Security for Changing Software and Systems

Jan Jürjens

TU Dortmund & Fraunhofer ISST

http://jan.jurjens.de
The Forgotten End of the System Life-cycle

Challenges:

• Software lifetime often longer than intended (cf. Year-2000-Bug).
• Systems evolve during their lifetime.
• In practice evolution is difficult to handle.

Problem: Critical requirements (e.g. security) preserved?
Challenge: Evolution

Each artifact may evolve.

To reduce costs, reuse verification results as far as possible.

⇒ Under which conditions does evolution preserve security?

Even better: examine possible future evolution for effects on security.

- Check *beforehand* whether potential evolution will preserve security.
- Choose an architecture during the design phase which will support future evolution best wrt. security.
Model Formalization

Formalize model execution. For transition \( t=(\text{source}, \text{msg}, \text{cond}[\text{msg}], \text{action}[\text{msg}], \text{target}) \) and message \( m \), execution formalized as:

\[
\text{Exec}(t,m) = [\text{state}_{\text{current}}=\text{source} \land m=\text{msg} \land \text{cond}[m]=\text{true} \\
\quad \Rightarrow \text{action}[m] \land \text{state}_{\text{current}.t(m)}=\text{target} ].
\]

(where \( \text{state}_{\text{current}} \) current state; \( \text{state}_{\text{current}.t(m)} \) state after executing \( t \)).

Example: Transition \( t_0 \):

\[
\text{Exec}(t_0,m)= \\
[ \text{state}_{\text{current}}=\text{NoExtraService} \\
\land m=\text{wm}(x) \land \text{money}_{\text{current}}+x=1000 \\
\Rightarrow \text{money}_{\text{current}.t_0(m)}=\text{money}_{\text{current}}+x \land \text{state}_{\text{current}.t_0(m)}=\text{ExtraService} ].
\]
Formalization of Requirements

Example „secure information flow“:
No information flow from confidential to public data.

**Analysis:** If states $state_{\text{current}}$, $state'_{\text{current}}$ differ only in confidential attributes, then publically observable behaviour should be same:

$$state_{\text{current}} \approx_{\text{pub}} state'_{\text{current}} \Rightarrow state_{\text{current}.t(m)} \approx_{\text{pub}} state'_{\text{current}.t(m)}$$

($state_{\text{current}} \approx_{\text{pub}} state'_{\text{current}}$ : same publically observable behaviour;
$state_{\text{current}.t(m)}$ : next state after executing $t(m)$).

**Example:** Insecure: confidential attribute $money$ influences return value of public method $rx()$.

---

**ExtraService $\approx_{\text{pub}}$ NoExtraService**
aber nicht:

ExtraService.$rx()$ $\approx_{\text{pub}}$ NoExtraService.$rx()$
Evolution vs. Design - Architectural Principles

Consider design- / architectural principles supporting evolution. Under which conditions are requirements preserved?

**Design technique: Refinement of specifications.** Supports evolution between refinements of an abstract specification.

**Architectural principle: Modularization** supports evolution by restricting impact of change to modules. Different dimensions:

- **Architectural layers**
- **Component**-oriented architectures
- **Service**-oriented architectures
- **Aspect**-oriented architectures

For each discovered conditions under which requirements are preserved. Explain this at the hand of security requirements.

Ochoa, Jürjens, Warzecha: A Sound Decision Procedure for the Compositionality of Secrecy. ESSoS’12
Ruhroth, Jürjens. Supporting Security Assurance in the Context of Evolution: Modular Modeling and Analysis with UMLsec. HASE’12
Schmidt, Jürjens: Connecting Security Requirements Analysis and Secure Design Using Patterns and UMLsec. CAiSE’11
Hatebur, Heisel, Jürjens, Schmidt: Systematic Development of UMLsec Design Models Based on Security Requirements. FASE’11
Refinement: Problem

For behaviour preserving refinement, one would expect preservation of behavioural requirements.

„Refinement Paradox“: Surprisingly, in general not true [Roscoe‘96].

Example: In above example, transition \( rx() / \text{return}(\text{true}) \)
(resp. \( \text{false} \)) is refinement of „secure “ transition \( rx() / \text{return}(\text{random_bool}) \).

Problem: Mixing non-determinism as under-specification resp. as security mechanism.
Refinement: Solution

Our specification approach separates non-determinism as under-specification resp. as security mechanism.

**Result:** Refinement now preserves behavioural requirements.

\[
\text{Definition } Q \text{ refines } P \ (P \sim Q) \text{ if for each } \bar{s} \in \text{Stream}_{IF} \\
\text{have } [P](\bar{s}) \supseteq [Q](\bar{s}).
\]

**Theorem** If \( P \) preserves secrecy of \( m \) and \( P \sim Q \) then \( Q \) preserves secrecy of \( m \).

**Proof:** using formal semantics.

**Above example:** with our approach: not a refinement.
Modularization: Problem

**Problem**: Behavioural requirements in general not compositional.

**Above example**: States *ExtraService* and *NoExtraService* each „secure“ (only one return value for *rx*), but composition in statechart not.

Under which condition are requirements preserved?
Modularization: (A) Solution

Solution: Formalize requirement as „rely-guarantee“-property.

Result: Using this formalization, get conditions for compositionality.

Proof: using formal semantics.

Above example: Rely-guarantee formalization shows that secure composition impossible.
Evolution-based Verification

Evolution-based Verification – Idea:

• Initial verification: Tool registers which **model elements** relevant for verification of given requirement.

• Store in verified model, together with partial results („proof-carrying models“).

• Discovered **conditions on changes** such that requirement preserved.

• **Compute difference** between old and new model (e.g. using SiDiff [Kelter]).

• Only need to re-verify **model parts** which
  1) were **relevant** in the initial verification,
  2) have **changed**, and
  3) which don’t satisfy the above-mentioned conditions.

**Significant verification speed-up compared to simple re-verification.**


**Theorem 1** Assume that the program $p'$ evolved from the program $p$ where $p$ and $p'$ are related as in the following cases

$p = \text{either } p' \text{ or } p''$: This implies $p \supseteq p'$ and $p \supseteq p''$.

$p = \text{if } E = E' \text{ then } p' \text{ else } p'': \text{ For any expression } X \in \text{Exp}$ such that $p$ preserves the secrecy of $X$:

$p'$ preserves the secrecy of $X$ assuming $E = E'$ and

$p''$ preserves the secrecy of $X$ assuming $E \neq E'$.

Performance measurement comparison for verification of secure dependency
Evolution-based Verification: Example

Preservation condition for secure information flow at evolution $M \rightarrow M'$: Only consider states $s, s'$ for which:

- $s \approx_{pub} s'$ in $M'$ but not in $M$, or
- $s.t(m) \approx_{pub} s'.t(m)$ in $M$ but not in $M'$.

Example: $wm(0).rx() \approx_{pub} wm(1000).rx()$ in $M$ but not in $M'$. Shows that $M'$ violates secure information flow (confidential data $0$ and $1000$ distinguishable).
Model-code Traceability under Evolution

Goal: Preserve model-code traceability during evolution.

Idea: Reduce evolution to:
- Adding / deleting model elements.
- Supporting refactoring operations.

=> Approach for automated model-code traceability based on refactoring scripts in Eclipse.

Code Verification subject to Evolution

Use evolution-based model verification and model-code traceability for evolution-aware code verification using static analysis.

**Example:** Condition in sequence diagram correctly checked in implementation.

Project Csec (with Microsoft Research Cambridge): Implemented static analysis, found several weaknesses.

<table>
<thead>
<tr>
<th></th>
<th>C LOC</th>
<th>IML LOC</th>
<th>outcome</th>
<th>result type</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple mac</td>
<td>(~250)</td>
<td>12</td>
<td>verified</td>
<td>symbolic</td>
<td>4s</td>
</tr>
<tr>
<td>RPC</td>
<td>(~600)</td>
<td>35</td>
<td>verified</td>
<td>symbolic</td>
<td>5s</td>
</tr>
<tr>
<td>NSL</td>
<td>(~450)</td>
<td>40</td>
<td>verified</td>
<td>computat.</td>
<td>5s</td>
</tr>
<tr>
<td>CSur</td>
<td>(~600)</td>
<td>20</td>
<td>flaws found</td>
<td>—</td>
<td>5s</td>
</tr>
<tr>
<td>Metering</td>
<td>(~1000)</td>
<td>51</td>
<td>flaws found</td>
<td>—</td>
<td>15s</td>
</tr>
</tbody>
</table>

Aizatulin, Gordon, Jürjens: Computational Verification of C Protocol Implementations by Symbolic Execution. CCS’12
Aizatulin, Gordon, Jürjens: Extracting and verifying cryptographic models from C protocol code by symbolic execution. CCS’11
Run-time Verification subject to Evolution

Relevant versions of source code not always available => run-time monitoring.

Relevant approach in the literature: Security Automata [F.B. Schneider 2000].

**Problem: no evolution** and only „safety“-**properties** supported
(too restrictive e.g. for secure information flow).

**So:** New approach, based on runtime verification (based on techniques from model-checking and testing).

Formalize requirement to be monitored in LTL.

Continuous monitoring of system events through monitors generated from the models, with **evolution-based traceability**.

Including non-safety-properties (using 3-valued LTL-semantics).

**Example results:**

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
<th>No Monitor [s]</th>
<th>Monitor [s]</th>
<th>Overhead [s]</th>
<th>Overhead [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>GnuTLS</td>
<td>GnuTLS</td>
<td>0.109</td>
<td>0.120</td>
<td>0.011</td>
<td>10.313</td>
</tr>
<tr>
<td>OpenSSL</td>
<td>JESSIE</td>
<td>0.158</td>
<td>0.172</td>
<td>0.014</td>
<td>8.986</td>
</tr>
<tr>
<td>GnuTLS</td>
<td>JESSIE</td>
<td>0.144</td>
<td>0.148</td>
<td>0.004</td>
<td>2.788</td>
</tr>
</tbody>
</table>
Technical Validation

- **Correctness**: based on formal semantics.
- **Completeness**: view model transformation as sequence of deletions, modifications and additions of model elements.

Performance gain **maximal** where **difference << software**. Example result:

- Evolution-based verification: Performance **linear** in software size (given constant size of differences)
- Complete Re-Verification: Performance **exponential** in software size.

This condition is satisfied e.g. for:

- **Maintenance of stabilé software**
- **QA tightly integrated with evolution** (e.g. nightly builds)
Practical Validation

Application of UMLsec in practice (examples):

• Global Platform (smartcard software updates, Gemalto)
• Mobile software architecture (Telefonica O2 Germany)
• Biometric authentication system
• German Health Card
• Health information systems

Detected signification weaknesses for some of these.

Empirical comparison model-based vs. traditional QA (testing):

Example: **Model-checking vs. simulation / testen:**

Door control unit (coop. w. BMW). Model-checking: Additional effort (1-2 days / LTL formula), but detects also obscure bugs.
Conclusion:  
Security for Changing Software and Systems

Evolution: challenging for QA.

Question: Can reuse QA results after evolution?

Result: Conditions for requirements preservation...

• … in context of design-/architectural techniques for evolution (e.g. refinement, modularization).
• … under model evolution (“evolution-based verification“).
• evolution-based static analysis and run-time verification.
• Tool-implementation: significant performance and scalability gains wrt. simple re-verification.

Validation: Successful use in practice.

Current work: SecVolution @ DFG-PP Design for Future