Towards an Independent Semantics and Verification Technology for the HLPSL Specification Language

ARSPA’05 Workshop, Lisbon, 16 July 2005

Alexey Gotsman

joint work with Fabio Massacci and Marco Pistore

University of Trento (Italy)

{gotsman, massacci, pistore}@dit.unitn.it

http://www.dit.unitn.it/˜{gotsman, massacci, pistore}

About the Talk

- An algorithm for the translation of security protocol specifications in a subset of the HLPSL specification language to the applied pi calculus
  - An independent semantics of HLPSL
  - A way to verify HLPSL specifications through a process algebra

Outline

1. Motivation and an outline of the proposed approach
2. Description of HLPSL and the applied pi calculus
3. Main ideas underlying the translation algorithm
4. Semantical issues arising in connection with the translation
5. Experimental results

A Protocol Formalization Pitfall

- Informal specification languages:
  - Security research papers and standard bodies
- Formal languages:
  - Experts in formal verification
- **Problem:** The gaps between these can lead to misunderstandings in the meaning of the protocol and its goals
- **Solution:** Using formal protocol specification languages

Security Protocol Specification Languages

- ALSP
- BRUTUS
- CAPSL and CIL
- CASPER
- CVS
- HLPSL
- NAPTRL
- Spi calculus-based (e. g. ProVerif)

  - Many languages but no “dominant” one
  - Languages are too tied to back-ends?

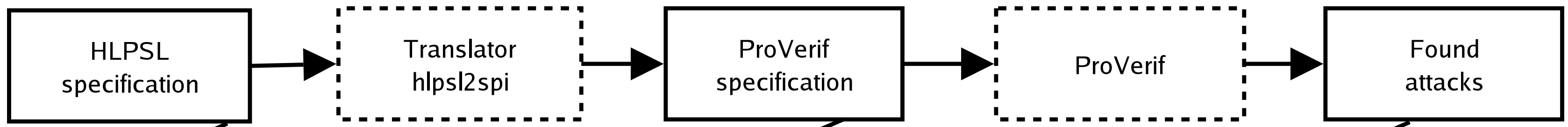
HLPSL Advantages

- Independently motivated semantics (Lamport's temporal logic of actions)
- Verification of HLPSL specifications (AVISPA tool):
  - SATMC – bounded model checking and satisfiability
  - OFMC – on-the-fly model checking
  - CL-AtSe – term rewriting
  - TA4SP – abstraction-based verification
  - **?** – process algebras

Proposed Approach and Its Outcomes

- Translation of specifications in a subset of HLPSL to a process algebra
  - The dialect of the applied pi calculus supported by the ProVerif tool
- Translation algorithm lets us verify protocols specified in HLPSL with the ProVerif tool
  - It completes the formalisms available for HLPSL
- Translation algorithm provides an independent semantics of HLPSL
  - It can be used to clarify ambiguities in specifications of HLPSL

Verification Scheme




```
role Alice (A,B : agent, Ka,Kb : public_key,
            Snd,Rcv : channel(dy)) played_by A def=
local State : nat, Na : text(fresh), Nb : text
init State = 0
accept State = 2
transition
   1. State=0 /\ Rcv(start) =|>
        Snd({A.Na'}Kb)
      /\ State'=1
      /\ witness(A,B,na,Na')
      /\ secret(Na',A)
      /\ secret(Na',B)
   2. State=1 /\ Rcv({Na.Nb'}Ka) =|>
        Snd({Nb'}Kb)
      /\ State'=2
      /\ wrequest (A,B,nb,Nb')
end role
...
```

```
...
let Alice =
      new Na;
      new Nb;
      Alice0.
let Alice0 =
      in(c, =start);
      new Na;
      event witness(A, B, na, Na);
      out(c, pencrypt((A, Na), Kb));
      if A <> i then
      if B <> i then
      out(c, sencrypt(secret1, Na));
      Alice1
      else Alice1
      else Alice1.
let Alice1 =
      in(c, m1);
      let (=Na, Nb) = pdecrypt(m1, inv(Ka)) in
      out(c, pencrypt(Nb, Kb));
      event wrequest(A, B, nb, Nb);
      Alice2.
let Alice2 = 0.
...
```

```
Goal of the attack:
end:wrequest(b[],a[],na[],Na_1[])
out(c, inv(ki))
out(c, ki)
out(c, kb)
out(c, ka)
out(c, i)
out(c, b)
out(c, a)
event(player(a,alice))
event(player(b,bob))
event(player(i,bob))
in(c, start)
event(witness(a,i,na,Na_1))
out(c, pencrypt((a,Na_1),ki))
in(c, pencrypt((a,Na_1),kb))
event(witness(b,a,nb,Nb_2))
out(c, pencrypt((Na_1,Nb_2),ka))
out(c, sencrypt(secret2,Nb_2))
in(c, pencrypt((Na_1,Nb_2),ka))
out(c, pencrypt(Nb_2,ki))
event(wrequest(a,i,nb,Nb_2))
in(c, pencrypt(Nb_2,kb))
event(wrequest(b,a,na,Na_1))
```

HLPSL – Role Specifications

- Protocol specifications are divided into roles
- Basic roles
  - Actions of one kind of participant:
    - parameters
    - initial state
    - transitions
- Composed roles
  - Role instantiations joined together

HLPSL – Transitions

- Transitions: $ev =|> act$
  - trigger event ev
  - action act
- Events:
  - comparisons of expressions
  - receiving of messages
- Actions:
  - assignments to variables
  - sending of messages
- The communication is synchronous and takes place over channels
- HLPSL allows for modeling protocols with non-linear structure

HLPSL – Goals

- Goals:
  - secrecy:
    `secrecy_of m`
  - weak authentication:
    `Alice weakly authenticates Bob on p`
    (`wrequest(b,a,p,m),witness(a,b,p,m)`)
  - strong authentication:
    `Alice authenticates Bob on p`
- Each goal corresponds to a temporal formula
- Goal facts:
  - `secret(m,a)`
  - `witness(a,b,p,m)`
  - `wrequest(a,b,p,m)`
  - `request(a,b,p,m)`

HLPSL – An Example (1)

$$
\begin{array}{llll}
(1) & A \rightarrow B & : & \{Na, A\}_{Kb} \\
(2) & B \rightarrow A & : & \{Na, Nb\}_{Ka} \\
(3) & A \rightarrow B & : & \{Nb\}_{Kb}
\end{array}
$$

```
role Alice (A,B : agent, Ka,Kb : public_key,
  Snd,Rcv : channel (dy)) played_by A def=
local State : nat, Na : text (fresh), Nb : text
init State = 0
accept State = 2
transition
    1. State = 0 /\ Rcv(start) =|>
            Snd({A.Na'}Kb)
         /\ State' = 1
         /\ witness(A,B,na,Na')
         /\ secret(Na',A)
         /\ secret(Na',B)
    2. State = 1 /\ Rcv({Na.Nb'}Ka) =|>
            Snd({Nb'}Kb)
         /\ State' = 2
         /\ wrequest (A,B,nb,Nb')
end role
```

HLPSL – An Example (2)


```
role Session (A,B: agent, Ka,Kb : public_key,
                  SA,RA,SB,RB : channel(dy)) def=
  composition
      Alice (A,B,Ka,Kb,SA,RA) /\
      Bob   (B,A,Kb,Ka,SB,RB)
end role

role Environment() def=
  const
      a,b,i : agent,
      ka,kb,ki : public_key,
      sa1,ra1,sb1,rb1,sa2,ra2,sb2,rb2 : channel(dy),
      na,nb : protocol_id
  knowledge(i) = {a,b,i,ka,kb,ki,inv(ki)}
  composition
      Session(a,b,ka,kb,sa1,ra1,sb1,rb1) /\
      Session(a,i,ka,ki,sa2,ra2,sb2,rb2)
end role

goal
      Alice weakly authenticates Bob on na
      Bob weakly authenticates Alice on nb
      secrecy_of Na, Nb
end goal
```

The Target Applied Pi Calculus

- The dialect of the applied pi calculus supported by the ProVerif tool
- The dialect extends the classical pi calculus with cryptographic primitives
- Destructors defined by reduction relations for defining cryptographic primitives
  - This approach is used in the ProVerif tool
  - It allows for more efficient verification
- Goals can be defined as restricted temporal formulas
- Events can be defined for stating goals

Translation Algorithm – Basic Ideas

- Send and receive operators of the applied pi calculus for translation of send and receive actions of HLPSL.
- DY intruder model $\Rightarrow$ receving/sending channel is irrelevant
- The restriction operator of the pi calculus for modeling the generation of fresh values for variables
- The + operator of the applied pi calculus for modeling choice in the execution of the role

Translation Algorithm – An Optimization

- A possible translation: $\left[\!\left[\begin{array}{c} ev_1 \;=|>\; act_1 \\ \ldots \\ ev_n \;=|>\; act_n \end{array} \right]\!\right] \triangleq \sum_{i=1}^{n} [\![ev_i]\!][\![act_i]\!]$
- We require that each basic role have an integer local variable `State` representing the state of the agent playing the role
  - $ev_i \triangleq \texttt{State} = s \wedge ev_i'$
  - $act_i \triangleq \texttt{State'} = s_1 \wedge act_i'$
- A basic role is translated to a set of processes B_s each acting as the role in the state s: $B_s \triangleq \sum_{k \in Tr(s)} [\![ev_k']\!][\![act_k']\!].B_{s_1^k}$
- The initial value of `State` determines the starting process

  - The optimization simplifies the translation
  - It facilitates mapping back found attacks into the protocol domain
  - All available HLPSL specifications are defined in this way

Translation Algorithm – An Example


```
role Alice (A,B : agent,
    Ka,Kb : public_key,
    Snd,Rcv : channel (dy))
    played_by A def=
local State : nat,
        Na : text (fresh),
        Nb : text
init State = 0
accept State = 2
transition
    1. State = 0 /\
          Rcv(start)=|>
        Snd({A.Na'}Kb)
    /\ State' = 1
    /\ witness(A,B,na,Na')
    /\ secret(Na',A)
    /\ secret(Na',B)
    2. State = 1 /\
          Rcv({Na.Nb'}Ka)=|>
        Snd({Nb'}Kb)
    /\ State' = 2
    /\ wrequest(A,B,nb,Nb')
end role
```

```
let Alice =
 new Na;
 new Nb;
 Alice0.
let Alice0 =
 in(c,=start);
 new Na;
 event witness(A,B,na,Na);
 out(c,pencrypt((A,Na),Kb));
 if A <> i then
 if B <> i then
 out(c,sencrypt(secr1,Na));
 Alice1
 else Alice1
 else Alice1.
let Alice1 =
 in(c,m1);
 let (=Na,Nb) =
   pdecrypt(m1,inv(Ka)) in
 out(c,pencrypt(Nb,Kb));
 event wrequest(A,B,nb,Nb);
 Alice2.
let Alice2 = 0.
```

Translation algorithm – Composed Roles

- We flatten the tree of composed roles
- We obtain only instantiations of basic roles with constants as arguments
- Instantiations are joined by the parallel composition operator of the applied pi calculus
- For each instantiation we introduce an instantiation identifier
- Flattening and introducing instantiation identifiers are useful:
  - for keeping track of roles played by agents
  - for formulating strong authentication goals

Translation Algorithm – Goals (1)

- Weak authentication goal
  `Alice weakly authenticates Bob on p`
- A possible translation:
  ```
  query ev:wrequest(x2,x1,p,m) ==>
  ev:witness(x1,x2,p,m)
  ```
- But this also translates
  `Bob weakly authenticates Alice on p`
  - A problem with AVISPA
- We must require that `x1` play `Alice` and `x2` play `Bob`

Translation Algorithm – Goals (2)

- We must require that `x1` play `Alice` and `x2` play `Bob`
- **Solution:** introduce an event `player(a,r)` – the agent `a` plays the role `r`
- **`role`** `Alice (...)` **`played_by`** `A...`
  $\Rightarrow$ `player(A,alice)`
- `query (ev:player(x2,bob)&ev:wrequest(x2,x1,p,m))==> (ev:witness(x1,x2,p,m)&ev:player(x1,alice))`
  - ProVerif does not allow such a query

Translation Algorithm – Goals (2)

- We must require that `x1` play `Alice` and `x2` play `Bob`
- **Solution:** introduce an event `player(a,r)` – the agent `a` plays the role `r`
- **`role`** `Alice (...)` **`played_by`** `A...`
  $\Rightarrow$ `player(A,alice)`
- `query (ev:player(x2,bob)&ev:wrequest(x2,x1,p,m))==> (ev:witness(x1,x2,p,m)&ev:player(x1,alice))`
  - ProVerif does not allow such a query
- For each b playing `Bob`
  `query ev:wrequest(`b`,x1,p,m) ==> (ev:witness(x1,`b`,p,m)&ev:player(x1,alice))`

Translation Algorithm – Goals (2)

- We must require that `x1` play `Alice` and `x2` play `Bob`
- **Solution:** introduce an event `player(a,r)` – the agent `a` plays the role `r`
- **`role`** `Alice (...)` **`played_by`** `A...`
  $\Rightarrow$ `player(A,alice)`
- `query (ev:player(x2,bob)&ev:wrequest(x2,x1,p,m))==> (ev:witness(x1,x2,p,m)&ev:player(x1,alice))`
  - ProVerif does not allow such a query
- For each b playing `Bob`
  `query ev:wrequest(`b`,x1,p,m) ==> (ev:witness(x1,`b`,p,m)&ev:player(x1,alice))`
- For each b playing `Bob`
  `query ev:wrequest(`b`,x1,p,m) ==> (ev:witness(x1,`b`,p,m)&ev:player(x1,alice))`$\,|\,$`x1=i,` $b \neq$ `i`

Semantical Issues

- Receiving of a private key through a channel
  - It is not possible either in AVISPA or in our tool
- Creation of fresh values
  - In the beginning of the protocol in an earlier version of AVISPA
  - Each time the transition is performed in subsequent versions and in our tool
- Taking into account roles in the authentication goals
  - No, in AVISPA at the moment
  - Yes, in our tool
- ProVerif does not support the $+$ operator
  - Our implementation performs the translation only for protocols without forks in computation

Experimental Results

Protocol	Time (s)	Properties			Attacks found
		Secrecy	Auth.	Type of auth.	
NSPK	0.04	2	2	Weak	Yes
NSPK-Lowe	0.07	2	2	Strong	No
SHARE	0.09	1	2	Weak	Yes
EKE	0.08	1	2	Weak	Yes
Chapv2	0.08	1	2	Strong	No
ISO1	0.02		1	Strong	Yes
ISO2	0.04		1	Strong	No
ISO3	–		2	Weak	–
ISO4	0.03		2	Strong	No
UMTS-AKA	0.05	1	2	Strong	No

Conclusions

- An algorithm for the translation of specifications in a subset of HLPSL to the applied pi calculus
- The usability of algorithm and the implementation
  - Analyzing different HLPSL specifications
- An independent semantics of HLPSL
- A way to verify HLPSL specifications through a process algebra
  - The algorithm completes the formalisms available for HLPSL

Future Work

- Extending the translator to handle the whole HLPSL language
- Giving a formal proof of correctness for the translation algorithm
  - Once HLPSL syntax and semantics stabilize
- Identifying more sophisticated protocols to assess the scalability of our approach
- Using other verification engines for pi calculi

Thank you