Formal Methods for Security Protocol Engineering

Riccardo Sisto
Security Protocols

• Secure information exchange over insecure networks using cryptography

• Typical goals: authentication, key exchange, data integrity, confidentiality

Our focus

\[ \begin{array}{c}
\text{applications} \\
\text{cryptographic protocols} \\
\text{communication primitives} & \text{cryptographic primitives}
\end{array} \]

Well standardized. Implementations commonly available as libraries
Example: Needham-Shroeder Public-key Authentication

Believed secure for years!
Example:
N-S Public Key Authentication
Man-in-the-middle Attack

Discovered by Formal Methods (model checking)
Security Protocols: Challenges

• Simple protocols but with many different scenarios
  – Concurrent sessions
  – Attackers can behave in **any** way

• Difficult to discover faults by hand

• Implementation bugs may disrupt protocol security
  – Testing may not reveal some mistakes
Security Proofs

• Objective: prove that no *reasonable* attacker can break a protocol *in practice*

• We need formal definitions: what is a reasonable attacker? what means it can break a protocol in practice?

• Different possible approaches:
  – use symbolic (high.level) models
  – use computational (more low-level) models
    • more accurate but harder
Rigorous Symbolic Modelling (Dolev-Yao)

• Abstract data types (data are symbolic terms)

• Crypto operations: algebraic operators with properties
  – e.g. $\text{decrypt}(\text{encrypt}(M,K),K) = M$

• Attacker can intercept/substitute messages, execute crypto operations, build new messages from current knowledge

• Attacker cannot guess secrets, get partial knowledge

• Prove attacks are **impossible** under these assumptions
Proofs on Dolev-Yao (Symbolic) Models

• Can be done automatically for standard properties (e.g. secrecy, authentication, data integrity) using model checking or automated theorem proving
  – Search for possible attacks
  – Search for formal correctness proofs

• Logical flaws are modeled

• Cryptosystem-related flaws are not modeled

• Side channels (e.g. related to timing) are not modeled
Computational Models

- Data modeled as bitstrings
- Crypto primitives modeled as algorithms
- Attacker: any polynomial-time algorithm
- Protocol runs modelled probabilistically
- Objective: prove that there does not exist an attacker that in polynomial time reaches a given goal with non-negligible probability
Proofs on Computational Models

• Generally based on complexity-theoretic reductions
  – “If there exists an attacker that runs in poly time and has non-negligible success probability then there exists an algorithm that solves a hard computational problem in poly time with non-negligible probability”

• Difficult to automate, but automatic provers based on game-theory already available

• Other approach: restrict crypto algorithms so that symbolic property implies computational property

• Cryptosystem-related flaws are modeled but side channels (e.g. related to timing) are not modeled
Proverif

• Good state-of-the-art automated theorem prover for security protocols based on Dolev-Yao modeling
• Developed by Bruno Blanchet (ENS, Paris)
• Protocol model expressed by a process calculus and then translated to a logic program
• Automated resolution-based algorithm
• Can deal with unbounded sessions (infinite state models)
• Web site: http://www.proverif.ens.fr/
How Proverif works

Extended pi-calculus model

Translation

Horn clauses + derivability queries

Resolution

Security Properties (queries)

Property cannot be proved

Analysis does not terminate

Potential attack

Property is true (and proved)

Attack Reconstruction

Analysis does not terminate

Property is true (and proved)
Specifying Protocols

• Two possibilities:
  – Horn clauses (low-level, only for experts)
  – Extended pi calculus (internally translated to Horn clauses)

• Each pi-calculus process models a protocol actor
  – Honest actors behave according to the protocol
  – Attacker can behave in any way
Extended Pi-Calculus: term Syntax

\[ M, N ::= \text{terms} \]

\[ x, y, z \quad \text{variables} \]
\[ a, b, c, k \quad \text{names} \]
\[ f(M_1, \ldots, M_n) \quad \text{constructor application} \]
\[ (M_1, \ldots, M_n) \quad \text{tuple} \]
Extended Pi-Calculus: main process Syntax

\[ P, \ Q ::= \]
\[ \text{out}(M,N).P \]
\[ \text{in}(M,x).P \]
\[ 0 \]
\[ P \mid Q \]
\[ \!P \]
\[ \text{new} \ a; \ P \]
\[ \text{let} \ x=g(M_1,\ldots,M_n) \ \text{in} \ P[\text{else} \ Q] \]
\[ \text{if} \ M = N \ \text{then} \ P[\text{else} \ Q] \]
\[ \text{let} \ x=M \ \text{in} \ P \]
\[ \text{event}(M).P \]
Formal Semantics

- Process evolution defined operationally by a transition relation on processes
  - process = state
- Destructor application defined by rewriting rules
Destructor Semantics

- Used to define the (ideal) properties of cryptographic and data manipulation primitives

- Example: modeling **Shared-key encryption**
  - Constructor: senc (x, y) encrypts x with key y
  - Destructor: sdec (x, y) decrypts x with key y
  - Rewrite rules: sdec(senc(x,y), y) \(\rightarrow\) x
  - Proverif syntax:

\[
\text{fun senc/2.} \\
\text{reduc sdec(senc(x,y), y) = x.}
\]
Other Example: Public-key Encryption

- Constructors: penc (x,y) encrypts x with public key y
  \( pk(x) \) returns the public key given the key pair x
  \( sk(x) \) returns the secret key given the key pair x
- Destructor: pdec (x,y) decrypts x with secret key y
- Rewrite rules: \( \text{pdec}(\text{penc}(x, pk(y)), sk(y)) \rightarrow x \)
- Proverif syntax:

```
fun penc/2.
fun pk/1.
fun sk/1.
reduc pdec(penc(x,pk(y)),sk(y)) = x.
```
Other Example: Digital Signature

- Constructors: sign (x,y) signs x with private key y
  
  pk(x) returns the public key given the key pair x
  sk(x) returns the secret key given the key pair x
- Destructors: getmess (x) extracts message from signature
  
  checksign(x,y) checks signature x with public key y
- Rewrite rules: getmess(sign(x,y)) → x
  
  checksign(sign(x,sk(y)), pk(y)) = ok
- Proverif syntax:

```
fun ok/0.
fun sign/2.
reduc getmess(sign(m,k)) = m.
reduc checksign(sign(m,k), pk(k)) = ok.
```
Other Example: Crypto Hashing

- Constructors: hash (x) computes the hash of x
- Destructors: no destructor defined (hashing cannot be inverted)
- Rewrite rules: no rewrite rule needed
- Proverif syntax:

```plaintext
fun hash/1.
```
Example: Handshake Protocol

- **Message 1**: $S \rightarrow C$: $\{\{k\}_s\}_k$ fresh
- **Message 2**: $C \rightarrow S$: $\{s\}_k$

**PS** = new $k$; out(c, penc (sign(k, sk(kpS)), pk(kpC))); in(c,x); let xs=sdec(x, k) in 0

**PC** = in(c, y); let y1=pdec(y, sk(kpC)) in if checksign(y1, pk(kpS))=ok then let xk=getmess(y1) in out(c, senc(s, xk)); 0

P = new kpS; new kpC; (!out(c, pk(kpS)); 0 | !out(c, pk(kpC)); 0) | !PA | !PB)
Specifying Properties: Secrecy

- **Intuitive property**: an attacker must not be able to get closed terms that are intended to be secret (e.g. names in the Handshake protocol)
Specification of Secrecy

• S-Adversary: any closed process Q with fn(Q) ⊆ S (fn(Q) is adversary initial knowledge: the unrestricted names of Q)

• Trace T outputs N iff T contains a step where N is output to channel M ∈ S

• Closed process P preserves the secrecy of N from S-Adversaries if
  ∀ S-Adversary Q, ∀ trace T executed by P|Q
  T does not output N.
Approximation

• The Horn clauses approximate the protocol behavior specified in extended pi calculus:
  – The number of times a message is sent is not represented by the Horn clauses => it is as though each message could be sent and received an arbitrary number of times
  – The Horn clauses distinguish different fresh names only partially => two fresh names could be represented by the same name

• These approximations have been proved sound:
  – If one proves a secrecy property holds in the Horn clauses model, the corresponding property holds in the pi model
Relating the two Models

• The logic theory described by the Horn clauses over-approximates the behavior of the real protocol
  – Freshness, repetition of send/receive

=> false positives are possible

Facts that are really true
Facts that can be derived
Example

Process

```
new privc;
  (out(privc,s); out(pubc,privc); 0 | in(privc,x); 0)
```

preserves the secrecy of $s$ against \{pubc\}-Adversaries

but Proverif cannot prove it, because the Proverif model corresponds to:

```
new privc;
  (! out(privc,s); out(pubc,privc); 0 | ! in(privc,x); 0)
```
Correspondence Properties

• Specify order relationships that should bind trace events

• Can be used to specify authentication (e.g. agreement)
Example: Authentication in the Handshake Protocol

PS= new k;
  event(bS(pk(kpS), pk(kpC), k));
  out(c, penc (sign(k, sk(kpS)), pk(kpC)));
  in(c, x); let xs=sdec(x, k) in 0

PC= in(c, y); let y1=pdec(y, sk(kpC)) in
  if checksign(y1, pk(kpS))=ok then let xk=getmess(y1) in
  event(eC(pk(kpS), pk(kpC), xk));
  out(c, senc(s, xk)); 0

• In each trace, if event eC(x,y,z) occurs, event bS(x,y,z) must have occurred before
  (ev:eC(x,y,z) ==> ev:bS(x,y,z)).
Injectivity of Correspondences

• The basic correspondence
  \[ \text{ev:}e(x_1,\ldots,x_n) \implies \text{ev:}e'(x_1,\ldots,x_n) \]

  is \textbf{non-injective}: it is true even when the same execution of event \( e'(x_1,\ldots,x_n) \) corresponds to more executions of event \( e(x_1,\ldots,x_n) \)

• \textbf{Injective} correspondence
  \[ \text{ev:}e(x_1,\ldots,x_n) \implies \text{evinj:}e'(x_1,\ldots,x_n) \]

  requires that each occurrence of event \( e(x_1,\ldots,x_n) \) corresponds to a \texttt{distinct} occurrence of event \( e'(x_1,\ldots,x_n) \)
General Correspondences

• Correspondences can be combined together to form more complex queries:
  – ev:e(x₁,x₂) ==> (evinj:e2(x₁,x₂) ==> evinj:e1(x₁))
  – ev:e(x₁,x₂) ==> (evinj:e2(x₁,x₂) ==> evinj:e1(x₁))
    | (evinj:e4(x₁,x₂) ==> evinj:e3(x₁))

• The query
  – ev:e(x₁,x₂) means event e(x₁,x₂) is never executed
Termination

• The resolution algorithm may not terminate

• Termination has been proved for a class of tagged protocols (Blanchet, Podelski, TCS 2005):
  – Limited set of primitives (including all the main ones)
  – No private channels
  – Crypto functions always applied to tagged data (with different tags for each occurrence of each function)
  – Tags always checked on application of destructors
  – No else in destructor applications
  – Atomic keys
Attack Reconstruction

• If the resolution algorithm does not find a proof, proverif performs a state exploration in order to look for an attack (counterexample)

• This search may yield false attacks

• If no attack trace is found, an attack may still exist
References

• **Verification of Secrecy**

• **Verification of Correspondences**
JavaSPI

• A "Java" interface to Proverif and a framework for model-based security protocol development:
  – write protocol models in Java (with some restrictions)
  – use regular Java IDE and debugger for model simulation
  – automatic generation of Proverif model from Java model
  – automatic translation of Java model to Java implementation code (annotation-guided automatic refinement)
    • The translation algorithm has been proved sound
    • => if a property has been proved by Proverif, it holds on the Java implementation too!
JavaSPI Architecture

Diagram showing the architecture of JavaSPI, including the steps from protocol definition, through parsing, to verification and testing. The diagram includes components like ProVerif, Java, and Spike2Java Framework.
References


– [http://spi2java.polito.it/](http://spi2java.polito.it/)