Model-based vs. Code-based Verification for Critical Systems

Jan Jürjens

Computing Department, The Open University, GB

from 1 Oct 2008 also:
Microsoft Research (Cambridge)

http://www.jurjens.de/jan
Personal Introduction

• Senior Lecturer (equiv. US Assoc. Prof.), Computing Departm., The Open University, GB
• From 1 Oct 2008: Royal Society Industrial Fellow at Microsoft Research (Cambridge)
• Extensive collaboration with industry (British Telecom, BMW, HypoVereinsbank, T-Systems, Munich Re, O2, Deutsche Bank, Siemens, Infineon, Allianz, …)
• PhD in Computer Science from Oxford Univ., Masters in Mathematics from Bremen Univ.
• Numerous publicat. inc. 2 books on secure software engineering
Verifying Critical Systems

Very challenging.
For high level of assurance, would need full coverage (test every possible execution).
Usually infeasible (especially reactive systems).
Have heuristics for trade-off between development effort and reliability.

Need to ask yourself:
• How complete is the heuristic?
• How can I validate it?

This talk: focus on security. Generalizes to other criticality requirements (fault-tolerance, reliability, …)
Problem: Security is Elusive

• Classical weakness in old Unix systems: “wrong password” message at first wrong letter in password. Using timing attack, reduce password space from $26^n$ to $26 \cdot n$ ($n =$ password length)

• More recent weakness on smart-card: reconstruct secret key by timed measurement of power consumption during crypto operations

→ How do you find these weaknesses using classical testing? (You don’t.)
Problem: Untrustworthy Programmer

- For security assurance, may not even trust the programmer of the code.
- May have intentionally built in back-door into code.
- May be impossible to find by random or black-box testing (e.g. hard-coded special password).
- Even worse when elusive weaknesses are used (previous slide).

What is the precaution in practice?

(usually none.)
Special Problem: Crypto

- Cryptography plays important role in many security-critical applications
- By definition, needs to be secure against brute-force attacks

Paradox: How do you get sufficient test coverage (for inputs accessible to a given attacker) of a system that needs to be secure against brute-force attacks on that input?

(Not using classical testing.)
Long-term goal: Tool-supported, theoretically sound, efficient automated security design & analysis.

Model-based System Assurance

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

Model-based System Assurance

**Design**: Encapsulate prudent engineering rules.

**Analysis**: Formally based, automated, efficient tools.

Note: emphasis on high-level requirements.
Architectural Layers

Models

Configurations

Security Archit.

Data

Applications

Operating System

Hardware

Crypto

Information

Network

Code

Access Control

Theoretical Foundations

Laws and Regulations

User

Developer

Administrator

Re-Engineer
Roadmap

Requirements

Weave in

(\textit{UML}) Models

Analyze against

Verify

Configure

Source Code

Generate/Verify

Verify

Configure

Runtime System

Configure

Execute

Verify

Configure

Verify

Verify

Configure

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify

Configure

Verify

Verify
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 mechanisms 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]
What Does UMLsec Cover?

Security requirements: <<secrecy>>, ...

Threat scenarios: Use Threats_{adv} (ster).

Security concepts: For example <<smart card>>.

Security mechanisms: E.g. <<guarded access>>.

Security primitives: Encryption built in.

Physical security: Given in deployment diagrams.

Security management: Use activity diagrams.

Technology specific: Java, CORBA security.
Security Protocols

System distributed over untrusted networks. „Adversary“ intercepts, modifies, deletes, inserts messages.

Cryptography provides security.

Cryptographic Protocol: Exchange of messages for distributing session keys, authenticating principals etc. using cryptographic algorithms.
Security Protocols: Problems

Many protocols have vulnerabilities or subtleties for various reasons

• weak cryptography
• core message exchange
• interfaces, prologues, epilogues
• deployment
• implementation bugs
Crypto-based Software (e.g. Protocols)

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

(Reasoned from [Dolev, Yao 1982])

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

(cf. [Dolev, Yao 1982])
Example: TLS Variant

Presented at IEEE Infocom 1999.
Goal: send secret protected by session key using fewer server resources.
Protocol

\[ \text{tls:} \quad \text{C:Client} \quad \text{S}_i : \text{Server} \]

\[
\begin{align*}
\text{init}(N_i, K_C, \text{Sign}_{K_C^{-1}}(C:: K_C)) \\
\rightarrow \text{resp} \left( \{\text{Sign}_{K_{S_i}^{-1}}(k_j :: N')\}_{K_C'}, \text{Sign}_{K_{CA}^{-1}}(S_i :: K_{S_i}) \right) \leftarrow \text{xchd}({\{s_i\}_k}) \\
\rightarrow [\text{snd}(\text{Ext}_{K_C'}(c_C)) = K_C']
\end{align*}
\]

\[
\begin{align*}
\text{fst}(\text{Ext}_{K_{CA}}(c_S)) = S_i \land \\
\text{snd}(\text{Ext}_{K_{S_i}'}(\text{Dec}_{K_C^{-1}}(c_k))) = N_i
\end{align*}
\]

\[
\begin{align*}
c_k & := \text{resp}_1 \\
c_S & := \text{resp}_2 \\
K_{S_i}' & := \text{snd}(\text{Ext}_{K_{CA}}(c_S)) \\
k & := \text{fst}(\text{Ext}_{K_{S_i}'}(\text{Dec}_{K_C^{-1}}(c_k)))
\end{align*}
\]

\[
\begin{align*}
N' & := \text{init}_1 \\
K_C' & := \text{init}_2 \\
c_C & := \text{init}_3
\end{align*}
\]
Define cryptosystem etc. E.g.: $\text{Dec}_{K^{-1}}(\{E\}_K) = E$

Bound on adversary knowledge set:
Predicate $\text{knows}(E)$, 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 $\text{knows}(s)$ from model-generated formulas using automated theorem prover. [ICSE05]

Formal foundations using streams. [JLAP08]
Example TLS Variant [IEEE Infocom 1999]

knows(N) ∧ knows(K_C) ∧ knows(Sign_{K_C^{-1}}(C::K_C))
∧ ∀ init_1, init_2, init_3. [knows(init_1) ∧ knows(init_2) ∧
knows(init_3) ∧ snd(Ext_{init_2}(init_3)) = init_2
⇒ knows({Sign_{K_S^{-1}}(…)}_{::}) ∧ [knows(Sign…)]
∧ ∀ resp_1, resp_2. […⇒…]]
Analysis

Check whether can derive $knows(s)$ e.g. using ATP for FOL.

Surprise: Yes!

→ Protocol does not preserve secrecy of $s$.

Why? Use Prolog-based attack generator.

```prolog
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), inv(k_c)) ) ) & % Server -> Attacker (2. message) ( ( knows(ArgS_11) & knows(ArgS_12) & knows(ArgS_13) & ( ? [X] : equal(sched(name), ... run entering next ... ) ) ) & % Client -> Attacker (3. mes. e-SETED done, exiting ... ) knows(enc(ArgS_11), ArgS_12) & knows(sign(conc(s, DataC_KK), ... ArgC_11 ) & equal(enc(sign(conc(DataC_k, DataC_n), inv(DataC_k_c), ArgC_11 ) & ( ? [DataC_ks] : equal(sign(conc(s, DataC_ks), inv(DataC_ks), ArgC_12 ) ) & equal(enc(sign(conc(DataC_k, n), inv(DataC_KK)), ArgC_11 ) & equal(enc(sign(conc(DataC_k, DataC_n), inv(DataC_KK)), ArgC_11 ) ) ) => ( knows(symenc(secret, DataC_k)) ) )))
```
Man-in-the-Middle Attack

$$N_i::K_C::\text{Sign}_{K_C^{-1}}(C::K_C)$$

$$C \rightarrow A \rightarrow S$$

$$N_i::K_A::\text{Sign}_{K_A^{-1}}(C::K_A)$$

$$A \rightarrow S$$

$$\{\text{Sign}_{K_S^{-1}}(K_j::N_i)\}_K_A::\text{Sign}_{K_C^{-1}}(S::K_S)$$

$$A \leftarrow S$$

$$\{\text{Sign}_{K_S^{-1}}(K_j::N_i)\}_K_C::\text{Sign}_{K_C^{-1}}(S::K_S)$$

$$C \leftarrow A$$

$$\{s\}_K_j$$

$$C \rightarrow A \rightarrow S$$

$$\{s\}_K_j$$

$$\{s\}_K_j$$
The Fix

e-Setheo: Proof that \textit{knows}(s) not derivable.
Note completeness of FOL (but also undecidability).
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.  

Similar: Established composability for certain security requirements under suitable assumptions.  
Also: Demonstrated how to apply security using aspect-oriented weaving / service orientation.

[FME01] [Concur01] [ICSOC 04, Models 05]
Layered Security Protocols

System layer on top uses security services below.

- Confidentiality, integrity, server authenticity
- Client authenticity

Security properties additive? [Safecomp03]

Theorem: Yes, under suitable conditions.

Model Verification

\[ \forall \text{arg}_1, \ldots, \text{arg}_n \ . \ (\text{knows}(\text{arg}_1) \land \ldots \land \text{knows}(\text{arg}_n) \land \text{cond}(\text{arg}_1, \ldots, \text{arg}_n) \Rightarrow \text{knows}(\text{exp}(\text{arg}_1, \ldots, \text{arg}_n))) \]

...)

 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) |)

analyzing results ...
model found/total failure
time limit information: 19 total / 18 strategy
(leading 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]
Tool-support: Pragmatics

Commercial modelling tools: so far mainly syntactic checks and code-generation.

Goal: sophisticated analysis. Solution:

• Draw UML models with editor.
• Save UML models as XMI (XML dialect).
• Connect to verification tools (automated theorem prover, model-checker …), e.g. using XMI Data Binding.
CSDUML Framework: Features

Framework for analysis plug-ins to access UML models on conceptual level over various UI’s. Exposes a set of commands. Has internal state (preserved between command calls).


Upload UML model (as .xmi file) on website. Analyse model for included critical requirements. Download report and UML model with highlighted weaknesses.
Tool Support

For example:
• consistency checks
• mechanical analysis of complicated requirements on model level (bindings to model-checkers, constraint solvers, automated theorem provers, …)
• code generation
• test-sequence generation
• configuration data analysis against UML.
Roadmap

- **Requirements**
  - Weave in
  - Analyze against

- **(UML) Models**
  - Code-/Testgen.
  - Generate/Verify
  - Configure/Verify

- **Source Code**
  - Configure/Verify

- **Configurations**
  - Configure

- **Runtime System**
  - Execute
Security Analysis: Model or Code?

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

Code:
+ „the real thing“ (which is executed)

$\Rightarrow$ Do both where feasible!
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
- Haneberg, Schellhorn, Grandy, Reif (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, Soot et al.): scalability wrt. code size / complexity; sophistication of properties (security).

→ Develop specialized verification approach based on these.
Model vs. Implementation

Sent and received data

Backtrace assignments

Compare meaning!

Defined during model creation

Elements of connections

Implement-ation

Java

Find

Has

Equal?

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

Transform to state machine:

\[ \text{trans}(\text{state}, \text{inputattern}, \text{condition}, \text{action}, \text{nextstate}) \]

where action can be outpattern or \( \text{localvar} := \text{value} \).

[ASE05, ASE06]
Real Life Challenges
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 (using run-time verification)
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.
I) Identify program points:
   value \((r)\), receive \((p)\), guard \((g)\), send \((q)\)

II) Check guards enforced
<table>
<thead>
<tr>
<th>Parameter der kryptographischen ClientHello Nachricht</th>
<th>Effektiv übertragene Daten der ClientHello Nachricht der Jessie Implementierung</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>type.getValue()</td>
</tr>
<tr>
<td>Pver</td>
<td>major</td>
</tr>
<tr>
<td></td>
<td>minor</td>
</tr>
<tr>
<td></td>
<td>((gmtUnixTime &gt;&gt; 16) &amp; 0xFFFF)</td>
</tr>
<tr>
<td></td>
<td>((gmtUnixTime &gt;&gt; 8) &amp; 0xFF)</td>
</tr>
<tr>
<td></td>
<td>(gmtUnixTime &amp; 0xFF)</td>
</tr>
<tr>
<td>r_c</td>
<td>randomBytes</td>
</tr>
<tr>
<td></td>
<td>sessionId.length</td>
</tr>
<tr>
<td>Sid</td>
<td>sessionId</td>
</tr>
<tr>
<td></td>
<td>((suites.size() &lt;&lt; 1) &gt;&gt; 8 &amp; 0xFF)</td>
</tr>
<tr>
<td></td>
<td>((suites.size() &lt;&lt; 1) &amp; 0xFF)</td>
</tr>
<tr>
<td>LCip</td>
<td>suites_1</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>suites_N</td>
</tr>
<tr>
<td>LKomp</td>
<td>comp.size()</td>
</tr>
<tr>
<td></td>
<td>comp_1</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>comp_N</td>
</tr>
</tbody>
</table>

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 ClientHello constructor

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

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

via Handshake.write()
initialized in SSLSocket.doClientHandshake()

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

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

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

```
<table>
<thead>
<tr>
<th>Send</th>
<th>Receive</th>
</tr>
</thead>
<tbody>
<tr>
<td>write(data₁),...,write(dataₙ)</td>
<td>read(data₁),...,read(dataₙ)</td>
</tr>
</tbody>
</table>
```

Buffer

bytes
Sending Messages

Automate this using patterns

SSLSocket.doClientHandshake()

Handshake.write()

ClientHello.write()

Random.write()

ProtocolVersion.write()

call of OutputStream.write()

traverse CFG
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 crypto for inequalities).

[ICFEM02]

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

[ASE06]
private void checkTrusted(X509Certificate[] chain, String authType) throws CertificateException {
    ...}

calls checkTrusted()

guard: checkServerTrusted()

calls verify() for every member of certificate chain

calls doVerify()

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

does not throw CertificateException

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"); ...
    }
}

java.security.Signature
• Initialize
• Update
• Verify
"verifies the signature"
msg = Handshake.read(din, certType);

try {
  session.trustManager.checkServerTrusted(peerCerts, suite.getAuthType());
  msg = new Handshake(Handshake.Type.CLIENT_KEY_EXCHANGE, ckex);
  msg.write(dout, version);
} catch {
  only possible way without throwing exception
}

[[equal(fst(ext_{K_c}(c_s)), S)]]
Roadmap

Requirements

(UML) Models

Source Code

Configurations

Runtime System

Weave in

Analyze against

Configure Verify

Generate/Verify

Execute

Configure Verify
Model-based Testing

Advantages over classical testing:

• Precise measures for completeness.
• Can be formally validated.

Two complementary strategies:

• Conformance testing
• Testing for criticality requirements
Conformance Testing

Classical approach in model-based test-generation (much literature).
Can be superfluous when using code-generation [except to check your code-generator, but only once and for all].
Works independently of real-time requirements.
Conformance Testing: Caveats

• Complete test-coverage still infeasible (although can measure coverage).

• Can only test code against what is contained in model. Usually, model more abstract than code. May lead to "blind spots".

For both reasons, may miss critical test-cases. Want: "criticality testing".
Criticality Testing: Strategies

**Internal:** Ensure test-case selection from models does not miss critical cases: Select according to information on criticality.

**External:** Test code against possible environment interaction generated from parts of the model (e.g. deployment diagram with information on physical environment).
Criticality Testing

Shortcoming of classical model-based test-generation (conformance testing) motivates „criticality testing“.

Goal: model-based test-generation adequate for critical real-time systems.
Internal Criticality Testing

Need behavioral semantics of used specification language (precise enough to be understood by a tool).

Here: semantics for simplified fragment of UML in „pseudo-code“ (ASMs).

Select test-cases according to criticality annotations in the class diagrams.

Test-cases: critical selections of intended behavior of the system.
External Criticality Testing

Generate test-sequences representing the environment behaviour from the criticality information in the deployment diagrams.
Automated White-Box Testing

- Generate control flow graph.
- Analyze for criticality requirements.
- Use to generate critical test-cases.
Model-based Testing with UML

Meaning of diagrams stated informally in (OMG 2003).

Ambiguities problem for

- tool support
- establishing behavioral properties (safety, security)

Need precise semantics for used part of UML, especially to ensure security requirements.
Vulnerability in SSL implementation

Analyzed open-source implementation Jessie of SSL protocol.

• According to SSL specification, a certificate with (issuedDate, expiredDate) should be checked whenever a message is received.
• 4 call sites of certificate() were found in the code.
• Only 3 of them call the Veri() function.
• Test cases were constructed to reveal the vulnerability.
• Fix of the vulnerability can be done using AOP techniques.

[ICSMM07, with Yijun Yu, J. Mylopoulos]
**Verification of Guards in Code**

**send**: represents send command

**g**: FOL formula with symbols $\text{msg}_n$ 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 $[\text{true}] \ p \models g$.

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

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

SSA:

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

TPTP:

```plaintext
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:

- $\phi$ from predicates $\text{PRED}(P_i)$ as above (for each path $i$)
- $\psi$ assigning variables according to given interleaving

Join formulas $\psi \Rightarrow \phi$ together by conjunction.
Abstraction by Code Annotations

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

// <<FOL axioms>>

Similarly for variables / constants.
Modular Verification

For program fragment \( p \), generate set of statements \( \text{derive}(L,C,E) \) 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.
Roadmap

**Requirements**

Weave in

Analyze against

(UML) Models

Configure

Verify.

Generate/ Verify

Source Code

Configure

Verify

Runtime System

Configure

Execute

Another Problem

How do I know the running implementation is still secure after deployment?

• Does system model capture all relevant aspects about a system?

• Are assumptions about influences from a system's operational environment reflected adequately?

• Are the abstractions that need to be made to enable automated static verification of non-trivial systems faithful wrt the verification result?

➔ Run-time verification.
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.

Runtime verification in a nutshell

- Property
- automatic generation of
- Monitor
- Property fulfilled?
- System
- Actions

[A. Bauer]
Formal underpinnings

- System (safety) property, $\varphi$ specified in terms of linear time temporal logic [Pnu77]:
  $$\varphi ::= \text{true} \mid p \mid \neg p \mid \varphi \land \varphi \mid \varphi U \varphi \mid X \varphi \quad (p \in AP)$$

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

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

  $$\varphi \rightarrow BA_{\varphi} \text{ s.t. } L(BA_{\varphi}) = L(\varphi)$$
Semantics

\[ w, i \models \text{true} \]
\[ w, i \models \neg \varphi \quad \iff \quad w, i \not\models \varphi \]
\[ w, i \models p \in AP \quad \iff \quad p \in w(i) \]
\[ w, i \models \varphi_1 \lor \varphi_2 \quad \iff \quad w, i \models \varphi_1 \lor w, i \models \varphi_2 \]
\[ w, i \models \varphi_1 U \varphi_2 \quad \iff \quad \exists k \geq i. \ w, k \models \varphi_2 \land \forall i \leq l < k. \ w, l \models \varphi_1 \]
\[ w, i \models X\varphi \quad \iff \quad w, i + 1 \models \varphi \]

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

Write \( F \, \text{phi} \) for true U phi (“eventually phi”); \( G \, \text{phi} \) for not F not phi (“globally phi”); \( \text{phi1 W phi2} \) 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:]

[A. Bauer]
ClientKeyExchange

Client will not send out \texttt{ClientKeyExchange} message until has received \texttt{Certificate} message and check is positive, and then sends it out.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig1.png}
\caption{FSM $\neg\text{ClientKeyExchange}_S \cup \neg\text{Certificate}_R$.}
\end{figure}
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).

\[ \varphi_3 = \neg Data \mathcal{W} \left( (MD5(\text{Finished}_R) = MD5(\text{Finished}_S)) \right) , \]

not co-safety but safety
Server Finished

Server will not send *Finished* message before MD5 received in Client`s *Finished* message equal to MD5 created by server. Then sends out eventually.

NB: Improves on Schneider’s security automata.

\[ \varphi_2 = (\neg \text{finished} \land W \text{ equal} \land (F \text{ equal} \Rightarrow F \text{ finished})) \]

not safety nor co-safety
Tracing Security Requirements

- Tracing security requirements to models...
- ... reconciling them with other non-functional requirements such as fault-tolerance, performance...
- ... and from models to code.
- For legacy systems need to extract security domain knowledge from the code.

[CAISE 06]
[UML 04, JSS 07]
[ASE 07, ICSM 08, ASE 08, w. Y. Yu]
[CSMR 07, CSMR 08, IPCP 08, w. D. Ratiu]
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.
German Health Card Architecture

- Analyzed architecture against security requirements using UMLsec
- Detected several security weaknesses in the architecture

[Meth. Inform. Medicine 08]

Application of Model-based Security Assurance at Mobile Communication Systems at O2 (Germany)

All 62 relevant security requirements from security policy successfully established using the approach.

Model-based development does incur extra effort.

Seems manageable when applied to critical system core.

Justifiable in case of high assurance needs (security).

Compares favorably with other assurance/same trustworthiness.

UMLsec well-suited for mobile communication systems.
MetaSearch Engine: Personalized search in company intranet (including password protected).

Some documents highly security-critical. 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.

Successfully analyzed using model-based security.
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
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]
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] Here: consider different protocol from public sources but with similar problems.
How does it compare?

- Empirical study to compare classical vs. model-based testing: embedded software / Automotive (window controller). In cooperation with colleagues from BMW / Elektrobit.

<table>
<thead>
<tr>
<th>Modelchecking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Examines an abstract model</td>
</tr>
<tr>
<td>Cheap and early verification</td>
</tr>
<tr>
<td>(without setting up complex in-the-loop-test environments)</td>
</tr>
<tr>
<td>Proof of correctness of properties possible</td>
</tr>
<tr>
<td>Uses selected user specific properties</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Examines a physical or concrete system</td>
</tr>
<tr>
<td>In-the-loop-tests take place in an environment near to the real one</td>
</tr>
<tr>
<td>No proof of correctness of properties possible</td>
</tr>
<tr>
<td>Uses often many, superficial test cases</td>
</tr>
</tbody>
</table>

THIS FRIDAY!
Conclusions

Model-based vs Code-based Verification using UMLsec:

- **formally** based approach
- **automated** tool support
- **industrially** used methods
- **integrated** approach (source-code, configuration data)

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

• Security Verification of Crypto Protocol Implementations in C: Use VCC to verify C code. (with Andy Gordon, MSR Cambridge; RS Industrial Fellowship & 2 PhD projects)
• Modelling for Compliance (EPSRC CASE PhD project with British Telecom)
• Security Engineering for Lifelong Evolvable Systems (EU FP7 Integrated Project): HIRING NOW: 2 Postdocs!
• RS Joint International Project with TU Munich on Formal Security Analysis of Cryptoprotocol Implementations
• RS Joint International Project with NII Tokyo Relating Security Requirements and Design
Overview

Security Engineering

Patterns
Architectures
Aspects
Requirements
Processes
Applications

Models
Components
Layers

UMLsec
Formal Semantics
Analysis Framework
Model Checking
Autom. Theorem Proving
Refinement

Runtime Checks
Model-based Testing
Autom. Theorem Proving

Code

Foundations
CEPS
Biometry
Cryptokey
Jessie
TLS variant
Wiesncard

IT Security

Fault-tolerance
Reliability
Real-time

Security

Engineering

Risk Assessment
Permissions
Business Processes
Security Investment
Firewall Configurations

Dependable Systems Development
Questions?

More information (papers, slides, tool etc.):
http://www.jurjens.de/jan
J.Jurjens@open.ac.uk
Roadmap

Requirements

Weave in

(UML) Models

Analyze against

Generate/
Verify

Source Code

Configure

Verify.

Runtime System

Configure

Verify

Execute