Protocol analysis using ProVerif
Attacker model

The attacker has full control over network. He can drop, halt, modify, substitute messages. The attacker decides who runs the protocols with whom.
Attacker model

The attacker has full control over the network. He can drop, halt, modify, and substitute messages. The attacker decides who runs the protocols with whom.
Attacker model

The attacker has full control over the network. He can drop, halt, modify, or substitute messages. The attacker decides who runs the protocols with whom.
**Attacker model**

- The attacker has full control over network.
  - He can drop, halt, modify, substitute messages.
Attacker model

- The attacker has full control over network.
  - He can drop, halt, modify, substitute messages.
- The attacker decides who runs the protocols with whom.
Attacker model

- The attacker has full control over network.
  - He can drop, halt, modify, substitute messages.
- The attacker decides who runs the protocols with whom.
ProVerif

- [ ] https://proverif.inria.fr
- Static analysis for cryptographic protocols under the perfect cryptography assumption
- Can check secrecy and correspondence properties
- Errs only to the safe side
  - If a protocol is insecure, then says so
  - If a protocol is secure, then sometimes may claim to have found an attack
- Principle: translate the protocol to a set of *Horn clauses*
  - Involves a little bit of abstraction
Horn clauses

\[ p_1(t_{11}, \ldots, t_{1k_1}) \land \cdots \land p_n(t_{n1}, \ldots, t_{nk_n}) \Rightarrow q(t'_1, \ldots, t'_m) \]

- \( p_1, \ldots, p_n, q \) — predicate symbols
  - from a fixed set; each with fixed arity
- \( t_*, t'_* \) — term
  - countable number of atoms (constants)
  - constructors (functional symbols) from a fixed set
- terms may contain term variables as subterms (in these slides, denoted with capital letters)
  - \( \land_i p_i(\ldots X \ldots) \Rightarrow q(\ldots X \ldots) \) means
    \[ \forall t \in T : \left( \land_i p_i(\ldots t \ldots) \Rightarrow q(\ldots t \ldots) \right) \]
- \( T \) — the set of all ground terms (without variables)
Examples

- A translation of a protocol always contains a unary predicate $a$
  - $a(X)$ means that the attacker can learn $X$
- A translation contains rules for composing and decomposing messages:
  - $a(pair(X, Y)) \Rightarrow a(X) \quad a(pair(X, Y)) \Rightarrow a(Y)$  \hspace{1cm} // (X,Y)
  - $a(X) \land a(Y) \Rightarrow a(pair(X, Y))$
  - $a(senc(K, X)) \land a(K) \Rightarrow a(X)$  \hspace{1cm} // symmetric encryption
  - $a(penc(pk(K), X)) \land a(K) \Rightarrow a(X)$  \hspace{1cm} // asymmetric encryption
  - $a(K) \land a(X) \Rightarrow a(sign(K, X))$  \hspace{1cm} // signature
  - $a(sign(K, X)) \Rightarrow a(X)$
  - $a(X) \Rightarrow a(h(X))$  \hspace{1cm} // hash
  - ...
- There are also rules for protocol steps
- There is a goal, stated as a boolean formula, whose truthfulness we need to verify.
Logic programming

- A logic program is a set of Horn clauses.
- $\forall X_1 \cdots \forall X_k (p_1 \land \cdots p_k \Rightarrow q) \equiv \forall X_1 \cdots \forall X_k (\neg p_1 \lor \cdots \neg p_k \lor q)$
- A formula is in CNF (conjunctive normal form) if it is of the form $\forall X( (L_{11} \lor \cdots \lor L_{1k_1}) \land \cdots \land (L_{n1} \lor \cdots \lor L_{nk_n}) )$ where
  - each $L$ is a literal — a predicate application or its negation.
- Denote this formula with $\{[L_{11}, \ldots, L_{1k_1}], \ldots, [L_{n1}, \ldots, L_{nk_n}]\}$
  - A set of sets, actually.
- There are known methods (resolution) that prove whether such a formula is satisfiable.
Recall our example

1. $A \rightarrow B : \{[K_A, N_A, K_{AB}]\}^{K_B}$
2. $B \rightarrow A : \{[N_A, N_B, K_B]\}^{K_A}$
3. $A \rightarrow B : \{[N_A, N_B]\}^{K_B}$
4. $B \rightarrow A : \{M\}^{K_{AB}}$

The attacker can have the 1st message by starting a new session

$$a(pk(A)) \land a(pk(B)) \Rightarrow a(penc(pk(B), triple(pk(A), na, k)))$$
Recall our example

1. $A \rightarrow B : \{[K_A, N_A, K_{AB}]\}_{K_B}$
2. $B \rightarrow A : \{[N_A, N_B, K_B]\}_{K_A}$
3. $A \rightarrow B : \{[N_A, N_B]\}_{K_B}$
4. $B \rightarrow A : \{M\}_{K_{AB}}$

- The attacker can have the 1st message by starting a new session

$$a(pk(A)) \land a(pk(B)) \Rightarrow a(penc(pk(B), triple(pk(A), na, k)))$$

Something is very wrong here... What $na$? What $k$?

- $na$ and $k$ would be different in each session. There must be a parameter “session ID”.
The first message

1. $A \rightarrow B : \{[K_A, N_A, K_{AB}]\}_{K_B}$
2. $B \rightarrow A : \{[N_A, N_B, K_B]\}_{K_A}$
3. $A \rightarrow B : \{[N_A, N_B]\}_{K_B}$
4. $B \rightarrow A : \{M\}_{K_{AB}}$

The attacker can have the 1st message by starting a new session

\[\text{a}(pk(A)) \wedge \text{a}(pk(B)) \wedge \text{a}(ld) \Rightarrow \text{a}(\text{penc}(pk(B),\ \text{triple}(pk(A),\ na[ld],\ k[ld])))\]
The first message

1. $A \rightarrow B : \{[K_A, N_A, K_{AB}]\}_{K_B}$
2. $B \rightarrow A : \{[N_A, N_B, K_B]\}_{K_A}$
3. $A \rightarrow B : \{[N_A, N_B]\}_{K_B}$
4. $B \rightarrow A : \{M\}_{K_{AB}}$

The attacker can have the 1st message by starting a new session

$$a(pk(A)) \land a(pk(B)) \land a(Id) \Rightarrow a(penc(pk(B), triple(pk(A), na[Id], k[Id])))$$

Attacker: “Dear Alice, please start session 5 with Bob”
- $k(5)$ will be exchanged

Attacker “Dear Alice, please start session 5 with me”
- Attacker learns $k(5)$
The first message (let us try again)

1. $A \rightarrow B : \{[K_A, N_A, K_{AB}]\}_{K_B}$
2. $B \rightarrow A : \{[N_A, N_B, K_B]\}_{K_A}$
3. $A \rightarrow B : \{[N_A, N_B]\}_{K_B}$
4. $B \rightarrow A : \{M\}_{K_{AB}}$

◎ Session ID must contain the roles of the parties.

\[
\text{a}(\text{pk}(A)) \land \text{a}(\text{pk}(B)) \land \text{a}(Id) \Rightarrow \\
\text{a}(\text{penc}(\text{pk}(B), \text{triple}(\text{pk}(A), \\
na[\text{pk}(A), \text{pk}(B), Id], k[\text{pk}(A), \text{pk}(B), Id])))}
\]
The second message

1. $A \rightarrow B : \{[K_A, N_A, K_{AB}]\}_{K_B}$
2. $B \rightarrow A : \{[N_A, N_B, K_B]\}_{K_A}$
3. $A \rightarrow B : \{[N_A, N_B]\}_{K_B}$
4. $B \rightarrow A : \{M\}_{K_{AB}}$

When Bob gets the 1st message, he responds with the 2nd

\[
\text{a}( \text{Id} \land \text{a}(\text{penc}(pk(B), \text{triple}(pk(A), Na, K)))) \Rightarrow \\
\text{a}(\text{penc}(pk(A), \text{triple}(Na, nb[pk(A), pk(B), Id], pk(B))))
\]
The third message

1. $A \rightarrow B : \{ [K_A, N_A, K_{AB}] \}_{K_B}$
2. $B \rightarrow A : \{ [N_A, N_B, K_B] \}_{K_A}$
3. $A \rightarrow B : \{ [N_A, N_B] \}_{K_B}$
4. $B \rightarrow A : \{ M \}_{K_{AB}}$

When Alice gets the 2nd message, she responds with the 3rd

\[ a(\text{penc}(pk(A), \text{triple}(na[pk(A), pk(B), Id], Nb, pk(B)))) \Rightarrow a(\text{penc}(pk(B), \text{pair}(na[pk(A), pk(B), Id], Nb)))) \]
The fourth message

1. $A \rightarrow B : \{[K_A, N_A, K_{AB}]\}_{K_B}$
2. $B \rightarrow A : \{[N_A, N_B, K_B]\}_{K_A}$
3. $A \rightarrow B : \{[N_A, N_B]\}_{K_B}$
4. $B \rightarrow A : \{M\}_{K_{AB}}$

- When Bob gets the 3rd message, he responds with the 4th...
- But only if he has participated in the session from the beginning
The fourth message

1. $A \rightarrow B : \{[K_A, N_A, K_{AB}]\}_{K_B}$
2. $B \rightarrow A : \{[N_A, N_B, K_B]\}_{K_A}$
3. $A \rightarrow B : \{[N_A, N_B]\}_{K_B}$
4. $B \rightarrow A : \{M\}_{K_{AB}}$

- When Bob gets the 3rd message, he responds with the 4th...
- But only if he has participated in the session from the beginning.
- When Bob has received the 1st and 3rd messages, he can respond with the 4th.

$$a(penc(pk(B), triple(pk(A), Na, K))) \land$$

$$a(penc(pk(B), pair(Na, nb[pk(A), pk(B), Id]))) \Rightarrow$$

$$a(senc(K, m))$$
Solving the system

- Is $a(m)$ derivable?
- You may ask a Prolog system (traditional logic programming). And it will answer...
Solving the system

- Is $a(m)$ derivable?
- You may ask a Prolog system (traditional logic programming). And it will answer...
- ...infinite loop.
  - To get $a(m)$, we could use some $a(f(m))$
  - To get $a(f(m))$, we could use some $a(f(f(m)))$
  - To get...
- The unification strategy of ProVerif is more geared towards such protocol representations.
Try to run ProVerif

- Demo
  Invoking the analyzer: ./proverif -in horn file
Try to run ProVerif

- Demo
  Invoking the analyzer: ./proverif -in horn file
- Try to reconstruct the attack
What went wrong

• The attacker gained access to the secret key of Alice and could decrypt her messages.
• Actually, ProVerif tells that the attacker generated himself the secret key of Alice.
• How could that have happened?
What went wrong

- The attacker gained access to the secret key of Alice and could decrypt her messages.
- Actually, ProVerif tells that the attacker generated himself the secret key of Alice.
- How could that have happened?
- Since A and B are term variables (i.e. can represent any party, as well as the attacker), the attacker will learn the secret if he takes the role A.
- We are interested in privacy only if Alice is an honest user.
The fourth message (revisited)

1. $A \rightarrow B : \{[K_A, N_A, K_{AB}]\}_{K_B}$
2. $B \rightarrow A : \{[N_A, N_B, K_B]\}_{K_A}$
3. $A \rightarrow B : \{[N_A, N_B]\}_{K_B}$
4. $B \rightarrow A : \{M\}_{K_{AB}}$

Let $sA$ and $sB$ be the secret keys (unknown to the attacker) of actual Alice and Bob (i.e. not the roles, but some honest users)

Only Bob will send $m$, and only to Alice.

\[
\text{a}(\text{penc}(\text{pk}(sB), \text{triple}(\text{pk}(sA), Na, K))) \wedge \\
\text{a}(\text{penc}(\text{pk}(sB), \text{pair}(Na, nb[\text{pk}(sA), \text{pk}(sB), Id]))) \Rightarrow \\
\text{a}(\text{senc}(K, m))
\]
Try to run ProVerif

- Demo
Try to run ProVerif

- Demo
- Try to reconstruct the attack
What went wrong

- Attacker plays Alice sending the first message to Bob
- Bob received it twice, responding to it both times
  - Fair enough
What went wrong

- Attacker plays Alice sending the first message to Bob
- Bob received it twice, responding to it both times
  - Fair enough
- But the adversary repeated the session identifier
  - Not good
  - To avoid that, newly generated values must contain all received messages so far.
The second message

1. $A \rightarrow B : \{[K_A, N_A, K_{AB}]\}_{K_B}$
2. $B \rightarrow A : \{[N_A, N_B, K_B]\}_{K_A}$
3. $A \rightarrow B : \{[N_A, N_B]\}_{K_B}$
4. $B \rightarrow A : \{M\}_{K_{AB}}$

⊙ When Bob gets the first message, he responds with the second

$$a(Id) \land a(penc(pk(B), triple(pk(A), N, K))) \Rightarrow a(penc(pk(A), triple(Na,

$nb[pk(A), pk(B), Id, penc(pk(B), triple(pk(A), Na, K))],

pk(B)])))$$
The fourth message

1. $A \rightarrow B : \{ [K_A, N_A, K_{AB}] \}_B$
2. $B \rightarrow A : \{ [N_A, N_B, K_B] \}_A$
3. $A \rightarrow B : \{ [N_A, N_B] \}_B$
4. $B \rightarrow A : \{ M \}_B$

$$a(penc(pk(sB), \text{triple}(pk(sA), Na, K))) \land a(penc(pk(sB), pair(Na, nb[pk(sA), pk(sB), Id, penc(pk(sB), \text{triple}(pk(sA), Na, K)]))) \Rightarrow$$

$$a(senc(K, m))$$
Try to run ProVerif

- Demo
Try to run ProVerif

- Demo
- A similar-looking attack...
Try to run ProVerif

- Demo
- A similar-looking attack...
  - The attacker messed up 1st and 2nd messages of different sessions.
  - This is actually a type flaw, as the attacker needs to make a key look like a nonce, and a symmetric key like an asymmetric key.
Try to run ProVerif

- Demo
- A similar-looking attack...
  - The attacker messed up 1st and 2nd messages of different sessions.
  - This is actually a type flaw, as the attacker needs to make a key look like a nonce, and a symmetric key like an asymmetric key.
- How to fix it?
  - We can use typed version of Horn clauses
  - We can add constants (e.g. fst() and snd()) to the first and the second messages respectively.
Correspondence assertions

- So far, we have analysed message secrecy.
- We may need some other important properties like "Does Bob always accept the same shared key as Alice does?"
  - The event \textbf{Bob accepts K} should happen only if \textbf{Alice accepts K} happened.
  - More generally, an event \textbf{end}(M) should happen only if \textbf{begin}(M) has happened (for some \(M\)).
  - ... take into account session IDs etc.
Correspondence assertions as Horn clauses

- Two more predicates, $b$ and $e$, for $\text{begin}$ and $\text{end}$.
- After a party has executed $\text{begin}(M)$, its following messages are translated with $b(M)$ as a premise.
  - $b(M) \land a(\cdots) \Rightarrow a(\cdots)$
  - $\cdots$ contains session IDs and received messages.
- Emitting $\text{end}(M)$ is adversary’s goal, hence it is the conclusion of a rule.
  - $a(\cdots) \Rightarrow e(M)$
- If $b(M)$ is necessary for $e(M)$, then we have (non-injective) agreement.
ISO 3-pass mutual authentication

1. $A \rightarrow B : N_A$
2. $B \rightarrow A : [\{N_A, N_B, K_A\}]_{K_B}$
3. $A \rightarrow B : [\{N_B, N_A, K_B\}]_{K_A}$

- From signature find the message.
- Public key $\equiv$ principal’s name.
- **end**($K_A, K_B$) executed by $B$ in the very end.
- **begin**($K_A, K_B$) executed by $A$ before 3rd message.
Injective agreement

- An agreement is **injective** if no two instances of **end** event can share the same **begin** event.
- Add the session identifier \( Z \) to the argument of \( e \).
- Add the session identifier and received messages \( Y \) to the argument of \( b \).
- If \( b((X, Y)) \) is necessary for \( e((X, Z)) \), and \( Z \) appears in \( Y \), then we have injective agreement.
Injective agreement (example)

Example that has agreement, which is not injective:

1. $A \rightarrow B : (A, B)$
2. $B \rightarrow A : [\{ N \}]_{K_B}$

Let **begin** event be executed by $B$ after 1st step, and **end** executed by $A$ after the 2nd step.

- There is agreement, as $A$’s signature verification fails, if $B$ has never signed anything.
- It is non-injective, as the attacker may resend the second message multiple times in different sessions.
Try to run ProVerif

- Demo
Intermediate take-aways

- Writing down protocols in Horn clauses is a non-trivial task.
- Technical transformations that we had to do (e.g. including previously received messages everywhere) could be done automatically.
- There exists more user-friendly *spi-calculus* interface of ProVerif.
ProVerif’s input/output 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 pitype file, if file contains the protocol specification in applied π-calculus.
  - ./proverif -in pitype -graph directory file, creates a .dot and a corresponding .pdf file with a picture representing the attack trace (if there was any) in the directory (that should already exist). It requires installation of graphviz.
A process $P$ is one of

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>does nothing</td>
</tr>
<tr>
<td>new $n; P'$</td>
<td>create new atom $n$, then $P'$</td>
</tr>
<tr>
<td>in($c, p$); $P'$</td>
<td>bind a message from channel $c$ to var. $p$, then $P'$</td>
</tr>
<tr>
<td>out($c, m$); $P'$</td>
<td>send the message $m$ on channel $c$, then $P'$</td>
</tr>
<tr>
<td>let $p = M$ in $P'$ else $P''$</td>
<td>bind $p$ to $M$, do $P'$ if success, $P''$ otherwise</td>
</tr>
<tr>
<td>$P_1</td>
<td>P_2$</td>
</tr>
<tr>
<td>!$P'$</td>
<td>replicate $P'$. We have !$P' \equiv P'</td>
</tr>
<tr>
<td>event $M; P'$</td>
<td>emit event $M$, then $P'$</td>
</tr>
</tbody>
</table>

A process represents all sessions of all parties.
Translation to Horn clauses (internal)

- Just two predicates:
  - \( \text{attacker}(\nu) \): the attacker can learn the value \( \nu \).
  - \( \text{mess}(c,\nu) \): message \( \nu \) can be transmitted over channel \( c \).
- If the attacker knows the channel name, he can read (i.e. intercept) and write on that channel.
- 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”.

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.
Recall our example

1. $A \rightarrow B : \{[K_A, N_A, K_{AB}]\}_{K_B}$
2. $B \rightarrow A : \{[N_A, N_B, K_B]\}_{K_A}$
3. $A \rightarrow B : \{[N_A, N_B]\}_{K_B}$
4. $B \rightarrow A : \{M\}_{K_{AB}}$

Let us write this down in pi-calculus.
Modeling an honest user

1. We can put the names of honest users onto a secret channel.
   private free honest : channel.

   let user = new Uname; (⟨user actions⟩ | !out(honest, Uname) ).

   let server = ...; let Uname = ... in
   ... in(honest, =Uname); ⟨sensitive stuff⟩.

2. We can use a table that is not accessible by the attacker.
   table honest(bitstring).

   let user = new Uname; insert honest(Uname); ⟨user actions⟩.

   let server = ...; let Uname = ... in
   ... get (honest, =Uname) in ⟨sensitive stuff⟩.
Global synchronization — phases

- ProVerif’s process definition allows the construct
  \[ \text{phase } n; P \]
  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.
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.
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 \( \text{out}(\text{net}, \text{cert}(p,k)) \).
- Adversary cannot construct certificates itself. Where does he get certificates for his own keys?

let simpleca = ! in(\text{net}, \text{pubkey}); \text{new} \, n; \text{out}(\text{net}, \text{cert}(n,\text{pubkey})).

Same with keys shared between phone and operator.

The process contains \( \ldots \mid \text{simpleca} \mid \ldots \)
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:
  
  \[
  \ldots | (\! \text{in}(\text{net}, x); \text{out}(\text{tlsmatch}, x)) | \ldots
  \]
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; (⟨user actions⟩ | !out(ServerOK, username) ).

let server = ... ! ... let username = ... in
    ... in(ServerOK, =username); ⟨sensitive stuff⟩.
```
Reading public databases

- How do the parties find certificates of other parties?
  - Just by
    - Receiving them from the network;
    - Checking that the name in the certificate is matches what they wanted to learn.

- This works for public registries where the rows have integrity.
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.