

Automated Protocol Analysis

The state of the art

By now there are a variety of methods for protocol analysis that rely at least in part on automatic tools.

They are effective on abstract but detailed models of important protocols.

Some of them employ elaborate proof techniques —some general, some specific to this area.

We will cover only a fraction of this work.

Decision procedures

Going back to Dolev and Yao, there has been much work on special decision procedures.

In recent years, these have been most successful for finite-state systems.

An important example: deduction

Static knowledge (of a participant or attacker):

$\nu\tilde{n}. \{M_1/x_1, \dots, M_l/x_l\}$

where

- \tilde{n} is a list of names
(fresh, not known to the attacker a priori),
- M_1, \dots, M_l are terms
(known to the attacker),
- x_1, \dots, x_l are variables
(which can be used to refer to the terms).

Example: $\nu k. \{\text{senc}(M, k)/x_1, k/x_2\}$

Someone with this knowledge can deduce $\text{sdec}(x_1, x_2)$,
which equals M if $\text{sdec}(\text{senc}(x, y), y) = x$.

Deduction is central to methods for protocol analysis.

Deduction (\vdash)

$$\frac{}{\nu\tilde{n}.\sigma \vdash x\sigma} \quad x \in \text{dom}(\sigma) \quad \frac{}{\nu\tilde{n}.\sigma \vdash s} \quad s \notin \tilde{n}$$
$$\frac{\phi \vdash M_1 \quad \dots \quad \phi \vdash M_k}{\phi \vdash f(M_1, \dots, M_k)} \quad f \in \Sigma \quad \frac{\phi \vdash M \quad M =_E M'}{\phi \vdash M'}$$

where Σ is the underlying signature and $M =_E M'$ is equality in the underlying equational theory E .

Example: $\nu k. \{\text{senc}(M, k)/x_1, k/x_2\} \vdash M$

Decidability of deduction

Deduction is undecidable for some theories (even some decidable ones).

Deduction is in PTIME for theories given by a finite, convergent set of rewrite rules of the form $M \rightarrow N$ where N is a proper subterm of M or a constant.

- These conditions hold often.
(And decidability was known in many special cases.)
- For example, $\text{sdec}(\text{senc}(x, y), y) \rightarrow x$.

And one can go a bit further ...

- Other classes of theories.
- Other decision problems.

Model checking

Since the mid 1990s, many general model checking techniques and tools have been applied in this area.

They are typically limited to finite-state systems.

Proof assistants

Some automated proofs require a fair amount of expert human guidance.

Paulson's work with Isabelle is probably the best example.

The results are sophisticated theorems (and attacks), for infinite-state systems.

Other techniques

Several other approaches rely on programming-language techniques:

- control-flow analysis,
- typing,
- abstract interpretation,
- logic-program analysis.

These techniques are often incomplete but useful in examples and (relatively) easy to use.

Some of these techniques are equivalent
—but not trivially.

Blanchet's ProVerif

A checker for the applied pi calculus.

- A somewhat modified input syntax.
- An internal representation in terms of Horn clauses.
- Algorithms for certain classes of proofs.
 - Secrecy.
 - Authenticity, as correspondence assertions.
 - Most recently, certain equivalences.

ProVerif input syntax

| | |
|---|-------------------------|
| $M, N ::=$ | terms |
| x | variable |
| n | name |
| $f(M_1, \dots, M_n)$ | constructor application |
| $P, Q ::=$ | processes |
| $\overline{M}\langle N \rangle.P$ | output |
| $M(x).P$ | input |
| 0 | nil |
| $P \mid Q$ | parallel composition |
| $!P$ | replication |
| $(\nu n)P$ | “new” |
| $let\ x = g(M_1, \dots, M_n)\ in$ $P\ else\ Q$ | destructor application |

A small process

$(\nu k)(\bar{c}(\text{senc}(M, k)) \cdot 0 \mid c(x)). \text{let } y = \text{sdec}(x, k) \text{ in } \bar{d}(H(y)) \cdot 0$

Internal syntax

Each protocol is compiled into a set of rules.

- Terms (patterns):

$$M ::= x \mid n[M_1, \dots, M_n] \mid f(M_1, \dots, M_n)$$

- Facts:

$$F ::= attacker(M) \mid mess(M, M')$$

$attacker(M)$: the attacker may have the term M

$mess(M, M')$: M' may be sent on M

- Rules:

$$F_1 \wedge \dots \wedge F_n \rightarrow F$$

Internal protocol representation

For example, if a process sends M on channel c when it receives N on channel d , then the rules imply

$$mess(d, N) \rightarrow mess(c, M)$$

In addition, some rules deal with communication and cryptographic functions, for example

$$attacker(x) \wedge attacker(y) \rightarrow mess(x, y)$$

$$attacker(x) \wedge attacker(y) \rightarrow attacker(penc(x, y))$$

Proofs with the checker

If $attacker(M)$ is derivable from the rules, then the protocol may reveal M . Otherwise, it keeps M secret.

In addition, the checker can establish correspondence assertions, of the form “if some event happens, then some other events must have happened” .

Sound approximations

The rules do not completely model protocols:

- **Freshness** (of keys, nonces) is only partially captured.
 - The **state** of principals is only partially captured.
 - The **number of times** a message appears is ignored.
- Only the fact that it appears is taken into account.

These approximations help for efficient verification
with infinite state spaces,
with any number of concurrent protocol runs.

(Cf. methods with linear-logic programming.)

A small protocol

Message 1. $A \rightarrow B : \text{penc}((n, pK_A), pK_B)$

Message 2. $B \rightarrow A : \text{penc}((n, k), pK_A)$

Message 3. $A \rightarrow B : \text{senc}(s, k)$

n is a fresh nonce, k is a fresh shared key.

The code

```
P  $\triangleq$  ( $\nu sK_A$ )( $\nu sK_B$ )  
  let  $pk_A = pk(sK_A)$  in let  $pk_B = pk(sK_B)$  in  
   $\bar{e}(pk_A).\bar{e}(pk_B).(A \mid B)$   
  
A  $\triangleq$  ( $\nu n$ )( $\bar{e}(penc((n, pk_A), pk_B)) \mid$   
   $e(z).let(x, y) = pdec(z, sK_A)$  in  
  if  $x = n$  then  $\bar{e}(senc(s, y))$ )  
  
B  $\triangleq$   $e(z).let(x, y) = pdec(z, sK_B)$  in  
  ( $\nu k$ )( $\bar{e}(penc((x, k), y)) \mid e(z').let s' = sdec(z', k)$  in 0)
```

(And there are more elaborate variants.)

Compilation into rules (a little optimized)

$$\begin{aligned} & \text{attacker}(x) \wedge \text{attacker}(y) \rightarrow \text{attacker}(\text{penc}(x, y)) \\ & \text{attacker}(x) \rightarrow \text{attacker}(\text{pk}(x)) \\ & \text{attacker}(\text{penc}(m, \text{pk}(k))) \wedge \text{attacker}(k) \rightarrow \text{attacker}(m) \\ & \text{attacker}(x) \wedge \text{attacker}(y) \rightarrow \text{attacker}(\text{senc}(x, y)) \\ & \text{attacker}(\text{senc}(m, k)) \wedge \text{attacker}(k) \rightarrow \text{attacker}(m) \\ & \text{attacker}(x) \wedge \text{attacker}(y) \rightarrow \text{mess}(x, y) \\ & \text{mess}(x, y) \wedge \text{attacker}(x) \rightarrow \text{attacker}(y) \\ & \text{attacker}(e[]) \\ & \text{attacker}(\text{pk}(sK_A[])) \\ & \text{attacker}(\text{pk}(sK_B[])) \\ & \text{attacker}(\text{penc}((n[], \text{pk}(sK_A[])), \text{pk}(sK_B[]))) \\ & \text{attacker}(\text{penc}((n[], x), \text{pk}(sK_A[]))) \rightarrow \text{attacker}(\text{senc}(s[], x)) \\ & \text{attacker}(\text{penc}((x, y), \text{pk}(sK_B[]))) \\ & \rightarrow \text{attacker}(\text{penc}((x, k[\text{penc}((x, y), \text{pk}(sK_B[])])], y)) \end{aligned}$$

Can we derive $\text{attacker}(s[])$?

Solving algorithm (sketch)

1. Completion of the rules by resolution (with a particular selection strategy). For example:

$$\begin{array}{l} \text{attacker}(x) \wedge \text{attacker}(y) \rightarrow \text{attacker}(\text{penc}(x, y)) \\ \text{attacker}(\text{penc}(\text{sign}(z, sk_A[]), \text{pk}(sk_B[]))) \\ \rightarrow \text{attacker}(\text{senc}(s[], z)) \\ \hline \text{attacker}(\text{sign}(z, sk_A[])) \wedge \text{attacker}(\text{pk}(sk_B[])) \\ \rightarrow \text{attacker}(\text{senc}(s[], z)) \end{array}$$

2. Backwards search (with loop detection).

The problem of looping

A natural tool for determining derivability of a fact from clauses is resolution.

Basic SLD-resolution (as in Prolog) would never terminate:

$$\text{attacker}(\text{senc}(m, k)) \wedge \text{attacker}(k) \rightarrow \text{attacker}(m)$$

A key idea is to avoid resolving on $\text{attacker}(x)$.

Optimizations

Various optimization help prune the search space:

- We can eliminate subsumed clauses.
- We can eliminate duplicate hypotheses in clauses.
- We can eliminate hypotheses *attacker(x)* when *x* does not appear elsewhere in the clause.
- We can assume, then verify secrecy assumptions.

Termination

The resolution algorithm does not always terminate, but it does terminate for tagged protocols:

A protocol is tagged when each encryption, signature, ... in the protocol is distinguished from others by a constant tag c_i

$$\{c_i, M_1, \dots, M_n\}_K$$

- This is a large class of protocols.
- It is easy to add tags.
- Tagging is generally a good design practice.

Examples

Pentium III, 1 GHz.

| Protocol | Result | ms |
|--|------------------|-----|
| Needham-Schroeder public key | Attack [Lowe] | 10 |
| Needham-Schroeder public key corrected | Secure | 7 |
| Needham-Schroeder shared key | Attack [Denning] | 52 |
| Needham-Schroeder shared key corrected | Secure | 109 |
| Denning-Sacco | Attack [AN] | 6 |
| Denning-Sacco corrected | Secure | 7 |
| Otway-Rees | Secure | 10 |
| Otway-Rees, variant of Paulson98 | Attack [Paulson] | 12 |
| Yahalom | Secure | 10 |
| Simpler Yahalom | Secure | 11 |
| Main mode of Skeme | Secure | 23 |

Some larger examples

- A protocol for certified email.
- The JFK (“Just Fast Keying”) protocol for IP security.
- Password-based protocols (such as the Encrypted Key Exchange).
- Web-services protocols.
- Electronic-voting protocols.

Some of the automatic proofs take minutes.

Some manual proofs are combined with automatic proofs.

Code (fragment)

```
let processS =
  (* The attacker chooses possible recipients of the message *)
  in(c, recipient);
  (* Build the message to send *)
  new msgid;
  let m = Message(recipient,msgid) in
  (* Step 1 *)
  new k;
  let em = senc(k,m) in
  let hs = H((cleartext, em)) in
  let S2TTP = penc(TTPEncKey, (Sname, (Give, k, recipient, hs))) in

  out(recipient, (TTPname, em, cleartext, S2TTP));

  (* Step 4 *)
  !
  in(Sname, inmess4);
  let (=Released, =S2TTP, =recipient) = checkS(TTPVerKey, inmess4) in
  (* S knows that R has read the message *)

  0
  else out(Sname, inmess4).
```