# Model-based Security Engineering: Models vs. Code

Jan Jürjens

Department of Computing
The Open University, GB

J.Jurjens@open.ac.uk

http://www.umlsec.org

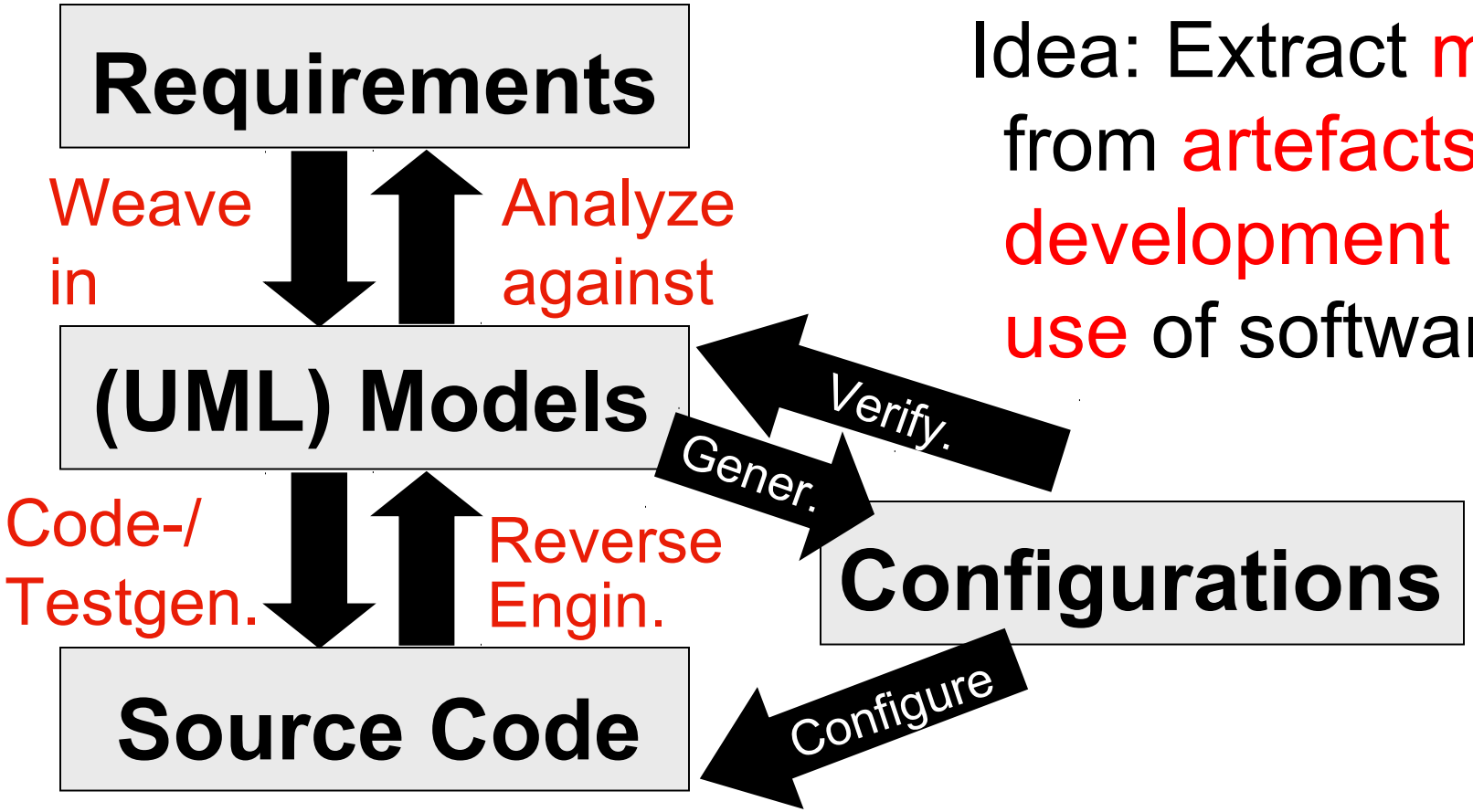
# Challenge: Security

Security is holistic property:

- Attackers often circumvent (not: break) mechanisms.
- Transform (in)secure components to secure systems ?

„Those who think that their problem can be solved by simply applying cryptography don`t understand cryptography and don`t understand their problem“ (B. Lampson / R. Needham).

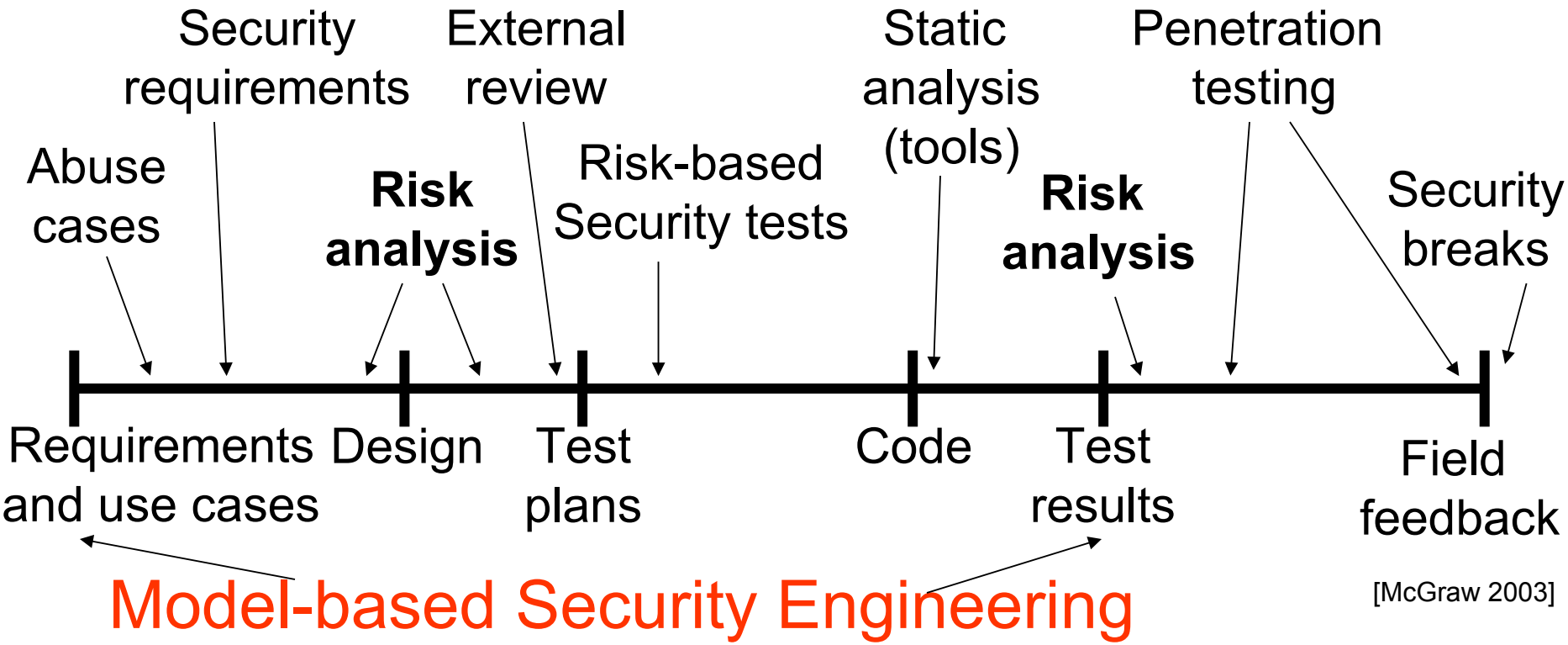
# Model-based Security Engineering




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

➔ Tool-supported, theoretically sound, efficient automated security design & analysis.

# Secure System Lifecycle




Model-based Security Engineering

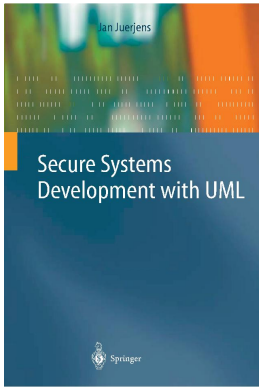
[McGraw 2003]

Design: Encapsulate prudent security engineering rules.
Analysis: Formally based, automated, efficient tools.
Note: emphasis on high-level requirements.

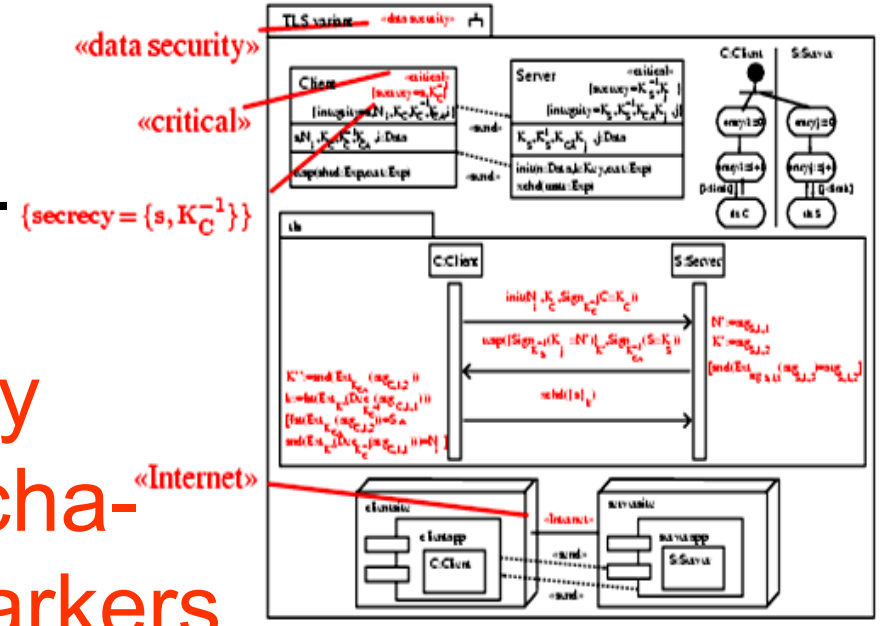
# Model-based Security with UMLsec

Extension of the Unified Modeling Language (UML) for secure systems development.

- evaluate UML models for security
- encapsulate established rules of prudent secure engineering
- make available to developers not specialized in secure systems
- consider security requirements from early design phases, in system context
- can use in certification



# UMLsec

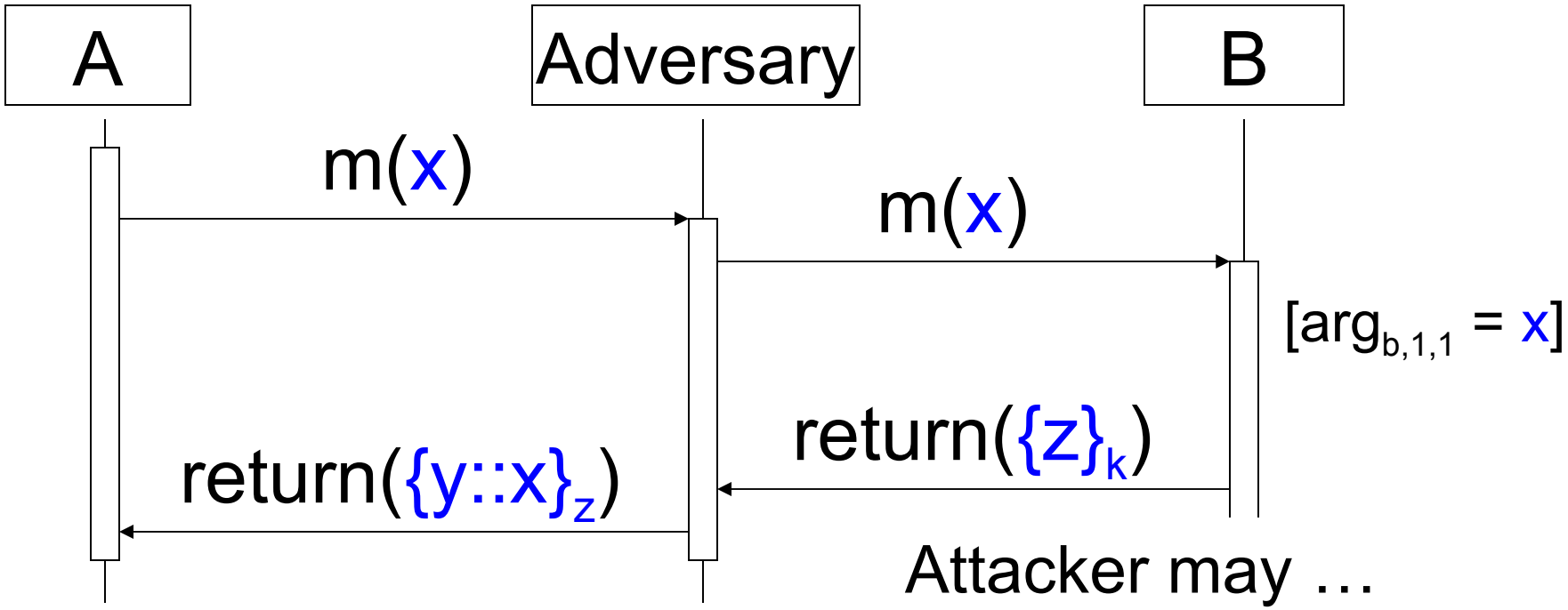



Insert recurring security requirements, adversary scenarios, security mecha-nisms as predefined markers.

Use associated logical constraints to verify specifications using model checkers and ATPs based on formal semantics.

Ensures that UML specification enforces the relevant security requirements wrt Dolev-Yao type adversaries. [FASE01,UML02,FOSAD05,ICSE05]

# Example: Crypto-based Distributed System




Adversary knowledge:

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

(cf. [Dolev, Yao 1982])

Attacker may …

- control system parts,
- know data in advance,
- intercept messages,
- delete messages,
- inject messages.

# Security Analysis in First-order Logic

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]

# 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)

under equations …

# Cryptographic Expressions II

$\forall\forall\ E,K.\ Dec_K^{-1}(\{E\}_K)=E$

$\forall\forall\ E,K.\ Ext_K(Sign_K^{-1}(E))=E$

$\forall\forall\ E_1,E_2.\ head(E_1::E_2)=E_1$

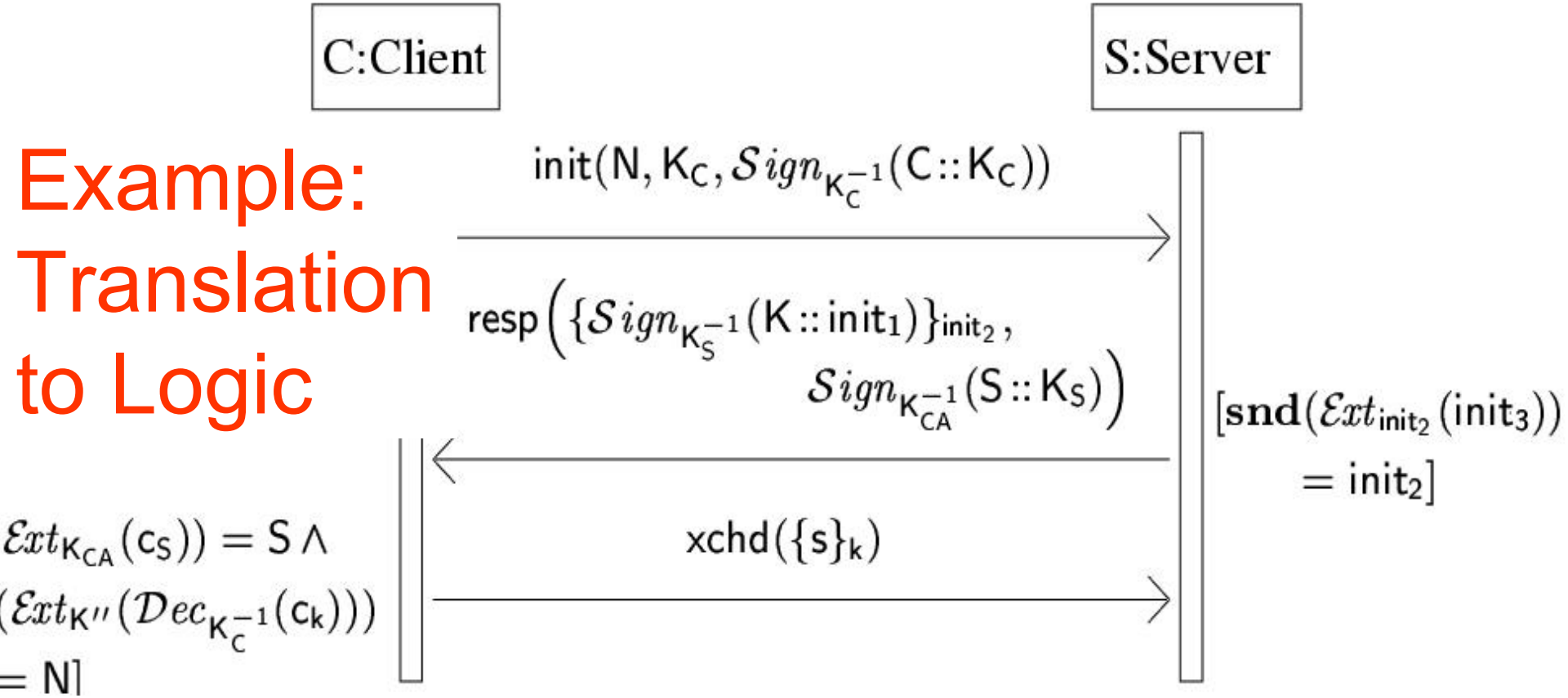
$\forall\forall\ E_1,E_2.\ tail(E_1::E_2)=E_2$

- Associativity for $::$.

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, …).

# Example: Translation to Logic




$$\mathit{knows}(N) \wedge \mathit{knows}(K_C) \wedge \mathit{knows}(\mathit{Sign}_{K_C^{-1}}(C::K_C))$$
$$\wedge\ \forall \mathit{init}_1, \mathit{init}_2, \mathit{init}_3.[\mathit{knows}(\mathit{init}_1) \wedge \mathit{knows}(\mathit{init}_2) \wedge$$
$$\mathit{knows}(\mathit{init}_3) \wedge \mathit{snd}(\mathit{Ext}_{\mathit{init}_2}(\mathit{init}_3)) = \mathit{init}_2$$
$$)\ \mathit{knows}(\{\mathit{Sign}_{K_S^{-1}}(\ldots)\}_{\ldots}) \wedge [\mathit{knows}(\mathit{Sign}\ldots)]$$
$$\wedge\ \forall \mathit{resp}_1, \mathit{resp}_2.\ [\ldots)\ldots]]$$

# Analysis

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

Surprise: Yes !

➔ Protocol does not preserve secrecy of *s*.

Why ? Use Prolog-based attack generator.

```
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),
   & % Server -> Attacker (2. mes
      (  (    knows(ArgS_11)
            & knows(ArgS_12)
            & knows(ArgS_13
            & ( ? [X] :

      => (    know

 & %

      & equal(
              ArgC_11 )
      & equal(enc(sign(conc(DataC_k, DataC_n), inv(DataC_
              ArgC_11 )
      )
    => ( knows(symenc(secret, DataC_k)) )  )
) )  )).
```

```
(E-SETHEO csp03 single processor running on host ...
(c) 2003 Max-Planck-Institut fuer Informatik and
Technische Universitaet Muenchen
analyzing results ...
proof found
time limit information: 298 total / 297 strategy
```

Attack

# Man-in-the-Middle Attack

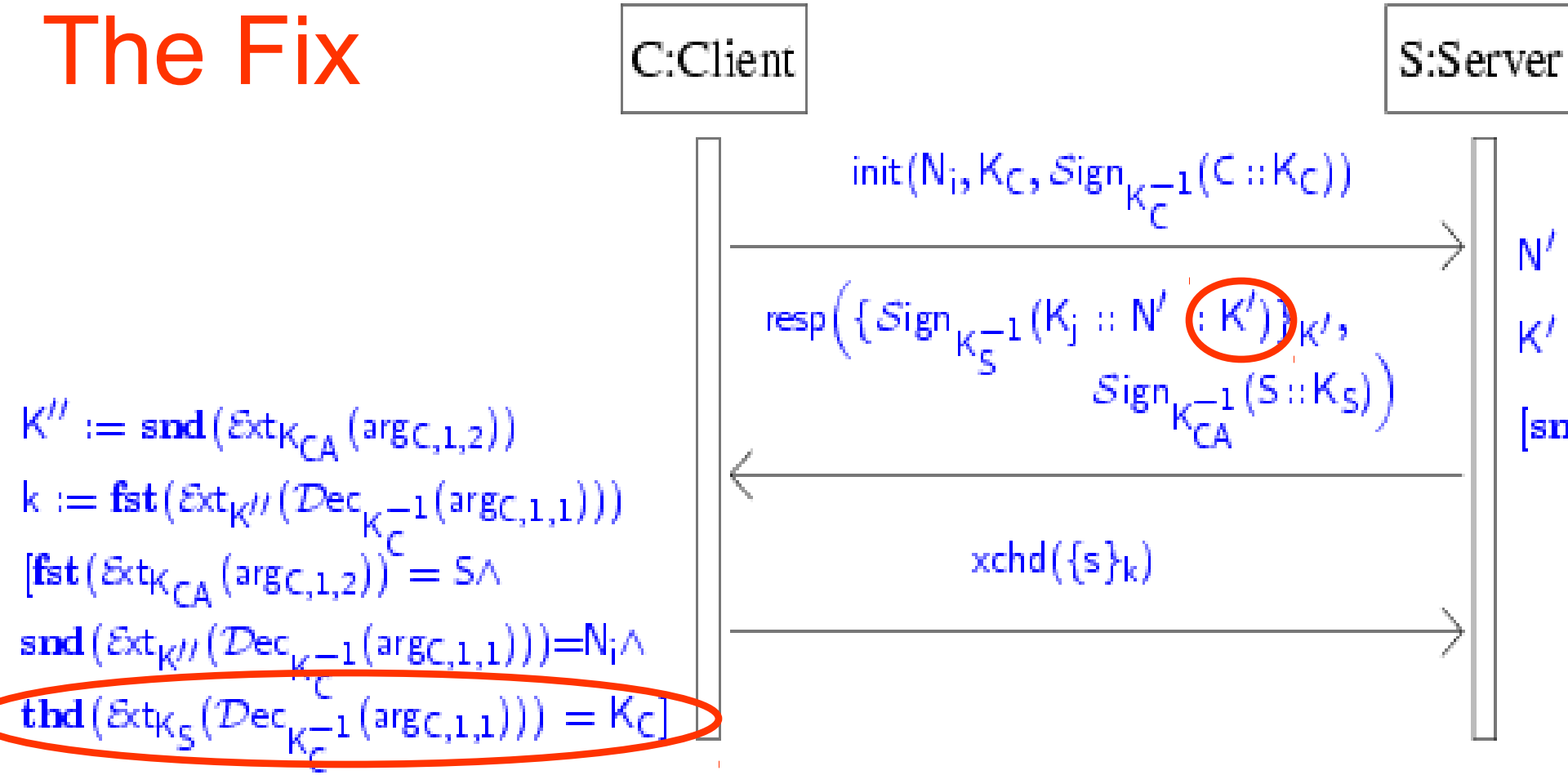
$$C \xrightarrow{N_i::K_C::Sign_{K_C^{-1}}(C::K_C)} A \xrightarrow{N_i::K_A::Sign_{K_A^{-1}}(C::K_A)} S$$

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

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

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

# The Fix




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

Note completeness of FOL (but also undecidability).

# Security Analysis: Model or 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)

➔ Do both !

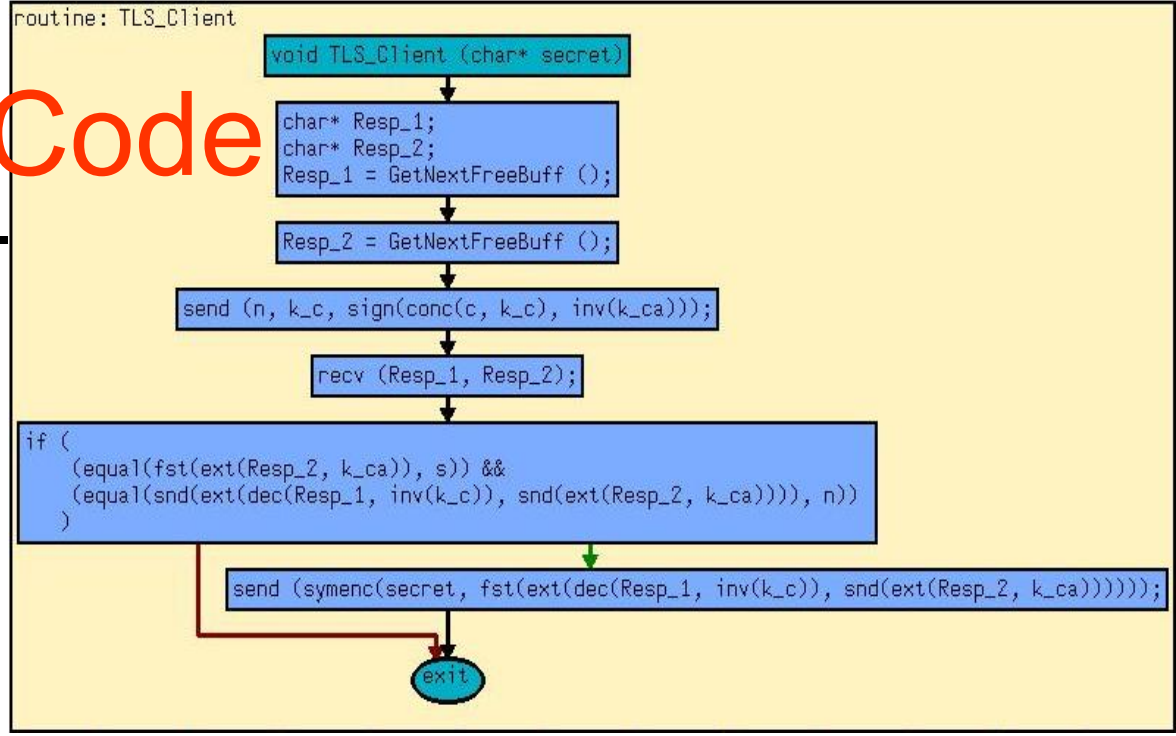
Surprise: Essentially no existing work (eg for crypto prots) !

# Code Analysis vs. Model Analysis

Options:

- generate code from models
  ➔ currently not available in general
- generate models from code
  ➔ next slides
- create models and code manually and verify code against models
  ➔ later in this talk

# Models from Code



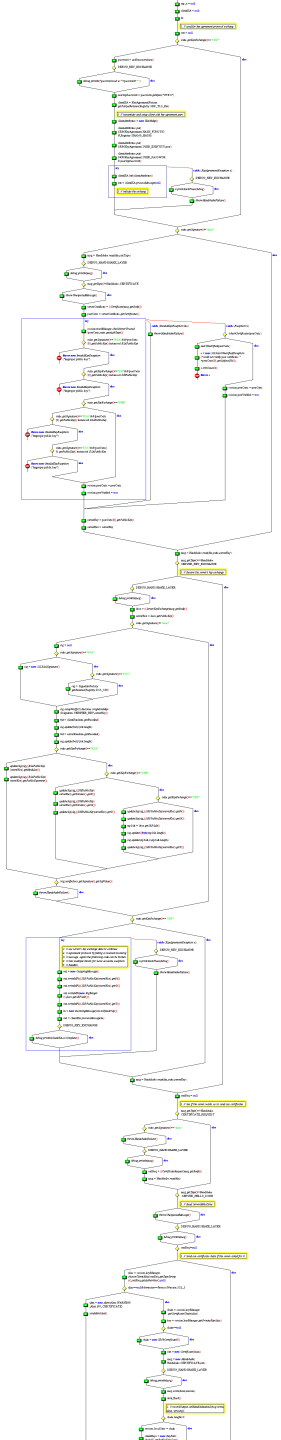

Generate control flow graph (e.g. aicall (Absint)).

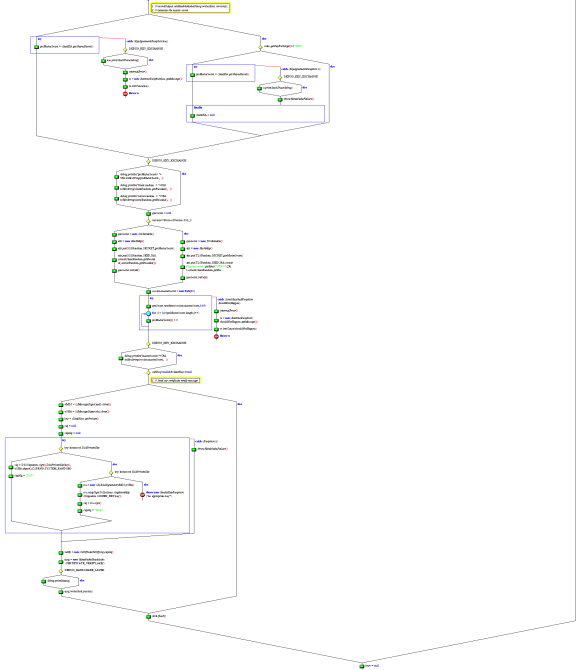
Transform to state machine:

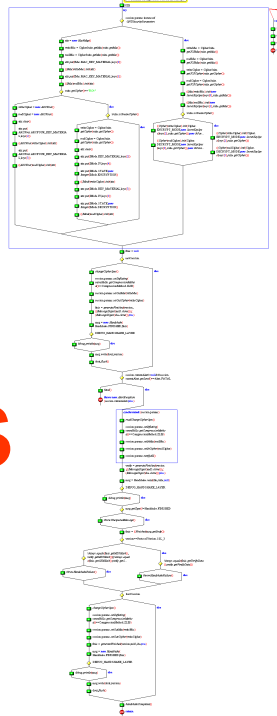
trans(state,inpattern,condition,action,nextstate)

where action can be outpattern or localvar:=value.

[ASE05,ASE06]







# Real Life Challenges

...

rjens,

# Experiences

Can generate behavioral models from code (e.g. CFGs). Problem: too concrete

➔ understanding + automated verification hard (even with annotations).

Constructing abstract specifications from practical software is manually intensive.

# Code Analysis vs. Model Analysis

Options:

- generate code from models
  ➔ currently not possible in general
- generate models from code
  ➔ challenging
- create models and code manually and verify code against models
  ➔ next slides

# Verify Code against Models

Assumption: Have textual specification.
Then:

- construct interface spec from textual spec
- analyze interface spec for security
- verify that software satisfies interface spec

# Model vs. Implementation

[with David Kirscheneder ]

„meaning“

„meaning“

**compare meaning!**

Backtrace assignments

Defined during model creation

Sent and received data

Elements of connections

Find

Has

**Implement -ation**

.java

Equal?

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$_c$))

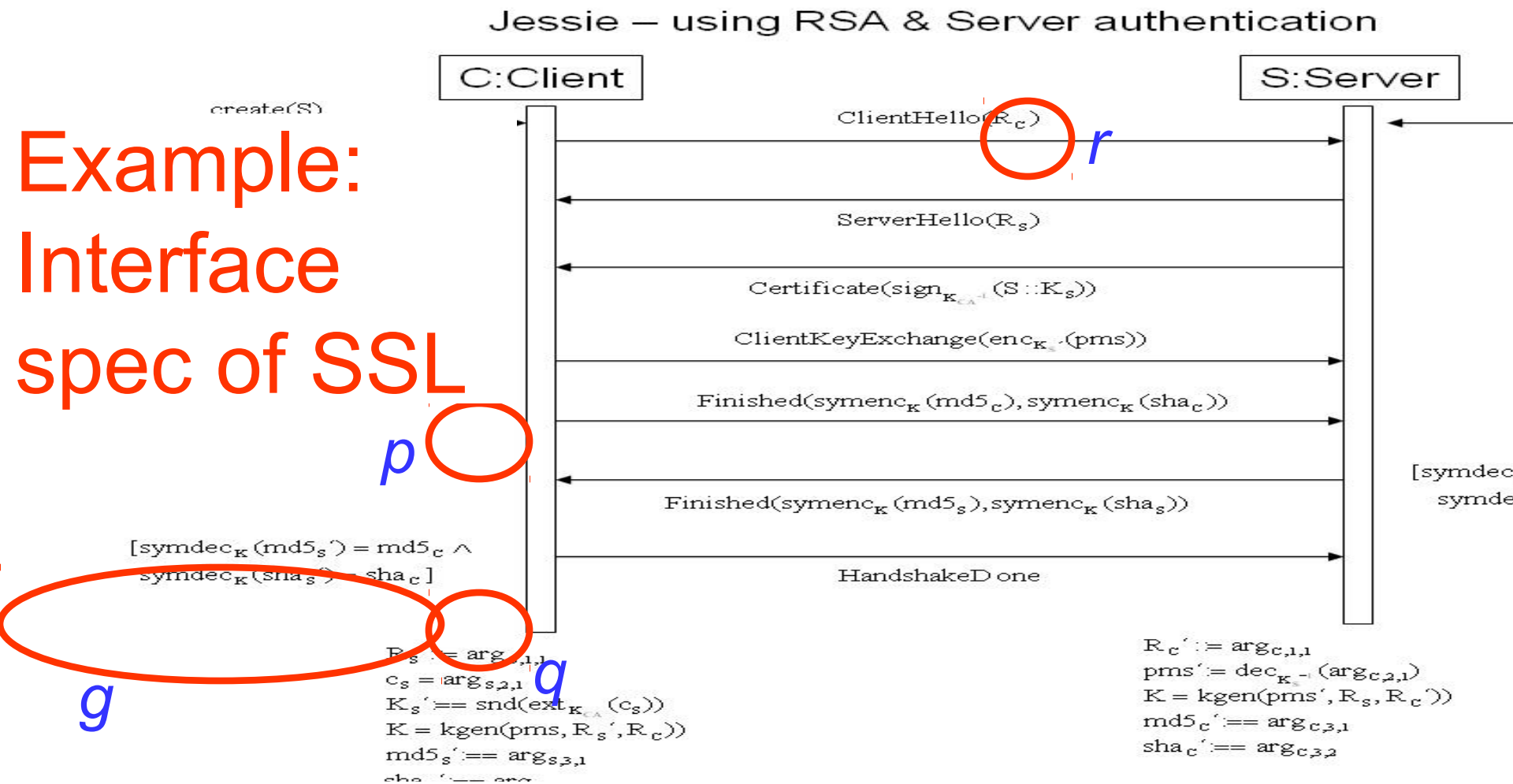
Finished(symenc$_K$(md5$_s$), symenc$_K$(sha$_s$))

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

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

**Abstract model**

# Example: Interface spec of SSL




I) Identify program points:

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

II) Check guards enforced

# Implementation of SSL: Identify Values

| Parameter der kryptographischen ClientHello Nachricht | Effektiv übertragene Daten der ClientHello Nachricht der Jessie Implementierung |
|---|---|
| C | type.getValue() |
| Pver | major |
| | minor |
| | ((gmtUnixTime >>> |
| | ((gmtUnixTime >>> 16) & 0xFF) |
| | ((gmtUnixTime >>> 8) & 0xFF) |
| | (gmtUnixTime & 0xFF) |
| r_c | randomBytes |
| | sessionId.length |
| Sid | sessionId |
| | ((suites.size() << 1) >>> 8 & 0xFF) |
| | ((suites.size() << 1) & 0xFF) |
| LCip | suites_1 |
| | ... |
| | suites_N |
| | comp.size() |
| LKomp | comp_1 |
| | ... |
| | comp_N |

```
public void write(OutputStream out) throws IOException
 {        ...    out.write(randomBytes);    …        }
```

**Identify: randomBytes**

(in message *ClientHello*)

2nd parameter of Random constructor called by ClientHello.write()

```
 public void write(OutputStream out)
throws IOException
{ ... random.write(out); ... }
```

2nd parameter of ClientHello constructor

```
ClientHello(… , Random random, )
{   ... this.random = random;    ... }
```

via Handshake.write()

initialized in SSLSocket.doClientHandshake()

```
ClientHello clientHello = new ClientHello(...,clientRandom,...);
```

initialization of the used Random object

```
Random clientRandom =
new Random(...,session.random.generateSeed(28));
```

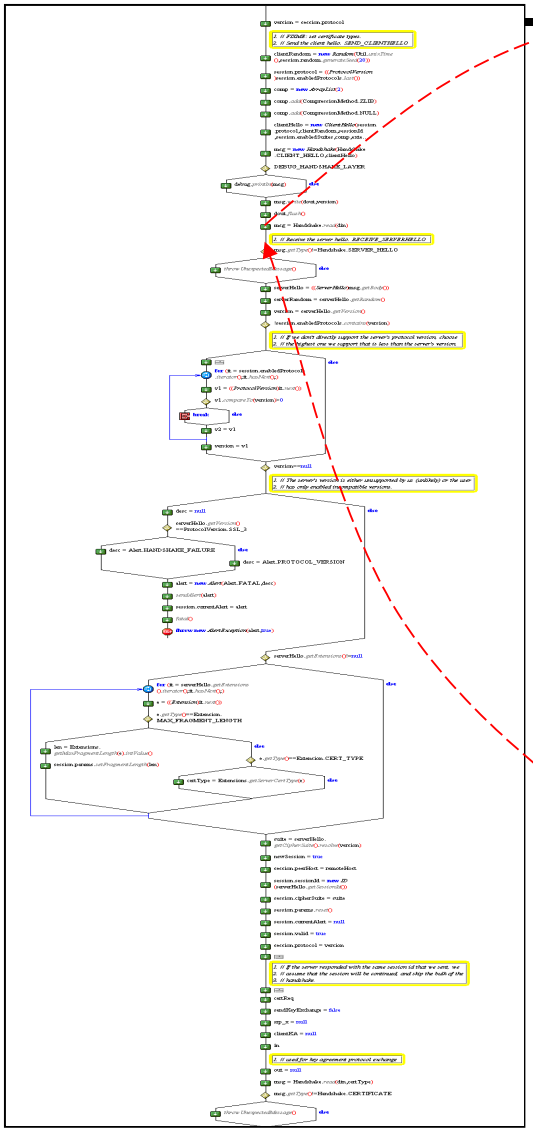
„meaning“

**class SecureRandom** (specified in: FIPS 140-2,RFC 1750) of package java.security

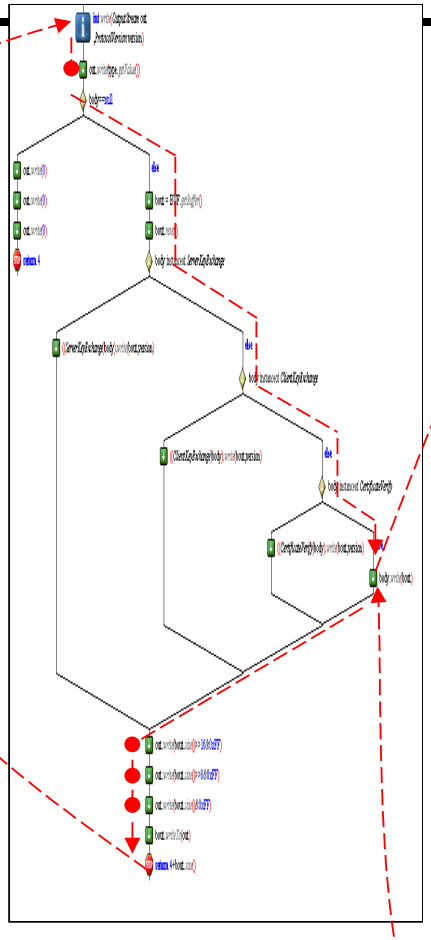
**Function: generateSeed**

# Sending Messages

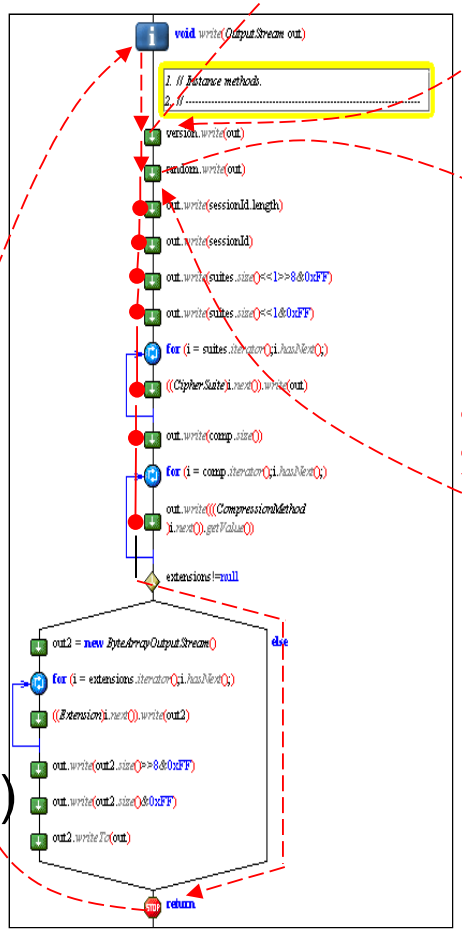
Automate this using patterns



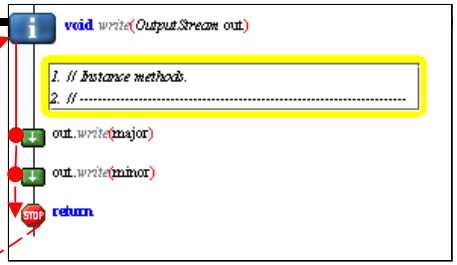
SSLSocket.doClientHandshake()



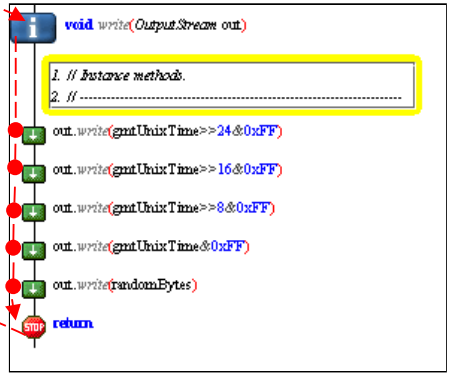
Handshake.write()



ClientHello.write()



ProtocolVersion.write()

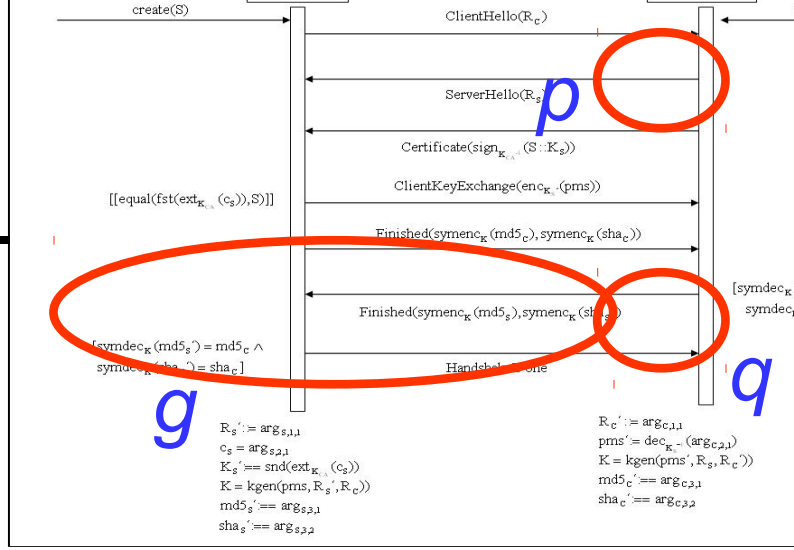


Random.write()

traverse CFG

call of OutputStream.write()

# Checking Guards




Guard *g* enforced by code?

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

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

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

```
public void checkServerTrusted(X509Certificate[] chain, String authType)
  throws CertificateException {…      checkTrusted(chain, authType);    }
```

calls checkTrusted()

Guard:
checkServerTrusted()

```
private void checkTrusted(X509Certificate[] chain,
String authType) throws CertificateException
   {      ...    }
```

calls verify() for every member of certificate chain

```
public void verify(PublicKey key, String provider)
 throws CertificateException, ...
   {      ...    }
```

„meaning
“

calls doVerify()

**va.security.Signature**
**Initializatize**
**Update**
**Verify**
**erifies the signature“**

```
private void doVerify(Signature sig,PublicKey key)
  throws CertificateException, ...
 {  ...    sig.initVerify(key);
  sig.update(tbsCertBytes);
  if (!sig.verify(signature))
 {… throw new CertificateException
      ("signature not validated"); …   }  }
```

Execution Path Diagram for method SSLSocket.doClientHandshake()

msg = Handshake.read(din, certType);

*p*

catch

try

Navigator

only possible way without throwing exception

*g*

serverKex = se

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

*q*

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

Class Sequence Flow

# 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

$[d]\ p\ ^2 g$ : $g$ checked in any execution of $p$ initially satisfying $d$ before any send

write $p\ ^2 g$ for $[true]\ p\ ^2 g$.

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

# Some Rules (Simplified)

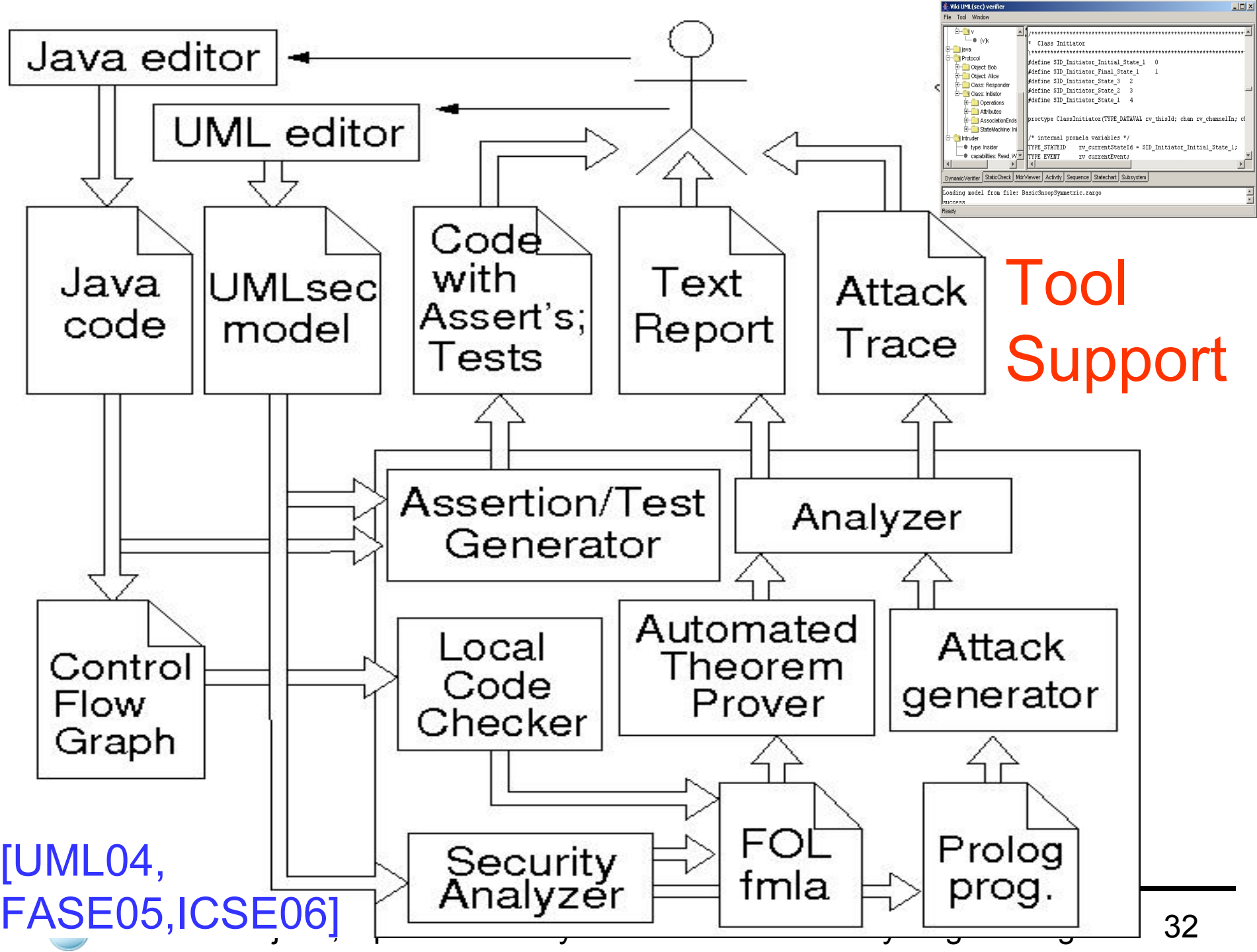
$$\frac{}{[d]\ \text{if}\ c\ \text{then}\ p\ \text{else}\ q \models g}(c \wedge d \Rightarrow g,\ \text{no send in}\ q)$$

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

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

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

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


Java editor
UML editor
Java code
UMLsec model
Code with Assert's; Tests
Text Report
Attack Trace
Tool Support
Assertion/Test Generator
Analyzer
Control Flow Graph
Local Code Checker
Automated Theorem Prover
Attack generator
Security Analyzer
FOL fmla
Prolog prog.
[UML04, FASE05,ICSE06]

# Some Applications

Analyzed designs / implementations / configurations for

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

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

# Conclusion

Seemingly first approach to formally based security verification for crypto-based Java implementations.

As far as possible automated and relatively efficient due to abstraction tailored to verification problem.

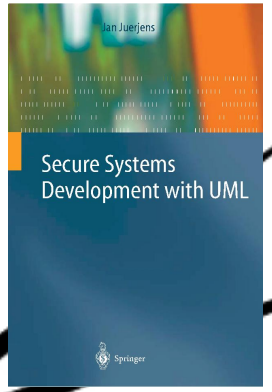
Still many challenges to address – collaboration always welcome !

Don‘t miss my Tutorial on Tuesday and Experience Track Talk on Wednesday !

# Overview

Security Engineering

- Patterns
- Services
- Architectures
- Components
- Aspects
- Layers
- Requirements
- UMLsec
  - Formal Semantics
  - Analysis Framework
  - Model Checking
  - Autom. Theorem Proving
  - Refinement
- Processes
- Models
- Applications
  - CEPS
  - Biometry
  - Cryptokey
  - Jessie
  - TLS variant
  - Wiesncard
- Code
  - Runtime Checks
  - Model-based Testing
  - Autom. Theorem Proving
- Foundations





Dependable Systems Development

- Fault-tolerance
- Reliability
- Real-time
- Security
  - Engineering
  - Management
    - Risk Assessment
    - Permissions
    - Business Processes
    - Security Investment
    - Firewall Configurations

# Questions ?

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

http://www.umlsec.org

J.Jurjens@open.ac.uk