Towards Verified Crypto-Protocol Implementations: The Java Secure Sockets Extension

Jan Jürjens

Computing Department, Open University, GB

[from 1 Oct 2008 also visiting at: Microsoft Research (Cambridge)]

http://www.jurjens.de/jan J.Jurjens@open.ac.uk
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
- Reif, Schellhorn et al (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) \(\Rightarrow\) 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 et al.): scalability wrt. code size/complexity; sophistication of properties (security).

⇒ Develop specialized verification approach.
Context: Model-based Security Engineering

Requirements

Weave in

Analyze against

(UML) Models

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

Verify.

Gener.

Generate/Verify

Configurations

Configure

Source Code

→ Long-term goal: Tool-supported, theoretically sound, efficient automated security design & analysis.
Security Analysis in First-order Logic

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.

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

Formal foundations using streams. [ICSE05]
Example TLS Variant [IEEE Infocom 1999]

knows(N) \land \text{knows}(K_C) \land \text{knows}(Sign_{K_{C^{-1}}}(C::K_C)) \\
\land \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)\}_{\ldots}) \land [\text{knows}(\text{Sign}\ldots)] \\
\land \forall \text{resp}_1, \text{resp}_2. [... \Rightarrow ...]]
Analysis

Check whether can derive $knows(s)$ e.g. using SPASS/e-Setheo.

Surprise: Yes!

→ Protocol does not preserve secrecy of $s$.

Why? Use Prolog-based attack generator.
Man-in-the-Middle Attack

\[
\begin{align*}
N_i::K_C::Sign_{K_C^{-1}}(C::K_C) & \quad N_i::K_A::Sign_{K_A^{-1}}(C::K_A) \\
C & \quad \rightarrow \quad A & \quad \rightarrow \quad S \\
\{Sign_{K_S^{-1}}(K_j::N_i)\}_K & \quad \rightarrow \quad S \\
A & \quad \leftarrow \quad C \\
\{Sign_{K_C^{-1}}(K_j::N_i)\}_K & \quad \rightarrow \quad S \\
C & \quad \leftarrow \quad A \\
\{s\}_K_j & \quad \rightarrow \quad S \\
\{s\}_K_j & \quad \rightarrow \quad S
\end{align*}
\]
The Fix

e-Setheo: Proof that *knows*(s) not derivable.

Note completeness of FOL (but also undecidability).
Interface: Model vs. Implementation

"meaning"

Backtrace assignments

Sent and received data

Implement -ation

Elements of connections

Find

Consistent?

has

Jessie – using RSA & Server authentication

Jan Jürjens, Open Univ.: Towards Verifi
Models vs Code

Generate control flow graph (e.g. CodeLogic).

Transform to labelled transition state machine:
\[
\text{trans}(\text{state}, \text{inpattern}, \text{condition}, \text{action}, \text{nextstate})
\]
where action can be outpattern or localvar:=value.

Need to link concrete data to abstract symbols.
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.
Interface spec of SSL

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

II) Check guards enforced
<table>
<thead>
<tr>
<th>in Model</th>
<th>Send: ClientHello</th>
<th>by OutputStream.write in</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>type.getValue()</td>
<td>Handshake.write</td>
</tr>
<tr>
<td></td>
<td>(bout.size() &gt;&gt;&gt; 16 &amp; 0xFF)</td>
<td>Handshake.write</td>
</tr>
<tr>
<td></td>
<td>(bout.size() &gt;&gt;&gt; 8 &amp; 0xFF)</td>
<td>Handshake.write</td>
</tr>
<tr>
<td></td>
<td>(bout.size() &amp; 0xFF)</td>
<td>Handshake.write</td>
</tr>
<tr>
<td>Pver</td>
<td>major</td>
<td>ProtocolVersion.write</td>
</tr>
<tr>
<td></td>
<td>minor</td>
<td>ProtocolVersion.write</td>
</tr>
<tr>
<td></td>
<td>((gmtUnixTime &gt;&gt;&gt; 24) &amp; 0xFF)</td>
<td>Random.write</td>
</tr>
<tr>
<td></td>
<td>((gmtUnixTime &gt;&gt;&gt; 16) &amp; 0xFF)</td>
<td>Random.write</td>
</tr>
<tr>
<td></td>
<td>((gmtUnixTime &gt;&gt;&gt; 8) &amp; 0xFF)</td>
<td>Random.write</td>
</tr>
<tr>
<td></td>
<td>(gmtUnixTime &amp; 0xFF)</td>
<td>Random.write</td>
</tr>
<tr>
<td>R_c</td>
<td>randomBytes</td>
<td>ClientHello.write</td>
</tr>
<tr>
<td></td>
<td>sessionId.length</td>
<td>ClientHello.write</td>
</tr>
<tr>
<td>Sid</td>
<td>sessionId</td>
<td>ClientHello.write</td>
</tr>
<tr>
<td></td>
<td>((suites.size() &lt;&lt; 1) &gt;&gt;&gt; 8 &amp; 0xFF)</td>
<td>ClientHello.write</td>
</tr>
<tr>
<td></td>
<td>((suites.size() &lt;&lt; 1) &amp; 0xFF)</td>
<td>ClientHello.write</td>
</tr>
<tr>
<td>Ciph[]</td>
<td>id[]</td>
<td>CipherSuite.write</td>
</tr>
<tr>
<td></td>
<td>comp.size()</td>
<td>ClientHello.write</td>
</tr>
<tr>
<td>Comp[]</td>
<td>comp[2]</td>
<td>ClientHello.write</td>
</tr>
</tbody>
</table>

Data vs. Symbols (SSL project Jessie)
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).
Sending Messages

SSLSocket.doClientHandshake()

ClientHello.write()

Random.write()

traverse CFG

call of OutputStream.write()
Checking Guards

Guard $g$ enforced by code?
b) Generate runtime check for $g$ at $q$ from diagram:
  + simple, effective.
  -- performance, code change

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).
Run-time Verification using LTL

Example: Client sends out `ClientKeyExchange` message eventually but only after received `Certificate` message, and check is positive. Improves on Schneider’s security automata.

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

[BCS-Visions08 w. Bauer / Yu]

not safety but co-safety
msg = Handshake.read(din, certType);

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

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

Static Verification

only possible way without throwing exception

[ASE06]
Verification of Guards in Code

**send**: represents send command

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

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

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

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

\[ \frac{[d]p \models g}{[d]p; q \models g} \quad (d \Rightarrow c) \]

\[ \frac{[d]q \models g}{[d]p \models g} \quad (d \Rightarrow \neg c) \]

\[ \frac{[d]p \models g}{x := e; p \models g} \quad d \Rightarrow x = e \]
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:
while (true)
{
    k = a + 1;
    a = b + k;
    b = b + 1;
}

SSA:
while (true)
{
    k = a0 + 1;
    a1 = b0 + k;
    b1 = b0 + 1;
}

TPTP:
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 $\psi \implies \phi$ together by conjunction.
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 $I$ 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 $I$

When considering single protocol run, can construct finite set of such statements similar to FOL formulas from security analysis.
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.
Conclusion

Seemingly first attempt at formally based security verification for crypto-based Java legacy implementations.

Goals: Emphasis on automation, achieve efficiency using abstraction tailored to verification problem. Experiences so far encouraging but still many challenges – collaboration welcome!

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

More information (papers, slides, tool etc.):
http://www.jurjens.de/jan

J.Jurjens@open.ac.uk