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

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-neglible 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 is true (and proved)

Analysis does not terminate

Property cannot be proved

Potential attack

Attack Reconstruction

Potential attack
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

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

\[ P, \; Q ::= \]

- \( \text{out}(M,N).P \) \quad & \quad \text{output } N \text{ to channel } M \\
- \( \text{in}(M,x).P \) \quad & \quad \text{input from channel } M \\
- \( 0 \) \quad & \quad \text{nil} \\
- \( P | Q \) \quad & \quad \text{parallel composition} \\
- \( !P \) \quad & \quad \text{replication} \\
- \( \text{new } a; \; P \) \quad & \quad \text{creation of restricted data} \\
- \( \text{let } x = g(M_1, \ldots, M_n) \text{ in } P[\text{else } Q] \) \quad & \quad \text{destructor application} \\
- \( \text{if } M = N \text{ then } P[\text{else } Q] \) \quad & \quad \text{equality test} \\
- \( \text{let } x = M \text{ in } P \) \quad & \quad \text{assignment} \\
- \( \text{event}(M).P \) \quad & \quad \text{event}
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: \( \text{senc}(x, y) \) encrypts \( x \) with key \( y \)
  – Destructor: \( \text{sdec}(x, y) \) decrypts \( x \) with key \( y \)
  – Rewrite rules: \( \text{sdec}(\text{senc}(x, y), y) \rightarrow x \)
  – Proverif syntax:

```
fun senc/2.
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: \(pdec(penc(x,pk(y)),sk(y)) \rightarrow x\)
- 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:
  
  \(\text{getmess}(\text{sign}(x,y)) \rightarrow x\)
  
  \(\text{checksing}(\text{sign}(x,sk(y)), pk(y)) = \text{ok}\)

- Proverif syntax:

  \begin{verbatim}
  fun ok/0.
  fun sign/2.
  reduc getmess(sign(m,k)) = m.
  reduc checksing(sign(m,k), pk(k)) = ok.
  \end{verbatim}
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:

```proverif
define hash/1.
```


Example: Handshake Protocol

- Message 1  \( S \rightarrow C: \)  \( \{\{k\}_{skS}\}_{pkC} \)  \( k \) fresh
- Message 2  \( C \rightarrow S: \)  \( \{s\}_k \)

\[
\begin{align*}
PS &= \text{new } k; \text{ out}(c, \text{ penc } (\text{sign}(k, sk(kpS)), pk(kpC))); \\
PC &= \text{in}(c, y); \text{ let } y1=\text{pdec}(y, sk(kpC)) \text{ in} \\
& \quad \text{if } \text{checksign}(y1, pk(kpS))=\text{ok} \text{ then} \\
& \quad \quad \text{let } xk=\text{getmess}(y1) \text{ in} \\
& \quad \quad \text{out}(c, \text{ senc}(s, xk)); 0 \\
P &= \text{new kpS; new kpC; } \\
& \quad (\text{!out}(c, pk(kpS)); 0 | \text{!out}(c, pk(kpC)); 0) | \text{!PA } | \text{!PB})
\end{align*}
\]
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
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;
\[\text{event}(bS(pk(kpS), pk(kpC), k))\];
\[\text{out}(c, \text{penc} (\text{sign}(k, \text{sk}(kpS)), pk(kpC)))\];
\[\text{in}(c, x); \text{let} x_s = \text{sdec}(x, k) \text{ in } 0\]

PC = \[\text{in}(c, y); \text{let} y_1 = \text{pdec}(y, \text{sk}(kpC)) \text{ in}\]
\[\text{if} \ \text{checksign}(y_1, pk(kpS)) = \text{ok} \ \text{then} \ \text{let} x_k = \text{getmess}(y_1) \ \text{in}\]
\[\text{event}(eC(pk(kpS), pk(kpC), x_k))\];
\[\text{out}(c, \text{senc}(s, x_k)); 0\]

- In each trace, if event eC(x, y, z) occurs, event bS(x, y, z) must have occurred before
  \[(ev:eC(x, y, z) \implies ev:bS(x, y, z)).\]
Injectivity of Correspondences

• The basic correspondence
  \[ \text{ev}:e(x_1,\ldots,x_n) \Rightarrow \text{ev}:e'(x_1,\ldots,x_n) \]
  is **non-injective**: it is true even when the same execution of event \( e'(x_1,\ldots,x_n) \) corresponds to more executions of event \( e(x_1,\ldots,x_n) \)

• **Injective** correspondence
  \[ \text{ev}:e(x_1,\ldots,x_n) \Rightarrow \text{evinj}:e'(x_1,\ldots,x_n) \]
  requires that each occurrence of event \( e(x_1,\ldots,x_n) \) corresponds to a **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**