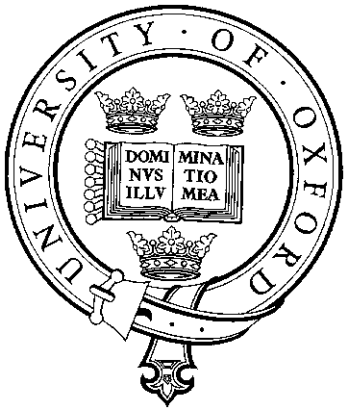
COSP-J: A Compiler for Security Protocols

Xavier Didelot
Lady Margaret Hall



Oxford University Computing Laboratory

Thesis Submitted in partial fulfillment of the requirements for the Degree of Master of Science in Computation

University of Oxford

Abstract

A security protocol consists of an exchange of messages between two or more agents, with goals such as establishing a cryptographic key, or authenticating the identities of the agents. These protocols are designed to operate in particularly hostile environments, where an intruder may be trying to attack the protocol, by intercepting some messages or creating new messages.

Typically, the design of a security protocols can be formally described in half a page using an abstract notation. However, the implementation tends to be much longer and may introduce new sources of attack if not done properly. The main goal of this project is to build a compiler that compiles the design of a given security protocol into an implementation in Java as secure as possible.

Acknowledgments

Firstly I would like to thank Dr. Gavin Lowe, my thesis supervisor, for proposing this very interesting subject and providing me all the help I needed to complete this work.

I am also grateful to Dr. Philippa Broadfoot for her guidance and her interest in my work.

Thank you to Dr. Christie Bolton for giving me the opportunity to give a talk to the concurrency group that was very fruitful. Thanks to everybody from the security and concurrency groups for some very interesting discussions that made me understand a lot of things.

I would like to give thanks to my french Engineering School, l'Institut d'Informatique d'Entreprise, for giving me the opportunity to follow this Master Of Science in the University of Oxford.

Finally I am greatly thankful to my parents for their love and support.

Contents

List of Figures

Chapter 1

Introduction

According to Gollman [Gol99], "Computer security deals with the prevention and detection of unauthorized actions by users of a computer system". A computer system can be a set of computers connected in some way to each other across a network. An important area of research in computer science is concerned with the security of communications over a public network such as the Internet. Its applications are extremely varied, for example the confidentiality of e-mails sent between companies, the growing use of e-commerce, the authentication of an entity before sending important data to it, or the anonymity of the users of a website.

In spite of this, the usual computer networks by themselves do not provide any form of security. When an agent sends a message to another agent, except if the two agents are linked directly, it will be transmitted from one host to another until it reaches its intended destination. However, nothing prevents one of the intermediary from destroying the message or modifying it. So how can we communicate securely over a network that is not secure? Security protocols are meant to remedy this issue.

1.1 Introducing security protocols

A security protocol is a sequence of messages exchanged between entities making use of cryptography, in order to establish security properties; for example, authentication (i.e. verification of identity) of agents, establishment of secret keys and ensuring the integrity of data received. These services have to be provided even in a highly hostile environment in which intruders have capabilities such as intercepting messages, inserting new messages, modifying messages sent, spoofing the identity of another agent and running the protocol with different agents at the same time. In order to do so, security protocols make use of cryptography. Cryptography is the science of rendering plain information unintelligible (this is called encryption) and restoring encrypted information to intelligible form (this is called de-

cryption). The design of such protocols is a difficult task as the following example will show.

The following security protocol is called the Needham-Schroeder Public Key protocol [NS78]. It was thought to be secure for many years:

$$\begin{aligned} &\text{Message 1.} \quad a \rightarrow b \ : \ \{a, na\}_{PK(b)} \\ &\text{Message 2.} \quad b \rightarrow a \ : \ \{na, nb\}_{PK(a)} \\ &\text{Message 3.} \quad a \rightarrow b \ : \ \{nb\}_{PK(b)} \end{aligned}$$

where a and b are agents, $PK(x)$ is the public key of agent x, na is a fresh nonce created by a and nb is a fresh nonce created by b. A nonce is a large random number that must be fresh for each run of the protocol.

$\{data\}_k$ denotes the value $data$ encrypted under the key k and m, n denotes the text m followed by the text n.

In this security protocol, two agents a and b communicate with the aim of mutually authenticating one another. This protocol was thought to satisfy the following security properties:

1. a must be authenticated by b;
2. b must be authenticated by a;
3. the value of the nonces na and nb must be known by agents a and b but nobody else.

The assumptions of knowledge at the start of a protocol run are as follows:

- each participating agent x knows its own secret key $SK(x)$, which remains secret;
- all public keys are known by all participating agents.

This protocol is composed of 3 messages. Firstly a creates a fresh nonce na and sends it along with its own identity to b, encrypting the message with b's public key so that only b will be able to decrypt it. b then sends to a the nonce na along with a newly created nonce nb, encrypting this message with a's public key so that only a should be able to decrypt it. When a reads na, he knows he was talking to b because nobody else could have been able to decrypt message 1. a finally sends nb encrypted with b's public key to b. When b reads nb, he thinks he was talking to a because nobody else should be able to decrypt message 2. After this, a and b apparently know they have been talking with each other, agree on the values of na and nb, and nobody else know those values.

After 17 years the following attack was discovered by Lowe [Low96a]:

$$\begin{array}{lrcl}
\text{Message } \alpha 1. & A \rightarrow Y & : & \{A, na\}_{PK(Y)} \\
\text{Message } \beta 1. & Y_A \rightarrow B & : & \{A, na\}_{PK(B)} \\
\text{Message } \beta 2. & B \rightarrow Y_A & : & \{na, nb\}_{PK(A)} \\
\text{Message } \alpha 2. & Y \rightarrow A & : & \{na, nb\}_{PK(A)} \\
\text{Message } \alpha 3. & A \rightarrow Y & : & \{nb\}_{PK(Y)} \\
\text{Message } \beta 3. & Y_A \rightarrow B & : & \{nb\}_{PK(B)}
\end{array}$$

where A and B are playing roles of a and b respectively, and I_X represents the intruder impersonating agent X.

In this attack, A runs the protocol with some agent Y who will use this opportunity to spoof A's identity in a run of the protocol with B. Y starts a second run of the protocol (run β) and sends to B the nonce na that he received from message $\alpha 1$, pretending to be A. When B sends him the values of na and nb encrypted with the public key of A, the intruder simply forwards this message to A who decrypts it for him and sends him the value of nb encrypted with Y's public key. Y now knows nb and is able to send message $\beta 3$ to B.

At the end of run β, B thinks he has completed a run of the protocol with A, whereas he was actually talking to Y impersonating A. This is a failure of authentication.

Lowe [Low96a] proposed the following correction to the NSPK protocol:

$$\begin{array}{lrcl}
\text{Message 1.} & a \rightarrow b & : & \{a, na\}_{PK(b)} \\
\text{Message 2.} & b \rightarrow a & : & \{b, na, nb\}_{PK(a)} \\
\text{Message 3.} & a \rightarrow b & : & \{nb\}_{PK(b)}
\end{array}$$

The former attack does not work anymore as an intruder Y will not be able to replay message $\beta 2$ to A: A expects to read the identity Y in the first field of the message once decrypted and will read the identity of B instead.

Abadi and Needham [AN96] have defined a number of principles to follow in order to avoid such attacks. Here the Needham-Schroeder Public Key protocol does not respect the third principle that reads: "If the identity of a principal is essential to the meaning of a message, it is prudent to mention the principal's name explicitly in the message".

But how can we be sure that there is not any other subtle attacks? There have been a number of formal approaches developed over the years to verify the correctness of security protocols. One of the most successful is the use of Casper [Low97] that will be presented in the next chapter.

The usual way to design a security protocol is therefore:

1. Establish how the protocol works, the requirements it has and the properties it is supposed to provide;

2. Fix any obvious issue that may be discovered just by thinking about how the protocol works;
3. Use a verification technique to search for other attacks and fix them until no problem remains;
4. Eventually implement the protocol in a given computing language.

1.2 The problem this thesis addresses

The aim of this project is to build a tool that compiles the design of a security protocol into a corresponding Java implementation.

The protocol compiler that we develop is named COSP-J and is meant to be used once a protocol was checked with Casper. This means that the Casper and COSP-J input files must be as similar as possible.

The creation of such implementations can introduce new ways of attacking the protocol. For example, a worm called "Slapper" infected 12,000 servers running the SSL module of Apache `mod_ssl`. The worm does not exploit an attack against the SSL protocol itself, but is a buffer overflow attack that works only against this specific implementation.

A buffer overflow occurs when there is more data stored in a buffer than it is intended to hold. The extra information can overflow into adjacent buffers, overwriting the valid data they hold. Extra data may contain instructions that will be executed by the attacked computer so that the attacker will really take control of its target.

In 2001, a buffer overflow attack was discovered against implementations of the SSH protocol (version 1) from OpenSSH and SSH Communications Security [Zal01]. Once again, it is not an attack against the protocol itself, but against some of its implementations. In fact, buffer overflow is just an example of programming error that can lead to an attack.

Therefore it would be interesting to have a tool that could generate the implementation automatically to remove the risk of error in this step, and also to make implementation faster and easier.

An additional aim is to prevent some attacks in the implementations generated by COSP-J, that Casper is unable to detect (cf. Chapter 7 and 8).

1.3 Thesis overview

In Chapter 2, we introduce the background material that is used in this thesis. In Chapter 3, we describe how to write a basic input file for our compiler. In Chapter 4, we describe how to write a clean implementation of a security protocol by writing as few lines as possible and trying to make the code as readable as possible. This is achieved by using external libraries

common for all protocols (some of them exist and some of them have to be written). In Chapter 5, we highlight the main difficulties encountered when compiling an input script into an implementation and how to deal with these issues. Chapter 6, introduces additional features that are useful for certain protocols, such as Vernam Encryption and hash functions, and how they are captured in an input file so as to be able to implement a very wide range of security protocols. In Chapter 7, we describe Type Flaw Attacks which are difficult to detect with Casper; we show how to prevent these attacks formally and how these preventative measures are implemented in our proposed compiler. Chapter 8 deals with Multi-protocols Attack and how they are prevented in COSP-J. Chapter 9, shows how to use our proposed compiler to implement a complex e-commerce security protocol. Finally, Chapter 10 presents the conclusions of our work, together with a discussion of future work.

Chapter 2

Introducing Casper

Casper [Low97] is a compiler that takes a high level description of a security protocol and generates a corresponding model in the CSP [Hoa85] language.

2.1 Introducing CSP and FDR

CSP (Communicating Sequential Processes) is a notation for describing a system made of processes communicating with each other. CSP lets its users describe a system and also write down some properties on the system. FDR [Ros94] is a model-checker that can check whether these properties are satisfied or not (and give counter-example if a property is not satisfied). It is possible to build a CSP model describing a distributed system with processes running a security protocol and other processes that try to interfere to create an attack. However, building these models by hand takes a lot of time and must be done very carefully to prevent any mistakes.

Casper is able to automatically compile a high level description of a protocol into such a CSP script. Then we can run FDR on this CSP script and automatically check whether the properties that the protocol is intended to achieve are satisfied no matter what the intruders may try. If a property is false, CSP returns a counter-example that can be interpreted by Casper as an attack against the protocol (Figure 2.1).

The use of Casper and CSP has been very successful over the past few years and has discovered many attacks against protocols that were thought to be correct. The next paragraph will describe a Casper input file.

2.2 A Casper input file

In this section, we describe the Casper script for the Needham-Schroeder Public Key (NSPK) protocol.

All Casper input scripts contain principally two kinds of information: information on the protocol we are analyzing and information describing

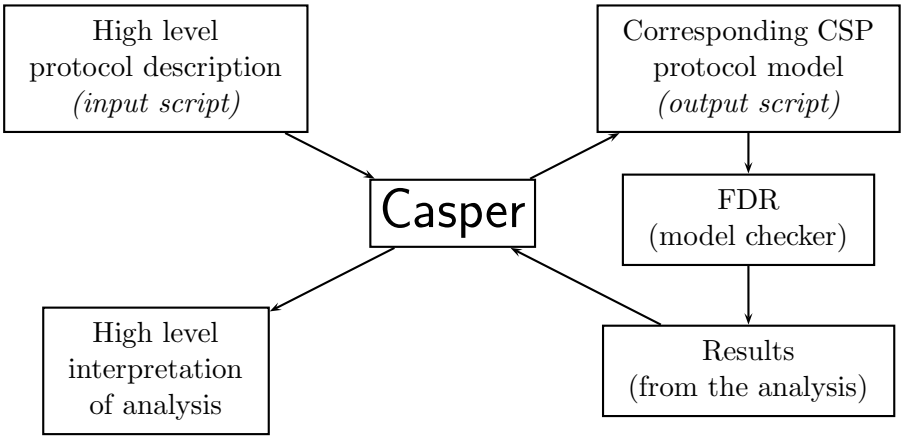


Figure 2.1: The use of Casper

the actual system on which the check will be based.

The sections of a Casper input file that deal with the protocol itself are:

The "Free variables" section

```
#Free variables
A, B : Agent
na, nb : Nonce
PK : Agent -> PublicKey
SK : Agent -> SecretKey
InverseKeys = (PK, SK)
```

This section declares all the variables used in the protocol and their corresponding type. It also specifies the relationship between encryption and decryption keys thanks to the `InverseKeys` keyword.

The "Processes" section

```
#Processes
INITIATOR(A,na) knows PK, SK(A)
RESPONDER(B,nb) knows PK, SK(B)
```

This section declares the agent processes that play a role in the protocol and their initial knowledge. The first parameter of each process is their identity.

The "Protocol description" section

```
#Protocol description
```

```
0.    -> A : B
1.  A -> B : {na, A}{PK(B)}
2.  B -> A : {na, nb}{PK(A)}
3.  A -> B : {nb}{PK(B)}
```

This section describes the sequence of messages that are exchanged during the run of the protocol. For each message, it gives the name of the sender, the name of the receiver and the content of the message.

The "Specification" section

```
#Specification
Secret(A, na, [B])
Secret(B, nb, [A])
Agreement(A,B,[na,nb])
Agreement(B,A,[na,nb])
```

This section is used to specify the requirements of the protocol to be verified. For example, at the end of the Needham-Schroeder Public Key protocol na and nb are secrets shared by agents a and b only, and a and b are authenticated to each other.

The sections of a Casper input file that deal with the actual system to check are:

The "Actual variables" section

```
#Actual variables
Alice, Bob, Mallory : Agent
Na, Nb, Nm : Nonce
```

This section declares all the variables used in the actual system to be verified and their corresponding type in a similar way as for the free variables.

The "System" section

```
#System
INITIATOR(Alice, Na)
RESPONDER(Bob, Nb)
```

This section defines the agents participating in the actual system to be checked.

The "Intruder Information" section

```
#Intruder Information
Intruder = Mallory
IntruderKnowledge = {Alice, Bob, Mallory, Nm, PK, SK(Mallory)}
```

This section describes the intruder's identity in the system to be verified and its initial knowledge.

The "Functions" section

```
#Functions
symbolic PK, SK
```

This section defines how functions used by the agents in the protocol description are performed. The keyword `symbolic` means that Casper will produce its own values representing the results of the applications of the functions.

As Casper and COSP-J are meant to be used together, COSP-J was designed so that their input files have a lot in common. The next chapter will first introduce COSP-J and then present the modifications between a Casper script and a COSP-J script.

Chapter 3

Introducing COSP-J

This chapter introduces the compiler of security protocols COSP-J and shows how to write a COSP-J input file.

3.1 Presentation of COSP-J

There presently exists no form of mechanical assistance to facilitate the implementation of security protocols once they have been analyzed with Casper. Even if the protocol is valid, there can be some errors in its implementation that can be exploited by intruders. The aim of this project is to build a tool called COSP-J that compiles the design of a security protocol into an implementation written in Java (Figure 3.1).

COSP-J stands for Compiler Of Security Protocols into Java. It is meant to be used in addition to Casper in order to first analyze a protocol and then compile it.

It is important to keep in mind that Casper and COSP-J, despite their similarities, act at two different levels of abstraction. Casper considers a simple network of agents that communicate together (Figure 3.2). Casper abstracts away from what these agents physically are and how they communicate together. If we consider that these agents are human beings, then we do not take into account the physical network that they use, on which computers they are logged and how the messages are transmitted from one computer to another.

COSP-J acts at a much more concrete level of abstraction. COSP-J considers a network of computers that communicate with each other and on which some users can be logged (Figure 3.3). This means that we have to make a clear distinction between computers and users. There is no way to map a computer to a user, because a user can log on to different computers at different times, a user can be logged on to more than one computer at a time, and more than one user can be logged at a computer at a time. In the rest of this thesis, we will use the word host or agent to designate a computer

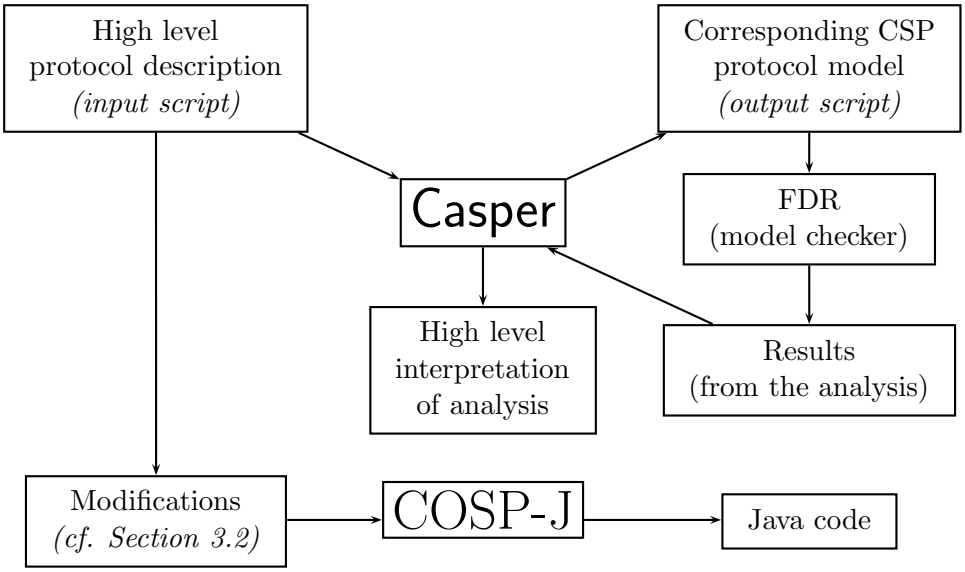


Figure 3.1: The use of Casper and COSP-J together

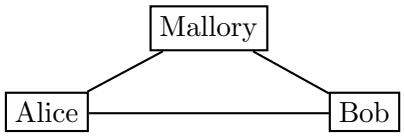


Figure 3.2: Illustration of a network for Casper

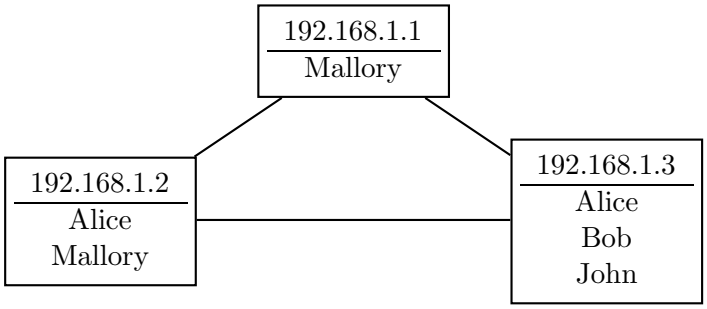


Figure 3.3: Illustration of a network for COSP-J

on which a given user is logged. The identity of the host is the name of its user and the address of the host is the IP address of the computer.

3.2 Description of an input file

The structure of the input files of COSP-J is adapted from the general structure of the input files of Casper. In this section, we define the sections that compose a COSP-J input file and their meaning. Throughout the section, we will refer to the Needham-Schroeder Public Key protocol as a running example: we will look at the sections of a Casper input script that we will need to keep and see what modifications have to be made to get a COSP-J input script.

3.2.1 "Protocol description" section

The main subsection of a Casper input file, under the heading "`#Protocol description`" is the list of messages to be exchanged between the participating agents. For the Needham-Schroeder Public Key protocol, described in Chapter 1, this part of a Casper input file would be:

```
#Protocol description
0.       -> a : b
1.     a -> b : {a,na}{PK(b)}
2.     b -> a : {na,nb}{PK(a)}
3.     a -> b : {nb}{PK(b)}
```

Each step of the protocol is defined in a language very close to the usual notation used above. `{m}{k}` represents the message m encrypted with the key k. The only difference with the usual notation is the message 0. The message 0 means that a receives the identity of the agent b from the environment. This message may come from a user of agent a, who tells agent a that he wants him to run the protocol with agent b.

The following two modifications are made to create a correct COSP-J input file from this:

- We do not use the environmental message 0 anymore in COSP-J: the fact that a initially knows the identity of b with whom he has to run the protocol will be expressed in the section "`#Processes`" described below. The difference between an environmental message and an initial knowledge is important for the analysis of a protocol with Casper but does not change anything for an implementation;

- We need a way to express what the results are of the protocol run once it is finished. In the case of the Needham-Schroeder Public Key Protocol, the agents share the secret values of na and nb and the agent

b is sure of a's identity. If the protocol ends successfully, we want the agent a to return the values of na and nb and the agent b to return the values of a, na and nb. The method in the implementation returns these values to the calling code. These values may be useful after the end of the protocol, for example if this implementation of the NSPK is just a part of a big network application.

Therefore this part of the script becomes:

```
#Protocol description
1.     a -> b : {a,na}{PK(b)}
2.     b -> a : {na,nb}{PK(a)}
3.     a -> b : {nb}{PK(b)}
4.     a ->   : na,nb
5.     b ->   : a,na,nb
```

There are some cases where the protocol description needs further changes. For example, let us consider the Yahalom Protocol [BAN89], slightly modified by Lowe and Hui [HL99] so that the message 3 is split into messages 3a and 3b:

$$
\begin{array}{lll}
\text{Message 1.} & a \rightarrow b & : a, na \\
\text{Message 2.} & b \rightarrow s & : b, \{a, na, nb\}_{ServerKey(b)} \\
\text{Message 3}a. & s \rightarrow a & : \{b, k_{ab}, na, nb\}_{ServerKey(a)} \\
\text{Message 3}b. & s \rightarrow b & : \{a, k_{ab}\}_{ServerKey(b)} \\
\text{Message 4.} & a \rightarrow b & : \{nb\}_{k_{ab}}
\end{array}
$$

where a and b are agents, s is a trusted server, $ServerKey(x)$ is a secret key shared by s and x, k_{ab} is a fresh key created by s, na is a fresh nonce created by a and nb is a fresh nonce created by b.

This protocol establishes a shared secret key k_{ab} between agents a and b. It also mutually authenticates a and b. The protocol description section in a Casper input file would be:

```
#Protocol description
1.  a -> b : a, na
2.  b -> s : {a, na, nb}{ServerKey(b)}
3a. s -> a : {b, kab, na, nb}{ServerKey(a)}
3b. s -> b : {a, kab}{ServerKey(b)}
4.  a -> b : {nb}{kab}
```

But this description does not provide enough information for our new compiler for two reasons.

Firstly, if we try the protocol description above in COSP-J, an error will be raised explaining that agent s is unable to decrypt message 2. To decrypt

this message, s needs to know the value of $ServerKey(b)$, but s does not know the identity of b. In Casper it is assumed that when a sends a message to b, b automatically knows who a is, but this is not the case in a real life implementation; there is no way to map a host to the identity of its user. This means that when a host receives a message from another host, he only knows its IP address (in the case of an Internet network). So we need to add the identity of b in message 2; that is to say, the name used for b in the KeyStore of s. We add it in an unencrypted way, because Casper does not make a distinction between a computer and its user, so it is logical to send the identity of the user in the same way as the identity of the computer (which is sent unencrypted by the IP protocol for example=.

Secondly, an error will be raised explaining that agent s is unable to send message 3a, because he does not know who to send the message to. In message 2, b sends to s the identity of a, but there is no way to map an identity to a real host. Concretely, s knows the name of a but not its IP address (in the case of an Internet network). So we need to add the address of a in message 2. In order to do this, we use the keyword *addr* which represents the address of an agent. We add it inside of the encrypted part of the message, because Casper does not make a distinction between a computer and its user, so it is logical to send the identity of the user in the same way as the identity of the computer (which is the first field of the encrypted part).

Once we take these modifications into account, we get the correct protocol description section for COSP-J:

```
#Protocol description
1.  a -> b : a,na
2.  b -> s : b, {a, addr(a), na, nb}{ServerKey(b)}
3a. s -> a : {b, kab, na, nb}{ServerKey(a)}
3b. s -> b : {a, kab}{ServerKey(b)}
4.  a -> b : {nb}{kab}
5.  a ->   : b,kab
6.  b ->   : a,kab
```

3.2.2 "Free variables" section

The second section, under the heading "`#Free variables`" describes the types of the variables and functions that are used in the protocol definition. For the NSPK protocol, this section of an input file of Casper would be:

```
#Free variables
a, b : Agent
na,nb : Nonce
PK : Agent -> PublicKey
SK : Agent -> SecretKey
```

```
InverseKey : (PK, SK)
```

The second and third lines declare the types of the free variables. The names for the types do not really matter, as Casper does not know what a Nonce is. However, standard names are generally used. But COSP-J needs to recognize the keywords for the types and know what a Nonce is, for example.
The 4th and 5th lines declare the types of the functions PK and SK which take the identity of an agent and respectively return a PublicKey and a SecretKey.
The 6th line declares that $PK(x)$ and $SK(x)$ are inversed for every host x, so that anything encrypted with $PK(x)$ can be decrypted with $SK(x)$ and vice-versa.
For COSP-J we need to be more specific about the type of the function PK and SK and replace `PublicKey` with `RSAPublicKey` and also replace `SecretKey` with `RSASecretKey` if we want to use RSA encryption [RSA78]. Therefore this subsection simply becomes:

```
#Free variables
a, b : Agent
na,nb : Nonce
PK : Agent -> RSAPublicKey
SK : Agent -> RSASecretKey
InverseKey : (PK, SK)
```

3.2.3 "Processes" section

The next section, under the heading "`#Processes`", gives some information about the agents running the protocol and their initial knowledge. Here is this subsection of an input file of Casper for our example protocol:

```
#Processes
INITIATOR(a,na) knows PK,SK(a)
RESPONDER(b,nb) knows PK,SK(b)
```

The first line means that an agent playing role INITIATOR will be instantiated with identity a and nonce na, and initially knows PK (i.e. the public keys of all agents) and $SK(a)$ (i.e. his own secret key).

The second line means that an agent playing role RESPONDER will be instantiated with identity b and nonce nb, and initially knows PK (i.e. the public keys of all agents) and $SK(b)$ (i.e. his own secret key).

This section will have to be modified a bit for our compiler. Firstly the $INITIATOR$ has to know the name of b with whom it wants to run the protocol, because we do not use the message 0 anymore (cf. the paragraph on the protocol description section). Secondly we have to make it clear which values are fed to the agents and which values are generated during

the run of the protocol. In order to do so, we use another Casper notation and this section will become:

```
#Processes
INITIATOR(a,b) knows PK,SK(a) generates na
RESPONDER(b) knows PK,SK(b) generates nb
```

3.2.4 "Functions" section

A Casper input file includes a subsection called "`#Functions`" which defines how the functions used in the protocol must be created in CSP. For the NSPK protocol, this section is as follows:

```
#Functions
symbolic PK, SK
```

This means that the functions PK and SK are not really defined but will be simulated by the Casper code. Our compiler will need to know what the functions PK and SK really are and where to find the secret key or the public key of a given agent. The easiest way to store public and secret keys in Java is to use a KeyStore. See Chapter 5 of [Knu98] for more details on the class `java.security.keystore`. Therefore this subsection will become:

```
#Functions
PK = myKeyStore.PK
SK = myKeyStore.SK
```

This simply means that each time we need to apply the function PK or SK, we need to get the public key or the secret key from a KeyStore named myKeyStore.

This object is an external object: it is not part of the security protocol but can be specified when we run the protocol. In order to define external objects, we need an additional section described in the next paragraph.

3.2.5 "External" section

This section, that does not exist in Casper, describes the variables that are used in the input file and that will be provided at runtime. For example, the KeyStore used in the Needham-Schroeder Public Key protocol is not the same for every run of the protocol: it will be an argument of the protocol when we want to run it. Therefore we only need to tell the compiler that the variable $myKeyStore$ is of type $KeyStore$ and that it is externally defined:

```
#External
myKeyStore : KeyStore
```

3.3 Case studies

3.3.1 The Needham-Schroeder Public Key Protocol

We have built the complete input script of the Needham-Schroeder Public Key protocol. Below is the complete script that we built in this chapter:

```
#Protocol description
1.     a -> b : {a,na}{PK(b)}
2.     b -> a : {na,nb}{PK(a)}
3.     a -> b : {nb}{PK(b)}
4.     a ->   : b,na,nb
5.     b ->   : a,na,nb

#Free variables
a, b : Agent
na,nb : Nonce
PK : Agent -> RSAPublicKey
SK : Agent -> RSASecretKey
InverseKeys = (PK, SK)

#External
myKeyStore : KeyStore

#Functions
PK = myKeyStore.PK
SK = myKeyStore.SK

#Processes
INITIATOR(a,b) knows PK,SK(a) generates na
RESPONDER(b) knows PK,SK(b) generates nb
```

3.3.2 The Yahalom Protocol

In the "Protocol description" paragraph, we used the Yahalom protocol to illustrate the modification that have to be made on this section between a Casper input script and a COSP-J input script. Here is the complete input script for this protocol:

```
#Free variables
a, b, s : Agent
na, nb : Nonce
kab : DESKey
ServerKey : Agent -> DESKey
InverseKeys = (kab, kab), (ServerKey, ServerKey)
```

```
#External
sk : File

#Processes
INITYAHA(a,b) knows ServerKey(a) generates na
RESPYAHA(b,s) knows ServerKey(b) generates nb
SERVYAHA(s) knows ServerKey(a),ServerKey(b) generates kab

#Protocol description
1.  a -> b : a,na
2.  b -> s : b, {a, addr(a), na, nb}{ServerKey(b)}
3a. s -> a : {b, kab, na, nb}{ServerKey(a)}
3b. s -> b : {a, kab}{ServerKey(b)}
4.  a -> b : {nb}{kab}
5.  a ->   : b,kab
6.  b ->   : a,kab
#Functions
ServerKey = sk
```

These two case studies do not illustrate all the features that can be used in an input file of our compiler. Additional features will be introduced in Chapter 6 enabling a broader spectrum of security protocols to be implemented. But before having a more in depth vision of the input script, we need to have a look at the output code that is generated by COSP-J.

Chapter 4

The output code

4.1 Java as the output language

The output code will be a script written in Java. The choice of Java as a target language is mainly due to the fact that Java provide extensive support for cryptography in the sets of classes called Java Cryptography Architecture (JCA) and Java Cryptography Extension (JCE). These libraries are fully described in [Knu98].

Another reason for choosing Java is that it is an Object Oriented Language. We want our output code to be highly readable and as short as possible. Therefore we will make the code as reusable as possible through the mechanisms of Object Oriented Programming, for example inheritance and overloading. For a complete overview of these mechanisms and their use in Java, see [Bar00].

Finally, using Java automatically eliminates many implementation errors, such as buffer overflows, dangling pointers and memory leaks, because Java is a type-safe language with automatic memory management. Such implementation errors occur very often when using an unsafe language such as C or C++. For instance, from the top ten CERT/CC notes (as of August 2003) with highest vulnerability potential, seven are buffer overflows or integer overflows.

4.2 Structure of the output code

The structure of the output code is the list of actions that each agent has to perform concerning each message of the protocol. Three cases can occur for each agent concerning a given message.

Case 1 If the agent is neither sending nor receiving the message, it does not perform any action.

Case 2 If the agent is sending a message, it first builds the message and then sends it. Each message is composed of fields. A message is built by building all the fields and then concatenating them. A field can be an atomic value (for example, the identity of an agent or the value of a nonce) or an encrypted message. In the second case, the agent has to build the sub-message first and then encrypt it.

Case 3 If the agent is receiving a message, it decomposes and decrypts it, stores the values that were unknown and checks if the values that were already known are correct.

For example, here is the structure of the output code for the Needham-Shroeder Public Key Protocol described in Chapter 1:

For agent A

1. create a new nonce na and remember its value;
2. create a message containing A and na;
3. encrypt it with $PK(B)$;
4. send it to B;
5. wait for a message;
6. try to decrypt it with $SK(A)$, if impossible throw an exception message;
7. check that na is right, if not throw an exception;
8. remember the value of nb;
9. create a new message containing nb;
10. encrypt it with $PK(B)$;
11. send it to B;
12. return the values of na and nb.

For agent B

1. wait for a message;
2. try to decrypt it with $SK(B)$, if impossible throw an exception message;
3. remember the values of A and na;
4. create a new nonce nb and remember its value;
5. create a message containing na and nb;
6. encrypt it with $PK(A)$;
7. send it to A;
8. wait for a message;
9. try to decrypt it with $SK(B)$, if impossible throw an exception message;
10. check that nb is right, if not throw an exception;
11. return the values of na and nb and the identity of agent a.

The structure of the output code has to be deduced from the input code. As we want the output code to be easy to read and understand, we need each action to be expressed in as few code as possible (preferably just a line). This means that we need to identify the actions that are performed and design an Object Oriented Model that provides them in the best way possible. For example, we notice that some actions happen quite often, such as encryption, decryption, creation of a nonce, reception of a message, dispatch of a message, creation of a message from its fields and so on.

The next section will describe an (much simplified) Object Oriented Model to provide tools that let us perform all this actions easily.

4.3 Object Oriented Model for the implementation of security protocols

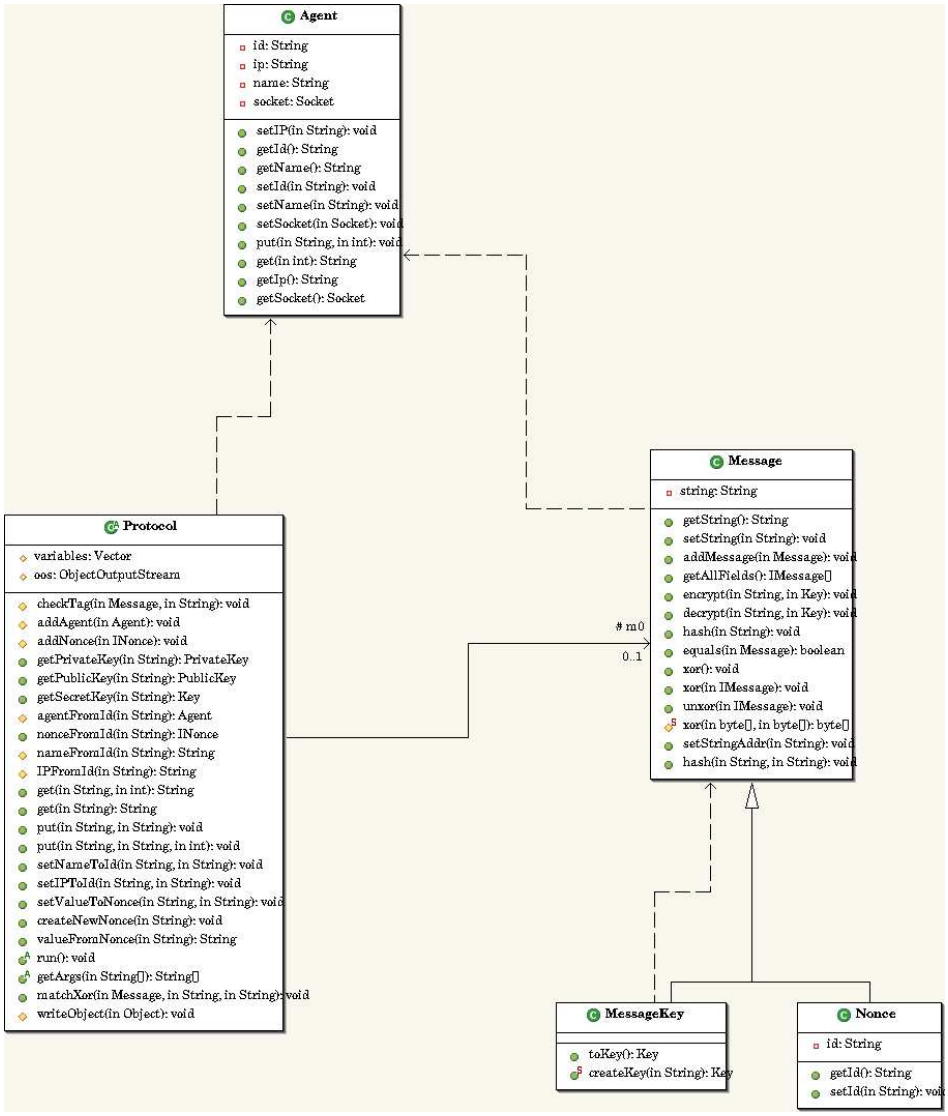


Figure 4.1: UML schema of the main classes of the implementation

The UML [BMF01] schema in Figure 3.1 represents a simplified view of the most important classes of the project. In this section, we describe the role of each of these classes in turn. For a more complete overview of all the classes of the project and their methods, see the Application Programming Interface (API) for the package "core" in directory `java/core/doc` of the

project.

4.3.1 The class Protocol

The main class is called Protocol. It is an abstract class that needs to be inherited by each agent taking part in a given protocol. For example, the compilation of the Needham Schroeder Public Key protocol generates two classes called INITIATOR and RESPONDER, both inheriting of the class Protocol. Someone who wants to run this protocol from the point of view of the initiator has to run the class INITIATOR and someone who wants to run the protocol from the viewpoint of the responder has to run the class RESPONDER.

The constructor for the classes inheriting the class Protocol needs the following arguments:

Variables An array of arguments representing the variables that the agent knows initially (as described in the "Processes" section of the input file). All these variables are stored in the private Vector "variables", together with all the new variables created during the run of the protocol. This array is optional: if it is not provided (i.e. if an empty array is provided or if the array does not contain the required number of arguments), then the program will prompt for the arguments one by one from the standard input of the program.

ObjectOutputStream An ObjectOutputStream writes primitive data types and graphs of Java objects to an OutputStream. The objects can be read (reconstituted) using an ObjectInputStream. In this ObjectOutputStream all the messages sent to the environment (i.e. such as messages 4 and 5 in the input file of the Needham-Schroeder Public Key protocol) will be sent. This lets us easily use the implementation generated by our compiler as part of a bigger project: the object that launches the protocol specifies from which ObjectOutputStream it will receive the output messages of the protocol. This ObjectOutputStream is stored in the private member called *oos*. The function *writeObject* is used to send an object through this stream. In the rest of this thesis, the ObjectOutputStream will be implicitly linked to a pretty printer which displays a readable representation of the objects returned in environmental messages on the standard output.

The class Protocol essentially provides functions to access the local variables of the agent. The most important methods of this class are:

addAgent This method adds a new agent to the list of known variables.

addNonce This method adds a new nonce to the list of known variables.

getPrivateKey, getPublicKey and getSecretKey These methods return the private key, the public key or the secret key respectively for a given agent by looking after it in the KeyStore.

agentFromId This method returns the Agent whose identity is provided (cf. the description of the class Agent).

nonceFromId This method returns the Nonce corresponding to the given identity (cf. the description of the class Nonce).

get This method waits for a message from the given agent and returns it.

put This method sends a message to another agent.

setNameToId This method sets the name of an Agent (cf. the description of class Agent). If the name of the agent is already known, it checks if the value is correct and returns an exception otherwise.

setValueToNonce This method sets the value of a Nonce (cf. the description of class Nonce). If the value of this nonce is already known, it checks if the value is correct and returns an exception otherwise.

createNewNonce This method creates a new nonce for the specified id.

valueFromNonce This method returns the value of the nonce whose id is provided.

run This method is abstract and so every class inheriting from the Protocol class will have to implement it. It describe the actions to be performed during the run of the protocol.

getArgs This method is abstract too. It represents the actions to be performed to get the arguments of the protocol. It will be used only if the arguments were not provided correctly during the instantiation of an implementation of the abstract class Protocol.

All other methods of this class are not represented on the UML schema and are less important, providing services useful only in certain cases or for certain protocols. Some of this features will be discussed in Chapter 6.

4.3.2 The class Agent

Each instance of the class Agent represents an Agent of the distributed system with whom the current agent might want to communicate. All the methods of the class Agent are means to set or access one of the following four members of an Agent:

id This member represents the identity of the agent as it is used in the protocol description section of the input script. The method *getId* returns this value and the method *setId* is used to set this value.

ip This member is the IP address of the agent. For example, in the second message of the Yahalom protocol studied in Chapter 3, b sends to a the field $addr(a)$. This is the IP address of agent a. The method *getIp* returns this value and the method *setIp* is used to set this value.

name This member represents the identity of the agent as it is used in the KeyStores. For example, in the first message of the NSPK protocol that we have studied in Chapter 3, agent A sends its *name* to agent B. The method *getName* returns this value and the method *setName* is used to set this value.

socket This object is an instance of the class Socket that is used to communicate with the agent. It can be accessed with the *getSocket* method and modified with the *setSocket* method.

4.3.3 The class Message

The class Message represents a message. Its only member is a String that represents the content of the message. This content can be accessed with the *getString* method or modified with the *setString* method. The class Message also provides all the methods that are useful to build a message or interpret it:

encrypt This method encrypts the content of a message with the algorithm and the key provided.

decrypt This method decrypts the content of a message with the algorithm and the key provided.

hash This method hashes the content of a message with the algorithm provided. See Section 6.2 for more information on this.

addMessage This method concatenates two messages and put an 'at' symbol between them, so that we can cut it later with the method *getAllFields* for example. This means that a field of a message must never contain the 'at' symbol. That is the reason why all the fields are encoded in Base64 format. Base64 is a system for representing raw byte data as ASCII characters (and the 'at' symbol is not an ASCII character).

getAllFields This method returns an array containing the different fields of a message, i.e. the different messages that have been concatenated in order to build this message.

xor This method performs a Vernam encryption between the current message and the message provided. See Section 6.3 for more information.

equals This method compares the current message with another and returns a boolean value *true* if they are equal and $false$ otherwise.

4.3.4 The class Nonce

This class inherits the class Message. It represents a nonce whose value is stored in the member of the Message class. There is another member in the class Nonce called Id. It represents the name of the nonce used in the Protocol Description section of an input script. This member is private, but can be accessed with the $getId$ method and modified with the $setId$ method.

4.3.5 The class MessageKey

This class also inherits from the class Message. It represents a message containing just a key. For example, in the message 3a of the Yahalom protocol, the field containing the key k_{ab} will be represented by this class. It is possible to get the key that is contained inside of a MessageKey object using the $toKey$ method. The method $createKey$ creates a new key for the algorithm provided.

4.4 An example of output code

COSP-J generates a file for each agent playing a role in the protocol we want to compile. These files contain a public class inheriting from the class Protocol described in the previous section.

The output code for the INITIATOR of the Needham-Schroeder Public Key Protocol defines the public class INITIATOR.

This script, like all the output files of COSP-J, is made of a class constructor and four methods called run, $subMain$, $getArgs$ and $main$.

4.4.1 The constructor method INITIATOR: lines 13-16

```
13  public class INITIATOR extends Protocol {
14
15  public INITIATOR (ObjectOutputStream oos) {
16     super("core.Message" ,"core.MessageKey" ,"core.Nonce" );
17     this.oos=oos;
18     }
```

This part of the output script defines the only constructor of the class INITIATOR. This constructor needs only one argument of type ObjectOutputStream that is used during the run of the protocol, to output the results

as in the messages 4 and 5 of the NSPK protocol input file. This constructor is used in the *subMain* static method described in the next paragraph.

Line 14 lets us use classes other than the Message, KeyMessage and Nonce classes defined in the previous section. The use of this feature will be illustrated in Chapter 7 when dealing with the prevention of type-flaw attacks.

This constructor should never be used outside the class, as the method *subMain* provides an easier way to call it.

4.4.2 The method *subMain*: lines 60-64

```
60   public static void subMain(String[] args,ObjectOutputStream oos) {
61      INITIATOR a = new INITIATOR(oos);
62      args=a.getArgs(args);
63      a.start();
64      }
```

The static method *subMain* first instantiates an object of type INITIATOR using the constructor described in the previous paragraph, then calls the method *getArgs* and finally starts the protocol by executing the method *run*. *subMain* is the method to be called in order to run an INITIATOR of the NSPK protocol.

This method takes two arguments: the first one is the list of parameters used for the run of the protocol and will be transmitted to the INITIATOR during the call of the *getArgs* method. The second one is the ObjectOutputStream needed to instantiate an INITIATOR, as explained in the previous paragraph.

4.4.3 The method *getArgs*: lines 66-94

```
66   public String[] getArgs(String[] args) {
67      try {
68         if (args.length != 5) {
69            args = (String[]) Array.newInstance("".getClass(), 5);
70            BufferedReader in2 =
71            new BufferedReader(new InputStreamReader(System.in));
72            println("Type the KeyStore file to use  [keystore]");
73            args[0]=in2.readLine();
74            if (args[0].compareTo("")==0) args[0]="keystore";
75            println("Type the password to access it [pass]");
76            args[1]=in2.readLine();
77            if (args[1].compareTo("")==0) args[1]="pass";
78            println("Type the DNS name of agent b [localhost]");
79            args[2]=in2.readLine();
80            if (args[2].compareTo("")==0) args[2]="localhost";
```

```
81            println("Type the name of agent a [a]");
82            args[3]=in2.readLine();
83            if (args[3].compareTo("")==0) args[3]="a";
84            println("Type the name of agent b [b]");
85            args[4]=in2.readLine();
86            if (args[4].compareTo("")==0) args[4]="b";
87            }
88         setKS(args[0],args[1]);
89         setIPToId(args[2],"b");
90         setNameToId(args[3],"a");
91         setNameToId(args[4],"b");
92         } catch (Exception e) {e.printStackTrace();}
93      return args;
94      }
```

The method *getArgs* sets the parameters of the INITIATOR that are initially known. In the case of the INITIATOR in the NSPK protocol, these parameters are:

- the KeyStore defined in the External section of the input file in which the SecretKey of agent A and the public key of the agent B are stored;
- the password that is used to access this KeyStore;
- the IP or DNS address of the RESPONDER agent so that we can send messages to it;
- the names of the initiator and responder agents are initial knowledge of the protocol as specified in the Processes section of the input file.

If the array provided to the *getArgs* method contains the right number of parameters, these parameters will be used. Otherwise, the program will prompt for those values.

The method *getArgs* should never be used outside the class as the method *subMain* provides an easier way to call it.

4.4.4 The method *run*: lines 18-58

```
18      public void run() {
19         try {
20
21            println("creating nonce na");
22            createNewNonce("na");
23
24            println("Building message m1");
25            Message m1 =message();
```

```
26            m1.addMessage(message(agentFromId("a")));
27            m1.addMessage(message(nonceFromId("na")));
28            println("Encrypting m1");
29            m1.encrypt("RSA", getPublicKey(nameFromId("b")));
30            println("Sending message m1");
31            put("b", m1.getString(),4000);
32
33            println("waiting for m2");
34            Message m2 = message(get("b"));
35            println("Decrypting m2");
36            m2.decrypt("RSA", getPrivateKey(nameFromId("a")));
37            addParts(m2.getAllFields(),2);
38            m0=getPart();
39            setValueToNonce(m0.getString(),"na");
40            m0=getPart();
41            setValueToNonce(m0.getString(),"nb");
42
43
44            println("Building message m3");
45            Message m3 =message();
46            m3.addMessage(message(nonceFromId("nb")));
47            println("Encrypting m3");
48            m3.encrypt("RSA", getPublicKey(nameFromId("b")));
49            println("Sending message m3");
50            put("b", m3.getString());
51
52            writeObject(nameFromId("b"));
53            writeObject(nonceFromId("na"));
54            writeObject(nonceFromId("nb"));
55
56
57            } catch (Exception e) {e.printStackTrace();System.exit(0);}
58        }
```

The method *run* defines the actions that have to be done by the initiator during a run of the NSPK protocol. The first thing to do is to create a fresh nonce *na*. This is done on line 22. After that, all messages in which agent A plays a role (namely messages 1,2,3 and 4) have to be interpreted in terms of actions as explained in Section 4.2.

Interpretation of message 1: lines 24-31

The actions associated with message 1 are:

- creating an empty message called $m1$ (line 25);

- adding the identity of agent A to this message (line 26);
- adding the value of nonce na to this message (line 27);
- encrypting this message using RSA algorithm with the public key of agent B (line 29);
- sending this message to agent B (line 31).

Interpretation of message 2: lines 33-41

The actions associated with message 2 are:

- waiting for a message from agent B (line 34);
- decrypting it using RSA algorithm with the secret key of agent A (line 36);
- decomposing the message once decrypted in 2 fields (line 37). The two fields are put inside of a stack;
- selecting the first field (line 38);
- checking that the value of this field is equal to the value of nonce na (line 39);
- selecting the second field (line 40);
- storing the value of this field into nonce nb (line 41).

Interpretation of message 3: lines 44-50

The actions associated with message 3 are:

- creating of an empty message called $m3$ (line 45);
- adding the value of nonce nb to this message (line 46);
- encrypting this message using RSA algorithm with the public key of agent B (line 48);
- sending this message to agent B (line 50).

Interpretation of message 4: lines 52-54

The message 4 expresses that the result of the run of the protocol are the identity of agent B and the values of nonces na and nb. Therefore these three values are respectively output on lines 53, 53 and 54.

4.4.5 The method *main*: lines 96-105

```
96      public static void main(String[] args) {
97         ObjectOutputStream oos=null;
98         try {
99            ByteArrayOutputStream os = new ByteArrayOutputStream();
100           oos=new ObjectOutputStream(os);
101           PrintOOS p=new PrintOOS("INITIATOR",oos,os);
102           p.start();}
103        catch (Exception e) {e.printStackTrace();}
104        subMain(args,oos);
105        }
106     }
```

The static method *main* is not really part of the protocol, but provides an easy way to test the protocol. It creates a new ObjectOutputStream (lines 98, 99 and 100) and connects it to a pretty printer called PrintOOS (line 101 and 102). Finally, the protocol is started (line 104). The examples of protocol runs shown in the next paragraph use this test function.

4.5 Examples of protocol runs

A trace is the list of all output generated by all agents of a given security protocol when they run altogether. This list is ordered chronologically and for each item, specifies which agent generated this output. Output lines can be of two kinds. Some of them correspond to the use of the *println* function inside of the *run* method of a class inheriting the abstract class Protocol. The other kind of output corresponds to the interpretation of environmental messages such as messages 4 and 5 in the Needham-Schroeder Public Key protocol whose input file is given in Chapter 3.

4.5.1 Trace of the Needham-Schroeder Public Key Protocol

The script used to produce the trace of the Needham-Schroeder Public Key Protocol is:

```
(sleep 5 ; java protocols/INITIATOR keystore1 *** localhost Alice Bob) &
java protocols/RESPONDER keystore2 *** Bob
```

Both agents A and B are running on the same computer *localhost*, but this does not change the way the protocol works. The file *keystore*1 contains the public key and secret key of *Alice* and a certificate containing the public key of *Bob*. The file *keystore*2 contains the public key and secret key of *Bob* and a certificate containing the public key of *Alice*. The method used to generate these two KeyStore files is presented in Appendix A. The trace

generated by this script illustrates the actions performed by agents A and B and also shows that they agree at the end on the values of nonces na and nb:

```
INITIATOR: creating nonce na
INITIATOR: Building message m1
INITIATOR: Encrypting m1
INITIATOR: Sending message m1
RESPONDER: creating nonce nb
INITIATOR: waiting for m2
RESPONDER: waiting for m1
RESPONDER: Decrypting m1
RESPONDER: Building message m2
RESPONDER: Encrypting m2
RESPONDER: Sending message m2
RESPONDER: waiting for m3
INITIATOR: Decrypting m2
INITIATOR: Building message m3
INITIATOR: Encrypting m3
INITIATOR: Sending message m3
RESPONDER: Decrypting m3
INITIATOR: Bob
INITIATOR: na: 0QT70Ge3Dq3Bvg==
INITIATOR: nb: CAz+ZgIgegbeEA==
RESPONDER: Alice
RESPONDER: na: 0QT70Ge3Dq3Bvg==
RESPONDER: nb: CAz+ZgIgegbeEA==
```

4.5.2 Trace of the Yahalom protocol

Once the input script of Section 3.6.2 is compiled with COSP-J, it generates files YAHAINIT.java, YAHARESP.java and YAHASERV.java. The compilation of these three files with a java compiler creates files YAHAINIT.class, YAHARESP.class and YAHASERV.class. We can then use these files in the following script:

```
(sleep 10 ; java protocols/INITYAHA secretkeyfile localhost a b) &
(sleep 5 ; java protocols/RESPYAHA secretkeyfile localhost b s) &
java protocols/SERVYAHA secretkeyfile s
```

The trace generated by this script illustrates the actions that each of the 3 agents perform, together with the interactions between these agents:

```
INITYAHA: creating nonce na
INITYAHA: Building message m1
INITYAHA: Sending message m1
```

```
RESPYAHA: creating nonce nb
INITYAHA: waiting for m3a
RESPYAHA: waiting for m1
RESPYAHA: Building message m2
RESPYAHA: Encrypting m0
RESPYAHA: Sending message m2
SERVYAHA: creating DESkey kab
RESPYAHA: waiting for m3b
SERVYAHA: waiting for m2
SERVYAHA: Decrypting m0
SERVYAHA: Building message m3a
SERVYAHA: Encrypting m3a
SERVYAHA: Sending message m3a
INITYAHA: Decrypting m3a
SERVYAHA: Building message m3b
SERVYAHA: Encrypting m3b
SERVYAHA: Sending message m3b
RESPYAHA: Decrypting m3b
RESPYAHA: waiting for m4
INITYAHA: Building message m4
INITYAHA: Encrypting m4
INITYAHA: Sending message m4
RESPYAHA: Decrypting m4
RESPYAHA: a
RESPYAHA: rO0ABXNyAB5jb20uc3VuLmNyeXB0by5wcm92aWRlc
i5ERVNLZXlrNJw12hVomAIAAVsAA2tleXQA
AltCeHB1cgACW0Ks8xf4BghU4AIAAHhwAAAACBwgSRk7L7z9
INITYAHA: b
INITYAHA: rO0ABXNyAB5jb20uc3VuLmNyeXB0by5wcm92aWRlc
i5ERVNLZXlrNJw12hVomAIAAVsAA2tleXQA
AltCeHB1cgACW0Ks8xf4BghU4AIAAHhwAAAACBwgSRk7L7z9
```

In the previous chapter, we described an input script of COSP-J. In this chapter, we described what the output code is. The next chapter will link these two together by explaining how COSP-J builds the output code from the input script.

Chapter 5

Building the output code

In this chapter, we describe the main algorithms of COSP-J. COSP-J is written in Haskell 98 [Pey03]. Haskell is a general purpose, purely functional programming language. The reason why COSP-J is written in Haskell is that it reuses some parts of the code of Casper which is also written in Haskell 98.

The execution of COSP-J, just like Casper, can be decomposed in 4 steps, namely the parsing, type checking, consistency checking and code generation. The Main.lhs file executes this four steps one by one and throws an error to the user if one of them fails.

This chapter will first describe the parts of the Casper code that are reused in COSP-J and the modifications that have been done on them, and then present the new algorithms that have been created for COSP-J.

5.1 The reuse of the Casper code

The similarity between a Casper input file and a COSP-J input file made it easier to adapt the parser, type checker and consistency checker of Casper for COSP-J instead of rewriting them.

5.1.1 The parser

A parser is a program that analyzes and organizes formal language statements into a usable form for a given purpose. For instance, the Casper parser analyzes a Casper input file, checks if its structure is correct and returns a data structure that contains all the information of the input script.

The parser of Casper is contained in the files Parse.lhs and Parse1.lhs. Parse1.lhs deals with the parsing of the protocol description part of an input file and is called by Parse.lhs. Parse.lhs defines the function `parse`:

```
parse :: String -> Maybe_ Input
```

The `Input` type is a tuple combining types for each of the sections of an Input file. The `Maybe_` expresses the fact that `parse` may also return an error message (if the input file does not have the expected structure).

The modifications done on the Casper parser for COSP-J correspond to the changes of structure of the Input file. For example, the addition of the "External" section, the suppression of the "Intruder Information", "System", "Specification" and "Actual Variables" sections and the complete change of the "Functions" section.

5.1.2 The type checker

A type checker is a function that checks whether there are any type error inside of a script. For example, if PK is defined by `PK : Agent -> RSAPublicKey` and na is defined by `na : Nonce`, then writing `PK(na)` will give an error in the type checker, because PK needs an argument of type Agent and is given one of type Nonce.

In Casper, the files TypeCheck.lhs, TypeCheckpd.lhs and TypeCheckDI.lhs implement the type checker. The TypeCheck.lhs file uses TypeCheckDI.lhs and TypeCheckpd.lhs which deals with the type checking of the protocol description section of the input scripts. It defines the function `typecheck`:

```
typecheck :: Input -> (String, String)
```

This function returns two strings: one contains the errors detected and the other contains the warning messages.

The modifications done on the Casper type checker for COSP-J concerns, for example, the fact that types have to be recognized by the compiler (for example, Nonce becomes a keyword), the different use of hash functions and the *addr* function (which is predefined).

5.1.3 The consistency checker

A consistency checker is a function that checks for consistency errors inside of a script. For example, if in the first message of a protocol, an agent a is supposed to send the value of nonce na to agent b but does not know this value, it will rise a consistency error.

In Casper, the files Consistency.lhs and ConsistencyDI.lhs define the consistency checker. Consistency.lhs defines the function `consistency`:

```
consistency :: Input -> (String, String)
```

This function returns the errors and warnings that have been detected.

The modifications done on the Casper consistency checker for COSP-J concern the addition of the consistency error described in Section 3.1 about the impossibility to send a message to an agent whose IP address is unknown and the fact that we do not know the name of an agent just by receiving a message from it.

5.2 The main routines of the code creation

The complete difference between the CSP code generated by Casper and the Java code generated by COSP-J made it easier to rewrite from scratch the code generation algorithms. The code generation routine of COSP-J are included inside of the Compile.lhs and CompileRun.lhs files. The Compile.lhs file defines a function *makeOutput* that creates the output of a given protocol *Input* for a given agent. It is called for each agents in the Main.lhs file and the results are stored in the corresponding java files. The CompileRun.lhs defines a function *makeRun* that generates the run method of the output. This is the most important part of the code generation.

The *makeRun* function is called for each of the agents of the protocol. For each messages in which the agent plays a role, *makeRun* calls one of the sub-functions called $makeMsg$ and *match*. The first one generates the code for a message to be built and sent. The second one generates the code for to be received and interpreted. The structure of these two functions is based on the definition of the type Msg that is used for all messages. This type is defined recursively in the Message.lhs file:

```
data Msg = Atom VarName
|  Encrypt Msg [Msg]
|  Sq [Msg]
|  Xor Msg Msg
|  Undec Msg VarName
|  Forwd VarName Msg
|  Apply VarName Msg
          deriving (Eq, Ord, Show)
```

This means that a message can respectively be an atomic value (for example, the value of a nonce or the identity of an agent) or a list of messages (usually called fields) encrypted with a message (usually called key), or a list of messages (usually called fields), or the Vernam encryption of two messages (see Section 6.3) or a message to store in a variable name without further interpretation (see Section 6.1) or a variable that will be interpreted as a message (see Section 6.1) or a function applied to a message (for example, a hash function, as described in Section 6.2).

The $makeMsg$ and *match* functions handle all these cases and recursively call themselves when needed. For example, to build a message made of a list of messages encrypted with a message (for example the first message of the NSPK protocol), we first generate this list of messages and then encrypt it. To generate the list of messages, we generate the messages one by one and concatenate them.

This chapter explored the main algorithms of COSP-J and showed how the output code is generated from an input file. The next chapter will give

a more complete overview of how to build an input script by presenting additional features that can be used in a COSP-J input script.

Chapter 6

Additional features

There exists a wide variety of security protocols. In this chapter, we explore some additional features that are useful in the input scripts of COSP-J for a broader class of security protocols than we have seen so far.

6.1 The use of the %-notation

The %-notation is a feature of Casper's input files that lets the user express the fact that a given part of a message is not intended to be decrypted, but just forwarded in a next step to another agent. For example, here is another version of the Yahalom protocol [BAN89]:

Message 1. $a \rightarrow b \ : \ a, na$
Message 2. $b \rightarrow s \ : \ b, \{a, na, nb\}_{ServerKey(b)}$
Message 3. $s \rightarrow a \ : \ \{b, kab, na, nb\}_{ServerKey(a)}, \{a, kab\}_{ServerKey(b)}$
Message 4. $a \rightarrow b \ : \ \{a, kab\}_{ServerKey(b)}, \{nb\}_{kab}$

where a and b are agents, s is a trusted server, $ServerKey(x)$ is a secret key shared by s and x, kab is a fresh key created by s, na is a fresh nonce created by a and nb is a fresh nonce created by b.

In this version of the Yahalom protocol, s sends to a both messages 3a and 3b of the previous version of the Yahalom protocol that we used in Section 3.1 as a single message 3.

This means that a receives $\{a, kab\}_{ServerKey(b)}$ but is not able to decrypt it: he simply forwards it on to b in the fourth message.

We write `m%v` where `m` is a message and `v` the name for a variable to denote that the receiver stores the message `m` into the variable `v`.

We write `v%m` to indicate that the receives should receive a message of structure `m` whereas the sender simply sends the value of variable `v`.

This very useful feature works in COSP-J exactly in the same way as in Casper.

The protocol description section of a COSP-J input file for this version of the Yahalom protocol is therefore:

```
#Protocol description
1.  a -> b : a,na
2.  b -> s : {a, b, addr(a), na, nb}{ServerKey(b)}
3.  s -> a : {b, kab, na, nb}{ServerKey(a)},{a, kab}{ServerKey(b)}%c
4.  a -> b : c%{a, kab}{ServerKey(b)}, {nb}{kab}
5.  a ->   : b,kab
6.  b ->   : a,kab
```

The rest of the input script is the same as for the previous version of the Yahalom protocol that we studied in Section 3.6.2.

6.2 The use of hash functions

A hash (also called message digest) is a special number calculated from an arbitrary amount of input data.

A signature (also called MAC for Message Authentication Code) is basically a hash encrypted with a key. When an agent receives a signature along with a message, he can be sure of who sent this message. For example, consider the following security protocol:

$$\begin{aligned} &\text{Message 1.} \quad a \rightarrow b \ : \ \{a, b, na\}_{PK(b)} \\ &\text{Message 2.} \quad b \rightarrow a \ : \ \{a, b, na\}_{PK(a)} \end{aligned}$$

where a and b are agents, $PK(x)$ is the public key of agent x and na is a fresh nonce created by a.

The aim of this protocol is to achieve mutual authentication of agents a and b, and to establish a secret value na known only by a and b.

When a sends $\{a, b, na\}_{PK(b)}$ to b, he can be sure that nobody else other than b can read it. But how can b be sure that the message comes from a? This protocol suffers the following obvious attack:

$$\begin{aligned} &\text{Message 1.} \quad I_A \rightarrow B \ : \ \{A, B, na\}_{PK(B)} \\ &\text{Message 2.} \quad B \rightarrow I_A \ : \ \{B, A, na\}_{PK(A)} \end{aligned}$$

where A and B are playing roles a and b respectively, and I_X represents the intruder impersonating agent X.

Even if the intruder I is unable to decrypt message 2, agent B thinks he completed a run of the protocol with agent A. We propose tackling this issue by adding a signature in the first message as follows:

$$\begin{aligned} &\text{Message 1.} \quad a \rightarrow b \ : \ \{a, b, na\}_{PK(b)}, \{hash(\{a, b, na\}_{PK(b)})\}_{SK(a)} \\ &\text{Message 2.} \quad b \rightarrow a \ : \ \{a, b, na\}_{PK(a)} \end{aligned}$$

And now an intruder is not able to forge message 1 anymore, because he cannot encrypt the message digest with $SK(a)$. This protocol was successfully checked with Casper. The Casper input script used for the check can be found in Appendix E.

To implement this protocol with COSP-J, we need to define a function f of type $MD5HashFunction$ and use it in the protocol description:

```
#Free variables
f : MD5HashFunction
A, B : Agent
na : Nonce
PK : Agent -> RSAPublicKey
SK : Agent -> RSASecretKey
InverseKeys = (PK, SK)

#External
myKeyStore : KeyStore

#Processes
INITXAV(A,B) knows PK, SK(A), f generates na
RESPXAV(B) knows PK, SK(B), f

#Protocol description
1.  A -> B : {A,B,na}{PK(B)}, {f({A,B,na}{PK(B)})}{SK(A)}
2.  B -> A : {A,B,na}{PK(A)}
3.  A ->   : B,na
4.  B ->   : A,na

#Functions
PK = myKeyStore.PK
SK = myKeyStore.SK
```

6.3 The use of Vernam encryption

Vernam encryption is a common form of encryption in which we form the bitwise exclusive-or of two arrays of bytes. We write $a \oplus b$ to indicate the Vernam encryption of a with b. If an agent knows $a \oplus b$ and a, he can find the value of b which is equal to $(a \oplus b) \oplus a$. If an agent knows $a \oplus b$ and b, he can find the value of a which is equal to $(a \oplus b) \oplus b$. But if an agent knows only $a \oplus b$, he cannot deduce a or b.

The TMN protocol [TMN90] makes use of Vernam encryption:

$$\begin{aligned}
&\text{Message 1.} \quad a \rightarrow s \ : \ b, \{ka\}_{PK(s)} \\
&\text{Message 2.} \quad s \rightarrow b \ : \ a \\
&\text{Message 3.} \quad b \rightarrow s \ : \ a, \{kb\}_{PK(s)} \\
&\text{Message 4.} \quad s \rightarrow a \ : \ kb \oplus ka
\end{aligned}$$

where a and b are agents, s is a trusted server, $PK(x)$ is the public key of agent x, ka and kb are session symmetric keys freshly created by a and b respectively.

The TMN protocol seeks to establish a session key kb between agents a and b.

To implement this protocol with COSP-J, we use the `(+)` notation that indicates the use of Vernam encryption:

```
#Free variables
a, b, s : Agent
ka, kb : Nonce
PK : Agent -> RSAPublicKey
SK : Agent -> RSASecretKey
InverseKeys = (PK, SK) , (ka, ka) , (kb, kb)

#External
myKeyStore : KeyStore

#Processes
INITTMN(a,b,s) knows PK, SK(a) generates ka
RESPTMN(b) knows PK, SK(b) generates kb
SERVTMN(s) knows PK, SK(s)

#Protocol description
1.  a -> s : a,b,addr(b), {ka}{PK(s)}
2.  s -> b : s, a
3.  b -> s : a, {kb}{PK(s)}
4.  s -> a : kb (+) ka
5.  a ->   : b,kb
6.  b ->   : a,kb

#Functions
PK = myKeyStore.PK
SK = myKeyStore.SK
```

6.4 The use of the angle brackets notation

The angle brackets notation is a Casper feature that is used to make some assignments.

In order to illustrate this, we will consider the famous Diffie-Hellman key exchange algorithm [DH76]:

$$\begin{aligned} &\text{Message 1.} \quad A \rightarrow B \ : \ P, G \\ &\text{Message 2.} \quad A \rightarrow B \ : \ G^X \bmod P \\ &\text{Message 3.} \quad B \rightarrow A \ : \ G^Y \bmod P \end{aligned}$$

In this protocol, P, G and X are big integers created by A and Y is a big integer created by B. Diffie-Hellman key establishment aims to establish a session key between agents A and B who do not already share a secret key. At the end of the protocol, A is able to calculate $K = (G^Y \bmod P)^X \bmod P$ and B can calculate $K = (G^X \bmod P)^Y \bmod P$. These are both equal to $G^{XY} \bmod P$.

An intruder who listens to this protocol will not be able to gain the value of k from the values of P, G, $G^X \bmod P$ and $G^Y \bmod P$.

Diffie-Hellman key establishment only works if the messages are authenticated in some way. Otherwise there is the following obvious attack in which an intruder I can calculate the key that B thinks he is sharing with A:

$$\begin{aligned} &\text{Message 1.} \quad I_A \rightarrow B \ : \ P, G \\ &\text{Message 2.} \quad I_A \rightarrow B \ : \ G^X \bmod P \\ &\text{Message 3.} \quad B \rightarrow I_A \ : \ G^Y \bmod P \end{aligned}$$

The Station-to-Station protocol [DvOW92] provide a way to run Diffie-Hellman key establishment in an authenticated way by having the agents sign the exponents. Appendix B presents the COSP-J input file of the Station-to-Station protocol.

To implement the Diffie-Hellman key establishment algorithm using COSP-J, we need the angle brackets notation:

```
#Free variables
A, B : Agent
g, m, x, y, gEx, gEy, k : BigInteger
InverseKeys = (k,k)

#Processes
INITDH(A,B) generates g,m,x
RESPDH(B) generates y
```

```
#Protocol description
1. A -> B : g,m
<gEx := g.modPow(x,m)>
2. A -> B : g,m,gEx
<k := gEx.modPow(y,m) ; gEy := g.modPow(y,m)>
3. B -> A : gEy
<k := gEy.modPow(x,m)>
4. A ->   : k
5. B ->   : k
```

In message 1, agent A sends the values of G and P to agent B. Before sending message 2, agent A calculates $G^X \bmod P$, bounds it in the variable gEx and send it to B in message 2. Then B calculates $(G^X \bmod P)^Y \bmod P$ and store it in variable k and also calculate $G^Y \bmod P$, bounds it in the variable gEy and send it to A in message 3. Finally, the agent A calculates $(G^Y \bmod P)^X \bmod P$ and assign it to variable k.

Chapter 7

Preventing type flaw attacks

This chapter presents the system that is implemented in COSP-J in order to prevent type flaw attacks.

7.1 Definition of a type flaw attack

Some protocols suffers from type flaw attacks, where a field of one type is interpreted as being of another type.

For example, consider the Woo and Lam Protocol Pi_1 [WL94]:

$$\begin{array}{lll}
\text{Message 1.} & a \rightarrow b & : \ a \\
\text{Message 2.} & b \rightarrow a & : \ nb \\
\text{Message 3.} & a \rightarrow b & : \ \{a, b, nb\}_{shared(a,s)} \\
\text{Message 4.} & b \rightarrow s & : \ \{a, b, \{a, b, nb\}_{shared(a,s)}\}_{shared(b,s)} \\
\text{Message 5.} & s \rightarrow b & : \ \{a, b, nb\}_{shared(b,s)}
\end{array}$$

where a and b are agents, s is a trusted server, $shared(x, s)$ is a secret key that agents x and s share, and nb is a nonce.

The aim of this protocol is to have agent b be sure that agent a is who he claims to be in message 1. But it suffers from the following attack:

$$\begin{array}{lll}
\text{Message 1.} & I_A \rightarrow B & : \ A \\
\text{Message 2.} & B \rightarrow I_A & : \ N_b \\
\text{Message 3.} & I_A \rightarrow B & : \ N_b \\
\text{Message 4.} & B \rightarrow I_S & : \ \{A, B, N_b\}_{shared(B,S)} \\
\text{Message 5.} & I_S \rightarrow B & : \ \{A, B, N_b\}_{shared(B,S)}
\end{array}$$

where A and B are playing roles of a and b respectively, and I_X represents the intruder impersonating agent X.

In this attack, the intruder I initiates a protocol run pretending to be agent A. I sends A's identity in message 1 to B, receives the nonce N_b

in message 2 from B and sends back N_b to B in message 3 instead of $\{A, B, N_b\}_{shared(A,S)}$. Since B is not supposed to decrypt this message, it will simply forward it to S in message 4. I intercepts message 4, is unable to decrypt it but simply sends it back to B in message 5. B receives exactly what it was expecting in message 5 (supposedly from S) and therefore thinks he has completed a run of the protocol with A.

As explained in [DNL99], Casper (in common with most of the protocol analysis techniques) is not good at finding type flaw attacks. Casper has a limited feature whereby an atomic value can be defined as having two different types. This feature can be used to find the attack upon the Woo and Lam Pi_1 protocol, as reported in [Low96b].

That is the reason why COSP-J includes a type flaw preventing system.

7.2 How to prevent type flaw attacks

Type flaw attacks can be prevented by tagging each field –atomic values, concatenations, encrypted components, etc.– with some information giving a claimed type. The tag for concatenations must include enough information to allow the concatenation to be split into components correctly. The tag for encryptions must include the type of the encryption key and of the body. All the honest agents check that the tags of a received message is as expected and tag their own messages accordingly. This technique is fully described and proved in [HLS00]. We illustrate how such a tag system would prevent the attack upon the Woo and Lam Pi_1 protocol as follows:

If the intruder I tries to perform the same attack above with the tagging system, he needs to put a tag in message 3, pretending that it is of the type of $\{a, b, nb\}_{shared(a,s)}$. B receives it and includes it in the message 4. I intercepts this message 4 and sends it back to B without being able to modify the tag inside of the encryption. When B receives message 5 and decrypts it, he see that the last field is of type of $\{a, b, nb\}_{shared(a,s)}$, whereas a simple nonce tag was expected. Therefore the attack described above does not work anymore thanks to the tagging system.

7.3 Implementation in COSP-J

This section describes how the system to prevent type flaw attacks is implemented in COSP-J.

7.3.1 Object-Oriented Model to prevent type flaw attacks

Figure 7.1 is a UML schema showing how the type flaw prevention system described above is implemented in COSP-J.

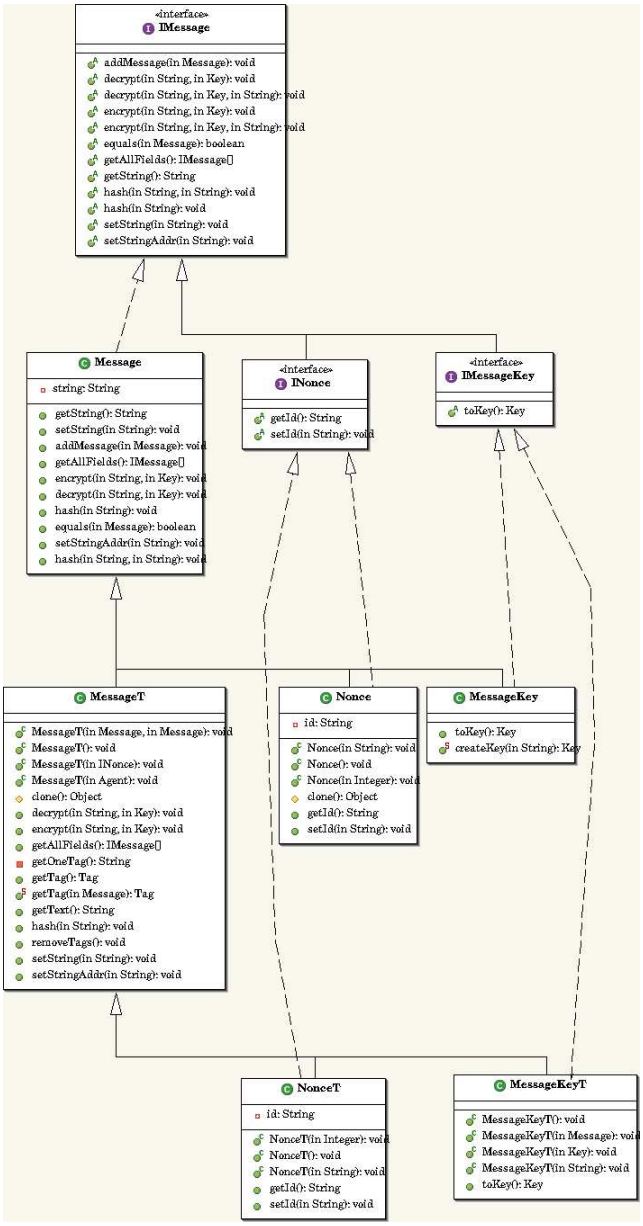


Figure 7.1: UML schema of the type flaw prevention system

Section 4.3 has already presented the Message, Nonce and MessageKey classes that are used to represent a message, a nonce and a message containing a key respectively. Figure 7.1 shows three more classes called MessageT, NonceT and MessageKeyT. They are used to represent a tagged message, a tagged nonce and a tagged message containing a key respectively. This means that, depending on whether or not we want to activate the type flaw

prevention system, we will use one set of classes or the other. In order to do so, COSP-J defines three interfaces: the IMessage interface implemented by the Message and MessageT classes, the INonce interface implemented by the Nonce and NonceT classes and the IMessageKey interface implemented by the MessageKey and MessageKeyT classes. These interfaces are useful for the rest of the code to handle the classes implementing the different kind of messages. For example, a method in the Protocol class can use an IMessage object without knowing if the message is tagged or not, because the IMessage interface defines all the abstract methods that the inheriting classes will need to implement (such as the encrypt or decrypt methods that have the same signature in the Message and MessageT classes, but are implemented differently because for a tagged message we need to check if the tag is valid).

7.3.2 Modifications of the output code when the type-flaw prevention system is active

The 11th line of the `CompileRun.lhs` file determines if the type-flaw prevention system is active or not: it is active if the *tag* variable is equal to $True$ and inactive if it is equal to $False$.

The Protocol abstract class contains three members that were hidden in Figure 4.1. These members are private, abstract and of the Class type. Instances of the class Class represent classes and interfaces in a running Java application. These members point to the three implementations of the IMessage, INonce and IMessageKey that will be used by the protocol. These members are set during the instantiation of the Protocol class. The Protocol class contains only one constructor (that is hidden on Figure 4.1), taking three parameters that are the names of the classes to use. For example, in the output code of the Needham-Schroeder Public Key protocol initiator that we studied in Section 4.4, the first line of the INITIATOR constructor (line 14) reads:

```
super("core.Message" ,"core.MessageKey" ,"core.Nonce" );
```

The call to the super method is in fact a call to the constructor of the Protocol class, because the INITIATOR class inherits the Protocol class. This line means that the Message, MessageKey and Nonce classes will be used throughout this implementation of the NSPK initiator. In turn, this means that the type flaw prevention system is deactivates.

If we activate the type flaw prevention system and recompile the NSPK protocol, this line becomes:

```
super("core.MessageT" ,"core.MessageKeyT" ,"core.NonceT" );
```

This means that now the MessageT, MessageKeyT and NonceT are used. The tags will therefore be created automatically when we call the methods of these classes. The only other change in the output code is the tag check

system. If we activate the type flaw prevention system and recompile the NSPK protocol, the run method becomes:

```
18  public void run() {
19     try {
20
21        println("creating nonce na");
22        createNewNonce("na");
23
24        println("Building message m1");
25        Message m1 =message();
26        m1.addMessage(message(agentFromId("a")));
27        m1.addMessage(message(nonceFromId("na")));
28        println("Encrypting m1");
29        m1.encrypt("RSA", getPublicKey(nameFromId("b")));
30        println("Sending message m1");
31        put("b", m1.getString(),4000);
32
33        println("waiting for m2");
34        Message m2 = message(get("b"));
35        checkTag(m2,"PNN)");
36        println("Decrypting m2");
37        m2.decrypt("RSA", getPrivateKey(nameFromId("a")));
38        checkTag(m2,"NN");
39        addParts(m2.getAllFields(),2);
40        m0=getPart();
41        setValueToNonce(m0.getString(),"na");
42        m0=getPart();
43        setValueToNonce(m0.getString(),"nb");
44
45        println("Building message m3");
46        Message m3 =message();
47        m3.addMessage(message(nonceFromId("nb")));
48        println("Encrypting m3");
49        m3.encrypt("RSA", getPublicKey(nameFromId("b")));
50        println("Sending message m3");
51        put("b", m3.getString());
52
53        writeObject(nameFromId("b"));
54        writeObject(nonceFromId("na"));
55        writeObject(nonceFromId("nb"));
56
57     } catch (Exception e) {e.printStackTrace();System.exit(0);}
58  }
```

If we compare this output code with the one generated without the type flaw prevention system in Section 4.4, we find that it is exactly the same, except that lines 35 and 38 have been added. These lines correspond to the tag check system.

In line 35, the tag for the whole message m2 that was received in line 34 is compared to what it is supposed to be, namely a tag representing two nonces encrypted with a public key. The "N" letter represents a nonce and the "Pxxx)" form represents the message xxx encrypted with a public key. Therefore the correct tag for the second message of the Needham-Schroeder Public Key protocol is "PNN)".

In line 38, the tag inside of the encryption of message m2 is compared to what it is supposed to be, namely a tag "NN" representing two nonces unencrypted.

The checkTag method will check if the tag is as expected and will return an exception if it is not the case.

7.3.3 The tagging system

The tag string is created according to the following rules:

- the letter "N" represents a nonce;
- the letter "A" represents the identity of an agent;
- the letter "p" represents a public key;
- the letter "s" represents a private key;
- the letter "k" represents another type of key;
- the letter "I" represents a big integer;
- the concatenation of two tags represents the concatenation of two messages. For example "NN" is the tag representing a message containing two nonces unencrypted;
- the "Pxxx)" form represents the message xxx encrypted with a public key;
- the "Sxxx)" form represents the message xxx encrypted with a secret key;
- the "Kxxx)" form represents the message xxx encrypted with another type of key.

7.3.4 Example: NSPK protocol

When the type-flaw prevention system is activated, the trace of the Needham-Schroeder Public Key protocol run becomes:

```
INITIATOR: creating nonce na
INITIATOR: Building message m1
INITIATOR: Encrypting m1
INITIATOR: Sending message m1
RESPONDER: creating nonce nb
INITIATOR: waiting for m2
RESPONDER: waiting for m1
RESPONDER: Tag check successful for PAN)
RESPONDER: Decrypting m1
RESPONDER: Tag check successful for AN
RESPONDER: Building message m2
RESPONDER: Encrypting m2
RESPONDER: Sending message m2
RESPONDER: waiting for m3
INITIATOR: Tag check successful for PNN)
INITIATOR: Decrypting m2
INITIATOR: Tag check successful for NN
INITIATOR: Building message m3
INITIATOR: Encrypting m3
INITIATOR: Sending message m3
RESPONDER: Tag check successful for PN)
RESPONDER: Decrypting m3
INITIATOR: Bob
RESPONDER: Tag check successful for N
RESPONDER: Alice
```

This trace is the same as the one we studied in Section 4.5.1, except that each time an agent receives a message, it checks the validity of all tags.

This chapter showed what type flaw attacks are, how to prevent them formally and how they are prevented in COSP-J. The next chapter will deal with multi-protocol attacks.

Chapter 8

Preventing multi-protocol attacks

In this chapter, we deal with another kind of attack that can not be detected by Casper called multi-protocol attack. Firstly we define what multi-protocol attacks are, then we formally present a system to prevent them and finally we show how this system is implemented in COSP-J. Our goal is to ensure that the implementations of security protocols created by COSP-J are not vulnerable to multi-protocol attacks.

8.1 Definition of a multi-protocol attack

It is sometimes possible to replay a message from a protocol P_1 in a run of some other protocol P_2 that is being executed concurrently with P_1; attacks arising from such replays are known as *multi-protocol attacks*. Casper can check the validity of a protocol, but is unable to check a set of protocols running at the same time. This means that Casper is unable to detect multi-protocol attacks.

8.2 Preventing multi-protocol attacks

In order to prevent multi-protocol attacks, we need to make it possible to tell from which protocol a component comes. If we can achieve this and ensure that all honest agents check that the messages really come from the expected protocol, then it is impossible for an attacker to replay an encrypted component from one protocol to another.

One solution is to include a protocol identification field inside each encrypted component. [GT00] proves that this prevents multi-protocol attacks.

This method is the one we will implement in COSP-J to prevent multi-protocols attacks.

8.3 Implementation within COSP-J

The 12th line of the `CompileRun.lhs` file determines if the system preventing multi-protocol attacks is active or not: it is active if the *protoTag* variable is equal to $True$ and inactive if it is equal to $False$.

The tags we use in COSP-J to identify from which protocol a given component comes, are generated by using a hash function on the input script. The hash function we use is a MD5 hash function. The haskell code for the MD5 function we use in COSP-J was created by Ian Lynagh, a research student from the Oxford University Computing Laboratory. It is included is the MD5.lhs file.

The Protocol class contains a static protected member called protoTag which is a String representing the tag to use for the current protocol. This member is hidden in Figure 4.1. If we activate the multi-protocol attack prevention system and compile the Needham-Schroeder Public Key protocol for example, an additional line is generated for the initiator compared to the output code in Section 4.4:

```
static String protoTag="6adc1b80ddc08e287b90e9b4da968770";
```

This line sets the value of the protoTag member to be used in the protocol.

The Message class defines three methods that are hidden in both Figures 4.1 and 7.1. They are similar to the encrypt, decrypt and hash methods except that they take an additional argument which is the tag for the protocol. The encrypt method will include this tag in the encrypted part of the message, the hash method will hash the tag along with the message and the decrypt method will check that the tag is correct after the decryption of the message; an exception is returned in the case where it is not correct.

Therefore the run method for the initiator of the Needham-Schroeder Public Key protocol becomes:

```
20      public void run() {
21         try {
22
23            println("creating nonce na");
24            createNewNonce("na");
25
26            println("Building message m1");
27            Message m1 =message();
28            m1.addMessage(message(agentFromId("a")));
29            m1.addMessage(message(nonceFromId("na")));
30            println("Encrypting m1");
31            m1.encrypt("RSA", getPublicKey(nameFromId("b")),protoTag);
32            println("Sending message m1");
33            put("b", m1.getString(),4000);
```

```
34
35            println("waiting for m2");
36            Message m2 = message(get("b"));
37            println("Decrypting m2");
38            m2.decrypt("RSA", getPrivateKey(nameFromId("a")),protoTag);
39            addParts(m2.getAllFields(),2);
40            m0=getPart();
41            setValueToNonce(m0.getString(),"na");
42            m0=getPart();
43            setValueToNonce(m0.getString(),"nb");
44
45
46            println("Building message m3");
47            Message m3 =message();
48            m3.addMessage(message(nonceFromId("nb")));
49            println("Encrypting m3");
50            m3.encrypt("RSA", getPublicKey(nameFromId("b")),protoTag);
51            println("Sending message m3");
52            put("b", m3.getString());
53
54            writeObject(nameFromId("b"));
55            writeObject(nonceFromId("na"));
56            writeObject(nonceFromId("nb"));
57
58
59            } catch (Exception e) {e.printStackTrace();System.exit(0);}
60        }
```

The only difference with the run method that we studied in Section 4.4, without the multi-protocols attacks prevention system, is that all encrypt and decrypt methods use the additional "protoTag" argument.

Thanks to this system, the implementation of security protocols generated by COSP-J are not vulnerable to multi-protocols attacks. The next chapter presents a complete case study for the implementation of an e-commerce protocol.

Chapter 9

Case-study: An e-commerce protocol

This chapter is a case study of the implementation of a complex protocol. E-commerce (electronic commerce or EC) is the buying and selling of goods and services on the Internet, usually through the World Wide Web. Clearly, we want e-commerce to be carried out securely: a customer wants to be sure of what he is buying and the price he is paying; the merchant wants to be sure of receiving payment; both sides want to end up with evidence of the transaction, in case either side denies it took place; and the act of purchase should not leak secrets, such as credit card details, to an eavesdropper.

9.1 An e-commerce protocol

The e-commerce protocol that we implement in this chapter was designed in [Men02] but never implemented. This protocol is completely based on Public-Key Cryptography. Three actors are involved in this protocol: a customer a, who wants to purchase goods on the Internet, a merchant c, who sells goods, and a bank b where customer a has an account.

The bank b is the only trusted operator of the system. This means that the protocol must protect participants from malicious customers and merchants, but will assume the bank to be trustworthy.

The goals of the protocol are to provide:

1. authentication between the different agents;
2. agreement and confidentiality of the goods sold and their corresponding prices;
3. non-repudiation, i.e. evidence of the transaction for each participant.

Figure 10.1 illustrates the messages sent during a run of the protocol.

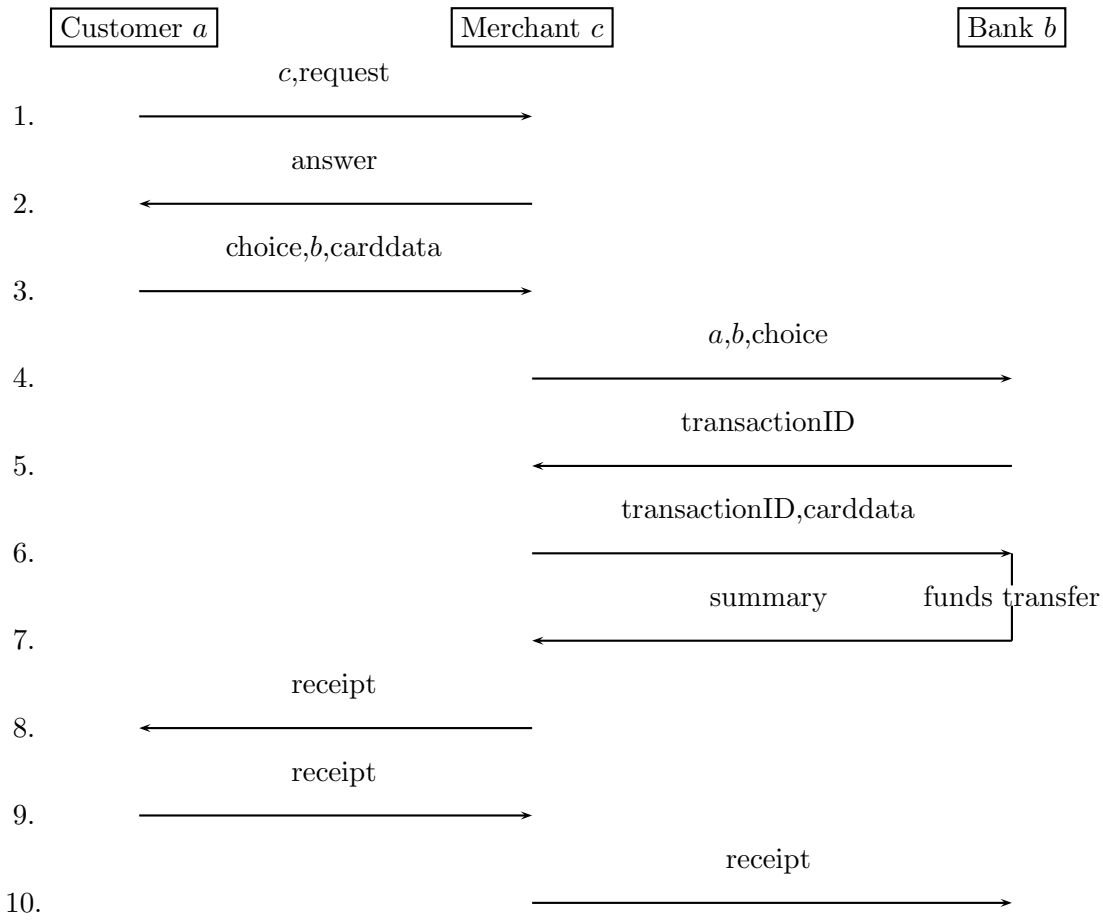


Figure 9.1: The e-commerce protocol

The complete protocol is as follows:

Message 1. $a \rightarrow c \ : \ \{\{c\}_{SK(a)}, request\}_{PK(c)}$
Message 2. $c \rightarrow a \ : \ \{request, answer\}_{PK(a)}$
Message 3. $a \rightarrow c \ : \ \{answer, choice, \{b\}_{SK(a)}, \{\{carddata\}_{PK(b)}\}_{SK(a)}\}_{PK(c)}$
Message 4. $c \rightarrow b \ : \ \{a, \{b\}_{SK(c)}, price\}_{PK(b)}$
Message 5. $b \rightarrow c \ : \ \{price, transactionID\}_{PK(c)}$
Message 6. $c \rightarrow b \ : \ \{transactionID, \{\{carddata\}_{PK(b)}\}_{SK(a)}\}_{PK(b)}$
Message 7. $b \rightarrow c \ : \ \{\{c\}_{SK(b)}, \{\{carddata\}_{PK(a)}\}_{SK(b)}, summary\}_{PK(c)}$
Message 8. $c \rightarrow a \ : \ \{\{a\}_{SK(c)}, \{\{carddata\}_{PK(a)}\}_{SK(b)}, receipt\}_{PK(a)}$
Message 9. $a \rightarrow c \ : \ \{receipt, acknowledgment\}_{PK(c)}$
Message 10. $c \rightarrow b \ : \ \{\{b\}_{SK(c)}, acknowledgment, end\}_{PK(b)}$

The design and verification of this protocol is out of the scope of this dissertation. This protocol was checked using Casper with the script in Appendix C.

The goal of the first three messages is to have a and c mutually authenticated and agreeing on the goods to buy and their price: a sends a request to c concerning goods of interest, c replies with a list of available goods and their prices, and a then chooses which good he wants to buy and sends his card data to b.

The goal of messages 4 to 6 is to have c and b mutually authenticated and agree on the fund transaction: firstly c sends a's identity to b as well as the goods that were ordered by a. Then b sends back the order and an identification number (transactionID). Finally c answers with the transactionID (achieving authentication between c and b) and the card data that it received in message 3. At that point, the bank can securely transfer the funds to the merchant.

The goal of messages 7 to 10 is the establishment of evidences of the transaction. Firstly b sends to c a summary of the transaction. Then c sends a receipt of the transaction to a, who answers with the receipt so that c can be sure that a has received the receipt. Finally c sends the receipt to b, concluding the transaction.

9.2 Implementation with COSP-J

In order to implement the e-commerce protocol, we have to build the input script for COSP-J, compile it with COSP-J, modify the output code if necessary, compile the output code and finally test it. We describe each of these phases in turn.

9.2.1 Building the input script

The COSP-J input file is built from the Casper code presented in Appendix C. We will consider all the sections of the COSP-J input script in turn.

The "Free variables" section

The Casper script in Appendix C uses many types for the different data transferred during the protocol (for example REQUEST, ANSWER and CHOICE). Here we will simulate all these types by the single type Nonce. The "Free variables" section therefore becomes:

```
#Free variables
A, B, C : Agent
request, answer, choice, carddata, transactionID, summary, \
receipt, acknowledgment, end, price : Nonce
PK : Agent -> RSAPublicKey
SK : Agent -> RSASecretKey
InverseKeys = (PK, SK)
```

The "Protocol description" section

The "Protocol description" section of the COSP-J input file is based on the Casper input file with the following three modifications:

Firstly, Casper assumes that when c receives message 1, he knows the identity of the sender a. This is not the case in COSP-J, as we discussed in Section 3.1. Therefore, we need a to send its identity in message 1.

Secondly, in message 3, the user a sends the identity of his bank b to the merchant c. However, c needs to be able to contact b and therefore we also need a to send the address of his bank (as described in Section 3.1). So we add $addr(b)$ in this message.

Finally, Casper assumes that when b receives message 4, he knows the identity of the sender c. This is not the case in COSP-J as we discussed in Section 3.1. Therefore we need c to send its identity along with this message.

Once we apply these modifications, we get the following COSP-J "Protocol description" section:

```
#Protocol description
-- AUTHENTICATION A-C AND REQUEST
1.  A -> C : A,{{C}{SK(A)}, request}{PK(C)}
2.  C -> A : {request, answer}{PK(A)}
3.  A -> C : {answer, choice, {B,addr(B)}{SK(A)}, \
{{carddata}{PK(B)}}{SK(A)}%c}{PK(C)}
-- AUTHENTICATION C-B AND TRANSACTION
4.  C -> B : C,{A, {B}{SK(C)}, price}{PK(B)}
5.  B -> C : {price, transactionID}{PK(C)}
```

```
6.  C -> B : {transactionID, \
c%{{carddata}{PK(B)}}{SK(A)}}{PK(B)}
-- ### B transfers funds ###
-- EVIDENCE OF THE TRANSACTION
7.  B -> C : {{C}{SK(B)}, \
{{carddata}{PK(A)}}{SK(B)}%e, summary}{PK(C)}
8.  C -> A : {{A}{SK(C)}, \
e%{{carddata}{PK(A)}}{SK(B)}, receipt}{PK(A)}
9.  A -> C : {receipt, acknowledgment}{PK(C)}
10. C -> B : {{B}{SK(C)}, acknowledgment, end}{PK(B)}
11. A ->   : choice,receipt
12. B ->   : A,C,price,summary
13. C ->   : A,choice,price,receipt
```

The "Processes" section

This section is based on the Casper input file, with the following two modifications.

Firstly, we have to make it clear that the nonces are generated during the run of the protocol thanks to the "generates" keyword.

Secondly, we do not use an environmental message 0 anymore to transmit the identity of the merchant to the customer anymore. This means that the customer needs to be instantiated with the identity of the merchant.

This section therefore becomes:

```
#Processes
CUSTOMER(A,B,C) knows PK, SK(A) generates carddata, \
choice, request, acknowledgment
BANK(B) knows PK, SK(B) generates transactionID, summary
MERCHANT(C) knows PK,SK(C) generates answer, price, receipt, end
```

The "Functions" section

The only functions that this script uses are PK and SK in order to get the public and secret key of an agent respectively. In our implementation, these keys will be stored in a KeyStore. Therefore, we specify that PK and SK need to get the keys from the KeyStore as follows:

```
#Functions
PK = myKeyStore.PK
SK = myKeyStore.SK
```

The "External" section

In this section, we define the only external object that we use in this protocol, namely the KeyStore "myKeyStore" that we use in the "Functions" section.

The "External" section of our COSP-J input script is therefore:

```
#External
myKeyStore : KeyStore
```

This completes the COSP-J input file. We can compile it with our COSP-J compiler to generate the following three java files: `CUSTOMER.java`, `MERCHANT.java` and `BANK.java`.

9.2.2 Modifying and compiling the output code

It is sometime useful to apply a few modifications to the java files before compiling them. For example, we may want the bank to transfer the funds between the receipt of message 6 and the sending of message 7. Or we may want the bank to check that the customer has enough money in his account to perform the transaction.

Another way to do this is to use the angle brackets notation (as described in Section 6.4) but this feature is limited to making a few assignments. Depending on the number of actions to be performed, we will either choose the angle brackets notation or modify the output of COSP-J. These modifications are not really part of the security protocol so we will not discuss them further in this dissertation, but they are another motivation for making the output of COSP-J as accessible as possible.

Once the modifications of the java files are completed, we compile all three java files `CUSTOMER.java`, `MERCHANT.java` and `BANK.java`.

9.2.3 Running the e-commerce protocol

We are now able to run the implementation of the protocol we have created.

The following script generates the trace for the e-commerce protocol:

```
(sleep 10 ; java protocols/CUSTOMER keystoreA *** \
localhost localhost A B C) &
(sleep 5 ; java protocols/MERCHANT keystoreC *** C) &
java protocols/BANK keystoreB *** B
```

The first line runs an instance of the customer class. Its parameters are the KeyStore where are keys are stored, the password used to access it, the DNS addresses of the merchant and the bank, and the names of the customer, bank and customer. The file keystoreA contains the secret key of agent A and the public keys of agents A, B and C.

The second line runs an instance of the merchant class. Its parameters are the KeyStore where are keys are stored, the password used to access it, and the name of the merchant. The file keystoreC contains the secret key of agent C and the public keys of agents A, B and C.

The last line runs an instance of the bank class. Its parameters are the KeyStore where are keys are stored, the password used to access it, and the name of the bank. The file keystoreB contains the secret key of agent B and the public keys of agents A, B and C.

The complete trace of the e-commerce protocol generated by this script can be found in Appendix D. In this trace, the type-flaw prevention system described in Chapter 7 is also active.

In addition to this case study, we successfully compiled many other protocols into COSP-J, as summarized in Chapter 10. Their input scripts can be found in the main directory of the COSP-J project. They all use the "spl" file extension. All the output codes generated during compilation of these security protocols are stored in the `java/protocols` directory of COSP-J.

The next chapter presents the conclusions of our work, together with a discussion of future work.

Chapter 10

Conclusion

10.1 Summary

In this thesis, we have presented COSP-J, a compiler for security protocols into Java implementation. The following three principles were followed when developing COSP-J:

Firstly, we designed the input script of COSP-J upon that of Casper. This lets users easily use COSP-J with Casper. Chapters 3 and 6 presented the design of the input file for COSP-J. The use of Casper and COSP-J together makes it very easy to firstly verify the validity of a security protocol with Casper and then generate a corresponding java implementation with COSP-J. The many examples studied in this thesis, and especially the e-commerce protocol studied in Chapter 9, has demonstrated how successful this approach is. The Casper input file structure was designed to make it easy to verify a very wide scope of protocols. As COSP-J is based on the same input file structure, it is able to compile many different security protocols. The list of protocols successfully implemented by COSP-J is as follows: Andrew Secure RPC, CCITT X.509, Denning-Sacco shared key, Diffie Helman key exchange algorithm, Kao Chow Authentication, Needham-Schroeder Public Key, Neumann Stubblebine, Otway Rees, Station-to-Station, TMN, Yahalom. The description of all these protocols can be found in [CJ97]. The COSP-J input files for these protocols are in the main directory of the COSP-J project, under the `.spl` extension.

Secondly, we wanted the Java code generated by COSP-J to be accessible, even for someone who does not know the classes described in Section 4.3. We believe this aim was reached: a protocol such as the Needham-Schroeder Public Key protocol generates around 100 lines of fully commented code for each agent, and the names of the functions used are very explicit. The Application Programming Interface (API) for the package "core" that is located in directory `java/core/doc` is also a precious help for anyone who wants to make the most of COSP-J: it describes all the classes used by the

implementations generated by COSP-J.

Thirdly, our aim was to make the implementations generated by COSP-J as secure as possible. The fact that these implementations are written in Java is a first step towards achieving this, as we showed that Java is a secure language in Section 4.1. As Java is an Object-Oriented Language, we used a very clean object oriented model (cf. Section 4.3) that minimizes human errors. The systems preventing type-flaw attacks (cf. Chapter 7) and multi-protocol attacks (cf. Chapter 8) also improves the security of our implementations. This is especially useful as Casper is not good at detecting type-flaw attacks and unable to find multi-protocol attacks.

10.2 Comparison with other existing tools

There are few compilers for generating implementations of security protocols. The only one that we are aware of was created in the Computer Science Division of the University of California, Berkeley. It is part of the AGVI toolkit (Automatic Generation, Verification, and Implementation of Security Protocols) [SPP01]. This toolkit is made of three tools. APG (Automatic Protocol Generator) takes a system specification and the security requirements, and automatically designs a security protocol. APV (Automatic Protocol Analyzer) is an analyzer of security protocols: it can both produce a proof (when a protocol is correct), and generate a counter-example (when a protocol is flawed). ACG (Automatic Code Generator) is an automatic compiler that translates high level specifications of security protocols into low level implementations (Java source code).

It is not possible to directly compare ACG and COSP-J, because the first one is made to be used along with APG and APV, whereas COSP-J is designed to be used together with Casper: the leading principles that were followed when creating COSP-J (summarized in the previous section) do not apply to ACG. A comparison of Casper with other automatic checking systems is beyond the scope of this thesis, but the CSP/Casper approach is one of the most successful concerning protocol analysis (see [DNL99] for further details). This makes COSP-J (which is the only security protocol compiler into implementations based on Casper) a very useful tool.

Although COSP-J was designed to work with Casper, the input script is very accessible and can be used with other verification methods as well.

10.3 Future Work

One of the aims of our work was to make COSP-J able to compile as many security protocols as possible as discussed in Section 10.1. But there exists a very wide variety of security protocols. One possible direction for further work would be to try to compile different security protocols and adapt

COSP-J if needed. For example, we could try to use COSP-J to implement the CAM [OR01] protocol which is a protocol used by mobile computers to inform their peers when their network address has changed, and the SK3 protocol [SR96] which deals with symmetric key distribution using Smart Cards. More examples can be found in [CJ97].

Another interesting avenue for future research that builds upon the work presented in this thesis would be to try and create a formal proof of the correctness of COSP-J. The aim would be to formally prove that if Casper did not detect an error in a given protocol, its implementation with COSP-J will be free of error. In order to do so, we need to prove that the assumptions made by Casper are true in COSP-J. The proof may involve a few changes in the COSP-J code. For example, Casper assumes that if a message is encrypted with a key k, it cannot be decrypted with anything other than k. This is not always true in COSP-J: if a nonce na is encrypted with a key $k1$, it may be possible (depending on the encryption and decryption algorithms that are used) to decrypt it with a key $k2$, even if the key $k2$ is not the inverse of key $k1$. Of course the result would not be the value of na, but another random amount of data. It would be surprising that this would lead to an actual attack; however, a formal proof is needed to ensure that such an assumption made by Casper remains true in COSP-J. The proof will also probably need to assume that the Java Virtual Machine (JVM) on which the protocol is run and the Java classes used are error-free. See [Oak01] for a complete discussion on this topic.

Bibliography

[AN96] Martín Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, 1996. Also in Proceedings of the 1994 IEEE Symposium on Security and Privacy, and Digital Equipment Corporation Systems Research Center, Research Report 125.

[BAN89] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989. A preliminary version appeared as Digital Equipment Corporation Systems Research Center report No. 39, 1989.

[Bar00] David Barnes. *Object-oriented Programming with Java*. Prentice Hall, 2000.

[BMF01] Simon Bennett, Steve McRobb, and Ray Farmer. *Object-oriented Systems Analysis and Design Using UML 2/e*. McGraw-Hill Education, 2001.

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

[DH76] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, 1976.

[DNL99] Ben Donovan, Paul Norris, and Gavin Lowe. Analyzing a library of security protocols using Casper and FDR. In *Proceedings of the Workshop on Formal Methods and Security Protocols*, 1999.

[DvOW92] Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2:107–125, 1992.

[Gol99] Dieter Gollmann. *Computer security*. John Wiley & Sons, 1999.

[GT00] Joshua D. Guttman and F. Javier Thayer Fábrega. Protocol independence through disjoint encryption. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 24–34, 2000.

[HL99] Mei Lin Hui and Gavin Lowe. Safe simplifying transformations for security protocols. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, 1999.

[HLS00] James Heather, Gavin Lowe, and Steve Schneider. How to prevent type flaw attacks on security protocols. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 255–268, 2000.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Knu98] Jonathan Knudsen. *Java Cryptography*. O'Reilly, 1998.

[Low96a] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of TACAS*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer Verlag, 1996. Also in *Software—Concepts and Tools*, 17:93–102, 1996.

[Low96b] Gavin Lowe. Some new attacks upon security protocols. In *Proceedings 9th IEEE Computer Security Foundations Workshop*, pages 162–169, 1996.

[Low97] Gavin Lowe. Casper: A compiler for the analysis of security protocols. In *Proceedings of 10th IEEE Computer Security Foundations Workshop*, pages 18–30, 1997.

[Men02] Antoine Menard. An e-commerce protocol. Master's thesis, Oxford University Computing Laboratory, 2002.

[NS78] Roger Needham and Michael Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[Oak01] Scott Oaks. *Java Security (2nd Edition)*. O'Reilly & Associates, May 2001.

[OR01] Greg O'Shea and Michael Roe. Child-proof authentication for MIPv6 (CAM). *Computer Communications Review*, April 2001.

[Pey03] Simon Peyton Jones. *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, 2003.

[Ros94] A. W. Roscoe. Model-checking CSP. In *A Classical Mind, Essays in Honour of C. A. R. Hoare*. Prentice-Hall, 1994.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[SPP01] Dawn Song, Adrian Perrig, and Doantam Phan. AGVI — automatic generation, verification, and implementation of security protocols. In *13th Conference on Computer Aided Verification (CAV)*, 2001.

[SR96] Victor Shoup and Avi Rubin. Session key distribution using smart cards. In Springer-Verlag, editor, *Advances in Cryptology, EUROCRYPT'96*, volume 1070, 1996.

[TMN90] Makoto Tatebayashi, Natsume Matsuzaki, and David B. Newman, Jr. Key distribution protocol for digital mobile communication systems. In *Advances in Cryptology: Proceedings of Crypto '89*, volume 435 of *Lecture Notes in Computer Science*, pages 324–333. Springer-Verlag, 1990.

[WL94] Thomas Y. C. Woo and Simon S. Lam. A lesson on authentication protocol design. *Operating Systems Review*, 28(3):24–37, 1994.

[Zal01] Michal Zalewski. Remote vulnerability in SSH daemon crc32 compensation attack detector. Available from `razor.bindview.com/publish/advisories/adv_ssh1crc.html`, February 2001.

Appendix A

Script generating keystore1 and keystore2

```
# This script creates files keystore1 and keystore2

# First we delete keystore1 and keystore2
rm keystore1 2>/dev/null
rm keystore2 2>/dev/null

# Then we create keystore1 with A's public and secret key
keytool -genkey -alias Alice -keyalg RSA -keysize 1024 \
-keystore keystore1 -dname "CN=Alice" -storepass duitama -keypass duitama

# Then we create keystore2 with B's public and secret key
keytool -genkey -alias Bob -keyalg RSA -keysize 1024 \
-keystore keystore2 -dname "CN=Bob" -storepass duitama -keypass duitama

# Now we export the certificate containing A's public key from keystore1
keytool -export -keystore keystore1 -alias Alice -file output \
-storepass duitama 2>/dev/null

# And we import it in keystore2
echo Yes | keytool -import -alias Alice -keystore keystore2 \
-file output -storepass duitama >/dev/null 2>/dev/null

# Finally we export the certificate containing B's public key from keystore2
keytool -export -keystore keystore2 -alias Bob -file output2 \
-storepass duitama 2>/dev/null

# And we import it in keystore1
echo Yes | keytool -import -alias Bob -keystore keystore1 \
```

```
-file output2 -storepass duitama >/dev/null 2>/dev/null

# Clean tempory files
rm output
rm output2

# Display keystore1
echo CONTENT OF KEYSTORE1:
keytool -keystore keystore1 -list -storepass duitama

# Display keystore2
echo -e \\n\\n\\nCONTENT OF KEYSTORE2:
keytool -keystore keystore2 -list -storepass duitama
```

Appendix B

The Station-to-Station protocol

```
#Free variables
A, B : Agent
g, m, x, y, gEx, gEy, k : BigInteger
PK : Agent -> RSAPublicKey
SK : Agent -> RSASecretKey
InverseKeys = (PK, SK),(k,k)

#Processes
INITSTS(A,B) knows PK,SK(A) generates g,m,x
RESPSTS(B) knows PK,SK(B) generates y

#Protocol description
<gEx := g.modPow(x,m)>
1. A -> B : g,m,gEx
<k := gEx.modPow(y,m) ; gEy := g.modPow(y,m)>
2. B -> A : gEy
<k := gEy.modPow(x,m)>
2b.A -> B : A
2c.B -> A : {{gEy,gEx}{SK(B)}}{k}
3. A -> B : {{gEx,gEy}{SK(A)}}{k}
4. A ->   : B,k
5. B ->   : A,k

#External
myKeyStore : KeyStore
#Functions
PK = myKeyStore.PK
SK = myKeyStore.SK
```

Appendix C

Casper script: E-commerce protocol

```
#Free variables
A, B, C : Agent
request : REQUEST
answer : ANSWER
choice : CHOICE
carddata : CARDDATA
transactionID : TRANSACTIONID
summary : SUMMARY
receipt : RECEIPT
acknowledgement : ACKNOWLEDGEMENT
end : END
price : PRICE
PK : Agent -> PublicKey
SK : Agent -> SecretKey
InverseKeys = (PK, SK)

#Processes
ANTOINE(A,B,carddata, choice, request, acknowledgement) knows PK, SK(A)
BANK(B,transactionID, summary) knows PK, SK(B)
COMPANY(C,answer, price, receipt, end) knows PK,SK(C)

#Protocol description
0.    -> A : C
1.  A -> C : {{C}{SK(A)}, request}{PK(C)}
2.  C -> A : {request, answer}{PK(A)}
3.  A -> C : {answer, choice, {B}{SK(A)}, \
{{carddata}{PK(B)}}{SK(A)}%c}{PK(C)}
4.  C -> B : {A, {B}{SK(C)}, price}{PK(B)}
```

```
5.  B -> C : {price, transactionID}{PK(C)}
6.  C -> B : {transactionID, c%{{carddata}{PK(B)}}{SK(A)}}{PK(B)}
7.  B -> C : {{C}{SK(B)}, {{carddata}{PK(A)}}{SK(B)}%e, summary}{PK(C)}
8.  C -> A : {{A}{SK(C)}, e%{{carddata}{PK(A)}}{SK(B)}, receipt}{PK(A)}
9.  A -> C : {receipt, acknowledgement}{PK(C)}
10. C -> B : {{B}{SK(C)}, acknowledgement, end}{PK(B)}

#Functions
symbolic PK,SK

#Specification
Secret(A, request, [C])
Secret(A, choice, [C])
Secret(A, carddata, [B])
Secret(B, transactionID, [C])
Secret(B, summary, [C])
Secret(C, answer, [A])
Secret(C, receipt, [A])
Secret(A, acknowledgement, [C,B])
Secret(C, price, [B])
Secret(C, end, [B])
Agreement(A,C,[request,choice,answer,receipt,acknowledgement])
Agreement(C,A,[request,answer,receipt])
Agreement(C,B,[price,transactionID,acknowledgement,end])
Agreement(B,C,[price,transactionID,summary])
Agreement(B,A,[carddata])

#System
ANTOINE(Antoine, Bank, Carddata, Choice, Request, Acknowledgement)
BANK(Bank, TransactionID, Summary)
COMPANY(Company, Answer, Price, Receipt, End)

#Intruder Information
Intruder = Mallory
IntruderKnowledge = {Antoine, Bank, Company, Mallory, Answerm, Choicem, \
Carddatam, TransactionIDm, Summarym, Receiptm, Acknowledgementm, Endm, \
Pricem, PK, SK(Mallory)}

#Actual variables
Antoine, Bank, Company, Mallory : Agent
Request : REQUEST
Answer : ANSWER
Choice : CHOICE
Carddata : CARDDATA
```

```
TransactionID : TRANSACTIONID
Summary : SUMMARY
Receipt : RECEIPT
Acknowledgement : ACKNOWLEDGEMENT
End : END
Price : PRICE
Requestm : REQUEST
Answerm : ANSWER
Choicem : CHOICE
Carddatam : CARDDATA
TransactionIDm : TRANSACTIONID
Summarym : SUMMARY
Receiptm : RECEIPT
Acknowledgementm : ACKNOWLEDGEMENT
Endm : END
Pricem : PRICE
```

Appendix D

Trace of the e-commerce protocol

```
ANTOINE: creating nonce carddata
ANTOINE: creating nonce choice
ANTOINE: creating nonce request
ANTOINE: creating nonce acknowledgement
ANTOINE: Building message m1
ANTOINE: Encrypting m0
ANTOINE: Encrypting m0
ANTOINE: Sending message m1
COMPANY: creating nonce answer
ANTOINE: waiting for m2
COMPANY: creating nonce price
COMPANY: creating nonce receipt
COMPANY: creating nonce end
COMPANY: waiting for m1
COMPANY: Tag check successful for APSA)N)
COMPANY: Decrypting m0
COMPANY: Tag check successful for SA)N
COMPANY: Decrypting m0
COMPANY: Tag check successful for A
COMPANY: Building message m2
COMPANY: Encrypting m2
COMPANY: Sending message m2
ANTOINE: Tag check successful for PNN)
ANTOINE: Decrypting m2
COMPANY: waiting for m3
ANTOINE: Tag check successful for NN
ANTOINE: Building message m3
ANTOINE: Encrypting m0
```

```
ANTOINE: Encrypting m0
ANTOINE: Encrypting m0
ANTOINE: Encrypting m3
ANTOINE: Sending message m3
COMPANY: Tag check successful for PNNSAD)SPN)))
COMPANY: Decrypting m3
ANTOINE: waiting for m8
COMPANY: Tag check successful for NNSAD)SPN))
COMPANY: Decrypting m0
COMPANY: Tag check successful for AD
COMPANY: Building message m4
COMPANY: Encrypting m0
COMPANY: Encrypting m0
COMPANY: Sending message m4
BANK: creating nonce transactionID
COMPANY: waiting for m5
BANK: creating nonce summary
BANK: waiting for m4
BANK: Tag check successful for APASA)N)
BANK: Decrypting m0
BANK: Tag check successful for ASA)N
BANK: Decrypting m0
BANK: Tag check successful for A
BANK: Building message m5
BANK: Encrypting m5
BANK: Sending message m5
COMPANY: Tag check successful for PNN)
COMPANY: Decrypting m5
BANK: waiting for m6
COMPANY: Tag check successful for NN
COMPANY: Building message m6
COMPANY: Encrypting m6
COMPANY: Sending message m6
BANK: Tag check successful for PNSPN)))
BANK: Decrypting m6
COMPANY: waiting for m7
BANK: Tag check successful for NSPN))
BANK: Decrypting m0
BANK: Tag check successful for PN)
BANK: Decrypting m0
BANK: Tag check successful for N
BANK: Building message m7
BANK: Encrypting m0
BANK: Encrypting m0
```

```
BANK: Encrypting m0
BANK: Encrypting m7
BANK: Sending message m7
COMPANY: Tag check successful for PSA)SPN))N)
COMPANY: Decrypting m7
BANK: waiting for m10
COMPANY: Tag check successful for SA)SPN))N
COMPANY: Decrypting m0
COMPANY: Tag check successful for A
COMPANY: Building message m8
COMPANY: Encrypting m0
COMPANY: Encrypting m8
COMPANY: Sending message m8
ANTOINE: Tag check successful for PSA)SPN))N)
ANTOINE: Decrypting m8
COMPANY: waiting for m9
ANTOINE: Tag check successful for SA)SPN))N
ANTOINE: Decrypting m0
ANTOINE: Tag check successful for A
ANTOINE: Decrypting m0
ANTOINE: Tag check successful for PN)
ANTOINE: Decrypting m0
ANTOINE: Tag check successful for N
ANTOINE: Building message m9
ANTOINE: Encrypting m9
ANTOINE: Sending message m9
COMPANY: Tag check successful for PNN)
COMPANY: Decrypting m9
COMPANY: Tag check successful for NN
COMPANY: Building message m10
COMPANY: Encrypting m0
COMPANY: Encrypting m10
COMPANY: Sending message m10
BANK: Tag check successful for PSA)NN)
BANK: Decrypting m10
COMPANY: A
BANK: Tag check successful for SA)NN
BANK: Decrypting m0
BANK: Tag check successful for A
BANK: A
BANK: C
```

Appendix E

Casper script: protocol in Section 6.2

```
#Free variables
f : HashFunction
A, B : Agent
na : Nonce
PK : Agent -> PublicKey
SK : Agent -> SecretKey
InverseKeys = (PK, SK)

#Processes
INITIATOR(A,na) knows PK, SK(A), f
RESPONDER(B) knows PK, SK(B), f

#Protocol description
0.    -> A : B
[A!=B]
1.  A -> B : {A,B,na}{PK(B)},{f({A,B,na}{PK(B)})}{SK(A)}
2.  B -> A : {A,B,na}{PK(A)}

#Specification
Secret(A, na, [B])
Agreement(A,B,[na])
Agreement(B,A,[na])

#Actual variables
Alice, Bob, Mallory : Agent
Na , Nm : Nonce

#Functions
```

```
symbolic PK, SK

#System
INITIATOR(Alice, Na)
RESPONDER(Bob)

#Intruder Information
Intruder = Mallory
IntruderKnowledge = {Alice, Bob, Mallory, Nm, PK, SK(Mallory)}
```