Verification of Low-level Crypto-Protocol Implementations Using Automated Theorem Proving

Jan Jürjens
(contrib. Mark Yampolskiy)

Software & Systems Engineering
TU Munich

juerjens@in.tum.de

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

Software Engineering & Security

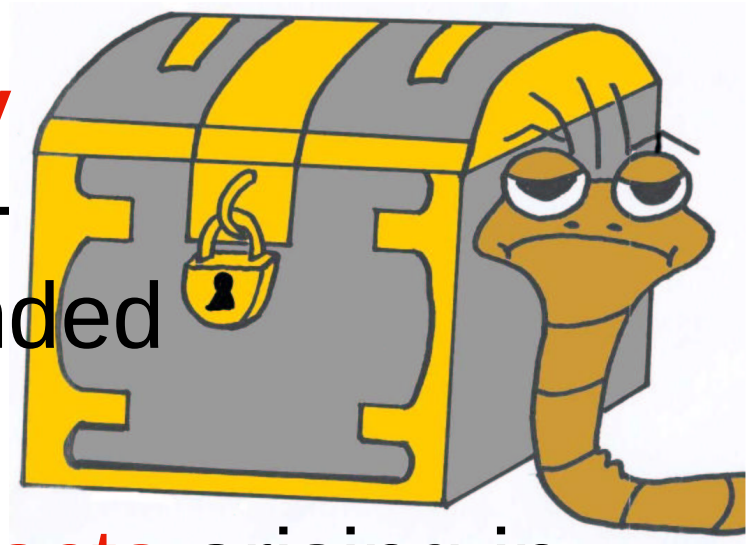
„Penetrate-and-patch“ (aka „banana strategy):

- insecure
- disruptive

Traditional formal methods: limited adoptation in industry.

- training people
- constructing formal specifications.

Model-based Security



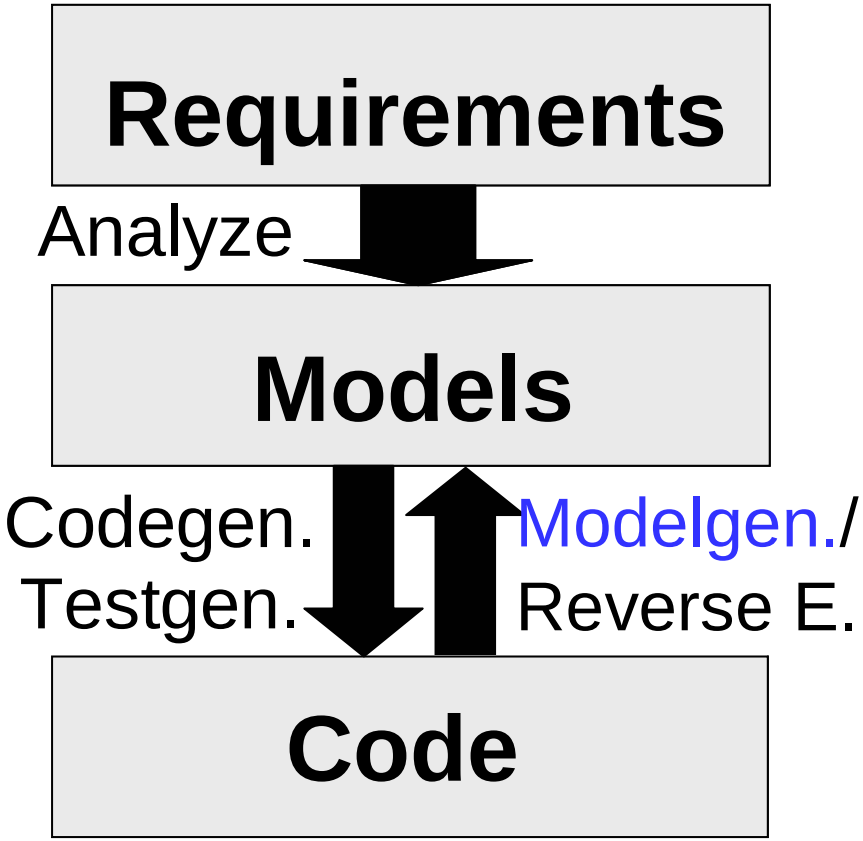
Increase security with bounded investment in time, costs:

- Extract models from artefacts arising in industrial development and use of security-critical systems (UML models, source code, configuration data).
- Tool-supported, theoretically sound, efficient automated security analysis.

Model-based Security Engineering

- Analyze (UMLsec) models against security requirements.
- Generate code (or tests) from models.
- Generate models from evolving or legacy code.

Goal: model-based = source-code based.



Security Analysis of C-Programs

Goal: Logic-based security verification of C programs which is as

- automatic and
- complete

as possible.

Note: can‘t be both perfectly automated and complete: Security in general undecidable.

Here: emphasize automation.

Crypto-Hardware API‘s (PKCS 11)

Here: support correct use of PKCS 11 API‘s.

Keep track of session handles.

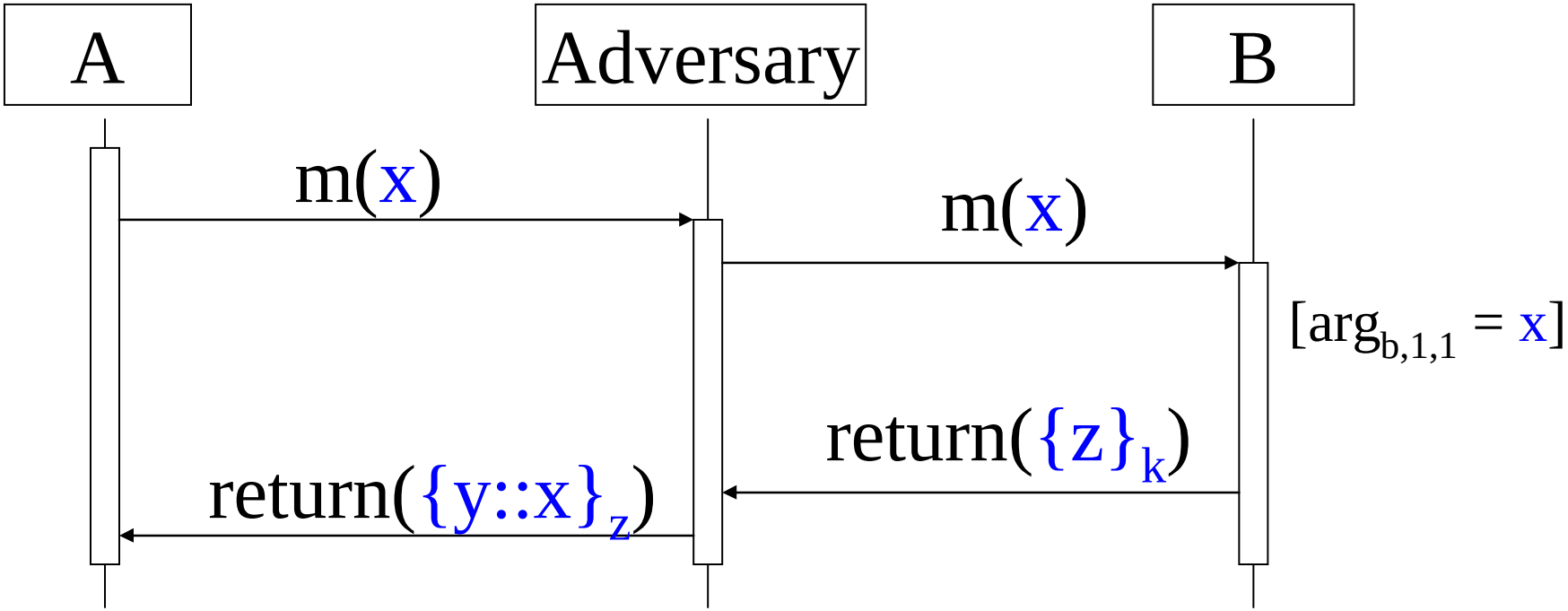
C_GetAttributeValue	obtains attribute value
C_SetAttributeValue	modifies attribute value
C_Encrypt	encrypts single-part data
C_Decrypt	decrypts encrypted data
C_Digest	digests single-part data
C_Sign	signs single-part data
C_VerifyRecover	verifies signature, data recovered
C_GenerateKey	generates a secret key
C_GenerateKeyPair	generates a key pair
C_GenerateRandom	generates random data

Security Analysis

Following Dolev, Yao (1982): To analyze system, verify against attacker model from threat scenarios in deployment diagrams who

- may participate in some protocol runs,
- knows some data in advance,
- may intercept messages on some links,
- injects messages that it can produce in some links
- may access certain nodes.

Adversary: Simulation

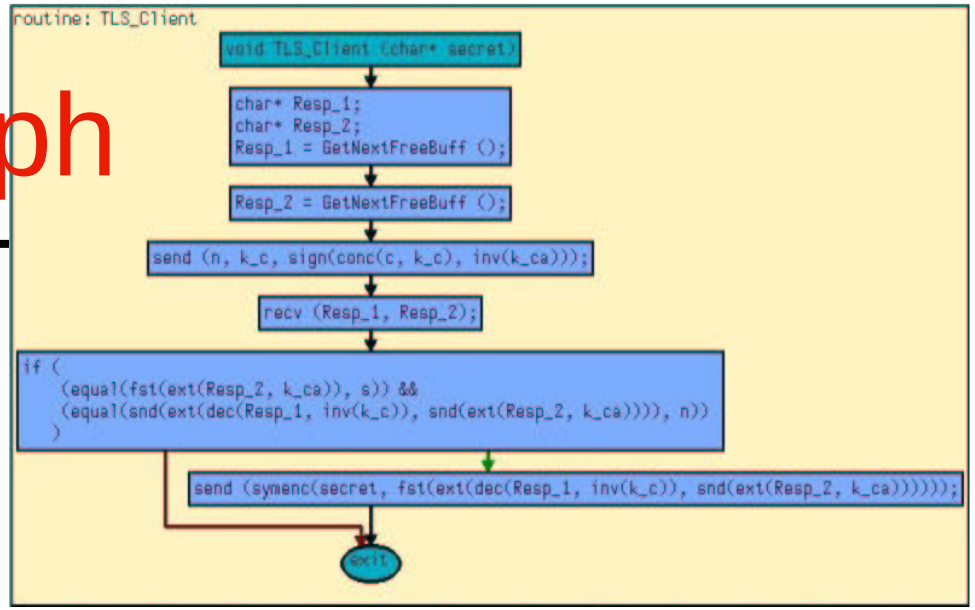


Adversary knowledge:

k^{-1}, y, x
$\{z\}_k$, z

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

Control Flow Graph



Generate control flow graph (e.g. with aicall (Absint, Germany)).

Transform to Mealy Machines:

trans (state,inpattern,condition,action,nextstate)

where action can be outpattern or localvar:=value.

Translate to abstract interpretation in FOL formula.

Security Analysis in First-order Logic

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.

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

Cryptographic Expressions I

Exp: quotient of term algebra generated from sets *Data*, *Keys*, *Var* of symbols using

- *_::_* (concatenation), *head(_)*, *tail(_)*,
- $(\_)^{-1}$ (inverse keys)
- *{ _ }_* (encryption)
- *Dec_( )* (decryption)
- *Sign_( )* (signing)
- *Ext_( )* (extracting from signature)

(each w. session parameter) and equations:

Cryptographic Expressions II

- $\forall E,K,S,S',S''.Dec_{K;S''}^{-1}(\{E;S\}_K;S')=E$
- $\forall E,K,S,S',S''.Ext_K(Sign_{K;S''}^{-1}(E;S);S')=E$
- $\forall E_1,E_2,S,S'.head(E_1::_S E_2;S')=E_1$
- $\forall E_1,E_2,S,S'.tail(E_1::_S E_2;S')=E_2$

Write $E_1::E_2::E_3$ for $E_1::(E_2::E_3)$ and $fst(E_1::E_2)$ for $head(E_1::E_2)$ etc.

Can include further crypto-specific primitives and laws (XOR, …).

First-order Logic: Basic Rules

Define *knows(E)* for any *E* initially known to the adversary.

For evolving knowledge define

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

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

… Translate to 1st Order Logic

Graph transition

$TR1=(in(msg\_in),cond(msg\_in),out(msg\_out))$

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

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

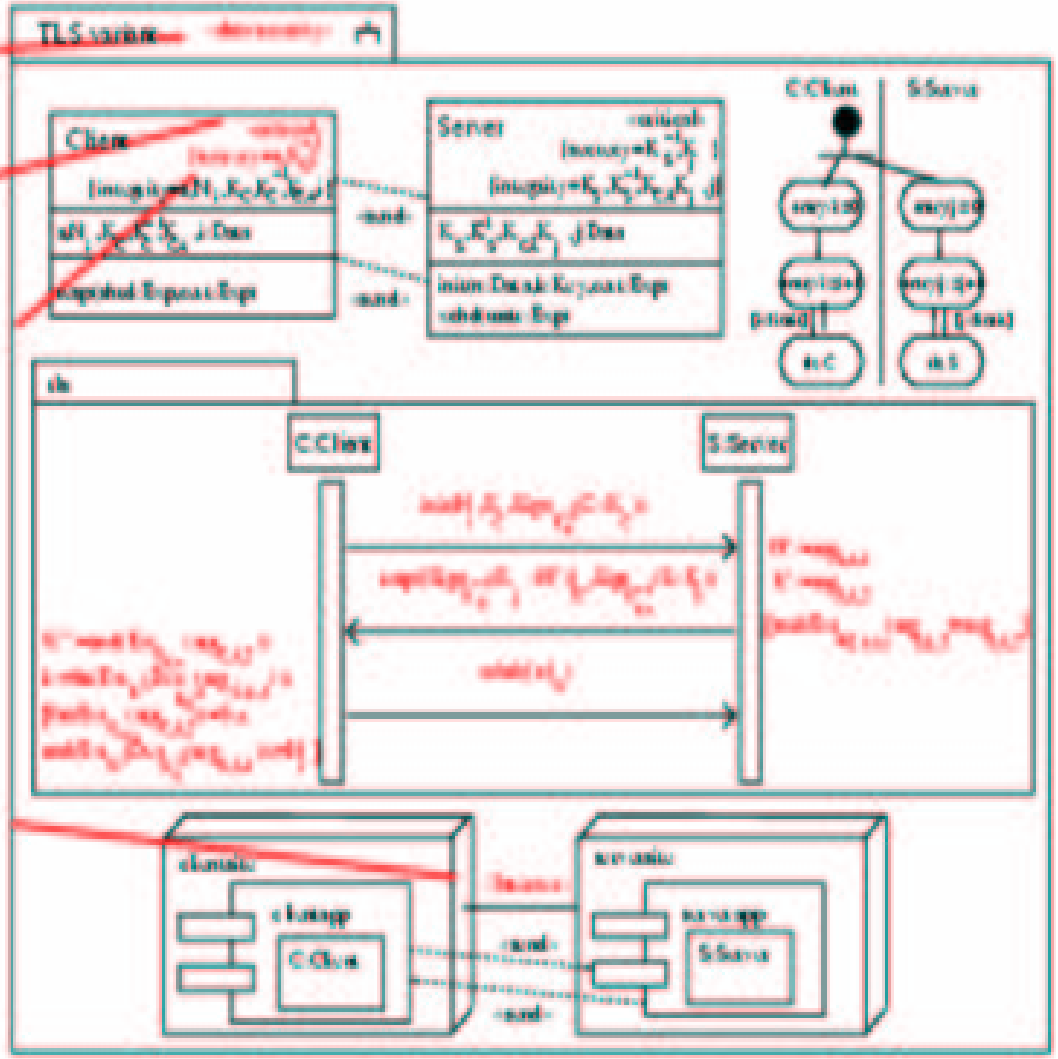
Abstraction (e.g. from senders, receivers): find all attacks, may have false positives.

Check whether can derive threat conjecture (e.g. $knows(s)$ for a secret s) from axioms.

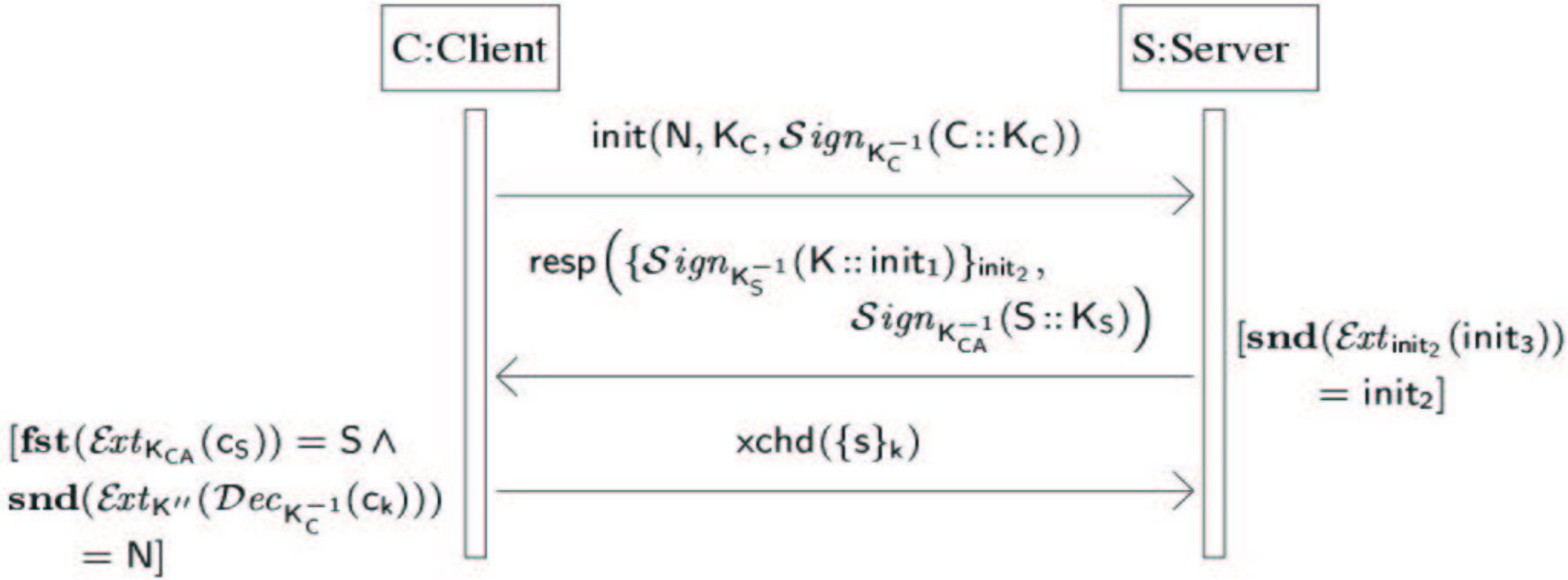
Example: Proposed Variant of TLS (SSL)

Presented at IEEE Infocom 1999.

Goal: send secret protected by session key using fewer server resources.



Example: Translation to Logic



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

Preprocessing of code

For simplification and efficiency apply semantics-preserving transformations to the C program:

- use of single-assignment variables
- eliminate side effects
- factor out pointer arithmetic

Analyze loops up to a fixed number of iterations. No so much of a restriction when analyzing secure interactions in crypto protocols.

Include annotations to abstract cryptofunctions etc.

routine: TLS_Client

```
void TLS_Client (
                CK_SESSION_HANDLE hSession,    // session handle
                CK_BYTE_PTR       pSecretData, // secret as plaintext data
                CK_ULONG          ulDataLen    // plaintext bytes
               )
```

```
CK_BYTE     Resp_1 [MESSAGEBUFF_MAXLEN];
CK_BYTE     Resp_2 [MESSAGEBUFF_MAXLEN];
CK_BYTE_PTR pEncryptedData = NULL;
// allocate and prepare buffers
pEncryptedData = (CK_BYTE_PTR) malloc (ulDataLen);
```

```
memset (pEncryptedData, 0x00, ulDataLen);
```

```
send (n); // C->S: Init
```

```
send (k_c);
```

```
send (Cert_c_ca); // => sign(conc(c, k_c), inv(k_ca))
```

```
recv (Resp_1); // S->C: Receive Server's respond
```

Client I

Client II

```
recv (Resp_2);
```

```
// Check Guards
if ((memcmp(fst(C_Ext(hSession, Resp_2, ID_k_ca)),
            s, MESSAGEBUFF_MAXLEN) == 0) &&
    (memcmp(snd(C_Ext(hSession, C_Dec(hSession, Resp_1,
                                C_Inv(hSession, ID_k_c)),
                snd(C_Ext(hSession, Resp_2, ID_k_ca)))),
                              n, MESSAGEBUFF_MAXLEN) == 0))
```

```
// C->S: Send Secret
send (C_Symenc(hSession, pSecretData, fst(C_Ext(hSession,
      C_Dec(hSession, Resp_1, C_Inv(ID_k_c)),
      snd(C_Ext(hSession, Resp_2, ID_k_ca))))));
```

```
// free temporare buffers
free (pEncryptedData);
```

exit

Surprise …

```
E-SETHEO csp03 single processor running on host ..
(c) 2003 Max-Planck-Institut fuer Informatik and
    Technische Universitaet Muenchen

tlsvariant-freshkey-check.tptp
...
analyzing results ...
proof found
time limit information: 298 total / 297 strateg
...
e-SETHEO done. exiting
```

Attack

Can derive *knows(s)*.

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

➔ Completely insecure wrt stated goals.

But why ? Use prolog-based attack generator.

Man-in-the-Middle Attack

$$C \xrightarrow{N_i::K_C::Sign_{K_C^{-1}}(C::K_C)} A \xrightarrow{N_i::K_A::Sign_{K_A^{-1}}(C::K_A)} S$$

$$A \xleftarrow{\{Sign_{K_S^{-1}}(K_j::N_i)\}_{K_A}::Sign_{K_{CA}^{-1}}(S::K_S)} S$$

$$C \xleftarrow{\{Sign_{K_S^{-1}}(K_j::N_i)\}_{K_C}::Sign_{K_{CA}^{-1}}(S::K_S)} A$$

$$C \xrightarrow{\{s\}_{K_j}} A \xrightarrow{\{s\}_{K_j}} S$$

Biometric Authentication System

In development by large German company.

In joint project, use presented security analysis tools at given UML specification.

So far, have discovered three major attacks against subsequently improved versions (misuse counter circumvented by dropping / replaying messages, smart-card insufficiently authenticated by recombing sessions).

Conclusions

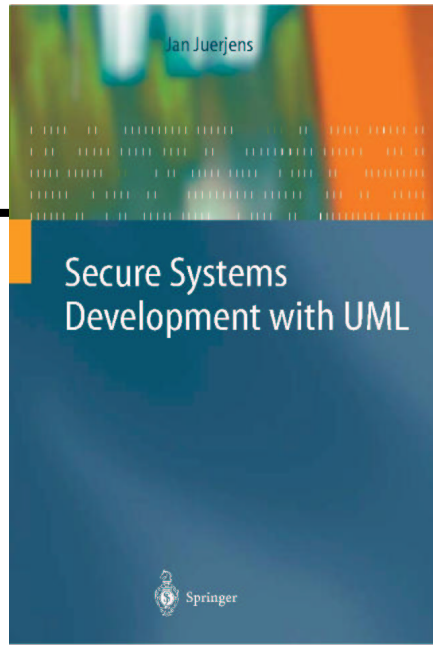
Code security verification using assertions:

- formally based approach
- automated tool support
- industrially used programming language
- integrated approach (specifications, source-code, configuration data)

Future:

- Virtual C execution machine in tptp instead of manual code tptp annotations…

Resources



TUM TB Dez. '04 (with T. Kuhn)

International Conference for Software Maintenance 2005

Models / UML 2005 (with S. Houmb)

More information (papers, slides, tool, …):
http://www4.in.tum.de/~juerjens

Abstraction

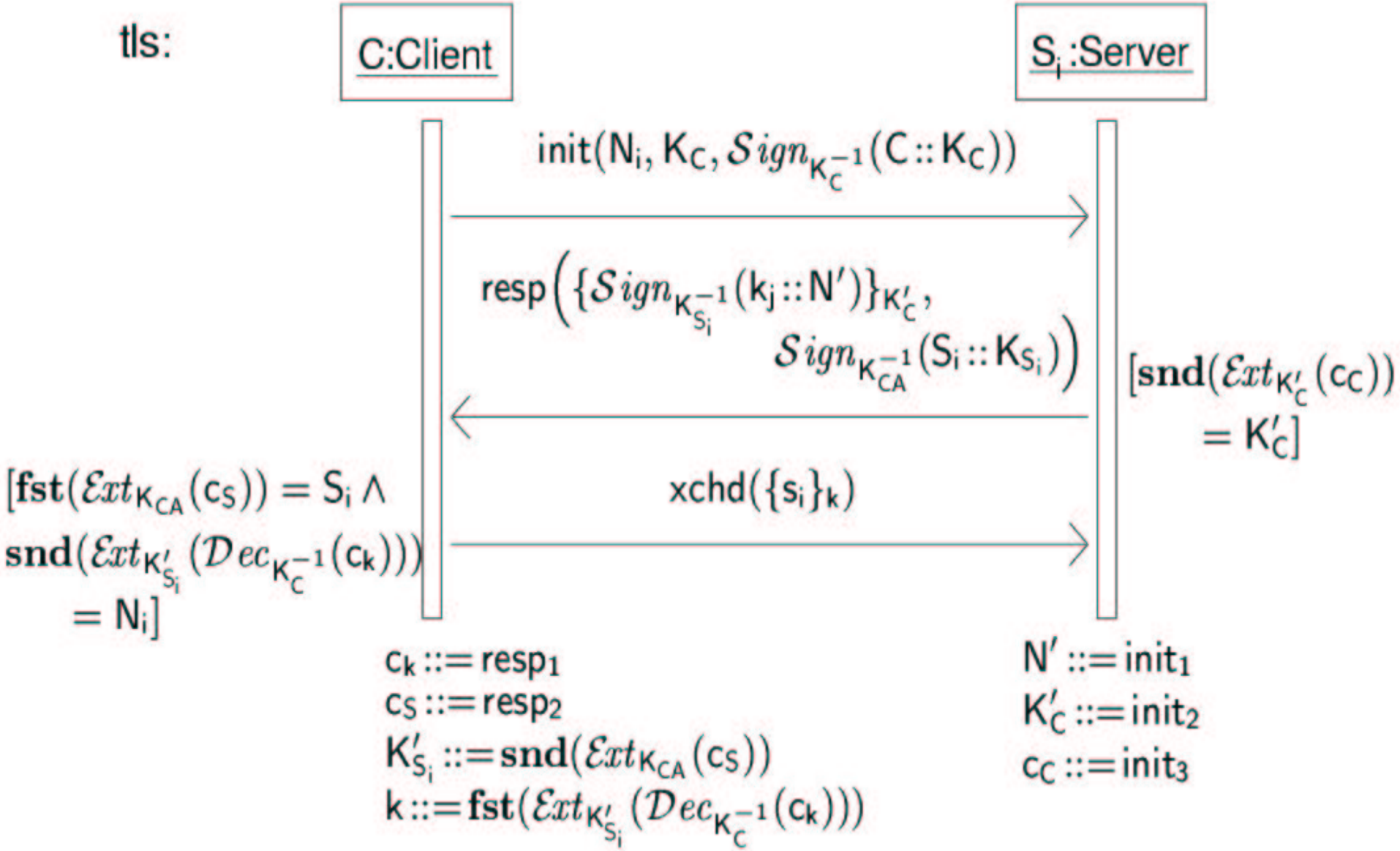
Enable efficient automated analysis by abstraction (e.g. functions or code-blocks):

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

Factor out pointers usage.

Sequence Diagram Specification

tls:



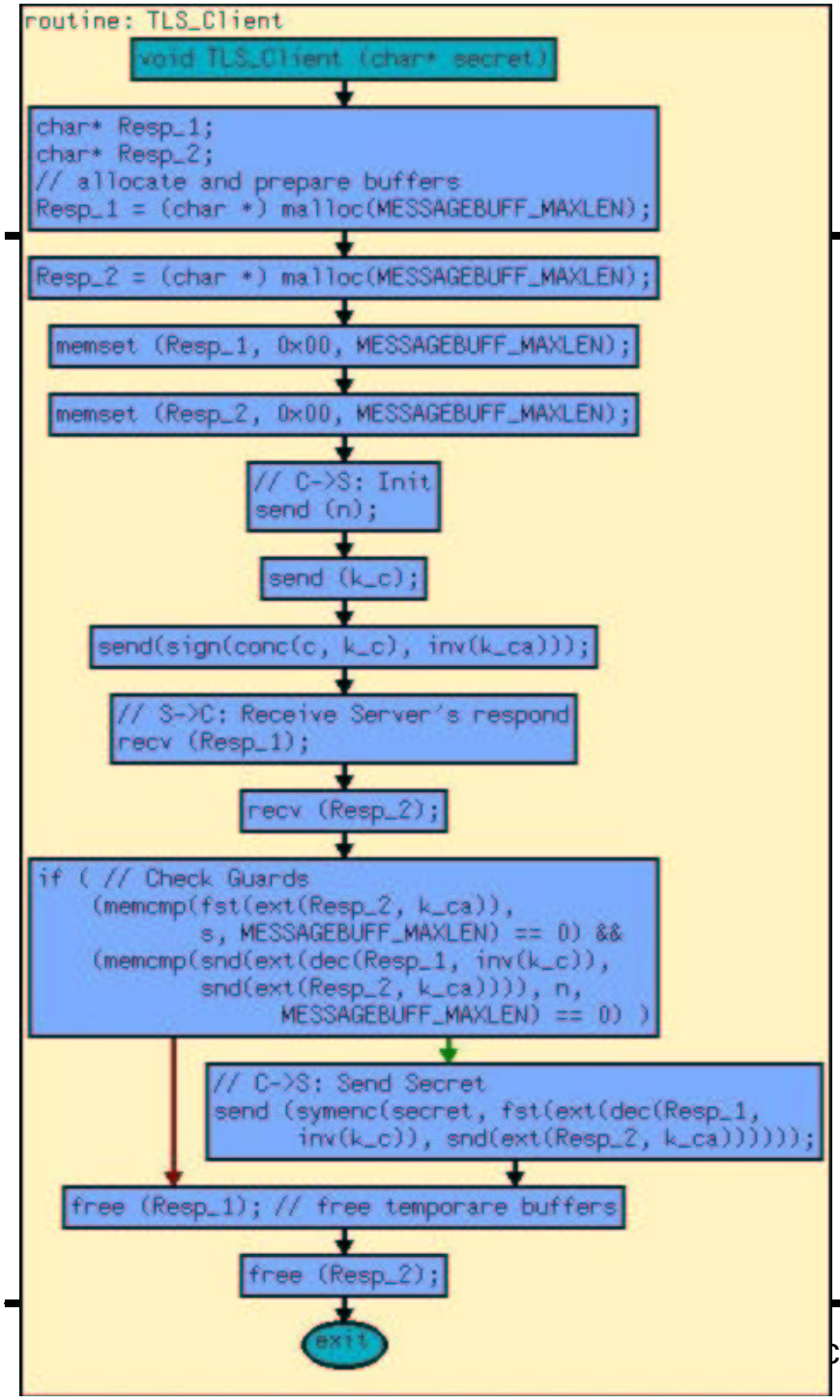
Code Annotations

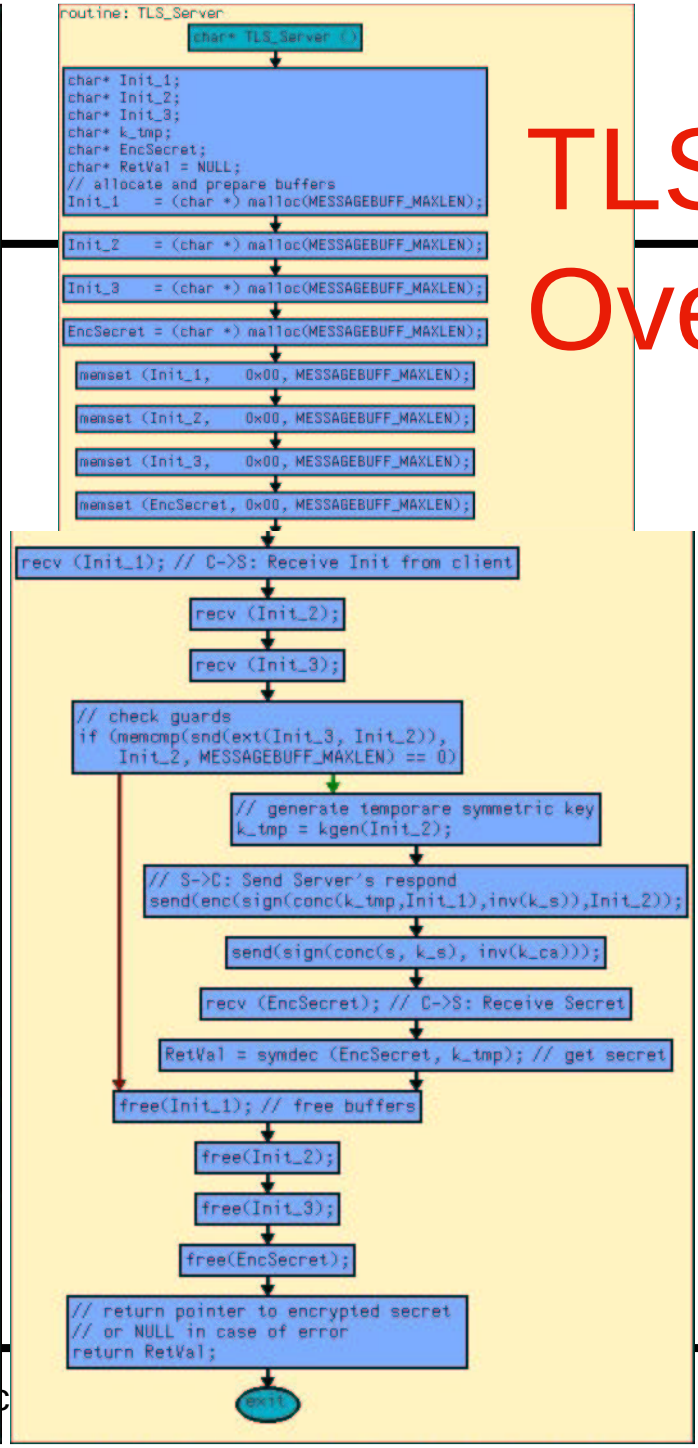
```
#ifndef _TLS_C2TPTP_H
#define _TLS_C2TPTP_H

//@C2TPTP_START
// SEND (send, 3)
// RECEIVE (rcv, 2)
// CONST const_c_cert_ca := sign(conc(c, k_c), inv(k_ca))
// CONST const_s_cert_ca := sign(conc(s, k_s), inv(k_ca))
//@C2TPTP_END

#endif // _TLS_C2TPTP_H
```

TLS Overview





Client I

routine: TLS_Client

```
void TLS_Client (char* secret)
```

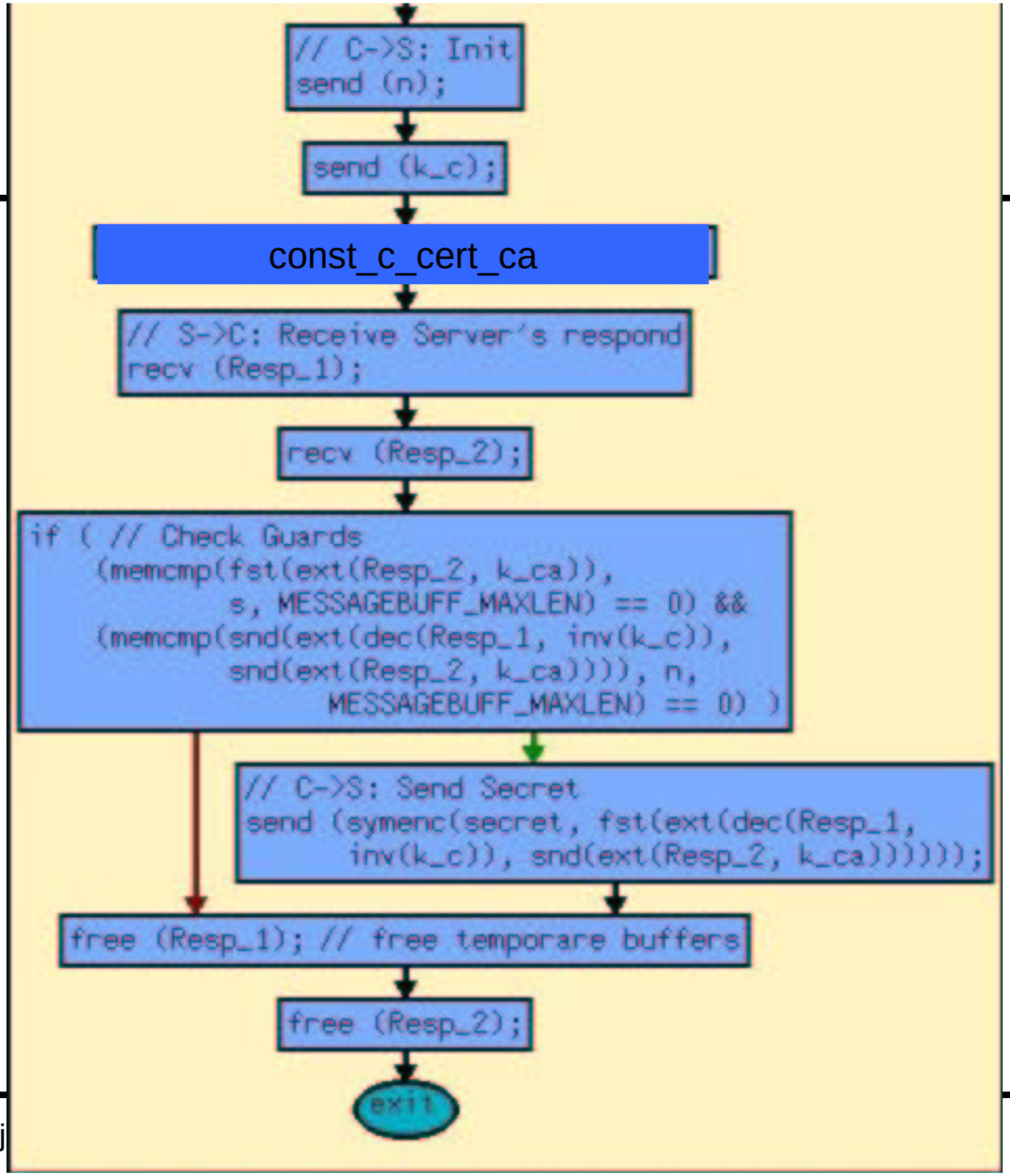
```
char* Resp_1;
char* Resp_2;
// allocate and prepare buffers
Resp_1 = (char *) malloc(MESSAGEBUFF_MAXLEN);
```

```
Resp_2 = (char *) malloc(MESSAGEBUFF_MAXLEN);
```

```
memset (Resp_1, 0x00, MESSAGEBUFF_MAXLEN);
```

```
memset (Resp_2, 0x00, MESSAGEBUFF_MAXLEN);
```

Client II


// C->S: Init
send (n);
send (k_c);
const_c_cert_ca
// S->C: Receive Server's respond
recv (Resp_1);
recv (Resp_2);
if (// Check Guards
(memcmp(fst(ext(Resp_2, k_ca)),
s, MESSAGEBUFF_MAXLEN) == 0) &&
(memcmp(snd(ext(dec(Resp_1, inv(k_c)),
snd(ext(Resp_2, k_ca)))), n,
MESSAGEBUFF_MAXLEN) == 0))
// C->S: Send Secret
send (symenc(secret, fst(ext(dec(Resp_1,
inv(k_c)), snd(ext(Resp_2, k_ca))))));
free (Resp_1); // free temporare buffers
free (Resp_2);
exit

Server I

routine: TLS_Server

```
char* TLS_Server ()
```

```
char* Init_1;
char* Init_2;
char* Init_3;
char* k_tmp;
char* EncSecret;
char* RetVal = NULL;
// allocate and prepare buffers
Init_1    = (char *) malloc(MESSAGEBUFF_MAXLEN);
```

```
Init_2    = (char *) malloc(MESSAGEBUFF_MAXLEN);
```

```
Init_3    = (char *) malloc(MESSAGEBUFF_MAXLEN);
```

```
EncSecret = (char *) malloc(MESSAGEBUFF_MAXLEN);
```

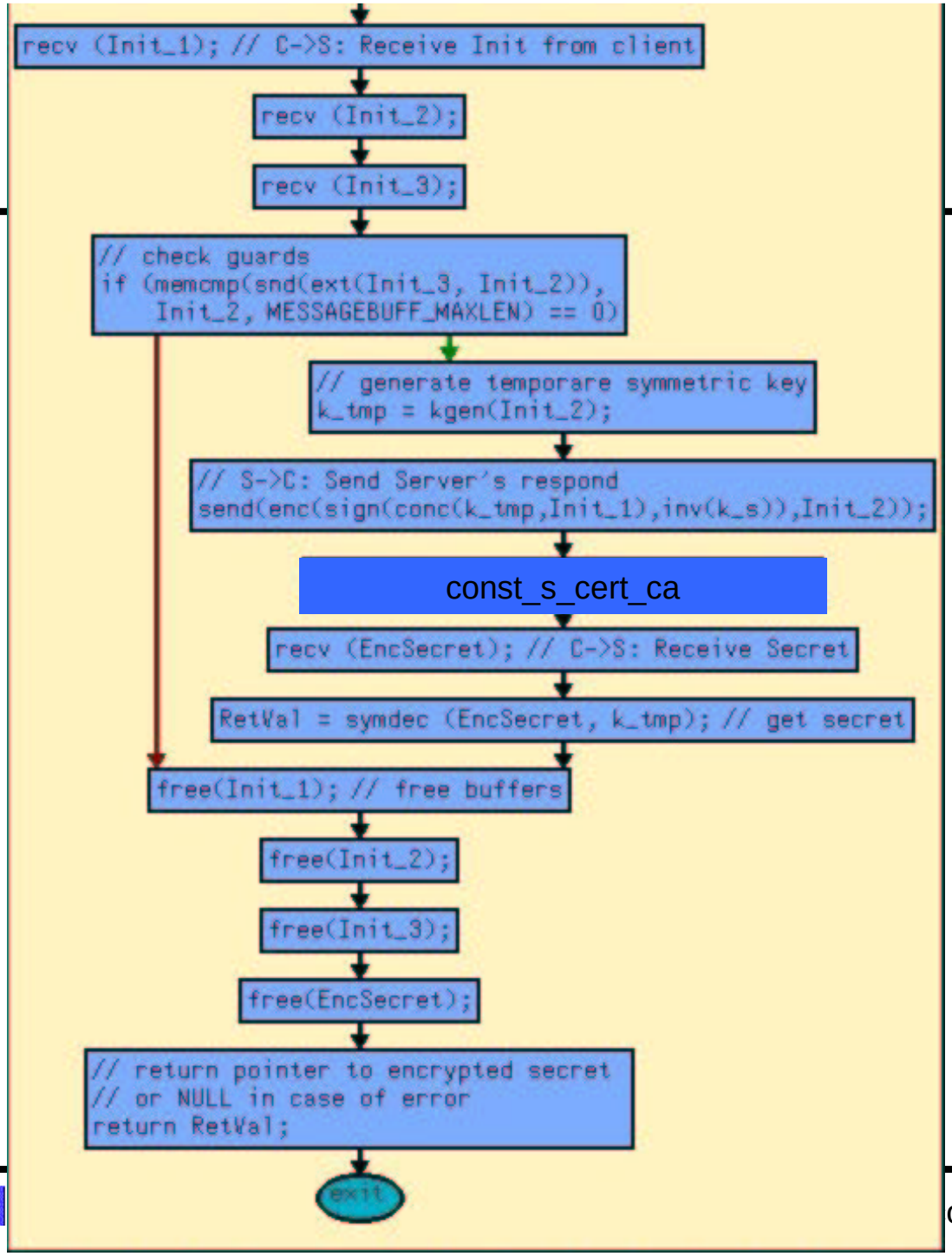
```
memset (Init_1,    0x00, MESSAGEBUFF_MAXLEN);
```

```
memset (Init_2,    0x00, MESSAGEBUFF_MAXLEN);
```

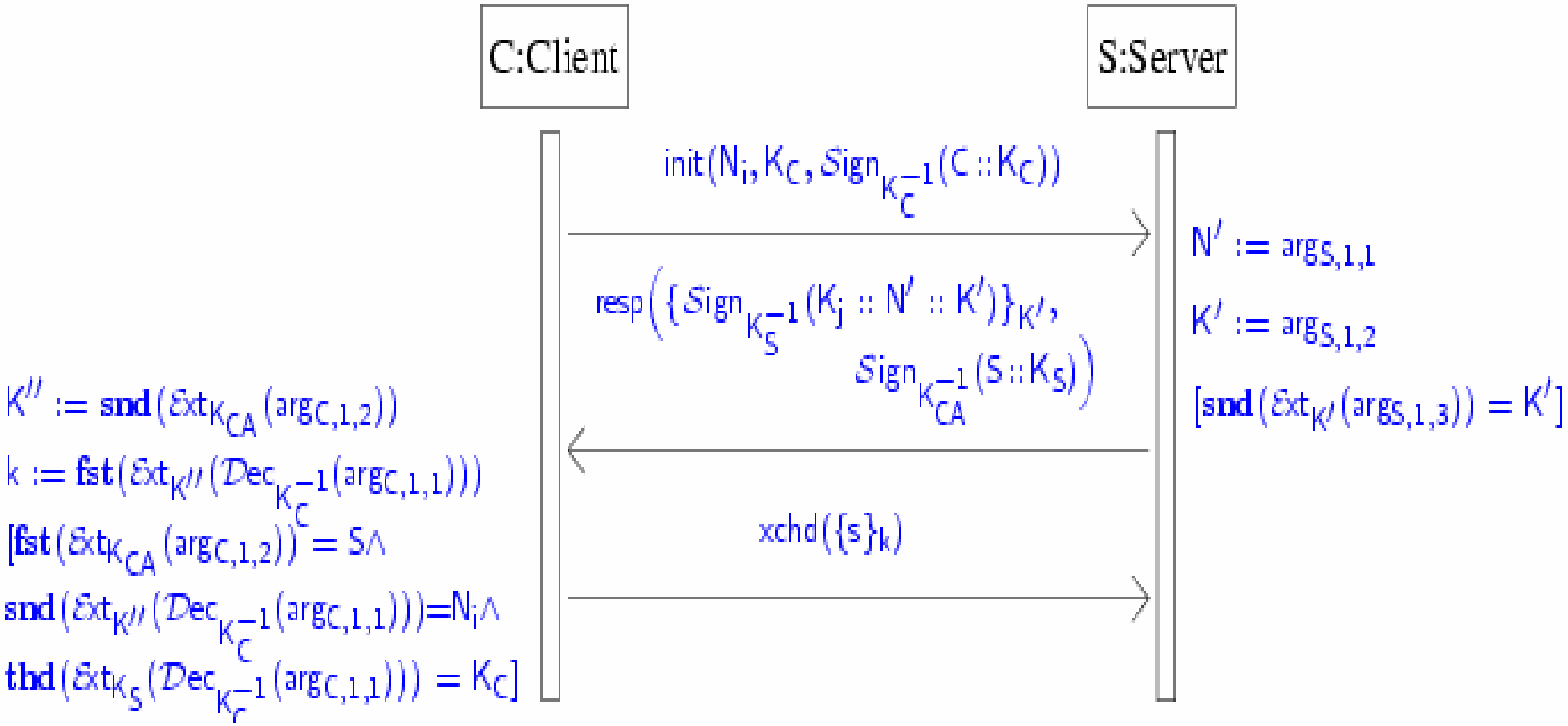
```
memset (Init_3,    0x00, MESSAGEBUFF_MAXLEN);
```

```
memset (EncSecret, 0x00, MESSAGEBUFF_MAXLEN);
```

Server II


recv (Init_1); // C->S: Receive Init from client
recv (Init_2);
recv (Init_3);
// check guards
if (memcmp(snd(ext(Init_3, Init_2)),
Init_2, MESSAGEBUFF_MAXLEN) == 0)
// generate temporare symmetric key
k_tmp = kgen(Init_2);
// S->C: Send Server's respond
send(enc(sign(conc(k_tmp,Init_1),inv(k_s)),Init_2));
const_s_cert_ca
recv (EncSecret); // C->S: Receive Secret
RetVal = symdec (EncSecret, k_tmp); // get secret
free(Init_1); // free buffers
free(Init_2);
free(Init_3);
free(EncSecret);
// return pointer to encrypted secret
// or NULL in case of error
return RetVal;
exit

The Fix



e-Setheo: *knows(s)* not derivable. Thus „secure“ in above sense.

Interpretation

Why examine knows(s) rather than secret(s) ?

➔Really only care about initial models of the axioms (basic axioms, protocol axioms and initial adversary knowledge) - our „idealized implementation“. Models basic assumptions on the security of the cryptographic algorithms (viewed at our level of abstraction): Cryptographic data does not fulfill any unintended equations (or the adversary cannot learn these).

Modular Verification: Idea

For given protocol try to establish assertions

a1 → g1 .

Include into C code as annotations.

Check whether axioms together with annotations entail threat conjecture.

Modular Verification: Example

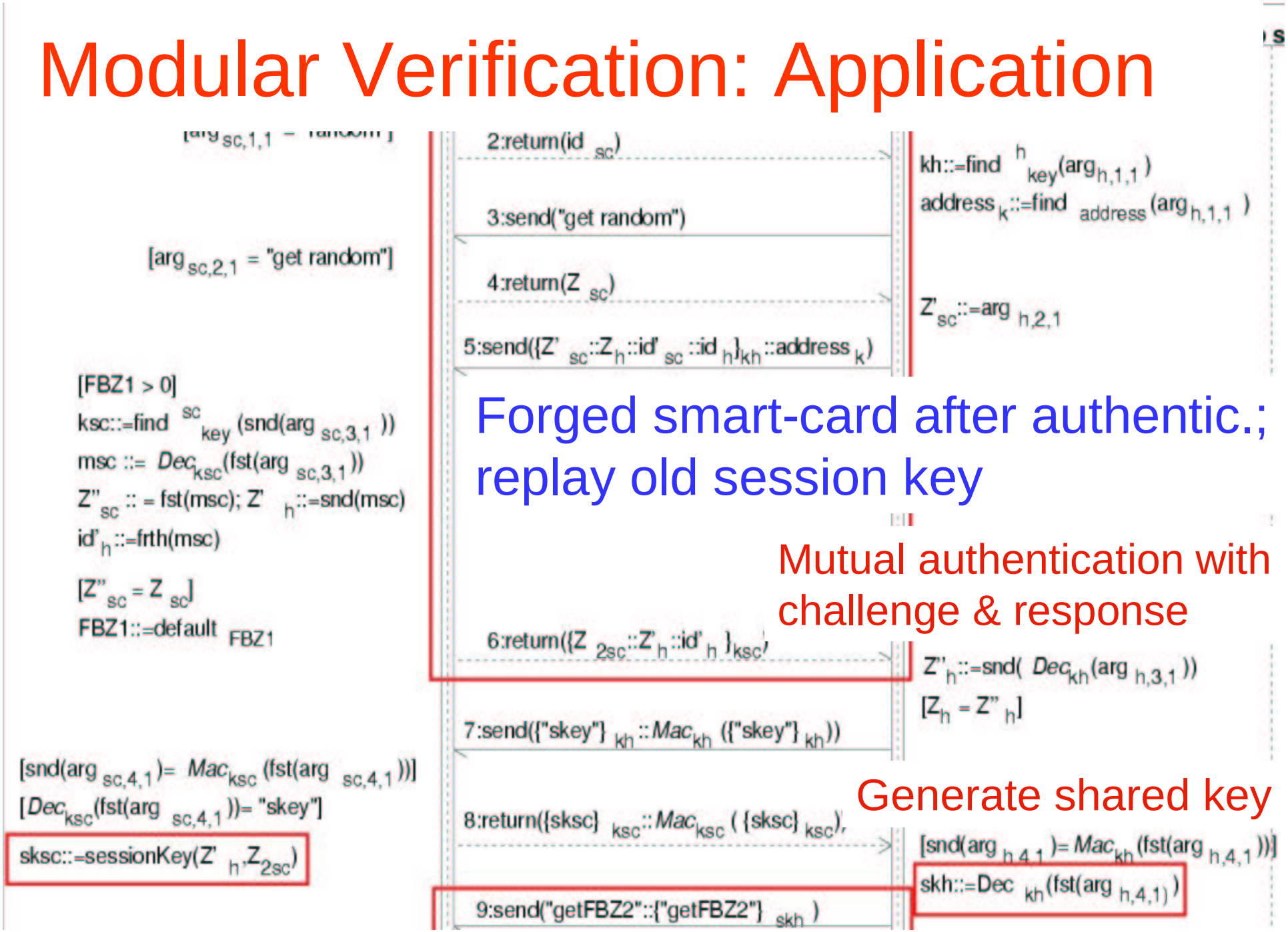
If for protocol p1, basic axioms, protocol axioms and initial adversary knowledge entails knows(s) (have counter-example) then in the initial model satisfying these axioms (our „idealized implementation“) the adversary can learn s.

We generate the assertion

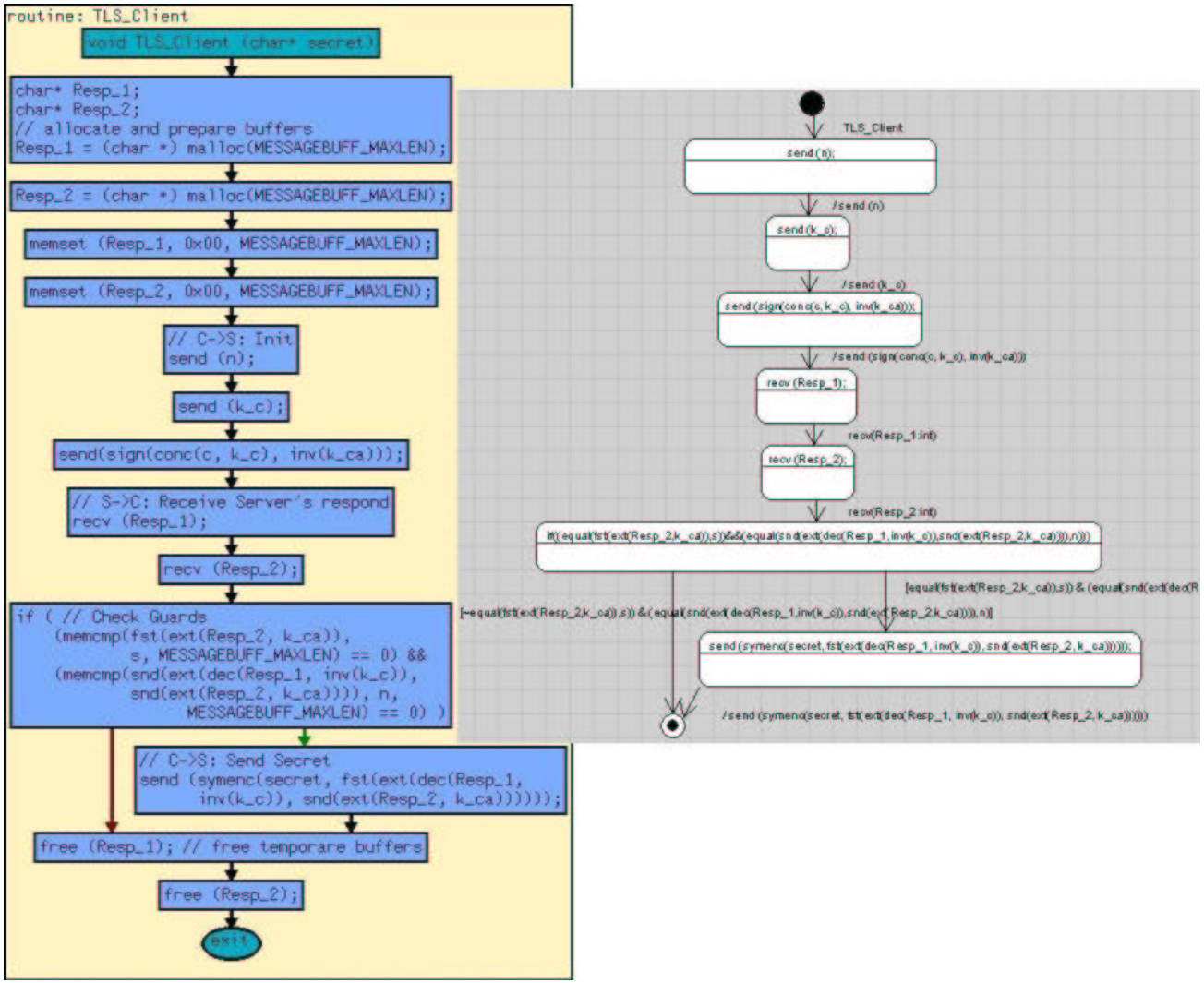
init_knowledge_axioms → knows(s)

(Careful: cannot just generate … → not(knows(s)) otherwise without possible inconsistency.)

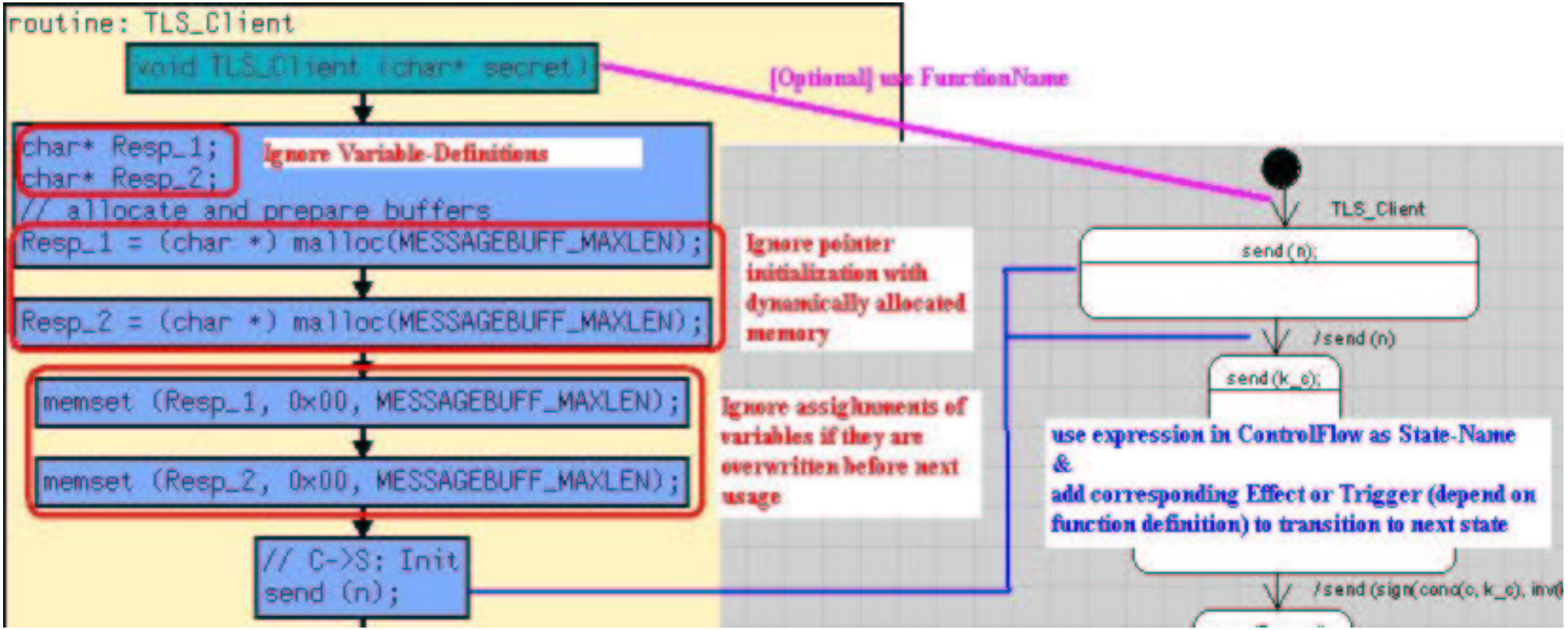
Modular Verification: Application



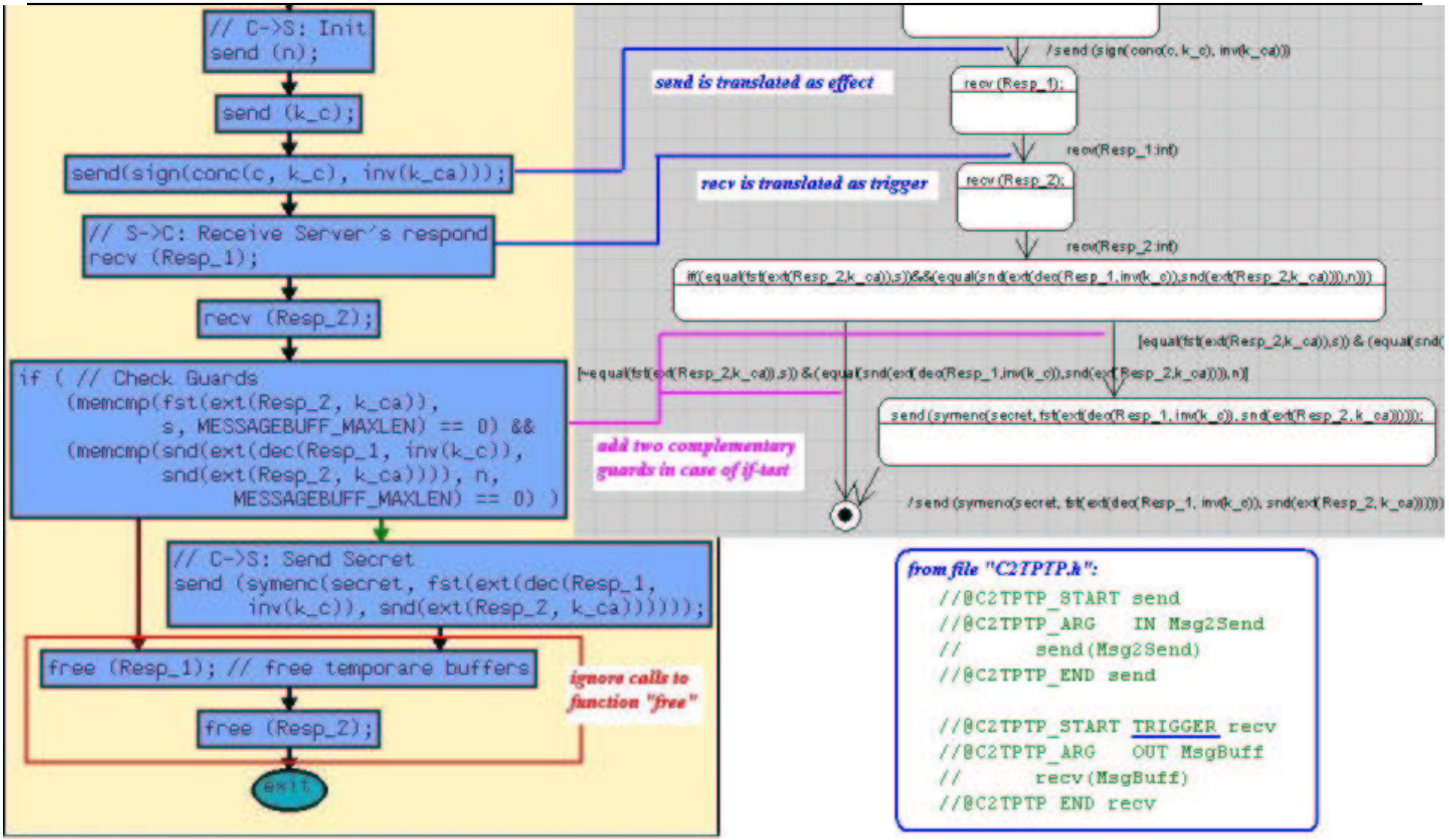
Control Flow graph to State Machine



CFG to State Machine I



CFG to State Machine II



TLS Variant in TPTP notation I

```
input_formula(tls_abstract_protocol,axiom,(
 ![ArgS_11, ArgS_12, ArgS_13, ArgC_11, ArgC_12] : (
   ![DataC_KK, DataC_k, DataC_n] : (
      % Client -> Attacker (1. message)
      (       knows(n)
           & knows(k_c)
           & knows(sign(conc(c, k_c), inv(k_c) ) )  )
    & % Server -> Attacker (2. message)
      (  (    knows(ArgS_11)
           & knows(ArgS_12)
           & knows(ArgS_13)
           & ( ? [X] : equal( sign(conc(X, ArgS_12), inv(ArgS_12) ),
                              ArgS_13 ) )  )
      => (    knows(enc(sign(conc(kgen(ArgS_12), ArgS_11), inv(k_s) ),
                        ArgS_12 ) )
           & knows(sign(conc(s, k_s), inv(k_ca) ) ) )  )
```

TLS Variant in TPTP notation II

```
  & % Client -> Attacker (3. message)
    (  (   knows(ArgC_11)
         & knows(ArgC_12)
         & equal(sign(conc(s, DataC_KK), inv(k_ca)), ArgC_12 )
         & equal(enc(sign(conc(DataC_k, DataC_n), inv(DataC_KK) ),
                     k_c), ArgC_11 )
         & ( ? [DataC_ks] : equal(sign(conc(s, DataC_ks), inv(k_ca) ),
                                  ArgC_12 ) )
         & equal(enc(sign(conc(DataC_k, n), inv(DataC_KK) ), k_c),
                 ArgC_11 )
         & equal(enc(sign(conc(DataC_k, DataC_n), inv(DataC_KK) ), k_c),
                 ArgC_11 )
       )
    => ( knows(symenc(secret, DataC_k)) )  )
) )  )).
```