`spyer` user's guide

Sébastien BRIAIS

January 28, 2008

Abstract

In this document, we describe how to use `spyer` which is a cryptographic protocol compiler. It is based on the work presented in [BN05].

`spyer` is released under the GPL. The distribution can be downloaded from the `spyer` homepage located at:

`http://lampwww.epfl.ch/~sbriais/spyer/spyer.html`

1 What is `spyer`? How to use it?

1.1 Presentation

`spyer` is a cryptographic protocol compiler. More precisely, `spyer` takes as input a cryptographic protocol expressed with so-called protocol narrations and gives back the definition of a spi calculus system which corresponds. It can also give back what we call an executable narration. This executable narration can also be translated in spi calculus. The compilation algorithm is precisely given in [BN05]. For more information about the spi calculus, we point the reader to the foundation article of Abadi and Gordon [AG99]. Lots of cryptographic protocols can be found in [CJ97].

1.2 Usage of `spyer`

`spyer` works as a simple command line tool. The valid command line arguments include:

1. `-help`, `--help`:

   To display all valid command line arguments and what they are useful for.

2. `--pp`, `--no-pp`:

   To display (or not) the protocol narrations after preprocessing.

3. `--xnarr`, `--no-xnarr`:

   To display (or not) the executable narrations after compilation.

4. `--spi`, `--no-spi`:

   To display (or not) the spi calculus system translated from the executable narration obtained after compilation.

   The definitions can then be used with `sbc`.

5. `--sysname` agent

   To specify the name of the global spi calculus term representing the system. By default, the name is `System`.

6. `--auth` name

   To make the given name private to the system and known by all agents. The name should be generated by an agent in the protocol narration.

7. `--read-xnarr`

   To read directly an executable narration instead of a protocol narration. This is useful for translating an executable narration in spi calculus.

8. `--stdin`

   To read the input file on standard input instead of a file.

By default, the option `--spi` is activated.

1.3 Input syntax

1.3.1 Common definitions

There are three base entities: agents, names and variables.

Agents: An agent is any string beginning by a capital letter and followed by any sequence of letters, digits or underscore character.

For example,

- valid agents are: `A`, `A_`, `A_1`, `AB`, `Ab`, ...
- and invalid agents are: `a`, `_A`, ...

Names: A name is built with the same rule as an agent except that it begins with a small letter instead of a capital letter. Moreover, it should not begin by the string `x_` or the string `agent_`.

For example,

- valid names are: `a`, `a_`, `a_1`, `ab`, ...
- and invalid names are: `A`, `a#`, `x_`, `agent_A`, ...

Variables: A variable is any sequence of digits not beginning with 0 or the simple digit 0.

For example,

- valid variables are: `0`, `12`, `2713`, ...
- and invalid variables are: `00`, `a0`, ...

Additionnal rules:
If a string begins with `agent_` and the remaining of the string is a valid agent then the entire string represents the *same* agent as the second part of the string.

For example: `A` and `agent_A` represents the same agent.
If a string begins with `x_` and the remaining of the string is a valid variable then the entire string represents the *same* variable as the second part of the string.

For example: `0` and `x_0` represents the same variable.

1.3.2 Protocol narrations

A protocol narration is composed of two parts: first a sequence of declarations and then the narration itself.

```
      agents  ::=  agent | agent "," agents

     message  ::=  name | agent
                |  "<" message "," message ">"
                |  "enc" "(" message "," message ")"
                |  "hash" "(" message ")"
                |  "pub" "(" message ")"
                |  "priv" "(" message ")"

    messages  ::=  message | message messages

 declaration  ::=  "private" name
                |  agent "generates" name
                |  agents "know" messages
                |  agents "share" name

declarations  ::=  ϵ | declaration declarations

    exchange  ::=  agent "->" agent ":" message

   narration  ::=  ϵ | exchange narration

    protocol  ::=  declarations narration
```

1.3.3 Executable narrations

An executable narration is a sequence of simple actions.

```
 expression  ::=  name | agent | variable
               |  "<" expression "," expression ">"
               |  "fst" "(" expression ")"
               |  "snd" "(" expression ")"
               |  "enc" "(" expression "," expression ")"
               |  "dec" "(" expression "," expression ")"
               |  "hash" "(" expression ")"
               |  "pub" "(" expression ")"
               |  "priv" "(" expression ")"

       atom  ::=  "wff" "(" expression ")"
               |  "inv" "(" expression "," expression ")"
               |  "[" expression "=" expression "]"

    formula  ::=  atom | atom formula

     action  ::=  "new" name
               |  agent ":" "new" name
               |  agent ":" "?" variable
               |  agent ":" expression "!" expression
               |  agent ":" formula

 xnarration  ::=  ϵ | action xnarration
```

2 The Denning-Sacco protocol

2.1 A complete protocol narration

We can find the Denning-Sacco protocol in [CJ97] (page 47). Actually, the following narration represents this protocol:

```
(1) A -> S: A,B
(2) S -> A: E(Kas:B,Kab,T,E(Kbs:A,Kab,T))
(3) A -> B: E(Kbs:A,Kab,T)
```

The protocol narration (with declaration) is expressed in our syntax by:

```
$ cat denning-sacco.spyer
(* initial knowledge *)
A,B,S know A B S
A,B,S know t
A,S share kAS
B,S share kBS
```

```
S generates kAB

(* protocol narration *)
A -> S : <A,B>
S -> A : enc(<B,kAB,t,enc(<A,kAB,t>,kBS)>,kAS)
A -> B : enc(<A,kAB,t>,kBS)

(* Denning-Sacco protocol *)
```

2.2 After preprocessing

If we look at the protocol narration after preprocessing, this gives

```
$ ./spyer --pp --no-spi denning-sacco.spyer
A know A
A know B
A know S
B know A
B know B
B know S
S know A
S know B
S know S
A know t
B know t
S know t
private kAS
A know kAS
S know kAS
private kBS
B know kBS
S know kBS
S generates kAB
A -> S : <A,B>
S -> A : enc(<B,<kAB,<t,enc(<A,<kAB,t>>,kBS)>>>,kAS)
A -> B : enc(<A,<kAB,t>>,kBS)
```

Some syntactic sugar is indeed removed. This shows for example that the declaration

```
A,B,S know t
```

is actually expanded in

```
A know t
B know t
S know t
```

and the declaration

```
A,S share kAS
```

is expanded in

```
private kAS
A know kAS
S know kAS
```

Finally, note that for messages (not expressions), some syntactic sugar is also offered for tuples.

2.3 The compiled executable narration

We can now look at the compiled executable narration.

```
$ ./spyer --xnarr --no-spi denning-sacco.spyer
new kAS
new kBS
S: new kAB
A: S!<A,B>
S: ?0
S: (*2*)
   [B = snd(0)]
   [A = fst(0)]
S: A!enc(<B,<kAB,<t,enc(<A,<kAB,t>>,kBS)>>>,kAS)
A: ?1
A: (*4*)
   inv(snd(snd(snd(dec(1,kAS)))),snd(snd(snd(dec(1,kAS)))))
   inv(fst(snd(dec(1,kAS))),fst(snd(dec(1,kAS))))
   [B = fst(dec(1,kAS))]
   [t = fst(snd(snd(dec(1,kAS))))]
A: B!snd(snd(snd(dec(1,kAS))))
B: ?2
B: (*3*)
   inv(fst(snd(dec(2,kBS))),fst(snd(dec(2,kBS))))
   [A = fst(dec(2,kBS))]
   [t = snd(snd(dec(2,kBS)))]
```

The integer in commentary before each formula indicates the number of atoms in the formula.

2.4 The spi calculus system

We can finally get the spi calculus system. We give it the name DenningSacco.

```
$ ./spyer --sysname DenningSacco denning-sacco.spyer
agent A(agent_A, agent_B, agent_S, kAS, t) =
  'agent_S<<agent_A, agent_B>>.
    agent_A(x_1).
      {[agent_B = fst(dec(x_1, kAS))]
    /\ [t = fst(snd(snd(dec(x_1, kAS))))]
    /\ wff(dec(enc(kAS, snd(snd(snd(dec(x_1, kAS))))),
               snd(snd(snd(dec(x_1, kAS))))))
    /\ wff(dec(enc(kAS, fst(snd(dec(x_1, kAS)))),
               fst(snd(dec(x_1, kAS)))))}
        'agent_B<snd(snd(snd(dec(x_1, kAS))))>.0

agent B(agent_A, agent_B, kBS, t) =
  agent_B(x_2).
    {[agent_A = fst(dec(x_2, kBS))]
  /\ [t = snd(snd(dec(x_2, kBS)))]
  /\ wff(dec(enc(kBS, fst(snd(dec(x_2, kBS)))),
         fst(snd(dec(x_2, kBS)))))}0

agent S(agent_A, agent_B, agent_S, kAS, kBS, t) =
  (^kAB)
    (agent_S(x_0).
      {[agent_B = snd(x_0)]
    /\ [agent_A = fst(x_0)]}
        'agent_A<enc(<agent_B, <kAB, <t,
                     enc(<agent_A, <kAB, t>>, kBS)>>>, kAS)>.0)

agent DenningSacco(agent_A, agent_B, agent_S, t) =
  (^kAS, kBS)(A(agent_A, agent_B, agent_S, kAS, t)
           | B(agent_A, agent_B, kBS, t)
           | S(agent_A, agent_B, agent_S, kAS, kBS, t))
```

Note that the same result can be obtained with the following command:

```
$ ./spyer --xnarr --no-spi denning-sacco.spyer | \
  ./spyer --sysname DenningSacco --read-xnarr --stdin
```

2.5 Changing the status of a generated name

It is possible with the option `--auth` to change the status of a generated name. The name indicated become a private name shared by all agents of the system. By doing this, we simulate the fact that every agent learns magically the generated name in question. Thus, it can be used to perform some more checks.

For example, in the Denning-Sacco protocol, we can make as if the generated key `kAB` is known in advance by `A` and `B`.

First, let us see how the file is preprocessed:

```
$ ./spyer --auth kAB --pp --no-spi denning-sacco.spyer
A know A
A know B
A know S
B know A
B know B
B know S
S know A
S know B
S know S
A know t
B know t
S know t
private kAS
A know kAS
S know kAS
private kBS
B know kBS
S know kBS
private kAB
A know kAB
B know kAB
S know kAB
A -> S : <A,B>
S -> A : enc(<B,<kAB,<t,enc(<A,<kAB,t>>,kBS)>>>,kAS)
A -> B : enc(<A,<kAB,t>>,kBS)
```

This gives the following executable narration:

```
$ ./spyer --auth kAB --xnarr --no-spi denning-sacco.spyer
new kAS
new kBS
new kAB
A: S!<A,B>
S: ?0
S: (*2*)
   [B = snd(0)]
   [A = fst(0)]
S: A!enc(<B,<kAB,<t,enc(<A,<kAB,t>>,kBS)>>>,kAS)
A: ?1
A: (*4*)
   inv(snd(snd(snd(dec(1,kAS)))),snd(snd(snd(dec(1,kAS)))))
   [B = fst(dec(1,kAS))]
   [t = fst(snd(snd(dec(1,kAS))))]
   [kAB = fst(snd(dec(1,kAS)))] (* new *)
A: B!snd(snd(snd(dec(1,kAS))))
B: ?2
```

```
B: (*3*)
   [A = fst(dec(2,kBS))]
   [t = snd(snd(dec(2,kBS)))]
   [kAB = fst(snd(dec(2,kBS)))] (* new *)
```

We have indicated with the commentary `(* new *)` the new tests generated by this small change in the protocol narration.

References

[AG99] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The Spi calculus. *Journal of Information and Computation*, 148(1):1–70, 1999. An extended abstract appeared in the *Proceedings of the Fourth ACM Conference on Computer and Communications Security (Zürich, April 1997)*. An extended version of this paper appears as Research Report 149, Digital Equipment Corporation Systems Research Center, January 1998, and, in preliminary form, as Technical Report 414, University of Cambridge Computer Laboratory, January 1997.

[BN05] Sébastien Briais and Uwe Nestmann. A formal semantics for protocol narrations. In Springer-Verlag, editor, *Proceedings of TGC 05*, 2005.

[CJ97] John A. Clark and Jeremy L. Jacob. A survey of authentication protocol literature. Technical Report 1.0, University of York, 1997.