Formally Linking
Security Protocol Models and Implementations

Alfredo Pironti

Microsoft Research – INRIA
http://alfredo.pironti.eu/research/

Politecnico di Torino
Formal Methods for Security Protocol Engineering
28th October, 2011
In a nutshell

- Protect exchange of valuable assets
  - E-banking, e-mail, e-voting...
- Communication protocols based on cryptographic primitives
- Base for reliable, pervasive and dependable services
- Concise, yet difficult to get right
- Exhaustive testing not feasible

The TLS example

- Design and crypto
  - Since 1996, 4 revisions, 1 extension patch, attacks still found
- Implementation
  - About 2-3 OpenSSL security advisories per year
The weapon...
- Unambiguous specifications
- Rigorous proofs of correctness
- Possibly automatic

...without silver bullet
- Full protocol implementations too complex
- Need to consider models instead of programs
  - Dolev-Yao cryptography: $\{M\}_k$ is the symbolic encryption of $M$ under key $k$; $M$ can be retrieved only if $k$ is available
  - High level message exchange: $A \rightarrow B : \{M\}_k$
  - Don’t care about cryptographic algorithms (3DES, AES...) or message format
Bridging the Gap

Motivation

- From protocol informal specification (e.g. TLS 1.2, RFC 5246)
  - Researchers verify formal models ($\pi$-calculus, CSP, HLPSL...)
  - Developers write implementations (C, C#, Java...)
- No link between models and implementations
- No verification of details neglected in models
How to Bridge the Gap

Possible approaches

- Model extraction [ASPIER; Elyjah; fs2pv...]
  - From code, *abstract* a model
  - Doesn’t work yet on legacy code

- Code generation [AGC-C#; AGVI; CACE; Spi2Java...]
  - From model, *refine* to code
  - Need to know modeling language
How to Bridge the Gap

Possible approaches

- Code/model conformance (refinement types) [Jif; F7; F*…]
  - Given code and model, \((\text{type-})\text{check}\) code against model
  - Need to specify the model

- Executable models [JavaSPI]
  - The model is itself an executable program
  - Still a relation with a verifiable model is needed
  - Approach detailed in this lecture
Agenda

Writing Java models
- What is a Java model?
- Model execution, verification, refinement

Linking Java to $\pi$-calculus
- Define semantics of both languages
- Relate the two semantics
- Theorem of simulation
Background

A specific extended pi calculus
- Fixed set of operators
  - Easier to reason about and to implement
- Backs up the Java model

Asymmetric key cryptography

fun PubPart/1.
fun PriPart/1.
fun PriEncrypt/2.
fun PubEncrypt/2.
reduc PriDecrypt(PriPart(Key),PubEncrypt(PubPart(Key),M)) = M.
reduc PubDecrypt(PubPart(Key),PriEncrypt(PriPart(Key),M)) = M.
data true/0.
reduc SignCheck(PriEncrypt(PriPart(Key),M),M,PubPart(Key)) = true.
Background

Shared key cryptography

fun SharedKey/1.
reduc MaterialShareKey(SharedKey(x)) = x.

fun SymEncrypt/2.
reduc SymDecrypt(Key, SymEncrypt(Key, M)) = M.

Hash function

fun H/1.

Diffie-Hellman

fun DHPub/1.
fun DHKey/2.
equation DHKey(x,DHPub(y)) = DHKey(y,DHPub(x)).
The JavaSPI Framework – I

Features

- The model *is* a Java program
  - Executable
  - Not yet a fully fledged implementation
- Standard Java debugger realizes model simulation
- Implementation details as standard Java annotations
  - Finally translated into the “concrete” application
- Formal verification done via ProVerif (Dolev-Yao model)
  - Implies some relationship between Java and ProVerif
Restrictions on Java Code

- All `new` objects must belong to the SpiWrapperSim library
  - Side-effect free library implementing pi calculus terms
- Other accidental restrictions
  - No nested `new` objects (create tmp variables instead)
  - Restricted loops, recursion and Java lists (to ease verification)
Reference Example

The 6.1.5 (sof) Clark-Jacob Protocol

1. $B \rightarrow A : B, R_b$
2. $A \rightarrow B : A, \{H(R_b), R_a, A, K\}_{K_{ab}}$
3. $B \rightarrow A : B, \{H(R_a)\}_K$
The Reference Example in JavaSPI

Application Architecture

- One Java class per protocol actor
- As abstract as the Spi Calculus model

Demo

- Via the Eclipse IDE
  - Code
  - Simulation
  - Generated “prototype”
The Java-ProVerif Tool

- From a JavaSPI model
- To a ProVerif model
- Security queries as Java annotations in the JavaSPI model

Demo

- Security Java annotations
- Resulting ProVerif model
Formal Verification – II

Java-ProVerif Syntax Translation Rules

- Syntactical translation between semantically equivalent languages

A Taste of the Translation Rules

<table>
<thead>
<tr>
<th>Statement</th>
<th>Java</th>
<th>ProVerif</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fresh</td>
<td>Type ( a = \text{new} \text{Type()} );</td>
<td>( \text{new} \ a; )</td>
</tr>
<tr>
<td>Assign</td>
<td>Type ( a = b; )</td>
<td>( \text{let} \ a = b \text{ in} )</td>
</tr>
<tr>
<td>Send</td>
<td>( cAB.\text{send}(a); )</td>
<td>( \text{out}(cAB, a); )</td>
</tr>
<tr>
<td>Receive</td>
<td>Type ( a = cAB.\text{receive(\text{Type.class});} )</td>
<td>( \text{in}(cAB, a); )</td>
</tr>
<tr>
<td>Constructor</td>
<td>Type ( a = \text{new} \text{Type}(b); )</td>
<td>( \text{let} \ a = \text{C}(b) \text{ in} )</td>
</tr>
<tr>
<td>Destructor</td>
<td>Type ( a = b\text{.method(args)} )</td>
<td>( \text{let} \ a = \text{D}(\text{args}) )</td>
</tr>
<tr>
<td>Match case</td>
<td>( \text{if} \ (a.\text{equals}(b)) {...} )</td>
<td>( \text{if} \ a = b \text{ then} {...} )</td>
</tr>
</tbody>
</table>
Concrete Implementation

Java Annotations

- Natural coupling between model and implementation details
- Exploiting annotations scope to reduce burden
  - Sensible default values significantly reduce required annotations
- Java-Java tool
  - From annotated JavaSPI to “concrete” Java implementation
  - With cryptographic algorithms (e.g. AES, SHA-1) and parameters (e.g. key size, TCP port)
  - Marshaling layer (network format and byte order)

Demo

- Not much sense on the “sof” protocol
  - Artificial implementation details, as there is no standard
- Explained in a minute along with bigger SSL example
A Significant Subset of SSL

**Features**

- Both client and server sides
- Restricted scenario
  - Hardcoded ciphersuites
  - DH key exchange
  - Run-time resolution of IVs
- Formally verified (ProVerif)
- Interoperable (OpenSSL)
The JavaSPI Model of (the subset of) SSL

Code Annotations
- Do not burden the code
  - ~60 annotations in total (Client + Server)
  - ~10% of the whole model size

Demo
- Show refinement annotations
- Show and run refined implementation
Linking Java to pi calculus

The problem
- Java code is executed
- ProVerif model is verified

Intuition
- JavaSPI code has restrictions
  - It “looks like” a ProVerif model
- Show that
  - Whatever Java can do
  - There exists a pi calculus process that does the same (and is verified)
- So that we can verify ProVerif models and run Java code
In practice
/* let plain = SymDecrypt(key,cipher) in */
Message plain = cipher.decrypt(key);

Formally
- The other way round
- Function $tr_p : pi \rightarrow Java$
- In: pi calculus process
- Out: well formed (i.e. that “compiles”) Java code
Intuition

- The generated Java program correctly refines the Spi calculus specification it was generated from.
- Labelled Transition Systems (LTS) for Spi calculus and Java
  - Weak simulation relation $S$ between the two
Dynamic Semantic Correctness

**Pi Calculus Operational Semantics**

\[ P\sigma \xrightarrow{L} P'\sigma\sigma' \]

where \( P, P' \) are open pi calculus processes and \( \sigma, \sigma' \) are sets of variable substitutions

**Example**

\[
\begin{align*}
\text{out}(c,M); P & \xrightarrow{c!M} P \\
\text{let plain} = \text{SymDecrypt(Key, SymEncrypt(Key, M))}; P & \xrightarrow{\tau} P\left[M/\text{plain}\right]
\end{align*}
\]
Dynamic Semantic Correctness

Java Operational Semantics

\[ j, \text{Mem} \xrightarrow{\mathcal{L}} j', \text{Mem}' \]

where \( j, j' \) are sequences of Java statements to be executed and \( \text{Mem}, \text{Mem}' \) are partial relations representing the current memory configuration.

Example

\{c.send(M); j\}, Mem \xrightarrow{c!M} \{j\}, Mem
\{Message plain = cipher.decrypt(key); j\}, Mem \xrightarrow{\tau} \{j\}, Mem[plain := M] (if cipher is \{M\}_{key})
Dynamic Semantic Correctness

Simulation Relation $S$

\[ S(P\sigma, (j, Mem)) \iff j = tr_p(P, RI) \land \sigma = Mem \]

for some refinement information $RI$

- The Java code is actually implementing a translated Spi calculus process
Dynamic Semantic Correctness

Theorem [HASE08; COSE10]

If the SpiWrapper library behaves as assumed, then

\[ S(P\sigma, (j, \text{Mem})) \land j, \text{Mem} \xrightarrow{\tau^*} \xrightarrow{\mathcal{L}} j', \text{Mem}' \Rightarrow \]

\[ P\sigma \xrightarrow{\mathcal{L}} P'\sigma' \land S(P'\sigma', (j', \text{Mem}')) \]