COL876: SPECIAL TOPICS IN FORMAL METHODS

Formal verification of security protocols

Lecture 6, 14 August 2023

APPLIED-PI CALCULUS: GRAMMAR

P, Q :=	plain process
0	[null process]
P | Q	[parallel composition]
!P	[replication]
νn.P	[name restriction]
if $t_1 = t_2$ then P else Q	[conditional branching]
in(c, x).P	[receive action]
out(c, t).P	[send action]
let x = t in P	[let binding]

RECAP

- Saw how to convert protocols in arrow notation to programs at each end-point
- Convert these programs into applied-pi notation
- Put these together to get the whole protocol in applied-pi
- Suffices to consider in parallel the following for every generated agent:
  - replicated instance of each role $!P_i(\mathit{ski}, \mathit{pkr})$, $!P_r(\mathit{skr})$
  - replicated instance of an intruder supplying the public key for any such parameter in any role $!in(c, xpk). P_i(\mathit{ski}, \mathit{xpk})$
- Add any extra bookkeeping (monitor processes, events etc) for verifying properties

TODAY

- Easy to write out protocols and expected properties
- What does it mean to verify them? Consider all
  - possible instantiations of variables
  - possible unfoldings of any replication
  - reduction sequences starting at the initial configuration
- Non-trivial while also relatively mechanical; needs automation

PROVERIF PROTOCOL VERIFIER

- https://bblanche.gitlabpages.inria.fr/proverif/
- Automatic cryptographic protocol verifier
- Can handle unboundedly many sessions of the protocol
- Tries to prove a property; if it cannot be proved, tries to produce an attack trace
- Suffers from false negatives (a claimed attack might not “really” be an attack) but is sound; if a property is proved true, it is indeed true

PROVERIF: UNDER THE HOOD

- Horn clauses + resolution for the protocol and *negated* property
  - Any derivation of this provides an attack trace
  - Attack might be due to some abstraction with Horn clauses, but if not, it violates the property
  - Otherwise, property holds of the protocol

PROVERIF: SYNTAX

- Input: Protocol in ~applied-pi calculus and security property
- Terms appearing in the process must be typed
- ProVerif checks for well-typedness of the process
  - But not of the property! Allows detection of type-flaw attacks
- Crypto operations specified using equations or *rewrite rules*
- $\mathsf{fst}(x, y) \rightarrow x \quad \mathsf{snd}(x, y) \rightarrow y \quad \mathsf{adec}(\mathsf{aenc}(x, \mathsf{pk}(y)), y) \rightarrow x$

RUNNING EXAMPLE

$$A \rightarrow B : A, \mathsf{enc}(m, \mathsf{pk}(B))$$
$$B \rightarrow A : \mathsf{enc}(m, \mathsf{pk}(A))$$

- $\mathrm{P_i}(ski, pkr) \triangleq \nu n.\ \mathrm{out}(c, \mathrm{aenc}(n, pkr)).\ \mathrm{in}(c, x).\ \mathrm{if}(\mathrm{adec}(x, ski) == n)$ then SUCCESS
- $\mathrm{P_r}(skr) \triangleq \mathrm{in}(c, y).\ \mathrm{let}\ pka = \mathrm{fst}(y)\ \mathrm{in}.\ \mathrm{let}\ z = \mathrm{adec}(y, skr)\ \mathrm{in}.\ \mathrm{out}(c, \mathrm{aenc}(z, pka))$
- $\mathrm{Pr} \triangleq\ !\nu sk.(\ !\mathrm{in}(c, x_{pk}).\ \mathrm{P_i}(sk, x_{pk})\ |\ !\mathrm{P_r}(sk)\ |\ \mathrm{out}(c, \mathrm{pk}(sk))\ )$

PROVERIF: CRYPTO OPERATIONS

- Declare two types, pkey and skey, using the **`type`** keyword
- Declare two functions pk and aenc along with params and types
  - Constructors declared using **`fun`** keyword
- Declare a equation defining the operation of the adec function
  - Using **`reduc`** and universally quantified terms
- Tuples have in-built support; no need to do anything explicitly

EXAMPLE: CRYPTO OPERATIONS

```
type skey.

type pkey.


fun pk(skey): pkey.

fun aenc(bitstring, pkey): bitstring.


reduc forall t: bitstring, k: skey; adec(aenc(t, pk(k)), k) = t.
```

PROVERIF: SPECIFYING PROTOCOLS

- The `channel` keyword declares a public channel
- For any other free name, use `free` keyword
- Free names and constructors known to intruder by default
  - If not, modify using the `private` keyword
- Can specify reachability/secrecy checks using `query attacker`
- Then specify roles and the overall protocol process

EXAMPLE: ROLES

```
let init(ski:skey, pkr:pkey) =
    new s: bitstring;
    out(c, (pk(ski), aenc(s, pkr))) ;
    in(c, x: bitstring);
    let y = adec(x, ski) in
    if (y = s) then out(c, SUCCESS).

let resp(skr:skey) =
    in(c, (k: pkey, x: bitstring));
    let z = adec(x, skr) in
    out(c, aenc(z, k)).
```

EXAMPLE: PROTOCOL

```
process
!new sk:skey;
  (
     out(c, pk(sk)) |
     ( !in(c, x:pkey);init(sk,x) ) |
     ( !resp(sk))
  )
```

PROVERIF SYNTAX

- Identifiers: an unlimited sequence of letters, digits, _, and '.
  - But must begin with a letter!
- Boolean operators: &&, ||, not Constants: true, false Equality: = and <>
- ProVerif does some minimal pattern matching; can use in **`let`**
  - x : t matches any term of type t and stores it in x
  - Similarly a tuple pattern (t_1, ..., t_n) matches tuples of this type
  - =M matches any term equal to M; basically an equality check!

PROVERIF SYNTAX

- Is !P | Q the same as !(P | Q) or (!P) | Q?
  - Parallelism | binds most closely
  - Then **if**… **then**… **else** and **let**… **in**
  - Finally unary operations (replication, name restriction etc)
- Where do the parentheses go in the following?

new n : t; **out**(c, n) | **new** n : t; **in**(c, x : t) | **if** x = n **then** 0 | **out**(c, n)

PROVERIF SYNTAX

- Parallelism | binds most closely
- Then **if**… **then**… **else** and **let**… **in**
- Finally unary operations (replication, name restriction etc)
- Where do the parentheses go in the following?

new n : t; (**out**(c, n) | **new** n : t; **in**(c, x : t) | **if** x = n **then** (0 | **out**(c, n)))

PROVERIF SYNTAX

- Parallelism | binds most closely
- Then **if**… **then**… **else** and **let**… **in**
- Finally unary operations (replication, name restriction etc)
- Where do the parentheses go in the following?

if $t = t'$ **then** **if** $u = u'$ **then** P **else** Q

PROVERIF SYNTAX

- Parallelism | binds most closely
- Then **if**… **then**… **else** and **let**… **in**
- Finally unary operations (replication, name restriction etc)
- Where do the parentheses go in the following?

$$\textbf{if } t = t' \textbf{ then } (\textbf{if } u = u' \textbf{ then } P \textbf{ else } Q)$$