APPLIED-PI CALCULUS: GRAMMAR

\[ P, Q := \text{plain process} \]

\[ \circ \quad \text{[null process]} \]

\[ P \mid Q \quad \text{[parallel composition]} \]

\[ !P \quad \text{[replication]} \]

\[ \nu n.P \quad \text{[name restriction]} \]

\[ \text{if } t_1 = t_2 \text{ then } P \text{ else } Q \quad \text{[conditional branching]} \]

\[ \text{in}(c, x).P \quad \text{[receive action]} \]

\[ \text{out}(c, t).P \quad \text{[send action]} \]

\[ \text{let } x = t \text{ in } P \quad \text{[let binding]} \]
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 \( \Pi_i(ski, pkr), \Pi_r(skr) \)
  - replicated instance of an intruder supplying the public key for any such parameter in any role \( \text{!in}(c, xpk). \Pi_i(ski, xpk) \)
- 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/](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
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
  - \( \text{fst}(x, y) \rightarrow x \)  \( \text{snd}(x, y) \rightarrow y \)  \( \text{adec}(\text{aenc}(x, \text{pk}(y)), y) \rightarrow x \)
RUNNING EXAMPLE

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

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

- $P_i(ski, pkr) \triangleq \nu n. \text{out}(c, \text{aenc}(n, pkr)). \text{in}(c, x). \text{if}(\text{adec}(x, ski) = n) \text{ then SUCCESS}$

- $P_r(skr) \triangleq \text{in}(c, y). \text{let } pka = \text{fst}(y) \text{ in. let } z = \text{adec}(y, skr) \text{ in. out}(c, \text{aenc}(z, pka))$

- $Pr \triangleq !\nu sk.\big( !\text{in}(c, x_{pk}). P_i(sk, x_{pk}) \mid !P_r(sk) \mid \text{out}(c, pk(sk)) \big)$
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 `aenc` 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; adep(aenc(t, pk(k)), k) = t.
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

\[
\text{let init}(\text{ski:skey}, \text{pkr:pkey}) = \\
\quad \text{new } s: \text{bitstring}; \\
\quad \text{out}(c, (\text{pk(}\text{ski}\text{)}, \text{aenc}(s, \text{pkr}))) ; \\
\quad \text{in}(c, x: \text{bitstring}); \\
\quad \text{let } y = \text{adec}(x, \text{ski}) \text{ in} \\
\quad \text{if } (y = s) \text{ then out}(c, \text{SUCCESS}).
\]

\[
\text{let resp}(\text{skr:skey}) = \\
\quad \text{in}(c, (k: \text{pkey, x: bitstring})); \\
\quad \text{let } z = \text{adec}(x, \text{skr}) \text{ in} \\
\quad \text{out}(c, \text{aenc}(z, k)).
\]
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 o | 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 (o | 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
Parallelism binds most closely

Then \textbf{if}... \textbf{then}... \textbf{else} and \textbf{let}... \textbf{in}

Finally unary operations (replication, name restriction etc)

Where do the parentheses go in the following?

\begin{align*}
\textbf{if } t = t' \textbf{ then } (\textbf{if } u = u' \textbf{ then } P \textbf{ else } Q)
\end{align*}