Model-based Security Testing
Supporting Evolution
Jan Jürjens

Open University (GB)
Microsoft Research (Cambridge)
Robinson College (Univ. Cambridge)

http://www.umlsec.de
J.Jurjens@open.ac.uk

http://www.securechange.eu
3 Claims for this Talk

- Quality Assurance for security-critical systems is **very difficult**: some security properties **cannot be established** by conventional testing.

- To save effort, a security assurance approach needs to support system evolution.

- **Model-based Security Testing** can deal with these challenges.
Security: Some Problems

„Blind“ use of security mechanisms:

- Security usually compromised by circumventing (rather than breaking) them.

- Assumptions on system context, physical environment.

- Attacker may use unintended/unnoticed functionality

„Those who think that their problem can be solved by simply applying cryptography don`t understand cryptography and don`t understand their problem“ (R. Needham).
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 $26*n$ ($n = \text{password length}$)

- 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 your precaution? (Probably 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?

→ What`s your answer?

(Not using classical testing.)
Testing Security-Critical Systems

Very challenging.

Given motivated adversary, would need full coverage (test every possible execution).

Usually infeasible (especially open systems).

Need heuristics for trade-off between testing effort and reliability.

Need to ask yourself:

- How complete is the heuristic?
- How can I validate it?
Model-based Security Assurance

Requirements

Weave in

Requirements

Analyze against

Configure

Verify

Monitor

Execute

Reverse Engin.

UML Models

Code-/Testgen.

Source Code

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

Model-based Security Assurance

- Critical requirements
- Abuse cases
- Risk analysis
- Design
- Test plans
- Risk-based tests
- Code
- Test results
- Static analysis (tools)
- System Monitoring
- System breaks
- Field feedback

Security Assurance: Model vs Code

Model:
+ earlier (less expensive to fix flaws)
+ more abstract ➜ more efficient

➤ Analyze both model and code:
  ▪ construct interface spec model
  ▪ analyze interface spec for critical requirements
  ▪ verify that software satisfies interface spec

Important: to save effort need to automatically trace code evolution back to model level.
➤ Automatically reuse earlier tests.
**UMLsec**

Extension for secure systems development.

- evaluate UML specifications for weaknesses in design
- encapsulate established rules of prudent secure engineering as checklist
- make available to developers not specialized in secure systems
- consider security requirements from early design phases, in system context
- make certification cost-effective
Tool Support

Java editor

UML editor

Java code

UMLsec model

Code with Assert’s; Tests

Text Report

Attack Trace

Assertion/Test Generator

Analyzer

Local Code Checker

Automated Theorem Prover

Attack generator

Security Analyzer

FOL fmla

Prolog prog.

Works also with C.
Model-based Testing Example: Protocols

Sent / received data  Testing  Test cases

Execute  Generate

Incorrect Implementation?

Model-based Security Testing: Strategies

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

**External:** Test code against possible environment interaction generated from parts of the model (e.g. deployment diagram with information on physical environment).
To address problems mentioned earlier, test cases generation needs to be informed by actual code.

- Generate control flow graph.
- Analyze for criticality requirements (e.g. security).
- Use to generate critical test-cases by mutation testing.
Example: SSL

I) Identify program points:
   value (r), receive (p), guard (g), send (q)

II) Check guards enforced

msg = Handshake.read(din, certType);

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

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

only possible way without throwing exception
Maintaining Traceability under Evolution

Code evolution can potentially multiply testing effort. Need to re-verify code parts that changed, re-do integration testing etc.

Particular problem when testing for sophisticated properties (such as security) since requires particular effort.

→ Want to automatically trace code evolution to model level so can automatically reuse earlier tests.

[Bauer, Jurjens, Yu 09]
Model-Code Co-Evolution

Basic observation: Most system changes can be reduced to two kinds:
- Adding / removing parts of the system.
- Basic refactoring operations to hold system parts together despite changes.

When adding / removing code parts we need to assume that the corresponding models are also added / removed.

For evolution by refactoring can achieve automated model-code traceability e.g. using Eclipse Refactoring Language Toolkit (LTK) / XML based refactoring scripts.

Maintain model-code synchrony using continuous integration scripts (with CruiseControl / Apache Ant).
Evolution as Code Refactoring

DESIGN

Virtualizing refactorings

Concretizing refactorings

IMPLEMENTATION

Programs

Tests

Virtualizing refactorings

Concretizing refactorings

Programs

Tests
Refactoring: Tool Supports

Motivation - Model-based Security - Some details - Applications - Evaluation
Java Secure Sockets Extension

Applied our approach to a series of implementations of the Java Secure Sockets Extension library:

- Jessie 1.0.0
- Jessie 1.0.1
- JSSE 1.6

Demonstrated that our model-code co-evolution approach is robust even across major software changes.
### Refactoring JSSE

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Program entities</th>
<th>Identif.</th>
<th>Refactoring op.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $C$</td>
<td>clientHello</td>
<td>$C$</td>
<td>rename.type</td>
</tr>
<tr>
<td>2. $S$</td>
<td>serverHello</td>
<td>$S$</td>
<td>rename.type</td>
</tr>
<tr>
<td>3. $P_{\text{ver}}$</td>
<td>session.protocol</td>
<td>$P_{\text{ver}}$</td>
<td>extract.temp</td>
</tr>
<tr>
<td></td>
<td>version</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. $R_C$</td>
<td>clientRandom</td>
<td>$R_C$</td>
<td>rename.local.variable</td>
</tr>
<tr>
<td>$R_S$</td>
<td>serverRandom</td>
<td>$R_S$</td>
<td>rename.local.variable</td>
</tr>
<tr>
<td>5. $S_{\text{id}}$</td>
<td>sessionId</td>
<td>$S_{\text{id}}$</td>
<td>rename.field</td>
</tr>
<tr>
<td></td>
<td>sessionId</td>
<td>$S_{\text{id}}$</td>
<td>rename.local.variable</td>
</tr>
<tr>
<td>6. $\text{Ciph}[]$</td>
<td>session.enabledSuites</td>
<td>$\text{Ciph}$</td>
<td>extract.temp</td>
</tr>
<tr>
<td>7. $\text{Comp}[]$</td>
<td>comp</td>
<td>$\text{Comp}$</td>
<td>extract.temp</td>
</tr>
<tr>
<td>8. Veri</td>
<td>Lines 1518–1557</td>
<td>Veri</td>
<td>extract.method</td>
</tr>
</tbody>
</table>
## Refactoring: Performance

<table>
<thead>
<tr>
<th>Messages in sequence</th>
<th>op.</th>
<th>diff</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1: $C \rightarrow S : (P_{\text{ver}}, R_C, S_{id}, \text{Ciph}[], \text{Comp}[])$</td>
<td>7</td>
<td>31</td>
<td>13.891</td>
</tr>
<tr>
<td>S2. $S \rightarrow C : (P_{\text{ver}}, R_S, S_{id}, \text{Ciph}[], \text{Comp}[])$</td>
<td>5</td>
<td>20</td>
<td>9.437</td>
</tr>
<tr>
<td>S3. $S \rightarrow C : \text{Certificate}[X509Cert_s]$</td>
<td>2</td>
<td>2</td>
<td>1.474</td>
</tr>
<tr>
<td>S4. $C : \text{Veri}(X509Cert_s)$</td>
<td>2</td>
<td>2</td>
<td>3.854</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>Total of 7 messages and 3 checks</td>
<td>27</td>
<td>86</td>
<td>40.303</td>
</tr>
</tbody>
</table>
Vulnerability in SSL implementation

Analyzed open-source Java 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.

Note similar bug to this was recently found in widely used C implementation openSSL.
Some other Applications

Analyzed designs / implementations / configurations e.g. for

- Biometry- or smart-card-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
- Investigated potential security weaknesses in the architecture

[Meth. Inform. Medicine 08]

Jan Jürjens (OU / MSRC)
Intranet Information System

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.
Mobile Communications

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

Successfully verified using model-based security.
Common Electronic Purse Spec.

Global elec. purse standard (Visa et al., 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.
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).

Finally developed version secure by our analysis.
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. From the comparison:

<table>
<thead>
<tr>
<th>Modelchecking</th>
<th>Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Examine an abstract model</td>
<td>Examines a physical or concrete system</td>
</tr>
<tr>
<td>Cheap and early verification (without setting up complex in-the-loop-test environments)</td>
<td>In-the-loop-tests take place in an environment near to the real one</td>
</tr>
<tr>
<td>Proof of correctness of properties possible</td>
<td>No proof of correctness of properties possible</td>
</tr>
<tr>
<td>Uses selected user specific properties</td>
<td>Uses often many, superficial test cases</td>
</tr>
</tbody>
</table>
3 Things to Take Away

- Quality Assurance for security-critical systems is **very difficult**: some security properties **cannot be established** by conventional testing.

- To save effort, a security assurance approach needs to support system evolution.

- **Model-based Security Testing** can deal with these challenges.

More information: [www.umlsec.de](http://www.umlsec.de), [J.Jurjens@open.ac.uk](mailto:J.Jurjens@open.ac.uk)

www.securechange.eu