Security Analysis of Crypto-based Java Programs using Automated Theorem Provers

Jan Jürjens

Competence Center for IT Security

Software & Systems Engineering

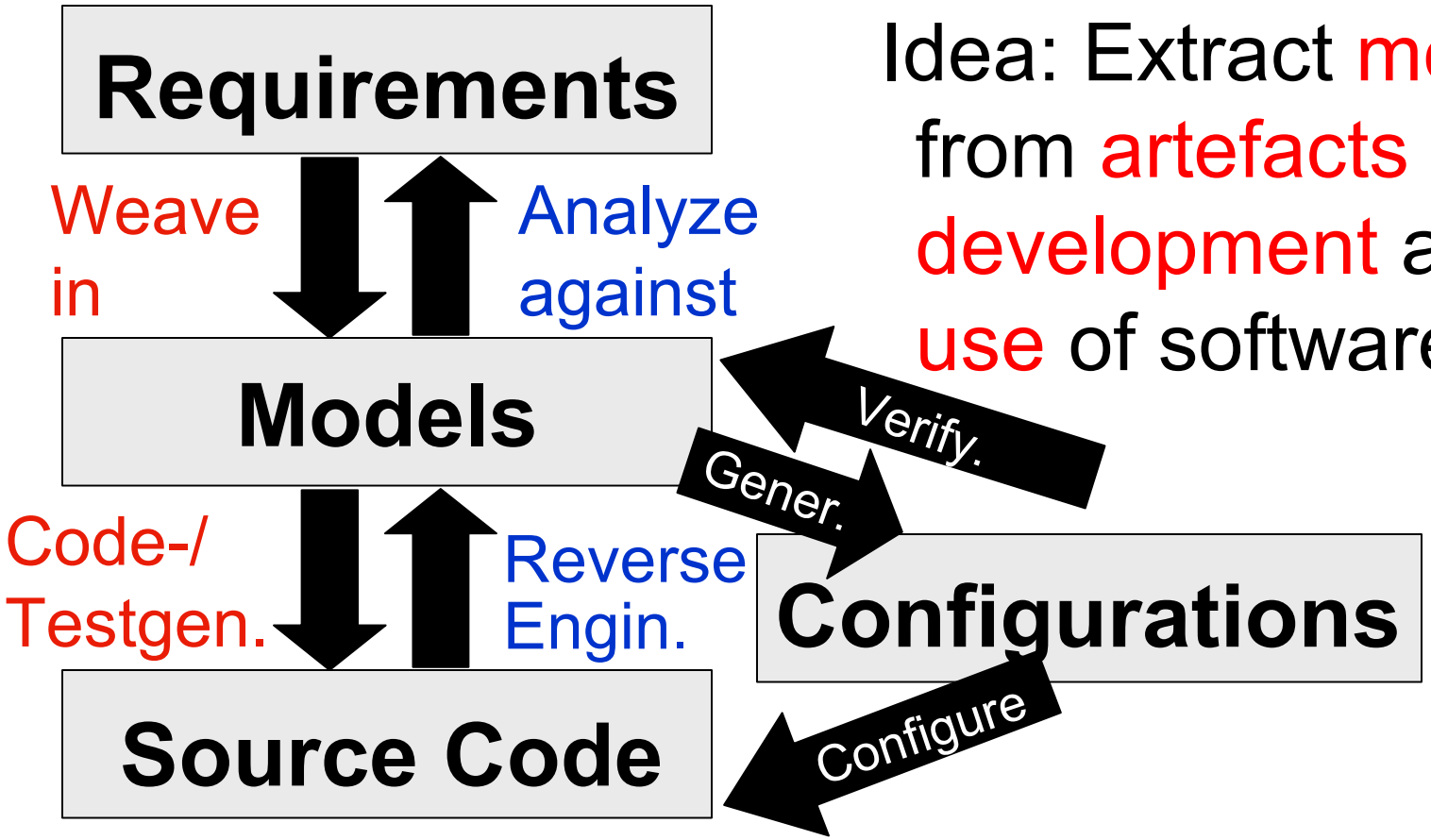
Informatics, Technical University Munich

from 1 Oct: Open University, GB

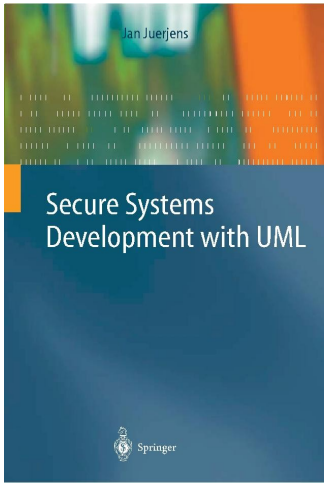
juerjens@in.tum.de

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

Model-based Security Engineering



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



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

Security Analysis: Model or Code ?

Model:

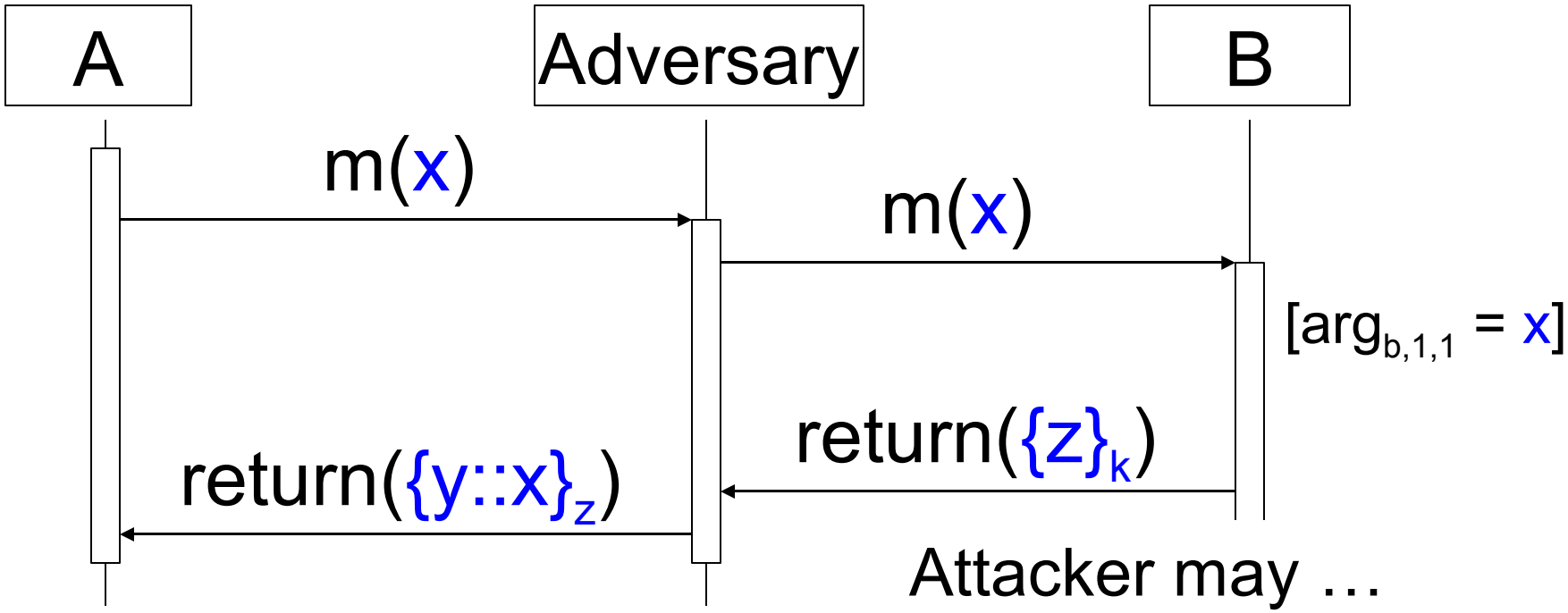
+ earlier (less expensive to fix flaws)
+ more abstract ➔ more efficient
- more abstract ➔ may miss attacks
- programmers may introduce security flaws
- even code generators, if not formally verified

Code:

+ „the real thing“ (which is executed)

➔ Do both where feasible !

Crypto-based Software (e.g. Protocols)



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 automatically generated formulas using automatic theorem prover.

First-order Logic: Basic Rules

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

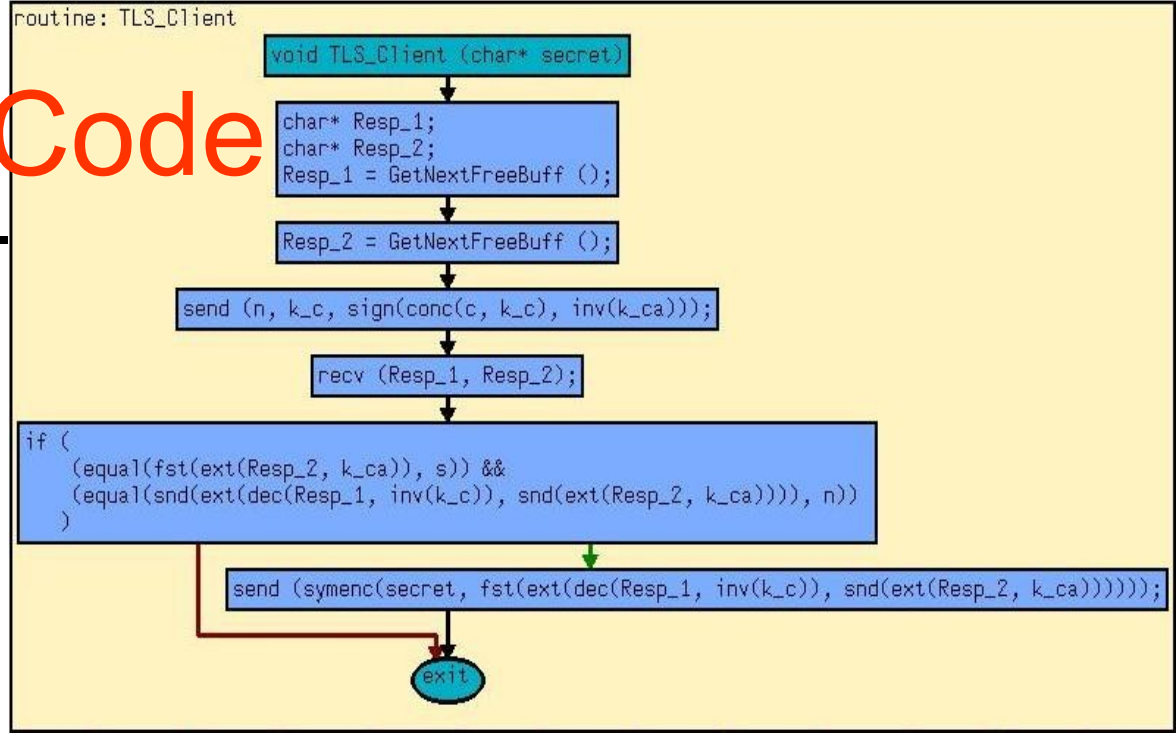
Define cryptosystem. E.g.: $Dec_{K^{-1}}(\{E\}_K)=E$

For evolving adversary knowledge define

$$\forall E_1,E_2.(knows(E_1) \wedge knows(E_2) \Rightarrow knows(\{E_1\}_{E_2}) \wedge knows(Dec_{E_2}(E_1)) \wedge \ldots)$$

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.

… Translate to First 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)Æ cond(msg_in)*

$\Rightarrow$*knows(msg_out)*

Æ PRED(TR2)]

Verify with automated FOL prover (e-SETHEO, SPASS, …), e.g. whether can derive *knows(s)* from generated formulas.

Example: TLS Variant

«data security»

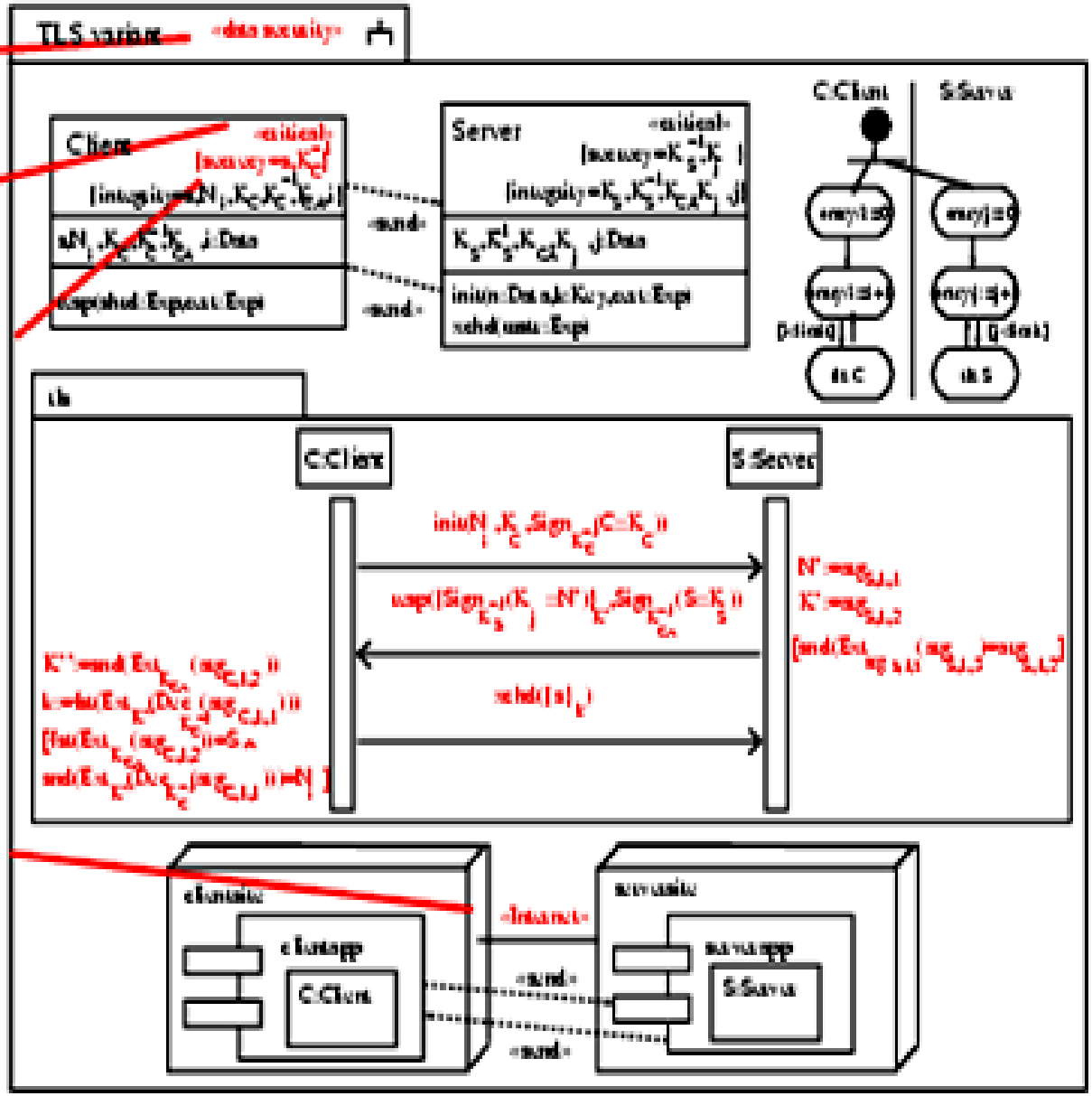
Presented at IEEE Infocom 1999

«critical»

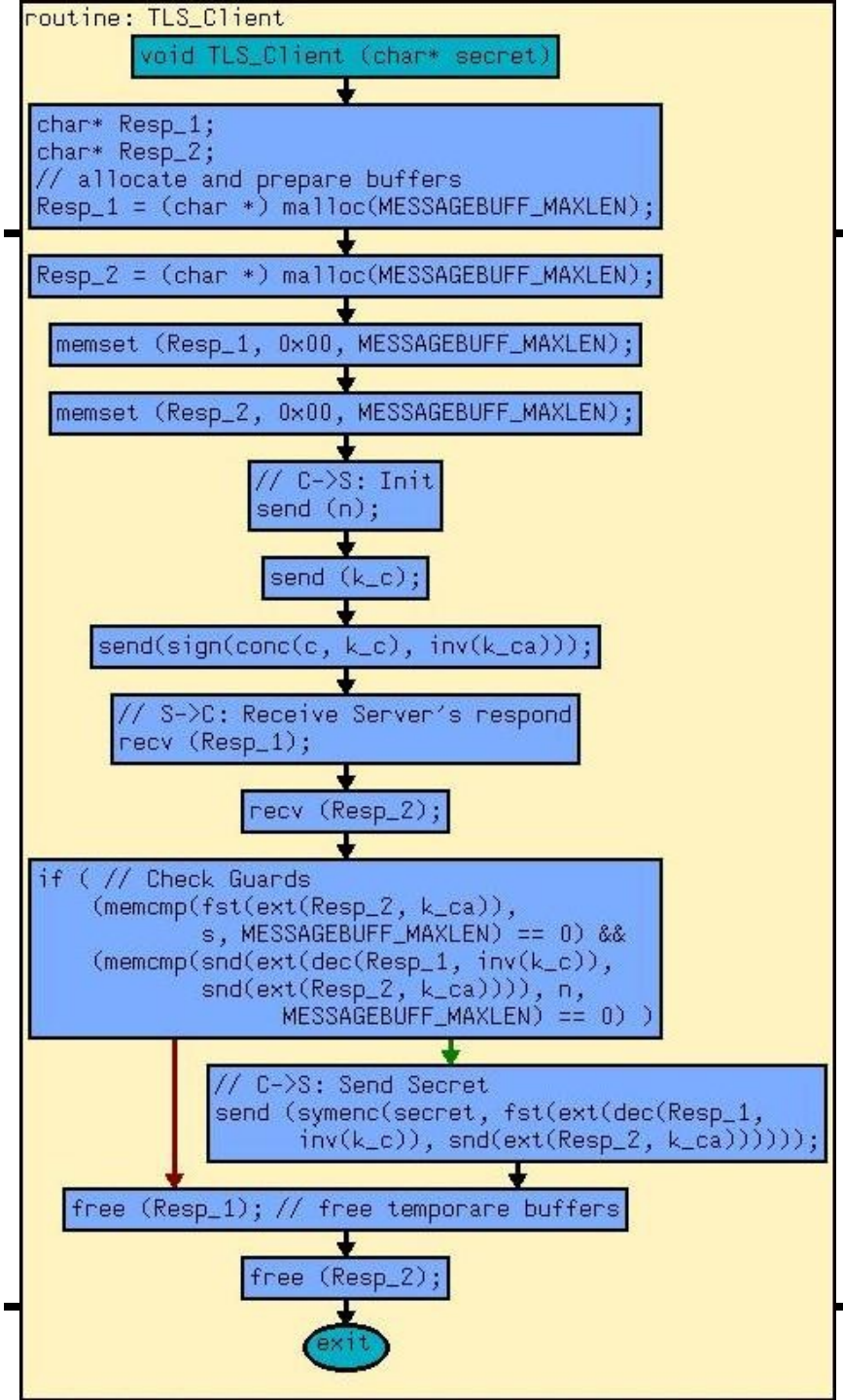
$\{secrecy = \{s, K_C^{-1}\}\}$

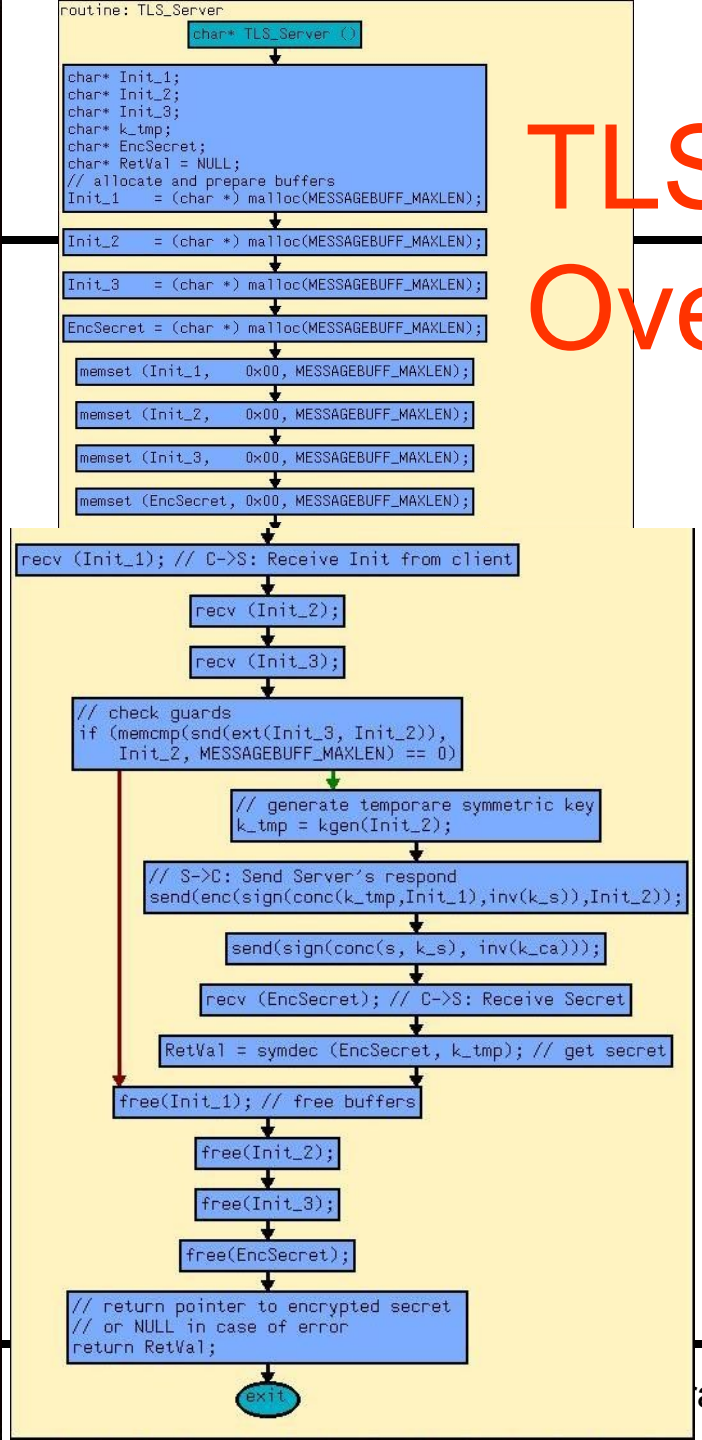
Goal: send secret protected by session key using fewer server resources.

«Internet»

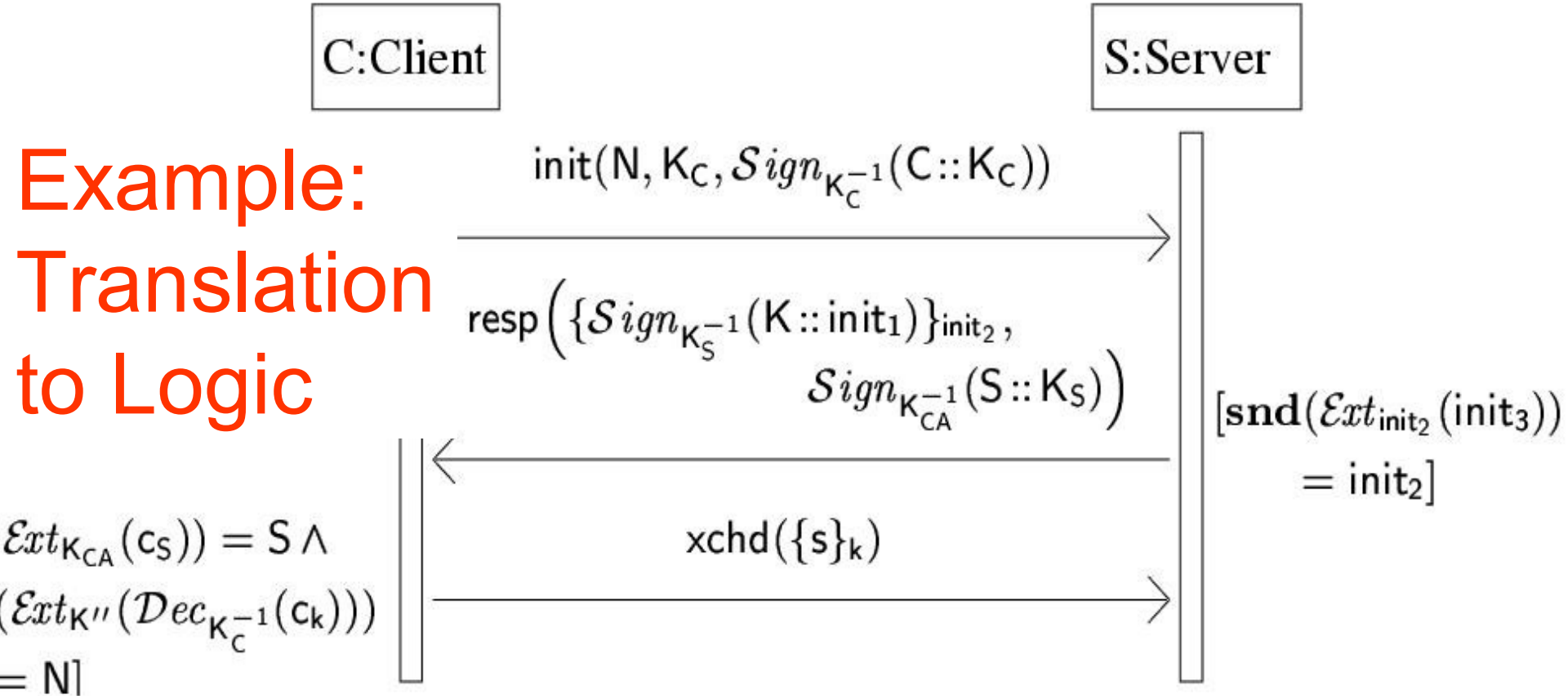


TLS Overview





Example: Translation to Logic



$$knows(N) \text{Æ} knows(K_C) \text{Æ} knows(Sign_{K_C^{-1}}(C::K_C))$$

$$\text{Æ}\ \forall\, init_1, init_2, init_3.[knows(init_1) \text{Æ}\ knows(init_2) \text{Æ}$$

$$knows(init_3)\ \text{Æ}\ snd(Ext_{init_2}(init_3)) = init_2$$

$$\Rightarrow knows(\{Sign_{K_S^{-1}}(\ldots)\}_{\ldots}) \text{Æ} [knows(Sign\ldots)]$$

$$\text{Æ} \forall\, resp_1, resp_2\, [\quad \Rightarrow \quad]]$$

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)) )  )
 ) )  )).
```

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.

Experiences

Since consider bound to adversary knowledge, may get false positives (but not too many in our practical experience).

Security verification problem theoretically undecidable, but reasonably practically decidable.

Challenge: handle large CFGs generated by CASE tools.

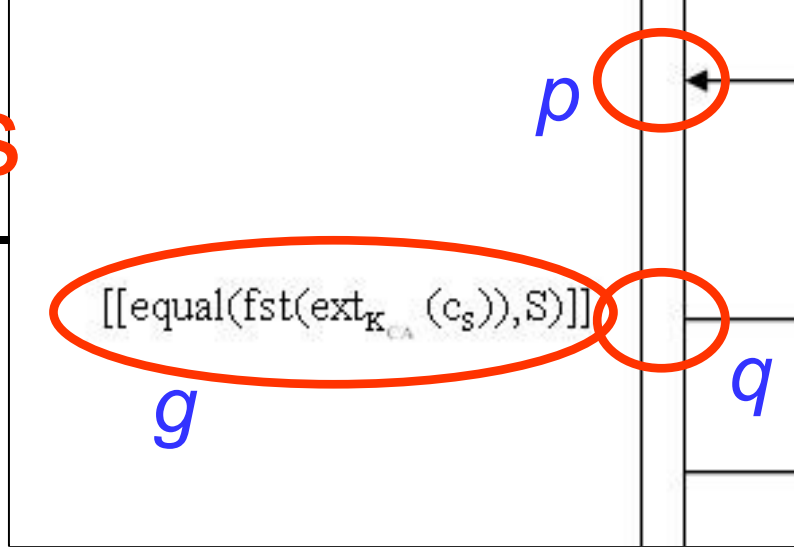
Proposed solution: abstraction using annotations.

Abstraction by Code Annotations

```
//@J2SD_ANN (<<method name>>)
//@J2SD_CONN (<<trigger>>; <<guard>>;
   <<effect>>)
//@J2SD_INSERT (<<value>>)
//@J2SD_AXIOMS (<<value>>)
// <<FOL axioms>>
//@J2SD_AXIOMS_END
```

Similarly for variables / constants.

Validating Assertions



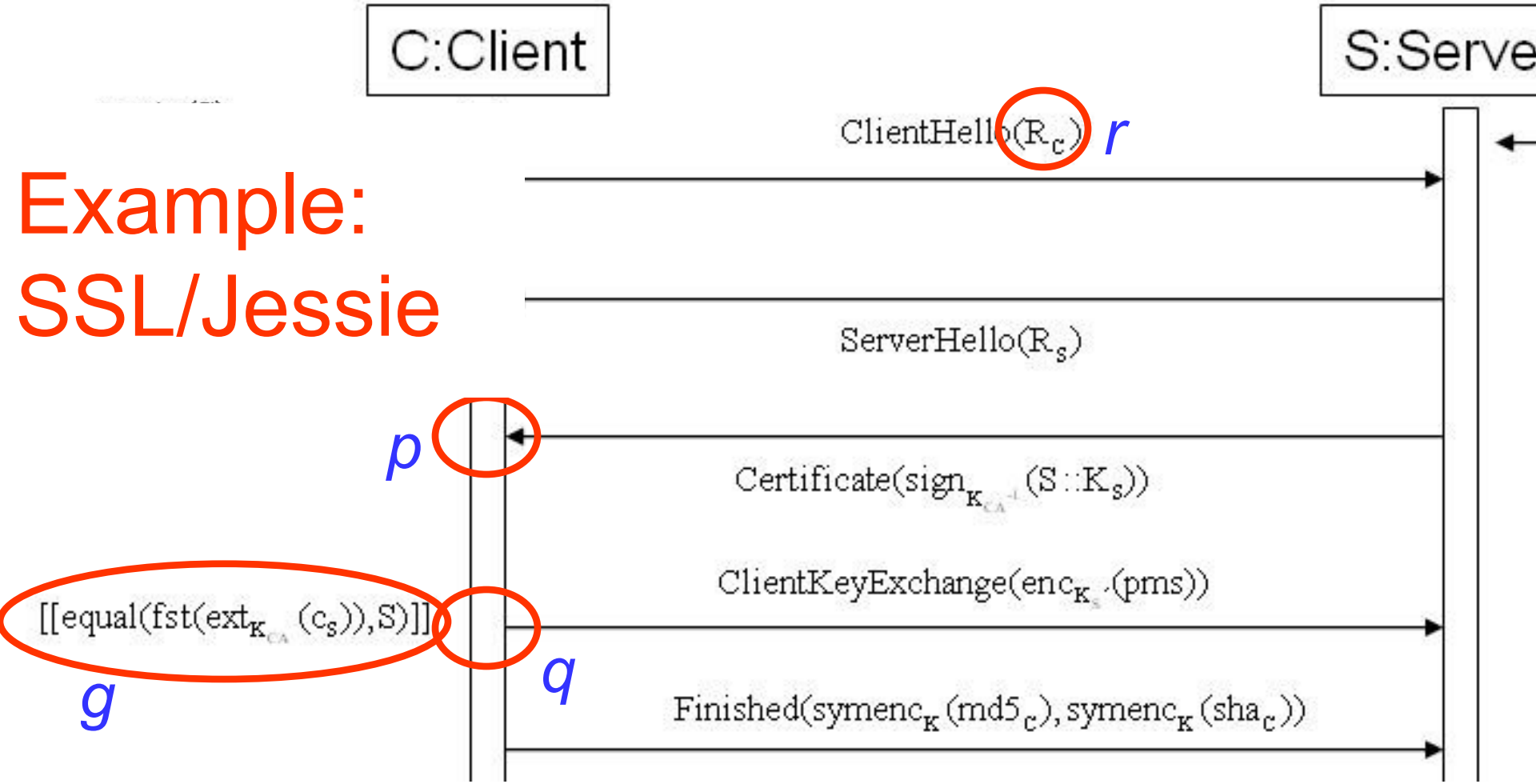
Guard *g* enforced by code?

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

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

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

Example: SSL/Jessie



I) Identify program points:

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

II) Check guards enforced

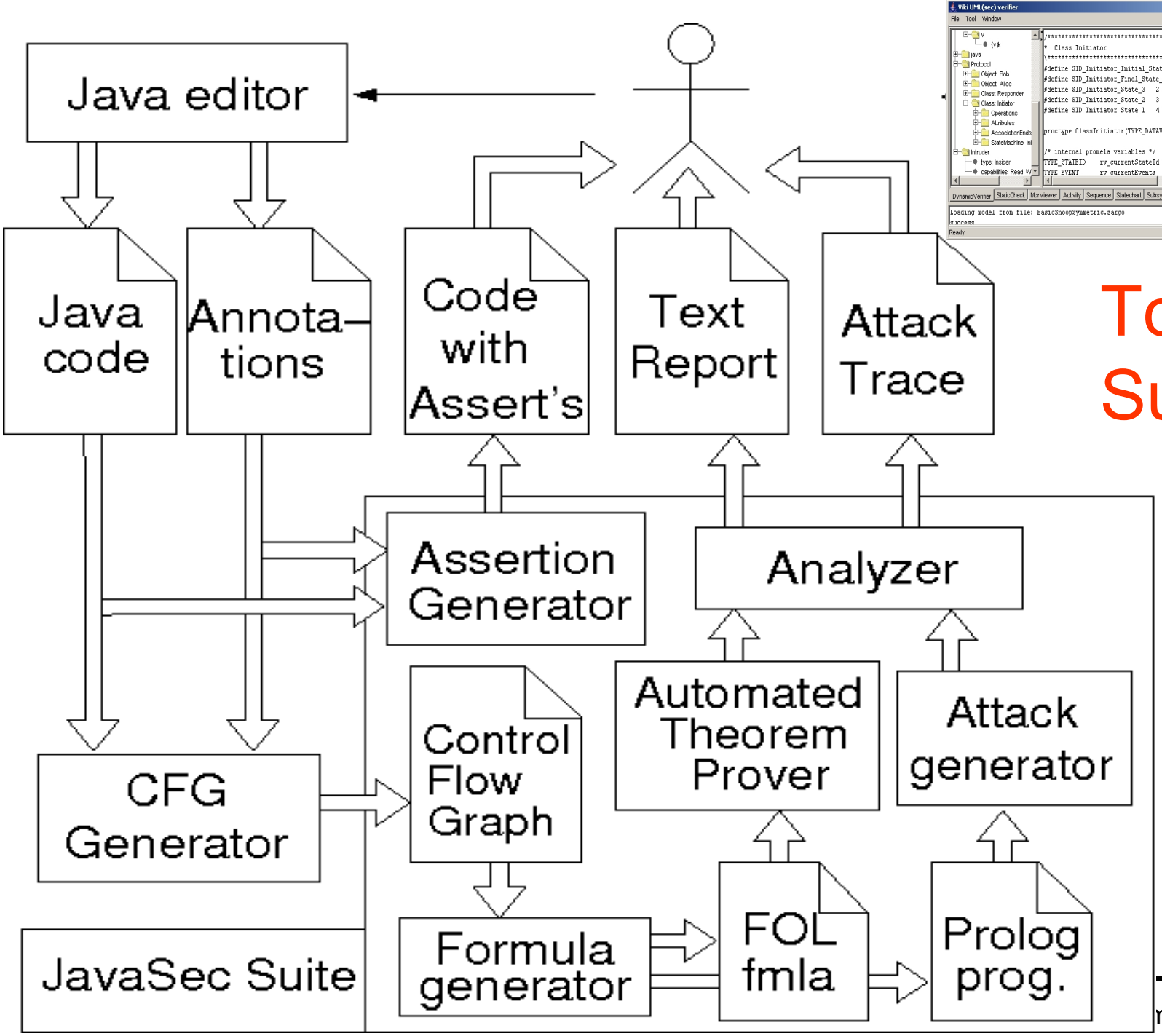
Modular Verification

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

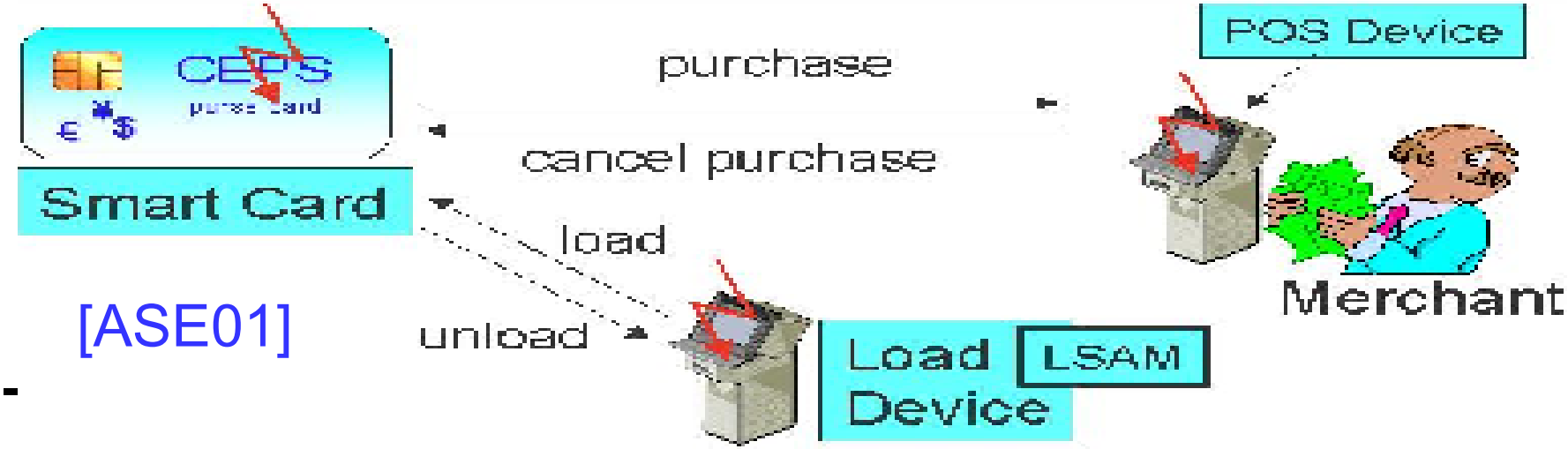
When considering single protocol run, can construct finite set of such statements similar to FOL formulas from security analysis.

Tool Support



Common Electronic Purse Specifications

Global elec. purse standard (Visa, 90% market).
Smart card contains account balance, performs crypto operations securing each transaction.
Formal analysis of load and purchase protocols: three significant weaknesses: purchase redirection, fraud bank vs. load device owner.



[ASE01]

Conclusion

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

Automated and with sufficient efficiency due to abstraction tailored to verification problem.

Challenge: size of CFGs automatically generated by CASE tools.

Adress this using annotations.

Current work: automatically generate annotations from UML models.

Questions ?

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

http://www.umlsec.org

from 1 Oct: Open University, UK

Applications of MBSE

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.

Model vs. Implementation

[with David Kirscheneder]

„meaning“

„meaning“

compare meaning!

Backtrace assignments

Defined during model creation

Sent and received data

Elements of connections

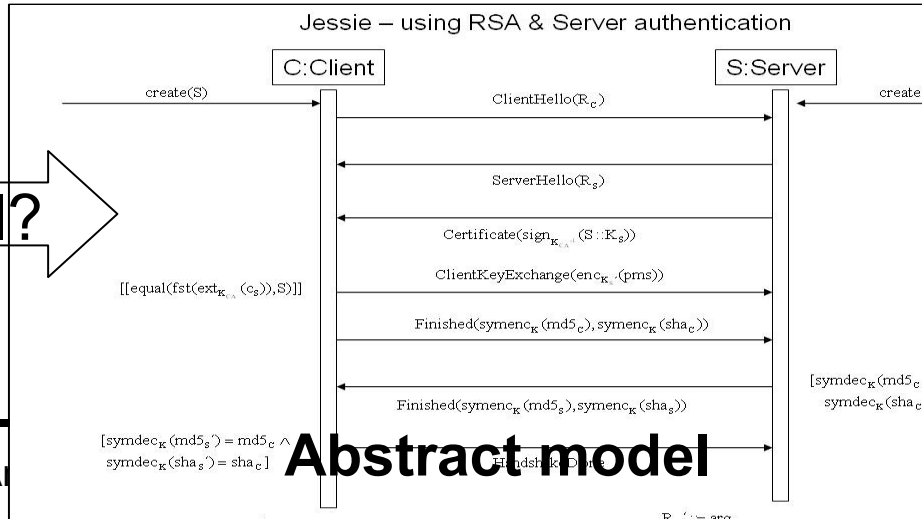
Find

Has

Implement -ation

.java

Equal?



Abstract model

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

Implementation (Jessie): Identify Values

Currently do this manually using code assertions

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

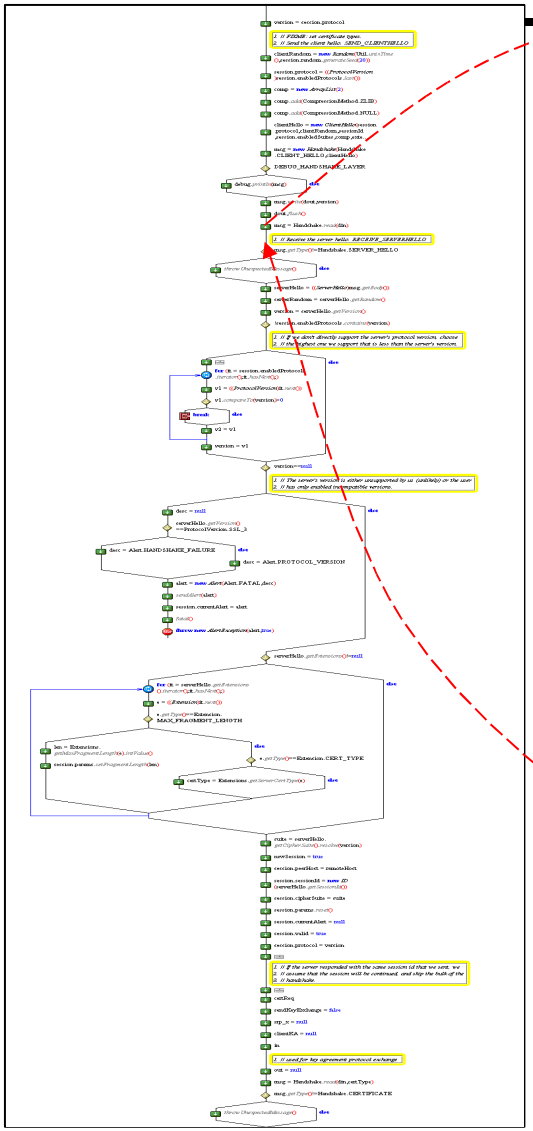
Random clientRandom =
new Random(...,**session.random.generateSeed(28)**);

„meaning“

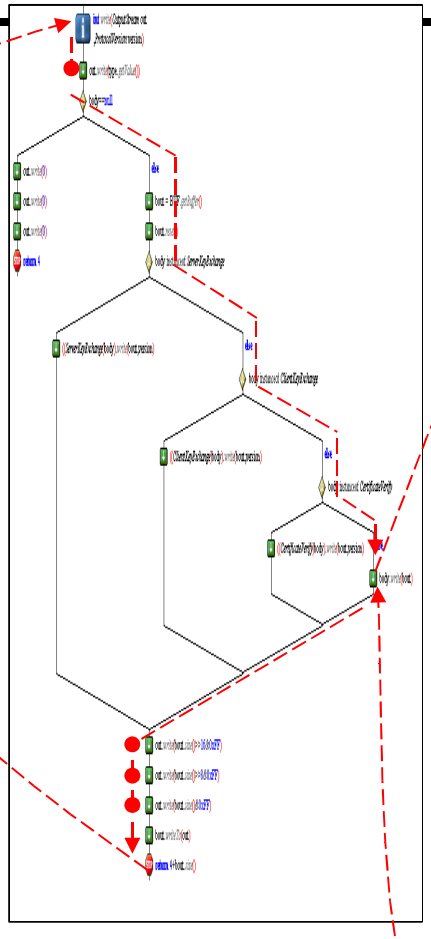
class SecureRandom (specified in: FIPS 140-2,RFC 1750) of package java.security
Function: generateSeed

Sending Messages

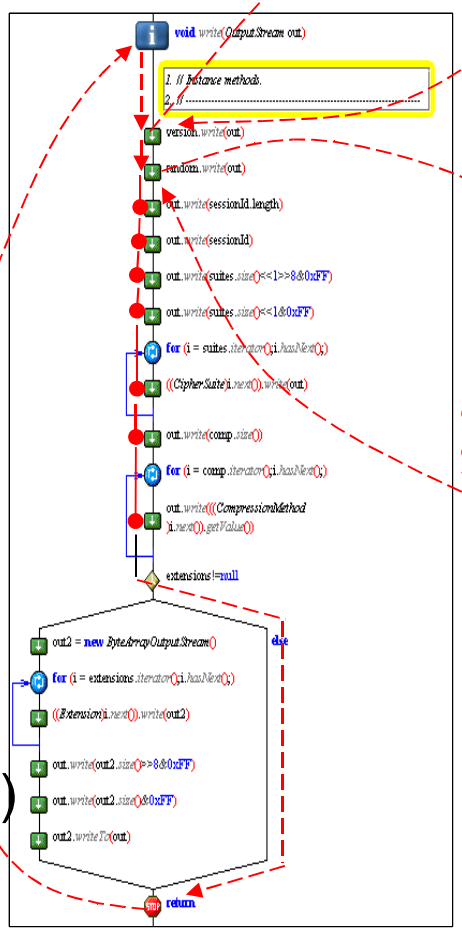
Automate this using patterns



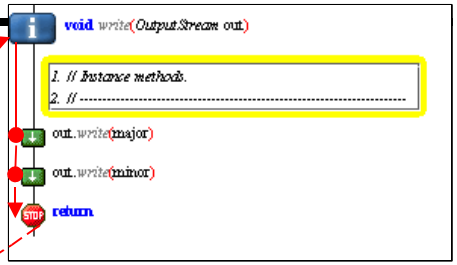
SSLSocket.doClientHandshake()



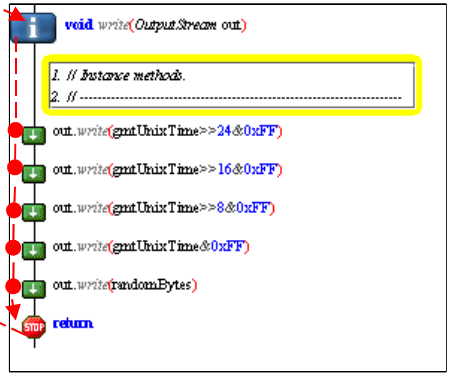
Handshake.write()



ClientHello.write()



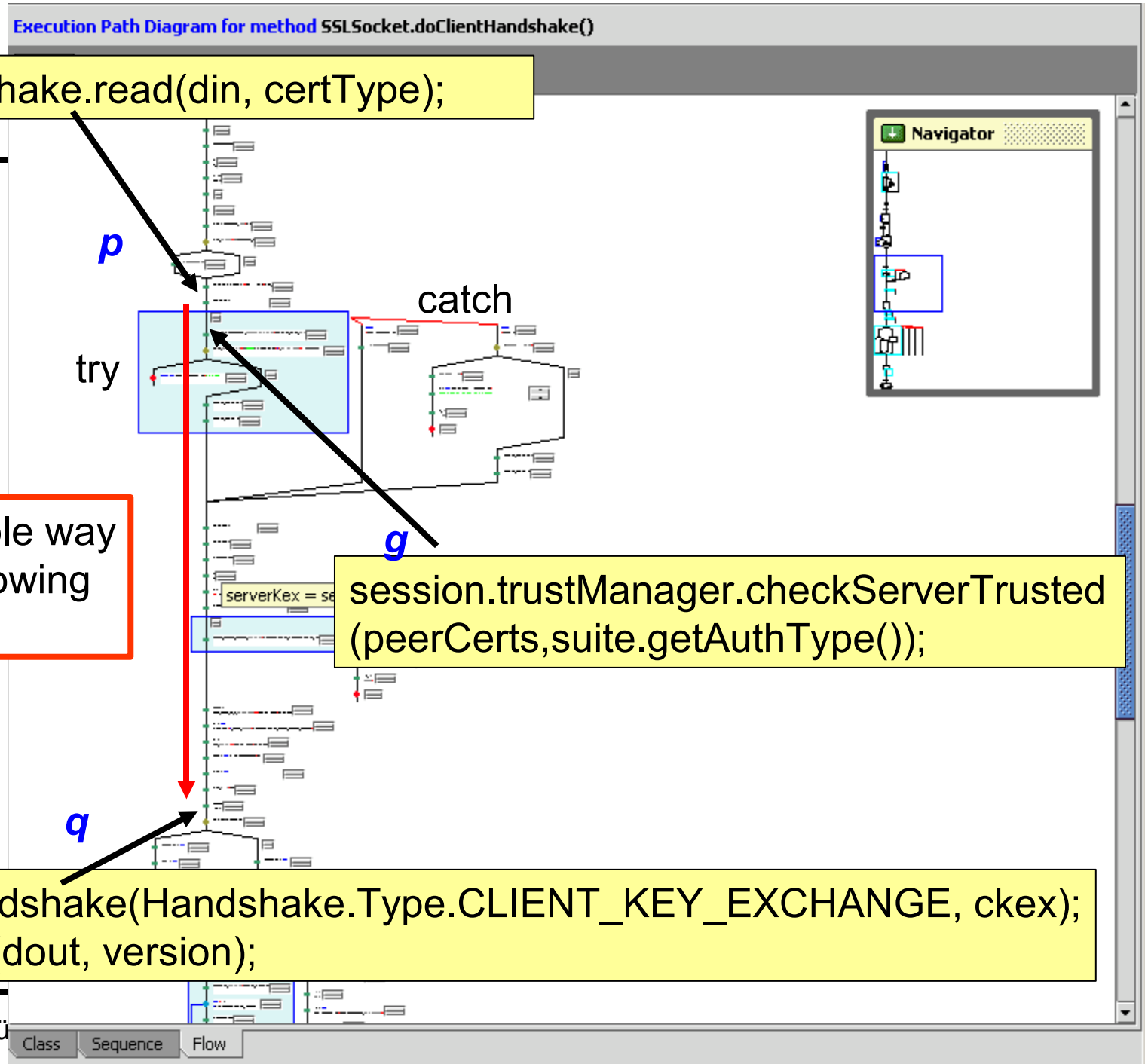
ProtocolVersion.write()



Random.write()

traverse CFG

call of OutputStream.write()


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

Man-in-the-Middle Attack

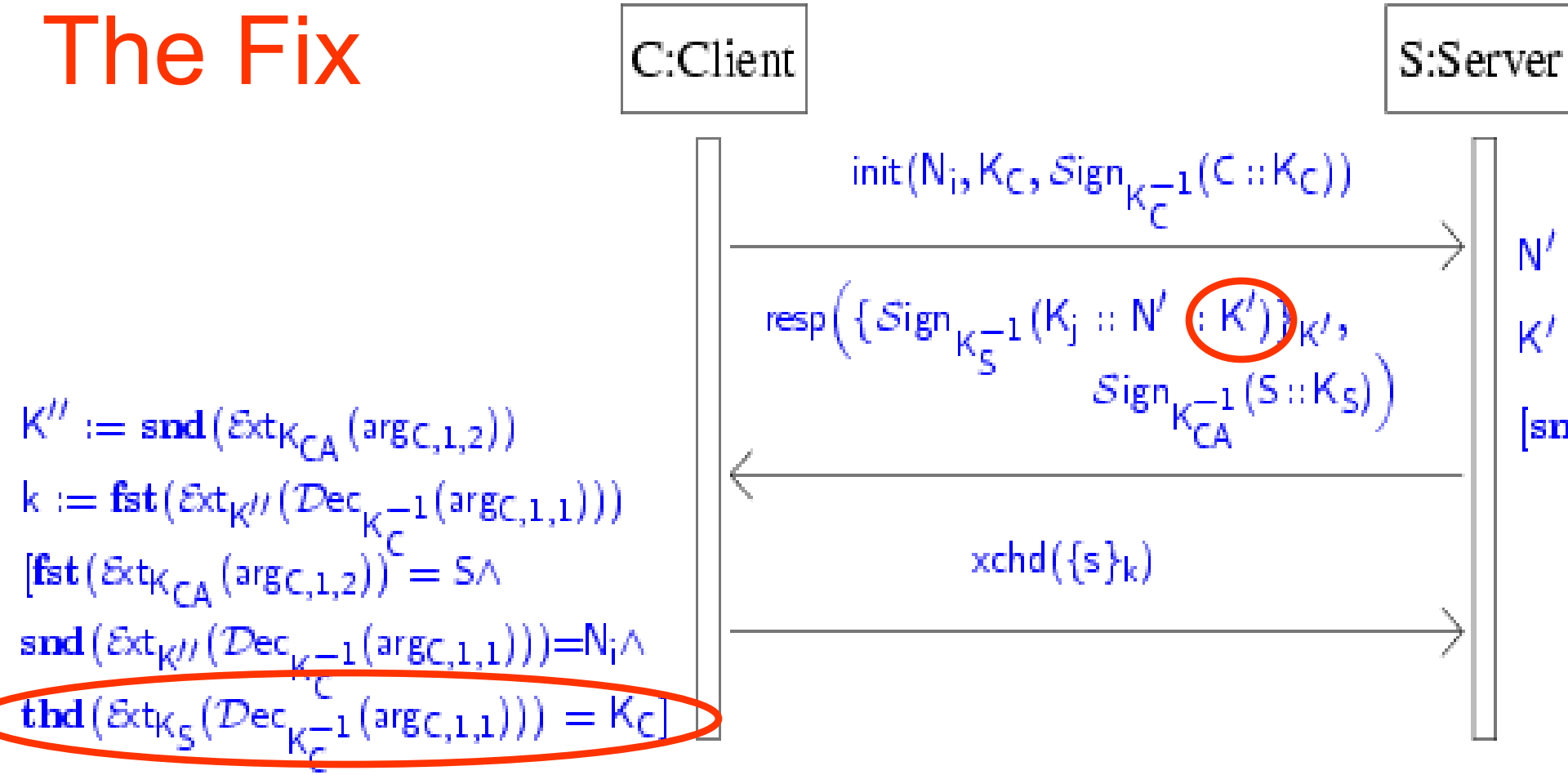
$$C \xrightarrow{N_i::K_C::Sign_{K_C^{-1}}(C::K_C)} A \xrightarrow{N_i::K_A::Sign_{K_A^{-1}}(C::K_A)} S$$

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

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

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

The Fix



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

Note completeness of FOL (but also undecidability).

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

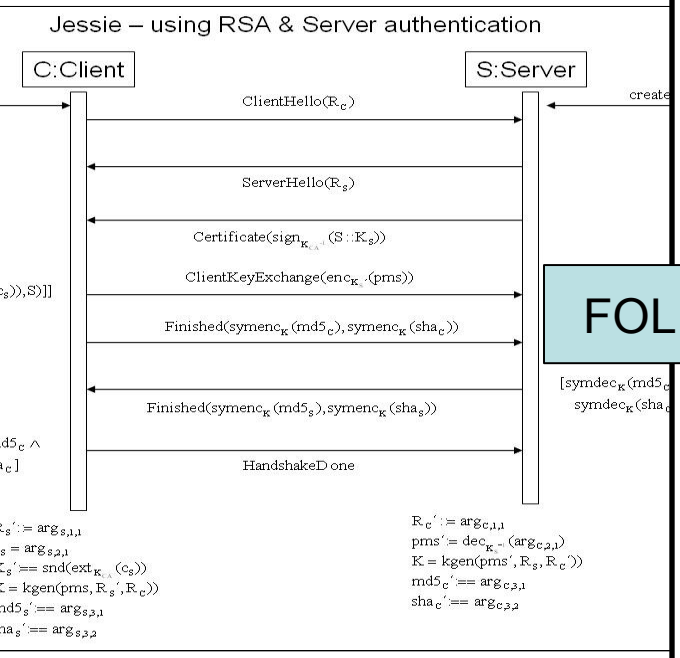
$[d]\ p \models g$: g checked in any execution of p initially satisfying d before any send

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

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

Model Verification

$$\forall arg_1, \ldots, arg_n \;.\; (knows(arg_1) \wedge \ldots \wedge knows(arg_n) \wedge cond(arg_1, \ldots, arg_n) \Rightarrow knows(exp(arg_1, \ldots, arg_n)))$$



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
```

[FASE05,ICSE05, ICSE06]

Biometric Authentication System

In development by company in joint project.

Uses bio-reference template on smart-card.

Analyze given UML spec.

Discovered three major weaknesses in subsequently improved versions (misuse counter circumvented by dropping / replaying messages, smart-card insufficiently authenticated by mixing sessions). [ACSAC05]

Bank Application

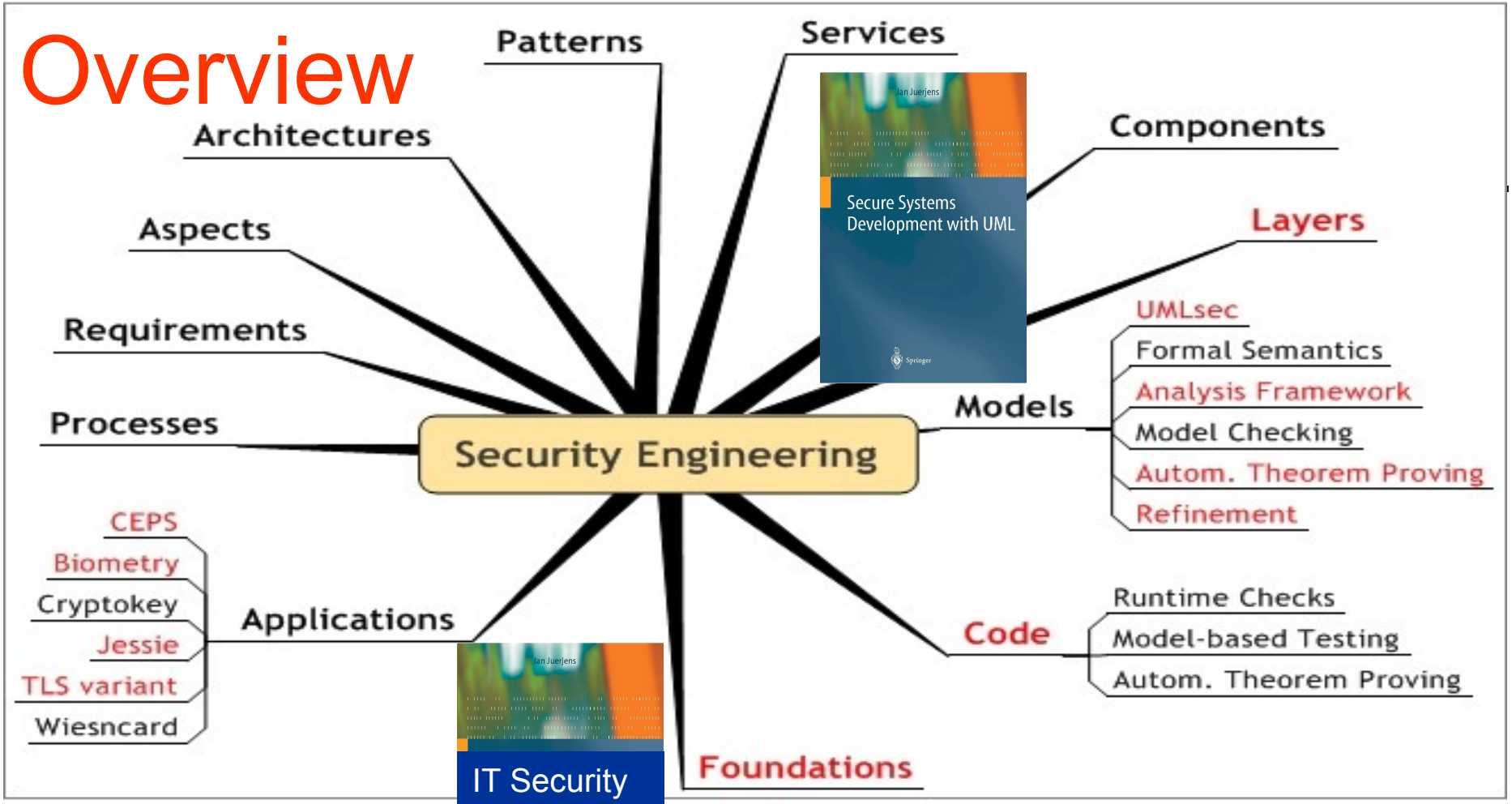
Security analysis of web-based banking application, to be put to commercial use (clients fill out and sign digital order forms).

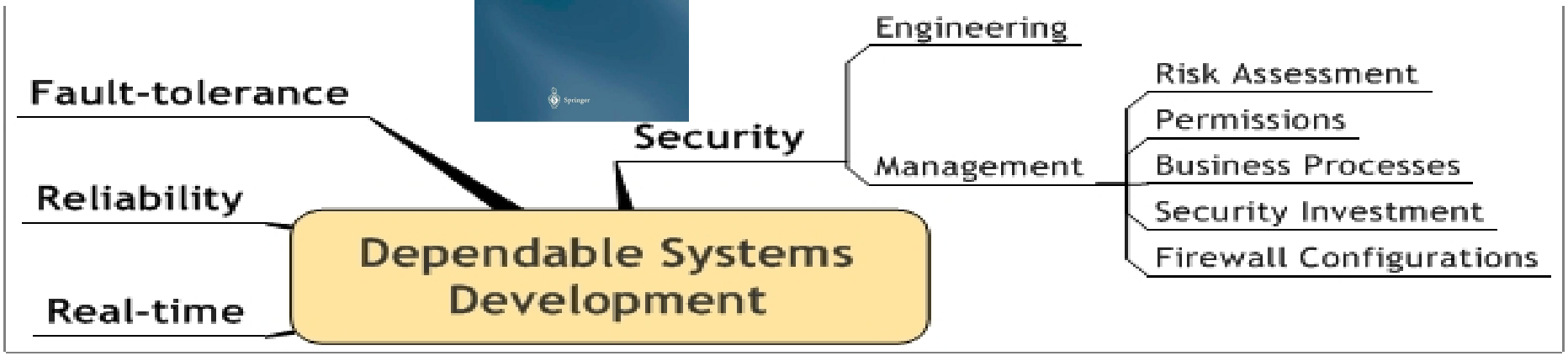
Layered security protocol (first layer: SSL protocol, second layer: client authentication protocol)

Security requirements:

- confidentiality
- authenticity

Overview


Patterns
Services
Architectures
Components
Aspects
Layers
Requirements
UMLsec
Formal Semantics
Analysis Framework
Processes
Models
Security Engineering
Model Checking
Autom. Theorem Proving
Refinement
CEPS
Biometry
Cryptokey
Jessie
TLS variant
Wiesncard
Applications
Runtime Checks
Code
Model-based Testing
Autom. Theorem Proving
Foundations
Secure Systems Development with UML
Jan Juerjens
Springer


Engineering
Fault-tolerance
Risk Assessment
Permissions
Security
Management
Business Processes
Reliability
Security Investment
Firewall Configurations
Dependable Systems Development
Real-time
IT Security
Jan Juerjens
Springer

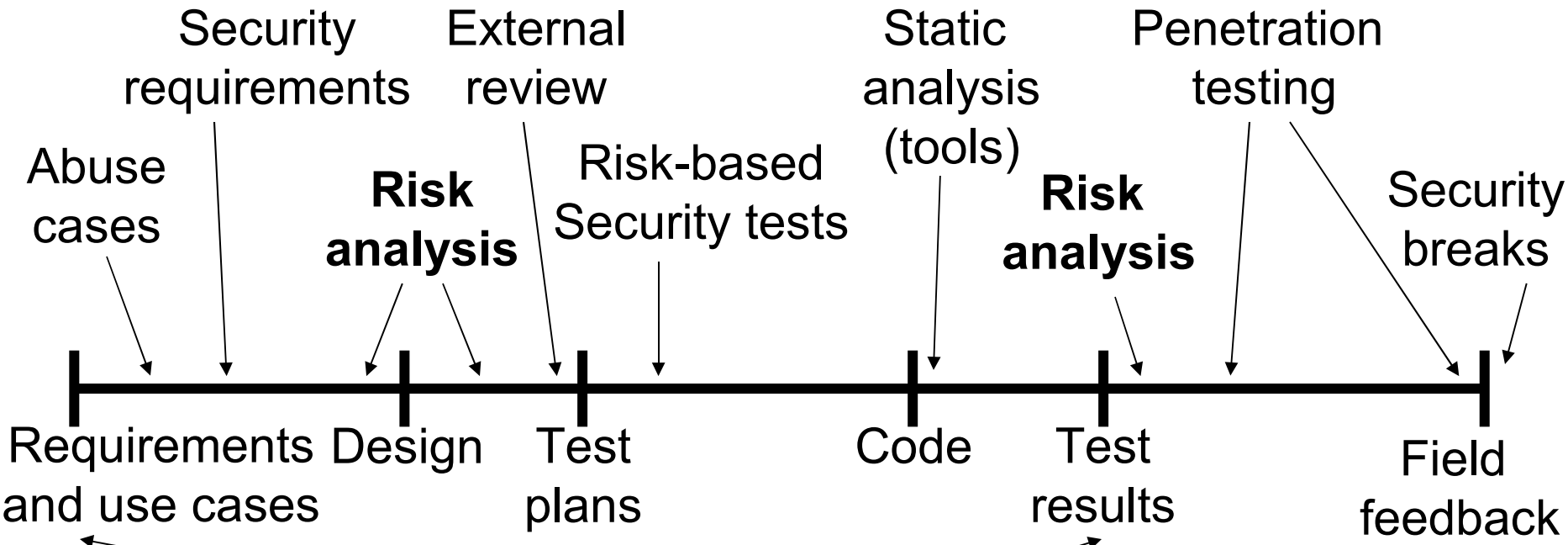
Challenge: Security

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).

Secure System Lifecycle

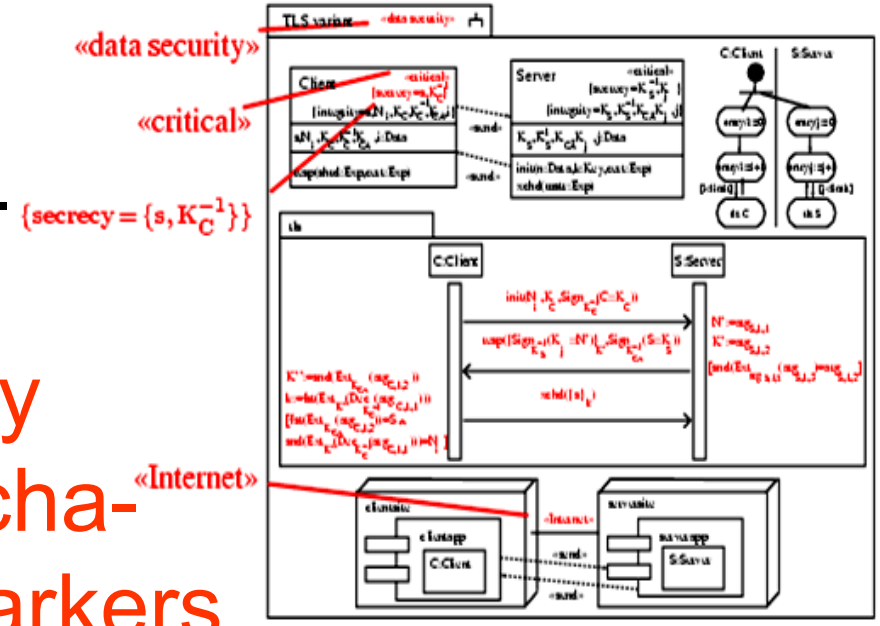


Model-based Security Engineering

[McGraw 2003]

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

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]

Why Prove *knows(s)*, not *secret(s)* ?

To prove *secret(s)*, would need to specify inequalities explicitly to avoid unrealistic confusion.

Problem: completeness and consistency.

Alternative: try to prove *knows(s)*.

Proof then gives existence of attack in all models, in particular confusion-free one.

Problem: Refutation counter-example may again have confusion !

Confusion-free Counterexample Problem

Result: established conditions on axioms and conjecture such that counterexample can be constructed to be confusion-free.

Proof (using logical model-theory): can construct all models from confusion-free ones while preserving the threat conjecture.

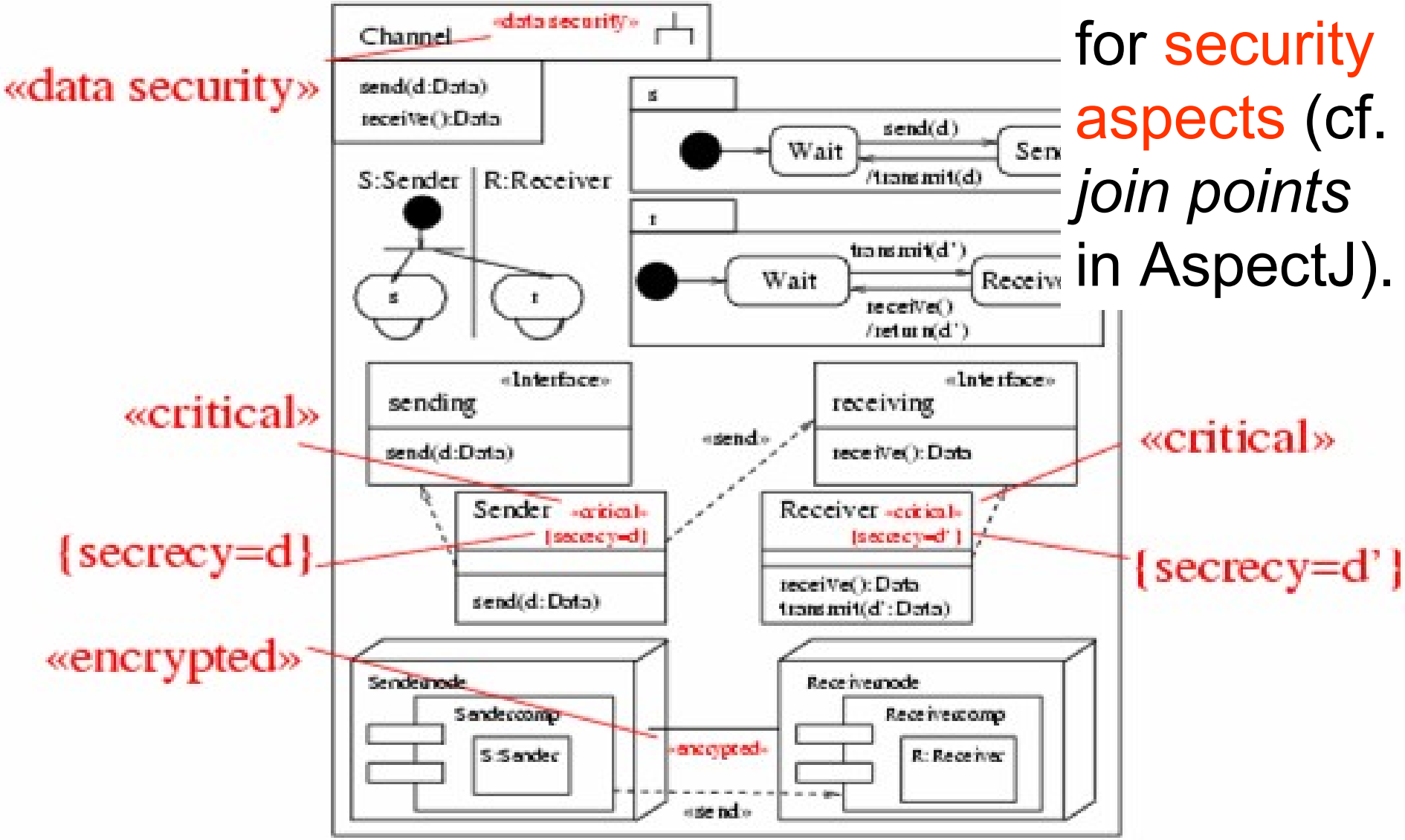
[In variety, all models quotients of free ones – can generalize.]

Model-based Security Aspects

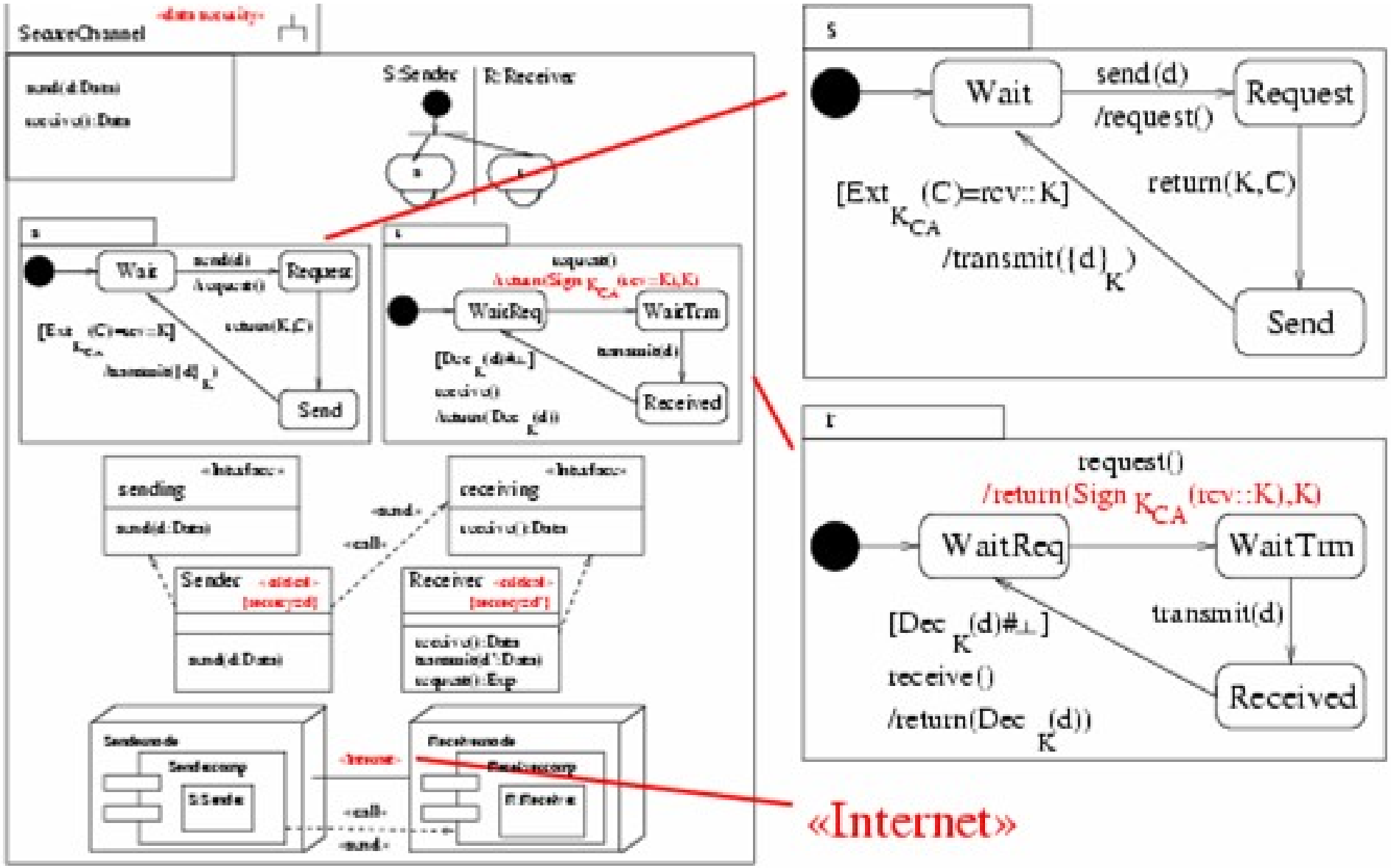
- Define abstract security aspect.
- Define concretization (e.g. protocol).
- Applying aspects results in transformation on models.
- If possible, give conditions under which it is secure to weave in aspect using concretization, e.g. by simulation argument.

Secure Channel Aspect

Primary model with *directives* for security aspects (cf. *join points* in AspectJ).



Secure Channel Aspect: Weaving



Aspect Validation

Need to prove concretization securely refines abstract aspect. Challenging problem in security.

For secure channel, have generic result.

Not always possible. Then use translation validation on the weaving transformation, before or after code generation.

Refinement & Composability

Need to refine models down to code.

Common formalizations of security properties not preserved by refinement.

Bad: re-verify after each step (incl code).

Theorem: Our notion of model refinement preserves security requirements. [FME01]

Similar: Established composability for certain security requirements under suitable assumptions. [Concur01]

Layered Security Protocols

System layer on top uses security services below.

client authenticity

confidentiality, integrity, server authenticity

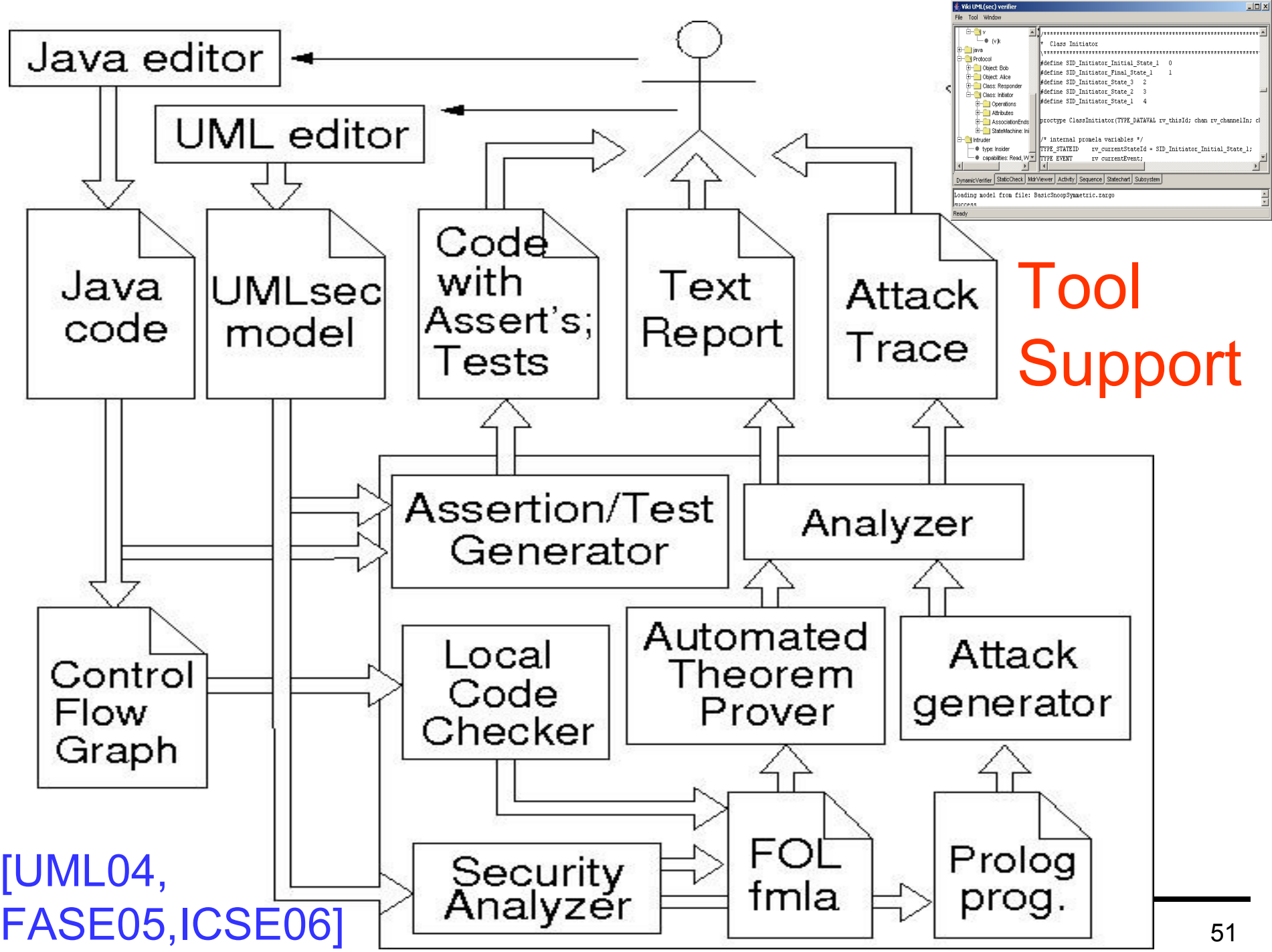
=

confidentiality, … + client authenticity

?

Security properties additive ? [Safecomp03]

Theorem: Yes, under suitable conditions.


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]