PennState

CSE 597: Security of Emerging Technologies

Module: Cryptographic Protocol Verification

Prof. Syed Rafiul Hussain

Systems and Network Security (SyNSec) Research Group

Department of Computer Science and Engineering

The Pennsylvania State University

A Recipe for Automated Proof

- Translate program to a ProVerif script
  - Programs become pi calculus processes
  - Symbolic libraries become equational theories
  - Security goals become ProVerif queries

- Prove that if the ProVerif script is secure, so is the original program
  - Hand proof: the translation preserves reductions
  - Need to prove this only once

- Use ProVerif to automatically prove security

ProVerif

PennState

- It can handle an unbounded number of sessions of the protocol, thanks to some well-chosen approximations
- It can give false attacks, but if it claims that the protocol satisfies some property, then the property is actually satisfied.
- When the tool cannot prove a property, it tries to reconstruct an attack, that is, an execution trace of the protocol that falsifies the desired property

Simple Example

```
free app.
free net.

data msg/1.

fun enc/2.
reduc dec(enc(x,k),k) = x.

query ev:Accept(x) ==> ev:Send(x).
```

```
let client =
    in(app,x);
    event Send(x);
    let e = enc(x,kab) in
    out(net,msg(e)).

let server =
    in(app,z);
    in(net,msg(e));
    let x = dec(e,kab) in
    event Accept(x).

process
        new kab;
        (!client | !server)
```

- **msg:** constructor tag
- **enc-dec:** constructor--destructor pair
- **kab:** fresh name (key) Generated for this script
- **client, server:** replicated processes (share kab)

Running ProVerif

Terminal — bash — 88×25

```
bash-3.2$ proverif -in pi encdec.pv
Process:
new kab_6;
{1}!
(
    {7}in(app, x_10);
    {8}event Send(x_10);
    {9}let e_11 = enc(x_10,kab_6) in
    {10}out(net, msg(e_11));
    0
) | (
    {2}!
    {3}in(app, z_7);
    {4}in(net, msg(e_8));
    {5}let x_9 = dec(e_8,kab_6) in
    {6}event Accept(x_9);
    0
)

-- Query ev:Accept(x_12) ==> ev:Send(x_12)
Completing...
Starting query ev:Accept(x_12) ==> ev:Send(x_12)
goal reachable: begin:Send(x_53) & attacker:x_53 -> end:Accept(x_53)
RESULT ev:Accept(x_12) ==> ev:Send(x_12) is true.
bash-3.2$
```

PV's input language as a generic variant of the pi-calculus

- Syntax of Terms:

  - M,N::= terms
    - x,y,z variable
    - a,b,c,k name
    - f(M1,…,Mn) constructor application

- Allows users to define their own cryptographic primitives
  - E.g. encrypt((x,n), k)
  - Specify properties with destructors (more below).

Names, Channels, and Communication

$\mathbf{in}(M,x);P$	input of x from M (x has scope P)
$\mathbf{out}(M,N);P$	output of N on M
$\mathbf{new}\ a;P$	make new name a (a has scope P)

- Any value can be used as a channel
- Send and Receive are synchronous
  - But the continuation P may be 0
- Fresh names are generated by new
  - Such names may be used as private channels, or as nonces, or keys for crypto operations

Parallel Processes and Replication

$!P$	replication of P
$P \mid Q$	parallel composition
$\mathbf{0}$	inactivity

- !P is an unlimited number of copies of P

  !new a; P

  - This process generates fresh names a1, a2, … and uses each in a different copy of P

- Parallel composition is symmetric, associative

- 0 represents a finished process

Events

event *M*	event *M*

- A global log of events
- Any value can be logged as an event

PV's input language as a generic variant of the spi-calculus

- Syntax of Processes:
  - P,Q::= processes
    - out(c,M);P output
    - in(c,x);P input (this also declares the variable x)
    - 0 nil
    - P|Q parallel composition
    - !P replication
    - (new n);P restriction
    - let x=g(M1,…,Mn) in P else Q destructor application
    - let x=M in P local definition (this also declares the variable x)
    - if M=N then P else Q conditional

- e.g. new n; out(net, encrypt((x,n), k))

Three ways to generate names

- In a process: new a; P
  - Creates a fresh name, known only to P
  - P may choose to send it to other processes
  - E.g., new kab; (client | server)

- In a declaration: free a
  - Creates a fresh name known to all processes including the attacker.
  - E.g., free net, or free timestamp

- In a declaration: private free a
  - Creates a fresh name known only to good processes, and not known to the attacker
  - E.g., private free passwordDatabase

Free names

`[private] free id₁, ..., idₙ`

- Free names:
  - are public by default (e.g., untrusted channels, agent ids)
  - can optionally be declared private (e.g., trusted channels, global keys)
- Private free names are equivalent to names that are new-bound in front of the main process

Three kinds of constructors

- Invertible data constructors: data f/n
  - Both f(x) and $f^{-1}(x)$ are easily computable
  - E.g., data utf8/1.

- Functions: fun f/n
  - f(x) is computable, but $f^{-1}(x)$ may not be –
  - E.g., fun enc/2, fun sha1/1, fun hmacsha1/2

- • Private functions: private fun f/n
  - f(x) can be called by good processes but not by attacker
  - E.g., fun cookie/3.

Constructors

`[private] fun id/n`

- **Examples:**
  - `fun encrypt/2`
  - `fun sign/2`
  - `fun hash/1`

You can also declare constructors as private; it is kind of uncommon but useful, for instance for declaring the function that the server uses to retrieve the key she shares with a given user

Destructor Rules

- Destructors are defined by reduction rules
  - Forms a set of (directed) equations
  - E.g., **reduc** iutf8(utf8(x)) = x
  - E.g., **reduc** dec(enc(x,k), k) = x

- Multiple rules may apply
  - reduc errorCode(y, utf8(base64(x))) = Error1()
  - reduc errorCode(x, utf8(x)) = Error2()

- Private destructors: **private reduc**
  - Defines function that may not be used by attacker
  - E.g., reduc icookie(cookie(x,y,z)) = (x, y, z)

Constructors/destructors

- Constructors (-expected parameters):
  - `encrypt/2-(M,K)`: `M` encrypted with symmetric key `K`
  - `pencrypt/2-(M,enc(K))`: `M` encrypted with encryption key `enc(K)`
  - `enc/1-(K), dec/1-(K)`: key extraction
  - `ntuple/n-(`M_1`, ... ,`M_n`)`: n-tupling
  - `hash/1-(M)`: hashing
- Destructors:
  - `reduc decrypt(encrypt(x,y),y)=x`: symmetric key decryption
  - `reduc pdecrypt(pencrypt(x,enc(y)),dec(y))=x`: asymmetric key decryption
  - `reduc` *`ithOfn`*`(ntuple(`x_1`,... ,`x_n`))=`x_i

Process macros

```
let id = ⟨process⟩
```

- After this declaration you can refer to the `⟨process⟩` by `id`
- ProVerif textually replaces the `id` by the `⟨process⟩`

Pattern matching

- ProVerif supports pattern-matching both at input and in let-expressions
  - To match you need to precede the id with =
  - To bind omit the =

- Example:
  - let (=tag,=B, x) = decrypt(ctext,k) in …
    - This pattern is matched by a triple (tag, B, M) binding M to x
    - In this case x is used as a variable, but tag and B are not
  - it is syntactic suggar; how would you write it in SPI?

Events

- Events can be inserted into processes
- Used for correspondence assertions
  - We will see more on this when talking about authentication.
- They have no effect at runtime

- Examples:
  - `event beginSend(A,B,m)`
  - `event endSend(A,B,m)`

Queries: examples

- In the declaration section, you need to query for the properties that you want ProVerif to analyze:
  - Secrecy: queries if the attacker can obtain M
    - `query attacker : M`
  - Weak Authenticity: Many-one correspondence: queries if event M is always preceded by event N
    - `query ev: M ==> ev: N` (∀ parameters)
  - Strong Authenticity: One-one correspondence: queries if event M is always preceded by event N, and every trace contains at least as many N-events as M-events
    - `query evinj : M ==> evinj : N`

All you need to know is if the attacker has a given message.

Thanks

PennState

Thanks to Karthikeyan Bhargavan for some slides.