

# Inductive Study of Confidentiality

Giampaolo Bella

Dipartimento di Matematica e Informatica, Università di Catania, Italy

February 6, 2026

## 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

<b>1 Theory of Agents and Messages for Security Protocols against Dolev-Yao</b>	<b>4</b>
1.1 Inductive definition of all parts of a message . . . . .	5
1.2 Inverse of keys . . . . .	5
1.3 keysFor operator . . . . .	5
1.4 Inductive relation "parts" . . . . .	6
1.4.1 Unions . . . . .	7
1.4.2 Idempotence and transitivity . . . . .	8
1.4.3 Rewrite rules for pulling out atomic messages . . . . .	8
1.5 Inductive relation "analz" . . . . .	10
1.5.1 General equational properties . . . . .	11
1.5.2 Rewrite rules for pulling out atomic messages . . . . .	11
1.5.3 Idempotence and transitivity . . . . .	13
1.6 Inductive relation "synth" . . . . .	14
1.6.1 Unions . . . . .	15
1.6.2 Idempotence and transitivity . . . . .	15

1.6.3	Combinations of parts, <i>analz</i> and <i>synth</i> . . . . .	16
1.6.4	For reasoning about the Fake rule in traces . . . . .	16
1.7	HPair: a combination of Hash and MPair . . . . .	17
1.7.1	Freeness . . . . .	17
1.7.2	Specialized laws, proved in terms of those for Hash and MPair . . . . .	18
1.8	The set of key-free messages . . . . .	19
1.9	Tactics useful for many protocol proofs . . . . .	20
<b>2</b>	<b>Theory of Events for Security Protocols against Dolev-Yao</b>	<b>22</b>
2.1	Function <i>knows</i> . . . . .	23
2.2	Knowledge of Agents . . . . .	24
<b>3</b>	<b>Theory of Cryptographic Keys for Security Protocols against Dolev-Yao</b>	<b>28</b>
3.1	Asymmetric Keys . . . . .	28
3.2	Basic properties of <i>pubEK</i> and <i>priEK</i> . . . . .	29
3.3	"Image" equations that hold for injective functions . . . . .	30
3.4	Symmetric Keys . . . . .	30
3.5	Initial States of Agents . . . . .	31
3.6	Function <i>knows Spy</i> . . . . .	33
3.7	Fresh Nonces . . . . .	34
3.8	Supply fresh nonces for possibility theorems . . . . .	35
3.9	Specialized Rewriting for Theorems About <i>analz</i> and Image . . . . .	35
3.10	Specialized Methods for Possibility Theorems . . . . .	36
<b>4</b>	<b>The Needham-Schroeder Public-Key Protocol against Dolev- Yao — with Gets event, hence with Reception rule</b>	<b>37</b>
<b>5</b>	<b>Inductive Study of Confidentiality against Dolev-Yao</b>	<b>41</b>
<b>6</b>	<b>Existing study - fully spelled out</b>	<b>41</b>
6.1	On static secrets . . . . .	42
6.2	On dynamic secrets . . . . .	42
<b>7</b>	<b>Novel study</b>	<b>43</b>
7.1	Protocol independent study . . . . .	43
7.2	Protocol-dependent study . . . . .	48
<b>8</b>	<b>Theory of Agents and Messages for Security Protocols against the General Attacker</b>	<b>51</b>
8.1	Inductive definition of all parts of a message . . . . .	52
8.2	Inverse of keys . . . . .	53
8.3	keysFor operator . . . . .	53
8.4	Inductive relation "parts" . . . . .	54

8.4.1	Unions . . . . .	54
8.4.2	Idempotence and transitivity . . . . .	55
8.4.3	Rewrite rules for pulling out atomic messages . . . . .	56
8.5	Inductive relation "analz" . . . . .	57
8.5.1	General equational properties . . . . .	58
8.5.2	Rewrite rules for pulling out atomic messages . . . . .	58
8.5.3	Idempotence and transitivity . . . . .	60
8.6	Inductive relation "synth" . . . . .	61
8.6.1	Unions . . . . .	62
8.6.2	Idempotence and transitivity . . . . .	62
8.6.3	Combinations of parts, analz and synth . . . . .	63
8.6.4	For reasoning about the Fake rule in traces . . . . .	64
8.7	HPair: a combination of Hash and MPair . . . . .	65
8.7.1	Freeness . . . . .	65
8.7.2	Specialized laws, proved in terms of those for Hash and MPair . . . . .	66
8.8	The set of key-free messages . . . . .	67
8.9	Tactics useful for many protocol proofs . . . . .	67
<b>9</b>	<b>Theory of Events for Security Protocols against the General Attacker</b>	<b>69</b>
9.1	Function <i>knows</i> . . . . .	70
9.2	Knowledge of generic agents . . . . .	71
<b>10</b>	<b>Theory of Cryptographic Keys for Security Protocols against the General Attacker</b>	<b>73</b>
10.1	Asymmetric Keys . . . . .	73
10.2	Basic properties of <i>pubEK</i> and <i>priEK</i> . . . . .	75
10.3	"Image" equations that hold for injective functions . . . . .	75
10.4	Symmetric Keys . . . . .	76
10.5	Initial States of Agents . . . . .	77
10.6	Function <i>knows Spy</i> . . . . .	78
10.7	Fresh Nonces . . . . .	79
10.8	Supply fresh nonces for possibility theorems . . . . .	79
10.9	Specialized Rewriting for Theorems About <i>analz</i> and Image . . . . .	80
10.10	Specialized Methods for Possibility Theorems . . . . .	81
<b>11</b>	<b>The Needham-Schroeder Public-Key Protocol against the General Attacker</b>	<b>81</b>
<b>12</b>	<b>Inductive Study of Confidentiality against the General At- tacker</b>	<b>82</b>
12.1	Protocol independent study . . . . .	83
12.2	Protocol dependent study . . . . .	85

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

```
theory Message
imports Main
begin
```

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

```
type-synonym
key = nat
```

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

```
specification (invKey)
invKey [simp]: invKey (invKey K) = K
invKey-symmetric: all-symmetric --> invKey = id
by (rule exI [of - id], auto)
```

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

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

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

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

Concrete syntax: messages appear as  $\{A,B,NA\}$ , etc...

```
syntax
-MTuple :: ['a, args] => 'a * 'b ((⟨⟨indent=2 notation=⟨mixfix message tuple⟩⟩{-/-}⟩))
syntax-consts
```

$-MTuple == MPair$

**translations**

$\{x, y, z\} == \{x, \{y, z\}\}$   
 $\{x, y\} == CONST MPair x y$

**definition**  $HPair :: [msg, msg] ==> msg$  ( $\langle (4Hash[-] /-) \rangle [0, 1000]$ ) **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**  $parts-mono: G \subseteq H ==> parts(G) \subseteq parts(H)$

**apply** *auto*

**apply** (*erule parts.induct*)

**apply** (*blast dest: parts.Fst parts.Snd parts.Body*)**+**

**done**

Equations hold because constructors are injective.

**lemma**  $Friend-image-eq [simp]: (Friend x \in Friend'A) = (x:A)$   
**by** *auto*

**lemma**  $Key-image-eq [simp]: (Key x \in Key'A) = (x \in A)$   
**by** *auto*

**lemma**  $Nonce-Key-image-eq [simp]: (Nonce x \notin Key'A)$   
**by** *auto*

## 1.2 Inverse of keys

**lemma**  $invKey-eq [simp]: (invKey K = invKey K') = (K=K')$   
**by** (*metis invKey*)

## 1.3 keysFor operator

**lemma**  $keysFor-empty [simp]: keysFor \{\} = \{\}$

**by** (*unfold keysFor-def, blast*)

**lemma** *keysFor-Un* [*simp*]:  $keysFor (H \cup H') = keysFor H \cup keysFor H'$   
**by** (*unfold keysFor-def, blast*)

**lemma** *keysFor-UN* [*simp*]:  $keysFor (\bigcup_{i \in A} H i) = (\bigcup_{i \in A} keysFor (H i))$   
**by** (*unfold keysFor-def, blast*)

Monotonicity

**lemma** *keysFor-mono*:  $G \subseteq H \implies keysFor(G) \subseteq keysFor(H)$   
**by** (*unfold keysFor-def, blast*)

**lemma** *keysFor-insert-Agent* [*simp*]:  $keysFor (insert (Agent A) H) = keysFor H$   
**by** (*unfold keysFor-def, auto*)

**lemma** *keysFor-insert-Nonce* [*simp*]:  $keysFor (insert (Nonce N) H) = keysFor H$   
**by** (*unfold keysFor-def, auto*)

**lemma** *keysFor-insert-Number* [*simp*]:  $keysFor (insert (Number N) H) = keysFor H$   
**by** (*unfold keysFor-def, auto*)

**lemma** *keysFor-insert-Key* [*simp*]:  $keysFor (insert (Key K) H) = keysFor H$   
**by** (*unfold keysFor-def, auto*)

**lemma** *keysFor-insert-Hash* [*simp*]:  $keysFor (insert (Hash X) H) = keysFor H$   
**by** (*unfold keysFor-def, auto*)

**lemma** *keysFor-insert-MPair* [*simp*]:  $keysFor (insert \{X, Y\} H) = keysFor H$   
**by** (*unfold keysFor-def, auto*)

**lemma** *keysFor-insert-Crypt* [*simp*]:  
 $keysFor (insert (Crypt K X) H) = insert (invKey K) (keysFor H)$   
**by** (*unfold keysFor-def, auto*)

**lemma** *keysFor-image-Key* [*simp*]:  $keysFor (Key'E) = \{\}$   
**by** (*unfold keysFor-def, auto*)

**lemma** *Crypt-imp-invKey-keysFor*:  $Crypt K X \in H \implies invKey K \in keysFor H$   
**by** (*unfold keysFor-def, blast*)

## 1.4 Inductive relation "parts"

**lemma** *MPair-parts*:  
 $[\{X, Y\} \in parts H;$   
 $[\{X \in parts H; Y \in parts H\}] \implies P \implies P$   
**by** (*blast dest: parts.Fst parts.Snd*)

**declare** *MPair-parts* [*elim!*] *parts.Body* [*dest!*]

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

**lemma** *parts-increasing*:  $H \subseteq \text{parts}(H)$   
**by** *blast*

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

**lemma** *parts-empty* [*simp*]:  $\text{parts}\{\} = \{\}$   
**apply** *safe*  
**apply** (*erule parts.induct*, *blast+*)  
**done**

**lemma** *parts-emptyE* [*elim!*]:  $X \in \text{parts}\{\} \implies P$   
**by** *simp*

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

**lemma** *parts-singleton*:  $X \in \text{parts } H \implies \exists Y \in H. X \in \text{parts } \{Y\}$   
**by** (*erule parts.induct*, *fast+*)

#### 1.4.1 Unions

**lemma** *parts-Un-subset1*:  $\text{parts}(G) \cup \text{parts}(H) \subseteq \text{parts}(G \cup H)$   
**by** (*intro Un-least parts-mono Un-upper1 Un-upper2*)

**lemma** *parts-Un-subset2*:  $\text{parts}(G \cup H) \subseteq \text{parts}(G) \cup \text{parts}(H)$   
**apply** (*rule subsetI*)  
**apply** (*erule parts.induct*, *blast+*)  
**done**

**lemma** *parts-Un* [*simp*]:  $\text{parts}(G \cup H) = \text{parts}(G) \cup \text{parts}(H)$   
**by** (*intro equalityI parts-Un-subset1 parts-Un-subset2*)

**lemma** *parts-insert*:  $\text{parts}(\text{insert } X H) = \text{parts } \{X\} \cup \text{parts } H$   
**by** (*metis insert-is-Un parts-Un*)

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

**lemma** *parts-insert2*:  
 $\text{parts}(\text{insert } X (\text{insert } Y H)) = \text{parts } \{X\} \cup \text{parts } \{Y\} \cup \text{parts } H$   
**by** (*metis Un-commute Un-empty-right Un-insert-right insert-is-Un parts-Un*)

Added to simplify arguments to *parts*, *analz* and *synth*.

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

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

**declare** *in-parts-UnE* [*elim!*]

**lemma** *parts-insert-subset*:  $\text{insert } X \text{ (parts } H) \subseteq \text{parts}(\text{insert } X \ H)$   
**by** (*blast intro: parts-mono [THEN [2] rev-subsetD]*)

### 1.4.2 Idempotence and transitivity

**lemma** *parts-partsD* [*dest!*]:  $X \in \text{parts} \text{ (parts } H) \implies X \in \text{parts } H$   
**by** (*erule parts.induct, blast+*)

**lemma** *parts-idem* [*simp*]:  $\text{parts} \text{ (parts } H) = \text{parts } H$   
**by** *blast*

**lemma** *parts-subset-iff* [*simp*]:  $(\text{parts } G \subseteq \text{parts } H) = (G \subseteq \text{parts } H)$   
**by** (*metis parts-idem parts-increasing parts-mono subset-trans*)

**lemma** *parts-trans*:  $[\!| X \in \text{parts } G; G \subseteq \text{parts } H \!|] \implies X \in \text{parts } H$   
**by** (*metis parts-subset-iff subsetD*)

Cut

**lemma** *parts-cut*:  
 $[\!| Y \in \text{parts} \text{ (insert } X \ G); X \in \text{parts } H \!|] \implies Y \in \text{parts} \text{ (} G \cup H \text{)}$   
**by** (*blast intro: parts-trans*)

**lemma** *parts-cut-eq* [*simp*]:  $X \in \text{parts } H \implies \text{parts} \text{ (insert } X \ H) = \text{parts } H$   
**by** (*metis insert-absorb parts-idem parts-insert*)

### 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*]:  
 $\text{parts} \text{ (insert (Agent } agt) \ H) = \text{insert} \text{ (Agent } agt) \text{ (parts } H)$   
**apply** (*rule parts-insert-eq-I*)  
**apply** (*erule parts.induct, auto*)  
**done**

**lemma** *parts-insert-Nonce* [*simp*]:  
 $\text{parts} \text{ (insert (Nonce } N) \ H) = \text{insert} \text{ (Nonce } N) \text{ (parts } H)$   
**apply** (*rule parts-insert-eq-I*)  
**apply** (*erule parts.induct, auto*)  
**done**

**lemma** *parts-insert-Number* [*simp*]:  
 $\text{parts} \text{ (insert (Number } N) \ H) = \text{insert} \text{ (Number } N) \text{ (parts } H)$   
**apply** (*rule parts-insert-eq-I*)  
**apply** (*erule parts.induct, auto*)  
**done**

```

lemma parts-insert-Key [simp]:
   $parts (insert (Key K) H) = insert (Key K) (parts H)$ 
apply (rule parts-insert-eq-I)
apply (erule parts.induct, auto)
done

lemma parts-insert-Hash [simp]:
   $parts (insert (Hash X) H) = insert (Hash X) (parts H)$ 
apply (rule parts-insert-eq-I)
apply (erule parts.induct, auto)
done

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

lemma parts-insert-MPair [simp]:
   $parts (insert \{X, Y\} H) =$ 
     $insert \{X, Y\} (parts (insert X (insert Y H)))$ 
apply (rule equalityI)
apply (rule subsetI)
apply (erule parts.induct, auto)
apply (blast intro: parts.Fst parts.Snd)+
done

lemma parts-image-Key [simp]:  $parts (Key'N) = Key'N$ 
apply auto
apply (erule parts.induct, auto)
done

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

lemma msg-Nonce-supply:  $\exists N. \forall n. N \leq n \longrightarrow Nonce\ n \notin parts\ \{msg\}$ 
apply (induct msg)
apply (simp-all (no-asm-simp) add: exI parts-insert2)

Nonce case
apply (metis Suc-n-not-le-n)

MPair case: metis works out the necessary sum itself!
apply (metis le-trans nat-le-linear)
done

```

## 1.5 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*] :  $X \in H \implies X \in \text{analz } H$

| *Fst*:  $\{X, Y\} \in \text{analz } H \implies X \in \text{analz } H$

| *Snd*:  $\{X, Y\} \in \text{analz } H \implies Y \in \text{analz } H$

| *Decrypt* [*dest*]:

$[\text{Crypt } K X \in \text{analz } H; \text{Key}(\text{invKey } K) \in \text{analz } H] \implies X \in \text{analz } H$

Monotonicity; Lemma 1 of Lowe's paper

**lemma** *analz-mono*:  $G \subseteq H \implies \text{analz}(G) \subseteq \text{analz}(H)$

**apply** *auto*

**apply** (*erule analz.induct*)

**apply** (*auto dest: analz.Fst analz.Snd*)

**done**

Making it safe speeds up proofs

**lemma** *MPair-analz* [*elim!*]:

$[\{X, Y\} \in \text{analz } H;$

$[\ X \in \text{analz } H; Y \in \text{analz } H \ ] \implies P$

$]\implies P$

**by** (*blast dest: analz.Fst analz.Snd*)

**lemma** *analz-increasing*:  $H \subseteq \text{analz}(H)$

**by** *blast*

**lemma** *analz-subset-parts*:  $\text{analz } H \subseteq \text{parts } H$

**apply** (*rule subsetI*)

**apply** (*erule analz.induct, blast+*)

**done**

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

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

**lemma** *parts-analz* [*simp*]:  $\text{parts}(\text{analz } H) = \text{parts } H$

**by** (*metis analz-increasing analz-subset-parts equalityI parts-mono parts-subset-iff*)

**lemma** *analz-parts* [*simp*]:  $\text{analz}(\text{parts } H) = \text{parts } H$

**apply** *auto*

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

**done**

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

### 1.5.1 General equational properties

**lemma** *analz-empty [simp]: analz{} = {}*

**apply** *safe*

**apply** (*erule analz.induct, blast+*)

**done**

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

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

**by** (*intro Un-least analz-mono Un-upper1 Un-upper2*)

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

**by** (*blast intro: analz-mono [THEN [2] rev-subsetD]*)

### 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)*

**apply** (*rule analz-insert-eq-I*)

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

**done**

**lemma** *analz-insert-Nonce [simp]:*

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

**apply** (*rule analz-insert-eq-I*)

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

**done**

**lemma** *analz-insert-Number [simp]:*

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

**apply** (*rule analz-insert-eq-I*)

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

**done**

**lemma** *analz-insert-Hash [simp]:*

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

**apply** (*rule analz-insert-eq-I*)

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

**done**

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

**lemma** *analz-insert-Key [simp]:*

```

     $K \notin \text{keysFor} (\text{analz } H) \implies$ 
       $\text{analz} (\text{insert} (\text{Key } K) H) = \text{insert} (\text{Key } K) (\text{analz } H)$ 
apply (unfold keysFor-def)
apply (rule analz-insert-eq-I)
apply (erule analz.induct, auto)
done

```

```

lemma analz-insert-MPair [simp]:
   $\text{analz} (\text{insert} \{\!| X, Y |\!\} H) =$ 
     $\text{insert} \{\!| X, Y |\!\} (\text{analz} (\text{insert } X (\text{insert } Y H)))$ 
apply (rule equalityI)
apply (rule subsetI)
apply (erule analz.induct, auto)
apply (erule analz.induct)
apply (blast intro: analz.Fst analz.Snd)+
done

```

Can pull out enCrypted message if the Key is not known

```

lemma analz-insert-Crypt:
   $\text{Key} (\text{invKey } K) \notin \text{analz } H$ 
   $\implies \text{analz} (\text{insert} (\text{Crypt } K X) H) = \text{insert} (\text{Crypt } K X) (\text{analz } H)$ 
apply (rule analz-insert-eq-I)
apply (erule analz.induct, auto)

done

```

```

lemma lemma1:  $\text{Key} (\text{invKey } K) \in \text{analz } H \implies$ 
   $\text{analz} (\text{insert} (\text{Crypt } K X) H) \subseteq$ 
   $\text{insert} (\text{Crypt } K X) (\text{analz} (\text{insert } X H))$ 
apply (rule subsetI)
apply (erule-tac  $x = x$  in analz.induct, auto)
done

```

```

lemma lemma2:  $\text{Key} (\text{invKey } K) \in \text{analz } H \implies$ 
   $\text{insert} (\text{Crypt } K X) (\text{analz} (\text{insert } X H)) \subseteq$ 
   $\text{analz} (\text{insert} (\text{Crypt } K X) H)$ 
apply auto
apply (erule-tac  $x = x$  in analz.induct, auto)
apply (blast intro: analz-insertI analz.Decrypt)
done

```

```

lemma analz-insert-Decrypt:
   $\text{Key} (\text{invKey } K) \in \text{analz } H \implies$ 
   $\text{analz} (\text{insert} (\text{Crypt } K X) H) =$ 
   $\text{insert} (\text{Crypt } K X) (\text{analz} (\text{insert } X H))$ 
by (intro equalityI lemma1 lemma2)

```

Case analysis: either the message is secure, or it is not! Effective, but can cause subgoals to blow up! Use with *if-split*; apparently *split-tac* does not

cope with patterns such as  $\text{analz} (\text{insert} (\text{Crypt } K \ X) \ H)$

**lemma** *analz-Crypt-if* [simp]:  
 $\text{analz} (\text{insert} (\text{Crypt } K \ X) \ H) =$   
 $(\text{if } (\text{Key} (\text{invKey } K) \in \text{analz } H)$   
 $\text{then } \text{insert} (\text{Crypt } K \ X) (\text{analz} (\text{insert } X \ H))$   
 $\text{else } \text{insert} (\text{Crypt } K \ X) (\text{analz } H))$   
**by** (*simp add: analz-insert-Crypt analz-insert-Decrypt*)

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

**lemma** *analz-insert-Crypt-subset*:  
 $\text{analz} (\text{insert} (\text{Crypt } K \ X) \ H) \subseteq$   
 $\text{insert} (\text{Crypt } K \ X) (\text{analz} (\text{insert } X \ H))$   
**apply** (*rule subsetI*)  
**apply** (*erule analz.induct, auto*)  
**done**

**lemma** *analz-image-Key* [simp]:  $\text{analz} (\text{Key}'N) = \text{Key}'N$   
**apply** *auto*  
**apply** (*erule analz.induct, auto*)  
**done**

### 1.5.3 Idempotence and transitivity

**lemma** *analz-analzD* [dest!]:  $X \in \text{analz} (\text{analz } H) \implies X \in \text{analz } H$   
**by** (*erule analz.induct, blast+*)

**lemma** *analz-idem* [simp]:  $\text{analz} (\text{analz } H) = \text{analz } H$   
**by** *blast*

**lemma** *analz-subset-iff* [simp]:  $(\text{analz } G \subseteq \text{analz } H) = (G \subseteq \text{analz } H)$   
**by** (*metis analz-idem analz-increasing analz-mono subset-trans*)

**lemma** *analz-trans*:  $[| X \in \text{analz } G; G \subseteq \text{analz } H |] \implies X \in \text{analz } H$   
**by** (*drule analz-mono, blast*)

Cut; Lemma 2 of Lowe

**lemma** *analz-cut*:  $[| Y \in \text{analz} (\text{insert } X \ H); X \in \text{analz } H |] \implies Y \in \text{analz } H$   
**by** (*erule analz-trans, blast*)

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

**lemma** *analz-insert-eq*:  $X \in \text{analz } H \implies \text{analz} (\text{insert } X \ H) = \text{analz } H$   
**by** (*metis analz-cut analz-insert-eq-I insert-absorb*)

A congruence rule for "analz"

**lemma** *analz-subset-cong*:

$$\llbracket \text{analz } G \subseteq \text{analz } G'; \text{analz } H \subseteq \text{analz } H' \rrbracket$$

$$\implies \text{analz } (G \cup H) \subseteq \text{analz } (G' \cup H')$$
**by** (*metis Un-mono analz-Un analz-subset-iff subset-trans*)

**lemma analz-cong:**  

$$\llbracket \text{analz } G = \text{analz } G'; \text{analz } H = \text{analz } H' \rrbracket$$

$$\implies \text{analz } (G \cup H) = \text{analz } (G' \cup H')$$
**by** (*intro equalityI analz-subset-cong, simp-all*)

**lemma analz-insert-cong:**  

$$\text{analz } H = \text{analz } H' \implies \text{analz}(\text{insert } X H) = \text{analz}(\text{insert } X H')$$
**by** (*force simp only: insert-def intro!: analz-cong*)

If there are no pairs or encryptions then analz does nothing

**lemma analz-trivial:**  

$$\llbracket \forall X Y. \{X, Y\} \notin H; \forall X K. \text{Crypt } K X \notin H \rrbracket \implies \text{analz } H = H$$
**apply** *safe*  
**apply** (*erule analz.induct, blast+*)  
**done**

## 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**  
 $\text{synth} :: \text{msg set} \implies \text{msg set}$   
**for**  $H :: \text{msg set}$   
**where**  
 $\text{Inj} \quad [\text{intro}]: X \in H \implies X \in \text{synth } H$   
 $\text{Agent} \quad [\text{intro}]: \text{Agent } \text{agt} \in \text{synth } H$   
 $\text{Number} \quad [\text{intro}]: \text{Number } n \in \text{synth } H$   
 $\text{Hash} \quad [\text{intro}]: X \in \text{synth } H \implies \text{Hash } X \in \text{synth } H$   
 $\text{MPair} \quad [\text{intro}]: \llbracket X \in \text{synth } H; Y \in \text{synth } H \rrbracket \implies \{X, Y\} \in \text{synth } H$   
 $\text{Crypt} \quad [\text{intro}]: \llbracket X \in \text{synth } H; \text{Key}(K) \in H \rrbracket \implies \text{Crypt } K X \in \text{synth } H$

Monotonicity

**lemma synth-mono:**  $G \subseteq H \implies \text{synth}(G) \subseteq \text{synth}(H)$   
**by** (*auto, erule synth.induct, auto*)

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

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

$\{X, Y\} \in \text{synth } H$   
*Crypt*  $K X \in \text{synth } H$

**lemma** *synth-increasing*:  $H \subseteq \text{synth}(H)$   
**by** *blast*

### 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*:  $\text{synth}(G) \cup \text{synth}(H) \subseteq \text{synth}(G \cup H)$   
**by** (*intro Un-least synth-mono Un-upper1 Un-upper2*)

**lemma** *synth-insert*:  $\text{insert } X (\text{synth } H) \subseteq \text{synth}(\text{insert } X H)$   
**by** (*blast intro: synth-mono [THEN [2] rev-subsetD]*)

### 1.6.2 Idempotence and transitivity

**lemma** *synth-synthD* [*dest!*]:  $X \in \text{synth} (\text{synth } H) \implies X \in \text{synth } H$   
**by** (*erule synth.induct, auto*)

**lemma** *synth-idem*:  $\text{synth} (\text{synth } H) = \text{synth } H$   
**by** *blast*

**lemma** *synth-subset-iff* [*simp*]:  $(\text{synth } G \subseteq \text{synth } H) = (G \subseteq \text{synth } H)$   
**by** (*metis subset-trans synth-idem synth-increasing synth-mono*)

**lemma** *synth-trans*:  $[[ X \in \text{synth } G; G \subseteq \text{synth } H ]] \implies X \in \text{synth } H$   
**by** (*drule synth-mono, blast*)

Cut; Lemma 2 of Lowe

**lemma** *synth-cut*:  $[[ Y \in \text{synth} (\text{insert } X H); X \in \text{synth } H ]] \implies Y \in \text{synth } H$   
**by** (*erule synth-trans, blast*)

**lemma** *Agent-synth* [*simp*]:  $\text{Agent } A \in \text{synth } H$   
**by** *blast*

**lemma** *Number-synth* [*simp*]:  $\text{Number } n \in \text{synth } H$   
**by** *blast*

**lemma** *Nonce-synth-eq* [*simp*]:  $(\text{Nonce } N \in \text{synth } H) = (\text{Nonce } N \in H)$   
**by** *blast*

**lemma** *Key-synth-eq* [*simp*]:  $(\text{Key } K \in \text{synth } H) = (\text{Key } K \in H)$   
**by** *blast*

**lemma** *Crypt-synth-eq* [*simp*]:  
 $\text{Key } K \notin H \implies (\text{Crypt } K X \in \text{synth } H) = (\text{Crypt } K X \in H)$   
**by** *blast*

**lemma** *keysFor-synth* [*simp*]:  
 $keysFor (synth H) = keysFor H \cup invKey\{K. Key K \in H\}$   
**by** (*unfold keysFor-def, blast*)

### 1.6.3 Combinations of parts, analz and synth

**lemma** *parts-synth* [*simp*]:  $parts (synth H) = parts H \cup synth H$   
**apply** (*rule equalityI*)  
**apply** (*rule subsetI*)  
**apply** (*erule parts.induct*)  
**apply** (*blast intro: synth-increasing [THEN parts-mono, THEN subsetD]*  
 $parts.Fst parts.Snd parts.Body$ )  
**done**

**lemma** *analz-analz-Un* [*simp*]:  $analz (analz G \cup H) = analz (G \cup H)$   
**apply** (*intro equalityI analz-subset-cong*)  
**apply** *simp-all*  
**done**

**lemma** *analz-synth-Un* [*simp*]:  $analz (synth G \cup H) = analz (G \cup H) \cup synth G$   
**apply** (*rule equalityI*)  
**apply** (*rule subsetI*)  
**apply** (*erule analz.induct*)  
**prefer** 5 **apply** (*blast intro: analz-mono [THEN [2] rev-subsetD]*)  
**apply** (*blast intro: analz.Fst analz.Snd analz.Decrypt*)  
**done**

**lemma** *analz-synth* [*simp*]:  $analz (synth H) = analz H \cup synth H$   
**by** (*metis Un-empty-right analz-synth-Un*)

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

**lemma** *parts-insert-subset-Un*:  $X \in G \implies parts(insert X H) \subseteq parts G \cup parts H$   
**by** (*metis UnCI Un-upper2 insert-subset parts-Un parts-mono*)

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

**lemma** *Fake-parts-insert*:  
 $X \in synth (analz H) \implies$   
 $parts (insert X H) \subseteq synth (analz H) \cup parts H$   
**by** (*metis Un-commute analz-increasing insert-subset parts-analz parts-mono*  
 $parts-synth synth-mono synth-subset-iff$ )

**lemma** *Fake-parts-insert-in-Un*:  
 $[[Z \in parts (insert X H); X: synth (analz H)]]$   
 $\implies Z \in synth (analz H) \cup parts H$   
**by** (*metis Fake-parts-insert subsetD*)

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

**lemma** *Fake-analz-insert*:

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

**apply** (*rule subsetI*)

**apply** (*subgoal-tac*  $x \in \text{analz} (\text{synth} (\text{analz } G) \cup H)$ , *force*)

**apply** (*blast intro: analz-mono [THEN [2] rev-subsetD] analz-mono [THEN synth-mono, THEN [2] rev-subsetD]*)

**done**

**lemma** *analz-conj-parts [simp]*:

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

**by** (*blast intro: analz-subset-parts [THEN subsetD]*)

**lemma** *analz-disj-parts [simp]*:

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

**by** (*blast intro: analz-subset-parts [THEN subsetD]*)

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

**lemma** *MPair-synth-analz [iff]*:

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

**by** *blast*

**lemma** *Crypt-synth-analz*:

$$[\text{Key } K \in \text{analz } H; \text{Key} (\text{invKey } K) \in \text{analz } H] \\ \implies (\text{Crypt } K X \in \text{synth} (\text{analz } H)) = (X \in \text{synth} (\text{analz } H))$$

**by** *blast*

**lemma** *Hash-synth-analz [simp]*:

$$X \notin \text{synth} (\text{analz } H) \\ \implies (\text{Hash}\{X, Y\} \in \text{synth} (\text{analz } H)) = (\text{Hash}\{X, Y\} \in \text{analz } H)$$

**by** *blast*

## 1.7 HPair: a combination of Hash and MPair

### 1.7.1 Freeness

**lemma** *Agent-neq-HPair*:  $\text{Agent } A \sim = \text{Hash}[X] Y$

**by** (*unfold HPair-def, simp*)

**lemma** *Nonce-neq-HPair*:  $\text{Nonce } N \sim = \text{Hash}[X] Y$

**by** (*unfold HPair-def, simp*)

**lemma** *Number-neq-HPair*:  $\text{Number } N \sim = \text{Hash}[X] Y$

**by** (*unfold HPair-def, simp*)

**lemma** *Key-neq-HPair*:  $Key\ K \sim = Hash[X]\ Y$   
**by** (*unfold HPair-def, simp*)

**lemma** *Hash-neq-HPair*:  $Hash\ Z \sim = Hash[X]\ Y$   
**by** (*unfold HPair-def, simp*)

**lemma** *Crypt-neq-HPair*:  $Crypt\ K\ X' \sim = Hash[X]\ Y$   
**by** (*unfold HPair-def, simp*)

**lemmas** *HPair-neqs = Agent-neq-HPair Nonce-neq-HPair Number-neq-HPair  
Key-neq-HPair Hash-neq-HPair Crypt-neq-HPair*

**declare** *HPair-neqs* [*iff*]  
**declare** *HPair-neqs* [*symmetric, iff*]

**lemma** *HPair-eq* [*iff*]:  $(Hash[X]\ Y' = Hash[X]\ Y) = (X' = X \wedge Y' = Y)$   
**by** (*simp add: HPair-def*)

**lemma** *MPair-eq-HPair* [*iff*]:  
 $(\{X', Y'\} = Hash[X]\ Y) = (X' = Hash\{X, Y\} \wedge Y' = Y)$   
**by** (*simp add: HPair-def*)

**lemma** *HPair-eq-MPair* [*iff*]:  
 $(Hash[X]\ Y = \{X', Y'\}) = (X' = Hash\{X, Y\} \wedge Y' = Y)$   
**by** (*auto simp add: HPair-def*)

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

**lemma** *keysFor-insert-HPair* [*simp*]:  $keysFor\ (insert\ (Hash[X]\ Y)\ H) = keysFor\ H$   
**by** (*simp add: HPair-def*)

**lemma** *parts-insert-HPair* [*simp*]:  
 $parts\ (insert\ (Hash[X]\ Y)\ H) =$   
 $insert\ (Hash[X]\ Y)\ (insert\ (Hash\{X, Y\})\ (parts\ (insert\ Y\ H)))$   
**by** (*simp add: HPair-def*)

**lemma** *analz-insert-HPair* [*simp*]:  
 $analz\ (insert\ (Hash[X]\ Y)\ H) =$   
 $insert\ (Hash[X]\ Y)\ (insert\ (Hash\{X, Y\})\ (analz\ (insert\ Y\ H)))$   
**by** (*simp add: HPair-def*)

**lemma** *HPair-synth-analz* [*simp*]:  
 $X \notin synth\ (analz\ H)$   
 $\implies (Hash[X]\ Y \in synth\ (analz\ H)) =$   
 $(Hash\{X, Y\} \in analz\ H \wedge Y \in synth\ (analz\ H))$   
**by** (*auto simp add: HPair-def*)

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

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

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

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

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

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

**lemmas** *pushes* = *pushKeys pushCrypts*

## 1.8 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*

**by** (*clarify*, *erule parts.induct*, *auto elim!*: *keyfree-KeyE keyfree-MPairE keyfree-CryptE*)

```

lemma analz-keyfree-into-Un:  $\llbracket X \in \text{analz } (G \cup H); G \subseteq \text{keyfree} \rrbracket \implies X \in \text{parts } G \cup \text{analz } H$ 
apply (erule analz.induct, auto)
apply (blast dest:parts.Body)
apply (blast dest: parts.Body)
apply (metis Un-absorb2 keyfree-KeyE parts-Un parts-keyfree UnI2)
done

```

## 1.9 Tactics useful for many protocol proofs

ML

```

<
(*Analysis of Fake cases. Also works for messages that forward unknown parts,
but this application is no longer necessary if analz-insert-eq is used.
DEPENDS UPON X REFERRING TO THE FRADULENT MESSAGE *)

```

```

fun impOfSubs th = th RSN (2, @{thm rev-subsetD})

```

```

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

```

```

fun Fake-insert-tac ctxt =
  dresolve-tac ctxt [impOfSubs @{thm Fake-analz-insert},
    impOfSubs @{thm Fake-parts-insert}] THEN'
  eresolve-tac ctxt [asm-rl, @{thm synth.Inj}];

```

```

fun Fake-insert-simp-tac ctxt i =
  REPEAT (Fake-insert-tac ctxt i) THEN asm-full-simp-tac ctxt i;

```

```

fun atomic-spy-analz-tac ctxt =
  SELECT-GOAL
  (Fake-insert-simp-tac ctxt 1 THEN
  IF-UNSOLVED
  (Blast.depth-tac
  (ctxt addIs [@{thm analz-insertI}, impOfSubs @{thm analz-subset-parts}])
  4 1));

```

```

fun spy-analz-tac ctxt i =
  DETERM
  (SELECT-GOAL
  (EVERY
  [ (*push in occurrences of X...*)
    (REPEAT o CHANGED)
    (Rule-Insts.res-inst-tac ctxt [((x, 1), Position.none), X]) []
    (@{thm insert-commute} RS ssubst) 1),
  (*...allowing further simplifications*)
  simp-tac ctxt 1,
  REPEAT (FIRSTGOAL (resolve-tac ctxt [allI, impI, notI, conjI, iffI])),
  DEPTH-SOLVE (atomic-spy-analz-tac ctxt 1)) i);

```

›

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

**declare** *o-def* [simp]

**lemma** *Crypt-notin-image-Key* [simp]:  $\text{Crypt } K \ X \notin \text{Key } A$   
**by** *auto*

**lemma** *Hash-notin-image-Key* [simp]:  $\text{Hash } X \notin \text{Key } A$   
**by** *auto*

**lemma** *synth-analz-mono*:  $G \subseteq H \implies \text{synth } (\text{analz}(G)) \subseteq \text{synth } (\text{analz}(H))$   
**by** (*iprover intro: synth-mono analz-mono*)

**lemma** *Fake-analz-eq* [simp]:  
 $X \in \text{synth}(\text{analz } H) \implies \text{synth } (\text{analz } (\text{insert } X \ H)) = \text{synth } (\text{analz } H)$   
**by** (*metis Fake-analz-insert Un-absorb Un-absorb1 Un-commute subset-insertI synth-analz-mono synth-increasing synth-subset-iff*)

Two generalizations of *analz-insert-eq*

**lemma** *gen-analz-insert-eq* [rule-format]:  
 $X \in \text{analz } H \implies \forall G. H \subseteq G \dashrightarrow \text{analz } (\text{insert } X \ G) = \text{analz } G$   
**by** (*blast intro: analz-cut analz-insertI analz-mono [THEN [2] rev-subsetD]*)

**lemma** *synth-analz-insert-eq* [rule-format]:  
 $X \in \text{synth } (\text{analz } H) \implies \forall G. H \subseteq G \dashrightarrow (\text{Key } K \in \text{analz } (\text{insert } X \ G)) = (\text{Key } K \in \text{analz } G)$   
**apply** (*erule synth.induct*)  
**apply** (*simp-all add: gen-analz-insert-eq subset-trans [OF - subset-insertI]*)  
**done**

**lemma** *Fake-parts-sing*:  
 $X \in \text{synth } (\text{analz } H) \implies \text{parts}\{X\} \subseteq \text{synth } (\text{analz } H) \cup \text{parts } H$   
**by** (*metis Fake-parts-insert empty-subsetI insert-mono parts-mono subset-trans*)

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

**method-setup** *spy-analz* = ‹  
  *Scan.succeed (SIMPLE-METHOD' o spy-analz-tac)*›  
  *for proving the Fake case when analz is involved*

**method-setup** *atomic-spy-analz* = ‹  
  *Scan.succeed (SIMPLE-METHOD' o atomic-spy-analz-tac)*›  
  *for debugging spy-analz*

**method-setup** *Fake-insert-simp* = ‹

*Scan.succeed (SIMPLE-METHOD' o Fake-insert-simp-tac)*  
*for debugging spy-analz*

**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* ∈ *bad*

*Server-not-bad* [iff]: *Server* ∉ *bad*

**by** (*rule exI* [*of* - {*Spy*}], *simp*)

**primrec** *knows* :: *agent* => *event list* => *msg set*

**where**

*knows-Nil*: *knows* *A* [] = *initState A*

| *knows-Cons*:

*knows* *A* (*ev* # *evs*) =

(*if* *A* = *Spy* then

(*case ev* of

*Says* *A' B X* => *insert X (knows Spy evs)*

| *Gets* *A' X* => *knows Spy evs*

| *Notes* *A' X* =>

*if* *A' ∈ bad* then *insert X (knows Spy evs)* else *knows Spy evs*)

else

(*case ev* of

*Says* *A' B X* =>

*if* *A'=A* then *insert X (knows A evs)* else *knows A evs*

| *Gets* *A' X* =>

*if* *A'=A* then *insert X (knows A evs)* else *knows A evs*

| *Notes* *A' X* =>

*if* *A'=A* then *insert X (knows A evs)* else *knows A evs*))

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

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

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

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

**lemma** *Notes-imp-used* [*rule-format*]: *Notes A X ∈ set evs* --> *X ∈ used evs*  
**apply** (*induct-tac evs*)  
**apply** (*auto split: event.split*)  
**done**

**lemma** *Says-imp-used* [*rule-format*]: *Says A B X ∈ set evs* --> *X ∈ used evs*  
**apply** (*induct-tac evs*)  
**apply** (*auto split: event.split*)  
**done**

## 2.1 Function *knows*

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

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

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

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

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

**lemma** *knows-Spy-subset-knows-Spy-Says*:  
*knows Spy evs ⊆ knows Spy (Says A B X # evs)*  
**by** (*simp add: subset-insertI*)

**lemma** *knows-Spy-subset-knows-Spy-Notes*:  
 $knows\ Spy\ evs \subseteq knows\ Spy\ (Notes\ A\ X\ \# \ evs)$   
**by** *force*

**lemma** *knows-Spy-subset-knows-Spy-Gets*:  
 $knows\ Spy\ evs \subseteq knows\ Spy\ (Gets\ A\ X\ \# \ evs)$   
**by** (*simp add: subset-insertI*)

Spy sees what is sent on the traffic

**lemma** *Says-imp-knows-Spy* [*rule-format*]:  
 $Says\ A\ B\ X \in\ set\ evs \longrightarrow X \in\ knows\ Spy\ evs$   
**apply** (*induct-tac evs*)  
**apply** (*simp-all (no-asm-simp) split: event.split*)  
**done**

**lemma** *Notes-imp-knows-Spy* [*rule-format*]:  
 $Notes\ A\ X \in\ set\ evs \longrightarrow A: bad \longrightarrow X \in\ knows\ Spy\ evs$   
**apply** (*induct-tac evs*)  
**apply** (*simp-all (no-asm-simp) split: event.split*)  
**done**

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

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

**lemmas** *knows-Spy-partsEs* =  
 $knows-Spy-partsEs\ parts.Body\ [THEN\ revcut-rl]$

**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*:  $knows\ A\ (Says\ A\ B\ X\ \# \ evs) = insert\ X\ (knows\ A\ evs)$   
**by** *simp*

**lemma** *knows-Notes*:  $knows\ A\ (Notes\ A\ X\ \# \ evs) = insert\ X\ (knows\ A\ evs)$   
**by** *simp*

**lemma** *knows-Gets*:

$A \neq\ Spy \longrightarrow knows\ A\ (Gets\ A\ X\ \# \ evs) = insert\ X\ (knows\ A\ evs)$   
**by** *simp*

**lemma** *knows-subset-knows-Says*:  $knows\ A\ evs \subseteq knows\ A\ (Says\ A'\ B\ X\ \# \ evs)$   
**by** (*simp add: subset-insertI*)

**lemma** *knows-subset-knows-Notes*:  $knows\ A\ evs \subseteq knows\ A\ (Notes\ A'\ X\ \# \ evs)$   
**by** (*simp add: subset-insertI*)

**lemma** *knows-subset-knows-Gets*:  $knows\ A\ evs \subseteq knows\ A\ (Gets\ A'\ X\ \# \ evs)$   
**by** (*simp add: subset-insertI*)

Agents know what they say

**lemma** *Says-imp-knows* [*rule-format*]:  $Says\ A\ B\ X \in\ set\ evs \longrightarrow X \in\ knows\ A\ evs$   
**apply** (*induct-tac evs*)  
**apply** (*simp-all (no-asm-simp) split: event.split*)  
**apply** *blast*  
**done**

Agents know what they note

**lemma** *Notes-imp-knows* [*rule-format*]:  $Notes\ A\ X \in\ set\ evs \longrightarrow X \in\ knows\ A\ evs$   
**apply** (*induct-tac evs*)  
**apply** (*simp-all (no-asm-simp) split: event.split*)  
**apply** *blast*  
**done**

Agents know what they receive

**lemma** *Gets-imp-knows-agents* [*rule-format*]:  
 $A \neq\ Spy \longrightarrow Gets\ A\ X \in\ set\ evs \longrightarrow X \in\ knows\ A\ evs$   
**apply** (*induct-tac evs*)  
**apply** (*simp-all (no-asm-simp) split: event.split*)  
**done**

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

**lemma** *knows-imp-Says-Gets-Notes-initState* [*rule-format*]:  
 $[| X \in\ knows\ A\ evs; A \neq\ Spy |] \implies \exists B.$   
 $Says\ A\ B\ X \in\ set\ evs \mid Gets\ A\ X \in\ set\ evs \mid Notes\ A\ X \in\ set\ evs \mid X \in\ initState\ A$   
**apply** (*erule rev-mp*)  
**apply** (*induct-tac evs*)  
**apply** (*simp-all (no-asm-simp) split: event.split*)  
**apply** *blast*  
**done**

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

**lemma** *knows-Spy-imp-Says-Notes-initState* [*rule-format*]:

```

    [| X ∈ knows Spy evs |] ==> ∃ A B.
      Says A B X ∈ set evs | Notes A X ∈ set evs | X ∈ initState Spy
apply (erule rev-mp)
apply (induct-tac evs)
apply (simp-all (no-asm-simp) split: event.split)
apply blast
done

lemma parts-knows-Spy-subset-used: parts (knows Spy evs) ⊆ used evs
apply (induct-tac evs, force)
apply (simp add: parts-insert-knows-A knows-Cons add: event.split, blast)
done

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

lemma initState-into-used: X ∈ parts (initState B) ==> X ∈ used evs
apply (induct-tac evs)
apply (simp-all add: parts-insert-knows-A split: event.split, blast)
done

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

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

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

lemma used-nil-subset: used [] ⊆ used evs
apply simp
apply (blast intro: initState-into-used)
done

```

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

```

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

```

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

```

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

```

```

lemma knows-subset-knows-Cons: knows A evs ⊆ knows A (e # evs)

```

by (induct e, auto simp: knows-Cons)

**lemma** *initState-subset-knows*: *initState A*  $\subseteq$  *knows A evs*  
apply (induct-tac evs, simp)  
apply (blast intro: knows-subset-knows-Cons [THEN subsetD])  
done

For proving *new-keys-not-used*

**lemma** *keysFor-parts-insert*:  
[[ *K*  $\in$  *keysFor* (*parts* (*insert X G*)); *X*  $\in$  *synth* (*analz H*) ]]  
==> *K*  $\in$  *keysFor* (*parts* (*G*  $\cup$  *H*)) | *Key* (*invKey K*)  $\in$  *parts H*  
by (force  
dest!: *parts-insert-subset-Un* [THEN *keysFor-mono*, THEN [2] *rev-subsetD*]  
analz-subset-parts [THEN *keysFor-mono*, THEN [2] *rev-subsetD*]  
intro: *analz-subset-parts* [THEN *subsetD*] *parts-mono* [THEN [2] *rev-subsetD*])

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

ML

<  
fun *analz-mono-contra-tac* ctxt =  
  *resolve-tac* ctxt @ {thms *analz-impI*} THEN'  
  REPEAT1 o (*dresolve-tac* ctxt @ {thms *analz-mono-contra*})  
  THEN' *mp-tac* ctxt  
>

**method-setup** *analz-mono-contra* = <  
  *Scan.succeed* (fn ctxt => *SIMPLE-METHOD* (*REPEAT-FIRST* (*analz-mono-contra-tac*  
  *ctxt*)))>

for proving theorems of the form *X*  $\notin$  *analz* (*knows Spy evs*)  $\dashv\vdash$  *P*

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

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

ML

<  
fun *synth-analz-mono-contra-tac* ctxt =  
  *resolve-tac* ctxt @ {thms *synt-impI*} THEN'  
  REPEAT1 o  
    (*dresolve-tac* ctxt  
      [@ {thm *knows-Spy-subset-knows-Spy-Says*} *RS* @ {thm *synth-analz-mono*} *RS*  
      @ {thm *contra-subsetD*},  
      @ {thm *knows-Spy-subset-knows-Spy-Notes*} *RS* @ {thm *synth-analz-mono*} *RS*  
      @ {thm *contra-subsetD*},  
      @ {thm *knows-Spy-subset-knows-Spy-Gets*} *RS* @ {thm *synth-analz-mono*} *RS*  
      @ {thm *contra-subsetD*])  
  THEN'

```

    mp-tac ctxt
  ›

method-setup synth-analz-mono-contr = ⟨
  Scan.succeed (fn ctxt => SIMPLE-METHOD (REPEAT-FIRST (synth-analz-mono-contr-tac
  ctxt))))⟩
  for proving theorems of the form  $X \notin \text{synth} (\text{analz} (\text{knows } \text{Spy } \text{evs})) \dashrightarrow P$ 

end

```

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

```

theory Public
imports Event
begin

```

```

lemma invKey-K:  $K \in \text{symKeys} \implies \text{invKey } K = K$ 
by (simp add: symKeys-def)

```

#### 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** (*input*)

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

**abbreviation** (*input*)

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

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

**specification** (*publicKey*)

```
injective-publicKey:
publicKey b A = publicKey c A' ==> b=c ∧ A=A'
apply (rule exI [of -
  %b A. 2 * case-agent 0 (λn. n + 2) 1 A + case-keymode 0 1 b])
apply (auto simp add: inj-on-def split: agent.split keymode.split)
apply presburger
apply presburger
done
```

**axiomatization where**

```
privateKey-neq-publicKey [iff]: privateKey b A ≠ publicKey c A'
```

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

### 3.2 Basic properties of *pubEK* and *priEK*

```
lemma publicKey-inject [iff]: (publicKey b A = publicKey c A') = (b=c ∧ A=A')
by (blast dest!: injective-publicKey)
```

```
lemma not-symKeys-pubK [iff]: publicKey b A ∉ symKeys
by (simp add: symKeys-def)
```

```
lemma not-symKeys-priK [iff]: privateKey b A ∉ symKeys
by (simp add: symKeys-def)
```

```
lemma symKey-neq-priEK: K ∈ symKeys ==> K ≠ priEK A
by auto
```

```
lemma symKeys-neq-imp-neq: (K ∈ symKeys) ≠ (K' ∈ symKeys) ==> K ≠ K'
by blast
```

```
lemma symKeys-invKey-iff [iff]: (invKey K ∈ symKeys) = (K ∈ symKeys)
by (unfold symKeys-def, auto)
```

**lemma** *analz-symKeys-Decrypt*:  

$$\llbracket \text{Crypt } K \ X \in \text{analz } H; \ K \in \text{symKeys}; \ \text{Key } K \in \text{analz } H \rrbracket$$

$$\implies X \in \text{analz } H$$
**by** (*auto simp add: symKeys-def*)

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

**lemma** *invKey-image-eq* [*simp*]:  $(\text{invKey } x \in \text{invKey } A) = (x \in A)$   
**by** *auto*

**lemma** *publicKey-image-eq* [*simp*]:  

$$(\text{publicKey } b \ x \in \text{publicKey } c \ A) = (b=c \wedge x \in A)$$
**by** *auto*

**lemma** *privateKey-notin-image-publicKey* [*simp*]:  $\text{privateKey } b \ x \notin \text{publicKey } c \ A$   
**by** *auto*

**lemma** *privateKey-image-eq* [*simp*]:  

$$(\text{privateKey } b \ A \in \text{invKey } c \ AS) = (b=c \wedge A \in AS)$$
**by** *auto*

**lemma** *publicKey-notin-image-privateKey* [*simp*]:  $\text{publicKey } b \ A \notin \text{invKey } c \ AS$   
**by** *auto*

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

**specification** (*shrK*)  
 $\text{inj-shrK}: \text{inj } \text{shrK}$   
— No two agents have the same long-term key  
**apply** (*rule exI [of - case-agent 0 (\lambda n. n + 2) 1]*)  
**apply** (*simp add: inj-on-def split: agent.split*)  
**done**

**axiomatization where**  
 $\text{sym-shrK} \text{ [iff]: } \text{shrK } X \in \text{symKeys} \text{ — All shared keys are symmetric}$

Injectiveness: Agents' long-term keys are distinct.

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

**lemma** *invKey-shrK* [*simp*]:  $\text{invKey } (\text{shrK } A) = \text{shrK } A$

**by** (*simp add: invKey-K*)

**lemma** *analz-shrK-Decrypt*:

$[[ \text{Crypt } (\text{shrK } A) \ X \in \text{analz } H; \text{Key}(\text{shrK } A) \in \text{analz } H ]] \implies X \in \text{analz } H$   
**by** *auto*

**lemma** *analz-Decrypt'*:

$[[ \text{Crypt } K \ X \in \text{analz } H; K \in \text{symKeys}; \text{Key } K \in \text{analz } H ]] \implies X \in \text{analz } H$   
**by** (*auto simp add: invKey-K*)

**lemma** *priK-neq-shrK [iff]*:  $\text{shrK } A \neq \text{privateKey } b \ C$

**by** (*simp add: symKeys-neq-imp-neq*)

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

**declare** *shrK-neq-priK [simp]*

**lemma** *pubK-neq-shrK [iff]*:  $\text{shrK } A \neq \text{publicKey } b \ C$

**by** (*simp add: symKeys-neq-imp-neq*)

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

**declare** *shrK-neq-pubK [simp]*

**lemma** *priEK-noteq-shrK [simp]*:  $\text{priEK } A \neq \text{shrK } B$

**by** *auto*

**lemma** *publicKey-notin-image-shrK [simp]*:  $\text{publicKey } b \ x \notin \text{shrK } 'AA$

**by** *auto*

**lemma** *privateKey-notin-image-shrK [simp]*:  $\text{privateKey } b \ x \notin \text{shrK } 'AA$

**by** *auto*

**lemma** *shrK-notin-image-publicKey [simp]*:  $\text{shrK } x \notin \text{publicKey } b \ 'AA$

**by** *auto*

**lemma** *shrK-notin-image-privateKey [simp]*:  $\text{shrK } x \notin \text{invKey } ' \text{publicKey } b \ 'AA$

**by** *auto*

**lemma** *shrK-image-eq [simp]*:  $(\text{shrK } x \in \text{shrK } 'AA) = (x \in AA)$

**by** *auto*

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*  $\equiv$  *initState*  
**begin**

**primrec** *initState* **where**

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

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

| *initState-Spy*:  
*initState Spy* =  
 $(Key\ 'invKey\ 'pubEK\ 'bad) \cup (Key\ 'invKey\ 'pubSK\ 'bad) \cup$   
 $(Key\ 'shrK\ 'bad) \cup$   
 $(Key\ 'range\ pubEK) \cup (Key\ 'range\ pubSK)$

**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*]:

$\forall X \in used\ evs.\ parts\ \{X\} \subseteq used\ evs$

**apply** (*induct evs*)

**prefer** 2

**apply** (*simp add: used-Cons split: event.split*)

**apply** (*metis Un-iff empty-subsetI insert-subset le-supI1 le-supI2 parts-subset-iff*)

Base case

**apply** (*auto dest!: parts-cut simp add: used-Nil*)

**done**

**lemma** *MPair-used-D*:  $\{\{X, Y\} \in used\ H \implies X \in used\ H \wedge Y \in used\ H$

**by** (*drule used-parts-subset-parts, simp, blast*)

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

**lemma** *MPair-used* [*elim!*]:

$[\{X, Y\} \in used\ H;$

$[\{X \in used\ H; Y \in used\ H\}] \implies P$

$\implies P$

**by** (*blast dest: MPair-used-D*)

Rewrites should not refer to *initState (Friend i)* because that expression is

not in normal form.

**lemma** *keysFor-parts-initState* [simp]:  $keysFor (parts (initState C)) = \{\}$   
**apply** (*unfold keysFor-def*)  
**apply** (*induct-tac C*)  
**apply** (*auto intro: range-eqI*)  
**done**

**lemma** *Crypt-notin-initState*:  $Crypt K X \notin parts (initState B)$   
**by** (*induct B, auto*)

**lemma** *Crypt-notin-used-empty* [simp]:  $Crypt K X \notin used []$   
**by** (*simp add: Crypt-notin-initState used-Nil*)

**lemma** *shrK-in-initState* [iff]:  $Key (shrK A) \in initState A$   
**by** (*induct-tac A, auto*)

**lemma** *shrK-in-knows* [iff]:  $Key (shrK A) \in knows A evs$   
**by** (*simp add: initState-subset-knows [THEN subsetD]*)

**lemma** *shrK-in-used* [iff]:  $Key (shrK A) \in used evs$   
**by** (*rule initState-into-used, blast*)

**lemma** *Key-not-used* [simp]:  $Key K \notin used evs \implies K \notin range shrK$   
**by** *blast*

**lemma** *shrK-neq*:  $Key K \notin used evs \implies shrK B \neq K$   
**by** *blast*

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

### 3.6 Function *knows Spy*

**lemma** *not-SignatureE* [elim!]:  $b \neq Signature \implies b = Encryption$   
**by** (*cases b, auto*)

Agents see their own private keys!

**lemma** *priK-in-initState* [iff]:  $Key (privateKey b A) \in initState A$   
**by** (*cases A, auto*)

Agents see all public keys!

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

**by** (*cases B, auto*)

All public keys are visible

**lemma** *spies-pubK* [*iff*]:  $Key (publicKey\ b\ A) \in spies\ evs$   
**apply** (*induct-tac evs*)  
**apply** (*auto simp add: imageI knows-Cons split: event.split*)  
**done**

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

Spy sees private keys of bad agents!

**lemma** *Spy-spies-bad-privateKey* [*intro!*]:  
 $A \in bad \implies Key (privateKey\ b\ A) \in spies\ evs$   
**apply** (*induct-tac evs*)  
**apply** (*auto simp add: imageI knows-Cons split: event.split*)  
**done**

Spy sees long-term shared keys of bad agents!

**lemma** *Spy-spies-bad-shrK* [*intro!*]:  
 $A \in bad \implies Key (shrK\ A) \in spies\ evs$   
**apply** (*induct-tac evs*)  
**apply** (*simp-all add: imageI knows-Cons split: event.split*)  
**done**

**lemma** *publicKey-into-used* [*iff*]:  $Key (publicKey\ b\ A) \in used\ evs$   
**apply** (*rule initState-into-used*)  
**apply** (*rule publicKey-in-initState [THEN parts.Inj]*)  
**done**

**lemma** *privateKey-into-used* [*iff*]:  $Key (privateKey\ b\ A) \in used\ evs$   
**apply** (*rule initState-into-used*)  
**apply** (*rule priK-in-initState [THEN parts.Inj]*)  
**done**

**lemma** *Crypt-Spy-analz-bad*:  
 $[ [ Crypt (shrK\ A)\ X \in analz (knows\ Spy\ evs); A \in bad ] ]$   
 $\implies X \in analz (knows\ Spy\ evs)$   
**by** *force*

### 3.7 Fresh Nonces

**lemma** *Nonce-notin-initState* [*iff*]:  $Nonce\ N \notin parts (initState\ B)$   
**by** (*induct-tac B, auto*)

**lemma** *Nonce-notin-used-empty* [*simp*]:  $Nonce\ N \notin used\ []$   
**by** (*simp add: used-Nil*)

### 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*:  $\exists N. \forall n. N \leq n \longrightarrow \text{Nonce } n \notin \text{used evs}$   
**apply** (*induct-tac evs*)  
**apply** (*rule-tac x = 0 in exI*)  
**apply** (*simp-all (no-asm-simp) add: used-Cons split: event.split*)  
**apply** *safe*  
**apply** (*rule msg-Nonce-supply [THEN exE], blast elim!: add-leE*)  
**done**

**lemma** *Nonce-supply1*:  $\exists N. \text{Nonce } N \notin \text{used evs}$   
**by** (*rule Nonce-supply-lemma [THEN exE], blast*)

**lemma** *Nonce-supply*:  $\text{Nonce } (\text{SOME } N. \text{Nonce } N \notin \text{used evs}) \notin \text{used evs}$   
**apply** (*rule Nonce-supply-lemma [THEN exE]*)  
**apply** (*rule someI, fast*)  
**done**

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

**lemma** *insert-Key-singleton*:  $\text{insert } (\text{Key } K) H = \text{Key } \{K\} \cup H$   
**by** *blast*

**lemma** *insert-Key-image*:  $\text{insert } (\text{Key } K) (\text{Key } KK \cup C) = \text{Key } (\text{insert } K KK) \cup C$   
**by** *blast*

**lemma** *Crypt-imp-keysFor*:  $[\text{Crypt } K X \in H; K \in \text{symKeys}] \implies K \in \text{keysFor } H$   
**by** (*drule Crypt-imp-invKey-keysFor, simp*)

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

**lemma** *analz-image-freshK-lemma*:  
 $(\text{Key } K \in \text{analz } (\text{Key } nE \cup H)) \longrightarrow (K \in nE \mid \text{Key } K \in \text{analz } H) \implies$   
 $(\text{Key } K \in \text{analz } (\text{Key } nE \cup H)) = (K \in nE \mid \text{Key } K \in \text{analz } H)$   
**by** (*blast intro: analz-mono [THEN [2] rev-subsetD]*)

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

```

ML ‹
structure Public =
struct

val analz-image-freshK-ss =
  simpset-of (@{context}
    delsimps @{thms insert-commute image-Un}
    delsimps @{thms imp-disjL} (*reduces blow-up*)
    addsimps @{thms analz-image-freshK-simps})

(*Tactic for possibility theorems*)
fun possibility-tac ctxt =
  REPEAT (*omit used-Says so that Nonces start from different traces!*)
  (ALLGOALS (simp-tac (ctxt |> Simplifier.set-unsafe-solver safe-solver |> Simplifier.del-simp @{thm used-Says})))
  THEN
  REPEAT-FIRST (eq-assume-tac ORELSE'
    resolve-tac ctxt [refl, conjI, @{thm Nonce-supply}]))

(*For harder protocols (such as Recur) where we have to set up some
nonces and keys initially*)
fun basic-possibility-tac ctxt =
  REPEAT
  (ALLGOALS (asm-simp-tac (ctxt |> Simplifier.set-unsafe-solver safe-solver))
  THEN
  REPEAT-FIRST (resolve-tac ctxt [refl, conjI]))

end
›

method-setup analz-freshK = ‹
  Scan.succeed (fn ctxt =>
    (SIMPLE-METHOD
      (EVERY [REPEAT-FIRST (resolve-tac ctxt @{thms allI ballI impI}),
        REPEAT-FIRST (resolve-tac ctxt @{thms analz-image-freshK-lemma}),
        ALLGOALS (asm-simp-tac (put-simpset Public.analz-image-freshK-ss
          ctxt))])))›
  for proving the Session Key Compromise theorem

```

### 3.10 Specialized Methods for Possibility Theorems

```

method-setup possibility = ‹
  Scan.succeed (SIMPLE-METHOD o Public.possibility-tac)›
  for proving possibility theorems

method-setup basic-possibility = ‹
  Scan.succeed (SIMPLE-METHOD o Public.basic-possibility-tac)›
  for proving possibility theorems

```

end

## 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*: [] ∈ *ns-public*

| *Fake*: [ *evsf* ∈ *ns-public*; *X* ∈ synth (analz (knows Spy *evsf*)) ]  
⇒ Says Spy *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*

declare *knows-Spy-partsEs* [elim] thm *knows-Spy-partsEs*

declare *analz-into-parts* [dest]

declare *Fake-parts-insert-in-Un* [dest]

lemma ∃ *NB*. ∃ *evs* ∈ *ns-public*. Says *A B* (Crypt (pubEK *B*) (Nonce *NB*)) ∈ set *evs*

apply (intro exI bexI)

apply (rule-tac [2] *ns-public.Nil* [THEN *ns-public.NS1*, THEN *ns-public.Reception*,

*THEN ns-public.NS2, THEN ns-public.Reception,  
THEN ns-public.NS3])*

by *possibility*

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

**lemma** *Gets-imp-Says* :

$\llbracket \text{Gets } B \ X \in \text{set } \text{evs}; \text{evs} \in \text{ns-public} \rrbracket \implies \exists A. \text{Says } A \ B \ X \in \text{set } \text{evs}$

**apply** (*erule rev-mp*)

**apply** (*erule ns-public.induct*)

**apply** *auto*

**done**

**lemma** *Gets-imp-knows-Spy*:

$\llbracket \text{Gets } B \ X \in \text{set } \text{evs}; \text{evs} \in \text{ns-public} \rrbracket \implies X \in \text{knows } \text{Spy } \text{evs}$

**apply** (*blast dest!: Gets-imp-Says Says-imp-knows-Spy*)

**done**

**lemma** *Gets-imp-knows-Spy-parts[dest]*:

$\llbracket \text{Gets } B \ X \in \text{set } \text{evs}; \text{evs} \in \text{ns-public} \rrbracket \implies X \in \text{parts } (\text{knows } \text{Spy } \text{evs})$

**apply** (*blast dest: Gets-imp-knows-Spy [THEN parts.Inj]*)

**done**

**lemma** *Spy-see-priEK [simp]*:

$\text{evs} \in \text{ns-public} \implies (\text{Key } (\text{priEK } A) \in \text{parts } (\text{knows } \text{Spy } \text{evs})) = (A \in \text{bad})$

**by** (*erule ns-public.induct, auto*)

**lemma** *Spy-analz-priEK [simp]*:

$\text{evs} \in \text{ns-public} \implies (\text{Key } (\text{priEK } A) \in \text{analz } (\text{knows } \text{Spy } \text{evs})) = (A \in \text{bad})$

**by** *auto*

**lemma** *no-nonce-NS1-NS2 [rule-format]*:

$\text{evs} \in \text{ns-public}$

$\implies \text{Crypt } (\text{pubEK } C) \ \{\!| \text{NA}', \text{Nonce } \text{NA} |\!\} \in \text{parts } (\text{knows } \text{Spy } \text{evs}) \longrightarrow$

$\text{Crypt } (\text{pubEK } B) \ \{\!| \text{Nonce } \text{NA}, \text{Agent } A |\!\} \in \text{parts } (\text{knows } \text{Spy } \text{evs}) \longrightarrow$

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

**apply** (*erule ns-public.induct, simp-all*)

**apply** (*blast intro: analz-insertI*)<sup>+</sup>

**done**

**lemma** *unique-NA*:

$\llbracket \text{Crypt}(\text{pubEK } B) \{ \text{Nonce } NA, \text{Agent } A \} \in \text{parts}(\text{knows Spy } \text{evs});$   
 $\text{Crypt}(\text{pubEK } B') \{ \text{Nonce } NA, \text{Agent } A' \} \in \text{parts}(\text{knows Spy } \text{evs});$   
 $\text{Nonce } NA \notin \text{analz}(\text{knows Spy } \text{evs}); \text{evs} \in \text{ns-public} \rrbracket$   
 $\implies A=A' \wedge B=B'$

**apply** (*erule rev-mp, erule rev-mp, erule rev-mp*)

**apply** (*erule ns-public.induct, simp-all*)

**apply** (*blast intro!: analz-insertI*)**+**

**done**

**theorem** *Spy-not-see-NA*:

$\llbracket \text{Says } A \ B (\text{Crypt}(\text{pubEK } B) \{ \text{Nonce } NA, \text{Agent } A \}) \in \text{set } \text{evs};$   
 $A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{ns-public} \rrbracket$   
 $\implies \text{Nonce } NA \notin \text{analz}(\text{knows Spy } \text{evs})$

**apply** (*erule rev-mp*)

**apply** (*erule ns-public.induct, simp-all, spy-analz*)

**apply** (*blast dest: unique-NA intro: no-nonce-NS1-NS2*)**+**

**done**

**lemma** *A-trusts-NS2-lemma* [*rule-format*]:

$\llbracket A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{ns-public} \rrbracket$   
 $\implies \text{Crypt}(\text{pubEK } A) \{ \text{Nonce } NA, \text{Nonce } NB \} \in \text{parts}(\text{knows Spy } \text{evs}) \longrightarrow$   
 $\text{Says } A \ B (\text{Crypt}(\text{pubEK } B) \{ \text{Nonce } NA, \text{Agent } A \}) \in \text{set } \text{evs} \longrightarrow$   
 $\text{Says } B \ A (\text{Crypt}(\text{pubEK } A) \{ \text{Nonce } NA, \text{Nonce } NB \}) \in \text{set } \text{evs}$

**apply** (*erule ns-public.induct*)

**apply** (*auto dest: Spy-not-see-NA unique-NA*)

**done**

**theorem** *A-trusts-NS2*:

$\llbracket \text{Says } A \ B (\text{Crypt}(\text{pubEK } B) \{ \text{Nonce } NA, \text{Agent } A \}) \in \text{set } \text{evs};$   
 $\text{Gets } A (\text{Crypt}(\text{pubEK } A) \{ \text{Nonce } NA, \text{Nonce } NB \}) \in \text{set } \text{evs};$   
 $A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{ns-public} \rrbracket$   
 $\implies \text{Says } B \ A (\text{Crypt}(\text{pubEK } A) \{ \text{Nonce } NA, \text{Nonce } NB \}) \in \text{set } \text{evs}$

**by** (*blast intro: A-trusts-NS2-lemma*)

**lemma** *B-trusts-NS1* [*rule-format*]:

$\text{evs} \in \text{ns-public}$   
 $\implies \text{Crypt}(\text{pubEK } B) \{ \text{Nonce } NA, \text{Agent } A \} \in \text{parts}(\text{knows Spy } \text{evs}) \longrightarrow$   
 $\text{Nonce } NA \notin \text{analz}(\text{knows Spy } \text{evs}) \longrightarrow$   
 $\text{Says } A \ B (\text{Crypt}(\text{pubEK } B) \{ \text{Nonce } NA, \text{Agent } A \}) \in \text{set } \text{evs}$

**apply** (*erule ns-public.induct, simp-all*)

**apply** (*blast intro!: analz-insertI*)  
**done**

**lemma** *unique-NB* [*dest*]:

$\llbracket \text{Crypt}(\text{pubEK } A) \{ \text{Nonce } NA, \text{Nonce } NB \} \in \text{parts}(\text{knows } \text{Spy } \text{evs});$   
 $\text{Crypt}(\text{pubEK } A') \{ \text{Nonce } NA', \text{Nonce } NB \} \in \text{parts}(\text{knows } \text{Spy } \text{evs});$   
 $\text{Nonce } NB \notin \text{analz}(\text{knows } \text{Spy } \text{evs}); \text{evs} \in \text{ns-public} \rrbracket$   
 $\implies A=A' \wedge NA=NA'$

**apply** (*erule rev-mp, erule rev-mp, erule rev-mp*)

**apply** (*erule ns-public.induct, simp-all*)

**apply** (*blast intro!: analz-insertI*)+  
**done**

**theorem** *Spy-not-see-NB* [*dest*]:

$\llbracket \text{Says } B \ A \ (\text{Crypt}(\text{pubEK } A) \{ \text{Nonce } NA, \text{Nonce } NB \}) \in \text{set } \text{evs};$   
 $\forall C. \text{Says } A \ C \ (\text{Crypt}(\text{pubEK } C) (\text{Nonce } NB)) \notin \text{set } \text{evs};$   
 $A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{ns-public} \rrbracket$   
 $\implies \text{Nonce } NB \notin \text{analz}(\text{knows } \text{Spy } \text{evs})$

**apply** (*erule rev-mp, erule rev-mp*)

**apply** (*erule ns-public.induct, simp-all, spy-analz*)

**apply** (*simp-all add: all-conj-distrib*)

**apply** (*blast intro: no-nonce-NS1-NS2*)+

**done**

**lemma** *B-trusts-NS3-lemma* [*rule-format*]:

$\llbracket A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{ns-public} \rrbracket$   
 $\implies \text{Crypt}(\text{pubEK } B) (\text{Nonce } NB) \in \text{parts}(\text{knows } \text{Spy } \text{evs}) \longrightarrow$   
 $\text{Says } B \ A \ (\text{Crypt}(\text{pubEK } A) \{ \text{Nonce } NA, \text{Nonce } NB \}) \in \text{set } \text{evs} \longrightarrow$   
 $(\exists C. \text{Says } A \ C \ (\text{Crypt}(\text{pubEK } C) (\text{Nonce } NB)) \in \text{set } \text{evs})$

**apply** (*erule ns-public.induct, auto*)

**by** (*blast intro: no-nonce-NS1-NS2*)+

**theorem** *B-trusts-NS3*:

$\llbracket \text{Says } B \ A \ (\text{Crypt}(\text{pubEK } A) \{ \text{Nonce } NA, \text{Nonce } NB \}) \in \text{set } \text{evs};$   
 $\text{Gets } B \ (\text{Crypt}(\text{pubEK } B) (\text{Nonce } NB)) \in \text{set } \text{evs};$   
 $A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{ns-public} \rrbracket$

$\implies \exists C. \text{Says } A \ C \ (\text{Crypt } (\text{pubEK } C) \ (\text{Nonce } NB)) \in \text{set evs}$   
**by** (*blast intro: B-trusts-NS3-lemma*)

**lemma**  $\llbracket A \notin \text{bad}; B \notin \text{bad}; \text{evs} \in \text{ns-public} \rrbracket$   
 $\implies \text{Says } B \ A \ (\text{Crypt } (\text{pubEK } A) \ \{\text{Nonce } NA, \text{Nonce } NB\}) \in \text{set evs}$   
 $\longrightarrow \text{Nonce } NB \notin \text{analz } (\text{knows } \text{Spy } \text{evs})$   
**apply** (*erule ns-public.induct, simp-all, spy-analz*)

**apply** *blast*

**apply** (*blast intro: no-nonce-NS1-NS2*)

**apply** *clarify*  
**apply** (*frule-tac A' = A in*  
*Says-imp-knows-Spy [THEN parts.Inj, THEN unique-NB], auto*)  
**apply** (*rename-tac evs3 B' C*)

This is the attack!

1.  $\bigwedge \text{evs3a } \text{evs3 } B' \ C.$   
 $\llbracket A \notin \text{bad}; B \notin \text{bad}; \text{evs3a} \in \text{ns-public};$   
 $\text{Says } \text{evs3 } B' \ (\text{Crypt } (\text{pubEK } B') \ \{\text{Nonce } C, \text{Agent } \text{evs3}\})$   
 $\in \text{set } \text{evs3a};$   
 $\text{Gets } \text{evs3 } (\text{Crypt } (\text{pubEK } \text{evs3}) \ \{\text{Nonce } C, \text{Nonce } NB\})$   
 $\in \text{set } \text{evs3a};$   
 $B' \in \text{bad};$   
 $\text{Says } B \ A \ (\text{Crypt } (\text{pubEK } A) \ \{\text{Nonce } NA, \text{Nonce } NB\})$   
 $\in \text{set } \text{evs3a};$   
 $\text{Nonce } NB \notin \text{analz } (\text{knows } \text{Spy } \text{evs3a}) \rrbracket$   
 $\implies \text{False}$

**oops**

**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*:

$evs \in ns\text{-public} \implies (Key (priEK A) \in parts (spies evs)) = (A \in bad)$

**apply** (*erule ns-public.induct, simp-all*)

**apply** (*cases A:bad*)

**thm** *ccontr*

**apply** *blast*

**apply** *clarify*

**thm** *Fake-parts-insert* [THEN subsetD]

**apply** (*drule Fake-parts-insert* [THEN subsetD], *simp*)

**apply** (*blast dest: analz-into-parts*)

**done**

**lemma** *Spy-analz-priEK*:

$evs \in ns\text{-public} \implies (Key (priEK A) \in analz (spies evs)) = (A \in bad)$

**apply** (*erule ns-public.induct, simp-all*)

**thm** *analz-image-freshK-simps*

**apply** (*simp add: analz-image-freshK-simps*)

**apply** (*cases A:bad*)

**apply** *clarify* **apply** *simp*

**apply** *blast*

**apply** *clarsimp*

**apply** (*drule Fake-analz-insert* [THEN subsetD], *simp*)

**apply** *clarsimp*

**done**

## 6.2 On dynamic secrets

**lemma** *Spy-not-see-NA*:

$\llbracket \text{Says } A \ B \ (\text{Crypt}(\text{pubEK } B) \ \{\text{Nonce } NA, \text{Agent } A\}) \in \text{set } evs;$

$A \notin bad; B \notin bad; evs \in ns\text{-public}$

$\implies \text{Nonce } NA \notin \text{analz } (spies \ evs)$

**apply** (*erule rev-mp, erule ns-public.induct*)

**apply** (*simp-all add: Spy-analz-priEK*)

**thm** *conjI*

**apply** (*rule conjI*)

**apply** *clarify*

**apply** *clarify*

**apply** (*drule Fake-analz-insert [THEN subsetD], simp*)

**apply** *simp*

**apply** (*blast dest: unique-NA analz-into-parts intro: no-nonce-NS1-NS2*)+  
**done**

**lemma** *Spy-not-see-NB*:

$\llbracket \text{Says } B \ A \ (\text{Crypt } (\text{pubEK } A) \ \{\!\{ \text{Nonce } NA, \text{Nonce } NB \}\!\}) \in \text{set } \text{evs};$

$\forall C. \ \text{Says } A \ C \ (\text{Crypt } (\text{pubEK } C) \ (\text{Nonce } NB)) \notin \text{set } \text{evs};$

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

$\implies \text{Nonce } NB \notin \text{analz } (\text{spies } \text{evs})$

**apply** (*erule rev-mp, erule rev-mp*)

**apply** (*erule ns-public.induct*)

**apply** (*simp-all add: Spy-analz-priEK*)

apply *spy-analz* is replaced here with the following list...

**apply** (*rule ccontr*)

**apply** *clarsimp*

**apply** (*erule disjE*)

**apply** *simp*

**apply** *simp*

**apply** *clarify*

**apply** (*drule Fake-analz-insert [THEN subsetD], simp*)

**apply** *simp*

...of commands!

**apply** (*simp-all add: all-conj-distrib*)

speeds up next

**apply** (*blast dest: analz-into-parts intro: no-nonce-NS1-NS2*)+

**done**

## 7 Novel study

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

**definition** *staticSecret* :: *agent*  $\Rightarrow$  *msg set* **where**

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

### 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*:

$\llbracket m \in \text{parts } (\text{knows } \text{Spy } \text{evs}); \ m \in \text{staticSecret } A \rrbracket \implies$

$A \in \text{bad} \vee$

$(\exists C \ B \ X. \ \text{Says } C \ B \ X \in \text{set } \text{evs} \wedge m \in \text{parts}\{X\}) \vee$

$(\exists C Y. \text{Notes } C Y \in \text{set evs} \wedge C \in \text{bad} \wedge m \in \text{parts}\{Y\})$   
**apply** (*erule rev-mp*)  
**apply** (*induct-tac evs*)  
**apply** *force*  
**apply** (*induct-tac a*)

Says

1.  $\bigwedge a \text{ list } x1 \ x2 \ x3.$ 

$$\begin{aligned} & \llbracket m \in \text{staticSecret } A; \\ & m \in \text{parts } (\text{knows Spy list}) \longrightarrow \\ & A \in \text{bad} \vee \\ & (\exists C B X. \text{Says } C B X \in \text{set list} \wedge m \in \text{parts } \{X\}) \vee \\ & (\exists C Y. \text{Notes } C Y \in \text{set list} \wedge C \in \text{bad} \wedge m \in \text{parts } \{Y\}) \rrbracket \\ \implies & m \in \text{parts } (\text{knows Spy } (\text{Says } x1 \ x2 \ x3 \ \# \ \text{list})) \longrightarrow \\ & A \in \text{bad} \vee \\ & (\exists C B X. \\ & \quad \text{Says } C B X \in \text{set } (\text{Says } x1 \ x2 \ x3 \ \# \ \text{list}) \wedge m \in \text{parts } \{X\}) \vee \\ & (\exists C Y. \text{Notes } C Y \in \text{set } (\text{Says } x1 \ x2 \ x3 \ \# \ \text{list}) \wedge \\ & \quad C \in \text{bad} \wedge m \in \text{parts } \{Y\}) \end{aligned}$$
2.  $\bigwedge a \text{ list } x1 \ x2.$ 

$$\begin{aligned} & \llbracket m \in \text{staticSecret } A; \\ & m \in \text{parts } (\text{knows Spy list}) \longrightarrow \\ & A \in \text{bad} \vee \\ & (\exists C B X. \text{Says } C B X \in \text{set list} \wedge m \in \text{parts } \{X\}) \vee \\ & (\exists C Y. \text{Notes } C Y \in \text{set list} \wedge C \in \text{bad} \wedge m \in \text{parts } \{Y\}) \rrbracket \\ \implies & m \in \text{parts } (\text{knows Spy } (\text{Gets } x1 \ x2 \ \# \ \text{list})) \longrightarrow \\ & A \in \text{bad} \vee \\ & (\exists C B X. \text{Says } C B X \in \text{set } (\text{Gets } x1 \ x2 \ \# \ \text{list}) \wedge m \in \text{parts } \{X\}) \vee \\ & (\exists C Y. \text{Notes } C Y \in \text{set } (\text{Gets } x1 \ x2 \ \# \ \text{list}) \wedge \\ & \quad C \in \text{bad} \wedge m \in \text{parts } \{Y\}) \end{aligned}$$
3.  $\bigwedge a \text{ list } x1 \ x2.$ 

$$\begin{aligned} & \llbracket m \in \text{staticSecret } A; \\ & m \in \text{parts } (\text{knows Spy list}) \longrightarrow \\ & A \in \text{bad} \vee \\ & (\exists C B X. \text{Says } C B X \in \text{set list} \wedge m \in \text{parts } \{X\}) \vee \\ & (\exists C Y. \text{Notes } C Y \in \text{set list} \wedge C \in \text{bad} \wedge m \in \text{parts } \{Y\}) \rrbracket \\ \implies & m \in \text{parts } (\text{knows Spy } (\text{Notes } x1 \ x2 \ \# \ \text{list})) \longrightarrow \\ & A \in \text{bad} \vee \\ & (\exists C B X. \text{Says } C B X \in \text{set } (\text{Notes } x1 \ x2 \ \# \ \text{list}) \wedge m \in \text{parts } \{X\}) \vee \\ & (\exists C Y. \text{Notes } C Y \in \text{set } (\text{Notes } x1 \ x2 \ \# \ \text{list}) \wedge \\ & \quad C \in \text{bad} \wedge m \in \text{parts } \{Y\}) \end{aligned}$$

**apply** (*rule impI*)  
**apply** *simp*

1.  $\bigwedge \text{list } x1 \ x2 \ x3.$ 

$$\llbracket m = \text{Key } (\text{priEK } A) \vee m = \text{Key } (\text{priSK } A) \vee m = \text{Key } (\text{shrK } A);$$

$$\begin{aligned}
& m \in \text{parts } (\text{knows Spy list}) \longrightarrow \\
& A \in \text{bad} \vee \\
& (\exists C B X. \text{Says } C B X \in \text{set list} \wedge m \in \text{parts } \{X\}) \vee \\
& (\exists C Y. \text{Notes } C Y \in \text{set list} \wedge C \in \text{bad} \wedge m \in \text{parts } \{Y\}); \\
& m \in \text{parts } (\text{insert } x3 \text{ (knows Spy list)}) \\
\implies & A \in \text{bad} \vee \\
& (\exists C B X. \\
& \quad (C = x1 \wedge B = x2 \wedge X = x3 \vee \text{Says } C B X \in \text{set list}) \wedge \\
& \quad m \in \text{parts } \{X\}) \vee \\
& (\exists C Y. \text{Notes } C Y \in \text{set list} \wedge C \in \text{bad} \wedge m \in \text{parts } \{Y\})
\end{aligned}$$

2.  $\bigwedge$  a list  $x1\ x2$ .

$$\begin{aligned}
& \llbracket m \in \text{staticSecret } A; \\
& m \in \text{parts } (\text{knows Spy list}) \longrightarrow \\
& A \in \text{bad} \vee \\
& (\exists C B X. \text{Says } C B X \in \text{set list} \wedge m \in \text{parts } \{X\}) \vee \\
& (\exists C Y. \text{Notes } C Y \in \text{set list} \wedge C \in \text{bad} \wedge m \in \text{parts } \{Y\}) \\
\implies & m \in \text{parts } (\text{knows Spy (Gets } x1\ x2 \text{ \# list)}) \longrightarrow \\
& A \in \text{bad} \vee \\
& (\exists C B X. \text{Says } C B X \in \text{set (Gets } x1\ x2 \text{ \# list)} \wedge m \in \text{parts } \{X\}) \vee \\
& (\exists C Y. \text{Notes } C Y \in \text{set (Gets } x1\ x2 \text{ \# list)} \wedge \\
& \quad C \in \text{bad} \wedge m \in \text{parts } \{Y\})
\end{aligned}$$

3.  $\bigwedge$  a list  $x1\ x2$ .

$$\begin{aligned}
& \llbracket m \in \text{staticSecret } A; \\
& m \in \text{parts } (\text{knows Spy list}) \longrightarrow \\
& A \in \text{bad} \vee \\
& (\exists C B X. \text{Says } C B X \in \text{set list} \wedge m \in \text{parts } \{X\}) \vee \\
& (\exists C Y. \text{Notes } C Y \in \text{set list} \wedge C \in \text{bad} \wedge m \in \text{parts } \{Y\}) \\
\implies & m \in \text{parts } (\text{knows Spy (Notes } x1\ x2 \text{ \# list)}) \longrightarrow \\
& A \in \text{bad} \vee \\
& (\exists C B X. \text{Says } C B X \in \text{set (Notes } x1\ x2 \text{ \# list)} \wedge m \in \text{parts } \{X\}) \vee \\
& (\exists C Y. \text{Notes } C Y \in \text{set (Notes } x1\ x2 \text{ \# list)} \wedge \\
& \quad C \in \text{bad} \wedge m \in \text{parts } \{Y\})
\end{aligned}$$

**apply** (drule parts-insert [THEN equalityD1, THEN subsetD])

1.  $\bigwedge$  list  $x1\ x2\ x3$ .

$$\begin{aligned}
& \llbracket m = \text{Key } (\text{priEK } A) \vee m = \text{Key } (\text{priSK } A) \vee m = \text{Key } (\text{shrK } A); \\
& m \in \text{parts } (\text{knows Spy list}) \longrightarrow \\
& A \in \text{bad} \vee \\
& (\exists C B X. \text{Says } C B X \in \text{set list} \wedge m \in \text{parts } \{X\}) \vee \\
& (\exists C Y. \text{Notes } C Y \in \text{set list} \wedge C \in \text{bad} \wedge m \in \text{parts } \{Y\}); \\
& m \in \text{parts } \{x3\} \cup \text{parts } (\text{knows Spy list}) \\
\implies & A \in \text{bad} \vee \\
& (\exists C B X. \\
& \quad (C = x1 \wedge B = x2 \wedge X = x3 \vee \text{Says } C B X \in \text{set list}) \wedge \\
& \quad m \in \text{parts } \{X\}) \vee \\
& (\exists C Y. \text{Notes } C Y \in \text{set list} \wedge C \in \text{bad} \wedge m \in \text{parts } \{Y\})
\end{aligned}$$

2.  $\bigwedge$  a list  $x1\ x2$ .

$$\llbracket m \in \text{staticSecret } A;$$

$$\begin{aligned}
& m \in \text{parts } (\text{knows Spy list}) \longrightarrow \\
& A \in \text{bad} \vee \\
& (\exists C B X. \text{Says } C B X \in \text{set list} \wedge m \in \text{parts } \{X\}) \vee \\
& (\exists C Y. \text{Notes } C Y \in \text{set list} \wedge C \in \text{bad} \wedge m \in \text{parts } \{Y\}) \\
\implies & m \in \text{parts } (\text{knows Spy } (\text{Gets } x1 x2 \# \text{list})) \longrightarrow \\
& A \in \text{bad} \vee \\
& (\exists C B X. \text{Says } C B X \in \text{set } (\text{Gets } x1 x2 \# \text{list}) \wedge m \in \text{parts } \{X\}) \vee \\
& (\exists C Y. \text{Notes } C Y \in \text{set } (\text{Gets } x1 x2 \# \text{list}) \wedge \\
& \quad C \in \text{bad} \wedge m \in \text{parts } \{Y\})
\end{aligned}$$

3.  $\wedge a \text{ list } x1 x2.$

$$\begin{aligned}
& \llbracket m \in \text{staticSecret } A; \\
& m \in \text{parts } (\text{knows Spy list}) \longrightarrow \\
& A \in \text{bad} \vee \\
& (\exists C B X. \text{Says } C B X \in \text{set list} \wedge m \in \text{parts } \{X\}) \vee \\
& (\exists C Y. \text{Notes } C Y \in \text{set list} \wedge C \in \text{bad} \wedge m \in \text{parts } \{Y\}) \\
\implies & m \in \text{parts } (\text{knows Spy } (\text{Notes } x1 x2 \# \text{list})) \longrightarrow \\
& A \in \text{bad} \vee \\
& (\exists C B X. \text{Says } C B X \in \text{set } (\text{Notes } x1 x2 \# \text{list}) \wedge m \in \text{parts } \{X\}) \vee \\
& (\exists C Y. \text{Notes } C Y \in \text{set } (\text{Notes } x1 x2 \# \text{list}) \wedge \\
& \quad C \in \text{bad} \wedge m \in \text{parts } \{Y\})
\end{aligned}$$

**apply** *blast*

Gets

**apply** *simp*

Notes

**apply** (*rule impI*)

**apply** *simp*

**apply** (*rename-tac agent msg*)

**apply** (*case-tac agent*  $\notin$  *bad*)

**apply** *simp* **apply** *blast*

**apply** *simp*

**apply** (*drule parts-insert* [*THEN equalityD1*, *THEN subsetD*])

**apply** *blast*

**done**

**lemma** *staticSecret-analz-Spy*:

$$\begin{aligned}
& \llbracket m \in \text{analz } (\text{knows Spy evs}); m \in \text{staticSecret } A \rrbracket \implies \\
& A \in \text{bad} \vee \\
& (\exists C B X. \text{Says } C B X \in \text{set evs} \wedge m \in \text{parts}\{X\}) \vee \\
& (\exists C Y. \text{Notes } C Y \in \text{set evs} \wedge C \in \text{bad} \wedge m \in \text{parts}\{Y\}) \\
& \text{by } (\text{drule } \text{analz-into-parts } [\text{THEN } \text{staticSecret-parts-Spy}])
\end{aligned}$$

**lemma** *secret-parts-Spy*:

$$\begin{aligned}
& m \in \text{parts } (\text{knows Spy evs}) \implies \\
& m \in \text{initState } \text{Spy} \vee \\
& (\exists C B X. \text{Says } C B X \in \text{set evs} \wedge m \in \text{parts}\{X\}) \vee
\end{aligned}$$

$(\exists C Y. \text{Notes } C Y \in \text{set evs} \wedge C \in \text{bad} \wedge m \in \text{parts}\{Y\})$   
**apply** (*erule rev-mp*)  
**apply** (*induct-tac evs*)  
**apply** *simp*  
**apply** (*induct-tac a*)

Says

**apply** (*rule impI*)  
**apply** (*simp del: initState-Spy*)  
**apply** (*drule parts-insert [THEN equalityD1, THEN subsetD]*)  
**apply** (*simp only: Un-iff*)  
**apply** *blast*

Gets

**apply** *simp*

Notes

**apply** (*rule impI*)  
**apply** (*simp del: initState-Spy*)  
**apply** (*rename-tac agent msg*)  
**apply** (*case-tac agent $\notin$ bad*)  
**apply** (*simp del: initState-Spy*)  
**apply** *blast*  
**apply** (*simp del: initState-Spy*)  
**apply** (*drule parts-insert [THEN equalityD1, THEN subsetD]*)  
**apply** *blast*  
**done**

**lemma** *secret-parts-Spy-converse:*

$m \in \text{initState } \text{Spy} \vee$   
 $(\exists C B X. \text{Says } C B X \in \text{set evs} \wedge m \in \text{parts}\{X\}) \vee$   
 $(\exists C Y. \text{Notes } C Y \in \text{set evs} \wedge C \in \text{bad} \wedge m \in \text{parts}\{Y\})$   
 $\implies m \in \text{parts}(\text{knows } \text{Spy } \text{evs})$   
**apply** (*erule disjE*)  
**apply** *force*  
**apply** (*erule disjE*)  
**apply** (*blast dest: Says-imp-knows-Spy [THEN parts.Inj] parts-trans*)  
**apply** (*blast dest: Notes-imp-knows-Spy [THEN parts.Inj] parts-trans*)  
**done**

**lemma** *secret-analz-Spy:*

$m \in \text{analz } (\text{knows } \text{Spy } \text{evs}) \implies$   
 $m \in \text{initState } \text{Spy} \vee$   
 $(\exists C B X. \text{Says } C B X \in \text{set evs} \wedge m \in \text{parts}\{X\}) \vee$   
 $(\exists C Y. \text{Notes } C Y \in \text{set evs} \wedge C \in \text{bad} \wedge m \in \text{parts}\{Y\})$   
**by** (*blast dest: analz-into-parts secret-parts-Spy*)

## 7.2 Protocol-dependent study

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

**lemma** *NS-Spy-see-staticSecret:*

$\llbracket m \in staticSecret A; evs \in ns-public \rrbracket \implies$   
 $m \in parts(knows Spy evs) = (A \in bad)$

**apply** (*erule ns-public.induct*)

**apply** *simp-all*

**prefer** 2

**apply** (*cases A:bad*)

**apply** *blast*

**apply** *clarify*

**apply** (*drule Fake-parts-insert [THEN subsetD], simp*)

1.  $\bigwedge evsf X.$

$\llbracket m = Key (priEK A) \vee m = Key (priSK A) \vee m = Key (shrK A);$   
 $evsf \in ns-public; A \notin bad; m \notin parts (knows Spy evsf); A \notin bad;$   
 $m \in parts (insert X (knows Spy evsf));$   
 $m \in synth (analz (knows Spy evsf)) \cup parts (knows Spy evsf) \rrbracket$   
 $\implies A \in bad$

*A total of 5 subgoals...*

**apply** (*blast dest: analz-into-parts*)

**apply** *force+*

**done**

Seeking a proof of  $\llbracket ?m \in staticSecret ?A; ?evs \in ns-public \rrbracket \implies (?m \in parts (knows Spy ?evs)) = (?A \in bad)$  using an alternative, "specialisation" strategy

**lemma** *NS-no-Notes:*

$evs \in ns-public \implies Notes A X \notin set evs$

**apply** (*erule ns-public.induct*)

**apply** *simp-all*

**done**

**lemma** *NS-staticSecret-parts-Spy-weak:*

$\llbracket m \in parts (knows Spy evs); m \in staticSecret A;$

$evs \in ns-public \rrbracket \implies A \in bad \vee$

$(\exists C B X. Says C B X \in set evs \wedge m \in parts\{X\})$

**apply** (*blast dest: staticSecret-parts-Spy NS-no-Notes*)

**done**

**lemma** *NS-Says-staticSecret:*

$\llbracket Says A B X \in set evs; m \in staticSecret C; m \in parts\{X\};$

$evs \in ns-public \rrbracket \implies A=Spy$

**apply** (*erule rev-mp*)

**apply** (*erule ns-public.induct*)  
**apply** *force+*  
**done**

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

**lemma** *staticSecret-synth-eq*:  
 $m \in\ staticSecret\ A \implies (m \in\ synth\ H) = (m \in\ H)$   
**apply** *force*  
**done**

**lemma** *NS-Says-Spy-staticSecret*:  
 $\llbracket Says\ Spy\ B\ X \in\ set\ evs; m \in\ parts\ \{X\};$   
 $m \in\ staticSecret\ A; evs \in\ ns-public \rrbracket \implies A \in\ bad$

**apply** (*drule Says-imp-knows-Spy [THEN parts.Inj]*)

1.  $\llbracket m \in\ parts\ \{X\}; m \in\ staticSecret\ A; evs \in\ ns-public;$   
 $X \in\ parts\ (knows\ Spy\ evs) \rrbracket$   
 $\implies A \in\ bad$

**apply** (*drule parts-trans, simp*)  
**apply** (*rotate-tac -1*)

1.  $\llbracket m \in\ parts\ (knows\ Spy\ evs); m \in\ staticSecret\ A; evs \in\ ns-public;$   
 $X \in\ parts\ (knows\ Spy\ evs) \rrbracket$   
 $\implies A \in\ bad$

**apply** (*erule rev-mp*)  
**apply** (*erule ns-public.induct*)  
**apply** *simp-all*  
**prefer** 2  
**apply** (*cases A:bad*)  
**apply** *blast*  
**apply** *clarsimp*  
**apply** (*drule Fake-parts-insert [THEN subsetD], simp*)  
**apply** (*blast dest: analz-into-parts*)  
**apply** *force+*  
**done**

Here's the specialised version of  $\llbracket ?m \in\ parts\ (knows\ Spy\ ?evs); ?m \in\ staticSecret\ ?A \rrbracket \implies ?A \in\ bad \vee (\exists C\ B\ X. Says\ C\ B\ X \in\ set\ ?evs \wedge ?m \in\ parts\ \{X\}) \vee (\exists C\ Y. Notes\ C\ Y \in\ set\ ?evs \wedge C \in\ bad \wedge ?m \in\ parts\ \{Y\})$

**lemma** *NS-staticSecret-parts-Spy*:  
 $\llbracket m \in\ parts\ (knows\ Spy\ evs); m \in\ staticSecret\ A;$   
 $evs \in\ ns-public \rrbracket \implies A \in\ bad$   
**apply** (*drule staticSecret-parts-Spy*)  
**apply** *assumption*

1.  $\llbracket m \in \text{staticSecret } A; \text{ evs} \in \text{ns-public};$   
 $A \in \text{bad} \vee$   
 $(\exists C B X. \text{Says } C B X \in \text{set evs} \wedge m \in \text{parts } \{X\}) \vee$   
 $(\exists C Y. \text{Notes } C Y \in \text{set evs} \wedge C \in \text{bad} \wedge m \in \text{parts } \{Y\}) \rrbracket$   
 $\implies A \in \text{bad}$

**apply** (*erule disjE*)

**apply** *assumption*  
**apply** (*erule disjE*)

**apply** (*erule exE*)<sup>+</sup>

1.  $\wedge C B X.$   
 $\llbracket m \in \text{staticSecret } A; \text{ evs} \in \text{ns-public};$   
 $\text{Says } C B X \in \text{set evs} \wedge m \in \text{parts } \{X\} \rrbracket$   
 $\implies A \in \text{bad}$
2.  $\llbracket m \in \text{staticSecret } A; \text{ evs} \in \text{ns-public};$   
 $\exists C Y. \text{Notes } C Y \in \text{set evs} \wedge C \in \text{bad} \wedge m \in \text{parts } \{Y\} \rrbracket$   
 $\implies A \in \text{bad}$

**apply** (*case-tac C=Spy*)  
**apply** (*blast dest: NS-Says-Spy-staticSecret*)  
**apply** (*blast dest: NS-Says-staticSecret*)  
**apply** (*blast dest: NS-no-Notes*)  
**done**

Concluding the specialisation proof strategy...

**lemma** *NS-Spy-see-staticSecret-spec*:  
 $\llbracket m \in \text{staticSecret } A; \text{ evs} \in \text{ns-public} \rrbracket \implies$   
 $m \in \text{parts } (\text{knows } \text{Spy } \text{evs}) = (A \in \text{bad})$

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

**apply** *safe*  
**apply** (*blast dest: NS-staticSecret-parts-Spy*)

one line proof: *force*

**apply** *simp*  
**apply** (*erule disjE*)  
**apply** *clarify*  
**apply** (*drule-tac b=Encryption and evs=evs in Spy-spies-bad-privateKey*)  
**apply** (*drule parts.Inj, assumption*)  
**apply** (*erule disjE*)  
**apply** *clarify*  
**apply** (*drule-tac b=Signature and evs=evs in Spy-spies-bad-privateKey*)  
**apply** (*drule parts.Inj, assumption*)  
**apply** *clarify*

**apply** (*drule-tac evs=evs in Spy-spies-bad-shrK*)  
**apply** (*drule parts.Inj, assumption*)  
**done**

**lemma** *NS-Spy-analz-staticSecret*:  
 $\llbracket m \in \text{staticSecret } A; \text{ evs} \in \text{ns-public} \rrbracket \implies$   
 $m \in \text{analz } (\text{knows Spy evs}) = (A \in \text{bad})$   
**apply** (*force dest: analz-into-parts NS-staticSecret-parts-Spy*)  
**done**

**lemma** *NS-staticSecret-subset-parts-knows-Spy*:  
 $\text{evs} \in \text{ns-public} \implies$   
 $\text{staticSecret } A \subseteq \text{parts } (\text{knows Spy evs}) = (A \in \text{bad})$   
**apply** (*force dest: NS-staticSecret-parts-Spy*)  
**done**

**lemma** *NS-staticSecret-subset-analz-knows-Spy*:  
 $\text{evs} \in \text{ns-public} \implies$   
 $\text{staticSecret } A \subseteq \text{analz } (\text{knows Spy evs}) = (A \in \text{bad})$   
**apply** (*force dest: analz-into-parts NS-staticSecret-parts-Spy*)  
**done**

**end**

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

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

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

**type-synonym**  
 $\text{key} = \text{nat}$

**consts**  
 $\text{all-symmetric} :: \text{bool}$  — true if all keys are symmetric  
 $\text{invKey} :: \text{key} \Rightarrow \text{key}$  — inverse of a symmetric key

**specification** (*invKey*)  
 $\text{invKey } [\text{simp}]: \text{invKey } (\text{invKey } K) = K$   
 $\text{invKey-symmetric}: \text{all-symmetric} \longrightarrow \text{invKey} = \text{id}$   
**by** (*rule exI [of - id], auto*)

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

key and vice versa

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

**datatype** — We 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* ( $\langle\langle\text{indent}=2\ \text{notation}=\langle\text{mixfix message tuple}\rangle\{-, / -\}\rangle\rangle$ )

**syntax-consts**

-*MTuple* == *MPair*

**translations**

$\{x, y, z\}$  ==  $\{x, \{y, z\}\}$   
 $\{x, y\}$  == *CONST MPair* *x y*

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

**definition** *keysFor* :: *msg set* => *key set* **where**

— Keys useful to decrypt elements of a message set  
*keysFor* *H* == *invKey* ' {*K*.  $\exists X. \text{Crypt } K X \in H$ }

## 8.1 Inductive definition of all parts of a message

**inductive-set**

*parts* :: *msg set* => *msg set*

**for** *H* :: *msg set*

**where**

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

Monotonicity

**lemma** *parts-mono*:  $G \subseteq H \implies \text{parts}(G) \subseteq \text{parts}(H)$

**apply** *auto*

**apply** (*erule parts.induct*)  
**apply** (*blast dest: parts.Fst parts.Snd parts.Body*)  
**done**

Equations hold because constructors are injective.

**lemma** *Friend-image-eq* [*simp*]: (*Friend x*  $\in$  *Friend'A*) = (*x:A*)  
**by** *auto*

**lemma** *Key-image-eq* [*simp*]: (*Key x*  $\in$  *Key'A*) = (*x* $\in$ *A*)  
**by** *auto*

**lemma** *Nonce-Key-image-eq* [*simp*]: (*Nonce x*  $\notin$  *Key'A*)  
**by** *auto*

## 8.2 Inverse of keys

**lemma** *invKey-eq* [*simp*]: (*invKey K* = *invKey K'*) = (*K=K'*)  
**by** (*metis invKey*)

## 8.3 keysFor operator

**lemma** *keysFor-empty* [*simp*]: *keysFor* {} = {}  
**by** (*unfold keysFor-def, blast*)

**lemma** *keysFor-Un* [*simp*]: *keysFor* (*H*  $\cup$  *H'*) = *keysFor H*  $\cup$  *keysFor H'*  
**by** (*unfold keysFor-def, blast*)

**lemma** *keysFor-UN* [*simp*]: *keysFor* ( $\bigcup_{i \in A} H i$ ) = ( $\bigcup_{i \in A} \textit{keysFor} (H i)$ )  
**by** (*unfold keysFor-def, blast*)

Monotonicity

**lemma** *keysFor-mono*: *G*  $\subseteq$  *H*  $\implies$  *keysFor(G)*  $\subseteq$  *keysFor(H)*  
**by** (*unfold keysFor-def, blast*)

**lemma** *keysFor-insert-Agent* [*simp*]: *keysFor* (*insert (Agent A) H*) = *keysFor H*  
**by** (*unfold keysFor-def, auto*)

**lemma** *keysFor-insert-Nonce* [*simp*]: *keysFor* (*insert (Nonce N) H*) = *keysFor H*  
**by** (*unfold keysFor-def, auto*)

**lemma** *keysFor-insert-Number* [*simp*]: *keysFor* (*insert (Number N) H*) = *keysFor H*  
**by** (*unfold keysFor-def, auto*)

**lemma** *keysFor-insert-Key* [*simp*]: *keysFor* (*insert (Key K) H*) = *keysFor H*  
**by** (*unfold keysFor-def, auto*)

**lemma** *keysFor-insert-Hash* [*simp*]: *keysFor* (*insert (Hash X) H*) = *keysFor H*  
**by** (*unfold keysFor-def, auto*)

**lemma** *keysFor-insert-MPair* [simp]:  $keysFor (insert \{X, Y\} H) = keysFor H$   
**by** (*unfold keysFor-def, auto*)

**lemma** *keysFor-insert-Crypt* [simp]:  
 $keysFor (insert (Crypt K X) H) = insert (invKey K) (keysFor H)$   
**by** (*unfold keysFor-def, auto*)

**lemma** *keysFor-image-Key* [simp]:  $keysFor (Key E) = \{\}$   
**by** (*unfold keysFor-def, auto*)

**lemma** *Crypt-imp-invKey-keysFor*:  $Crypt K X \in H \implies invKey K \in keysFor H$   
**by** (*unfold keysFor-def, blast*)

## 8.4 Inductive relation "parts"

**lemma** *MPair-parts*:  
 $[\{X, Y\} \in parts H;$   
 $[\{ X \in parts H; Y \in parts H \}] \implies P \implies P$   
**by** (*blast dest: parts.Fst parts.Snd*)

**declare** *MPair-parts* [elim!] *parts.Body* [dest!]

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

**lemma** *parts-increasing*:  $H \subseteq parts(H)$   
**by** *blast*

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

**lemma** *parts-empty* [simp]:  $parts\{\} = \{\}$   
**apply** *safe*  
**apply** (*erule parts.induct, blast+*)  
**done**

**lemma** *parts-emptyE* [elim!]:  $X \in parts\{\} \implies P$   
**by** *simp*

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

**lemma** *parts-singleton*:  $X \in parts H \implies \exists Y \in H. X \in parts \{Y\}$   
**by** (*erule parts.induct, fast+*)

### 8.4.1 Unions

**lemma** *parts-Un-subset1*:  $parts(G) \cup parts(H) \subseteq parts(G \cup H)$   
**by** (*intro Un-least parts-mono Un-upper1 Un-upper2*)

**lemma** *parts-Un-subset2*:  $parts(G \cup H) \subseteq parts(G) \cup parts(H)$   
**apply** (*rule subsetI*)  
**apply** (*erule parts.induct, blast+*)  
**done**

**lemma** *parts-Un [simp]*:  $parts(G \cup H) = parts(G) \cup parts(H)$   
**by** (*intro equalityI parts-Un-subset1 parts-Un-subset2*)

**lemma** *parts-insert*:  $parts(insert X H) = parts \{X\} \cup parts H$   
**by** (*metis insert-is-Un parts-Un*)

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

**lemma** *parts-insert2*:  
 $parts(insert X (insert Y H)) = parts \{X\} \cup parts \{Y\} \cup parts H$   
**by** (*metis Un-commute Un-empty-right Un-insert-right insert-is-Un parts-Un*)

Added to simplify arguments to parts, analz and synth.

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*:  $insert X (parts H) \subseteq parts(insert X H)$   
**by** (*blast intro: parts-mono [THEN [2] rev-subsetD]*)

## 8.4.2 Idempotence and transitivity

**lemma** *parts-partsD [dest!]*:  $X \in parts (parts H) \implies X \in parts H$   
**by** (*erule parts.induct, blast+*)

**lemma** *parts-idem [simp]*:  $parts (parts H) = parts H$   
**by** *blast*

**lemma** *parts-subset-iff [simp]*:  $(parts G \subseteq parts H) = (G \subseteq parts H)$   
**by** (*metis parts-idem parts-increasing parts-mono subset-trans*)

**lemma** *parts-trans*:  $[| X \in parts G; G \subseteq parts H |] \implies X \in parts H$   
**by** (*drule parts-mono, blast*)

Cut

**lemma** *parts-cut*:  
 $[| Y \in parts (insert X G); X \in parts H |] \implies Y \in parts (G \cup H)$   
**by** (*blast intro: parts-trans*)

**lemma** *parts-cut-eq [simp]*:  $X \in parts H \implies parts (insert X H) = parts H$   
**by** (*force dest!: parts-cut intro: parts-insertI*)

### 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*]:

$parts\ (insert\ (Agent\ agt)\ H) = insert\ (Agent\ agt)\ (parts\ H)$

**apply** (*rule parts-insert-eq-I*)

**apply** (*erule parts.induct, auto*)

**done**

**lemma** *parts-insert-Nonce* [*simp*]:

$parts\ (insert\ (Nonce\ N)\ H) = insert\ (Nonce\ N)\ (parts\ H)$

**apply** (*rule parts-insert-eq-I*)

**apply** (*erule parts.induct, auto*)

**done**

**lemma** *parts-insert-Number* [*simp*]:

$parts\ (insert\ (Number\ N)\ H) = insert\ (Number\ N)\ (parts\ H)$

**apply** (*rule parts-insert-eq-I*)

**apply** (*erule parts.induct, auto*)

**done**

**lemma** *parts-insert-Key* [*simp*]:

$parts\ (insert\ (Key\ K)\ H) = insert\ (Key\ K)\ (parts\ H)$

**apply** (*rule parts-insert-eq-I*)

**apply** (*erule parts.induct, auto*)

**done**

**lemma** *parts-insert-Hash* [*simp*]:

$parts\ (insert\ (Hash\ X)\ H) = insert\ (Hash\ X)\ (parts\ H)$

**apply** (*rule parts-insert-eq-I*)

**apply** (*erule parts.induct, auto*)

**done**

**lemma** *parts-insert-Crypt* [*simp*]:

$parts\ (insert\ (Crypt\ K\ X)\ H) = insert\ (Crypt\ K\ X)\ (parts\ (insert\ X\ H))$

**apply** (*rule equalityI*)

**apply** (*rule subsetI*)

**apply** (*erule parts.induct, auto*)

**apply** (*blast intro: parts.Body*)

**done**

**lemma** *parts-insert-MPair* [*simp*]:

$parts\ (insert\ \{\!|X, Y|\!\} H) =$   
 $insert\ \{\!|X, Y|\!\} (parts\ (insert\ X\ (insert\ Y\ H)))$

**apply** (*rule equalityI*)

**apply** (*rule subsetI*)

**apply** (*erule parts.induct, auto*)

**apply** (*blast intro: parts.Fst parts.Snd*)<sup>+</sup>

**done**

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

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

```
lemma msg-Nonce-supply:  $\exists N. \forall n. N \leq n \longrightarrow \text{Nonce } n \notin \text{parts } \{msg\}$   
apply (induct msg)  
apply (simp-all (no-asm-simp) add: exI parts-insert2)
```

Nonce case

```
apply (metis Suc-n-not-le-n)
```

MPair case: metis works out the necessary sum itself!

```
apply (metis le-trans nat-le-linear)  
done
```

## 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*] :  $X \in H \Longrightarrow X \in \text{analz } H$

| *Fst*:  $\{X, Y\} \in \text{analz } H \Longrightarrow X \in \text{analz } H$

| *Snd*:  $\{X, Y\} \in \text{analz } H \Longrightarrow Y \in \text{analz } H$

| *Decrypt* [*dest*]:

$[\text{Crypt } K X \in \text{analz } H; \text{Key}(\text{invKey } K) \in \text{analz } H] \Longrightarrow X \in \text{analz } H$

Monotonicity; Lemma 1 of Lowe's paper

```
lemma analz-mono:  $G \subseteq H \Longrightarrow \text{analz}(G) \subseteq \text{analz}(H)$ 
```

```
apply auto
```

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

```
apply (auto dest: analz.Fst analz.Snd)
```

```
done
```

Making it safe speeds up proofs

```
lemma MPair-analz [elim!]:
```

```
  [|  $\{X, Y\} \in \text{analz } H;$ 
```

```
   |  $X \in \text{analz } H; Y \in \text{analz } H$  |] ==> P
```

```
  |] ==> P
```

```
by (blast dest: analz.Fst analz.Snd)
```

**lemma** *analz-increasing*:  $H \subseteq \text{analz}(H)$   
**by** *blast*

**lemma** *analz-subset-parts*:  $\text{analz } H \subseteq \text{parts } H$   
**apply** (*rule subsetI*)  
**apply** (*erule analz.induct, blast+*)  
**done**

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

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

**lemma** *parts-analz* [*simp*]:  $\text{parts } (\text{analz } H) = \text{parts } H$   
**by** (*metis analz-increasing analz-subset-parts equalityI parts-mono parts-subset-iff*)

**lemma** *analz-parts* [*simp*]:  $\text{analz } (\text{parts } H) = \text{parts } H$   
**apply** *auto*  
**apply** (*erule analz.induct, auto*)  
**done**

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

### 8.5.1 General equational properties

**lemma** *analz-empty* [*simp*]:  $\text{analz } \{\} = \{\}$   
**apply** *safe*  
**apply** (*erule analz.induct, blast+*)  
**done**

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

**lemma** *analz-Un*:  $\text{analz}(G) \cup \text{analz}(H) \subseteq \text{analz}(G \cup H)$   
**by** (*intro Un-least analz-mono Un-upper1 Un-upper2*)

**lemma** *analz-insert*:  $\text{insert } X (\text{analz } H) \subseteq \text{analz}(\text{insert } X H)$   
**by** (*blast intro: analz-mono [THEN [2] rev-subsetD]*)

### 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*]:  
 $\text{analz } (\text{insert } (\text{Agent } \text{agt}) H) = \text{insert } (\text{Agent } \text{agt}) (\text{analz } H)$   
**apply** (*rule analz-insert-eq-I*)  
**apply** (*erule analz.induct, auto*)  
**done**

**lemma** *analz-insert-Nonce* [simp]:  
 $\text{analz} (\text{insert} (\text{Nonce } N) H) = \text{insert} (\text{Nonce } N) (\text{analz } H)$   
**apply** (rule *analz-insert-eq-I*)  
**apply** (erule *analz.induct*, auto)  
**done**

**lemma** *analz-insert-Number* [simp]:  
 $\text{analz} (\text{insert} (\text{Number } N) H) = \text{insert} (\text{Number } N) (\text{analz } H)$   
**apply** (rule *analz-insert-eq-I*)  
**apply** (erule *analz.induct*, auto)  
**done**

**lemma** *analz-insert-Hash* [simp]:  
 $\text{analz} (\text{insert} (\text{Hash } X) H) = \text{insert} (\text{Hash } X) (\text{analz } H)$   
**apply** (rule *analz-insert-eq-I*)  
**apply** (erule *analz.induct*, auto)  
**done**

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

**lemma** *analz-insert-Key* [simp]:  
 $K \notin \text{keysFor} (\text{analz } H) \implies$   
 $\text{analz} (\text{insert} (\text{Key } K) H) = \text{insert} (\text{Key } K) (\text{analz } H)$   
**apply** (unfold *keysFor-def*)  
**apply** (rule *analz-insert-eq-I*)  
**apply** (erule *analz.induct*, auto)  
**done**

**lemma** *analz-insert-MPair* [simp]:  
 $\text{analz} (\text{insert} \{\!| X, Y |\!\} H) =$   
 $\text{insert} \{\!| X, Y |\!\} (\text{analz} (\text{insert } X (\text{insert } Y H)))$   
**apply** (rule *equalityI*)  
**apply** (rule *subsetI*)  
**apply** (erule *analz.induct*, auto)  
**apply** (erule *analz.induct*)  
**apply** (blast intro: *analz.Fst analz.Snd*)  
**done**

Can pull out enCrypted message if the Key is not known

**lemma** *analz-insert-Crypt*:  
 $\text{Key} (\text{invKey } K) \notin \text{analz } H$   
 $\implies \text{analz} (\text{insert} (\text{Crypt } K X) H) = \text{insert} (\text{Crypt } K X) (\text{analz } H)$   
**apply** (rule *analz-insert-eq-I*)  
**apply** (erule *analz.induct*, auto)

**done**

**lemma** *lemma1*:  $\text{Key} (\text{invKey } K) \in \text{analz } H \implies$   
 $\text{analz} (\text{insert} (\text{Crypt } K X) H) \subseteq$   
 $\text{insert} (\text{Crypt } K X) (\text{analz} (\text{insert } X H))$

**apply** (*rule subsetI*)  
**apply** (*erule-tac x = x in analz.induct, auto*)  
**done**

**lemma** *lemma2*:  $Key (invKey K) \in analz H \implies$   
 $insert (Crypt K X) (analz (insert X H)) \subseteq$   
 $analz (insert (Crypt K X) H)$

**apply** *auto*  
**apply** (*erule-tac x = x in analz.induct, auto*)  
**apply** (*blast intro: analz-insertI analz.Decrypt*)  
**done**

**lemma** *analz-insert-Decrypt*:  
 $Key (invKey K) \in analz H \implies$   
 $analz (insert (Crypt K X) H) =$   
 $insert (Crypt K X) (analz (insert X H))$

**by** (*intro equalityI lemma1 lemma2*)

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

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

**by** (*simp add: analz-insert-Crypt analz-insert-Decrypt*)

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

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

**apply** (*rule subsetI*)  
**apply** (*erule analz.induct, auto*)  
**done**

**lemma** *analz-image-Key [simp]*:  $analz (Key'N) = Key'N$   
**apply** *auto*  
**apply** (*erule analz.induct, auto*)  
**done**

### 8.5.3 Idempotence and transitivity

**lemma** *analz-analzD [dest!]*:  $X \in analz (analz H) \implies X \in analz H$   
**by** (*erule analz.induct, blast+*)

**lemma** *analz-idem [simp]*:  $analz (analz H) = analz H$   
**by** *blast*

**lemma** *analz-subset-iff* [*simp*]:  $(\text{analz } G \subseteq \text{analz } H) = (G \subseteq \text{analz } H)$   
**by** (*metis analz-idem analz-increasing analz-mono subset-trans*)

**lemma** *analz-trans*:  $[[ X \in \text{analz } G; G \subseteq \text{analz } H ]] \implies X \in \text{analz } H$   
**by** (*drule analz-mono, blast*)

Cut; Lemma 2 of Lowe

**lemma** *analz-cut*:  $[[ Y \in \text{analz } (\text{insert } X H); X \in \text{analz } H ]] \implies Y \in \text{analz } H$   
**by** (*erule analz-trans, blast*)

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

**lemma** *analz-insert-eq*:  $X \in \text{analz } H \implies \text{analz } (\text{insert } X H) = \text{analz } H$   
**by** (*blast intro: analz-cut analz-insertI*)

A congruence rule for "analz"

**lemma** *analz-subset-cong*:  
 $[[ \text{analz } G \subseteq \text{analz } G'; \text{analz } H \subseteq \text{analz } H' ]] \implies \text{analz } (G \cup H) \subseteq \text{analz } (G' \cup H')$   
**apply** *simp*  
**apply** (*iprover intro: conjI subset-trans analz-mono Un-upper1 Un-upper2*)  
**done**

**lemma** *analz-cong*:  
 $[[ \text{analz } G = \text{analz } G'; \text{analz } H = \text{analz } H' ]] \implies \text{analz } (G \cup H) = \text{analz } (G' \cup H')$   
**by** (*intro equalityI analz-subset-cong, simp-all*)

**lemma** *analz-insert-cong*:  
 $\text{analz } H = \text{analz } H' \implies \text{analz}(\text{insert } X H) = \text{analz}(\text{insert } X H')$   
**by** (*force simp only: insert-def intro!: analz-cong*)

If there are no pairs or encryptions then analz does nothing

**lemma** *analz-trivial*:  
 $[[ \forall X Y. \{X, Y\} \notin H; \forall X K. \text{Crypt } K X \notin H ]] \implies \text{analz } H = H$   
**apply** *safe*  
**apply** (*erule analz.induct, blast+*)  
**done**

## 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]:  $X \in H \implies X \in \text{synth } H$   
| *Agent* [intro]:  $\text{Agent } agt \in \text{synth } H$   
| *Number* [intro]:  $\text{Number } n \in \text{synth } H$   
| *Hash* [intro]:  $X \in \text{synth } H \implies \text{Hash } X \in \text{synth } H$   
| *MPair* [intro]:  $[X \in \text{synth } H; Y \in \text{synth } H] \implies \{X, Y\} \in \text{synth } H$   
| *Crypt* [intro]:  $[X \in \text{synth } H; \text{Key}(K) \in H] \implies \text{Crypt } K X \in \text{synth } H$

### Monotonicity

**lemma** *synth-mono*:  $G \subseteq H \implies \text{synth}(G) \subseteq \text{synth}(H)$

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

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

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

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

**lemma** *synth-increasing*:  $H \subseteq \text{synth}(H)$

by *blast*

### 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*:  $\text{synth}(G) \cup \text{synth}(H) \subseteq \text{synth}(G \cup H)$

by (*intro Un-least synth-mono Un-upper1 Un-upper2*)

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

by (*blast intro: synth-mono [THEN [2] rev-subsetD]*)

### 8.6.2 Idempotence and transitivity

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

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

**lemma** *synth-idem*:  $\text{synth} (\text{synth } H) = \text{synth } H$

by *blast*

**lemma** *synth-subset-iff* [*simp*]:  $(\text{synth } G \subseteq \text{synth } H) = (G \subseteq \text{synth } H)$

by (*metis subset-trans synth-idem synth-increasing synth-mono*)

**lemma** *synth-trans*:  $[X \in \text{synth } G; G \subseteq \text{synth } H] \implies X \in \text{synth } H$   
**by** (*drule synth-mono, blast*)

Cut; Lemma 2 of Lowe

**lemma** *synth-cut*:  $[Y \in \text{synth } (\text{insert } X \ H); X \in \text{synth } H] \implies Y \in \text{synth } H$   
**by** (*erule synth-trans, blast*)

**lemma** *Agent-synth* [*simp*]:  $\text{Agent } A \in \text{synth } H$   
**by** *blast*

**lemma** *Number-synth* [*simp*]:  $\text{Number } n \in \text{synth } H$   
**by** *blast*

**lemma** *Nonce-synth-eq* [*simp*]:  $(\text{Nonce } N \in \text{synth } H) = (\text{Nonce } N \in H)$   
**by** *blast*

**lemma** *Key-synth-eq* [*simp*]:  $(\text{Key } K \in \text{synth } H) = (\text{Key } K \in H)$   
**by** *blast*

**lemma** *Crypt-synth-eq* [*simp*]:  
 $\text{Key } K \notin H \implies (\text{Crypt } K \ X \in \text{synth } H) = (\text{Crypt } K \ X \in H)$   
**by** *blast*

**lemma** *keysFor-synth* [*simp*]:  
 $\text{keysFor } (\text{synth } H) = \text{keysFor } H \cup \text{invKey}\{K. \text{Key } K \in H\}$   
**by** (*unfold keysFor-def, blast*)

### 8.6.3 Combinations of parts, analz and synth

**lemma** *parts-synth* [*simp*]:  $\text{parts } (\text{synth } H) = \text{parts } H \cup \text{synth } H$   
**apply** (*rule equalityI*)  
**apply** (*rule subsetI*)  
**apply** (*erule parts.induct*)  
**apply** (*blast intro: synth-increasing [THEN parts-mono, THEN subsetD]*)  
 $\text{parts.Fst parts.Snd parts.Body}+$   
**done**

**lemma** *analz-analz-Un* [*simp*]:  $\text{analz } (\text{analz } G \cup H) = \text{analz } (G \cup H)$   
**apply** (*intro equalityI analz-subset-cong*)  
**apply** *simp-all*  
**done**

**lemma** *analz-synth-Un* [*simp*]:  $\text{analz } (\text{synth } G \cup H) = \text{analz } (G \cup H) \cup \text{synth } G$   
**apply** (*rule equalityI*)  
**apply** (*rule subsetI*)  
**apply** (*erule analz.induct*)  
**prefer** 5 **apply** (*blast intro: analz-mono [THEN [2] rev-subsetD]*)  
**apply** (*blast intro: analz.Fst analz.Snd analz.Decrypt*)  
 $+$

done

**lemma** *analz-synth [simp]*:  $\text{analz} (\text{synth } H) = \text{analz } H \cup \text{synth } H$   
**by** (*metis Un-empty-right analz-synth-Un*)

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

**lemma** *parts-insert-subset-Un*:  $X \in G \implies \text{parts}(\text{insert } X H) \subseteq \text{parts } G \cup \text{parts } H$   
**by** (*metis UnCI Un-upper2 insert-subset parts-Un parts-mono*)

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

**lemma** *Fake-parts-insert*:  
 $X \in \text{synth} (\text{analz } H) \implies$   
 $\text{parts} (\text{insert } X H) \subseteq \text{synth} (\text{analz } H) \cup \text{parts } H$   
**by** (*metis Un-commute analz-increasing insert-subset parts-analz parts-mono parts-synth synth-mono synth-subset-iff*)

**lemma** *Fake-parts-insert-in-Un*:  
 $[[Z \in \text{parts} (\text{insert } X H); X: \text{synth} (\text{analz } H)]]$   
 $\implies Z \in \text{synth} (\text{analz } H) \cup \text{parts } H$   
**by** (*metis Fake-parts-insert subsetD*)

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

**lemma** *Fake-analz-insert*:  
 $X \in \text{synth} (\text{analz } G) \implies$   
 $\text{analz} (\text{insert } X H) \subseteq \text{synth} (\text{analz } G) \cup \text{analz} (G \cup H)$   
**apply** (*rule subsetI*)  
**apply** (*subgoal-tac x \in analz (synth (analz G) \cup H), force*)  
**apply** (*blast intro: analz-mono [THEN [2] rev-subsetD] analz-mono [THEN synth-mono, THEN [2] rev-subsetD]*)  
**done**

**lemma** *analz-conj-parts [simp]*:  
 $(X \in \text{analz } H \wedge X \in \text{parts } H) = (X \in \text{analz } H)$   
**by** (*blast intro: analz-subset-parts [THEN subsetD]*)

**lemma** *analz-disj-parts [simp]*:  
 $(X \in \text{analz } H \mid X \in \text{parts } H) = (X \in \text{parts } H)$   
**by** (*blast intro: analz-subset-parts [THEN subsetD]*)

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

**lemma** *MPair-synth-analz [iff]*:  
 $(\{X, Y\} \in \text{synth} (\text{analz } H)) =$   
 $(X \in \text{synth} (\text{analz } H) \wedge Y \in \text{synth} (\text{analz } H))$   
**by** *blast*

**lemma** *Crypt-synth-analz*:

$$\llbracket \text{Key } K \in \text{analz } H; \text{Key } (\text{invKey } K) \in \text{analz } H \rrbracket$$

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

**lemma** *Hash-synth-analz* [*simp*]:  

$$X \notin \text{synth } (\text{analz } H)$$

$$\implies (\text{Hash } \{X, Y\} \in \text{synth } (\text{analz } H)) = (\text{Hash } \{X, Y\} \in \text{analz } H)$$
**by** *blast*

## 8.7 HPair: a combination of Hash and MPair

### 8.7.1 Freeness

**lemma** *Agent-neq-HPair*:  $\text{Agent } A \sim = \text{Hash}[X] \ Y$   
**by** (*unfold HPair-def, simp*)

**lemma** *Nonce-neq-HPair*:  $\text{Nonce } N \sim = \text{Hash}[X] \ Y$   
**by** (*unfold HPair-def, simp*)

**lemma** *Number-neq-HPair*:  $\text{Number } N \sim = \text{Hash}[X] \ Y$   
**by** (*unfold HPair-def, simp*)

**lemma** *Key-neq-HPair*:  $\text{Key } K \sim = \text{Hash}[X] \ Y$   
**by** (*unfold HPair-def, simp*)

**lemma** *Hash-neq-HPair*:  $\text{Hash } Z \sim = \text{Hash}[X] \ Y$   
**by** (*unfold HPair-def, simp*)

**lemma** *Crypt-neq-HPair*:  $\text{Crypt } K \ X' \sim = \text{Hash}[X] \ Y$   
**by** (*unfold HPair-def, simp*)

**lemmas** *HPair-neqs* = *Agent-neq-HPair Nonce-neq-HPair Number-neq-HPair*  
*Key-neq-HPair Hash-neq-HPair Crypt-neq-HPair*

**declare** *HPair-neqs* [*iff*]  
**declare** *HPair-neqs* [*symmetric, iff*]

**lemma** *HPair-eq* [*iff*]:  $(\text{Hash}[X'] \ Y' = \text{Hash}[X] \ Y) = (X' = X \wedge Y' = Y)$   
**by** (*simp add: HPair-def*)

**lemma** *MPair-eq-HPair* [*iff*]:  

$$(\{X', Y'\} = \text{Hash}[X] \ Y) = (X' = \text{Hash}\{X, Y\} \wedge Y' = Y)$$
**by** (*simp add: HPair-def*)

**lemma** *HPair-eq-MPair* [*iff*]:  

$$(\text{Hash}[X] \ Y = \{X', Y'\}) = (X' = \text{Hash}\{X, Y\} \wedge Y' = Y)$$
**by** (*auto simp add: HPair-def*)

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

**lemma** *keysFor-insert-HPair* [simp]:  $keysFor (insert (Hash[X] Y) H) = keysFor H$   
**by** (simp add: HPair-def)

**lemma** *parts-insert-HPair* [simp]:  
 $parts (insert (Hash[X] Y) H) =$   
 $insert (Hash[X] Y) (insert (Hash\{X, Y\}) (parts (insert Y H)))$   
**by** (simp add: HPair-def)

**lemma** *analz-insert-HPair* [simp]:  
 $analz (insert (Hash[X] Y) H) =$   
 $insert (Hash[X] Y) (insert (Hash\{X, Y\}) (analz (insert Y H)))$   
**by** (simp add: HPair-def)

**lemma** *HPair-synth-analz* [simp]:  
 $X \notin synth (analz H)$   
 $\implies (Hash[X] Y \in synth (analz H)) =$   
 $(Hash\{X, Y\} \in analz H \wedge Y \in synth (analz H))$   
**by** (auto simp add: HPair-def)

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

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

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

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

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

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

**lemmas** *pushes* = *pushKeys pushCrypts*

## 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*

**by** (*clarify*, *erule parts.induct*, *auto elim!*: *keyfree-KeyE keyfree-MPairE keyfree-CryptE*)

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

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

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

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

**apply** (*metis Un-absorb2 keyfree-KeyE parts-Un parts-keyfree UnI2*)

**done**

## 8.9 Tactics useful for many protocol proofs

**ML**

<

(\**Analysis of Fake cases. Also works for messages that forward unknown parts, but this application is no longer necessary if analz-insert-eq is used.*

*DEPENDS UPON X REFERRING TO THE FRADULENT MESSAGE* \*)

*fun impOfSubs th = th RSN (2, @{thm rev-subsetD})*

(\**Apply rules to break down assumptions of the form*

*Y* ∈ *parts(insert X H)* and *Y* ∈ *analz(insert X H)*

*\*)*

*fun Fake-insert-tac ctxt =*

*dresolve-tac ctxt [impOfSubs @{thm Fake-analz-insert},*

*impOfSubs @{thm Fake-parts-insert}] THEN'*

*eresolve-tac ctxt [asm-rl, @{thm synth.Inj}];*

*fun Fake-insert-simp-tac ctxt i =*

```

    REPEAT (Fake-insert-tac ctxt i) THEN asm-full-simp-tac ctxt i;

fun atomic-spy-analz-tac ctxt =
  SELECT-GOAL
  (Fake-insert-simp-tac ctxt 1 THEN
   IF-UNSOLVED
   (Blast.depth-tac
    (ctxt addIs [@{thm analz-insertI}, impOfSubs @{thm analz-subset-parts}])
    4 1));

fun spy-analz-tac ctxt i =
  DETERM
  (SELECT-GOAL
   (EVERY
    [ (*push in occurrences of X...*)
      (REPEAT o CHANGED)
        (Rule-Insts.res-inst-tac ctxt [((x, 1), Position.none), X] []
         (@{thm insert-commute} RS ssubst) 1),
        (*...allowing further simplifications*)
        simp-tac ctxt 1,
        REPEAT (FIRSTGOAL (resolve-tac ctxt [allI,impI,notI,conjI,iffI])),
        DEPTH-SOLVE (atomic-spy-analz-tac ctxt 1)]) i);
  >

```

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

**declare** *o-def* [*simp*]

**lemma** *Crypt-notin-image-Key* [*simp*]: *Crypt K X*  $\notin$  *Key 'A*  
**by** *auto*

**lemma** *Hash-notin-image-Key* [*simp*]: *Hash X*  $\notin$  *Key 'A*  
**by** *auto*

**lemma** *synth-analz-mono*:  $G \subseteq H \implies \text{synth}(\text{analz}(G)) \subseteq \text{synth}(\text{analz}(H))$   
**by** (*iprover intro: synth-mono analz-mono*)

**lemma** *Fake-analz-eq* [*simp*]:  
 $X \in \text{synth}(\text{analz } H) \implies \text{synth}(\text{analz}(\text{insert } X H)) = \text{synth}(\text{analz } H)$   
**by** (*metis Fake-analz-insert Un-absorb Un-absorb1 Un-commute*  
*subset-insertI synth-analz-mono synth-increasing synth-subset-iff*)

Two generalizations of *analz-insert-eq*

**lemma** *gen-analz-insert-eq* [*rule-format*]:  
 $X \in \text{analz } H \implies \forall G. H \subseteq G \longrightarrow \text{analz}(\text{insert } X G) = \text{analz } G$   
**by** (*blast intro: analz-cut analz-insertI analz-mono [THEN [2] rev-subsetD]*)

**lemma** *synth-analz-insert-eq* [rule-format]:  
 $X \in \text{synth } (\text{analz } H)$   
 $\implies \forall G. H \subseteq G \longrightarrow (\text{Key } K \in \text{analz } (\text{insert } X G)) = (\text{Key } K \in \text{analz } G)$   
**apply** (*erule synth.induct*)  
**apply** (*simp-all add: gen-analz-insert-eq subset-trans [OF - subset-insertI]*)  
**done**

**lemma** *Fake-parts-sing*:  
 $X \in \text{synth } (\text{analz } H) \implies \text{parts}\{X\} \subseteq \text{synth } (\text{analz } H) \cup \text{parts } H$   
**by** (*metis Fake-parts-insert empty-subsetI insert-mono parts-mono subset-trans*)

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

**method-setup** *spy-analz* =  $\langle$   
*Scan.succeed (SIMPLE-METHOD' o spy-analz-tac)*  
*for proving the Fake case when analz is involved*  
 $\rangle$

**method-setup** *atomic-spy-analz* =  $\langle$   
*Scan.succeed (SIMPLE-METHOD' o atomic-spy-analz-tac)*  
*for debugging spy-analz*  
 $\rangle$

**method-setup** *Fake-insert-simp* =  $\langle$   
*Scan.succeed (SIMPLE-METHOD' o Fake-insert-simp-tac)*  
*for debugging spy-analz*  
 $\rangle$

**end**

## 9 Theory of Events for Security Protocols against the General Attacker

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

**consts**  
*initState* :: *agent*  $\Rightarrow$  *msg set*

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

**primrec** *knows* :: *agent*  $\Rightarrow$  *event list*  $\Rightarrow$  *msg set* **where**  
*knows-Nil*: *knows* *A* [] = *initState A*  
| *knows-Cons*:  
*knows* *A* (*ev* # *evs*) =  
(case *ev* of  
*Says A' B X*  $\Rightarrow$  *insert X (knows A evs)*  
| *Gets A' X*  $\Rightarrow$  *knows A evs*  
| *Notes A' X*  $\Rightarrow$

*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
apply (induct-tac evs)
apply (auto split: event.split)
done

```

```

lemma Says-imp-used [rule-format]: Says A B X ∈ set evs ⟶ X ∈ used evs
apply (induct-tac evs)
apply (auto split: event.split)
done

```

## 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)
by simp

```

```

lemma knows-Notes [simp]:
  knows A (Notes A' X # evs) =
    (if A=A' then insert X (knows A evs) else knows A evs)
by simp

```

```

lemma knows-Gets [simp]: knows A (Gets A' X # evs) = knows A evs
by simp

```

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)
apply (induct-tac evs)
apply (simp-all (no-asm-simp) split: event.split)
apply auto
done

```

```

lemma Notes-imp-knows [rule-format]:

```

*Notes*  $A' X \in \text{set evs} \longrightarrow X \in \text{knows } A' \text{ evs}$   
**apply** (*induct-tac evs*)  
**apply** (*simp-all (no-asm-simp) split: event.split*)  
**done**

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)*  
**by** (*simp add: subset-insertI*)

**lemma** *knows-subset-knows-Notes: knows A evs  $\subseteq$  knows A (Notes A' X # evs)*  
**by** (*simp add: subset-insertI*)

**lemma** *knows-subset-knows-Gets: knows A evs  $\subseteq$  knows A (Gets A' X # evs)*  
**by** (*simp add: subset-insertI*)

**lemma** *knows-imp-Says-Gets-Notes-initState [rule-format]:*  
 $X \in \text{knows } A \text{ evs} \implies \exists A' B.$   
 $\text{Says } A' B X \in \text{set evs} \vee \text{Notes } A X \in \text{set evs} \vee X \in \text{initState } A$   
**apply** (*erule rev-mp*)  
**apply** (*induct-tac evs*)  
**apply** (*simp-all (no-asm-simp) split: event.split*)  
**apply** *auto*  
**done**

**lemma** *parts-knows-subset-used: parts (knows A evs)  $\subseteq$  used evs*  
**apply** (*induct-tac evs, force*)  
**apply** (*simp add: parts-insert-knows-A add: event.split, blast*)  
**done**

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

**lemma** *initState-into-used: X  $\in$  parts (initState B)  $\implies$  X  $\in$  used evs*  
**apply** (*induct-tac evs*)  
**apply** (*simp-all add: parts-insert-knows-A split: event.split, blast*)  
**done**

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

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

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

**lemma** *used-nil-subset*:  $used [] \subseteq used\ evs$   
**apply** *simp*  
**apply** (*blast intro: initState-into-used*)  
**done**

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

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

**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*:  $knows A evs \subseteq knows A (e \# evs)$   
**by** (*induct e, auto simp: knows-Cons*)

**lemma** *initState-subset-knows*:  $initState A \subseteq knows A evs$   
**apply** (*induct-tac evs, simp*)  
**apply** (*blast intro: knows-subset-knows-Cons [THEN subsetD]*)  
**done**

For proving *new-keys-not-used*

**lemma** *keysFor-parts-insert*:  
 $[| K \in keysFor (parts (insert X G)); X \in synth (analz H) |]$   
 $==> K \in keysFor (parts (G \cup H)) \mid Key (invKey K) \in parts H$   
**by** (*force*  
*dest!*: *parts-insert-subset-Un* [*THEN keysFor-mono, THEN [2] rev-subsetD*]  
*analz-subset-parts* [*THEN keysFor-mono, THEN [2] rev-subsetD*]  
*intro: analz-subset-parts* [*THEN subsetD*] *parts-mono* [*THEN [2] rev-subsetD*])

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

**ML**

<  
*fun* *analz-mono-contra-tac* *ctxt* =  
*resolve-tac* *ctxt* @{*thms analz-impI*} *THEN'*  
*REPEAT1 o (dresolve-tac* *ctxt* @{*thms analz-mono-contra*})  
*THEN'* *mp-tac* *ctxt*

>

```
method-setup analz-mono-contra = ‹  
  Scan.succeed (fn ctxt => SIMPLE-METHOD (REPEAT-FIRST (analz-mono-contra-tac  
  ctxt)))›  
  for proving theorems of the form  $X \notin \text{analz}(\text{knows } A \text{ evs}) \longrightarrow P$ 
```

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

```
lemmas syant-impI = impI [where  $P = Y \notin \text{synth}(\text{analz}(\text{knows } A \text{ evs}))$ ] for  $Y$   
 $A \text{ evs}$ 
```

**ML**

<

```
fun synth-analz-mono-contra-tac ctxt =  
  resolve-tac ctxt @ {thms syant-impI} THEN'  
  REPEAT1 o  
    (dresolve-tac ctxt  
     [ @ {thm knows-subset-knows-Says} RS @ {thm synth-analz-mono} RS @ {thm  
     contra-subsetD},  
       @ {thm knows-subset-knows-Notes} RS @ {thm synth-analz-mono} RS @ {thm  
     contra-subsetD},  
       @ {thm knows-subset-knows-Gets} RS @ {thm synth-analz-mono} RS @ {thm  
     contra-subsetD}] )  
  THEN'  
  mp-tac ctxt  
>
```

```
method-setup synth-analz-mono-contra = ‹  
  Scan.succeed (fn ctxt => SIMPLE-METHOD (REPEAT-FIRST (synth-analz-mono-contra-tac  
  ctxt)))›  
  for proving theorems of the form  $X \notin \text{synth}(\text{analz}(\text{knows } A \text{ evs})) \longrightarrow P$ 
```

**end**

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

```
theory PublicGA imports EventGA begin
```

```
lemma invKey-K:  $K \in \text{symKeys} \implies \text{invKey } K = K$   
by (simp add: symKeys-def)
```

### 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** (*input*)

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

**abbreviation** (*input*)

```
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'  $\implies$  b=c  $\wedge$  A=A'
```

**apply** (*rule exI* [*of* -

```
%b A. 2 * case-agent ( $\lambda n. n + 2$ ) A + case-keymode 0 1 b])
```

**apply** (*auto simp add: inj-on-def split: agent.split keymode.split*)

**apply** *presburger*

**apply** *presburger*

**done**

**axiomatization where**

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

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

## 10.2 Basic properties of *pubEK* and *priEK*

**lemma** *publicKey-inject* [iff]:  $(\text{publicKey } b \ A = \text{publicKey } c \ A') = (b=c \wedge A=A')$   
**by** (*blast dest!: injective-publicKey*)

**lemma** *not-symKeys-pubK* [iff]:  $\text{publicKey } b \ A \notin \text{symKeys}$   
**by** (*simp add: symKeys-def*)

**lemma** *not-symKeys-priK* [iff]:  $\text{privateKey } b \ A \notin \text{symKeys}$   
**by** (*simp add: symKeys-def*)

**lemma** *symKey-neq-priEK*:  $K \in \text{symKeys} \implies K \neq \text{priEK } A$   
**by** *auto*

**lemma** *symKeys-neq-imp-neq*:  $(K \in \text{symKeys}) \neq (K' \in \text{symKeys}) \implies K \neq K'$   
**by** *blast*

**lemma** *symKeys-invKey-iff* [iff]:  $(\text{invKey } K \in \text{symKeys}) = (K \in \text{symKeys})$   
**by** (*unfold symKeys-def, auto*)

**lemma** *analz-symKeys-Decrypt*:  
 $[\text{Crypt } K \ X \in \text{analz } H; K \in \text{symKeys}; \text{Key } K \in \text{analz } H] \implies X \in \text{analz } H$   
**by** (*auto simp add: symKeys-def*)

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

**lemma** *invKey-image-eq* [simp]:  $(\text{invKey } x \in \text{invKey } A) = (x \in A)$   
**by** *auto*

**lemma** *publicKey-image-eq* [simp]:  
 $(\text{publicKey } b \ x \in \text{publicKey } c \ A) = (b=c \wedge x \in A)$   
**by** *auto*

**lemma** *privateKey-notin-image-publicKey* [simp]:  $\text{privateKey } b \ x \notin \text{publicKey } c \ A$   
**by** *auto*

**lemma** *privateKey-image-eq* [simp]:  
 $(\text{privateKey } b \ A \in \text{invKey } c \ AS) = (b=c \wedge A \in AS)$   
**by** *auto*

**lemma** *publicKey-notin-image-privateKey* [simp]:  $\text{publicKey } b \ A \notin \text{invKey } c \ AS$   
**by** *auto*

## 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

**apply** (*rule exI [of - case-agent ( $\lambda n. n + 2$ )]*)

**apply** (*simp add: inj-on-def split: agent.split*)

**done**

**axiomatization where**

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

Injectiveness: Agents' long-term keys are distinct.

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

**declare** *shrK-injective [iff]*

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

**by** (*simp add: invKey-K*)

**lemma** *analz-shrK-Decrypt*:

$[| \text{Crypt } (\text{shrK } A) \ X \in \text{analz } H; \text{Key}(\text{shrK } A) \in \text{analz } H |] \implies X \in \text{analz } H$

**by** *auto*

**lemma** *analz-Decrypt'*:

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

*H*

**by** (*auto simp add: invKey-K*)

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

**by** (*simp add: symKeys-neq-imp-neq*)

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

**declare** *shrK-neq-priK [simp]*

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

**by** (*simp add: symKeys-neq-imp-neq*)

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

**declare** *shrK-neq-pubK [simp]*

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

**by** *auto*

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

by auto

**lemma** *privateKey-notin-image-shrK* [simp]: *privateKey* *b*  $x \notin \text{shrK } 'AA$   
by auto

**lemma** *shrK-notin-image-publicKey* [simp]: *shrK*  $x \notin \text{publicKey } b 'AA$   
by auto

**lemma** *shrK-notin-image-privateKey* [simp]: *shrK*  $x \notin \text{invKey } 'publicKey } b 'AA$   
by auto

**lemma** *shrK-image-eq* [simp]:  $(\text{shrK } x \in \text{shrK } 'AA) = (x \in AA)$   
by auto

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

**declare** *invKey-K* [simp]

## 10.5 Initial States of Agents

**overloading**

*initState*  $\equiv$  *initState*

**begin**

**primrec** *initState* **where**

*initState-Friend*:

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

**end**

**lemma** *used-parts-subset-parts* [rule-format]:

$\forall X \in \text{used evs. parts } \{X\} \subseteq \text{used evs}$

**apply** (*induct evs*)

**prefer** 2

**apply** (*simp add: used-Cons split: event.split*)

**apply** (*metis Un-iff empty-subsetI insert-subset le-supI1 le-supI2 parts-subset-iff*)

Base case

**apply** (*auto dest!: parts-cut simp add: used-Nil*)

**done**

**lemma** *MPair-used-D*:  $\{X, Y\} \in \text{used } H \implies X \in \text{used } H \wedge Y \in \text{used } H$

**by** (*drule used-parts-subset-parts, simp, blast*)

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

**lemma** *MPair-used* [elim!]:

```

    [| {X, Y} ∈ used H;
      [| X ∈ used H; Y ∈ used H |] ==> P |]
    ==> P
  by (blast dest: MPair-used-D)

```

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

```

lemma keysFor-parts-initState [simp]: keysFor (parts (initState C)) = {}
apply (unfold keysFor-def)
apply (induct-tac C)
apply (auto intro: range-eqI)
done

```

```

lemma Crypt-notin-initState: Crypt K X ∉ parts (initState B)
by (induct B, auto)

```

```

lemma Crypt-notin-used-empty [simp]: Crypt K X ∉ used []
by (simp add: Crypt-notin-initState used-Nil)

```

```

lemma shrK-in-initState [iff]: Key (shrK A) ∈ initState A
by (induct-tac A, auto)

```

```

lemma shrK-in-knows [iff]: Key (shrK A) ∈ knows A evs
by (simp add: initState-subset-knows [THEN subsetD])

```

```

lemma shrK-in-used [iff]: Key (shrK A) ∈ used evs
by (rule initState-into-used, blast)

```

```

lemma Key-not-used [simp]: Key K ∉ used evs ==> K ∉ range shrK
by blast

```

```

lemma shrK-neq: Key K ∉ used evs ==> shrK B ≠ K
by blast

```

```

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

```

## 10.6 Function *knows Spy*

```

lemma not-SignatureE [elim!]: b ≠ Signature ==> b = Encryption
by (cases b, auto)

```

Agents see their own private keys!

**lemma** *priK-in-initState* [iff]:  $Key (privateKey\ b\ A) \in initState\ A$   
**by** (*cases A, auto*)

Agents see all public keys!

**lemma** *publicKey-in-initState* [iff]:  $Key (publicKey\ b\ A) \in initState\ B$   
**by** (*cases B, auto*)

All public keys are visible

**lemma** *spies-pubK* [iff]:  $Key (publicKey\ b\ A) \in knows\ B\ evs$   
**apply** (*induct-tac evs*)  
**apply** (*auto simp add: imageI knows-Cons split: event.split*)  
**done**

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

**lemma** *publicKey-into-used* [iff] :  $Key (publicKey\ b\ A) \in used\ evs$   
**apply** (*rule initState-into-used*)  
**apply** (*rule publicKey-in-initState [THEN parts.Inj]*)  
**done**

**lemma** *privateKey-into-used* [iff]:  $Key (privateKey\ b\ A) \in used\ evs$   
**apply**(*rule initState-into-used*)  
**apply**(*rule priK-in-initState [THEN parts.Inj]*)  
**done**

**lemma** *Crypt-analz-bad*:  
 $[[\ Crypt\ (shrK\ A)\ X \in analz\ (knows\ A\ evs) ]]$   
 $==> X \in analz\ (knows\ A\ evs)$   
**by** *force*

## 10.7 Fresh Nonces

**lemma** *Nonce-notin-initState* [iff]:  $Nonce\ N \notin parts\ (initState\ B)$   
**by** (*induct-tac B, auto*)

**lemma** *Nonce-notin-used-empty* [simp]:  $Nonce\ N \notin used\ []$   
**by** (*simp add: used-Nil*)

## 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*:  $\exists N. \forall n. N \leq n \longrightarrow Nonce\ n \notin used\ evs$   
**apply** (*induct-tac evs*)  
**apply** (*rule-tac x = 0 in exI*)

```

apply (simp-all (no-asm-simp) add: used-Cons split: event.split)
apply safe
apply (rule msg-Nonce-supply [THEN exE], blast elim!: add-leE)
done

```

```

lemma Nonce-supply1:  $\exists N. \text{Nonce } N \notin \text{used evs}$ 
by (rule Nonce-supply-lemma [THEN exE], blast)

```

```

lemma Nonce-supply: Nonce (SOME N. Nonce N  $\notin$  used evs)  $\notin$  used evs
apply (rule Nonce-supply-lemma [THEN exE])
apply (rule someI, fast)
done

```

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

```

lemma insert-Key-singleton: insert (Key K) H = Key ' {K}  $\cup$  H
by blast

```

```

lemma insert-Key-image: insert (Key K) (Key'KK  $\cup$  C) = Key ' (insert K KK)
 $\cup$  C
by blast

```

```

lemma Crypt-imp-keysFor:  $[[\text{Crypt } K \ X \in \ H; \ K \in \ \text{symKeys}]] \implies K \in \ \text{keysFor } H$ 
by (drule Crypt-imp-invKey-keysFor, simp)

```

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

```

lemma analz-image-freshK-lemma:
  (Key K  $\in$  analz (Key'nE  $\cup$  H))  $\longrightarrow$  (K  $\in$  nE | Key K  $\in$  analz H)  $\implies$ 
  (Key K  $\in$  analz (Key'nE  $\cup$  H)) = (K  $\in$  nE | Key K  $\in$  analz H)
by (blast intro: analz-mono [THEN [?] rev-subsetD])

```

```

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

```

```

ML <
structure Public =
struct

```

```

val analz-image-freshK-ss =
  simpset-of (@{context})

```

```

delsimps @{thms image-insert image-Un}
delsimps @{thms imp-disjL} (*reduces blow-up*)
addsimps @{thms analz-image-freshK-simps}

(*Tactic for possibility theorems*)
fun possibility-tac ctxt =
  REPEAT (*omit used-Says so that Nonces start from different traces!*)
  (ALLGOALS (simp-tac (ctxt delsimps [@{thm used-Says}])))
  THEN
  REPEAT-FIRST (eq-assume-tac ORELSE'
    resolve-tac ctxt [refl, conjI, @{thm Nonce-supply}]))

(*For harder protocols (such as Recur) where we have to set up some
nonces and keys initially*)
fun basic-possibility-tac ctxt =
  REPEAT
  (ALLGOALS (asm-simp-tac (ctxt |> Simplifier.set-unsafe-solver safe-solver))
  THEN
  REPEAT-FIRST (resolve-tac ctxt [refl, conjI]))

end
>

method-setup analz-freshK = <
  Scan.succeed (fn ctxt =>
    (SIMPLE-METHOD
      (EVERY [REPEAT-FIRST (resolve-tac ctxt @{thms allI ballI impI}],
        REPEAT-FIRST (resolve-tac ctxt @{thms analz-image-freshK-lemma}),
        ALLGOALS (asm-simp-tac (put-simpset Public.analz-image-freshK-ss
          ctxt))))))>
  for proving the Session Key Compromise theorem

```

## 10.10 Specialized Methods for Possibility Theorems

```

method-setup possibility = <
  Scan.succeed (SIMPLE-METHOD o Public.possibility-tac)>
  for proving possibility theorems

method-setup basic-possibility = <
  Scan.succeed (SIMPLE-METHOD o Public.basic-possibility-tac)>
  for proving possibility theorems

```

end

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

| *Fake*:  $\llbracket evsf \in ns\text{-}public; X \in synth (analz (knows A evsf)) \rrbracket$   
 $\implies Says A B X \# evsf \in ns\text{-}public$

| *Reception*:  $\llbracket evsr \in ns\text{-}public; Says A B X \in set evsr \rrbracket$   
 $\implies Gets B X \# evsr \in ns\text{-}public$

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

| *NS2*:  $\llbracket evs2 \in ns\text{-}public; Nonce NB \notin used evs2;$   
 $Gets B (Crypt (pubEK B) \{Nonce NA, Agent A\}) \in set evs2 \rrbracket$   
 $\implies Says B A (Crypt (pubEK A) \{Nonce NA, Nonce NB\})$   
 $\# evs2 \in ns\text{-}public$

| *NS3*:  $\llbracket evs3 \in ns\text{-}public;$   
 $Says A B (Crypt (pubEK B) \{Nonce NA, Agent A\}) \in set evs3;$   
 $Gets A (Crypt (pubEK A) \{Nonce NA, Nonce NB\}) \in set evs3 \rrbracket$   
 $\implies Says A B (Crypt (pubEK B) (Nonce NB)) \# evs3 \in ns\text{-}public$

**lemma** *NS-no-Notes*:

$evs \in ns\text{-}public \implies Notes A X \notin set evs$

**apply** (*erule ns-public.induct*)

**apply** (*simp-all*)

**done**

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*:  $parts(initState C) = initState C$

**by** (*cases C, simp*)

**lemma** *analz-initState*:  $analz(initState C) = initState C$

**by** (*cases C, force dest: analz-into-parts*)

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

**definition** *staticSecret* :: agent  $\Rightarrow$  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  $\subseteq$  *initState* A  
**by** (cases A, simp)  
**thm** parts-insert

**lemma** *staticSecretA-notin-initStateB*:  
 $m \in \text{staticSecret } A \implies m \in \text{initState } B = (A=B)$   
**by** (cases B, auto)

**lemma** *staticSecretA-notin-parts-initStateB*:  
 $m \in \text{staticSecret } A \implies m \in \text{parts}(\text{initState } B) = (A=B)$   
**by** (cases B, auto)

**lemma** *staticSecretA-notin-analz-initStateB*:  
 $m \in \text{staticSecret } A \implies m \in \text{analz}(\text{initState } B) = (A=B)$   
**by** (cases B, force dest: analz-into-parts)

**lemma** *staticSecret-synth-eq*:  
 $m \in \text{staticSecret } A \implies (m \in \text{synth } H) = (m \in H)$   
**apply** force  
**done**

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

**lemma** *nonce-notin-analz-initState*:  
 Nonce N  $\notin$  *analz*(*initState* A)  
**by**(cases A, auto dest: analz-into-parts)

## 12.1 Protocol independent study

**lemma** *staticSecret-parts-agent*:  
 $\llbracket m \in \text{parts} (\text{knows } C \text{ evs}); m \in \text{staticSecret } A \rrbracket \implies$   
 $A=C \vee$   
 $(\exists D E X. \text{Says } D E X \in \text{set evs} \wedge m \in \text{parts}\{X\}) \vee$   
 $(\exists Y. \text{Notes } C Y \in \text{set evs} \wedge m \in \text{parts}\{Y\})$   
**apply** (erule rev-mp)  
**apply** (induct-tac evs)

1.  $m \in \text{staticSecret } A \implies$   
 $m \in \text{parts} (\text{knows } C []) \longrightarrow$   
 $A = C \vee$

$(\exists D E X. \text{Says } D E X \in \text{set } [] \wedge m \in \text{parts } \{X\}) \vee$   
 $(\exists Y. \text{Notes } C Y \in \text{set } [] \wedge m \in \text{parts } \{Y\})$

2.  $\wedge$  a list.

$\llbracket m \in \text{staticSecret } A;$   
 $m \in \text{parts } (\text{knows } C \text{ list}) \longrightarrow$   
 $A = C \vee$   
 $(\exists D E X. \text{Says } D E X \in \text{set list} \wedge m \in \text{parts } \{X\}) \vee$   
 $(\exists Y. \text{Notes } C Y \in \text{set list} \wedge m \in \text{parts } \{Y\}) \rrbracket$   
 $\implies m \in \text{parts } (\text{knows } C (a \# \text{list})) \longrightarrow$   
 $A = C \vee$   
 $(\exists D E X. \text{Says } D E X \in \text{set } (a \# \text{list}) \wedge m \in \text{parts } \{X\}) \vee$   
 $(\exists Y. \text{Notes } C Y \in \text{set } (a \# \text{list}) \wedge m \in \text{parts } \{Y\})$

**apply** (*simp add: staticSecretA-notin-parts-initStateB*)  
**apply** (*induct-tac a*)

**apply** (*rule impI*)  
**apply** *simp*  
**apply** (*drule parts-insert [THEN equalityD1, THEN subsetD]*)  
**apply** *blast*

**apply** *simp*

**apply** *simp*  
**apply** *clarify*

1.  $\wedge$  list x1 x2.

$\llbracket m \in \text{staticSecret } A;$   
 $m \in \text{parts } (\text{knows } C \text{ list}) \longrightarrow$   
 $A = C \vee$   
 $(\exists D E X. \text{Says } D E X \in \text{set list} \wedge m \in \text{parts } \{X\}) \vee$   
 $(\exists Y. \text{Notes } C Y \in \text{set list} \wedge m \in \text{parts } \{Y\});$   
 $m \in \text{parts } (\text{insert } x2 (\text{knows } C \text{ list})); A \neq C;$   
 $\nexists Y. (Y = x2 \vee \text{Notes } C Y \in \text{set list}) \wedge m \in \text{parts } \{Y\} \rrbracket$   
 $\implies \exists D E X. \text{Says } D E X \in \text{set list} \wedge m \in \text{parts } \{X\}$

**apply** (*drule parts-insert [THEN equalityD1, THEN subsetD]*)  
**apply** *blast*  
**done**

**lemma** *staticSecret-analz-agent:*

$\llbracket m \in \text{analz } (\text{knows } C \text{ evs}); m \in \text{staticSecret } A \rrbracket \implies$   
 $A = C \vee$   
 $(\exists D E X. \text{Says } D E X \in \text{set evs} \wedge m \in \text{parts}\{X\}) \vee$   
 $(\exists Y. \text{Notes } C Y \in \text{set evs} \wedge m \in \text{parts}\{Y\})$   
**by** (*drule analz-into-parts [THEN staticSecret-parts-agent]*)

**lemma** *secret-parts-agent:*

```

   $m \in \text{parts}(\text{knows } C \text{ evs}) \implies m \in \text{initState } C \vee$ 
   $(\exists A B X. \text{Says } A B X \in \text{set evs} \wedge m \in \text{parts}\{X\}) \vee$ 
   $(\exists Y. \text{Notes } C Y \in \text{set evs} \wedge m \in \text{parts}\{Y\})$ 
apply (erule rev-mp)
apply (induct-tac evs)
apply (simp add: parts-initState)
apply (induct-tac a)

apply (rule impI)
apply simp
apply (drule parts-insert [THEN equalityD1, THEN subsetD])
apply blast

apply simp

apply simp
apply clarify
apply (drule parts-insert [THEN equalityD1, THEN subsetD])
apply blast
done

```

## 12.2 Protocol dependent study

```

lemma NS-staticSecret-parts-agent-weak:
   $\llbracket m \in \text{parts}(\text{knows } C \text{ evs}); m \in \text{staticSecret } A;$ 
   $\text{evs} \in \text{ns-public} \rrbracket \implies$ 
   $A=C \vee (\exists D E X. \text{Says } D E X \in \text{set evs} \wedge m \in \text{parts}\{X\})$ 
apply (blast dest: NS-no-Notes staticSecret-parts-agent)
done

```

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:
   $\llbracket m \in \text{parts}(\text{knows } C \text{ evs}); m \in \text{staticSecret } A; A \neq C; \text{evs} \in \text{ns-public} \rrbracket \implies$ 
   $m \in \text{parts}(\text{knows } D \text{ evs})$ 
apply (blast dest: NS-staticSecret-parts-agent-weak Says-imp-knows [THEN parts.Inj])
parts-trans)

done

```

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:
   $\llbracket m \in \text{parts}(\text{knows } C \text{ evs}); m \in \text{staticSecret } A;$ 

```

$C \neq A; evs \in ns-public$   
 $\implies \exists B X. Says A B X \in set\ evs \wedge m \in parts\ \{X\}$   
**apply** (*erule rev-mp, erule ns-public.induct*)

**apply** (*simp add: staticSecretA-notin-parts-initStateB*)

**apply** *simp*

**apply** *clarify*

**apply** (*drule parts-insert [THEN equalityD1, THEN subsetD]*)

**apply** (*case-tac Aa=A*)

**apply** *clarify*

1.  $\bigwedge evsf\ X\ Aa\ B.$

$\llbracket m \in staticSecret\ A; C \neq A; evsf \in ns-public;$   
 $m \in parts\ (knows\ C\ evsf) \longrightarrow$   
 $(\exists B X. Says\ A\ B\ X \in set\ evsf \wedge m \in parts\ \{X\});$   
 $X \in synth\ (analz\ (knows\ A\ evsf));$   
 $m \in parts\ \{X\} \cup parts\ (knows\ C\ evsf) \rrbracket$   
 $\implies \exists Ba\ Xa.$   
 $(A = A \wedge Ba = B \wedge Xa = X \vee Says\ A\ Ba\ Xa \in set\ evsf) \wedge$   
 $m \in parts\ \{Xa\}$

2.  $\bigwedge evsf\ X\ Aa\ B.$

$\llbracket m \in staticSecret\ A; C \neq A; evsf \in ns-public;$   
 $m \in parts\ (knows\ C\ evsf) \longrightarrow$   
 $(\exists B X. Says\ A\ B\ X \in set\ evsf \wedge m \in parts\ \{X\});$   
 $X \in synth\ (analz\ (knows\ Aa\ evsf));$   
 $m \in parts\ \{X\} \cup parts\ (knows\ C\ evsf); Aa \neq A \rrbracket$   
 $\implies \exists Ba\ Xa.$   
 $(A = Aa \wedge Ba = B \wedge Xa = X \vee Says\ A\ Ba\ Xa \in set\ evsf) \wedge$   
 $m \in parts\ \{Xa\}$

*A total of 6 subgoals...*

**apply** *blast*

**apply** *simp*

**apply** *clarify*

1.  $\bigwedge evsf\ X\ Aa.$

$\llbracket m \in staticSecret\ A; C \neq A; evsf \in ns-public;$   
 $X \in synth\ (analz\ (knows\ Aa\ evsf)); Aa \neq A;$   
 $m \notin parts\ (knows\ C\ evsf); m \in parts\ \{X\} \rrbracket$   
 $\implies \exists B X. Says\ A\ B\ X \in set\ evsf \wedge m \in parts\ \{X\}$

*A total of 5 subgoals...*

**apply** (*drule Fake-parts-sing [THEN subsetD], simp*)

**apply** (*simp add: staticSecret-synth-eq*)

1.  $\bigwedge \text{evsf } X \text{ Aa.}$   
 $\llbracket m \in \text{staticSecret } A; C \neq A; \text{evsf} \in \text{ns-public}; \text{Aa} \neq A;$   
 $m \notin \text{parts } (\text{knows } C \text{ evsf}); m \in \text{parts } \{X\}; m \in \text{parts } (\text{knows } \text{Aa } \text{evsf}) \rrbracket$   
 $\implies \exists B \ X. \text{Says } A \ B \ X \in \text{set evsf} \wedge m \in \text{parts } \{X\}$   
*A total of 5 subgoals...*

**apply** (*blast dest: NS-staticSecret-parts-agent-parts*)

**apply** (*force simp add: staticSecret-def*)+  
**done**

**lemma** *NS-agent-see-staticSecret:*

$\llbracket m \in \text{staticSecret } A; C \neq A; \text{evs} \in \text{ns-public} \rrbracket$   
 $\implies m \in \text{parts } (\text{knows } C \text{ evs}) = (\exists B \ X. \text{Says } A \ B \ X \in \text{set evs} \wedge m \in \text{parts } \{X\})$   
**apply** (*force dest: NS-staticSecret-parts-agent Says-imp-knows [THEN parts.Inj]*)  
*intro: parts-trans*)  
**done**

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

**lemma** *analz-insert-analz:*

$\llbracket c \notin \text{parts}\{Z\}; \forall K. \text{Key } K \notin \text{parts}\{Z\}; c \in \text{analz}(\text{insert } Z \ H) \rrbracket \implies c \in \text{analz } H$   
**apply** (*erule rev-mp, erule rev-mp*)  
**apply** (*erule analz.induct*)  
**prefer** 4  
**apply** *clarify*

1.  $\bigwedge K \ X. \llbracket \text{Crypt } K \ X \in \text{analz } (\text{insert } Z \ H);$   
 $\text{Key } (\text{invKey } K) \in \text{analz } (\text{insert } Z \ H); \forall K. \text{Key } K \notin \text{parts } \{Z\};$   
 $X \notin \text{parts } \{Z\}; \text{Crypt } K \ X \notin \text{parts } \{Z\} \longrightarrow \text{Crypt } K \ X \in \text{analz } H;$   
 $\text{Key } (\text{invKey } K) \notin \text{parts } \{Z\} \longrightarrow \text{Key } (\text{invKey } K) \in \text{analz } H \rrbracket$   
 $\implies X \in \text{analz } H$

*A total of 4 subgoals...*

**apply** (*blast dest: parts.Body analz.Decrypt*)

**apply** *blast+*

**done**

**lemma** *Agent-not-see-NA:*

$\llbracket \text{Key } (\text{priEK } B) \notin \text{analz}(\text{knows } C \ \text{evs});$   
 $\text{Key } (\text{priEK } A) \notin \text{analz}(\text{knows } C \ \text{evs});$   
 $\forall S \ R \ Y. \text{Says } S \ R \ Y \in \text{set evs} \longrightarrow$   
 $Y = \text{Crypt } (\text{pubEK } B) \ \{\!\! \{ \text{Nonce } NA, \text{Agent } A \}\!\! \} \vee$   
 $Y = \text{Crypt } (\text{pubEK } A) \ \{\!\! \{ \text{Nonce } NA, \text{Nonce } NB \}\!\! \} \vee$   
 $\text{Nonce } NA \notin \text{parts}\{Y\} \wedge (\forall K. \text{Key } K \notin \text{parts}\{Y\});$   
 $C \neq A; C \neq B; \text{evs} \in \text{ns-public} \rrbracket$

$\implies \text{Nonce } NA \notin \text{analz } (\text{knows } C \text{ evs})$   
**apply** (*erule rev-mp, erule rev-mp, erule rev-mp, erule ns-public.induct*)  
**apply** (*simp add: nonce-notin-analz-initState*)  
**apply** *clarify*  
**apply** *simp*  
**apply** (*drule subset-insertI [THEN analz-mono, THEN contra-subsetD]*)+

1.  $\bigwedge \text{evsf } X \text{ Aa } Ba.$

$\llbracket C \neq A; C \neq B; \text{evsf} \in \text{ns-public};$   
 $\text{Key } (\text{priEK } A) \notin \text{analz } (\text{knows } C \text{ evsf}) \longrightarrow$   
 $\text{Key } (\text{priEK } B) \notin \text{analz } (\text{knows } C \text{ evsf}) \longrightarrow$   
 $\text{Nonce } NA \notin \text{analz } (\text{knows } C \text{ evsf});$   
 $X \in \text{synth } (\text{analz } (\text{knows } Aa \text{ evsf}));$   
 $\forall S R Y.$   
 $(S = Aa \wedge R = Ba \wedge Y = X \longrightarrow$   
 $X = \text{Crypt } (\text{pubEK } B) \{ \text{Nonce } NA, \text{Agent } A \} \vee$   
 $X = \text{Crypt } (\text{pubEK } A) \{ \text{Nonce } NA, \text{Nonce } NB \} \vee$   
 $\text{Nonce } NA \notin \text{parts } \{X\} \wedge (\forall K. \text{Key } K \notin \text{parts } \{X\})) \wedge$   
 $(\text{Says } S R Y \in \text{set evsf} \longrightarrow$   
 $Y = \text{Crypt } (\text{pubEK } B) \{ \text{Nonce } NA, \text{Agent } A \} \vee$   
 $Y = \text{Crypt } (\text{pubEK } A) \{ \text{Nonce } NA, \text{Nonce } NB \} \vee$   
 $\text{Nonce } NA \notin \text{parts } \{Y\} \wedge (\forall K. \text{Key } K \notin \text{parts } \{Y\}));$   
 $\text{Nonce } NA \in \text{analz } (\text{insert } X (\text{knows } C \text{ evsf}));$   
 $\text{Key } (\text{priEK } A) \notin \text{analz } (\text{knows } C \text{ evsf});$   
 $\text{Key } (\text{priEK } B) \notin \text{analz } (\text{knows } C \text{ evsf})\rrbracket$   
 $\implies \text{False}$

A total of 5 subgoals...

**apply** (*subgoal-tac*

$\forall S R Y.$   
 $(S = Aa \wedge R = Ba \wedge Y = X \longrightarrow$   
 $X = \text{Crypt } (\text{pubK } B) \{ \text{Nonce } NA, \text{Agent } A \} \vee$   
 $X = \text{Crypt } (\text{pubK } A) \{ \text{Nonce } NA, \text{Nonce } NB \} \vee$   
 $\text{Nonce } NA \notin \text{parts } \{X\} \wedge (\forall K. \text{Key } K \notin \text{parts } \{X\}))$ )

**prefer** 2

**apply** *blast*

**apply** *simp*

1.  $\bigwedge \text{evsf } X \text{ Aa.}$

$\llbracket C \neq A; C \neq B; \text{evsf} \in \text{ns-public}; \text{Nonce } NA \notin \text{analz } (\text{knows } C \text{ evsf});$   
 $X \in \text{synth } (\text{analz } (\text{knows } Aa \text{ evsf}));$   
 $\forall S R Y.$   
 $\text{Says } S R Y \in \text{set evsf} \longrightarrow$   
 $Y = \text{Crypt } (\text{pubEK } B) \{ \text{Nonce } NA, \text{Agent } A \} \vee$   
 $Y = \text{Crypt } (\text{pubEK } A) \{ \text{Nonce } NA, \text{Nonce } NB \} \vee$   
 $\text{Nonce } NA \notin \text{parts } \{Y\} \wedge (\forall K. \text{Key } K \notin \text{parts } \{Y\});$   
 $\text{Nonce } NA \in \text{analz } (\text{insert } X (\text{knows } C \text{ evsf}));$

```

Key (priEK A) ∉ analz (knows C evsf);
Key (priEK B) ∉ analz (knows C evsf);
X = Crypt (pubEK B) {Nonce NA, Agent A} ∨
X = Crypt (pubEK A) {Nonce NA, Nonce NB} ∨
Nonce NA ∉ parts {X} ∧ (∀ K. Key K ∉ parts {X})]
⇒ False

```

A total of 5 subgoals...

```

apply (force dest!: analz-insert-analz)

```

```

apply auto
done

```

```

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 ∈ knows A evs; Notes A X ∉ set evs;
  X ∉ {Key (priEK A), Key (priSK A), Key (shrK A)} ]]
⇒ X ∈ knows B evs

```

```

apply (erule rev-mp, erule rev-mp, erule rev-mp)

```

```

apply (induct-tac A, induct-tac B, induct-tac evs)

```

```

apply (induct-tac [2] a) apply auto

```

```

done

```

```

lemma knowledge-equiv-bis:

```

```

[[ X ∈ knows A evs; Notes A X ∉ set evs ]]
⇒ X ∈ {Key (priEK A), Key (priSK A), Key (shrK A)} ∪ knows B evs

```

```

apply (blast dest: knowledge-equiv)

```

```

done

```

```

lemma knowledge-equiv-ter:

```

```

[[ X ∈ knows A evs; X ∉ {Key (priEK A), Key (priSK A), Key (shrK A)} ]]
⇒ X ∈ knows B evs ∨ Notes A X ∈ set evs

```

```

apply (blast dest: knowledge-equiv)

```

```

done

```

**lemma** *knowledge-equiv-quater*:

$X \in \text{knows } A \text{ evs}$   
 $\implies X \in \text{knows } B \text{ evs} \vee \text{Notes } A \ X \in \text{set evs} \vee$   
 $X \in \{\text{Key } (\text{priEK } A), \text{Key } (\text{priSK } A), \text{Key } (\text{shrK } A)\}$   
**apply** (*blast dest: knowledge-equiv*)  
**done**

**lemma** *setdiff-diff-insert*:  $A-B-C=D-E-F \implies \text{insert } m \ (A-B-C) = \text{insert } m \ (D-E-F)$   
**by** *simp*

**lemma**  $A-B-C=D-E-F \implies \text{insert } m \ A-B-C = \text{insert } m \ D-E-F$   
**oops**

**lemma** *knowledge-equiv-eq-setdiff*:

$\text{knows } A \text{ evs} -$   
 $\{\text{Key } (\text{priEK } A), \text{Key } (\text{priSK } A), \text{Key } (\text{shrK } A)\} -$   
 $\{X. \text{Notes } A \ X \in \text{set evs}\}$   
 $=$   
 $\text{knows } B \text{ evs} -$   
 $\{\text{Key } (\text{priEK } B), \text{Key } (\text{priSK } B), \text{Key } (\text{shrK } B)\} -$   
 $\{X. \text{Notes } B \ X \in \text{set evs}\}$   
**apply** (*induct-tac evs, induct-tac A, induct-tac B*)  
**apply** *force*  
**apply** (*induct-tac a*)

**apply** *simp-all*

**oops**

**lemma** *knowledge-equiv-eq-old*:

$\text{knows } A \text{ evs} \cup$   
 $\{\text{Key } (\text{priEK } B), \text{Key } (\text{priSK } B), \text{Key } (\text{shrK } B)\} \cup$   
 $\{X. \text{Notes } B \ X \in \text{set evs}\}$   
 $=$   
 $\text{knows } B \text{ evs} \cup$   
 $\{\text{Key } (\text{priEK } A), \text{Key } (\text{priSK } A), \text{Key } (\text{shrK } A)\} \cup$   
 $\{X. \text{Notes } A \ X \in \text{set evs}\}$   
**apply** (*induct-tac evs, induct-tac A, induct-tac B*)  
**apply** *force*  
**apply** (*induct-tac a*)

Gets case solves because this event doesn't touch any agent knowledge

**apply** *simp-all*  
**apply** *safe*

speeds up subsequent blasting

**apply** *blast+*

very very slow

**done**

**theorem** *knowledge-eval: knows A evs =*  
$$\{Key (priEK A), Key (priSK A), Key (shrK A)\} \cup$$
$$(Key \text{ ` } range \text{ pubEK}) \cup (Key \text{ ` } range \text{ pubSK}) \cup$$
$$\{X. \exists S R. Says S R X \in set \text{ evs}\} \cup$$
$$\{X. Notes A X \in set \text{ evs}\}$$

**apply** (*induct-tac A, induct-tac evs*)

**apply** (*induct-tac [2] a*)

**apply** *auto*

**done**

**lemma** *knowledge-eval-setdiff:*

*knows A evs -*

$$\{Key (priEK A), Key (priSK A), Key (shrK A)\} -$$
$$\{X. Notes A X \in set \text{ evs}\}$$

=

$$(Key \text{ ` } range \text{ pubEK}) \cup (Key \text{ ` } range \text{ pubSK}) \cup$$
$$\{X. \exists S R. Says S R X \in set \text{ evs}\}$$

**apply** (*simp only: knowledge-eval*) **apply** *auto*

**oops**

**theorem** *knowledge-equiv-eq: knows A evs  $\cup$*

$$\{Key (priEK B), Key (priSK B), Key (shrK B)\} \cup$$
$$\{X. Notes B X \in set \text{ evs}\}$$

=

*knows B evs  $\cup$*

$$\{Key (priEK A), Key (priSK A), Key (shrK A)\} \cup$$
$$\{X. Notes A X \in set \text{ evs}\}$$

**apply** (*force simp only: knowledge-eval*)

**done**

**lemma** *knows A evs  $\cup$*

$$\{Key (priEK B), Key (priSK B), Key (shrK B)\} \cup$$
$$\{X. Notes B X \in set \text{ evs}\} -$$

( 
$$\{Key (priEK B), Key (priSK B), Key (shrK B)\} \cup$$
$$\{X. Notes B X \in set \text{ evs}\} ) = \textit{knows A evs}$$

**apply** *auto*

**oops**

**theorem** *parts-knowledge-equiv-eq*:  
 $parts(knows\ A\ evs) \cup$   
 $\{Key\ (priEK\ B),\ Key\ (priSK\ B),\ Key\ (shrK\ B)\} \cup$   
 $parts(\{X.\ Notes\ B\ X \in\ set\ evs\})$   
 $=$   
 $parts(knows\ B\ evs) \cup$   
 $\{Key\ (priEK\ A),\ Key\ (priSK\ A),\ Key\ (shrK\ A)\} \cup$   
 $parts(\{X.\ Notes\ A\ X \in\ set\ evs\})$   
**apply** (*simp only: knowledge-eval parts-Un*) **apply force**  
**done**

**lemmas** *parts-knowledge-equiv = parts-knowledge-equiv-eq* [*THEN equalityD1, THEN subsetD*]

**thm** *parts-knowledge-equiv*

**theorem** *noprishr-parts-knowledge-equiv*:

$\llbracket X \notin \{Key\ (priEK\ A),\ Key\ (priSK\ A),\ Key\ (shrK\ A)\};$   
 $X \in parts(knows\ A\ evs) \rrbracket$   
 $\implies X \in parts(knows\ B\ evs) \cup$   
 $parts(\{X.\ Notes\ A\ X \in\ set\ evs\})$

**apply** (*force dest: UnI1 [THEN UnI1, THEN parts-knowledge-equiv]*)  
**done**

**lemma** *knowledge-equiv-eq-NS*:

$evs \in ns-public \implies$   
 $knows\ A\ evs \cup \{Key\ (priEK\ B),\ Key\ (priSK\ B),\ Key\ (shrK\ B)\} =$   
 $knows\ B\ evs \cup \{Key\ (priEK\ A),\ Key\ (priSK\ A),\ Key\ (shrK\ A)\}$

**apply** (*force simp only: knowledge-eval NS-no-Notes*)  
**done**

**lemma** *parts-knowledge-equiv-eq-NS*:

$evs \in ns-public \implies$   
 $parts(knows\ A\ evs) \cup \{Key\ (priEK\ B),\ Key\ (priSK\ B),\ Key\ (shrK\ B)\} =$   
 $parts(knows\ B\ evs) \cup \{Key\ (priEK\ A),\ Key\ (priSK\ A),\ Key\ (shrK\ A)\}$

**apply** (*simp only: knowledge-eval NS-no-Notes parts-Un*) **apply force**  
**done**

**theorem** *noprishr-parts-knowledge-equiv-NS*:

$\llbracket X \notin \{Key\ (priEK\ A),\ Key\ (priSK\ A),\ Key\ (shrK\ A)\};$   
 $X \in parts(knows\ A\ evs); evs \in ns-public \rrbracket$   
 $\implies X \in parts(knows\ B\ evs)$

**apply** (*drule noprishr-parts-knowledge-equiv, simp*)  
**apply** (*simp add: NS-no-Notes*)  
**done**

**theorem** *Agent-not-analz-N*:  
[[ *Nonce N*  $\notin$  *parts(knows A evs)*; *evs*  $\in$  *ns-public* ]]  
   $\implies$  *Nonce N*  $\notin$  *analz(knows B evs)*  
**apply** (*force intro: noprishr-parts-knowledge-equiv-NS analz-into-parts*)  
**done**

**end**

## References

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