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)
  - Unbounded model checking for crypto protocols (ProVerif, Tamarin, 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

<table>
<thead>
<tr>
<th>Attack</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Message 1.</strong> A → B: A.B.[{A,N_A}]_{PK(B)}</td>
</tr>
<tr>
<td><strong>Message 2.</strong> B → A: B.A.[{N_A,N_B}]_{PK(A)}</td>
</tr>
<tr>
<td><strong>Message 3.</strong> A → B: A.B.[{N_B}]_{PK(B)}</td>
</tr>
</tbody>
</table>

| Message 1a. | A → I: A.I.[{A,N_A}]_{PK(I)} |
| Message 1b. | I_A → B: A.B.[{A,N_A}]_{PK(B)} |
| Message 2a. | I_A → B: B.A.[{N_A,N_B}]_{PK(A)} |
| Message 2b. | I_A → B: I.A.[{N_A,N_B}]_{PK(A)} |
| Message 3a. | A → I: A.I.[{N_B}]_{PK(I)} |
| Message 3b. | I_A → 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 $F_s$ 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 $F_s \rightarrow P$;
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 Isabelle, higher-order inductive theorem prover for the verification of security protocols;
  - J. Bryans, S. Schenider, using interactive theorem prover PVS;
Other interesting approaches


  - 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;
  - Recent versions use pi-calculus as a language for front-end
ProVerif system

Denning-Sacco key distribution protocol

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

Message 2. $B \rightarrow A : \{s\}_{sk}$

Its representation in ProVerif system

Computation abilities of the attacker:

- $\text{pencrypt}$: $\text{attacker}(m) \land \text{attacker}(pk) \rightarrow \text{attacker}(\text{pencrypt}(m, pk))$
- $\text{pk}$: $\text{attacker}(sk) \rightarrow \text{attacker}(pk(sk))$
- $\text{decrypt}$: $\text{attacker}(\text{pencrypt}(m, pk(sk))) \land \text{attacker}(sk) \rightarrow \text{attacker}(m)$
- $\text{sign}$: $\text{attacker}(m) \land \text{attacker}(sk) \rightarrow \text{attacker}(\text{sign}(m, sk))$
- $\text{getmess}$: $\text{attacker}(\text{sign}(m, sk)) \rightarrow \text{attacker}(m)$
- $\text{checksign}$: removed since implied by $\text{getmess}$
- $\text{ssencrypt}$: $\text{attacker}(m) \land \text{attacker}(k) \rightarrow \text{attacker}(\text{ssencrypt}(m, k))$
- $\text{ssdecrypt}$: $\text{attacker}(\text{ssencrypt}(m, k)) \land \text{attacker}(k) \rightarrow \text{attacker}(m)$

Initial knowledge of the attacker:

- $\text{attacker}(pk(sk_A[]))$, $\text{attacker}(pk(sk_B[]))$, $\text{attacker}(a[])$

Protocol:

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

Second message: $\text{attacker}(\text{pencrypt}(\text{sign}(k', sk_A[]), pk(sk_B[]))) \rightarrow \text{attacker}(\text{ssencrypt}(s[], k'))$