Symbolic Constraint Solving
Overview

◆ Strand space model
◆ Protocol analysis with unbounded attacker
  • Parametric strands
  • Symbolic attack traces
  • Protocol analysis via constraint solving
◆ SRI constraint solver
Protocol Analysis Techniques

Crypto protocol analysis

- Probabilistic model checking
  - Inductive method
    - Finite-state checking
      - Finite processes, finite attacker
    - Symbolic analysis
      - Finite processes, infinite attacker
  - Model checking
    - Random oracle
      - Probabilistic ptime calculus
      - Probabilistic I/O automata
  - Protocol logics

- Computational models
  - Dolev-Yao (perfect cryptography)

Fully automated methods always terminate and give an answer
Obtaining a Finite Model

◆ Two sources of infinite behavior
  • Multiple protocol sessions, multiple participants
  • Message space or data space may be infinite

◆ Finite approximation
  • Assume finite sessions
    - Example: 2 clients, 2 servers
  • Assume finite message space
    - Represent random numbers by r1, r2, r3, ...
    - Do not allow encrypt(encrypt(encrypt(…))))

This restriction is necessary (or the problem is undecidable)

This restriction is not necessary for fully automated analysis!
Decidable Protocol Analysis

◆ Eliminate sources of undecidability
  • Bound the number of protocol sessions
    – Artificial bound, no guarantee of completeness
  • Bound structural size of messages by lazy instantiation of variables
  • Loops are simulated by multiple sessions

◆ Secrecy and authentication are NP-complete if the number of protocol instances is bounded
  [Rusinowitch, Turuani ’01]

◆ Search for solutions can be fully automated
  • Several tools; we’ll talk about SRI constraint solver
Strand Space Model

[Thayer, Herzog, Guttman ’98]

◆ A strand is a representation of a protocol “role”
  • Sequence of “nodes”
  • Describes what a participant playing one side of the protocol must do according to protocol specification

◆ A node is an observable action
  • “+” node: sending a message
  • “-” node: receiving a message

◆ Messages are ground terms
  • Standard formalization of cryptographic operations: pairing, encryption, one-way functions, …
Participant Roles in NSPK

Protocol

A → B \{n,A\}_{kb}
B → A \{n,r\}_{ka}
A → B \{r\}_{kb}

“A” role
A → \{n,A\}_{kb}
A ← \{n,r\}_{ka}
A → \{r\}_{kb}

“B” role
B ← \{n,A\}_{kb}
B → \{n,r\}_{ka}
B ← \{r\}_{kb}

Controls network and can schedule any consistent interleaving of these roles
NSPK in Strand Space Model

Each primitive capability of the attacker is a “penetrator” strand

Same set of attacker strands for every protocol
Bundles

◆ A bundle combines strands into a partial ordering
  • Nodes are ordered by internal strand order
  • “Send message” nodes of one strand are matched up with “receive message” nodes of another strand

◆ Infinitely many possible bundles for any given set of strands
  • No bound on the number of times any given attacker strand may be used

◆ Each bundle corresponds to a particular execution trace of the protocol
  • Conceptually similar to a Mur$\varphi$ trace
NSPK Attack Bundle
Parametric Strands

- Use a variable for every term whose value is not known to recipient in advance

Parametric “A” strand

- “Talk to B”
  + \{n,A\}_{pk(B)}
  - \{n,r\}_{pk(A)}
  + \{r\}_{pk(B)}

Parametric “B” strand

- “Talk to X”
  + \{n,A\}_{pk(X)}
  - \{n,Z\}_{pk(A)}
  + \{Z\}_{pk(X)}

- “Talk to B”
  + \{Y,r\}_{pk(A)}
  - \{Y,A\}_{pk(B)}
  + \{n,r\}_{pk(A)}

+ “Talk to B”
  - \{r\}_{pk(B)}
  - \{n,A\}_{pk(B)}
  - \{r\}_{pk(B)}
Properties of Parametric Strands

◆ Variables are untyped
  • Attacker may substitute a nonce for a key, an encrypted term for a nonce, etc.
  • More flexible; can discover more attacks

◆ Compound terms may be used as symmetric keys
  • Useful for modeling key establishment protocols
    - Keys constructed by exchanging and hashing random numbers
  • Public keys constructed with pk(A)

◆ Free term algebra
  • Simple, but cannot model some protocols
  • No explicit decryption, no cryptographic properties
Attack Scenario

Partial bundle corresponding to attack trace

- By contrast, in Mur$\varphi$ need to specify attack state
- Assume that the attacker will intercept all messages

Is there a way to insert attacker strands here so that attacker learns secret $r$ in the resulting bundle?
Attack Scenario Generation

◆ Choose a finite number of strands
◆ Try all combinations respecting partial order imposed by individual strands
  • If node L appears after node K in the same strand, then L must appear after K in the combination bundle
  • Two strands of size m & n ⇒ choose(m+n, n) variants
◆ Optimization to reduce number of variants
  • The order of “send message” nodes doesn’t matter: attacker will intercept all sent messages anyway
  • If this is the only difference between two combinations, throw one of them away
### Attack Scenario: Example

#### A’s role

- A→ "Talk to X"
- A→ \{n,A\}_{pk(X)}
- A← \{n,Z\}_{pk(A)}
- A→ \{Z\}_{pk(X)}

#### B’s role

- B→ "Talk to B"
- B← \{A,Y\}_{pk(B)}
- B→ \{Y,r\}_{pk(A)}
- B← \{r\}_{pk(B)}

Try all possible ways to plug attacker in the middle, for example:

- B→ E "Talk to B"
- A← E "Talk to X"
- A→ E \{n,A\}_{pk(X)}
- B← E \{A,Y\}_{pk(B)}
- B→ E \{Y,r\}_{pk(A)}
- A← E \{n,Z\}_{pk(A)}
- A→ E \{Z\}_{pk(X)}
- E r

◆ This is a symbolic attack trace
  - Variables are uninstantiated
◆ It may or may not correspond to a concrete trace
Symbolic Attack Scenarios

◆ Attack is modeled as a symbolic execution trace
  • Trace is a sequence of message send and receive events
  • Attack trace ends in a violation
    – E.g., attacker outputs the secret
  • Messages contain variables
    – Variables represent data controlled by attacker

◆ Adequate for trace-based security properties
  • Secrecy, authentication, some forms of fairness…

◆ A symbolic trace may or may not have a feasible concrete instantiation
  • Goal: discover whether a feasible instantiation exists
From Attack Traces to Constraints

- Any symbolic execution trace is equivalent to a sequence of symbolic constraints.

<table>
<thead>
<tr>
<th>[ m \text{ from } t_1, \ldots, t_n ]</th>
</tr>
</thead>
</table>

Can the attacker learn message \( m \) from terms \( t_1, \ldots, t_n \)?

- A constraint is satisfiable if and only if \( m \) can be derived from \( t_1, \ldots, t_n \) in attacker term algebra.
  - Attacker term algebra is an abstract representation of what the attacker can do.
Constraint Generation: Example

**Attack Trace**

- B → E \hspace{5pt} “Talk to B”
- A ← E \hspace{5pt} “Talk to X”
- A → E \hspace{5pt} \{n,A\}_{pk(X)}
- B ← E \hspace{5pt} \{A,Y\}_{pk(B)}
- B → E \hspace{5pt} \{Y,r\}_{pk(A)}
- A ← E \hspace{5pt} \{n,Z\}_{pk(A)}
- A → E \hspace{5pt} \{Z\}_{pk(X)}
- E ← \hspace{5pt} r

**Symbolic Constraints**

- “Talk to X” from $T_0$ (attacker’s initial knowledge)
- \{A,Y\}_{pk(B)} from $T_0$, \{n,A\}_{pk(X)}
- \{n,Z\}_{pk(A)} from $T_0$, \{n,A\}_{pk(X)}, \{Y,r\}_{pk(A)}
- r from $T_0$, \{n,A\}_{pk(X)}, \{Y,r\}_{pk(A)}, \{Z\}_{pk(X)}
Symbolic Constraint Generation

◆ For each message sent by the attacker in the attack trace, create symbolic constraint

\[
\text{m}_i \text{ from } t_1, \ldots, t_n
\]

• \( m_i \) is the message attacker needs to send
• \( t_1, \ldots, t_n \) are the messages observed by attacker up to this point (may contain variables)

◆ Attack is feasible if and only if all constraints are satisfiable simultaneously

• There exists an instantiation \( \sigma \) such that \( \forall i \ m_i \sigma \) can be derived from \( t_1 \sigma, \ldots, t_n \sigma \) in attacker’s term algebra
Dolev-Yao Term Algebra

Attacker’s term algebra is a set of derivation rules

\[
\begin{align*}
v \in T & \quad \text{if } u = v^\sigma \text{ for some } \sigma \\
T \arr u & \\
T \arr [u,v] & \\
T \arr \text{crypt}_u[v] & \\
T \arr [u,v] & \\
T \arr u & \\
T \arr v & \\
T \arr \text{crypt}_u[v] & \\
T \arr u & \\
T \arr v &
\end{align*}
\]

Symbolic constraint $m$ from $t_1, \ldots, t_n$ is satisfiable if and only if there is a substitution $\sigma$ such that $t_1^\sigma, \ldots, t_n^\sigma \arr m^\sigma$ is derivable using these rules
Solving Symbolic Constraints

Constraint reduction rules
- Replace each $m_i$ from $T_i$ with one or more simpler constraints
- Preserve essential properties of the constraint sequence

Nondeterministic reduction procedure
- Structure-driven, but several rules may apply in any state
- Exponential in the worst case (the problem is NP-complete)

The procedure is terminating and complete
- If $T_\sigma \triangleright m_\sigma$ is derivable in attacker’s term algebra,
  1. There exists reduction rule $r=r(\sigma)$ which is applicable to $m$ from $T$ and produces some $m'$ from $T'$ such that
  2. $T'\sigma \triangleright m'\sigma$ is derivable in attacker’s term algebra
Reduction Procedure

Initial constraint sequence

Nondeterministically apply special transformation rules to first $m$ from $T$ where $m$ is not a variable

... ...

No rule is applicable or

$\text{var}_1 \text{ from } T_1$
$\text{...}$
$\text{var}_N \text{ from } T_N$

If reduction tree has at least one such sequence as a leaf, there is a solution, and attack scenario is feasible
From Protocols to Constraints

Formal specification of protocol roles

Choose finite number of role instances

Choose an interleaving corresponding to an attack

Sequence of symbolic constraints

Constraint solving procedure

This is the only thing the user has to specify!

contains variables & may not have a feasible instantiation

satisfiable ⇔ there exists a feasible instantiation
SRI Constraint Solver

◆ Easy protocol specification
  • Specify only protocol rules and correctness condition
  • No explicit intruder rules!

◆ Fully automated protocol analysis
  • Generates all possible attack scenarios
  • Converts scenario into a constraint solving problem
  • Automatically solves the constraint sequence

◆ Fast implementation
  • Three-page program in standard Prolog (SWI, XSB, etc.)

http://www.csl.sri.com/users/millen/capsl/constraints.html
A Tiny Bit of Prolog (I)

◆ Atoms
  • a, foo_bar, 23, 'any.string'

◆ Variables
  • A, Foo, _G456

◆ Terms
  • f(N), [a,B], N+1
A Tiny Bit of Prolog (II)

Clauses define terms as relations or predicates

• factorial(1,1).  \(\text{Fact, true as given}\)
• factorial(N,M) :-  \(\text{...is true if...}\)
  \(\begin{align*}
    & N>1, \quad \text{condition for this case} \\
    & N1 \text{ is } N-1, \quad \text{"is" to do arithmetic} \\
    & \text{factorial}(N1,M1), \quad \text{recursive call to find } (N-1)! \\
    & M \text{ is } N \times M1. \quad M = N! = N(N-1)!
  \end{align*}\)
Using Prolog

◆ **Put definitions in a text file**

```
../factdef or ../\factdef.pl
```

◆ **Start Prolog**

```
swipl, pl or plwin.exe
```

```
Prolog prompt
```

◆ **Load definitions file**

```
?- reconsult(factdef).
?- [factdef].
?- ['examples/factdef'].
```

◆ **Execute query**

```
?- factorial(3,M).
M=6
Yes
?- halt.
```

```
Start search for true instance
Prolog responds
Quit Protocol session.
```

◆ **Both UNIX and Windows**

```
consult(factdef) in SWI-Prolog
```

```
subdirectory, need quotes
``
Defining a Protocol: Terms

◆ Constants
  • a, b, e, na, k, ...
    e is the name of the attacker

◆ Variables
  • A, M, ...
    by convention, names capitalized

◆ Compound terms
  • [A,B,C]
    n-ary concatenation, for all n > 1
  • A+K
    symmetric encryption
  • A*pk(B)
    public-key encryption
  • sha(X)
    hash function
  • f(X,Y)
    new function unknown to attacker
Specifying Protocol Roles

strand(roleA, A, B, Na, Nb, [send([A, Na]*pk(B)), recv([Na, Nb]*pk(A)), send(Nb*pk(B))]).

strand(roleB, A, B, Na, Nb, [recv([A, Na]*pk(B)), send([Na, Nb]*pk(A)), recv(Nb*pk(B))]).

- No need to specify rules for the intruder
- No need to check that messages have correct format
Specifying Secrecy Condition

◆ Special secrecy test strand

\[
\text{strand(secrecytest,X,[recv(X),send(stop)])}.
\]

◆ When the attacker has learned the secret, he’ll pass it to this strand to “announce” that the attack has succeeded
Choosing Number of Sessions

◆ Choose number of instances for each role
  • For example, one sender and two recipients
◆ In each instance, use different constants to instantiate nonces and keys created by that role

```prolog
nspk0([Sa, Sb1, Sb2]) :-
  strand(roleA, a, B1, Na, Nb, Sa),
  strand(roleB, a, b, Na1, nb1, Sb1),
  strand(roleB, A3, b, Na2, nb2, Sb2).
```

1 instance of role A, 2 instances of role B
Each nonce modeled by a separate constant
Each instance has its own name
Verifying Secrecy

◆ Add secrecy test strand to the bundle

\[\text{nspk0}([Sa,Sb1,Sb2,St]) \leftarrow \]
\[\text{strand(roleA,a,B1,na,Nb,Sa)}, \]
\[\text{strand(roleB,a,b,Na1,nb1,Sb1)}, \]
\[\text{strand(roleB,A3,b,Na2,nb2,Sb2)}, \]
\[\text{strand(secrecytest,nb1,St)}. \]

◆ This bundle is solvable if and only if the attacker can learn secret \text{nb1} and pass it to test strand

◆ Run the constraint solver to find out

\[\leftarrow \text{nspk0}(B), \text{search}(B, []). \]

◆ This is it! Will print the attack if there is one.
Specifying Authentication Condition

◆ What is authentication?
  • If B completes the protocol successfully, then there is or was an instance of A that agrees with B on certain values (each other’s identity, some key, some nonce)

◆ Use a special authentication message
  \[
  \text{send}(\text{roleA}(a,b,nb))
  \]
  “A believes he is talking to B and B’s nonce is nb”

◆ Attack succeeds if B completes protocol, but A’s doesn’t send authentication message
  • B thinks he is talking to A, but not vice versa
NSPK Strands for Authentication

strand(roleA,A,B,Na,Nb,[
    send([A,Na]*pk(B)),
    recv([Na,Nb]*pk(A)),
    send(roleA(A,B,Nb)),
    send(Nb*pk(B))
]).

strand(roleB,A,B,Na,Nb,[
    recv([A,Na]*pk(B)),
    send([Na,Nb]*pk(A)),
    recv(Nb*pk(B)),
    send(roleB(A,B,Na))
]).
Verifying Authentication

◆ Test for presence of authentication message

```prolog
nspk0([Sa, Sb, St], roleA(a, b, nb)) :-
    strand(roleA, a, B, na, Nb, Sa),
    strand(roleB, a, b, Na, nb, Sb),
    strand(secrecytest, roleB(a, b, na), St).
```

◆ This bundle is solvable if and only if the attacker can cause roleB(a, b, na) to appear in a trace that does not contain roleA(a, b, nb)

  • Convince B that he is talking A when A does not think he is talking to B.
Symbolic Analysis in a Nutshell

- Informal protocol description
- Participant roles
- All possible attack traces
- Symbolic constraints for each trace
- Automated constraint solving procedure

If constraints are satisfied, then there is an attack

This is the only thing user needs to specify