

# Inductive Study of Confidentiality

Giampaolo Bella

Dipartimento di Matematica e Informatica, Università di Catania, Italy

September 13, 2023

## 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 . . . . .	7
1.4.3 Rewrite rules for pulling out atomic messages . . . . .	8
1.5 Inductive relation "analz" . . . . .	9
1.5.1 General equational properties . . . . .	10
1.5.2 Rewrite rules for pulling out atomic messages . . . . .	10
1.5.3 Idempotence and transitivity . . . . .	11
1.6 Inductive relation "synth" . . . . .	12
1.6.1 Unions . . . . .	13
1.6.2 Idempotence and transitivity . . . . .	13

1.6.3	Combinations of parts, <i>analz</i> and <i>synth</i> . . . . .	14
1.6.4	For reasoning about the Fake rule in traces . . . . .	14
1.7	HPair: a combination of Hash and MPair . . . . .	15
1.7.1	Freeness . . . . .	15
1.7.2	Specialized laws, proved in terms of those for Hash and MPair . . . . .	16
1.8	The set of key-free messages . . . . .	17
1.9	Tactics useful for many protocol proofs . . . . .	18
<b>2</b>	<b>Theory of Events for Security Protocols against Dolev-Yao</b>	<b>19</b>
2.1	Function <i>knows</i> . . . . .	20
2.2	Knowledge of Agents . . . . .	21
<b>3</b>	<b>Theory of Cryptographic Keys for Security Protocols against Dolev-Yao</b>	<b>23</b>
3.1	Asymmetric Keys . . . . .	24
3.2	Basic properties of <i>pubK</i> and <i>priEK</i> . . . . .	25
3.3	"Image" equations that hold for injective functions . . . . .	25
3.4	Symmetric Keys . . . . .	26
3.5	Initial States of Agents . . . . .	27
3.6	Function <i>knows Spy</i> . . . . .	29
3.7	Fresh Nonces . . . . .	30
3.8	Supply fresh nonces for possibility theorems . . . . .	30
3.9	Specialized Rewriting for Theorems About <i>analz</i> and Image .	30
3.10	Specialized Methods for Possibility Theorems . . . . .	31
<b>4</b>	<b>The Needham-Schroeder Public-Key Protocol against Dolev- Yao — with Gets event, hence with Reception rule</b>	<b>31</b>
<b>5</b>	<b>Inductive Study of Confidentiality against Dolev-Yao</b>	<b>34</b>
<b>6</b>	<b>Existing study - fully spelled out</b>	<b>34</b>
6.1	On static secrets . . . . .	35
6.2	On dynamic secrets . . . . .	35
<b>7</b>	<b>Novel study</b>	<b>35</b>
7.1	Protocol independent study . . . . .	35
7.2	Protocol-dependent study . . . . .	36
<b>8</b>	<b>Theory of Agents and Messages for Security Protocols against the General Attacker</b>	<b>38</b>
8.1	Inductive definition of all parts of a message . . . . .	39
8.2	Inverse of keys . . . . .	39
8.3	keysFor operator . . . . .	39
8.4	Inductive relation "parts" . . . . .	40

8.4.1	Unions . . . . .	41
8.4.2	Idempotence and transitivity . . . . .	41
8.4.3	Rewrite rules for pulling out atomic messages . . . . .	42
8.5	Inductive relation "analz" . . . . .	43
8.5.1	General equational properties . . . . .	44
8.5.2	Rewrite rules for pulling out atomic messages . . . . .	44
8.5.3	Idempotence and transitivity . . . . .	45
8.6	Inductive relation "synth" . . . . .	46
8.6.1	Unions . . . . .	47
8.6.2	Idempotence and transitivity . . . . .	47
8.6.3	Combinations of parts, analz and synth . . . . .	48
8.6.4	For reasoning about the Fake rule in traces . . . . .	48
8.7	HPair: a combination of Hash and MPair . . . . .	49
8.7.1	Freeness . . . . .	49
8.7.2	Specialized laws, proved in terms of those for Hash and MPair . . . . .	50
8.8	The set of key-free messages . . . . .	51
8.9	Tactics useful for many protocol proofs . . . . .	52
<b>9</b>	<b>Theory of Events for Security Protocols against the General Attacker</b>	<b>53</b>
9.1	Function <i>knows</i> . . . . .	53
9.2	Knowledge of generic agents . . . . .	54
<b>10</b>	<b>Theory of Cryptographic Keys for Security Protocols against the General Attacker</b>	<b>56</b>
10.1	Asymmetric Keys . . . . .	56
10.2	Basic properties of <i>pubK</i> and <i>priEK</i> . . . . .	57
10.3	"Image" equations that hold for injective functions . . . . .	57
10.4	Symmetric Keys . . . . .	58
10.5	Initial States of Agents . . . . .	59
10.6	Function <i>knows Spy</i> . . . . .	60
10.7	Fresh Nonces . . . . .	61
10.8	Supply fresh nonces for possibility theorems . . . . .	61
10.9	Specialized Rewriting for Theorems About <i>analz</i> and Image . . . . .	62
10.10	Specialized Methods for Possibility Theorems . . . . .	62
<b>11</b>	<b>The Needham-Schroeder Public-Key Protocol against the General Attacker</b>	<b>62</b>
<b>12</b>	<b>Inductive Study of Confidentiality against the General At- tacker</b>	<b>63</b>
12.1	Protocol independent study . . . . .	64
12.2	Protocol dependent study . . . . .	65

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

```
theory Message
imports Main
begin
```

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

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

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

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

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

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

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

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

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

```
syntax
-MTuple :: ['a, args] => 'a * 'b ((2{-,/-})
```

```
translations
{x, y, z} == {x, {y, z}}
```

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

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

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

## 1.1 Inductive definition of all parts of a message

**inductive-set**

$\text{parts} :: \text{msg set} ==> \text{msg set}$

**for**  $H :: \text{msg set}$

**where**

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

Monotonicity

**lemma**  $\text{parts-mono}: G \subseteq H ==> \text{parts}(G) \subseteq \text{parts}(H)$   
 $\langle \text{proof} \rangle$

Equations hold because constructors are injective.

**lemma**  $\text{Friend-image-eq } [\text{simp}]: (\text{Friend } x \in \text{Friend}'A) = (x:A)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{Key-image-eq } [\text{simp}]: (\text{Key } x \in \text{Key}'A) = (x \in A)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{Nonce-Key-image-eq } [\text{simp}]: (\text{Nonce } x \notin \text{Key}'A)$   
 $\langle \text{proof} \rangle$

## 1.2 Inverse of keys

**lemma**  $\text{invKey-eq } [\text{simp}]: (\text{invKey } K = \text{invKey } K') = (K=K')$   
 $\langle \text{proof} \rangle$

## 1.3 keysFor operator

**lemma**  $\text{keysFor-empty } [\text{simp}]: \text{keysFor } \{\} = \{\}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{keysFor-Un } [\text{simp}]: \text{keysFor } (H \cup H') = \text{keysFor } H \cup \text{keysFor } H'$   
 $\langle \text{proof} \rangle$

**lemma** *keysFor-UN* [simp]:  $keysFor (\bigcup_{i \in A}. H\ i) = (\bigcup_{i \in A}. keysFor\ (H\ i))$   
(proof)

Monotonicity

**lemma** *keysFor-mono*:  $G \subseteq H \implies keysFor(G) \subseteq keysFor(H)$   
(proof)

**lemma** *keysFor-insert-Agent* [simp]:  $keysFor\ (insert\ (Agent\ A)\ H) = keysFor\ H$   
(proof)

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

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

**lemma** *keysFor-insert-Key* [simp]:  $keysFor\ (insert\ (Key\ K)\ H) = keysFor\ H$   
(proof)

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

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

**lemma** *keysFor-insert-Crypt* [simp]:  
 $keysFor\ (insert\ (Crypt\ K\ X)\ H) = insert\ (invKey\ K)\ (keysFor\ H)$   
(proof)

**lemma** *keysFor-image-Key* [simp]:  $keysFor\ (Key\ 'E) = \{\}$   
(proof)

**lemma** *Crypt-imp-invKey-keysFor*:  $Crypt\ K\ X \in H \implies invKey\ K \in keysFor\ H$   
(proof)

## 1.4 Inductive relation "parts"

**lemma** *MPair-parts*:  
 $[\![\ \{\!\! \{X, Y\}\!\!\} \in parts\ H;\ \{\!\! \{X \in parts\ H;\ Y \in parts\ H\}\!\!\} \implies P\ ]\!] \implies P$   
(proof)

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

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

**lemma** *parts-increasing*:  $H \subseteq \text{parts}(H)$   
*<proof>*

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

**lemma** *parts-empty* [*simp*]:  $\text{parts}\{\} = \{\}$   
*<proof>*

**lemma** *parts-emptyE* [*elim!*]:  $X \in \text{parts}\{\} \implies P$   
*<proof>*

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

**lemma** *parts-singleton*:  $X \in \text{parts } H \implies \exists Y \in H. X \in \text{parts } \{Y\}$   
*<proof>*

#### 1.4.1 Unions

**lemma** *parts-Un-subset1*:  $\text{parts}(G) \cup \text{parts}(H) \subseteq \text{parts}(G \cup H)$   
*<proof>*

**lemma** *parts-Un-subset2*:  $\text{parts}(G \cup H) \subseteq \text{parts}(G) \cup \text{parts}(H)$   
*<proof>*

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

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

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

**lemma** *parts-insert2*:  
 $\text{parts}(\text{insert } X \ (\text{insert } Y \ H)) = \text{parts } \{X\} \cup \text{parts } \{Y\} \cup \text{parts } H$   
*<proof>*

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*:  $\text{insert } X \ (\text{parts } H) \subseteq \text{parts}(\text{insert } X \ H)$   
*<proof>*

#### 1.4.2 Idempotence and transitivity

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

$\langle proof \rangle$

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

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

**lemma** *parts-trans*:  $[| X \in parts G; G \subseteq parts H |] ==> X \in parts H$   
 $\langle proof \rangle$

Cut

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

**lemma** *parts-cut-eq* [simp]:  $X \in parts H ==> parts (insert X H) = parts H$   
 $\langle proof \rangle$

### 1.4.3 Rewrite rules for pulling out atomic messages

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

**lemma** *parts-insert-Agent* [simp]:  
 $parts (insert (Agent agt) H) = insert (Agent agt) (parts H)$   
 $\langle proof \rangle$

**lemma** *parts-insert-Nonce* [simp]:  
 $parts (insert (Nonce N) H) = insert (Nonce N) (parts H)$   
 $\langle proof \rangle$

**lemma** *parts-insert-Number* [simp]:  
 $parts (insert (Number N) H) = insert (Number N) (parts H)$   
 $\langle proof \rangle$

**lemma** *parts-insert-Key* [simp]:  
 $parts (insert (Key K) H) = insert (Key K) (parts H)$   
 $\langle proof \rangle$

**lemma** *parts-insert-Hash* [simp]:  
 $parts (insert (Hash X) H) = insert (Hash X) (parts H)$   
 $\langle proof \rangle$

**lemma** *parts-insert-Crypt* [simp]:  
 $parts (insert (Crypt K X) H) = insert (Crypt K X) (parts (insert X H))$   
 $\langle proof \rangle$

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



$parts (insert \{X, Y\} H) =$   
 $insert \{X, Y\} (parts (insert X (insert Y H)))$   
 <proof>

**lemma** *parts-image-Key* [simp]:  $parts (Key'N) = Key'N$   
 <proof>

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

**lemma** *msg-Nonce-supply*:  $\exists N. \forall n. N \leq n \longrightarrow Nonce\ n \notin parts\ \{msg\}$   
 <proof>

## 1.5 Inductive relation "analz"

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

**inductive-set**

*analz* :: *msg set* ==> *msg set*

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

**where**

| *Inj* [intro,simp] :  $X \in H \implies X \in analz\ H$

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

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

| *Decrypt* [dest]:

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

Monotonicity; Lemma 1 of Lowe's paper

**lemma** *analz-mono*:  $G \subseteq H \implies analz(G) \subseteq analz(H)$   
 <proof>

Making it safe speeds up proofs

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

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

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

$] \implies P$

<proof>

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

<proof>

**lemma** *analz-subset-parts*:  $analz\ H \subseteq parts\ H$

<proof>

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

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

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

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

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

### 1.5.1 General equational properties

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

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

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

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

### 1.5.2 Rewrite rules for pulling out atomic messages

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

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

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

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

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

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

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

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

$insert \{X, Y\} (analz (insert X (insert Y H)))$   
 ⟨proof⟩

Can pull out enCrypted message if the Key is not known

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

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

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

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

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

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

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

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

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

### 1.5.3 Idempotence and transitivity

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

**lemma** *analz-idem* [simp]:  $\text{analz} (\text{analz } H) = \text{analz } H$   
 ⟨proof⟩

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

**lemma** *analz-trans*:  $[[ X \in \text{analz } G; G \subseteq \text{analz } H ]] \implies X \in \text{analz } H$   
 ⟨proof⟩

Cut; Lemma 2 of Lowe

**lemma** *analz-cut*:  $[[ Y \in \text{analz} (\text{insert } X H); X \in \text{analz } H ]] \implies Y \in \text{analz } H$   
 ⟨proof⟩

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

**lemma** *analz-insert-eq*:  $X \in \text{analz } H \implies \text{analz} (\text{insert } X H) = \text{analz } H$   
 ⟨proof⟩

A congruence rule for "analz"

**lemma** *analz-subset-cong*:  
 $[[ \text{analz } G \subseteq \text{analz } G'; \text{analz } H \subseteq \text{analz } H' ]] \implies \text{analz} (G \cup H) \subseteq \text{analz} (G' \cup H')$   
 ⟨proof⟩

**lemma** *analz-cong*:  
 $[[ \text{analz } G = \text{analz } G'; \text{analz } H = \text{analz } H' ]] \implies \text{analz} (G \cup H) = \text{analz} (G' \cup H')$   
 ⟨proof⟩

**lemma** *analz-insert-cong*:  
 $\text{analz } H = \text{analz } H' \implies \text{analz}(\text{insert } X H) = \text{analz}(\text{insert } X H')$   
 ⟨proof⟩

If there are no pairs or encryptions then analz does nothing

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

## 1.6 Inductive relation "synth"

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

**inductive-set**

```

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

```

Monotonicity

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

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

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

```

Nonce n ∈ synth H
Key K ∈ synth H
Hash X ∈ synth H
{|X, Y|} ∈ synth H
Crypt K X ∈ synth H

```

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

### 1.6.1 Unions

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

**lemma** *synth-Un*:  $\text{synth}(G) \cup \text{synth}(H) \subseteq \text{synth}(G \cup H)$   
 ⟨proof⟩

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

### 1.6.2 Idempotence and transitivity

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

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

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

**lemma** *synth-trans*:  $[| X \in \text{synth } G; G \subseteq \text{synth } H |] \implies X \in \text{synth } H$

$\langle proof \rangle$

Cut; Lemma 2 of Lowe

**lemma** *synth-cut*:  $[[ Y \in synth (insert X H); X \in synth H ]] \implies Y \in synth H$   
 $\langle proof \rangle$

**lemma** *Agent-synth [simp]*:  $Agent A \in synth H$   
 $\langle proof \rangle$

**lemma** *Number-synth [simp]*:  $Number n \in synth H$   
 $\langle proof \rangle$

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

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

**lemma** *Crypt-synth-eq [simp]*:  
 $Key K \notin H \implies (Crypt K X \in synth H) = (Crypt K X \in H)$   
 $\langle proof \rangle$

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

### 1.6.3 Combinations of parts, analz and synth

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

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

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

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

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

**lemma** *parts-insert-subset-Un*:  $X \in G \implies parts(insert X H) \subseteq parts G \cup parts H$   
 $\langle proof \rangle$

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

**lemma** *Fake-parts-insert*:

$X \in \text{synth} (\text{analz } H) \implies$   
 $\text{parts} (\text{insert } X H) \subseteq \text{synth} (\text{analz } H) \cup \text{parts } H$   
 ⟨proof⟩

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

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

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

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

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

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

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

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

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

## 1.7 HPair: a combination of Hash and MPair

### 1.7.1 Freeness

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

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

*<proof>*

**lemma** *Number-neq-HPair*:  $Number\ N \sim = Hash[X]\ Y$   
*<proof>*

**lemma** *Key-neq-HPair*:  $Key\ K \sim = Hash[X]\ Y$   
*<proof>*

**lemma** *Hash-neq-HPair*:  $Hash\ Z \sim = Hash[X]\ Y$   
*<proof>*

**lemma** *Crypt-neq-HPair*:  $Crypt\ K\ X' \sim = Hash[X]\ Y$   
*<proof>*

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

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

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

**lemma** *HPair-eq* [*iff*]:  $(Hash[X]\ Y' = Hash[X]\ Y) = (X' = X \wedge Y' = Y)$   
*<proof>*

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

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

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

**lemma** *keysFor-insert-HPair* [*simp*]:  $keysFor\ (insert\ (Hash[X]\ Y)\ H) = keysFor\ H$   
*<proof>*

**lemma** *parts-insert-HPair* [*simp*]:  
 $parts\ (insert\ (Hash[X]\ Y)\ H) =$   
 $insert\ (Hash[X]\ Y)\ (insert\ (Hash\{X, Y\})\ (parts\ (insert\ Y\ H)))$   
*<proof>*

**lemma** *analz-insert-HPair* [*simp*]:  
 $analz\ (insert\ (Hash[X]\ Y)\ H) =$   
 $insert\ (Hash[X]\ Y)\ (insert\ (Hash\{X, Y\})\ (analz\ (insert\ Y\ H)))$   
*<proof>*

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



$$\begin{aligned}
& X \notin \text{synth}(\text{analz } H) \\
\implies & (\text{Hash}[X] \ Y \in \text{synth}(\text{analz } H)) = \\
& (\text{Hash} \ \{X, Y\} \in \text{analz } H \wedge Y \in \text{synth}(\text{analz } H)) \\
\langle \text{proof} \rangle
\end{aligned}$$

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*:  $parts (keyfree) \subseteq keyfree$   
*<proof>*

**lemma** *analz-keyfree-into-Un*:  $[X \in analz (G \cup H); G \subseteq keyfree] \implies X \in parts G \cup analz H$   
*<proof>*

## 1.9 Tactics useful for many protocol proofs

*<ML>*

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

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

**lemma** *Crypt-notin-image-Key* [*simp*]:  $Crypt K X \notin Key ' A$   
*<proof>*

**lemma** *Hash-notin-image-Key* [*simp*]:  $Hash X \notin Key ' A$   
*<proof>*

**lemma** *synth-analz-mono*:  $G \subseteq H \implies synth (analz(G)) \subseteq synth (analz(H))$   
*<proof>*

**lemma** *Fake-analz-eq* [*simp*]:  
 $X \in synth (analz H) \implies synth (analz (insert X H)) = synth (analz H)$   
*<proof>*

Two generalizations of *analz-insert-eq*

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

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

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

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

*<ML>*

end

## 2 Theory of Events for Security Protocols against Dolev-Yao

theory *Event* imports *Message* begin

**consts**

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

**datatype**

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

**consts**

*bad* :: *agent set* — compromised agents

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

**specification** (*bad*)

*Spy-in-bad* [iff]: *Spy* ∈ *bad*

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

⟨*proof*⟩

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

**where**

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

| *knows-Cons*:

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

(if *A* = *Spy* then

(case *ev* of

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

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

| *Notes* *A'* *X* =>

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

else

(case *ev* of

*Says* *A'* *B* *X* =>

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

| *Gets* *A'* *X* =>

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

| *Notes* *A'* *X* =>

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

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

**abbreviation** (*input*)

*spies* :: *event list* => *msg set* **where**

$spies == knows\ Spy$

**primrec**  $used :: event\ list \Rightarrow msg\ set$

**where**

$used\ Nil: used\ [] = (UN\ B. parts\ (initState\ B))$   
 $| used\ Cons: used\ (ev\ \#\ evs) =$   
      $(case\ ev\ of$   
          $Says\ A\ B\ X \Rightarrow parts\ \{X\} \cup used\ evs$   
          $| Gets\ A\ X \Rightarrow used\ evs$   
          $| Notes\ A\ X \Rightarrow parts\ \{X\} \cup used\ evs)$

— 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]:  $Notes\ A\ X \in set\ evs \longrightarrow X \in used\ evs$   
 $\langle proof \rangle$

**lemma** *Says-imp-used* [rule-format]:  $Says\ A\ B\ X \in set\ evs \longrightarrow X \in used\ evs$   
 $\langle proof \rangle$

## 2.1 Function *knows*

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

**lemma** *knows-Spy-Says* [simp]:

$knows\ Spy\ (Says\ A\ B\ X\ \#\ evs) = insert\ X\ (knows\ Spy\ evs)$   
 $\langle proof \rangle$

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

**lemma** *knows-Spy-Notes* [simp]:

$knows\ Spy\ (Notes\ A\ X\ \#\ evs) =$   
      $(if\ A:bad\ then\ insert\ X\ (knows\ Spy\ evs)\ else\ knows\ Spy\ evs)$   
 $\langle proof \rangle$

**lemma** *knows-Spy-Gets* [simp]:  $knows\ Spy\ (Gets\ A\ X\ \#\ evs) = knows\ Spy\ evs$   
 $\langle proof \rangle$

**lemma** *knows-Spy-subset-knows-Spy-Says*:

$knows\ Spy\ evs \subseteq knows\ Spy\ (Says\ A\ B\ X\ \#\ evs)$   
 $\langle proof \rangle$

**lemma** *knows-Spy-subset-knows-Spy-Notes*:

$knows\ Spy\ evs \subseteq knows\ Spy\ (Notes\ A\ X\ \#\ evs)$   
 $\langle proof \rangle$

**lemma** *knows-Spy-subset-knows-Spy-Gets*:

$knows\ Spy\ evs \subseteq knows\ Spy\ (Gets\ A\ X\ \# \ evs)$   
 ⟨proof⟩

Spy sees what is sent on the traffic

**lemma** *Says-imp-knows-Spy* [rule-format]:  
 $Says\ A\ B\ X \in\ set\ evs \longrightarrow X \in\ knows\ Spy\ evs$   
 ⟨proof⟩

**lemma** *Notes-imp-knows-Spy* [rule-format]:  
 $Notes\ A\ X \in\ set\ evs \longrightarrow A: bad \longrightarrow X \in\ knows\ Spy\ evs$   
 ⟨proof⟩

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

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

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

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

Compatibility for the old "spies" function

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

## 2.2 Knowledge of Agents

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

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

**lemma** *knows-Gets*:  
 $A \neq Spy \longrightarrow knows\ A\ (Gets\ A\ X\ \# \ evs) = insert\ X\ (knows\ A\ evs)$   
 ⟨proof⟩

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

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

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

Agents know what they say

**lemma** *Says-imp-knows* [rule-format]: *Says A B X ∈ set evs*  $\implies X \in \text{knows } A \text{ evs}$

*<proof>*

Agents know what they note

**lemma** *Notes-imp-knows* [rule-format]: *Notes A X ∈ set evs*  $\implies X \in \text{knows } A \text{ evs}$

*<proof>*

Agents know what they receive

**lemma** *Gets-imp-knows-agents* [rule-format]:

*A ≠ Spy*  $\implies \text{Gets } A \text{ X} \in \text{set evs} \implies X \in \text{knows } A \text{ evs}$

*<proof>*

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

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

$[\text{X} \in \text{knows } A \text{ evs}; A \neq \text{Spy}] \implies \exists B.$

*Says A B X ∈ set evs | Gets A X ∈ set evs | Notes A X ∈ set evs | X ∈ initState A*

*<proof>*

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

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

$[\text{X} \in \text{knows } \text{Spy} \text{ evs}] \implies \exists A B.$

*Says A B X ∈ set evs | Notes A X ∈ set evs | X ∈ initState Spy*

*<proof>*

**lemma** *parts-knows-Spy-subset-used*: *parts (knows Spy evs) ⊆ used evs*

*<proof>*

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

**lemma** *initState-into-used*: *X ∈ parts (initState B) ⇒ X ∈ used evs*

*<proof>*

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

*<proof>*

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

*<proof>*

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

*<proof>*

**lemma** *used-nil-subset*: *used [] ⊆ used evs*

*<proof>*

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

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

For proving theorems of the form  $X \notin \text{analz} (\text{knows Spy evs}) \longrightarrow P$  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  $\subseteq$  knows A (e # evs)  
<proof>
```

```
lemma initState-subset-knows: initState A  $\subseteq$  knows A evs  
<proof>
```

For proving *new-keys-not-used*

```
lemma keysFor-parts-insert:  
  [| K  $\in$  keysFor (parts (insert X G)); X  $\in$  synth (analz H) |]  
  ==> K  $\in$  keysFor (parts (G  $\cup$  H)) | Key (invKey K)  $\in$  parts H  
<proof>
```

```
lemmas analz-impI = impI [where P = Y  $\notin$  analz (knows Spy evs)] for Y evs  
<ML>
```

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>
```

**end**

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

```
theory Public  
imports Event  
begin
```

**lemma** *invKey-K*:  $K \in \text{symKeys} \implies \text{invKey } K = K$   
*<proof>*

### 3.1 Asymmetric Keys

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

**consts**

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

**abbreviation**

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

**abbreviation**

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

**abbreviation**

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

**abbreviation**

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

**abbreviation**

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

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

**abbreviation**

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

**abbreviation**

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

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

**specification** (*publicKey*)

*injective-publicKey*:

$\text{publicKey } b A = \text{publicKey } c A' \implies b=c \wedge A=A'$   
*<proof>*



**axiomatization where**

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

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

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

**lemma** *publicKey-inject* [iff]:  $(\text{publicKey } b \ A = \text{publicKey } c \ A') = (b=c \wedge A=A')$   
<proof>

**lemma** *not-symKeys-pubK* [iff]:  $\text{publicKey } b \ A \notin \text{symKeys}$   
<proof>

**lemma** *not-symKeys-priK* [iff]:  $\text{privateKey } b \ A \notin \text{symKeys}$   
<proof>

**lemma** *symKey-neq-priEK*:  $K \in \text{symKeys} \implies K \neq \text{priEK } A$   
<proof>

**lemma** *symKeys-neq-imp-neq*:  $(K \in \text{symKeys}) \neq (K' \in \text{symKeys}) \implies K \neq K'$   
<proof>

**lemma** *symKeys-invKey-iff* [iff]:  $(\text{invKey } K \in \text{symKeys}) = (K \in \text{symKeys})$   
<proof>

**lemma** *analz-symKeys-Decrypt*:  
[[ Crypt  $K \ X \in \text{analz } H$ ;  $K \in \text{symKeys}$ ; Key  $K \in \text{analz } H$  ]]  
 $\implies X \in \text{analz } H$   
<proof>

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

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

**lemma** *publicKey-image-eq* [simp]:  
 $(\text{publicKey } b \ x \in \text{publicKey } c \ A) = (b=c \wedge x \in A)$   
<proof>

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

**lemma** *privateKey-image-eq* [simp]:  
 $(\text{privateKey } b \ A \in \text{privateKey } c \ A) = (b=c \wedge A \in A)$   
<proof>

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

### 3.4 Symmetric Keys

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

**consts**

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

**specification** (*shrK*)

*inj-shrK*: *inj shrK*

— No two agents have the same long-term key

⟨proof⟩

**axiomatization where**

*sym-shrK* [iff]:  $\text{shrK } X \in \text{symKeys}$  — 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]:  $\text{invKey } (\text{shrK } A) = \text{shrK } A$

⟨proof⟩

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

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

⟨proof⟩

**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*

⟨proof⟩

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

⟨proof⟩

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

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

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

⟨proof⟩

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

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

**lemma** *priEK-noteq-shrK* [simp]: *priEK A*  $\neq$  *shrK B*  
 ⟨*proof*⟩

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

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

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

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

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

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

**declare** *invKey-K* [simp]

### 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*]:

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

*<proof>*

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

*<proof>*

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

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

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

*<proof>*

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

**lemma** *keysFor-parts-initState* [*simp*]:  $\text{keysFor } (\text{parts } (\text{initState } C)) = \{\}$

*<proof>*

**lemma** *Crypt-notin-initState*:  $\text{Crypt } K X \notin \text{parts } (\text{initState } B)$

*<proof>*

**lemma** *Crypt-notin-used-empty* [*simp*]:  $\text{Crypt } K X \notin \text{used } []$

*<proof>*

**lemma** *shrK-in-initState* [*iff*]:  $\text{Key } (\text{shrK } A) \in \text{initState } A$

*<proof>*

**lemma** *shrK-in-knows* [*iff*]:  $\text{Key } (\text{shrK } A) \in \text{knows } A \text{ evs}$

*<proof>*

**lemma** *shrK-in-used* [*iff*]:  $\text{Key } (\text{shrK } A) \in \text{used evs}$

*<proof>*

**lemma** *Key-not-used* [*simp*]:  $\text{Key } K \notin \text{used evs} \implies K \notin \text{range } \text{shrK}$

*<proof>*

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

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

### 3.6 Function *knows Spy*

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

Agents see their own private keys!

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

Agents see all public keys!

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

All public keys are visible

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

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

Spy sees private keys of bad agents!

**lemma** *Spy-spies-bad-privateKey* [*intro!*]:  
 $A \in\ bad \implies Key\ (privateKey\ b\ A) \in\ spies\ evs$   
 ⟨*proof*⟩

Spy sees long-term shared keys of bad agents!

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

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

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

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

### 3.7 Fresh Nonces

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

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

### 3.8 Supply fresh nonces for possibility theorems

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

**lemma** *Nonce-supply-lemma*:  $\exists N. \forall n. N \leq n \longrightarrow \text{Nonce } n \notin \text{used evs}$   
 ⟨proof⟩

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

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

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

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

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

**lemma** *Crypt-imp-keysFor* : [| *Crypt*  $K X \in H$ ;  $K \in \text{symKeys}$  |]  $\implies K \in \text{keysFor } H$   
 ⟨proof⟩

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

**lemma** *analz-image-freshK-lemma*:  
 $(\text{Key } K \in \text{analz } (\text{Key}'nE \cup H)) \longrightarrow (K \in nE \mid \text{Key } K \in \text{analz } H) \implies$   
 $(\text{Key } K \in \text{analz } (\text{Key}'nE \cup H)) = (K \in nE \mid \text{Key } K \in \text{analz } H)$   
 ⟨proof⟩

**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*]

$\langle ML \rangle$

### 3.10 Specialized Methods for Possibility Theorems

$\langle ML \rangle$

end

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

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

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

*Nil*: []  $\in$  *ns-public*

| *Fake*:  $\llbracket \text{evsf} \in \text{ns-public}; X \in \text{synth} (\text{analz} (\text{knows Spy evsf})) \rrbracket$   
 $\implies \text{Says Spy } B \ X \ \# \ \text{evsf} \in \text{ns-public}$

| *Reception*:  $\llbracket \text{evsr} \in \text{ns-public}; \text{Says } A \ B \ X \in \text{set evsr} \rrbracket$   
 $\implies \text{Gets } B \ X \ \# \ \text{evsr} \in \text{ns-public}$

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

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

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

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

declare *analz-into-parts* [dest]

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

**lemma**  $\exists NB. \exists evs \in ns-public. Says\ A\ B\ (Crypt\ (pubEK\ B)\ (Nonce\ NB)) \in set\ evs$   
 $\langle proof \rangle$

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

**lemma** *Gets-imp-Says* :  
 $\llbracket Gets\ B\ X \in set\ evs; evs \in ns-public \rrbracket \implies \exists A. Says\ A\ B\ X \in set\ evs$   
 $\langle proof \rangle$

**lemma** *Gets-imp-knows-Spy*:  
 $\llbracket Gets\ B\ X \in set\ evs; evs \in ns-public \rrbracket \implies X \in knows\ Spy\ evs$   
 $\langle proof \rangle$

**lemma** *Gets-imp-knows-Spy-parts[dest]*:  
 $\llbracket Gets\ B\ X \in set\ evs; evs \in ns-public \rrbracket \implies X \in parts\ (knows\ Spy\ evs)$   
 $\langle proof \rangle$

**lemma** *Spy-see-priEK [simp]*:  
 $evs \in ns-public \implies (Key\ (priEK\ A) \in parts\ (knows\ Spy\ evs)) = (A \in bad)$   
 $\langle proof \rangle$

**lemma** *Spy-analz-priEK [simp]*:  
 $evs \in ns-public \implies (Key\ (priEK\ A) \in analz\ (knows\ Spy\ evs)) = (A \in bad)$   
 $\langle proof \rangle$

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

**lemma** *unique-NA*:  
 $\llbracket Crypt\ (pubEK\ B)\ \{\{Nonce\ NA, Agent\ A\}\} \in parts\ (knows\ Spy\ evs);$   
 $Crypt\ (pubEK\ B')\ \{\{Nonce\ NA, Agent\ A'\}\} \in parts\ (knows\ Spy\ evs);$



$$\text{Nonce } NA \notin \text{analz } (\text{knows Spy } \text{evs}); \text{evs} \in \text{ns-public}]$$

$$\implies A=A' \wedge B=B'$$
 <proof>

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

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

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

$$\implies \text{Nonce } NA \notin \text{analz } (\text{knows Spy } \text{evs})$$
 <proof>

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

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

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

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

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

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

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

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

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

$$\implies \text{Says } B \ A \ (\text{Crypt}(\text{pubEK } A) \ \{\!\{ \text{Nonce } NA, \text{Nonce } NB \}\!\}) \in \text{set } \text{evs}$$
 <proof>

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

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

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

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

$$\text{Says } A \ B \ (\text{Crypt}(\text{pubEK } B) \ \{\!\{ \text{Nonce } NA, \text{Agent } A \}\!\}) \in \text{set } \text{evs}$$
 <proof>

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

$$\llbracket \text{Crypt}(\text{pubEK } A) \ \{\!\{ \text{Nonce } NA, \text{Nonce } NB \}\!\} \in \text{parts}(\text{knows Spy } \text{evs});$$

$$\text{Crypt}(\text{pubEK } A') \ \{\!\{ \text{Nonce } NA', \text{Nonce } NB \}\!\} \in \text{parts}(\text{knows Spy } \text{evs});$$

$$\text{Nonce } NB \notin \text{analz } (\text{knows Spy } \text{evs}); \text{evs} \in \text{ns-public}]$$

$$\implies A=A' \wedge NA=NA'$$
 <proof>

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

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

*<proof>*

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

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

*<proof>*

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

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

*<proof>*

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

$\implies \text{Says } B \ A \ (\text{Crypt } (\text{pubEK } A) \ \{\!\! \{ \text{Nonce } NA, \text{Nonce } NB \}\!\! \}) \in \text{set } \text{evs}$   
 $\longrightarrow \text{Nonce } NB \notin \text{analz } (\text{knows } \text{Spy } \text{evs})$

*<proof>*

end

## 5 Inductive Study of Confidentiality against Dolev-Yao

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

## 6 Existing study - fully spelled out

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

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

## 6.1 On static secrets

**lemma** *Spy-see-priEK*:

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

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

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

## 6.2 On dynamic secrets

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

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

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

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

## 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**

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

### 7.1 Protocol independent study

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

**lemma** *staticSecret-parts-Spy*:

$\llbracket m \in parts (knows Spy evs); m \in staticSecret A \rrbracket \implies$   
 $A \in bad \vee$   
 $(\exists C B X. Says C B X \in set evs \wedge m \in parts\{X\}) \vee$   
 $(\exists C Y. Notes C Y \in set evs \wedge C \in bad \wedge m \in parts\{Y\})$   
 $\langle proof \rangle$

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

$\llbracket m \in analz (knows Spy evs); m \in staticSecret A \rrbracket \implies$   
 $A \in bad \vee$   
 $(\exists C B X. Says C B X \in set evs \wedge m \in parts\{X\}) \vee$

$(\exists C Y. \text{Notes } C Y \in \text{set evs} \wedge C \in \text{bad} \wedge m \in \text{parts}\{Y\})$   
 $\langle \text{proof} \rangle$

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

$m \in \text{parts}(\text{knows Spy evs}) \implies$   
 $m \in \text{initState Spy} \vee$   
 $(\exists C B X. \text{Says } C B X \in \text{set evs} \wedge m \in \text{parts}\{X\}) \vee$   
 $(\exists C Y. \text{Notes } C Y \in \text{set evs} \wedge C \in \text{bad} \wedge m \in \text{parts}\{Y\})$   
 $\langle \text{proof} \rangle$

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

$m \in \text{initState Spy} \vee$   
 $(\exists C B X. \text{Says } C B X \in \text{set evs} \wedge m \in \text{parts}\{X\}) \vee$   
 $(\exists C Y. \text{Notes } C Y \in \text{set evs} \wedge C \in \text{bad} \wedge m \in \text{parts}\{Y\})$   
 $\implies m \in \text{parts}(\text{knows Spy evs})$   
 $\langle \text{proof} \rangle$

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

$m \in \text{analz}(\text{knows Spy evs}) \implies$   
 $m \in \text{initState Spy} \vee$   
 $(\exists C B X. \text{Says } C B X \in \text{set evs} \wedge m \in \text{parts}\{X\}) \vee$   
 $(\exists C Y. \text{Notes } C Y \in \text{set evs} \wedge C \in \text{bad} \wedge m \in \text{parts}\{Y\})$   
 $\langle \text{proof} \rangle$

## 7.2 Protocol-dependent study

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

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

$\llbracket m \in \text{staticSecret } A; \text{evs} \in \text{ns-public} \rrbracket \implies$   
 $m \in \text{parts}(\text{knows Spy evs}) = (A \in \text{bad})$   
 $\langle \text{proof} \rangle$

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

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

$\text{evs} \in \text{ns-public} \implies \text{Notes } A X \notin \text{set evs}$   
 $\langle \text{proof} \rangle$

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

$\llbracket m \in \text{parts}(\text{knows Spy evs}); m \in \text{staticSecret } A; \text{evs} \in \text{ns-public} \rrbracket \implies A \in \text{bad} \vee$   
 $(\exists C B X. \text{Says } C B X \in \text{set evs} \wedge m \in \text{parts}\{X\})$   
 $\langle \text{proof} \rangle$

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

$\llbracket \text{Says } A \ B \ X \in \text{set } \text{evs}; m \in \text{staticSecret } C; m \in \text{parts}\{X\};$   
 $\text{evs} \in \text{ns-public} \rrbracket \implies A = \text{Spy}$   
 ⟨proof⟩

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

**lemma** *staticSecret-synth-eq:*

$m \in \text{staticSecret } A \implies (m \in \text{synth } H) = (m \in H)$   
 ⟨proof⟩

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

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

⟨proof⟩

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

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

$\llbracket m \in \text{parts } (\text{knows } \text{Spy } \text{evs}); m \in \text{staticSecret } A;$   
 $\text{evs} \in \text{ns-public} \rrbracket \implies A \in \text{bad}$   
 ⟨proof⟩

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*)

⟨proof⟩

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

$\llbracket m \in \text{staticSecret } A; \text{evs} \in \text{ns-public} \rrbracket \implies$   
 $m \in \text{analz } (\text{knows } \text{Spy } \text{evs}) = (A \in \text{bad})$   
 ⟨proof⟩

**lemma** *NS-staticSecret-subset-parts-knows-Spy:*

$\text{evs} \in \text{ns-public} \implies$   
 $\text{staticSecret } A \subseteq \text{parts } (\text{knows } \text{Spy } \text{evs}) = (A \in \text{bad})$   
 ⟨proof⟩

**lemma** *NS-staticSecret-subset-analz-knows-Spy:*

$\text{evs} \in \text{ns-public} \implies$   
 $\text{staticSecret } A \subseteq \text{analz } (\text{knows } \text{Spy } \text{evs}) = (A \in \text{bad})$   
 ⟨proof⟩

end

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

theory *MessageGA* imports *Main* begin

**lemma** [*simp*] :  $A \cup (B \cup A) = B \cup A$   
*<proof>*

**type-synonym**

*key* = *nat*

**consts**

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

**specification** (*invKey*)

*invKey* [*simp*]:  $\text{invKey} (\text{invKey } K) = K$   
*invKey-symmetric*:  $\text{all-symmetric} \longrightarrow \text{invKey} = \text{id}$   
*<proof>*

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

**definition** *symKeys* :: *key set* **where**

*symKeys* == {*K*. *invKey* *K* = *K*}

**datatype** — We only allow for any number of friendly agents

*agent* = *Friend nat*

**datatype**

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

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

**syntax**

*-MTuple* :: [*'a*, *args*] => *'a \* 'b* ((2 {*-*,/ *-*})

**translations**

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

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

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

## 8.1 Inductive definition of all parts of a message

### inductive-set

$parts :: msg set => msg set$

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

**where**

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

Monotonicity

**lemma**  $parts-mono: G \subseteq H \implies parts(G) \subseteq parts(H)$   
 $\langle proof \rangle$

Equations hold because constructors are injective.

**lemma**  $Friend-image-eq [simp]: (Friend x \in Friend'A) = (x:A)$   
 $\langle proof \rangle$

**lemma**  $Key-image-eq [simp]: (Key x \in Key'A) = (x \in A)$   
 $\langle proof \rangle$

**lemma**  $Nonce-Key-image-eq [simp]: (Nonce x \notin Key'A)$   
 $\langle proof \rangle$

## 8.2 Inverse of keys

**lemma**  $invKey-eq [simp]: (invKey K = invKey K') = (K=K')$   
 $\langle proof \rangle$

## 8.3 keysFor operator

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

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

**lemma**  $keysFor-UN [simp]: keysFor (\bigcup_{i \in A}. H i) = (\bigcup_{i \in A}. keysFor (H i))$   
 $\langle proof \rangle$

Monotonicity

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

**lemma** *keysFor-insert-Agent [simp]*:  $\text{keysFor} (\text{insert} (\text{Agent } A) H) = \text{keysFor } H$   
(proof)

**lemma** *keysFor-insert-Nonce [simp]*:  $\text{keysFor} (\text{insert} (\text{Nonce } N) H) = \text{keysFor } H$   
(proof)

**lemma** *keysFor-insert-Number [simp]*:  $\text{keysFor} (\text{insert} (\text{Number } N) H) = \text{keysFor } H$   
(proof)

**lemma** *keysFor-insert-Key [simp]*:  $\text{keysFor} (\text{insert} (\text{Key } K) H) = \text{keysFor } H$   
(proof)

**lemma** *keysFor-insert-Hash [simp]*:  $\text{keysFor} (\text{insert} (\text{Hash } X) H) = \text{keysFor } H$   
(proof)

**lemma** *keysFor-insert-MPair [simp]*:  $\text{keysFor} (\text{insert} \{\!\{X, Y\}\!\} H) = \text{keysFor } H$   
(proof)

**lemma** *keysFor-insert-Crypt [simp]*:  
 $\text{keysFor} (\text{insert} (\text{Crypt } K X) H) = \text{insert} (\text{invKey } K) (\text{keysFor } H)$   
(proof)

**lemma** *keysFor-image-Key [simp]*:  $\text{keysFor} (\text{Key}'E) = \{\}$   
(proof)

**lemma** *Crypt-imp-invKey-keysFor*:  $\text{Crypt } K X \in H \implies \text{invKey } K \in \text{keysFor } H$   
(proof)

## 8.4 Inductive relation "parts"

**lemma** *MPair-parts*:  
 $\llbracket \{\!\{X, Y\}\!\} \in \text{parts } H; \llbracket X \in \text{parts } H; Y \in \text{parts } H \rrbracket \implies P \rrbracket \implies P$   
(proof)

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

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

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



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

**lemma** *parts-empty* [*simp*]:  $\text{parts}\{\} = \{\}$   
*<proof>*

**lemma** *parts-emptyE* [*elim!*]:  $X \in \text{parts}\{\} \implies P$   
*<proof>*

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

**lemma** *parts-singleton*:  $X \in \text{parts } H \implies \exists Y \in H. X \in \text{parts } \{Y\}$   
*<proof>*

### 8.4.1 Unions

**lemma** *parts-Un-subset1*:  $\text{parts}(G) \cup \text{parts}(H) \subseteq \text{parts}(G \cup H)$   
*<proof>*

**lemma** *parts-Un-subset2*:  $\text{parts}(G \cup H) \subseteq \text{parts}(G) \cup \text{parts}(H)$   
*<proof>*

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

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

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

**lemma** *parts-insert2*:  
 $\text{parts}(\text{insert } X \ (\text{insert } Y \ H)) = \text{parts } \{X\} \cup \text{parts } \{Y\} \cup \text{parts } H$   
*<proof>*

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*:  $\text{insert } X \ (\text{parts } H) \subseteq \text{parts}(\text{insert } X \ H)$   
*<proof>*

### 8.4.2 Idempotence and transitivity

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

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

*<proof>*

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

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

Cut

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

**lemma** *parts-cut-eq* [*simp*]:  $X \in \text{parts } H \implies \text{parts } (\text{insert } X H) = \text{parts } H$   
*<proof>*

### 8.4.3 Rewrite rules for pulling out atomic messages

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

**lemma** *parts-insert-Agent* [*simp*]:  
 $\text{parts } (\text{insert } (\text{Agent } \text{agt}) H) = \text{insert } (\text{Agent } \text{agt}) (\text{parts } H)$   
*<proof>*

**lemma** *parts-insert-Nonce* [*simp*]:  
 $\text{parts } (\text{insert } (\text{Nonce } N) H) = \text{insert } (\text{Nonce } N) (\text{parts } H)$   
*<proof>*

**lemma** *parts-insert-Number* [*simp*]:  
 $\text{parts } (\text{insert } (\text{Number } N) H) = \text{insert } (\text{Number } N) (\text{parts } H)$   
*<proof>*

**lemma** *parts-insert-Key* [*simp*]:  
 $\text{parts } (\text{insert } (\text{Key } K) H) = \text{insert } (\text{Key } K) (\text{parts } H)$   
*<proof>*

**lemma** *parts-insert-Hash* [*simp*]:  
 $\text{parts } (\text{insert } (\text{Hash } X) H) = \text{insert } (\text{Hash } X) (\text{parts } H)$   
*<proof>*

**lemma** *parts-insert-Crypt* [*simp*]:  
 $\text{parts } (\text{insert } (\text{Crypt } K X) H) = \text{insert } (\text{Crypt } K X) (\text{parts } (\text{insert } X H))$   
*<proof>*

**lemma** *parts-insert-MPair* [*simp*]:  
 $\text{parts } (\text{insert } \{X, Y\} H) =$   
 $\text{insert } \{X, Y\} (\text{parts } (\text{insert } X (\text{insert } Y H)))$

*<proof>*

**lemma** *parts-image-Key* [simp]:  $parts (Key'N) = Key'N$   
*<proof>*

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

**lemma** *msg-Nonce-supply*:  $\exists N. \forall n. N \leq n \longrightarrow Nonce\ n \notin parts\ \{msg\}$   
*<proof>*

## 8.5 Inductive relation "analz"

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

**inductive-set**

*analz* :: *msg set* ==> *msg set*

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

**where**

*Inj* [intro,simp] :  $X \in H \implies X \in analz\ H$

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

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

| *Decrypt* [dest]:

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

Monotonicity; Lemma 1 of Lowe's paper

**lemma** *analz-mono*:  $G \subseteq H \implies analz(G) \subseteq analz(H)$   
*<proof>*

Making it safe speeds up proofs

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

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

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

$] \implies P$

*<proof>*

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

*<proof>*

**lemma** *analz-subset-parts*:  $analz\ H \subseteq parts\ H$

*<proof>*

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

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

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

*<proof>*

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

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

### 8.5.1 General equational properties

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

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

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

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

### 8.5.2 Rewrite rules for pulling out atomic messages

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

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

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

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

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

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

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

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

Can pull out enCrypted message if the Key is not known

**lemma** *analz-insert-Crypt*:

$Key (invKey K) \notin analz H$   
 $\implies analz (insert (Crypt K X) H) = insert (Crypt K X) (analz H)$   
 ⟨proof⟩

**lemma** *lemma1*:  $Key (invKey K) \in analz H \implies$

$analz (insert (Crypt K X) H) \subseteq$   
 $insert (Crypt K X) (analz (insert X H))$   
 ⟨proof⟩

**lemma** *lemma2*:  $Key (invKey K) \in analz H \implies$

$insert (Crypt K X) (analz (insert X H)) \subseteq$   
 $analz (insert (Crypt K X) H)$   
 ⟨proof⟩

**lemma** *analz-insert-Decrypt*:

$Key (invKey K) \in analz H \implies$   
 $analz (insert (Crypt K X) H) =$   
 $insert (Crypt K X) (analz (insert X H))$   
 ⟨proof⟩

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

**lemma** *analz-Crypt-if [simp]*:

$analz (insert (Crypt K X) H) =$   
 $(if (Key (invKey K) \in analz H)$   
 $then insert (Crypt K X) (analz (insert X H))$   
 $else insert (Crypt K X) (analz H))$   
 ⟨proof⟩

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

**lemma** *analz-insert-Crypt-subset*:

$analz (insert (Crypt K X) H) \subseteq$   
 $insert (Crypt K X) (analz (insert X H))$   
 ⟨proof⟩

**lemma** *analz-image-Key [simp]*:  $analz (Key'N) = Key'N$

⟨proof⟩

### 8.5.3 Idempotence and transitivity

**lemma** *analz-analzD [dest!]*:  $X \in analz (analz H) \implies X \in analz H$

⟨proof⟩

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

*<proof>*

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

**lemma** *analz-trans*:  $[[ X \in \text{analz } G; G \subseteq \text{analz } H ]] \implies X \in \text{analz } H$   
*<proof>*

Cut; Lemma 2 of Lowe

**lemma** *analz-cut*:  $[[ Y \in \text{analz } (\text{insert } X H); X \in \text{analz } H ]] \implies Y \in \text{analz } H$   
*<proof>*

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

**lemma** *analz-insert-eq*:  $X \in \text{analz } H \implies \text{analz } (\text{insert } X H) = \text{analz } H$   
*<proof>*

A congruence rule for "analz"

**lemma** *analz-subset-cong*:  
 $[[ \text{analz } G \subseteq \text{analz } G'; \text{analz } H \subseteq \text{analz } H' ]]$   
 $\implies \text{analz } (G \cup H) \subseteq \text{analz } (G' \cup H')$   
*<proof>*

**lemma** *analz-cong*:  
 $[[ \text{analz } G = \text{analz } G'; \text{analz } H = \text{analz } H' ]]$   
 $\implies \text{analz } (G \cup H) = \text{analz } (G' \cup H')$   
*<proof>*

**lemma** *analz-insert-cong*:  
 $\text{analz } H = \text{analz } H' \implies \text{analz}(\text{insert } X H) = \text{analz}(\text{insert } X H')$   
*<proof>*

If there are no pairs or encryptions then analz does nothing

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

## 8.6 Inductive relation "synth"

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

**inductive-set**  
*synth* :: *msg set* => *msg set*  
**for** *H* :: *msg set*

**where**

*Inj* [intro]:  $X \in H \implies X \in \text{synth } H$   
| *Agent* [intro]:  $\text{Agent } \text{agt} \in \text{synth } H$   
| *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*:  $G \subseteq H \implies \text{synth}(G) \subseteq \text{synth}(H)$   
*<proof>*

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*:  $H \subseteq \text{synth}(H)$   
*<proof>*

### 8.6.1 Unions

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

**lemma** *synth-Un*:  $\text{synth}(G) \cup \text{synth}(H) \subseteq \text{synth}(G \cup H)$   
*<proof>*

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

### 8.6.2 Idempotence and transitivity

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

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

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

**lemma** *synth-trans*:  $[[X \in \text{synth } G; G \subseteq \text{synth } H]] \implies X \in \text{synth } H$   
*<proof>*

Cut; Lemma 2 of Lowe

**lemma** *synth-cut*:  $[[ Y \in \text{synth} (\text{insert } X \ H); X \in \text{synth } H ]] \implies Y \in \text{synth } H$   
 $\langle \text{proof} \rangle$

**lemma** *Agent-synth* [*simp*]:  $\text{Agent } A \in \text{synth } H$   
 $\langle \text{proof} \rangle$

**lemma** *Number-synth* [*simp*]:  $\text{Number } n \in \text{synth } H$   
 $\langle \text{proof} \rangle$

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

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

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

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

### 8.6.3 Combinations of parts, analz and synth

**lemma** *parts-synth* [*simp*]:  $\text{parts} (\text{synth } H) = \text{parts } H \cup \text{synth } H$   
 $\langle \text{proof} \rangle$

**lemma** *analz-analz-Un* [*simp*]:  $\text{analz} (\text{analz } G \cup H) = \text{analz} (G \cup H)$   
 $\langle \text{proof} \rangle$

**lemma** *analz-synth-Un* [*simp*]:  $\text{analz} (\text{synth } G \cup H) = \text{analz} (G \cup H) \cup \text{synth } G$   
 $\langle \text{proof} \rangle$

**lemma** *analz-synth* [*simp*]:  $\text{analz} (\text{synth } H) = \text{analz } H \cup \text{synth } H$   
 $\langle \text{proof} \rangle$

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

**lemma** *parts-insert-subset-Un*:  $X \in G \implies \text{parts}(\text{insert } X \ H) \subseteq \text{parts } G \cup \text{parts } H$   
 $\langle \text{proof} \rangle$

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

**lemma** *Fake-parts-insert*:  
 $X \in \text{synth} (\text{analz } H) \implies$   
 $\text{parts} (\text{insert } X \ H) \subseteq \text{synth} (\text{analz } H) \cup \text{parts } H$   
 $\langle \text{proof} \rangle$



**lemma** *Fake-parts-insert-in-Un*:  

$$\llbracket Z \in \text{parts} (\text{insert } X \ H); \ X: \text{synth} (\text{analz } H) \rrbracket$$

$$\implies Z \in \text{synth} (\text{analz } H) \cup \text{parts } H$$
 $\langle \text{proof} \rangle$

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

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

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

$$\text{analz} (\text{insert } X \ H) \subseteq \text{synth} (\text{analz } G) \cup \text{analz} (G \cup H)$$
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

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

$$(X \in \text{synth} (\text{analz } H) \wedge Y \in \text{synth} (\text{analz } H))$$
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

## 8.7 HPair: a combination of Hash and MPair

### 8.7.1 Freeness

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

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

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

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

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

**lemma** *Crypt-neq-HPair*:  $Crypt\ K\ X' \sim = Hash[X]\ Y$   
 ⟨proof⟩

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

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

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

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

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

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

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

**lemma** *parts-insert-HPair* [simp]:  
 $parts\ (insert\ (Hash[X]\ Y)\ H) =$   
 $insert\ (Hash[X]\ Y)\ (insert\ (Hash\{X, Y\})\ (parts\ (insert\ Y\ H)))$   
 ⟨proof⟩

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

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

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

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

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

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

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

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

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

## 8.8 The set of key-free messages

**inductive-set**

*keyfree* :: *msg set*

**where**

*Agent*: *Agent A* ∈ *keyfree*

| *Number*: *Number N* ∈ *keyfree*

| *Nonce*: *Nonce N* ∈ *keyfree*

| *Hash*: *Hash X* ∈ *keyfree*

| *MPair*: [|*X* ∈ *keyfree*; *Y* ∈ *keyfree*] ==> {|*X*, *Y*} ∈ *keyfree*

| *Crypt*: [|*X* ∈ *keyfree*] ==> *Crypt K X* ∈ *keyfree*

**declare** *keyfree.intros* [intro]

**inductive-cases** *keyfree-KeyE*: *Key K* ∈ *keyfree*

**inductive-cases** *keyfree-MPairE*: {|*X*, *Y*} ∈ *keyfree*

**inductive-cases** *keyfree-CryptE*: *Crypt K X* ∈ *keyfree*

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

**lemma** *analz-keyfree-into-Un*:  $\llbracket X \in \text{analz } (G \cup H); G \subseteq \text{keyfree} \rrbracket \implies X \in \text{parts } G \cup \text{analz } H$   
 <proof>

## 8.9 Tactics useful for many protocol proofs

<ML>

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

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

**lemma** *Crypt-notin-image-Key* [*simp*]:  $\text{Crypt } K \ X \notin \text{Key } 'A$   
 <proof>

**lemma** *Hash-notin-image-Key* [*simp*]:  $\text{Hash } X \notin \text{Key } 'A$   
 <proof>

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

**lemma** *Fake-analz-eq* [*simp*]:  
 $X \in \text{synth}(\text{analz } H) \implies \text{synth } (\text{analz } (\text{insert } X \ H)) = \text{synth } (\text{analz } H)$   
 <proof>

Two generalizations of *analz-insert-eq*

**lemma** *gen-analz-insert-eq* [*rule-format*]:  
 $X \in \text{analz } H \implies \forall G. H \subseteq G \longrightarrow \text{analz } (\text{insert } X \ G) = \text{analz } G$   
 <proof>

**lemma** *synth-analz-insert-eq* [*rule-format*]:  
 $X \in \text{synth } (\text{analz } H) \implies \forall G. H \subseteq G \longrightarrow (\text{Key } K \in \text{analz } (\text{insert } X \ G)) = (\text{Key } K \in \text{analz } G)$   
 <proof>

**lemma** *Fake-parts-sing*:  
 $X \in \text{synth } (\text{analz } H) \implies \text{parts}\{X\} \subseteq \text{synth } (\text{analz } H) \cup \text{parts } H$   
 <proof>

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

<ML>

**end**

## 9 Theory of Events for Security Protocols against the General Attacker

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

**consts**

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

**datatype**

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

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

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

| *knows-Cons*:

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

(*case ev of*

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

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

| *Notes A' X* =>

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

**primrec**

*used* :: *event list* => *msg set* **where**

*used-Nil*: *used* [] = (*UN B. parts (initState B)*)

| *used-Cons*: *used* (*ev* # *evs*) =

(*case ev of*

*Says A B X* => *parts {X} ∪ used evs*

| *Gets A X* => *used evs*

| *Notes A X* => *parts {X} ∪ used evs*)

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

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

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

### 9.1 Function *knows*

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

**lemma** *knows-Says* [*simp*]:

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

⟨*proof*⟩

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

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

Everybody sees what is sent on the traffic

**lemma** *Says-imp-knows* [*rule-format*]:  
 $Says\ A'\ B\ X \in set\ evs \longrightarrow (\forall A. X \in knows\ A\ evs)$   
 ⟨*proof*⟩

**lemma** *Notes-imp-knows* [*rule-format*]:  
 $Notes\ A'\ X \in set\ evs \longrightarrow X \in knows\ A'\ evs$   
 ⟨*proof*⟩

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

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

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

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

## 9.2 Knowledge of generic agents

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

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

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

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

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

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

**lemma** *initState-into-used*:  $X \in parts\ (initState\ B) \implies X \in used\ evs$

*<proof>*

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

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

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

**lemma** *used-nil-subset*: *used [] ⊆ used evs*  
*<proof>*

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

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

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

**lemma** *knows-subset-knows-Cons*: *knows A evs ⊆ knows A (e # evs)*  
*<proof>*

**lemma** *initState-subset-knows*: *initState A ⊆ knows A evs*  
*<proof>*

For proving *new-keys-not-used*

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

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

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

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

*<ML>*

**end**

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

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

**lemma** *invKey-K*:  $K \in \text{symKeys} \implies \text{invKey } K = K$   
(*proof*)

### 10.1 Asymmetric Keys

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

**consts**

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

**abbreviation**

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

**abbreviation**

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

**abbreviation**

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

**abbreviation**

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

**abbreviation**

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

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

**abbreviation**

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

**abbreviation**

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

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

**specification** (*publicKey*)



*injective-publicKey*:  
 $publicKey\ b\ A = publicKey\ c\ A' \implies b=c \wedge A=A'$   
 ⟨proof⟩

**axiomatization where**

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

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

## 10.2 Basic properties of *pubK* and *priEK*

**lemma** *publicKey-inject* [iff]:  $(publicKey\ b\ A = publicKey\ c\ A') = (b=c \wedge A=A')$   
 ⟨proof⟩

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

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

**lemma** *symKey-neq-priEK*:  $K \in symKeys \implies K \neq priEK\ A$   
 ⟨proof⟩

**lemma** *symKeys-neq-imp-neq*:  $(K \in symKeys) \neq (K' \in symKeys) \implies K \neq K'$   
 ⟨proof⟩

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

**lemma** *analz-symKeys-Decrypt*:  
 [ Crypt  $K\ X \in analz\ H$ ;  $K \in symKeys$ ;  $Key\ K \in analz\ H$  ]  
 $\implies X \in analz\ H$   
 ⟨proof⟩

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

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

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

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

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

$$(privateKey\ b\ A \in invKey\ 'c\ AS) = (b=c \wedge A \in AS)$$

$\langle proof \rangle$

**lemma** *publicKey-notin-image-privateKey* [simp]:  $publicKey\ b\ A \notin invKey\ 'c\ AS$

$\langle proof \rangle$

## 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

$\langle proof \rangle$

**axiomatization where**

*sym-shrK* [iff]:  $shrK\ X \in symKeys$  — All shared keys are symmetric

Injectiveness: Agents' long-term keys are distinct.

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

**declare** *shrK-injective* [iff]

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

$\langle proof \rangle$

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

$$[| Crypt\ (shrK\ A)\ X \in analz\ H; Key\ (shrK\ A) \in analz\ H |] ==> X \in analz\ H$$

$\langle proof \rangle$

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

$$[| Crypt\ K\ X \in analz\ H; K \in symKeys; Key\ K \in analz\ H |] ==> X \in analz\ H$$

$\langle proof \rangle$

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

$\langle proof \rangle$

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

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

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

$\langle proof \rangle$

**lemmas**  $shrK\text{-neq-pubK} = pubK\text{-neq-shrK}$  [THEN not-sym]

**declare**  $shrK\text{-neq-pubK}$  [simp]

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

**declare**  $invKey\text{-K}$  [simp]

## 10.5 Initial States of Agents

**overloading**

$initState \equiv initState$

**begin**

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

$initState\text{-Friend}$ :

$initState (Friend i) =$

$\{Key (priEK (Friend i)), Key (priSK (Friend i)), Key (shrK (Friend i))\} \cup$   
 $(Key \text{ ` } range pubEK) \cup (Key \text{ ` } range pubSK)$

**end**

**lemma**  $used\text{-parts-subset-parts}$  [rule-format]:

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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!]:  

$$\begin{aligned} & \llbracket \{X, Y\} \in \text{used } H; \\ & \quad \llbracket X \in \text{used } H; Y \in \text{used } H \rrbracket \implies P \rrbracket \\ & \implies P \end{aligned}$$
 $\langle \text{proof} \rangle$

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

**lemma** *keysFor-parts-initState* [simp]:  $\text{keysFor } (\text{parts } (\text{initState } C)) = \{\}$   
 $\langle \text{proof} \rangle$

**lemma** *Crypt-notin-initState*:  $\text{Crypt } K X \notin \text{parts } (\text{initState } B)$   
 $\langle \text{proof} \rangle$

**lemma** *Crypt-notin-used-empty* [simp]:  $\text{Crypt } K X \notin \text{used } []$   
 $\langle \text{proof} \rangle$

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

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

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

**lemma** *Key-not-used* [simp]:  $\text{Key } K \notin \text{used evs} \implies K \notin \text{range } \text{shrK}$   
 $\langle \text{proof} \rangle$

**lemma** *shrK-neq*:  $\text{Key } K \notin \text{used evs} \implies \text{shrK } B \neq K$   
 $\langle \text{proof} \rangle$

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

## 10.6 Function *knows Spy*

**lemma** *not-SignatureE* [elim!]:  $b \neq \text{Signature} \implies b = \text{Encryption}$   
 $\langle \text{proof} \rangle$

Agents see their own private keys!

**lemma** *priK-in-initState* [iff]:  $\text{Key } (\text{privateKey } b A) \in \text{initState } A$

*<proof>*

Agents see all public keys!

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

All public keys are visible

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

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

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

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

**lemma** *Crypt-analz-bad*:  
[[ *Crypt (shrK A) X*  $\in$  *analz (knows A evs)* ]]  
==>  $X \in$  *analz (knows A evs)*  
*<proof>*

## 10.7 Fresh Nonces

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

**lemma** *Nonce-notin-used-empty* [simp]:  $Nonce\ N \notin used\ []$   
*<proof>*

## 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*:  $\exists N. \forall n. N \leq n \longrightarrow Nonce\ n \notin used\ evs$   
*<proof>*

**lemma** *Nonce-supply1*:  $\exists N. Nonce\ N \notin used\ evs$   
*<proof>*

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

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

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

**lemma** *insert-Key-image*:  $insert (Key K) (Key \text{ ' } KK \cup C) = Key \text{ ' } (insert K KK) \cup C$   
 ⟨proof⟩

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

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

**lemma** *analz-image-freshK-lemma*:  
 $(Key K \in analz (Key \text{ ' } nE \cup H)) \longrightarrow (K \in nE \mid Key K \in analz H) \implies$   
 $(Key K \in analz (Key \text{ ' } nE \cup H)) = (K \in nE \mid Key K \in analz H)$   
 ⟨proof⟩

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

⟨ML⟩

## 10.10 Specialized Methods for Possibility Theorems

⟨ML⟩

end

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

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

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

*Nil*:  $[] \in ns-public$

| *Fake*:  $\llbracket \text{evsf} \in \text{ns-public}; X \in \text{synth}(\text{analz}(\text{knows } A \text{ evsf})) \rrbracket$   
 $\implies \text{Says } A \ B \ X \ \# \ \text{evsf} \in \text{ns-public}$

| *Reception*:  $\llbracket \text{evsr} \in \text{ns-public}; \text{Says } A \ B \ X \in \text{set evsr} \rrbracket$   
 $\implies \text{Gets } B \ X \ \# \ \text{evsr} \in \text{ns-public}$

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

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

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

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

$\text{evs} \in \text{ns-public} \implies \text{Notes } A \ X \notin \text{set evs}$   
 $\langle \text{proof} \rangle$

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*:  $\text{parts}(\text{initState } C) = \text{initState } C$   
 $\langle \text{proof} \rangle$

**lemma** *analz-initState*:  $\text{analz}(\text{initState } C) = \text{initState } C$   
 $\langle \text{proof} \rangle$

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

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

More subsidiary lemmas combining initial secrets and knowledge of generic agent

**lemma** *staticSecret-in-initState* [simp]:

$staticSecret\ A \subseteq initState\ A$

$\langle proof \rangle$

**thm** *parts-insert*

**lemma** *staticSecretA-notin-initStateB*:

$m \in staticSecret\ A \implies m \in initState\ B = (A=B)$

$\langle proof \rangle$

**lemma** *staticSecretA-notin-parts-initStateB*:

$m \in staticSecret\ A \implies m \in parts(initState\ B) = (A=B)$

$\langle proof \rangle$

**lemma** *staticSecretA-notin-analz-initStateB*:

$m \in staticSecret\ A \implies m \in analz(initState\ B) = (A=B)$

$\langle proof \rangle$

**lemma** *staticSecret-synth-eq*:

$m \in staticSecret\ A \implies (m \in synth\ H) = (m \in H)$

$\langle proof \rangle$

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

**lemma** *nonce-notin-analz-initState*:

$Nonce\ N \notin analz(initState\ A)$

$\langle proof \rangle$

## 12.1 Protocol independent study

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

$\llbracket m \in parts\ (knows\ C\ evs); m \in staticSecret\ A \rrbracket \implies$   
 $A=C \vee$   
 $(\exists D\ E\ X. Says\ D\ E\ X \in set\ evs \wedge m \in parts\{X\}) \vee$   
 $(\exists Y. Notes\ C\ Y \in set\ evs \wedge m \in parts\{Y\})$   
 $\langle proof \rangle$

**lemma** *staticSecret-analz-agent*:

$\llbracket m \in analz\ (knows\ C\ evs); m \in staticSecret\ A \rrbracket \implies$   
 $A=C \vee$   
 $(\exists D\ E\ X. Says\ D\ E\ X \in set\ evs \wedge m \in parts\{X\}) \vee$   
 $(\exists Y. Notes\ C\ Y \in set\ evs \wedge m \in parts\{Y\})$   
 $\langle proof \rangle$

**lemma** *secret-parts-agent*:

$m \in parts\ (knows\ C\ evs) \implies m \in initState\ C \vee$   
 $(\exists A\ B\ X. Says\ A\ B\ X \in set\ evs \wedge m \in parts\{X\}) \vee$   
 $(\exists Y. Notes\ C\ Y \in set\ evs \wedge m \in parts\{Y\})$   
 $\langle proof \rangle$



## 12.2 Protocol dependent study

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

$$\begin{aligned} & \llbracket m \in \text{parts}(\text{knows } C \text{ evs}); m \in \text{staticSecret } A; \\ & \quad \text{evs} \in \text{ns-public} \rrbracket \implies \\ & \quad A=C \vee (\exists D E X. \text{Says } D E X \in \text{set evs} \wedge m \in \text{parts}\{X\}) \\ & \langle \text{proof} \rangle \end{aligned}$$

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:*

$$\begin{aligned} & \llbracket m \in \text{parts}(\text{knows } C \text{ evs}); m \in \text{staticSecret } A; A \neq C; \text{evs} \in \text{ns-public} \rrbracket \implies \\ & \quad m \in \text{parts}(\text{knows } D \text{ evs}) \\ & \langle \text{proof} \rangle \end{aligned}$$

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:*

$$\begin{aligned} & \llbracket m \in \text{parts}(\text{knows } C \text{ evs}); m \in \text{staticSecret } A; \\ & \quad C \neq A; \text{evs} \in \text{ns-public} \rrbracket \\ & \implies \exists B X. \text{Says } A B X \in \text{set evs} \wedge m \in \text{parts} \{X\} \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *NS-agent-see-staticSecret:*

$$\begin{aligned} & \llbracket m \in \text{staticSecret } A; C \neq A; \text{evs} \in \text{ns-public} \rrbracket \\ & \implies m \in \text{parts}(\text{knows } C \text{ evs}) = (\exists B X. \text{Says } A B X \in \text{set evs} \wedge m \in \text{parts} \{X\}) \\ & \langle \text{proof} \rangle \end{aligned}$$

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

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

$$\llbracket c \notin \text{parts}\{Z\}; \forall K. \text{Key } K \notin \text{parts}\{Z\}; c \in \text{analz}(\text{insert } Z H) \rrbracket \implies c \in \text{analz } H$$

$\langle \text{proof} \rangle$

**lemma** *Agent-not-see-NA:*

$$\begin{aligned} & \llbracket \text{Key}(\text{priEK } B) \notin \text{analz}(\text{knows } C \text{ evs}); \\ & \quad \text{Key}(\text{priEK } A) \notin \text{analz}(\text{knows } C \text{ evs}); \\ & \quad \forall S R Y. \text{Says } S R Y \in \text{set evs} \longrightarrow \\ & \quad \quad Y = \text{Crypt}(\text{pubEK } B) \{\text{Nonce } NA, \text{Agent } A\} \vee \\ & \quad \quad Y = \text{Crypt}(\text{pubEK } A) \{\text{Nonce } NA, \text{Nonce } NB\} \vee \\ & \quad \quad \text{Nonce } NA \notin \text{parts}\{Y\} \wedge (\forall K. \text{Key } K \notin \text{parts}\{Y\}); \\ & \quad C \neq A; C \neq B; \text{evs} \in \text{ns-public} \rrbracket \\ & \implies \text{Nonce } NA \notin \text{analz}(\text{knows } C \text{ evs}) \end{aligned}$$

*<proof>*

**end**

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

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

**theorem** *knowledge-equiv:*

$\llbracket X \in \text{knows } A \text{ evs}; \text{Notes } A \ X \notin \text{set evs};$   
 $X \notin \{\text{Key } (\text{priEK } A), \text{Key } (\text{priSK } A), \text{Key } (\text{shrK } A)\} \rrbracket$   
 $\implies X \in \text{knows } B \text{ evs}$   
*<proof>*

**lemma** *knowledge-equiv-bis:*

$\llbracket X \in \text{knows } A \text{ evs}; \text{Notes } A \ X \notin \text{set evs} \rrbracket$   
 $\implies X \in \{\text{Key } (\text{priEK } A), \text{Key } (\text{priSK } A), \text{Key } (\text{shrK } A)\} \cup \text{knows } B \text{ evs}$   
*<proof>*

**lemma** *knowledge-equiv-ter:*

$\llbracket X \in \text{knows } A \text{ evs}; X \notin \{\text{Key } (\text{priEK } A), \text{Key } (\text{priSK } A), \text{Key } (\text{shrK } A)\} \rrbracket$   
 $\implies X \in \text{knows } B \text{ evs} \vee \text{Notes } A \ X \in \text{set evs}$   
*<proof>*

**lemma** *knowledge-equiv-quater:*

$X \in \text{knows } A \text{ evs}$   
 $\implies X \in \text{knows } B \text{ evs} \vee \text{Notes } A \ X \in \text{set evs} \vee$   
 $X \in \{\text{Key } (\text{priEK } A), \text{Key } (\text{priSK } A), \text{Key } (\text{shrK } A)\}$   
*<proof>*

**lemma** *setdiff-diff-insert:*  $A-B-C=D-E-F \implies \text{insert } m \ (A-B-C) = \text{insert } m \ (D-E-F)$   
*<proof>*

**lemma**  $A-B-C=D-E-F \implies \text{insert } m \ A-B-C = \text{insert } m \ D-E-F$   
*<proof>*

**lemma** *knowledge-equiv-eq-setdiff:*  
*knows } A \text{ evs} -*

$$\begin{aligned}
& \{Key (priEK A), Key (priSK A), Key (shrK A)\} - \\
& \{X. Notes A X \in set evs\} \\
= & \\
& knows B evs - \\
& \{Key (priEK B), Key (priSK B), Key (shrK B)\} - \\
& \{X. Notes B X \in set evs\} \\
\langle proof \rangle
\end{aligned}$$

**lemma** *knowledge-equiv-eq-old:*

$$\begin{aligned}
& knows A evs \cup \\
& \{Key (priEK B), Key (priSK B), Key (shrK B)\} \cup \\
& \{X. Notes B X \in set evs\} \\
= & \\
& knows B evs \cup \\
& \{Key (priEK A), Key (priSK A), Key (shrK A)\} \cup \\
& \{X. Notes A X \in set evs\} \\
\langle proof \rangle
\end{aligned}$$

**theorem** *knowledge-eval: knows A evs =*

$$\begin{aligned}
& \{Key (priEK A), Key (priSK A), Key (shrK A)\} \cup \\
& (Key ' range pubEK) \cup (Key ' range pubSK) \cup \\
& \{X. \exists S R. Says S R X \in set evs\} \cup \\
& \{X. Notes A X \in set evs\} \\
\langle proof \rangle
\end{aligned}$$

**lemma** *knowledge-eval-setdiff:*

$$\begin{aligned}
& knows A evs - \\
& \{Key (priEK A), Key (priSK A), Key (shrK A)\} - \\
& \{X. Notes A X \in set evs\} \\
= & \\
& (Key ' range pubEK) \cup (Key ' range pubSK) \cup \\
& \{X. \exists S R. Says S R X \in set evs\} \\
\langle proof \rangle
\end{aligned}$$

**theorem** *knowledge-equiv-eq: knows A evs \cup*

$$\begin{aligned}
& \{Key (priEK B), Key (priSK B), Key (shrK B)\} \cup \\
& \{X. Notes B X \in set evs\} \\
= & \\
& knows B evs \cup \\
& \{Key (priEK A), Key (priSK A), Key (shrK A)\} \cup \\
& \{X. Notes A X \in set evs\} \\
\langle proof \rangle
\end{aligned}$$

**lemma** *knows A evs \cup*

$$\begin{aligned}
& \{Key (priEK B), Key (priSK B), Key (shrK B)\} \cup \\
& \{X. Notes B X \in set evs\} -
\end{aligned}$$

(  $\{Key (priEK B), Key (priSK B), Key (shrK B)\} \cup$   
 $\{X. Notes B X \in set evs\}$  ) =  $knows A evs$   
 $\langle proof \rangle$

**theorem** *parts-knowledge-equiv-eq*:

$parts(knows A evs) \cup$   
 $\{Key (priEK B), Key (priSK B), Key (shrK B)\} \cup$   
 $parts(\{X. Notes B X \in set evs\})$   
 $=$   
 $parts(knows B evs) \cup$   
 $\{Key (priEK A), Key (priSK A), Key (shrK A)\} \cup$   
 $parts(\{X. Notes A X \in set evs\})$   
 $\langle proof \rangle$

**lemmas** *parts-knowledge-equiv = parts-knowledge-equiv-eq* [THEN equalityD1, THEN subsetD]

**thm** *parts-knowledge-equiv*

**theorem** *noprishr-parts-knowledge-equiv*:

$\llbracket X \notin \{Key (priEK A), Key (priSK A), Key (shrK A)\};$   
 $X \in parts(knows A evs) \rrbracket$   
 $\implies X \in parts(knows B evs) \cup$   
 $parts(\{X. Notes A X \in set evs\})$   
 $\langle proof \rangle$

**lemma** *knowledge-equiv-eq-NS*:

$evs \in ns-public \implies$   
 $knows A evs \cup \{Key (priEK B), Key (priSK B), Key (shrK B)\} =$   
 $knows B evs \cup \{Key (priEK A), Key (priSK A), Key (shrK A)\}$   
 $\langle proof \rangle$

**lemma** *parts-knowledge-equiv-eq-NS*:

$evs \in ns-public \implies$   
 $parts(knows A evs) \cup \{Key (priEK B), Key (priSK B), Key (shrK B)\} =$   
 $parts(knows B evs) \cup \{Key (priEK A), Key (priSK A), Key (shrK A)\}$   
 $\langle proof \rangle$

**theorem** *noprishr-parts-knowledge-equiv-NS*:

$\llbracket X \notin \{Key (priEK A), Key (priSK A), Key (shrK A)\};$   
 $X \in parts(knows A evs); evs \in ns-public \rrbracket$   
 $\implies X \in parts(knows B evs)$   
 $\langle proof \rangle$

**theorem** *Agent-not-analz-N*:

$\llbracket \text{Nonce } N \notin \text{parts}(\text{knows } A \text{ evs}); \text{ evs} \in \text{ns-public} \rrbracket$   
 $\implies \text{Nonce } N \notin \text{analz}(\text{knows } B \text{ evs})$   
 $\langle \text{proof} \rangle$

**end**

## References

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