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

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]
Example: Translation to Logic

knows(N) ∧ knows(K_C) ∧ knows(Sign_{K_C^{-1}}(C::K_C))
∧ ∀init_1, init_2, init_3. [knows(init_1) ∧ knows(init_2) ∧
knows(init_3) ∧ snd(Ext_{init_2}(init_3)) = init_2
⇒ knows({Sign_{K_{S^{-1}}}(\ldots)}_{\ldots}) ∧ [knows(Sign_{\ldots})]
∧ ∀resp_1, resp_2. [...⇒...]]
**Model Verification**

Check whether can derive \textit{knows}(s).

If yes, generate attack scenario.

If no, s secret (wrt our attacker).

```
...  
| (  
| \text{knows} (\text{ArgC}_3)  
| \& \text{equal} (\text{fst} (\text{ArgC}_3), \text{type_serverkeyexchange}))  
| \& \text{equal} (\text{snd} (\text{ext} (\text{snd} (\text{snd} (\text{ArgC}_3)), \text{k_ca})), \text{skey}))  
| \& \text{equal} (\text{snd} (\text{ext} (\text{snd} (\text{ArgC}_2), \text{k_ca})), \text{fst} (\text{snd} (\text{ArgC}_3)))  
| )  
| \Rightarrow  
| (\text{knows} (\text{ArgC}_4_1)  
| \& \text{equal} (\text{ArgC}_4_1, \text{type_serverhelloworld}))  
| \Rightarrow  
| ( \{  
| \text{true} \& \text{equal} (\text{ClientKeyExchange}, \text{enc} (\text{premasterkey}, \text{skey}))  
| \}  
| ...  
| \%----------------------------- Conjecture --  
| input_formula (attack, conjecture, (  
| | \text{knows} (\text{mastersecret}) ))).  
```

**ATP**

analyzing results ...
```
model found/total failure
```
time limit information: 19 total / 18 strategy
(levering wrapper).
task myUML_PID1491 on atbroyl has status SUCCESS
(model found by strategy 300) consuming 1 seconds
deleting temporary files.
e-SETHEO done. exiting
Model vs 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)
+ applicable to legacy systems

⇒ Verify both!

(NB: No previous work eg for legacy crypto protocols.)
State of the art in code verification in practice: 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

Sent and received data

```
Implement .java
```

Elements of connections

```
Backtrace assignments
```

Jan Jürjens, Open Univ.: Linking Cryptoprotocol Verification to Reality
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).
Example

Sending a protocol message (e.g. ClientHello):
• create the clientHello object with appropriate message parameters
• create the message object msg by giving the clientHello object as an argument
• call the write method at the 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). E.g. Bandera.
msg = Handshake.read(din, certType);

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

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

only possible way without throwing exception
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.
Modular Verification: Formalisation

**send**: represents send command

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

\[
[d] \ p \vdash g : g \text{ checked in any execution of } p
\]

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

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

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

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

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

\[ x := e ; p \models g \quad d \Rightarrow x = e \quad d' \Rightarrow d \]
JSSE / Jessie

• Java Secure Sockets Extension contains implementation of SSL.
• Open-source clean-room reimplementaton Jessie.
• Case-study: applied above 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.
Tool Support

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

Open-source
Linking to Reality II …

Analyzed designs / implementations / configurations e.g. for

• Common Electronic Purse Specs
• biometric authentication system
• Internet banking security architecture
• intranet information system
• mobile communications architectures
Intranet Information System

MetaSearch Engine: **Personalized search** in company intranet (including **password protected**).

Some documents highly **security-critical**. [ICSE07]

More than 1,000 potential users, index 280,000 documents, allow 20,000 queries per day.

Seamlessly integrated in **enterprise-wide security reference architecture**. Provides security services to applications, including **user authentication**, **role-based access control**, **global single-sign-on** and hook-up of **new security apps**.

Used MBSE to demonstrate architecture is secure.
Mobile Communications

• Application of Model-based Security Assurance at Mobile Communication Systems at O2 (Germany) [ICSE08]

• All 62 relevant security requirements derived from the security policy could successfully be established using the approach
Future Work

• PhD project on “Verifying Implementations of Crypto-Protocols in C” (MSR Cambridge / Andy Gordon). Hiring now 😊.
• Goal: Verify implementations in practical use
• Most crypto protocols implemented in C (openSSL, openSSH, Kerberos, …)
• Initial experiments with SLAM, Blast (some technical challenges…)
• Hopeful about VCC
Conclusion

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

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

Experiences so far encouraging.

Still many challenges to address – collaboration always welcome!
Current Projects

• PhD project on Verifying Implementations of Cryptoprotocols in C (MSR Cambridge / A. Gordon)
• RoySoc JIP with TU Munich on Formal Model-based Analysis of Cryptoprotocol Implementations
• RoySoc JIP with NII (Tokyo) on Security Requirements vs Design
• PhD project on IT security risk assessment with Munich Re
• PhD project on Adaptive Security for Ambient Technology
• PhD project on fuzzy reasoning for IT security risks
• PhD project on model-based development for avionics
Questions?

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