Protocol Verification Techniques - Theorem Provers

Design and Verification of Security Protocols and Security Ceremonies

Programa de Pós-Graduacão em Ciências da Computacão
Dr. Jean Everson Martina

March-June 2018

Programa de Pós-Graduação em Ciência da Computação UFSC

UNIVERSIDADE FEDERAL DE SANTA CATARINA

Attention!

Attention!

This topic will be divided into two lectures. One will deal with automatic theorem provers using FOL and the second will deal with theorem provers using HOL

Higher-Order Logic

- Higher-order logic is a form of predicate logic that is distinguished from first-order logic by additional quantifiers and stronger semantics;

Higher-Order Logic

- Higher-order logic is a form of predicate logic that is distinguished from first-order logic by additional quantifiers and stronger semantics;
- Higher-order logics semantics are more expressive, but their model-theoretic properties are less well-behaved than those of first-order logic;

Higher-Order Logic

- Higher-order logic is a form of predicate logic that is distinguished from first-order logic by additional quantifiers and stronger semantics;
- Higher-order logics semantics are more expressive, but their model-theoretic properties are less well-behaved than those of first-order logic;
- The term "higher-order logic", abbreviated as HOL;

Higher-Order Logic

- Higher-order logic is a form of predicate logic that is distinguished from first-order logic by additional quantifiers and stronger semantics;
- Higher-order logics semantics are more expressive, but their model-theoretic properties are less well-behaved than those of first-order logic;
- The term "higher-order logic", abbreviated as HOL;
- HOL is any predicate logic that has greater order than Second-Order Logic;

Higher-Order Logic

- Second-Order Logic stands for the possibility of quantifying over sets;

Higher-Order Logic

- Second-Order Logic stands for the possibility of quantifying over sets;
- The idea is that you can put a quantifier over other quantifier;

Higher-Order Logic

- Second-Order Logic stands for the possibility of quantifying over sets;
- The idea is that you can put a quantifier over other quantifier;
- For example we can say $\forall x \exists P(x, y) \rightarrow y$

Higher-Order Logic

- Second-Order Logic stands for the possibility of quantifying over sets;
- The idea is that you can put a quantifier over other quantifier;
- For example we can say $\forall x \exists P(x, y) \rightarrow y$
- Third-Order Logic allows for quantification over Second-Order Logic;

Higher-Order Logic

- Second-Order Logic stands for the possibility of quantifying over sets;
- The idea is that you can put a quantifier over other quantifier;
- For example we can say $\forall x \exists P(x, y) \rightarrow y$
- Third-Order Logic allows for quantification over Second-Order Logic;
- Higher-order logic is the union of first-, second-, third-, ..., nth-order logic;

Higher-Order Logic

- Second-Order Logic stands for the possibility of quantifying over sets;
- The idea is that you can put a quantifier over other quantifier;
- For example we can say $\forall x \exists P(x, y) \rightarrow y$
- Third-Order Logic allows for quantification over Second-Order Logic;
- Higher-order logic is the union of first-, second-, third-, ..., nth-order logic;
- Higher-order logic admits quantification over sets that are nested arbitrarily deeply.

HOL Example

- First-order logic quantifies only variables that range over individuals;

HOL Example

- First-order logic quantifies only variables that range over individuals;
- Second-order logic, in addition, quantifies over sets;

HOL Example

- First-order logic quantifies only variables that range over individuals;
- Second-order logic, in addition, quantifies over sets;
- Third-order logic also quantifies over sets of sets, and so on;

HOL Example

- First-order logic quantifies only variables that range over individuals;
- Second-order logic, in addition, quantifies over sets;
- Third-order logic also quantifies over sets of sets, and so on;
- From Second-Order Logic on we are allowed to describe mathematical induction;

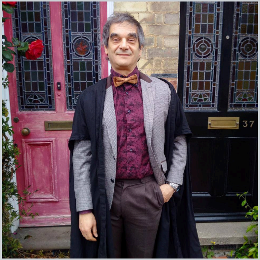
HOL Example

- First-order logic quantifies only variables that range over individuals;
- Second-order logic, in addition, quantifies over sets;
- Third-order logic also quantifies over sets of sets, and so on;
- From Second-Order Logic on we are allowed to describe mathematical induction;
- $\forall P((0 \in P \land \forall i(i \in P \rightarrow i + 1 \in P)) \rightarrow \forall n(n \in P))$

HOL Example

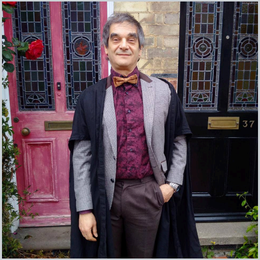
- First-order logic quantifies only variables that range over individuals;
- Second-order logic, in addition, quantifies over sets;
- Third-order logic also quantifies over sets of sets, and so on;
- From Second-Order Logic on we are allowed to describe mathematical induction;
- $\forall P((0 \in P \land \forall i(i \in P \rightarrow i + 1 \in P)) \rightarrow \forall n(n \in P))$
- This is the definition of the set of Natural Numbers.

Lawrence Charles Paulson - Larry



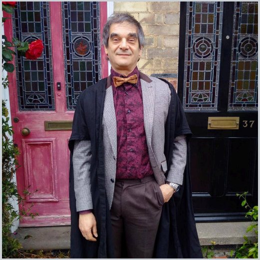
- Lawrence Charles Paulson (Larry) is a professor at the University of Cambridge;

Lawrence Charles Paulson - Larry



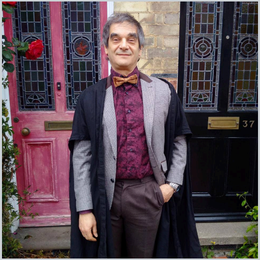
- Lawrence Charles Paulson (Larry) is a professor at the University of Cambridge;
- His research is based around the interactive theorem prover Isabelle, which he introduced in 1986;

Lawrence Charles Paulson - Larry



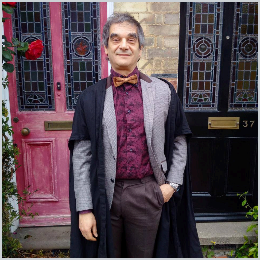
- Lawrence Charles Paulson (Larry) is a professor at the University of Cambridge;
- His research is based around the interactive theorem prover Isabelle, which he introduced in 1986;
- He has worked on the verification of cryptographic protocols using inductive definitions;

Lawrence Charles Paulson - Larry



- Lawrence Charles Paulson (Larry) is a professor at the University of Cambridge;
- His research is based around the interactive theorem prover Isabelle, which he introduced in 1986;
- He has worked on the verification of cryptographic protocols using inductive definitions;
- He has also formalized the constructible universe of Kurt Gödel;

Lawrence Charles Paulson - Larry



- Lawrence Charles Paulson (Larry) is a professor at the University of Cambridge;
- His research is based around the interactive theorem prover Isabelle, which he introduced in 1986;
- He has worked on the verification of cryptographic protocols using inductive definitions;
- He has also formalized the constructible universe of Kurt Gödel;

Curiosities about Larry

- He was one of the most cited researchers on a paper that demonstrated the existence of God by a machine on Gödel's world;

Curiosities about Larry

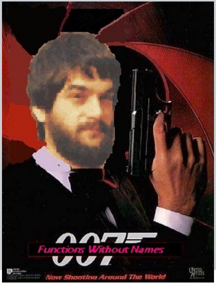
- He was one of the most cited researchers on a paper that demonstrated the existence of God by a machine on Gödel's world;
- He is a deep minded atheist;

Curiosities about Larry

- He was one of the most cited researchers on a paper that demonstrated the existence of God by a machine on Gödel's world;
- He is a deep minded atheist;
- He has a page called: "Larry Paulson - Portrait of a God";

Curiosities about Larry

- He was one of the most cited researchers on a paper that demonstrated the existence of God by a machine on Gödel's world;
- He is a deep minded atheist;
- He has a page called: "Larry Paulson - Portrait of a God";
- http://www.geocities.ws/robrich18/Larry.html



Larry's Protocol Verification Time-line

- Isabelle theorem prover;

Larry's Protocol Verification Time-line

- Isabelle theorem prover;
  - General tool;

Larry's Protocol Verification Time-line

- Isabelle theorem prover;
  - General tool;
  - Works with protocols since 1997;

Larry's Protocol Verification Time-line

- Isabelle theorem prover;
  - General tool;
  - Works with protocols since 1997;
- Many papers describing the Inductive Method he created;

Larry's Protocol Verification Time-line

- Isabelle theorem prover;
  - General tool;
  - Works with protocols since 1997;
- Many papers describing the Inductive Method he created;
- Many case studies, including:

Larry's Protocol Verification Time-line

- Isabelle theorem prover;
  - General tool;
  - Works with protocols since 1997;
- Many papers describing the Inductive Method he created;
- Many case studies, including:
  - Verification of SET protocol (6 papers)

Larry's Protocol Verification Time-line

- Isabelle theorem prover;
  - General tool;
  - Works with protocols since 1997;
- Many papers describing the Inductive Method he created;
- Many case studies, including:
  - Verification of SET protocol (6 papers)
  - Kerberos (3 papers)

Larry's Protocol Verification Time-line

- Isabelle theorem prover;
  - General tool;
  - Works with protocols since 1997;
- Many papers describing the Inductive Method he created;
- Many case studies, including:
  - Verification of SET protocol (6 papers)
  - Kerberos (3 papers)
  - TLS protocol

Larry's Protocol Verification Time-line

- Isabelle theorem prover;
  - General tool;
  - Works with protocols since 1997;
- Many papers describing the Inductive Method he created;
- Many case studies, including:
  - Verification of SET protocol (6 papers)
  - Kerberos (3 papers)
  - TLS protocol
  - Yahalom protocol, smart cards, etc

Larry's Protocol Verification Time-line

- Isabelle theorem prover;
  - General tool;
  - Works with protocols since 1997;
- Many papers describing the Inductive Method he created;
- Many case studies, including:
  - Verification of SET protocol (6 papers)
  - Kerberos (3 papers)
  - TLS protocol
  - Yahalom protocol, smart cards, etc
- Last work published in 2015: "Verifying multicast-based security protocols using the inductive method. Martina, J.E., Paulson, L.C."

The Inductive Method

- Starts with an informal protocol description;

The Inductive Method

- Starts with an informal protocol description;
- Out of that we extract:

The Inductive Method

- Starts with an informal protocol description;
- Out of that we extract:
  - An inductive abstract trace model;

The Inductive Method

- Starts with an informal protocol description;
- Out of that we extract:
  - An inductive abstract trace model;
  - Correctness theorem about the traces;

The Inductive Method

- Starts with an informal protocol description;
- Out of that we extract:
  - An inductive abstract trace model;
  - Correctness theorem about the traces;
- We then add the attacker inference rules;

The Inductive Method

- Starts with an informal protocol description;
- Out of that we extract:
  - An inductive abstract trace model;
  - Correctness theorem about the traces;
- We then add the attacker inference rules;
- We state the goals ad theorems and lemmas;

The Inductive Method

- Starts with an informal protocol description;
- Out of that we extract:
  - An inductive abstract trace model;
  - Correctness theorem about the traces;
- We then add the attacker inference rules;
- We state the goals ad theorems and lemmas;
- We prove the theorems inductively to demonstrate correctness.

The Inductive Method Mechanics

- Larry Paulson advocates a simple approach (called Inductive Method):

The Inductive Method Mechanics

- Larry Paulson advocates a simple approach (called Inductive Method):
  - A protocol in a context describes a set of traces;

The Inductive Method Mechanics

- Larry Paulson advocates a simple approach (called Inductive Method):
  - A protocol in a context describes a set of traces;
  - These traces are defined inductively;

The Inductive Method Mechanics

- Larry Paulson advocates a simple approach (called Inductive Method):
  - A protocol in a context describes a set of traces;
  - These traces are defined inductively;
  - A specification is again a property of traces;

The Inductive Method Mechanics

- Larry Paulson advocates a simple approach (called Inductive Method):
  - A protocol in a context describes a set of traces;
  - These traces are defined inductively;
  - A specification is again a property of traces;
  - Checking requires proving that all the traces satisfy the property, by induction on the construction of the traces;

The Inductive Method Mechanics

- Larry Paulson advocates a simple approach (called Inductive Method):
  - A protocol in a context describes a set of traces;
  - These traces are defined inductively;
  - A specification is again a property of traces;
  - Checking requires proving that all the traces satisfy the property, by induction on the construction of the traces;
  - Main point: these proofs are big, uninteresting, and better left to machines;

The Inductive Method Mechanics

- Larry Paulson advocates a simple approach (called Inductive Method):
  - A protocol in a context describes a set of traces;
  - These traces are defined inductively;
  - A specification is again a property of traces;
  - Checking requires proving that all the traces satisfy the property, by induction on the construction of the traces;
  - Main point: these proofs are big, uninteresting, and better left to machines;
  - Use a theorem prover (Isabelle)to write the proofs.

Isabelle

- Automated support for proof development, which supports:

Isabelle

- Automated support for proof development, which supports:
    - Higher-order logic;

Isabelle

- Automated support for proof development, which supports:
  - Higher-order logic;
  - Serves as a logical framework;

Isabelle

- Automated support for proof development, which supports:
  - Higher-order logic;
  - Serves as a logical framework;
  - Supports ZF set theory and HOL;

Isabelle

- Automated support for proof development, which supports:
  - Higher-order logic;
  - Serves as a logical framework;
  - Supports ZF set theory and HOL;
  - Generic treatment of inference rules;

Isabelle

- Automated support for proof development, which supports:
  - Higher-order logic;
  - Serves as a logical framework;
  - Supports ZF set theory and HOL;
  - Generic treatment of inference rules;
- Powerful simplifier, classical reasoner and connected tools;

Isabelle

- Automated support for proof development, which supports:
  - Higher-order logic;
  - Serves as a logical framework;
  - Supports ZF set theory and HOL;
  - Generic treatment of inference rules;
- Powerful simplifier, classical reasoner and connected tools;
- Strong support for inductive definitions.

Inductive Method Support in Isabelle

- There are several files to support security protocol verification in Isabelle:

Inductive Method Support in Isabelle

- There are several files to support security protocol verification in Isabelle:
    - *Message.thy*, which specifies how the messages exchanged in security protocols are constructed, as well as the main operators we use within the method;

Inductive Method Support in Isabelle

- There are several files to support security protocol verification in Isabelle:
  - *Message.thy*, which specifies how the messages exchanged in security protocols are constructed, as well as the main operators we use within the method;
  - *Event.thy*, which inherits from *Message.thy* and accounts for the specification of the communication layer with the sending, receiving and noting events;

Inductive Method Support in Isabelle

- There are several files to support security protocol verification in Isabelle:
  - *Message.thy*, which specifies how the messages exchanged in security protocols are constructed, as well as the main operators we use within the method;
  - *Event.thy*, which inherits from *Message.thy* and accounts for the specification of the communication layer with the sending, receiving and noting events;
  - *Public.thy* which inherits from *Event.thy*, which despite the narrowly chosen name, accounts for the specification of symmetric and asymmetric cryptographic primitives;

Inductive Method Support in Isabelle

- There are several files to support security protocol verification in Isabelle:
  - *Message.thy*, which specifies how the messages exchanged in security protocols are constructed, as well as the main operators we use within the method;
  - *Event.thy*, which inherits from *Message.thy* and accounts for the specification of the communication layer with the sending, receiving and noting events;
  - *Public.thy* which inherits from *Event.thy*, which despite the narrowly chosen name, accounts for the specification of symmetric and asymmetric cryptographic primitives;
  - We also have some other specialised theories for Smart-Cards, Threshold Cryptography and Multicast communication.

Inductive Method Details - Agents

Definition

Agent datatype definition

datatype

```
  agent = Friend nat
        | Server
        | Spy
```

Inductive Method Details - Agents

Definition

Agent datatype definition

datatype

```
  agent = Friend nat
        | Server
        | Spy
```

- First we have the definition of a friendly agent by the bijection explained above;

Inductive Method Details - Agents

Definition

Agent datatype definition

```
datatype
  agent = Friend nat
        | Server
        | Spy
```

- First we have the definition of a friendly agent by the bijection explained above;
- The second category regards the trusted third party;

Inductive Method Details - Agents

Definition

Agent datatype definition

```
datatype
  agent = Friend nat
        | Server
        | Spy
```

- First we have the definition of a friendly agent by the bijection explained above;
- The second category regards the trusted third party;
- Finally we have the attacker, which is categorised separately.

Inductive Method Details - Cryptographic Keys

Definition

shrK function definition

consts

```
  shrK :: "agent => key"
```

specification (shrK)

```
  inj_shrK: "inj shrK"
```

Inductive Method Details - Cryptographic Keys

Definition

shrK function definition

consts
```
  shrK :: "agent => key"
```
specification `(shrK)`
```
  inj_shrK: "inj shrK"
```

- The specification of cryptographic keys in the Inductive Method starts with the introduction of a free type *key* as a derivation of the type *nat*;

Inductive Method Details - Cryptographic Keys

Definition

shrK function definition

consts
  shrK :: "agent => key"
specification *(shrK)*
  inj_shrK: "inj shrK"

- The specification of cryptographic keys in the Inductive Method starts with the introduction of a free type *key* as a derivation of the type *nat*;
- A shared key is specified as an injective function taking an agent and returning a key.

Inductive Method Details - Cryptographic Keys

Definition

invKey function definition

```
consts
  invKey :: "key => key"
specification (invKey)
  invKey [simp]: "invKey (invKey K) = K"
  invKey_symmetric: "all_symmetric --> invKey = id"
```

Inductive Method Details - Cryptographic Keys

Definition

invKey function definition

consts

```
  invKey :: "key => key"
```

specification *(invKey)*

```
  invKey [simp]: "invKey (invKey K) = K"
  invKey_symmetric: "all_symmetric --> invKey = id"
```

- The function *invKey*is a function from data type key to key specified by two rules:

Inductive Method Details - Cryptographic Keys

Definition

invKey function definition

consts

```
  invKey :: "key => key"
```

specification `(invKey)`

```
  invKey [simp]: "invKey (invKey K) = K"
  invKey_symmetric: "all_symmetric --> invKey = id"
```

- The function *invKey*is a function from data type key to key specified by two rules:
  - The first rule is a simplification one that says that the double application of the function brings us back to the original value;

Inductive Method Details - Cryptographic Keys

Definition

invKey function definition

consts

```
  invKey :: "key => key"
```

specification *(invKey)*

```
  invKey [simp]: "invKey (invKey K) = K"
  invKey_symmetric: "all_symmetric --> invKey = id"
```

- The function *invKey*is a function from data type key to key specified by two rules:
  - The first rule is a simplification one that says that the double application of the function brings us back to the original value;

Inductive Method Details - Cryptographic Keys

Definition

symKeys set definition

constdefs

```
  symKeys :: "key set"
  "symKeys == K. invKey K = K"
```

Inductive Method Details - Cryptographic Keys

Definition

symKeys set definition

constdefs

```
symKeys :: "key set"
"symKeys == K. invKey K = K"
```

- The symmetric key set is defined as containing all keys where the inverse of the key by the application of the function *invKey* is itself;

Inductive Method Details - Cryptographic Keys

Definition

symKeys set definition

constdefs

```
symKeys :: "key set"
"symKeys == K. invKey K = K"
```

- The symmetric key set is defined as containing all keys where the inverse of the key by the application of the function *invKey* is itself;

Inductive Method Details - Cryptographic Keys

Definition

symKeys set definition

constdefs

```
  symKeys :: "key set"
  "symKeys == K. invKey K = K"
```

- By stating $K \in symKeys \wedge K \notin range\ shrK$ we can create a second class of symmetric keys that are not long-term so that it represents sessions keys in our protocols being verified.

Inductive Method Details - Cryptographic Keys

Definition

Axiom for symmetric usage of shared keys

axioms

```
  sym_shrK [iff]: "shrK X ∈ symKeys"
```

Inductive Method Details - Cryptographic Keys

Definition

Axiom for symmetric usage of shared keys

axioms

sym_shrK [iff]: "shrK X ∈ symKeys"

- Establishes that our long-term keys are symmetric with an axiom.

Inductive Method Details - Cryptographic Keys

Definition

publicKey definition

```
datatype
  keymode = Signature | Encryption

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

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

Inductive Method Details - Cryptographic Keys

Definition

publicKey definition

datatype

```
  keymode = Signature | Encryption
```

consts

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

specification *(publicKey)*

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

Inductive Method Details - Cryptographic Keys

Definition

privateKey axiom definition

axioms

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

Inductive Method Details - Cryptographic Keys

Definition

privateKey axiom definition

axioms

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

Inductive Method Details - Cryptographic Keys

Definition

Public Key abbreviations

abbreviation

```
pubEK :: "agent => key" where
"pubEK == publicKey Encryption"
pubSK :: "agent => key" where
"pubSK == publicKey Signature"
privateKey :: "[keymode, agent] => key" where
"privateKey b A == invKey (publicKey b A)"
priEK :: "agent => key" where
"priEK A == privateKey Encryption A"
priSK :: "agent => key" where
"priSK A == privateKey Signature A"
```

Inductive Method Details - Compromised Agents

Definition

bad set definition

consts

```
  bad :: "agent set"
```

specification *(bad)*

```
  Spy_in_bad [iff]: "Spy ∈ bad"
  Server_not_bad [iff]: "Server ∉ bad"
```

Inductive Method Details - Messages

Definition

msg datatype definition

datatype

```
msg = Agent agent
    | Number nat
    | Nonce nat
    | Key key
    | MPair msg msg
    | Hash msg
    | Crypt key msg
```

Inductive Method Details - Events

Definition

event datatype definition

datatype

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

Inductive Method Details - Initial Knowledge

Definition

initState definition

consts

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

Inductive Method Details - Initial Knowledge

Definition

Spy agent initial knowledge definition

primrec

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

Inductive Method Details - Knows

Definition

knows function definition

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

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

Inductive Method Details - Knows

Definition

knows function definition

consts

```
    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))"
```

Inductive Method Details - Operators

Definition

parts inductive set definition

inductive_set

```
  parts :: "msg set => msg set"
  for H :: "msg set"
  where
      Inj [intro]: "X ∈ H ==> X ∈ parts H"
      | Fst: "{|X,Y|} ∈ parts H ==> X ∈ parts H"
      | Snd: "{|X,Y|} ∈ parts H ==> Y ∈ parts H"
      | Body: "Crypt K X ∈ parts H ==> X ∈ parts H"
```

Inductive Method Details - Operators

Definition

analz inductive set definition

inductive_set

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

Inductive Method Details - Operators

Definition

synth inductive set definition

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"
```

FEDERAL
ARINA

Inductive Method Details - Operators

Definition

used function definition

consts

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

primrec

```
  used_Nil: "used [] = (UN B. parts (initState B))"
  used_Cons: "used (ev # evs) =
    (case ev of
      Says A B X => parts {X} ∪ used evs
      | Gets A X => used evs
      | Notes A X => parts {X} ∪ used evs)"
```

Dummy Protocol

$$\boxed{\begin{array}{llclcc} 1. & A & \rightarrow & B & : & \{|A, B, Na|\}_{K_B} \\ 2. & B & \rightarrow & A & : & \{|Na, Nb, K_{AB}|\}_{K_A} \\ 3. & A & \rightarrow & B & : & \{|Nb\}_{K_{AB}} \end{array}}$$

Figure: Example protocol

Dummy Protocol Specification

Definition

inductive definition of example protocol

```
inductive_set example :: "event list set"
where
    Nil: "[] ∈ example"

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

    |EX1: "[|evs1 ∈ example; Nonce NA ∉ used evs1|]
     ⇒Says A B (Crypt (pubK B){|Agent A, Agent B,
Nonce NA|}) #
     evs1 ∈ example"
```

Dummy Protocol Specification

Definition

```
|EX2: "[|evs2 ∈ example; Nonce NB ∉ used evs2;
    Key AB ∉ used evs1
    Says A' B (Crypt (pubK B){|Agent A, Agent B,
Nonce NA|})
    ∈ set evs2|]
  ⇒Says B A (Crypt (pubK A) {| Nonce NA, Nonce NB,
Key AB|})
  # evs2 ∈ example"
```

Dummy Protocol Specification

Definition

```
    |EX3: "[|evs3 ∈ example;
       Says A B (Crypt (pubK B){|Agent A, Agent B,
Nonce NA|})
       ∈ set evs3;
       Says B A (Crypt (pubK A) {| Nonce NA, Nonce NB,
Key AB|})
       ∈ set evs3 |]
     ⇒Says A B (Crypt (Key AB){| Nonce NB|}) # evs3 ∈
example"
```

Dummy Protocol Specification

Definition

```
    |Oops: "[|evso ∈ example;
     Says B A (Crypt (pubK A) {| Nonce NA, Nonce NB,
Key AB|})
     ∈ set evso|]
     ⇒ Notes Spy {|Nonce NA, Nonce NB, Key AB|}# evso
∈ example"
```

What to do next?

- We describe the protocol goals as Theorems and Lemmas;

What to do next?

- We describe the protocol goals as Theorems and Lemmas;
- We prove these inductively with the assistance of Isabelle;

What to do next?

- We describe the protocol goals as Theorems and Lemmas;
- We prove these inductively with the assistance of Isabelle;
- Proof tactics are known for most of the usual goals;

What to do next?

- We describe the protocol goals as Theorems and Lemmas;
- We prove these inductively with the assistance of Isabelle;
- Proof tactics are known for most of the usual goals;
- The problem starts when you want to prove a new goal....

Problems with the Inductive Method

- Training someone to use it proficiently takes 2 years;

Problems with the Inductive Method

- Training someone to use it proficiently takes 2 years;
- Understanding Isabelle cryptic messages is painful

Problems with the Inductive Method

- Training someone to use it proficiently takes 2 years;
- Understanding Isabelle cryptic messages is painful
- The person carrying the verification need to be clever and seasoned with the tool, otherwise there will be pain;

Problems with the Inductive Method

- Training someone to use it proficiently takes 2 years;
- Understanding Isabelle cryptic messages is painful
- The person carrying the verification need to be clever and seasoned with the tool, otherwise there will be pain;
- Too much freedom and power are sometimes difficult to use.

Discussion

- What else can you foresee modelled using this strategy?

Discussion

- What else can you foresee modelled using this strategy?
- Can this be extended?

Discussion

- What else can you foresee modelled using this strategy?
- Can this be extended?
- What this strategy can not do?

Questions????

UNIVERSIDADE FEDERAL
DE SANTA CATARINA

UNIVERSIDADE FEDERAL
DE SANTA CATARINA