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

- Weave in
- Analyze against
- Reverse Engin.
- Configure

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

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

Requirements -> Models -> Source Code

Verify. -> Gener. -> Configurations
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!

Jan Jürjens, TU Munich/Open Univ: Sec.Analysis of Crypto-based Java Programs
Crypto-based Software (e.g. Protocols)

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

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

(cf. [Dolev, Yao 1982])
Approximate adversary knowledge set from above:

Predicate $\textit{knows}(E)$ 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 $\textit{knows}(s)$ from automatically generated formulas using automatic theorem prover.
First-order Logic: Basic Rules

Define $\text{knows}(E)$ for any $E$ initially known to adversary.

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

For evolving adversary knowledge define

$$\forall E_1, E_2. (\text{knows}(E_1) \land \text{knows}(E_2) \Rightarrow \text{knows}(\{E_1\}_{E_2}) \land \text{knows}(\text{Dec}_{E_2}(E_1)) \land \ldots)$$
Models from Code

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

Transform to state machine:

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

where action can be outpattern or localvar:=value.
Graph transition

\[ TR1 = (\text{in}(\text{msg\_in}), \text{cond}(\text{msg\_in}), \text{out}(\text{msg\_out})) \]

followed by \( TR2 \) gives predicate

\[ \text{PRED}(TR1) = \forall \text{msg\_in}. \ [\text{knows}((\text{msg\_in}) \land \text{cond}(\text{msg\_in})) \Rightarrow \text{knows}(\text{msg\_out}) \land \text{PRED}(TR2)] \]

Verify with automated FOL prover (e-SETHEO, SPASS, …), e.g. whether can derive \( \text{knows}(s) \) from generated formulas.
Example: TLS Variant

Presented at IEEE Infocom 1999.

Goal: send secret protected by session key using fewer server resources.
Example: Translation to Logic

\[ \text{knows}(N) \land \text{knows}(K_C) \land \text{knows}(\text{Sign}_{K_C^{-1}}(C::K_C)) \]

\[ \forall \text{init}_1, \text{init}_2, \text{init}_3. [\text{knows}(\text{init}_1) \land \text{knows}(\text{init}_2) \land \text{knows}(\text{init}_3) \land \text{snd}(\text{Ext}_{\text{init}_2}(\text{init}_3)) = \text{init}_2] \]

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

\[ \forall \text{resp}_1, \text{resp}_2, \ldots, [\ldots \Rightarrow \ldots] \]
Analysis

Check whether can derive $\text{knows}(s)$ e.g. using e-Setheo.

Surprise: Yes!

→ Protocol does not preserve secrecy of $s$.

Why? Use Prolog-based attack generator.
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:
```java
while (true)
{
    k = a + 1;
    a = b + k;
    b = b + 1;
}
```

SSA:
```java
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 \( \text{PRED}(Pi) \) as above (for each path \( i \))

\( \psi \) 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).

\[
[p \leftarrow \text{AF}(\text{ext}_{k_{ca}} (c_s), S)]
\]
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 \( \text{derive}(L,C,E) \) 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.
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

- Sent and received data
- Implement-ation
  - java

Elements of connections

- Defined during model creation
- Backtrace assignments
- compare meaning!

"meaning"

Find

Has

Equal?

Jessie – using RSA & Server authentication

[with David Kirscheneder]
<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;&gt; 10) &amp; 0xFF)</td>
</tr>
<tr>
<td></td>
<td>((gmtUnixTime &gt;&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;&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; ... 
}

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

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

Handshake.write()

Random.write()

call of OutputStream.write()

traverse CFG

SSLSocket.doClientHandshake()

ClientHello.write()
```java
msg = Handshake.read(din, certType);
session.trustManager.checkServerTrusted(peerCerts, suite.getAuthType());
try {
    msg = new Handshake(Handshake.Type.CLIENT_KEY_EXCHANGE, ckex);
    msg.write(dout, version);
} catch (Exception e) {
    // only possible way without throwing exception
    session.trustManager.checkServerTrusted(peerCerts, suite.getAuthType());
}
```
Man-in-the-Middle Attack

\[ N_i::K_C::\text{Sign}_{K_C}^{-1}(C::K_C) \]
\[ C \rightarrow A \rightarrow S \]
\[ \{ \text{Sign}_{K_S}^{-1}(K_j::N_i) \}_{K_A}::\text{Sign}_{K_{CA}}^{-1}(S::K_S) \]
\[ N_i::K_A::\text{Sign}_{K_A}^{-1}(C::K_A) \]
\[ \{ s \}_{K_j} \]
\[ C \leftarrow A \leftarrow S \]
\[ \{ s \}_{K_j} \]

\[ \{ \text{Sign}_{K_S}^{-1}(K_j::N_i) \}_{K_C}::\text{Sign}_{K_{CA}}^{-1}(S::K_S) \]
The Fix

\[ K'' := \text{snd}(\text{Ext}_{K_{CA}}(\text{arg}_C,1,2)) \]
\[ k := \text{fst}(\text{Ext}_{K''}(\text{Dec}_{K_C^{-1}}(\text{arg}_C,1,1))) \]
\[ \text{fst}(\text{Ext}_{K_{CA}}(\text{arg}_C,1,2)) = S \land \]
\[ \text{snd}(\text{Ext}_{K_{CA}}(\text{arg}_C,1,2)) = N_i \land \]
\[ \text{thd}(\text{Ext}_{K_S}(\text{Dec}_{K_C^{-1}}(\text{arg}_C,1,1))) = K_C \]

**e-Setheo:** Proof that \textit{knows(s)} not derivable.

Note \textit{completeness} of FOL (but also undecidability).
Verification of Guards in Code

*send*: represents send command

*\textit{g}*: FOL formula with symbols *msg*\textsubscript{n} representing \(n\)\textsuperscript{th} argument of message received before program fragment *p* is executed

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

\[\text{write } p \models g \text{ for } [\text{true}] \ p \models g.\]

\[\text{[d] if c then } p \text{ else } q \models g (c \wedge d \Rightarrow g, \text{ no send in } q)\]
Jan Jürjens, TU Munich/Open Univ: Sec.Analysis of Crypto-based Java Programs

Model Verification

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

... 

\( \)

... 

... 

\( \)

... 

\$\text{-------------------------- Conjecture --} \$

\text{input_formula(attack, conjecture, ( \text{knows(mastersecret)})}).

[FASE05, ICSE05, ICSE06]

analyzing results ... model found/total failure

time limit information: 19 total / 18 strategy
(leave wrapper).
task myUML_PID1491 on atbroy1 has status SUCCESS
(model found by strategy 300) consuming 1 seconds
deleting temporary files.
e-SETHEO done. exiting
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
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

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 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]
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.

**Similar:** Established **composability** for certain security requirements under suitable assumptions. [FME01] [Concur01]
Layered Security Protocols

System layer on top uses security services below.

Theorem: Yes, under suitable conditions.

Security properties additive? [Safecomp03]

Layered Security Protocols

Confidentiality, integrity, server authenticity + client authenticity = confidentialitiy, ... + client authenticity

Confidentiality, integrity, server authenticity

Client authenticity