Automated formal analysis of security protocols

Automated verification

- It is not easy and is error-prone itself to do formal analysis manually;
- Development of methods for automated or semi-automated (interactive) validation and verification is important area, especially in the context of security protocols;

Different directions

- **Model checking** (state exploration tools);
  - specific (NRL Protocol Analyser,etc)
  - general purpose tools (SMV, SPIN, Mocha, etc)
  - general purpose tools combined with specific translators (Casper/FDR, etc)
- **Theorem proving**
  - Automated (TAPS, etc)
  - Interactive (Isabell, PVS, etc)
- **Combinations of above techniques**:
  - Athena, etc
- **Others:** decision procedures for specific theories, infinite state model checking,etc

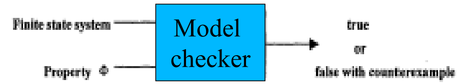
General questions

- How to represent a protocol (system) to be analysed?
- How to express properties to be verified?

Model checking

- A protocol (system executing a protocol) is represented as a transition system **M** with **finitely** many states;
- A property to be analysed is expressed by a formula of a logic (temporal, modal, etc) **f;**
- Then verification amounts to checking whether the formula ***f*** *is* true in **M;**
- Model checking is done via efficient state exploration techniques;

Model checking



Nice properties

- Fully automated procedures;
- Very efficient state exploration;

but

- Finite state abstraction is not always adequate, especially for protocols with unbounded number of participants or unbounded number of rounds.

Attack on Needham-Schroeder protocol

- A particular success of model checking methods in security protocol verification was discovery of a flaw in NS protocol based on public key cryptography (Gavin Lowe, 1995-1996);

Original protocol

Message 1. $A \to B$: $A.B.\{A, N_A\}_{PK(B)}$
Message 2. $B \to A$: $B.A.\{N_A, N_B\}_{PK(A)}$
Message 3. $A \to B$: $A.B.\{N_B\}_{PK(B)}$.

Attack

Message 1a. $A \to I$: $A.I.\{A, N_A\}_{PK(I)}$
Message 1b. $I_A \to B$: $A.B.\{A, N_A\}_{PK(B)}$
Message 2b. $B \to I_A$: $B.A.\{N_A, N_B\}_{PK(A)}$
Message 2a. $I \to A$: $I.A.\{N_A, N_B\}_{PK(A)}$
Message 3a. $A \to I$: $A.I.\{N_B\}_{PK(I)}$
Message 3b. $I_A \to B$: $A.B.\{N_B\}_{PK(B)}$.

Corrupt participant I impersonates A

Theorem Proving

- A protocol (a system) to be verified is described by a formula **Fs** of a logic (classical first-order, higher-order, modal, temporal, etc);
- A property to be verified is expressed by a formula **P** of the same logic;
- Then to establish the required property it is enough to prove the theorem **Fs → P;**

Theorem proving

Potential benefits:

- the systems with *unbounded* (infinite) number states can be analysed;

But:

- The problems here are, in general, *undecidable*;
- Procedures are *incomplete* and of high complexity.

Theorem proving

What to do?

- Apply automated procedures for fragments of first-order and higher-order logic
  - E.Cohen, TAPS system, Microsoft Research;
- Use interactive theorem proving
  - L.Paulson, Cambridge: using Isabell, higher-order inductive theorem prover for the verification of security protocols;
  - J.Bryans, S. Schenider, using interactive theorem prover PVS;

Other interesting approaches

- Bruno Blanchet, INRIA: approach based on ideas from Logic Programming (ProVerif, available online at http://www.di.ens.fr/~blanchet/crypto-eng.html):

- A protocol is presented as a set of Horn clauses (like a program in Prolog), defining capabilities of all participants);
- Verification then amounts to checking whether a security breaching goal can be reached (derived) from the set of clauses;
- If the system detects the goal is unreachable, then the protocol is correct;
- Standard operational semantics of Prolog is not very useful here due to undesirable looping;
- Novel operational semantics (search strategy) is defined;

ProVerif system

Denning-Sacco key distribution protocol

Message 1. $A \to B : \{\{k\}_{sk_A}\}_{pk_B}$

Message 2. $B \to A : \{s\}_k$

Its representation in ProVerif system

Computation abilities of the attacker:

pencrypt	$\mathsf{attacker}(m) \wedge \mathsf{attacker}(pk) \to \mathsf{attacker}(\mathsf{pencrypt}(m, pk))$
pk	$\mathsf{attacker}(sk) \to \mathsf{attacker}(\mathsf{pk}(sk))$
pdecrypt	$\mathsf{attacker}(\mathsf{pencrypt}(m, \mathsf{pk}(sk)) \wedge \mathsf{attacker}(sk) \to \mathsf{attacker}(m)$
sign	$\mathsf{attacker}(m) \wedge \mathsf{attacker}(sk) \to \mathsf{attacker}(\mathsf{sign}(m, sk))$
getmess	$\mathsf{attacker}(\mathsf{sign}(m, sk)) \to \mathsf{attacker}(m)$
checksign	removed since implied by **getmess**
sencrypt	$\mathsf{attacker}(m) \wedge \mathsf{attacker}(k) \to \mathsf{attacker}(\mathsf{sencrypt}(m, k))$
sdecrypt	$\mathsf{attacker}(\mathsf{sencrypt}(m, k)) \wedge \mathsf{attacker}(k) \to \mathsf{attacker}(m)$

Initial knowledge of the attacker:

$\mathsf{attacker}(\mathsf{pk}(sk_A[])), \quad \mathsf{attacker}(\mathsf{pk}(sk_B[])), \quad \mathsf{attacker}(a[])$

Protocol:

First message: $\mathsf{attacker}(\mathsf{pk}(x)) \to \mathsf{attacker}(\mathsf{pencrypt}(\mathsf{sign}(k[\mathsf{pk}(x)], sk_A[]), \mathsf{pk}(x)))$

Second message: $\mathsf{attacker}(\mathsf{pencrypt}(\mathsf{sign}(k', sk_A[]), \mathsf{pk}(sk_B[]))) \to \mathsf{attacker}(\mathsf{sencrypt}(s[], k'))$

Developments here at the Department

- Verification based on supercompilation (a program transformation technique);
- A system (protocol) is encoded as a functional program, then supercompilation is applied to get a simplified, but equivalent program for which correctness conditions may be easily checked;
- It has proved to be very efficient technique for verification of parameterised systems;
- **But,** it has not been tried yet for security protocols;
- Possible MSc (and PhD) projects. If interested, please contact A.Lisitsa.