

Spi2Java: an MDD Approach for Security Protocols

Alfredo Pironti, Riccardo Sisto
{alfredo.pironti,riccardo.sisto}@polito.it

Politecnico di Torino
Dipartimento di Automatica e Informatica

MSRC
6th November, 2008

Outline

Introduction

- Motivation

- Overall Architecture

The Spi2Java Framework

- Type Inference

- Data Marshalling

- Data Encoding

- Cryptographic Algorithms and Parameters

- Code Generation

Conclusions

- Future Work

Outline

Introduction

Motivation

Overall Architecture

The Spi2Java Framework

Type Inference

Data Marshalling

Data Encoding

Cryptographic Algorithms and Parameters

Code Generation

Conclusions

Future Work

Introduction

Distributed components

- ▶ Require communication protocols to
 - ▶ Cooperate
 - ▶ Exchange data

How to protect exchanged data

- ▶ Cryptographic protocols
- ▶ As simple as possible
- ▶ Still quite difficult to get right
 - ▶ No silver bullet
 - ▶ Code testing
 - ▶ Only check for a few scenarios
 - ▶ Formal methods can tackle some issues

Formal Methods for Security Protocols - I

Pros

- ▶ Can statically check execution scenarios
- ▶ Give a proof about intended program behavior

Cons

- ▶ Only work on abstract representations of problems
 - ▶ Full models are too complex to be handled
- ▶ Big gap between abstract formal models and implementations
- ▶ Security issues may be missed because of this gap

Formal Methods for Security Protocols - II

Code generation

- ▶ Start from an abstract model
 - ▶ Simple to be developed
 - ▶ Can be verified
- ▶ Obtain an implementation that
 - ▶ Refines the abstract model
 - ▶ Is (semi-)automatically generated
 - ▶ Need to fill the semantic gap between abstract and refined models
- ▶ Related work: [Tobler04, Jeon05]
 - ▶ Refinement soundness not proven
 - ▶ Lack of interoperability of implementation

Formal Methods for Security Protocols - III

Model extraction

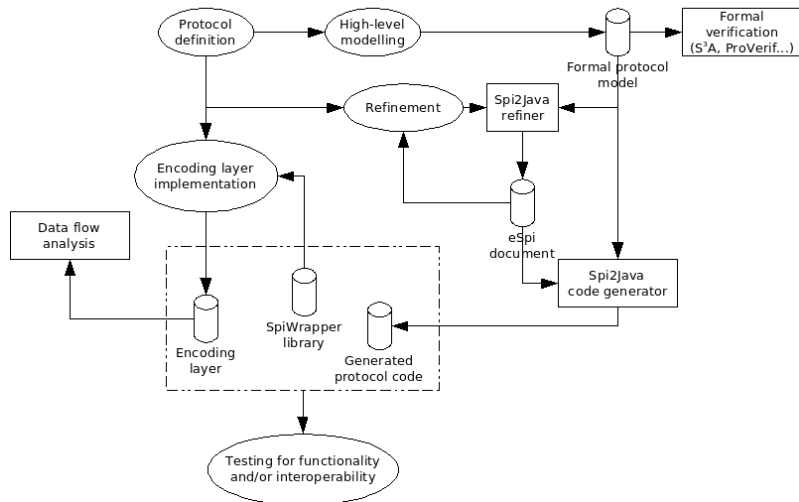
- ▶ Start from a full blown implementation
- ▶ Extract a model to verify the desired properties
- ▶ Abstract unnecessary details
- ▶ Related work: [Jürjens05, Bhargavan06]
 - ▶ Promising approach
 - ▶ Domain specific implementation language, with constraints
 - ▶ Over approximation (false positives)

Formal Methods for Security Protocols - IV

Spi2Java

- ▶ Prototype tool implementation
 - ▶ Supports code generation
- ▶ Abstract model: Spi Calculus
- ▶ Concrete implementation: Java
- ▶ Semantic gap
 - ▶ Assign types to terms
 - ▶ Set cryptographic and configuration parameters
 - ▶ Set marshalling and encoding functions

Spi2Java Workflow - Graphical



Spi2Java Workflow - Textual

- ▶ Design Spi Calculus specification
- ▶ Use Spi2Java refiner to
 - ▶ Semi-automatically assign types to terms
 - ▶ Fill implementation details
- ▶ Write the marshalling and encoding functions
- ▶ Use Spi2Java code generator to
 - ▶ Generate a Java protocol implementation

Outline

Introduction

Motivation

Overall Architecture

The Spi2Java Framework

Type Inference

Data Marshalling

Data Encoding

Cryptographic Algorithms and Parameters

Code Generation

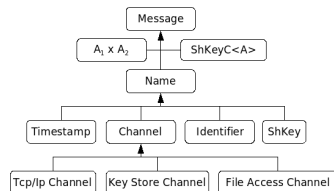
Conclusions

Future Work

The Type System

Types

- ▶ Hierarchically related
 - ▶ One type for each kind of term
 - ▶ Subtypes for different usage of terms
- ▶ Each Spi Calculus term is assigned a type



Type inference

- ▶ Automatically infer principal type
- ▶ Manually refine it
- ▶ Store result in eSpi document

```

<term id="3" name="{M.0}k_0"
type="Shared Key Ciphered">
[...]
```

Marshalling Functions

- ▶ From internal representation to external one, and vice versa
 - ▶ Interoperability

The problem

- ▶ Manually written
 - ▶ Security faults hidden in their wrong manual implementation?

The Abstract Model



- ▶ Formalism based on CSP (Roscoe 1997); protocol model extends the one by Hui and Lowe 2001
- ▶ Actor A is modeled by process P_A
- ▶ The whole system is defined as

$$SYSTEM \triangleq (|||_{A \in \text{Honest}} P_A) || INTRUDER(IK_0)$$

- ▶ The intruder is the medium (Dolev-Yao)
 - ▶ Able to see, modify, forge or drop any message
 - ▶ Has a knowledge derivation relation \vdash

The Refined Model - I



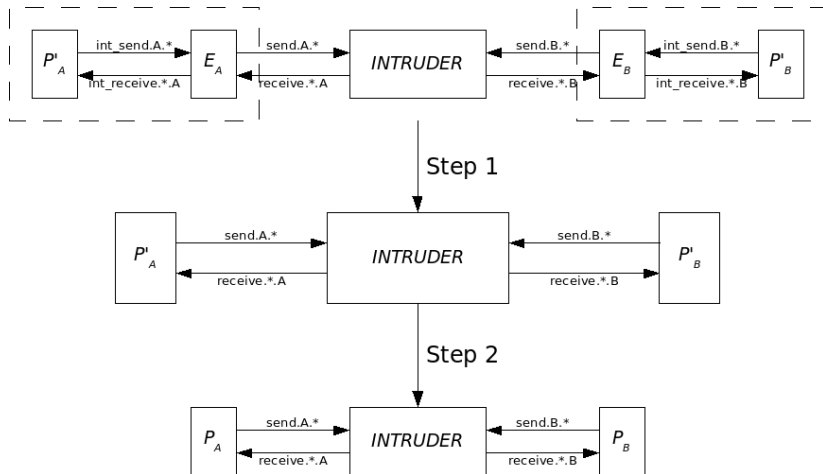
- ▶ Internal and external representation of data
- ▶ Protocol logic and marshalling functions are modeled separately (but coupled)

$$SYSTEM' \triangleq (((|||_{A \in Honest} P'_A) || (|||_{A \in Honest} E_A)) \setminus \{int\}) \\ || INTRUDER(IK'_0)$$

The Refined Model - II

- ▶ Each P'_A acts like P_A
 - ▶ But explicitly sends internal representation of data
- ▶ Each E_A models the marshalling functions
 - ▶ Information is taken only from
 - ▶ Internal representation of data
 - ▶ Marshalling parameters
 - ▶ Marshalling functions are assumed to be
 - ▶ Side-effects free
 - ▶ Memoryless
- ▶ *INTRUDER*(\cdot) is left untouched
 - ▶ Has the same power than in *SYSTEM*

Removing the marshalling functions



Results - I

Any Security Property (that can be defined on traces)

- ▶ Models of the marshalling functions E_A can be safely abstracted
 - ▶ Much of the complexity is in the E_A processes
- ▶ Marshalling parameters are still explicitly represented
- ▶ If the intruder knows all marshalling parameters (reasonable)

No assumptions are made

- ▶ On encoding scheme specification
 - ▶ Even erroneous encoding scheme specifications are safe (though not functional)
- ▶ On implementations
 - ▶ Provided they behave like the model
 - ▶ Memory less and side effect free; Information flow property

Results - II

Secrecy and Authentication

- ▶ Even marshalling parameters can be abstracted
 - ▶ With no other assumptions

Encoding Functions

- ▶ Encoding of data that are going to be ciphered or hashed
 - ▶ E.g. header, padding ...

The problem

- ▶ Manually written
 - ▶ Security faults hidden in their wrong manual implementation?

The Refined Model



- ▶ Protocol logic and encoding functions are modeled separately (but coupled)
- ▶ Again, DEC_A processes are assumed to be memory less and side effect free

$$SYSTEM' \triangleq (|||_{A \in Honest} ((P'_A \parallel DEC_A) \setminus \{priv\})) \parallel INTRUDER(IK'_0)$$

Example

- ▶ Abstract message: $\{M\}_K$
- ▶ Refined message: $\{enc(a, M)\}_{enc(b, K)}$
- ▶ Obtaining plaintext in the refined model is no more atomic
 1. Decipher $enc(a, M)$ from $\{enc(a, M)\}_{enc(b, K)}$
 2. Decode M from $enc(a, M)$
- ▶ Nested cryptographic operations
 - ▶ More iterations are required

In Practice

$$\begin{aligned}
 INTRUDER &\rightarrow P'_A : \{enc(a, M)\}_{enc(b, K)} \\
 P'_A &\rightarrow DEC_A : enc(a, M) \\
 DEC_A &\rightarrow P'_A : M
 \end{aligned}$$

Results - I

Any Security Property (that can be defined on traces)

- ▶ Models of the encoding functions DEC_A can be safely abstracted
 - ▶ Much of the complexity is in the DEC_A processes
- ▶ Encoding parameters are still explicitly represented
- ▶ If each actor locally implements invertible encoding and decoding functions
 - ▶ No assumptions on their correctness with respect to encoding scheme specification
 - ▶ Only local checks, can be done in isolation

Results - II

Secrecy

- ▶ Even encoding parameters can be abstracted
 - ▶ No other assumption required

Authentication

Weaker result: Encoding parameters can be “abstracted” if

- ▶ Implementation of encoding functions is correct with respect to specification (!)
 - ▶ Still viable in some specific cases (e.g. XML parsing for WS-Security implemented in ML)
- ▶ Encoding parameters are explicitly agreed in the authentication events (reasonable, but you do not get completely rid of encoding parameters)

Choosing Cryptographic Algorithms and Parameters - I

The problem

- ▶ One cryptographic operation
 - ▶ E.g. symmetric encryption
- ▶ Many cryptographic algorithms
 - ▶ AES, 3DES, IDEA ...
- ▶ Many Java implementations of one algorithm
 - ▶ SunJCE, Cryptix, Bouncy Castle ...

Choosing Cryptographic Algorithms and Parameters - II

A solution

```
<term id="3" name="{M_0}k_0" type="Shared Key Ciphered">  
  <parameters>  
    <param name="algorithm" type="const">DES</param>  
    <param name="mode" type="const">CBC</param>  
    <param name="padding" type="const">PKCS5Padding</param>  
    <param name="provider" type="const">SunJCE</param>  
  </parameters>  
</term>
```

- ▶ Note: run-time negotiation of cryptographic algorithms and parameters is supported

Spi to Java Translation Rules - I

Idea

- ▶ One Spi Calculus statement translated onto a sequence of Java statements

Example

```
1 /* let (ID_5,M_5) = _w0_4 in */  
2 Identifier ID_5 = (Identifier) _w0_4.getLeft();  
3 Message M_5 = (Message) _w0_4.getRight();
```

Spi to Java Translation Rules - II

SpiWrapper library

- ▶ Translation relies on custom library
- ▶ One Java class for each type of the type system
 - ▶ Implements type behavior (e.g. a pair can be split)
- ▶ Formal specification of expected implementation behavior

Formally

- ▶ Function $tr_p : Spi \rightarrow Java$
- ▶ Input: well typed Spi Calculus process
- ▶ Output: well formed (i.e. that “compiles”) Java code
 - ▶ Uses SpiWrapper library

Spi to Java Translation Properties - I

Static semantics correctness

- ▶ A well typed Spi Calculus process is translated into a well formed Java program
 - ▶ By inductive inspection of the translation function

Dynamic semantics correctness

- ▶ The generated Java program correctly refines the Spi Calculus specification it was generated from
 - ▶ By showing that a weak simulation relation exists
 - ▶ Assumption: SpiWrapper implementation is correct w.r.t. formal specification
 - ▶ Result: The generated code is Dolev-Yao attacks resilient

Spi to Java Translation Properties - II

Dynamic semantics correctness – Formally

$$S((\nu\bar{n})P\sigma, (j, Val, Res)) \wedge j, Val, Res \xrightarrow{\tau^*} \xrightarrow{\mathcal{L}} \xrightarrow{\tau^*} j', Val', Res' \Rightarrow \\ P\sigma \xrightarrow{\mathcal{L}} (\nu\bar{m})P'\sigma' \wedge S((\nu\bar{n})(\nu\bar{m})P'\sigma', (j', Val', Res'))$$

Verification of a SpiWrapper implementation

- ▶ Use Middleweight Java (MJ) framework
 - ▶ Formal semantics of rich subset of sequential Java
- ▶ For each method of each class:
 - ▶ Show that all possible execution paths behave as formally specified
 - ▶ Currently hand made for one simple class

Outline

Introduction

Motivation

Overall Architecture

The Spi2Java Framework

Type Inference

Data Marshalling

Data Encoding

Cryptographic Algorithms and Parameters

Code Generation

Conclusions

Future Work

Conclusions

Spi2Java framework

- ▶ Generation of Java code from Spi Calculus specifications
 - ▶ Provably correct refinement
- ▶ Some manually added information
 - ▶ Fill the semantic gap
 - ▶ Type assignment
 - ▶ Cryptographic algorithms and parameters
 - ▶ Marshalling and encoding functions
 - ▶ Sufficient conditions are stated
 - ▶ Filled details cannot introduce Dolev-Yao security faults

Future Work - I

Input language

- ▶ Spi Calculus: domain specific
 - ▶ Steep learning curve
- ▶ Small Java subset
 - ▶ Easily mapped back to Spi Calculus
 - ▶ No need for code generation
 - ▶ Model extraction like?

Soundness of generated code

- ▶ Verification of a complete SpiWrapper implementation
 - ▶ Make proofs (semi-)automatic
- ▶ Use provably correct implementations of cryptographic primitives

Future Work - II

Marshalling and encoding functions

- ▶ General method to check information flow requirements for arbitrary marshalling function implementations
 - ▶ Type system with non interference property for cryptographic primitives
 - ▶ Jif: Java + information flow
- ▶ Automatic generation of marshalling and encoding functions
 - ▶ Packet processing approach
 - ▶ Correctness of implementation (*needed* by encoding functions)
- ▶ Apply results to model extraction techniques

Future Work - III

Spi2Java framework

- ▶ Improve usability and functionality
- ▶ Extend to
 - ▶ Security-aware applications
 - ▶ A more powerful intruder model
 - ▶ Timed models
 - ▶ Computational models
 - ▶ Protocol sessions monitoring (with Jan Jürjens)

The End

Thank you!
Questions?

References - I



D. Pozza, R. Sisto, and L. Durante.

Spi2Java: Automatic cryptographic protocol Java code generation from spi calculus.

In International Conference on Advanced Information Networking and Applications, pages 400–405, 2004



A. Pironti and R. Sisto.

An Experiment in Interoperable Cryptographic Protocol Implementation Using Automatic Code Generation.

In IEEE Symposium on Computers and Communications, pages 839–844, 2007

References - II



A. Pironti and R. Sisto.

Soundness Conditions for Message Encoding Abstractions in Formal Security Protocol Models.

In *International Conference on Availability, Reliability and Security*, pages 72–79, 2008



A. Pironti and R. Sisto.

Formally Sound Refinement of Spi Calculus Protocol Specifications into Java Code.

In *IEEE High Assurance Systems Engineering Symposium*, to appear

References - III



B. Tobler and A. Hutchison.

Generating network security protocol implementations from formal specifications.

In Certification and Security in Inter-Organizational E-Services,
Toulouse, France, 2004



C.-W. Jeon, I.-G. Kim, and J.-Y. Choi.

Automatic generation of the C# code for security protocols verified with Casper/FDR.

In International Conference on Advanced Information Networking and Applications, pages 507–510, 2005

References - IV



J. Jürjens.

Verification of low-level crypto-protocol implementations using automated theorem proving.

In *Formal Methods and Models for Co-Design*, pages 89–98, 2005



K. Bhargavan et al.

Verified interoperable implementations of security protocols.

In *Computer Security Foundations Workshop*, pages 139–152, 2006