Security Protocols

Properties, Primitives, Models

Partly based on slides by Bruno Blanchet

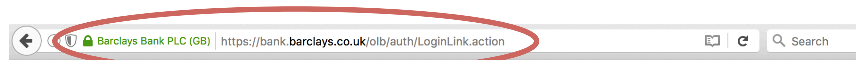
Alfredo Pironti – IOActive
alfredo.pironti@ioactive.com

FOSAD 2016 Summer School
29 Aug – 3 Sep 2016

Security Protocols

- Communication protocol
  - Set of rules to exchange data between entities
- Secure communication
  - According to security properties (e.g. confidentiality)
- Use of cryptographic primitives
  - E.g. encryption, hash functions

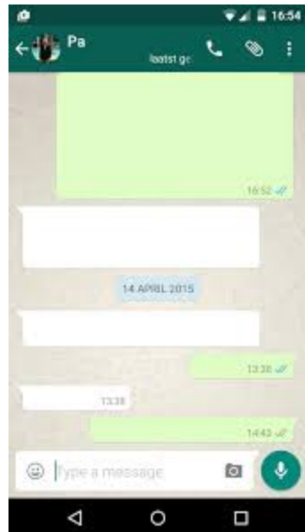
Usage of Security Protocols



Mobile

Quick, safe and convenient - Online Banking made easy

Not yet registered for Online Banking ? Register now.



SECURITY PROPERTIES

Security Properties

- Secrecy
  - An attacker cannot learn the content of a secret message
- Integrity
  - Any attempt to tamper with the message content will be reliably detected
- Authentication
  - Alice knows she's really talking to Bob

Security Properties

- Availability
  - The attacker cannot disrupt communication
- Fairness
  - Playing by the rules provides the maximum benefit
- Non repudiation
  - Alice sends a message to Bob. Alice cannot later deny she sent the message; Bob cannot later deny he received the message

Example: e-voting

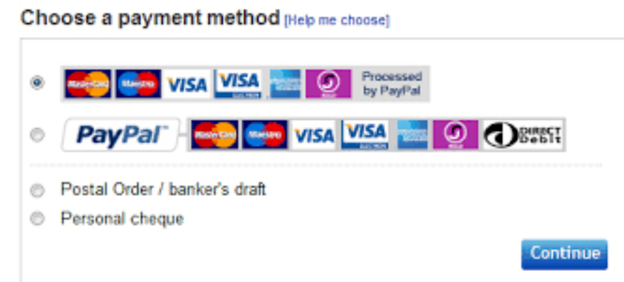
- Eligibility
  - Only legitimate voters can vote
  - Only one vote is counted
- Fairness
  - No early results can be obtained which could influence the remaining votes
- Verifiability
  - Individual: I can check my vote has been counted
  - Universal: The outcome corresponds to the votes on the bulletin board

Example: e-voting

- Privacy
  - My vote is not revealed to anyone
- Receipt-freeness
  - I cannot prove the way I voted
  - Defends from coercion
- Coercion-resistance
  - Coercer and voter collude

Types of Security Protocols

- Two parties, point-to-point
  - Client-server, instant messaging, VoIP
- Multi-party
  - Distributed, or centralized

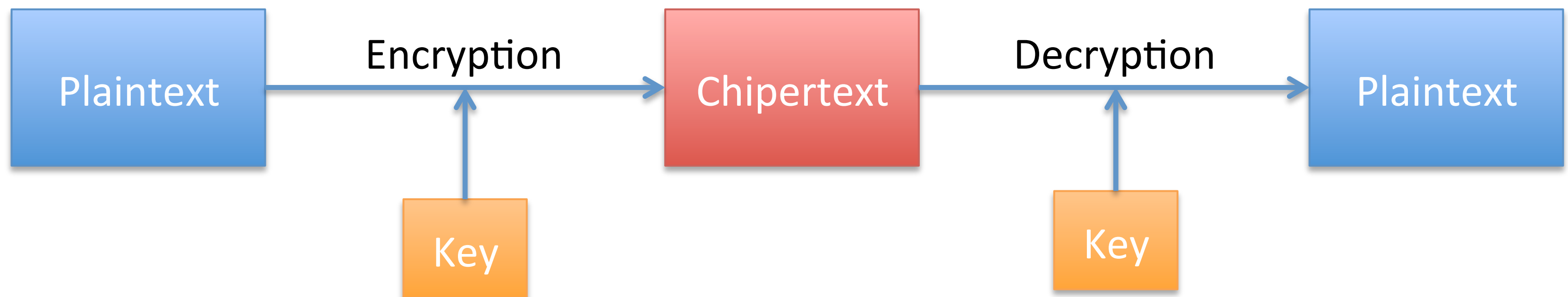


- Open ended
  - Group chat, domotics

CRYPTOGRAPHIC PRIMITIVES

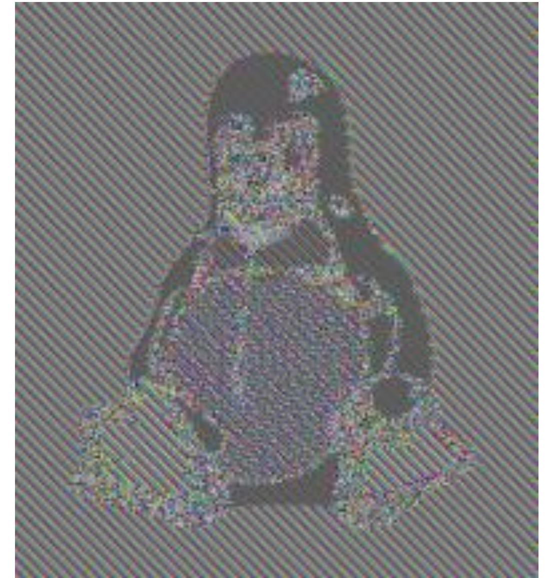
Cryptographic Primitives

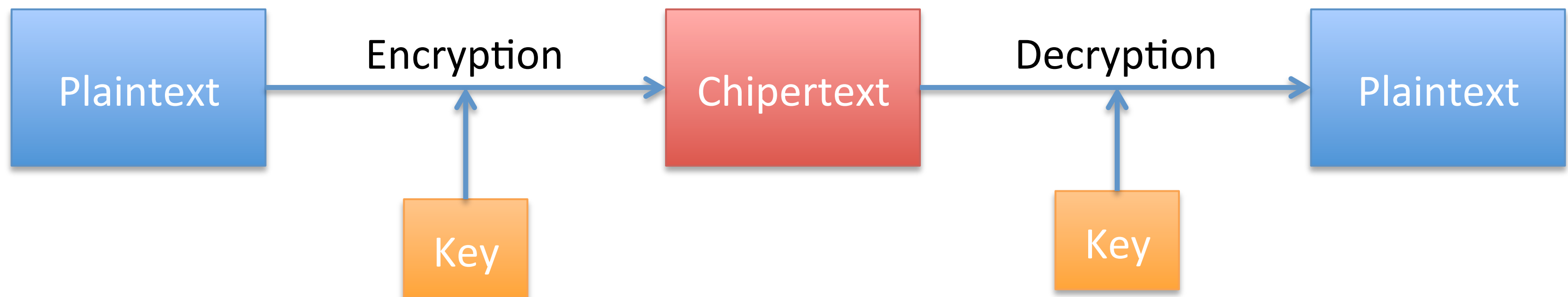
- Building blocks of security protocols
- Encryption
  - Use a key to turn a message into a pseudorandom sequence of bytes
- Symmetric encryption



Cryptographic Primitives

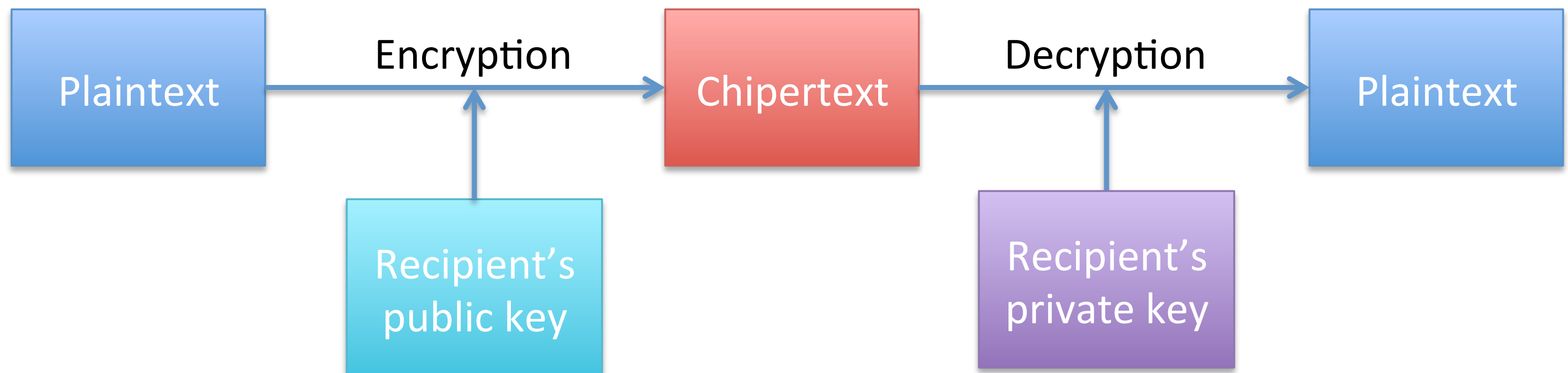
- Symmetric encryption
  - Examples: 3DES, AES
  - Provides secrecy (only!)
  - Not so trivial
    - Stream vs Block ciphers (ECB, CBC)
    - IVs





Cryptographic Primitives

- Asymmetric encryption
  - Examples: OpenPGP, RSA
  - Provides secrecy (only!)
  - Not so trivial
    - Equivalent of IV
    - Key distribution problem (CAs, WoT)



Cryptographic Primitives

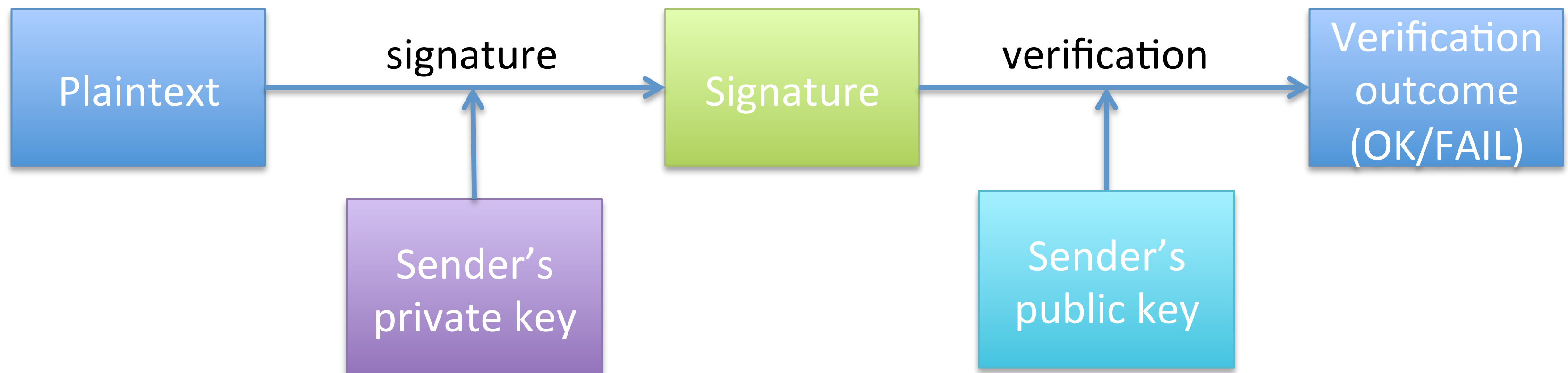
- Hash functions
  - Examples: MD5, SHA-1, SHA-256
  - Properties
    - Pre-image resistance
      - Given H(m), find m
    - Second pre-image resistance
      - Given m and H(m), find m’ such that H(m) = H(m’)
    - Collision resistance
      - Find m, m’ such that H(m) = H(m’)

Cryptographic Primitives

- Hash functions
  - Store passwords securely
    - Salted hash: s, H(s,m)
  - Message Authentication Code (MAC)
    - Provides message integrity
    - Keyed hash: H(k,m)
    - Use HMAC constructs!

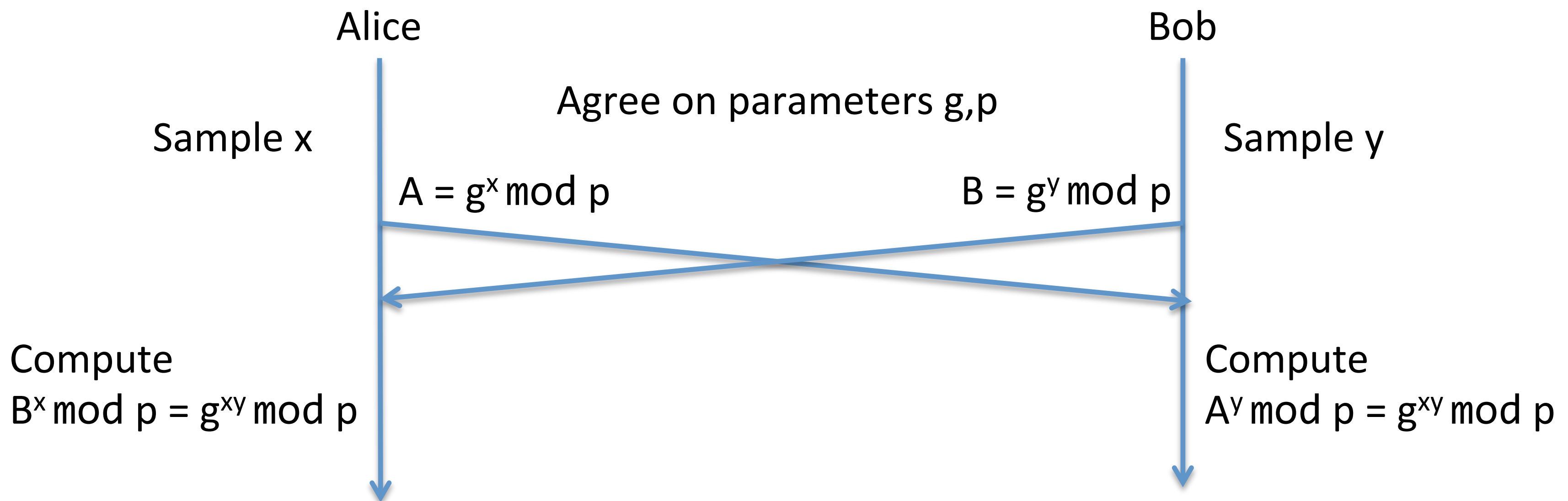
Cryptographic Primitives

- Signature
  - Examples: OpenPGP, RSA, DSA
  - Provides integrity and authentication
  - As usual, not so trivial
    - Randomized signature



Cryptographic Primitives

- Diffie-Hellman (DH) key exchange
  - Agree on shared secret
  - No info on peer's identity
  - Now more efficient with elliptic curves (ECDH)



Summary of Cryptographic Primitives

- Secrecy
  - Symmetric encryption (3DES, AES)
  - Asymmetric encryption (OpenPGP, RSA)
- Integrity / Authentication
  - HMAC
  - Signature (OpenPGP, RSA, DSA)
- Key Exchange
  - DH, ECDH

Combining Properties & Primitives

- How to get secrecy and authentication for a message?

- Exercise: Combine previous asymmetric primitives to get a new primitive that provides secrecy and authentication

Combining Properties & Primitives

- Symmetric secrecy and authentication
  - Examples: AES-CCM, AES-GCM
  - Authenticated Encryption with Additional Data (AEAD)
    - AD are only authentic, not secret

SECURITY PROTOCOL MODELS

Security Protocol Verification

- Why?
  - Security protocol design is error prone
  - Testing can be of little help
    - Distributed environment
    - Unbounded number of sessions
    - Attacker behaves in the worst possible way
  - Usually deployed in sensitive contexts

Symbolic Model

- AKA Dolev-Yao model
- Cryptographic primitives are black boxes
- Messages are symbolic terms
  - $\{m\}_k$ encryption of message *m* with key *k*
  - (m,n) pair of *m* and *n* messages

Symbolic Model

- Perfect cryptography assumption
  - Rules define how the attacker can manipulate terms
  - Example: Attacker learns *m* only if he has $\{m\}_k$ and *k*
  - Rules can be added (to a certain extent) to model cryptographic assumptions
- Amenable for automated reasoning
- High abstraction level – less realistic
  - Still useful to find relevant, real-world logical attacks

Symbolic Model

- Secrecy
  - Attacker can never learn a secret term
- Authentication
  - Correspondence property
  - Each time A believes she has engaged in a session with B, B has started a session with A, and both agree upon some session-specific data
  - "B is authenticated to A"

Symbolic Model

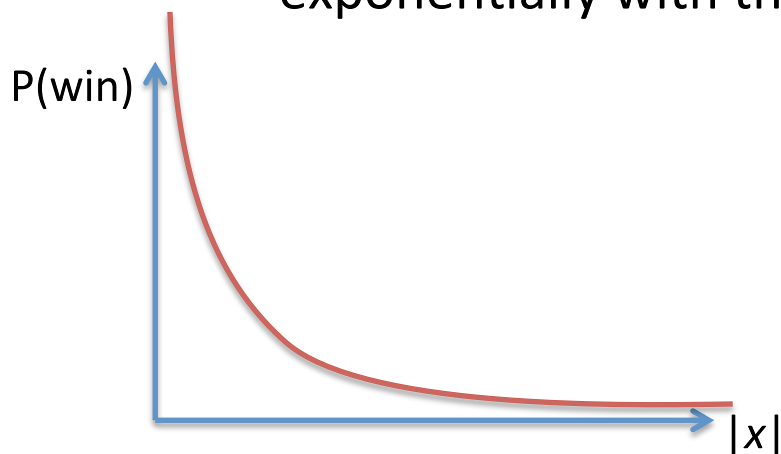
- How to verify
  - Compute all terms reachable by the attacker (infinite set)
  - Compute all possible sessions interleaving (unbound)
  - In general, it is a undecidable problem
- Verification techniques
  - Sound approximations
  - Pruning of redundant states
  - User interaction
  - Allow non-termination

Computational Model

- Messages are bitstrings: 00101101
- Cryptographic primitives are functions on bitstrings
  - E(011, 00101101) = 01001010
- Attacker is a probabilistic polynomial Turing machine
- More realistic, but harder for automatic reasoning

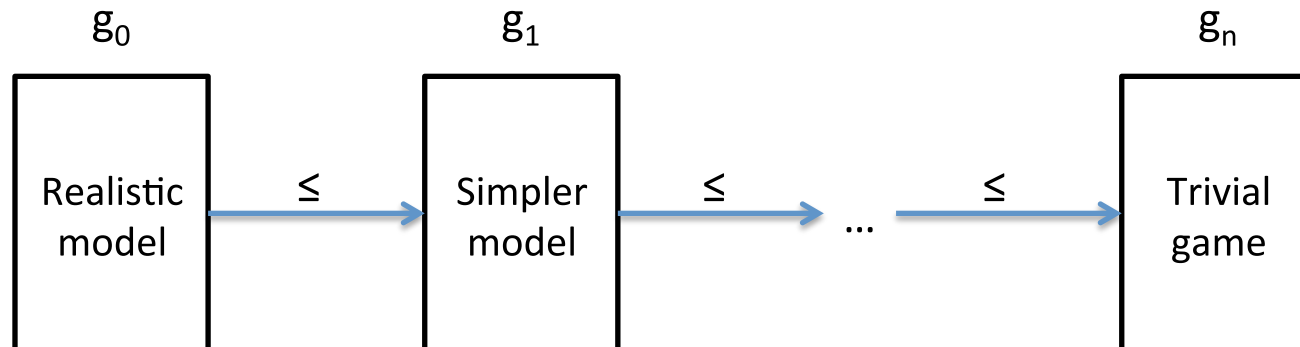
Computational Model

- Security properties
  - The attacker has a *negligible* probability of breaking the protocol
  - *Negligible* is formally defined
    - x is a security parameter (e.g. size in bits of a symmetric key)
    - Attacker probability of breaking the protocol decreases exponentially with the size of the security parameter x



Computational Model

- Typical verification technique: game hopping
  - Start with a realistic game
  - Small hops to simpler games, preserving same or higher attacker advantage
  - End with trivial game where attacker only wins with negligible probability
  - Usually done by hand, automatic and interactive tools emerging



Symbolic vs Computational Model

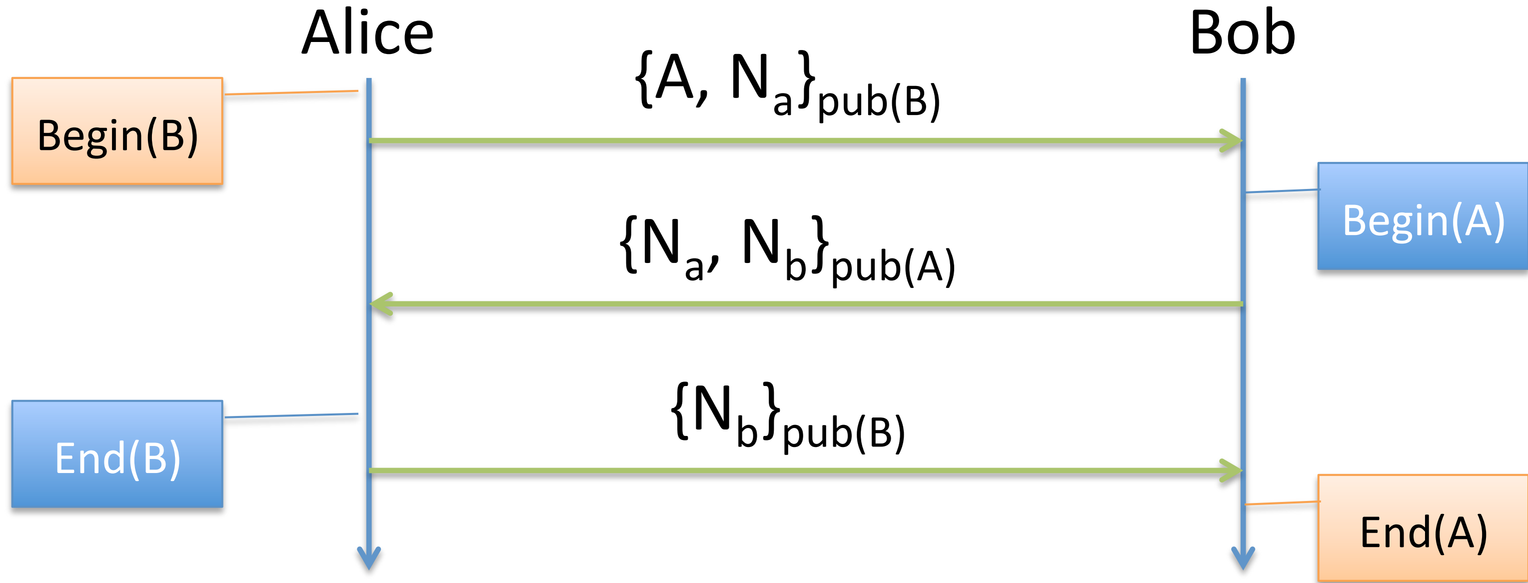
- Symbolic attack $\Rightarrow$ computational attack
- Symbolic proof $\not\Rightarrow$ computational proof

- Computational soundness of symbolic model
  - Symbolic proof $\Rightarrow$ computational proof
  - Under additional assumptions

Example

THE NEEDHAM-SCHROEDER PROTOCOL

Protocol definition (1978)



- Security properties
  - N_a, N_b are secret
  - Mutual authentication
    - A authenticates to B
    - B authenticates to A

Lowe's Man in the Middle Attack

- Discovered 17 years after protocol definition!

- Involves 2 concurrent protocol sessions
- *A* initiates a session with the attacker

- Discovered and fixed with the help of an automatic tool

Lowe's Attack (1996)

Alice Mallory Bob

Begin(M)

$\{A, N_a\}_{pub(M)}$

$\{A, N_a\}_{pub(B)}$

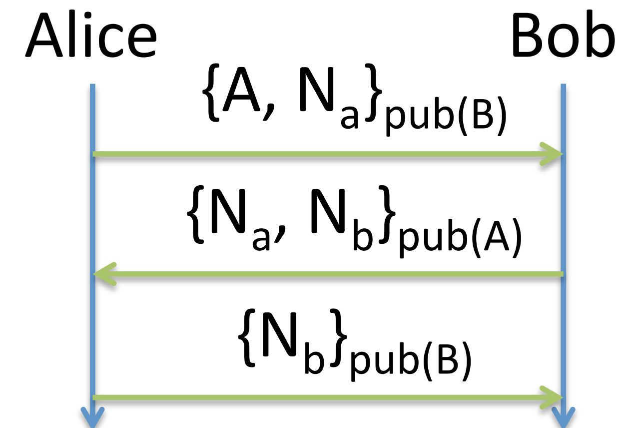
$\{N_a, N_b\}_{pub(A)}$ $\{N_a, N_b\}_{pub(A)}$

$\{N_b\}_{pub(M)}$

$\{N_b\}_{pub(B)}$

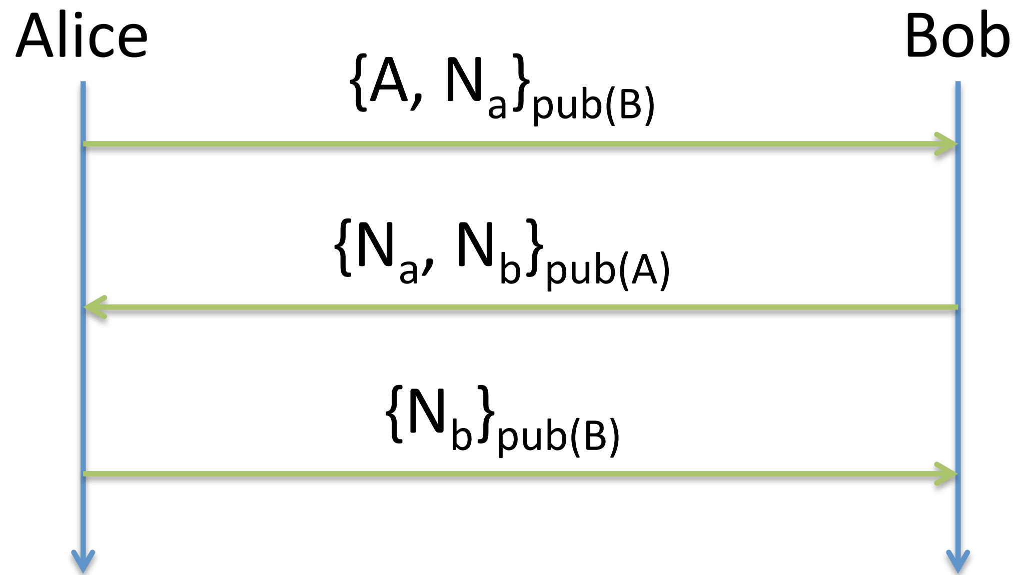
End(A)

- Attacker knows N_a, N_b
- B believes he ended a session with A, but A never started a session with B



Excercise

- Propose a fix for the Needham-Schroeder protocol



Formal Verification of Security Protocol Implementations

Approaches and Techniques

Alfredo Pironti – IOActive
alfredo.pironti@ioactive.com

FOSAD 2016 Summer School
29 Aug – 3 Sep 2016

About Models

- Always an abstraction of real world
  - Will only capture defined security properties
  - Typically ignore side channels
    - Message size
    - Timing
    - Power consumption
    - Noise
- Still significant gap between models and real implementations

Will your model catch this?

OpenSSL up to version 0.9.8i (2009)

```
ret = RSA_verify(NID_md5_sha1,buf 36,buf2,rsa_num,sa_key[j]);
if (ret == 0) {
  BIO_printf(bio_err,"RSA verify failure\n");
}
// Verification OK: continue with protocol execution
```

Typical symbolic model

```
if RSA_verify(md5_sha1, cert, key)
then ...
else event("FAILURE").
```

About Implementations

- Often manually developed
- Directly derived from informal documents
  - RFC, ISO standards
  - Ambiguities, incomplete specifications, logical mistakes
- Typically only tested for functionality

Verifying Implementations

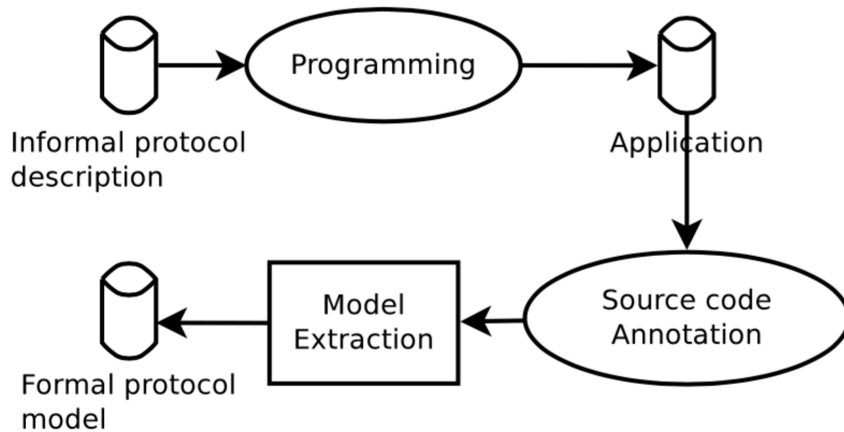
- Direct verification of arbitrary code
  - Too complex
- Is verifying code enough?
  - Trusted computing base
    - Implementation of cryptographic primitives
    - Compiler
    - OS
  - Network attacker
    - No attacker running parallel process on same machine

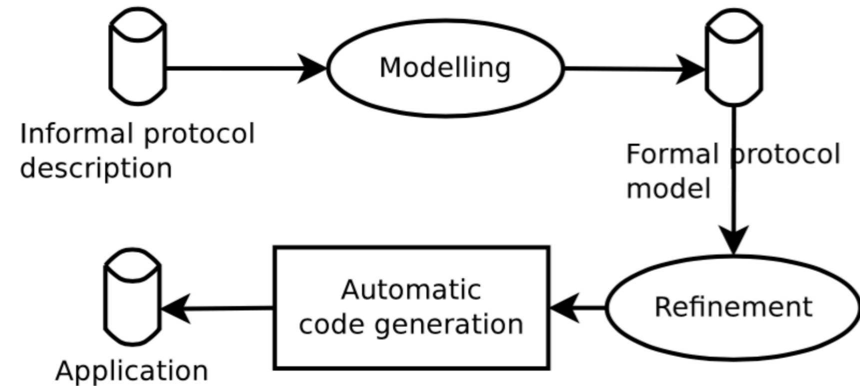
Overall approach

- Verify some formal model
- Formal model proof ⇒ source code proof
- Sound link between model and source code
- Assumptions about source code and its environment

- No general taxonomy for different approaches

A Classification of Approaches





Formal Soundness

$\alpha(M) \models \alpha(\phi) \Rightarrow M \models \phi$

Formal Soundness

$M_a \models \phi_a \Rightarrow \rho(M_a, C) \models \rho(\phi_a)$

Where α is "abstraction mapping"; ρ is "refinement mapping"; M is a model; ϕ is a property; C are implementation choices

Further Approaches

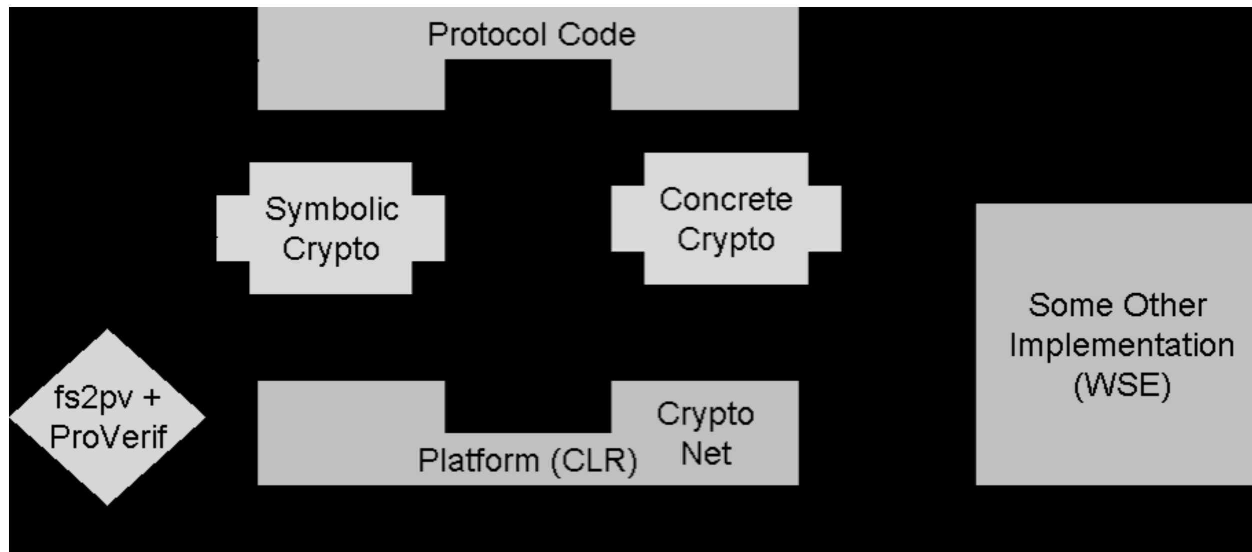
- Model-implementation conformance
  - Both model and implementation provided
  - Check model is secure
  - Check implementation refines model
  - Example: Refinement Type-Checking (F7)
- Abstract & concrete compilers
  - Intermediate version of the protocol provided
  - Abstract compiler generates model
  - Concrete compiler generates implementation
  - Example: JavaSPI

Model Extraction

- Pros
  - Handling of legacy code
  - No need to know modeling languages
  - Access to standard IDE features when developing
  - Efficiency of handwritten code
- Cons
  - Handling of all implementation details at once
  - No protection from low-level coding errors
    - Buffer overflows
  - Need to map verification results back to code
    - Or expose underlying modeling language

Model Extraction Example: FS2PV

- Implementation language: F#
- Formal model: ProVerif
- Cryptographic model: Symbolic



Model Extraction Example: FS2PV

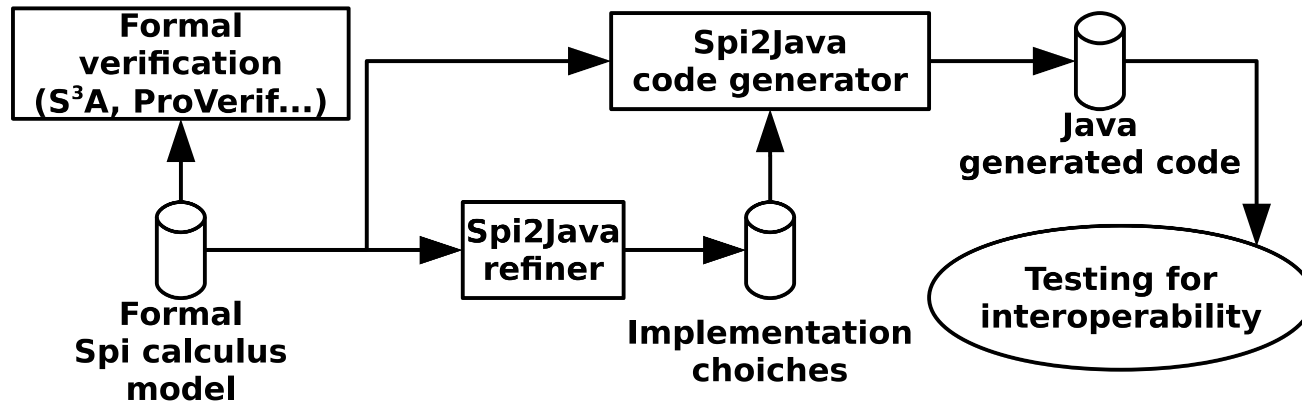
- Attacker model
  - Arbitrary F# program
  - Specific F# interfaces to interact with the protocol
- Security properties
  - Predicates on traces
    - Events (secrecy); correspondences (authentication)
  - Formally sound
- In practice
  - Case study on ad-hoc TLS implementation
  - Scalability issues
    - Small changes in code, unpredictable verification results

Code Generation

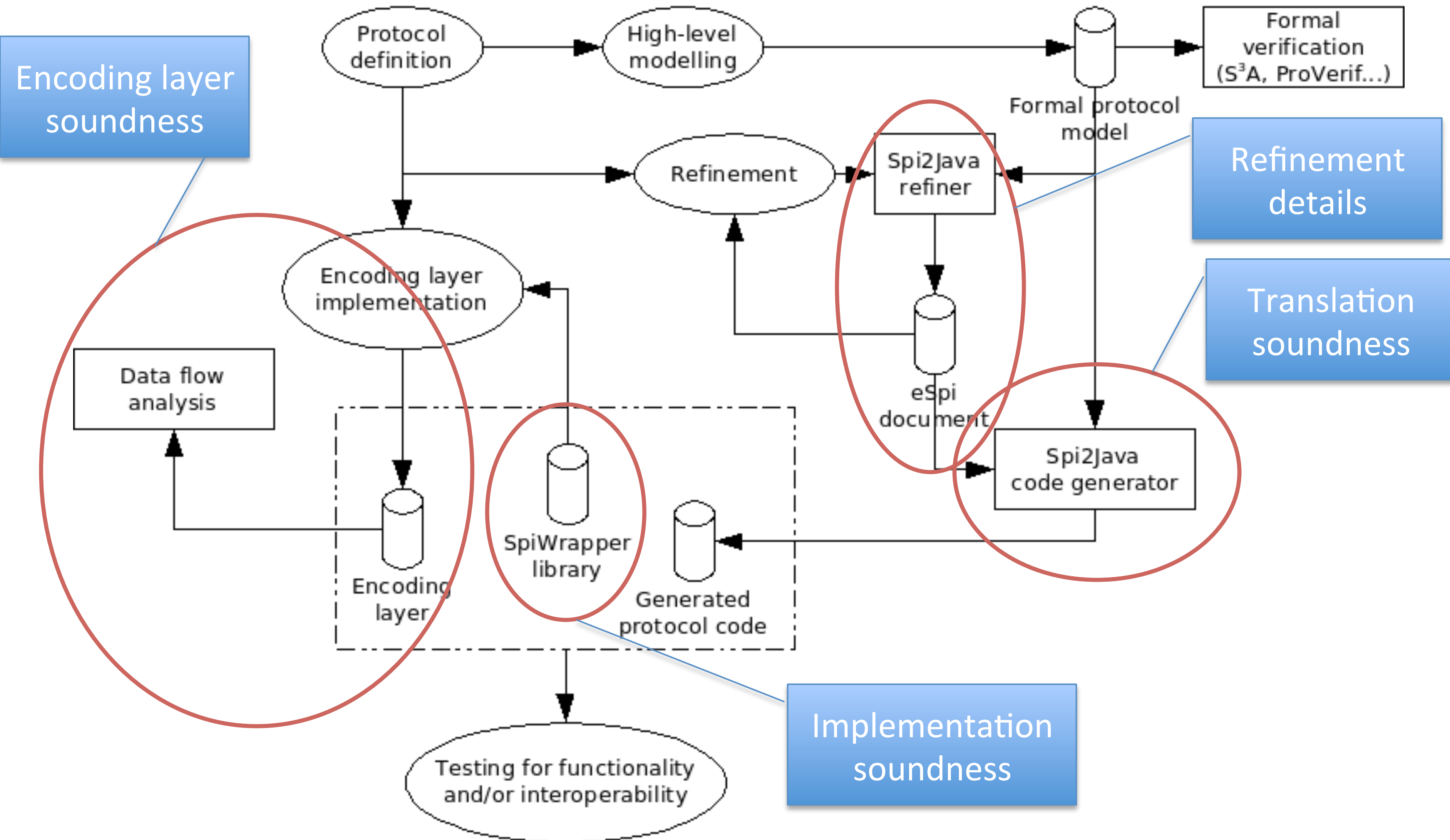
- Pros
  - Reduced complexity; separate development steps
    - First protocol logic; then implementation details
  - Free from low-level errors by construction
  - Fast prototyping
- Cons
  - Need to know modeling language
  - IDEs not mature or available
  - No handling of legacy code
  - Generated code unmodifiable
    - Or loss of soundness properties

Code Generation Example: Spi2Java

- Modeling language: Spi Calculus
- Implementation: Java
- Cryptographic model: Symbolic



Spi2Java Full Workflow



Spi Calculus Syntax – Terms

$L, M, N ::=$	terms
n	name
(M, N)	pair
0	zero
$suc(M)$	successor
x	variable
$\{M\}_N$	shared-key encryption
$H(M)$	hashing
M^+	public part
M^-	private part
$\{[M]\}_N$	public-key encryption
$[\{M\}]_N$	private-key signature

Spi Calculus Syntax – Processes

$P, Q, R ::=$	processes
$\overline{M}\langle N\rangle.P$	output
$M(x).P$	input
$P \mid Q$	composition
$(\nu n)P$	restriction
$!P$	replication
$[M\ is\ N]\,P$	match
$\mathbf{0}$	nil
$let\ (x, y) = M\ in\ P$	pair splitting
$case\ M\ of\ 0 : P\ suc(x) : Q$	integer case
$case\ L\ of\ \{x\}_N\ in\ P$	shared-key decryption
$case\ L\ of\ \{[x]\}_N\ in\ P$	decryption
$case\ L\ of\ [\{x\}]_N\ in\ P$	signature check

Spi Calculus Example: RPC Protocol

Informal notation

$A \to B : Ra, \{Req\}_{Kab}, H(Kab, Ra, Req)$

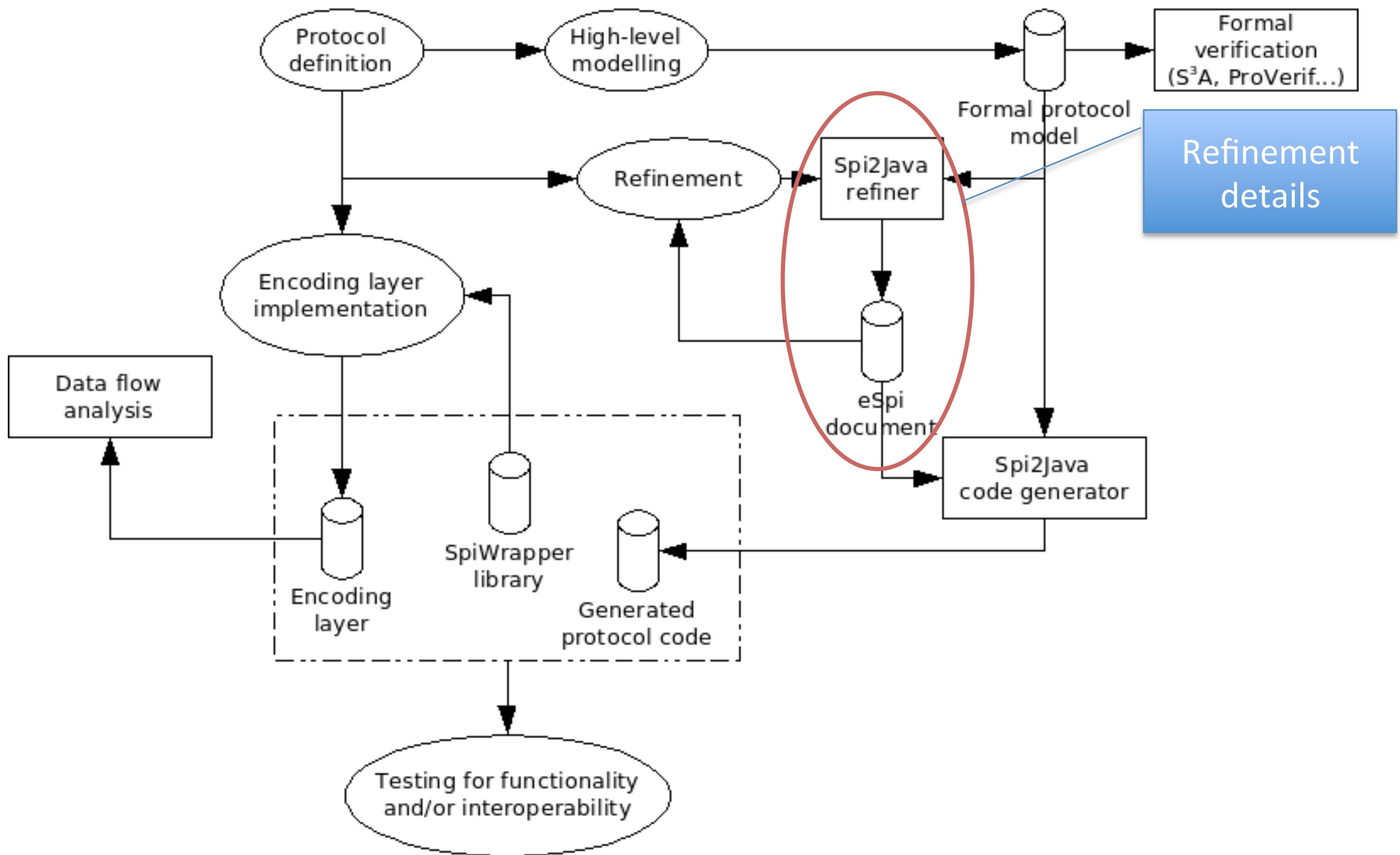
$B \to A : Rb, \{Resp\}_{Kab}, H(Kab, Ra, Rb, Req, Resp)$

A Actor in Spi Calculus

```
A(cAB,Kab,Req) :=
 (@Ra)
 cAB<(Ra,{Req}Kab,H(Kab,Ra,Req))>.
 cAB((Rb,encResp,MAC)).
 case encResp of {Resp}Kab in
 [ MAC is H(Kab,Ra,Rb,Req,Resp) ]
 0
```

B Actor in Spi Calculus

```
B(cAB,Kab) :=
 cAB<(Ra,encReq,MAC)>.
 case encReq of {Req}Kab in
 [ MAC is H(Kab,Ra,Req) ]
 (@Rb)
 cAB<(Rb,{f(Req)}Kab,
   H(Kab,Ra,Rb,Req,f(Req)))>.
 0
```


Protocol definition
High-level modelling
Formal protocol model
Formal verification (S³A, ProVerif...)
Refinement
Spi2Java refiner
Refinement details
Encoding layer implementation
eSpi document
Data flow analysis
Spi2Java code generator
SpiWrapper library
Encoding layer
Generated protocol code
Testing for functionality and/or interoperability

Spi2Java Translation Example

A Actor in Spi Calculus

```
A(cAB,Kab,Req) :=
 (@Ra)
 cAB<(Ra,{Req}Kab,H(Kab,Ra,Req))>.
 cAB((Rb,encResp,MAC)).
 case encResp of {Resp}Kab in
 [ MAC is H(Kab,Ra,Rb,Req,Resp) ]
 0
```

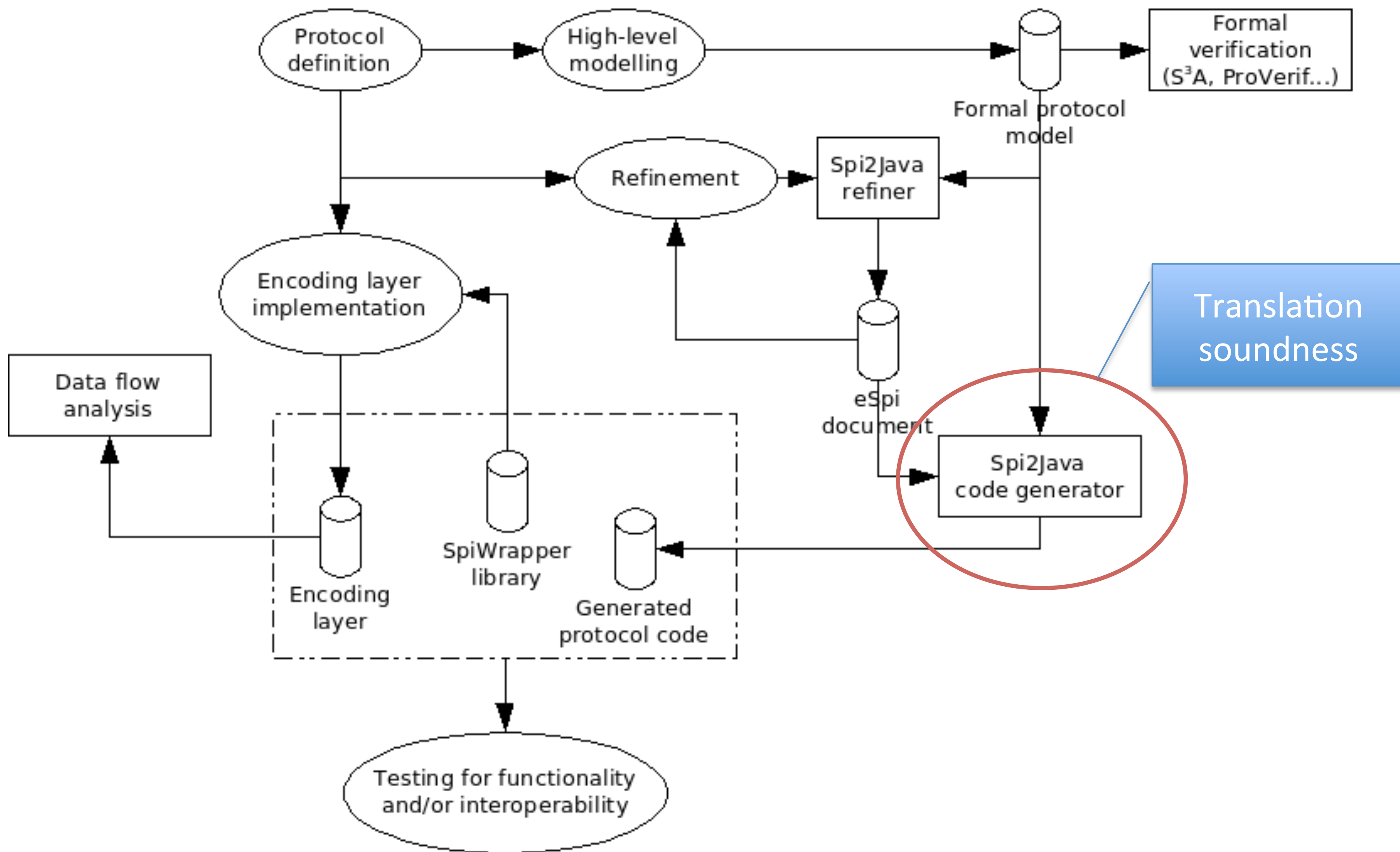
- Refinement details
  - Cryptographic algorithms
  - Message formatting

```
/* case encResp of {Resp}Kab in */
Message Resp =
  encResp.decrypt(new MessageSR(), Kab,
  "AES", iv.getText(), "CBC", "PKCS5Padding", "SunJCE");
```

Spi2Java Refinement Details

```
/* case encResp of {Resp}Kab in */
Message Resp =
  encResp.decrypt(new MessageSR(), Kab,
  "AES", iv.getText(), "CBC", "PKCS5Padding", "SunJCE");
```

```
<term id="3" name="encResp" type="Shared Key Ciphered">
  <parameters>
    <param name="algorithm" type="const">AES</param>
    <param name="iv" type="var">iv</param>
    <param name="mode" type="const">CBC</param>
    <param name="padding" type="const">PKCS5Padding</param>
    <param name="provider" type="const">SunJCE</param>
  </parameters>
</term>
```


Protocol definition
High-level modelling
Formal protocol model
Formal verification (S³A, ProVerif...)
Refinement
Spi2Java refiner
Encoding layer implementation
eSpi document
Translation soundness
Data flow analysis
Spi2Java code generator
Encoding layer
SpiWrapper library
Generated protocol code
Testing for functionality and/or interoperability

Spi2Java Translation Example

A Actor in Spi Calculus

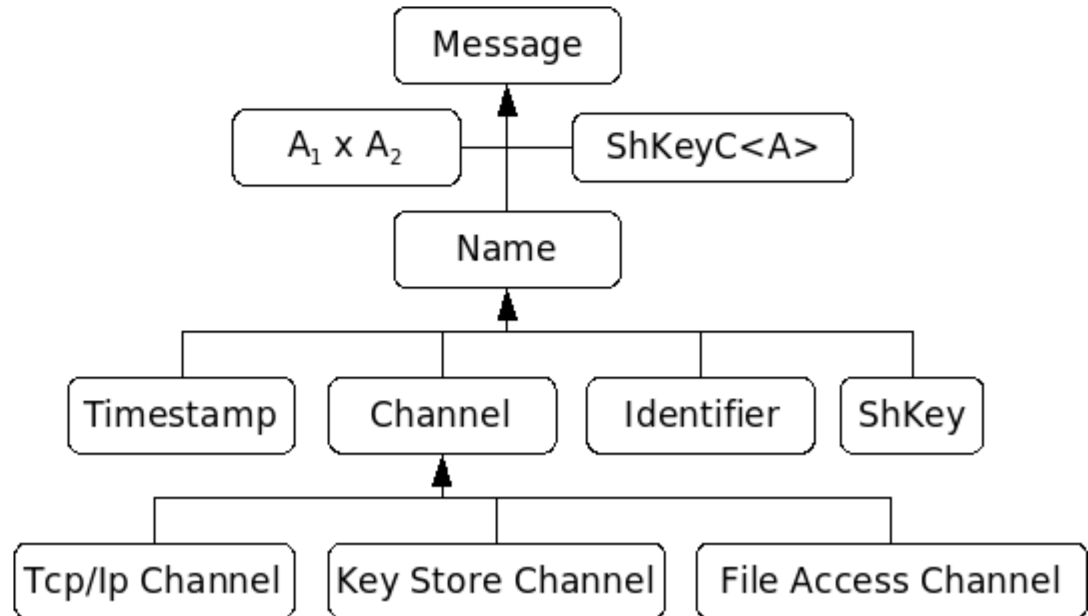
```
A(cAB,Kab,Req) :=
 (@Ra)
 cAB<(Ra,{Req}Kab,H(Kab,Ra,Req))>.
 cAB((Rb,encResp,MAC)).
 case encResp of {Resp}Kab in
 [ MAC is H(Kab,Ra,Rb,Req,Resp) ]
 0
```

- Type system
  - Spi Calculus is untyped

```
/* case encResp of {Resp}Kab in */
Message Resp =
  encResp.decrypt(new MessageSR(), Kab,
  "AES", iv.getText(), "CBC", "PKCS5Padding", "SunJCE");
```

A Spi Calculus Type System

- Hierarchical
  - Java inheritance
- Parametric
  - Java generics
- Only for terms
  - Java statements have no types



Type Inference

- ML-like
  - Automatic
  - User can provide coherent downcast

$$\frac{\Gamma \vdash M_1 : A_1 \qquad \Gamma \vdash M_2 : A_2}{\Gamma \vdash (M_1, M_2) : A_1 \times A_2} \text{(T-Pair)}$$

$$\frac{\Gamma \vdash M : \mathit{Channel} \qquad \Gamma, x : A \vdash P}{\Gamma \vdash M(x).P} \text{(P-In)}$$

$$\frac{\Gamma \vdash M : \mathit{Message} \qquad \Gamma \vdash N : \mathit{Message} \qquad \Gamma \vdash P}{\Gamma \vdash [M \ \mathit{is} \ N] \ P} \text{(P-Match)}$$

Spi2Java Translation Function

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

```
/* case encResp of {Resp}Kab in */
Message Resp =
  encResp.decrypt(new MessageSR(), Kab,
  "AES", iv.getText(), "CBC", "PKCS5Padding", "SunJCE");
```

```
1 /* let (ID,Msg) = aPair in */
2 Identifier ID = aPair.getLeft();
3 Message M = aPair.getRight();
```

Spi2Java Translation Function

Main function

- Function $tr_p : Spi \rightarrow Java$
- Input: well typed Spi Calculus process
- Output: well formed (i.e. that "compiles") Java code
  - Uses a custom library, called SpiWrapper

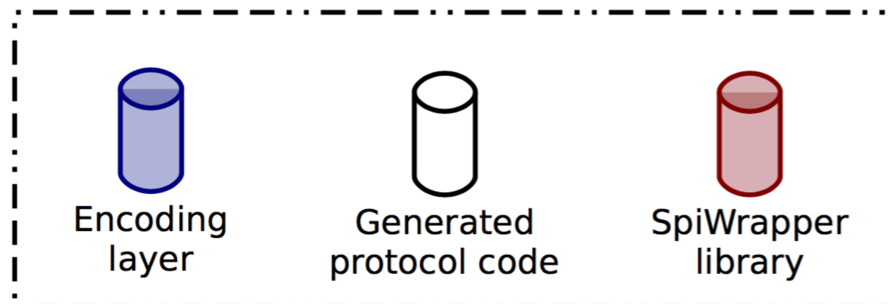
Auxiliary function

- Function $tr_t : SpiTerm \rightarrow Java$
- Input: a well typed Spi Calculus term
- Output: the Java code "building" (declaring and initialising) the corresponding Java variable

The SpiWrapper Library

- Implements Spi Calculus behavior in Java
  - A pair can be split; An encrypted term can be decrypted…
  - One class for each type in the type system
- Formal specification of expected implementation behavior

```
/* case encResp of {Resp}Kab in */
Message Resp =
  encResp.decrypt(new MessageSR(), Kab,
  "AES", iv.getText(), "CBC", "PKCS5Padding", "SunJCE");
```



Static Semantic Correctness

- A well typed Spi Calculus process is translated into a well formed Java program

Theorem

If $\Gamma \vdash P$*, then* $tr_p(P)$ *is well formed*

Proof Idea.

By inductive inspection of the translation function

Dynamic Semantic Correctness

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

Spi Calculus Operational Semantics

$$P\sigma \xrightarrow{\mathcal{L}} P'\sigma\sigma'$$

where P, P' are open Spi calculus processes
and σ, σ' are sets of variable substitutions

Example
Assume C(x).P' receives the term M over channel C
Then C(x).P' evolves into P', where x is substituted by M

Dynamic Semantic Correctness

Java Operational Semantics

$$j, Mem \xrightarrow{\mathcal{L}} j', Mem'$$

where j, j' are sequences of Java statements to be executed and Mem, Mem' are partial relations representing the current memory configuration

Simulation Relation S

$$S(P\sigma, (j, Mem)) \Leftrightarrow j = tr_p(P, RI) \ \wedge \ \sigma = Mem$$

for some refinement information RI

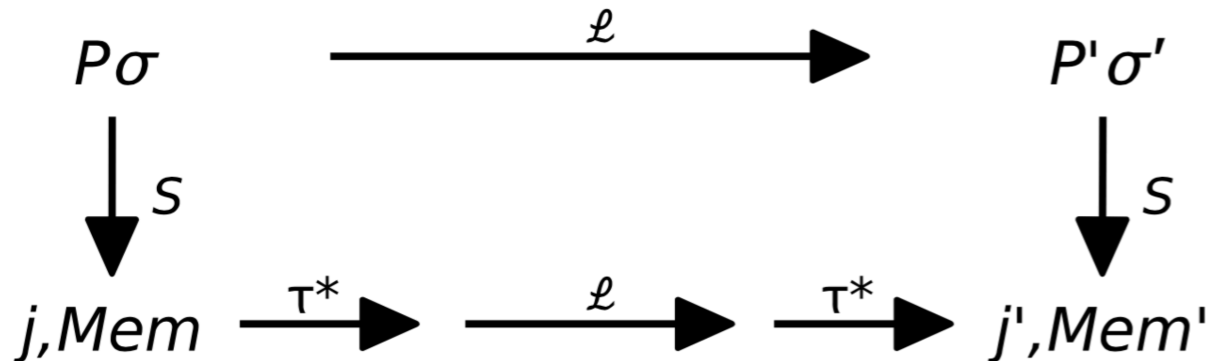
- The Java code is actually implementing a translated Spi calculus process

Dynamic Semantic Theorem

If the SpiWrapper library behaves as assumed, then

$$S(P\sigma, (j, Mem)) \;\wedge\; j, Mem \stackrel{\tau}{\rightarrow}^{*} \stackrel{\mathcal{L}}{\rightarrow} \stackrel{\tau}{\rightarrow}^{*} j', Mem' \Rightarrow$$

$$P\sigma \stackrel{\mathcal{L}}{\rightarrow} P'\sigma' \;\wedge\; S(P'\sigma', (j', Mem'))$$



Dynamic Semantic Theorem

If the SpiWrapper library behaves as assumed, then

$$S(P\sigma, (j, Mem)) \;\wedge\; j, Mem \xrightarrow{\tau}{}^{*} \xrightarrow{\mathcal{L}} \xrightarrow{\tau}{}^{*} j', Mem' \Rightarrow$$

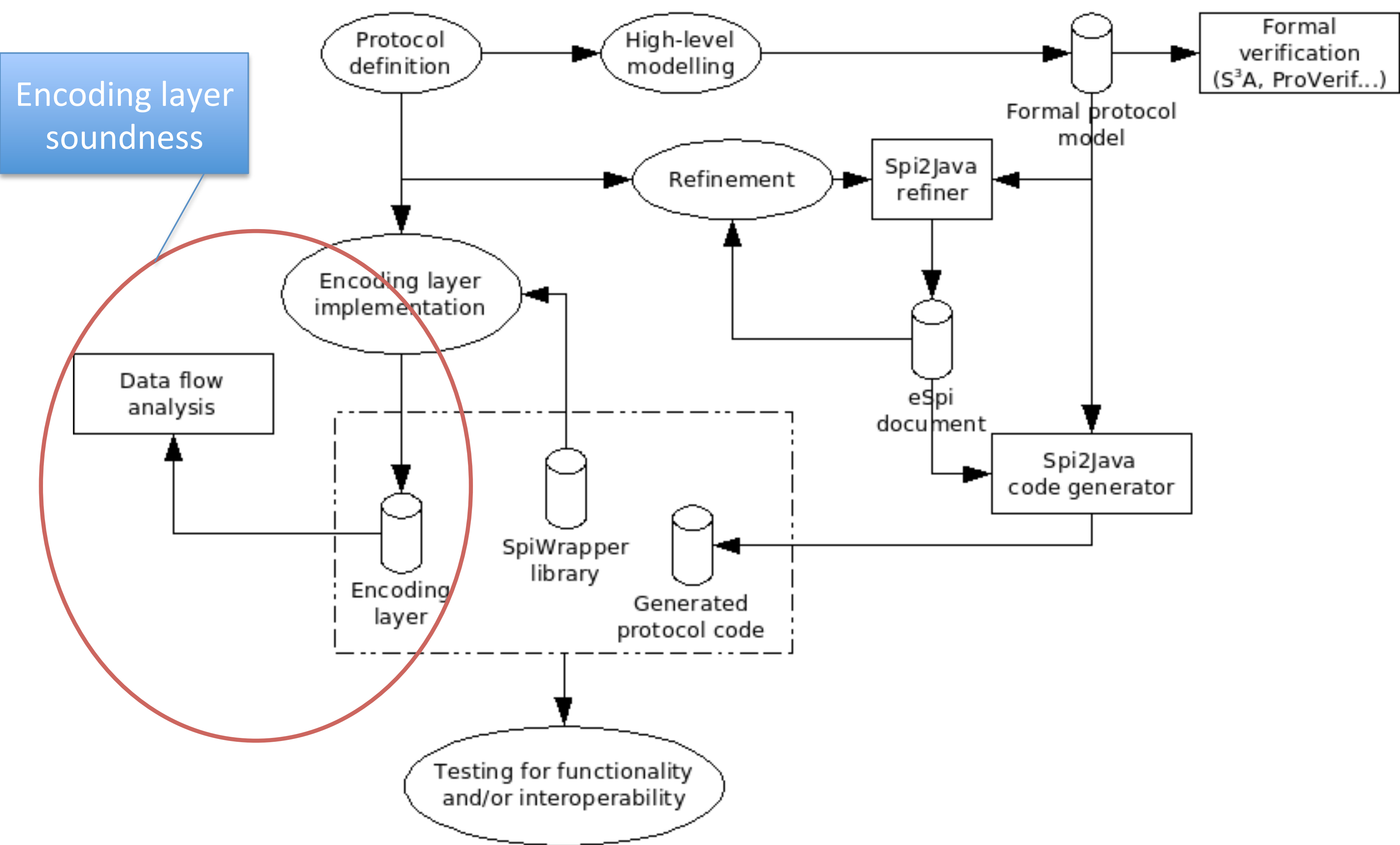
$$P\sigma \xrightarrow{\mathcal{L}} P'\sigma' \;\wedge\; S(P'\sigma', (j', Mem'))$$

- Exercise: describe the proof steps for the pair splitting case

```
1 /* let (ID,Msg) = aPair in */
2 Identifier ID = aPair.getLeft();
3 Message M = aPair.getRight();
```

- If `aPair` is implementing (M,N), and `M` is implementing M, then

$$\texttt{pair.getLeft()} \xrightarrow{\mathcal{L}} \texttt{M}$$


Encoding layer soundness
Protocol definition
High-level modelling
Formal protocol model
Formal verification (S^{3}A, ProVerif...)
Refinement
Spi2Java refiner
Encoding layer implementation
eSpi document
Data flow analysis
Spi2Java code generator
SpiWrapper library
Encoding layer
Generated protocol code
Testing for functionality and/or interoperability

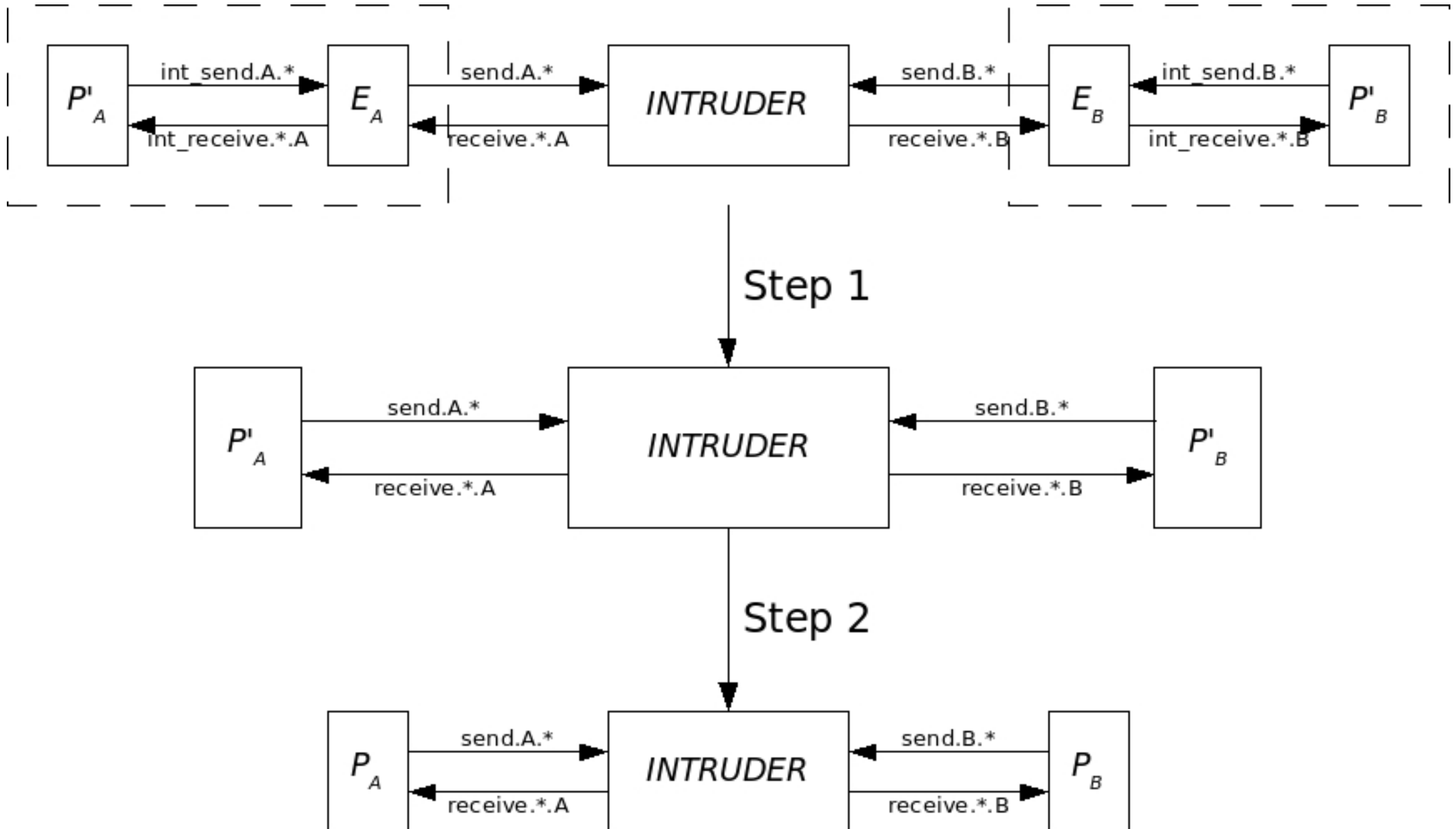
The Marshaling Problem

- Abstract protocol models deal with terms
- Java implementations exchange bitstrings

- Encoding functions manually written
  - Maximum flexibility on data format
  - How to ensure they do not break security?

- Two kinds of encoding functions
  - Data ready to be sent over the network
  - Data to be encrypted

Abstracting Channel Marshaling



Abstracting Channel Marshaling

- Only for secrecy and authentication
- Encoding functions part of the attacker
  - Attacker must know marshaling parameters
- Implementation must be a pure function
  - Stateless and side-effect free
  - Information flow property
- No assumptions on scheme specification
  - Even erroneous marshaling definitions are safe (though not functional)

Abstracting Encryption Marshaling

- Can’t be part of the attacker
  - Deal with secret data
- Abstraction for Secrecy
  - Each actor must implement invertible functions
  - Local property, can be checked in isolation
  - No assumptions on correctness with respect to encoding specification

Abstracting Encryption Marshaling

- Abstraction for Authentication
  - Correctness of implementation with respect to encoding specification
    - Weaker result, more complex to verify
  - Encoding parameters explicitly agreed upon
    - Example: Mavrogiannopoulos et al. 2012 attack on TLS