CS 259

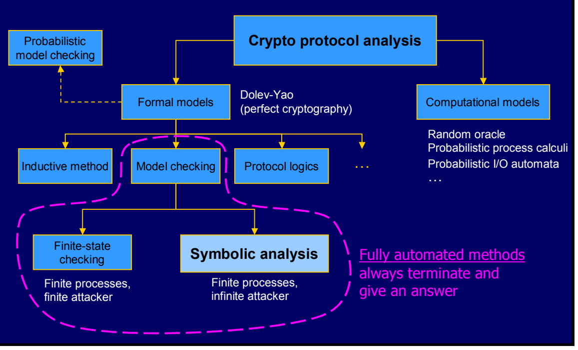
Symbolic Protocol Analysis

Vitaly Shmatikov

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



Obtaining a Finite Model

- Two sources of infinite behavior
  - Multiple protocol sessions, multiple participants
  - Message space or data space may be infinite
- Finite approximation (This restriction is necessary (or the problem is undecidable))
  - 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 **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

$A \rightarrow B \quad \{n,A\}_{kb}$

$B \rightarrow A \quad \{n,r\}_{ka}$

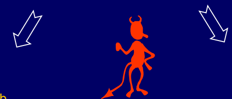
$A \rightarrow B \quad \{r\}_{kb}$

"A" role

$A \rightarrow \quad \{n,A\}_{kb}$

$A \leftarrow \quad \{n,r\}_{ka}$

$A \rightarrow \quad \{r\}_{kb}$



Controls network and can schedule any consistent interleaving of these roles

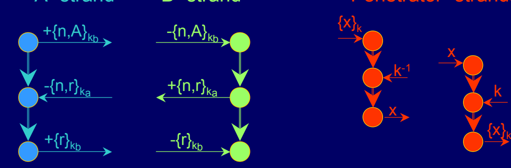
"B" role

$B \leftarrow \quad \{n,A\}_{kb}$

$B \rightarrow \quad \{n,r\}_{ka}$

$B \leftarrow \quad \{r\}_{kb}$

NSPK in Strand Space Model

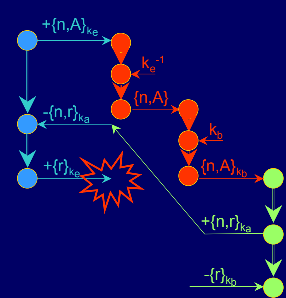


- Each primitive capability of the attacker is a "penetrator" strand
- Same set of attacker strands for every protocol

Bundles

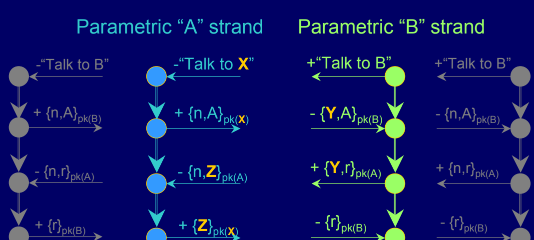
- A bundle combines strands into a partial ordering
  - Nodes are ordered by internal strand order
  - "Send message" nodes of one strand are matched up with "receive message" nodes of another strand
- Infinitely many possible bundles for any given set of strands
  - No bound on the number of times any given attacker strand may be used
- Each bundle corresponds to a particular execution trace of the protocol
  - Conceptually similar to a Murφ trace

NSPK Attack Bundle



Parametric Strands

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

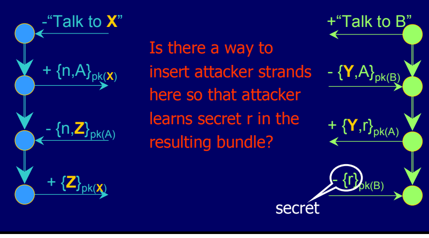


Properties of Parametric Strands

- 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 Murφ need to specify attack state
  - Assume that the attacker will intercept all messages



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 $\Rightarrow$ 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→ $\{Z\}_{pk(X)}$

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

B’s role

B→ “Talk to B”
B← $\{A,Y\}_{pk(B)}$
B→ $\{Y,r\}_{pk(A)}$
B← $\{r\}_{pk(B)}$

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

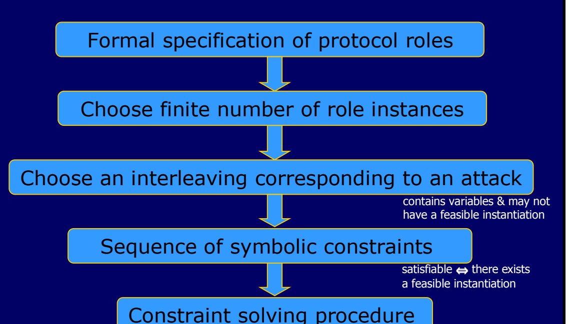
Symbolic Analysis Problem

- Attack modeled as a symbolic trace
  - Sequence of protocol messages with variables
  - Represents a successful attack
    - For example, attacker learns secret in the end
  - Adequate for secrecy, authentication, fairness
- Equivalent to a sequence of symbolic constraints

m **from** t_1, …, t_n	Can the attacker learn message m from terms t_1, …, t_n?

- This constraint is satisfiable *if and only if* there exists substitution σ such that attacker can derive $m\sigma$ from $t_1\sigma$, …, $t_n\sigma$

From Protocols to Constraints



Constraint Generation

- For each message attacker sends in the attack trace, create symbolic constraint m_i **from** T_i
  - m_i is the message attacker needs to send
  - T_i is set of messages previously observed by attacker
  - m_i,T_i may contain variables
- Attack is feasible *if and only if* all constraints are satisfiable simultaneously
  - There exists a substitution σ such that ∀i attacker can derive $m_i\sigma$ from $T_i\sigma$ using Dolev-Yao rules
  - Variables must be instantiated consistently in all terms

Constraint Generation: Example

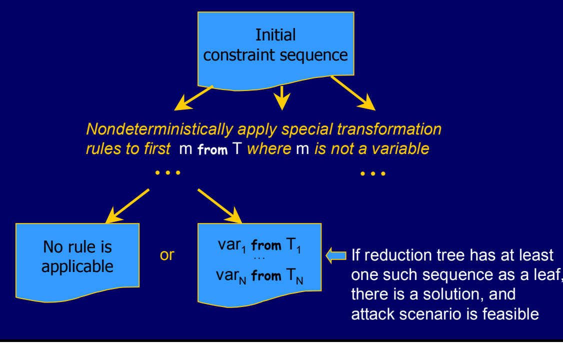
Attack Trace

- 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

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, $\{n,A\}_{pk(X)}$, $\{Y,r\}_{pk(A)}$, $\{Z\}_{pk(X)}$

Solving Constraint Sequence



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). *Fact, true as given*
  - factorial(N,M) :- *…is true if…*

```
    N>1,                 condition for this case
    N1 is N-1,           "is" to do arithmetic
    factorial(N1,M1),    recursive call to find (N-1)!
    M is N*M1.           M = N! = N(N-1)!
```

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). — consult(factdef) *in SWI-Prolog*
  - ?- [factdef]. — *Both UNIX and Windows*
  - ?- ['examples/factdef']. — *subdirectory, need quotes*
- Execute query
  - ?- factorial(3,M). — *Start search for true instance*
  - M=6 Yes — *Prolog responds*
  - ?- halt. — *Quit Protocol session.*

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

Name of the role — *Parameters of the role*

```
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))
]).
```

$A \rightarrow B: \{A,Na\}_{pk(B)}$
$B \rightarrow A: \{Na,Nb\}_{pk(A)}$
$A \rightarrow B: \{Nb\}_{pk(B)}$

Sending and receiving messages (just like in Murφ)

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

Specifying Secrecy Condition

- Special secrecy test strand

Forces analysis to stop as soon as this strand is executed

```
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

```
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

```
nspk0([Sa,Sb1,Sb2,St]) :-
      strand(roleA,a,B1,na,Nb,Sa),
      strand(roleB,A2,b,Na1,nb1,Sb1),
      strand(roleB,A3,b,Na2,nb2,Sb2),
      strand(secrecytest,nb1,St).
```

- This bundle is solvable if and only if the attacker can learn secret nb1 and pass it to test strand
- Run the constraint solver to find out

```
:- nspk0(B),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

  `send(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))
]).
```

A announces who he thinks he is talking to

B announces who he thinks he is talking to

Verifying Authentication

- Test for presence of authentication message

Only look at bundles where this message doesn't occur

```
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