

Special topics in Logic and Security I

Master Year II, Sem. I, 2025-2026

Ioana Leuştean
FMI, UB

Prolog and Horn Clauses

Prolog program = set of facts and rules

A Prolog program is a set of *facts* and *rules* which define a given problem using a set of *first-order predicates*.

- **Facts:**

```
father(eddard, sansa).  
father(eddard, jonSnow).  
stark(eddard).  
stark(catelyn).
```

- **Rules:**

```
stark(X) :- father(Y, X), stark(Y).
```

- **Predicates:**

father/2
stark/1

Predicates, rules and questions

father(eddard, sansa).

father(eddard, jonSnow).

stark(eddard).

stark(catelyn).

stark(X) :- father(Y, X), stark(Y).

- In order to "run" a program we ask **questions**:

? – *stark(jonSnow)*

true

? – *stark(arya)*

false

? – *stark(X)*

X = eddard;

X = catelyn;

X = sansa;

X = jonSnow;

Logic programming

- Horn formulas are particular FOL formulas:

- definite clauses:

$\top \rightarrow P$ (facts)

$Q_1 \wedge \dots \wedge Q_n \rightarrow P$ (rules)

- queries (questions)

$Q_1 \wedge \dots \wedge Q_n \rightarrow \perp$

- the empty clause: \square

where Q_i, P are atomic formulas.

- In PROLOG we represent our knowledge as a set of definite clauses KB and our goal is to find answers for questions of the form $Q_1 \wedge \dots \wedge Q_n$.

$$KB \models Q_1 \wedge \dots \wedge Q_n$$

- The clauses from KB universally quantified.
- Q_1, \dots, Q_n are existentially quantified.

Horn clauses:

- Knowledge base KB = set of definite clauses:

father(eddard, sansa).

father(eddard, jonSnow).

stark(eddard).

stark(catelyn).

father(Y, X) \wedge stark(Y) \rightarrow stark(X)

- Queries (questions):

stark(eddard)

$\exists X$ stark(X)

SLD (Selected, Linear, Definite) resolution

Let KB be a knowledge base.

$$\text{SLD} \quad \frac{\neg Q_1 \vee \cdots \vee \neg Q_i \vee \cdots \vee \neg Q_n}{\theta(\neg Q_1 \vee \cdots \vee \neg P_1 \vee \cdots \vee \neg P_m \vee \cdots \vee \neg Q_n)}$$

where

- $Q \vee \neg P_1 \vee \cdots \vee \neg P_m$ is a definite clause from KB
- the variables of $Q \vee \neg P_1 \vee \cdots \vee \neg P_m$ and Q_i are renamed
- θ is an m.g.u for Q_i and Q .

SLD resolution

father(eddard, sansa).

father(eddard, jonSnow).

stark(eddard).

stark(catelyn).

stark(X) : –father(Y, X), stark(Y).

? – *stark(jonSnow)*

$$\text{SLD} \quad \frac{\neg Q_1 \vee \cdots \vee \neg Q_i \vee \cdots \vee \neg Q_n}{\theta(\neg Q_1 \vee \cdots \vee \neg P_1 \vee \cdots \vee \neg P_m \vee \cdots \vee \neg Q_n)}$$

- $Q \vee \neg P_1 \vee \cdots \vee \neg P_m$ is a definite clause from KB
- the variables of $Q \vee \neg P_1 \vee \cdots \vee \neg P_m$ and Q_i are renamed
- θ is an m.g.u for Q_i and Q .

SLD resolution

$\text{father}(eddard, sansa)$

$\text{father}(eddard, \text{jonSnow})$

$\text{stark}(eddard)$

$\text{stark}(\text{catelyn})$

$\neg\text{stark}(\text{jonSnow})$

$\neg\text{father}(Y, \text{jonSnow}) \vee \neg\text{stark}(Y)$

$\theta(X) = \text{jonSnow}$

$\text{stark}(X) \vee \neg\text{father}(Y, X) \vee \neg\text{stark}(Y)$

SLD

$$\frac{\neg Q_1 \vee \dots \vee \neg Q_i \vee \dots \vee \neg Q_n}{\theta(\neg Q_1 \vee \dots \vee \neg P_1 \vee \dots \vee \neg P_m \vee \dots \vee \neg Q_n)}$$

- $Q \vee \neg P_1 \vee \dots \vee \neg P_m$ is a definite clause from KB
- the variables of $Q \vee \neg P_1 \vee \dots \vee \neg P_m$ and Q_i are renamed
- θ is an m.g.u for Q_i and Q .

SLD resolution

father(eddard, sansa)

father(eddard, jonSnow)

stark(eddard)

stark(catelyn)

stark(X) ∨ ¬father(Y, X) ∨ ¬stark(Y)

$$\frac{\neg stark(jonSnow)}{\neg father(Y, jonSnow) \vee \neg stark(Y)}$$

$$\frac{\neg father(Y, jonSnow) \vee \neg stark(Y)}{\neg stark(eddard)}$$

$$\frac{}{\neg stark(eddard)}$$

□

As consequence, the answer to ?- stark(jonSnow) is true.

Prolog inference

father(eddard, sansa)

father(eddard, jonSnow)

stark(eddard)

stark(catelyn)

stark(X) :- father(Y, X), stark(Y).

ruler(stark, kingdom(north, winterfell)).

ruler(lannister, kingdom(rock, casterlyRock)).

capital(X, Y) :- ruler(X, kingdom(Y)).

? - *capital(X, Y).*

X = stark,

Y = winterfell;

X = lannister,

Y = casterlyRock.

[Back to protocols](#)

The Dolev-Yao attacker

Recall that we defined the attacker knowledge using a deductive systems, e.g.

- if $M \vdash t$ and $M \vdash k$ then $M \vdash \{t\}_k$
- if $M \vdash t_1$ and $M \vdash t_2$ then $M \vdash (t_1, t_2)$
- if $M \vdash \{t\}_k$ and $M \vdash k$ then $M \vdash t$
- if $M \vdash (t_1, t_2)$ then $M \vdash t_1$ and $M \vdash t_2$
- ...

where $M \vdash t$ means that t can be deduced knowing M

Can we specify the attacker knowledge by Horn clauses?

The Dolev-Yao attacker

Recall that a *Horn clause* has the form

$$H_1 \wedge \cdots \wedge H_n \rightarrow C$$

where H_1, \dots, H_n and C are atomic formulas in FOL.

We consider a unary predicate att such that

$\text{att}(t)$ means that the attacker knows t .

The attacker deduction system can be defined using implications as follows:

$$\text{att}(t) \wedge \text{att}(k) \rightarrow \text{att}(\{ t \}_k)$$

$$\text{att}(t_1) \wedge \text{att}(t_2) \rightarrow \text{att}((t_1, t_2))$$

$$\text{att}(\{ t \}_k) \wedge \text{att}(k) \rightarrow \text{att}(t)$$

$$\text{att}(t_1, t_2) \rightarrow \text{att}(t_1)$$

$$\text{att}(t_1, t_2) \rightarrow \text{att}(t_2)$$

which are obviously Horn clauses.

The Dolev-Yao attacker

$$\begin{aligned}att(t) \wedge att(k) &\rightarrow att(\{ t \}_k) \\att(t_1) \wedge att(t_2) &\rightarrow att((t_1, t_2)) \\att(\{ t \}_k) \wedge att(k) &\rightarrow att(t) \\att(t_1, t_2) &\rightarrow att(t_1) \\att(t_1, t_2) &\rightarrow att(t_2)\end{aligned}$$

These can be expressed in Prolog as follows:

```
att(senc(X, Y)) :- att(X), att(Y).  
att(pair(X,Y)) :- att(X), att(Y).  
att(X) :- att(senc(X, Y)), att(Y).  
att(X) :- att(pair(X, _)).  
att(Y) :- att(pair(_, Y)).
```

Note that Prolog (compound) terms - `pair(X,Y)` and `senc(X,Y)` - are used in order to define pairing and symmetric encryption.

Prolog: the Dolev-Yao attacker

Is Prolog suitable for such inferences?

```
att(ka).  
att(kb).  
att(senc(secret,pair(ka,kb))).  
att(senc(X, Y)) :- att(X), att(Y).  
att(pair(X,Y)) :- att(X), att(Y).  
att(X) :- att(senc(X, Y)), att(Y).  
att(X) :- att(pair(X, _)).  
att(Y) :- att(pair(_, Y)).  
  
?- att(secret).  
true
```

Prolog: the Dolev-Yao attacker

```
?- trace.  
true.
```

```
[trace] ?- att(secret).  
Call: (10) att(secret) ? creep  
Call: (11) att(senc(secret, _21612)) ? creep  
Exit: (11) att(senc(secret, pair(ka, kb))) ? creep  
Call: (11) att(pair(ka, kb)) ? creep  
Call: (12) att(ka) ? creep  
Exit: (12) att(ka) ? creep  
Call: (12) att(kb) ? creep  
Exit: (12) att(kb) ? creep  
Exit: (11) att(pair(ka, kb)) ? creep  
Exit: (10) att(secret) ? creep  
true .
```

Prolog: the Dolev-Yao attacker

We can further add rules for public encryption and signing:

```
att(pk(X)) :- att(X).  
att(X) :- att(aenc(X, pk(Y))), att(Y).  
att(aenc(X, Y)) :- att(X), att(Y).  
  
att(X) :- att(sign(X, Y)).  
att(sign(X,Y)) :- att(X), att(Y).  
  
att(X) :- att(senc(X, Y)), att(Y).  
att(senc(X, Y)) :- att(X), att(Y).  
  
att(X) :- att(pair(X, _)).  
att(Y) :- att(pair(_, Y)).  
att(pair(X,Y)) :- att(X), att(Y).
```

Can we follow the same ideas for protocols?

The Denning-Sacco protocol

We consider the Denning-Sacco protocol (without timestamps):

- (1) $A \rightarrow B : \{\{k\}_{sk(A)}\}_{pk(B)}$
- (2) $B \rightarrow A : \{\text{secret}\}_k$

The protocol is vulnerable to the following attack

- (att1) $A \rightarrow E : \{\{k\}_{sk(A)}\}_{pk(E)}$
- (att2) $E \rightarrow B : \{\{k\}_{sk(A)}\}_{pk(B)}$
- (att3) $B \rightarrow E : \{\text{secret}\}_k$

Since E is a corrupted agent, by (att1) the attacker knows the key k so, by (att3), he knows the *secret* (which was meant to be a secret between B and A).

Solution:

- (1) $A \rightarrow B : \{\{A, B, k\}_{sk(A)}\}_{pk(B)}$

The Denning-Sacco protocol

We consider the Denning-Sacco protocol (without timestamps):

- (1) $A \rightarrow B : \{\{k\}_{sk(A)}\}_{pk(B)}$
- (2) $B \rightarrow A : \{secret\}_k$

The communication between A and B can be represented (from the attacker's point of view) by:

```
att(senc(secret, K)) :- att(aenc(sign(K, skA), pk(skB))).
```

If B receives what he expects, then he behaves following the protocols and the messages are "added" to the adversary knowledge.

Note that the agents are identified with their secret keys.

The Denning-Sacco protocol

- (1) $A \longrightarrow B : \{\{k\}_{sk(A)}\}_{pk(B)}$
- (2) $B \longrightarrow A : \{secret\}_k$

We've defined the fact that B acts when he receives a message, but we still need to define that fact that A starts the protocol. To do this we can think that " A wants to speak to any principal (who's public key is known)":

```
att(aenc(sign(k, skA), pk(X))) :- att(pk(X)).
```

Moreover, we can express the fact that k is a key that depends on the communication partner:

```
att(aenc(sign(k(pk(X)), skA), pk(X))) :- att(pk(X)).
```

Prolog: the Denning-Sacco protocol - naive approach

We've shown that the protocol

- (1) $A \longrightarrow B : \{\{k\}_{sk(A)}\}_{pk(B)}$
- (2) $B \longrightarrow A : \{\text{secret}\}_k$

can be defined in Prolog by:

```
att(aenc(sign(K, skA), pk(X))) :- att(pk(X)).  
att(senc(secret, K)) :- att(aenc(sign(K, skA), pk(skB))).
```

Can we find the attack? This means that:

```
?- att(secret).  
true
```

Prolog: the Denning-Sacco protocol - naive approach

In order to analyze the protocol we need to represent the attacker inference system, as well as the adversary initial knowledge:

```
% initial knowledge
att(skI).
att(pk(skA)).
att(pk(skB)).

% the attacker inference system
att(pk(X)) :- att(X).
att(X) :- att(aenc(X, pk(Y))), att(Y).
att(aenc(X, Y)) :- att(X), att(Y).
att(X) :- att(sign(X, Y)).
att(sign(X,Y)) :- att(X), att(Y).
att(X) :- att(senc(X, Y)), att(Y).
att(senc(X, Y)) :- att(X), att(Y).

%the protocol
att(aenc(sign(K, skA), pk(X))) :- att(pk(X)).
att(senc(secret, K)) :- att(aenc(sign(K, skA), pk(skB))).
```

Prolog: the Denning-Sacco protocol - naive approach

```
att(skI).  
att(pk(skA)).  
att(pk(skB)).  
att(aenc(sign(K, skA), pk(X))) :- att(pk(X)).  
att(senc(secret, K)) :- att(aenc(sign(K, skA), pk(skB))).  
att(pk(X)) :- att(X).  
att(X) :- att(senc(X, Y)), att(Y).  
att(aenc(X, Y)) :- att(X), att(Y).  
att(sign(X, Y)) :- att(X), att(Y).  
att(X) :- att(aenc(X, pk(Y))), att(Y).  
att(X) :- att(sign(X, Y)).  
att(senc(X, Y)) :- att(X), att(Y).  
  
?- att(secret).  
true
```

Prolog: the Denning-Sacco protocol - naive approach

```
?- trace.  
[trace] ?- att(secret).  
Call: (10) att(secret) ? creep  
Call: (11) att(senc(secret, _16792)) ? creep  
Call: (12) att(aenc(sign(_16792, skA), pk(skB))) ? creep  
Call: (13) att(pk(skB)) ? creep  
Exit: (13) att(pk(skB)) ? creep  
Exit: (12) att(aenc(sign(_16792, skA), pk(skB))) ? creep  
Exit: (11) att(senc(secret, _16792)) ? creep  
Call: (11) att(_16792) ? creep  
Exit: (11) att(skI) ? creep  
Exit: (10) att(secret) ? creep  
true
```

Prolog: the Denning-Sacco protocol - naive approach

Why "naive"?

- The clause ordering affects the results or it can provide false attacks.
- The clause ordering affects termination.
- The rules with the conclusion $\text{att}(X)$ can always be applied.

However, the idea of using Horn clauses for modeling and analyzing protocols is valuable, and few tools and theories are known.

"The ProVerif tool takes protocols written in a variant of **the applied pi calculus** as input together with a security property to verify. The protocol is then **automatically translated into a set of first-order Horn clauses** and the properties are translated into derivability queries. The verification algorithm is based on a **dedicated Horn clause resolution** procedure."

[1] Véronique Cortier, Steve Kremer. Formal Models and Techniques for Analyzing Security Protocols: A Tutorial. Foundations and Trends in Programming Languages, Now Publishers, 2014, 1 (3), pp.117.

<https://hal.archives-ouvertes.fr/hal-01090874>

<https://bblanche.gitlabpages.inria.fr/proverif/>