Protocol analysis using ProVerif, 2nd part

ProVerif's input language

- ProVerif internally represents protocols as sets of Horn clauses.
- The protocol can be entered as Horn clauses, or as a process in a language similar to spi-calculus.
- Invoking the analyzer:
  - `./proverif` *file*, if *file* contains the protocol specification as Horn clauses;
  - `./proverif -in pi` *file*, if *file* contains the protocol specification in applied π-calculus.

A process

A process P is one of

0	does nothing
$\texttt{new}\ n; P'$	create new atom n, then P'
$\texttt{in}(c, p); P'$	bind a msg from chan. c to var. p, then P'
$\texttt{out}(c, m); P'$	send the msg m on chan. c, then P'
$\texttt{let}\ p = M\ \texttt{in}\ P'\ \texttt{else}\ P''$	bind p to M, do P' if success, P'' otherwise
$P_1 \mid P_2$	do P_1 and P_2 in parallel
$!P'$	replicate P'. $!P' \equiv P' \mid !P'$
$\texttt{event}\ M; P'$	emit event M, then P'

A channel can be read (i.e. intercepted) and written by a party that knows its name.
A process represents all sessions of all parties.

Translation to Horn clauses

- Just two predicates:
  - attacker(v) means that the attacker can learn the value v.
  - mess(c,v) means that message v can be transmitted over channel c.
- Each output statement generates a Horn clause stating that if previous input messages have been transmitted on their channels, then the message from this statement will be transmitted on this channel.
  - Messages on channels do not have "direction of movement".
  - This is different from $\overline{c}\langle M\rangle.P \mid c(x).Q \rightarrow P \mid Q\{M/x\}$.

Protocol specification

Declare

- message constructors;
  - constants, channel names, event names, constructors, etc.
  - whether adversary has access to them or not
- message destructors;
  - whether adversary has access to them or not
  - In the ProVerif language, terms cannot be “automatically” taken apart or parsed
    - like we did with Horn clauses
- predicates (if you need them);
- queries;
- the process.

Demo. . .

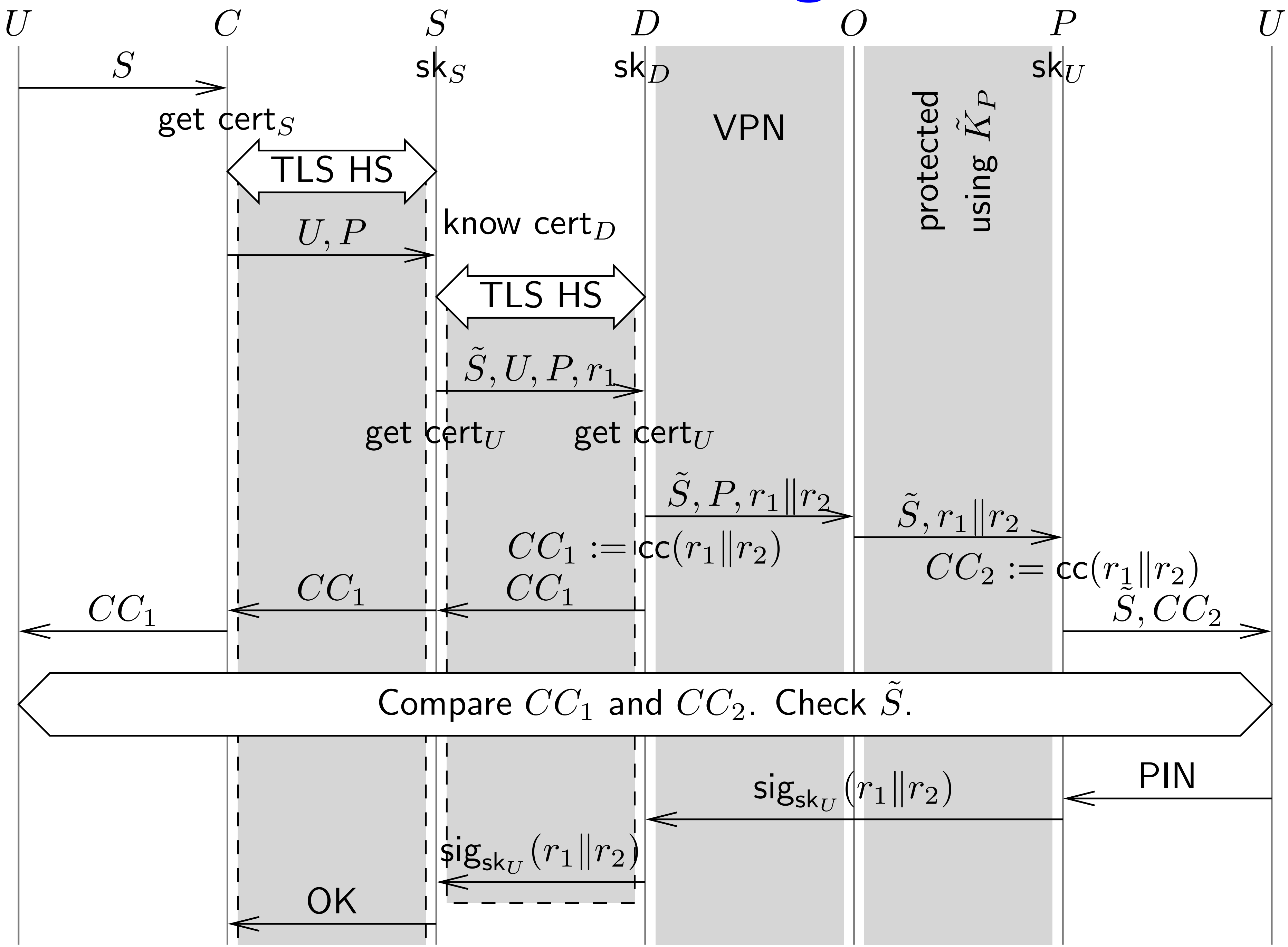
TODO:

- `proverif/examples/pi/secr-auth/piyahalom`
  - Analysis of the code and execution result
- `proverif/examples/pi/secr-auth/piyahalom-bid`

Analysis: Estonian Mobile-ID identification

- User's secret key contained in the SIM-card
- User establishes a TLS session with the server, server is authenticated.
- Server generates a challenge. Causes the phone to receive it.
- Phone shows a very short digest of the challenge.
- Server sends that digest also to user's computer, which shows it.
- User compares two digests, if OK, authorizes phone to sign the challenge.
- Challenge is sent back, server thinks it's talking to the user.

Parties and message flow



Modeling certificates

```
private fun cert/2.
reduc readcert(cert(x,y)) = (x,y).
```

- When an honest party p constructs a public key k for himself, he also executes out(net, cert(p,k)).
- Adversary cannot construct certificates itself. Where does he get certificates for his own keys?

```
let simpleca = ! in(net, pubkey); new n; out(net, cert(n,pubkey)).
```

Same with keys shared between phone and operator.

The process contains . . . | simpleca | . . .

Useful trick: procedures / functions

Function implementation

```
private free f_in

let f =
  in(f_in, (f_out,arg));
    ......
  out(f_out, result).
```

Function call:

```
...
new f_out;
out(f_in, (f_out, arg));
in(f_out, result);
...
```

The Process contains:

```
process ...| !f | ...
```

Abstracting TLS handshakes

- Assume TLS is secure. Other people have analysed it.
- Goal of TLS — creation of a secure channel.
  - Identifies the server.
- Write a "function" that
  - gets inputs from two places
  - constructs two new channels — client2server and server2client and sends them back to both places.
  - Verifies the identities, as necessary.

TLS handshaking process

```
private free tlsmatch.
let tlsmatcher =
    in(tlsmatch, TLSClient(username, servername, cl_back));
    in(tlsmatch, TLSServer(servercert, serversk, sr_back));
    let (=servername,serverpk) = readcert(servercert) in
    if pke(serversk) = serverpk then
    new cltosr; new srtocl;
    out(cl_back, (cltosr,srtocl));
    out(sr_back, (username, cltosr,srtocl)).
```

The process contains . . . | !tlsmatcher | . . .

What is wrong?

Handshake with the adversary

- Adversary should be able to write to tlsmatch.
  - But not read!
- Add to the process:

. . . | (! in(net,x); out(tlsmatch, x)) | . . .

Modeling collisions in the control code

- Given some x, it is easy to find y, such that $CC(x) = CC(y)$.
  - Even if the format of y is restricted.
- In our application, the challenge $x = (x_1, x_2)$ is a pair.
  - x_1 is chosen by the server we're protecting. x_2 might be adversarially chosen.
- We introduce a function $csc/2$, such that for each y and each code z, we have $CC(\,(y, csc(z, y))\,) = z$.

Modeling collisions in the control code

```
fun CCode/1.
fun ccodesuffixcoll/2.

equation CCode((x,ccodesuffixcoll(z,x))) = z.
```

- Support for equational theories is not a strong part of ProVerif.
- The equations must be convergent.

Performing security-sensitive operations

- Mobile-ID protocol protects the server — allows to identify clients.
- We verify the security of the protocol by letting the server
  - send a secret over the agreed TLS channel;
  - perform an end-event

  at the end of the protocol.
- How to model that the user is an honest one?

Modeling an honest user

- We put the names of honest users onto a secret channel.

private free ServerOK.

let user = new username; ($\langle$*user actions*$\rangle$ | !out(ServerOK, username)).

let server = . . . ! . . . let username = . . . in
. . . in(ServerOK, =username); $\langle$*sensitive stuff*$\rangle$.

Model of Mobile-ID

- Many users, some dishonest
- Many servers, some dishonest
- A single DigiDocService
- A single Mobile Operator

See the implementation

What if DDS is dishonest?

- Make DDS's secrets available to the adversary.
  - May delete DDS's process.

See the implementation.

Other properties: non-interference

- Let $P(\vec{x})$ be a process depending on variables $\vec{x}$.
- Informally, P does not preserve secrecy of $\vec{x}$, if
  - for some $\vec{M}$, $\vec{N}$
  - some attacker can observe the difference in behaviour of $P(\vec{M})$ and $P(\vec{N})$.
- e.g. $P(x, y) \equiv \texttt{new}\ k; \texttt{out}(c, (\{x\}_k, \{y\}_k))$ does not preserve the secrecy of (x, y).
- Indeed, the outputs made by $P(M, M)$ and $P(M, N)$ look different.
- Non-interference should be used if the set where the secrets come from is small.
- example: `proverif/examples/pi/noninterf/piyahalom`

Global synchronization — phases

- ProVerif's process definition allows the construct

$$\texttt{phase}\ n; P$$

  where n is an integer.

- P executes after the time point n has been reached. The commands preceding `phase` n execute before that point.

- Some applications, e.g. voting, have such synchronization points.

Observational equivalence

- ProVerif's messages may contain the construct

$$\texttt{choice}[M_1, M_2]$$

- This defines two processes:
  - One, where all `choice`-constructs are replaced with their left arguments.
  - Another, where all `choice`-constructs are replaced with their right arguments.
- ProVerif tries to find whether some attacker can observe the difference in behaviour of these two processes.
- example: `proverif/examples/pi/choice/pivote`
- A form of offline guessing attack