
Jan Jürgens

TU Dortmund & Fraunhofer ISST

http://jan.jurjens.de
What is Software Evolution?

Software today often long-living (cf. Year-2000-Bug).

Continual change during lifetime.
- Changing requirements, changing environment
- Bug fixing

→ Software evolution!
What is the Challenge about Software Evolution?

After each change: redo software tests (regression test).

High costs:

- E.g. Y2k-problem:
  Ca. 600 Bill. US$ world-wide.

→ Often insufficient regression tests:

- Explosion of Ariane 5 (1996):
  Software from Ariane 4 reused in Ariane 5 without sufficient regression tests.
  → 370 Mio US$ damage.
How to solve this Problem?

Goal: Integrate quality assurance with evolution:

• Reuse QA results as far as possible to reduce costs.

→ Under which conditions requirements preserved?

• Only re-verify when conditions not fulfilled.
  • And only systems parts where necessary.

• Check at model level before evolution carried out.
Secure Software Evolution

- Challenge and Scientific Basis
- Secure Software Evolution
  - General Approach
  - Preservation Results
- Validation
Model-based Security Assurance

Security Requirements

Annotate

Analyse

UML Model

Generate

Static Analysis

Configuration Data

Configure

Runtime System

Test generation

Monitor

Execute

Jan Jürjens: Secure Evolution: Challenge – Approach – Results – Validation

Refinement of specifications

➤ Evolution between different refinements.

Applying refactorings

➤ Carry out evolution in controllable way.

Applying design patterns

➤ Reduce complexity of evolution.

Architectural principle modularization

➤ Limit impact of change to system parts.

Question: when do these preserve security properties?

Evolution vs. Design / Architecture
When Properties Preserved?

**Refinement:** Developed refinement which *preserves* security.

**Conditions** for preservation of security through…

… **Refactoring:** using Eclipse refactoring scripts

… Applying design patterns: „Gang of Four“ patterns

… **Modularization:**

- Layering of *architectural levels*
- **Component**-oriented architectures
- **Service**-oriented architectures
- **Aspect**-oriented development

**Experiments:** Relevant in 57% of code base.

Evolution-based Verification

- Initial verification: Register which **system parts** relevant.
- Store partial results in model („proof-carrying models“).
- **Compute difference:** old vs. new model (e.g. with SiDiff [Kelter]).
- Only re-verify **system parts** that:
  1) **relevant** in initial verification,
  2) **changed**, such that
  3) conditions on **preservation** of security not fulfilled.

**Significant speed-up vs. complete** re-verification.

Model-Code Traceability at Evolution

**Goal:** Preserve model-code traceability of security properties during evolution.

**Idea:** Reduce evolution to:

- **Adding / deleting** system elements.
- **Supporting** refactoring operations.

→ Approach for automated model-code traceability based on refactoring scripts in Eclipse.
Vulnerability in OpenSSL:


“Several functions inside OpenSSL incorrectly checked the result after calling the EVP_VerifyFinal function, allowing a malformed signature to be treated as a good signature rather than as an error.”

Feb/Mar 2014: “goto fail”-vulnerabilities in SSL @ iPhone, GnuTLS.
Security Analysis by Symbolic Execution

Compile protocol implementation to **symbolic model** for security analysis.

Example message: \[ A \xrightarrow{m, \text{hash}(m,k_{\text{shared}})} B \]

**C code:**
```c
client(char * payload, int payload_len){
    int msg_len = 5 + len + SHA1_LEN;
    char * msg = malloc(msg_len);
    char * p = msg;
    *p = len; p += 4; // add length
    *(p++) = 1; // add the tag
    memcpy(payload, p, len); // add the payload
    sha1(msg, 5 + len, p); // add the hash
    send(msg, msg_len); // send
}
```

**Model from symbolic execution:**
\[
\text{out}(\text{len}(\text{payload}) \mid 01 \mid \text{payload} \\
| \text{sha1}(\text{len}(\text{payload}) \mid 01 \mid \text{payload}, k_{\text{shared}}))
\]
Code-level Security Verification Subject to Evolution

Project Csec (with Microsoft Research Cambridge):

**Static code analysis** against **security** properties.

**Evolution-based** model analysis + model-code traceability $\Rightarrow$ **evolution-sensitive static code security analysis**.

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>RPC</td>
<td>~ 600</td>
<td>35</td>
<td>verified</td>
<td>symbolic</td>
</tr>
<tr>
<td>NSL</td>
<td>~ 450</td>
<td>40</td>
<td>verified</td>
<td>computat.</td>
</tr>
<tr>
<td>CSur</td>
<td>~ 600</td>
<td>20</td>
<td>flaws found</td>
<td>—</td>
</tr>
<tr>
<td>Metering</td>
<td>~ 1000</td>
<td>51</td>
<td>flaws found</td>
<td>—</td>
</tr>
</tbody>
</table>

Run-time Verification subject to Evolution

Source code not available ➔ run-time monitoring.

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

**Problem:** no evolution, only „safety“-properties.

➔ New approach, based on runtime verification¹:

- Security property in LTL.
- Generate monitors with evolution-based traceability.

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

<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>

¹ Havelund, Grosu 2002
**Goal** for monitoring:
Detect as *early* as possible that property **will** be violated.

**Use 3-valued semantics** for this.

- Finite automata for **minimal prefix** of violating state.

Also **non-safety-properties**:
Boolean combinations of safety and co-safety

---

Server sends message Finished (event “finished”) only after message Finished received from Client and MD5-hash equals MD5 at Server side (event “equal”). Then sends it out.

LTL formalization:
\[
\varphi_2 = (\neg \text{finished} \lor \text{equal}) \land (F \text{equal} \Rightarrow F \text{finished})
\]

(F phi: “eventually phi”; phi1 W phi2: “phi1 weak-until phi2”)

Non-safety-property (not with Security Automata).

Generated monitor:
Several versions of **Java security library** “Java Secure Sockets Extension (JSSE)” and open-source re-implementation (Jessie).

Found several **weaknesses** (similar “goto fail” in SSL@iPhone).

Works also for **large evolutions** (re-implementation).

---

**Table:**

<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. $P_{ver}$</td>
<td>session.protocol version</td>
<td>$P_{ver}$</td>
<td>extract.temp</td>
</tr>
<tr>
<td>4. $R_C$</td>
<td>clientRandom</td>
<td>$R_C$</td>
<td>rename.local.variable</td>
</tr>
<tr>
<td>5. $R_S$</td>
<td>serverRandom</td>
<td>$R_S$</td>
<td>rename.local.variable</td>
</tr>
<tr>
<td>5. S</td>
<td>session.Id</td>
<td>S</td>
<td>rename.field</td>
</tr>
</tbody>
</table>

**Messages in sequence**

<table>
<thead>
<tr>
<th>op.</th>
<th>diff</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>31</td>
<td>13.891</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>9.437</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1.474</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3.854</td>
</tr>
<tr>
<td>27</td>
<td>86</td>
<td>40.303</td>
</tr>
</tbody>
</table>
Secure Software Evolution

- Challenge and Scientific Basis
- Secure Software Evolution
  - General Approach
  - Preservation Results
- Validation
**Security Analysis: Formalization of Model**

**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:

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

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

**Example:** Transition \( t_0 \):

\[
\text{Exec}(t_0,m) = \lbrack \text{state}_{\text{current}} = \text{NoExtraService} \\
\land m = \text{wm}(x) \land \text{money}_{\text{current}} + x \geq 1000 \\
\Rightarrow \text{money}_{\text{current}.t_0(m)} = \text{money} + x, \text{state}_{\text{current}.t_0(m)} = \text{ExtraService} \rbrack.
\]
Security Analysis: Formalization of Property

Example „secure information flow“: No information flow from confidential to non-confidential data (not even partial / indirect).

Analysis: States differ only in confidential variables ➞ same impact on non-confidential variables:

\[
\text{state}_{current} \approx_{pub} \text{state'}_{current} = \text{state}_{current}.t(m) \approx_{pub} \text{state'}_{current}.t(m)
\]

\((\text{state}_{current} \approx_{pub} \text{state'}_{current}) : \text{differ only in confidential variables} \). 

Example insecure: Partial information flow from confidential attribute money to return value of non-confidential method rx().

Diagram:
- Customer account «no down−flow»
- ExtraService \(\approx_{pub}\) NoExtraService
  - aber nicht:
  - ExtraService.rx() \(\approx_{pub}\) NoExtraService.rx()
- Account «critical»
  - \(\text{secret} = \{\text{wm}, \text{rm}, \text{money}\}\}
  - \text{money}: Integer
  - \text{rm}(): Integer
  - \text{wm}(x): Integer
  - \text{rx}(): Boolean
- ExtraService
  - \text{rx}():return(true)
  - \text{rm}():return(money)
  - \text{wm}(x)
  - \[/money:=\text{money}+x\]
  - \[\text{money}+x\geq1000\]
  - \[\text{money}+x<1000\]
- NoExtraService
  - \text{rx}():return(false)
  - \text{rm}():return(money)
  - \text{wm}(x)
  - \[/money:=\text{money}+x\]
  - \[\text{money}+x<1000\]

\(\text{ExtraService} \approx_{pub} \text{NoExtraService}\)
Security vs. Refinement: Problem

For behaviour preserving refinement:
Would expect preservation of security.

„Refinement Paradox“: In general not true!

Example:
Transition \( rx() / \text{return}(true) \)
(resp. \( false \)) is refinement of „secure“ transition
\( rx() / \text{return}(\text{random\_bool}) \).

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

Our specification approach **distinguishes** the two kinds of non-determinism.

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

**Definition** \( Q \) refines \( P \) \((P \leadsto Q)\) if for each \( \bar{s} \in \text{Stream}_{IF} \) have \( \lbrack P \rbrack(\bar{s}) \supseteq \lbrack Q \rbrack(\bar{s}) \).

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

**Proof:** using formal semantics.

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

**Problem:** Security in general **not compositional**.

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

![Diagram](image-url)

Under which **condition** security properties **preserved**?
Security vs. Modularization: Solution

„Rely-guarantee“-formalization\(^1\) of property.

**Result:** Get conditions for compositionality.

---

**Theorem 5.** Let \(P_1, P_2, D\) and \(U\) be processes with \(I_{P_1} = I_D\), \(O_D = I_{P_2}\), \(O_{P_2} = I_U\) and \(O_U = O_{P_1}\) and such that \(D\) has a left inverse \(D'\) and \(U\) a right inverse \(U'\). Let \(m \in (\text{Secret} \cup \text{Keys}) \setminus \bigcup_{Q \in \{D', U'\}} (S_Q \cup K_Q)\).

If \(P_1\) preserves the secrecy of \(m\) and \(P_1 \stackrel{(D,U)}{\sim} P_2\) then \(P_2\) preserves the secrecy of \(m\).

**Proof:** using formal semantics.

**Above example:** Rely-guarantee formalization shows that secure composition impossible.

---

\(^1\) C.B. Jones 1981
Evolution-based Verification: Example

Secure information flow:

\[ \text{state}_{\text{current}} \approx \text{pub} \text{state}'_{\text{current}} \Rightarrow \text{state}_{\text{current}.t(m)} \approx \text{pub} \text{state}'_{\text{current}.t(m)} \]

Evolution \( M \rightarrow M' \): Only consider states for which:

- \( \text{state}_{\text{current}} \approx \text{pub} \text{state}'_{\text{current}} \) in \( M' \), but not in \( M \), or
- \( \text{state}_{\text{current}.t(m)} \approx \text{pub} \text{state}'_{\text{current}.t(m)} \) in \( M \), but not in \( M' \).

Example: \( \text{wm}(0).\text{rx}() \approx \text{pub} \text{wm}(1000).\text{rx}() \) in \( M \) but not in \( M' \).

\( \Rightarrow M' \) violates secure information flow.
Secure Software Evolution

- Challenge and Scientific Basis
- Secure Software Evolution
  - General Approach
  - Preservation Results
- Validation
Validation

- **Correctness**: Based on formal semantics.
- **Completeness**: Evolution: sequence of deletions / modifications / additions of system elements.

Performance gain **maximal** when **difference \(<<\) software**. Example result:

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

Assumption satisfied e.g. for:

- **Maintenance** of stable software versions
- **Verification** **integrated** with development
Current Project: Beyond One-Shot Security: Keeping Information Systems Secure through Environment-Driven Knowledge Evolution

Joint project with Stefan Gärtner, Kurt Schneider (Univ. Hannover) and Jens Bürger, Thomas Ruhroth, Johannes Zweihoff (TU Dortmund)

Goals

- Systematic co-evolution of knowledge and software for security.
- Develop techniques, tools, and processes that support security requirements and design analysis techniques for evolving, long-living systems in order to ensure "lifelong" compliance to security requirements.

Approach

Some publications

Conclusion: Secure Software Evolution

Integrate security verification with evolution:
- **Reuse** verification results by conditions for security preservation (e.g. refinement, refactoring, patterns, modularization).

→ **Evolution-based verification:**
- Model-code **traceability** at evolution
- Evolution-based **static analysis** and **run-time verification**.

**Validation:** Successful application projects.
- Significant **performance gains**.

**Current work:** **Security certification for clouds.**