Tools for Traceable Security Verification

Jan Jürjens*, Yijun Yu, Andreas Bauer

Computing Department
The Open University, GB

Computer Sciences Lab
The Australian National University

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

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) ➔ 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, Soot et al.): scalability wrt. code size / complexity; sophistication of properties (security).

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

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

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

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

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.

Formal foundations using streams.

[ICSE05]

[JLAP08]
Example TLS Variant [IEEE Infocom 1999]

\[
\begin{align*}
\text{knows}(N) & \land \text{knows}(K_C) \land \text{knows}(\text{Sign}_{K_C^{-1}}(C::K_C)) \\
\land \forall \text{init}_1, \text{init}_2, \text{init}_3. \left[ \text{knows}(\text{init}_1) \land \text{knows}(\text{init}_2) \land \text{knows}(\text{init}_3) \land \text{snd}(\text{Ext}_{\text{init}_2}(\text{init}_3)) = \text{init}_2 \right] \\
\Rightarrow \text{knows}(\{\text{Sign}_{K_S^{-1}}(...}\}) & \land \left[ \text{knows}(\text{Sign}...) \right] \\
\land \forall \text{resp}_1, \text{resp}_2. \left[ [... \Rightarrow ...] \right]
\end{align*}
\]
Analysis

Check whether can derive $\text{knows}(s)$ e.g. using ATP for FOL.

Surprise: Yes!

→ Protocol does not preserve secrecy of $s$.

Why? Use Prolog-based attack generator.
The Fix

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

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

Note completeness of FOL (but also undecidability).
Linking Models to Code

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

II) Check guards enforced
Static Verification

- Guard \( g \) enforced?
- Testing (vs. crypto?)
- Automated formal local verification: Conditionals between \( p \) and \( q \) must imply \( g \).

\[
\text{msg} = \text{Handshake.read}(\text{din}, \text{certType});
\]

\[
\text{session.trustManager.checkServerTrusted}(\text{peerCerts}, \text{suite.getAuthType()});
\]

\[
\text{msg} = \text{new Handshake(Handshake.Type.CLIENT_KEY_EXCHANGE, ckex)};
\]

\[
\text{msg.write}(\text{dout}, \text{version});
\]

Only path without exception.

\[
\text{try}
\]

\[
\text{catch}
\]

[[equal(fst(ex_{K \text{CA}}(c_s)), S)]]
Vulnerability in SSL implementation

Analyzed open-source 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.
Another Problem

How do I know the running implementation is still secure after deployment?

• Does system model capture all relevant aspects about a system?

• Are assumptions about influences from a system's operational environment reflected adequately?

• Are the abstractions that need to be made to enable automated static verification of non-trivial systems faithful wrt the verification result?

→ Run-time verification.
Runtime Verification using Monitors

Dynamic verification technique on the actual system.

Essentially a symbiosis of model-checking and testing.

“Lazy model-checking”: only check the system traces which are executed, when they are executed.
Formal underpinnings

- System (safety) property, $\varphi$ specified in terms of linear time temporal logic [Pnu77]:
  $$\varphi ::= true \mid p \mid \neg p \mid \varphi \circ p \varphi \mid \varphi U \varphi \mid X \varphi \quad (p \in AP)$$

- Continuous interpretation of $\varphi$ over sequence of system events (behaviours), $u \in (2^{AP})^*$

- **Automatic monitor generation:** “Inspired” by translation of LTL to Büchi-automata

  $$\varphi \rightarrow \mathcal{BA}_\varphi \text{ s.t. } L(\mathcal{BA}_\varphi) = L(\varphi)$$
Monitoring-friendly LTL semantics

3-valued semantics:

\[ [u \models \varphi] = \begin{cases} \top & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \models \varphi \\ \bot & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \not\models \varphi \\ \ ? & \text{otherwise} \end{cases} \]

Gives finite-state machines for detecting *minimal* bad prefixes:

![Diagram](diagram.png)

*inconclusive*

*false*

*true*

*Predictiveness*

*inconclusive*
ClientKeyExchange

Client will not send out ClientKeyExchange message until has received Certificate message and check is positive, and then sends it out.

\[
\begin{align*}
q_1 \quad \text{“⊥”} & \quad \text{ClientKeyExchange}_S \\
\neg \text{Certificate}_R & \quad \text{“?”} \\
\text{Certificate}_R & \quad q_2 \quad \text{“⊤”}
\end{align*}
\]

not safety but co-safety

Figure 1: FSM $\neg \text{ClientKeyExchange}_S \cup \text{Certificate}_R$. 
Client Transport Data

Client will not send any transport data before has checked that MD5 hash received in Server`s **Finished** message is equal to MD5 created by Client (and correspondingly for SHA hash).

\[ \varphi_3 = \neg DataW((MD5(\text{Finished}_R) = MD5(\text{Finished}_S))) \]

not co-safety but safety
Server Finished

Server will not send **Finished** message before MD5 received in Client`s **Finished** message equal to MD5 created by server. Then sends out eventually.

NB: Improves on Schneider’s security automata.

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

Not safety nor co-safety
Some Applications

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.
Conclusion

Seemingly first approach to run-time security verification for crypto-based Java implementations.

Integrated with static verification of UMLsec models.

Exceeds previous approaches such as Fred Schneider’s security automata in expressivity.

Future work: collaboration with Andy Gordon (MSRC) on verifying crypto protocol implementations in C.
Overview

Security Engineering

IT Security

Dependable Systems Development

Fault-tolerance

Reliability

Real-time

Security

Management

Engineering

Risk Assessment

Permissions

Business Processes

Security Investment

Firewall Configurations
Questions?

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

J.Jurjens@open.ac.uk