Towards Using Game Semantics for Cryptoprotocol Verification: Lorenzen Games

Jan Jürjens

Informatics
Technical University of Munich

juerjens@in.tum.de

http://www4.in.tum.de/~juerjens

Why semantics ?

Semantics: define translation of something into something else so that

- understanding the latter helps us understand the former
- we can use the translation as a basis for (mechanized) verification

Game semantics for verification

Verify crypto protocol implementations for security requirements (source code level).

Idea: Use game semantics as interaction-oriented foundation for program verification ?

Here as first step: Have translation from control flow graphs (e.g. generated from C source code) to first-order logic such as attacks are represented as proofs of the formula.

Use Lorenzen games to exhibit these attacks.

Recall: Lorenzen Games

1	$A \vee B$
2	?
1	choose A or B
2	attack chosen formula

1	$A \wedge B$
2	attack A or B
1	defend chosen formula

1	$A \Rightarrow B$
2	?
1	attack A or defend B

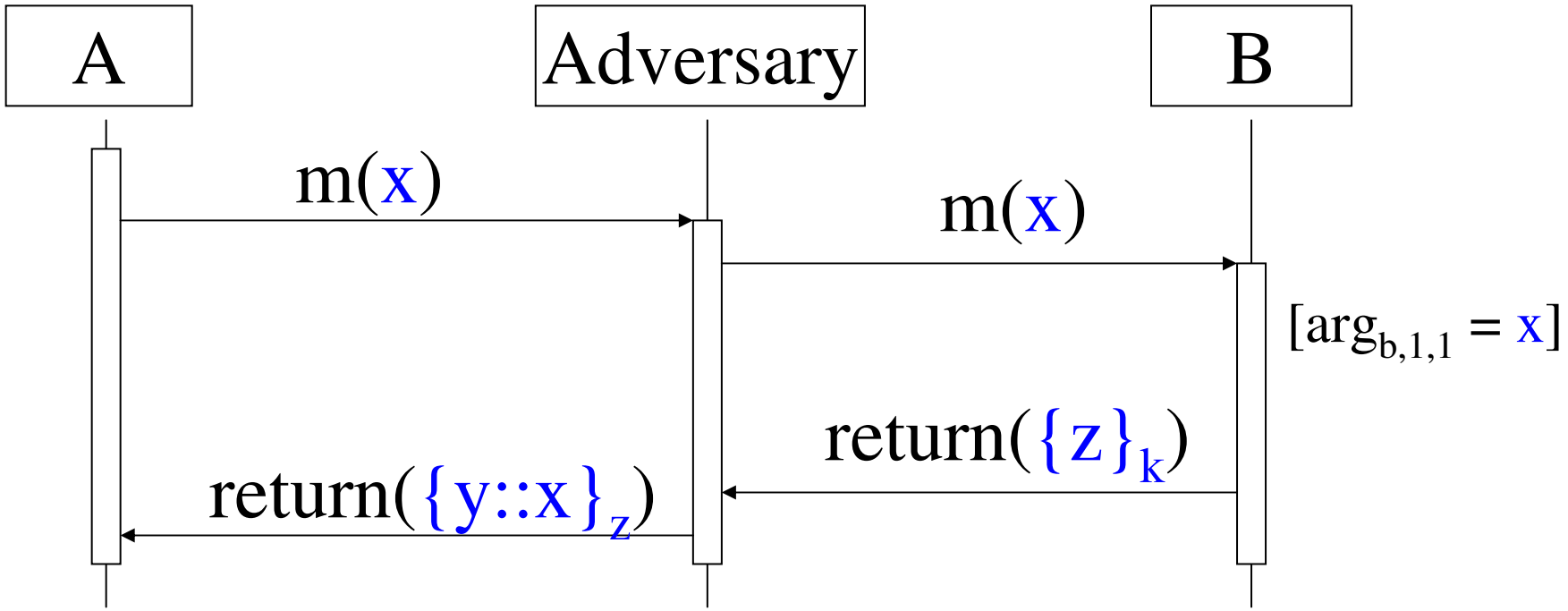
1	$\neg A$
2	?
1	attack A

Lorenzen Games: Modens Ponens

Modus Ponens:

1	$((A$	$\Rightarrow$	$B)$	$\wedge$	$A)$	$\Rightarrow$	B	
2						?		
1		?						
2	?							
1					?			
2					A			
1	A							
2			B					
1							B	

Dolev-Yao Adversary



Adversary knowledge:

k^{-1}, y, x

$\{z\}_k$, z

- $\forall e, k. Dec_{k^{-1}}(\{e\}_k) = e$

Cryptographic Expressions

Exp: term algebra generated by Var ∪ Keys ∪ Data and

- *_ :: _* (concatenation) and empty expression $\mathcal{E}$,
- *{ _ } _* (encryption)
- *Dec ( )* (decryption)
- *Sign ( )* (signing)
- *Ext_( )* (extracting from signature)
- *Hash(_)* (hashing)

by factoring out the equations $Dec_{K^{-1}}(\{E\}_k) = E$ and $Ext_K(Sign_{K^{-1}}(E)) = E$ (for *K*∈ Keys).

Security Analysis in First-order Logic

Idea: approximate set of possible data values flowing through system from above.

Predicate *knows(E)* meaning that the adversary may get to know *E* during the execution of the protocol.

For any secret *s*, check whether can derive *knows(s)* using automated theorem prover.

First-order Logic: Basic Rules

For initial adversary knowledge (K^0): Define *knows(E)* for any *E* initially known to the adversary (protocol-specific, e.g. K_A, K_A^{-1}).

Define above equations.

For evolving knowledge (K^n) define

$$\forall E_1, E_2.(knows(E_1) \wedge knows(E_2) \Rightarrow knows(E_1 :: E_2) \wedge knows(\{E_1\}_{E2}) \wedge knows(Dec_{E2}(E_1)) \wedge knows(Sign_{E2}(E_1)) \wedge knows(Ext_{E2}(E_1)))$$

$$\forall E.(knows(E) \Rightarrow knows(head(E)) \wedge knows(tail(E)))$$

Translate Protocol to FOL

Control flow graph transition

TR1=(in(msg_in),cond(msg_in),out(msg_out))

followed by *TR2* gives predicate *PRED(TR1)=*

$$\forall msg\_in.\ [knows(msg\_in) \wedge cond(msg\_in) \Rightarrow knows(msg\_out) \wedge PRED(TR2)]$$

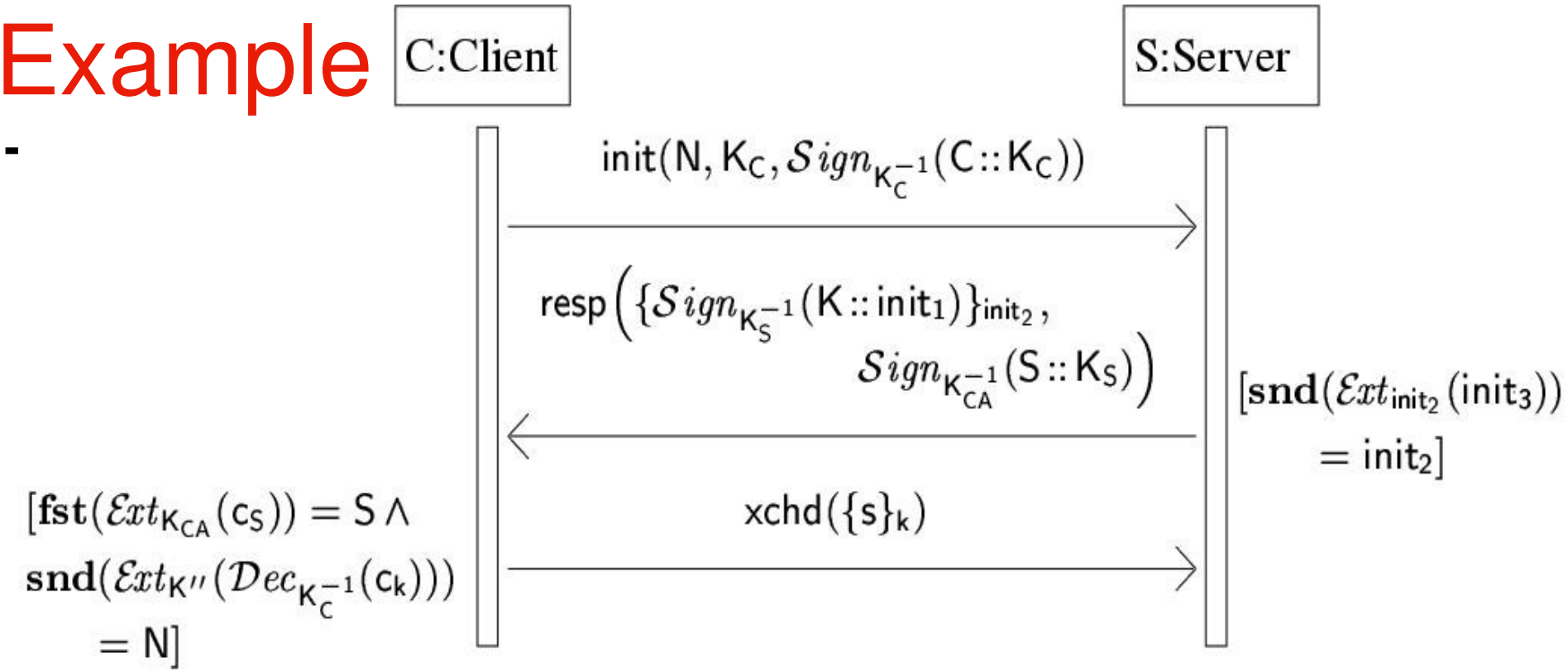
(Assume: order enforced (!).)

Can include senders, receivers in messages.

Abstraction: find all attacks, may have false positives, but efficient.

Example

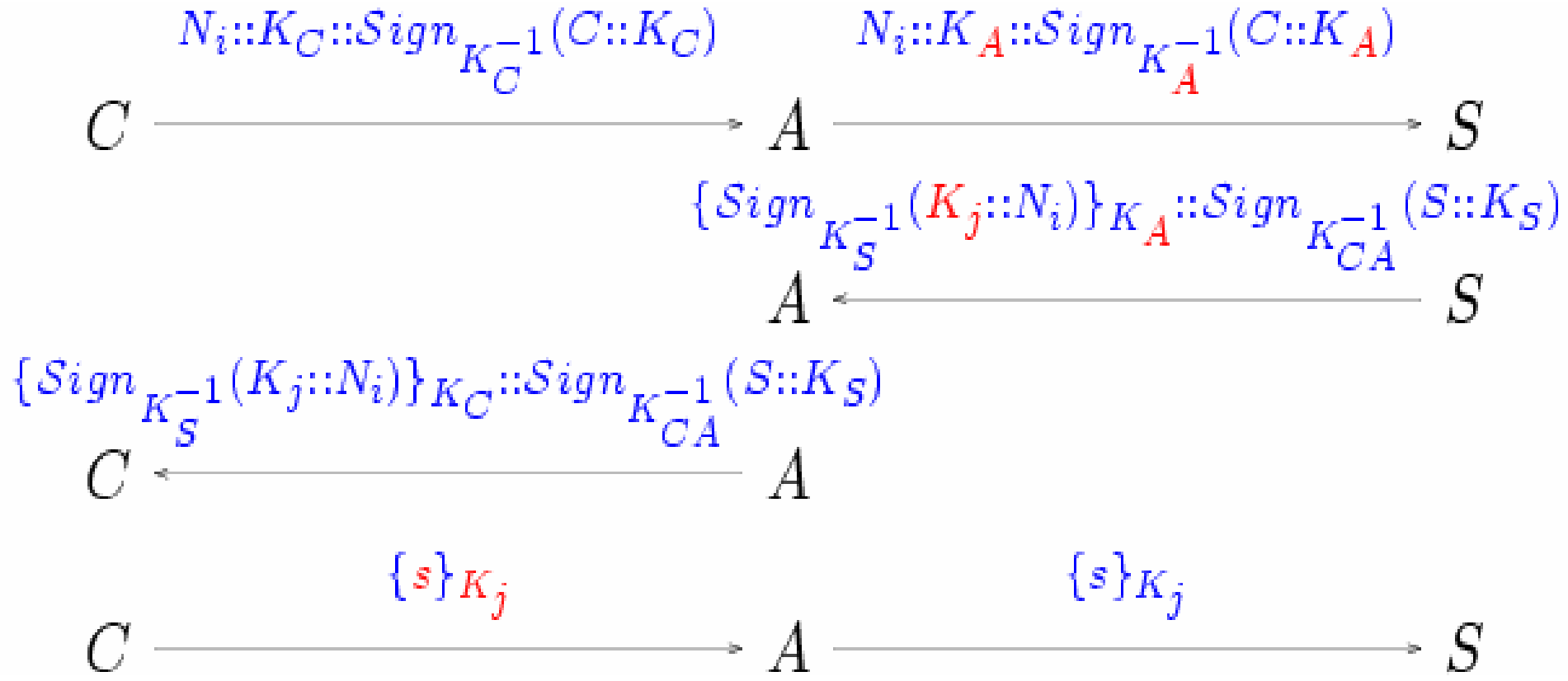
-



$knows(N) \wedge knows(K_C) \wedge knows(Sign_{KC\text{-}1}(C::K_C))$
$\wedge \forall init_1, init_2, init_3.[knows(init_1) \wedge knows(init_2) \wedge knows(init_3) \wedge snd(Ext_{init2}(init_3)) = init_2 \Rightarrow knows(\{Sign_{KS\text{-}1}(\ldots)\}_\ldots) \wedge [\ldots] \wedge [\ldots \Rightarrow \ldots]\ldots]$

Variant of TLS (IEEE INFOCOM 1999). knows(s) ?

Man-in-the-Middle Attack



Conclusions

First step towards using game semantics for security verification of crypto protocol implementation on source code level.

Next: from Lorenzen Games to game semantics.

Comments ?

… Which Means:

Can derive *knows(s)* (!).

That is: Protocol does not preserve secrecy of *s* against adversaries.

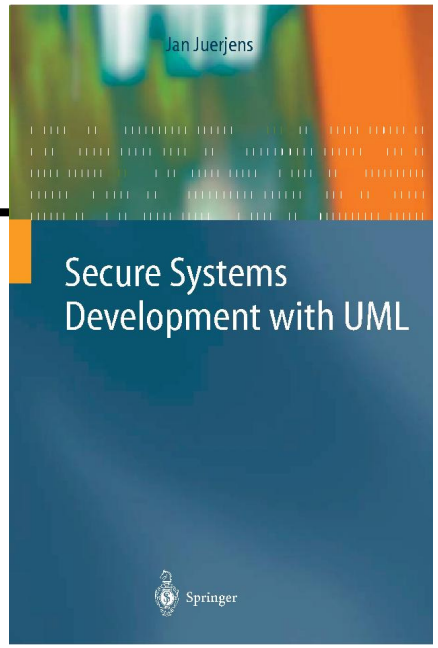
➔ Completely insecure wrt stated goals.

But why ?

Could look at proof tree.

Or: use prolog-based attack generator.

Resources



Jan Jürjens, Secure Systems Development with UML, Springer 2004

ICSE 2005

Spring School: May 2005, Carlos III Univ. Madrid

Workshops: WITS05, CSDUML05, …

More information (papers, slides, tool, …):
http://www.umlsec.org

Abstraction

Enable efficient automated analysis by abstraction.

Use lookup tables for functions or code-blocks:

- symbolic representation of cryptographic or arithmetic routines
- technical infrastructure (packet_send, buffer_copy, …)
- data structures (e.g. a->b)

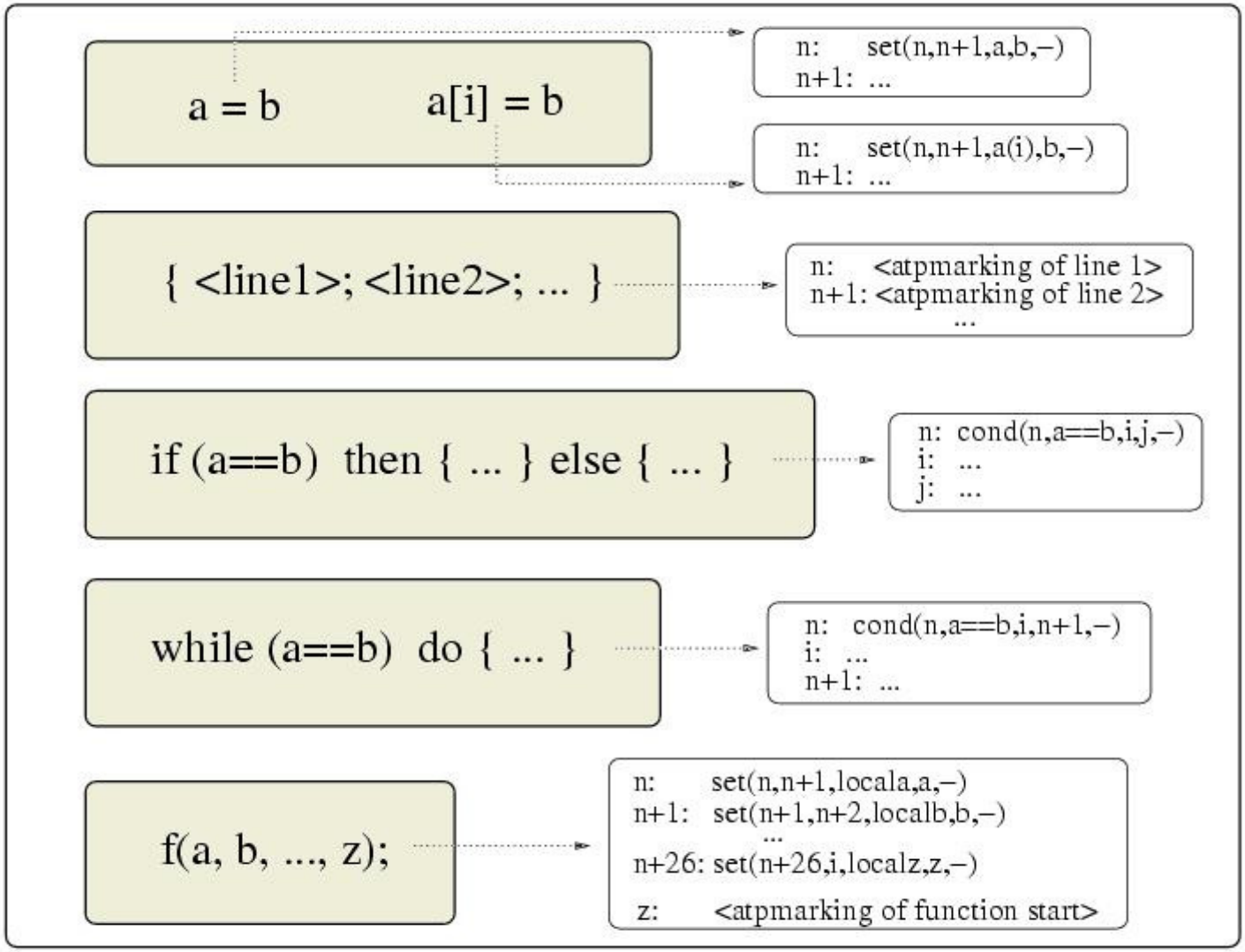
No pointers usage and arithmetic.

Can verify e.g. using Isabelle that abstractions are faithful.

Example Lookup Table

```
secret                                          => <data secretdata>
signature                                       => <data sig>;

server_host_key                                 => <data serverhostkey>;
shared_secret                                   => <data sharedsecret>;

simple_crypto_hash(.1.,.2.,.3.)                 => <lib hash(conc(.1.,.3.)>;
tmp_hash                                        => <data tmphash>;

key_verify(.1.,.2.,.3.,.4.,.5.)                 => <lib verify(.2.,.1.,.4.)>,
fatal(.1.)                                      => <end>;

DES_encryption(.1.,.2.)                         => <lib enc(.1.,.2.)>;

in_init() * in_read(.1.) * in_close()           => <in .1.>;
out_init() * out_write(.1.) * out_close()       => <out .1.>;
```

Annotated Code



```
void simplecall(char *shared_secret, char *kbuf, int klen, char *secret)
/* #ATP# state(1,2) */
{
   char *signature;
   int slen;

   /* read input */
   in_init();
   in_read(server_host_key);
   /* #ATP# io(2,3,serverhostkey,-,-) */
   in_read(signature);
   /* #ATP# io(3,4,sig),-,-) */
   in_read(slen);

   /* compute hash of server_host_key and length with shared secret */
   tmp_hash = simple_crypto_hash(
      server_host_key,     /* server public key (ks) */
      shared_secret   );  /* shared secret for the crypto hash (k) */
   /* #ATP# set(4,5,tmphash,hash(conc(serverhostkey,sharedsecret)),-) */

   /* check signature where hash is the signed value */
   if (key_verify(server_host_key, signature, slen, tmp_hash, 20) == false)
      fatal("key_verify failed for server_host_key");
   /* #ATP# cond(5,verify(sig,serverhostkey,tmphash)==false,0,6,-) */

   /* send out data 'secret' encrypted with server_host_key
   out_init();
   out_write(DES_encryption(secret,server_host_key));
   /* #ATP# io(6,7,-,enc(secretdata,serverhostkey),-) */

   out_close();
   in_close();
   xfree(kbuf);
}
```

Translation to Logic

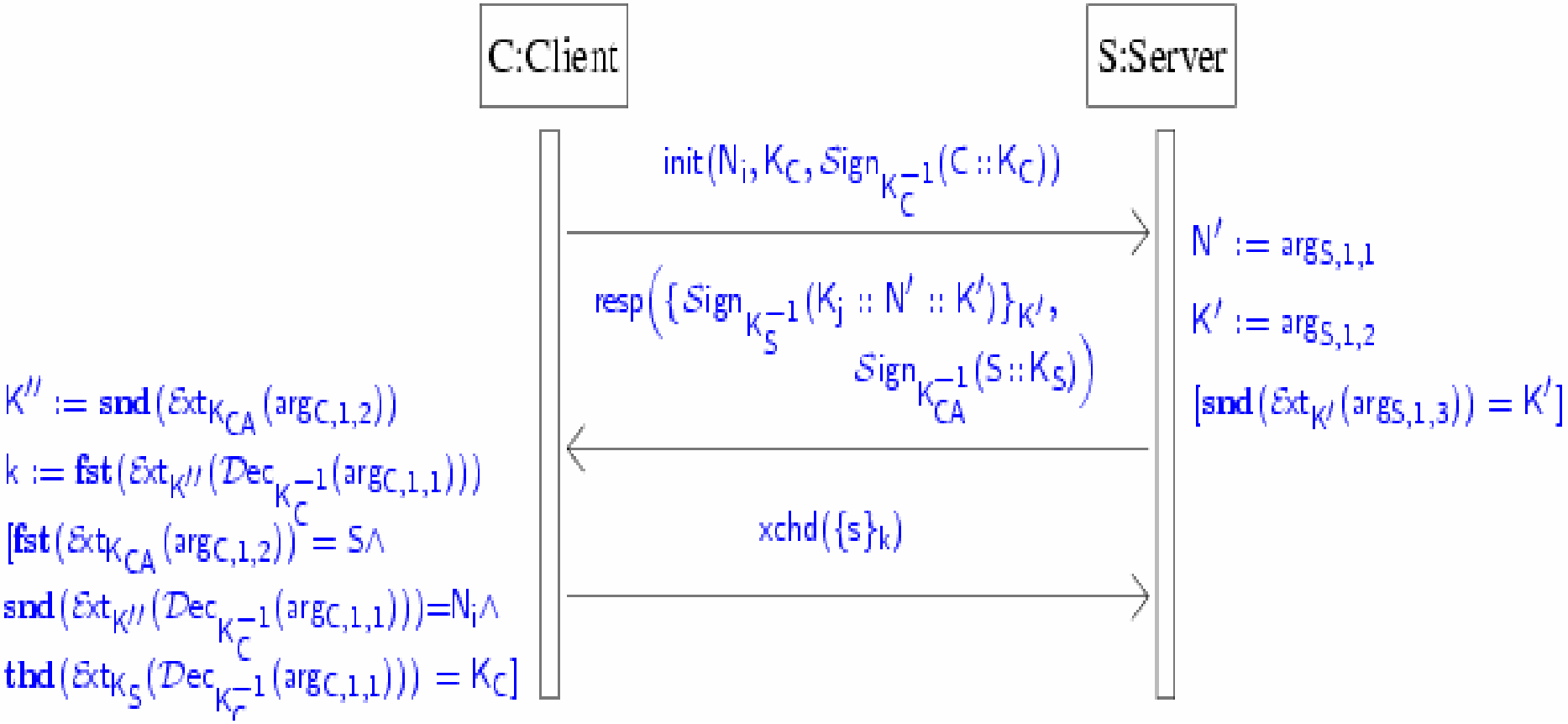
Similar as explained earlier.

Loops: Abstract away or break up after maximal iteration number.

Example:

```
(true & true => true % state 1
  & (knows(serverhostkey) & true => true % state 2
    & (knows(sig) & true => true % state 3
      & (true & tmphash = hash(conc(serverhostkey,sharedsecret)) => true % state 4
        & (true & not(verify(sig,serverhostkey,tmphash)=false) => true % state 5
          & (true & true => knows(enc(secretdata,serverhostkey)))))))) % state 6
```

The Fix



e-Setheo: *knows(s)* not derivable. Thus secure.