Linking Cryptoprotocol Verification to Reality

Jan Jürjens

Computing Department
The Open University, GB

J.Jurjens@open.ac.uk

http://www.jurjens.de/jan

Crypto-Protocol Analysis

State of the affairs:

A *lot* of very successful work in formally verifying abstract models of crypto-protocol design.

- virtually every formal method has been applied
- seemingly more people working on verification than on designing protocols
- efficient tool-support usable by academics or specialists
- sometimes used at industrial size protocols (usually by tool developers themselves)

(Almost) solves the problem whether design is secure.

Problem

How do I know a crypto-protocol implementation is secure ?

Possible solution:

Verify design model, 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 verified design or directly against security requirements.

So far applied to self-written or restricted code.

Surprisingly few approaches so far:

- J. Jürjens, M. Yampolski (ASE´05): 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

May reduce first problem. How about other two ?

Towards Verifying Legacy Implementations

Goal: Verify implementation created independently.

Options:

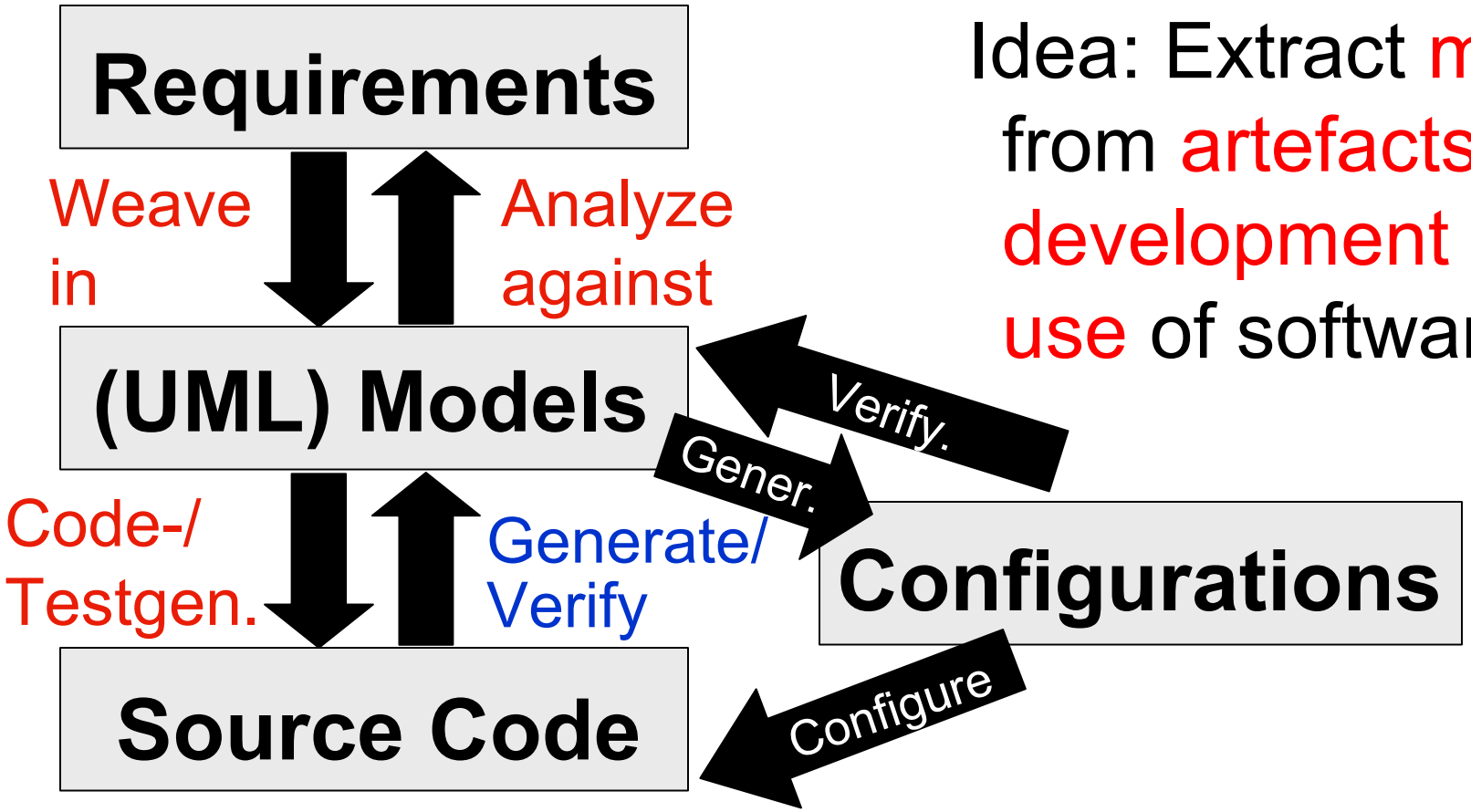
3) Generate models from code and verify these.

- 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.

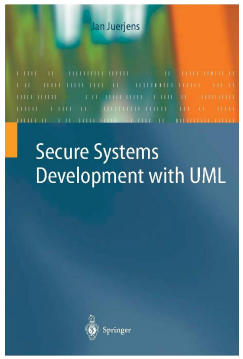
2) Create models and code manually and verify code against models.

- Advantages: Split heavy verification burden. Get some verification result already in design phase (for non-legacy implementations).

Background: 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

Based on usual Dolev-Yao model.

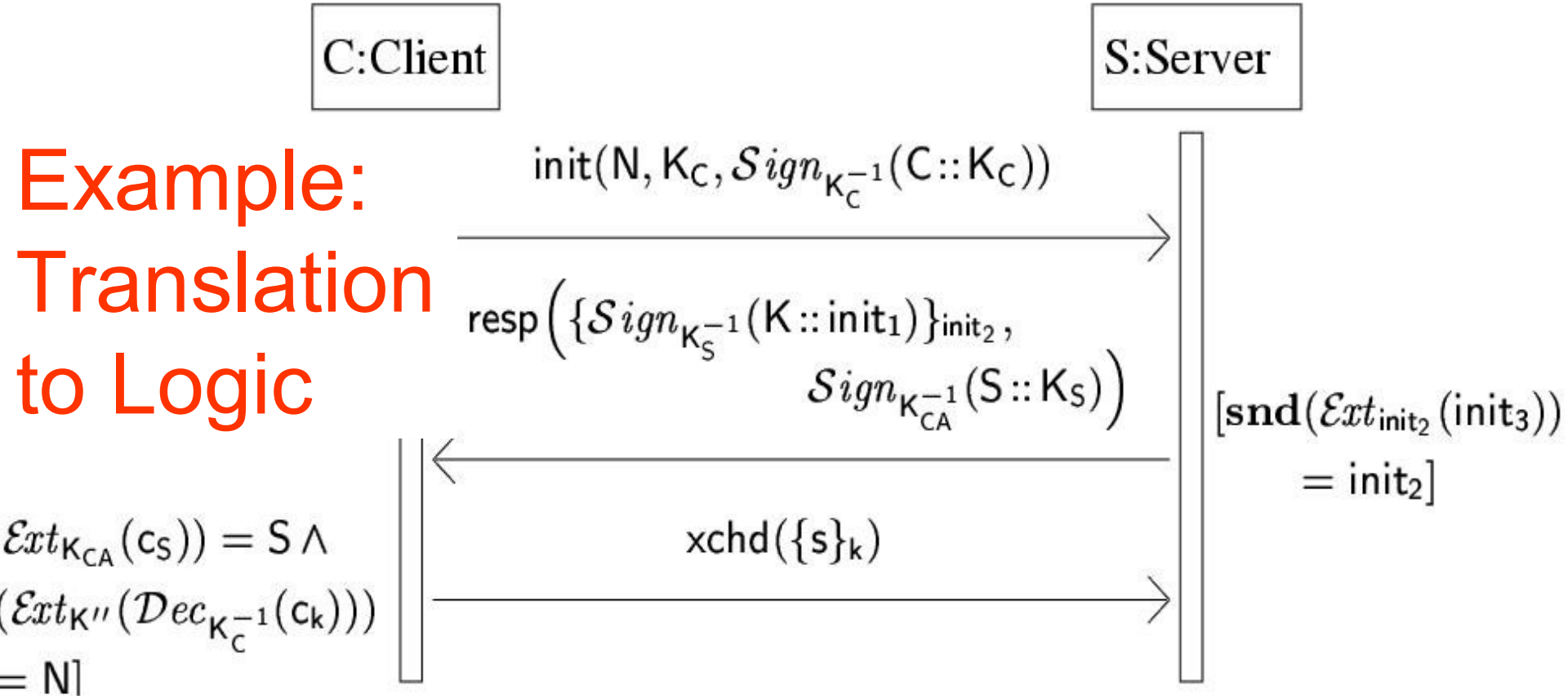
Approximate adversary knowledge set from above:

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

E.g. secrecy requirement:

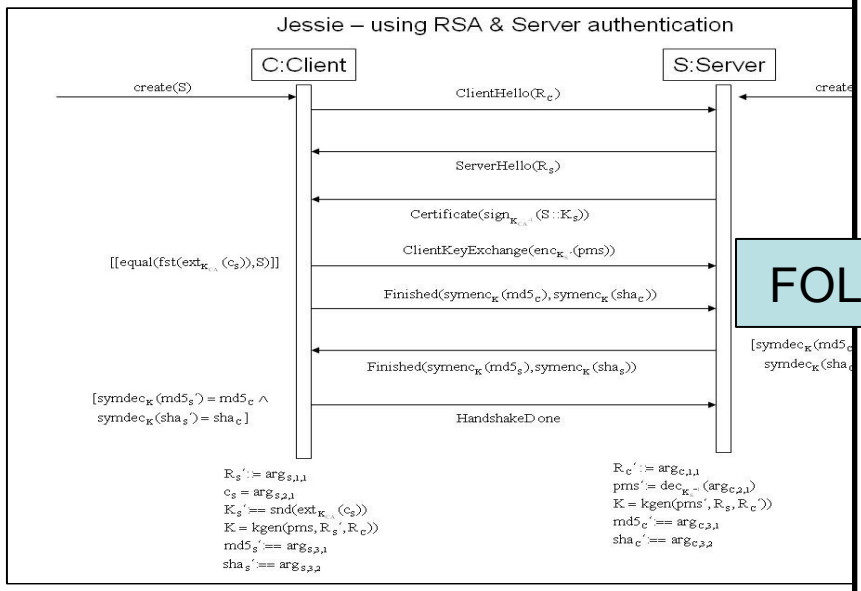
For any secret *s*, check whether can derive *knows(s)* from model-generated formulas using automatic theorem prover. [ICSE05]

Example: Translation to Logic



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

Model Verification



FOL

```
...
((
  knows(ArgC_3)
  & (equal(fst(ArgC_3), type_serverkeyexchange))
  & (equal(snd(ext(snd(snd(ArgC_3)), k_ca)), skey))
  & (equal(snd(ext(snd(ArgC_2), k_ca)), fst(snd(ArgC_3))))
)
=>(
  ((knows(ArgC_4_1)
   & equal(ArgC_4_1, type_serverhellodone))
   =>(
  ( ( true & equal(ClientKeyExchange, enc(premasterkey, skey)
)
...
%------------------------ Conjecture --

input_formula(attack,conjecture,(
   knows(mastersecret) )).
```

ATP

```
analyzing results ...
model found/total failure
time limit information: 19 total / 18 strategy
(leaving wrapper).
task myUML_PID1491 on atbroy1 has status SUCCESS
(model found by strategy 300) consuming 1 seconds
deleting temporary files.
e-SETHEO done. exiting
```

Check whether can derive *knows(s)*.

If yes, generate attack scenario.

If no, *s* secret (wrt our attacker).

Model vs Code ?

Model:

+ earlier (less expensive to fix flaws)
+ more abstract ➔ more efficient
- more abstract ➔ may miss attacks
- programmers may introduce security flaws
- even code generators, if not formally verified

Code:

+ „the real thing“ (which is executed)
+ applicable to legacy systems

[ASE05, ASE06]

➔ Verify both !

(NB: No previous work eg for legacy crypto protocols.)

Just an Exercise in Code Verification ?

State of the art in code verification in practice: execution exploration by testing (possibly generated from models). 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.

General approaches to formal software verification exist (Isabelle et al), but limited use by (civilian) software engineers, and usually not for sophisticated properties like Dolev-Yao security.

➔ Develop specialized verification approach.

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$_{K_{CA}^{-1}}$(S::K_s))

ClientKeyExchange(enc$_{K_{CA}}$(pms))

[[equal(fst(ext$_{K_{CA}}$(c_s)),S)]]

Finished(symenc$_K$(md5$_c$), symenc$_K$(sha$_c$))

Finished(symenc$_K$(md5$_s$), symenc$_K$(sha$_s$))

[symdec$_K$(md5$_s$') = md5$_c$ ∧ symdec$_K$(sha$_s$') = sha$_c$]

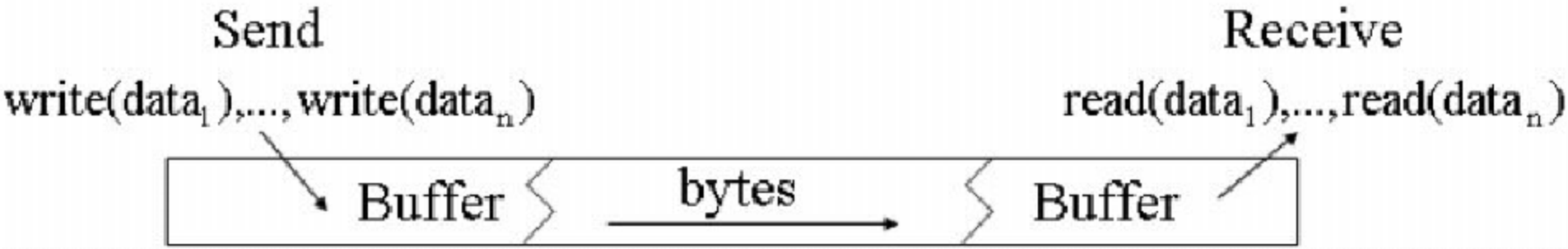
[symdec$_K$(md5$_c$ symdec$_K$(sha$_c$

Abstract model

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).



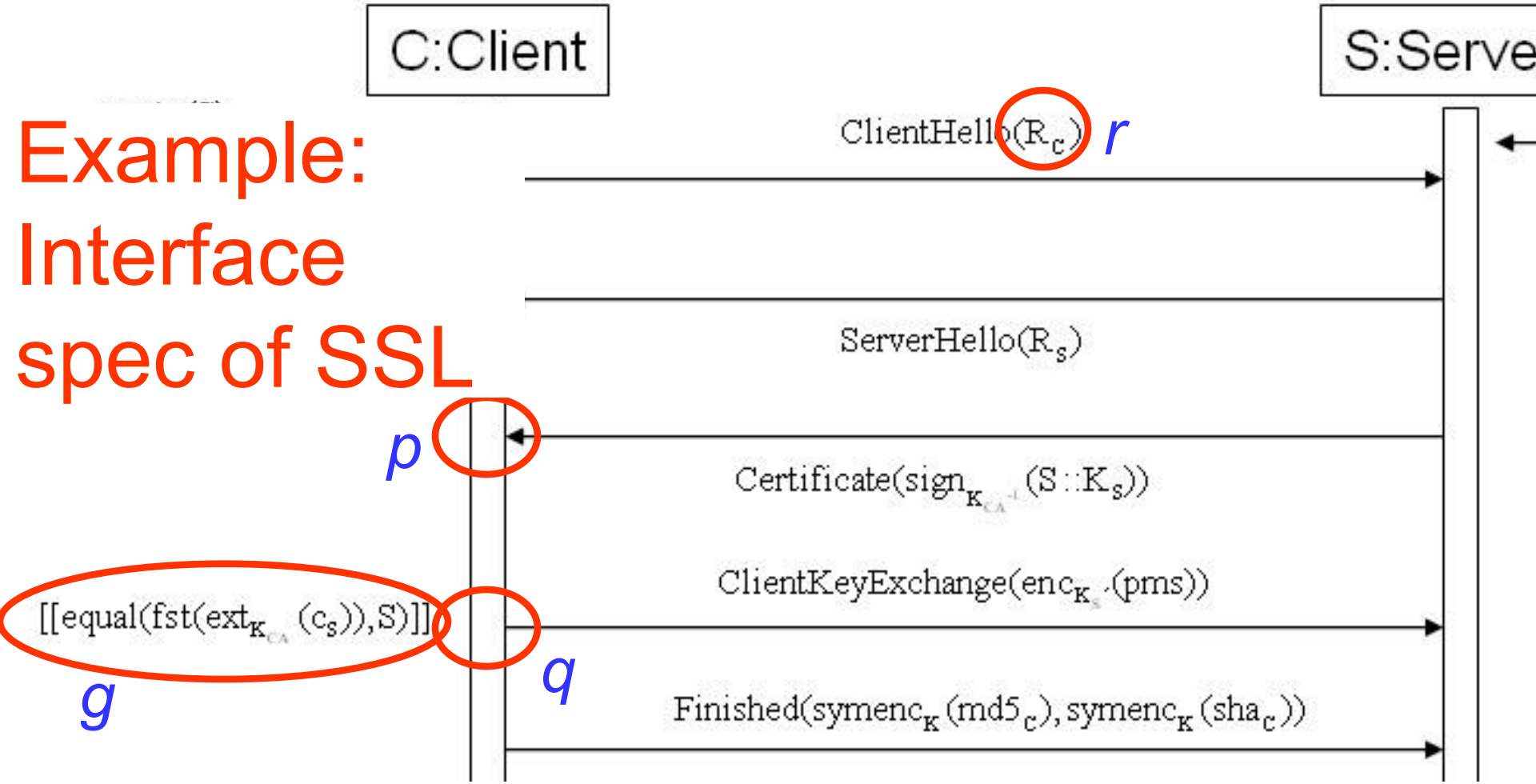
Example

Sending a protocol message (e.g. ClientHello):

- create the clientHello object with appropriate message parameters
- create the message object msg by giving the clientHello object as an argument
- call the write method at the msg object

```
ClientHello clientHello = new ClientHello(session.protocol,clientRandom,sessionId,
                                          session.enabledSuites,comp, extensions);
Handshake msg = new Handshake(Handshake.Type.CLIENT_HELLO, clientHello);
msg.write (dout, version);
```

Example: Interface spec of SSL



I) Identify program points:

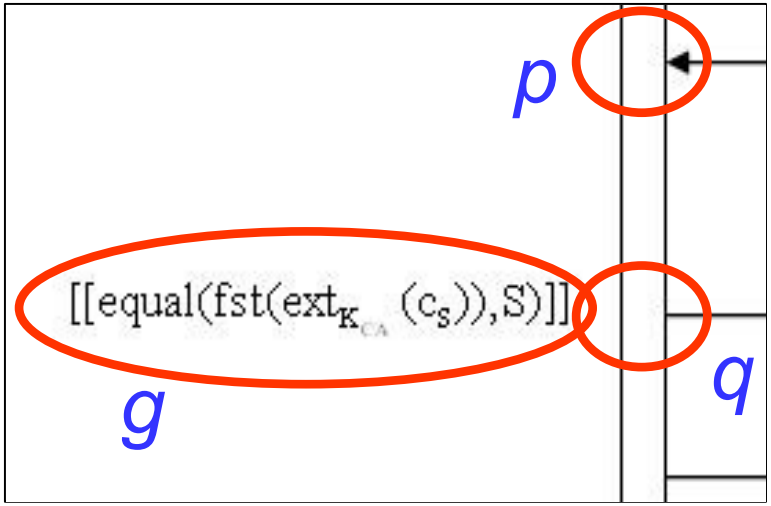
value (*r*), receive (*p*), guard (*g*), send (*q*)

II) Check guards enforced

Checking Guards

Guard *g* enforced by code?

b) Generate runtime check for *g* at *q* from diagram: simple + effective, but performance penalty.

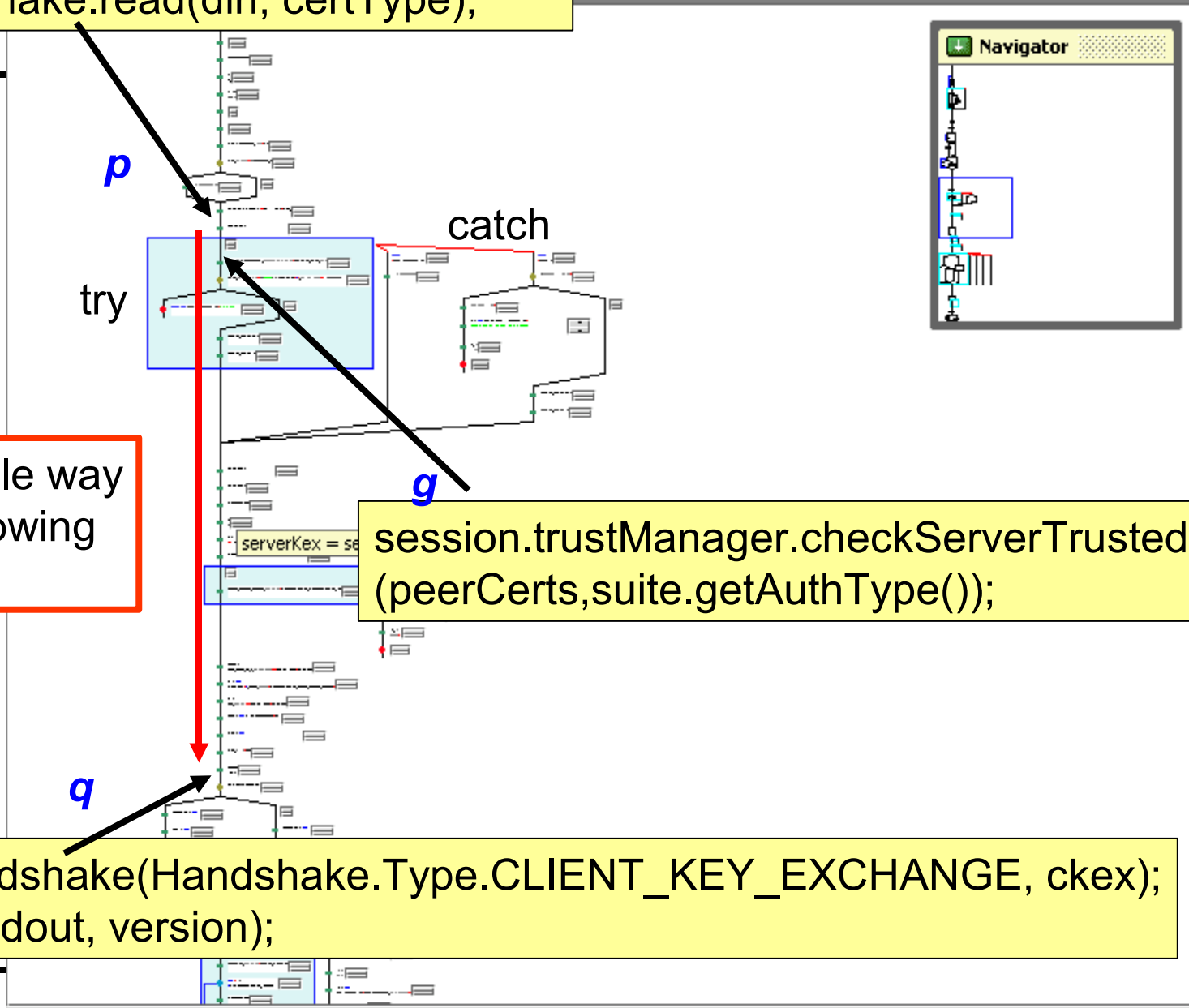


c) Testing against checks (symbolic abstractions for crypto).

d) Automated formal local verification: conditionals between *p* and *q* logically imply *g* (uses Prolog). E.g. Bandera.

Execution Path Diagram for method SSLSocket.doClientHandshake()

msg = Handshake.read(din, certType);



only possible way without throwing exception

session.trustManager.checkServerTrusted (peerCerts,suite.getAuthType());

msg = new Handshake(Handshake.Type.CLIENT_KEY_EXCHANGE, ckex);
msg.write (dout, version);

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.

Modular Verification: Formalisation

send: represents send command

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

$[d]\ p \models g$: g checked in any execution of p initially satisfying d before any send

write $p \models g$ for $[true]\ p \models g$.

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

Modular Verification: Some Rules

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

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

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

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

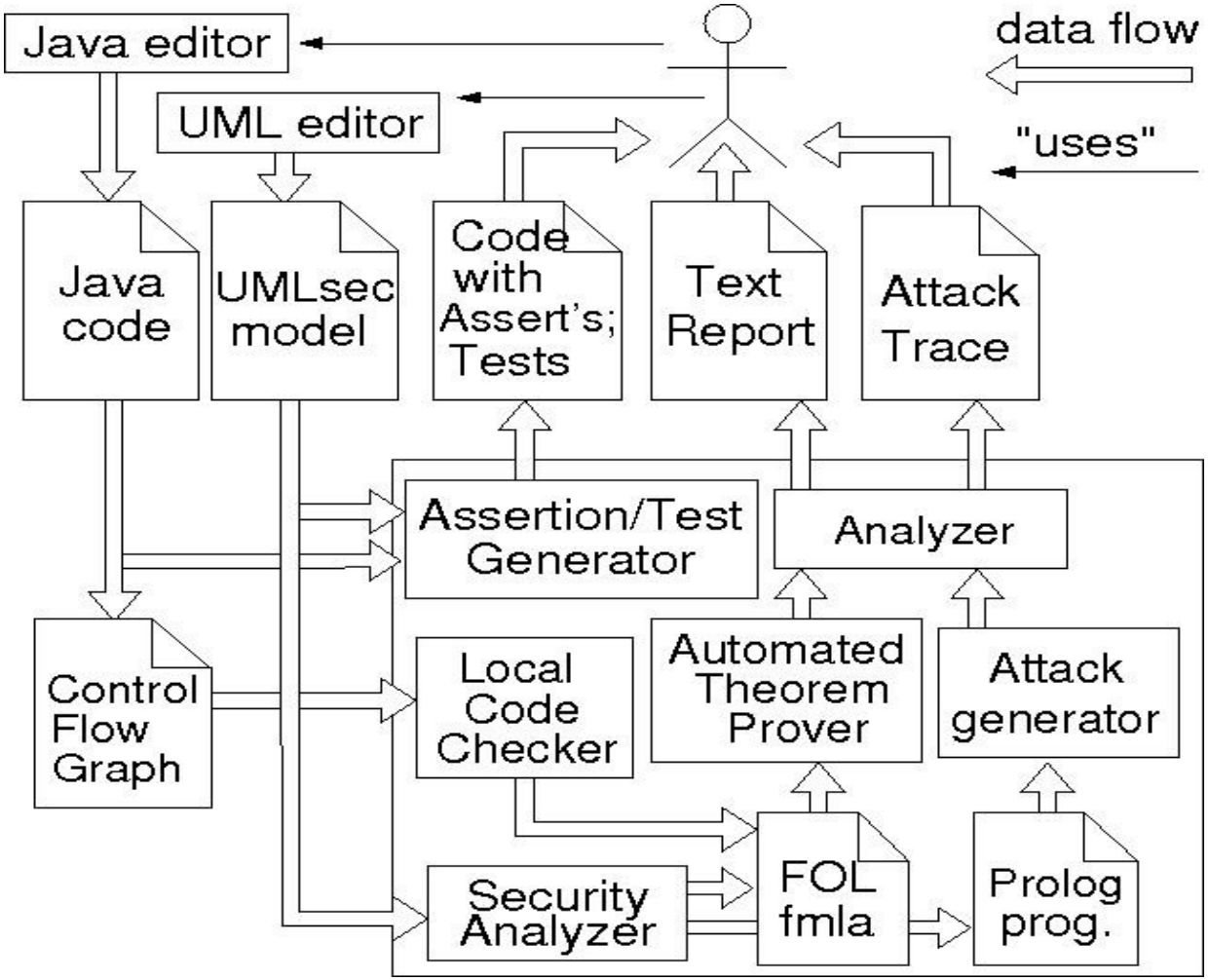
$$\frac{[d]p \models g}{x := e; p \models g} d \Rightarrow x = e$$

JSSE / Jessie

- Java Secure Sockets Extension contains implementation of SSL.
- Open-source clean-room reimplementation Jessie.
- Case-study: applied above 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.

Tool Support

[FASE05,ICSE06,ASE07, STTT07,ICSE08]



Also:

- configuration analysis: (user permissions, firewall rules/ policies) [FASE08]
- code traceability (with Yijun Yu)

Open-source

Linking to Reality II …

Analyzed designs / implementations / configurations e.g. for

- Common Electronic Purse Specs
- biometric authentication system
- Internet banking security architecture
- intranet information system
- mobile communications architectures

Allianz

Deutsche Bank

HypoVereinsbank

CEPS™

BMW Group

Münchener Rück
Munich Re Group

Bundesministerium für Bildung und Forschung

Bundesministerium für Wirtschaft und Technologie

O2

infineon

Intranet Information System

BMW Group

MetaSearch Engine: Personalized search in company intranet (including password protected).

Some documents highly security-critical. [ICSE07]

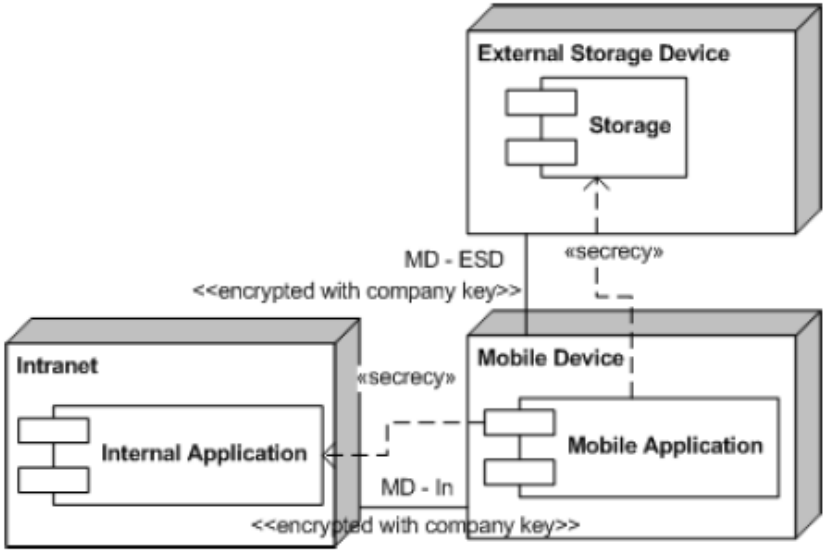
More than 1,000 potential users, index 280,000 documents, allow 20,000 queries per day.

Seamlessly integrated in enterprise-wide security reference architecture. Provides security services to applications, including user authentication, role-based access control, global single-sign-on and hook-up of new security apps.

Used MBSE to demonstrate architecture is secure.

Mobile Communications

- Application of Model-based Security Assurance at Mobile Communication Systems at O2 (Germany) [ICSE08]
- All 62 relevant security requirements derived from the security policy could success-fully be established using the approach



Future Work

- PhD project on “Verifying Implementations of Crypto-Protocols in C” (MSR Cambridge / Andy Gordon). Hiring now ☺.
- Goal: Verify implementations in practical use
- Most crypto protocols implemented in C (openSSL, openSSH, Kerberos, …)
- Initial experiments with SLAM, Blast (some technical challenges…)
- Hopeful about VCC

Conclusion

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

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

Experiences so far encouraging.

Still many challenges to address – collaboration always welcome !

Current Projects

- PhD project on Verifying Implementations of Crypto-Protocols in C (MSR Cambridge / A. Gordon)
- RoySoc JIP with TU Munich on Formal Model-based Analysis of Cryptoprotocol Implementations
- RoySoc JIP with NII (Tokyo) on Security Requirements vs Design
- PhD project on IT security risk assessment with Munich Re
- PhD project on Adaptive Security for Ambient Technology
- PhD project on fuzzy reasoning for IT security risks
- PhD project on model-based development for avionics

Questions ?

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

http://www.jurjens.de/jan

J.Jurjens@open.ac.uk