Scyther 1.0

User Manual
DRAFT

Cas Cremers

July 12, 2012

Contents

1 Introduction

Disclaimer: This is an early draft of the manual. Constructive criticism is very welcome, please contact Cas Cremers or report it on the scyther-users mailing list.

This is the user manual for the Scyther security protocol verification tool.

This manual consists of several parts. Some background is given in Section 2. Installing Scyther is explained in Section 3. In Section 4 we give a brief tutorial with some very simple examples to show the basics of the tool. Then we discuss things in more detail as we introduce the input language of the tool in Section 5, and modeling protocols is briefly discussed in Section 6. The usage of the tool is then explained in more detail in Section 7. Then, in Section 8 we discuss the output formats of Scyther and how these should be interpreted. Some more advanced topics are discussed in Section 9.

Online information

More help can be found online: see `http://people.inf.ethz.ch/cremersc/scyther/index.html` for the Scyther webpages, where the most up-to-date information can be found, and where users can subscribe to the Scyther mailing list.

2 Background

Scyther is a tool for the formal analysis of security protocols under the *perfect cryptography assumption*, in which it is assumed that all cryptographic functions are perfect: the adversary learns nothing from an encrypted message unless he knows the decryption key. The tool can be used to find problems that arise from the way the protocol is constructed. This problem is undecidable in general, but in practice many protocols can be either proven correct or attacks can be found.

It is not our intention to describe the full protocol model, nor any possible security properties here. For such matters we refer the reader to e.g. [1–4]. Thus, in this manual we assume the reader is familiar with the formal modeling of security protocols and their properties.

Not only is knowledge of security protocol models needed to use the Scyther tool, further knowledge is needed to know how to interpret the results that the tool produces in any useful way. In fact, the reader should be very cautious: security protocol models and their properties are intricate and it is easy to misinterpret the results.

Having said that, one of the main goals of Scyther is to help with the analysis of a protocol in such a way that for example attacks can be understood well. Thus, wherever possible the tool will give useful information on the results.

3 Installation

Scyther can be downloaded from the following website:

`http://people.inf.ethz.ch/cremersc/scyther/index.html`

Installation instruction are included. Scyther is available for the Windows, Linux and Mac OS platforms.

4 Quick start tutorial

Scyther takes as input a security protocol description that includes security claims, and evaluates these.

Start Scyther by executing the `scyther-gui.py` program in the Scyther directory. The program will launch two windows: the main window, in which files are edited, and the `about` window, which shows some information about the tool.

As an introductory example, we will verify the Needham-Schroeder protocol, and investigate an attack on it.

Go to the file→open dialog, and open the file `ns3.spdl` in the Scyther directory. Your main window should look like the one in Figure 4.

By convention, protocol description files have the extension `.spdl` (Security Protocol Description Language), but it can have any name. The file used in this example is shown in Appendix A.

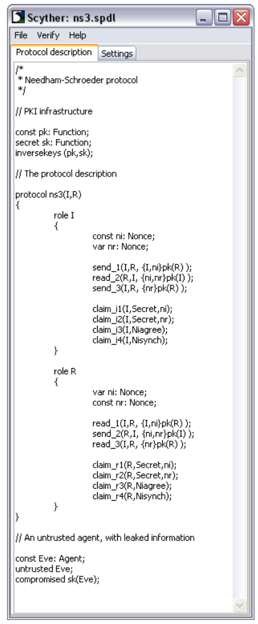

Figure 1: Scyther main window with the file ns3.spdl opened

Run the verification tool by selecting verify→verify_claims in the menu. A new window will appear during the verification process. Once verification is completed, the window will be replaced by the result window, as shown in Figure 4.

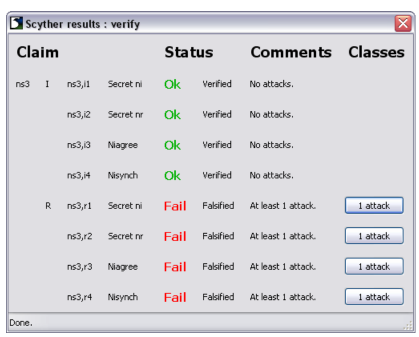

Figure 2: Scyther result window

The result window shows a summary of the claims in the protocol, and the verification results. Here one can find whether the protocol is correct, or false. In the next section there will be a full explanation of the possible outcomes of the verification process. The most important thing here is that if a protocol claim is incorrect, there exists at least one attack on the protocol. A button is shown next to the claim: press this button to view the attacks on the claim, as in Figure 4.

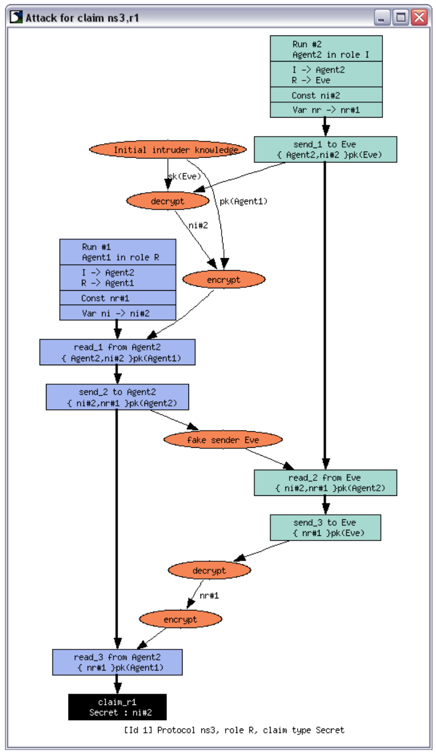

Figure 3: Scyther attack window

5 Input Language

Some initial remarks on the language:

- Comments can start with `//` or `#` (for single-line comments) or be enclosed by `/*` and `*/` (for multi-line comments). Note that the multi-line comments cannot be nested.
- Any whitespace between elements is ignored. It is therefore possible to use whitespace (spaces, tabs, newlines) to improve readability.
- A basic identifier consists of a string of characters from the set of alphanumeric characters as well as the symbols `^` and `-`.

- The language is case-sensitive, thus `NS3` is not the same identifier as `ns3`.

5.1 Terms

At the most basic level, Scyther manipulates terms.

5.1.1 Atomic terms

An atomic term can be any identifier, which is usually a string of alphanumeric characters.

Atomic terms can be combined into more complex terms by several operators, such as tupling and encryption.

5.1.2 Tupling

Any two terms can combined into a term tuple: we write `(x,y)` for the tupling of terms `x` and `y`. It is also allowed to write n-tuples as `(x,y,z)`.

5.1.3 Symmetric keys

Any term can act as a key for symmetrical encryption.

The encryption of `ni` with a term `kir` is written as:

```
{ ni }kir
```

Unless `kir` is explicitly defined as being part of an asymmetric key pair (explained below), this is interpreted as symmetric encryption.

5.1.4 Asymmetric keys

A public-key infrastructure (PKI) is predefined: `sk(X)` denotes the long-term private key of X, and `pk(X)` denotes the corresponding public key.

As an example, consider the following term. It represents the encryption of some term `ni` by the term `pk(I)`. Under normal conventions, this means that the nonce of the initiator (`ni`) is encrypted with the public key of the initiator.

```
{ ni }pk(I)
```

This term can only be decrypted by an agent who knows the secret key `sk(I)`.

Section 9.1 describes how to model more than one key pair per agent.

5.1.5 Hash functions

Hash functions are essentially encryptions with a function, of which the inverse is not known by anybody.

They can be used by first globally declaring an identifier to be a hashfunction, e.g.:

```
hashfunction H1;
```

Once declared, they can be used in protocol messages, e.g.:

```
H1(ni)
```

5.1.6 Predefined types

`Agent` Type used for agents.

`Function` A special type that defines a function term that can take a list of terms as parameter. By default, it behaves like a hash function: given the term `h(x)` where `h` is of type `Function`, it is impossible to derive `x`.

`Nonce` A standard type that is often used and therefore defined inside the tool.

`Ticket` A variable of type `Ticket` can be substituted by any term.

5.1.7 Usertypes

It is possible to define a new type. This can be done using the `usertype` command:

```
usertype MyAtomicMessage;

protocol X(I,R) {
   role I {
      var y: MyAtomicMessage;

      recv_1(I,R, y );
```

The effect of such a declaration is that variables of the new type can only be instantiated with messages `m` of that type, i.e., that have been declared by the global declaration `const m: MyAtomicMessage` or the freshly generated `fresh m: MyAtomicMessage` within a role.

5.2 Events

5.2.1 Recv and Send events

The `recv` and `send` events mark receiving and sending a message, respectively.

Note that in some protocol description files one may find the `read` keyword: this is obsolete syntax and can safely be substituted by `recv`.

In most cases, each `send` event will have a corresponding `recv` event. We specify this correspondence by giving such events the same label, denoted by a subscript.

For some protocols we may want to model sending or receiving to the adversary directly, in which case we have no corresponding event. If a `send` or `recv` event has no corresponding event, Scyther will output a warning. To surpress this warning, the label can be prefixed by a bang `!`, e.g.:

```
send_!1(I,I, LeakToAdversary );
```

5.2.2 Claim events and Security properties

Claim events are used in role specifications to model intended security properties. For example, the following claim event models that `sessKey` is meant to be secret.

```
claim(I, Secret, sessKey);
```

There are several predefined claim types.

`Secret` This claim requires a parameter term. Secrecy of this term is claimed as defined in [2].

`Alive` Aliveness (of all roles) as defined in [5].

`Weakagree` Weak agreement (of all roles) as defined in [5].

`Commit`, `Running` Non-injective agreement with a role on a set data items [5] can be defined by inserting the appropriate signal claims. In this context, `Commit` marks the effective claim, whose correctness requires the existence of a corresponding `Running` signal in the trace.

For an example of their use, see the "ns3.spdl" input file in the Scyther distribution. For a formal definition of the signals, see [5].

`Nisynch` Non-injective synchronisation as defined in [4].

`Niagree` Non-injective agreement as defined in [4].

`Reachable` When this claim is verified, Scyther will check whether this claim can be reached at all. It is true iff there exists a trace in which this claim occurs. This can be useful to check if there is no obvious error in the protocol specification, and is in fact inserted when the `--check` mode of Scyther is used.

`Empty` This claim will not be verified, but simply ignored. It is only useful when Scyther is used as a back-end for other verification means. For more on this, see Section 9.

5.3 Role definitions

Role definitions are sequences of events, i.e., declarations, send, receive, or claim events.

```
role Server {
  var x,y,z: Nonce;
  fresh n,m: Nonce;

  send_1(Server,Init, m,n );
  recv_2(Init,Server, x,y, { z }pk(Server) );
}
```

5.4 Protocol definitions

A protocol definition takes as a parameter a sequence of roles, which are then defined within its body.

```
protocol MyProt(Init,Resp,Server)
{
  role Init {
    ...
  }
  role Resp {
    ...
  }
  role Server {
    ...
  }
}
```

5.5 Global declarations

In many applications global constants are used. These include, for example, string constants, labels, or protocol identifiers.

They are modeled and used in the following way:

```
usertype String;
const HelloWorld: String;

protocol hello(I,R)
{
  role I {
    send_1(I,R, HelloWorld);
  }
  role R {
```

```
    recv_1(I,R, HelloWorld);
  }
}
```

5.6 Miscellaneous

The language also contains a command to include other files:

```
include "filename";
```

where *filename* denotes the name of the file that will be included at this point. Using this command, it is possible to share e.g. a set of common definitions between files. Typically this will include definitions for the key structures, and (untrusted) agent names. Nested use of this command is possible.

5.7 Language BNF

The full BNF grammar for the input language is given below. In the strict language definition, there are no claim terms such as `Niagree` and `Nisynch`, and neither are there any predefined type classes such as `Agent`. Instead, they are predefined constant terms in the Scyther tool itself.

5.7.1 Input file

An input file is simply a list of spdl constructions, which are global declarations or protocol descriptions.

⟨*spdlcomplete*⟩ ::= ⟨*spdl*⟩ { ';' ⟨*spdl*⟩ }

⟨*spdl*⟩ ::= ⟨*globaldeclaration*⟩
 | ⟨*protocol*⟩

5.7.2 Protocols

Note that a protocol is simply a container for a set of roles. Because we use a role-based approach to describing roles, this affects only the naming of the roles: a role "I" in a protocol "ns3" will be assigned the global name "ns3.I".

⟨*protocol*⟩ ::= '`protocol`' ⟨*id*⟩ '`(`' ⟨*termlist*⟩ '`)`' '`{`' ⟨*roles*⟩ '`}`' ['`;`']

5.7.3 Roles

⟨*roles*⟩ ::= ⟨*role*⟩ [⟨*roles*⟩]
 | ⟨*declaration*⟩ [⟨*roles*⟩]

⟨*role*⟩ ::= ['`singular`'] '`role`' ⟨*id*⟩ '`{`' ⟨*roledef*⟩ '`}`' ['`;`']

⟨*roledef*⟩ ::= ⟨*event*⟩ [⟨*roledef*⟩]
 | ⟨*declaration*⟩ [⟨*roledef*⟩]

5.7.4 Events

⟨*event*⟩ ::= ‘`recv`’ ⟨*label*⟩ ‘`(`’ ⟨*from*⟩ ‘`,`’ ⟨*to*⟩ ‘`,`’ ⟨*termlist*⟩ ‘`)`’ ‘`;`’
| ‘`send`’ ⟨*label*⟩ ‘`(`’ ⟨*from*⟩ ‘`,`’ ⟨*to*⟩ ‘`,`’ ⟨*termlist*⟩ ‘`)`’ ‘`;`’
| ‘`claim`’ [⟨*label*⟩] ‘`(`’ ⟨*from*⟩ ‘`,`’ ⟨*claim*⟩ [‘`,`’ ⟨*termlist*⟩] ‘`)`’ ‘`;`’

⟨*label*⟩ ::= ‘`_`’ ⟨*term*⟩

⟨*from*⟩ ::= ⟨*id*⟩

⟨*to*⟩ ::= ⟨*id*⟩

⟨*claim*⟩ ::= ⟨*id*⟩

5.7.5 Declarations

⟨*globaldeclaration*⟩ ::= ⟨*declaration*⟩
| ‘`untrusted`’ ⟨*termlist*⟩ ‘`;`’
| ‘`usertype`’ ⟨*termlist*⟩ ‘`;`’

⟨*declaration*⟩ ::= [‘`secret`’] ‘`const`’ ⟨*termlist*⟩ [‘`:`’ ⟨*type*⟩] ‘`;`’
| [‘`secret`’] ‘`fresh`’ ⟨*termlist*⟩ [‘`:`’ ⟨*typelist*⟩] ‘`;`’
| [‘`secret`’] ‘`var`’ ⟨*termlist*⟩ [‘`:`’ ⟨*typelist*⟩] ‘`;`’
| ‘`secret`’ ⟨*termlist*⟩ [⟨*type*⟩] ‘`;`’
| ‘`inversekeys`’ ‘`(`’ ⟨*term*⟩ ‘`,`’ ⟨*term*⟩ ‘`)`’ ‘`;`’
| ‘`compromised`’ ⟨*termlist*⟩ ‘`;`’

⟨*type*⟩ ::= ⟨*id*⟩

⟨*typelist*⟩ ::= ⟨*type*⟩ { ‘`,`’ ⟨*type*⟩ }

5.7.6 Terms

⟨*term*⟩ ::= ⟨*id*⟩
| ‘`{`’ ⟨*termlist*⟩ ‘`}`’ ⟨*key*⟩
| ‘`(`’ ⟨*termlist*⟩ ‘`)`’
| ⟨*id*⟩ ‘`(`’ ⟨*termlist*⟩ ‘`)`’

⟨*key*⟩ ::= ⟨*term*⟩

⟨*termlist*⟩ ::= ⟨*term*⟩ { ‘`,`’ ⟨*term*⟩ }

6 Protocol modeling

The initial step of modeling a protocol typically takes the most time of the verification process. Most protocols are not very well documented, and because we work here with abstracted protocols, it is very easy to make a wrong abstraction in the process and miss out on a crucial feature. Once the protocol is modeled, the issue of deciding which security properties need to be included is often also unclear. Secrecy of some terms is fairly straightforward, but informal notions of authentication are potential minefields, and should be carefully examined.

Once this difficult phase is over, and we are left with a suitable abstracted protocol, the tools can be used to quickly find attacks on the protocol model. It is often easy to check whether an attack on the abstract protocol constitutes an attack on the real protocol.

6.1 Example: Needham-Schroeder Public Key

We discuss now construction of a such protocol model in stages.

In figure 4 the Needham-Schroeder Public Key protocol is shown. For simplicity, we have only displayed the claim by each role that the initiator nonce ni is secret.

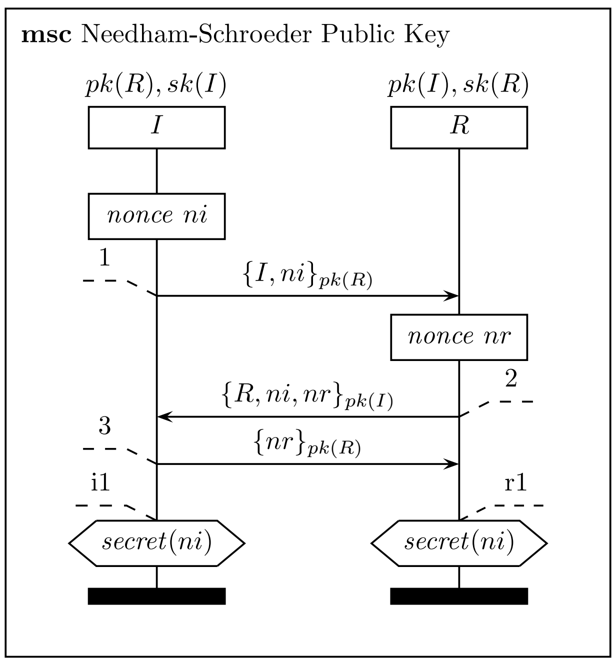

Figure 4: A message sequence chart description

We start off the protocol description by adding a multi-line comment that describes the protocol and other interesting details. Multi-line comments start with `/*` and end with `*/`.

```
/*
 * Needham-Schroeder protocol
 */
```

The protocol uses the default public/private key infrastructure: an agent `A` has a key pair `(pk(A),sk(A))`.

The protocol has two roles: the intiator role `I` and the responder role `R`. We also add a single line comment, starting with `//`.

```
// The protocol description

protocol ns3(I,R)
{
```

Scyther works with a role-based description of the protocols. We first model the initiator role. This role has two values that are local to the role: the nonce that is created by `I` and the nonce that is received. We have to declare them both.

```
  role I
  {
    fresh ni: Nonce;
    var nr: Nonce;
```

We now model the communication behaviour of the protocol. Needham-Schroeder has three messages, and the initiator role sends the first and last of these. Note the labels (e.g. `_1`) at the end of the `send` and `recv` keywords: these serve merely to retain the information of the connected arrows in the message sequence chart.

```
    send_1(I,R, {I,ni}pk(R) );
    recv_2(R,I, {ni,nr}pk(I) );
    send_3(I,R, {nr}pk(R) );
```

Finally, we add the security requirements of the protocol. Without such claims, Scyther does not know[1] what needs to be checked.

Here we have chosen to check for secrecy of the generated and received nonce, and will check for non-injective agreement and synchronisation.

```
    claim_i1(I,Secret,ni);
    claim_i2(I,Secret,nr);
    claim_i3(I,Niagree);
    claim_i4(I,Nisynch);
  }
```

This completes the specification of the initiator role.

For this simple protocol, the responder role is very similar to the initiator role[2]. In fact, there are only a few differences:

[1] If you are unsure about the claims, you can also use the `--auto-claims` switch to automatically generate these at run-time.

[2] In general, the transformation is not that simple, but for many protocols this will suffice.

1. The keywords `var` and `fresh` have swapped places: `ni` was created by `I` and a freshly generated value there, but for the role `R` it is the received value and thus a variable.

2. The keywords `send` and `recv` have swapped places.

3. The claims should have unique labels, so they have changed, and the role executing the claim is now `R` instead of `I`.

The complete role description for the responder looks like this:

```
  role R
  {
    var ni: Nonce;
    fresh nr: Nonce;

    recv_1(I,R, {I,ni}pk(R) );
    send_2(R,I, {ni,nr}pk(I) );
    recv_3(I,R, {nr}pk(R) );

    claim_r1(R,Secret,ni);
    claim_r2(R,Secret,nr);
    claim_r3(R,Niagree);
    claim_r4(R,Nisynch);
  }
}
```

The full protocol description file for the *Needham-Schroeder* protocol can be found in Appendix A.

More text will be supplied at a later stage.

7 Using the Scyther tool

The Scyther tool can be used in two main ways. First, through the graphical user interface (GUI) and second, through the command-line interface. For most users the first option is preferred.

7.1 Using the graphical user interface (GUI)

7.2 Using the command-line interface

8 Scyther output

In this section we detail the Scyther output when used through the GUI.

8.1 Results

As shown before, verifying the Needham-Schroeder public key protocol yields the following results as in Figure 8.1.

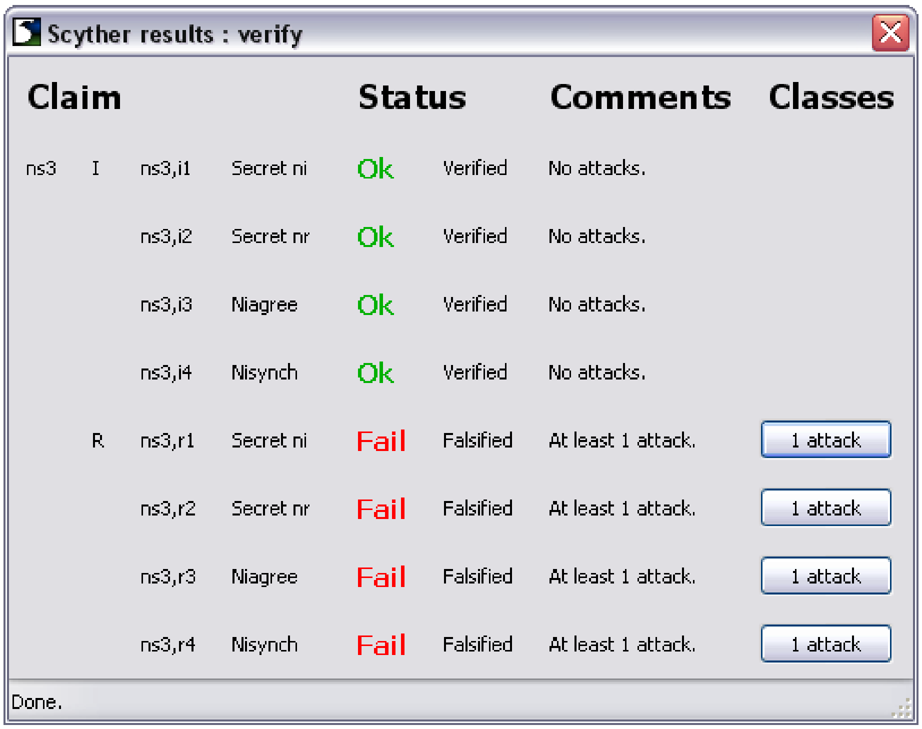

Figure 5: Scyther results for the Needham-Schroeder protocol

The interpretation is as follows: all the claims of the initiator role `ns3,I` are correct for an unbounded number of runs.

Unfortunately, all the claims of the responder role are false. Scyther reports that it found at least one attack for each of those four claims. We could choose to view these attacks: this will be shown in Section 8.3.

In the result window, Scyther will output a single line for each claim. The line is divided into several columns. The first column shows the protocol in which the claim occurs, and the second shows the role. In the third column a unique claim identifier is shown, of the form `p,l`, where `p` is the protocol and `l` is the claim label.[3]. The fourth column displays the claim type and the claim parameter.

Under the header `Status` we find two columns. The fifth column gives the actual result of the verification process: it will yield `Fail` when the claim is

[3] This includes the protocol name, which is important when doing multi-protocol analysis.

false, and `Ok` when the claim is correct. The sixth column refines the previous statement: in some cases, the Scyther verification process is not complete (which will be explored in more detail in the next section). If this column states `Verified`, then the claim is provably true. If the column states `Falsified`, then the claim is provably false. If the column is empty, then the statement of fail/ok depends on the specific bounds setting.

The seventh column, `Comments`, serves to explain the status of the results further. In particular, the column contains a single sentences. We describe the possible results below.

- `At least X attack(s)`

  Some attacks were found in the state space: however, due to the undecidability of the problem, or because of the branch and bround structure of the search, we cannot be sure that there are no other attack states.

  In the default setup, Scyther will stop the verification process after an attack is found.

- `Exactly X attack(s)`

  Within the statespace, there are exactly this many attacks, and no others.

- `At least X pattern(s)`

- `Exactly X pattern(s)`

  These correspond exactly to the previous two, but occur in case of a '`Reachable`' claim. Thus, the states that are found are not really attacks but classes of reachable states.

- `No attacks within bounds`

  No attack was found within the bounded statespace, but there can possibly be an attack outside the bounded statespace.

- `No attacks`

  No attack was found within the (bounded or unbounded) statespace, and a proof can be constructed that there is no attack even when the statespace is unbounded.

  Note that because of the nature of the algorithm, this result can even be obtained when the statespace is bounded.

8.2 Bounding the statespace

During the verification process, the Scyther tool explores a proof tree that covers all possible protocol behaviours. The default setting is to *bound* the size of this tree in some way, ensuring that the verification procedure terminates. However, importantly, even if the size of this proof tree is bounded, unbounded verification may still be achieved.

In most cases, the verification procedure will terminate and return results before ever reaching the bound. However, if the verification procedure reaches the bound, this is reported in the result window, e.g.:

```
No attack within bounds
```

This should be interpreted as: Scyther did not find any attacks, but because it reached the bound, it did not explore the full tree, and it is possible that there are still attacks on the protocol.

The default way of bounding the *maximum number of runs*, or protocol instances. This can be changed in the `Settings` tab of the main window. If the maximum number of runs is e.g. 5, and Scyther reports `No attack within bounds`, this means that there exist no attacks that involve 5 runs or less. However, there might exist attacks that involve 6 runs or more.

For some protocols, increasing the maximum number of runs can lead to complete results (i.e. finding an attack or being sure that there is no attack), but for other protocols the result will always be `No attack within bounds`.

Note that the verification time usually grows exponentially with respect to the maximum number of runs.

8.3 Attack graphs

In Figure 8.3 we show an attack window in more detail.

The basic elements are arrows and several kinds of boxes. The arrows in the graph represent ordering constraints (caused by the prefix-closedness of events in the protocol roles, or by dependencies in the intruder knowledge). The boxes represent creation of a run, communication events of a run, and claim events.

8.3.1 Runs

Each vertical axis represents a run (an instance of a protocol role). Thus, in this attack we see that there are two runs involved. Each run starts with a diamond shaped box. This represents the creation of a run, and is used to give information about the run.

For the run on the left-hand side in the attack we have this information:

```
Run #1
Agent2 in role I
I -> Agent2
R -> Agent1
```

Each run is assigned a run identifier (here 1), which is an arbitrary number that enables us to uniquely identify each run. This run executes the `R` role of the protocol. It is being executed by an agent called `Agent1`, who thinks he is talking to `Agent2`. Note that although run 2 is being executed by `Agent2`, this agent does not believe he is talking to `Agent1`.

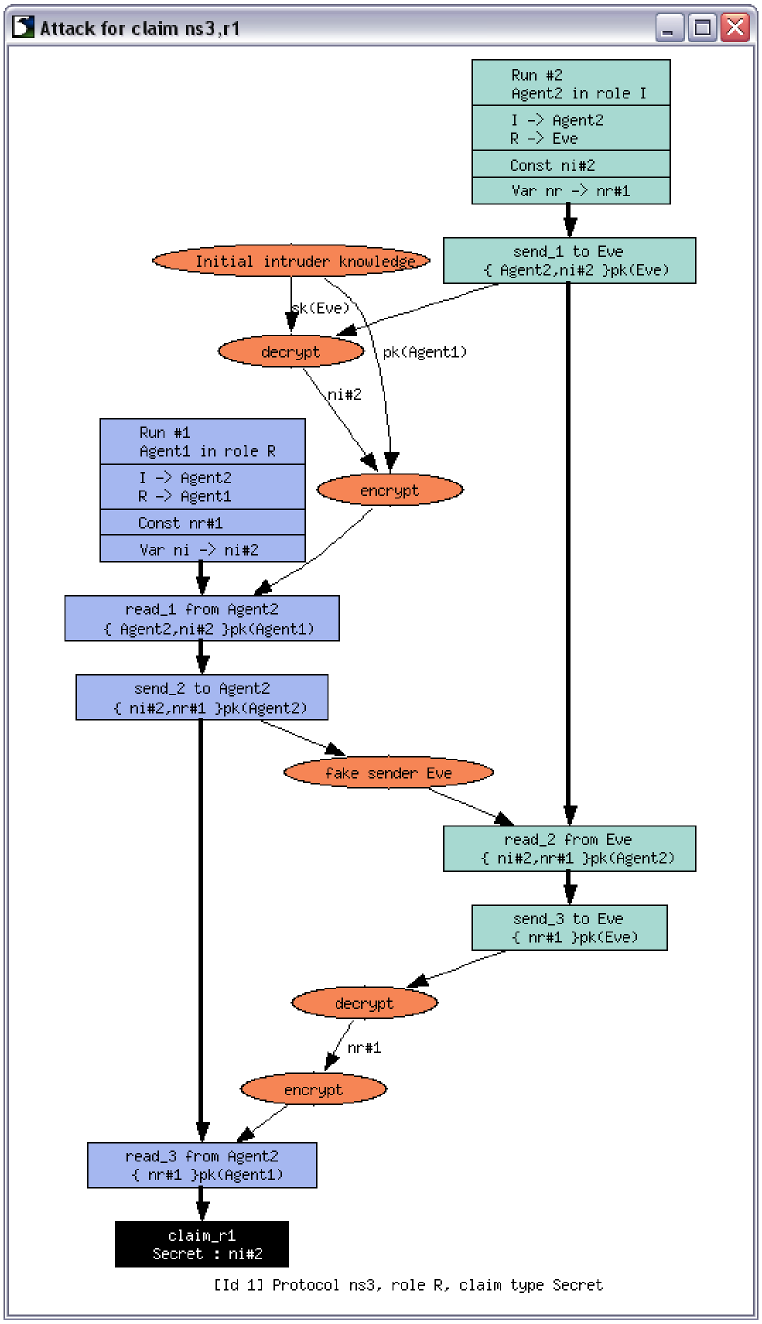

Figure 6: Scyther attack window

```
Run #2
```

```
Agent2 in role I
I -> Agent2
R -> Eve
```

In the run on the right, we see This run represents an instance of the role `I`. From the second line we can see which agent is executing the run, and who he thinks he is talking to. In this example, the run is executed by an agent called `Agent2`, who thinks the responder role is being executed by the untrusted agent `Eve`.[4]

Additionally, the run headers contain information on the freshly generated values (e.g. run 1 generates `nr#1`) and information on the instantiation of the local variables (e.g. run 1 instantiates its variable `ni` with the nonce `ni#2` or run 2.

8.3.2 Communication events

Send events denote the sending of a message. The first send occurs in this attack is the first send event of run 2.

```
send_1(Eve, { Agent#0, ni#2 }pk(Eve) )
```

Every time a message is sent, it is effectively given to the intruder. In this case, because the intruder knows the secret key `sk(Eve)` of the agent `Eve`, he can decrypt the message and learns the value of the nonce `ni#2`.

recv events denote the reading of a message. The first recv that can occur in this attack is the first recv event of run 0.

```
recv_1(Agent#0, { Agent#0, ni#2 }pk(Agent#1) )
```

This tells us that the agent executing this run, `Agent#1`, reads a message that is apparently coming from `Agent#1`. The message that is received is { `Agent#0, ni#2` }`pk(Agent#1)` : the name of the agent he thinks he is communicating with and the nonce `ni#2`, encrypted with his public key.

The incoming arrow does not indicate a direct sending of the message. Rather, it denotes an ordering constraint: this message can only be received *after* something else has happened. In this case, we see that the message can only be received after run 2 sends his initial message. The reason for this is the nonce `ni#2`: the intruder cannot predict this nonce, and thus has to wait until run 2 has generated it.

In the graph the connecting arrow is red and has a label "construct" with it: this is caused by the fact that the message sent does not correspond to the message that is received. We know the intruder can only construct the message to be received after the sent message, and thus it must be the case that he uses

[4]Because this agent is talking to the untrusted agent, of course all information is leaked, and no guarantees can be given.

information from the sent message to construct the message that is received. Other possibilities include a green and a yellow arrow. A yellow arrow indicates that a message was sent, and received in exactly the same form: however, the agents disagree about who was sending a message to whom. It is therefore labeled with "redirect" because the intruder must have redirected the message. A green arrow (not in the picture) indicating that a message is received exactly the same as it was sent, representing a normal message communication between two agents.

Note that a `recv` event without an incoming arrow denotes that a term is received that can be generated from the initial knowledge of the intruder. There is no such event in the example, but this can occur often. For example, if a role reads a plain message containing only an agent name, the intruder can generate the term from his initial knowledge.

8.3.3 Claims

9 Advanced topics

9.1 Modeling more than one asymmetric key pair

Asymmetric keys are typically modeled as two functions: one function that maps the agents to their public keys, and another function that maps agents to their secret keys.

By default, each agent x has a public/private key pair $(\mathtt{pk}(\mathtt{x}), \mathtt{sk}(\mathtt{x}))$.

To model other asymmetric keys, we first define the two functions, which are for example named `pk2` for the public key function, and `sk2` for the secret key function.

```
const pk2: Function;
secret sk2: Function;
```

We also declare that these functions represent asymmetric key pairs:

```
inversekeys (pk2,sk2);
```

If defined in this way, a term encrypted with `pk2(x)` can only be decrypted with `sk2(x)` and vice versa.

References

[1] C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols.* Ph.D. dissertation, Eindhoven University of Technology, 2006.

[2] C.J.F. Cremers and S. Mauw. Operational semantics of security protocols. In S. Leue and T. Systä, editors, *Scenarios: Models, Transformations and Tools, International Workshop, Dagstuhl Castle, Germany, September 7-12, 2003, Revised Selected Papers*, volume 3466 of *LNCS*. Springer, 2005.

[3] C.J.F. Cremers, S. Mauw, and E.P. de Vink. Formal methods for security protocols: Three examples of the black-box approach. *NVTI newsletter*, 7:21–32, 2003. Newsletter of the Dutch Association for Theoretical Computing Scientists.

[4] C.J.F. Cremers, S. Mauw, and E.P. de Vink. Defining authentication in a trace model. In T. Dimitrakos and F. Martinelli, editors, *Proc. FAST 2003*, pages 131–145, Pisa, 2003. IITT-CNR technical report.

[5] G. Lowe. A hierarchy of authentication specifications. In *Proc. 10th IEEE Computer Security Foundations Workshop (CSFW)*, pages 31–44. IEEE, 1997.

A Full specification for Needham-Schroeder public key

```
/*
 * Needham-Schroeder protocol
 */


// The protocol description

protocol ns3(I,R)
{
  role I
  {
    fresh ni: Nonce;
    var nr: Nonce;

    send_1(I,R, {I,ni}pk(R) );
    recv_2(R,I, {ni,nr}pk(I) );
    claim(I,Running,R,ni,nr);
    send_3(I,R, {nr}pk(R) );

    claim_i1(I,Secret,ni);
    claim_i2(I,Secret,nr);
    claim_i3(I,Alive);
    claim_i4(I,Weakagree);
    claim_i5(I,Commit,R,ni,nr);
    claim_i6(I,Niagree);
    claim_i7(I,Nisynch);
  }

  role R
```

```
  {
    var ni: Nonce;
    fresh nr: Nonce;

    recv_1(I,R, {I,ni}pk(R) );
    claim(R,Running,I,ni,nr);
    send_2(R,I, {ni,nr}pk(I) );
    recv_3(I,R, {nr}pk(R) );

    claim_r1(R,Secret,ni);
    claim_r2(R,Secret,nr);
    claim_r3(R,Alive);
    claim_r4(R,Weakagree);
    claim_r5(R,Commit,I,ni,nr);
    claim_r6(R,Niagree);
    claim_r7(R,Nisynch);
  }
}
```