Towards Verified Crypto-Protocol Implementations:

The Java Secure Sockets Extension

Jan Jürjens

Computing Department, Open University, GB

[from 1 Oct 2008 also visiting at:
Microsoft Research (Cambridge)]

http://www.jurjens.de/jan J.Jurjens@open.ac.uk

Problem

How do I know a crypto-protocol implementation (as opposed to specification) is secure ?

Possible solution:

Verify specification, write code generator, verify code generator.

Problems:

- very challenging to verify code generator
- generated code satisfactory for given requirements (maintainability, performance, size, …) ?
- not applicable to existing implementations

Alternative Solution

Verify implementation against security requirements.

So far applied to self-written or restricted code.

Surprisingly few approaches so far:

- J. Jürjens, M. Yampolski (ASE´05, ASE'06, …): methodology + initial results for restricted C code
- J. Goubault-Larrecq, F. Parrennes (VMCAI´05): self-coded client-side of Needham-Schroeder in C
- K. Bhargavan, C. Fournet, A. Gordon (CSFW´06, …): self-coded implementations in F-sharp
- Reif, Schellhorn et al (forthcoming): self-constructed code

May reduce first problem (verify code generator). How about other two (requirements on code; legacy code)?

Towards Verifying Legacy Implementations

Goal: Verify pre-existing implementation. Options:

2) Generate models from code and verify these.
   - Advantages:
     -- Seems more automatic.
     -- Users in practice can work on familiar artifact (code), don´t need to otherwise change development process (!).
   - Challenges: Currently possible for restricted code or using significant annotations. Need to verify model generator.

2) Create models and code manually and verify code against models. Advantages:
   - Split heavy verification burden (Model-level analysis more efficient).
   - Get some verification result already in design phase (for non-legacy implementations) ➔ cheaper to fix.

Just an Exercise in Code Verification ?

State of the art in code verification in practice: execution exploration by testing. Limitations:

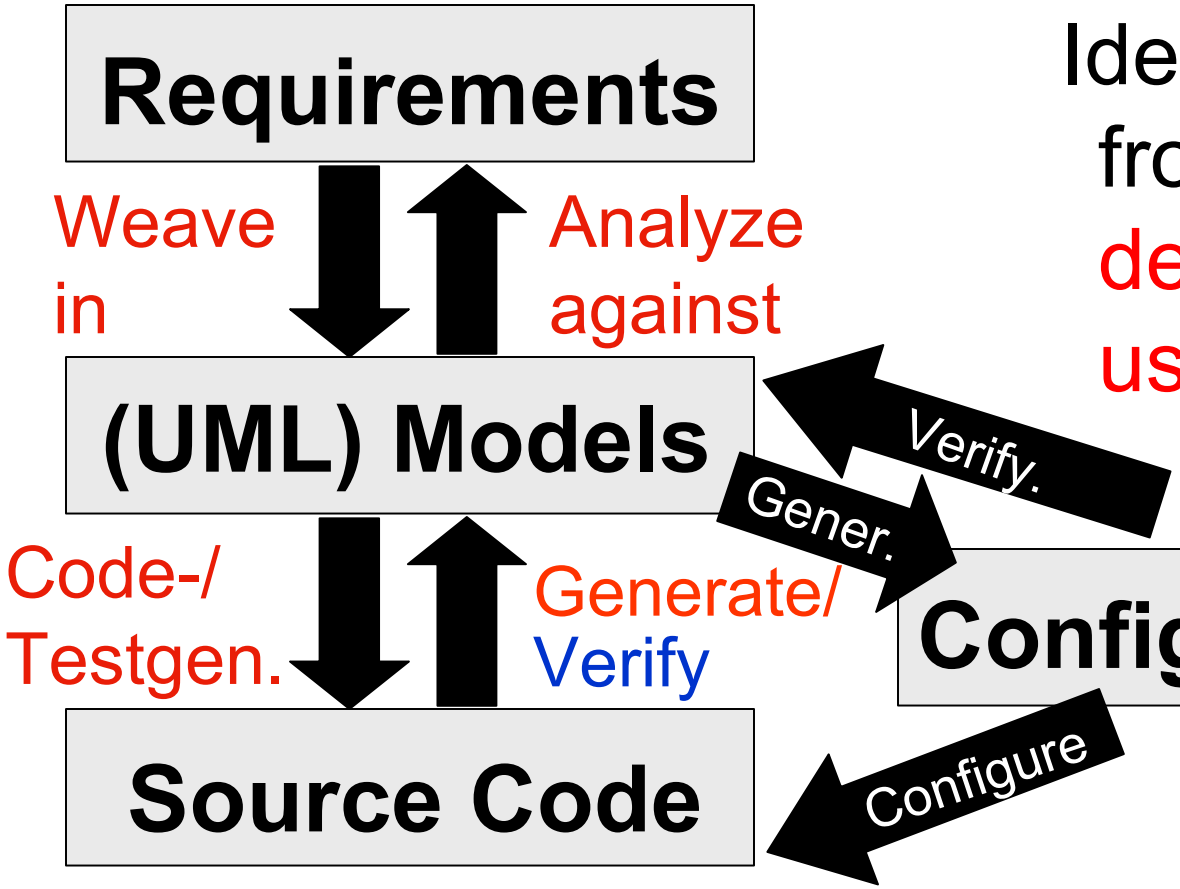
- For highly interactive systems usually only partial test coverage due to test-space explosion.
- Cryptography inherently un-testable since resilient to brute-force attack.

Interactive formal software verification (Isabelle et al): assumes specialist users.

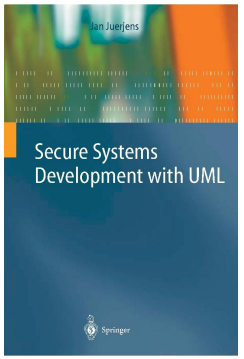
Automated … (Bandera et al.): scalability wrt. code size/ complexity; sophistication of properties (security).

➔ Develop specialized verification approach.

Context: Model-based Security Engineering



Idea: Extract models from artefacts in development and use of software.



➔ Long-term goal: Tool-supported, theoretically sound, efficient automated security design & analysis.

Security Analysis in First-order Logic

Bound on adversary knowledge set:

Predicate *knows(E)*, means adversary may get to know *E* during the execution of the system.

E.g. secrecy requirement:

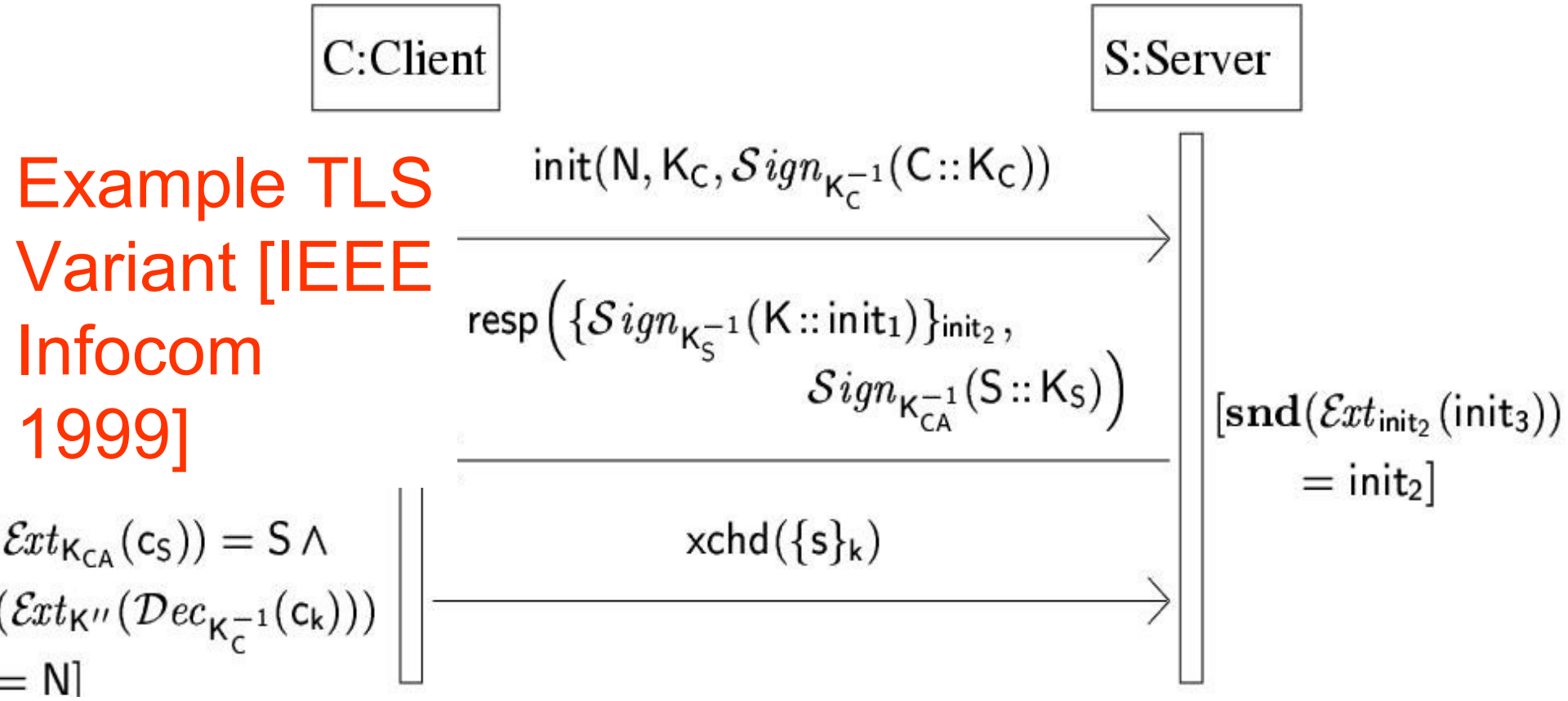
For any secret *s*, check whether can derive *knows(s)* from model-generated formulas using automated theorem prover.

Define cryptosystem. E.g.: $Dec_{K^{-1}}(\{E\}_K)=E$

Formal foundations using streams.

[ICSE05]

Example TLS Variant [IEEE Infocom 1999]



$knows(N) \wedge knows(K_C) \wedge knows(Sign_{Kc^{-1}}(C::K_C))$
$\wedge \forall init_1, init_2, init_3.[knows(init_1) \wedge knows(init_2) \wedge$
$knows(init_3) \wedge snd(Ext_{init_2}(init_3)) = init_2$
$\Rightarrow knows(\{Sign_{Ks^{-1}}(\ldots)\}_{\ldots}) \wedge [knows(Sign\ldots)]$
$\wedge \forall resp_1, resp_2.[\ldots \Rightarrow \ldots]]$

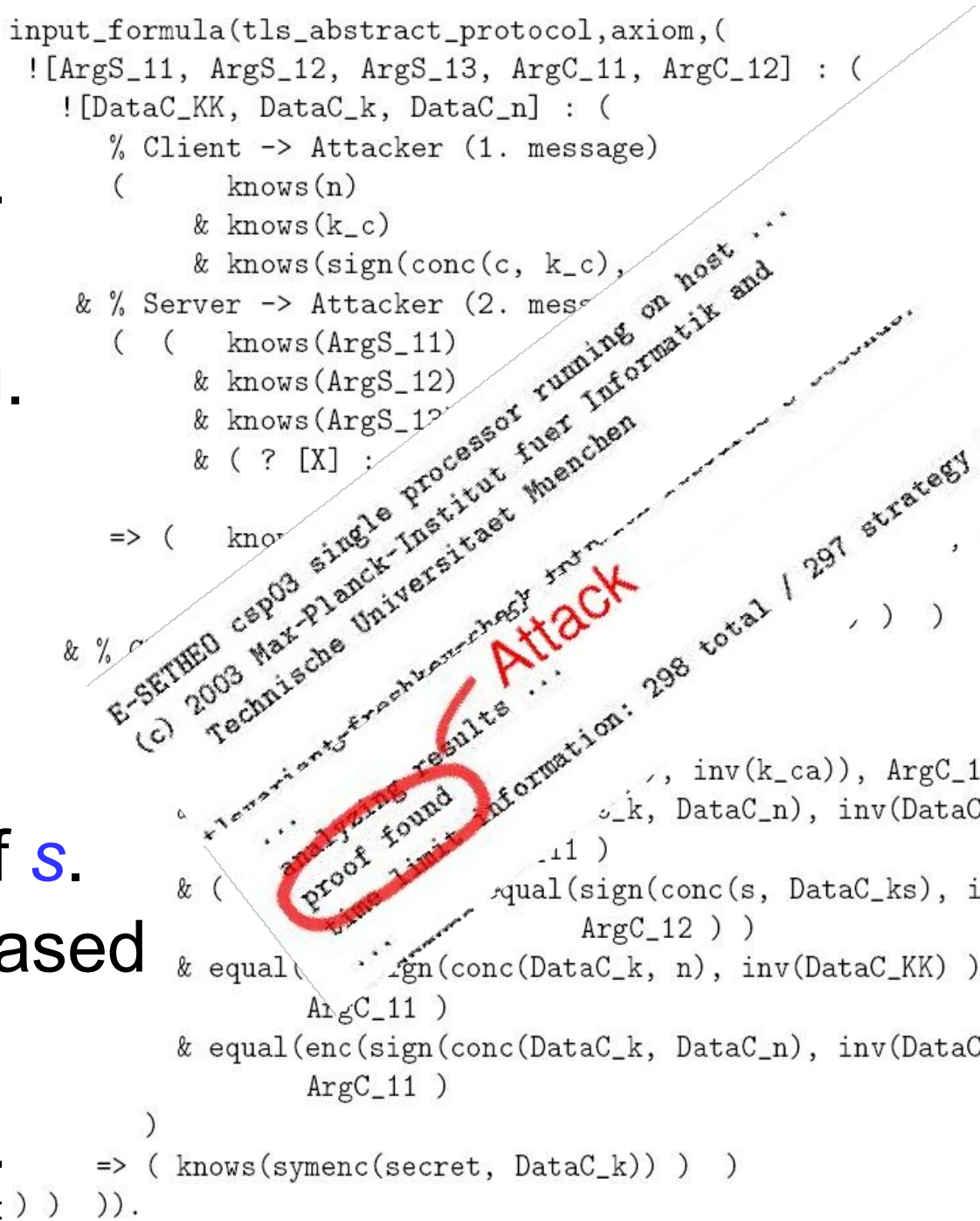
Analysis

Check whether can derive *knows(s)* e.g. using SPASS/ e-Setheo.

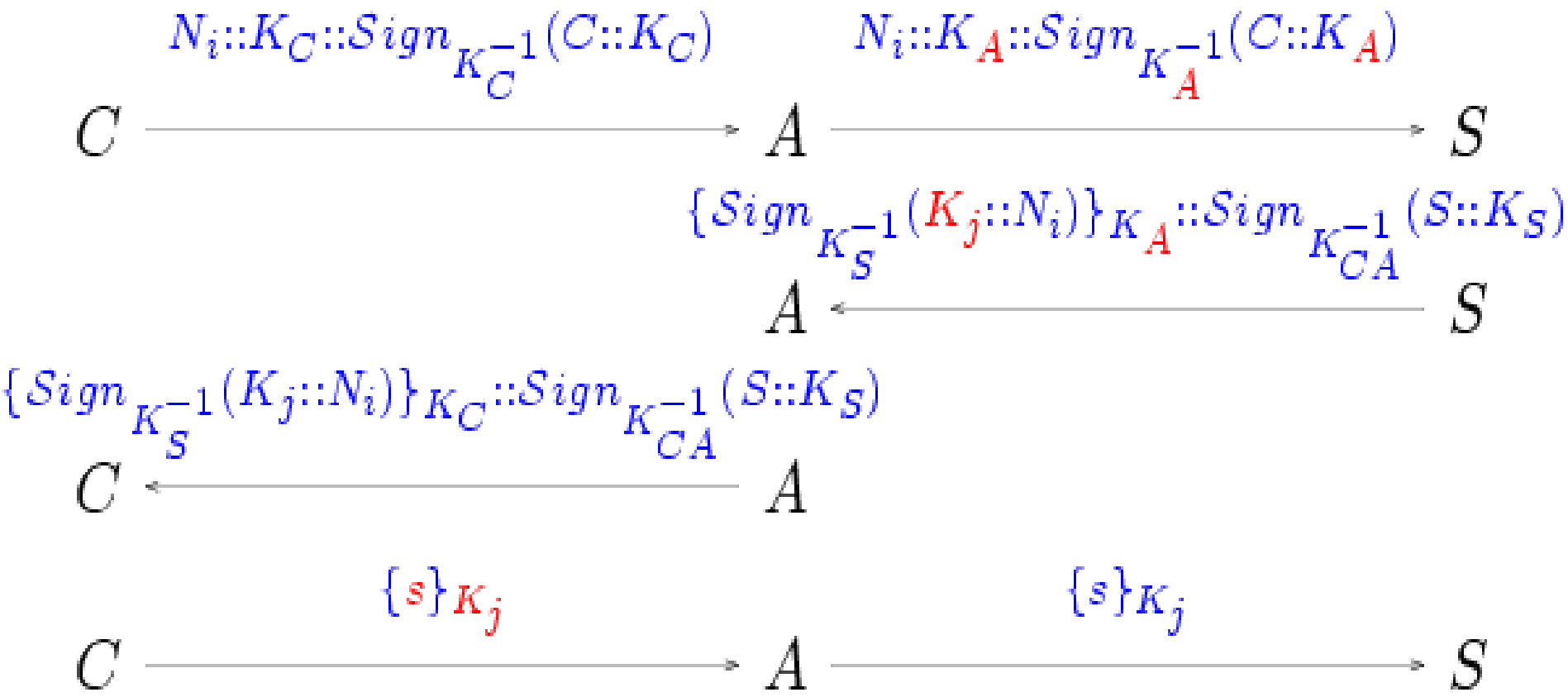
Surprise: Yes !

➔ Protocol does not preserve secrecy of *s*.

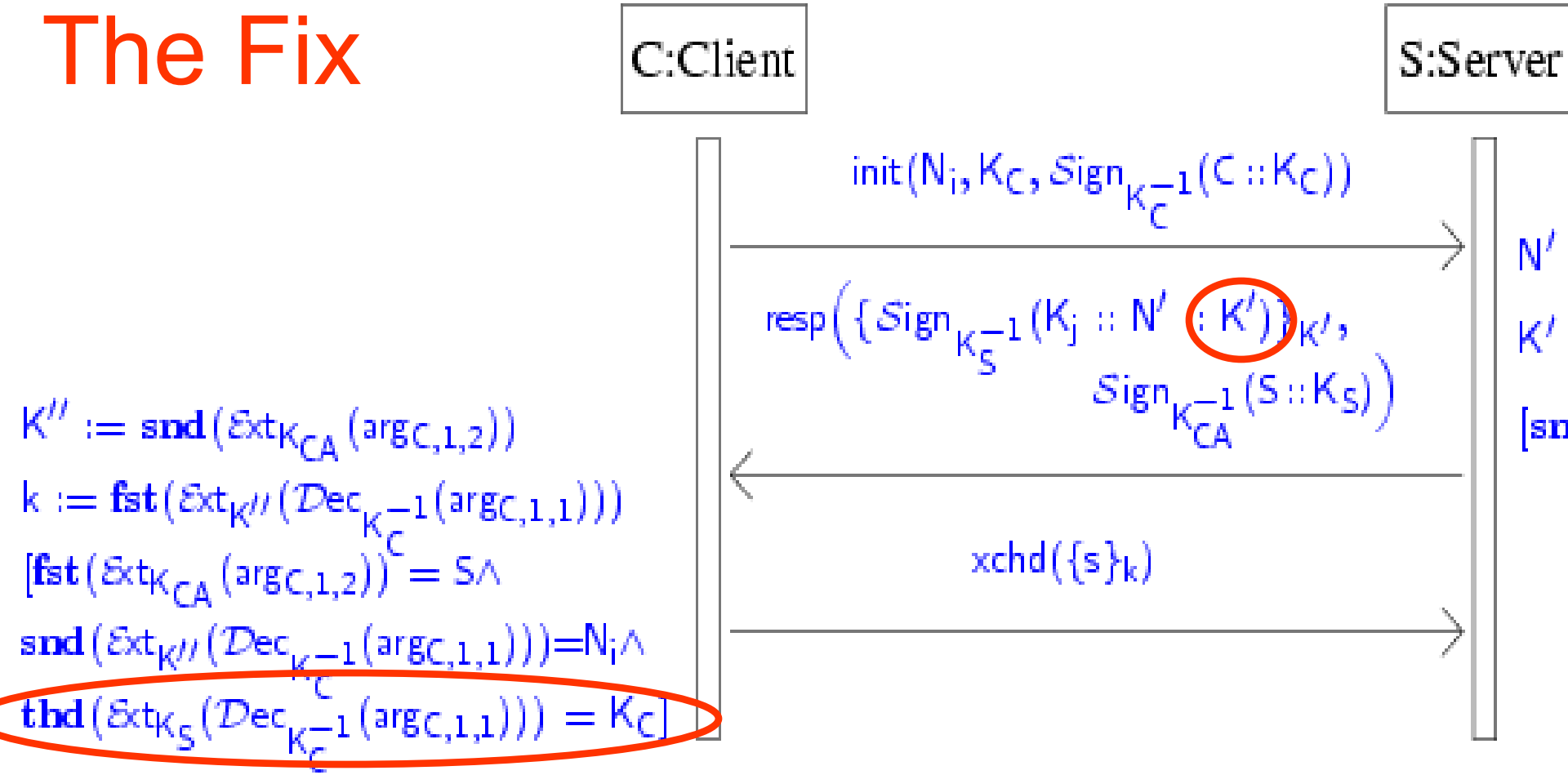
Why ? Use Prolog-based attack generator.



Man-in-the-Middle Attack



The Fix



e-Setheo: Proof that *knows(s)* not derivable.

Note completeness of FOL (but also undecidability).

Interface: Model vs. Implementation

„meaning“

„meaning“

compare meaning!

Backtrace assignments

Defined during model creation

Sent and received data

Elements of connections

Find

Has

Implement -ation

.java

Consistent?

Jessie – using RSA & Server authentication

C:Client

S:Server

create(S)

ClientHello(R_c)

create

ServerHello(R_s)

Certificate($sign_{K_{\cdot\cdot}}(S::K_s)$)

[[equal(fst($ext_{K_{\cdot\cdot}}(c_s)$),S)]]

ClientKeyExchange($enc_{K_{\cdot\cdot}}$(pms))

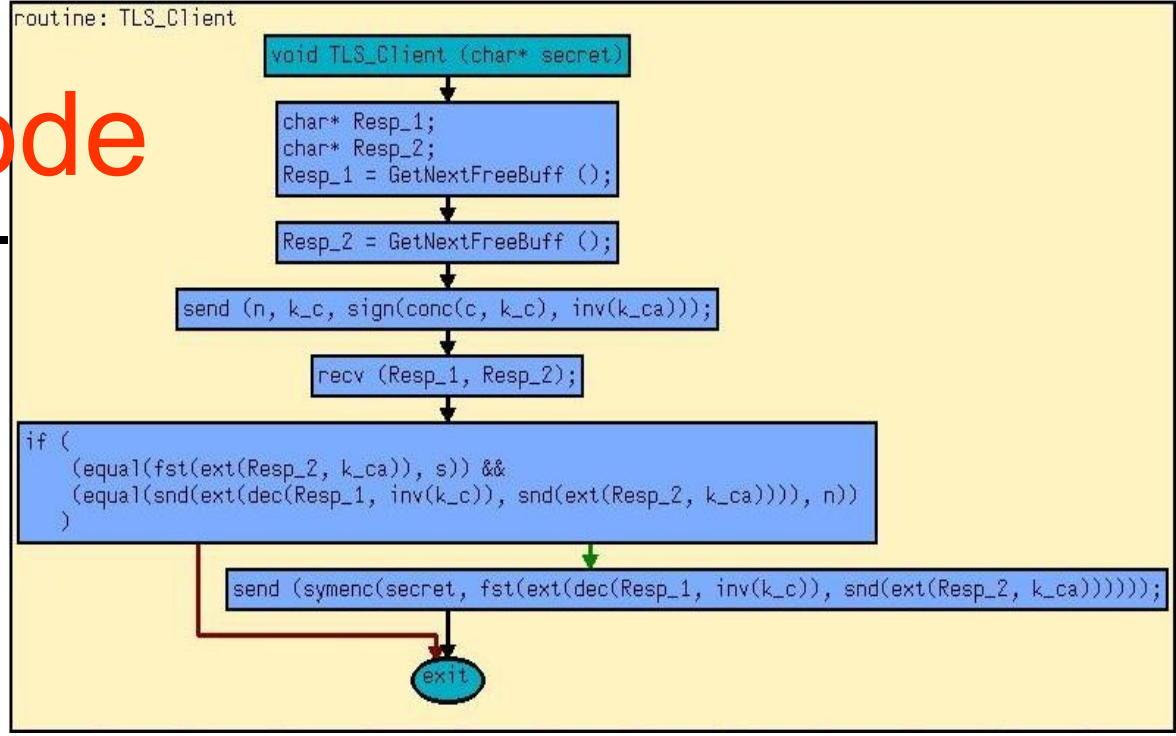
Finished($symenc_K(md5_c)$, $symenc_K(sha_c)$)

Finished($symenc_K(md5_s)$, $symenc_K(sha_s)$)

[$symdec_K(md5_s') = md5_c \wedge$ $symdec_K(sha_s') = sha_c$]

Abstract model

Models vs Code



Generate control flow graph (e.g. CodeLogic).

Transform to labelled transition state machine:

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

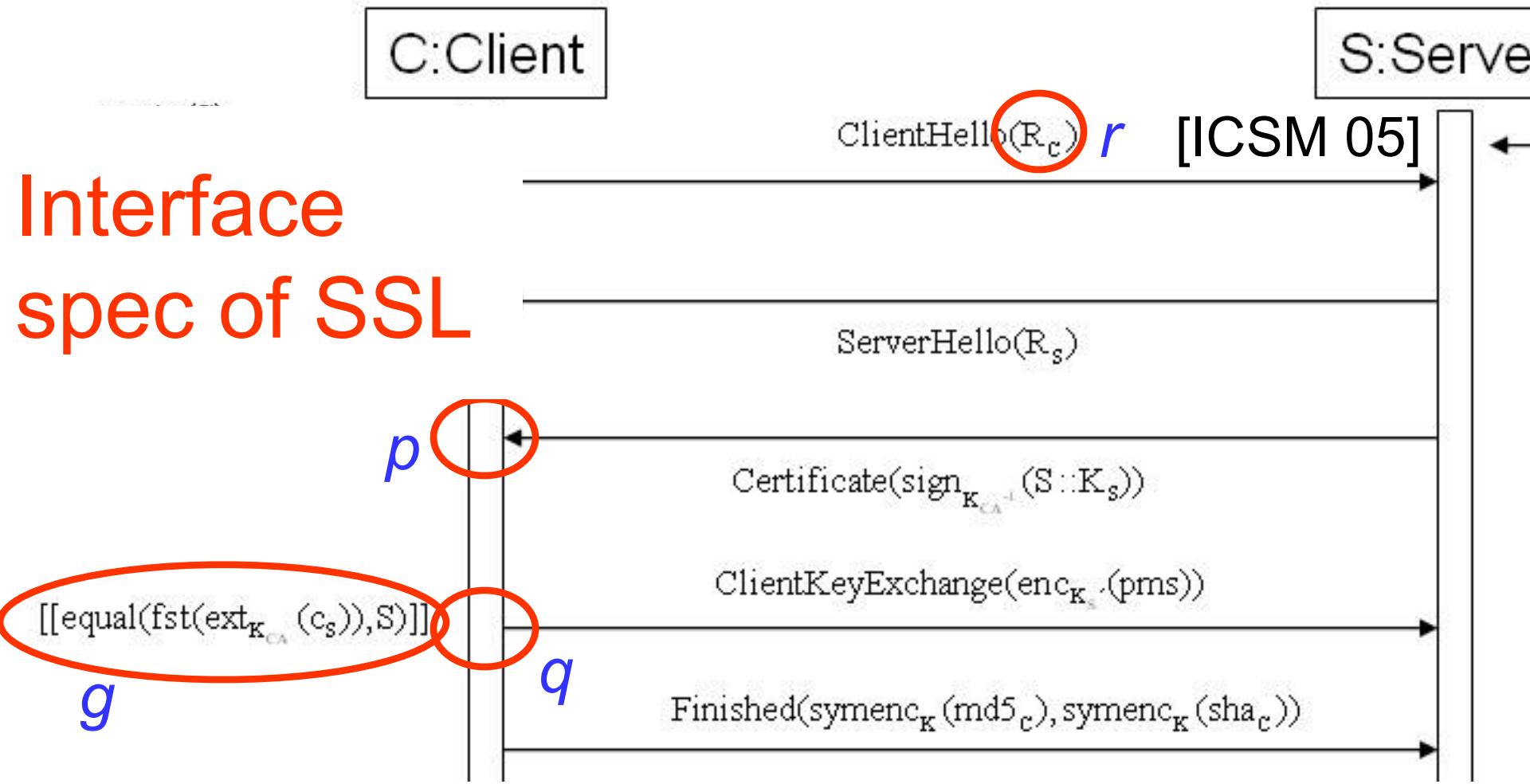
where action can be outpattern or localvar:=value.

Need to link concrete data to abstract symbols.

JSSE / Jessie

- Java Secure Sockets Extension (JSSE) contains implementation of SSL.
- Open-source clean-room reimplementation Jessie.
- Applied our approach to fragment of Jessie (SSL handshake using RSA, verifying secrecy of exchanged secret).
- Currently extending the work to JSSE recently made open-source by Sun.

Interface spec of SSL



I) Identify program points:
value (*r*), receive (*p*), guard (*g*), send (*q*)

II) Check guards enforced

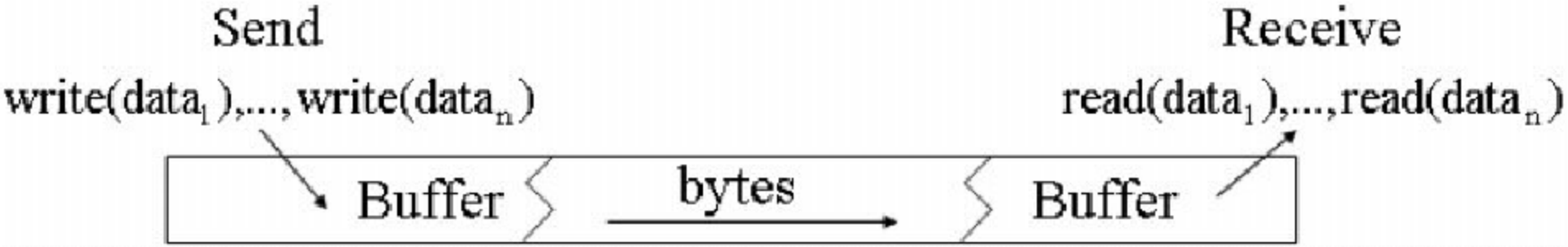
in Model	Send: ClientHello	by Outputstream.write in
	type.getValue()	Handshake.write
	(bout.size() >>> 16 & 0xFF)	Handshake.write
	(bout.size() >>> 8 & 0xFF)	Handshake.write
	(bout.size() & 0xFF)	Handshake.write
Pver	major	ProtocolVersion.write
	minor	ProtocolVersion.write
	((gmtUnixTime >>> 24) & 0xFF)	Random.write
	((gmtUnixTime >>> 16) & 0xFF)	Random.write
	((gmtUnixTime >>> 8) & 0xFF)	Random.write
	(gmtUnixTime & 0xFF)	Random.write
R_C	randomBytes	ClientHello.write
	sessionId.length	ClientHello.write
Sid	sessionId	ClientHello.write
	((suites.size() << 1) >>> 8 & 0xFF)	ClientHello.write
	((suites.size() << 1) & 0xFF)	ClientHello.write
Ciph[]	id[]	CipherSuite.write
	comp.size()	ClientHello.write
Comp[]	comp[2]	ClientHello.write

Data vs. Symbols (SSL project Jessie)

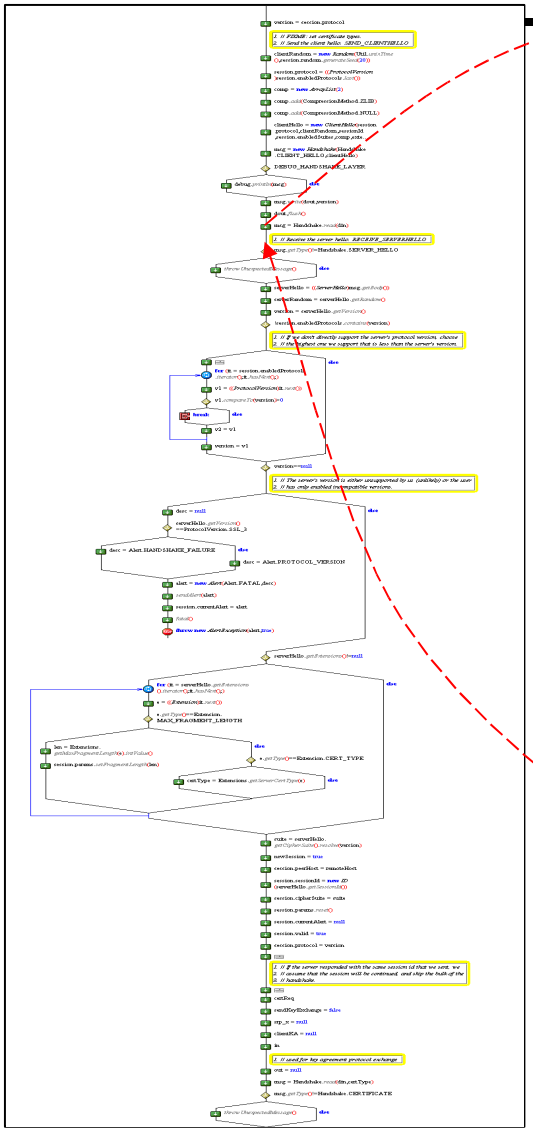
Input / Output

To extract input/output labels for state machine transitions, analyze input / output mechanism used in the implementation.

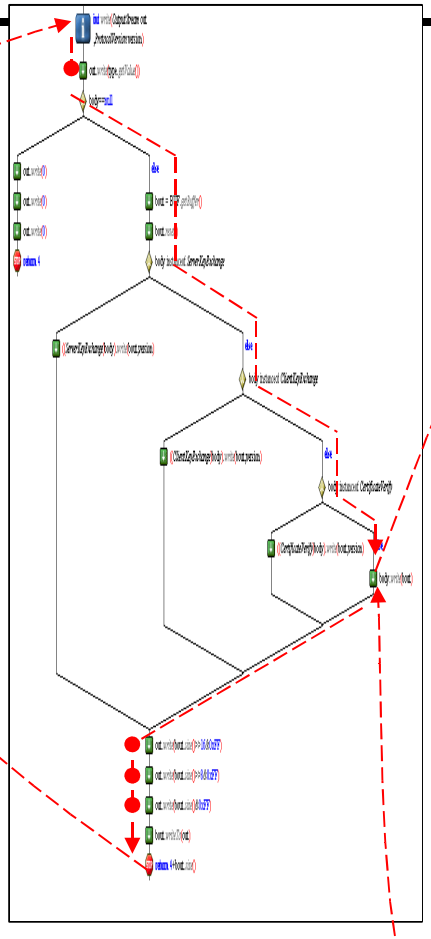
Many implementations (e.g. Jessie and JSSE) use buffered communication where the message objects implement read and write methods. Translate these method calls to input / output labels (need to track successive subcalls).



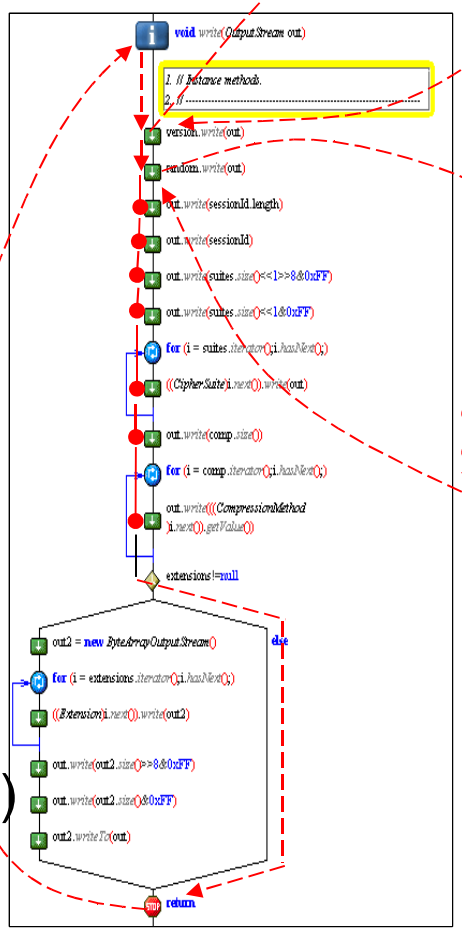
Sending Messages



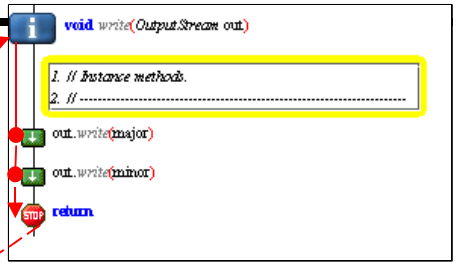
SSLSocket.doClientHandshake()



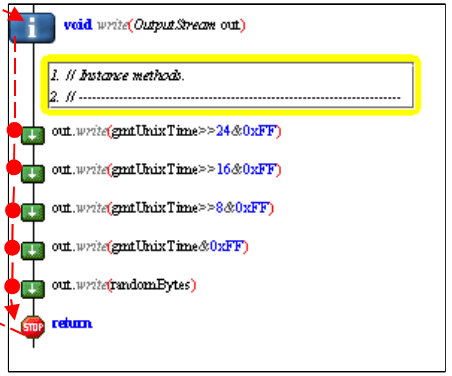
Handshake.write()



ClientHello.write()



ProtocolVersion.write()

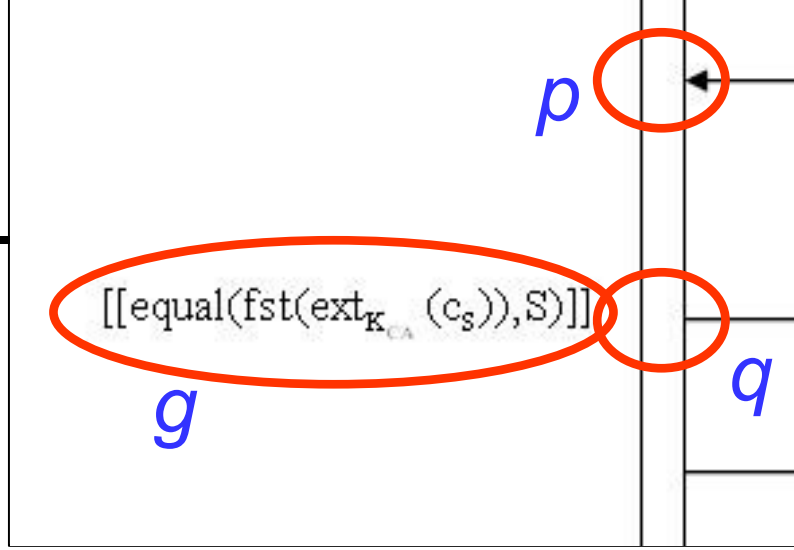


Random.write()

traverse CFG

call of OutputStream.write()

Checking Guards



[BCS-Visions08]

Guard *g* enforced by code?

b) Generate runtime check for *g* at *q* from diagram:

+ simple, effective.

-- performance, code change

c) Testing against checks (symbolic crypto for inequalities). [ICFEM02]

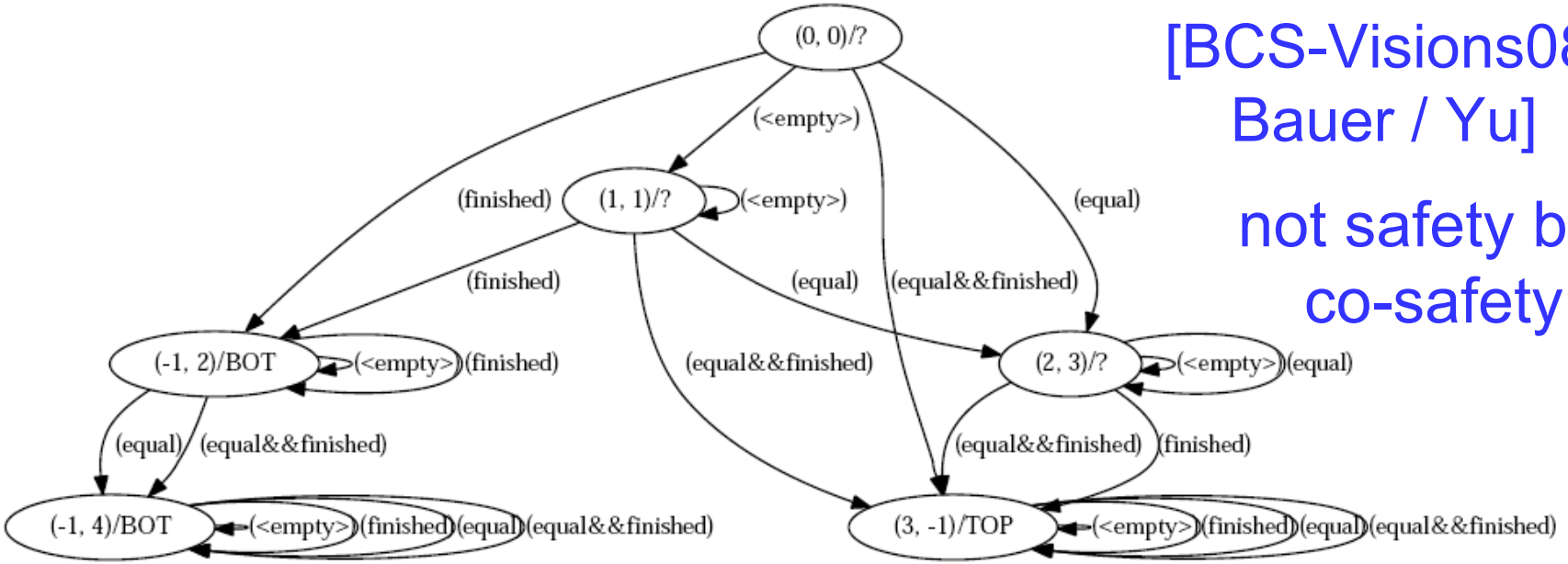
d) Automated formal local verification: conditionals between *p* and *q* logically imply *g* (using ATP for FOL). [ASE06]

Run-time Verification using LTL

Example: Client sends out ClientKeyExchange message eventually but only after received Certificate message, and check is positive.

Improves on Schneider's security automata.

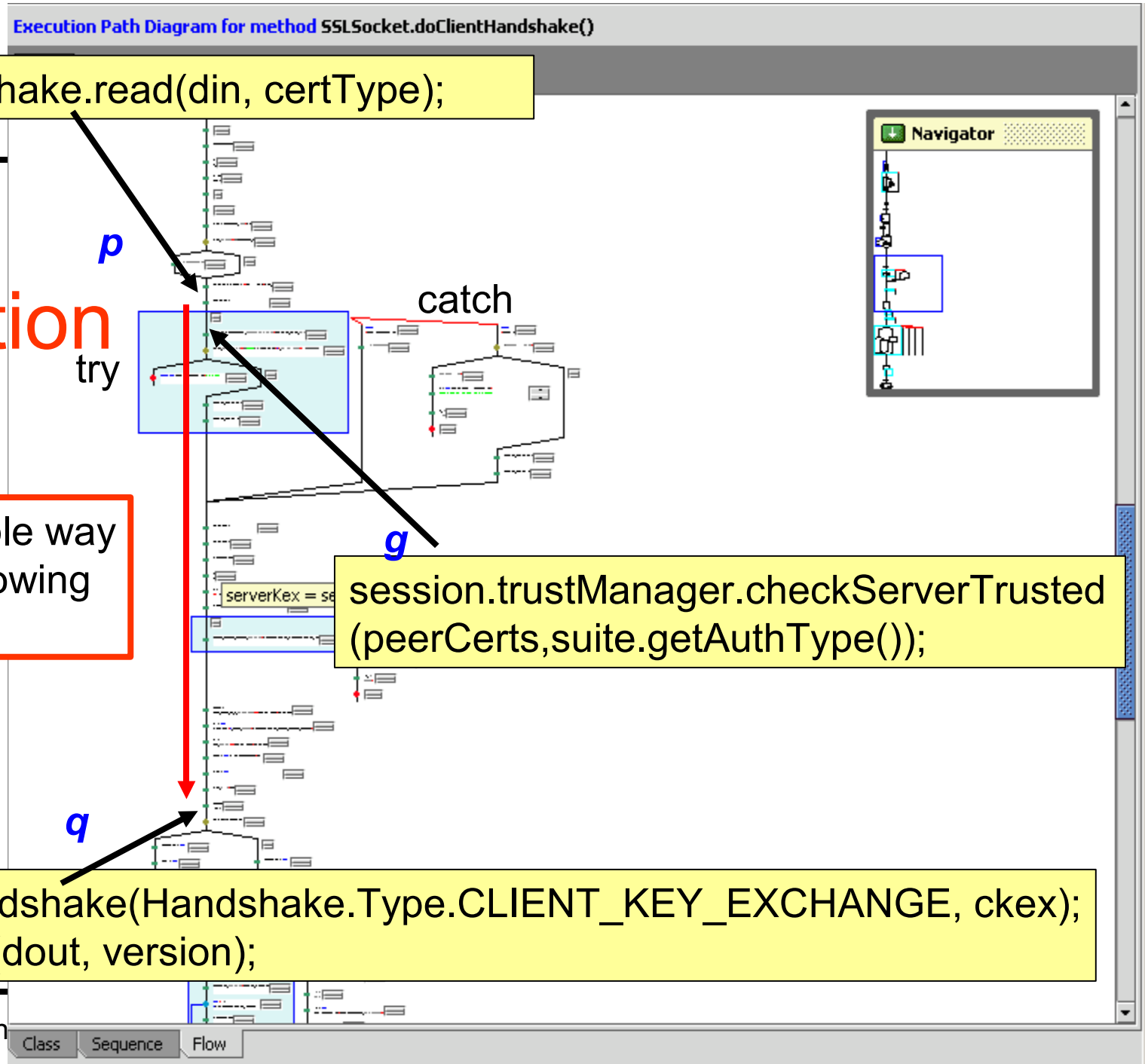
$$\varphi_2 = (\neg \text{finished } \mathbf{W} \text{ equal} \wedge (\mathbf{F} \text{ equal} \Rightarrow \mathbf{F} \text{ finished}))$$



[BCS-Visions08 w. Bauer / Yu]

not safety but co-safety

Static Verification



Verification of Guards in Code

send: represents send command

g: FOL formula with symbols msg_n representing n^{th} argument of message received before program fragment *p* is executed

[*d*] *p* ⊨*g* : *g* checked in any execution of *p* initially satisfying *d* before any send

write *p* ⊨*g* for [true] *p* ⊨*g*.

$$\frac{}{\text{[d] if } c \text{ then } p \text{ else } q \models g}\,(c \wedge d \Rightarrow g,\ \text{no send in } q)$$

Some Rules (Simplified)

$$\frac{}{[\mathsf{d}]\ \mathsf{if}\ c\ \mathsf{then}\ p\ \mathsf{else}\ q \models g}(c \wedge d \Rightarrow g,\ \mathsf{no\ send\ in}\ q)$$

$$\frac{}{[\mathsf{d}]\ \mathsf{if}\ c\ \mathsf{then}\ p\ \mathsf{else}\ q \models g}(\neg c \wedge d \Rightarrow g,\ \mathsf{no\ send\ in}\ p)$$

$$\frac{[d]p \models g}{[\mathsf{d}]\ \mathsf{if}\ c\ \mathsf{then}\ p\ \mathsf{else}\ q \models g}(d \Rightarrow c) \qquad \frac{[d]p \models g}{[d]p;q \models g}$$

$$\frac{[d]q \models g}{[\mathsf{d}]\ \mathsf{if}\ c\ \mathsf{then}\ p\ \mathsf{else}\ q \models g}(d \Rightarrow \neg c) \qquad \frac{[d]p \models g}{[d']p \models g} d' \Rightarrow d$$

$$\frac{[d]p \models g}{x := e; p \models g} d \Rightarrow x = e$$

Loops

In automated verification, often only consider finite number of iterations.

Here: in translation to logic, replace variables in loops by infinite arrays (index: loop counter).

Note: using ATP, don‘t need to worry about finding loop invariants.

General problem undecidable, but at our level of abstraction for crypto-protocols not a problem since emphasis on interaction rather than computation.

Loops: Example

```
Example:
while (true)
{ k = a + 1;
  a = b + k;
  b = b + 1; }
```

```
SSA:
while (true)
{ k = a0 + 1;
  a1 = b0 + k;
  b1 = b0 + 1; }
```

```
TPTP:
input_formula(ForLoop_axiom_ID1,axiom,(
![I]: (equal (k[I], sum(a0[I],1)) &
       equal (a1[I], sum(b0[I],k[I])) &
       equal (b1[I], sum(b0[I],1)) &
       equal (a0[succ(I)],a1[I]) &
       equal (b0[succ(I)],b1[I])))).
```

Concurrent threads

Identify maximal transition paths in CFG between points where shared variables written or read.

In translation to logic, consider possible interleavings of threads by defining:

ϕ from predicates PRED(Pi) as above (for each path i)

ψ assigning variables according to given interleaving

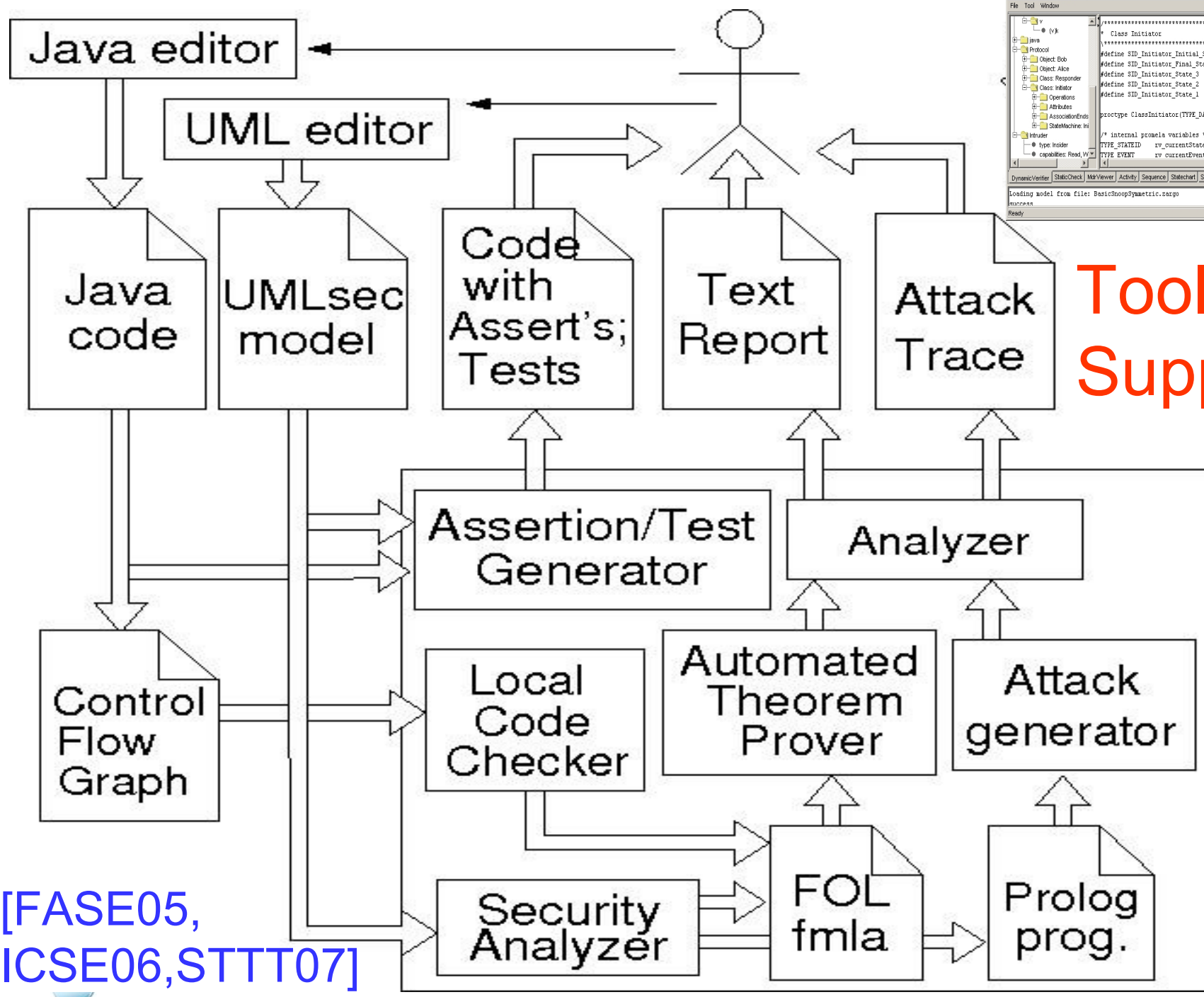
Join formulas $\psi \Rightarrow \phi$ together by conjunction.

Modular Verification

For program fragment p, generate set of statements derive(L,C,E) such that adversary knowledge is contained in every set K that:

- for every list l of values for the variables in L that satisfy the conditions in C contains the value constructed by instantiating the variables in the expression E with the values from l

When considering single protocol run, can construct finite set of such statements similar to FOL formulas from security analysis.



Tool Support

[FASE05, ICSE06,STTT07]

Other Applications

Analyzed designs / implementations / configurations e.g. for

- Biometry- or smart-card-based identification
- authentication (crypto protocols)
- authorization (user permissions, e.g. SAP systems)

Analyzed security policies, e.g. for privacy regulations.

Conclusion

Seemingly first attempt at formally based security verification for crypto-based Java legacy implementations.

Goals: Emphasis on automation, achieve efficiency using abstraction tailored to verification problem.

Experiences so far encouraging but still many challenges – collaboration welcome !

Future work: collaboration with Andy Gordon (MSRC) on verifying cryptoprotocol implementations in C.

Questions ?

More information (papers, slides, tool etc.):

http://www.jurjens.de/jan

J.Jurjens@open.ac.uk