Inductive Study of Confidentiality

Giampaolo Bella
Dipartimento di Matematica e Informatica, Università di Catania, Italy

September 23, 2021

Abstract

This document contains the full theory files accompanying article "Inductive Study of Confidentiality — for Everyone" [1]. They aim at an illustrative and didactic presentation of the Inductive Method of protocol analysis, focusing on the treatment of one of the main goals of security protocols: confidentiality against a threat model. The treatment of confidentiality, which in fact forms a key aspect of all protocol analysis tools, has been found cryptic by many learners of the Inductive Method, hence the motivation for this work. The theory files in this document guide the reader step by step towards design and proof of significant confidentiality theorems. These are developed against two threat models, the standard Dolev-Yao and a more audacious one, the General Attacker, which turns out to be particularly useful also for teaching purposes.

Contents

1 Theory of Agents and Messages for Security Protocols against Dolev-Yao

theory *Message*
imports *Main*
begin

lemma [*simp*] : $A \cup (B \cup A) = B \cup A$
⟨*proof*⟩

type-synonym
key = *nat*

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

specification (*invKey*)
invKey [*simp*]: *invKey* (*invKey K*) = *K*
invKey-symmetric: *all-symmetric* −−> *invKey* = *id*
⟨*proof*⟩

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

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

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

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

Concrete syntax: messages appear as {|*A*,*B*,*NA*|}, etc...

syntax
-*MTuple* :: [′*a*, *args*] => ′*a* ∗ ′*b* ((*2*{|-,/ -|}))
translations
{|*x*, *y*, *z*|} == {|*x*, {|*y*, *z*|}|}

$\{\!|x,\ y|\!\}$ == *CONST MPair x y*

definition *HPair* :: [*msg*,*msg*] => *msg* ((*4Hash*[-] /-) [*0*, *1000*]) **where**
— Message Y paired with a MAC computed with the help of X
Hash[*X*] *Y* == $\{\!|$ *Hash*$\{\!|X,Y|\!\}$, $Y|\!\}$

definition *keysFor* :: *msg set* => *key set* **where**
— Keys useful to decrypt elements of a message set
keysFor H == *invKey* ‘ $\{K.\ \exists X.\ Crypt\ K\ X \in H\}$

1.1 Inductive definition of all parts of a message

inductive-set
parts :: *msg set* => *msg set*
for *H* :: *msg set*
where
Inj [*intro*]: $X \in H$ ==> $X \in$ *parts H*
| *Fst*: $\{\!|X,Y|\!\} \in$ *parts H* ==> $X \in$ *parts H*
| *Snd*: $\{\!|X,Y|\!\} \in$ *parts H* ==> $Y \in$ *parts H*
| *Body*: *Crypt K X* $\in$ *parts H* ==> $X \in$ *parts H*

Monotonicity

lemma *parts-mono*: $G \subseteq H$ ==> *parts*(G) $\subseteq$ *parts*(H)
⟨*proof*⟩

Equations hold because constructors are injective.

lemma *Friend-image-eq* [*simp*]: (*Friend x* $\in$ *Friend*‘*A*) = (*x*:*A*)
⟨*proof*⟩

lemma *Key-image-eq* [*simp*]: (*Key x* $\in$ *Key*‘*A*) = ($x \in A$)
⟨*proof*⟩

lemma *Nonce-Key-image-eq* [*simp*]: (*Nonce x* $\notin$ *Key*‘*A*)
⟨*proof*⟩

1.2 Inverse of keys

lemma *invKey-eq* [*simp*]: (*invKey K* = *invKey K'*) = ($K=K'$)
⟨*proof*⟩

1.3 keysFor operator

lemma *keysFor-empty* [*simp*]: *keysFor* {} = {}
⟨*proof*⟩

lemma *keysFor-Un* [*simp*]: *keysFor* ($H \cup H'$) = *keysFor H* $\cup$ *keysFor H'*
⟨*proof*⟩

lemma *keysFor-UN* [*simp*]: *keysFor* ($\bigcup$*i*∈*A*. *H i*) = ($\bigcup$*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*⟩

1.4 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*: $H \subseteq parts(H)$
⟨*proof*⟩

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

lemma *parts-empty* [*simp*]: $parts\{\} = \{\}$
⟨*proof*⟩

lemma *parts-emptyE* [*elim*!]: $X \in parts\{\} ==> P$
⟨*proof*⟩

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

lemma *parts-singleton*: $X \in parts\ H ==> \exists\, Y \in H.\ X \in parts\ \{Y\}$
⟨*proof*⟩

1.4.1 Unions

lemma *parts-Un-subset1*: $parts(G) \cup parts(H) \subseteq parts(G \cup H)$
⟨*proof*⟩

lemma *parts-Un-subset2*: $parts(G \cup H) \subseteq parts(G) \cup parts(H)$
⟨*proof*⟩

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

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

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

lemma *parts-insert2*:
$parts\ (insert\ X\ (insert\ Y\ H)) = parts\ \{X\} \cup parts\ \{Y\} \cup parts\ H$
⟨*proof*⟩

lemma *parts-UN-subset1*: $(\bigcup x \in A.\ parts(H\ x)) \subseteq parts(\bigcup x \in A.\ H\ x)$
⟨*proof*⟩

lemma *parts-UN-subset2*: $parts(\bigcup x \in A.\ H\ x) \subseteq (\bigcup x \in A.\ parts(H\ x))$
⟨*proof*⟩

lemma *parts-UN* [*simp*]: $parts(\bigcup x \in A.\ H\ x) = (\bigcup x \in A.\ parts(H\ x))$
⟨*proof*⟩

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

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

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

lemma *parts-insert-subset*: *insert X* (*parts H*) ⊆ *parts*(*insert X H*)
⟨*proof*⟩

1.4.2 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*⟩

1.4.3 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*⟩

1.5 Inductive relation "analz"

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

inductive-set
 analz :: *msg set* => *msg set*
 for *H* :: *msg set*
 where
  Inj [*intro*,*simp*] : *X* ∈ *H* ==> *X* ∈ *analz H*
 | *Fst*: ⦃*X*, *Y*⦄ ∈ *analz H* ==> *X* ∈ *analz H*
 | *Snd*: ⦃*X*, *Y*⦄ ∈ *analz H* ==> *Y* ∈ *analz H*
 | *Decrypt* [*dest*]:
    [|*Crypt K X* ∈ *analz H*; *Key*(*invKey K*): *analz H*|] ==> *X* ∈ *analz H*

Monotonicity; Lemma 1 of Lowe's paper

lemma *analz-mono*: *G*⊆*H* ==> *analz*(*G*) ⊆ *analz*(*H*)
⟨*proof*⟩

Making it safe speeds up proofs

lemma *MPair-analz* [*elim!*]:
  [| ⦃*X*, *Y*⦄ ∈ *analz H*;
    [| *X* ∈ *analz H*; *Y* ∈ *analz H* |] ==> *P*
   |] ==> *P*
⟨*proof*⟩

lemma *analz-increasing*: *H* ⊆ *analz*(*H*)

⟨*proof*⟩

lemma *analz-subset-parts*: *analz H* ⊆ *parts H*
⟨*proof*⟩

lemmas *analz-into-parts* = *analz-subset-parts* [*THEN subsetD*]

lemmas *not-parts-not-analz* = *analz-subset-parts* [*THEN contra-subsetD*]

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*]

1.5.1 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*⟩

1.5.2 Rewrite rules for pulling out atomic messages

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

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

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

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

lemma *analz-insert-Hash* [*simp*]:
  analz (*insert* (*Hash X*) *H*) = *insert* (*Hash X*) (*analz H*)

⟨*proof*⟩

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

lemma *analz-insert-Key* [*simp*]:
  $K \notin keysFor\ (analz\ H) ==>$
    $analz\ (insert\ (Key\ K)\ H) = insert\ (Key\ K)\ (analz\ H)$
⟨*proof*⟩

lemma *analz-insert-MPair* [*simp*]:
  $analz\ (insert\ \{\!|X, Y|\!\}\ H) =$
    $insert\ \{\!|X, Y|\!\}\ (analz\ (insert\ X\ (insert\ Y\ H)))$
⟨*proof*⟩

Can pull out enCrypted message if the Key is not known

lemma *analz-insert-Crypt*:
  $Key\ (invKey\ K) \notin analz\ H$
  $==> analz\ (insert\ (Crypt\ K\ X)\ H) = insert\ (Crypt\ K\ X)\ (analz\ H)$
⟨*proof*⟩

lemma *lemma1*: $Key\ (invKey\ K) \in analz\ H ==>$
    $analz\ (insert\ (Crypt\ K\ X)\ H) \subseteq$
    $insert\ (Crypt\ K\ X)\ (analz\ (insert\ X\ H))$
⟨*proof*⟩

lemma *lemma2*: $Key\ (invKey\ K) \in analz\ H ==>$
    $insert\ (Crypt\ K\ X)\ (analz\ (insert\ X\ H)) \subseteq$
    $analz\ (insert\ (Crypt\ K\ X)\ H)$
⟨*proof*⟩

lemma *analz-insert-Decrypt*:
  $Key\ (invKey\ K) \in analz\ H ==>$
    $analz\ (insert\ (Crypt\ K\ X)\ H) =$
    $insert\ (Crypt\ K\ X)\ (analz\ (insert\ X\ H))$
⟨*proof*⟩

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

lemma *analz-Crypt-if* [*simp*]:
  $analz\ (insert\ (Crypt\ K\ X)\ H) =$
    $(if\ (Key\ (invKey\ K) \in analz\ H)$
     $then\ insert\ (Crypt\ K\ X)\ (analz\ (insert\ X\ H))$
     $else\ insert\ (Crypt\ K\ X)\ (analz\ H))$
⟨*proof*⟩

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

lemma *analz-insert-Crypt-subset*:
  $analz\ (insert\ (Crypt\ K\ X)\ H) \subseteq$

insert (*Crypt K X*) (*analz* (*insert X H*))
⟨*proof*⟩

lemma *analz-image-Key* [*simp*]: *analz* (*Key`N*) = *Key`N*
⟨*proof*⟩

1.5.3 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*:

[| $\forall X\ Y.\ \{|X,Y|\} \notin H;\ \ \forall X\ K.\ Crypt\ K\ X \notin H$ |] ==> $analz\ H = H$
⟨*proof*⟩

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

lemma *analz-UN-analz-lemma*:
$X\in analz\ (\bigcup i\in A.\ analz\ (H\ i))$ ==> $X\in analz\ (\bigcup i\in A.\ H\ i)$
⟨*proof*⟩

lemma *analz-UN-analz* [*simp*]: $analz\ (\bigcup i\in A.\ analz\ (H\ i)) = analz\ (\bigcup i\in A.\ H\ i)$
⟨*proof*⟩

1.6 Inductive relation "synth"

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

inductive-set
synth :: *msg set* => *msg set*
for *H* :: *msg set*
where
Inj [*intro*]: $X \in H$ ==> $X \in synth\ H$
| *Agent* [*intro*]: $Agent\ agt \in synth\ H$
| *Number* [*intro*]: $Number\ n \in synth\ H$
| *Hash* [*intro*]: $X \in synth\ H$ ==> $Hash\ X \in synth\ H$
| *MPair* [*intro*]: [|$X \in synth\ H$; $Y \in synth\ H$|] ==> $\{|X,Y|\} \in synth\ H$
| *Crypt* [*intro*]: [|$X \in synth\ H$; $Key(K) \in H$|] ==> $Crypt\ K\ X \in synth\ H$

Monotonicity

lemma *synth-mono*: $G\subseteq H$ ==> $synth(G) \subseteq synth(H)$
⟨*proof*⟩

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

inductive-simps *synth-simps* [*iff*]:
$Nonce\ n \in synth\ H$
$Key\ K \in synth\ H$
$Hash\ X \in synth\ H$
$\{|X,Y|\} \in synth\ H$
$Crypt\ K\ X \in synth\ H$

lemma *synth-increasing*: $H \subseteq synth(H)$
⟨*proof*⟩

1.6.1 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*: $synth(G) \cup synth(H) \subseteq synth(G \cup H)$
⟨*proof*⟩

lemma *synth-insert*: $insert\ X\ (synth\ H) \subseteq synth(insert\ X\ H)$
⟨*proof*⟩

1.6.2 Idempotence and transitivity

lemma *synth-synthD* [*dest*!]: $X \in synth\ (synth\ H) ==> X \in synth\ H$
⟨*proof*⟩

lemma *synth-idem*: $synth\ (synth\ H) = synth\ H$
⟨*proof*⟩

lemma *synth-subset-iff* [*simp*]: $(synth\ G \subseteq synth\ H) = (G \subseteq synth\ H)$
⟨*proof*⟩

lemma *synth-trans*: $[|\ X \in synth\ G;\ \ G \subseteq synth\ H\ |] ==> X \in synth\ H$
⟨*proof*⟩

Cut; Lemma 2 of Lowe

lemma *synth-cut*: $[|\ Y \in synth\ (insert\ X\ H);\ \ X \in synth\ H\ |] ==> Y \in synth\ H$
⟨*proof*⟩

lemma *Agent-synth* [*simp*]: $Agent\ A \in synth\ H$
⟨*proof*⟩

lemma *Number-synth* [*simp*]: $Number\ n \in synth\ H$
⟨*proof*⟩

lemma *Nonce-synth-eq* [*simp*]: $(Nonce\ N \in synth\ H) = (Nonce\ N \in H)$
⟨*proof*⟩

lemma *Key-synth-eq* [*simp*]: $(Key\ K \in synth\ H) = (Key\ K \in H)$
⟨*proof*⟩

lemma *Crypt-synth-eq* [*simp*]:
$Key\ K \notin H ==> (Crypt\ K\ X \in synth\ H) = (Crypt\ K\ X \in H)$
⟨*proof*⟩

lemma *keysFor-synth* [*simp*]:
$keysFor\ (synth\ H) = keysFor\ H \cup invKey`\{K.\ Key\ K \in H\}$
⟨*proof*⟩

1.6.3 Combinations of parts, analz and synth

lemma *parts-synth* [*simp*]: $parts\ (synth\ H) = parts\ H \cup synth\ H$
⟨*proof*⟩

lemma *analz-analz-Un* [*simp*]: *analz* (*analz G* ∪ *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*⟩

1.6.4 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. See also *Fake-parts-sing* below

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* |]

$\quad ==> (Crypt\ K\ X \in synth\ (analz\ H)) = (X \in synth\ (analz\ H))$
⟨proof⟩

lemma *Hash-synth-analz* [*simp*]:
$\quad X \notin synth\ (analz\ H)$
$\quad ==> (Hash\{\!|X,Y|\!\} \in synth\ (analz\ H)) = (Hash\{\!|X,Y|\!\} \in analz\ H)$
⟨proof⟩

1.7 HPair: a combination of Hash and MPair

1.7.1 Freeness

lemma *Agent-neq-HPair*: $Agent\ A\ {}^{\sim}= Hash[X]\ Y$
⟨proof⟩

lemma *Nonce-neq-HPair*: $Nonce\ N\ {}^{\sim}= Hash[X]\ Y$
⟨proof⟩

lemma *Number-neq-HPair*: $Number\ N\ {}^{\sim}= Hash[X]\ Y$
⟨proof⟩

lemma *Key-neq-HPair*: $Key\ K\ {}^{\sim}= Hash[X]\ Y$
⟨proof⟩

lemma *Hash-neq-HPair*: $Hash\ Z\ {}^{\sim}= Hash[X]\ Y$
⟨proof⟩

lemma *Crypt-neq-HPair*: $Crypt\ K\ X'\ {}^{\sim}= Hash[X]\ Y$
⟨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*]: $(Hash[X']\ Y' = Hash[X]\ Y) = (X' = X \wedge Y'=Y)$
⟨proof⟩

lemma *MPair-eq-HPair* [*iff*]:
$\quad (\{\!|X',Y'|\!\} = Hash[X]\ Y) = (X' = Hash\{\!|X,Y|\!\} \wedge Y'=Y)$
⟨proof⟩

lemma *HPair-eq-MPair* [*iff*]:
$\quad (Hash[X]\ Y = \{\!|X',Y'|\!\}) = (X' = Hash\{\!|X,Y|\!\} \wedge Y'=Y)$
⟨proof⟩

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

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

lemma *parts-insert-HPair* [*simp*]:
parts (*insert* (*Hash*[*X*] *Y*) *H*) =
insert (*Hash*[*X*] *Y*) (*insert* (*Hash*⦃*X*,*Y*⦄) (*parts* (*insert* *Y* *H*)))
⟨*proof*⟩

lemma *analz-insert-HPair* [*simp*]:
analz (*insert* (*Hash*[*X*] *Y*) *H*) =
insert (*Hash*[*X*] *Y*) (*insert* (*Hash*⦃*X*,*Y*⦄) (*analz* (*insert* *Y* *H*)))
⟨*proof*⟩

lemma *HPair-synth-analz* [*simp*]:
X ∉ *synth* (*analz* *H*)
==> (*Hash*[*X*] *Y* ∈ *synth* (*analz* *H*)) =
(*Hash* ⦃*X*, *Y*⦄ ∈ *analz* *H* ∧ *Y* ∈ *synth* (*analz* *H*))
⟨*proof*⟩

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

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

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

lemmas *pushKeys* =
insert-commute [*of Key K Agent C*]
insert-commute [*of Key K Nonce N*]
insert-commute [*of Key K Number N*]
insert-commute [*of Key K Hash X*]
insert-commute [*of Key K MPair X Y*]
insert-commute [*of Key K Crypt X K′*]
for *K C N X Y K′*

lemmas *pushCrypts* =
insert-commute [*of Crypt X K Agent C*]
insert-commute [*of Crypt X K Agent C*]
insert-commute [*of Crypt X K Nonce N*]
insert-commute [*of Crypt X K Number N*]
insert-commute [*of Crypt X K Hash X′*]
insert-commute [*of Crypt X K MPair X′ Y*]
for *X K C N X′ Y*

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

lemmas *pushes* = *pushKeys pushCrypts*

1.8 The set of key-free messages

inductive-set
  keyfree :: *msg set*
  where
    Agent: *Agent A* ∈ *keyfree*
  | *Number*: *Number N* ∈ *keyfree*
  | *Nonce*: *Nonce N* ∈ *keyfree*
  | *Hash*: *Hash X* ∈ *keyfree*
  | *MPair*: [|*X* ∈ *keyfree*; *Y* ∈ *keyfree*|] ==> $\{\!|X, Y|\!\}$ ∈ *keyfree*
  | *Crypt*: [|*X* ∈ *keyfree*|] ==> *Crypt K X* ∈ *keyfree*

declare *keyfree.intros* [*intro*]

inductive-cases *keyfree-KeyE*: *Key K* ∈ *keyfree*
inductive-cases *keyfree-MPairE*: $\{\!|X, Y|\!\}$ ∈ *keyfree*
inductive-cases *keyfree-CryptE*: *Crypt K X* ∈ *keyfree*

lemma *parts-keyfree*: *parts* (*keyfree*) ⊆ *keyfree*
  ⟨*proof*⟩

lemma *analz-keyfree-into-Un*: [[*X* ∈ *analz* (*G* ∪ *H*); *G* ⊆ *keyfree*]] ⟹ *X* ∈ *parts* *G* ∪ *analz H*
⟨*proof*⟩

1.9 Tactics useful for many protocol proofs

⟨*ML*⟩

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

declare *o-def* [*simp*]

lemma *Crypt-notin-image-Key* [*simp*]: *Crypt K X* ∉ *Key* ‘ *A*
⟨*proof*⟩

lemma *Hash-notin-image-Key* [*simp*] :*Hash X* ∉ *Key* ‘ *A*
⟨*proof*⟩

lemma *synth-analz-mono*: *G*⊆*H* ==> *synth* (*analz*(*G*)) ⊆ *synth* (*analz*(*H*))
⟨*proof*⟩

lemma *Fake-analz-eq* [*simp*]:
    X ∈ *synth*(*analz H*) ==> *synth* (*analz* (*insert X H*)) = *synth* (*analz H*)
⟨*proof*⟩

Two generalizations of *analz-insert-eq*

lemma *gen-analz-insert-eq* [*rule-format*]:
$X \in analz\ H ==> \forall G.\ H \subseteq G --> analz\ (insert\ X\ G) = analz\ G$
⟨*proof*⟩

lemma *synth-analz-insert-eq* [*rule-format*]:
$X \in synth\ (analz\ H)$
$==> \forall G.\ H \subseteq G --> (Key\ K \in analz\ (insert\ X\ G)) = (Key\ K \in analz\ G)$
⟨*proof*⟩

lemma *Fake-parts-sing*:
$X \in synth\ (analz\ H) ==> parts\{X\} \subseteq synth\ (analz\ H) \cup parts\ H$
⟨*proof*⟩

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

⟨*ML*⟩

end

2 Theory of Events for Security Protocols against Dolev-Yao

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

consts
initState :: *agent* => *msg set*

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

consts
bad :: *agent set* — compromised agents

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

specification (*bad*)
Spy-in-bad [*iff*]: $Spy \in bad$
Server-not-bad [*iff*]: $Server \notin bad$
⟨*proof*⟩

primrec *knows* :: *agent* => *event list* => *msg set*
where
knows-Nil: *knows A* [] = *initState A*
| *knows-Cons*:
knows A (*ev* # *evs*) =
(*if A* = *Spy then*

```
(case ev of
   Says A' B X => insert X (knows Spy evs)
 | Gets A' X => knows Spy evs
 | Notes A' X =>
     if A' ∈ bad then insert X (knows Spy evs) else knows Spy evs)
else
(case ev of
   Says A' B X =>
     if A'=A then insert X (knows A evs) else knows A evs
 | Gets A' X    =>
     if A'=A then insert X (knows A evs) else knows A evs
 | Notes A' X    =>
     if A'=A then insert X (knows A evs) else knows A evs))
```

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

abbreviation (*input*)
spies :: *event list => msg set* **where**
spies == knows Spy

primrec *used* :: *event list => msg set*
where
used-Nil: *used* [] = (*UN B. parts* (*initState B*))
| *used-Cons*: *used* (*ev # evs*) =
(*case ev of*
Says A B X => parts {*X*} ∪ *used evs*
| *Gets A X => used evs*
| *Notes A X => parts* {*X*} ∪ *used evs*)

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

lemma *Notes-imp-used* [*rule-format*]: *Notes A X* ∈ *set evs* −−> *X* ∈ *used evs*
⟨*proof*⟩

lemma *Says-imp-used* [*rule-format*]: *Says A B X* ∈ *set evs* −−> *X* ∈ *used evs*
⟨*proof*⟩

2.1 Function *knows*

lemmas *parts-insert-knows-A* = *parts-insert* [*of - knows A evs*] **for** *A evs*

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

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

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

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

lemma *knows-Spy-subset-knows-Spy-Says*:
knows Spy evs ⊆ *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 *Says-imp-parts-knows-Spy* =
Says-imp-knows-Spy [*THEN parts.Inj*, *THEN revcut-rl*]

lemmas *knows-Spy-partsEs* =
Says-imp-parts-knows-Spy parts.Body [*THEN revcut-rl*]

lemmas *Says-imp-analz-Spy* = *Says-imp-knows-Spy* [*THEN analz.Inj*]

Compatibility for the old "spies" function

lemmas *spies-partsEs* = *knows-Spy-partsEs*
lemmas *Says-imp-spies* = *Says-imp-knows-Spy*
lemmas *parts-insert-spies* = *parts-insert-knows-A* [*of - Spy*]

2.2 Knowledge of Agents

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

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

lemma *knows-Gets*:
  $A \neq Spy$ −−> *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*]:
  $A \neq Spy$ −−> *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*; $A \neq Spy$ |] ==> ∃ *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* |] ==> ∃ *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*: $X \in$ *parts* (*initState B*) ==> $X \in$ *used evs*
⟨*proof*⟩

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

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

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

lemma *used-nil-subset*: *used* [] ⊆ *used evs*
⟨*proof*⟩

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

declare *knows-Cons* [*simp del*]
used-Nil [*simp del*] *used-Cons* [*simp del*]

For proving theorems of the form $X \notin$ *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*]

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

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

⟨*ML*⟩

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

lemmas *syan-impI* = *impI* [**where** $P = Y \notin$ *synth* (*analz* (*knows Spy evs*))] **for** *Y evs*

⟨*ML*⟩

end

3 Theory of Cryptographic Keys for Security Protocols against Dolev-Yao

theory *Public*
imports *Event*
begin

lemma *invKey-K*: $K \in$ *symKeys* ==> *invKey K* = *K*
⟨*proof*⟩

3.1 Asymmetric Keys

datatype *keymode* = *Signature* | *Encryption*

consts
publicKey :: [*keymode*,*agent*] => *key*

abbreviation
pubEK :: *agent* => *key* **where**
pubEK == *publicKey Encryption*

abbreviation
pubSK :: *agent* => *key* **where**
pubSK == *publicKey Signature*

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

abbreviation

priEK :: *agent* => *key* **where**
priEK A == *privateKey Encryption A*

abbreviation
priSK :: *agent* => *key* **where**
priSK A == *privateKey Signature A*

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

abbreviation
  pubK :: *agent* => *key* **where**
  pubK A == *pubEK A*

abbreviation
  priK :: *agent* => *key* **where**
  priK A == *invKey* (*pubEK A*)

By freeness of agents, no two agents have the same key. Since *True* ≠ *False*, no agent has identical signing and encryption keys

specification (*publicKey*)
  injective-publicKey:
    publicKey b A = *publicKey c A′* ==> *b*=*c* ∧ *A*=*A′*
  ⟨*proof*⟩

axiomatization where

  privateKey-neq-publicKey [*iff*]: *privateKey b A* ≠ *publicKey c A′*

lemmas *publicKey-neq-privateKey* = *privateKey-neq-publicKey* [*THEN not-sym*]
declare *publicKey-neq-privateKey* [*iff*]

3.2 Basic properties of *pubK* and *priEK*

lemma *publicKey-inject* [*iff*]: (*publicKey b A* = *publicKey c A′*) = (*b*=*c* ∧ *A*=*A′*)
⟨*proof*⟩

lemma *not-symKeys-pubK* [*iff*]: *publicKey b A* ∉ *symKeys*
⟨*proof*⟩

lemma *not-symKeys-priK* [*iff*]: *privateKey b A* ∉ *symKeys*
⟨*proof*⟩

lemma *symKey-neq-priEK*: *K* ∈ *symKeys* ==> *K* ≠ *priEK A*
⟨*proof*⟩

lemma *symKeys-neq-imp-neq*: (*K* ∈ *symKeys*) ≠ (*K′* ∈ *symKeys*) ==> *K* ≠ *K′*
⟨*proof*⟩

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

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

3.3 "Image" equations that hold for injective functions

lemma *invKey-image-eq* [*simp*]: (*invKey* $x \in$ *invKey*‘*A*) = ($x \in A$)
⟨*proof*⟩

lemma *publicKey-image-eq* [*simp*]:
(*publicKey* b $x \in$ *publicKey* c ‘ *AA*) = (b=c ∧ $x \in AA$)
⟨*proof*⟩

lemma *privateKey-notin-image-publicKey* [*simp*]: *privateKey* b $x \notin$ *publicKey* c ‘ *AA*
⟨*proof*⟩

lemma *privateKey-image-eq* [*simp*]:
(*privateKey* b $A \in$ *invKey* ‘ *publicKey* c ‘ *AS*) = (b=c ∧ $A \in AS$)
⟨*proof*⟩

lemma *publicKey-notin-image-privateKey* [*simp*]: *publicKey* b $A \notin$ *invKey* ‘ *publicKey* c ‘ *AS*
⟨*proof*⟩

3.4 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*⟩

axiomatization where
sym-shrK [*iff*]: *shrK X* ∈ *symKeys* — All shared keys are symmetric

Injectiveness: Agents' long-term keys are distinct.

lemmas *shrK-injective* = *inj-shrK* [*THEN inj-eq*]
declare *shrK-injective* [*iff*]

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

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

lemma *analz-Decrypt'*:

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

lemma *priK-neq-shrK* [*iff*]: *shrK A* ≠ *privateKey b C*
⟨*proof*⟩

lemmas *shrK-neq-priK* = *priK-neq-shrK* [*THEN not-sym*]
declare *shrK-neq-priK* [*simp*]

lemma *pubK-neq-shrK* [*iff*]: *shrK A* ≠ *publicKey b C*
⟨*proof*⟩

lemmas *shrK-neq-pubK* = *pubK-neq-shrK* [*THEN not-sym*]
declare *shrK-neq-pubK* [*simp*]

lemma *priEK-noteq-shrK* [*simp*]: *priEK A* ≠ *shrK B*
⟨*proof*⟩

lemma *publicKey-notin-image-shrK* [*simp*]: *publicKey b x* ∉ *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*]

3.5 Initial States of Agents

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

overloading
initState ≡ *initState*
begin

primrec *initState* **where**

initState-Server:

initState Server =
{*Key* (*priEK Server*), *Key* (*priSK Server*)} ∪
(*Key* ‘ *range pubEK*) ∪ (*Key* ‘ *range pubSK*) ∪ (*Key* ‘ *range shrK*)

| *initState-Friend*:
initState (*Friend i*) =
{*Key* (*priEK*(*Friend i*)), *Key* (*priSK*(*Friend i*)), *Key* (*shrK*(*Friend i*))} ∪
(*Key* ‘ *range pubEK*) ∪ (*Key* ‘ *range pubSK*)

| *initState-Spy*:
initState Spy =
(*Key* ‘ *invKey* ‘ *pubEK* ‘ *bad*) ∪ (*Key* ‘ *invKey* ‘ *pubSK* ‘ *bad*) ∪
(*Key* ‘ *shrK* ‘ *bad*) ∪
(*Key* ‘ *range pubEK*) ∪ (*Key* ‘ *range pubSK*)

end

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

lemma *used-parts-subset-parts* [*rule-format*]:
∀ *X* ∈ *used evs*. *parts* {*X*} ⊆ *used evs*
⟨*proof*⟩

lemma *MPair-used-D*: $\{\!|X, Y|\!\}$ ∈ *used H* ==> *X* ∈ *used H* ∧ *Y* ∈ *used H*
⟨*proof*⟩

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

lemma *MPair-used* [*elim*!]:
[| $\{\!|X, Y|\!\}$ ∈ *used H*;
[| *X* ∈ *used H*; *Y* ∈ *used H* |] ==> *P* |]
==> *P*
⟨*proof*⟩

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

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

lemma *Crypt-notin-initState*: *Crypt K X* ∉ *parts* (*initState B*)
⟨*proof*⟩

lemma *Crypt-notin-used-empty* [*simp*]: *Crypt K X* ∉ *used* []
⟨*proof*⟩

lemma *shrK-in-initState* [*iff*]: *Key* (*shrK A*) ∈ *initState A*
⟨*proof*⟩

lemma *shrK-in-knows* [*iff*]: *Key* (*shrK A*) ∈ *knows A evs*
⟨*proof*⟩

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

lemma *Key-not-used* [*simp*]: *Key K* ∉ *used evs* ==> *K* ∉ *range shrK*
⟨*proof*⟩

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

lemmas *neq-shrK* = *shrK-neq* [*THEN not-sym*]
declare *neq-shrK* [*simp*]

3.6 Function *knows Spy*

lemma *not-SignatureE* [*elim*!]: *b* ≠ *Signature* ⟹ *b* = *Encryption*
⟨*proof*⟩

Agents see their own private keys!

lemma *priK-in-initState* [*iff*]: *Key* (*privateKey b A*) ∈ *initState A*
⟨*proof*⟩

Agents see all public keys!

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

All public keys are visible

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

lemmas *analz-spies-pubK* = *spies-pubK* [*THEN analz.Inj*]
declare *analz-spies-pubK* [*iff*]

Spy sees private keys of bad agents!

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

Spy sees long-term shared keys of bad agents!

lemma *Spy-spies-bad-shrK* [*intro*!]:
$A \in bad$ ==> *Key* (*shrK A*) $\in$ *spies evs*
⟨*proof*⟩

lemma *publicKey-into-used* [*iff*] :*Key* (*publicKey b A*) $\in$ *used evs*
⟨*proof*⟩

lemma *privateKey-into-used* [*iff*]: *Key* (*privateKey b A*) $\in$ *used evs*
⟨*proof*⟩

lemma *Crypt-Spy-analz-bad*:
[| *Crypt* (*shrK A*) $X \in$ *analz* (*knows Spy evs*); $A \in bad$ |]
==> $X \in$ *analz* (*knows Spy evs*)
⟨*proof*⟩

3.7 Fresh Nonces

lemma *Nonce-notin-initState* [*iff*]: *Nonce N* $\notin$ *parts* (*initState B*)
⟨*proof*⟩

lemma *Nonce-notin-used-empty* [*simp*]: *Nonce N* $\notin$ *used* []
⟨*proof*⟩

3.8 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*: $\exists N.\ \forall n.\ N \leq n$ --> *Nonce n* $\notin$ *used evs*
⟨*proof*⟩

lemma *Nonce-supply1*: $\exists N$. *Nonce N* $\notin$ *used evs*
⟨*proof*⟩

lemma *Nonce-supply*: *Nonce* (*SOME N. Nonce N* $\notin$ *used evs*) $\notin$ *used evs*
⟨*proof*⟩

3.9 Specialized Rewriting for Theorems About *analz* and Image

lemma *insert-Key-singleton*: *insert* (*Key K*) H = *Key* ‘ $\{K\} \cup H$
⟨*proof*⟩

lemma *insert-Key-image*: *insert* (*Key K*) (*Key*‘*KK* $\cup$ *C*) = *Key* ‘ (*insert K KK*) $\cup$ *C*
⟨*proof*⟩

lemma *Crypt-imp-keysFor* :[|*Crypt K X* $\in H$; $K \in$ *symKeys*|] ==> $K \in$ *keysFor H*

⟨*proof*⟩

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

lemma *analz-image-freshK-lemma*:
  $(Key\ K \in analz\ (Key`nE \cup H))\ --{>}\ (K \in nE\ |\ Key\ K \in analz\ H)\ \ ==>$
    $(Key\ K \in analz\ (Key`nE \cup H)) = (K \in nE\ |\ Key\ K \in analz\ H)$
⟨*proof*⟩

lemmas *analz-image-freshK-simps* =
  simp-thms mem-simps — these two allow its use with *only*:
  disj-comms
  image-insert [*THEN sym*] *image-Un* [*THEN sym*] *empty-subsetI insert-subset*
  analz-insert-eq Un-upper2 [*THEN analz-mono*, *THEN subsetD*]
  insert-Key-singleton
  Key-not-used insert-Key-image Un-assoc [*THEN sym*]

⟨*ML*⟩

3.10 Specialized Methods for Possibility Theorems

⟨*ML*⟩

end

4 The Needham-Schroeder Public-Key Protocol against Dolev-Yao — with Gets event, hence with Reception rule

theory *NS-Public-Bad* **imports** *Public* **begin**

inductive-set *ns-public* :: *event list set*
 where

 Nil: $[] \in ns\text{-}public$

| *Fake*: $[\![evsf \in ns\text{-}public;\ \ X \in synth\ (analz\ (knows\ Spy\ evsf))]\!]$
  $\Longrightarrow Says\ Spy\ B\ X\ \ \#\ evsf \in ns\text{-}public$

| *Reception*: $[\![evsr \in ns\text{-}public;\ Says\ A\ \ B\ X \in set\ evsr]\!]$
   $\Longrightarrow Gets\ B\ X\ \#\ evsr \in ns\text{-}public$

| *NS1*: $[\![evs1 \in ns\text{-}public;\ \ Nonce\ NA \notin used\ evs1]\!]$
  $\Longrightarrow Says\ A\ \ B\ (Crypt\ (pubEK\ B)\ \{\!|Nonce\ NA,\ Agent\ A|\!\})$
   $\#\ evs1\ \in\ \ ns\text{-}public$

| $NS2$: $[\![evs2 \in ns\text{-}public;\ \ Nonce\ NB \notin used\ evs2;$
$Gets\ B\ (Crypt\ (pubEK\ B)\ \{\!|Nonce\ NA,\ Agent\ A|\!\})\ \in set\ evs2]\!]$
$\Longrightarrow Says\ B\ A\ (Crypt\ (pubEK\ A)\ \{\!|Nonce\ NA,\ Nonce\ NB|\!\})$
$\#\ evs2\ \in\ ns\text{-}public$

| $NS3$: $[\![evs3 \in ns\text{-}public;$
$Says\ A\ \ B\ (Crypt\ (pubEK\ B)\ \{\!|Nonce\ NA,\ Agent\ A|\!\})\ \in set\ evs3;$
$Gets\ A\ (Crypt\ (pubEK\ A)\ \{\!|Nonce\ NA,\ Nonce\ NB|\!\})\ \in set\ evs3]\!]$
$\Longrightarrow Says\ A\ B\ (Crypt\ (pubEK\ B)\ (Nonce\ NB))\ \#\ evs3 \in ns\text{-}public$

declare *knows-Spy-partsEs* [*elim*] **thm** *knows-Spy-partsEs*
declare *analz-into-parts* [*dest*]
declare *Fake-parts-insert-in-Un* [*dest*]

lemma $\exists NB.\ \exists evs \in ns\text{-}public.\ Says\ A\ B\ (Crypt\ (pubEK\ B)\ (Nonce\ NB)) \in set\ evs$
⟨*proof*⟩

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

lemma *Gets-imp-Says* :
$[\![\ Gets\ B\ X \in set\ evs;\ evs \in ns\text{-}public\ ]\!] \Longrightarrow \exists A.\ Says\ A\ B\ X \in set\ evs$
⟨*proof*⟩

lemma *Gets-imp-knows-Spy*:
$[\![\ Gets\ B\ X \in set\ evs;\ evs \in ns\text{-}public\ ]\!]\ \Longrightarrow X \in knows\ Spy\ evs$
⟨*proof*⟩

lemma *Gets-imp-knows-Spy-parts*[*dest*]:
$[\![\ Gets\ B\ X \in set\ evs;\ evs \in ns\text{-}public\ ]\!]\ \Longrightarrow X \in parts\ (knows\ Spy\ evs)$
⟨*proof*⟩

lemma *Spy-see-priEK* [*simp*]:
$evs \in ns\text{-}public \Longrightarrow (Key\ (priEK\ A) \in parts\ (knows\ Spy\ evs)) = (A \in bad)$
⟨*proof*⟩

lemma *Spy-analz-priEK* [*simp*]:
$evs \in ns\text{-}public \Longrightarrow (Key\ (priEK\ A) \in analz\ (knows\ Spy\ evs)) = (A \in bad)$
⟨*proof*⟩

lemma *no-nonce-NS1-NS2* [*rule-format*]:
$evs \in ns\text{-}public$
$\Longrightarrow Crypt\ (pubEK\ C)\ \{\!|NA', Nonce\ NA|\!\} \in parts\ (knows\ Spy\ evs) \longrightarrow$
$Crypt\ (pubEK\ B)\ \{\!|Nonce\ NA, Agent\ A|\!\} \in parts\ (knows\ Spy\ evs) \longrightarrow$
$Nonce\ NA \in analz\ (knows\ Spy\ evs)$
⟨*proof*⟩

lemma *unique-NA*:
$[\![Crypt(pubEK\ B)\ \ \{\!|Nonce\ NA, Agent\ A\ |\!\} \in parts(knows\ Spy\ evs);$
$Crypt(pubEK\ B')\ \{\!|Nonce\ NA, Agent\ A'|\!\} \in parts(knows\ Spy\ evs);$
$Nonce\ NA \notin analz\ (knows\ Spy\ evs);\ evs \in ns\text{-}public]\!]$
$\Longrightarrow A{=}A' \wedge B{=}B'$
⟨*proof*⟩

theorem *Spy-not-see-NA*:
$[\![Says\ A\ B\ (Crypt(pubEK\ B)\ \{\!|Nonce\ NA, Agent\ A|\!\}) \in set\ evs;$
$A \notin bad;\ \ B \notin bad;\ \ evs \in ns\text{-}public]\!]$
$\Longrightarrow Nonce\ NA \notin analz\ (knows\ Spy\ evs)$
⟨*proof*⟩

lemma *A-trusts-NS2-lemma* [*rule-format*]:
$[\![A \notin bad;\ \ B \notin bad;\ \ evs \in ns\text{-}public]\!]$
$\Longrightarrow Crypt\ (pubEK\ A)\ \{\!|Nonce\ NA, Nonce\ NB|\!\} \in parts\ (knows\ Spy\ evs) \longrightarrow$
$Says\ A\ B\ (Crypt(pubEK\ B)\ \{\!|Nonce\ NA, Agent\ A|\!\}) \in set\ evs \longrightarrow$
$Says\ B\ A\ (Crypt(pubEK\ A)\ \{\!|Nonce\ NA, Nonce\ NB|\!\}) \in set\ evs$
⟨*proof*⟩

theorem *A-trusts-NS2*:
$[\![Says\ A\ \ B\ (Crypt(pubEK\ B)\ \{\!|Nonce\ NA, Agent\ A|\!\}) \in set\ evs;$
$Gets\ A\ (Crypt(pubEK\ A)\ \{\!|Nonce\ NA, Nonce\ NB|\!\}) \in set\ evs;$
$A \notin bad;\ \ B \notin bad;\ \ evs \in ns\text{-}public]\!]$
$\Longrightarrow Says\ B\ A\ (Crypt(pubEK\ A)\ \{\!|Nonce\ NA, Nonce\ NB|\!\}) \in set\ evs$
⟨*proof*⟩

lemma *B-trusts-NS1* [*rule-format*]:
$evs \in ns\text{-}public$
$\Longrightarrow Crypt\ (pubEK\ B)\ \{\!|Nonce\ NA, Agent\ A|\!\} \in parts\ (knows\ Spy\ evs) \longrightarrow$

$Nonce\ NA \notin analz\ (knows\ Spy\ evs) \longrightarrow$
$Says\ A\ B\ (Crypt\ (pubEK\ B)\ \{|Nonce\ NA,\ Agent\ A|\}) \in set\ evs$
⟨*proof*⟩

lemma *unique-NB* [*dest*]:
$[\![Crypt(pubEK\ A)\ \ \{|Nonce\ NA,\ Nonce\ NB|\} \in parts(knows\ Spy\ evs);$
$Crypt(pubEK\ A')\ \{|Nonce\ NA',\ Nonce\ NB|\} \in parts(knows\ Spy\ evs);$
$Nonce\ NB \notin analz\ (knows\ Spy\ evs);\ evs \in ns\text{-}public]\!]$
$\Longrightarrow A{=}A' \wedge NA{=}NA'$
⟨*proof*⟩

theorem *Spy-not-see-NB* [*dest*]:
$[\![Says\ B\ A\ (Crypt\ (pubEK\ A)\ \{|Nonce\ NA,\ Nonce\ NB|\}) \in set\ evs;$
$\forall C.\ Says\ A\ C\ (Crypt\ (pubEK\ C)\ (Nonce\ NB)) \notin set\ evs;$
$A \notin bad;\ \ B \notin bad;\ \ evs \in ns\text{-}public]\!]$
$\Longrightarrow Nonce\ NB \notin analz\ (knows\ Spy\ evs)$
⟨*proof*⟩

lemma *B-trusts-NS3-lemma* [*rule-format*]:
$[\![A \notin bad;\ \ B \notin bad;\ \ evs \in ns\text{-}public]\!]$
$\Longrightarrow Crypt\ (pubEK\ B)\ (Nonce\ NB) \in parts\ (knows\ Spy\ evs) \longrightarrow$
$Says\ B\ A\ \ (Crypt\ (pubEK\ A)\ \{|Nonce\ NA,\ Nonce\ NB|\}) \in set\ evs \longrightarrow$
$(\exists C.\ Says\ A\ C\ (Crypt\ (pubEK\ C)\ (Nonce\ NB)) \in set\ evs)$
⟨*proof*⟩

theorem *B-trusts-NS3*:
$[\![Says\ B\ A\ \ (Crypt\ (pubEK\ A)\ \{|Nonce\ NA,\ Nonce\ NB|\}) \in set\ evs;$
$Gets\ B\ (Crypt\ (pubEK\ B)\ (Nonce\ NB)) \in set\ evs;$
$A \notin bad;\ \ B \notin bad;\ \ evs \in ns\text{-}public]\!]$
$\Longrightarrow \exists C.\ Says\ A\ C\ (Crypt\ (pubEK\ C)\ (Nonce\ NB)) \in set\ evs$
⟨*proof*⟩

lemma $[\![A \notin bad;\ \ B \notin bad;\ \ evs \in ns\text{-}public]\!]$
$\Longrightarrow Says\ B\ A\ (Crypt\ (pubEK\ A)\ \{|Nonce\ NA,\ Nonce\ NB|\}) \in set\ evs$
$\longrightarrow Nonce\ NB \notin analz\ (knows\ Spy\ evs)$
⟨*proof*⟩

end

5 Inductive Study of Confidentiality against Dolev-Yao

theory *ConfidentialityDY* **imports** *NS-Public-Bad* **begin**

6 Existing study - fully spelled out

In order not to leave hidden anything of the line of reasoning, we cancel some relevant lemmas that were installed previously

declare *Spy-see-priEK* [*simp del*]
Spy-analz-priEK [*simp del*]
analz-into-parts [*rule del*]

6.1 On static secrets

lemma *Spy-see-priEK*:
$evs \in ns\text{-}public \Longrightarrow (Key\ (priEK\ A) \in parts\ (spies\ evs)) = (A \in bad)$
⟨*proof*⟩

lemma *Spy-analz-priEK*:
$evs \in ns\text{-}public \Longrightarrow (Key\ (priEK\ A) \in analz\ (spies\ evs)) = (A \in bad)$

⟨*proof*⟩

6.2 On dynamic secrets

lemma *Spy-not-see-NA*:
$[\![Says\ A\ B\ (Crypt(pubEK\ B)\ \{\!|Nonce\ NA,\ Agent\ A|\!\})\in set\ evs;$
$A \notin bad;\ \ B \notin bad;\ \ evs \in ns\text{-}public]\!]$
$\Longrightarrow Nonce\ NA \notin analz\ (spies\ evs)$
⟨*proof*⟩

lemma *Spy-not-see-NB*:
$[\![Says\ B\ A\ (Crypt\ (pubEK\ A)\ \{\!|Nonce\ NA,\ Nonce\ NB|\!\}) \in set\ evs;$
$\forall C.\ Says\ A\ C\ (Crypt\ (pubEK\ C)\ (Nonce\ NB)) \notin set\ evs;$
$A \notin bad;\ \ B \notin bad;\ \ evs \in ns\text{-}public]\!]$
$\Longrightarrow Nonce\ NB \notin analz\ (spies\ evs)$
⟨*proof*⟩

7 Novel study

Generalising over all initial secrets the existing treatment, which is limited to private encryption keys

definition *staticSecret* :: *agent* ⇒ *msg set* **where**

[*simp*]: *staticSecret A* $\equiv$ {*Key* (*priEK A*), *Key* (*priSK A*), *Key* (*shrK A*)}

7.1 Protocol independent study

Converse doesn't hold because something that is said or noted is not necessarily an initial secret

lemma *staticSecret-parts-Spy*:
$[\![m \in parts\ (knows\ Spy\ evs);\ m \in staticSecret\ A]\!] \implies$
$A \in bad\ \vee$
$(\exists\ C\ B\ X.\ Says\ C\ B\ X \in set\ evs \wedge m \in parts\{X\})\ \vee$
$(\exists\ C\ Y.\ Notes\ C\ Y \in set\ evs \wedge C \in bad \wedge m \in parts\{Y\})$
⟨*proof*⟩

lemma *staticSecret-analz-Spy*:
$[\![m \in analz\ (knows\ Spy\ evs);\ m \in staticSecret\ A]\!] \implies$
$A \in bad\ \vee$
$(\exists\ C\ B\ X.\ Says\ C\ B\ X \in set\ evs \wedge m \in parts\{X\})\ \vee$
$(\exists\ C\ Y.\ Notes\ C\ Y \in set\ evs \wedge C \in bad \wedge m \in parts\{Y\})$
⟨*proof*⟩

lemma *secret-parts-Spy*:
$m \in parts\ (knows\ Spy\ evs) \implies$
$m \in initState\ Spy\ \vee$
$(\exists\ C\ B\ X.\ Says\ C\ B\ X \in set\ evs \wedge m \in parts\{X\})\ \vee$
$(\exists\ C\ Y.\ Notes\ C\ Y \in set\ evs \wedge C \in bad \wedge m \in parts\{Y\})$
⟨*proof*⟩

lemma *secret-parts-Spy-converse*:
$m \in initState\ Spy\ \vee$
$(\exists\ C\ B\ X.\ Says\ C\ B\ X \in set\ evs \wedge m \in parts\{X\})\ \vee$
$(\exists\ C\ Y.\ Notes\ C\ Y \in set\ evs \wedge C \in bad \wedge m \in parts\{Y\})$
$\implies m \in parts(knows\ Spy\ evs)$
⟨*proof*⟩

lemma *secret-analz-Spy*:
$m \in analz\ (knows\ Spy\ evs) \implies$
$m \in initState\ Spy\ \vee$
$(\exists\ C\ B\ X.\ Says\ C\ B\ X \in set\ evs \wedge m \in parts\{X\})\ \vee$
$(\exists\ C\ Y.\ Notes\ C\ Y \in set\ evs \wedge C \in bad \wedge m \in parts\{Y\})$
⟨*proof*⟩

7.2 Protocol-dependent study

Proving generalised version of $?evs \in ns\text{-}public \implies (Key\ (priEK\ ?A) \in parts\ (knows\ Spy\ ?evs)) = (?A \in bad)$ using same strategy, the "direct" strategy

lemma *NS-Spy-see-staticSecret*:

$[\![m \in \mathit{staticSecret}\ A;\ \mathit{evs} \in \mathit{ns\text{-}public}]\!] \Longrightarrow$
$m \in \mathit{parts}(\mathit{knows}\ \mathit{Spy}\ \mathit{evs}) = (A \in \mathit{bad})$
⟨*proof*⟩

Seeking a proof of $[\![?m \in \mathit{staticSecret}\ ?A;\ ?\mathit{evs} \in \mathit{ns\text{-}public}]\!] \Longrightarrow (?m \in \mathit{parts}\ (\mathit{knows}\ \mathit{Spy}\ ?\mathit{evs})) = (?A \in \mathit{bad})$ using an alternative, "specialisation" strategy

lemma *NS-no-Notes*:
$\mathit{evs} \in \mathit{ns\text{-}public} \Longrightarrow \mathit{Notes}\ A\ X \notin \mathit{set}\ \mathit{evs}$
⟨*proof*⟩

lemma *NS-staticSecret-parts-Spy-weak*:
$[\![m \in \mathit{parts}\ (\mathit{knows}\ \mathit{Spy}\ \mathit{evs});\ m \in \mathit{staticSecret}\ A;$
$\mathit{evs} \in \mathit{ns\text{-}public}]\!] \Longrightarrow A \in \mathit{bad} \vee$
$(\exists\, C\ B\ X.\ \mathit{Says}\ C\ B\ X \in \mathit{set}\ \mathit{evs} \wedge m \in \mathit{parts}\{X\})$
⟨*proof*⟩

lemma *NS-Says-staticSecret*:
$[\![\mathit{Says}\ A\ B\ X \in \mathit{set}\ \mathit{evs};\ m \in \mathit{staticSecret}\ C;\ m \in \mathit{parts}\{X\};$
$\mathit{evs} \in \mathit{ns\text{-}public}]\!] \Longrightarrow A{=}\mathit{Spy}$
⟨*proof*⟩

This generalises $(\mathit{Key}\ ?K \in \mathit{synth}\ ?H) = (\mathit{Key}\ ?K \in ?H)$

lemma *staticSecret-synth-eq*:
$m \in \mathit{staticSecret}\ A \Longrightarrow (m \in \mathit{synth}\ H) = (m \in H)$
⟨*proof*⟩

lemma *NS-Says-Spy-staticSecret*:
$[\![\mathit{Says}\ \mathit{Spy}\ B\ X \in \mathit{set}\ \mathit{evs};\ m \in \mathit{parts}\{X\};$
$m \in \mathit{staticSecret}\ A;\ \mathit{evs} \in \mathit{ns\text{-}public}]\!] \Longrightarrow A \in \mathit{bad}$

⟨*proof*⟩

Here's the specialised version of $[\![?m \in \mathit{parts}\ (\mathit{knows}\ \mathit{Spy}\ ?\mathit{evs});\ ?m \in \mathit{staticSecret}\ ?A]\!] \Longrightarrow ?A \in \mathit{bad} \vee (\exists\, C\ B\ X.\ \mathit{Says}\ C\ B\ X \in \mathit{set}\ ?\mathit{evs} \wedge ?m \in \mathit{parts}\ \{X\}) \vee (\exists\, C\ Y.\ \mathit{Notes}\ C\ Y \in \mathit{set}\ ?\mathit{evs} \wedge C \in \mathit{bad} \wedge ?m \in \mathit{parts}\ \{Y\})$

lemma *NS-staticSecret-parts-Spy*:
$[\![m \in \mathit{parts}\ (\mathit{knows}\ \mathit{Spy}\ \mathit{evs});\ m \in \mathit{staticSecret}\ A;$
$\mathit{evs} \in \mathit{ns\text{-}public}]\!] \Longrightarrow A \in \mathit{bad}$
⟨*proof*⟩

Concluding the specialisation proof strategy...

lemma *NS-Spy-see-staticSecret-spec*:
$[\![m \in \mathit{staticSecret}\ A;\ \mathit{evs} \in \mathit{ns\text{-}public}]\!] \Longrightarrow$
$m \in \mathit{parts}\ (\mathit{knows}\ \mathit{Spy}\ \mathit{evs}) = (A \in \mathit{bad})$

one line proof: apply (force dest: *NS-staticSecret-parts-Spy*)

⟨*proof*⟩

lemma *NS-Spy-analz-staticSecret*:
$[\![m \in \mathit{staticSecret}\ A;\ \mathit{evs} \in \mathit{ns\text{-}public}]\!] \Longrightarrow$
$m \in \mathit{analz}\ (\mathit{knows}\ \mathit{Spy}\ \mathit{evs}) = (A \in \mathit{bad})$
⟨*proof*⟩

lemma *NS-staticSecret-subset-parts-knows-Spy*:
$\mathit{evs} \in \mathit{ns\text{-}public} \Longrightarrow$
$\mathit{staticSecret}\ A \subseteq \mathit{parts}\ (\mathit{knows}\ \mathit{Spy}\ \mathit{evs}) = (A \in \mathit{bad})$
⟨*proof*⟩

lemma *NS-staticSecret-subset-analz-knows-Spy*:
$\mathit{evs} \in \mathit{ns\text{-}public} \Longrightarrow$
$\mathit{staticSecret}\ A \subseteq \mathit{analz}\ (\mathit{knows}\ \mathit{Spy}\ \mathit{evs}) = (A \in \mathit{bad})$
⟨*proof*⟩

end

8 Theory of Agents and Messages for Security Protocols against the General Attacker

theory *MessageGA* **imports** *Main* **begin**

lemma [*simp*] : $A \cup (B \cup A) = B \cup A$
⟨*proof*⟩

type-synonym
key = *nat*

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

specification (*invKey*)
invKey [*simp*]: $\mathit{invKey}\ (\mathit{invKey}\ K) = K$
invKey-symmetric: $\mathit{all\text{-}symmetric} \longrightarrow \mathit{invKey} = \mathit{id}$
⟨*proof*⟩

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

definition *symKeys* :: *key set* **where**
$\mathit{symKeys} == \{K.\ \mathit{invKey}\ K = K\}$

datatype — We only allow for any number of friendly agents
agent = *Friend nat*

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
$-MTuple$ :: $['a,\ args] => 'a * 'b$ $((2\{|-,/\ -|\}))$
translations
$\{|x,\ y,\ z|\}$ == $\{|x,\ \{|y,\ z|\}|\}$
$\{|x,\ y|\}$ == $CONST\ MPair\ x\ y$

definition $HPair$ :: $[msg,msg] => msg$ $((4Hash[-]\ /-)\ [0,\ 1000])$ **where**
— Message Y paired with a MAC computed with the help of X
$Hash[X]\ \ Y == \{|\ Hash\{|X,Y|\},\ Y|\}$

definition $keysFor$ :: $msg\ set => key\ set$ **where**
— Keys useful to decrypt elements of a message set
$keysFor\ H == invKey\ `\ \{K.\ \exists X.\ Crypt\ K\ X \in H\}$

8.1 Inductive definition of all parts of a message

inductive-set
$parts$ :: $msg\ set => msg\ set$
for H :: $msg\ set$
where
$Inj\ [intro]$: $X \in H \Longrightarrow X \in parts\ H$
| Fst: $\{|X,Y|\} \in parts\ H \Longrightarrow X \in parts\ H$
| Snd: $\{|X,Y|\} \in parts\ H \Longrightarrow Y \in parts\ H$
| $Body$: $Crypt\ K\ X \in parts\ H \Longrightarrow X \in parts\ H$

Monotonicity

lemma $parts\text{-}mono$: $G \subseteq H \Longrightarrow parts(G) \subseteq parts(H)$
⟨*proof*⟩

Equations hold because constructors are injective.

lemma $Friend\text{-}image\text{-}eq$ $[simp]$: $(Friend\ x \in Friend`A) = (x{:}A)$
⟨*proof*⟩

lemma $Key\text{-}image\text{-}eq$ $[simp]$: $(Key\ x \in Key`A) = (x{\in}A)$
⟨*proof*⟩

lemma $Nonce\text{-}Key\text{-}image\text{-}eq$ $[simp]$: $(Nonce\ x \notin Key`A)$

⟨*proof*⟩

8.2 Inverse of keys

lemma *invKey-eq* [*simp*]: (*invKey K* = *invKey K′*) = (*K*=*K′*)
⟨*proof*⟩

8.3 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*⟩

8.4 Inductive relation "parts"

lemma *MPair-parts*:
  $[| \{|X,Y|\} \in parts\ H;$
    $[| X \in parts\ H;\ Y \in parts\ H\ |] ==> P\ |] ==> P$
⟨*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*: $H \subseteq parts(H)$
⟨*proof*⟩

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

lemma *parts-empty* [*simp*]: $parts\{\} = \{\}$
⟨*proof*⟩

lemma *parts-emptyE* [*elim*!]: $X \in parts\{\} \Longrightarrow P$
⟨*proof*⟩

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

lemma *parts-singleton*: $X \in parts\ H \Longrightarrow \exists\ Y \in H.\ X \in parts\ \{Y\}$
⟨*proof*⟩

8.4.1 Unions

lemma *parts-Un-subset1*: $parts(G) \cup parts(H) \subseteq parts(G \cup H)$
⟨*proof*⟩

lemma *parts-Un-subset2*: $parts(G \cup H) \subseteq parts(G) \cup parts(H)$
⟨*proof*⟩

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

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

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

lemma *parts-insert2*:
  $parts\ (insert\ X\ (insert\ Y\ H)) = parts\ \{X\} \cup parts\ \{Y\} \cup parts\ H$
⟨*proof*⟩

lemma *parts-UN-subset1*: $(\bigcup x \in A.\ parts(H\ x)) \subseteq parts(\bigcup x \in A.\ H\ x)$

⟨*proof*⟩

lemma *parts-UN-subset2*: $parts(\bigcup x{\in}A.\ H\ x) \subseteq (\bigcup x{\in}A.\ parts(H\ x))$
⟨*proof*⟩

lemma *parts-UN* [*simp*]: $parts(\bigcup x{\in}A.\ H\ x) = (\bigcup x{\in}A.\ parts(H\ x))$
⟨*proof*⟩

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

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

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

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

8.4.2 Idempotence and transitivity

lemma *parts-partsD* [*dest*!]: $X \in parts\ (parts\ H) \Longrightarrow X{\in}\ parts\ H$
⟨*proof*⟩

lemma *parts-idem* [*simp*]: $parts\ (parts\ H) = parts\ H$
⟨*proof*⟩

lemma *parts-subset-iff* [*simp*]: $(parts\ G \subseteq parts\ H) = (G \subseteq parts\ H)$
⟨*proof*⟩

lemma *parts-trans*: $[|\ X \in parts\ G;\ \ G \subseteq parts\ H\ |] ==> X{\in}\ parts\ H$
⟨*proof*⟩

Cut

lemma *parts-cut*:
  $[|\ Y{\in}\ parts\ (insert\ X\ G);\ \ X \in parts\ H\ |] ==> Y{\in}\ parts\ (G \cup H)$
⟨*proof*⟩

lemma *parts-cut-eq* [*simp*]: $X \in parts\ H \Longrightarrow parts\ (insert\ X\ H) = parts\ H$
⟨*proof*⟩

8.4.3 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*: $\exists N.\ \forall n.\ N \le n \longrightarrow Nonce\ n \notin parts\ \{msg\}$
⟨*proof*⟩

8.5 Inductive relation "analz"

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

inductive-set
analz :: *msg set* => *msg set*
for *H* :: *msg set*
where
Inj [*intro*,*simp*] : $X \in H \Longrightarrow X \in analz\ H$
| *Fst*: ⦃*X*, *Y*⦄ ∈ *analz H* ⟹ *X* ∈ *analz H*
| *Snd*: ⦃*X*, *Y*⦄ ∈ *analz H* ⟹ *Y* ∈ *analz H*
| *Decrypt* [*dest*]:

$[| Crypt\ K\ X \in analz\ H;\ Key(invKey\ K): analz\ H |] ==> X \in analz\ H$

Monotonicity; Lemma 1 of Lowe's paper

lemma *analz-mono*: $G \subseteq H \Longrightarrow analz(G) \subseteq analz(H)$
⟨*proof*⟩

Making it safe speeds up proofs

lemma *MPair-analz* [*elim!*]:
$[| \{|X, Y|\} \in analz\ H;$
$[| X \in analz\ H;\ Y \in analz\ H |] ==> P$
$|] ==> P$
⟨*proof*⟩

lemma *analz-increasing*: $H \subseteq analz(H)$
⟨*proof*⟩

lemma *analz-subset-parts*: $analz\ H \subseteq parts\ H$
⟨*proof*⟩

lemmas *analz-into-parts* = *analz-subset-parts* [*THEN subsetD*]

lemmas *not-parts-not-analz* = *analz-subset-parts* [*THEN contra-subsetD*]

lemma *parts-analz* [*simp*]: $parts\ (analz\ H) = parts\ H$
⟨*proof*⟩

lemma *analz-parts* [*simp*]: $analz\ (parts\ H) = parts\ H$
⟨*proof*⟩

lemmas *analz-insertI* = *subset-insertI* [*THEN analz-mono*, *THEN* [*2*] *rev-subsetD*]

8.5.1 General equational properties

lemma *analz-empty* [*simp*]: $analz\{\} = \{\}$
⟨*proof*⟩

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

lemma *analz-Un*: $analz(G) \cup analz(H) \subseteq analz(G \cup H)$
⟨*proof*⟩

lemma *analz-insert*: $insert\ X\ (analz\ H) \subseteq analz(insert\ X\ H)$
⟨*proof*⟩

8.5.2 Rewrite rules for pulling out atomic messages

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

lemma *analz-insert-Agent* [*simp*]:
  $analz\ (insert\ (Agent\ agt)\ H) = insert\ (Agent\ agt)\ (analz\ H)$
⟨*proof*⟩

lemma *analz-insert-Nonce* [*simp*]:
  $analz\ (insert\ (Nonce\ N)\ H) = insert\ (Nonce\ N)\ (analz\ H)$
⟨*proof*⟩

lemma *analz-insert-Number* [*simp*]:
  $analz\ (insert\ (Number\ N)\ H) = insert\ (Number\ N)\ (analz\ H)$
⟨*proof*⟩

lemma *analz-insert-Hash* [*simp*]:
  $analz\ (insert\ (Hash\ X)\ H) = insert\ (Hash\ X)\ (analz\ H)$
⟨*proof*⟩

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

lemma *analz-insert-Key* [*simp*]:
  $K \notin keysFor\ (analz\ H) \Longrightarrow$
    $analz\ (insert\ (Key\ K)\ H) = insert\ (Key\ K)\ (analz\ H)$
⟨*proof*⟩

lemma *analz-insert-MPair* [*simp*]:
  $analz\ (insert\ \{\!|X, Y|\!\}\ H) =$
    $insert\ \{\!|X, Y|\!\}\ (analz\ (insert\ X\ (insert\ Y\ H)))$
⟨*proof*⟩

Can pull out enCrypted message if the Key is not known

lemma *analz-insert-Crypt*:
  $Key\ (invKey\ K) \notin analz\ H$
  $\Longrightarrow analz\ (insert\ (Crypt\ K\ X)\ H) = insert\ (Crypt\ K\ X)\ (analz\ H)$
⟨*proof*⟩

lemma *lemma1*: $Key\ (invKey\ K) \in analz\ H \Longrightarrow$
    $analz\ (insert\ (Crypt\ K\ X)\ H) \subseteq$
    $insert\ (Crypt\ K\ X)\ (analz\ (insert\ X\ H))$
⟨*proof*⟩

lemma *lemma2*: $Key\ (invKey\ K) \in analz\ H \Longrightarrow$
    $insert\ (Crypt\ K\ X)\ (analz\ (insert\ X\ H)) \subseteq$
    $analz\ (insert\ (Crypt\ K\ X)\ H)$
⟨*proof*⟩

lemma *analz-insert-Decrypt*:
  $Key\ (invKey\ K) \in analz\ H \Longrightarrow$
    $analz\ (insert\ (Crypt\ K\ X)\ H) =$
    $insert\ (Crypt\ K\ X)\ (analz\ (insert\ X\ H))$
⟨*proof*⟩

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

lemma *analz-Crypt-if* [*simp*]:
  analz (*insert* (*Crypt K X*) *H*) =
    (*if* (*Key* (*invKey K*) $\in$ *analz H*)
     then insert (*Crypt K X*) (*analz* (*insert X H*))
     else insert (*Crypt K X*) (*analz H*))
⟨*proof*⟩

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

lemma *analz-insert-Crypt-subset*:
  analz (*insert* (*Crypt K X*) *H*) $\subseteq$
     insert (*Crypt K X*) (*analz* (*insert X H*))
⟨*proof*⟩

lemma *analz-image-Key* [*simp*]: *analz* (*Key'N*) = *Key'N*
⟨*proof*⟩

8.5.3 Idempotence and transitivity

lemma *analz-analzD* [*dest*!]: *X*$\in$ *analz* (*analz H*) $\Longrightarrow$ *X* $\in$ *analz H*
⟨*proof*⟩

lemma *analz-idem* [*simp*]: *analz* (*analz H*) = *analz H*
⟨*proof*⟩

lemma *analz-subset-iff* [*simp*]: (*analz G* $\subseteq$ *analz H*) = (*G* $\subseteq$ *analz H*)
⟨*proof*⟩

lemma *analz-trans*: [| *X*$\in$ *analz G*; *G* $\subseteq$ *analz H* |] ==> *X*$\in$ *analz H*
⟨*proof*⟩

Cut; Lemma 2 of Lowe

lemma *analz-cut*: [| *Y*$\in$ *analz* (*insert X H*); *X*$\in$ *analz H* |] ==> *Y*$\in$ *analz H*
⟨*proof*⟩

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

lemma *analz-insert-eq*: *X* $\in$ *analz H* $\Longrightarrow$ *analz* (*insert X H*) = *analz H*
⟨*proof*⟩

A congruence rule for "analz"

lemma *analz-subset-cong*:
  [| *analz G* $\subseteq$ *analz G'*; *analz H* $\subseteq$ *analz H'* |]
  ==> *analz* ($G \cup H$) $\subseteq$ *analz* ($G' \cup H'$)
⟨*proof*⟩

lemma *analz-cong*:
$[| \; analz\ G = analz\ G'; \; analz\ H = analz\ H' \; |]$
$==> analz\ (G \cup H) = analz\ (G' \cup H')$
⟨*proof*⟩

lemma *analz-insert-cong*:
$analz\ H = analz\ H' \Longrightarrow analz(insert\ X\ H) = analz(insert\ X\ H')$
⟨*proof*⟩

If there are no pairs or encryptions then analz does nothing

lemma *analz-trivial*:
$[| \; \forall X\ Y.\ \{|X,Y|\} \notin H; \;\; \forall X\ K.\ Crypt\ K\ X \notin H \; |] ==> analz\ H = H$
⟨*proof*⟩

These two are obsolete but cost little to prove...

lemma *analz-UN-analz-lemma*:
$X \in analz\ (\bigcup i \in A.\ analz\ (H\ i)) \Longrightarrow X \in analz\ (\bigcup i \in A.\ H\ i)$
⟨*proof*⟩

lemma *analz-UN-analz* [*simp*]: $analz\ (\bigcup i \in A.\ analz\ (H\ i)) = analz\ (\bigcup i \in A.\ H\ i)$
⟨*proof*⟩

8.6 Inductive relation "synth"

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

inductive-set
synth :: *msg set* => *msg set*
for *H* :: *msg set*
where
Inj [*intro*]: $X \in H \Longrightarrow X \in synth\ H$
| *Agent* [*intro*]: $Agent\ agt \in synth\ H$
| *Number* [*intro*]: $Number\ n \in synth\ H$
| *Hash* [*intro*]: $X \in synth\ H \Longrightarrow Hash\ X \in synth\ H$
| *MPair* [*intro*]: $[|X \in synth\ H; \;\; Y \in synth\ H|] ==> \{|X,Y|\} \in synth\ H$
| *Crypt* [*intro*]: $[|X \in synth\ H; \;\; Key(K) \in H|] ==> Crypt\ K\ X \in synth\ H$

Monotonicity

lemma *synth-mono*: $G \subseteq H \Longrightarrow synth(G) \subseteq synth(H)$
⟨*proof*⟩

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

inductive-simps *synth-simps* [*iff*]:

Nonce $n \in$ *synth* H
Key $K \in$ *synth* H
Hash $X \in$ *synth* H
$\{\!|X, Y|\!\} \in$ *synth* H
Crypt K $X \in$ *synth* H

lemma *synth-increasing*: $H \subseteq$ *synth*(H)
⟨*proof*⟩

8.6.1 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*$(G) \cup$ *synth*$(H) \subseteq$ *synth*$(G \cup H)$
⟨*proof*⟩

lemma *synth-insert*: *insert* X (*synth* H) $\subseteq$ *synth*(*insert* X H)
⟨*proof*⟩

8.6.2 Idempotence and transitivity

lemma *synth-synthD* [*dest*!]: $X \in$ *synth* (*synth* H) $\Longrightarrow X \in$ *synth* H
⟨*proof*⟩

lemma *synth-idem*: *synth* (*synth* H) = *synth* H
⟨*proof*⟩

lemma *synth-subset-iff* [*simp*]: (*synth* $G \subseteq$ *synth* H) = ($G \subseteq$ *synth* H)
⟨*proof*⟩

lemma *synth-trans*: [| $X \in$ *synth* G; $G \subseteq$ *synth* H |] ==> $X \in$ *synth* H
⟨*proof*⟩

Cut; Lemma 2 of Lowe

lemma *synth-cut*: [| $Y \in$ *synth* (*insert* X H); $X \in$ *synth* H |] ==> $Y \in$ *synth* H
⟨*proof*⟩

lemma *Agent-synth* [*simp*]: *Agent* $A \in$ *synth* H
⟨*proof*⟩

lemma *Number-synth* [*simp*]: *Number* $n \in$ *synth* H
⟨*proof*⟩

lemma *Nonce-synth-eq* [*simp*]: (*Nonce* $N \in$ *synth* H) = (*Nonce* $N \in H$)
⟨*proof*⟩

lemma *Key-synth-eq* [*simp*]: (*Key* $K \in$ *synth* H) = (*Key* $K \in H$)
⟨*proof*⟩

lemma *Crypt-synth-eq* [*simp*]:
$Key\ K \notin H \Longrightarrow (Crypt\ K\ X \in synth\ H) = (Crypt\ K\ X \in H)$
⟨*proof*⟩

lemma *keysFor-synth* [*simp*]:
$keysFor\ (synth\ H) = keysFor\ H \cup invKey`\{K.\ Key\ K \in H\}$
⟨*proof*⟩

8.6.3 Combinations of parts, analz and synth

lemma *parts-synth* [*simp*]: $parts\ (synth\ H) = parts\ H \cup synth\ H$
⟨*proof*⟩

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

lemma *analz-synth-Un* [*simp*]: $analz\ (synth\ G \cup H) = analz\ (G \cup H) \cup synth\ G$
⟨*proof*⟩

lemma *analz-synth* [*simp*]: $analz\ (synth\ H) = analz\ H \cup synth\ H$
⟨*proof*⟩

8.6.4 For reasoning about the Fake rule in traces

lemma *parts-insert-subset-Un*: $X \in G \Longrightarrow parts(insert\ X\ H) \subseteq parts\ G \cup parts\ H$
⟨*proof*⟩

More specifically for Fake. See also *Fake-parts-sing* below

lemma *Fake-parts-insert*:
$X \in synth\ (analz\ H) \Longrightarrow$
$parts\ (insert\ X\ H) \subseteq synth\ (analz\ H) \cup parts\ H$
⟨*proof*⟩

lemma *Fake-parts-insert-in-Un*:
$[|Z \in parts\ (insert\ X\ H);\ \ X{:}\ synth\ (analz\ H)|]$
$==> Z \in\ synth\ (analz\ H) \cup parts\ H$
⟨*proof*⟩

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

lemma *Fake-analz-insert*:
$X \in synth\ (analz\ G) \Longrightarrow$
$analz\ (insert\ X\ H) \subseteq synth\ (analz\ G) \cup analz\ (G \cup H)$
⟨*proof*⟩

lemma *analz-conj-parts* [*simp*]:
$(X \in analz\ H \wedge X \in parts\ H) = (X \in analz\ H)$
⟨*proof*⟩

lemma *analz-disj-parts* [*simp*]:
$(X \in analz\ H \mid X \in parts\ H) = (X \in parts\ H)$
⟨*proof*⟩

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

lemma *MPair-synth-analz* [*iff*]:
$(\{|X, Y|\} \in synth\ (analz\ H)) =$
$(X \in synth\ (analz\ H) \wedge Y \in synth\ (analz\ H))$
⟨*proof*⟩

lemma *Crypt-synth-analz*:
$[|\ Key\ K \in analz\ H;\ \ Key\ (invKey\ K) \in analz\ H\ |]$
$==> (Crypt\ K\ X \in synth\ (analz\ H)) = (X \in synth\ (analz\ H))$
⟨*proof*⟩

lemma *Hash-synth-analz* [*simp*]:
$X \notin synth\ (analz\ H)$
$\Longrightarrow (Hash\{|X, Y|\} \in synth\ (analz\ H)) = (Hash\{|X, Y|\} \in analz\ H)$
⟨*proof*⟩

8.7 HPair: a combination of Hash and MPair

8.7.1 Freeness

lemma *Agent-neq-HPair*: $Agent\ A\ \tilde{}= Hash[X]\ \ Y$
⟨*proof*⟩

lemma *Nonce-neq-HPair*: $Nonce\ N\ \tilde{}= Hash[X]\ \ Y$
⟨*proof*⟩

lemma *Number-neq-HPair*: $Number\ N\ \tilde{}= Hash[X]\ \ Y$
⟨*proof*⟩

lemma *Key-neq-HPair*: $Key\ K\ \tilde{}= Hash[X]\ \ Y$
⟨*proof*⟩

lemma *Hash-neq-HPair*: $Hash\ Z\ \tilde{}= Hash[X]\ \ Y$
⟨*proof*⟩

lemma *Crypt-neq-HPair*: $Crypt\ K\ X'\ \tilde{}= Hash[X]\ \ Y$
⟨*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*]: $(Hash[X']\ Y' = Hash[X]\ Y) = (X' = X \wedge Y'{=}Y)$
⟨*proof*⟩

lemma *MPair-eq-HPair* [*iff*]:
$(\{|X',Y'|\} = Hash[X]\ Y) = (X' = Hash\{|X,Y|\} \wedge Y'{=}Y)$
⟨*proof*⟩

lemma *HPair-eq-MPair* [*iff*]:
$(Hash[X]\ Y = \{|X',Y'|\}) = (X' = Hash\{|X,Y|\} \wedge Y'{=}Y)$
⟨*proof*⟩

8.7.2 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*]:
$X \notin synth\ (analz\ H)$
$\Longrightarrow (Hash[X]\ Y \in synth\ (analz\ H)) =$
$(Hash\ \{|X,\ Y|\} \in analz\ H \wedge Y \in synth\ (analz\ H))$
⟨*proof*⟩

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

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

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

lemmas *pushKeys* =
insert-commute [*of Key K Agent C*]
insert-commute [*of Key K Nonce N*]
insert-commute [*of Key K Number N*]
insert-commute [*of Key K Hash X*]
insert-commute [*of Key K MPair X Y*]
insert-commute [*of Key K Crypt X K'*]
for *K C N X Y K'*

lemmas *pushCrypts* =
  insert-commute [*of Crypt X K Agent C*]
  insert-commute [*of Crypt X K Agent C*]
  insert-commute [*of Crypt X K Nonce N*]
  insert-commute [*of Crypt X K Number N*]
  insert-commute [*of Crypt X K Hash X'*]
  insert-commute [*of Crypt X K MPair X' Y*]
  for *X K C N X' Y*

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

lemmas *pushes* = *pushKeys pushCrypts*

8.8 The set of key-free messages

inductive-set
  keyfree :: *msg set*
  where
    Agent: *Agent A* ∈ *keyfree*
  | *Number*: *Number N* ∈ *keyfree*
  | *Nonce*: *Nonce N* ∈ *keyfree*
  | *Hash*:   *Hash X* ∈ *keyfree*
  | *MPair*: [|*X* ∈ *keyfree*; *Y* ∈ *keyfree*|] ==> ⦃*X*, *Y*⦄ ∈ *keyfree*
  | *Crypt*: [|*X* ∈ *keyfree*|] ==> *Crypt K X* ∈ *keyfree*

declare *keyfree.intros* [*intro*]

inductive-cases *keyfree-KeyE*: *Key K* ∈ *keyfree*
inductive-cases *keyfree-MPairE*: ⦃*X*, *Y*⦄ ∈ *keyfree*
inductive-cases *keyfree-CryptE*: *Crypt K X* ∈ *keyfree*

lemma *parts-keyfree*: *parts* (*keyfree*) ⊆ *keyfree*
  ⟨*proof*⟩

lemma *analz-keyfree-into-Un*: ⟦*X* ∈ *analz* (*G* ∪ *H*); *G* ⊆ *keyfree*⟧ ⟹ *X* ∈ *parts* *G* ∪ *analz H*
⟨*proof*⟩

8.9 Tactics useful for many protocol proofs

⟨*ML*⟩

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

declare *o-def* [*simp*]

lemma *Crypt-notin-image-Key* [*simp*]: *Crypt K X* $\notin$ *Key* ‘ *A*
⟨*proof*⟩

lemma *Hash-notin-image-Key* [*simp*] :*Hash X* $\notin$ *Key* ‘ *A*
⟨*proof*⟩

lemma *synth-analz-mono*: $G \subseteq H \Longrightarrow$ *synth* (*analz*(*G*)) $\subseteq$ *synth* (*analz*(*H*))
⟨*proof*⟩

lemma *Fake-analz-eq* [*simp*]:
  $X \in$ *synth*(*analz H*) $\Longrightarrow$ *synth* (*analz* (*insert X H*)) = *synth* (*analz H*)
⟨*proof*⟩

Two generalizations of *analz-insert-eq*

lemma *gen-analz-insert-eq* [*rule-format*]:
  $X \in$ *analz H* $\Longrightarrow \forall$ *G*. $H \subseteq G \longrightarrow$ *analz* (*insert X G*) = *analz G*
⟨*proof*⟩

lemma *synth-analz-insert-eq* [*rule-format*]:
  $X \in$ *synth* (*analz H*)
  $\Longrightarrow \forall$ *G*. $H \subseteq G \longrightarrow$ (*Key K* $\in$ *analz* (*insert X G*)) = (*Key K* $\in$ *analz G*)
⟨*proof*⟩

lemma *Fake-parts-sing*:
  $X \in$ *synth* (*analz H*) $\Longrightarrow$ *parts*{*X*} $\subseteq$ *synth* (*analz H*) $\cup$ *parts H*
⟨*proof*⟩

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

⟨*ML*⟩

end

9 Theory of Events for Security Protocols against the General Attacker

theory *EventGA* **imports** *MessageGA* **begin**

consts
  initState :: *agent* => *msg set*

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

primrec *knows* :: *agent* => *event list* => *msg set* **where**
  knows-Nil: *knows A* [] = *initState A*

| *knows-Cons*:
  knows A (ev # evs) =
    (case ev of
      Says A′ B X ⇒ insert X (knows A evs)
    | Gets A′ X ⇒ knows A evs
    | Notes A′ X ⇒
      if A′=A then insert X (knows A evs) else knows A evs)

primrec

used :: *event list => msg set* **where**
  used-Nil: *used* [] = (*UN B. parts* (*initState B*))
| *used-Cons*: *used* (*ev # evs*) =
    (*case ev of*
      Says A B X => parts {*X*} ∪ *used evs*
    | *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*⟩

9.1 Function *knows*

lemmas *parts-insert-knows-A* = *parts-insert* [*of* - *knows A evs*] **for** *A evs*

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

lemma *knows-Notes* [*simp*]:
  knows A (*Notes A′ X # evs*) =
    (*if A=A′ then insert X* (*knows A evs*) *else knows A evs*)
⟨*proof*⟩

lemma *knows-Gets* [*simp*]: *knows A* (*Gets A′ X # evs*) = *knows A evs*
⟨*proof*⟩

Everybody sees what is sent on the traffic

lemma *Says-imp-knows* [*rule-format*]:
  Says A′ B X ∈ *set evs* ⟶ (∀ *A*. *X* ∈ *knows A evs*)
⟨*proof*⟩

lemma *Notes-imp-knows* [*rule-format*]:
Notes A′ X ∈ *set evs* ⟶ *X* ∈ *knows A′ evs*

⟨*proof*⟩

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

lemmas *Says-imp-parts-knows* =
Says-imp-knows [*THEN parts.Inj*, *THEN revcut-rl*]

lemmas *knows-partsEs* =
Says-imp-parts-knows parts.Body [*THEN revcut-rl*]

lemmas *Says-imp-analz* = *Says-imp-knows* [*THEN analz.Inj*]

9.2 Knowledge of generic agents

lemma *knows-subset-knows-Says*: $knows\ A\ evs \subseteq knows\ A\ (Says\ A'\ B\ X\ \#\ evs)$
⟨*proof*⟩

lemma *knows-subset-knows-Notes*: $knows\ A\ evs \subseteq knows\ A\ (Notes\ A'\ X\ \#\ evs)$
⟨*proof*⟩

lemma *knows-subset-knows-Gets*: $knows\ A\ evs \subseteq knows\ A\ (Gets\ A'\ X\ \#\ evs)$
⟨*proof*⟩

lemma *knows-imp-Says-Gets-Notes-initState* [*rule-format*]:
$X \in knows\ A\ evs \Longrightarrow \exists A'\ B.$
$Says\ A'\ B\ X \in set\ evs \lor Notes\ A\ X \in set\ evs \lor X \in initState\ A$
⟨*proof*⟩

lemma *parts-knows-subset-used*: $parts\ (knows\ A\ evs) \subseteq used\ evs$
⟨*proof*⟩

lemmas *usedI* = *parts-knows-subset-used* [*THEN subsetD*, *intro*]

lemma *initState-into-used*: $X \in parts\ (initState\ B) \Longrightarrow X \in used\ evs$
⟨*proof*⟩

lemma *used-Says* [*simp*]: $used\ (Says\ A\ B\ X\ \#\ evs) = parts\{X\} \cup used\ evs$
⟨*proof*⟩

lemma *used-Notes* [*simp*]: $used\ (Notes\ A\ X\ \#\ evs) = parts\{X\} \cup used\ evs$
⟨*proof*⟩

lemma *used-Gets* [*simp*]: $used\ (Gets\ A\ X\ \#\ evs) = used\ evs$
⟨*proof*⟩

lemma *used-nil-subset*: $used\ [] \subseteq used\ evs$
⟨*proof*⟩

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

declare *knows-Cons* [*simp del*]
used-Nil [*simp del*] *used-Cons* [*simp del*]

lemmas *analz-mono-contra* =
knows-subset-knows-Says [*THEN analz-mono*, *THEN contra-subsetD*]
knows-subset-knows-Notes [*THEN analz-mono*, *THEN contra-subsetD*]
knows-subset-knows-Gets [*THEN analz-mono*, *THEN contra-subsetD*]

lemma *knows-subset-knows-Cons*: *knows A evs* ⊆ *knows A* (*e* # *evs*)
⟨*proof*⟩

lemma *initState-subset-knows*: *initState A* ⊆ *knows A evs*
⟨*proof*⟩

For proving *new-keys-not-used*

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

lemmas *analz-impI* = *impI* [**where** *P* = *Y* ∉ *analz* (*knows A evs*)] **for** *Y A evs*

⟨*ML*⟩

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

lemmas *syan-impI* = *impI* [**where** *P* = *Y* ∉ *synth* (*analz* (*knows A evs*))] **for** *Y A evs*

⟨*ML*⟩

end

10 Theory of Cryptographic Keys for Security Protocols against the General Attacker

theory *PublicGA* **imports** *EventGA* **begin**

lemma *invKey-K*: *K* ∈ *symKeys* ⟹ *invKey K* = *K*
⟨*proof*⟩

10.1 Asymmetric Keys

datatype *keymode* = *Signature* | *Encryption*

consts

publicKey :: [*keymode*,*agent*] => *key*

abbreviation
pubEK :: *agent* => *key* **where**
pubEK == *publicKey Encryption*

abbreviation
pubSK :: *agent* => *key* **where**
pubSK == *publicKey Signature*

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

abbreviation

priEK :: *agent* => *key* **where**
priEK A == *privateKey Encryption A*

abbreviation
priSK :: *agent* => *key* **where**
priSK A == *privateKey Signature A*

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

abbreviation
pubK :: *agent* => *key* **where**
pubK A == *pubEK A*

abbreviation
priK :: *agent* => *key* **where**
priK A == *invKey* (*pubEK A*)

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

specification (*publicKey*)
injective-publicKey:
publicKey b A = *publicKey c A'* $\Longrightarrow$ *b*=*c* $\wedge$ *A*=*A'*
$\langle proof \rangle$

axiomatization where

privateKey-neq-publicKey [*iff*]: *privateKey b A* $\neq$ *publicKey c A'*

lemmas *publicKey-neq-privateKey* = *privateKey-neq-publicKey* [*THEN not-sym*]
declare *publicKey-neq-privateKey* [*iff*]

10.2 Basic properties of *pubK* and *priEK*

lemma *publicKey-inject* [*iff*]: (*publicKey b A* = *publicKey c A*′) = (*b*=*c* ∧ *A*=*A*′)
⟨*proof*⟩

lemma *not-symKeys-pubK* [*iff*]: *publicKey b A* ∉ *symKeys*
⟨*proof*⟩

lemma *not-symKeys-priK* [*iff*]: *privateKey b A* ∉ *symKeys*
⟨*proof*⟩

lemma *symKey-neq-priEK*: *K* ∈ *symKeys* ⟹ *K* ≠ *priEK A*
⟨*proof*⟩

lemma *symKeys-neq-imp-neq*: (*K* ∈ *symKeys*) ≠ (*K*′ ∈ *symKeys*) ⟹ *K* ≠ *K*′
⟨*proof*⟩

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

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

10.3 "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*⟩

10.4 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*⟩

axiomatization where
sym-shrK [*iff*]: *shrK X* ∈ *symKeys* — All shared keys are symmetric

Injectiveness: Agents' long-term keys are distinct.

lemmas *shrK-injective* = *inj-shrK* [*THEN inj-eq*]
declare *shrK-injective* [*iff*]

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

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

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

lemma *priK-neq-shrK* [*iff*]: *shrK A* ≠ *privateKey b C*
⟨*proof*⟩

lemmas *shrK-neq-priK* = *priK-neq-shrK* [*THEN not-sym*]
declare *shrK-neq-priK* [*simp*]

lemma *pubK-neq-shrK* [*iff*]: *shrK A* ≠ *publicKey b C*
⟨*proof*⟩

lemmas *shrK-neq-pubK* = *pubK-neq-shrK* [*THEN not-sym*]
declare *shrK-neq-pubK* [*simp*]

lemma *priEK-noteq-shrK* [*simp*]: *priEK A* ≠ *shrK B*
⟨*proof*⟩

lemma *publicKey-notin-image-shrK* [*simp*]: *publicKey b x* ∉ *shrK* ‘ *AA*
⟨*proof*⟩

lemma *privateKey-notin-image-shrK* [*simp*]: *privateKey b x* $\notin$ *shrK* ‘ *AA*
⟨*proof*⟩

lemma *shrK-notin-image-publicKey* [*simp*]: *shrK x* $\notin$ *publicKey b* ‘ *AA*
⟨*proof*⟩

lemma *shrK-notin-image-privateKey* [*simp*]: *shrK x* $\notin$ *invKey* ‘ *publicKey b* ‘ *AA*
⟨*proof*⟩

lemma *shrK-image-eq* [*simp*]: (*shrK x* $\in$ *shrK* ‘ *AA*) = (*x* $\in$ *AA*)
⟨*proof*⟩

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

declare *invKey-K* [*simp*]

10.5 Initial States of Agents

overloading
  initState ≡ *initState*
begin

primrec *initState* **where**

  initState-Friend:
    initState (*Friend i*) =
      {*Key* (*priEK* (*Friend i*)), *Key* (*priSK* (*Friend i*)), *Key* (*shrK* (*Friend i*))} ∪
       (*Key* ‘ *range pubEK*) ∪ (*Key* ‘ *range pubSK*)
end

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*⟩

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

lemma *MPair-used* [*elim*!]:
     [| ⦃*X*, *Y*⦄ ∈ *used H*;
         [| *X* ∈ *used H*; *Y* ∈ *used H* |] ==> *P* |]
      ==> *P*
⟨*proof*⟩

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

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

lemma *Crypt-notin-initState*: *Crypt K X* $\notin$ *parts* (*initState B*)
⟨*proof*⟩

lemma *Crypt-notin-used-empty* [*simp*]: *Crypt K X* $\notin$ *used* []
⟨*proof*⟩

lemma *shrK-in-initState* [*iff*]: *Key* (*shrK A*) $\in$ *initState A*
⟨*proof*⟩

lemma *shrK-in-knows* [*iff*]: *Key* (*shrK A*) $\in$ *knows A evs*
⟨*proof*⟩

lemma *shrK-in-used* [*iff*]: *Key* (*shrK A*) $\in$ *used evs*
⟨*proof*⟩

lemma *Key-not-used* [*simp*]: *Key K* $\notin$ *used evs* $\Longrightarrow$ *K* $\notin$ *range shrK*
⟨*proof*⟩

lemma *shrK-neq*: *Key K* $\notin$ *used evs* $\Longrightarrow$ *shrK B* $\neq$ *K*
⟨*proof*⟩

lemmas *neq-shrK* = *shrK-neq* [*THEN not-sym*]
declare *neq-shrK* [*simp*]

10.6 Function *knows Spy*

lemma *not-SignatureE* [*elim*!]: *b* $\neq$ *Signature* $\Longrightarrow$ *b* = *Encryption*
⟨*proof*⟩

Agents see their own private keys!

lemma *priK-in-initState* [*iff*]: *Key* (*privateKey b A*) $\in$ *initState A*
⟨*proof*⟩

Agents see all public keys!

lemma *publicKey-in-initState* [*iff*]: *Key* (*publicKey b A*) $\in$ *initState B*
⟨*proof*⟩

All public keys are visible

lemma *spies-pubK* [*iff*]: *Key* (*publicKey b A*) $\in$ *knows B evs*
⟨*proof*⟩

lemmas *analz-spies-pubK* = *spies-pubK* [*THEN analz.Inj*]
declare *analz-spies-pubK* [*iff*]

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-analz-bad*:
[| *Crypt* (*shrK A*) *X* ∈ *analz* (*knows A evs*) |]
==> *X* ∈ *analz* (*knows A evs*)
⟨*proof*⟩

10.7 Fresh Nonces

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

lemma *Nonce-notin-used-empty* [*simp*]: *Nonce N* ∉ *used* []
⟨*proof*⟩

10.8 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*: ∃ *N*. ∀ *n*. *N*≤*n* ⟶ *Nonce n* ∉ *used evs*
⟨*proof*⟩

lemma *Nonce-supply1*: ∃ *N*. *Nonce N* ∉ *used evs*
⟨*proof*⟩

lemma *Nonce-supply*: *Nonce* (*SOME N*. *Nonce N* ∉ *used evs*) ∉ *used evs*
⟨*proof*⟩

10.9 Specialized Rewriting for Theorems About *analz* and Image

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*⟩

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*:
$(Key\ K \in analz\ (Key`nE \cup H)) \longrightarrow (K \in nE \mid Key\ K \in analz\ H) \implies$
$(Key\ K \in analz\ (Key`nE \cup H)) = (K \in nE \mid Key\ K \in analz\ H)$
⟨*proof*⟩

lemmas *analz-image-freshK-simps* =
simp-thms mem-simps — these two allow its use with *only*:
disj-comms
image-insert [*THEN sym*] *image-Un* [*THEN sym*] *empty-subsetI insert-subset*
analz-insert-eq Un-upper2 [*THEN analz-mono*, *THEN subsetD*]
insert-Key-singleton
Key-not-used insert-Key-image Un-assoc [*THEN sym*]

⟨*ML*⟩

10.10 Specialized Methods for Possibility Theorems

⟨*ML*⟩

end

11 The Needham-Schroeder Public-Key Protocol against the General Attacker

theory *NS-Public-Bad-GA* **imports** *PublicGA* **begin**

inductive-set *ns-public* :: *event list set*
where

Nil: [] ∈ *ns-public*

| *Fake*: ⟦*evsf* ∈ *ns-public*; *X* ∈ *synth* (*analz* (*knows A evsf*))⟧
⟹ *Says A B X* # *evsf* ∈ *ns-public*

| *Reception*: ⟦ *evsr* ∈ *ns-public*; *Says A B X* ∈ *set evsr* ⟧
⟹ *Gets B X* # *evsr* ∈ *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*;

Gets 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*;
Gets A (*Crypt* (*pubEK A*) {|*Nonce NA*, *Nonce NB*|}) ∈ *set evs3*]]
⟹ *Says A B* (*Crypt* (*pubEK B*) (*Nonce NB*)) # *evs3* ∈ *ns-public*

lemma *NS-no-Notes*:
evs ∈ *ns-public* ⟹ *Notes A X* ∉ *set evs*
⟨*proof*⟩

Confidentiality treatment in separate theory file

end

12 Inductive Study of Confidentiality against the General Attacker

theory *ConfidentialityGA* **imports** *NS-Public-Bad-GA* **begin**

New subsidiary lemmas to reason on a generic agent initial state

lemma *parts-initState*: *parts*(*initState C*) = *initState C*
⟨*proof*⟩

lemma *analz-initState*: *analz*(*initState C*) = *initState C*
⟨*proof*⟩

Generalising over all initial secrets the existing treatment, which is limited to private encryption keys

definition *staticSecret* :: *agent* ⇒ *msg set* **where**
[*simp*]: *staticSecret A* == {*Key* (*priEK A*), *Key* (*priSK A*), *Key* (*shrK A*)}

More subsidiary lemmas combining initial secrets and knowledge of generic agent

lemma *staticSecret-in-initState* [*simp*]:
staticSecret A ⊆ *initState A*
⟨*proof*⟩
thm *parts-insert*

lemma *staticSecretA-notin-initStateB*:
m ∈ *staticSecret A* ⟹ *m* ∈ *initState B* = (*A*=*B*)
⟨*proof*⟩

lemma *staticSecretA-notin-parts-initStateB*:
m ∈ *staticSecret A* ⟹ *m* ∈ *parts*(*initState B*) = (*A*=*B*)

⟨*proof*⟩

lemma *staticSecretA-notin-analz-initStateB*:
$m \in staticSecret\ A \Longrightarrow m \in analz(initState\ B) = (A{=}B)$
⟨*proof*⟩

lemma *staticSecret-synth-eq*:
$m \in staticSecret\ A \Longrightarrow (m \in synth\ H) = (m \in H)$
⟨*proof*⟩

declare *staticSecret-def* [*simp del*]

lemma *nonce-notin-analz-initState*:
$Nonce\ N \notin analz(initState\ A)$
⟨*proof*⟩

12.1 Protocol independent study

lemma *staticSecret-parts-agent*:
$[\![m \in parts\ (knows\ C\ evs);\ m \in staticSecret\ A]\!] \Longrightarrow$
$A{=}C\ \vee$
$(\exists D\ E\ X.\ Says\ D\ E\ X \in set\ evs \wedge m \in parts\{X\}) \vee$
$(\exists Y.\ Notes\ C\ Y \in set\ evs \wedge m \in parts\{Y\})$
⟨*proof*⟩

lemma *staticSecret-analz-agent*:
$[\![m \in analz\ (knows\ C\ evs);\ m \in staticSecret\ A]\!] \Longrightarrow$
$A{=}C\ \vee$
$(\exists D\ E\ X.\ Says\ D\ E\ X \in set\ evs \wedge m \in parts\{X\}) \vee$
$(\exists Y.\ Notes\ C\ Y \in set\ evs \wedge m \in parts\{Y\})$
⟨*proof*⟩

lemma *secret-parts-agent*:
$m \in parts\ (knows\ C\ evs) \Longrightarrow m \in initState\ C\ \vee$
$(\exists A\ B\ X.\ Says\ A\ B\ X \in set\ evs \wedge m \in parts\{X\}) \vee$
$(\exists Y.\ Notes\ C\ Y \in set\ evs \wedge m \in parts\{Y\})$
⟨*proof*⟩

12.2 Protocol dependent study

lemma *NS-staticSecret-parts-agent-weak*:
$[\![m \in parts\ (knows\ C\ evs);\ m \in staticSecret\ A;$
$evs \in ns\text{-}public]\!] \Longrightarrow$
$A{=}C \vee (\exists D\ E\ X.\ Says\ D\ E\ X \in set\ evs \wedge m \in parts\{X\})$
⟨*proof*⟩

Can't prove the homologous theorem of NS_Says_Spy_staticSecret, hence the specialisation proof strategy cannot be applied

lemma *NS-staticSecret-parts-agent-parts*:

$[\![m \in parts\ (knows\ C\ evs);\ m \in staticSecret\ A;\ A \neq C;\ evs \in ns\text{-}public]\!] \Longrightarrow$
$m \in parts(knows\ D\ evs)$
⟨*proof*⟩

The previous theorems show that in general any agent could send anybody's initial secret, namely the threat model does not impose anything against it. However, the actual protocol specification will, where agents either follow the protocol or build messages out of their traffic analysis - this is actually the same in DY

Therefore, we are only left with the direct proof strategy.

lemma *NS-staticSecret-parts-agent*:
$[\![m \in parts\ (knows\ C\ evs);\ m \in staticSecret\ A;$
$C \neq A;\ evs \in ns\text{-}public]\!]$
$\Longrightarrow \exists\ B\ X.\ Says\ A\ B\ X \in set\ evs \wedge m \in parts\ \{X\}$
⟨*proof*⟩

lemma *NS-agent-see-staticSecret*:
$[\![m \in staticSecret\ A;\ C \neq A;\ evs \in ns\text{-}public]\!]$
$\Longrightarrow m \in parts\ (knows\ C\ evs) = (\exists\ B\ X.\ Says\ A\ B\ X \in set\ evs \wedge m \in parts\ \{X\})$
⟨*proof*⟩

declare *analz.Decrypt* [*rule del*]

lemma *analz-insert-analz*:
$[\![\ c \notin parts\{Z\};\ \forall K.\ Key\ K \notin parts\{Z\};\ c \in analz(insert\ Z\ H)]\!] \Longrightarrow c \in analz\ H$
⟨*proof*⟩

lemma *Agent-not-see-NA*:
$[\![\ Key\ (priEK\ B) \notin analz(knows\ C\ evs);$
$Key\ (priEK\ A) \notin analz(knows\ C\ evs);$
$\forall S\ R\ Y.\ Says\ S\ R\ Y \in set\ evs \longrightarrow$
$Y = Crypt\ (pubEK\ B)\ \{\!|Nonce\ NA,\ Agent\ A|\!\} \vee$
$Y = Crypt\ (pubEK\ A)\ \{\!|Nonce\ NA,\ Nonce\ NB|\!\} \vee$
$Nonce\ NA \notin parts\{Y\} \wedge (\forall K.\ Key\ K \notin parts\{Y\});$
$C \neq A;\ C \neq B;\ \ evs \in ns\text{-}public]\!]$
$\Longrightarrow Nonce\ NA \notin analz\ (knows\ C\ evs)$
⟨*proof*⟩

end

13 Study on knowledge equivalence — results to relate the knowledge of an agent to that of another's

theory *Knowledge*
imports *NS-Public-Bad-GA*
begin

theorem *knowledge-equiv*:
$[\![X \in knows\ A\ evs;\ Notes\ A\ X \notin set\ evs;$
$X \notin \{Key\ (priEK\ A),\ Key\ (priSK\ A),\ Key\ (shrK\ A)\}\ ]\!]$
$\Longrightarrow X \in knows\ B\ evs$
⟨*proof*⟩

lemma *knowledge-equiv-bis*:
$[\![X \in knows\ A\ evs;\ Notes\ A\ X \notin set\ evs\ ]\!]$
$\Longrightarrow X \in \{Key\ (priEK\ A),\ Key\ (priSK\ A),\ Key\ (shrK\ A)\} \cup knows\ B\ evs$
⟨*proof*⟩

lemma *knowledge-equiv-ter*:
$[\![X \in knows\ A\ evs;\ X \notin \{Key\ (priEK\ A),\ Key\ (priSK\ A),\ Key\ (shrK\ A)\}\ ]\!]$
$\Longrightarrow X \in knows\ B\ evs \lor Notes\ A\ X \in set\ evs$
⟨*proof*⟩

lemma *knowledge-equiv-quater*:
$X \in knows\ A\ evs$
$\Longrightarrow X \in knows\ B\ evs \lor Notes\ A\ X \in set\ evs \lor$
$X \in \{Key\ (priEK\ A),\ Key\ (priSK\ A),\ Key\ (shrK\ A)\}$
⟨*proof*⟩

lemma *setdiff-diff-insert*: $A-B-C=D-E-F \Longrightarrow insert\ m\ (A-B-C) = insert\ m\ (D-E-F)$
⟨*proof*⟩

lemma $A-B-C=D-E-F \Longrightarrow insert\ m\ A-B-C = insert\ m\ D-E-F$
⟨*proof*⟩

lemma *knowledge-equiv-eq-setdiff*:
$knows\ A\ evs\ -$
$\{Key\ (priEK\ A),\ Key\ (priSK\ A),\ Key\ (shrK\ A)\}\ -$
$\{X.\ Notes\ A\ X \in set\ evs\}$
$=$
$knows\ B\ evs\ -$
$\{Key\ (priEK\ B),\ Key\ (priSK\ B),\ Key\ (shrK\ B)\}\ -$
$\{X.\ Notes\ B\ X \in set\ evs\}$

⟨*proof*⟩

lemma *knowledge-equiv-eq-old*:
knows A evs ∪
{*Key* (*priEK B*), *Key* (*priSK B*), *Key* (*shrK B*)} ∪
{*X*. *Notes B X* ∈ *set evs*}
=
knows B evs ∪
{*Key* (*priEK A*), *Key* (*priSK A*), *Key* (*shrK A*)} ∪
{*X*. *Notes A X* ∈ *set evs*}
⟨*proof*⟩

theorem *knowledge-eval*: *knows A evs* =
{*Key* (*priEK A*), *Key* (*priSK A*), *Key* (*shrK A*)} ∪
(*Key* ‘ *range pubEK*) ∪ (*Key* ‘ *range pubSK*) ∪
{*X*. ∃ *S R*. *Says S R X* ∈ *set evs*} ∪
{*X*. *Notes A X* ∈ *set evs*}
⟨*proof*⟩

lemma *knowledge-eval-setdiff*:
knows A evs −
{*Key* (*priEK A*), *Key* (*priSK A*), *Key* (*shrK A*)} −
{*X*. *Notes A X* ∈ *set evs*}
=
(*Key* ‘ *range pubEK*) ∪ (*Key* ‘ *range pubSK*) ∪
{*X*. ∃ *S R*. *Says S R X* ∈ *set evs*}
⟨*proof*⟩

theorem *knowledge-equiv-eq*: *knows A evs* ∪
{*Key* (*priEK B*), *Key* (*priSK B*), *Key* (*shrK B*)} ∪
{*X*. *Notes B X* ∈ *set evs*}
=
knows B evs ∪
{*Key* (*priEK A*), *Key* (*priSK A*), *Key* (*shrK A*)} ∪
{*X*. *Notes A X* ∈ *set evs*}
⟨*proof*⟩

lemma *knows A evs* ∪
{*Key* (*priEK B*), *Key* (*priSK B*), *Key* (*shrK B*)} ∪
{*X*. *Notes B X* ∈ *set evs*} −
({*Key* (*priEK B*), *Key* (*priSK B*), *Key* (*shrK B*)} ∪
{*X*. *Notes B X* ∈ *set evs*}) = *knows A evs*
⟨*proof*⟩

theorem *parts-knowledge-equiv-eq*:

parts(*knows A evs*) ∪
{*Key* (*priEK B*), *Key* (*priSK B*), *Key* (*shrK B*)} ∪
parts({*X*. *Notes B X* ∈ *set evs*})
=
parts(*knows B evs*) ∪
{*Key* (*priEK A*), *Key* (*priSK A*), *Key* (*shrK A*)} ∪
parts({*X*. *Notes A X* ∈ *set evs*})
⟨*proof*⟩

lemmas *parts-knowledge-equiv* = *parts-knowledge-equiv-eq* [*THEN equalityD1*, *THEN subsetD*]
thm *parts-knowledge-equiv*
theorem *noprishr-parts-knowledge-equiv*:
⟦ *X* ∉ {*Key* (*priEK A*), *Key* (*priSK A*), *Key* (*shrK A*)};
X ∈ *parts*(*knows A evs*) ⟧
⟹ *X* ∈ *parts*(*knows B evs*) ∪
parts({*X*. *Notes A X* ∈ *set evs*})
⟨*proof*⟩

lemma *knowledge-equiv-eq-NS*:
evs ∈ *ns-public* ⟹
knows A evs ∪ {*Key* (*priEK B*), *Key* (*priSK B*), *Key* (*shrK B*)} =
knows B evs ∪ {*Key* (*priEK A*), *Key* (*priSK A*), *Key* (*shrK A*)}
⟨*proof*⟩

lemma *parts-knowledge-equiv-eq-NS*:
evs ∈ *ns-public* ⟹
parts(*knows A evs*) ∪ {*Key* (*priEK B*), *Key* (*priSK B*), *Key* (*shrK B*)} =
parts(*knows B evs*) ∪ {*Key* (*priEK A*), *Key* (*priSK A*), *Key* (*shrK A*)}
⟨*proof*⟩

theorem *noprishr-parts-knowledge-equiv-NS*:
⟦ *X* ∉ {*Key* (*priEK A*), *Key* (*priSK A*), *Key* (*shrK A*)};
X ∈ *parts*(*knows A evs*); *evs* ∈ *ns-public* ⟧
⟹ *X* ∈ *parts*(*knows B evs*)
⟨*proof*⟩

theorem *Agent-not-analz-N*:
⟦ *Nonce N* ∉ *parts*(*knows A evs*); *evs* ∈ *ns-public* ⟧
⟹ *Nonce N* ∉ *analz*(*knows B evs*)
⟨*proof*⟩

end

References

[1] G. Bella. Inductive study of confidentiality — for everyone. *Formal Aspects of Computing*, 2012. In press.