Automated Security Verification for Crypto-Protocol Implementations: Verifying the Jessie Project

Jan Jürjens

Department of Computing
The Open University, GB

J.Jurjens@open.ac.uk

http://www.umlsec.org
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 (cf talk yesterday) 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).
History: Model-based Security Engineering

Long-term goal: Tool-supported, theoretically sound, efficient automated security design & analysis.
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.
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]
Cryptographic Expressions I

\( \text{Exp} \): quotient of term algebra generated from sets \( \text{Data}, \text{Keys}, \text{Var} \) of symbols using

- \( _::_ \) (concatenation), \( \text{head}(\_) \), \( \text{tail}(\_) \)
- \( (\_)^{-1} \) (inverse keys)
- \( \{\_\}_\_ \) (encryption)
- \( \text{Dec}(\_) \) (decryption)
- \( \text{Sign}(\_) \) (signing)
- \( \text{Ext}(\_) \) (extracting from signature)

under equations …
Cryptographic Expressions II

\[ \forall E, K. \ Dec_K^{-1}({E}_K) = E \]
\[ \forall E, K. \ Ext_K(Sign_K^{-1}(E)) = E \]
\[ \forall E_1, E_2. \ head(E_1 :: E_2) = E_1 \]
\[ \forall E_1, E_2. \ tail(E_1 :: E_2) = E_2 \]

• Associativity for ::.
Write \( E_1 :: E_2 :: E_3 \) for \( E_1 :: (E_2 :: E_3) \) and \( \text{fst}(E_1 :: E_2) \) for \( \text{head}(E_1 :: E_2) \) etc.

Can include further crypto-specific primitives and laws (XOR, \ldots).
First-order Logic: Basic Rules

Define $knows(E)$ for any $E$ initially known to adversary.

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

For evolving adversary knowledge define

$$\forall E_1, E_2. (knows(E_1) \land knows(E_2) \Rightarrow$$

$$knows(\{E_1\}_{E_2}) \land$$

$$knows(\text{Dec}_{E_2}(E_1)) \land$$

$$\ldots)$$
Models from Code

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

Transform to labelled transition state machine:
\[
\text{trans(state, inpattern, condition, action, nextstate)}
\]

where action can be outpattern or localvar:=value.

Need to link concrete data to abstract symbols.
<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>Rc</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>
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; ... }

2nd parameter of ClientHello write() called by ClientHello.write()

public void write(OutputStream out) throws IOException
{ ... random.write(out); ... }

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

ClientHello clientHello = new ClientHello(..., clientRandom, ...);

Initialization of the used Random object

Random clientRandom = new Random(..., session.random.generateSeed(28));

class SecureRandom (specified in: FIPS 140-2, RFC 1750) of package java.security
Function: generateSeed
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()

ProtocolVersion.write()

Handshake.write()
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

```java
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);
```
Guards

Finally, need to determine the crypto protocol conditions to add as guards to the state machine.

Similarly done by tracking the relevant message calls.
public void checkServerTrusted(X509Certificate[] chain, String authType) throws CertificateException {
    // ...
    checkTrusted(chain, authType);
}

calls checkTrusted()

Guard:
checkServerTrusted()

calls verify() for every member of certificate chain

private void checkTrusted(X509Certificate[] chain, String authType) throws CertificateException
{
    // ...
}

public void verify(PublicKey key, String provider) throws CertificateException, ...
{
    // ...
}

calls doVerify()

private void doVerify(Signature sig, PublicKey key) throws CertificateException, ...
{
    sig.initVerify(key);
    sig.update(tbsCertBytes);
    if (!sig.verify(signature))
    {
        // ... throw new CertificateException
        // "signature not validated";
        // ...
    }
}

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()

Guard:
checkServerTrusted()

calls checkTrusted()
Translate to First Order Logic

State machine transition

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

followed by \( TR2 \) gives predicate

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

Added to axioms. Verify against security conjecture using automated FOL prover (e-SETHEO, SPASS, ...).
Routine: TLS_Client

```c
void TLS_Client (char* secret)
{
    char* Resp_1;
    char* Resp_2;
    // allocate and prepare buffers
    Resp_1 = (char *) malloc(MESSAGEBUFF_MAXLEN);
    Resp_2 = (char *) malloc(MESSAGEBUFF_MAXLEN);
    memset (Resp_1, 0x00, MESSAGEBUFF_MAXLEN);
    memset (Resp_2, 0x00, MESSAGEBUFF_MAXLEN);

    // C->S: Init
    send (n);
    send (k_c);
    send (sign(conc(c, k_c), inv(k_ca)));

    // S->C: Receive Server's respond
    recv (Resp_1);
    recv (Resp_2);

    if ( // Check Guards
        (memcmp(lst(ex(t(Resp_2, k_c))), s, MESSAGEBUFF_MAXLEN) == 0) &
        (memcmp(lst(ex(t(dec(Resp_1, inv(k_c)),
                        ex(t(Resp_2, k_c))))), n, MESSAGEBUFF_MAXLEN) == 0) )
    {
        // C->S: Send Secret
        send (sym enc(secret, lst(ex(t(dec(Resp_1, inv(k_c)),
                                   ex(t(Resp_2, k_c)))))));
        free (Resp_1); // free temporary buffers
    }

    free (Resp_2);
}
```

Routine: TLS_Server

```c
char* Init_1;
char* Init_2;
char* Init_3;
char* k_tmp;
char* EncSecret;
char* RetVal = NULL;
// allocate and prepare buffers
Init_1 = (char *) malloc(MESSAGEBUFF_MAXLEN);
Init_2 = (char *) malloc(MESSAGEBUFF_MAXLEN);
Init_3 = (char *) malloc(MESSAGEBUFF_MAXLEN);
EncSecret = (char *) malloc(MESSAGEBUFF_MAXLEN);
memset (Init_1, 0x00, MESSAGEBUFF_MAXLEN);
memset (Init_2, 0x00, MESSAGEBUFF_MAXLEN);
memset (Init_3, 0x00, MESSAGEBUFF_MAXLEN);
memset (EncSecret, 0x00, MESSAGEBUFF_MAXLEN);

recv (Init_1); // C->S: Receive Init from client
recv (Init_2);
recv (Init_3);

// check guards
if ( (memcmp(lst(ex(t(Init_3, Init_2))),
            Init_2, MESSAGEBUFF_MAXLEN) == 0) )
{
    // generate temporary symmetric key
    k_tmp = kmg (Init_2);

    // S->C: Send Server's respond
    send (enc (sign (conc (k_tmp, Init_1), inv (k_s)), Init_2));

    send (sym enc (s, k_s), inv (k_ca)));

    recv (EncSecret); // C->S: Receive Secret
    RetVal = sym dec (EncSecret, k_tmp); // get secret
}
Free (Init_1); // free buffers
Free (Init_2);
Free (Init_3);
Free (EncSecret);

// return pointer to encrypted secret
// or NULL in case of error
return RetVal;
```
Example: Translation to Logic

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

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

Surprise: Yes!

⇒ Protocol does not preserve secrecy of s.

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

\[ N_i :: K_C :: Sign_{K_C^{-1}}(C :: K_C) \quad \Rightarrow \quad A \]

\[ N_i :: K_A :: Sign_{K_A^{-1}}(C :: K_A) \quad \Rightarrow \quad S \]

\[ \{Sign_{K_S^{-1}}(K_j :: N_i)\}_{K_A} :: Sign_{K_{CA}^{-1}}(S :: K_S) \quad \Rightarrow \quad S \]

\[ \{Sign_{K_S^{-1}}(K_j :: N_i)\}_{K_C} :: Sign_{K_{CA}^{-1}}(S :: K_S) \quad \Rightarrow \quad A \]

\[ \{s\}_{K_j} \quad \Rightarrow \quad A \]

\[ \{s\}_{K_j} \quad \Rightarrow \quad S \]
The Fix

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

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

Note completeness of FOL (but also undecidability).
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:

φ from predicates PRED(Pi) as above (for each path i)

ψ assigning variables according to given interleaving

Join formulas $\psi \Rightarrow \phi$ together by conjunction.
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.
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.
Experiences

Can generate behavioral models from code (e.g. CFGs).

Problem: concrete (no increased abstraction) ➔ understanding + automated verification hard.

Can raise abstraction using annotations (ideally provided by programmers).

Problem: „Reverse engineering“ annotations from legacy implementations is manually intensive.
Verify Code against Models

Assumption: Have textual specification. Then:

• construct interface spec from textual spec
• analyze interface spec for security
• verify that software satisfies interface spec
Interface: Model vs. Implementation

Backtrace assignments

Sent and received data

Implement -ation

Java

define

Elements of connections

find

Has

Consistent?

Jessie – using RSA & Server authentication

Abstract model

By Jan Jürjens, Open Univ.: Automated Verification
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 crypto for inequalities).

\[ [\text{equal}(\text{fst}(\text{ext}_{k_{ca}}(c_8)), S)] \]

\( g \)

\( q \)

\( p \)

\[ \text{Checking Guards} \]

[ICFEM02]

[ASE06]

d) Automated formal local verification: conditionals between \( p \) and \( q \) logically imply \( g \) (using ATP for FOL).
msg = Handshake.read(din, certType);

try

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

catch

only possible way without throwing exception

msg = new Handshake(Handshake.Type.CLIENT_KEY_EXCHANGE, ckex);
msg.write (dout, version);
Verification of Guards in Code

send: represents send command

g: FOL formula with symbols msg_n representing n\textsuperscript{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 [true] p \vDash g.

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

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

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

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

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

\[
[d]q \models g \quad (d \Rightarrow x = e)
\]

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

\[
x := e; p \models g
\]
Jan Jürjens, Open Univ.: Automated Verification for Crypto-Protocol Implementations

[Tool Support]

[UML04, FASE05, ICSE06]
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.
This talk: Feasability study: Jessie project.
Experiences so far encouraging.
Still many challenges to address – collaboration always welcome!
Questions?

More information (papers, slides, tool etc.):
http://www.umlsec.org

J.Jurjens@open.ac.uk
Adversary Interactions

Protocol interaction

\[ TR_1 = (\text{in}(msg_{in}), \text{cond}(msg_{in}), \text{out}(msg_{out})) \]
followed by \( TR_2 \) gives predicate \( PRED(TR_1) = \forall msg_{in}. [\text{knows}(msg_{in}) \land \text{cond}(msg_{in}) \Rightarrow \text{knows}(msg_{out}) \land PRED(TR_2)] \)

Abstraction (e.g. from senders, receivers): find all attacks, may have false positives.

Forall-quantify over iteration counters.
Example: TLS Variant

Presented at IEEE Infocom 1999.

Goal: send secret protected by session key using fewer server resources.