Using Interface Specifications for Verifying Cryptoprotocol Implementations

Jan Jürjens

Computing Department
The Open University, GB

J.Jurjens@open.ac.uk

http://www.jurjens.de/jan
Crypto-Protocol Analysis

State of the affairs:

A *lot* of very successful work in formally verifying abstract models of crypto-protocol design.

- virtually every formal method has been applied
- seemingly more people working on verification than on designing protocols
- efficient tool-support usable by academics or specialists
- sometimes used at industrial size protocols (usually by tool developers themselves)

(Almost) solves the problem whether design is secure.
Problem

How do I know a crypto-protocol implementation is secure?

Possible solution:
Verify design model, 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 verified design or directly against security requirements.
So far applied to self-written or restricted code.
Surprisingly few approaches so far:
• J. Jürjens, M. Yampolski (ASE´05): 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
May reduce first problem. How about other two?
Towards Verifying Legacy Implementations

Goal: Verify implementation created independently.

Options:

3) 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. Get some verification result already in design phase (for non-legacy implementations).
Background: Model-based Security Engineering

- **Long-term goal:** Tool-supported, theoretically sound, efficient automated security design & analysis.

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

```
Weave in (UML) Models

Analyze against Configurations

Generate/ Verify Source Code

Verify, Gener. Configure

Requirements
```

- Code-/ Testgen.
Why Behavioural Interfaces?

Goal: verify implementations of significant complexity automatically and exhaustively against non-trivial requirements.

Have software model-checkers, but so far not used for very complex implementations and very sophisticated requirements (e.g. involving Dolev-Yao type attacker models).

Do have powerful type checkers.

Idea: push the envelope by introducing behaviour into types \(\Rightarrow\) behavioural interfaces

Long line of foundational work (rely/guarantee etc.), some tools (SLAM, Blast)
Interface based Security Analysis in FOL

Based on usual Dolev-Yao model. **Approximate adversary knowledge** set from above:

Predicate $\text{knows}(E)$ 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 $\text{knows}(s)$ from model-generated formulas using automatic theorem prover.  

[ICSE05]
\[ \text{Interface to FOL} \]

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

Jessie – using RSA & Server authentication

Check whether can derive \textit{knows(s)}. If yes, generate attack scenario.
If no, \textit{s} secret (wrt our attacker).

```
... 
!(
  \text{knows(ArgC\_3)}
  & \text{equal(fst(ArgC\_3), type_serverkeyexchange))}
  & \text{equal(snd(ext(snd(snd(ArgC\_3)), k\_ca)), skey))}
  & \text{equal(snd(ext(snd(snd(ArgC\_2), k\_ca)), fst(snd(ArgC\_3))))})
)
=>
  ((\text{knows(ArgC\_4\_1)}
    & \text{equal(ArgC\_4\_1, type_serverhelloworld)})
  =>
    ( [ true \& equal(ClientKeyExchange, enc(premasterkey, skey)) ]
  )
...
%----------------------------- Conjecture --
input_formula(attack,conjecture, ( 
  \text{knows(mastersecret)} ).
```

analyzing results ...

model found/total failure

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

Jan Jürjens, OU: Using Interface Specifications for Verifying Cryptoprotocols
Just an Exercise in Code Verification?

State of the art in practical code verification: execution exploration by testing (possibly generated from models). 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.

General approaches to formal software verification exist (Isabelle et al), but limited use by (civilian) software engineers, and usually not for sophisticated properties like Dolev-Yao security.

➡ Develop specialized verification approach.
Interface: Model vs. Implementation

"meaning"

Backtrace assignments

Sent and received data

Implement\-ation

Elements of connections

Elements of connections

Implement\-ation

"meaning"

Define during model creation

Compare meaning!

Find

Has

Consistent?

Jessie – using RSA & Server authentication

Jan Jürjens, OU: Using Interface Specifications
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).
Example

Sending a protocol message (e.g. ClientHello):
• create the clientHello object with appropriate message parameters
• create the message object \texttt{msg} by giving the clientHello object as an argument
• call the write method at the \texttt{msg} object

ClientHello clientHello = new ClientHello(session.protocol, clientRandom, sessionId, session.enabledSuites, comp, extensions);
Handshake msg = new Handshake(Handshake.Type.CLIENT_HELLO, clientHello);
msg.write (dout, version);
Example: Interface spec of SSL

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

II) Check guards enforced
Checking Guards

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 abstractions for crypto).

d) Automated formal local verification: conditionals between $p$ and $q$ logically imply $g$ (uses Prolog).

[ICFEM02]

[ASE06]
msg = Handshake.read(din, certType);

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

only possible way without throwing exception

msg = new Handshake(Handshake.Type.CLIENT_KEY_EXCHANGE, ckex);
msg.write (dout, version);
Modular Verification with Interfaces

For program fragment p implementing a given interface, generate set of statements 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.
Modular Verification: Formalisation

**send**: represents send command

\( g \): FOL formula with symbols \( msg_n \) representing \( n^{th} \) argument of message received before program fragment \( p \) is executed

\([d] \ p \vdash g : g \) checked in any execution of \( p \) initially satisfying \( d \) before any send

write \( p \vdash g \) for \([\text{true}] \ p \vdash g\).

\([d] \ \text{if } \ c \ \text{then } p \ \text{else } q \vdash g \) \( (c \land d \Rightarrow g, \text{ no send in } q) \)
Modular Verification: Some Rules

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

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

\[
[d] p \models g \quad (d \Rightarrow c)
\]

\[
[d] p \models g \quad [d] p ; q \models g
\]

\[
[d] q \models g \quad (d \Rightarrow \neg c)
\]

\[
[d] q \models g \quad [d'] p \models g \quad d' \Rightarrow d
\]

\[
[d] p \models g \quad x := e ; p \models g \quad d \Rightarrow x = e
\]
Tool Support

Also:
- configuration analysis: (user permissions, firewall rules/policies)
- code traceability (with Yijun Yu)

Open-source
Some 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.

Use interface specification to make verification of large-scale implementations feasible.

Goals: Emphasis on automation, reach efficiency using abstraction tailored to verification problem.

Experiences so far encouraging.

Still many challenges to address – collaboration always welcome!
Questions?

More information (papers, slides, tool etc.):
http://www.jurjens.de/jan
J.Jurjens@open.ac.uk