Security protocols, properties, and their monitoring

Andreas Bauer, Jan Jürjens

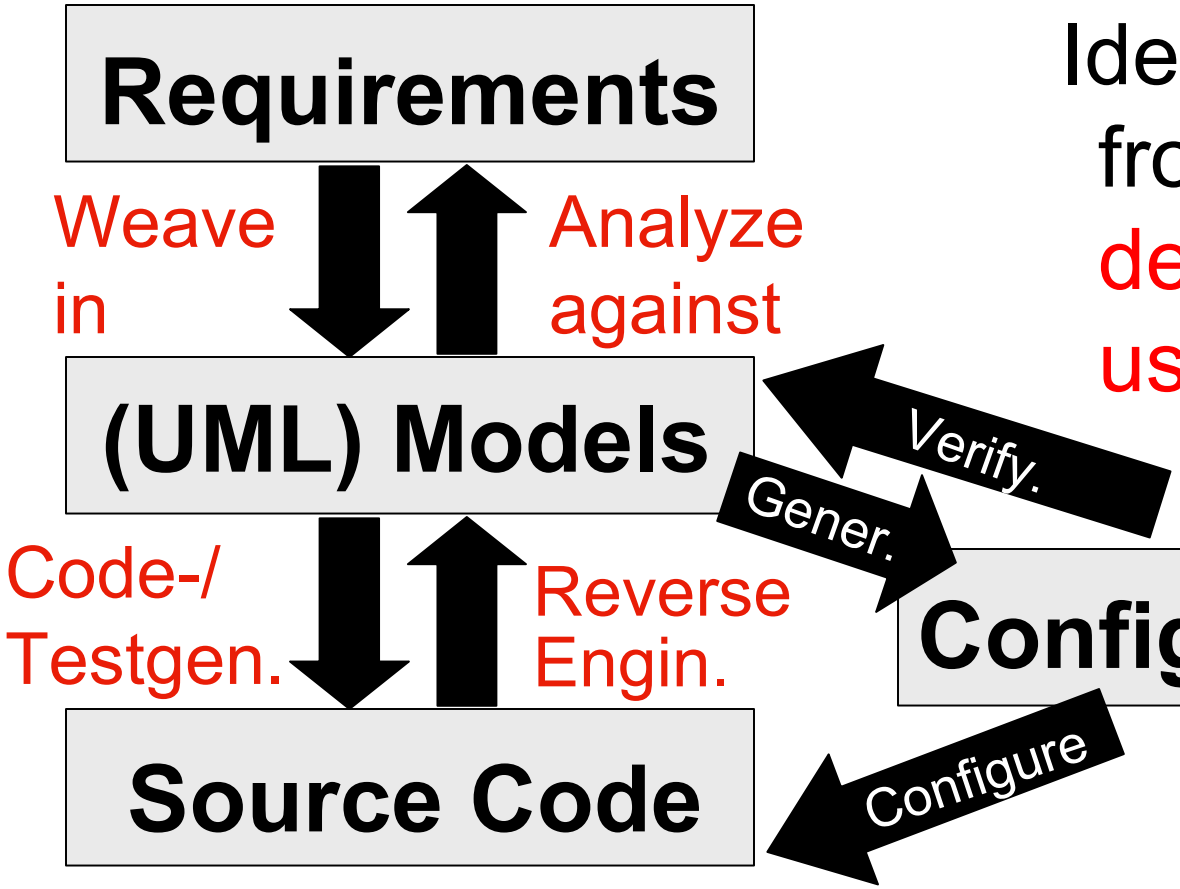
Computer Sciences Lab
The Australian National University

Computing Department
The Open University, GB

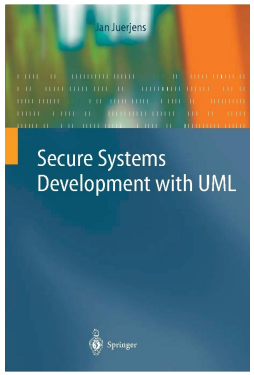
J.Jurjens@open.ac.uk

http://www.umlsec.org

Model-based Security Engineering

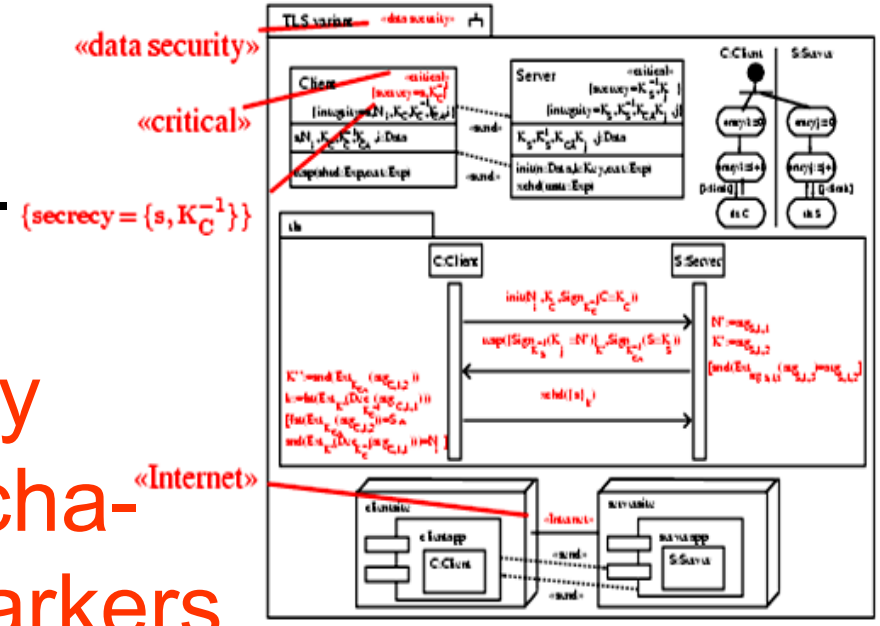


Idea: Extract models from artefacts in development and use of software.



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

UMLsec

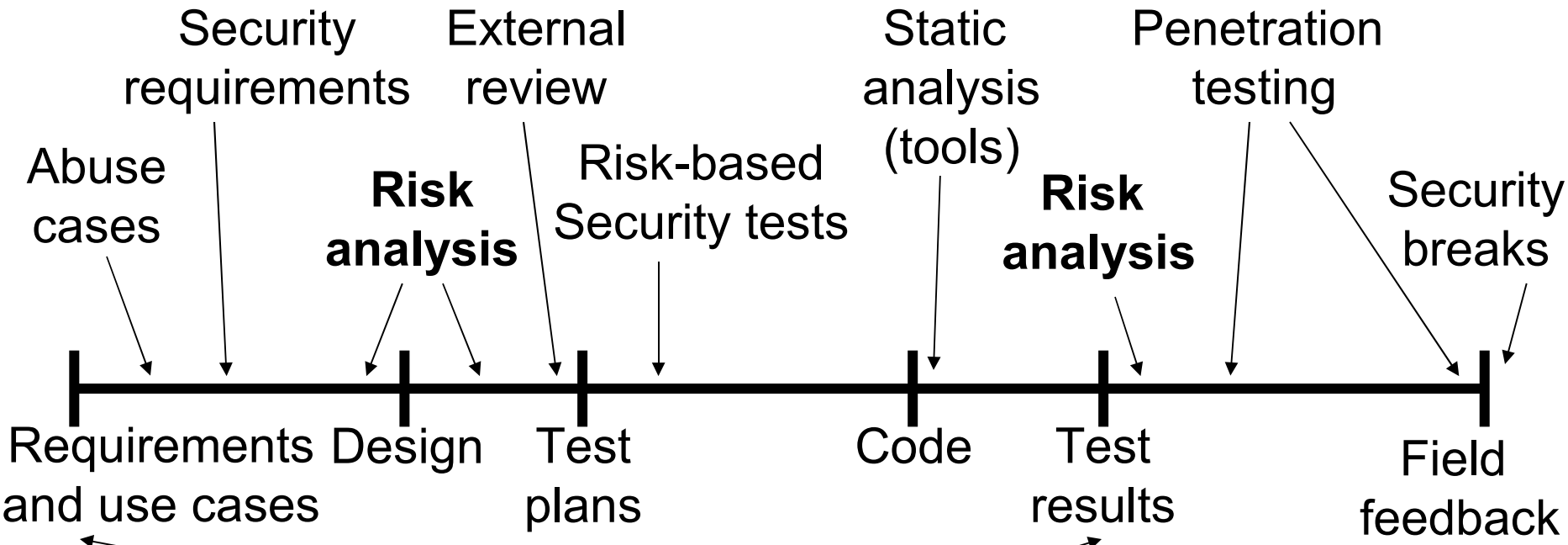


Insert recurring security requirements, adversary scenarios, security mecha-nisms as predefined markers.

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]

Secure System Lifecycle

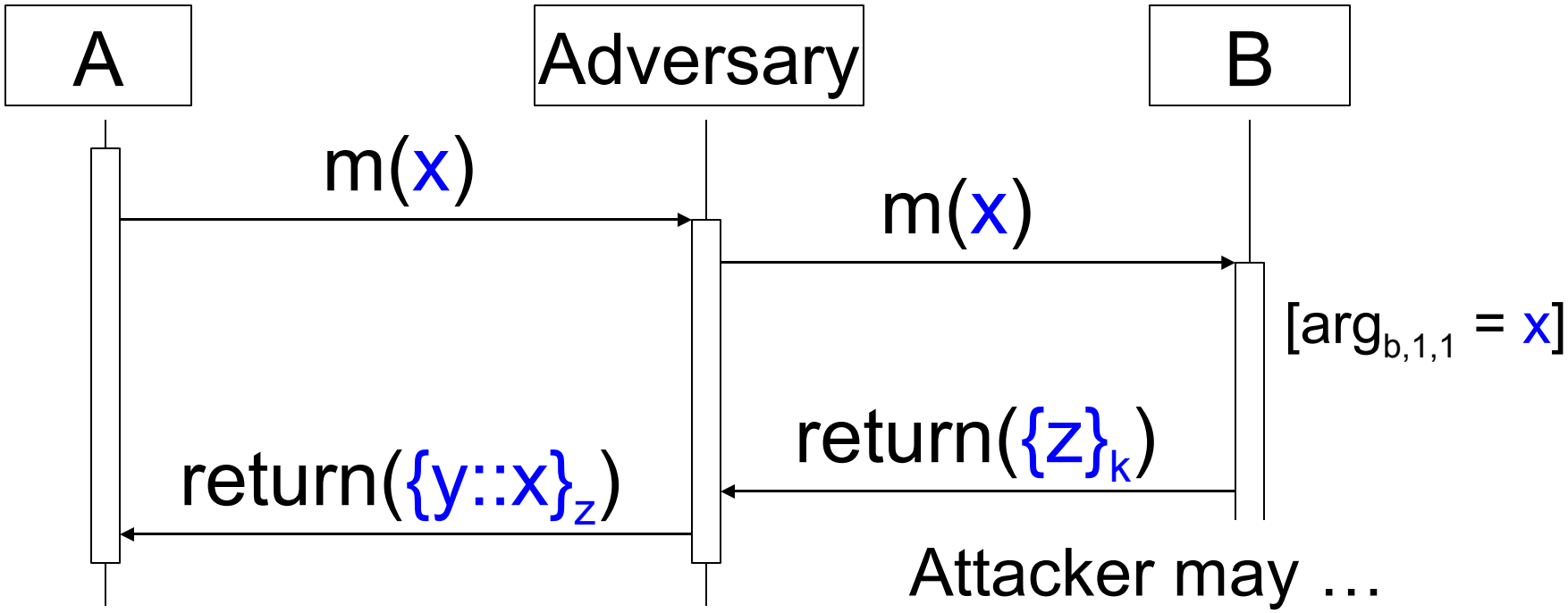


Model-based Security Engineering

[McGraw 2003]

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

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.

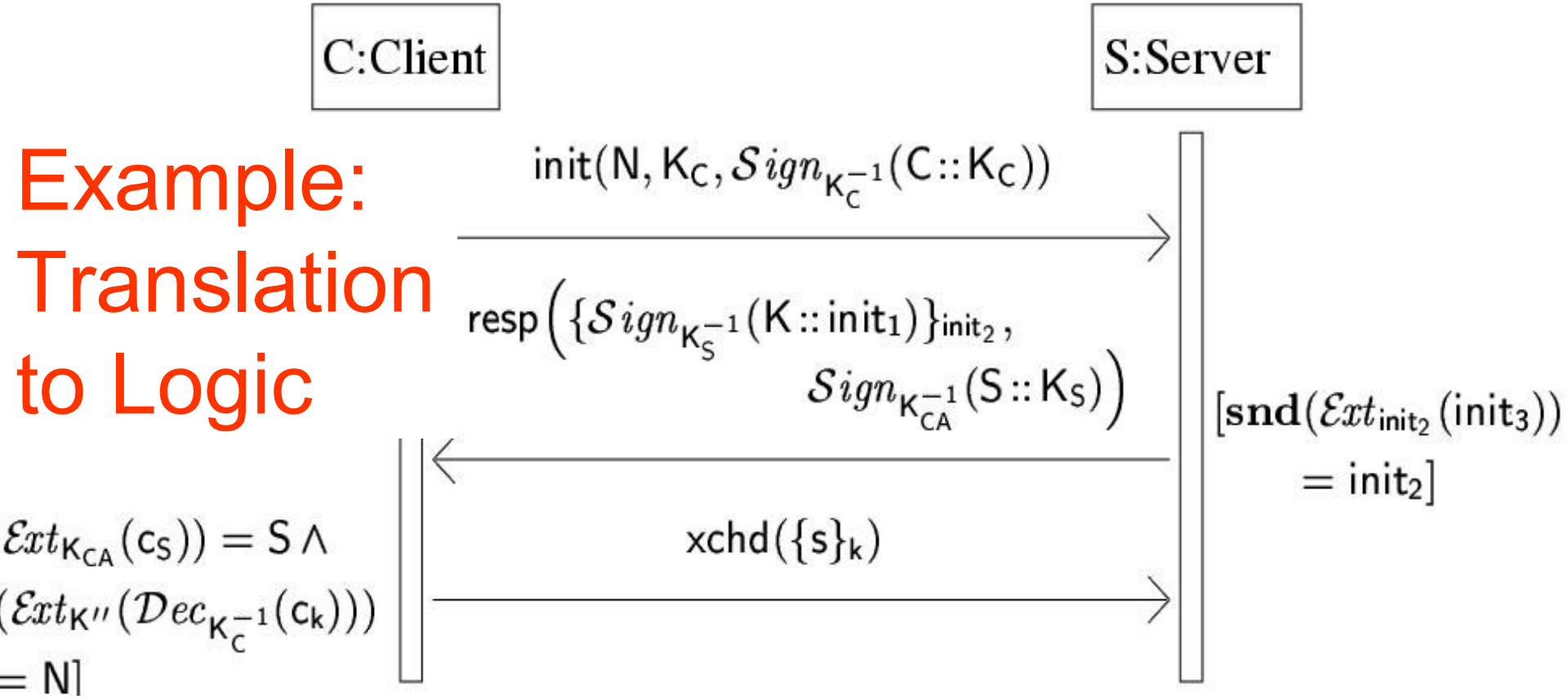
Security Analysis in First-order Logic

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]

Example: Translation to Logic



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

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_1
            & ( ? [X] :

      => (    kno

 & % c
```

```
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

```
                                  , inv(k_ca)), ArgC_1
                                  _k, DataC_n), inv(DataC
                                  11 )
      & (                         qual(sign(conc(s, DataC_ks), i
                                       ArgC_12 ) )
      & equal(                    gn(conc(DataC_k, n), inv(DataC_KK) )
              ArgC_11 )
      & equal(enc(sign(conc(DataC_k, DataC_n), inv(DataC
              ArgC_11 )
    )
   => ( knows(symenc(secret, DataC_k)) )  )
 ) )  )).
```

Static Model Verification

Static model verification works well if sufficient to check the model of a system (and not the physical asset).

- System model captures all relevant aspects about a system.
  - No problem, if code is generated from it, but in practice, this is not always the case.
- Assumptions about influences from a system's operational environment are reflected adequately.
  - Influences from the physical environment potentially infinitely many, e.g., think of a control system for a combustion engine.

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 !

Note: Almost no existing work (wrt crypto protocols).

Code Analysis vs. Model Analysis

Options:

- generate code from models
  ➔ currently not available in general
- generate models from code
  ➔ currently not available (for real-life software and right level of abstraction)

➔create models and code manually and verify code against models

➔This talk: run-time verification.

Run-time Verification

- Checking the actual behaviour of a system in its operational environment can be important.
- Need to validate the hidden assumptions that need to be made to reach a sufficient level for abstraction to enable automated static verification of non-trivial systems.

➔ Do run-time verification / monitoring.

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 (using run-time verification)

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

Implements?

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$_c$

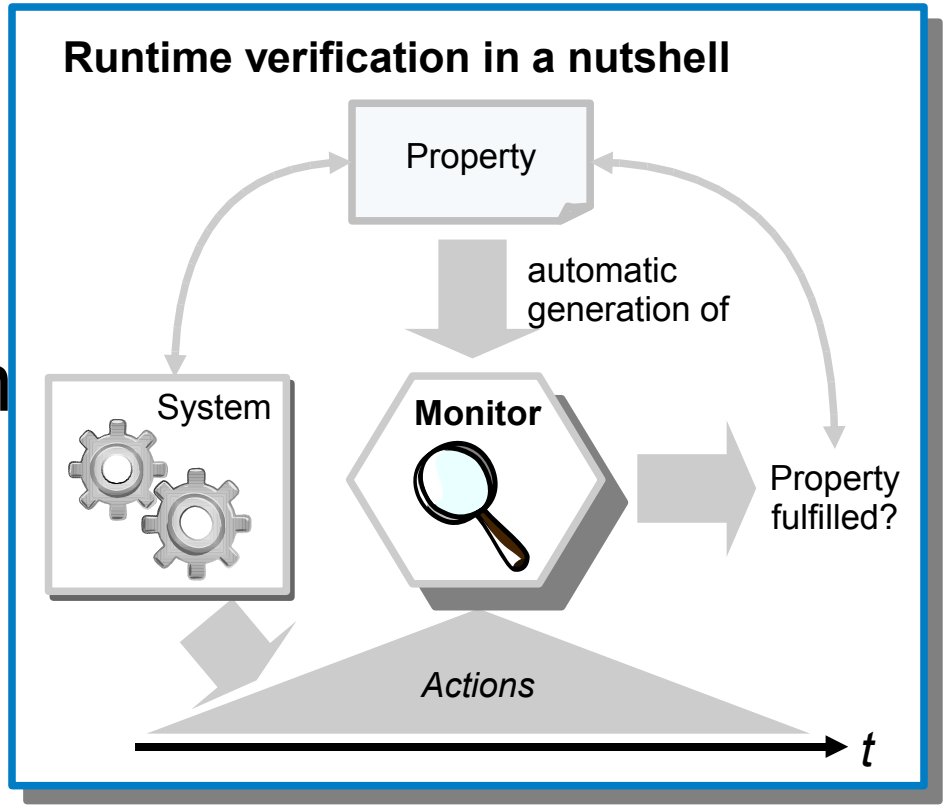
Abstract model

Runtime Verification using Monitors

Dynamic verification technique on the actual system.

Essentially a symbiosis of model-checking and testing.

“Lazy model-checking”: only check the system traces which are executed, when they are executed.

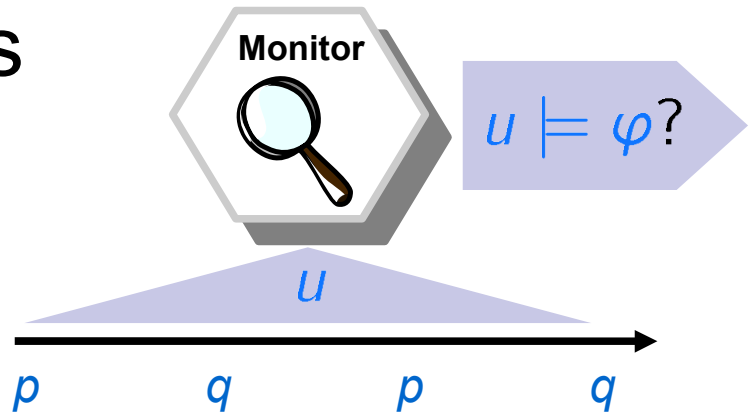


Formal underpinnings

- System (safety) property, φ specified in terms of linear time temporal logic [Pnu77]:

  $\varphi ::= true \mid p \mid \neg p \mid \varphi \; op \; \varphi \mid \varphi \mathbf{U} \varphi \mid \mathbf{X}\varphi \quad (p \in AP)$

- Continuous interpretation of φ over sequence of system events (behaviours), $u \in (2^{AP})^*$



- ***Automatic monitor generation:*** *"Inspired" by translation of LTL to Büchi-automata*

  $\varphi \rightarrow BA_\varphi \; s.t. \; L(BA_\varphi) = L(\varphi)$

Semantics

$$
\begin{array}{lcl}
w, i \models \mathit{true} & & \\
w, i \models \neg\varphi & \Leftrightarrow & w, i \not\models \varphi \\
w, i \models p \in AP & \Leftrightarrow & p \in w(i) \\
w, i \models \varphi_1 \vee \varphi_2 & \Leftrightarrow & w, i \models \varphi_1 \vee w, i \models \varphi_2 \\
w, i \models \varphi_1 \boldsymbol{U} \varphi_2 & \Leftrightarrow & \exists k \geq i.\ w, k \models \varphi_2 \wedge \\
 & & \forall i \leq l < k.\ w, l \models \varphi_1 \\
w, i \models \boldsymbol{X}\varphi & \Leftrightarrow & w, i+1 \models \varphi
\end{array}
$$

We write $w \models \varphi$, if and only if $w, 0 \models \varphi$, and use $w(i)$ to denote the ith element in w. (w word, i position)

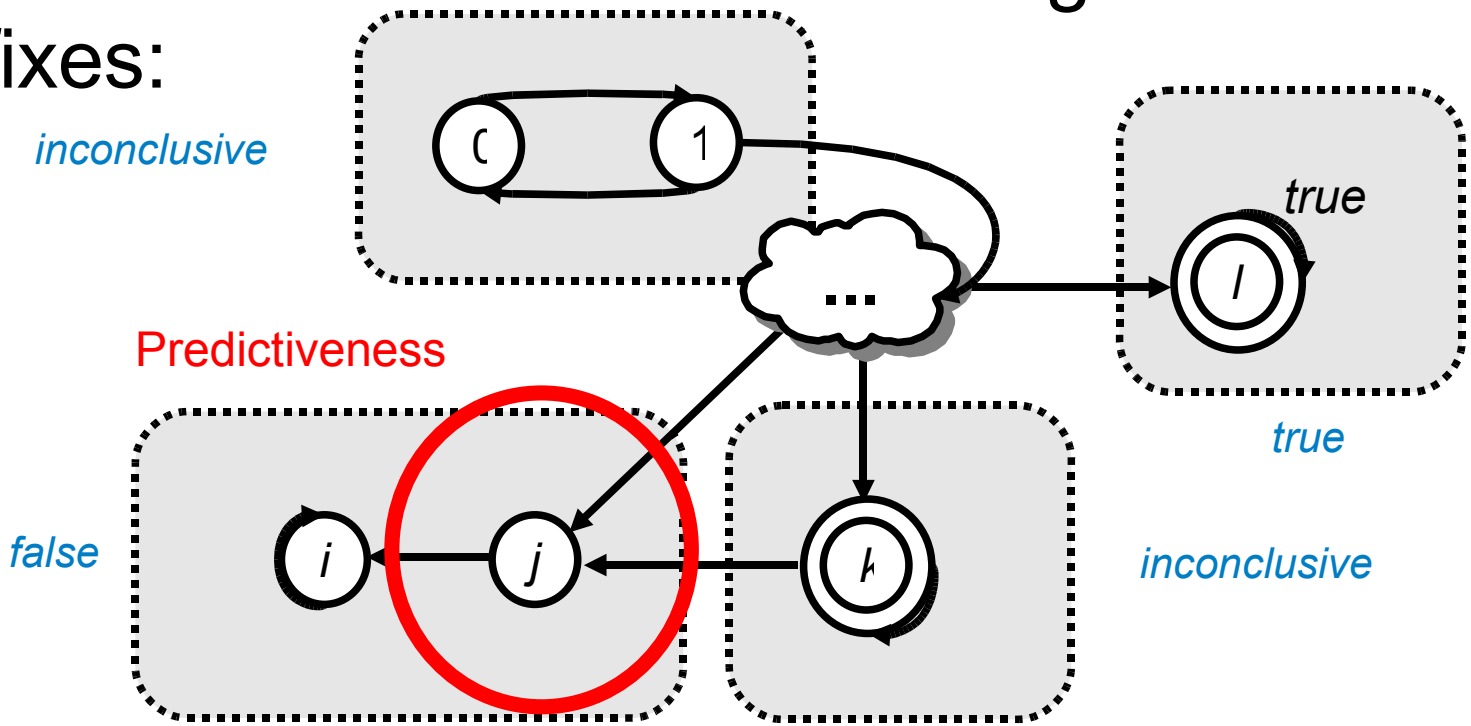
Write F phi for true U phi ("eventually phi"); G phi for not F not phi ("globally phi"); phi1 W phi2 for G phi1 or (phi1 U phi2) (weak-until)

Monitoring-friendly LTL semantics

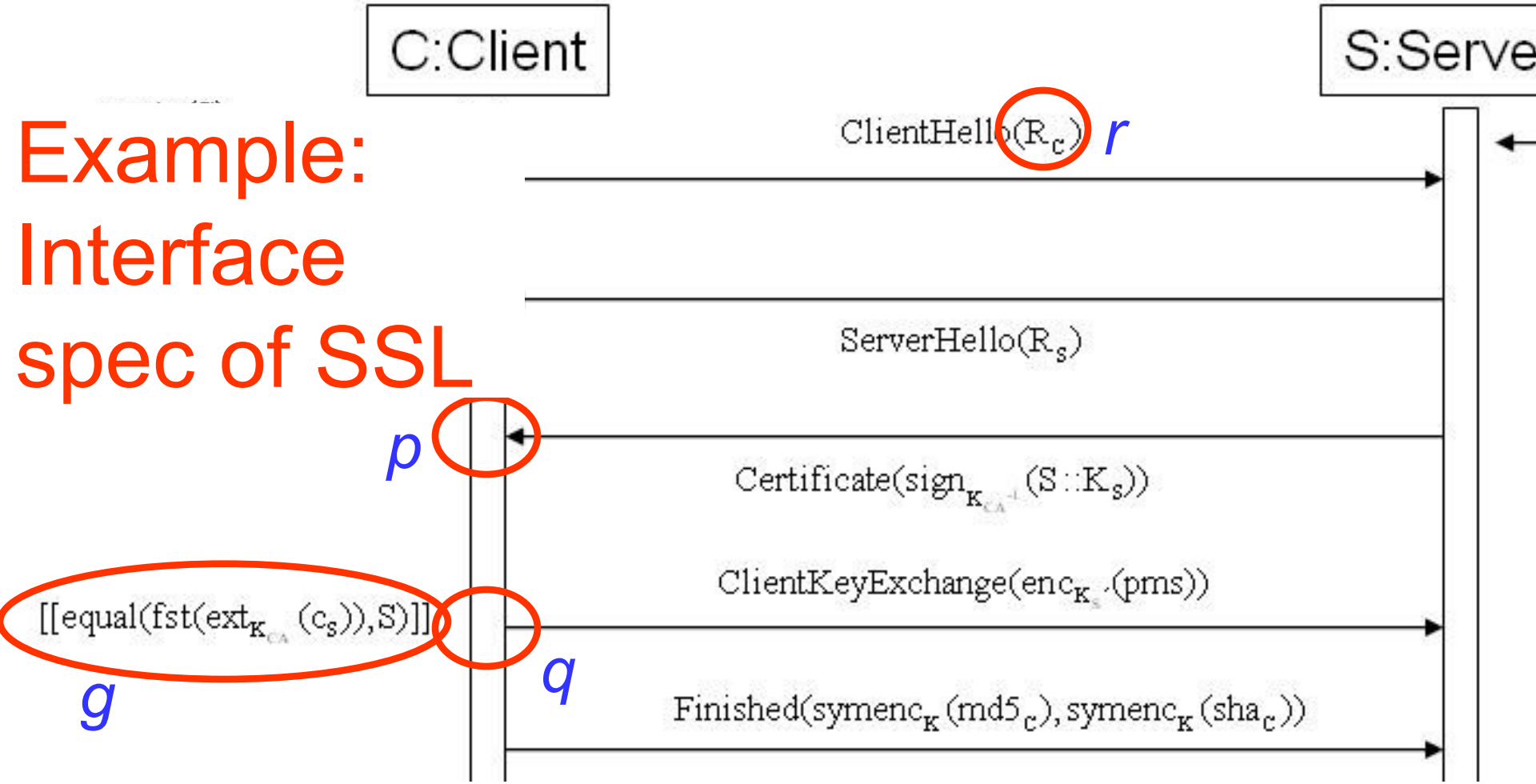
3-valued semantics:

$$[u \models \varphi] = \begin{cases} \top & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \models \varphi \\ \bot & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \not\models \varphi \\ ? & \text{otherwise} \end{cases}$$

Gives finite-state machines for detecting *minimal* bad prefixes:



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

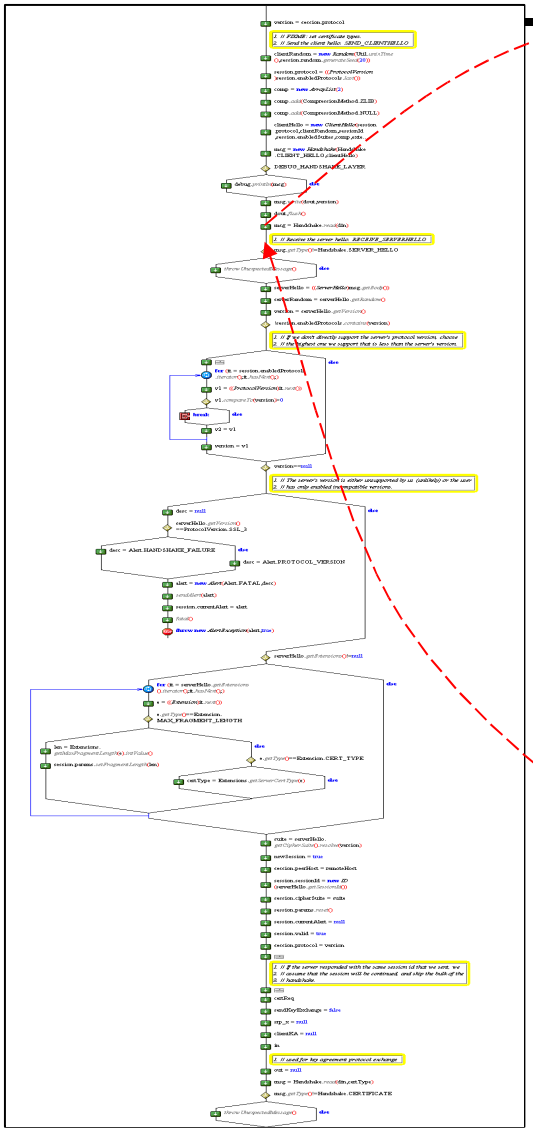
Random clientRandom =
new Random(...,**session.random.generateSeed(28)**);

„meaning“

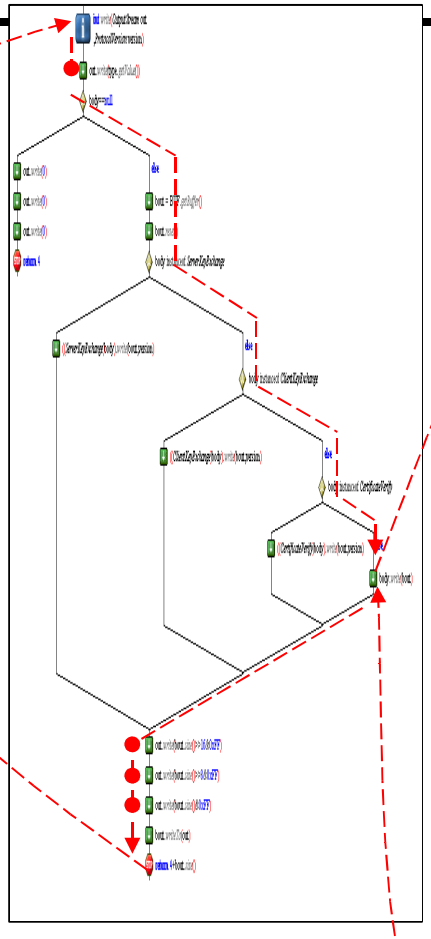
class SecureRandom (specified in: FIPS 140-2,RFC 1750) of package java.security
Function: generateSeed

Sending Messages

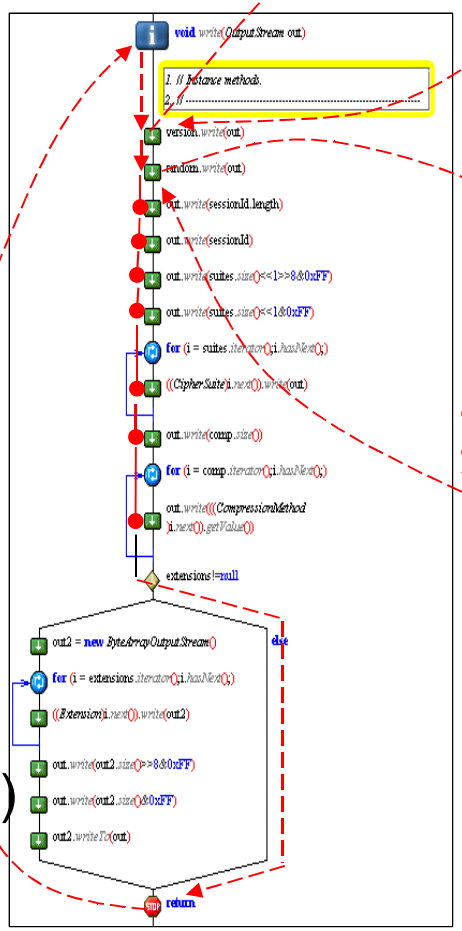
Automate this using patterns



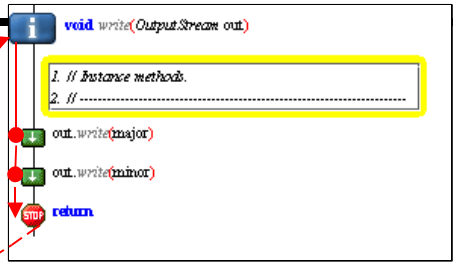
SSLSocket.doClientHandshake()



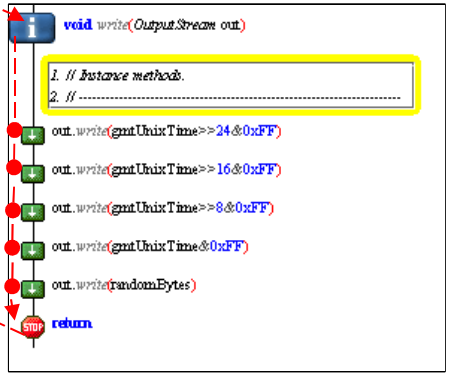
Handshake.write()



ClientHello.write()



ProtocolVersion.write()

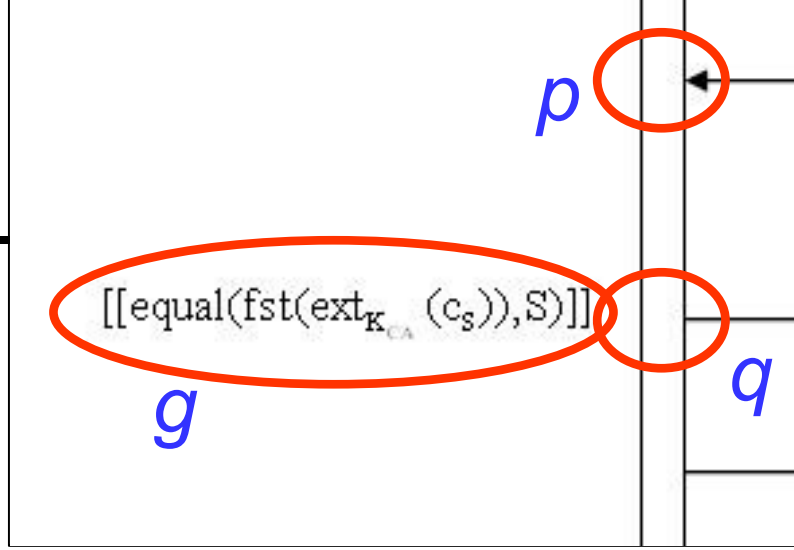


Random.write()

traverse CFG

call of OutputStream.write()

Checking Guards



Guard *g* enforced by code?

Here: Generate runtime-verification checks for *g* at *q* from diagram.

[Alternatives: Testing against checks (symbolic crypto for inequalities); [ICFEM02]
Automated formal local verification: conditionals between *p* and *q* logically imply *g* (using ATP for FOL).] [ASE05,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()

java.security.Signature

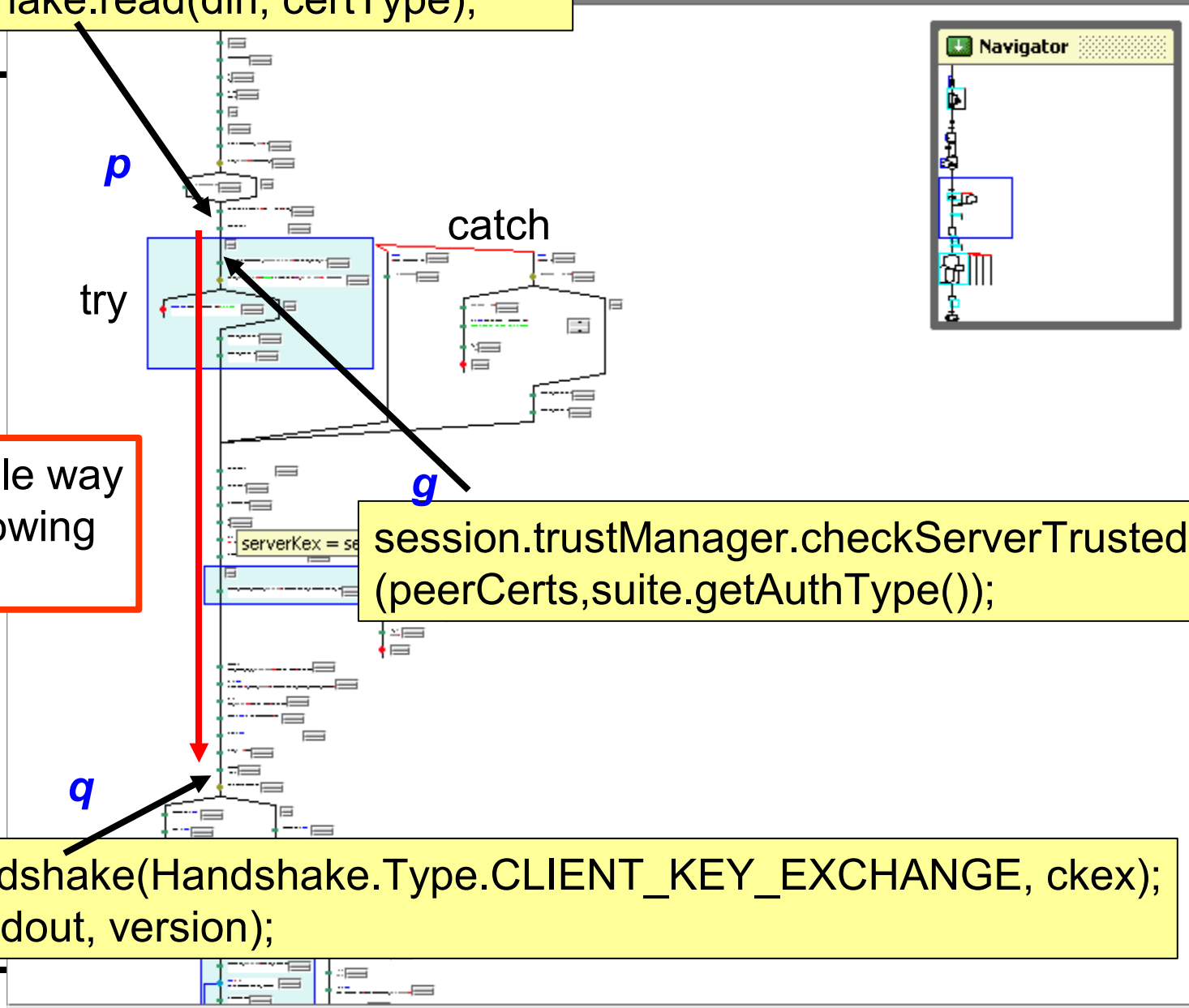
- **Initializatize**
- **Update**
- **Verify**

„verifies 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);



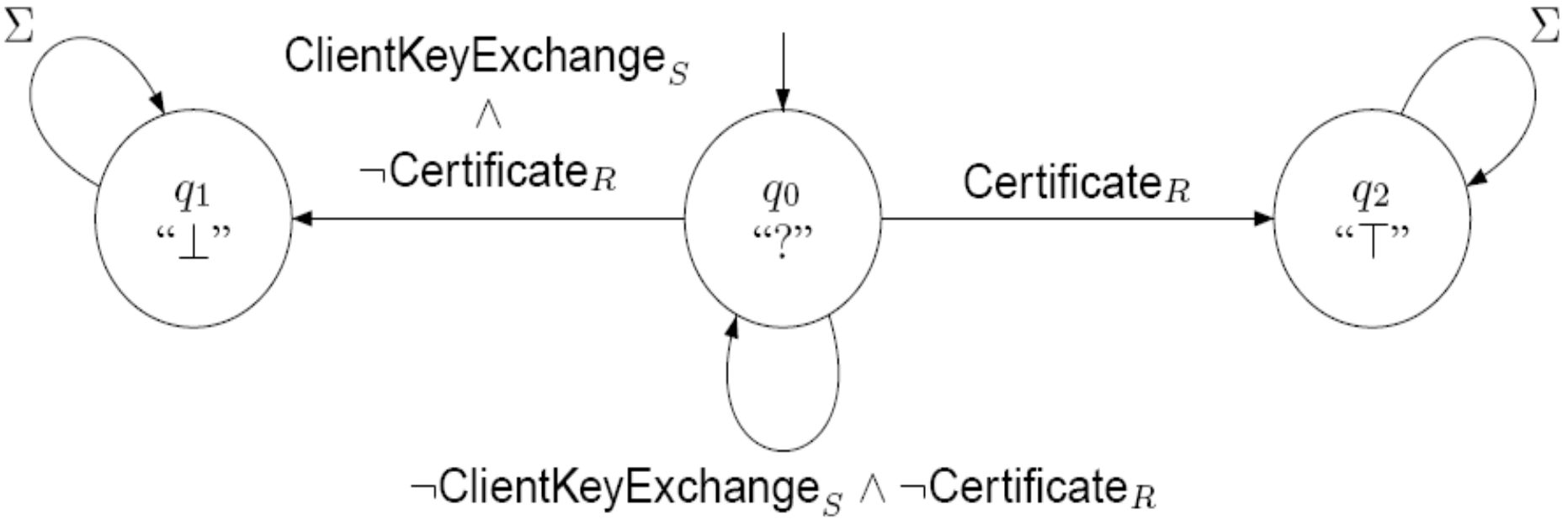
only possible way without throwing exception

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

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

ClientKeyExchange

Client will not send out ClientKeyExchange message until has received Certificate message, and validity check is positive. If that is the case, client will send out ClientKeyExchange message eventually.



not safety but co-safety

Figure 1: FSM ¬ClientKeyExchange$_S$ UCertificate$_R$.

Server Finished

Server will not send Finished message before has checked that MD5 received in Client`s Finished message is equal to MD5 created by server, and correspondingly for SHA hash. Will send it out eventually after that has been established.

$$\varphi_2 = (\neg \mathsf{Finished}_S \mathbf{W}(MD5(\mathsf{Finished}_R) = MD5(\mathsf{Finished}_S))) \wedge (\mathbf{F}(MD5(\mathsf{Finished}_R) = MD5(\mathsf{Finished}_S)) \Rightarrow \mathbf{F}\mathsf{Finished}_S),$$

(and similar for SHA).

neither safety nor co-safety

Client Transport Data

Client will not send any transport data before has checked that MD5 hash received in Server`s Finished message is equal to MD5 created by Client (and correspondingly for SHA hash).

$$\varphi_3 = \neg Data\mathbf{W}((MD5(\mathsf{Finished}_R) = MD5(\mathsf{Finished}_S)),$$

not co-safety but safety

Conclusion

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

Integrated with static verification of UMLsec models.

Exceeds previous approaches such as Fred Schneider's security automata in expressivity.

Run-time verification approach used optimally efficient (regarding space complexity).

Questions ?

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

J.Jurjens@open.ac.uk