!

lemma *new_keys_not_used [simp]:*
```
"[|Key K ∉ used evs; K ∈ symKeys; evs ∈ kerberos|]
 ==> K ∉ keysFor (parts (spies evs))"
```
⟨*proof*⟩

lemma *new_keys_not_analzd:*
```
"[|evs ∈ kerberos; K ∈ symKeys; Key K ∉ used evs|]
 ==> K ∉ keysFor (analz (spies evs))"
```
⟨*proof*⟩

5.4 Regularity Lemmas

These concern the form of items passed in messages

Describes the form of AuthKey, AuthTicket, and K sent by Kas

lemma *Says_Kas_message_form:*
```
"[| Says Kas A (Crypt K {|Key AuthKey, Agent Peer, Tk, AuthTicket|})
      ∈ set evs;
    evs ∈ kerberos |]
 ==> AuthKey ∉ range shrK & AuthKey ∈ AuthKeys evs & AuthKey ∈ symKeys
&
  AuthTicket = (Crypt (shrK Tgs) {|Agent A, Agent Tgs, Key AuthKey, Tk|}) &
```

```
            K = shrK A  & Peer = Tgs"
```
⟨*proof*⟩

lemma *SesKey_is_session_key:*
```
     "[| Crypt (shrK Tgs_B) {|Agent A, Agent Tgs_B, Key SesKey, Number T|}
           ∈ parts (spies evs); Tgs_B ∉ bad;
         evs ∈ kerberos |]
      ==> SesKey ∉ range shrK"
```
⟨*proof*⟩

lemma *A_trusts_AuthTicket:*
```
     "[| Crypt (shrK Tgs) {|Agent A, Agent Tgs, Key AuthKey, Tk|}
           ∈ parts (spies evs);
         evs ∈ kerberos |]
      ==> Says Kas A (Crypt (shrK A) {|Key AuthKey, Agent Tgs, Tk,
                Crypt (shrK Tgs) {|Agent A, Agent Tgs, Key AuthKey, Tk|}|})
           ∈ set evs"
```
⟨*proof*⟩

lemma *AuthTicket_crypt_AuthKey:*
```
     "[| Crypt (shrK Tgs) {|Agent A, Agent Tgs, Key AuthKey, Number Tk|}
           ∈ parts (spies evs);
         evs ∈ kerberos |]
      ==> AuthKey ∈ AuthKeys evs"
```
⟨*proof*⟩

Describes the form of ServKey, ServTicket and AuthKey sent by Tgs

lemma *Says_Tgs_message_form:*
```
     "[| Says Tgs A (Crypt AuthKey {|Key ServKey, Agent B, Tt, ServTicket|})
           ∈ set evs;
         evs ∈ kerberos |]
   ==> B ≠ Tgs &
       ServKey ∉ range shrK & ServKey ∉ AuthKeys evs & ServKey ∈ symKeys
&
       ServTicket = (Crypt (shrK B) {|Agent A, Agent B, Key ServKey, Tt|})
&
       AuthKey ∉ range shrK & AuthKey ∈ AuthKeys evs & AuthKey ∈ symKeys"
```
⟨*proof*⟩

Authenticity of AuthKey for A: If a certain encrypted message appears then it originated with Kas

lemma *A_trusts_AuthKey:*
```
     "[| Crypt (shrK A) {|Key AuthKey, Peer, Tk, AuthTicket|}
           ∈ parts (spies evs);
         A ∉ bad;  evs ∈ kerberos |]
      ==> Says Kas A (Crypt (shrK A) {|Key AuthKey, Peer, Tk, AuthTicket|})
           ∈ set evs"
```
⟨*proof*⟩

If a certain encrypted message appears then it originated with Tgs

lemma *A_trusts_K4:*
```
     "[| Crypt AuthKey {|Key ServKey, Agent B, Tt, ServTicket|}
```

```
          ∈ parts (spies evs);
         Key AuthKey ∉ analz (spies evs);
         AuthKey ∉ range shrK;
         evs ∈ kerberos |]
 ==> ∃A. Says Tgs A (Crypt AuthKey {|Key ServKey, Agent B, Tt, ServTicket|})
       ∈ set evs"
```
⟨*proof*⟩

lemma *AuthTicket_form:*
```
     "[| Crypt (shrK A) {|Key AuthKey, Agent Tgs, Tk, AuthTicket|}
           ∈ parts (spies evs);
         A ∉ bad;
         evs ∈ kerberos |]
    ==> AuthKey ∉ range shrK & AuthKey ∈ symKeys &
        AuthTicket = Crypt (shrK Tgs) {|Agent A, Agent Tgs, Key AuthKey, Tk|}"
```
⟨*proof*⟩

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

lemma *ServTicket_form:*
```
     "[| Crypt AuthKey {|Key ServKey, Agent B, Tt, ServTicket|}
              ∈ parts (spies evs);
            Key AuthKey ∉ analz (spies evs);
            evs ∈ kerberos |]
         ==> ServKey ∉ range shrK & ServKey ∈ symKeys &
    (∃A. ServTicket = Crypt (shrK B) {|Agent A, Agent B, Key ServKey, Tt|})"
```
⟨*proof*⟩

Essentially the same as *AuthTicket_form*

lemma *Says_kas_message_form:*
```
     "[| Says Kas' A (Crypt (shrK A)
              {|Key AuthKey, Agent Tgs, Tk, AuthTicket|}) ∈ set evs;
         evs ∈ kerberos |]
      ==> AuthKey ∉ range shrK & AuthKey ∈ symKeys &
          AuthTicket =
                  Crypt (shrK Tgs) {|Agent A, Agent Tgs, Key AuthKey, Tk|}
          | AuthTicket ∈ analz (spies evs)"
```
⟨*proof*⟩

lemma *Says_tgs_message_form:*
```
 "[| Says Tgs' A (Crypt AuthKey {|Key ServKey, Agent B, Tt, ServTicket|})
       ∈ set evs;  AuthKey ∈ symKeys;
     evs ∈ kerberos |]
  ==> ServKey ∉ range shrK &
      (∃A. ServTicket =
              Crypt (shrK B) {|Agent A, Agent B, Key ServKey, Tt|})
       | ServTicket ∈ analz (spies evs)"
```
⟨*proof*⟩

5.5 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_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 ∈ kerberos |]
 ==> A=A' & B=B' & T=T'"
```
⟨*proof*⟩

An AuthKey is encrypted by one and only one Shared key. A ServKey is encrypted by one and only one AuthKey.

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 ∈ kerberos |]
 ==> K=K' & B=B' & T=T' & Ticket=Ticket'"
```
⟨*proof*⟩

lemma *unique_AuthKeys:*
```
"[| Says Kas A
          (Crypt Ka {|Key AuthKey, Agent Tgs, Tk, X|}) ∈ set evs;
    Says Kas A'
          (Crypt Ka' {|Key AuthKey, Agent Tgs, Tk', X'|}) ∈ set evs;
    evs ∈ kerberos |] ==> A=A' & Ka=Ka' & Tk=Tk' & X=X'"
```
⟨*proof*⟩

ServKey uniquely identifies the message from Tgs

lemma *unique_ServKeys:*
```
"[| Says Tgs A
          (Crypt K {|Key ServKey, Agent B, Tt, X|}) ∈ set evs;
    Says Tgs A'
          (Crypt K' {|Key ServKey, Agent B', Tt', X'|}) ∈ set evs;
    evs ∈ kerberos |] ==> A=A' & B=B' & K=K' & Tt=Tt' & X=X'"
```
⟨*proof*⟩

5.6 Lemmas About the Predicate *KeyCryptKey*

lemma *not_KeyCryptKey_Nil [iff]: "~ KeyCryptKey AuthKey ServKey []"*
⟨*proof*⟩

lemma *KeyCryptKeyI:*
```
 "[| Says Tgs A (Crypt AuthKey {|Key ServKey, Agent B, tt, X |}) ∈ set evs;
     evs ∈ kerberos |] ==> KeyCryptKey AuthKey ServKey evs"
```
⟨*proof*⟩

lemma *KeyCryptKey_Says [simp]:*
```
   "KeyCryptKey AuthKey ServKey (Says S A X # evs) =
     (Tgs = S &
      (∃B tt. X = Crypt AuthKey
```

```
                {|Key ServKey, Agent B, tt,
                  Crypt (shrK B) {|Agent A, Agent B, Key ServKey, tt|} |})
     | KeyCryptKey AuthKey ServKey evs)"
```
⟨*proof*⟩

lemma *Auth_fresh_not_KeyCryptKey:*
```
     "[| Key AuthKey ∉ used evs; evs ∈ kerberos |]
      ==> ~ KeyCryptKey AuthKey ServKey evs"
```
⟨*proof*⟩

lemma *Serv_fresh_not_KeyCryptKey:*
```
 "Key ServKey ∉ used evs ==> ~ KeyCryptKey AuthKey ServKey evs"
```
⟨*proof*⟩

lemma *AuthKey_not_KeyCryptKey:*
```
     "[| Crypt (shrK Tgs) {|Agent A, Agent Tgs, Key AuthKey, tk|}
           ∈ parts (spies evs);  evs ∈ kerberos |]
      ==> ~ KeyCryptKey K AuthKey evs"
```
⟨*proof*⟩

A secure serverkey cannot have been used to encrypt others

lemma *ServKey_not_KeyCryptKey:*
```
 "[| Crypt (shrK B) {|Agent A, Agent B, Key SK, tt|} ∈ parts (spies evs);
     Key SK ∉ analz (spies evs);  SK ∈ symKeys;
     B ≠ Tgs;  evs ∈ kerberos |]
  ==> ~ KeyCryptKey SK K evs"
```
⟨*proof*⟩

Long term keys are not issued as ServKeys

lemma *shrK_not_KeyCryptKey:*
```
     "evs ∈ kerberos ==> ~ KeyCryptKey K (shrK A) evs"
```
⟨*proof*⟩

The Tgs message associates ServKey with AuthKey and therefore not with any other key AuthKey.

lemma *Says_Tgs_KeyCryptKey:*
```
     "[| Says Tgs A (Crypt AuthKey {|Key ServKey, Agent B, tt, X |})
           ∈ set evs;
        AuthKey' ≠ AuthKey;  evs ∈ kerberos |]
      ==> ~ KeyCryptKey AuthKey' ServKey evs"
```
⟨*proof*⟩

lemma *KeyCryptKey_not_KeyCryptKey:*
```
     "[| KeyCryptKey AuthKey ServKey evs;  evs ∈ kerberos |]
      ==> ~ KeyCryptKey ServKey K evs"
```
⟨*proof*⟩

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

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

lemma *Key_analz_image_Key_lemma:*
```
"P --> (Key K ∈ analz (Key‘KK Un H)) --> (K:KK | Key K ∈ analz H)
 ==>
 P --> (Key K ∈ analz (Key‘KK Un H)) = (K:KK | Key K ∈ analz H)"
```
⟨*proof*⟩

lemma *KeyCryptKey_analz_insert:*
```
"[| KeyCryptKey K K' evs; K ∈ symKeys; evs ∈ kerberos |]
 ==> Key K' ∈ analz (insert (Key K) (spies evs))"
```
⟨*proof*⟩

lemma *AuthKeys_are_not_KeyCryptKey:*
```
"[| K ∈ AuthKeys evs Un range shrK;  evs ∈ kerberos |]
 ==> ∀SK. ~ KeyCryptKey SK K evs"
```
⟨*proof*⟩

lemma *not_AuthKeys_not_KeyCryptKey:*
```
"[| K ∉ AuthKeys evs;
    K ∉ range shrK; evs ∈ kerberos |]
 ==> ∀SK. ~ KeyCryptKey K SK evs"
```
⟨*proof*⟩

5.7 Secrecy Theorems

For the Oops2 case of the next theorem

lemma *Oops2_not_KeyCryptKey:*
```
"[| evs ∈ kerberos;
    Says Tgs A (Crypt AuthKey
                {|Key ServKey, Agent B, Number Tt, ServTicket|})
      ∈ set evs |]
 ==> ~ KeyCryptKey ServKey SK evs"
```
⟨*proof*⟩

Big simplification law for keys SK that are not crypted by keys in KK It helps prove three, otherwise hard, facts about keys. These facts are exploited as simplification laws for analz, and also "limit the damage" in case of loss of a key to the spy. See ESORICS98. [simplified by LCP]

lemma *Key_analz_image_Key [rule_format (no_asm)]:*
```
"evs ∈ kerberos ==>
 (∀SK KK. SK ∈ symKeys & KK <= -(range shrK) -->
  (∀K ∈ KK. ~ KeyCryptKey K SK evs)   -->
  (Key SK ∈ analz (Key‘KK Un (spies evs))) =
  (SK ∈ KK | Key SK ∈ analz (spies evs)))"
```
⟨*proof*⟩

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

lemma *analz_insert_freshK1:*
```
"[| evs ∈ kerberos;  K ∈ (AuthKeys evs) Un range shrK;
    K ∈ symKeys;
    SesKey ∉ range shrK |]
```

```
      ==> (Key K ∈ analz (insert (Key SesKey) (spies evs))) =
          (K = SesKey | Key K ∈ analz (spies evs))"
```
⟨*proof*⟩

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

lemma *analz_insert_freshK2:*
```
     "[| evs ∈ kerberos;  ServKey ∉ (AuthKeys evs); ServKey ∉ range shrK;
         K ∈ symKeys|]
      ==> (Key K ∈ analz (insert (Key ServKey) (spies evs))) =
          (K = ServKey | Key K ∈ analz (spies evs))"
```
⟨*proof*⟩

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

lemma *analz_insert_freshK3:*
```
 "[| Says Tgs A
             (Crypt AuthKey {|Key ServKey, Agent B, Number Tt, ServTicket|})
        ∈ set evs;  ServKey ∈ symKeys;
     AuthKey ≠ AuthKey'; AuthKey' ∉ range shrK; evs ∈ kerberos |]
        ==> (Key ServKey ∈ analz (insert (Key AuthKey') (spies evs))) =
                (ServKey = AuthKey' | Key ServKey ∈ analz (spies evs))"
```
⟨*proof*⟩

a weakness of the protocol

lemma *AuthKey_compromises_ServKey:*
```
     "[| Says Tgs A
              (Crypt AuthKey {|Key ServKey, Agent B, Number Tt, ServTicket|})
           ∈ set evs;  AuthKey ∈ symKeys;
         Key AuthKey ∈ analz (spies evs); evs ∈ kerberos |]
      ==> Key ServKey ∈ analz (spies evs)"
```
⟨*proof*⟩

5.8 Guarantees for Kas

lemma *ServKey_notin_AuthKeysD:*
```
     "[| Crypt AuthKey {|Key ServKey, Agent B, Tt,
                         Crypt (shrK B) {|Agent A, Agent B, Key ServKey, Tt|}|}
           ∈ parts (spies evs);
         Key ServKey ∉ analz (spies evs);
         B ≠ Tgs; evs ∈ kerberos |]
      ==> ServKey ∉ AuthKeys evs"
```
⟨*proof*⟩

If Spy sees the Authentication Key sent in msg K2, then the Key has expired.

lemma *Confidentiality_Kas_lemma [rule_format]:*
```
     "[| AuthKey ∈ symKeys; A ∉ bad;  evs ∈ kerberos |]
      ==> Says Kas A
               (Crypt (shrK A)
                  {|Key AuthKey, Agent Tgs, Number Tk,
           Crypt (shrK Tgs) {|Agent A, Agent Tgs, Key AuthKey, Number Tk|}|})
             ∈ set evs -->
           Key AuthKey ∈ analz (spies evs) -->
           ExpirAuth Tk evs"
```

⟨*proof*⟩

lemma *Confidentiality_Kas:*
```
"[| Says Kas A
          (Crypt Ka {|Key AuthKey, Agent Tgs, Number Tk, AuthTicket|})
       ∈ set evs;
     ~ ExpirAuth Tk evs;
     A ∉ bad;  evs ∈ kerberos |]
  ==> Key AuthKey ∉ analz (spies evs)"
```
⟨*proof*⟩

5.9 Guarantees for Tgs

If Spy sees the Service Key sent in msg K4, then the Key has expired.

lemma *Confidentiality_lemma [rule_format]:*
```
"[| Says Tgs A
        (Crypt AuthKey
           {|Key ServKey, Agent B, Number Tt,
             Crypt (shrK B) {|Agent A, Agent B, Key ServKey, Number Tt|}|})
       ∈ set evs;
    Key AuthKey ∉ analz (spies evs);
    ServKey ∈ symKeys;
    A ∉ bad;  B ∉ bad; B ≠ Tgs; evs ∈ kerberos |]
  ==> Key ServKey ∈ analz (spies evs) -->
      ExpirServ Tt evs"
```
⟨*proof*⟩

In the real world Tgs can't check wheter AuthKey is secure!

lemma *Confidentiality_Tgs1:*
```
"[| Says Tgs A
          (Crypt AuthKey {|Key ServKey, Agent B, Number Tt, ServTicket|})
       ∈ set evs;
     Key AuthKey ∉ analz (spies evs);
     ~ ExpirServ Tt evs;
     A ∉ bad;  B ∉ bad; B ≠ Tgs; evs ∈ kerberos |]
  ==> Key ServKey ∉ analz (spies evs)"
```
⟨*proof*⟩

In the real world Tgs CAN check what Kas sends!

lemma *Confidentiality_Tgs2:*
```
"[| Says Kas A
           (Crypt Ka {|Key AuthKey, Agent Tgs, Number Tk, AuthTicket|})
       ∈ set evs;
     Says Tgs A
          (Crypt AuthKey {|Key ServKey, Agent B, Number Tt, ServTicket|})
       ∈ set evs;
     ~ ExpirAuth Tk evs; ~ ExpirServ Tt evs;
     A ∉ bad;  B ∉ bad; B ≠ Tgs; evs ∈ kerberos |]
  ==> Key ServKey ∉ analz (spies evs)"
```
⟨*proof*⟩

Most general form

lemmas *Confidentiality_Tgs3 = A_trusts_AuthTicket [THEN Confidentiality_Tgs2]*

5.10 Guarantees for Alice

lemmas *Confidentiality_Auth_A = A_trusts_AuthKey [THEN Confidentiality_Kas]*

lemma *A_trusts_K4_bis:*
```
 "[| Says Kas A
       (Crypt (shrK A) {|Key AuthKey, Agent Tgs, Tk, AuthTicket|}) ∈ set evs;
     Crypt AuthKey {|Key ServKey, Agent B, Tt, ServTicket|}
       ∈ parts (spies evs);
     Key AuthKey ∉ analz (spies evs);
     evs ∈ kerberos |]
 ==> Says Tgs A (Crypt AuthKey {|Key ServKey, Agent B, Tt, ServTicket|})
       ∈ set evs"
```
⟨*proof*⟩

lemma *Confidentiality_Serv_A:*
```
     "[| Crypt (shrK A) {|Key AuthKey, Agent Tgs, Number Tk, AuthTicket|}
           ∈ parts (spies evs);
         Crypt AuthKey {|Key ServKey, Agent B, Number Tt, ServTicket|}
           ∈ parts (spies evs);
         ~ ExpirAuth Tk evs; ~ ExpirServ Tt evs;
         A ∉ bad;  B ∉ bad; B ≠ Tgs; evs ∈ kerberos |]
      ==> Key ServKey ∉ analz (spies evs)"
```
⟨*proof*⟩

5.11 Guarantees for Bob

Theorems for the refined model have suffix "refined"

lemma *K4_imp_K2:*
```
"[| Says Tgs A (Crypt AuthKey {|Key ServKey, Agent B, Number Tt, ServTicket|})
      ∈ set evs;  evs ∈ kerberos|]
   ==> ∃Tk. Says Kas A
        (Crypt (shrK A)
         {|Key AuthKey, Agent Tgs, Number Tk,
           Crypt (shrK Tgs) {|Agent A, Agent Tgs, Key AuthKey, Number Tk|}|})
        ∈ set evs"
```
⟨*proof*⟩

lemma *K4_imp_K2_refined:*
```
"[| Says Tgs A (Crypt AuthKey {|Key ServKey, Agent B, Number Tt, ServTicket|})
      ∈ set evs; evs ∈ kerberos|]
   ==> ∃Tk. (Says Kas A (Crypt (shrK A) {|Key AuthKey, Agent Tgs, Number Tk,
           Crypt (shrK Tgs) {|Agent A, Agent Tgs, Key AuthKey, Number Tk|}|})
             ∈ set evs
         & ServLife + Tt <= AuthLife + Tk)"
```
⟨*proof*⟩

Authenticity of ServKey for B

lemma *B_trusts_ServKey:*
```
     "[| Crypt (shrK B) {|Agent A, Agent B, Key ServKey, Tt|}
           ∈ parts (spies evs);  B ≠ Tgs;  B ∉ bad;
         evs ∈ kerberos |]
 ==> ∃AuthKey.
```

```
       Says Tgs A (Crypt AuthKey {|Key ServKey, Agent B, Tt,
                   Crypt (shrK B) {|Agent A, Agent B, Key ServKey, Tt|}|})
       ∈ set evs"
```
⟨*proof*⟩

lemma *B_trusts_ServTicket_Kas:*
```
     "[| Crypt (shrK B) {|Agent A, Agent B, Key ServKey, Number Tt|}
           ∈ parts (spies evs);  B ≠ Tgs;  B ∉ bad;
         evs ∈ kerberos |]
  ==> ∃AuthKey Tk.
       Says Kas A
         (Crypt (shrK A) {|Key AuthKey, Agent Tgs, Number Tk,
            Crypt (shrK Tgs) {|Agent A, Agent Tgs, Key AuthKey, Number Tk|}|})
        ∈ set evs"
```
⟨*proof*⟩

lemma *B_trusts_ServTicket_Kas_refined:*
```
     "[| Crypt (shrK B) {|Agent A, Agent B, Key ServKey, Number Tt|}
           ∈ parts (spies evs);  B ≠ Tgs;  B ∉ bad;
         evs ∈ kerberos |]
  ==> ∃AuthKey Tk. Says Kas A (Crypt(shrK A) {|Key AuthKey, Agent Tgs, Number
Tk,
           Crypt (shrK Tgs) {|Agent A, Agent Tgs, Key AuthKey, Number Tk|}|})
             ∈ set evs
           & ServLife + Tt <= AuthLife + Tk"
```
⟨*proof*⟩

lemma *B_trusts_ServTicket:*
```
     "[| Crypt (shrK B) {|Agent A, Agent B, Key ServKey, Number Tt|}
           ∈ parts (spies evs);  B ≠ Tgs;  B ∉ bad;
         evs ∈ kerberos |]
 ==> ∃Tk AuthKey.
     Says Kas A (Crypt (shrK A) {|Key AuthKey, Agent Tgs, Number Tk,
                   Crypt (shrK Tgs) {|Agent A, Agent Tgs, Key AuthKey, Number
Tk|}|})
       ∈ set evs
     & Says Tgs A (Crypt AuthKey {|Key ServKey, Agent B, Number Tt,
                   Crypt (shrK B) {|Agent A, Agent B, Key ServKey, Number Tt|}|})
       ∈ set evs"
```
⟨*proof*⟩

lemma *B_trusts_ServTicket_refined:*
```
     "[| Crypt (shrK B) {|Agent A, Agent B, Key ServKey, Number Tt|}
           ∈ parts (spies evs);  B ≠ Tgs;  B ∉ bad;
         evs ∈ kerberos |]
 ==> ∃Tk AuthKey.
     (Says Kas A (Crypt (shrK A) {|Key AuthKey, Agent Tgs, Number Tk,
                   Crypt (shrK Tgs) {|Agent A, Agent Tgs, Key AuthKey, Number
Tk|}|})
       ∈ set evs
     & Says Tgs A (Crypt AuthKey {|Key ServKey, Agent B, Number Tt,
                   Crypt (shrK B) {|Agent A, Agent B, Key ServKey, Number Tt|}|})
       ∈ set evs
     & ServLife + Tt <= AuthLife + Tk)"
```

⟨*proof*⟩

lemma *NotExpirServ_NotExpirAuth_refined:*
```
"[| ~ ExpirServ Tt evs; ServLife + Tt <= AuthLife + Tk |]
 ==> ~ ExpirAuth Tk evs"
```
⟨*proof*⟩

lemma *Confidentiality_B:*
```
"[| Crypt (shrK B) {|Agent A, Agent B, Key ServKey, Number Tt|}
      ∈ parts (spies evs);
    Crypt AuthKey {|Key ServKey, Agent B, Number Tt, ServTicket|}
      ∈ parts (spies evs);
    Crypt (shrK A) {|Key AuthKey, Agent Tgs, Number Tk, AuthTicket|}
      ∈ parts (spies evs);
    ~ ExpirServ Tt evs; ~ ExpirAuth Tk evs;
    A ∉ bad;  B ∉ bad; B ≠ Tgs; evs ∈ kerberos |]
 ==> Key ServKey ∉ analz (spies evs)"
```
⟨*proof*⟩

Most general form – only for refined model!

lemma *Confidentiality_B_refined:*
```
"[| Crypt (shrK B) {|Agent A, Agent B, Key ServKey, Number Tt|}
      ∈ parts (spies evs);
    ~ ExpirServ Tt evs;
    A ∉ bad;  B ∉ bad; B ≠ Tgs; evs ∈ kerberos |]
 ==> Key ServKey ∉ analz (spies evs)"
```
⟨*proof*⟩

5.12 Authenticity theorems

1. Session Keys authenticity: they originated with servers.

Authenticity of ServKey for A

lemma *A_trusts_ServKey:*
```
"[| Crypt (shrK A) {|Key AuthKey, Agent Tgs, Number Tk, AuthTicket|}
      ∈ parts (spies evs);
    Crypt AuthKey {|Key ServKey, Agent B, Number Tt, ServTicket|}
      ∈ parts (spies evs);
    ~ ExpirAuth Tk evs; A ∉ bad; evs ∈ kerberos |]
==>Says Tgs A (Crypt AuthKey {|Key ServKey, Agent B, Number Tt, ServTicket|})
     ∈ set evs"
```
⟨*proof*⟩

Note: requires a temporal check

B checks authenticity of A by theorems *A_Authenticity* and *A_authenticity_refined*

lemma *Says_Auth:*
```
"[| Crypt ServKey {|Agent A, Number Ta|} ∈ parts (spies evs);
    Says Tgs A (Crypt AuthKey {|Key ServKey, Agent B, Number Tt,
                                ServTicket|}) ∈ set evs;
    Key ServKey ∉ analz (spies evs);
```

```
        A ∉ bad; B ∉ bad; evs ∈ kerberos |]
 ==> Says A B {|ServTicket, Crypt ServKey {|Agent A, Number Ta|}|} ∈ set evs"
```
⟨*proof*⟩

The second assumption tells B what kind of key ServKey is.

lemma *A_Authenticity:*
```
     "[| Crypt ServKey {|Agent A, Number Ta|} ∈ parts (spies evs);
         Crypt (shrK B) {|Agent A, Agent B, Key ServKey, Number Tt|}
           ∈ parts (spies evs);
         Crypt AuthKey {|Key ServKey, Agent B, Number Tt, ServTicket|}
           ∈ parts (spies evs);
         Crypt (shrK A) {|Key AuthKey, Agent Tgs, Number Tk, AuthTicket|}
           ∈ parts (spies evs);
         ~ ExpirServ Tt evs; ~ ExpirAuth Tk evs;
         B ≠ Tgs; A ∉ bad;  B ∉ bad;  evs ∈ kerberos |]
   ==> Says A B {|Crypt (shrK B) {|Agent A, Agent B, Key ServKey, Number Tt|},
                  Crypt ServKey {|Agent A, Number Ta|} |} ∈ set evs"
```
⟨*proof*⟩

Stronger form in the refined model

lemma *A_Authenticity_refined:*
```
     "[| Crypt ServKey {|Agent A, Number Ta2|} ∈ parts (spies evs);
         Crypt (shrK B) {|Agent A, Agent B, Key ServKey, Number Tt|}
           ∈ parts (spies evs);
         ~ ExpirServ Tt evs;
         B ≠ Tgs; A ∉ bad;  B ∉ bad;  evs ∈ kerberos |]
   ==> Says A B {|Crypt (shrK B) {|Agent A, Agent B, Key ServKey, Number Tt|},
                  Crypt ServKey {|Agent A, Number Ta2|} |} ∈ set evs"
```
⟨*proof*⟩

A checks authenticity of B by theorem *B_authenticity*

lemma *Says_K6:*
```
     "[| Crypt ServKey (Number Ta) ∈ parts (spies evs);
         Says Tgs A (Crypt AuthKey {|Key ServKey, Agent B, Number Tt,
                                     ServTicket|}) ∈ set evs;
         Key ServKey ∉ analz (spies evs);
         A ∉ bad; B ∉ bad; evs ∈ kerberos |]
      ==> Says B A (Crypt ServKey (Number Ta)) ∈ set evs"
```
⟨*proof*⟩

lemma *K4_trustworthy:*
```
     "[| Crypt AuthKey {|Key ServKey, Agent B, T, ServTicket|}
           ∈ parts (spies evs);
         Key AuthKey ∉ analz (spies evs);  AuthKey ∉ range shrK;
         evs ∈ kerberos |]
  ==> ∃A. Says Tgs A (Crypt AuthKey {|Key ServKey, Agent B, T, ServTicket|})
             ∈ set evs"
```
⟨*proof*⟩

lemma *B_Authenticity:*
```
     "[| Crypt ServKey (Number Ta) ∈ parts (spies evs);
         Crypt AuthKey {|Key ServKey, Agent B, Number Tt, ServTicket|}
           ∈ parts (spies evs);
```

```
        Crypt (shrK A) {|Key AuthKey, Agent Tgs, Number Tk, AuthTicket|}
          ∈ parts (spies evs);
        ~ ExpirAuth Tk evs; ~ ExpirServ Tt evs;
        A ∉ bad;  B ∉ bad; B ≠ Tgs; evs ∈ kerberos |]
     ==> Says B A (Crypt ServKey (Number Ta)) ∈ set evs"
```
⟨*proof*⟩

lemma *B_Knows_B_Knows_ServKey_lemma:*
```
     "[| Says B A (Crypt ServKey (Number Ta)) ∈ set evs;
        Key ServKey ∉ analz (spies evs);
        A ∉ bad;  B ∉ bad; B ≠ Tgs; evs ∈ kerberos |]
     ==> B Issues A with (Crypt ServKey (Number Ta)) on evs"
```
⟨*proof*⟩

lemma *B_Knows_B_Knows_ServKey:*
```
     "[| Says B A (Crypt ServKey (Number Ta)) ∈ set evs;
        Crypt (shrK B) {|Agent A, Agent B, Key ServKey, Number Tt|}
          ∈ parts (spies evs);
        Crypt AuthKey {|Key ServKey, Agent B, Number Tt, ServTicket|}
          ∈ parts (spies evs);
        Crypt (shrK A) {|Key AuthKey, Agent Tgs, Number Tk, AuthTicket|}
          ∈ parts (spies evs);
        ~ ExpirServ Tt evs; ~ ExpirAuth Tk evs;
        A ∉ bad;  B ∉ bad; B ≠ Tgs; evs ∈ kerberos |]
     ==> B Issues A with (Crypt ServKey (Number Ta)) on evs"
```
⟨*proof*⟩

lemma *B_Knows_B_Knows_ServKey_refined:*
```
     "[| Says B A (Crypt ServKey (Number Ta)) ∈ set evs;
        Crypt (shrK B) {|Agent A, Agent B, Key ServKey, Number Tt|}
          ∈ parts (spies evs);
        ~ ExpirServ Tt evs;
        A ∉ bad;  B ∉ bad; B ≠ Tgs; evs ∈ kerberos |]
     ==> B Issues A with (Crypt ServKey (Number Ta)) on evs"
```
⟨*proof*⟩

lemma *A_Knows_B_Knows_ServKey:*
```
     "[| Crypt ServKey (Number Ta) ∈ parts (spies evs);
        Crypt AuthKey {|Key ServKey, Agent B, Number Tt, ServTicket|}
          ∈ parts (spies evs);
        Crypt (shrK A) {|Key AuthKey, Agent Tgs, Number Tk, AuthTicket|}
          ∈ parts (spies evs);
        ~ ExpirAuth Tk evs; ~ ExpirServ Tt evs;
        A ∉ bad;  B ∉ bad; B ≠ Tgs; evs ∈ kerberos |]
     ==> B Issues A with (Crypt ServKey (Number Ta)) on evs"
```
⟨*proof*⟩

lemma *K3_imp_K2:*
```
     "[| Says A Tgs
            {|AuthTicket, Crypt AuthKey {|Agent A, Number Ta|}, Agent B|}
```

```
          ∈ set evs;
        A ∉ bad;  evs ∈ kerberos |]
     ==> ∃Tk. Says Kas A (Crypt (shrK A)
                   {|Key AuthKey, Agent Tgs, Tk, AuthTicket|})
                 ∈ set evs"
```
⟨*proof*⟩

lemma *K4_trustworthy':*
```
     "[| Crypt AuthKey {|Key ServKey, Agent B, Number Tt, ServTicket|}
           ∈ parts (spies evs);
         Says Kas A (Crypt (shrK A)
                     {|Key AuthKey, Agent Tgs, Tk, AuthTicket|})
         ∈ set evs;
         Key AuthKey ∉ analz (spies evs);
         B ≠ Tgs; A ∉ bad;  B ∉ bad;  evs ∈ kerberos |]
   ==> Says Tgs A (Crypt AuthKey
                     {|Key ServKey, Agent B, Number Tt, ServTicket|})
         ∈ set evs"
```
⟨*proof*⟩

lemma *A_Knows_A_Knows_ServKey_lemma:*
```
     "[| Says A B {|ServTicket, Crypt ServKey {|Agent A, Number Ta|}|}
           ∈ set evs;
         Key ServKey ∉ analz (spies evs);
         B ≠ Tgs; A ∉ bad;  B ∉ bad;  evs ∈ kerberos |]
   ==> A Issues B with (Crypt ServKey {|Agent A, Number Ta|}) on evs"
```
⟨*proof*⟩

lemma *A_Knows_A_Knows_ServKey:*
```
     "[| Says A B {|ServTicket, Crypt ServKey {|Agent A, Number Ta|}|}
           ∈ set evs;
         Crypt (shrK A) {|Key AuthKey, Agent Tgs, Number Tk, AuthTicket|}
           ∈ parts (spies evs);
         Crypt AuthKey {|Key ServKey, Agent B, Number Tt, ServTicket|}
           ∈ parts (spies evs);
         ~ ExpirAuth Tk evs; ~ ExpirServ Tt evs;
         B ≠ Tgs; A ∉ bad;  B ∉ bad;  evs ∈ kerberos |]
   ==> A Issues B with (Crypt ServKey {|Agent A, Number Ta|}) on evs"
```
⟨*proof*⟩

lemma *B_Knows_A_Knows_ServKey:*
```
     "[| Crypt ServKey {|Agent A, Number Ta|} ∈ parts (spies evs);
         Crypt (shrK B) {|Agent A, Agent B, Key ServKey, Number Tt|}
           ∈ parts (spies evs);
         Crypt AuthKey {|Key ServKey, Agent B, Number Tt, ServTicket|}
           ∈ parts (spies evs);
         Crypt (shrK A) {|Key AuthKey, Agent Tgs, Number Tk, AuthTicket|}
           ∈ parts (spies evs);
         ~ ExpirServ Tt evs; ~ ExpirAuth Tk evs;
         B ≠ Tgs; A ∉ bad;  B ∉ bad;  evs ∈ kerberos |]
   ==> A Issues B with (Crypt ServKey {|Agent A, Number Ta|}) on evs"
```
⟨*proof*⟩

lemma *B_Knows_A_Knows_ServKey_refined:*
```
     "[| Crypt ServKey {|Agent A, Number Ta|} ∈ parts (spies evs);
         Crypt (shrK B) {|Agent A, Agent B, Key ServKey, Number Tt|}
           ∈ parts (spies evs);
         ~ ExpirServ Tt evs;
         B ≠ Tgs; A ∉ bad;  B ∉ bad;  evs ∈ kerberos |]
   ==> A Issues B with (Crypt ServKey {|Agent A, Number Ta|}) on evs"
```
⟨*proof*⟩

end

6 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.

```
consts  otway   :: "event list set"
inductive "otway"
  intros

   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_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 {|Nonce NA, Crypt (shrK A) {|Nonce NA, Key K|}|}
              ∈ set evs"
```

⟨*proof*⟩

```
lemma Gets_imp_Says [dest!]:
     "[| Gets B X ∈ set evs; evs ∈ otway |] ==> ∃A. Says A B X ∈ set evs"
```

⟨*proof*⟩

```
lemma OR2_analz_knows_Spy:
     "[| Gets B {|N, Agent A, Agent B, X|} ∈ set evs;  evs ∈ otway |]
      ==> X ∈ analz (knows Spy evs)"
```

⟨*proof*⟩

```
lemma OR4_analz_knows_Spy:
     "[| Gets B {|N, X, Crypt (shrK B) X'|} ∈ set evs;  evs ∈ otway |]
      ==> X ∈ analz (knows Spy evs)"
```

⟨*proof*⟩

lemmas *OR2_parts_knows_Spy =*
OR2_analz_knows_Spy [THEN analz_into_parts, standard]

Theorems of the form *X ∉ parts (knows Spy evs)* imply that NOBODY sends messages containing X!

Spy never sees a good agent's shared key!

lemma *Spy_see_shrK [simp]:*
"evs ∈ otway ==> (Key (shrK A) ∈ parts (knows Spy evs)) = (A ∈ bad)"
⟨*proof*⟩

lemma *Spy_analz_shrK [simp]:*
"evs ∈ otway ==> (Key (shrK A) ∈ analz (knows Spy evs)) = (A ∈ bad)"
⟨*proof*⟩

lemma *Spy_see_shrK_D [dest!]:*
"[|Key (shrK A) ∈ parts (knows Spy evs); evs ∈ otway|] ==> A ∈ bad"
⟨*proof*⟩

6.1 Towards Secrecy: Proofs Involving *analz*

lemma *Says_Server_message_form:*
"[| Says Server B {|NA, X, Crypt (shrK B) {|NB, Key K|}|} ∈ set evs;
evs ∈ otway |]
==> K ∉ range shrK & (∃i. NA = Nonce i) & (∃j. NB = Nonce j)"
⟨*proof*⟩

Session keys are not used to encrypt other session keys

The equality makes the induction hypothesis easier to apply

lemma *analz_image_freshK [rule_format]:*
"evs ∈ otway ==>
∀K KK. KK <= -(range shrK) -->
(Key K ∈ analz (Key‘KK Un (knows Spy evs))) =
(K ∈ KK | Key K ∈ analz (knows Spy evs))"
⟨*proof*⟩

lemma *analz_insert_freshK:*
"[| evs ∈ otway; KAB ∉ range shrK |] ==>
(Key K ∈ analz (insert (Key KAB) (knows Spy evs))) =
(K = KAB | Key K ∈ analz (knows Spy evs))"
⟨*proof*⟩

The Key K uniquely identifies the Server's message.

lemma *unique_session_keys:*
"[| Says Server B {|NA, X, Crypt (shrK B) {|NB, K|}|} ∈ set evs;
Says Server B' {|NA',X',Crypt (shrK B') {|NB',K|}|} ∈ set evs;
evs ∈ otway |] ==> X=X' & B=B' & NA=NA' & NB=NB'"
⟨*proof*⟩

6.2 Authenticity properties relating to NA

Only OR1 can have caused such a part of a message to appear.

lemma *Crypt_imp_OR1 [rule_format]:*
```
 "[| A ∉ bad;  evs ∈ otway |]
  ==> Crypt (shrK A) {|NA, Agent A, Agent B|} ∈ parts (knows Spy evs) -->
      Says A B {|NA, Agent A, Agent B,
                 Crypt (shrK A) {|NA, Agent A, Agent B|}|}
        ∈ set evs"
```
⟨*proof*⟩

lemma *Crypt_imp_OR1_Gets:*
```
     "[| Gets B {|NA, Agent A, Agent B,
                  Crypt (shrK A) {|NA, Agent A, Agent B|}|} ∈ set evs;
         A ∉ bad; evs ∈ otway |]
       ==> Says A B {|NA, Agent A, Agent B,
                      Crypt (shrK A) {|NA, Agent A, Agent B|}|}
             ∈ set evs"
```
⟨*proof*⟩

The Nonce NA uniquely identifies A's message

lemma *unique_NA:*
```
     "[| Crypt (shrK A) {|NA, Agent A, Agent B|} ∈ parts (knows Spy evs);
         Crypt (shrK A) {|NA, Agent A, Agent C|} ∈ parts (knows Spy evs);
         evs ∈ otway;  A ∉ bad |]
      ==> B = C"
```
⟨*proof*⟩

It is impossible to re-use a nonce in both OR1 and OR2. This holds because OR2 encrypts Nonce NB. It prevents the attack that can occur in the over-simplified version of this protocol: see *OtwayRees_Bad*.

lemma *no_nonce_OR1_OR2:*
```
   "[| Crypt (shrK A) {|NA, Agent A, Agent B|} ∈ parts (knows Spy evs);
       A ∉ bad;  evs ∈ otway |]
    ==> Crypt (shrK A) {|NA', NA, Agent A', Agent A|} ∉ parts (knows Spy evs)"
```
⟨*proof*⟩

Crucial property: If the encrypted message appears, and A has used NA to start a run, then it originated with the Server!

lemma *NA_Crypt_imp_Server_msg [rule_format]:*
```
     "[| A ∉ bad;  evs ∈ otway |]
      ==> Says A B {|NA, Agent A, Agent B,
                     Crypt (shrK A) {|NA, Agent A, Agent B|}|} ∈ set evs -->
          Crypt (shrK A) {|NA, Key K|} ∈ parts (knows Spy evs)
          --> (∃NB. Says Server B
                        {|NA,
                          Crypt (shrK A) {|NA, Key K|},
                          Crypt (shrK B) {|NB, Key K|}|} ∈ set evs)"
```
⟨*proof*⟩

Corollary: if A receives B's OR4 message and the nonce NA agrees then the key really did come from the Server! CANNOT prove this of the bad form of this protocol, even though we can prove *Spy_not_see_encrypted_key*

lemma *A_trusts_OR4:*
```
"[| Says A  B {|NA, Agent A, Agent B,
                Crypt (shrK A) {|NA, Agent A, Agent B|}|} ∈ set evs;
    Says B' A {|NA, Crypt (shrK A) {|NA, Key K|}|} ∈ set evs;
 A ∉ bad;  evs ∈ otway |]
 ==> ∃NB. Says Server B
           {|NA,
             Crypt (shrK A) {|NA, Key K|},
             Crypt (shrK B) {|NB, Key K|}|}
             ∈ set evs"
```
⟨*proof*⟩

Crucial secrecy property: Spy does not see the keys sent in msg OR3 Does not in itself guarantee security: an attack could violate the premises, e.g. by having *A = Spy*

lemma *secrecy_lemma:*
```
"[| A ∉ bad;  B ∉ bad;  evs ∈ otway |]
 ==> Says Server B
       {|NA, Crypt (shrK A) {|NA, Key K|},
         Crypt (shrK B) {|NB, Key K|}|} ∈ set evs -->
     Notes Spy {|NA, NB, Key K|} ∉ set evs -->
     Key K ∉ analz (knows Spy evs)"
```
⟨*proof*⟩

theorem *Spy_not_see_encrypted_key:*
```
"[| Says Server B
     {|NA, Crypt (shrK A) {|NA, Key K|},
           Crypt (shrK B) {|NB, Key K|}|} ∈ set evs;
    Notes Spy {|NA, NB, Key K|} ∉ set evs;
    A ∉ bad;  B ∉ bad;  evs ∈ otway |]
 ==> Key K ∉ analz (knows Spy evs)"
```
⟨*proof*⟩

This form is an immediate consequence of the previous result. It is similar to the assertions established by other methods. It is equivalent to the previous result in that the Spy already has *analz* and *synth* at his disposal. However, the conclusion *Key K ∉ knows Spy evs* appears not to be inductive: all the cases other than Fake are trivial, while Fake requires *Key K ∉ analz (knows Spy evs)*.

lemma *Spy_not_know_encrypted_key:*
```
"[| Says Server B
     {|NA, Crypt (shrK A) {|NA, Key K|},
           Crypt (shrK B) {|NB, Key K|}|} ∈ set evs;
    Notes Spy {|NA, NB, Key K|} ∉ set evs;
    A ∉ bad;  B ∉ bad;  evs ∈ otway |]
 ==> Key K ∉ knows Spy evs"
```
⟨*proof*⟩

A's guarantee. The Oops premise quantifies over NB because A cannot know what it is.

lemma *A_gets_good_key:*
```
"[| Says A  B {|NA, Agent A, Agent B,
                Crypt (shrK A) {|NA, Agent A, Agent B|}|} ∈ set evs;
    Says B' A {|NA, Crypt (shrK A) {|NA, Key K|}|} ∈ set evs;
```

```
        ∀NB. Notes Spy {|NA, NB, Key K|} ∉ set evs;
        A ∉ bad;  B ∉ bad;  evs ∈ otway |]
     ==> Key K ∉ analz (knows Spy evs)"
```
⟨*proof*⟩

6.3 Authenticity properties relating to NB

Only OR2 can have caused such a part of a message to appear. We do not know anything about X: it does NOT have to have the right form.

lemma *Crypt_imp_OR2:*
```
     "[| Crypt (shrK B) {|NA, NB, Agent A, Agent B|} ∈ parts (knows Spy evs);
         B ∉ bad;  evs ∈ otway |]
      ==> ∃X. Says B Server
                 {|NA, Agent A, Agent B, X,
                   Crypt (shrK B) {|NA, NB, Agent A, Agent B|}|}
                 ∈ set evs"
```
⟨*proof*⟩

The Nonce NB uniquely identifies B's message

lemma *unique_NB:*
```
     "[| Crypt (shrK B) {|NA, NB, Agent A, Agent B|} ∈ parts(knows Spy evs);
         Crypt (shrK B) {|NC, NB, Agent C, Agent B|} ∈ parts(knows Spy evs);
           evs ∈ otway;  B ∉ bad |]
         ==> NC = NA & C = A"
```
⟨*proof*⟩

If the encrypted message appears, and B has used Nonce NB, then it originated with the Server! Quite messy proof.

lemma *NB_Crypt_imp_Server_msg [rule_format]:*
```
 "[| B ∉ bad;  evs ∈ otway |]
  ==> Crypt (shrK B) {|NB, Key K|} ∈ parts (knows Spy evs)
      --> (∀X'. Says B Server
                     {|NA, Agent A, Agent B, X',
                       Crypt (shrK B) {|NA, NB, Agent A, Agent B|}|}
           ∈ set evs
           --> Says Server B
                {|NA, Crypt (shrK A) {|NA, Key K|},
                      Crypt (shrK B) {|NB, Key K|}|}
               ∈ set evs)"
```
⟨*proof*⟩

Guarantee for B: if it gets a message with matching NB then the Server has sent the correct message.

theorem *B_trusts_OR3:*
```
     "[| Says B Server {|NA, Agent A, Agent B, X',
                        Crypt (shrK B) {|NA, NB, Agent A, Agent B|} |}
           ∈ set evs;
         Gets B {|NA, X, Crypt (shrK B) {|NB, Key K|}|} ∈ set evs;
         B ∉ bad;  evs ∈ otway |]
      ==> Says Server B
               {|NA,
                 Crypt (shrK A) {|NA, Key K|},
```

```
                Crypt (shrK B) {|NB, Key K|}|}
                ∈ set evs"
```
⟨*proof*⟩

The obvious combination of *B_trusts_OR3* with *Spy_not_see_encrypted_key*

lemma *B_gets_good_key:*
```
     "[| Says B Server {|NA, Agent A, Agent B, X',
                         Crypt (shrK B) {|NA, NB, Agent A, Agent B|} |}
           ∈ set evs;
         Gets B {|NA, X, Crypt (shrK B) {|NB, Key K|}|} ∈ set evs;
         Notes Spy {|NA, NB, Key K|} ∉ set evs;
         A ∉ bad;  B ∉ bad;  evs ∈ otway |]
       ==> Key K ∉ analz (knows Spy evs)"
```
⟨*proof*⟩

lemma *OR3_imp_OR2:*
```
     "[| Says Server B
              {|NA, Crypt (shrK A) {|NA, Key K|},
                Crypt (shrK B) {|NB, Key K|}|} ∈ set evs;
         B ∉ bad;  evs ∈ otway |]
  ==> ∃X. Says B Server {|NA, Agent A, Agent B, X,
                          Crypt (shrK B) {|NA, NB, Agent A, Agent B|} |}
              ∈ set evs"
```
⟨*proof*⟩

After getting and checking OR4, agent A can trust that B has been active. We could probably prove that X has the expected form, but that is not strictly necessary for authentication.

theorem *A_auths_B:*
```
     "[| Says B' A {|NA, Crypt (shrK A) {|NA, Key K|}|} ∈ set evs;
         Says A  B {|NA, Agent A, Agent B,
                     Crypt (shrK A) {|NA, Agent A, Agent B|}|} ∈ set evs;
         A ∉ bad;  B ∉ bad;  evs ∈ otway |]
  ==> ∃NB X. Says B Server {|NA, Agent A, Agent B, X,
                             Crypt (shrK B)  {|NA, NB, Agent A, Agent B|}
|}
                ∈ set evs"
```
⟨*proof*⟩

end

7 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)

```
consts  otway   :: "event list set"
inductive "otway"
  intros
   Nil: — The empty trace
        "[] ∈ otway"
```

Fake: — The Spy may say anything he can say. The sender field is correct, but agents don't use that information.

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

Reception: — A message that has been sent can be received by the intended recipient.

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

OR1: — Alice initiates a protocol run

```
        "evs1 ∈ otway
         ==> Says A B {|Agent A, Agent B, Nonce NA|} # evs1 ∈ otway"
```

OR2: — Bob's response to Alice's message.

```
        "[| evs2 ∈ otway;
            Gets B {|Agent A, Agent B, Nonce NA|} ∈set evs2 |]
         ==> Says B Server {|Agent A, Agent B, Nonce NA, Nonce NB|}
               # evs2 ∈ otway"
```

OR3: — The Server receives Bob's message. Then he sends a new session key to Bob with a packet for forwarding to Alice.

```
        "[| evs3 ∈ otway;  Key KAB ∉ used evs3;
            Gets Server {|Agent A, Agent B, Nonce NA, Nonce NB|}
              ∈set evs3 |]
         ==> Says Server B
              {|Crypt (shrK A) {|Nonce NA, Agent A, Agent B, Key KAB|},
                Crypt (shrK B) {|Nonce NB, Agent A, Agent B, Key KAB|}|}
             # evs3 ∈ otway"
```

OR4: — Bob receives the Server's (?) message and compares the Nonces with those in the message he previously sent the Server. Need *B* ≠ *Server* because we allow messages to self.

```
        "[| evs4 ∈ otway;  B ≠ Server;
            Says B Server {|Agent A, Agent B, Nonce NA, Nonce NB|} ∈set evs4;
            Gets B {|X, Crypt(shrK B){|Nonce NB,Agent A,Agent B,Key K|}|}
              ∈set evs4 |]
         ==> Says B A X # evs4 ∈ otway"
```

Oops: — This message models possible leaks of session keys. The nonces identify the protocol run.

```
        "[| evso ∈ otway;
            Says Server B
                     {|Crypt (shrK A) {|Nonce NA, Agent A, Agent B, Key K|},
                       Crypt (shrK B) {|Nonce NB, Agent A, Agent B, Key K|}|}
```

```
          ∈set evso |]
       ==> Notes Spy {|Nonce NA, Nonce NB, Key K|} # evso ∈ otway"
```

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

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

lemma *"[| B ≠ Server; Key K ∉ used [] |]*
==> ∃evs ∈ otway.
Says B A (Crypt (shrK A) {|Nonce NA, Agent A, Agent B, Key K|})
∈ set evs"
⟨*proof*⟩

lemma *Gets_imp_Says [dest!]:*
"[| Gets B X ∈ set evs; evs ∈ otway |] ==> ∃A. Says A B X ∈ set evs"
⟨*proof*⟩

For reasoning about the encrypted portion of messages

lemma *OR4_analz_knows_Spy:*
"[| Gets B {|X, Crypt(shrK B) X'|} ∈ set evs; evs ∈ otway |]
==> X ∈ analz (knows Spy evs)"
⟨*proof*⟩

Theorems of the form *X ∉ parts (knows Spy evs)* imply that NOBODY sends messages containing X!

Spy never sees a good agent's shared key!

lemma *Spy_see_shrK [simp]:*
"evs ∈ otway ==> (Key (shrK A) ∈ parts (knows Spy evs)) = (A ∈ bad)"
⟨*proof*⟩

lemma *Spy_analz_shrK [simp]:*
"evs ∈ otway ==> (Key (shrK A) ∈ analz (knows Spy evs)) = (A ∈ bad)"
⟨*proof*⟩

lemma *Spy_see_shrK_D [dest!]:*
"[|Key (shrK A) ∈ parts (knows Spy evs); evs ∈ otway|] ==> A ∈ bad"
⟨*proof*⟩

7.1 Proofs involving analz

Describes the form of K and NA when the Server sends this message.

lemma *Says_Server_message_form:*
"[| Says Server B
{|Crypt (shrK A) {|NA, Agent A, Agent B, Key K|},
Crypt (shrK B) {|NB, Agent A, Agent B, Key K|}|}
∈ set evs;
evs ∈ otway |]
==> K ∉ range shrK & (∃i. NA = Nonce i) & (∃j. NB = Nonce j)"
⟨*proof*⟩

Session keys are not used to encrypt other session keys

The equality makes the induction hypothesis easier to apply

lemma *analz_image_freshK [rule_format]:*
```
 "evs ∈ otway ==>
   ∀K KK. KK <= -(range shrK) -->
          (Key K ∈ analz (Key‘KK Un (knows Spy evs))) =
          (K ∈ KK | Key K ∈ analz (knows Spy evs))"
```
⟨*proof*⟩

lemma *analz_insert_freshK:*
```
  "[| evs ∈ otway;  KAB ∉ range shrK |] ==>
      (Key K ∈ analz (insert (Key KAB) (knows Spy evs))) =
      (K = KAB | Key K ∈ analz (knows Spy evs))"
```
⟨*proof*⟩

The Key K uniquely identifies the Server's message.

lemma *unique_session_keys:*
```
     "[| Says Server B
          {|Crypt (shrK A) {|NA, Agent A, Agent B, K|},
            Crypt (shrK B) {|NB, Agent A, Agent B, K|}|}
         ∈ set evs;
        Says Server B'
          {|Crypt (shrK A') {|NA', Agent A', Agent B', K|},
            Crypt (shrK B') {|NB', Agent A', Agent B', K|}|}
         ∈ set evs;
        evs ∈ otway |]
     ==> A=A' & B=B' & NA=NA' & NB=NB'"
```
⟨*proof*⟩

7.2 Authenticity properties relating to NA

If the encrypted message appears then it originated with the Server!

lemma *NA_Crypt_imp_Server_msg [rule_format]:*
```
    "[| A ∉ bad;  A ≠ B;  evs ∈ otway |]
     ==> Crypt (shrK A) {|NA, Agent A, Agent B, Key K|} ∈ parts (knows Spy
evs)
       --> (∃NB. Says Server B
                    {|Crypt (shrK A) {|NA, Agent A, Agent B, Key K|},
                      Crypt (shrK B) {|NB, Agent A, Agent B, Key K|}|}
                    ∈ set evs)"
```
⟨*proof*⟩

Corollary: if A receives B's OR4 message then it originated with the Server. Freshness may be inferred from nonce NA.

lemma *A_trusts_OR4:*
```
     "[| Says B' A (Crypt (shrK A) {|NA, Agent A, Agent B, Key K|}) ∈ set
evs;
         A ∉ bad;  A ≠ B;  evs ∈ otway |]
      ==> ∃NB. Says Server B
                  {|Crypt (shrK A) {|NA, Agent A, Agent B, Key K|},
                    Crypt (shrK B) {|NB, Agent A, Agent B, Key K|}|}
```

```
                ∈ set evs"
```
⟨*proof*⟩

Crucial secrecy property: Spy does not see the keys sent in msg OR3 Does not in itself guarantee security: an attack could violate the premises, e.g. by having `A = Spy`

lemma *secrecy_lemma:*
```
     "[| A ∉ bad;  B ∉ bad;  evs ∈ otway |]
      ==> Says Server B
           {|Crypt (shrK A) {|NA, Agent A, Agent B, Key K|},
             Crypt (shrK B) {|NB, Agent A, Agent B, Key K|}|}
          ∈ set evs -->
          Notes Spy {|NA, NB, Key K|} ∉ set evs -->
          Key K ∉ analz (knows Spy evs)"
```
⟨*proof*⟩

lemma *Spy_not_see_encrypted_key:*
```
     "[| Says Server B
           {|Crypt (shrK A) {|NA, Agent A, Agent B, Key K|},
             Crypt (shrK B) {|NB, Agent A, Agent B, Key K|}|}
          ∈ set evs;
         Notes Spy {|NA, NB, Key K|} ∉ set evs;
         A ∉ bad;  B ∉ bad;  evs ∈ otway |]
      ==> Key K ∉ analz (knows Spy evs)"
```
⟨*proof*⟩

A's guarantee. The Oops premise quantifies over NB because A cannot know what it is.

lemma *A_gets_good_key:*
```
     "[| Says B' A (Crypt (shrK A) {|NA, Agent A, Agent B, Key K|}) ∈ set
evs;
         ∀NB. Notes Spy {|NA, NB, Key K|} ∉ set evs;
         A ∉ bad;  B ∉ bad;  A ≠ B;  evs ∈ otway |]
      ==> Key K ∉ analz (knows Spy evs)"
```
⟨*proof*⟩

7.3 Authenticity properties relating to NB

If the encrypted message appears then it originated with the Server!

lemma *NB_Crypt_imp_Server_msg [rule_format]:*
```
 "[| B ∉ bad;  A ≠ B;  evs ∈ otway |]
  ==> Crypt (shrK B) {|NB, Agent A, Agent B, Key K|} ∈ parts (knows Spy evs)
      --> (∃NA. Says Server B
                  {|Crypt (shrK A) {|NA, Agent A, Agent B, Key K|},
                    Crypt (shrK B) {|NB, Agent A, Agent B, Key K|}|}
                  ∈ set evs)"
```
⟨*proof*⟩

Guarantee for B: if it gets a well-formed certificate then the Server has sent the correct message in round 3.

lemma *B_trusts_OR3:*

```
"[| Says S B {|X, Crypt (shrK B) {|NB, Agent A, Agent B, Key K|}|}
      ∈ set evs;
   B ∉ bad;  A ≠ B;  evs ∈ otway |]
 ==> ∃NA. Says Server B
              {|Crypt (shrK A) {|NA, Agent A, Agent B, Key K|},
                Crypt (shrK B) {|NB, Agent A, Agent B, Key K|}|}
              ∈ set evs"
```

⟨*proof*⟩

The obvious combination of *B_trusts_OR3* with *Spy_not_see_encrypted_key*

lemma *B_gets_good_key:*

```
"[| Gets B {|X, Crypt (shrK B) {|NB, Agent A, Agent B, Key K|}|}
     ∈ set evs;
   ∀NA. Notes Spy {|NA, NB, Key K|} ∉ set evs;
   A ∉ bad;  B ∉ bad;  A ≠ B;  evs ∈ otway |]
 ==> Key K ∉ analz (knows Spy evs)"
```

⟨*proof*⟩

end

8 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.

consts *otway :: "event list set"*
inductive *"otway"*
intros
Nil: — The empty trace

```
"[] ∈ otway"
```

Fake: — The Spy may say anything he can say. The sender field is correct, but agents don't use that information.

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

Reception: — A message that has been sent can be received by the intended recipient.

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

OR1: — Alice initiates a protocol run

```
"[| evs1 ∈ otway;  Nonce NA ∉ used evs1 |]
```

```
==> Says A B {|Nonce NA, Agent A, Agent B,
               Crypt (shrK A) {|Nonce NA, Agent A, Agent B|} |}
      # evs1 ∈ otway"
```

OR2: — Bob's response to Alice's message. This variant of the protocol does NOT encrypt NB.

```
"[| evs2 ∈ otway;  Nonce NB ∉ used evs2;
    Gets B {|Nonce NA, Agent A, Agent B, X|} ∈ set evs2 |]
 ==> Says B Server
        {|Nonce NA, Agent A, Agent B, X, Nonce NB,
          Crypt (shrK B) {|Nonce NA, Agent A, Agent B|}|}
       # evs2 ∈ otway"
```

OR3: — The Server receives Bob's message and checks that the three NAs match. Then he sends a new session key to Bob with a packet for forwarding to Alice.

```
"[| evs3 ∈ otway;  Key KAB ∉ used evs3;
    Gets Server
         {|Nonce NA, Agent A, Agent B,
           Crypt (shrK A) {|Nonce NA, Agent A, Agent B|},
           Nonce NB,
           Crypt (shrK B) {|Nonce NA, Agent A, Agent B|}|}
      ∈ set evs3 |]
 ==> Says Server B
        {|Nonce NA,
          Crypt (shrK A) {|Nonce NA, Key KAB|},
          Crypt (shrK B) {|Nonce NB, Key KAB|}|}
       # evs3 ∈ otway"
```

OR4: — Bob receives the Server's (?) message and compares the Nonces with those in the message he previously sent the Server. Need *B ≠ Server* because we allow messages to self.

```
"[| evs4 ∈ otway;  B ≠ Server;
    Says B Server {|Nonce NA, Agent A, Agent B, X', Nonce NB,
                    Crypt (shrK B) {|Nonce NA, Agent A, Agent B|}|}
      ∈ set evs4;
    Gets B {|Nonce NA, X, Crypt (shrK B) {|Nonce NB, Key K|}|}
      ∈ set evs4 |]
 ==> Says B A {|Nonce NA, X|} # evs4 ∈ otway"
```

Oops: — This message models possible leaks of session keys. The nonces identify the protocol run.

```
"[| evso ∈ otway;
    Says Server B {|Nonce NA, X, Crypt (shrK B) {|Nonce NB, Key K|}|}
      ∈ set evso |]
 ==> Notes Spy {|Nonce NA, Nonce NB, Key K|} # evso ∈ otway"
```

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

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

lemma *"[| B ≠ Server; Key K ∉ used [] |]*

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

lemma *Gets_imp_Says [dest!]:*
```
     "[| Gets B X ∈ set evs; evs ∈ otway |] ==> ∃A. Says A B X ∈ set evs"
```
⟨*proof*⟩

8.1 For reasoning about the encrypted portion of messages

lemma *OR2_analz_knows_Spy:*
```
     "[| Gets B {|N, Agent A, Agent B, X|} ∈ set evs;  evs ∈ otway |]
      ==> X ∈ analz (knows Spy evs)"
```
⟨*proof*⟩

lemma *OR4_analz_knows_Spy:*
```
     "[| Gets B {|N, X, Crypt (shrK B) X'|} ∈ set evs;  evs ∈ otway |]
      ==> X ∈ analz (knows Spy evs)"
```
⟨*proof*⟩

lemma *Oops_parts_knows_Spy:*
```
     "Says Server B {|NA, X, Crypt K' {|NB,K|}|} ∈ set evs
      ==> K ∈ parts (knows Spy evs)"
```
⟨*proof*⟩

Forwarding lemma: see comments in OtwayRees.thy

lemmas *OR2_parts_knows_Spy =*
```
    OR2_analz_knows_Spy [THEN analz_into_parts, standard]
```

Theorems of the form *X ∉ parts (knows Spy evs)* imply that NOBODY sends messages containing X!

Spy never sees a good agent's shared key!

lemma *Spy_see_shrK [simp]:*
```
     "evs ∈ otway ==> (Key (shrK A) ∈ parts (knows Spy evs)) = (A ∈ bad)"
```
⟨*proof*⟩

lemma *Spy_analz_shrK [simp]:*
```
     "evs ∈ otway ==> (Key (shrK A) ∈ analz (knows Spy evs)) = (A ∈ bad)"
```
⟨*proof*⟩

lemma *Spy_see_shrK_D [dest!]:*
```
     "[|Key (shrK A) ∈ parts (knows Spy evs);  evs ∈ otway|] ==> A ∈ bad"
```
⟨*proof*⟩

8.2 Proofs involving analz

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

lemma *Says_Server_message_form:*

```
     "[| Says Server B {|NA, X, Crypt (shrK B) {|NB, Key K|}|} ∈ set evs;
         evs ∈ otway |]
      ==> K ∉ range shrK & (∃i. NA = Nonce i) & (∃j. NB = Nonce j)"
```
⟨*proof*⟩

Session keys are not used to encrypt other session keys

The equality makes the induction hypothesis easier to apply

lemma *analz_image_freshK [rule_format]:*
```
 "evs ∈ otway ==>
   ∀K KK. KK <= -(range shrK) -->
          (Key K ∈ analz (Key‘KK Un (knows Spy evs))) =
          (K ∈ KK | Key K ∈ analz (knows Spy evs))"
```
⟨*proof*⟩

lemma *analz_insert_freshK:*
```
  "[| evs ∈ otway;  KAB ∉ range shrK |] ==>
      (Key K ∈ analz (insert (Key KAB) (knows Spy evs))) =
      (K = KAB | Key K ∈ analz (knows Spy evs))"
```
⟨*proof*⟩

The Key K uniquely identifies the Server's message.

lemma *unique_session_keys:*
```
     "[| Says Server B {|NA, X, Crypt (shrK B) {|NB, K|}|}   ∈ set evs;
         Says Server B' {|NA',X',Crypt (shrK B') {|NB',K|}|} ∈ set evs;
         evs ∈ otway |] ==> X=X' & B=B' & NA=NA' & NB=NB'"
```
⟨*proof*⟩

Crucial secrecy property: Spy does not see the keys sent in msg OR3 Does not in itself guarantee security: an attack could violate the premises, e.g. by having `A = Spy`

lemma *secrecy_lemma:*
```
 "[| A ∉ bad;  B ∉ bad;  evs ∈ otway |]
  ==> Says Server B
        {|NA, Crypt (shrK A) {|NA, Key K|},
          Crypt (shrK B) {|NB, Key K|}|} ∈ set evs -->
      Notes Spy {|NA, NB, Key K|} ∉ set evs -->
      Key K ∉ analz (knows Spy evs)"
```
⟨*proof*⟩

lemma *Spy_not_see_encrypted_key:*
```
     "[| Says Server B
          {|NA, Crypt (shrK A) {|NA, Key K|},
                Crypt (shrK B) {|NB, Key K|}|} ∈ set evs;
         Notes Spy {|NA, NB, Key K|} ∉ set evs;
         A ∉ bad;  B ∉ bad;  evs ∈ otway |]
      ==> Key K ∉ analz (knows Spy evs)"
```
⟨*proof*⟩

8.3 Attempting to prove stronger properties

Only OR1 can have caused such a part of a message to appear. The premise `A ≠ B` prevents OR2's similar-looking cryptogram from being picked up. Original

Otway-Rees doesn't need it.

lemma *Crypt_imp_OR1 [rule_format]:*
```
     "[| A ∉ bad;  A ≠ B;  evs ∈ otway |]
      ==> Crypt (shrK A) {|NA, Agent A, Agent B|} ∈ parts (knows Spy evs)
-->
          Says A B {|NA, Agent A, Agent B,
                     Crypt (shrK A) {|NA, Agent A, Agent B|}|}  ∈ set evs"
```
⟨*proof*⟩

Crucial property: If the encrypted message appears, and A has used NA to start a run, then it originated with the Server! The premise `A ≠ B` allows use of *Crypt_imp_OR1*

Only it is FALSE. Somebody could make a fake message to Server substituting some other nonce NA' for NB.

lemma
```
"[| A ∉ bad;  A ≠ B;  evs ∈ otway |]
       ==> Crypt (shrK A) {|NA, Key K|} ∈ parts (knows Spy evs) -->
           Says A B {|NA, Agent A, Agent B,
                      Crypt (shrK A) {|NA, Agent A, Agent B|}|}
            ∈ set evs -->
           (∃B NB. Says Server B
                {|NA,
                  Crypt (shrK A) {|NA, Key K|},
                  Crypt (shrK B) {|NB, Key K|}|}  ∈ set evs)"
```
⟨*proof*⟩

end

9 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

consts
```
woolam  :: "event list set"
```
inductive *woolam*
intros

```
Nil:  "[] ∈ woolam"


Fake: "[| evsf ∈ woolam;  X ∈ synth (analz (spies evsf)) |]
       ==> Says Spy B X  # evsf ∈ woolam"
```

```
WL1:  "evs1 ∈ woolam ==> Says A B (Agent A) # evs1 ∈ woolam"


WL2:  "[| evs2 ∈ woolam;  Says A' B (Agent A) ∈ set evs2 |]
       ==> Says B A (Nonce NB) # evs2 ∈ woolam"


WL3:  "[| evs3 ∈ woolam;
          Says A  B (Agent A)  ∈ set evs3;
          Says B' A (Nonce NB) ∈ set evs3 |]
       ==> Says A B (Crypt (shrK A) (Nonce NB)) # evs3 ∈ woolam"


WL4:  "[| evs4 ∈ woolam;
          Says A'  B X         ∈ set evs4;
          Says A'' B (Agent A) ∈ set evs4 |]
       ==> Says B Server {|Agent A, Agent B, X|} # evs4 ∈ woolam"


WL5:  "[| evs5 ∈ woolam;
          Says B' Server {|Agent A, Agent B, Crypt (shrK A) (Nonce NB)|}
            ∈ set evs5 |]
       ==> Says Server B (Crypt (shrK B) {|Agent A, Nonce NB|})
             # evs5 ∈ woolam"
```

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

lemma *"∃NB. ∃evs ∈ woolam.*
Says Server B (Crypt (shrK B) {|Agent A, Nonce NB|}) ∈ set evs"
⟨*proof*⟩

lemma *Spy_see_shrK [simp]:*
"evs ∈ woolam ==> (Key (shrK A) ∈ parts (spies evs)) = (A ∈ bad)"
⟨*proof*⟩

lemma *Spy_analz_shrK [simp]:*
"evs ∈ woolam ==> (Key (shrK A) ∈ analz (spies evs)) = (A ∈ bad)"
⟨*proof*⟩

lemma *Spy_see_shrK_D [dest!]:*
"[|Key (shrK A) ∈ parts (knows Spy evs); evs ∈ woolam|] ==> A ∈ bad"

⟨*proof*⟩

lemma *NB_Crypt_imp_Alice_msg:*
```
"[| Crypt (shrK A) (Nonce NB) ∈ parts (spies evs);
    A ∉ bad;  evs ∈ woolam |]
 ==> ∃B. Says A B (Crypt (shrK A) (Nonce NB)) ∈ set evs"
```
⟨*proof*⟩

lemma *Server_trusts_WL4 [dest]:*
```
"[| Says B' Server {|Agent A, Agent B, Crypt (shrK A) (Nonce NB)|}
      ∈ set evs;
    A ∉ bad;  evs ∈ woolam |]
 ==> ∃B. Says A B (Crypt (shrK A) (Nonce NB)) ∈ set evs"
```
⟨*proof*⟩

lemma *Server_sent_WL5 [dest]:*
```
"[| Says Server B (Crypt (shrK B) {|Agent A, NB|}) ∈ set evs;
    evs ∈ woolam |]
 ==> ∃B'. Says B' Server {|Agent A, Agent B, Crypt (shrK A) NB|}
        ∈ set evs"
```
⟨*proof*⟩

lemma *NB_Crypt_imp_Server_msg [rule_format]:*
```
"[| Crypt (shrK B) {|Agent A, NB|} ∈ parts (spies evs);
    B ∉ bad;  evs ∈ woolam |]
 ==> Says Server B (Crypt (shrK B) {|Agent A, NB|}) ∈ set evs"
```
⟨*proof*⟩

lemma *B_trusts_WL5:*
```
"[| Says S B (Crypt (shrK B) {|Agent A, Nonce NB|}): set evs;
    A ∉ bad;  B ∉ bad;  evs ∈ woolam  |]
 ==> ∃B. Says A B (Crypt (shrK A) (Nonce NB)) ∈ set evs"
```
⟨*proof*⟩

lemma *B_said_WL2:*
```
"[| Says B A (Nonce NB) ∈ set evs;  B ≠ Spy;  evs ∈ woolam |]
 ==> ∃A'. Says A' B (Agent A) ∈ set evs"
```
⟨*proof*⟩

```
lemma "[| A ∉ bad;  B ≠ Spy;  evs ∈ woolam |]
  ==> Crypt (shrK A) (Nonce NB) ∈ parts (spies evs) &
      Says B A (Nonce NB) ∈ set evs
      --> Says A B (Crypt (shrK A) (Nonce NB)) ∈ set evs"
```
⟨*proof*⟩

end

10 The Otway-Bull Recursive Authentication Protocol

theory *Recur* **imports** *Public* **begin**

End marker for message bundles

```
syntax       END :: "msg"
translations "END" == "Number 0"
```

```
consts     respond :: "event list => (msg*msg*key)set"
inductive "respond evs"
  intros
   One:  "Key KAB ∉ used evs
          ==> (Hash[Key(shrK A)] {|Agent A, Agent B, Nonce NA, END|},
               {|Crypt (shrK A) {|Key KAB, Agent B, Nonce NA|}, END|},
               KAB)   ∈ respond evs"


   Cons: "[| (PA, RA, KAB) ∈ respond evs;
             Key KBC ∉ used evs;  Key KBC ∉ parts {RA};
             PA = Hash[Key(shrK A)] {|Agent A, Agent B, Nonce NA, P|} |]
          ==> (Hash[Key(shrK B)] {|Agent B, Agent C, Nonce NB, PA|},
               {|Crypt (shrK B) {|Key KBC, Agent C, Nonce NB|},
                 Crypt (shrK B) {|Key KAB, Agent A, Nonce NB|},
                 RA|},
               KBC)
              ∈ respond evs"
```

```
consts     responses :: "event list => msg set"
inductive "responses evs"
  intros

   Nil:  "END ∈ responses evs"

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

```
consts     recur   :: "event list set"
inductive "recur"
  intros

   Nil:  "[] ∈ recur"


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


   RA1:  "[| evs1 ∈ recur;  Nonce NA ∉ used evs1 |]
          ==> Says A B (Hash[Key(shrK A)] {|Agent A, Agent B, Nonce NA, END|})
              # evs1 ∈ recur"


   RA2:  "[| evs2 ∈ recur;  Nonce NB ∉ used evs2;
             Says A' B PA ∈ set evs2 |]
          ==> Says B C (Hash[Key(shrK B)] {|Agent B, Agent C, Nonce NB, PA|})
              # evs2 ∈ recur"


   RA3:  "[| evs3 ∈ recur;  Says B' Server PB ∈ set evs3;
             (PB,RB,K) ∈ respond evs3 |]
          ==> Says Server B RB # evs3 ∈ recur"


   RA4:  "[| evs4 ∈ recur;
             Says B  C {|XH, Agent B, Agent C, Nonce NB,
                         XA, Agent A, Agent B, Nonce NA, P|} ∈ set evs4;
             Says C' B {|Crypt (shrK B) {|Key KBC, Agent C, Nonce NB|},
                         Crypt (shrK B) {|Key KAB, Agent A, Nonce NB|},
                         RA|} ∈ set evs4 |]
          ==> Says B A RA # evs4 ∈ recur"
```

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

Simplest case: Alice goes directly to the server

```
lemma "Key K ∉ used []
       ==> ∃NA. ∃evs ∈ recur.
             Says Server A {|Crypt (shrK A) {|Key K, Agent Server, Nonce NA|},
                   END|}  ∈ set evs"
```

⟨*proof*⟩

Case two: Alice, Bob and the server

```
lemma "[| Key K ∉ used []; Key K' ∉ used []; K ≠ K';
          Nonce NA ∉ used []; Nonce NB ∉ used []; NA < NB |]
       ==> ∃NA. ∃evs ∈ recur.
```

```
        Says B A {|Crypt (shrK A) {|Key K, Agent B, Nonce NA|},
                   END|}  ∈ set evs"
```
⟨*proof*⟩

lemma *"[| Key K ∉ used []; Key K' ∉ used [];*
Key K'' ∉ used []; K ≠ K'; K' ≠ K''; K ≠ K'';
Nonce NA ∉ used []; Nonce NB ∉ used []; Nonce NC ∉ used [];
NA < NB; NB < NC |]
==> ∃K. ∃NA. ∃evs ∈ recur.
Says B A {|Crypt (shrK A) {|Key K, Agent B, Nonce NA|},
END|} ∈ set evs"
⟨*proof*⟩

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

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

Simple inductive reasoning about responses

lemma *respond_imp_responses:*
"(PA,RB,KAB) ∈ respond evs ==> RB ∈ responses evs"
⟨*proof*⟩

lemmas *RA2_analz_spies = Says_imp_spies [THEN analz.Inj]*

lemma *RA4_analz_spies:*
"Says C' B {|Crypt K X, X', RA|} ∈ set evs ==> RA ∈ analz (spies evs)"
⟨*proof*⟩

lemmas *RA2_parts_spies = RA2_analz_spies [THEN analz_into_parts]*
lemmas *RA4_parts_spies = RA4_analz_spies [THEN analz_into_parts]*

lemma *Spy_see_shrK [simp]:*
"evs ∈ recur ==> (Key (shrK A) ∈ parts (spies evs)) = (A ∈ bad)"
⟨*proof*⟩

lemma *Spy_analz_shrK [simp]:*

```
    "evs ∈ recur ==> (Key (shrK A) ∈ analz (spies evs)) = (A ∈ bad)"
```
⟨*proof*⟩

lemma *Spy_see_shrK_D [dest!]:*
```
    "[|Key (shrK A) ∈ parts (knows Spy evs);  evs ∈ recur|] ==> A ∈ bad"
```
⟨*proof*⟩

lemma *resp_analz_image_freshK_lemma:*
```
    "[| RB ∈ responses evs;
        ∀K KK. KK ⊆ - (range shrK) -->
                  (Key K ∈ analz (Key‘KK Un H)) =
                  (K ∈ KK | Key K ∈ analz H) |]
    ==> ∀K KK. KK ⊆ - (range shrK) -->
                  (Key K ∈ analz (insert RB (Key‘KK Un H))) =
                  (K ∈ KK | Key K ∈ analz (insert RB H))"
```
⟨*proof*⟩

Version for the protocol. Proof is easy, thanks to the lemma.

lemma *raw_analz_image_freshK:*
```
 "evs ∈ recur ==>
   ∀K KK. KK ⊆ - (range shrK) -->
          (Key K ∈ analz (Key‘KK Un (spies evs))) =
          (K ∈ KK | Key K ∈ analz (spies evs))"
```
⟨*proof*⟩

lemmas *resp_analz_image_freshK =*
```
      resp_analz_image_freshK_lemma [OF _ raw_analz_image_freshK]
```

lemma *analz_insert_freshK:*
```
    "[| evs ∈ recur;  KAB ∉ range shrK |]
     ==> (Key K ∈ analz (insert (Key KAB) (spies evs))) =
         (K = KAB | Key K ∈ analz (spies evs))"
```
⟨*proof*⟩

Everything that's hashed is already in past traffic.

lemma *Hash_imp_body:*
```
    "[| Hash {|Key(shrK A), X|} ∈ parts (spies evs);
        evs ∈ recur;  A ∉ bad |] ==> X ∈ parts (spies evs)"
```
⟨*proof*⟩

lemma *unique_NA:*
```
  "[| Hash {|Key(shrK A), Agent A, B, NA, P|} ∈ parts (spies evs);
      Hash {|Key(shrK A), Agent A, B',NA, P'|} ∈ parts (spies evs);
```

```
      evs ∈ recur;  A ∉ bad |]
    ==> B=B' & P=P'"
```
⟨*proof*⟩

lemma *shrK_in_analz_respond [simp]:*
```
     "[| RB ∈ responses evs;  evs ∈ recur |]
  ==> (Key (shrK B) ∈ analz (insert RB (spies evs))) = (B:bad)"
```
⟨*proof*⟩

lemma *resp_analz_insert_lemma:*
```
     "[| Key K ∈ analz (insert RB H);
         ∀K KK. KK ⊆ - (range shrK) -->
                   (Key K ∈ analz (Key`KK Un H)) =
                   (K ∈ KK | Key K ∈ analz H);
         RB ∈ responses evs |]
     ==> (Key K ∈ parts{RB} | Key K ∈ analz H)"
```
⟨*proof*⟩

lemmas *resp_analz_insert =*
```
       resp_analz_insert_lemma [OF _ raw_analz_image_freshK]
```

The last key returned by respond indeed appears in a certificate

lemma *respond_certificate:*
```
     "(Hash[Key(shrK A)] {|Agent A, B, NA, P|}, RA, K) ∈ respond evs
      ==> Crypt (shrK A) {|Key K, B, NA|} ∈ parts {RA}"
```
⟨*proof*⟩

lemma *unique_lemma [rule_format]:*
```
     "(PB,RB,KXY) ∈ respond evs ==>
      ∀A B N. Crypt (shrK A) {|Key K, Agent B, N|} ∈ parts {RB} -->
      (∀A' B' N'. Crypt (shrK A') {|Key K, Agent B', N'|} ∈ parts {RB} -->
      (A'=A & B'=B) | (A'=B & B'=A))"
```
⟨*proof*⟩

lemma *unique_session_keys:*
```
     "[| Crypt (shrK A) {|Key K, Agent B, N|} ∈ parts {RB};
         Crypt (shrK A') {|Key K, Agent B', N'|} ∈ parts {RB};
         (PB,RB,KXY) ∈ respond evs |]
      ==> (A'=A & B'=B) | (A'=B & B'=A)"
```
⟨*proof*⟩

lemma *respond_Spy_not_see_session_key [rule_format]:*
```
     "[| (PB,RB,KAB) ∈ respond evs;  evs ∈ recur |]
      ==> ∀A A' N. A ∉ bad & A' ∉ bad -->
          Crypt (shrK A) {|Key K, Agent A', N|} ∈ parts{RB} -->
          Key K ∉ analz (insert RB (spies evs))"
```

⟨*proof*⟩

lemma *Spy_not_see_session_key:*
```
"[| Crypt (shrK A) {|Key K, Agent A', N|} ∈ parts (spies evs);
    A ∉ bad;  A' ∉ bad;  evs ∈ recur |]
 ==> Key K ∉ analz (spies evs)"
```
⟨*proof*⟩

The response never contains Hashes

lemma *Hash_in_parts_respond:*
```
"[| Hash {|Key (shrK B), M|} ∈ parts (insert RB H);
    (PB,RB,K) ∈ respond evs |]
 ==> Hash {|Key (shrK B), M|} ∈ parts H"
```
⟨*proof*⟩

Only RA1 or RA2 can have caused such a part of a message to appear. This result is of no use to B, who cannot verify the Hash. Moreover, it can say nothing about how recent A's message is. It might later be used to prove B's presence to A at the run's conclusion.

lemma *Hash_auth_sender [rule_format]:*
```
"[| Hash {|Key(shrK A), Agent A, Agent B, NA, P|} ∈ parts(spies evs);
    A ∉ bad;  evs ∈ recur |]
 ==> Says A B (Hash[Key(shrK A)] {|Agent A, Agent B, NA, P|}) ∈ set evs"
```
⟨*proof*⟩

Certificates can only originate with the Server.

lemma *Cert_imp_Server_msg:*
```
"[| Crypt (shrK A) Y ∈ parts (spies evs);
    A ∉ bad;  evs ∈ recur |]
 ==> ∃C RC. Says Server C RC ∈ set evs  &
              Crypt (shrK A) Y ∈ parts {RC}"
```
⟨*proof*⟩

end

11 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.

consts *yahalom :: "event list set"*
inductive *"yahalom"*
intros

Nil: "[] ∈ yahalom"

Fake: "[| evsf ∈ yahalom; X ∈ synth (analz (knows Spy evsf)) |]

```
        ==> Says Spy B X  # evsf ∈ yahalom"


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


  YM1:  "[| evs1 ∈ yahalom;  Nonce NA ∉ used evs1 |]
         ==> Says A B {|Agent A, Nonce NA|} # evs1 ∈ yahalom"


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


  YM3:  "[| evs3 ∈ yahalom;  Key KAB ∉ used evs3;  KAB ∈ symKeys;
            Gets Server
                 {|Agent B, Crypt (shrK B) {|Agent A, Nonce NA, Nonce NB|}|}
              ∈ set evs3 |]
         ==> Says Server A
                  {|Crypt (shrK A) {|Agent B, Key KAB, Nonce NA, Nonce NB|},
                    Crypt (shrK B) {|Agent A, Key KAB|}|}
               # evs3 ∈ yahalom"

  YM4:
```

— Alice receives the Server's (?) message, checks her Nonce, and uses the new session key to send Bob his Nonce. The premise *A ≠ 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"


  Oops: "[| evso ∈ yahalom;
            Says Server A {|Crypt (shrK A)
                                  {|Agent B, Key K, Nonce NA, Nonce NB|},
                            X|}  ∈ set evso |]
         ==> Notes Spy {|Nonce NA, Nonce NB, Key K|} # evso ∈ yahalom"
```

constdefs

```
  KeyWithNonce :: "[key, nat, event list] => bool"
  "KeyWithNonce K NB evs ==
     ∃A B na X.
       Says Server A {|Crypt (shrK A) {|Agent B, Key K, na, Nonce NB|}, X|}

         ∈ set evs"
```

declare *Says_imp_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; K ∈ symKeys; Key K ∉ used [] |]*
==> ∃X NB. ∃evs ∈ yahalom.
Says A B {|X, Crypt K (Nonce NB)|} ∈ set evs"
⟨*proof*⟩

11.1 Regularity Lemmas for Yahalom

lemma *Gets_imp_Says:*
"[| Gets B X ∈ set evs; evs ∈ yahalom |] ==> ∃A. Says A B X ∈ set evs"
⟨*proof*⟩

Must be proved separately for each protocol

lemma *Gets_imp_knows_Spy:*
"[| Gets B X ∈ set evs; evs ∈ yahalom |] ==> X ∈ knows Spy evs"
⟨*proof*⟩

declare *Gets_imp_knows_Spy [THEN analz.Inj, dest]*

Lets us treat YM4 using a similar argument as for the Fake case.

lemma *YM4_analz_knows_Spy:*
"[| Gets A {|Crypt (shrK A) Y, X|} ∈ set evs; evs ∈ yahalom |]
==> X ∈ analz (knows Spy evs)"
⟨*proof*⟩

lemmas *YM4_parts_knows_Spy =*
YM4_analz_knows_Spy [THEN analz_into_parts, standard]

For Oops

lemma *YM4_Key_parts_knows_Spy:*
"Says Server A {|Crypt (shrK A) {|B,K,NA,NB|}, X|} ∈ set evs
==> K ∈ parts (knows Spy evs)"
⟨*proof*⟩

Theorems of the form *X ∉ parts (knows Spy evs)* imply that NOBODY sends messages containing X!

Spy never sees a good agent's shared key!

lemma *Spy_see_shrK [simp]:*
"evs ∈ yahalom ==> (Key (shrK A) ∈ parts (knows Spy evs)) = (A ∈ bad)"
⟨*proof*⟩

lemma *Spy_analz_shrK [simp]:*
"evs ∈ yahalom ==> (Key (shrK A) ∈ analz (knows Spy evs)) = (A ∈ bad)"
⟨*proof*⟩

lemma *Spy_see_shrK_D [dest!]:*

```
     "[|Key (shrK A) ∈ parts (knows Spy evs);  evs ∈ yahalom|] ==> A ∈ bad"
```
⟨*proof*⟩

Nobody can have used non-existent keys! Needed to apply `analz_insert_Key`

lemma *new_keys_not_used [simp]:*
```
    "[|Key K ∉ used evs; K ∈ symKeys; evs ∈ yahalom|]
     ==> K ∉ keysFor (parts (spies evs))"
```
⟨*proof*⟩

Earlier, all protocol proofs declared this theorem. But only a few proofs need it, e.g. Yahalom and Kerberos IV.

lemma *new_keys_not_analzd:*
```
 "[|K ∈ symKeys; evs ∈ yahalom; Key K ∉ used evs|]
  ==> K ∉ keysFor (analz (knows Spy evs))"
```
⟨*proof*⟩

Describes the form of K when the Server sends this message. Useful for Oops as well as main secrecy property.

lemma *Says_Server_not_range [simp]:*
```
     "[| Says Server A {|Crypt (shrK A) {|Agent B, Key K, na, nb|}, X|}
           ∈ set evs;   evs ∈ yahalom |]
      ==> K ∉ range shrK"
```
⟨*proof*⟩

11.2 Secrecy Theorems

Session keys are not used to encrypt other session keys

lemma *analz_image_freshK [rule_format]:*
```
 "evs ∈ yahalom ==>
   ∀K KK. KK <= - (range shrK) -->
          (Key K ∈ analz (Key‘KK Un (knows Spy evs))) =
          (K ∈ KK | Key K ∈ analz (knows Spy evs))"
```
⟨*proof*⟩

lemma *analz_insert_freshK:*
```
     "[| evs ∈ yahalom;  KAB ∉ range shrK |] ==>
      (Key K ∈ analz (insert (Key KAB) (knows Spy evs))) =
      (K = KAB | Key K ∈ analz (knows Spy evs))"
```
⟨*proof*⟩

The Key K uniquely identifies the Server's message.

lemma *unique_session_keys:*
```
     "[| Says Server A
          {|Crypt (shrK A) {|Agent B, Key K, na, nb|}, X|} ∈ set evs;
         Says Server A'
          {|Crypt (shrK A') {|Agent B', Key K, na', nb'|}, X'|} ∈ set evs;
         evs ∈ yahalom |]
     ==> A=A' & B=B' & na=na' & nb=nb'"
```
⟨*proof*⟩

Crucial secrecy property: Spy does not see the keys sent in msg YM3

lemma *secrecy_lemma:*

```
"[| A ∉ bad;  B ∉ bad;  evs ∈ yahalom |]
 ==> Says Server A
       {|Crypt (shrK A) {|Agent B, Key K, na, nb|},
         Crypt (shrK B) {|Agent A, Key K|}|}
      ∈ set evs -->
     Notes Spy {|na, nb, Key K|} ∉ set evs -->
     Key K ∉ analz (knows Spy evs)"
```

⟨*proof*⟩

Final version

lemma *Spy_not_see_encrypted_key:*

```
"[| Says Server A
       {|Crypt (shrK A) {|Agent B, Key K, na, nb|},
         Crypt (shrK B) {|Agent A, Key K|}|}
      ∈ set evs;
    Notes Spy {|na, nb, Key K|} ∉ set evs;
    A ∉ bad;  B ∉ bad;  evs ∈ yahalom |]
 ==> Key K ∉ analz (knows Spy evs)"
```

⟨*proof*⟩

11.2.1 Security Guarantee for A upon receiving YM3

If the encrypted message appears then it originated with the Server

lemma *A_trusts_YM3:*

```
"[| Crypt (shrK A) {|Agent B, Key K, na, nb|} ∈ parts (knows Spy evs);
    A ∉ bad;  evs ∈ yahalom |]
  ==> Says Server A
       {|Crypt (shrK A) {|Agent B, Key K, na, nb|},
         Crypt (shrK B) {|Agent A, Key K|}|}
      ∈ set evs"
```

⟨*proof*⟩

The obvious combination of *A_trusts_YM3* with *Spy_not_see_encrypted_key*

lemma *A_gets_good_key:*

```
"[| Crypt (shrK A) {|Agent B, Key K, na, nb|} ∈ parts (knows Spy evs);
    Notes Spy {|na, nb, Key K|} ∉ set evs;
    A ∉ bad;  B ∉ bad;  evs ∈ yahalom |]
 ==> Key K ∉ analz (knows Spy evs)"
```

⟨*proof*⟩

11.2.2 Security Guarantees for B upon receiving YM4

B knows, by the first part of A's message, that the Server distributed the key for A and B. But this part says nothing about nonces.

lemma *B_trusts_YM4_shrK:*

```
"[| Crypt (shrK B) {|Agent A, Key K|} ∈ parts (knows Spy evs);
    B ∉ bad;  evs ∈ yahalom |]
 ==> ∃NA NB. Says Server A
                {|Crypt (shrK A) {|Agent B, Key K,
                                   Nonce NA, Nonce NB|},
                  Crypt (shrK B) {|Agent A, Key K|}|}
               ∈ set evs"
```

⟨*proof*⟩

B knows, by the second part of A's message, that the Server distributed the key quoting nonce NB. This part says nothing about agent names. Secrecy of NB is crucial. Note that *Nonce NB ∉ analz (knows Spy evs)* must be the FIRST antecedent of the induction formula.

lemma *B_trusts_YM4_newK [rule_format]:*
```
"[|Crypt K (Nonce NB) ∈ parts (knows Spy evs);
   Nonce NB ∉ analz (knows Spy evs);  evs ∈ yahalom|]
 ==> ∃A B NA. Says Server A
                {|Crypt (shrK A) {|Agent B, Key K, Nonce NA, Nonce NB|},
                  Crypt (shrK B) {|Agent A, Key K|}|}
               ∈ set evs"
```
⟨*proof*⟩

11.2.3 Towards proving secrecy of Nonce NB

Lemmas about the predicate KeyWithNonce

lemma *KeyWithNonceI:*
```
"Says Server A
        {|Crypt (shrK A) {|Agent B, Key K, na, Nonce NB|}, X|}
      ∈ set evs ==> KeyWithNonce K NB evs"
```
⟨*proof*⟩

lemma *KeyWithNonce_Says [simp]:*
```
"KeyWithNonce K NB (Says S A X # evs) =
   (Server = S &
    (∃B n X'. X = {|Crypt (shrK A) {|Agent B, Key K, n, Nonce NB|}, X'|})
    | KeyWithNonce K NB evs)"
```
⟨*proof*⟩

lemma *KeyWithNonce_Notes [simp]:*
```
"KeyWithNonce K NB (Notes A X # evs) = KeyWithNonce K NB evs"
```
⟨*proof*⟩

lemma *KeyWithNonce_Gets [simp]:*
```
"KeyWithNonce K NB (Gets A X # evs) = KeyWithNonce K NB evs"
```
⟨*proof*⟩

A fresh key cannot be associated with any nonce (with respect to a given trace).

lemma *fresh_not_KeyWithNonce:*
```
"Key K ∉ used evs ==> ~ KeyWithNonce K NB evs"
```
⟨*proof*⟩

The Server message associates K with NB' and therefore not with any other nonce NB.

lemma *Says_Server_KeyWithNonce:*
```
"[| Says Server A {|Crypt (shrK A) {|Agent B, Key K, na, Nonce NB'|}, X|}
      ∈ set evs;
    NB ≠ NB';  evs ∈ yahalom |]
 ==> ~ KeyWithNonce K NB evs"
```

⟨*proof*⟩

The only nonces that can be found with the help of session keys are those distributed as nonce NB by the Server. The form of the theorem recalls `analz_image_freshK`, but it is much more complicated.

As with `analz_image_freshK`, we take some pains to express the property as a logical equivalence so that the simplifier can apply it.

lemma *Nonce_secrecy_lemma:*
```
"P --> (X ∈ analz (G Un H)) --> (X ∈ analz H)  ==>
 P --> (X ∈ analz (G Un H)) = (X ∈ analz H)"
```
⟨*proof*⟩

lemma *Nonce_secrecy:*
```
"evs ∈ yahalom ==>
 (∀KK. KK <= - (range shrK) -->
      (∀K ∈ KK. K ∈ symKeys --> ~ KeyWithNonce K NB evs)   -->
      (Nonce NB ∈ analz (Key‘KK Un (knows Spy evs))) =
      (Nonce NB ∈ analz (knows Spy evs)))"
```
⟨*proof*⟩

Version required below: if NB can be decrypted using a session key then it was distributed with that key. The more general form above is required for the induction to carry through.

lemma *single_Nonce_secrecy:*
```
"[| Says Server A
     {|Crypt (shrK A) {|Agent B, Key KAB, na, Nonce NB'|}, X|}
    ∈ set evs;
    NB ≠ NB';  KAB ∉ range shrK;  evs ∈ yahalom |]
 ==> (Nonce NB ∈ analz (insert (Key KAB) (knows Spy evs))) =
     (Nonce NB ∈ analz (knows Spy evs))"
```
⟨*proof*⟩

11.2.4 The Nonce NB uniquely identifies B's message.

lemma *unique_NB:*
```
"[| Crypt (shrK B) {|Agent A, Nonce NA, nb|} ∈ parts (knows Spy evs);
    Crypt (shrK B') {|Agent A', Nonce NA', nb|} ∈ parts (knows Spy evs);
   evs ∈ yahalom;  B ∉ bad;  B' ∉ bad |]
 ==> NA' = NA & A' = A & B' = B"
```
⟨*proof*⟩

Variant useful for proving secrecy of NB. Because nb is assumed to be secret, we no longer must assume B, B' not bad.

lemma *Says_unique_NB:*
```
"[| Says C S   {|X,  Crypt (shrK B) {|Agent A, Nonce NA, nb|}|}
      ∈ set evs;
    Gets S' {|X', Crypt (shrK B') {|Agent A', Nonce NA', nb|}|}
      ∈ set evs;
    nb ∉ analz (knows Spy evs);  evs ∈ yahalom |]
 ==> NA' = NA & A' = A & B' = B"
```
⟨*proof*⟩

11.2.5 A nonce value is never used both as NA and as NB

lemma *no_nonce_YM1_YM2:*

```
"[|Crypt (shrK B') {|Agent A', Nonce NB, nb'|} ∈ parts(knows Spy evs);
   Nonce NB ∉ analz (knows Spy evs);  evs ∈ yahalom|]
 ==> Crypt (shrK B)  {|Agent A, na, Nonce NB|} ∉ parts(knows Spy evs)"
```

⟨*proof*⟩

The Server sends YM3 only in response to YM2.

lemma *Says_Server_imp_YM2:*

```
"[| Says Server A {|Crypt (shrK A) {|Agent B, k, na, nb|}, X|} ∈ set
evs;
    evs ∈ yahalom |]
 ==> Gets Server {| Agent B, Crypt (shrK B) {|Agent A, na, nb|} |}
       ∈ set evs"
```

⟨*proof*⟩

A vital theorem for B, that nonce NB remains secure from the Spy.

lemma *Spy_not_see_NB :*

```
"[| Says B Server
          {|Agent B, Crypt (shrK B) {|Agent A, Nonce NA, Nonce NB|}|}
      ∈ set evs;
    (∀k. Notes Spy {|Nonce NA, Nonce NB, k|} ∉ set evs);
    A ∉ bad;  B ∉ bad;  evs ∈ yahalom |]
 ==> Nonce NB ∉ analz (knows Spy evs)"
```

⟨*proof*⟩

B's session key guarantee from YM4. The two certificates contribute to a single conclusion about the Server's message. Note that the "Notes Spy" assumption must quantify over ∀ POSSIBLE keys instead of our particular K. If this run is broken and the spy substitutes a certificate containing an old key, B has no means of telling.

lemma *B_trusts_YM4:*

```
"[| Gets B {|Crypt (shrK B) {|Agent A, Key K|},
             Crypt K (Nonce NB)|} ∈ set evs;
    Says B Server
      {|Agent B, Crypt (shrK B) {|Agent A, Nonce NA, Nonce NB|}|}
      ∈ set evs;
    ∀k. Notes Spy {|Nonce NA, Nonce NB, k|} ∉ set evs;
    A ∉ bad;  B ∉ bad;  evs ∈ yahalom |]
  ==> Says Server A
            {|Crypt (shrK A) {|Agent B, Key K,
                      Nonce NA, Nonce NB|},
              Crypt (shrK B) {|Agent A, Key K|}|}
      ∈ set evs"
```

⟨*proof*⟩

The obvious combination of *B_trusts_YM4* with *Spy_not_see_encrypted_key*

lemma *B_gets_good_key:*

```
"[| Gets B {|Crypt (shrK B) {|Agent A, Key K|},
             Crypt K (Nonce NB)|} ∈ set evs;
    Says B Server
      {|Agent B, Crypt (shrK B) {|Agent A, Nonce NA, Nonce NB|}|}
```

```
          ∈ set evs;
        ∀k. Notes Spy {|Nonce NA, Nonce NB, k|} ∉ set evs;
        A ∉ bad;  B ∉ bad;  evs ∈ yahalom |]
     ==> Key K ∉ analz (knows Spy evs)"
```

⟨*proof*⟩

11.3 Authenticating B to A

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

lemma *B_Said_YM2 [rule_format]:*

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

⟨*proof*⟩

If the server sends YM3 then B sent YM2

lemma *YM3_auth_B_to_A_lemma:*

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

⟨*proof*⟩

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

lemma *YM3_auth_B_to_A:*

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

⟨*proof*⟩

11.4 Authenticating A to B using the certificate `Crypt K (Nonce NB)`

Assuming the session key is secure, if both certificates are present then A has said NB. We can't be sure about the rest of A's message, but only NB matters for freshness.

lemma *A_Said_YM3_lemma [rule_format]:*

```
     "evs ∈ yahalom
      ==> Key K ∉ analz (knows Spy evs) -->
          Crypt K (Nonce NB) ∈ parts (knows Spy evs) -->
          Crypt (shrK B) {|Agent A, Key K|} ∈ parts (knows Spy evs) -->
          B ∉ bad -->
          (∃X. Says A B {|X, Crypt K (Nonce NB)|} ∈ set evs)"
```

⟨*proof*⟩

If B receives YM4 then A has used nonce NB (and therefore is alive). Moreover, A associates K with NB (thus is talking about the same run). Other premises guarantee secrecy of K.

```
lemma YM4_imp_A_Said_YM3 [rule_format]:
     "[| Gets B {|Crypt (shrK B) {|Agent A, Key K|},
                  Crypt K (Nonce NB)|} ∈ set evs;
         Says B Server
           {|Agent B, Crypt (shrK B) {|Agent A, Nonce NA, Nonce NB|}|}
           ∈ set evs;
         (∀NA k. Notes Spy {|Nonce NA, Nonce NB, k|} ∉ set evs);
         A ∉ bad;  B ∉ bad;  evs ∈ yahalom |]
      ==> ∃X. Says A B {|X, Crypt K (Nonce NB)|} ∈ set evs"
```
⟨*proof*⟩

end

12 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.

```
consts  yahalom   :: "event list set"
inductive "yahalom"
  intros

   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 ∉ used []*
*   ==> ∃X NB. ∃evs ∈ yahalom.*
*   Says A B {|X, Crypt K (Nonce NB)|} ∈ set evs"*
⟨*proof*⟩

lemma *Gets_imp_Says:*
*   "[| Gets B X ∈ set evs; evs ∈ yahalom |] ==> ∃A. Says A B X ∈ set evs"*
⟨*proof*⟩

Must be proved separately for each protocol

lemma *Gets_imp_knows_Spy:*
*   "[| Gets B X ∈ set evs; evs ∈ yahalom |] ==> X ∈ knows Spy evs"*
⟨*proof*⟩

declare *Gets_imp_knows_Spy [THEN analz.Inj, dest]*

12.1 Inductive Proofs

Result for reasoning about the encrypted portion of messages. Lets us treat YM4 using a similar argument as for the Fake case.

lemma *YM4_analz_knows_Spy:*
*   "[| Gets A {|NB, Crypt (shrK A) Y, X|} ∈ set evs; evs ∈ yahalom |]*
*   ==> X ∈ analz (knows Spy evs)"*
⟨*proof*⟩

lemmas *YM4_parts_knows_Spy =*
YM4_analz_knows_Spy [THEN analz_into_parts, standard]

Spy never sees a good agent's shared key!

lemma *Spy_see_shrK [simp]:*
"evs ∈ yahalom ==> (Key (shrK A) ∈ parts (knows Spy evs)) = (A ∈ bad)"
⟨*proof*⟩

lemma *Spy_analz_shrK [simp]:*
"evs ∈ yahalom ==> (Key (shrK A) ∈ analz (knows Spy evs)) = (A ∈ bad)"
⟨*proof*⟩

lemma *Spy_see_shrK_D [dest!]:*
"[|Key (shrK A) ∈ parts (knows Spy evs); evs ∈ yahalom|] ==> A ∈ bad"
⟨*proof*⟩

Nobody can have used non-existent keys! Needed to apply *analz_insert_Key*

lemma *new_keys_not_used [simp]:*
"[|Key K ∉ used evs; K ∈ symKeys; evs ∈ yahalom|]
==> K ∉ keysFor (parts (spies evs))"
⟨*proof*⟩

Describes the form of K when the Server sends this message. Useful for Oops as well as main secrecy property.

lemma *Says_Server_message_form:*
"[| Says Server A {|nb', Crypt (shrK A) {|Agent B, Key K, na|}, X|}
∈ set evs; evs ∈ yahalom |]
==> K ∉ range shrK"
⟨*proof*⟩

lemma *analz_image_freshK [rule_format]:*
"evs ∈ yahalom ==>
∀K KK. KK <= - (range shrK) -->
(Key K ∈ analz (Key'KK Un (knows Spy evs))) =
(K ∈ KK | Key K ∈ analz (knows Spy evs))"
⟨*proof*⟩

lemma *analz_insert_freshK:*
"[| evs ∈ yahalom; KAB ∉ range shrK |] ==>
(Key K ∈ analz (insert (Key KAB) (knows Spy evs))) =
(K = KAB | Key K ∈ analz (knows Spy evs))"
⟨*proof*⟩

The Key K uniquely identifies the Server's message

lemma *unique_session_keys:*
"[| Says Server A
{|nb, Crypt (shrK A) {|Agent B, Key K, na|}, X|} ∈ set evs;
Says Server A'

```
        {|nb', Crypt (shrK A') {|Agent B', Key K, na'|}, X'|} ∈ set evs;
       evs ∈ yahalom |]
    ==> A=A' & B=B' & na=na' & nb=nb'"
```
⟨*proof*⟩

12.2 Crucial Secrecy Property: Spy Does Not See Key KAB

lemma *secrecy_lemma:*
```
    "[| A ∉ bad;  B ∉ bad;  evs ∈ yahalom |]
     ==> Says Server A
           {|nb, Crypt (shrK A) {|Agent B, Key K, na|},
                 Crypt (shrK B) {|Agent A, Agent B, Key K, nb|}|}
          ∈ set evs -->
         Notes Spy {|na, nb, Key K|} ∉ set evs -->
         Key K ∉ analz (knows Spy evs)"
```
⟨*proof*⟩

Final version

lemma *Spy_not_see_encrypted_key:*
```
    "[| Says Server A
           {|nb, Crypt (shrK A) {|Agent B, Key K, na|},
                 Crypt (shrK B) {|Agent A, Agent B, Key K, nb|}|}
       ∈ set evs;
       Notes Spy {|na, nb, Key K|} ∉ set evs;
       A ∉ bad;  B ∉ bad;  evs ∈ yahalom |]
     ==> Key K ∉ analz (knows Spy evs)"
```
⟨*proof*⟩

This form is an immediate consequence of the previous result. It is similar to the assertions established by other methods. It is equivalent to the previous result in that the Spy already has *analz* and *synth* at his disposal. However, the conclusion *Key K ∉ knows Spy evs* appears not to be inductive: all the cases other than Fake are trivial, while Fake requires *Key K ∉ analz (knows Spy evs)*.

lemma *Spy_not_know_encrypted_key:*
```
    "[| Says Server A
           {|nb, Crypt (shrK A) {|Agent B, Key K, na|},
                 Crypt (shrK B) {|Agent A, Agent B, Key K, nb|}|}
       ∈ set evs;
       Notes Spy {|na, nb, Key K|} ∉ set evs;
       A ∉ bad;  B ∉ bad;  evs ∈ yahalom |]
     ==> Key K ∉ knows Spy evs"
```
⟨*proof*⟩

12.3 Security Guarantee for A upon receiving YM3

If the encrypted message appears then it originated with the Server. May now apply *Spy_not_see_encrypted_key*, subject to its conditions.

lemma *A_trusts_YM3:*
```
    "[| Crypt (shrK A) {|Agent B, Key K, na|} ∈ parts (knows Spy evs);
       A ∉ bad;  evs ∈ yahalom |]
     ==> ∃nb. Says Server A
                  {|nb, Crypt (shrK A) {|Agent B, Key K, na|},
```

```
                        Crypt (shrK B) {|Agent A, Agent B, Key K, nb|}|}
                  ∈ set evs"
```
⟨*proof*⟩

The obvious combination of *A_trusts_YM3* with *Spy_not_see_encrypted_key*

theorem *A_gets_good_key:*
```
     "[| Crypt (shrK A) {|Agent B, Key K, na|} ∈ parts (knows Spy evs);
         ∀nb. Notes Spy {|na, nb, Key K|} ∉ set evs;
         A ∉ bad;  B ∉ bad;  evs ∈ yahalom |]
      ==> Key K ∉ analz (knows Spy evs)"
```
⟨*proof*⟩

12.4 Security Guarantee for B upon receiving YM4

B knows, by the first part of A's message, that the Server distributed the key for A and B, and has associated it with NB.

lemma *B_trusts_YM4_shrK:*
```
     "[| Crypt (shrK B) {|Agent A, Agent B, Key K, Nonce NB|}
           ∈ parts (knows Spy evs);
         B ∉ bad;  evs ∈ yahalom |]
  ==> ∃NA. Says Server A
            {|Nonce NB,
              Crypt (shrK A) {|Agent B, Key K, Nonce NA|},
              Crypt (shrK B) {|Agent A, Agent B, Key K, Nonce NB|}|}
            ∈ set evs"
```
⟨*proof*⟩

With this protocol variant, we don't need the 2nd part of YM4 at all: Nonce NB is available in the first part.

What can B deduce from receipt of YM4? Stronger and simpler than Yahalom because we do not have to show that NB is secret.

lemma *B_trusts_YM4:*
```
     "[| Gets B {|Crypt (shrK B) {|Agent A, Agent B, Key K, Nonce NB|},  X|}
           ∈ set evs;
         A ∉ bad;  B ∉ bad;  evs ∈ yahalom |]
  ==> ∃NA. Says Server A
            {|Nonce NB,
              Crypt (shrK A) {|Agent B, Key K, Nonce NA|},
              Crypt (shrK B) {|Agent A, Agent B, Key K, Nonce NB|}|}
           ∈ set evs"
```
⟨*proof*⟩

The obvious combination of *B_trusts_YM4* with *Spy_not_see_encrypted_key*

theorem *B_gets_good_key:*
```
     "[| Gets B {|Crypt (shrK B) {|Agent A, Agent B, Key K, Nonce NB|}, X|}
           ∈ set evs;
         ∀na. Notes Spy {|na, Nonce NB, Key K|} ∉ set evs;
         A ∉ bad;  B ∉ bad;  evs ∈ yahalom |]
      ==> Key K ∉ analz (knows Spy evs)"
```
⟨*proof*⟩

12.5 Authenticating B to A

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

lemma *B_Said_YM2:*
```
"[| Crypt (shrK B) {|Agent A, Nonce NA|} ∈ parts (knows Spy evs);
    B ∉ bad;  evs ∈ yahalom |]
 ==> ∃NB. Says B Server {|Agent B, Nonce NB,
                          Crypt (shrK B) {|Agent A, Nonce NA|}|}
                ∈ set evs"
```
⟨*proof*⟩

If the server sends YM3 then B sent YM2, perhaps with a different NB

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

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

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

12.6 Authenticating A to B

using the certificate `Crypt K (Nonce NB)`

Assuming the session key is secure, if both certificates are present then A has said NB. We can't be sure about the rest of A's message, but only NB matters for freshness. Note that `Key K ∉ analz (knows Spy evs)` must be the FIRST antecedent of the induction formula.

This lemma allows a use of `unique_session_keys` in the next proof, which otherwise is extremely slow.

lemma *secure_unique_session_keys:*
```
"[| Crypt (shrK A) {|Agent B, Key K, na|} ∈ analz (spies evs);
    Crypt (shrK A') {|Agent B', Key K, na'|} ∈ analz (spies evs);
    Key K ∉ analz (knows Spy evs);  evs ∈ yahalom |]
 ==> A=A' & B=B'"
```
⟨*proof*⟩

lemma *Auth_A_to_B_lemma [rule_format]:*
```
"evs ∈ yahalom
```

```
      ==> Key K ∉ analz (knows Spy evs) -->
          K ∈ symKeys -->
          Crypt K (Nonce NB) ∈ parts (knows Spy evs) -->
          Crypt (shrK B) {|Agent A, Agent B, Key K, Nonce NB|}
            ∈ parts (knows Spy evs) -->
          B ∉ bad -->
          (∃X. Says A B {|X, Crypt K (Nonce NB)|} ∈ set evs)"
```

⟨*proof*⟩

If B receives YM4 then A has used nonce NB (and therefore is alive). Moreover, A associates K with NB (thus is talking about the same run). Other premises guarantee secrecy of K.

```
theorem YM4_imp_A_Said_YM3 [rule_format]:
     "[| Gets B {|Crypt (shrK B) {|Agent A, Agent B, Key K, Nonce NB|},
                 Crypt K (Nonce NB)|} ∈ set evs;
         (∀NA. Notes Spy {|Nonce NA, Nonce NB, Key K|} ∉ set evs);
         K ∈ symKeys;  A ∉ bad;  B ∉ bad;  evs ∈ yahalom |]
      ==> ∃X. Says A B {|X, Crypt K (Nonce NB)|} ∈ set evs"
```

⟨*proof*⟩

end

13 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.

```
consts  yahalom   :: "event list set"
inductive "yahalom"
  intros

   Nil:  "[] ∈ yahalom"


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


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


   YM1:  "[| evs1 ∈ yahalom;  Nonce NA ∉ used evs1 |]
          ==> Says A B {|Agent A, Nonce NA|} # evs1 ∈ yahalom"


   YM2:  "[| evs2 ∈ yahalom;  Nonce NB ∉ used evs2;
              Gets B {|Agent A, Nonce NA|} ∈ set evs2 |]
          ==> Says B Server
                  {|Agent B, Nonce NB, Crypt (shrK B) {|Agent A, Nonce NA|}|}
```

```
              # evs2 ∈ yahalom"


   YM3:  "[| evs3 ∈ yahalom;  Key KAB ∉ used evs3;  KAB ∈ symKeys;
             Gets Server
                  {|Agent B, Nonce NB, Crypt (shrK B) {|Agent A, Nonce NA|}|}
               ∈ set evs3 |]
          ==> Says Server A
                   {|Crypt (shrK A) {|Agent B, Key KAB, Nonce NA, Nonce NB|},
                     Crypt (shrK B) {|Agent A, Key KAB|}|}
               # evs3 ∈ yahalom"


   YM4:  "[| evs4 ∈ yahalom;  A ≠ Server;  K ∈ symKeys;
             Gets A {|Crypt(shrK A) {|Agent B, Key K, Nonce NA, Nonce NB|},
X|}
                ∈ set evs4;
             Says A B {|Agent A, Nonce NA|} ∈ set evs4 |]
          ==> Says A B {|X, Crypt K (Nonce NB)|} # evs4 ∈ yahalom"


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

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

```
lemma "[| A ≠ Server; Key K ∉ used []; K ∈ symKeys |]
       ==> ∃X NB. ∃evs ∈ yahalom.
             Says A B {|X, Crypt K (Nonce NB)|} ∈ set evs"
```
⟨*proof*⟩

13.1 Regularity Lemmas for Yahalom

```
lemma Gets_imp_Says:
     "[| Gets B X ∈ set evs; evs ∈ yahalom |] ==> ∃A. Says A B X ∈ set evs"
```
⟨*proof*⟩

```
lemma Gets_imp_knows_Spy:
     "[| Gets B X ∈ set evs; evs ∈ yahalom |]  ==> X ∈ knows Spy evs"
```
⟨*proof*⟩

```
declare Gets_imp_knows_Spy [THEN analz.Inj, dest]
```

13.2 For reasoning about the encrypted portion of messages

Lets us treat YM4 using a similar argument as for the Fake case.

```
lemma YM4_analz_knows_Spy:
     "[| Gets A {|Crypt (shrK A) Y, X|} ∈ set evs;  evs ∈ yahalom |]
      ==> X ∈ analz (knows Spy evs)"
```
⟨*proof*⟩

lemmas *YM4_parts_knows_Spy =*
YM4_analz_knows_Spy [THEN analz_into_parts, standard]

Theorems of the form *X ∉ parts (knows Spy evs)* imply that NOBODY sends messages containing X!

Spy never sees a good agent's shared key!

lemma *Spy_see_shrK [simp]:*
"evs ∈ yahalom ==> (Key (shrK A) ∈ parts (knows Spy evs)) = (A ∈ bad)"
⟨*proof*⟩

lemma *Spy_analz_shrK [simp]:*
"evs ∈ yahalom ==> (Key (shrK A) ∈ analz (knows Spy evs)) = (A ∈ bad)"
⟨*proof*⟩

lemma *Spy_see_shrK_D [dest!]:*
"[|Key (shrK A) ∈ parts (knows Spy evs); evs ∈ yahalom|] ==> A ∈ bad"
⟨*proof*⟩

Nobody can have used non-existent keys! Needed to apply *analz_insert_Key*

lemma *new_keys_not_used [simp]:*
"[|Key K ∉ used evs; K ∈ symKeys; evs ∈ yahalom|]
==> K ∉ keysFor (parts (spies evs))"
⟨*proof*⟩

13.3 Secrecy Theorems

13.4 Session keys are not used to encrypt other session keys

lemma *analz_image_freshK [rule_format]:*
"evs ∈ yahalom ==>
∀K KK. KK <= - (range shrK) -->
(Key K ∈ analz (Key‘KK Un (knows Spy evs))) =
(K ∈ KK | Key K ∈ analz (knows Spy evs))"
⟨*proof*⟩

lemma *analz_insert_freshK:*
"[| evs ∈ yahalom; KAB ∉ range shrK |] ==>
(Key K ∈ analz (insert (Key KAB) (knows Spy evs))) =
(K = KAB | Key K ∈ analz (knows Spy evs))"
⟨*proof*⟩

The Key K uniquely identifies the Server's message.

lemma *unique_session_keys:*
"[| Says Server A
{|Crypt (shrK A) {|Agent B, Key K, na, nb|}, X|} ∈ set evs;
Says Server A'
{|Crypt (shrK A') {|Agent B', Key K, na', nb'|}, X'|} ∈ set evs;
evs ∈ yahalom |]
==> A=A' & B=B' & na=na' & nb=nb'"
⟨*proof*⟩

Crucial secrecy property: Spy does not see the keys sent in msg YM3

lemma *secrecy_lemma:*
```
"[| A ∉ bad;  B ∉ bad;  evs ∈ yahalom |]
 ==> Says Server A
       {|Crypt (shrK A) {|Agent B, Key K, na, nb|},
         Crypt (shrK B) {|Agent A, Key K|}|}
      ∈ set evs -->
     Key K ∉ analz (knows Spy evs)"
```
⟨*proof*⟩

Final version

lemma *Spy_not_see_encrypted_key:*
```
"[| Says Server A
       {|Crypt (shrK A) {|Agent B, Key K, na, nb|},
         Crypt (shrK B) {|Agent A, Key K|}|}
      ∈ set evs;
    A ∉ bad;  B ∉ bad;  evs ∈ yahalom |]
 ==> Key K ∉ analz (knows Spy evs)"
```
⟨*proof*⟩

13.5 Security Guarantee for A upon receiving YM3

If the encrypted message appears then it originated with the Server

lemma *A_trusts_YM3:*
```
"[| Crypt (shrK A) {|Agent B, Key K, na, nb|} ∈ parts (knows Spy evs);
    A ∉ bad;  evs ∈ yahalom |]
  ==> Says Server A
       {|Crypt (shrK A) {|Agent B, Key K, na, nb|},
         Crypt (shrK B) {|Agent A, Key K|}|}
      ∈ set evs"
```
⟨*proof*⟩

The obvious combination of *A_trusts_YM3* with *Spy_not_see_encrypted_key*

lemma *A_gets_good_key:*
```
"[| Crypt (shrK A) {|Agent B, Key K, na, nb|} ∈ parts (knows Spy evs);
    A ∉ bad;  B ∉ bad;  evs ∈ yahalom |]
 ==> Key K ∉ analz (knows Spy evs)"
```
⟨*proof*⟩

13.6 Security Guarantees for B upon receiving YM4

B knows, by the first part of A's message, that the Server distributed the key for A and B. But this part says nothing about nonces.

lemma *B_trusts_YM4_shrK:*
```
"[| Crypt (shrK B) {|Agent A, Key K|} ∈ parts (knows Spy evs);
    B ∉ bad;  evs ∈ yahalom |]
 ==> ∃NA NB. Says Server A
                {|Crypt (shrK A) {|Agent B, Key K, Nonce NA, Nonce NB|},
                  Crypt (shrK B) {|Agent A, Key K|}|}
               ∈ set evs"
```
⟨*proof*⟩

13.7 The Flaw in the Model

Up to now, the reasoning is similar to standard Yahalom. Now the doubtful reasoning occurs. We should not be assuming that an unknown key is secure, but the model allows us to: there is no Oops rule to let session keys become compromised.

B knows, by the second part of A's message, that the Server distributed the key quoting nonce NB. This part says nothing about agent names. Secrecy of K is assumed; the valid Yahalom proof uses (and later proves) the secrecy of NB.

lemma *B_trusts_YM4_newK [rule_format]:*
```
"[|Key K ∉ analz (knows Spy evs);  evs ∈ yahalom|]
 ==> Crypt K (Nonce NB) ∈ parts (knows Spy evs) -->
     (∃A B NA. Says Server A
                 {|Crypt (shrK A) {|Agent B, Key K,
                                    Nonce NA, Nonce NB|},
                   Crypt (shrK B) {|Agent A, Key K|}|}
                ∈ set evs)"
```
⟨*proof*⟩

B's session key guarantee from YM4. The two certificates contribute to a single conclusion about the Server's message.

lemma *B_trusts_YM4:*
```
"[| Gets B {|Crypt (shrK B) {|Agent A, Key K|},
             Crypt K (Nonce NB)|} ∈ set evs;
    Says B Server
      {|Agent B, Nonce NB, Crypt (shrK B) {|Agent A, Nonce NA|}|}
      ∈ set evs;
    A ∉ bad;  B ∉ bad;  evs ∈ yahalom |]
  ==> ∃na nb. Says Server A
                {|Crypt (shrK A) {|Agent B, Key K, na, nb|},
                  Crypt (shrK B) {|Agent A, Key K|}|}
        ∈ set evs"
```
⟨*proof*⟩

The obvious combination of *B_trusts_YM4* with *Spy_not_see_encrypted_key*

lemma *B_gets_good_key:*
```
"[| Gets B {|Crypt (shrK B) {|Agent A, Key K|},
             Crypt K (Nonce NB)|} ∈ set evs;
    Says B Server
      {|Agent B, Nonce NB, Crypt (shrK B) {|Agent A, Nonce NA|}|}
      ∈ set evs;
    A ∉ bad;  B ∉ bad;  evs ∈ yahalom |]
 ==> Key K ∉ analz (knows Spy evs)"
```
⟨*proof*⟩

Assuming the session key is secure, if both certificates are present then A has said NB. We can't be sure about the rest of A's message, but only NB matters for freshness.

lemma *A_Said_YM3_lemma [rule_format]:*
```
"evs ∈ yahalom
 ==> Key K ∉ analz (knows Spy evs) -->
```

```
          Crypt K (Nonce NB) ∈ parts (knows Spy evs) -->
          Crypt (shrK B) {|Agent A, Key K|} ∈ parts (knows Spy evs) -->
          B ∉ bad -->
          (∃X. Says A B {|X, Crypt K (Nonce NB)|} ∈ set evs)"
```

⟨*proof*⟩

If B receives YM4 then A has used nonce NB (and therefore is alive). Moreover, A associates K with NB (thus is talking about the same run). Other premises guarantee secrecy of K.

lemma *YM4_imp_A_Said_YM3 [rule_format]:*

```
     "[| Gets B {|Crypt (shrK B) {|Agent A, Key K|},
                  Crypt K (Nonce NB)|} ∈ set evs;
         Says B Server
           {|Agent B, Nonce NB, Crypt (shrK B) {|Agent A, Nonce NA|}|}
           ∈ set evs;
         A ∉ bad;  B ∉ bad;  evs ∈ yahalom |]
      ==> ∃X. Says A B {|X, Crypt K (Nonce NB)|} ∈ set evs"
```

⟨*proof*⟩

end

theory *ZhouGollmann* **imports** *Public* **begin**

syntax

```
  TTP :: agent
```

translations

```
  "TTP" == " Server "
```

syntax

```
  f_sub :: nat
  f_nro :: nat
  f_nrr :: nat
  f_con :: nat
```

translations

```
  "f_sub" == " 5 "
  "f_nro" == " 2 "
  "f_nrr" == " 3 "
  "f_con" == " 4 "
```

constdefs

```
  broken :: "agent set"
```

— the compromised honest agents; TTP is included as it's not allowed to use the protocol

```
   "broken == bad - {Spy}"
```

declare *broken_def [simp]*

consts *zg :: "event list set"*

```
inductive zg
  intros

  Nil:  "[] ∈ zg"

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

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


  ZG1: "[| evs1 ∈ zg;  Nonce L ∉ used evs1; C = Crypt K (Number m);
           K ∈ symKeys;
           NRO = Crypt (priK A) {|Number f_nro, Agent B, Nonce L, C|}|]
        ==> Says A B {|Number f_nro, Agent B, Nonce L, C, NRO|} # evs1 ∈ zg"


  ZG2: "[| evs2 ∈ zg;
           Gets B {|Number f_nro, Agent B, Nonce L, C, NRO|} ∈ set evs2;
           NRO = Crypt (priK A) {|Number f_nro, Agent B, Nonce L, C|};
           NRR = Crypt (priK B) {|Number f_nrr, Agent A, Nonce L, C|}|]
        ==> Says B A {|Number f_nrr, Agent A, Nonce L, NRR|} # evs2  ∈  zg"


  ZG3: "[| evs3 ∈ zg; C = Crypt K M; K ∈ symKeys;
           Says A B {|Number f_nro, Agent B, Nonce L, C, NRO|} ∈ set evs3;
           Gets A {|Number f_nrr, Agent A, Nonce L, NRR|} ∈ set evs3;
           NRR = Crypt (priK B) {|Number f_nrr, Agent A, Nonce L, C|};
           sub_K = Crypt (priK A) {|Number f_sub, Agent B, Nonce L, Key K|}|]
        ==> Says A TTP {|Number f_sub, Agent B, Nonce L, Key K, sub_K|}
              # evs3 ∈ zg"


  ZG4: "[| evs4 ∈ zg; K ∈ symKeys;
           Gets TTP {|Number f_sub, Agent B, Nonce L, Key K, sub_K|}
             ∈ set evs4;
           sub_K = Crypt (priK A) {|Number f_sub, Agent B, Nonce L, Key K|};
           con_K = Crypt (priK TTP) {|Number f_con, Agent A, Agent B,
                                      Nonce L, Key K|}|]
        ==> Says TTP Spy con_K
           #
           Notes TTP {|Number f_con, Agent A, Agent B, Nonce L, Key K, con_K|}
           # evs4 ∈ zg"


declare Says_imp_knows_Spy [THEN analz.Inj, dest]
declare Fake_parts_insert_in_Un  [dest]
declare analz_into_parts [dest]

declare symKey_neq_priEK [simp]
declare symKey_neq_priEK [THEN not_sym, simp]
```

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

lemma *"[|A ≠ B; TTP ≠ A; TTP ≠ B; K ∈ symKeys|] ==>*
∃L. ∃evs ∈ zg.
Notes TTP {|Number f_con, Agent A, Agent B, Nonce L, Key K,
Crypt (priK TTP) {|Number f_con, Agent A, Agent B, Nonce L,
Key K|} |}
∈ set evs"
⟨*proof*⟩

13.8 Basic Lemmas

lemma *Gets_imp_Says:*
"[| Gets B X ∈ set evs; evs ∈ zg |] ==> ∃A. Says A B X ∈ set evs"
⟨*proof*⟩

lemma *Gets_imp_knows_Spy:*
"[| Gets B X ∈ set evs; evs ∈ zg |] ==> X ∈ spies evs"
⟨*proof*⟩

Lets us replace proofs about *used evs* by simpler proofs about *parts (knows Spy evs)*.

lemma *Crypt_used_imp_spies:*
"[| Crypt K X ∈ used evs; evs ∈ zg |]
==> Crypt K X ∈ parts (spies evs)"
⟨*proof*⟩

lemma *Notes_TTP_imp_Gets:*
"[|Notes TTP {|Number f_con, Agent A, Agent B, Nonce L, Key K, con_K |}
∈ set evs;
sub_K = Crypt (priK A) {|Number f_sub, Agent B, Nonce L, Key K|};
evs ∈ zg|]
==> Gets TTP {|Number f_sub, Agent B, Nonce L, Key K, sub_K|} ∈ set evs"
⟨*proof*⟩

For reasoning about C, which is encrypted in message ZG2

lemma *ZG2_msg_in_parts_spies:*
"[|Gets B {|F, B', L, C, X|} ∈ set evs; evs ∈ zg|]
==> C ∈ parts (spies evs)"
⟨*proof*⟩

lemma *Spy_see_priK [simp]:*
"evs ∈ zg ==> (Key (priK A) ∈ parts (spies evs)) = (A ∈ bad)"
⟨*proof*⟩

So that blast can use it too

declare *Spy_see_priK [THEN [2] rev_iffD1, dest!]*

lemma *Spy_analz_priK [simp]:*
"evs ∈ zg ==> (Key (priK A) ∈ analz (spies evs)) = (A ∈ bad)"
⟨*proof*⟩

13.9 About NRO: Validity for B

Below we prove that if NRO exists then A definitely sent it, provided A is not broken.

Strong conclusion for a good agent

lemma *NRO_validity_good:*
```
"[|NRO = Crypt (priK A) {|Number f_nro, Agent B, Nonce L, C|};
   NRO ∈ parts (spies evs);
   A ∉ bad;  evs ∈ zg |]
 ==> Says A B {|Number f_nro, Agent B, Nonce L, C, NRO|} ∈ set evs"
```
⟨*proof*⟩

lemma *NRO_sender:*
```
"[|Says A' B {|n, b, l, C, Crypt (priK A) X|} ∈ set evs; evs ∈ zg|]
==> A' ∈ {A,Spy}"
```
⟨*proof*⟩

Holds also for A = Spy!

theorem *NRO_validity:*
```
"[|Gets B {|Number f_nro, Agent B, Nonce L, C, NRO|} ∈ set evs;
   NRO = Crypt (priK A) {|Number f_nro, Agent B, Nonce L, C|};
   A ∉ broken;  evs ∈ zg |]
 ==> Says A B {|Number f_nro, Agent B, Nonce L, C, NRO|} ∈ set evs"
```
⟨*proof*⟩

13.10 About NRR: Validity for A

Below we prove that if NRR exists then B definitely sent it, provided B is not broken.

Strong conclusion for a good agent

lemma *NRR_validity_good:*
```
"[|NRR = Crypt (priK B) {|Number f_nrr, Agent A, Nonce L, C|};
   NRR ∈ parts (spies evs);
   B ∉ bad;  evs ∈ zg |]
 ==> Says B A {|Number f_nrr, Agent A, Nonce L, NRR|} ∈ set evs"
```
⟨*proof*⟩

lemma *NRR_sender:*
```
"[|Says B' A {|n, a, l, Crypt (priK B) X|} ∈ set evs; evs ∈ zg|]
==> B' ∈ {B,Spy}"
```
⟨*proof*⟩

Holds also for B = Spy!

theorem *NRR_validity:*
```
"[|Says B' A {|Number f_nrr, Agent A, Nonce L, NRR|} ∈ set evs;
   NRR = Crypt (priK B) {|Number f_nrr, Agent A, Nonce L, C|};
   B ∉ broken; evs ∈ zg|]
==> Says B A {|Number f_nrr, Agent A, Nonce L, NRR|} ∈ set evs"
```
⟨*proof*⟩

13.11 Proofs About *sub_K*

Below we prove that if *sub_K* exists then *A* definitely sent it, provided *A* is not broken.

Strong conclusion for a good agent

lemma *sub_K_validity_good:*

```
"[|sub_K = Crypt (priK A) {|Number f_sub, Agent B, Nonce L, Key K|};
   sub_K ∈ parts (spies evs);
   A ∉ bad;  evs ∈ zg |]
 ==> Says A TTP {|Number f_sub, Agent B, Nonce L, Key K, sub_K|} ∈ set
evs"
```

⟨*proof*⟩

lemma *sub_K_sender:*

```
"[|Says A' TTP {|n, b, l, k, Crypt (priK A) X|} ∈ set evs;  evs ∈ zg|]
==> A' ∈ {A,Spy}"
```

⟨*proof*⟩

Holds also for *A = Spy*!

theorem *sub_K_validity:*

```
"[|Gets TTP {|Number f_sub, Agent B, Nonce L, Key K, sub_K|} ∈ set evs;
   sub_K = Crypt (priK A) {|Number f_sub, Agent B, Nonce L, Key K|};
   A ∉ broken;  evs ∈ zg |]
 ==> Says A TTP {|Number f_sub, Agent B, Nonce L, Key K, sub_K|} ∈ set
evs"
```

⟨*proof*⟩

13.12 Proofs About *con_K*

Below we prove that if *con_K* exists, then *TTP* has it, and therefore *A* and *B*) can get it too. Moreover, we know that *A* sent *sub_K*

lemma *con_K_validity:*

```
"[|con_K ∈ used evs;
   con_K = Crypt (priK TTP)
             {|Number f_con, Agent A, Agent B, Nonce L, Key K|};
   evs ∈ zg |]
==> Notes TTP {|Number f_con, Agent A, Agent B, Nonce L, Key K, con_K|}
      ∈ set evs"
```

⟨*proof*⟩

If *TTP* holds *con_K* then *A* sent *sub_K*. We assume that *A* is not broken. Importantly, nothing needs to be assumed about the form of *con_K*!

lemma *Notes_TTP_imp_Says_A:*

```
"[|Notes TTP {|Number f_con, Agent A, Agent B, Nonce L, Key K, con_K|}
      ∈ set evs;
   sub_K = Crypt (priK A) {|Number f_sub, Agent B, Nonce L, Key K|};
   A ∉ broken; evs ∈ zg|]
 ==> Says A TTP {|Number f_sub, Agent B, Nonce L, Key K, sub_K|} ∈ set
evs"
```

⟨*proof*⟩

If *con_K* exists, then *A* sent *sub_K*. We again assume that *A* is not broken.

theorem *B_sub_K_validity:*
```
"[|con_K ∈ used evs;
   con_K = Crypt (priK TTP) {|Number f_con, Agent A, Agent B,
                              Nonce L, Key K|};
   sub_K = Crypt (priK A) {|Number f_sub, Agent B, Nonce L, Key K|};
   A ∉ broken; evs ∈ zg|]
 ==> Says A TTP {|Number f_sub, Agent B, Nonce L, Key K, sub_K|} ∈ set
evs"
```
⟨*proof*⟩

13.13 Proving fairness

Cannot prove that, if `B` has NRO, then `A` has her NRR. It would appear that `B` has a small advantage, though it is useless to win disputes: `B` needs to present `con_K` as well.

Strange: unicity of the label protects `A`?

lemma *A_unicity:*
```
"[|NRO = Crypt (priK A) {|Number f_nro, Agent B, Nonce L, Crypt K M|};
   NRO ∈ parts (spies evs);
   Says A B {|Number f_nro, Agent B, Nonce L, Crypt K M', NRO'|}
     ∈ set evs;
   A ∉ bad; evs ∈ zg |]
 ==> M'=M"
```
⟨*proof*⟩

Fairness lemma: if `sub_K` exists, then `A` holds NRR. Relies on unicity of labels.

lemma *sub_K_implies_NRR:*
```
"[| NRO = Crypt (priK A) {|Number f_nro, Agent B, Nonce L, Crypt K M|};
    NRR = Crypt (priK B) {|Number f_nrr, Agent A, Nonce L, Crypt K M|};
    sub_K ∈ parts (spies evs);
    NRO ∈ parts (spies evs);
    sub_K = Crypt (priK A) {|Number f_sub, Agent B, Nonce L, Key K|};
    A ∉ bad;  evs ∈ zg |]
 ==> Gets A {|Number f_nrr, Agent A, Nonce L, NRR|} ∈ set evs"
```
⟨*proof*⟩

lemma *Crypt_used_imp_L_used:*
```
"[| Crypt (priK TTP) {|F, A, B, L, K|} ∈ used evs; evs ∈ zg |]
 ==> L ∈ used evs"
```
⟨*proof*⟩

Fairness for `A`: if `con_K` and `NRO` exist, then `A` holds NRR. `A` must be uncompromised, but there is no assumption about `B`.

theorem *A_fairness_NRO:*
```
"[|con_K ∈ used evs;
   NRO ∈ parts (spies evs);
   con_K = Crypt (priK TTP)
                {|Number f_con, Agent A, Agent B, Nonce L, Key K|};
   NRO = Crypt (priK A) {|Number f_nro, Agent B, Nonce L, Crypt K M|};
   NRR = Crypt (priK B) {|Number f_nrr, Agent A, Nonce L, Crypt K M|};
   A ∉ bad;  evs ∈ zg |]
```

```
    ==> Gets A {|Number f_nrr, Agent A, Nonce L, NRR|} ∈ set evs"
```
⟨*proof*⟩

Fairness for `B`: NRR exists at all, then `B` holds NRO. `B` must be uncompromised, but there is no assumption about `A`.

theorem *B_fairness_NRR:*
```
     "[|NRR ∈ used evs;
        NRR = Crypt (priK B) {|Number f_nrr, Agent A, Nonce L, C|};
        NRO = Crypt (priK A) {|Number f_nro, Agent B, Nonce L, C|};
        B ∉ bad; evs ∈ zg |]
    ==> Gets B {|Number f_nro, Agent B, Nonce L, C, NRO|} ∈ set evs"
```
⟨*proof*⟩

If `con_K` exists at all, then `B` can get it, by `con_K_validity`. Cannot conclude that also NRO is available to `B`, because if `A` were unfair, `A` could build message 3 without building message 1, which contains NRO.

end

14 Verifying the Needham-Schroeder Public-Key Protocol

theory *NS_Public_Bad* **imports** *Public* **begin**

consts *ns_public :: "event list set"*

inductive *ns_public*
intros

```
  Nil:  "[] ∈ ns_public"


  Fake: "⟦evsf ∈ ns_public;  X ∈ synth (analz (spies evsf))⟧
         ⟹ Says Spy B X  # evsf ∈ ns_public"


  NS1:  "⟦evs1 ∈ ns_public;  Nonce NA ∉ used evs1⟧
         ⟹ Says A B (Crypt (pubEK B) {|Nonce NA, Agent A|})
              # evs1  ∈  ns_public"


  NS2:  "⟦evs2 ∈ ns_public;  Nonce NB ∉ used evs2;
           Says A' B (Crypt (pubEK B) {|Nonce NA, Agent A|}) ∈ set evs2⟧
         ⟹ Says B A (Crypt (pubEK A) {|Nonce NA, Nonce NB|})
              # evs2  ∈  ns_public"


  NS3:  "⟦evs3 ∈ ns_public;
           Says A  B (Crypt (pubEK B) {|Nonce NA, Agent A|}) ∈ set evs3;
           Says B' A (Crypt (pubEK A) {|Nonce NA, Nonce NB|}) ∈ set evs3⟧
         ⟹ Says A B (Crypt (pubEK B) (Nonce NB)) # evs3 ∈ ns_public"
```

declare *knows_Spy_partsEs [elim]*
declare *analz_into_parts [dest]*
declare *Fake_parts_insert_in_Un [dest]*
declare *image_eq_UN [simp]*

lemma *"∃NB. ∃evs ∈ ns_public. Says A B (Crypt (pubEK B) (Nonce NB)) ∈ set evs"*
⟨*proof*⟩

lemma *Spy_see_priEK [simp]:*
"evs ∈ ns_public ⟹ (Key (priEK A) ∈ parts (spies evs)) = (A ∈ bad)"
⟨*proof*⟩

lemma *Spy_analz_priEK [simp]:*
"evs ∈ ns_public ⟹ (Key (priEK A) ∈ analz (spies evs)) = (A ∈ bad)"
⟨*proof*⟩

lemma *no_nonce_NS1_NS2 [rule_format]:*
"evs ∈ ns_public
⟹ Crypt (pubEK C) {|NA', Nonce NA|} ∈ parts (spies evs) ⟶
Crypt (pubEK B) {|Nonce NA, Agent A|} ∈ parts (spies evs) ⟶
Nonce NA ∈ analz (spies evs)"
⟨*proof*⟩

lemma *unique_NA:*
"⟦Crypt(pubEK B) {|Nonce NA, Agent A |} ∈ parts(spies evs);
Crypt(pubEK B') {|Nonce NA, Agent A'|} ∈ parts(spies evs);
Nonce NA ∉ analz (spies evs); evs ∈ ns_public⟧
⟹ A=A' ∧ B=B'"
⟨*proof*⟩

theorem *Spy_not_see_NA:*
"⟦Says A B (Crypt(pubEK B) {|Nonce NA, Agent A|}) ∈ set evs;
A ∉ bad; B ∉ bad; evs ∈ ns_public⟧
⟹ Nonce NA ∉ analz (spies evs)"
⟨*proof*⟩

lemma *A_trusts_NS2_lemma [rule_format]:*
```
"⟦A ∉ bad;  B ∉ bad;  evs ∈ ns_public⟧
 ⟹ Crypt (pubEK A) {|Nonce NA, Nonce NB|} ∈ parts (spies evs) ⟶
    Says A B (Crypt(pubEK B) {|Nonce NA, Agent A|}) ∈ set evs ⟶
    Says B A (Crypt(pubEK A) {|Nonce NA, Nonce NB|}) ∈ set evs"
```
⟨*proof*⟩

theorem *A_trusts_NS2:*
```
"⟦Says A  B (Crypt(pubEK B) {|Nonce NA, Agent A|}) ∈ set evs;
  Says B' A (Crypt(pubEK A) {|Nonce NA, Nonce NB|}) ∈ set evs;
  A ∉ bad;  B ∉ bad;  evs ∈ ns_public⟧
 ⟹ Says B A (Crypt(pubEK A) {|Nonce NA, Nonce NB|}) ∈ set evs"
```
⟨*proof*⟩

lemma *B_trusts_NS1 [rule_format]:*
```
"evs ∈ ns_public
 ⟹ Crypt (pubEK B) {|Nonce NA, Agent A|} ∈ parts (spies evs) ⟶
    Nonce NA ∉ analz (spies evs) ⟶
    Says A B (Crypt (pubEK B) {|Nonce NA, Agent A|}) ∈ set evs"
```
⟨*proof*⟩

lemma *unique_NB [dest]:*
```
"⟦Crypt(pubEK A)  {|Nonce NA, Nonce NB|} ∈ parts(spies evs);
  Crypt(pubEK A') {|Nonce NA', Nonce NB|} ∈ parts(spies evs);
  Nonce NB ∉ analz (spies evs); evs ∈ ns_public⟧
 ⟹ A=A' ∧ NA=NA'"
```
⟨*proof*⟩

theorem *Spy_not_see_NB [dest]:*
```
"⟦Says B A (Crypt (pubEK A) {|Nonce NA, Nonce NB|}) ∈ set evs;
  ∀C. Says A C (Crypt (pubEK C) (Nonce NB)) ∉ set evs;
  A ∉ bad;  B ∉ bad;  evs ∈ ns_public⟧
 ⟹ Nonce NB ∉ analz (spies evs)"
```
⟨*proof*⟩

lemma *B_trusts_NS3_lemma [rule_format]:*
```
"⟦A ∉ bad;  B ∉ bad;  evs ∈ ns_public⟧
 ⟹ Crypt (pubEK B) (Nonce NB) ∈ parts (spies evs) ⟶
    Says B A  (Crypt (pubEK A) {|Nonce NA, Nonce NB|}) ∈ set evs ⟶
    (∃C. Says A C (Crypt (pubEK C) (Nonce NB)) ∈ set evs)"
```
⟨*proof*⟩

theorem *B_trusts_NS3:*
```
"⟦Says B A (Crypt (pubEK A) {|Nonce NA, Nonce NB|}) ∈ set evs;
  Says A' B (Crypt (pubEK B) (Nonce NB)) ∈ set evs;
  A ∉ bad; B ∉ bad; evs ∈ ns_public⟧
 ⟹ ∃C. Says A C (Crypt (pubEK C) (Nonce NB)) ∈ set evs"
```
⟨*proof*⟩

lemma
```
"⟦A ∉ bad; B ∉ bad; evs ∈ ns_public⟧
 ⟹ Says B A (Crypt (pubEK A) {|Nonce NA, Nonce NB|}) ∈ set evs
     ⟶ Nonce NB ∉ analz (spies evs)"
```
⟨*proof*⟩

end

15 Verifying the Needham-Schroeder-Lowe Public-Key Protocol

theory *NS_Public* **imports** *Public* **begin**

consts
```
ns_public :: "event list set"
```

inductive *ns_public*
intros

```
Nil: "[] ∈ ns_public"

Fake: "⟦evsf ∈ ns_public; X ∈ synth (analz (spies evsf))⟧
       ⟹ Says Spy B X # evsf ∈ ns_public"

NS1: "⟦evs1 ∈ ns_public; Nonce NA ∉ used evs1⟧
       ⟹ Says A B (Crypt (pubEK B) {|Nonce NA, Agent A|})
            # evs1 ∈ ns_public"

NS2: "⟦evs2 ∈ ns_public; Nonce NB ∉ used evs2;
        Says A' B (Crypt (pubEK B) {|Nonce NA, Agent A|}) ∈ set evs2⟧
       ⟹ Says B A (Crypt (pubEK A) {|Nonce NA, Nonce NB, Agent B|})
            # evs2 ∈ ns_public"

NS3: "⟦evs3 ∈ ns_public;
        Says A B (Crypt (pubEK B) {|Nonce NA, Agent A|}) ∈ set evs3;
        Says B' A (Crypt (pubEK A) {|Nonce NA, Nonce NB, Agent B|})
          ∈ set evs3⟧
       ⟹ Says A B (Crypt (pubEK B) (Nonce NB)) # evs3 ∈ ns_public"
```

declare *knows_Spy_partsEs [elim]*
declare *knows_Spy_partsEs [elim]*
declare *analz_into_parts [dest]*
declare *Fake_parts_insert_in_Un [dest]*
declare *image_eq_UN [simp]*

lemma *"∃NB. ∃evs ∈ ns_public. Says A B (Crypt (pubEK B) (Nonce NB)) ∈ set evs"*
⟨*proof*⟩

lemma *Spy_see_priEK [simp]:*
"evs ∈ ns_public ⟹ (Key (priEK A) ∈ parts (spies evs)) = (A ∈ bad)"
⟨*proof*⟩

lemma *Spy_analz_priEK [simp]:*
"evs ∈ ns_public ⟹ (Key (priEK A) ∈ analz (spies evs)) = (A ∈ bad)"
⟨*proof*⟩

15.1 Authenticity properties obtained from NS2

lemma *no_nonce_NS1_NS2 [rule_format]:*
"evs ∈ ns_public
⟹ Crypt (pubEK C) {|NA', Nonce NA, Agent D|} ∈ parts (spies evs) ⟶
Crypt (pubEK B) {|Nonce NA, Agent A|} ∈ parts (spies evs) ⟶
Nonce NA ∈ analz (spies evs)"
⟨*proof*⟩

lemma *unique_NA:*
"⟦Crypt(pubEK B) {|Nonce NA, Agent A |} ∈ parts(spies evs);
Crypt(pubEK B') {|Nonce NA, Agent A'|} ∈ parts(spies evs);
Nonce NA ∉ analz (spies evs); evs ∈ ns_public⟧
⟹ A=A' ∧ B=B'"
⟨*proof*⟩

theorem *Spy_not_see_NA:*
"⟦Says A B (Crypt(pubEK B) {|Nonce NA, Agent A|}) ∈ set evs;
A ∉ bad; B ∉ bad; evs ∈ ns_public⟧
⟹ Nonce NA ∉ analz (spies evs)"
⟨*proof*⟩

lemma *A_trusts_NS2_lemma [rule_format]:*
"⟦A ∉ bad; B ∉ bad; evs ∈ ns_public⟧
⟹ Crypt (pubEK A) {|Nonce NA, Nonce NB, Agent B|} ∈ parts (spies evs)
⟶
Says A B (Crypt(pubEK B) {|Nonce NA, Agent A|}) ∈ set evs ⟶

```
        Says B A (Crypt(pubEK A) {|Nonce NA, Nonce NB, Agent B|}) ∈ set evs"
```
⟨*proof*⟩

theorem *A_trusts_NS2:*
```
     "⟦Says A  B (Crypt(pubEK B) {|Nonce NA, Agent A|}) ∈ set evs;
       Says B' A (Crypt(pubEK A) {|Nonce NA, Nonce NB, Agent B|}) ∈ set evs;
       A ∉ bad;  B ∉ bad;  evs ∈ ns_public⟧
      ⟹ Says B A (Crypt(pubEK A) {|Nonce NA, Nonce NB, Agent B|}) ∈ set evs"
```
⟨*proof*⟩

lemma *B_trusts_NS1 [rule_format]:*
```
     "evs ∈ ns_public
      ⟹ Crypt (pubEK B) {|Nonce NA, Agent A|} ∈ parts (spies evs) ⟶
         Nonce NA ∉ analz (spies evs) ⟶
         Says A B (Crypt (pubEK B) {|Nonce NA, Agent A|}) ∈ set evs"
```
⟨*proof*⟩

15.2 Authenticity properties obtained from NS2

lemma *unique_NB [dest]:*
```
     "⟦Crypt(pubEK A)  {|Nonce NA, Nonce NB, Agent B|} ∈ parts(spies evs);
       Crypt(pubEK A') {|Nonce NA', Nonce NB, Agent B'|} ∈ parts(spies evs);
       Nonce NB ∉ analz (spies evs); evs ∈ ns_public⟧
      ⟹ A=A' ∧ NA=NA' ∧ B=B'"
```
⟨*proof*⟩

theorem *Spy_not_see_NB [dest]:*
```
     "⟦Says B A (Crypt (pubEK A) {|Nonce NA, Nonce NB, Agent B|}) ∈ set evs;
       A ∉ bad;  B ∉ bad;  evs ∈ ns_public⟧
      ⟹ Nonce NB ∉ analz (spies evs)"
```
⟨*proof*⟩

lemma *B_trusts_NS3_lemma [rule_format]:*
```
     "⟦A ∉ bad;  B ∉ bad;  evs ∈ ns_public⟧ ⟹
      Crypt (pubEK B) (Nonce NB) ∈ parts (spies evs) ⟶
      Says B A (Crypt (pubEK A) {|Nonce NA, Nonce NB, Agent B|}) ∈ set evs
⟶
      Says A B (Crypt (pubEK B) (Nonce NB)) ∈ set evs"
```
⟨*proof*⟩

theorem *B_trusts_NS3:*
```
     "⟦Says B A  (Crypt (pubEK A) {|Nonce NA, Nonce NB, Agent B|}) ∈ set evs;
       Says A' B (Crypt (pubEK B) (Nonce NB)) ∈ set evs;
       A ∉ bad;  B ∉ bad;  evs ∈ ns_public⟧
      ⟹ Says A B (Crypt (pubEK B) (Nonce NB)) ∈ set evs"
```
⟨*proof*⟩

15.3 Overall guarantee for B

theorem *B_trusts_protocol:*
```
"⟦A ∉ bad;  B ∉ bad;  evs ∈ ns_public⟧ ⟹
 Crypt (pubEK B) (Nonce NB) ∈ parts (spies evs) ⟶
 Says B A  (Crypt (pubEK A) {|Nonce NA, Nonce NB, Agent B|}) ∈ set evs
⟶
 Says A B (Crypt (pubEK B) {|Nonce NA, Agent A|}) ∈ set evs"
```
⟨*proof*⟩

end

16 The TLS Protocol: Transport Layer Security

theory *TLS* **imports** *Public NatPair* **begin**

constdefs
```
  certificate      :: "[agent,key] => msg"
    "certificate A KA == Crypt (priSK Server) {|Agent A, Key KA|}"
```

TLS apparently does not require separate keypairs for encryption and signature. Therefore, we formalize signature as encryption using the private encryption key.

datatype *role = ClientRole | ServerRole*

consts

```
  PRF  :: "nat*nat*nat => nat"


  sessionK :: "(nat*nat*nat) * role => key"
```

syntax
```
    clientK :: "nat*nat*nat => key"
    serverK :: "nat*nat*nat => key"
```

translations
```
  "clientK X" == "sessionK(X, ClientRole)"
  "serverK X" == "sessionK(X, ServerRole)"
```

specification *(PRF)*
```
  inj_PRF: "inj PRF"
```
— the pseudo-random function is collision-free
⟨*proof*⟩

specification *(sessionK)*
```
  inj_sessionK: "inj sessionK"
```
— sessionK is collision-free; also, no clientK clashes with any serverK.
⟨*proof*⟩

axioms
— sessionK makes symmetric keys
```
  isSym_sessionK: "sessionK nonces ∈ symKeys"
```

— sessionK never clashes with a long-term symmetric key (they don't exist in TLS anyway)

```
sessionK_neq_shrK [iff]: "sessionK nonces ≠ shrK A"
```

consts `tls :: "event list set"`
inductive `tls`
intros

`Nil:` — The initial, empty trace

```
"[] ∈ tls"
```

`Fake:` — The Spy may say anything he can say. The sender field is correct, but agents don't use that information.

```
"[| evsf ∈ tls;  X ∈ synth (analz (spies evsf)) |]
 ==> Says Spy B X # evsf ∈ tls"
```

`SpyKeys:` — The spy may apply `PRF` and `sessionK` to available nonces

```
"[| evsSK ∈ tls;
    {Nonce NA, Nonce NB, Nonce M} <= analz (spies evsSK) |]
 ==> Notes Spy {| Nonce (PRF(M,NA,NB)),
                  Key (sessionK((NA,NB,M),role)) |} # evsSK ∈ tls"
```

`ClientHello:`

— (7.4.1.2) PA represents `CLIENT_VERSION`, `CIPHER_SUITES` and `COMPRESSION_METHODS`. It is uninterpreted but will be confirmed in the FINISHED messages. NA is CLIENT RANDOM, while SID is `SESSION_ID`. UNIX TIME is omitted because the protocol doesn't use it. May assume `NA ∉ range PRF` because CLIENT RANDOM is 28 bytes while MASTER SECRET is 48 bytes

```
"[| evsCH ∈ tls;  Nonce NA ∉ used evsCH;  NA ∉ range PRF |]
 ==> Says A B {|Agent A, Nonce NA, Number SID, Number PA|}
       # evsCH  ∈  tls"
```

`ServerHello:`

— 7.4.1.3 of the TLS Internet-Draft PB represents `CLIENT_VERSION`, `CIPHER_SUITE` and `COMPRESSION_METHOD`. SERVER CERTIFICATE (7.4.2) is always present. `CERTIFICATE_REQUEST` (7.4.4) is implied.

```
"[| evsSH ∈ tls;  Nonce NB ∉ used evsSH;  NB ∉ range PRF;
    Says A' B {|Agent A, Nonce NA, Number SID, Number PA|}
      ∈ set evsSH |]
 ==> Says B A {|Nonce NB, Number SID, Number PB|} # evsSH  ∈  tls"
```

`Certificate:`

— SERVER (7.4.2) or CLIENT (7.4.6) CERTIFICATE.

```
"evsC ∈ tls ==> Says B A (certificate B (pubK B)) # evsC  ∈  tls"
```

`ClientKeyExch:`

— CLIENT KEY EXCHANGE (7.4.7). The client, A, chooses PMS, the PREMASTER SECRET. She encrypts PMS using the supplied KB, which ought to be pubK B. We assume `PMS ∉ range PRF` because a clash betweem the PMS and another MASTER SECRET is highly unlikely (even though both items have the same length, 48 bytes). The Note event records in the trace that she knows PMS (see REMARK at top).

```
"[| evsCX ∈ tls;  Nonce PMS ∉ used evsCX;  PMS ∉ range PRF;
    Says B' A (certificate B KB) ∈ set evsCX |]
```

```
    ==> Says A B (Crypt KB (Nonce PMS))
        # Notes A {|Agent B, Nonce PMS|}
        # evsCX  ∈  tls"
```

CertVerify:

— The optional Certificate Verify (7.4.8) message contains the specific components listed in the security analysis, F.1.1.2. It adds the pre-master-secret, which is also essential! Checking the signature, which is the only use of A's certificate, assures B of A's presence

```
    "[| evsCV ∈ tls;
        Says B' A {|Nonce NB, Number SID, Number PB|} ∈ set evsCV;
        Notes A {|Agent B, Nonce PMS|} ∈ set evsCV |]
     ==> Says A B (Crypt (priK A) (Hash{|Nonce NB, Agent B, Nonce PMS|}))
         # evsCV  ∈  tls"
```

— Finally come the FINISHED messages (7.4.8), confirming PA and PB among other things. The master-secret is PRF(PMS,NA,NB). Either party may send its message first.

ClientFinished:

— The occurrence of Notes A —Agent B, Nonce PMS— stops the rule's applying when the Spy has satisfied the "Says A B" by repaying messages sent by the true client; in that case, the Spy does not know PMS and could not send ClientFinished. One could simply put $A \neq Spy$ 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 en-

crypted 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 FINISHED message using the recovered MASTER SECRET

```
"[| evsSR ∈ tls;
    Says A' B {|Agent A, Nonce NA, Number SID, Number PA|}: set evsSR;
    Says B  A {|Nonce NB, Number SID, Number PB|} ∈ set evsSR;
    Notes B {|Number SID, Agent A, Agent B, Nonce M|} ∈ set evsSR
|]
 ==> Says B A (Crypt (serverK(NA,NB,M))
```

```
                    (Hash{|Number SID, Nonce M,
                           Nonce NA, Number PA, Agent A,
                           Nonce NB, Number PB, Agent B|})) # evsSR
              ∈  tls"
```

Oops:
— The most plausible compromise is of an old session key. Losing the MASTER SECRET or PREMASTER SECRET is more serious but rather unlikely. The assumption $A \neq Spy$ is essential: otherwise the Spy could learn session keys merely by replaying messages!

```
"[| evso ∈ tls;  A ≠ Spy;
    Says A B (Crypt (sessionK((NA,NB,M),role)) X) ∈ set evso |]
 ==> Says A Spy (Key (sessionK((NA,NB,M),role))) # evso  ∈  tls"
```

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

Automatically unfold the definition of "certificate"

declare *certificate_def [simp]*

Injectiveness of key-generating functions

declare *inj_PRF [THEN inj_eq, iff]*
declare *inj_sessionK [THEN inj_eq, iff]*
declare *isSym_sessionK [simp]*

lemma *pubK_neq_sessionK [iff]: "publicKey b A ≠ sessionK arg"*
⟨*proof*⟩

declare *pubK_neq_sessionK [THEN not_sym, iff]*

lemma *priK_neq_sessionK [iff]: "invKey (publicKey b A) ≠ sessionK arg"*
⟨*proof*⟩

declare *priK_neq_sessionK [THEN not_sym, iff]*

lemmas *keys_distinct = pubK_neq_sessionK priK_neq_sessionK*

16.1 Protocol Proofs

Possibility properties state that some traces run the protocol to the end. Four paths and 12 rules are considered.

Possibility property ending with ClientAccepts.

lemma *"[| ∀evs. (@ N. Nonce N ∉ used evs) ∉ range PRF; A ≠ B |]*
==> ∃SID M. ∃evs ∈ tls.
Notes A {|Number SID, Agent A, Agent B, Nonce M|} ∈ set evs"

⟨*proof*⟩

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

lemma *"[| ∀evs. (@ N. Nonce N ∉ used evs) ∉ range PRF; A ≠ B |]*
==> ∃SID NA PA NB PB M. ∃evs ∈ tls.
Notes B {|Number SID, Agent A, Agent B, Nonce M|} ∈ set evs"
⟨*proof*⟩

Another one, for CertVerify (which is optional)

lemma *"[| ∀evs. (@ N. Nonce N ∉ used evs) ∉ range PRF; A ≠ B |]*
==> ∃NB PMS. ∃evs ∈ tls.
Says A B (Crypt (priK A) (Hash{|Nonce NB, Agent B, Nonce PMS|}))

∈ set evs"
⟨*proof*⟩

Another one, for session resumption (both ServerResume and ClientResume). NO tls.Nil here: we refer to a previous session, not the empty trace.

lemma *"[| evs0 ∈ tls;*
Notes A {|Number SID, Agent A, Agent B, Nonce M|} ∈ set evs0;
Notes B {|Number SID, Agent A, Agent B, Nonce M|} ∈ set evs0;
∀evs. (@ N. Nonce N ∉ used evs) ∉ range PRF;
A ≠ B |]
==> ∃NA PA NB PB X. ∃evs ∈ tls.
X = Hash{|Number SID, Nonce M,
Nonce NA, Number PA, Agent A,
Nonce NB, Number PB, Agent B|} &
Says A B (Crypt (clientK(NA,NB,M)) X) ∈ set evs &
Says B A (Crypt (serverK(NA,NB,M)) X) ∈ set evs"
⟨*proof*⟩

16.2 Inductive proofs about tls

Spy never sees a good agent's private key!

lemma *Spy_see_priK [simp]:*
"evs ∈ tls ==> (Key (privateKey b A) ∈ parts (spies evs)) = (A ∈ bad)"
⟨*proof*⟩

lemma *Spy_analz_priK [simp]:*
"evs ∈ tls ==> (Key (privateKey b A) ∈ analz (spies evs)) = (A ∈ bad)"
⟨*proof*⟩

lemma *Spy_see_priK_D [dest!]:*
"[|Key (privateKey b A) ∈ parts (knows Spy evs); evs ∈ tls|] ==> A ∈ bad"
⟨*proof*⟩

This lemma says that no false certificates exist. One might extend the model to include bogus certificates for the agents, but there seems little point in doing so: the loss of their private keys is a worse breach of security.

lemma *certificate_valid:*

```
    "[| certificate B KB ∈ parts (spies evs);  evs ∈ tls |] ==> KB = pubK
B"
```
⟨*proof*⟩

lemmas *CX_KB_is_pubKB = Says_imp_spies [THEN parts.Inj, THEN certificate_valid]*

16.2.1 Properties of items found in Notes

lemma *Notes_Crypt_parts_spies:*
```
     "[| Notes A {|Agent B, X|} ∈ set evs;  evs ∈ tls |]
      ==> Crypt (pubK B) X ∈ parts (spies evs)"
```
⟨*proof*⟩

C may be either A or B

lemma *Notes_master_imp_Crypt_PMS:*
```
     "[| Notes C {|s, Agent A, Agent B, Nonce(PRF(PMS,NA,NB))|} ∈ set evs;
         evs ∈ tls |]
      ==> Crypt (pubK B) (Nonce PMS) ∈ parts (spies evs)"
```
⟨*proof*⟩

Compared with the theorem above, both premise and conclusion are stronger

lemma *Notes_master_imp_Notes_PMS:*
```
     "[| Notes A {|s, Agent A, Agent B, Nonce(PRF(PMS,NA,NB))|} ∈ set evs;
         evs ∈ tls |]
      ==> Notes A {|Agent B, Nonce PMS|} ∈ set evs"
```
⟨*proof*⟩

16.2.2 Protocol goal: if B receives CertVerify, then A sent it

B can check A's signature if he has received A's certificate.

lemma *TrustCertVerify_lemma:*
```
     "[| X ∈ parts (spies evs);
         X = Crypt (priK A) (Hash{|nb, Agent B, pms|});
         evs ∈ tls;  A ∉ bad |]
      ==> Says A B X ∈ set evs"
```
⟨*proof*⟩

Final version: B checks X using the distributed KA instead of priK A

lemma *TrustCertVerify:*
```
     "[| X ∈ parts (spies evs);
         X = Crypt (invKey KA) (Hash{|nb, Agent B, pms|});
         certificate A KA ∈ parts (spies evs);
         evs ∈ tls;  A ∉ bad |]
      ==> Says A B X ∈ set evs"
```
⟨*proof*⟩

If CertVerify is present then A has chosen PMS.

lemma *UseCertVerify_lemma:*
```
     "[| Crypt (priK A) (Hash{|nb, Agent B, Nonce PMS|}) ∈ parts (spies evs);
         evs ∈ tls;  A ∉ bad |]
      ==> Notes A {|Agent B, Nonce PMS|} ∈ set evs"
```
⟨*proof*⟩

Final version using the distributed KA instead of priK A

lemma *UseCertVerify:*
```
"[| Crypt (invKey KA) (Hash{|nb, Agent B, Nonce PMS|})
      ∈ parts (spies evs);
    certificate A KA ∈ parts (spies evs);
    evs ∈ tls;  A ∉ bad |]
 ==> Notes A {|Agent B, Nonce PMS|} ∈ set evs"
```
⟨*proof*⟩

lemma *no_Notes_A_PRF [simp]:*
```
"evs ∈ tls ==> Notes A {|Agent B, Nonce (PRF x)|} ∉ set evs"
```
⟨*proof*⟩

lemma *MS_imp_PMS [dest!]:*
```
"[| Nonce (PRF (PMS,NA,NB)) ∈ parts (spies evs);  evs ∈ tls |]
 ==> Nonce PMS ∈ parts (spies evs)"
```
⟨*proof*⟩

16.2.3 Unicity results for PMS, the pre-master-secret

PMS determines B.

lemma *Crypt_unique_PMS:*
```
"[| Crypt(pubK B)  (Nonce PMS) ∈ parts (spies evs);
    Crypt(pubK B') (Nonce PMS) ∈ parts (spies evs);
    Nonce PMS ∉ analz (spies evs);
    evs ∈ tls |]
 ==> B=B'"
```
⟨*proof*⟩

In A's internal Note, PMS determines A and B.

lemma *Notes_unique_PMS:*
```
"[| Notes A  {|Agent B,  Nonce PMS|} ∈ set evs;
    Notes A' {|Agent B', Nonce PMS|} ∈ set evs;
    evs ∈ tls |]
 ==> A=A' & B=B'"
```
⟨*proof*⟩

16.3 Secrecy Theorems

Key compromise lemma needed to prove *analz_image_keys*. No collection of keys can help the spy get new private keys.

lemma *analz_image_priK [rule_format]:*
```
"evs ∈ tls
 ==> ∀KK. (Key(priK B) ∈ analz (Key‘KK Un (spies evs))) =
     (priK B ∈ KK | B ∈ bad)"
```
⟨*proof*⟩

slightly speeds up the big simplification below

lemma *range_sessionkeys_not_priK:*
```
"KK <= range sessionK ==> priK B ∉ KK"
```

⟨*proof*⟩

Lemma for the trivial direction of the if-and-only-if

lemma *analz_image_keys_lemma:*
```
"(X ∈ analz (G Un H)) --> (X ∈ analz H)  ==>
 (X ∈ analz (G Un H))  =  (X ∈ analz H)"
```
⟨*proof*⟩

lemma *analz_image_keys [rule_format]:*
```
"evs ∈ tls ==>
 ∀KK. KK <= range sessionK -->
         (Nonce N ∈ analz (Key‘KK Un (spies evs))) =
         (Nonce N ∈ analz (spies evs))"
```
⟨*proof*⟩

Knowing some session keys is no help in getting new nonces

lemma *analz_insert_key [simp]:*
```
"evs ∈ tls ==>
 (Nonce N ∈ analz (insert (Key (sessionK z)) (spies evs))) =
 (Nonce N ∈ analz (spies evs))"
```
⟨*proof*⟩

16.3.1 Protocol goal: serverK(Na,Nb,M) and clientK(Na,Nb,M) remain secure

Lemma: session keys are never used if PMS is fresh. Nonces don't have to agree, allowing session resumption. Converse doesn't hold; revealing PMS doesn't force the keys to be sent. THEY ARE NOT SUITABLE AS SAFE ELIM RULES.

lemma *PMS_lemma:*
```
"[| Nonce PMS ∉ parts (spies evs);
    K = sessionK((Na, Nb, PRF(PMS,NA,NB)), role);
    evs ∈ tls |]
 ==> Key K ∉ parts (spies evs) & (∀Y. Crypt K Y ∉ parts (spies evs))"
```
⟨*proof*⟩

lemma *PMS_sessionK_not_spied:*
```
"[| Key (sessionK((Na, Nb, PRF(PMS,NA,NB)), role)) ∈ parts (spies evs);
    evs ∈ tls |]
 ==> Nonce PMS ∈ parts (spies evs)"
```
⟨*proof*⟩

lemma *PMS_Crypt_sessionK_not_spied:*
```
"[| Crypt (sessionK((Na, Nb, PRF(PMS,NA,NB)), role)) Y
      ∈ parts (spies evs);  evs ∈ tls |]
 ==> Nonce PMS ∈ parts (spies evs)"
```
⟨*proof*⟩

Write keys are never sent if M (MASTER SECRET) is secure. Converse fails; betraying M doesn't force the keys to be sent! The strong Oops condition can be weakened later by unicity reasoning, with some effort. NO LONGER USED: see *clientK_not_spied* and *serverK_not_spied*

lemma *sessionK_not_spied:*
```
"[| ∀A. Says A Spy (Key (sessionK((NA,NB,M),role))) ∉ set evs;
    Nonce M ∉ analz (spies evs);  evs ∈ tls |]
 ==> Key (sessionK((NA,NB,M),role)) ∉ parts (spies evs)"
```
⟨*proof*⟩

If A sends ClientKeyExch to an honest B, then the PMS will stay secret.

lemma *Spy_not_see_PMS:*
```
"[| Notes A {|Agent B, Nonce PMS|} ∈ set evs;
    evs ∈ tls;  A ∉ bad;  B ∉ bad |]
 ==> Nonce PMS ∉ analz (spies evs)"
```
⟨*proof*⟩

If A sends ClientKeyExch to an honest B, then the MASTER SECRET will stay secret.

lemma *Spy_not_see_MS:*
```
"[| Notes A {|Agent B, Nonce PMS|} ∈ set evs;
    evs ∈ tls;  A ∉ bad;  B ∉ bad |]
 ==> Nonce (PRF(PMS,NA,NB)) ∉ analz (spies evs)"
```
⟨*proof*⟩

16.3.2 Weakening the Oops conditions for leakage of clientK

If A created PMS then nobody else (except the Spy in replays) would send a message using a clientK generated from that PMS.

lemma *Says_clientK_unique:*
```
"[| Says A' B' (Crypt (clientK(Na,Nb,PRF(PMS,NA,NB))) Y) ∈ set evs;
    Notes A {|Agent B, Nonce PMS|} ∈ set evs;
    evs ∈ tls;  A' ≠ Spy |]
 ==> A = A'"
```
⟨*proof*⟩

If A created PMS and has not leaked her clientK to the Spy, then it is completely secure: not even in parts!

lemma *clientK_not_spied:*
```
"[| Notes A {|Agent B, Nonce PMS|} ∈ set evs;
    Says A Spy (Key (clientK(Na,Nb,PRF(PMS,NA,NB)))) ∉ set evs;
    A ∉ bad;  B ∉ bad;
    evs ∈ tls |]
 ==> Key (clientK(Na,Nb,PRF(PMS,NA,NB))) ∉ parts (spies evs)"
```
⟨*proof*⟩

16.3.3 Weakening the Oops conditions for leakage of serverK

If A created PMS for B, then nobody other than B or the Spy would send a message using a serverK generated from that PMS.

lemma *Says_serverK_unique:*
```
"[| Says B' A' (Crypt (serverK(Na,Nb,PRF(PMS,NA,NB))) Y) ∈ set evs;
    Notes A {|Agent B, Nonce PMS|} ∈ set evs;
    evs ∈ tls;  A ∉ bad;  B ∉ bad;  B' ≠ Spy |]
 ==> B = B'"
```
⟨*proof*⟩

If A created PMS for B, and B has not leaked his serverK to the Spy, then it is completely secure: not even in parts!

lemma *serverK_not_spied:*
```
"[| Notes A {|Agent B, Nonce PMS|} ∈ set evs;
    Says B Spy (Key(serverK(Na,Nb,PRF(PMS,NA,NB)))) ∉ set evs;
    A ∉ bad;  B ∉ bad;  evs ∈ tls |]
 ==> Key (serverK(Na,Nb,PRF(PMS,NA,NB))) ∉ parts (spies evs)"
```
⟨*proof*⟩

16.3.4 Protocol goals: if A receives ServerFinished, then B is present and has used the quoted values PA, PB, etc. Note that it is up to A to compare PA with what she originally sent.

The mention of her name (A) in X assures A that B knows who she is.

lemma *TrustServerFinished [rule_format]:*
```
"[| X = Crypt (serverK(Na,Nb,M))
          (Hash{|Number SID, Nonce M,
                 Nonce Na, Number PA, Agent A,
                 Nonce Nb, Number PB, Agent B|});
    M = PRF(PMS,NA,NB);
    evs ∈ tls;  A ∉ bad;  B ∉ bad |]
 ==> Says B Spy (Key(serverK(Na,Nb,M))) ∉ set evs -->
     Notes A {|Agent B, Nonce PMS|} ∈ set evs -->
     X ∈ parts (spies evs) --> Says B A X ∈ set evs"
```
⟨*proof*⟩

This version refers not to ServerFinished but to any message from B. We don't assume B has received CertVerify, and an intruder could have changed A's identity in all other messages, so we can't be sure that B sends his message to A. If CLIENT KEY EXCHANGE were augmented to bind A's identity with PMS, then we could replace A' by A below.

lemma *TrustServerMsg [rule_format]:*
```
"[| M = PRF(PMS,NA,NB);  evs ∈ tls;  A ∉ bad;  B ∉ bad |]
 ==> Says B Spy (Key(serverK(Na,Nb,M))) ∉ set evs -->
     Notes A {|Agent B, Nonce PMS|} ∈ set evs -->
     Crypt (serverK(Na,Nb,M)) Y ∈ parts (spies evs)  -->
     (∃A'. Says B A' (Crypt (serverK(Na,Nb,M)) Y) ∈ set evs)"
```
⟨*proof*⟩

16.3.5 Protocol goal: if B receives any message encrypted with clientK then A has sent it

ASSUMING that A chose PMS. Authentication is assumed here; B cannot verify it. But if the message is ClientFinished, then B can then check the quoted values PA, PB, etc.

lemma *TrustClientMsg [rule_format]:*
```
"[| M = PRF(PMS,NA,NB);  evs ∈ tls;  A ∉ bad;  B ∉ bad |]
 ==> Says A Spy (Key(clientK(Na,Nb,M))) ∉ set evs -->
     Notes A {|Agent B, Nonce PMS|} ∈ set evs -->
     Crypt (clientK(Na,Nb,M)) Y ∈ parts (spies evs) -->
     Says A B (Crypt (clientK(Na,Nb,M)) Y) ∈ set evs"
```
⟨*proof*⟩

16.3.6 Protocol goal: if B receives ClientFinished, and if B is able to check a CertVerify from A, then A has used the quoted values PA, PB, etc. Even this one requires A to be uncompromised.

lemma *AuthClientFinished:*

```
"[| M = PRF(PMS,NA,NB);
    Says A Spy (Key(clientK(Na,Nb,M))) ∉ set evs;
    Says A' B (Crypt (clientK(Na,Nb,M)) Y) ∈ set evs;
    certificate A KA ∈ parts (spies evs);
    Says A'' B (Crypt (invKey KA) (Hash{|nb, Agent B, Nonce PMS|}))
      ∈ set evs;
    evs ∈ tls;  A ∉ bad;  B ∉ bad |]
 ==> Says A B (Crypt (clientK(Na,Nb,M)) Y) ∈ set evs"
```

⟨*proof*⟩

end

17 The Certified Electronic Mail Protocol by Abadi et al.

theory *CertifiedEmail* **imports** *Public* **begin**

syntax

```
TTP        :: agent
RPwd       :: "agent => key"
```

translations

```
"TTP"   == "Server "
"RPwd"  == "shrK "
```

consts

```
NoAuth   :: nat
TTPAuth  :: nat
SAuth    :: nat
BothAuth :: nat
```

We formalize a fixed way of computing responses. Could be better.

constdefs

```
  "response"    :: "agent => agent => nat => msg"
   "response S R q == Hash {|Agent S, Key (shrK R), Nonce q|}"
```

consts *certified_mail* `:: "event list set"`
inductive `"certified_mail"`
intros

Nil: — The empty trace
```
     "[] ∈ certified_mail"
```

Fake: — The Spy may say anything he can say. The sender field is correct, but agents don't use that information.
```
      "[| evsf ∈ certified_mail; X ∈ synth(analz(spies evsf))|]
       ==> Says Spy B X # evsf ∈ certified_mail"
```

FakeSSL: — The Spy may open SSL sessions with TTP, who is the only agent equipped with the necessary credentials to serve as an SSL server.
```
         "[| evsfssl ∈ certified_mail; X ∈ synth(analz(spies evsfssl))|]
          ==> Notes TTP {|Agent Spy, Agent TTP, X|} # evsfssl ∈ certified_mail"
```

CM1: — The sender approaches the recipient. The message is a number.
```
 "[|evs1 ∈ certified_mail;
    Key K ∉ used evs1;
    K ∈ symKeys;
    Nonce q ∉ used evs1;
    hs = Hash{|Number cleartext, Nonce q, response S R q, Crypt K (Number m)|};
    S2TTP = Crypt(pubEK TTP) {|Agent S, Number BothAuth, Key K, Agent R, hs|}|]
  ==> Says S R {|Agent S, Agent TTP, Crypt K (Number m), Number BothAuth,
                 Number cleartext, Nonce q, S2TTP|} # evs1
       ∈ certified_mail"
```

CM2: — The recipient records *S2TTP* while transmitting it and her password to *TTP* over an SSL channel.
```
 "[|evs2 ∈ certified_mail;
    Gets R {|Agent S, Agent TTP, em, Number BothAuth, Number cleartext,
             Nonce q, S2TTP|} ∈ set evs2;
    TTP ≠ R;
    hr = Hash {|Number cleartext, Nonce q, response S R q, em|} |]
  ==>
   Notes TTP {|Agent R, Agent TTP, S2TTP, Key(RPwd R), hr|} # evs2
     ∈ certified_mail"
```

CM3: — *TTP* simultaneously reveals the key to the recipient and gives a receipt to the sender. The SSL channel does not authenticate the client (*R*), but *TTP* accepts the message only if the given password is that of the claimed sender, *R*. He replies over the established SSL channel.
```
 "[|evs3 ∈ certified_mail;
    Notes TTP {|Agent R, Agent TTP, S2TTP, Key(RPwd R), hr|} ∈ set evs3;
    S2TTP = Crypt (pubEK TTP)
                    {|Agent S, Number BothAuth, Key k, Agent R, hs|};
    TTP ≠ R;  hs = hr;  k ∈ symKeys|]
  ==>
   Notes R {|Agent TTP, Agent R, Key k, hr|} #
```

```
   Gets S (Crypt (priSK TTP) S2TTP) #
   Says TTP S (Crypt (priSK TTP) S2TTP) # evs3 ∈ certified_mail"

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

declare *Says_imp_knows_Spy [THEN analz.Inj, dest]*
declare *analz_into_parts [dest]*

lemma *"[| Key K ∉ used []; K ∈ symKeys |] ==>*
∃S2TTP. ∃evs ∈ certified_mail.
Says TTP S (Crypt (priSK TTP) S2TTP) ∈ set evs"
⟨*proof*⟩

lemma *Gets_imp_Says:*
"[| Gets B X ∈ set evs; evs ∈ certified_mail |] ==> ∃A. Says A B X ∈ set evs"
⟨*proof*⟩

lemma *Gets_imp_parts_knows_Spy:*
"[|Gets A X ∈ set evs; evs ∈ certified_mail|] ==> X ∈ parts(spies evs)"
⟨*proof*⟩

lemma *CM2_S2TTP_analz_knows_Spy:*
```
 "[|Gets R {|Agent A, Agent B, em, Number AO, Number cleartext,
             Nonce q, S2TTP|} ∈ set evs;
    evs ∈ certified_mail|]
  ==> S2TTP ∈ analz(spies evs)"
```
⟨*proof*⟩

lemmas *CM2_S2TTP_parts_knows_Spy =*
CM2_S2TTP_analz_knows_Spy [THEN analz_subset_parts [THEN subsetD]]

lemma *hr_form_lemma [rule_format]:*
```
 "evs ∈ certified_mail
  ==> hr ∉ synth (analz (spies evs)) -->
      (∀S2TTP. Notes TTP {|Agent R, Agent TTP, S2TTP, pwd, hr|}
          ∈ set evs -->
      (∃clt q S em. hr = Hash {|Number clt, Nonce q, response S R q, em|}))"
```
⟨*proof*⟩

Cannot strengthen the first disjunct to $R \neq Spy$ because the fakessl rule allows Spy to spoof the sender's name. Maybe can strengthen the second disjunct with $R \neq Spy$.

lemma *hr_form:*
```
 "[|Notes TTP {|Agent R, Agent TTP, S2TTP, pwd, hr|} ∈ set evs;
    evs ∈ certified_mail|]
  ==> hr ∈ synth (analz (spies evs)) |
      (∃clt q S em. hr = Hash {|Number clt, Nonce q, response S R q, em|})"
```

⟨*proof*⟩

lemma *Spy_dont_know_private_keys [dest!]:*
```
    "[|Key (privateKey b A) ∈ parts (spies evs); evs ∈ certified_mail|]
     ==> A ∈ bad"
```
⟨*proof*⟩

lemma *Spy_know_private_keys_iff [simp]:*
```
    "evs ∈ certified_mail
     ==> (Key (privateKey b A) ∈ parts (spies evs)) = (A ∈ bad)"
```
⟨*proof*⟩

lemma *Spy_dont_know_TTPKey_parts [simp]:*
```
     "evs ∈ certified_mail ==> Key (privateKey b TTP) ∉ parts(spies evs)"
```

⟨*proof*⟩

lemma *Spy_dont_know_TTPKey_analz [simp]:*
```
     "evs ∈ certified_mail ==> Key (privateKey b TTP) ∉ analz(spies evs)"
```
⟨*proof*⟩

Thus, prove any goal that assumes that *Spy* knows a private key belonging to *TTP*

declare *Spy_dont_know_TTPKey_parts [THEN [2] rev_notE, elim!]*

lemma *CM3_k_parts_knows_Spy:*
```
 "[| evs ∈ certified_mail;
     Notes TTP {|Agent A, Agent TTP,
                 Crypt (pubEK TTP) {|Agent S, Number AO, Key K,
                 Agent R, hs|}, Key (RPwd R), hs|} ∈ set evs|]
  ==> Key K ∈ parts(spies evs)"
```
⟨*proof*⟩

lemma *Spy_dont_know_RPwd [rule_format]:*
```
    "evs ∈ certified_mail ==> Key (RPwd A) ∈ parts(spies evs) --> A ∈ bad"
```
⟨*proof*⟩

lemma *Spy_know_RPwd_iff [simp]:*
```
    "evs ∈ certified_mail ==> (Key (RPwd A) ∈ parts(spies evs)) = (A∈bad)"
```
⟨*proof*⟩

lemma *Spy_analz_RPwd_iff [simp]:*
```
    "evs ∈ certified_mail ==> (Key (RPwd A) ∈ analz(spies evs)) = (A∈bad)"
```
⟨*proof*⟩

Unused, but a guarantee of sorts

theorem *CertAutenticity:*
```
     "[|Crypt (priSK TTP) X ∈ parts (spies evs); evs ∈ certified_mail|]
      ==> ∃A. Says TTP A (Crypt (priSK TTP) X) ∈ set evs"
```
⟨*proof*⟩

17.1 Proving Confidentiality Results

lemma *analz_image_freshK [rule_format]:*

```
 "evs ∈ certified_mail ==>
   ∀K KK. invKey (pubEK TTP) ∉ KK -->
          (Key K ∈ analz (Key‘KK Un (spies evs))) =
          (K ∈ KK | Key K ∈ analz (spies evs))"
```

⟨*proof*⟩

lemma *analz_insert_freshK:*

```
  "[| evs ∈ certified_mail;  KAB ≠ invKey (pubEK TTP) |] ==>
      (Key K ∈ analz (insert (Key KAB) (spies evs))) =
      (K = KAB | Key K ∈ analz (spies evs))"
```

⟨*proof*⟩

S2TTP must have originated from a valid sender provided *K* is secure. Proof is surprisingly hard.

lemma *Notes_SSL_imp_used:*

```
     "[|Notes B {|Agent A, Agent B, X|} ∈ set evs|] ==> X ∈ used evs"
```

⟨*proof*⟩

lemma *S2TTP_sender_lemma [rule_format]:*

```
 "evs ∈ certified_mail ==>
    Key K ∉ analz (spies evs) -->
    (∀AO. Crypt (pubEK TTP)
           {|Agent S, Number AO, Key K, Agent R, hs|} ∈ used evs -->
    (∃m ctxt q.
        hs = Hash{|Number ctxt, Nonce q, response S R q, Crypt K (Number m)|}
&
        Says S R
           {|Agent S, Agent TTP, Crypt K (Number m), Number AO,
             Number ctxt, Nonce q,
             Crypt (pubEK TTP)
              {|Agent S, Number AO, Key K, Agent R, hs |}|} ∈ set evs))"
```

⟨*proof*⟩

lemma *S2TTP_sender:*

```
 "[|Crypt (pubEK TTP) {|Agent S, Number AO, Key K, Agent R, hs|} ∈ used evs;
    Key K ∉ analz (spies evs);
    evs ∈ certified_mail|]
  ==> ∃m ctxt q.
        hs = Hash{|Number ctxt, Nonce q, response S R q, Crypt K (Number m)|}
&
        Says S R
           {|Agent S, Agent TTP, Crypt K (Number m), Number AO,
             Number ctxt, Nonce q,
             Crypt (pubEK TTP)
              {|Agent S, Number AO, Key K, Agent R, hs |}|} ∈ set evs"
```

⟨*proof*⟩

Nobody can have used non-existent keys!

lemma *new_keys_not_used [simp]:*
```
"[|Key K ∉ used evs; K ∈ symKeys; evs ∈ certified_mail|]
 ==> K ∉ keysFor (parts (spies evs))"
```
⟨*proof*⟩

Less easy to prove `m' = m`. Maybe needs a separate unicity theorem for ciphertexts of the form `Crypt K (Number m)`, where `K` is secure.

lemma *Key_unique_lemma [rule_format]:*
```
"evs ∈ certified_mail ==>
  Key K ∉ analz (spies evs) -->
  (∀m cleartext q hs.
   Says S R
      {|Agent S, Agent TTP, Crypt K (Number m), Number AO,
        Number cleartext, Nonce q,
        Crypt (pubEK TTP) {|Agent S, Number AO, Key K, Agent R, hs|}|}
     ∈ set evs -->
  (∀m' cleartext' q' hs'.
  Says S' R'
      {|Agent S', Agent TTP, Crypt K (Number m'), Number AO',
        Number cleartext', Nonce q',
        Crypt (pubEK TTP) {|Agent S', Number AO', Key K, Agent R', hs'|}|}
     ∈ set evs --> R' = R & S' = S & AO' = AO & hs' = hs))"
```
⟨*proof*⟩

The key determines the sender, recipient and protocol options.

lemma *Key_unique:*
```
"[|Says S R
      {|Agent S, Agent TTP, Crypt K (Number m), Number AO,
        Number cleartext, Nonce q,
        Crypt (pubEK TTP) {|Agent S, Number AO, Key K, Agent R, hs|}|}
     ∈ set evs;
   Says S' R'
     {|Agent S', Agent TTP, Crypt K (Number m'), Number AO',
       Number cleartext', Nonce q',
       Crypt (pubEK TTP) {|Agent S', Number AO', Key K, Agent R', hs'|}|}
    ∈ set evs;
   Key K ∉ analz (spies evs);
   evs ∈ certified_mail|]
 ==> R' = R & S' = S & AO' = AO & hs' = hs"
```
⟨*proof*⟩

17.2 The Guarantees for Sender and Recipient

A Sender's guarantee: If Spy gets the key then `R` is bad and `S` moreover gets his return receipt (and therefore has no grounds for complaint).

theorem *S_fairness_bad_R:*
```
"[|Says S R {|Agent S, Agent TTP, Crypt K (Number m), Number AO,
              Number cleartext, Nonce q, S2TTP|} ∈ set evs;
   S2TTP = Crypt (pubEK TTP) {|Agent S, Number AO, Key K, Agent R, hs|};
   Key K ∈ analz (spies evs);
   evs ∈ certified_mail;
   S≠Spy|]
 ==> R ∈ bad & Gets S (Crypt (priSK TTP) S2TTP) ∈ set evs"
```

⟨*proof*⟩

Confidentially for the symmetric key

theorem *Spy_not_see_encrypted_key:*
```
"[|Says S R {|Agent S, Agent TTP, Crypt K (Number m), Number AO,
               Number cleartext, Nonce q, S2TTP|} ∈ set evs;
   S2TTP = Crypt (pubEK TTP) {|Agent S, Number AO, Key K, Agent R, hs|};
   evs ∈ certified_mail;
   S≠Spy; R ∉ bad|]
 ==> Key K ∉ analz(spies evs)"
```
⟨*proof*⟩

Agent `R`, who may be the Spy, doesn't receive the key until `S` has access to the return receipt.

theorem *S_guarantee:*
```
"[|Says S R {|Agent S, Agent TTP, Crypt K (Number m), Number AO,
              Number cleartext, Nonce q, S2TTP|} ∈ set evs;
   S2TTP = Crypt (pubEK TTP) {|Agent S, Number AO, Key K, Agent R, hs|};
   Notes R {|Agent TTP, Agent R, Key K, hs|} ∈ set evs;
   S≠Spy;  evs ∈ certified_mail|]
 ==> Gets S (Crypt (priSK TTP) S2TTP) ∈ set evs"
```
⟨*proof*⟩

If `R` sends message 2, and a delivery certificate exists, then `R` receives the necessary key. This result is also important to `S`, as it confirms the validity of the return receipt.

theorem *RR_validity:*
```
"[|Crypt (priSK TTP) S2TTP ∈ used evs;
   S2TTP = Crypt (pubEK TTP)
             {|Agent S, Number AO, Key K, Agent R,
               Hash {|Number cleartext, Nonce q, r, em|}|};
   hr = Hash {|Number cleartext, Nonce q, r, em|};
   R≠Spy;  evs ∈ certified_mail|]
 ==> Notes R {|Agent TTP, Agent R, Key K, hr|} ∈ set evs"
```
⟨*proof*⟩

end

18 Extensions to Standard Theories

theory *Extensions* **imports** *Event* **begin**

18.1 Extensions to Theory *Set*

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

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

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

18.2 Extensions to Theory *List*

18.2.1 "minus l x" erase the first element of "l" equal to "x"

consts *minus :: "'a list => 'a => 'a list"*

primrec
"minus [] y = []"
"minus (x#xs) y = (if x=y then xs else x # minus xs y)"

lemma *set_minus: "set (minus l x) <= set l"*
⟨*proof*⟩

18.3 Extensions to Theory *Message*

18.3.1 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]*

18.3.2 extract the agent number of an Agent message

consts *agt_nb :: "msg => agent"*

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

18.3.3 messages that are pairs

constdefs *is_MPair :: "msg => bool"*
"is_MPair X == EX Y Z. X = {|Y,Z|}"

declare *is_MPair_def [simp]*

lemma *MPair_is_MPair [iff]: "is_MPair {|X,Y|}"*
⟨*proof*⟩

lemma *Agent_isnt_MPair [iff]: "~ is_MPair (Agent A)"*
⟨*proof*⟩

lemma *Number_isnt_MPair [iff]: "~ is_MPair (Number n)"*
⟨*proof*⟩

lemma *Key_isnt_MPair [iff]: "~ is_MPair (Key K)"*
⟨*proof*⟩

lemma *Nonce_isnt_MPair [iff]: "~ is_MPair (Nonce n)"*
⟨*proof*⟩

lemma *Hash_isnt_MPair [iff]: "~ is_MPair (Hash X)"*
⟨*proof*⟩

lemma *Crypt_isnt_MPair [iff]: "~ is_MPair (Crypt K X)"*
⟨*proof*⟩

syntax *not_MPair :: "msg => bool"*

translations *"not_MPair X" == "~ is_MPair X"*

lemma *is_MPairE: "[| is_MPair X ==> P; not_MPair X ==> P |] ==> P"*
⟨*proof*⟩

declare *is_MPair_def [simp del]*

constdefs *has_no_pair :: "msg set => bool"*
"has_no_pair H == ALL X Y. {|X,Y|} ~:H"

declare *has_no_pair_def [simp]*

18.3.4 well-foundedness of messages

lemma *wf_Crypt1 [iff]: "Crypt K X ~= X"*
⟨*proof*⟩

lemma *wf_Crypt2 [iff]: "X ~= Crypt K X"*
⟨*proof*⟩

lemma *parts_size: "X:parts {Y} ==> X=Y | size X < size Y"*
⟨*proof*⟩

lemma *wf_Crypt_parts [iff]: "Crypt K X ~:parts {X}"*
⟨*proof*⟩

18.3.5 lemmas on keysFor

constdefs *usekeys :: "msg set => key set"*
"usekeys G == {K. EX Y. Crypt K Y:G}"

lemma *finite_keysFor [intro]: "finite G ==> finite (keysFor G)"*
⟨*proof*⟩

18.3.6 lemmas on parts

lemma *parts_sub: "[| X:parts G; G<=H |] ==> X:parts H"*
⟨*proof*⟩

lemma *parts_Diff [dest]: "X:parts (G - H) ==> X:parts G"*
⟨*proof*⟩

lemma *parts_Diff_notin: "[| Y ~:H; Nonce n ~:parts (H - {Y}) |]*
==> Nonce n ~:parts H"
⟨*proof*⟩

lemmas *parts_insert_substI = parts_insert [THEN ssubst]*
lemmas *parts_insert_substD = parts_insert [THEN sym, THEN ssubst]*

lemma *finite_parts_msg [iff]: "finite (parts {X})"*
⟨*proof*⟩

lemma *finite_parts [intro]: "finite H ==> finite (parts H)"*
⟨*proof*⟩

lemma *parts_parts: "[| X:parts {Y}; Y:parts G |] ==> X:parts G"*
⟨*proof*⟩

lemma *parts_parts_parts: "[| X:parts {Y}; Y:parts {Z}; Z:parts G |] ==> X:parts G"*
⟨*proof*⟩

lemma *parts_parts_Crypt: "[| Crypt K X:parts G; Nonce n:parts {X} |] ==> Nonce n:parts G"*
⟨*proof*⟩

18.3.7 lemmas on synth

lemma *synth_sub: "[| X:synth G; G<=H |] ==> X:synth H"*
⟨*proof*⟩

lemma *Crypt_synth [rule_format]: "[| X:synth G; Key K ~:G |] ==> Crypt K Y:parts {X} --> Crypt K Y:parts G"*
⟨*proof*⟩

18.3.8 lemmas on analz

lemma *analz_UnI1 [intro]: "X:analz G ==> X:analz (G Un H)"*
⟨*proof*⟩

lemma *analz_sub: "[| X:analz G; G <= H |] ==> X:analz H"*
⟨*proof*⟩

lemma *analz_Diff [dest]: "X:analz (G - H) ==> X:analz G"*
⟨*proof*⟩

lemmas *in_analz_subset_cong = analz_subset_cong [THEN subsetD]*

lemma *analz_eq: "A=A' ==> analz A = analz A'"*
⟨*proof*⟩

lemmas *insert_commute_substI = insert_commute [THEN ssubst]*

lemma *analz_insertD:*
"[| Crypt K Y:H; Key (invKey K):H |] ==> analz (insert Y H) = analz H"
⟨*proof*⟩

lemma *must_decrypt [rule_format,dest]: "[| X:analz H; has_no_pair H |] ==> X ~:H --> (EX K Y. Crypt K Y:H & Key (invKey K):H)"*
⟨*proof*⟩

lemma *analz_needs_only_finite: "X:analz H ==> EX G. G <= H & finite G"*
⟨*proof*⟩

lemma *notin_analz_insert: "X ~:analz (insert Y G) ==> X ~:analz G"*
⟨*proof*⟩

18.3.9 lemmas on parts, synth and analz

lemma *parts_invKey [rule_format,dest]:"X:parts {Y} ==>*
X:analz (insert (Crypt K Y) H) --> X ~:analz H --> Key (invKey K):analz H"
⟨*proof*⟩

lemma *in_analz: "Y:analz H ==> EX X. X:H & Y:parts {X}"*
⟨*proof*⟩

lemmas *in_analz_subset_parts = analz_subset_parts [THEN subsetD]*

lemma *Crypt_synth_insert: "[| Crypt K X:parts (insert Y H);*
Y:synth (analz H); Key K ~:analz H |] ==> Crypt K X:parts H"
⟨*proof*⟩

18.3.10 greatest nonce used in a message

consts *greatest_msg :: "msg => nat"*

recdef *greatest_msg "measure size"*
"greatest_msg (Nonce n) = n"
"greatest_msg {|X,Y|} = max (greatest_msg X) (greatest_msg Y)"
"greatest_msg (Crypt K X) = greatest_msg X"
"greatest_msg other = 0"

lemma *greatest_msg_is_greatest: "Nonce n:parts {X} ==> n <= greatest_msg*
X"
⟨*proof*⟩

18.3.11 sets of keys

constdefs *keyset :: "msg set => bool"*
"keyset G == ALL X. X:G --> (EX K. X = Key K)"

lemma *keyset_in [dest]: "[| keyset G; X:G |] ==> EX K. X = Key K"*
⟨*proof*⟩

lemma *MPair_notin_keyset [simp]: "keyset G ==> {|X,Y|} ~:G"*
⟨*proof*⟩

lemma *Crypt_notin_keyset [simp]: "keyset G ==> Crypt K X ~:G"*
⟨*proof*⟩

lemma *Nonce_notin_keyset [simp]: "keyset G ==> Nonce n ~:G"*
⟨*proof*⟩

lemma *parts_keyset [simp]: "keyset G ==> parts G = G"*
⟨*proof*⟩

18.3.12 keys a priori necessary for decrypting the messages of G

constdefs *keysfor :: "msg set => msg set"*
"keysfor G == Key ‘ keysFor (parts G)"

lemma *keyset_keysfor [iff]: "keyset (keysfor G)"*
⟨*proof*⟩

lemma *keyset_Diff_keysfor [simp]: "keyset H ==> keyset (H - keysfor G)"*
⟨*proof*⟩

lemma *keysfor_Crypt: "Crypt K X:parts G ==> Key (invKey K):keysfor G"*
⟨*proof*⟩

lemma *no_key_no_Crypt: "Key K ~:keysfor G ==> Crypt (invKey K) X ~:parts G"*
⟨*proof*⟩

lemma *finite_keysfor [intro]: "finite G ==> finite (keysfor G)"*
⟨*proof*⟩

18.3.13 only the keys necessary for G are useful in analz

lemma *analz_keyset: "keyset H ==>*
analz (G Un H) = H - keysfor G Un (analz (G Un (H Int keysfor G)))"
⟨*proof*⟩

lemmas *analz_keyset_substD = analz_keyset [THEN sym, THEN ssubst]*

18.4 Extensions to Theory *Event*

18.4.1 general protocol properties

constdefs *is_Says :: "event => bool"*
"is_Says ev == (EX A B X. ev = Says A B X)"

lemma *is_Says_Says [iff]: "is_Says (Says A B X)"*
⟨*proof*⟩

constdefs *Gets_correct :: "event list set => bool"*
"Gets_correct p == ALL evs B X. evs:p --> Gets B X:set evs
--> (EX A. Says A B X:set evs)"

lemma *Gets_correct_Says: "[| Gets_correct p; Gets B X # evs:p |]*
==> EX A. Says A B X:set evs"
⟨*proof*⟩

constdefs *one_step :: "event list set => bool"*
"one_step p == ALL evs ev. ev#evs:p --> evs:p"

lemma *one_step_Cons [dest]: "[| one_step p; ev#evs:p |] ==> evs:p"*
⟨*proof*⟩

lemma *one_step_app: "[| evs@evs’:p; one_step p; []:p |] ==> evs’:p"*

⟨*proof*⟩

lemma *trunc: "[| evs @ evs':p; one_step p |] ==> evs':p"*
⟨*proof*⟩

constdefs *has_only_Says :: "event list set => bool"*
"has_only_Says p == ALL evs ev. evs:p --> ev:set evs
--> (EX A B X. ev = Says A B X)"

lemma *has_only_SaysD: "[| ev:set evs; evs:p; has_only_Says p |]*
==> EX A B X. ev = Says A B X"
⟨*proof*⟩

lemma *in_has_only_Says [dest]: "[| has_only_Says p; evs:p; ev:set evs |]*
==> EX A B X. ev = Says A B X"
⟨*proof*⟩

lemma *has_only_Says_imp_Gets_correct [simp]: "has_only_Says p*
==> Gets_correct p"
⟨*proof*⟩

18.4.2 lemma on knows

lemma *Says_imp_spies2: "Says A B {|X,Y|}:set evs ==> Y:parts (spies evs)"*
⟨*proof*⟩

lemma *Says_not_parts: "[| Says A B X:set evs; Y ~:parts (spies evs) |]*
==> Y ~:parts {X}"
⟨*proof*⟩

18.4.3 knows without initState

consts *knows' :: "agent => event list => msg set"*

primrec
knows'_Nil:
"knows' A [] = {}"

knows'_Cons0:
"knows' A (ev # evs) = (
if A = Spy then (
case ev of
Says A' B X => insert X (knows' A evs)
| Gets A' X => knows' A evs
| Notes A' X => if A':bad then insert X (knows' A evs) else knows' A evs
) else (
case ev of
Says A' B X => if A=A' then insert X (knows' A evs) else knows' A evs
| Gets A' X => if A=A' then insert X (knows' A evs) else knows' A evs
| Notes A' X => if A=A' then insert X (knows' A evs) else knows' A evs
))"

translations *"spies" == "knows Spy"*

syntax *spies' :: "event list => msg set"*

translations *"spies'" == "knows' Spy"*

18.4.4 decomposition of knows into knows' and initState

lemma *knows_decomp: "knows A evs = knows' A evs Un (initState A)"*
⟨*proof*⟩

lemmas *knows_decomp_substI = knows_decomp [THEN ssubst]*
lemmas *knows_decomp_substD = knows_decomp [THEN sym, THEN ssubst]*

lemma *knows'_sub_knows: "knows' A evs <= knows A evs"*
⟨*proof*⟩

lemma *knows'_Cons: "knows' A (ev#evs) = knows' A [ev] Un knows' A evs"*
⟨*proof*⟩

lemmas *knows'_Cons_substI = knows'_Cons [THEN ssubst]*
lemmas *knows'_Cons_substD = knows'_Cons [THEN sym, THEN ssubst]*

lemma *knows_Cons: "knows A (ev#evs) = initState A Un knows' A [ev]*
Un knows A evs"
⟨*proof*⟩

lemmas *knows_Cons_substI = knows_Cons [THEN ssubst]*
lemmas *knows_Cons_substD = knows_Cons [THEN sym, THEN ssubst]*

lemma *knows'_sub_spies': "[| evs:p; has_only_Says p; one_step p |]*
==> knows' A evs <= spies' evs"
⟨*proof*⟩

18.4.5 knows' is finite

lemma *finite_knows' [iff]: "finite (knows' A evs)"*
⟨*proof*⟩

18.4.6 monotonicity of knows

lemma *knows_sub_Cons: "knows A evs <= knows A (ev#evs)"*
⟨*proof*⟩

lemma *knows_ConsI: "X:knows A evs ==> X:knows A (ev#evs)"*
⟨*proof*⟩

lemma *knows_sub_app: "knows A evs <= knows A (evs @ evs')"*
⟨*proof*⟩

18.4.7 maximum knowledge an agent can have includes messages sent to the agent

consts *knows_max' :: "agent => event list => msg set"*

primrec
knows_max'_def_Nil: "knows_max' A [] = {}"

```
knows_max'_def_Cons: "knows_max' A (ev # evs) = (
  if A=Spy then (
    case ev of
      Says A' B X => insert X (knows_max' A evs)
    | Gets A' X => knows_max' A evs
    | Notes A' X =>
      if A':bad then insert X (knows_max' A evs) else knows_max' A evs
  ) else (
    case ev of
      Says A' B X =>
      if A=A' | A=B then insert X (knows_max' A evs) else knows_max' A evs
    | Gets A' X =>
      if A=A' then insert X (knows_max' A evs) else knows_max' A evs
    | Notes A' X =>
      if A=A' then insert X (knows_max' A evs) else knows_max' A evs
  ))"

constdefs knows_max :: "agent => event list => msg set"
"knows_max A evs == knows_max' A evs Un initState A"

consts spies_max :: "event list => msg set"

translations "spies_max evs" == "knows_max Spy evs"
```

18.4.8 basic facts about *knows_max*

lemma *spies_max_spies [iff]: "spies_max evs = spies evs"*
⟨*proof*⟩

lemma *knows_max'_Cons: "knows_max' A (ev#evs)*
= knows_max' A [ev] Un knows_max' A evs"
⟨*proof*⟩

lemmas *knows_max'_Cons_substI = knows_max'_Cons [THEN ssubst]*
lemmas *knows_max'_Cons_substD = knows_max'_Cons [THEN sym, THEN ssubst]*

lemma *knows_max_Cons: "knows_max A (ev#evs)*
= knows_max' A [ev] Un knows_max A evs"
⟨*proof*⟩

lemmas *knows_max_Cons_substI = knows_max_Cons [THEN ssubst]*
lemmas *knows_max_Cons_substD = knows_max_Cons [THEN sym, THEN ssubst]*

lemma *finite_knows_max' [iff]: "finite (knows_max' A evs)"*
⟨*proof*⟩

lemma *knows_max'_sub_spies': "[| evs:p; has_only_Says p; one_step p |]*
==> knows_max' A evs <= spies' evs"
⟨*proof*⟩

lemma *knows_max'_in_spies' [dest]: "[| evs:p; X:knows_max' A evs;*
has_only_Says p; one_step p |] ==> X:spies' evs"
⟨*proof*⟩

lemma *knows_max'_app: "knows_max' A (evs @ evs')*
= knows_max' A evs Un knows_max' A evs'"
⟨*proof*⟩

lemma *Says_to_knows_max': "Says A B X:set evs ==> X:knows_max' B evs"*
⟨*proof*⟩

lemma *Says_from_knows_max': "Says A B X:set evs ==> X:knows_max' A evs"*
⟨*proof*⟩

18.4.9 used without initState

consts *used' :: "event list => msg set"*

primrec
"used' [] = {}"
"used' (ev # evs) = (
case ev of
Says A B X => parts {X} Un used' evs
| Gets A X => used' evs
| Notes A X => parts {X} Un used' evs
)"

constdefs *init :: "msg set"*
"init == used []"

lemma *used_decomp: "used evs = init Un used' evs"*
⟨*proof*⟩

lemma *used'_sub_app: "used' evs <= used' (evs@evs')"*
⟨*proof*⟩

lemma *used'_parts [rule_format]: "X:used' evs ==> Y:parts {X} --> Y:used'*
evs"
⟨*proof*⟩

18.4.10 monotonicity of used

lemma *used_sub_Cons: "used evs <= used (ev#evs)"*
⟨*proof*⟩

lemma *used_ConsI: "X:used evs ==> X:used (ev#evs)"*
⟨*proof*⟩

lemma *notin_used_ConsD: "X ~:used (ev#evs) ==> X ~:used evs"*
⟨*proof*⟩

lemma *used_appD [dest]: "X:used (evs @ evs') ==> X:used evs | X:used evs'"*
⟨*proof*⟩

lemma *used_ConsD: "X:used (ev#evs) ==> X:used [ev] | X:used evs"*
⟨*proof*⟩

lemma *used_sub_app: "used evs <= used (evs@evs')"*
⟨*proof*⟩

lemma *used_appIL: "X:used evs ==> X:used (evs' @ evs)"*
⟨*proof*⟩

lemma *used_appIR: "X:used evs ==> X:used (evs @ evs')"*
⟨*proof*⟩

lemma *used_parts: "[| X:parts {Y}; Y:used evs |] ==> X:used evs"*
⟨*proof*⟩

lemma *parts_Says_used: "[| Says A B X:set evs; Y:parts {X} |] ==> Y:used evs"*
⟨*proof*⟩

lemma *parts_used_app: "X:parts {Y} ==> X:used (evs @ Says A B Y # evs')"*
⟨*proof*⟩

18.4.11 lemmas on used and knows

lemma *initState_used: "X:parts (initState A) ==> X:used evs"*
⟨*proof*⟩

lemma *Says_imp_used: "Says A B X:set evs ==> parts {X} <= used evs"*
⟨*proof*⟩

lemma *not_used_not_spied: "X ~:used evs ==> X ~:parts (spies evs)"*
⟨*proof*⟩

lemma *not_used_not_parts: "[| Y ~:used evs; Says A B X:set evs |]*
==> Y ~:parts {X}"
⟨*proof*⟩

lemma *not_used_parts_false: "[| X ~:used evs; Y:parts (spies evs) |]*
==> X ~:parts {Y}"
⟨*proof*⟩

lemma *known_used [rule_format]: "[| evs:p; Gets_correct p; one_step p |]*
==> X:parts (knows A evs) --> X:used evs"
⟨*proof*⟩

lemma *known_max_used [rule_format]: "[| evs:p; Gets_correct p; one_step p*
|]
==> X:parts (knows_max A evs) --> X:used evs"
⟨*proof*⟩

lemma *not_used_not_known: "[| evs:p; X ~:used evs;*
Gets_correct p; one_step p |] ==> X ~:parts (knows A evs)"
⟨*proof*⟩

lemma *not_used_not_known_max: "[| evs:p; X ~:used evs;*
Gets_correct p; one_step p |] ==> X ~:parts (knows_max A evs)"
⟨*proof*⟩

18.4.12 a nonce or key in a message cannot equal a fresh nonce or key

lemma *Nonce_neq [dest]: "[| Nonce n' ~:used evs;*
Says A B X:set evs; Nonce n:parts {X} |] ==> n ~= n'"
⟨*proof*⟩

lemma *Key_neq [dest]: "[| Key n' ~:used evs;*
Says A B X:set evs; Key n:parts {X} |] ==> n ~= n'"
⟨*proof*⟩

18.4.13 message of an event

consts *msg :: "event => msg"*

recdef *msg "measure size"*
"msg (Says A B X) = X"
"msg (Gets A X) = X"
"msg (Notes A X) = X"

lemma *used_sub_parts_used: "X:used (ev # evs) ==> X:parts {msg ev} Un used evs"*
⟨*proof*⟩

end

19 Decomposition of Analz into two parts

theory *Analz* **imports** *Extensions* **begin**

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

19.1 messages that do not contribute to analz

consts *pparts :: "msg set => msg set"*

inductive *"pparts H"*
intros
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"

19.2 basic facts about *pparts*

lemma *pparts_is_MPair [dest]: "X:pparts H ==> is_MPair X"*
⟨*proof*⟩

lemma *Crypt_notin_pparts [iff]: "Crypt K X ~:pparts H"*
⟨*proof*⟩

lemma *Key_notin_pparts [iff]: "Key K ~:pparts H"*

⟨*proof*⟩

lemma *Nonce_notin_pparts [iff]: "Nonce n ~:pparts H"*
⟨*proof*⟩

lemma *Number_notin_pparts [iff]: "Number n ~:pparts H"*
⟨*proof*⟩

lemma *Agent_notin_pparts [iff]: "Agent A ~:pparts H"*
⟨*proof*⟩

lemma *pparts_empty [iff]: "pparts {} = {}"*
⟨*proof*⟩

lemma *pparts_insertI [intro]: "X:pparts H ==> X:pparts (insert Y H)"*
⟨*proof*⟩

lemma *pparts_sub: "[| X:pparts G; G<=H |] ==> X:pparts H"*
⟨*proof*⟩

lemma *pparts_insert2 [iff]: "pparts (insert X (insert Y H))*
= pparts {X} Un pparts {Y} Un pparts H"
⟨*proof*⟩

lemma *pparts_insert_MPair [iff]: "pparts (insert {|X,Y|} H)*
= insert {|X,Y|} (pparts ({X,Y} Un H))"
⟨*proof*⟩

lemma *pparts_insert_Nonce [iff]: "pparts (insert (Nonce n) H) = pparts H"*
⟨*proof*⟩

lemma *pparts_insert_Crypt [iff]: "pparts (insert (Crypt K X) H) = pparts*
H"
⟨*proof*⟩

lemma *pparts_insert_Key [iff]: "pparts (insert (Key K) H) = pparts H"*
⟨*proof*⟩

lemma *pparts_insert_Agent [iff]: "pparts (insert (Agent A) H) = pparts H"*
⟨*proof*⟩

lemma *pparts_insert_Number [iff]: "pparts (insert (Number n) H) = pparts*
H"
⟨*proof*⟩

lemma *pparts_insert_Hash [iff]: "pparts (insert (Hash X) H) = pparts H"*
⟨*proof*⟩

lemma *pparts_insert: "X:pparts (insert Y H) ==> X:pparts {Y} Un pparts H"*
⟨*proof*⟩

lemma *insert_pparts: "X:pparts {Y} Un pparts H ==> X:pparts (insert Y H)"*
⟨*proof*⟩

lemma *pparts_Un [iff]: "pparts (G Un H) = pparts G Un pparts H"*
⟨*proof*⟩

lemma *pparts_pparts [iff]: "pparts (pparts H) = pparts H"*
⟨*proof*⟩

lemma *pparts_insert_eq: "pparts (insert X H) = pparts {X} Un pparts H"*
⟨*proof*⟩

lemmas *pparts_insert_substI = pparts_insert_eq [THEN ssubst]*

lemma *in_pparts: "Y:pparts H ==> EX X. X:H & Y:pparts {X}"*
⟨*proof*⟩

19.3 facts about *pparts* and *parts*

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

19.4 facts about *pparts* and *analz*

lemma *pparts_analz: "X:pparts H ==> X:analz H"*
⟨*proof*⟩

lemma *pparts_analz_sub: "[| X:pparts G; G<=H |] ==> X:analz H"*
⟨*proof*⟩

19.5 messages that contribute to analz

consts *kparts :: "msg set => msg set"*

inductive *"kparts H"*
intros
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"

19.6 basic facts about *kparts*

lemma *kparts_not_MPair [dest]: "X:kparts H ==> not_MPair X"*
⟨*proof*⟩

lemma *kparts_empty [iff]: "kparts {} = {}"*
⟨*proof*⟩

lemma *kparts_insertI [intro]: "X:kparts H ==> X:kparts (insert Y H)"*
⟨*proof*⟩

lemma *kparts_insert2 [iff]: "kparts (insert X (insert Y H))*
= kparts {X} Un kparts {Y} Un kparts H"
⟨*proof*⟩

lemma *kparts_insert_MPair [iff]: "kparts (insert {|X,Y|} H)*

```
= kparts ({X,Y} Un H)"
```
⟨*proof*⟩

lemma *kparts_insert_Nonce [iff]: "kparts (insert (Nonce n) H)*
= insert (Nonce n) (kparts H)"
⟨*proof*⟩

lemma *kparts_insert_Crypt [iff]: "kparts (insert (Crypt K X) H)*
= insert (Crypt K X) (kparts H)"
⟨*proof*⟩

lemma *kparts_insert_Key [iff]: "kparts (insert (Key K) H)*
= insert (Key K) (kparts H)"
⟨*proof*⟩

lemma *kparts_insert_Agent [iff]: "kparts (insert (Agent A) H)*
= insert (Agent A) (kparts H)"
⟨*proof*⟩

lemma *kparts_insert_Number [iff]: "kparts (insert (Number n) H)*
= insert (Number n) (kparts H)"
⟨*proof*⟩

lemma *kparts_insert_Hash [iff]: "kparts (insert (Hash X) H)*
= insert (Hash X) (kparts H)"
⟨*proof*⟩

lemma *kparts_insert: "X:kparts (insert X H) ==> X:kparts {X} Un kparts H"*
⟨*proof*⟩

lemma *kparts_insert_fst [rule_format,dest]: "X:kparts (insert Z H) ==>*
X ~:kparts H --> X:kparts {Z}"
⟨*proof*⟩

lemma *kparts_sub: "[| X:kparts G; G<=H |] ==> X:kparts H"*
⟨*proof*⟩

lemma *kparts_Un [iff]: "kparts (G Un H) = kparts G Un kparts H"*
⟨*proof*⟩

lemma *pparts_kparts [iff]: "pparts (kparts H) = {}"*
⟨*proof*⟩

lemma *kparts_kparts [iff]: "kparts (kparts H) = kparts H"*
⟨*proof*⟩

lemma *kparts_insert_eq: "kparts (insert X H) = kparts {X} Un kparts H"*
⟨*proof*⟩

lemmas *kparts_insert_substI = kparts_insert_eq [THEN ssubst]*

lemma *in_kparts: "Y:kparts H ==> EX X. X:H & Y:kparts {X}"*
⟨*proof*⟩

lemma *kparts_has_no_pair [iff]: "has_no_pair (kparts H)"*
⟨*proof*⟩

19.7 facts about *kparts* and *parts*

lemma *kparts_no_Nonce [dest]: "[| X:kparts {Y}; Nonce n ~:parts {Y} |] ==> Nonce n ~:parts {X}"*
⟨*proof*⟩

lemma *kparts_parts: "X:kparts H ==> X:parts H"*
⟨*proof*⟩

lemma *parts_kparts: "X:parts (kparts H) ==> X:parts H"*
⟨*proof*⟩

lemma *Crypt_kparts_Nonce_parts [dest]: "[| Crypt K Y:kparts {Z}; Nonce n:parts {Y} |] ==> Nonce n:parts {Z}"*
⟨*proof*⟩

19.8 facts about *kparts* and *analz*

lemma *kparts_analz: "X:kparts H ==> X:analz H"*
⟨*proof*⟩

lemma *kparts_analz_sub: "[| X:kparts G; G<=H |] ==> X:analz H"*
⟨*proof*⟩

lemma *analz_kparts [rule_format,dest]: "X:analz H ==> Y:kparts {X} --> Y:analz H"*
⟨*proof*⟩

lemma *analz_kparts_analz: "X:analz (kparts H) ==> X:analz H"*
⟨*proof*⟩

lemma *analz_kparts_insert: "X:analz (kparts (insert Z H)) ==> X:analz (kparts {Z} Un kparts H)"*
⟨*proof*⟩

lemma *Nonce_kparts_synth [rule_format]: "Y:synth (analz G) ==> Nonce n:kparts {Y} --> Nonce n:analz G"*
⟨*proof*⟩

lemma *kparts_insert_synth: "[| Y:parts (insert X G); X:synth (analz G); Nonce n:kparts {Y}; Nonce n ~:analz G |] ==> Y:parts G"*
⟨*proof*⟩

lemma *Crypt_insert_synth: "[| Crypt K Y:parts (insert X G); X:synth (analz G); Nonce n:kparts {Y}; Nonce n ~:analz G |] ==> Crypt K Y:parts G"*
⟨*proof*⟩

19.9 analz is pparts + analz of kparts

lemma *analz_pparts_kparts: "X:analz H ==> X:pparts H | X:analz (kparts H)"*

⟨*proof*⟩

lemma *analz_pparts_kparts_eq: "analz H = pparts H Un analz (kparts H)"*
⟨*proof*⟩

lemmas *analz_pparts_kparts_substI = analz_pparts_kparts_eq [THEN ssubst]*
lemmas *analz_pparts_kparts_substD*
= analz_pparts_kparts_eq [THEN sym, THEN ssubst]

end

20 Protocol-Independent Confidentiality Theorem on Nonces

theory *Guard* **imports** *Analz Extensions* **begin**

consts *guard :: "nat => key set => msg set"*

inductive *"guard n Ks"*
intros
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"

20.1 basic facts about *guard*

lemma *Key_is_guard [iff]: "Key K:guard n Ks"*
⟨*proof*⟩

lemma *Agent_is_guard [iff]: "Agent A:guard n Ks"*
⟨*proof*⟩

lemma *Number_is_guard [iff]: "Number r:guard n Ks"*
⟨*proof*⟩

lemma *Nonce_notin_guard: "X:guard n Ks ==> X ~= Nonce n"*
⟨*proof*⟩

lemma *Nonce_notin_guard_iff [iff]: "Nonce n ~:guard n Ks"*
⟨*proof*⟩

lemma *guard_has_Crypt [rule_format]: "X:guard n Ks ==> Nonce n:parts {X}*
--> (EX K Y. Crypt K Y:kparts {X} & Nonce n:parts {Y})"
⟨*proof*⟩

lemma *Nonce_notin_kparts_msg: "X:guard n Ks ==> Nonce n ~:kparts {X}"*
⟨*proof*⟩

lemma *Nonce_in_kparts_imp_no_guard: "Nonce n:kparts H*

```
==> EX X. X:H & X ~:guard n Ks"
```
⟨*proof*⟩

lemma *guard_kparts [rule_format]: "X:guard n Ks ==>*
Y:kparts {X} --> Y:guard n Ks"
⟨*proof*⟩

lemma *guard_Crypt: "[| Crypt K Y:guard n Ks; K ~:invKey‘Ks |] ==> Y:guard*
n Ks"
⟨*proof*⟩

lemma *guard_MPair [iff]: "({|X,Y|}:guard n Ks) = (X:guard n Ks & Y:guard*
n Ks)"
⟨*proof*⟩

lemma *guard_not_guard [rule_format]: "X:guard n Ks ==>*
Crypt K Y:kparts {X} --> Nonce n:kparts {Y} --> Y ~:guard n Ks"
⟨*proof*⟩

lemma *guard_extand: "[| X:guard n Ks; Ks <= Ks’ |] ==> X:guard n Ks’"*
⟨*proof*⟩

20.2 guarded sets

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

20.3 basic facts about *Guard*

lemma *Guard_empty [iff]: "Guard n Ks {}"*
⟨*proof*⟩

lemma *notin_parts_Guard [intro]: "Nonce n ~:parts G ==> Guard n Ks G"*
⟨*proof*⟩

lemma *Nonce_notin_kparts [simplified]: "Guard n Ks H ==> Nonce n ~:kparts*
H"
⟨*proof*⟩

lemma *Guard_must_decrypt: "[| Guard n Ks H; Nonce n:analz H |] ==>*
EX K Y. Crypt K Y:kparts H & Key (invKey K):kparts H"
⟨*proof*⟩

lemma *Guard_kparts [intro]: "Guard n Ks H ==> Guard n Ks (kparts H)"*
⟨*proof*⟩

lemma *Guard_mono: "[| Guard n Ks H; G <= H |] ==> Guard n Ks G"*
⟨*proof*⟩

lemma *Guard_insert [iff]: "Guard n Ks (insert X H)*
= (Guard n Ks H & X:guard n Ks)"
⟨*proof*⟩

lemma *Guard_Un [iff]: "Guard n Ks (G Un H) = (Guard n Ks G & Guard n Ks H)"*

⟨*proof*⟩

lemma *Guard_synth [intro]: "Guard n Ks G ==> Guard n Ks (synth G)"*
⟨*proof*⟩

lemma *Guard_analz [intro]: "[| Guard n Ks G; ALL K. K:Ks --> Key K ~:analz G |]*
==> Guard n Ks (analz G)"
⟨*proof*⟩

lemma *in_Guard [dest]: "[| X:G; Guard n Ks G |] ==> X:guard n Ks"*
⟨*proof*⟩

lemma *in_synth_Guard: "[| X:synth G; Guard n Ks G |] ==> X:guard n Ks"*
⟨*proof*⟩

lemma *in_analz_Guard: "[| X:analz G; Guard n Ks G;*
ALL K. K:Ks --> Key K ~:analz G |] ==> X:guard n Ks"
⟨*proof*⟩

lemma *Guard_keyset [simp]: "keyset G ==> Guard n Ks G"*
⟨*proof*⟩

lemma *Guard_Un_keyset: "[| Guard n Ks G; keyset H |] ==> Guard n Ks (G Un H)"*
⟨*proof*⟩

lemma *in_Guard_kparts: "[| X:G; Guard n Ks G; Y:kparts {X} |] ==> Y:guard n Ks"*
⟨*proof*⟩

lemma *in_Guard_kparts_neq: "[| X:G; Guard n Ks G; Nonce n':kparts {X} |]*
==> n ~= n'"
⟨*proof*⟩

lemma *in_Guard_kparts_Crypt: "[| X:G; Guard n Ks G; is_MPair X;*
Crypt K Y:kparts {X}; Nonce n:kparts {Y} |] ==> invKey K:Ks"
⟨*proof*⟩

lemma *Guard_extand: "[| Guard n Ks G; Ks <= Ks' |] ==> Guard n Ks' G"*
⟨*proof*⟩

lemma *guard_invKey [rule_format]: "[| X:guard n Ks; Nonce n:kparts {Y} |]*
==>
Crypt K Y:kparts {X} --> invKey K:Ks"
⟨*proof*⟩

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

20.4 set obtained by decrypting a message

syntax *decrypt :: "msg set => key => msg => msg set"*

translations *"decrypt H K Y" => "insert Y (H - {Crypt K Y})"*

lemma *analz_decrypt: "[| Crypt K Y:H; Key (invKey K):H; Nonce n:analz H |] ==> Nonce n:analz (decrypt H K Y)"*
⟨*proof*⟩

lemma *parts_decrypt: "[| Crypt K Y:H; X:parts (decrypt H K Y) |] ==> X:parts H"*
⟨*proof*⟩

20.5 number of Crypt's in a message

consts *crypt_nb :: "msg => nat"*

recdef *crypt_nb "measure size"*
"crypt_nb (Crypt K X) = Suc (crypt_nb X)"
"crypt_nb {|X,Y|} = crypt_nb X + crypt_nb Y"
"crypt_nb X = 0"

20.6 basic facts about *crypt_nb*

lemma *non_empty_crypt_msg: "Crypt K Y:parts {X} ==> 0 < crypt_nb X"*
⟨*proof*⟩

20.7 number of Crypt's in a message list

consts *cnb :: "msg list => nat"*

recdef *cnb "measure size"*
"cnb [] = 0"
"cnb (X#l) = crypt_nb X + cnb l"

20.8 basic facts about *cnb*

lemma *cnb_app [simp]: "cnb (l @ l') = cnb l + cnb l'"*
⟨*proof*⟩

lemma *mem_cnb_minus: "x mem l ==> cnb l = crypt_nb x + (cnb l - crypt_nb x)"*
⟨*proof*⟩

lemmas *mem_cnb_minus_substI = mem_cnb_minus [THEN ssubst]*

lemma *cnb_minus [simp]: "x mem l ==> cnb (minus l x) = cnb l - crypt_nb x"*
⟨*proof*⟩

lemma *parts_cnb: "Z:parts (set l) ==>*
cnb l = (cnb l - crypt_nb Z) + crypt_nb Z"
⟨*proof*⟩

lemma *non_empty_crypt: "Crypt K Y:parts (set l) ==> 0 < cnb l"*
⟨*proof*⟩

20.9 list of kparts

lemma *kparts_msg_set: "EX l. kparts {X} = set l & cnb l = crypt_nb X"*
⟨*proof*⟩

lemma *kparts_set: "EX l'. kparts (set l) = set l' & cnb l' = cnb l"*
⟨*proof*⟩

20.10 list corresponding to "decrypt"

constdefs *decrypt' :: "msg list => key => msg => msg list"*
"decrypt' l K Y == Y # minus l (Crypt K Y)"

declare *decrypt'_def [simp]*

20.11 basic facts about *decrypt'*

lemma *decrypt_minus: "decrypt (set l) K Y <= set (decrypt' l K Y)"*
⟨*proof*⟩

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

lemma *Guard_invKey_by_list [rule_format]: "ALL l. cnb l = p*
--> Guard n Ks (set l) --> Nonce n:analz (set l)
--> (EX K. K:Ks & Key K:analz (set l))"
⟨*proof*⟩

lemma *Guard_invKey_finite: "[| Nonce n:analz G; Guard n Ks G; finite G |]*
==> EX K. K:Ks & Key K:analz G"
⟨*proof*⟩

lemma *Guard_invKey: "[| Nonce n:analz G; Guard n Ks G |]*
==> EX K. K:Ks & Key K:analz G"
⟨*proof*⟩

20.13 if the analyse of a finite guarded set and a (possibly infinite) set of keys gives n then it must also gives Ks

lemma *Guard_invKey_keyset: "[| Nonce n:analz (G Un H); Guard n Ks G; finite G;*
keyset H |] ==> EX K. K:Ks & Key K:analz (G Un H)"
⟨*proof*⟩

end

theory *Guard_Public* **imports** *Guard Public Extensions* **begin**

20.14 Extensions to Theory *Public*

declare *initState.simps [simp del]*

20.14.1 signature

constdefs *sign :: "agent => msg => msg"*
"sign A X == {|Agent A, X, Crypt (priK A) (Hash X)|}"

lemma *sign_inj [iff]: "(sign A X = sign A' X') = (A=A' & X=X')"*
⟨*proof*⟩

20.14.2 agent associated to a key

constdefs *agt :: "key => agent"*
"agt K == @A. K = priK A | K = pubK A"

lemma *agt_priK [simp]: "agt (priK A) = A"*
⟨*proof*⟩

lemma *agt_pubK [simp]: "agt (pubK A) = A"*
⟨*proof*⟩

20.14.3 basic facts about *initState*

lemma *no_Crypt_in_parts_init [simp]: "Crypt K X ~:parts (initState A)"*
⟨*proof*⟩

lemma *no_Crypt_in_analz_init [simp]: "Crypt K X ~:analz (initState A)"*
⟨*proof*⟩

lemma *no_priK_in_analz_init [simp]: "A ~:bad*
==> Key (priK A) ~:analz (initState Spy)"
⟨*proof*⟩

lemma *priK_notin_initState_Friend [simp]: "A ~= Friend C*
==> Key (priK A) ~: parts (initState (Friend C))"
⟨*proof*⟩

lemma *keyset_init [iff]: "keyset (initState A)"*
⟨*proof*⟩

20.14.4 sets of private keys

constdefs *priK_set :: "key set => bool"*
"priK_set Ks == ALL K. K:Ks --> (EX A. K = priK A)"

lemma *in_priK_set: "[| priK_set Ks; K:Ks |] ==> EX A. K = priK A"*
⟨*proof*⟩

lemma *priK_set1 [iff]: "priK_set {priK A}"*
⟨*proof*⟩

lemma *priK_set2 [iff]: "priK_set {priK A, priK B}"*
⟨*proof*⟩

20.14.5 sets of good keys

constdefs *good :: "key set => bool"*
"good Ks == ALL K. K:Ks --> agt K ~:bad"

lemma *in_good: "[| good Ks; K:Ks |] ==> agt K ~:bad"*
⟨*proof*⟩

lemma *good1 [simp]: "A ~:bad ==> good {priK A}"*
⟨*proof*⟩

lemma *good2 [simp]: "[| A ~:bad; B ~:bad |] ==> good {priK A, priK B}"*
⟨*proof*⟩

20.14.6 greatest nonce used in a trace, 0 if there is no nonce

consts *greatest :: "event list => nat"*

recdef *greatest "measure size"*
"greatest [] = 0"
"greatest (ev # evs) = max (greatest_msg (msg ev)) (greatest evs)"

lemma *greatest_is_greatest: "Nonce n:used evs ==> n <= greatest evs"*
⟨*proof*⟩

20.14.7 function giving a new nonce

constdefs *new :: "event list => nat"*
"new evs == Suc (greatest evs)"

lemma *new_isnt_used [iff]: "Nonce (new evs) ~:used evs"*
⟨*proof*⟩

20.15 Proofs About Guarded Messages

20.15.1 small hack necessary because priK is defined as the inverse of pubK

lemma *pubK_is_invKey_priK: "pubK A = invKey (priK A)"*
⟨*proof*⟩

lemmas *pubK_is_invKey_priK_substI = pubK_is_invKey_priK [THEN ssubst]*

lemmas *invKey_invKey_substI = invKey [THEN ssubst]*

lemma *"Nonce n:parts {X} ==> Crypt (pubK A) X:guard n {priK A}"*
⟨*proof*⟩

20.15.2 guardedness results

lemma *sign_guard [intro]: "X:guard n Ks ==> sign A X:guard n Ks"*
⟨*proof*⟩

lemma *Guard_init [iff]: "Guard n Ks (initState B)"*
⟨*proof*⟩

lemma *Guard_knows_max': "Guard n Ks (knows_max' C evs)*
==> Guard n Ks (knows_max C evs)"
⟨*proof*⟩

lemma *Nonce_not_used_Guard_spies [dest]: "Nonce n ~:used evs ==> Guard n Ks (spies evs)"*
⟨*proof*⟩

lemma *Nonce_not_used_Guard [dest]: "[| evs:p; Nonce n ~:used evs; Gets_correct p; one_step p |] ==> Guard n Ks (knows (Friend C) evs)"*
⟨*proof*⟩

lemma *Nonce_not_used_Guard_max [dest]: "[| evs:p; Nonce n ~:used evs; Gets_correct p; one_step p |] ==> Guard n Ks (knows_max (Friend C) evs)"*
⟨*proof*⟩

lemma *Nonce_not_used_Guard_max' [dest]: "[| evs:p; Nonce n ~:used evs; Gets_correct p; one_step p |] ==> Guard n Ks (knows_max' (Friend C) evs)"*
⟨*proof*⟩

20.15.3 regular protocols

constdefs *regular :: "event list set => bool" "regular p == ALL evs A. evs:p --> (Key (priK A):parts (spies evs)) = (A:bad)"*

lemma *priK_parts_iff_bad [simp]: "[| evs:p; regular p |] ==> (Key (priK A):parts (spies evs)) = (A:bad)"*
⟨*proof*⟩

lemma *priK_analz_iff_bad [simp]: "[| evs:p; regular p |] ==> (Key (priK A):analz (spies evs)) = (A:bad)"*
⟨*proof*⟩

lemma *Guard_Nonce_analz: "[| Guard n Ks (spies evs); evs:p; priK_set Ks; good Ks; regular p |] ==> Nonce n ~:analz (spies evs)"*
⟨*proof*⟩

end

21 Lists of Messages and Lists of Agents

theory *List_Msg* **imports** *Extensions* **begin**

21.1 Implementation of Lists by Messages

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

syntax *cons :: "msg => msg => msg"*

translations *"cons x l" => "{|x,l|}"*

21.1.2 induction principle

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

21.1.3 head

consts *head :: "msg => msg"*

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

21.1.4 tail

consts *tail :: "msg => msg"*

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

21.1.5 length

consts *len :: "msg => nat"*

recdef *len "measure size"*
"len (cons x l) = Suc (len l)"
"len other = 0"

lemma *len_not_empty: "n < len l ==> EX x l'. l = cons x l'"*
⟨*proof*⟩

21.1.6 membership

consts *isin :: "msg * msg => bool"*

recdef *isin "measure (%(x,l). size l)"*
"isin (x, cons y l) = (x=y | isin (x,l))"
"isin (x, other) = False"

21.1.7 delete an element

consts *del :: "msg * msg => msg"*

recdef *del "measure (%(x,l). size l)"*
"del (x, cons y l) = (if x=y then l else cons y (del (x,l)))"
"del (x, other) = other"

lemma *notin_del [simp]: "~ isin (x,l) ==> del (x,l) = l"*
⟨*proof*⟩

lemma *isin_del [rule_format]: "isin (y, del (x,l)) --> isin (y,l)"*
⟨*proof*⟩

21.1.8 concatenation

consts *app :: "msg * msg => msg"*

recdef *app "measure (%(l,l'). size l)"*
"app (cons x l, l') = cons x (app (l,l'))"
"app (other, l') = l'"

lemma *isin_app [iff]: "isin (x, app(l,l')) = (isin (x,l) | isin (x,l'))"*

⟨*proof*⟩

21.1.9 replacement

consts *repl :: "msg * nat * msg => msg"*

recdef *repl "measure (%(l,i,x'). i)"*
"repl (cons x l, Suc i, x') = cons x (repl (l,i,x'))"
"repl (cons x l, 0, x') = cons x' l"
"repl (other, i, M') = other"

21.1.10 ith element

consts *ith :: "msg * nat => msg"*

recdef *ith "measure (%(l,i). i)"*
"ith (cons x l, Suc i) = ith (l,i)"
"ith (cons x l, 0) = x"
"ith (other, i) = other"

lemma *ith_head: "0 < len l ==> ith (l,0) = head l"*
⟨*proof*⟩

21.1.11 insertion

consts *ins :: "msg * nat * msg => msg"*

recdef *ins "measure (%(l,i,x). i)"*
"ins (cons x l, Suc i, y) = cons x (ins (l,i,y))"
"ins (l, 0, y) = cons y l"

lemma *ins_head [simp]: "ins (l,0,y) = cons y l"*
⟨*proof*⟩

21.1.12 truncation

consts *trunc :: "msg * nat => msg"*

recdef *trunc "measure (%(l,i). i)"*
"trunc (l,0) = l"
"trunc (cons x l, Suc i) = trunc (l,i)"

lemma *trunc_zero [simp]: "trunc (l,0) = l"*
⟨*proof*⟩

21.2 Agent Lists

21.2.1 set of well-formed agent-list messages

syntax *nil :: msg*

translations *"nil" == "Number 0"*

consts *agl :: "msg set"*

inductive *agl*
intros
Nil[intro]: "nil:agl"
Cons[intro]: "[| A:agent; I:agl |] ==> cons (Agent A) I :agl"

21.2.2 basic facts about agent lists

lemma *del_in_agl [intro]: "I:agl ==> del (a,I):agl"*
⟨*proof*⟩

lemma *app_in_agl [intro]: "[| I:agl; J:agl |] ==> app (I,J):agl"*
⟨*proof*⟩

lemma *no_Key_in_agl: "I:agl ==> Key K ~:parts {I}"*
⟨*proof*⟩

lemma *no_Nonce_in_agl: "I:agl ==> Nonce n ~:parts {I}"*
⟨*proof*⟩

lemma *no_Key_in_appdel: "[| I:agl; J:agl |] ==>*
Key K ~:parts {app (J, del (Agent B, I))}"
⟨*proof*⟩

lemma *no_Nonce_in_appdel: "[| I:agl; J:agl |] ==>*
Nonce n ~:parts {app (J, del (Agent B, I))}"
⟨*proof*⟩

lemma *no_Crypt_in_agl: "I:agl ==> Crypt K X ~:parts {I}"*
⟨*proof*⟩

lemma *no_Crypt_in_appdel: "[| I:agl; J:agl |] ==>*
Crypt K X ~:parts {app (J, del (Agent B,I))}"
⟨*proof*⟩

end

22 Protocol P1

theory *P1* **imports** *Guard_Public List_Msg* **begin**

22.1 Protocol Definition

22.1.1 offer chaining: B chains his offer for A with the head offer of L for sending it to C

constdefs *chain :: "agent => nat => agent => msg => agent => msg"*
"chain B ofr A L C ==
let m1= Crypt (pubK A) (Nonce ofr) in
let m2= Hash {|head L, Agent C|} in
sign B {|m1,m2|}"

declare *Let_def [simp]*

lemma *chain_inj [iff]: "(chain B ofr A L C = chain B' ofr' A' L' C')*
= (B=B' & ofr=ofr' & A=A' & head L = head L' & C=C')"
⟨*proof*⟩

lemma *Nonce_in_chain [iff]: "Nonce ofr:parts {chain B ofr A L C}"*
⟨*proof*⟩

22.1.2 agent whose key is used to sign an offer

consts *shop :: "msg => msg"*

recdef *shop "measure size"*
"shop {|B,X,Crypt K H|} = Agent (agt K)"

lemma *shop_chain [simp]: "shop (chain B ofr A L C) = Agent B"*
⟨*proof*⟩

22.1.3 nonce used in an offer

consts *nonce :: "msg => msg"*

recdef *nonce "measure size"*
"nonce {|B,{|Crypt K ofr,m2|},CryptH|} = ofr"

lemma *nonce_chain [simp]: "nonce (chain B ofr A L C) = Nonce ofr"*
⟨*proof*⟩

22.1.4 next shop

consts *next_shop :: "msg => agent"*

recdef *next_shop "measure size"*
"next_shop {|B,{|m1,Hash{|headL,Agent C|}|},CryptH|} = C"

lemma *next_shop_chain [iff]: "next_shop (chain B ofr A L C) = C"*
⟨*proof*⟩

22.1.5 anchor of the offer list

constdefs *anchor :: "agent => nat => agent => msg"*
"anchor A n B == chain A n A (cons nil nil) B"

lemma *anchor_inj [iff]: "(anchor A n B = anchor A' n' B')*
= (A=A' & n=n' & B=B')"
⟨*proof*⟩

lemma *Nonce_in_anchor [iff]: "Nonce n:parts {anchor A n B}"*
⟨*proof*⟩

lemma *shop_anchor [simp]: "shop (anchor A n B) = Agent A"*
⟨*proof*⟩

lemma *nonce_anchor [simp]: "nonce (anchor A n B) = Nonce n"*
⟨*proof*⟩

lemma *next_shop_anchor [iff]: "next_shop (anchor A n B) = B"*
⟨*proof*⟩

22.1.6 request event

constdefs *reqm :: "agent => nat => nat => msg => agent => msg"*
"reqm A r n I B == {|Agent A, Number r, cons (Agent A) (cons (Agent B) I), cons (anchor A n B) nil|}"

lemma *reqm_inj [iff]: "(reqm A r n I B = reqm A' r' n' I' B')*
= (A=A' & r=r' & n=n' & I=I' & B=B')"
⟨*proof*⟩

lemma *Nonce_in_reqm [iff]: "Nonce n:parts {reqm A r n I B}"*
⟨*proof*⟩

constdefs *req :: "agent => nat => nat => msg => agent => event"*
"req A r n I B == Says A B (reqm A r n I B)"

lemma *req_inj [iff]: "(req A r n I B = req A' r' n' I' B')*
= (A=A' & r=r' & n=n' & I=I' & B=B')"
⟨*proof*⟩

22.1.7 propose event

constdefs *prom :: "agent => nat => agent => nat => msg => msg => msg => agent => msg"*
"prom B ofr A r I L J C == {|Agent A, Number r, app (J, del (Agent B, I)), cons (chain B ofr A L C) L|}"

lemma *prom_inj [dest]: "prom B ofr A r I L J C*
= prom B' ofr' A' r' I' L' J' C'
==> B=B' & ofr=ofr' & A=A' & r=r' & L=L' & C=C'"
⟨*proof*⟩

lemma *Nonce_in_prom [iff]: "Nonce ofr:parts {prom B ofr A r I L J C}"*
⟨*proof*⟩

constdefs *pro :: "agent => nat => agent => nat => msg => msg => msg => agent => event"*
"pro B ofr A r I L J C == Says B C (prom B ofr A r I L J C)"

lemma *pro_inj [dest]: "pro B ofr A r I L J C = pro B' ofr' A' r' I' L' J' C'*
==> B=B' & ofr=ofr' & A=A' & r=r' & L=L' & C=C'"
⟨*proof*⟩

22.1.8 protocol

consts *p1 :: "event list set"*

inductive *p1*
intros

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"
```

22.1.9 Composition of Traces

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

22.1.10 Valid Offer Lists

consts *valid :: "agent => nat => agent => msg set"*

inductive *"valid A n B"*
intros
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"

22.1.11 basic properties of valid

lemma *valid_not_empty: "L:valid A n B ==> EX M L'. L = cons M L'"*
⟨*proof*⟩

lemma *valid_pos_len: "L:valid A n B ==> 0 < len L"*
⟨*proof*⟩

22.1.12 offers of an offer list

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

22.1.13 the originator can get the offers

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

22.1.14 list of offers

consts *offers :: "msg => msg"*

recdef *offers "measure size"*
"offers (cons M L) = cons {|shop M, nonce M|} (offers L)"
"offers other = nil"

22.1.15 list of agents whose keys are used to sign a list of offers

consts *shops :: "msg => msg"*

recdef *shops "measure size"*
"shops (cons M L) = cons (shop M) (shops L)"
"shops other = other"

lemma *shops_in_agl: "L:valid A n B ==> shops L:agl"*
⟨*proof*⟩

22.1.16 builds a trace from an itinerary

consts *offer_list :: "agent * nat * agent * msg * nat => msg"*

recdef *offer_list "measure (%(A,n,B,I,ofr). size I)"*
"offer_list (A,n,B,nil,ofr) = cons (anchor A n B) nil"
"offer_list (A,n,B,cons (Agent C) I,ofr) = (
let L = offer_list (A,n,B,I,Suc ofr) in
cons (chain (next_shop (head L)) ofr A L C) L)"

lemma *"I:agl ==> ALL ofr. offer_list (A,n,B,I,ofr):valid A n B"*
⟨*proof*⟩

consts *trace :: "agent * nat * agent * nat * msg * msg * msg*
=> event list"

recdef *trace "measure (%(B,ofr,A,r,I,L,K). size K)"*
"trace (B,ofr,A,r,I,L,nil) = []"
"trace (B,ofr,A,r,I,L,cons (Agent D) K) = (
let C = (if K=nil then B else agt_nb (head K)) in
let I' = (if K=nil then cons (Agent A) (cons (Agent B) I)
else cons (Agent A) (app (I, cons (head K) nil))) in
let I'' = app (I, cons (head K) nil) in
pro C (Suc ofr) A r I' L nil D
 # trace (B,Suc ofr,A,r,I'',tail L,K))"

constdefs *trace' :: "agent => nat => nat => msg => agent => nat => event list"*
"trace' A r n I B ofr == (
let AI = cons (Agent A) I in
let L = offer_list (A,n,B,AI,ofr) in
trace (B,ofr,A,r,nil,L,AI))"

declare *trace'_def [simp]*

22.1.17 there is a trace in which the originator receives a valid answer

lemma *p1_not_empty: "evs:p1 ==> req A r n I B:set evs -->*
(EX evs'. evs'@evs:p1 & pro B' ofr A r I' L J A:set evs' & L:valid A n B)"
⟨*proof*⟩

22.2 properties of protocol P1

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

22.2.1 strong forward integrity: except the last one, no offer can be modified

lemma *strong_forward_integrity: "ALL L. Suc i < len L*
--> L:valid A n B & repl (L,Suc i,M):valid A n B --> M = ith (L,Suc i)"
⟨*proof*⟩

22.2.2 insertion resilience: except at the beginning, no offer can be inserted

lemma *chain_isnt_head [simp]: "L:valid A n B ==>*
head L ~= chain (next_shop (head L)) ofr A L C"
⟨*proof*⟩

lemma *insertion_resilience: "ALL L. L:valid A n B --> Suc i < len L*
--> ins (L,Suc i,M) ~:valid A n B"
⟨*proof*⟩

22.2.3 truncation resilience: only shop i can truncate at offer i

lemma *truncation_resilience: "ALL L. L:valid A n B --> Suc i < len L*
--> cons M (trunc (L,Suc i)):valid A n B --> shop M = shop (ith (L,i))"
⟨*proof*⟩

22.2.4 declarations for tactics

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

22.2.5 get components of a message

lemma *get_ML [dest]: "Says A' B {|A,r,I,M,L|}:set evs ==>*
M:parts (spies evs) & L:parts (spies evs)"
⟨*proof*⟩

22.2.6 general properties of p1

lemma *reqm_neq_prom [iff]:*
"reqm A r n I B ~= prom B' ofr A' r' I' (cons M L) J C"
⟨*proof*⟩

lemma *prom_neq_reqm [iff]:*
"prom B' ofr A' r' I' (cons M L) J C ~= reqm A r n I B"
⟨*proof*⟩

lemma *req_neq_pro [iff]: "req A r n I B ~= pro B' ofr A' r' I' (cons M L)*
J C"
⟨*proof*⟩

lemma *pro_neq_req [iff]: "pro B' ofr A' r' I' (cons M L) J C ~= req A r n I B"*
⟨*proof*⟩

lemma *p1_has_no_Gets: "evs:p1 ==> ALL A X. Gets A X ~:set evs"*
⟨*proof*⟩

lemma *p1_is_Gets_correct [iff]: "Gets_correct p1"*
⟨*proof*⟩

lemma *p1_is_one_step [iff]: "one_step p1"*
⟨*proof*⟩

lemma *p1_has_only_Says' [rule_format]: "evs:p1 ==> ev:set evs --> (EX A B X. ev=Says A B X)"*
⟨*proof*⟩

lemma *p1_has_only_Says [iff]: "has_only_Says p1"*
⟨*proof*⟩

lemma *p1_is_regular [iff]: "regular p1"*
⟨*proof*⟩

22.2.7 private keys are safe

lemma *priK_parts_Friend_imp_bad [rule_format,dest]:*
"[| evs:p1; Friend B ~= A |] ==> (Key (priK A):parts (knows (Friend B) evs)) --> (A:bad)"
⟨*proof*⟩

lemma *priK_analz_Friend_imp_bad [rule_format,dest]:*
"[| evs:p1; Friend B ~= A |] ==> (Key (priK A):analz (knows (Friend B) evs)) --> (A:bad)"
⟨*proof*⟩

lemma *priK_notin_knows_max_Friend: "[| evs:p1; A ~:bad; A ~= Friend C |] ==> Key (priK A) ~:analz (knows_max (Friend C) evs)"*
⟨*proof*⟩

22.2.8 general guardedness properties

lemma *agl_guard [intro]: "I:agl ==> I:guard n Ks"*
⟨*proof*⟩

lemma *Says_to_knows_max'_guard: "[| Says A' C {|A'',r,I,L|}:set evs; Guard n Ks (knows_max' C evs) |] ==> L:guard n Ks"*
⟨*proof*⟩

lemma *Says_from_knows_max'_guard: "[| Says C A' {|A'',r,I,L|}:set evs; Guard n Ks (knows_max' C evs) |] ==> L:guard n Ks"*
⟨*proof*⟩

lemma *Says_Nonce_not_used_guard: "[| Says A' B {|A'',r,I,L|}:set evs; Nonce n ~:used evs |] ==> L:guard n Ks"*
⟨*proof*⟩

22.2.9 guardedness of messages

lemma *chain_guard [iff]: "chain B ofr A L C:guard n {priK A}"*
⟨*proof*⟩

lemma *chain_guard_Nonce_neq [intro]: "n ~= ofr*
==> chain B ofr A' L C:guard n {priK A}"
⟨*proof*⟩

lemma *anchor_guard [iff]: "anchor A n' B:guard n {priK A}"*
⟨*proof*⟩

lemma *anchor_guard_Nonce_neq [intro]: "n ~= n'*
==> anchor A' n' B:guard n {priK A}"
⟨*proof*⟩

lemma *reqm_guard [intro]: "I:agl ==> reqm A r n' I B:guard n {priK A}"*
⟨*proof*⟩

lemma *reqm_guard_Nonce_neq [intro]: "[| n ~= n'; I:agl |]*
==> reqm A' r n' I B:guard n {priK A}"
⟨*proof*⟩

lemma *prom_guard [intro]: "[| I:agl; J:agl; L:guard n {priK A} |]*
==> prom B ofr A r I L J C:guard n {priK A}"
⟨*proof*⟩

lemma *prom_guard_Nonce_neq [intro]: "[| n ~= ofr; I:agl; J:agl;*
L:guard n {priK A} |] ==> prom B ofr A' r I L J C:guard n {priK A}"
⟨*proof*⟩

22.2.10 Nonce uniqueness

lemma *uniq_Nonce_in_chain [dest]: "Nonce k:parts {chain B ofr A L C} ==>*
k=ofr"
⟨*proof*⟩

lemma *uniq_Nonce_in_anchor [dest]: "Nonce k:parts {anchor A n B} ==> k=n"*
⟨*proof*⟩

lemma *uniq_Nonce_in_reqm [dest]: "[| Nonce k:parts {reqm A r n I B};*
I:agl |] ==> k=n"
⟨*proof*⟩

lemma *uniq_Nonce_in_prom [dest]: "[| Nonce k:parts {prom B ofr A r I L J*
C};
I:agl; J:agl; Nonce k ~:parts {L} |] ==> k=ofr"
⟨*proof*⟩

22.2.11 requests are guarded

lemma *req_imp_Guard [rule_format]: "[| evs:p1; A ~:bad |] ==>*
req A r n I B:set evs --> Guard n {priK A} (spies evs)"
⟨*proof*⟩

lemma *req_imp_Guard_Friend: "[| evs:p1; A ~:bad; req A r n I B:set evs |]*
==> Guard n {priK A} (knows_max (Friend C) evs)"
⟨*proof*⟩

22.2.12 propositions are guarded

lemma *pro_imp_Guard [rule_format]: "[| evs:p1; B ~:bad; A ~:bad |] ==>*
pro B ofr A r I (cons M L) J C:set evs --> Guard ofr {priK A} (spies evs)"
⟨*proof*⟩

lemma *pro_imp_Guard_Friend: "[| evs:p1; B ~:bad; A ~:bad;*
pro B ofr A r I (cons M L) J C:set evs |]
==> Guard ofr {priK A} (knows_max (Friend D) evs)"
⟨*proof*⟩

22.2.13 data confidentiality: no one other than the originator can decrypt the offers

lemma *Nonce_req_notin_spies: "[| evs:p1; req A r n I B:set evs; A ~:bad |]*
==> Nonce n ~:analz (spies evs)"
⟨*proof*⟩

lemma *Nonce_req_notin_knows_max_Friend: "[| evs:p1; req A r n I B:set evs;*
A ~:bad; A ~= Friend C |] ==> Nonce n ~:analz (knows_max (Friend C) evs)"
⟨*proof*⟩

lemma *Nonce_pro_notin_spies: "[| evs:p1; B ~:bad; A ~:bad;*
pro B ofr A r I (cons M L) J C:set evs |] ==> Nonce ofr ~:analz (spies evs)"
⟨*proof*⟩

lemma *Nonce_pro_notin_knows_max_Friend: "[| evs:p1; B ~:bad; A ~:bad;*
A ~= Friend D; pro B ofr A r I (cons M L) J C:set evs |]
==> Nonce ofr ~:analz (knows_max (Friend D) evs)"
⟨*proof*⟩

22.2.14 non repudiability: an offer signed by B has been sent by B

lemma *Crypt_reqm: "[| Crypt (priK A) X:parts {reqm A' r n I B}; I:agl |]*
==> A=A'"
⟨*proof*⟩

lemma *Crypt_prom: "[| Crypt (priK A) X:parts {prom B ofr A' r I L J C};*
I:agl; J:agl |] ==> A=B | Crypt (priK A) X:parts {L}"
⟨*proof*⟩

lemma *Crypt_safeness: "[| evs:p1; A ~:bad |] ==> Crypt (priK A) X:parts (spies*
evs)
--> (EX B Y. Says A B Y:set evs & Crypt (priK A) X:parts {Y})"
⟨*proof*⟩

lemma *Crypt_Hash_imp_sign: "[| evs:p1; A ~:bad |] ==>*
Crypt (priK A) (Hash X):parts (spies evs)
--> (EX B Y. Says A B Y:set evs & sign A X:parts {Y})"
⟨*proof*⟩

lemma *sign_safeness: "[| evs:p1; A ~:bad |] ==> sign A X:parts (spies evs)*
--> (EX B Y. Says A B Y:set evs & sign A X:parts {Y})"
⟨*proof*⟩

end

23 Protocol P2

theory *P2* **imports** *Guard_Public List_Msg* **begin**

23.1 Protocol Definition

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

23.1.1 offer chaining: B chains his offer for A with the head offer of L for sending it to C

constdefs *chain :: "agent => nat => agent => msg => agent => msg"*
"chain B ofr A L C ==
let m1= sign B (Nonce ofr) in
let m2= Hash {|head L, Agent C|} in
{|Crypt (pubK A) m1, m2|}"

declare *Let_def [simp]*

lemma *chain_inj [iff]: "(chain B ofr A L C = chain B' ofr' A' L' C')*
= (B=B' & ofr=ofr' & A=A' & head L = head L' & C=C')"
⟨*proof*⟩

lemma *Nonce_in_chain [iff]: "Nonce ofr:parts {chain B ofr A L C}"*
⟨*proof*⟩

23.1.2 agent whose key is used to sign an offer

consts *shop :: "msg => msg"*

recdef *shop "measure size"*
"shop {|Crypt K {|B,ofr,Crypt K' H|},m2|} = Agent (agt K')"

lemma *shop_chain [simp]: "shop (chain B ofr A L C) = Agent B"*
⟨*proof*⟩

23.1.3 nonce used in an offer

consts *nonce :: "msg => msg"*

recdef *nonce "measure size"*
"nonce {|Crypt K {|B,ofr,CryptH|},m2|} = ofr"

lemma *nonce_chain [simp]: "nonce (chain B ofr A L C) = Nonce ofr"*
⟨*proof*⟩

23.1.4 next shop

consts *next_shop :: "msg => agent"*

recdef *next_shop "measure size"*
"next_shop {|m1,Hash {|headL,Agent C|}|} = C"

lemma *"next_shop (chain B ofr A L C) = C"*
⟨*proof*⟩

23.1.5 anchor of the offer list

constdefs *anchor :: "agent => nat => agent => msg"*
"anchor A n B == chain A n A (cons nil nil) B"

lemma *anchor_inj [iff]:*
"(anchor A n B = anchor A' n' B') = (A=A' & n=n' & B=B')"
⟨*proof*⟩

lemma *Nonce_in_anchor [iff]: "Nonce n:parts {anchor A n B}"*
⟨*proof*⟩

lemma *shop_anchor [simp]: "shop (anchor A n B) = Agent A"*
⟨*proof*⟩

23.1.6 request event

constdefs *reqm :: "agent => nat => nat => msg => agent => msg"*
"reqm A r n I B == {|Agent A, Number r, cons (Agent A) (cons (Agent B) I),
cons (anchor A n B) nil|}"

lemma *reqm_inj [iff]: "(reqm A r n I B = reqm A' r' n' I' B')*
= (A=A' & r=r' & n=n' & I=I' & B=B')"
⟨*proof*⟩

lemma *Nonce_in_reqm [iff]: "Nonce n:parts {reqm A r n I B}"*
⟨*proof*⟩

constdefs *req :: "agent => nat => nat => msg => agent => event"*
"req A r n I B == Says A B (reqm A r n I B)"

lemma *req_inj [iff]: "(req A r n I B = req A' r' n' I' B')*
= (A=A' & r=r' & n=n' & I=I' & B=B')"
⟨*proof*⟩

23.1.7 propose event

constdefs *prom :: "agent => nat => agent => nat => msg => msg =>*
msg => agent => msg"
"prom B ofr A r I L J C == {|Agent A, Number r,
app (J, del (Agent B, I)), cons (chain B ofr A L C) L|}"

lemma *prom_inj [dest]: "prom B ofr A r I L J C = prom B' ofr' A' r' I' L'*
J' C'
==> B=B' & ofr=ofr' & A=A' & r=r' & L=L' & C=C'"

⟨*proof*⟩

lemma *Nonce_in_prom [iff]: "Nonce ofr:parts {prom B ofr A r I L J C}"*
⟨*proof*⟩

constdefs *pro :: "agent => nat => agent => nat => msg => msg =>*
msg => agent => event"
"pro B ofr A r I L J C == Says B C (prom B ofr A r I L J C)"

lemma *pro_inj [dest]: "pro B ofr A r I L J C = pro B' ofr' A' r' I' L' J' C'*
==> B=B' & ofr=ofr' & A=A' & r=r' & L=L' & C=C'"
⟨*proof*⟩

23.1.8 protocol

consts *p2 :: "event list set"*

inductive *p2*
intros

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"

23.1.9 valid offer lists

consts *valid :: "agent => nat => agent => msg set"*

inductive *"valid A n B"*
intros
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"

23.1.10 basic properties of valid

lemma *valid_not_empty: "L:valid A n B ==> EX M L'. L = cons M L'"*
⟨*proof*⟩

lemma *valid_pos_len: "L:valid A n B ==> 0 < len L"*
⟨*proof*⟩

23.1.11 list of offers

consts *offers :: "msg => msg"*

recdef *offers "measure size"*
"offers (cons M L) = cons {|shop M, nonce M|} (offers L)"
"offers other = nil"

23.2 Properties of Protocol P2

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

23.3 strong forward integrity: except the last one, no offer can be modified

lemma *strong_forward_integrity: "ALL L. Suc i < len L*
--> L:valid A n B --> repl (L,Suc i,M):valid A n B --> M = ith (L,Suc i)"
⟨*proof*⟩

23.4 insertion resilience: except at the beginning, no offer can be inserted

lemma *chain_isnt_head [simp]: "L:valid A n B ==>*
head L ~= chain (next_shop (head L)) ofr A L C"
⟨*proof*⟩

lemma *insertion_resilience: "ALL L. L:valid A n B --> Suc i < len L*
--> ins (L,Suc i,M) ~:valid A n B"
⟨*proof*⟩

23.5 truncation resilience: only shop i can truncate at offer i

lemma *truncation_resilience: "ALL L. L:valid A n B --> Suc i < len L*
--> cons M (trunc (L,Suc i)):valid A n B --> shop M = shop (ith (L,i))"
⟨*proof*⟩

23.6 declarations for tactics

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

23.7 get components of a message

lemma *get_ML [dest]: "Says A' B {|A,R,I,M,L|}:set evs ==>*
M:parts (spies evs) & L:parts (spies evs)"
⟨*proof*⟩

23.8 general properties of p2

lemma *reqm_neq_prom [iff]:*
"reqm A r n I B ~= prom B' ofr A' r' I' (cons M L) J C"
⟨*proof*⟩

lemma *prom_neq_reqm [iff]:*
"prom B' ofr A' r' I' (cons M L) J C ~= reqm A r n I B"
⟨*proof*⟩

lemma *req_neq_pro [iff]: "req A r n I B ~= pro B' ofr A' r' I' (cons M L) J C"*
⟨*proof*⟩

lemma *pro_neq_req [iff]: "pro B' ofr A' r' I' (cons M L) J C ~= req A r n I B"*
⟨*proof*⟩

lemma *p2_has_no_Gets: "evs:p2 ==> ALL A X. Gets A X ~:set evs"*
⟨*proof*⟩

lemma *p2_is_Gets_correct [iff]: "Gets_correct p2"*
⟨*proof*⟩

lemma *p2_is_one_step [iff]: "one_step p2"*
⟨*proof*⟩

lemma *p2_has_only_Says' [rule_format]: "evs:p2 ==>*
ev:set evs --> (EX A B X. ev=Says A B X)"
⟨*proof*⟩

lemma *p2_has_only_Says [iff]: "has_only_Says p2"*
⟨*proof*⟩

lemma *p2_is_regular [iff]: "regular p2"*
⟨*proof*⟩

23.9 private keys are safe

lemma *priK_parts_Friend_imp_bad [rule_format,dest]:*
"[| evs:p2; Friend B ~= A |]
==> (Key (priK A):parts (knows (Friend B) evs)) --> (A:bad)"
⟨*proof*⟩

lemma *priK_analz_Friend_imp_bad [rule_format,dest]:*
"[| evs:p2; Friend B ~= A |]
==> (Key (priK A):analz (knows (Friend B) evs)) --> (A:bad)"
⟨*proof*⟩

lemma *priK_notin_knows_max_Friend:*
"[| evs:p2; A ~:bad; A ~= Friend C |]
==> Key (priK A) ~:analz (knows_max (Friend C) evs)"
⟨*proof*⟩

23.10 general guardedness properties

lemma *agl_guard [intro]: "I:agl ==> I:guard n Ks"*
⟨*proof*⟩

lemma *Says_to_knows_max'_guard: "[| Says A' C {|A'',r,I,L|}:set evs;*

```
Guard n Ks (knows_max' C evs) |] ==> L:guard n Ks"
```
⟨*proof*⟩

lemma *Says_from_knows_max'_guard: "[| Says C A' {|A'',r,I,L|}:set evs;*
Guard n Ks (knows_max' C evs) |] ==> L:guard n Ks"
⟨*proof*⟩

lemma *Says_Nonce_not_used_guard: "[| Says A' B {|A'',r,I,L|}:set evs;*
Nonce n ~:used evs |] ==> L:guard n Ks"
⟨*proof*⟩

23.11 guardedness of messages

lemma *chain_guard [iff]: "chain B ofr A L C:guard n {priK A}"*
⟨*proof*⟩

lemma *chain_guard_Nonce_neq [intro]: "n ~= ofr*
==> chain B ofr A' L C:guard n {priK A}"
⟨*proof*⟩

lemma *anchor_guard [iff]: "anchor A n' B:guard n {priK A}"*
⟨*proof*⟩

lemma *anchor_guard_Nonce_neq [intro]: "n ~= n'*
==> anchor A' n' B:guard n {priK A}"
⟨*proof*⟩

lemma *reqm_guard [intro]: "I:agl ==> reqm A r n' I B:guard n {priK A}"*
⟨*proof*⟩

lemma *reqm_guard_Nonce_neq [intro]: "[| n ~= n'; I:agl |]*
==> reqm A' r n' I B:guard n {priK A}"
⟨*proof*⟩

lemma *prom_guard [intro]: "[| I:agl; J:agl; L:guard n {priK A} |]*
==> prom B ofr A r I L J C:guard n {priK A}"
⟨*proof*⟩

lemma *prom_guard_Nonce_neq [intro]: "[| n ~= ofr; I:agl; J:agl;*
L:guard n {priK A} |] ==> prom B ofr A' r I L J C:guard n {priK A}"
⟨*proof*⟩

23.12 Nonce uniqueness

lemma *uniq_Nonce_in_chain [dest]: "Nonce k:parts {chain B ofr A L C} ==>*
k=ofr"
⟨*proof*⟩

lemma *uniq_Nonce_in_anchor [dest]: "Nonce k:parts {anchor A n B} ==> k=n"*
⟨*proof*⟩

lemma *uniq_Nonce_in_reqm [dest]: "[| Nonce k:parts {reqm A r n I B};*
I:agl |] ==> k=n"
⟨*proof*⟩

lemma *uniq_Nonce_in_prom [dest]: "[| Nonce k:parts {prom B ofr A r I L J C};*
I:agl; J:agl; Nonce k ~:parts {L} |] ==> k=ofr"
⟨*proof*⟩

23.13 requests are guarded

lemma *req_imp_Guard [rule_format]: "[| evs:p2; A ~:bad |] ==>*
req A r n I B:set evs --> Guard n {priK A} (spies evs)"
⟨*proof*⟩

lemma *req_imp_Guard_Friend: "[| evs:p2; A ~:bad; req A r n I B:set evs |]*
==> Guard n {priK A} (knows_max (Friend C) evs)"
⟨*proof*⟩

23.14 propositions are guarded

lemma *pro_imp_Guard [rule_format]: "[| evs:p2; B ~:bad; A ~:bad |] ==>*
pro B ofr A r I (cons M L) J C:set evs --> Guard ofr {priK A} (spies evs)"
⟨*proof*⟩

lemma *pro_imp_Guard_Friend: "[| evs:p2; B ~:bad; A ~:bad;*
pro B ofr A r I (cons M L) J C:set evs |]
==> Guard ofr {priK A} (knows_max (Friend D) evs)"
⟨*proof*⟩

23.15 data confidentiality: no one other than the originator can decrypt the offers

lemma *Nonce_req_notin_spies: "[| evs:p2; req A r n I B:set evs; A ~:bad |]*
==> Nonce n ~:analz (spies evs)"
⟨*proof*⟩

lemma *Nonce_req_notin_knows_max_Friend: "[| evs:p2; req A r n I B:set evs;*
A ~:bad; A ~= Friend C |] ==> Nonce n ~:analz (knows_max (Friend C) evs)"
⟨*proof*⟩

lemma *Nonce_pro_notin_spies: "[| evs:p2; B ~:bad; A ~:bad;*
pro B ofr A r I (cons M L) J C:set evs |] ==> Nonce ofr ~:analz (spies evs)"
⟨*proof*⟩

lemma *Nonce_pro_notin_knows_max_Friend: "[| evs:p2; B ~:bad; A ~:bad;*
A ~= Friend D; pro B ofr A r I (cons M L) J C:set evs |]
==> Nonce ofr ~:analz (knows_max (Friend D) evs)"
⟨*proof*⟩

23.16 forward privacy: only the originator can know the identity of the shops

lemma *forward_privacy_Spy: "[| evs:p2; B ~:bad; A ~:bad;*
pro B ofr A r I (cons M L) J C:set evs |]
==> sign B (Nonce ofr) ~:analz (spies evs)"
⟨*proof*⟩

lemma *forward_privacy_Friend: "[| evs:p2; B ~:bad; A ~:bad; A ~= Friend D; pro B ofr A r I (cons M L) J C:set evs |] ==> sign B (Nonce ofr) ~:analz (knows_max (Friend D) evs)"*
⟨*proof*⟩

23.17 non repudiability: an offer signed by B has been sent by B

lemma *Crypt_reqm: "[| Crypt (priK A) X:parts {reqm A' r n I B}; I:agl |] ==> A=A'"*
⟨*proof*⟩

lemma *Crypt_prom: "[| Crypt (priK A) X:parts {prom B ofr A' r I L J C}; I:agl; J:agl |] ==> A=B | Crypt (priK A) X:parts {L}"*
⟨*proof*⟩

lemma *Crypt_safeness: "[| evs:p2; A ~:bad |] ==> Crypt (priK A) X:parts (spies evs) --> (EX B Y. Says A B Y:set evs & Crypt (priK A) X:parts {Y})"*
⟨*proof*⟩

lemma *Crypt_Hash_imp_sign: "[| evs:p2; A ~:bad |] ==> Crypt (priK A) (Hash X):parts (spies evs) --> (EX B Y. Says A B Y:set evs & sign A X:parts {Y})"*
⟨*proof*⟩

lemma *sign_safeness: "[| evs:p2; A ~:bad |] ==> sign A X:parts (spies evs) --> (EX B Y. Says A B Y:set evs & sign A X:parts {Y})"*
⟨*proof*⟩

end

24 Needham-Schroeder-Lowe Public-Key Protocol

theory *Guard_NS_Public* **imports** *Guard_Public* **begin**

24.1 messages used in the protocol

syntax *ns1 :: "agent => agent => nat => event"*

translations *"ns1 A B NA" => "Says A B (Crypt (pubK B) {|Nonce NA, Agent A|})"*

syntax *ns1' :: "agent => agent => agent => nat => event"*

translations *"ns1' A' A B NA" => "Says A' B (Crypt (pubK B) {|Nonce NA, Agent A|})"*

syntax *ns2 :: "agent => agent => nat => nat => event"*

translations *"ns2 B A NA NB" => "Says B A (Crypt (pubK A) {|Nonce NA, Nonce NB, Agent B|})"*

syntax *ns2' :: "agent => agent => agent => nat => nat => event"*

translations *"ns2' B' B A NA NB"*
=> "Says B' A (Crypt (pubK A) {|Nonce NA, Nonce NB, Agent B|})"

syntax *ns3 :: "agent => agent => nat => event"*

translations *"ns3 A B NB" => "Says A B (Crypt (pubK B) (Nonce NB))"*

24.2 definition of the protocol

consts *nsp :: "event list set"*

inductive *nsp*
intros

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: "[| evs3:nsp; ns1 A B NA:set evs3; ns2' B' B A NA NB:set evs3 |] ==> ns3 A B NB # evs3:nsp"

24.3 declarations for tactics

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

24.4 general properties of nsp

lemma *nsp_has_no_Gets: "evs:nsp ==> ALL A X. Gets A X ~:set evs"*
⟨*proof*⟩

lemma *nsp_is_Gets_correct [iff]: "Gets_correct nsp"*
⟨*proof*⟩

lemma *nsp_is_one_step [iff]: "one_step nsp"*
⟨*proof*⟩

lemma *nsp_has_only_Says' [rule_format]: "evs:nsp ==> ev:set evs --> (EX A B X. ev=Says A B X)"*
⟨*proof*⟩

lemma *nsp_has_only_Says [iff]: "has_only_Says nsp"*
⟨*proof*⟩

lemma *nsp_is_regular [iff]: "regular nsp"*
⟨*proof*⟩

24.5 nonce are used only once

lemma *NA_is_uniq [rule_format]: "evs:nsp ==>*
Crypt (pubK B) {|Nonce NA, Agent A|}:parts (spies evs)
--> Crypt (pubK B') {|Nonce NA, Agent A'|}:parts (spies evs)
--> Nonce NA ~:analz (spies evs) --> A=A' & B=B'"
⟨*proof*⟩

lemma *no_Nonce_NS1_NS2 [rule_format]: "evs:nsp ==>*
Crypt (pubK B') {|Nonce NA', Nonce NA, Agent A'|}:parts (spies evs)
--> Crypt (pubK B) {|Nonce NA, Agent A|}:parts (spies evs)
--> Nonce NA:analz (spies evs)"
⟨*proof*⟩

lemma *no_Nonce_NS1_NS2' [rule_format]:*
"[| Crypt (pubK B') {|Nonce NA', Nonce NA, Agent A'|}:parts (spies evs);
Crypt (pubK B) {|Nonce NA, Agent A|}:parts (spies evs); evs:nsp |]
==> Nonce NA:analz (spies evs)"
⟨*proof*⟩

lemma *NB_is_uniq [rule_format]: "evs:nsp ==>*
Crypt (pubK A) {|Nonce NA, Nonce NB, Agent B|}:parts (spies evs)
--> Crypt (pubK A') {|Nonce NA', Nonce NB, Agent B'|}:parts (spies evs)
--> Nonce NB ~:analz (spies evs) --> A=A' & B=B' & NA=NA'"
⟨*proof*⟩

24.6 guardedness of NA

lemma *ns1_imp_Guard [rule_format]: "[| evs:nsp; A ~:bad; B ~:bad |] ==>*
ns1 A B NA:set evs --> Guard NA {priK A,priK B} (spies evs)"
⟨*proof*⟩

24.7 guardedness of NB

lemma *ns2_imp_Guard [rule_format]: "[| evs:nsp; A ~:bad; B ~:bad |] ==>*
ns2 B A NA NB:set evs --> Guard NB {priK A,priK B} (spies evs)"
⟨*proof*⟩

24.8 Agents' Authentication

lemma *B_trusts_NS1: "[| evs:nsp; A ~:bad; B ~:bad |] ==>*
Crypt (pubK B) {|Nonce NA, Agent A|}:parts (spies evs)
--> Nonce NA ~:analz (spies evs) --> ns1 A B NA:set evs"
⟨*proof*⟩

lemma *A_trusts_NS2: "[| evs:nsp; A ~:bad; B ~:bad |] ==> ns1 A B NA:set evs*
--> Crypt (pubK A) {|Nonce NA, Nonce NB, Agent B|}:parts (spies evs)
--> ns2 B A NA NB:set evs"
⟨*proof*⟩

lemma *B_trusts_NS3: "[| evs:nsp; A ~:bad; B ~:bad |] ==> ns2 B A NA NB:set evs*
--> Crypt (pubK B) (Nonce NB):parts (spies evs) --> ns3 A B NB:set evs"
⟨*proof*⟩

end

25 protocol-independent confidentiality theorem on keys

theory *GuardK* **imports** *Analz Extensions* **begin**

consts *guardK :: "nat => key set => msg set"*

inductive *"guardK n Ks"*
intros
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"

25.1 basic facts about *guardK*

lemma *Nonce_is_guardK [iff]: "Nonce p:guardK n Ks"*
⟨*proof*⟩

lemma *Agent_is_guardK [iff]: "Agent A:guardK n Ks"*
⟨*proof*⟩

lemma *Number_is_guardK [iff]: "Number r:guardK n Ks"*
⟨*proof*⟩

lemma *Key_notin_guardK: "X:guardK n Ks ==> X ~= Key n"*
⟨*proof*⟩

lemma *Key_notin_guardK_iff [iff]: "Key n ~:guardK n Ks"*
⟨*proof*⟩

lemma *guardK_has_Crypt [rule_format]: "X:guardK n Ks ==> Key n:parts {X}*
--> (EX K Y. Crypt K Y:kparts {X} & Key n:parts {Y})"
⟨*proof*⟩

lemma *Key_notin_kparts_msg: "X:guardK n Ks ==> Key n ~:kparts {X}"*
⟨*proof*⟩

lemma *Key_in_kparts_imp_no_guardK: "Key n:kparts H*
==> EX X. X:H & X ~:guardK n Ks"
⟨*proof*⟩

lemma *guardK_kparts [rule_format]: "X:guardK n Ks ==>*

```
Y:kparts {X} --> Y:guardK n Ks"
```
⟨*proof*⟩

lemma *guardK_Crypt: "[| Crypt K Y:guardK n Ks; K ~:invKey‘Ks |] ==> Y:guardK n Ks"*
⟨*proof*⟩

lemma *guardK_MPair [iff]: "({|X,Y|}:guardK n Ks) = (X:guardK n Ks & Y:guardK n Ks)"*
⟨*proof*⟩

lemma *guardK_not_guardK [rule_format]: "X:guardK n Ks ==> Crypt K Y:kparts {X} --> Key n:kparts {Y} --> Y ~:guardK n Ks"*
⟨*proof*⟩

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

25.2 guarded sets

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

25.3 basic facts about *GuardK*

lemma *GuardK_empty [iff]: "GuardK n Ks {}"*
⟨*proof*⟩

lemma *Key_notin_kparts [simplified]: "GuardK n Ks H ==> Key n ~:kparts H"*
⟨*proof*⟩

lemma *GuardK_must_decrypt: "[| GuardK n Ks H; Key n:analz H |] ==> EX K Y. Crypt K Y:kparts H & Key (invKey K):kparts H"*
⟨*proof*⟩

lemma *GuardK_kparts [intro]: "GuardK n Ks H ==> GuardK n Ks (kparts H)"*
⟨*proof*⟩

lemma *GuardK_mono: "[| GuardK n Ks H; G <= H |] ==> GuardK n Ks G"*
⟨*proof*⟩

lemma *GuardK_insert [iff]: "GuardK n Ks (insert X H) = (GuardK n Ks H & X:guardK n Ks)"*
⟨*proof*⟩

lemma *GuardK_Un [iff]: "GuardK n Ks (G Un H) = (GuardK n Ks G & GuardK n Ks H)"*
⟨*proof*⟩

lemma *GuardK_synth [intro]: "GuardK n Ks G ==> GuardK n Ks (synth G)"*
⟨*proof*⟩

lemma *GuardK_analz [intro]: "[| GuardK n Ks G; ALL K. K:Ks --> Key K ~:analz*

```
G |]
==> GuardK n Ks (analz G)"
```
⟨*proof*⟩

lemma *in_GuardK [dest]: "[| X:G; GuardK n Ks G |] ==> X:guardK n Ks"*
⟨*proof*⟩

lemma *in_synth_GuardK: "[| X:synth G; GuardK n Ks G |] ==> X:guardK n Ks"*
⟨*proof*⟩

lemma *in_analz_GuardK: "[| X:analz G; GuardK n Ks G;*
ALL K. K:Ks --> Key K ~:analz G |] ==> X:guardK n Ks"
⟨*proof*⟩

lemma *GuardK_keyset [simp]: "[| keyset G; Key n ~:G |] ==> GuardK n Ks G"*
⟨*proof*⟩

lemma *GuardK_Un_keyset: "[| GuardK n Ks G; keyset H; Key n ~:H |]*
==> GuardK n Ks (G Un H)"
⟨*proof*⟩

lemma *in_GuardK_kparts: "[| X:G; GuardK n Ks G; Y:kparts {X} |] ==> Y:guardK*
n Ks"
⟨*proof*⟩

lemma *in_GuardK_kparts_neq: "[| X:G; GuardK n Ks G; Key n':kparts {X} |]*
==> n ~= n'"
⟨*proof*⟩

lemma *in_GuardK_kparts_Crypt: "[| X:G; GuardK n Ks G; is_MPair X;*
Crypt K Y:kparts {X}; Key n:kparts {Y} |] ==> invKey K:Ks"
⟨*proof*⟩

lemma *GuardK_extand: "[| GuardK n Ks G; Ks <= Ks';*
[| K:Ks'; K ~:Ks |] ==> Key K ~:parts G |] ==> GuardK n Ks' G"
⟨*proof*⟩

25.4 set obtained by decrypting a message

syntax *decrypt :: "msg set => key => msg => msg set"*

translations *"decrypt H K Y" => "insert Y (H - {Crypt K Y})"*

lemma *analz_decrypt: "[| Crypt K Y:H; Key (invKey K):H; Key n:analz H |]*
==> Key n:analz (decrypt H K Y)"
⟨*proof*⟩

lemma *parts_decrypt: "[| Crypt K Y:H; X:parts (decrypt H K Y) |] ==> X:parts*
H"
⟨*proof*⟩

25.5 number of Crypt's in a message

consts *crypt_nb :: "msg => nat"*

recdef *crypt_nb "measure size"*
"crypt_nb (Crypt K X) = Suc (crypt_nb X)"
"crypt_nb {|X,Y|} = crypt_nb X + crypt_nb Y"
"crypt_nb X = 0"

25.6 basic facts about *crypt_nb*

lemma *non_empty_crypt_msg: "Crypt K Y:parts {X} ==> 0 < crypt_nb X"*
⟨*proof*⟩

25.7 number of Crypt's in a message list

consts *cnb :: "msg list => nat"*

recdef *cnb "measure size"*
"cnb [] = 0"
"cnb (X#l) = crypt_nb X + cnb l"

25.8 basic facts about *cnb*

lemma *cnb_app [simp]: "cnb (l @ l') = cnb l + cnb l'"*
⟨*proof*⟩

lemma *mem_cnb_minus: "x mem l ==> cnb l = crypt_nb x + (cnb l - crypt_nb x)"*
⟨*proof*⟩

lemmas *mem_cnb_minus_substI = mem_cnb_minus [THEN ssubst]*

lemma *cnb_minus [simp]: "x mem l ==> cnb (minus l x) = cnb l - crypt_nb x"*
⟨*proof*⟩

lemma *parts_cnb: "Z:parts (set l) ==>*
cnb l = (cnb l - crypt_nb Z) + crypt_nb Z"
⟨*proof*⟩

lemma *non_empty_crypt: "Crypt K Y:parts (set l) ==> 0 < cnb l"*
⟨*proof*⟩

25.9 list of kparts

lemma *kparts_msg_set: "EX l. kparts {X} = set l & cnb l = crypt_nb X"*
⟨*proof*⟩

lemma *kparts_set: "EX l'. kparts (set l) = set l' & cnb l' = cnb l"*
⟨*proof*⟩

25.10 list corresponding to "decrypt"

constdefs *decrypt' :: "msg list => key => msg => msg list"*
"decrypt' l K Y == Y # minus l (Crypt K Y)"

declare *decrypt'_def [simp]*

25.11 basic facts about *decrypt'*

lemma *decrypt_minus: "decrypt (set l) K Y <= set (decrypt' l K Y)"*
⟨*proof*⟩

if the analysis of a finite guarded set gives n then it must also give one of the keys of Ks

lemma *GuardK_invKey_by_list [rule_format]: "ALL l. cnb l = p*
--> GuardK n Ks (set l) --> Key n:analz (set l)
--> (EX K. K:Ks & Key K:analz (set l))"
⟨*proof*⟩

lemma *GuardK_invKey_finite: "[| Key n:analz G; GuardK n Ks G; finite G |]*
==> EX K. K:Ks & Key K:analz G"
⟨*proof*⟩

lemma *GuardK_invKey: "[| Key n:analz G; GuardK n Ks G |]*
==> EX K. K:Ks & Key K:analz G"
⟨*proof*⟩

if the analyse of a finite guarded set and a (possibly infinite) set of keys gives n then it must also gives Ks

lemma *GuardK_invKey_keyset: "[| Key n:analz (G Un H); GuardK n Ks G; finite G;*
keyset H; Key n ~:H |] ==> EX K. K:Ks & Key K:analz (G Un H)"
⟨*proof*⟩

end

theory *Shared* **imports** *Event* **begin**

consts
shrK :: "agent => key"

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

All keys are symmetric

defs *all_symmetric_def: "all_symmetric == True"*

lemma *isSym_keys: "K ∈ symKeys"*
⟨*proof*⟩

Server knows all long-term keys; other agents know only their own

primrec
initState_Server: "initState Server = Key ‘ range shrK"
initState_Friend: "initState (Friend i) = {Key (shrK (Friend i))}"
initState_Spy: "initState Spy = Key‘shrK‘bad"

25.12 Basic properties of shrK

declare *inj_shrK [THEN inj_eq, iff]*

lemma *invKey_K [simp]: "invKey K = K"*
⟨*proof*⟩

lemma *analz_Decrypt' [dest]:*
"[| Crypt K X ∈ analz H; Key K ∈ analz H |] ==> X ∈ analz H"
⟨*proof*⟩

Now cancel the *dest* attribute given to *analz.Decrypt* in its declaration.

declare *analz.Decrypt [rule del]*

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

lemma *keysFor_parts_initState [simp]: "keysFor (parts (initState C)) = {}"*
⟨*proof*⟩

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

lemma *Crypt_imp_keysFor: "Crypt K X ∈ H ==> K ∈ keysFor H"*
⟨*proof*⟩

25.13 Function "knows"

lemma *Spy_knows_Spy_bad [intro!]: "A: bad ==> Key (shrK A) ∈ knows Spy evs"*
⟨*proof*⟩

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

lemma *shrK_in_initState [iff]: "Key (shrK A) ∈ initState A"*
⟨*proof*⟩

lemma *shrK_in_used [iff]: "Key (shrK A) ∈ used evs"*
⟨*proof*⟩

lemma *Key_not_used [simp]: "Key K ∉ used evs ==> K ∉ range shrK"*
⟨*proof*⟩

lemma *shrK_neq [simp]: "Key K ∉ used evs ==> shrK B ≠ K"*
⟨*proof*⟩

declare *shrK_neq [THEN not_sym, simp]*

25.14 Fresh nonces

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

lemma *Nonce_notin_used_empty [simp]: "Nonce N ∉ used []"*
⟨*proof*⟩

25.15 Supply fresh nonces for possibility theorems.

lemma *Nonce_supply_lemma: "∃N. ALL n. N<=n --> Nonce n ∉ used evs"*
⟨*proof*⟩

lemma *Nonce_supply1: "∃N. Nonce N ∉ used evs"*
⟨*proof*⟩

lemma *Nonce_supply2: "∃N N'. Nonce N ∉ used evs & Nonce N' ∉ used evs' & N ≠ N'"*
⟨*proof*⟩

lemma *Nonce_supply3: "∃N N' N''. Nonce N ∉ used evs & Nonce N' ∉ used evs' &*
Nonce N'' ∉ used evs'' & N ≠ N' & N' ≠ N'' & N ≠ N''"
⟨*proof*⟩

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

Unlike the corresponding property of nonces, we cannot prove *finite KK* $\Longrightarrow$ *∃K. K ∉ KK ∧ Key K ∉ 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.

25.16 Tactics for possibility theorems

⟨*ML*⟩

25.17 Specialized Rewriting for Theorems About `analz` and Image

lemma *subset_Compl_range: "A <= - (range shrK) ==> shrK x ∉ A"*
⟨*proof*⟩

lemma *insert_Key_singleton: "insert (Key K) H = Key ‘ {K} ∪ H"*
⟨*proof*⟩

lemma *insert_Key_image: "insert (Key K) (Key‘KK ∪ C) = Key‘(insert K KK) ∪ C"*

⟨*proof*⟩

lemmas *analz_image_freshK_simps =*
simp_thms mem_simps — these two allow its use with *only:*
disj_comms
image_insert [THEN sym] image_Un [THEN sym] empty_subsetI insert_subset
analz_insert_eq Un_upper2 [THEN analz_mono, THEN [2] rev_subsetD]
insert_Key_singleton subset_Compl_range
Key_not_used insert_Key_image Un_assoc [THEN sym]

lemma *analz_image_freshK_lemma:*
"(Key K ∈ analz (Key‘nE ∪ H)) --> (K ∈ nE | Key K ∈ analz H) ==>
(Key K ∈ analz (Key‘nE ∪ H)) = (K ∈ nE | Key K ∈ analz H)"
⟨*proof*⟩

⟨*ML*⟩

lemma *invKey_shrK_iff [iff]:*
"(Key (invKey K) ∈ X) = (Key K ∈ X)"
⟨*proof*⟩

⟨*ML*⟩

lemma *knows_subset_knows_Cons: "knows A evs <= knows A (e # evs)"*
⟨*proof*⟩

end

26 lemmas on guarded messages for protocols with symmetric keys

theory *Guard_Shared* **imports** *Guard GuardK Shared* **begin**

26.1 Extensions to Theory *Shared*

declare *initState.simps [simp del]*

26.1.1 a little abbreviation

syntax *Ciph :: "agent => msg"*

translations *"Ciph A X" == "Crypt (shrK A) X"*

26.1.2 agent associated to a key

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

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

26.1.3 basic facts about *initState*

lemma *no_Crypt_in_parts_init [simp]: "Crypt K X ~:parts (initState A)"*
⟨*proof*⟩

lemma *no_Crypt_in_analz_init [simp]: "Crypt K X ~:analz (initState A)"*
⟨*proof*⟩

lemma *no_shrK_in_analz_init [simp]: "A ~:bad*
==> Key (shrK A) ~:analz (initState Spy)"
⟨*proof*⟩

lemma *shrK_notin_initState_Friend [simp]: "A ~= Friend C*
==> Key (shrK A) ~: parts (initState (Friend C))"
⟨*proof*⟩

lemma *keyset_init [iff]: "keyset (initState A)"*
⟨*proof*⟩

26.1.4 sets of symmetric keys

constdefs *shrK_set :: "key set => bool"*
"shrK_set Ks == ALL K. K:Ks --> (EX A. K = shrK A)"

lemma *in_shrK_set: "[| shrK_set Ks; K:Ks |] ==> EX A. K = shrK A"*
⟨*proof*⟩

lemma *shrK_set1 [iff]: "shrK_set {shrK A}"*
⟨*proof*⟩

lemma *shrK_set2 [iff]: "shrK_set {shrK A, shrK B}"*
⟨*proof*⟩

26.1.5 sets of good keys

constdefs *good :: "key set => bool"*
"good Ks == ALL K. K:Ks --> agt K ~:bad"

lemma *in_good: "[| good Ks; K:Ks |] ==> agt K ~:bad"*
⟨*proof*⟩

lemma *good1 [simp]: "A ~:bad ==> good {shrK A}"*
⟨*proof*⟩

lemma *good2 [simp]: "[| A ~:bad; B ~:bad |] ==> good {shrK A, shrK B}"*
⟨*proof*⟩

26.2 Proofs About Guarded Messages

26.2.1 small hack

lemma *shrK_is_invKey_shrK: "shrK A = invKey (shrK A)"*
⟨*proof*⟩

lemmas *shrK_is_invKey_shrK_substI = shrK_is_invKey_shrK [THEN ssubst]*

lemmas *invKey_invKey_substI = invKey [THEN ssubst]*

lemma *"Nonce n:parts {X} ==> Crypt (shrK A) X:guard n {shrK A}"*
⟨*proof*⟩

26.2.2 guardedness results on nonces

lemma *guard_ciph [simp]: "shrK A:Ks ==> Ciph A X:guard n Ks"*
⟨*proof*⟩

lemma *guardK_ciph [simp]: "shrK A:Ks ==> Ciph A X:guardK n Ks"*
⟨*proof*⟩

lemma *Guard_init [iff]: "Guard n Ks (initState B)"*
⟨*proof*⟩

lemma *Guard_knows_max': "Guard n Ks (knows_max' C evs)*
==> Guard n Ks (knows_max C evs)"
⟨*proof*⟩

lemma *Nonce_not_used_Guard_spies [dest]: "Nonce n ~:used evs*
==> Guard n Ks (spies evs)"
⟨*proof*⟩

lemma *Nonce_not_used_Guard [dest]: "[| evs:p; Nonce n ~:used evs;*
Gets_correct p; one_step p |] ==> Guard n Ks (knows (Friend C) evs)"
⟨*proof*⟩

lemma *Nonce_not_used_Guard_max [dest]: "[| evs:p; Nonce n ~:used evs;*
Gets_correct p; one_step p |] ==> Guard n Ks (knows_max (Friend C) evs)"
⟨*proof*⟩

lemma *Nonce_not_used_Guard_max' [dest]: "[| evs:p; Nonce n ~:used evs;*
Gets_correct p; one_step p |] ==> Guard n Ks (knows_max' (Friend C) evs)"
⟨*proof*⟩

26.2.3 guardedness results on keys

lemma *GuardK_init [simp]: "n ~:range shrK ==> GuardK n Ks (initState B)"*
⟨*proof*⟩

lemma *GuardK_knows_max': "[| GuardK n A (knows_max' C evs); n ~:range shrK*
|]
==> GuardK n A (knows_max C evs)"
⟨*proof*⟩

lemma *Key_not_used_GuardK_spies [dest]: "Key n ~:used evs*
==> GuardK n A (spies evs)"
⟨*proof*⟩

lemma *Key_not_used_GuardK [dest]: "[| evs:p; Key n ~:used evs;*
Gets_correct p; one_step p |] ==> GuardK n A (knows (Friend C) evs)"
⟨*proof*⟩

lemma *Key_not_used_GuardK_max [dest]: "[| evs:p; Key n ~:used evs;*
Gets_correct p; one_step p |] ==> GuardK n A (knows_max (Friend C) evs)"
⟨*proof*⟩

lemma *Key_not_used_GuardK_max' [dest]: "[| evs:p; Key n ~:used evs;*
Gets_correct p; one_step p |] ==> GuardK n A (knows_max' (Friend C) evs)"
⟨*proof*⟩

26.2.4 regular protocols

constdefs *regular :: "event list set => bool"*
"regular p == ALL evs A. evs:p --> (Key (shrK A):parts (spies evs)) = (A:bad)"

lemma *shrK_parts_iff_bad [simp]: "[| evs:p; regular p |] ==>*
(Key (shrK A):parts (spies evs)) = (A:bad)"
⟨*proof*⟩

lemma *shrK_analz_iff_bad [simp]: "[| evs:p; regular p |] ==>*
(Key (shrK A):analz (spies evs)) = (A:bad)"
⟨*proof*⟩

lemma *Guard_Nonce_analz: "[| Guard n Ks (spies evs); evs:p;*
shrK_set Ks; good Ks; regular p |] ==> Nonce n ~:analz (spies evs)"
⟨*proof*⟩

lemma *GuardK_Key_analz: "[| GuardK n Ks (spies evs); evs:p;*
shrK_set Ks; good Ks; regular p; n ~:range shrK |] ==> Key n ~:analz (spies
evs)"
⟨*proof*⟩

end

27 Otway-Rees Protocol

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

27.1 messages used in the protocol

syntax *nil :: "msg"*

translations *"nil" == "Number 0"*

syntax *or1 :: "agent => agent => nat => event"*

translations *"or1 A B NA"*
=> "Says A B {|Nonce NA, Agent A, Agent B,

```
              Ciph A {|Nonce NA, Agent A, Agent B|}|}"

syntax or1' :: "agent => agent => agent => nat => msg => event"

translations "or1' A' A B NA X"
=> "Says A' B {|Nonce NA, Agent A, Agent B, X|}"

syntax or2 :: "agent => agent => nat => nat => msg => event"

translations "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|}|}"

syntax or2' :: "agent => agent => agent => nat => nat => event"

translations "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|}|}"

syntax or3 :: "agent => agent => nat => nat => key => event"

translations "or3 A B NA NB K"
=> "Says Server B {|Nonce NA, Ciph A {|Nonce NA, Key K|},
                    Ciph B {|Nonce NB, Key K|}|}"

syntax or3':: "agent => msg => agent => agent => nat => nat => key => event"

translations "or3' S Y A B NA NB K"
=> "Says S B {|Nonce NA, Y, Ciph B {|Nonce NB, Key K|}|}"

syntax or4 :: "agent => agent => nat => msg => event"

translations "or4 A B NA X" => "Says B A {|Nonce NA, X, nil|}"

syntax or4' :: "agent => agent => nat => msg => event"

translations "or4' B' A NA K" =>
"Says B' A {|Nonce NA, Ciph A {|Nonce NA, Key K|}, nil|}"
```

27.2 definition of the protocol

```
consts or :: "event list set"

inductive or
intros

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"
```

27.3 declarations for tactics

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

27.4 general properties of or

lemma *or_has_no_Gets: "evs:or ==> ALL A X. Gets A X ~:set evs"*
⟨*proof*⟩

lemma *or_is_Gets_correct [iff]: "Gets_correct or"*
⟨*proof*⟩

lemma *or_is_one_step [iff]: "one_step or"*
⟨*proof*⟩

lemma *or_has_only_Says' [rule_format]: "evs:or ==>*
ev:set evs --> (EX A B X. ev=Says A B X)"
⟨*proof*⟩

lemma *or_has_only_Says [iff]: "has_only_Says or"*
⟨*proof*⟩

27.5 or is regular

lemma *or1'_parts_spies [dest]: "or1' A' A B NA X:set evs*
==> X:parts (spies evs)"
⟨*proof*⟩

lemma *or2_parts_spies [dest]: "or2 A B NA NB X:set evs*
==> X:parts (spies evs)"
⟨*proof*⟩

lemma *or3_parts_spies [dest]: "Says S B {|NA, Y, Ciph B {|NB, K|}|}:set evs*
==> K:parts (spies evs)"
⟨*proof*⟩

lemma *or_is_regular [iff]: "regular or"*
⟨*proof*⟩

27.6 guardedness of KAB

lemma *Guard_KAB [rule_format]: "[| evs:or; A ~:bad; B ~:bad |] ==>*
or3 A B NA NB K:set evs --> GuardK K {shrK A,shrK B} (spies evs)"
⟨*proof*⟩

27.7 guardedness of NB

lemma *Guard_NB [rule_format]: "[| evs:or; B ~:bad |] ==>*
or2 A B NA NB X:set evs --> Guard NB {shrK B} (spies evs)"
⟨*proof*⟩

end

28 Yahalom Protocol

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

28.1 messages used in the protocol

syntax *ya1 :: "agent => agent => nat => event"*

translations *"ya1 A B NA" => "Says A B {|Agent A, Nonce NA|}"*

syntax *ya1' :: "agent => agent => agent => nat => event"*

translations *"ya1' A' A B NA" => "Says A' B {|Agent A, Nonce NA|}"*

syntax *ya2 :: "agent => agent => nat => nat => event"*

translations *"ya2 A B NA NB"*
=> "Says B Server {|Agent B, Ciph B {|Agent A, Nonce NA, Nonce NB|}|}"

syntax *ya2' :: "agent => agent => agent => nat => nat => event"*

translations *"ya2' B' A B NA NB"*
=> "Says B' Server {|Agent B, Ciph B {|Agent A, Nonce NA, Nonce NB|}|}"

syntax *ya3 :: "agent => agent => nat => nat => key => event"*

translations *"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|}|}"

syntax *ya3':: "agent => msg => agent => agent => nat => nat => key => event"*

translations *"ya3' S Y A B NA NB K"*
=> "Says S A {|Ciph A {|Agent B, Key K, Nonce NA, Nonce NB|}, Y|}"

syntax *ya4 :: "agent => agent => nat => nat => msg => event"*

translations *"ya4 A B K NB Y" => "Says A B {|Y, Crypt K (Nonce NB)|}"*

syntax *ya4' :: "agent => agent => nat => nat => msg => event"*

translations *"ya4' A' B K NB Y" => "Says A' B {|Y, Crypt K (Nonce NB)|}"*

28.2 definition of the protocol

consts *ya :: "event list set"*

inductive *ya*
intros

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"

28.3 declarations for tactics

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

28.4 general properties of ya

lemma *ya_has_no_Gets: "evs:ya ==> ALL A X. Gets A X ~:set evs"*
⟨*proof*⟩

lemma *ya_is_Gets_correct [iff]: "Gets_correct ya"*
⟨*proof*⟩

lemma *ya_is_one_step [iff]: "one_step ya"*
⟨*proof*⟩

lemma *ya_has_only_Says' [rule_format]: "evs:ya ==>*
ev:set evs --> (EX A B X. ev=Says A B X)"
⟨*proof*⟩

lemma *ya_has_only_Says [iff]: "has_only_Says ya"*
⟨*proof*⟩

lemma *ya_is_regular [iff]: "regular ya"*
⟨*proof*⟩

28.5 guardedness of KAB

lemma *Guard_KAB [rule_format]: "[| evs:ya; A ~:bad; B ~:bad |] ==>*
ya3 A B NA NB K:set evs --> GuardK K {shrK A,shrK B} (spies evs)"
⟨*proof*⟩

28.6 session keys are not symmetric keys

lemma *KAB_isnt_shrK [rule_format]: "evs:ya ==>*
ya3 A B NA NB K:set evs --> K ~:range shrK"
⟨*proof*⟩

lemma *ya3_shrK: "evs:ya ==> ya3 A B NA NB (shrK C) ~:set evs"*
⟨*proof*⟩

28.7 ya2' implies ya1'

lemma *ya2'_parts_imp_ya1'_parts [rule_format]:*
"[| evs:ya; B ~:bad |] ==>
Ciph B {|Agent A, Nonce NA, Nonce NB|}:parts (spies evs) -->
{|Agent A, Nonce NA|}:spies evs"
⟨*proof*⟩

lemma *ya2'_imp_ya1'_parts: "[| ya2' B' A B NA NB:set evs; evs:ya; B ~:bad*
|]
==> {|Agent A, Nonce NA|}:spies evs"
⟨*proof*⟩

28.8 uniqueness of NB

lemma *NB_is_uniq_in_ya2'_parts [rule_format]: "[| evs:ya; B ~:bad; B' ~:bad*
|] ==>
Ciph B {|Agent A, Nonce NA, Nonce NB|}:parts (spies evs) -->
Ciph B' {|Agent A', Nonce NA', Nonce NB|}:parts (spies evs) -->
A=A' & B=B' & NA=NA'"
⟨*proof*⟩

lemma *NB_is_uniq_in_ya2': "[| ya2' C A B NA NB:set evs;*
ya2' C' A' B' NA' NB:set evs; evs:ya; B ~:bad; B' ~:bad |]
==> A=A' & B=B' & NA=NA'"
⟨*proof*⟩

28.9 ya3' implies ya2'

lemma *ya3'_parts_imp_ya2'_parts [rule_format]: "[| evs:ya; A ~:bad |] ==>*
Ciph A {|Agent B, Key K, Nonce NA, Nonce NB|}:parts (spies evs)
--> Ciph B {|Agent A, Nonce NA, Nonce NB|}:parts (spies evs)"
⟨*proof*⟩

lemma *ya3'_parts_imp_ya2' [rule_format]: "[| evs:ya; A ~:bad |] ==>*
Ciph A {|Agent B, Key K, Nonce NA, Nonce NB|}:parts (spies evs)
--> (EX B'. ya2' B' A B NA NB:set evs)"
⟨*proof*⟩

lemma *ya3'_imp_ya2': "[| ya3' S Y A B NA NB K:set evs; evs:ya; A ~:bad |]*
==> (EX B'. ya2' B' A B NA NB:set evs)"
⟨*proof*⟩

28.10 ya3' implies ya3

lemma *ya3'_parts_imp_ya3 [rule_format]: "[| evs:ya; A ~:bad |] ==>*

```
Ciph A {|Agent B, Key K, Nonce NA, Nonce NB|}:parts(spies evs)
--> ya3 A B NA NB K:set evs"
```
⟨*proof*⟩

lemma *ya3'_imp_ya3: "[| ya3' S Y A B NA NB K:set evs; evs:ya; A ~:bad |]*
==> ya3 A B NA NB K:set evs"
⟨*proof*⟩

28.11 guardedness of NB

constdefs *ya_keys :: "agent => agent => nat => nat => event list => key set"*
"ya_keys A B NA NB evs == {shrK A,shrK B} Un {K. ya3 A B NA NB K:set evs}"

lemma *Guard_NB [rule_format]: "[| evs:ya; A ~:bad; B ~:bad |] ==>*
ya2 A B NA NB:set evs --> Guard NB (ya_keys A B NA NB evs) (spies evs)"
⟨*proof*⟩

end

29 Other Protocol-Independent Results

theory *Proto* **imports** *Guard_Public* **begin**

29.1 protocols

types *rule = "event set * event"*

syntax *msg' :: "rule => msg"*

translations *"msg' R" == "msg (snd R)"*

types *proto = "rule set"*

constdefs *wdef :: "proto => bool"*
"wdef p == ALL R k. R:p --> Number k:parts {msg' R}
--> Number k:parts (msg`(fst R))"

29.2 substitutions

record *subs =*
  agent :: "agent => agent"
  nonce :: "nat => nat"
  nb   :: "nat => msg"
  key   :: "key => key"

consts *apm :: "subs => msg => msg"*

primrec
"apm s (Agent A) = Agent (agent s A)"
"apm s (Nonce n) = Nonce (nonce s n)"
"apm s (Number n) = nb s n"
"apm s (Key K) = Key (key s K)"
"apm s (Hash X) = Hash (apm s X)"

```
"apm s (Crypt K X) = (
if (EX A. K = pubK A) then Crypt (pubK (agent s (agt K))) (apm s X)
else if (EX A. K = priK A) then Crypt (priK (agent s (agt K))) (apm s X)
else Crypt (key s K) (apm s X))"
"apm s {|X,Y|} = {|apm s X, apm s Y|}"
```

lemma *apm_parts: "X:parts {Y} ==> apm s X:parts {apm s Y}"*
⟨*proof*⟩

lemma *Nonce_apm [rule_format]: "Nonce n:parts {apm s X} ==>*
(ALL k. Number k:parts {X} --> Nonce n ~:parts {nb s k}) -->
(EX k. Nonce k:parts {X} & nonce s k = n)"
⟨*proof*⟩

lemma *wdef_Nonce: "[| Nonce n:parts {apm s X}; R:p; msg' R = X; wdef p;*
Nonce n ~:parts (apm s `(msg `(fst R))) |] ==>
(EX k. Nonce k:parts {X} & nonce s k = n)"
⟨*proof*⟩

consts *ap :: "subs => event => event"*

primrec
"ap s (Says A B X) = Says (agent s A) (agent s B) (apm s X)"
"ap s (Gets A X) = Gets (agent s A) (apm s X)"
"ap s (Notes A X) = Notes (agent s A) (apm s X)"

syntax
ap' :: "rule => msg"
apm' :: "rule => msg"
priK' :: "subs => agent => key"
pubK' :: "subs => agent => key"

translations
"ap' s R" == "ap s (snd R)"
"apm' s R" == "apm s (msg' R)"
"priK' s A" == "priK (agent s A)"
"pubK' s A" == "pubK (agent s A)"

29.3 nonces generated by a rule

constdefs *newn :: "rule => nat set"*
"newn R == {n. Nonce n:parts {msg (snd R)} & Nonce n ~:parts (msg`(fst R))}"

lemma *newn_parts: "n:newn R ==> Nonce (nonce s n):parts {apm' s R}"*
⟨*proof*⟩

29.4 traces generated by a protocol

constdefs *ok :: "event list => rule => subs => bool"*
"ok evs R s == ((ALL x. x:fst R --> ap s x:set evs)
& (ALL n. n:newn R --> Nonce (nonce s n) ~:used evs))"

consts *tr :: "proto => event list set"*

inductive *"tr p"* **intros**

```
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"
```

29.5 general properties

lemma *one_step_tr [iff]: "one_step (tr p)"*
⟨*proof*⟩

constdefs *has_only_Says' :: "proto => bool"*
"has_only_Says' p == ALL R. R:p --> is_Says (snd R)"

lemma *has_only_Says'D: "[| R:p; has_only_Says' p |]*
==> (EX A B X. snd R = Says A B X)"
⟨*proof*⟩

lemma *has_only_Says_tr [simp]: "has_only_Says' p ==> has_only_Says (tr p)"*
⟨*proof*⟩

lemma *has_only_Says'_in_trD: "[| has_only_Says' p; list @ ev # evs1 ∈ tr p |]*
==> (EX A B X. ev = Says A B X)"
⟨*proof*⟩

lemma *ok_not_used: "[| Nonce n ~:used evs; ok evs R s;*
ALL x. x:fst R --> is_Says x |] ==> Nonce n ~:parts (apm s `(msg `(fst R)))"
⟨*proof*⟩

lemma *ok_is_Says: "[| evs' @ ev # evs:tr p; ok evs R s; has_only_Says' p;*
R:p; x:fst R |] ==> is_Says x"
⟨*proof*⟩

29.6 types

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

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

29.7 introduction of a fresh guarded nonce

constdefs *fresh :: "proto => rule => subs => nat => key set => event list*
=> bool"
"fresh p R s n Ks evs == (EX evs1 evs2. evs = evs2 @ ap' s R # evs1
& Nonce n ~:used evs1 & R:p & ok evs1 R s & Nonce n:parts {apm' s R}
& apm' s R:guard n Ks)"

lemma *freshD: "fresh p R s n Ks evs ==> (EX evs1 evs2.*
evs = evs2 @ ap' s R # evs1 & Nonce n ~:used evs1 & R:p & ok evs1 R s
& Nonce n:parts {apm' s R} & apm' s R:guard n Ks)"

⟨*proof*⟩

lemma *freshI [intro]: "[| Nonce n ~:used evs1; R:p; Nonce n:parts {apm' s R};*
ok evs1 R s; apm' s R:guard n Ks |]
==> fresh p R s n Ks (list @ ap' s R # evs1)"
⟨*proof*⟩

lemma *freshI': "[| Nonce n ~:used evs1; (l,r):p;*
Nonce n:parts {apm s (msg r)}; ok evs1 (l,r) s; apm s (msg r):guard n Ks |]
==> fresh p (l,r) s n Ks (evs2 @ ap s r # evs1)"
⟨*proof*⟩

lemma *fresh_used: "[| fresh p R' s' n Ks evs; has_only_Says' p |]*
==> Nonce n:used evs"
⟨*proof*⟩

lemma *fresh_newn: "[| evs' @ ap' s R # evs:tr p; wdef p; has_only_Says' p;*
Nonce n ~:used evs; R:p; ok evs R s; Nonce n:parts {apm' s R} |]
==> EX k. k:newn R & nonce s k = n"
⟨*proof*⟩

lemma *fresh_rule: "[| evs' @ ev # evs:tr p; wdef p; Nonce n ~:used evs;*
Nonce n:parts {msg ev} |] ==> EX R s. R:p & ap' s R = ev"
⟨*proof*⟩

lemma *fresh_ruleD: "[| fresh p R' s' n Ks evs; keys R' s' n evs <= Ks; wdef p;*
has_only_Says' p; evs:tr p; ALL R k s. nonce s k = n --> Nonce n:used evs -->
R:p --> k:newn R --> Nonce n:parts {apm' s R} --> apm' s R:guard n Ks -->
apm' s R:parts (spies evs) --> keys R s n evs <= Ks --> P |] ==> P"
⟨*proof*⟩

29.8 safe keys

constdefs *safe :: "key set => msg set => bool"*
"safe Ks G == ALL K. K:Ks --> Key K ~:analz G"

lemma *safeD [dest]: "[| safe Ks G; K:Ks |] ==> Key K ~:analz G"*
⟨*proof*⟩

lemma *safe_insert: "safe Ks (insert X G) ==> safe Ks G"*
⟨*proof*⟩

lemma *Guard_safe: "[| Guard n Ks G; safe Ks G |] ==> Nonce n ~:analz G"*
⟨*proof*⟩

29.9 guardedness preservation

constdefs *preserv :: "proto => keyfun => nat => key set => bool"*
"preserv p keys n Ks == (ALL evs R' s' R s. evs:tr p -->
Guard n Ks (spies evs) --> safe Ks (spies evs) --> fresh p R' s' n Ks evs -->
keys R' s' n evs <= Ks --> R:p --> ok evs R s --> apm' s R:guard n Ks)"

lemma *preservD: "[| preserv p keys n Ks; evs:tr p; Guard n Ks (spies evs); safe Ks (spies evs); fresh p R' s' n Ks evs; R:p; ok evs R s; keys R' s' n evs <= Ks |] ==> apm' s R:guard n Ks"*
⟨*proof*⟩

lemma *preservD': "[| preserv p keys n Ks; evs:tr p; Guard n Ks (spies evs); safe Ks (spies evs); fresh p R' s' n Ks evs; (l,Says A B X):p; ok evs (l,Says A B X) s; keys R' s' n evs <= Ks |] ==> apm s X:guard n Ks"*
⟨*proof*⟩

29.10 monotonic keyfun

constdefs *monoton :: "proto => keyfun => bool"*
"monoton p keys == ALL R' s' n ev evs. ev # evs:tr p --> keys R' s' n evs <= keys R' s' n (ev # evs)"

lemma *monotonD [dest]: "[| keys R' s' n (ev # evs) <= Ks; monoton p keys; ev # evs:tr p |] ==> keys R' s' n evs <= Ks"*
⟨*proof*⟩

29.11 guardedness theorem

lemma *Guard_tr [rule_format]: "[| evs:tr p; has_only_Says' p; preserv p keys n Ks; monoton p keys; Guard n Ks (initState Spy) |] ==> safe Ks (spies evs) --> fresh p R' s' n Ks evs --> keys R' s' n evs <= Ks --> Guard n Ks (spies evs)"*
⟨*proof*⟩

29.12 useful properties for guardedness

lemma *newn_neq_used: "[| Nonce n:used evs; ok evs R s; k:newn R |] ==> n ~= nonce s k"*
⟨*proof*⟩

lemma *ok_Guard: "[| ok evs R s; Guard n Ks (spies evs); x:fst R; is_Says x |] ==> apm s (msg x):parts (spies evs) & apm s (msg x):guard n Ks"*
⟨*proof*⟩

lemma *ok_parts_not_new: "[| Y:parts (spies evs); Nonce (nonce s n):parts {Y}; ok evs R s |] ==> n ~:newn R"*
⟨*proof*⟩

29.13 unicity

constdefs *uniq :: "proto => secfun => bool"*
"uniq p secret == ALL evs R R' n n' Ks s s'. R:p --> R':p --> n:newn R --> n':newn R' --> nonce s n = nonce s' n' --> Nonce (nonce s n):parts {apm' s R} --> Nonce (nonce s n):parts {apm' s' R'} --> apm' s R:guard (nonce s n) Ks --> apm' s' R':guard (nonce s n) Ks --> evs:tr p --> Nonce (nonce s n) ~:analz (spies evs) -->

```
secret R n s Ks:parts (spies evs) --> secret R' n' s' Ks:parts (spies evs)
-->
secret R n s Ks = secret R' n' s' Ks"
```

lemma *uniqD: "[| uniq p secret; evs: tr p; R:p; R':p; n:newn R; n':newn R';*
nonce s n = nonce s' n'; Nonce (nonce s n) ~:analz (spies evs);
Nonce (nonce s n):parts {apm' s R}; Nonce (nonce s n):parts {apm' s' R'};
secret R n s Ks:parts (spies evs); secret R' n' s' Ks:parts (spies evs);
apm' s R:guard (nonce s n) Ks; apm' s' R':guard (nonce s n) Ks |] ==>
secret R n s Ks = secret R' n' s' Ks"
⟨*proof*⟩

constdefs *ord :: "proto => (rule => rule => bool) => bool"*
"ord p inf == ALL R R'. R:p --> R':p --> ~ inf R R' --> inf R' R"

lemma *ordD: "[| ord p inf; ~ inf R R'; R:p; R':p |] ==> inf R' R"*
⟨*proof*⟩

constdefs *uniq' :: "proto => (rule => rule => bool) => secfun => bool"*
"uniq' p inf secret == ALL evs R R' n n' Ks s s'. R:p --> R':p -->
inf 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 inf secret; evs: tr p; inf R R'; R:p; R':p; n:newn*
R;
n':newn R'; nonce s n = nonce s' n'; Nonce (nonce s n) ~:analz (spies evs);
Nonce (nonce s n):parts {apm' s R}; Nonce (nonce s n):parts {apm' s' R'};
secret R n s Ks:parts (spies evs); secret R' n' s' Ks:parts (spies evs);
apm' s R:guard (nonce s n) Ks; apm' s' R':guard (nonce s n) Ks |] ==>
secret R n s Ks = secret R' n' s' Ks"
⟨*proof*⟩

lemma *uniq'_imp_uniq: "[| uniq' p inf secret; ord p inf |] ==> uniq p secret"*
⟨*proof*⟩

29.14 Needham-Schroeder-Lowe

constdefs
a :: agent "a == Friend 0"
b :: agent "b == Friend 1"
a' :: agent "a' == Friend 2"
b' :: agent "b' == Friend 3"
Na :: nat "Na == 0"
Nb :: nat "Nb == 1"

consts
ns :: proto
ns1 :: rule

```
ns2 :: rule
ns3 :: rule
```

translations
```
"ns1" == "({}, Says a b (Crypt (pubK b) {|Nonce Na, Agent a|}))"

"ns2" == "({Says a' b (Crypt (pubK b) {|Nonce Na, Agent a|})},
Says b a (Crypt (pubK a) {|Nonce Na, Nonce Nb, Agent b|}))"

"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 *ns* **intros**
```
[iff]: "ns1:ns"
[iff]: "ns2:ns"
[iff]: "ns3:ns"
```

syntax
```
ns3a :: msg
ns3b :: msg
```

translations
```
"ns3a" => "Says a b (Crypt (pubK b) {|Nonce Na, Agent a|})"
"ns3b" => "Says b' a (Crypt (pubK a) {|Nonce Na, Nonce Nb, Agent b|})"
```

constdefs *keys :: "keyfun"*
```
"keys R' s' n evs == {priK' s' a, priK' s' b}"
```

lemma *"monoton ns keys"*
⟨*proof*⟩

constdefs *secret :: "secfun"*
```
"secret R n s Ks ==
(if R=ns1 then apm s (Crypt (pubK b) {|Nonce Na, Agent a|})
else if R=ns2 then apm s (Crypt (pubK a) {|Nonce Na, Nonce Nb, Agent b|})
else Number 0)"
```

constdefs *inf :: "rule => rule => bool"*
```
"inf R R' == (R=ns1 | (R=ns2 & R'~=ns1) | (R=ns3 & R'=ns3))"
```

lemma *inf_is_ord [iff]: "ord ns inf"*
⟨*proof*⟩

29.15 general properties

lemma *ns_has_only_Says' [iff]: "has_only_Says' ns"*
⟨*proof*⟩

lemma *newn_ns1 [iff]: "newn ns1 = {Na}"*
⟨*proof*⟩

lemma *newn_ns2 [iff]: "newn ns2 = {Nb}"*
⟨*proof*⟩

lemma *newn_ns3 [iff]: "newn ns3 = {}"*
⟨*proof*⟩

lemma *ns_wdef [iff]: "wdef ns"*
⟨*proof*⟩

29.16 guardedness for NSL

lemma *"uniq ns secret ==> preserv ns keys n Ks"*
⟨*proof*⟩

29.17 unicity for NSL

lemma *"uniq' ns inf secret"*
⟨*proof*⟩

end