Analysis and Reconstruction of Attacks on Authentication Protocols

Master Thesis by
Nikolaj Hjelm Kaplan

Institute of Informatics and Mathematical Modelling
The Technical University of Denmark
IMM-THESIS-2004-36
July 1, 2004

Abstract

In this project I use and develop methods for analyzing and reconstructing attacks on authentication protocols. I will combine two approaches to analysis of authentication protocols: Static analysis and model checking. The static analysis gives an over-approximation of the breaches of a protocol. My task is to choose a potential breach of a certain protocol, and if possible reconstruct the trace of events leading to this breach using techniques from model checking. The model checking will be aided by additional results from the analysis. The result of my work is a tool which makes it possible to efficiently find and reconstruct a certain breach of a given authentication protocol.

Keywords: *Authentication Protocols, Security protocols, Static Analysis, Control flow analysis, Model Checking,* LySa*,* samc*.*

Resumé

I dette projekt bruger og udvikler jeg metoder til at analysere og rekonstruere angreb på autentifikationsprotokoller. Jeg kombinerer to tilgange til analyse af autentifikationsprotokoller: Statisk analyse og model checking. Statisk analyse giver en overapproksimation til fejl i en protokol. Min opgave er at vælge en potentiel fejl i en given protokol og hvis muligt rekonstruere den sekvens af kommunikation, der forårsager denne fejl. Til denne rekonstruktion benyttes model checking teknikker. Resultatet af mit arbejde er et værktøj, der gør det muligt effektivt at finde og rekonstruere en fejl i en given autentifikationsprotokol.

Nøgleord: *Autentifikationsprotokoller, Sikkerhedsprotokoller, Statisk Analyse, Control flow analyse, Model Checking,* LySa, samc.

Preface

This report describes and documents the M.Sc. thesis project by Nikolaj Hjelm Kaplan. The project corresponds to 35 ECTS points and was carried out in the period from October 2003 to July 2004. The project was carried out at the Technical University of Denmark, Department of Informatics and Mathematical Modeling under the supervision of Professor Hanne Riis Nielson.

I would like to thank a number of people who have helped me in one way or another during my work. First of all, thanks to Hanne Riis Nielson. It has been amazing to have a supervisor with such an interest in my project. You have had a great patience and willingness to discuss my project on any level. Mikael Buchholtz, co-author to [1], has also been a great help in many phases of my work. DTU is a great place to do a project! Furthermore, I would like to thank Sebastian Mödersheim, co-author to [8], for taking time to write very good answers to each of my mails. On the personal side I would like to thank my family. My wife Anne Kaplan has been amazing and very supportive and my children, Silas and Bastian, has made some beautiful drawings on many of the draft versions of this report. The grand parents of my children have as always been a great help as well.

Nikolaj Hjelm Kaplan
Lyngby, July 1, 2004

Contents

Chapter 1

Introduction

1.1 Authentication Protocols and Insecure Networks

The purpose of an authentication protocol is to enable a number of principals to communicate securely over an insecure network. By secure communication we mean that the principals in general have to be able to establish some common knowledge, which enables some form of authentication and secrecy between the principals. Encryption of messages is used. In this project we only consider symmetric encryption. We do not consider any details as to how encryption is performed.

When considering an insecure network we assume the existence of a malicious intruder, who has the complete control over the network. The Dolev-Yao threat model describes the capabilities of the intruder as follows:

- The intruder can intercept any message.
- The intruder can decrypt encrypted messages if and only if he knows the encryption key. He cannot guess a key.
- The intruder can construct new messages and new encryptions.
- The intruder can send any message which he has constructed or previously intercepted.
- The intruder may be a legitimate principal of the system and has initial knowledge corresponding to a legitimate principal.

The sender of a message cannot be determined simply by looking at the message, and since the intruder can intercept and redirect any message, we can in general not be certain about the actual origin or destination of a given message.

An authentication protocol defines a sequence of communication steps between a number of principals, which are carried out in order to ensure some sort of security goal. The running example of this report will be a variant of the protocol "Wide Mouthed Frog". It is a very simple protocol which is intended to enable two principals, A and B, to establish a shared key via a trusted server, S. The protocol narration is as follows:

$$\begin{array}{l} A \rightarrow S : A, B, \{K\}_{K_A} \\ S \rightarrow B : A, \{K\}_{K_B} \\ A \rightarrow B : \{Msg\}_K \end{array}$$

$\{M\}_K$ denotes the symmetric encryption of message M with the key K. It is assumed that each of the principals share a secret key with the trusted server, S. Namely K_A and K_B. In step one the principal A initiates the protocol by sending the message $A, B, \{K\}_{K_A}$ to S. The message states that A wants to initiate a communication with B using the key K. Since K_A is a shared key between S and A, S can decrypt the term $\{K\}_{K_A}$ and retrieve the key K. The function of S is to inform B that A wants to communicate using the key K. This time K is encrypted with K_B, the shared key of S and B. B recieves the encryption $\{K\}_{K_B}$ and retrieves the key K. Finally A can send a message to B encrypted with the key K, which is now known to both A and B.

Validating protocols. It is amazingly difficult to validate an authentication protocol and many different approaches have been investigated through the last 30 years. The main problem is that the possibilities for various interactions by the intruder are infinite. There are examples of simple protocols that have been used for 10 or 15 years before a flaw was discovered.

A first look at the running example protocol may give the impression that it is working as it is supposed to, but consider the attack specified by the following narration:

$$\begin{array}{lll} A & \rightarrow I(S) & : A, B, \{K\}_{K_A} \\ I(A) & \rightarrow S & : A, I, \{K\}_{K_A} \\ S & \rightarrow I(B) & : A, \{K\}_{K_I} \\ A & \rightarrow I(B) & : \{Msg\}_K \end{array}$$

I denotes the intruder. The notation $I(n)$ expresses that I takes the role of principal n.

The intruder, I, intercepts the first message from A and tricks the server into believing that A actually wants to communicate with I. The consequence is that I obtains the session key K and finally is able to decrypt the message, *Msg*, intended for B. We use the fact that the intruder is a legitimate principal in the system and is therefor accepted by S as having the name I and the key K_I. This is just one of the possible attacks on the running example protocol.

1.2 Overview of Strategy and Concepts

In this project I have developed a tool, (called the SAMC tool[1]), which performs the following tasks:

- A static analysis of a protocol is carried out. This analysis might reveal potential breaches in the protocol.
- A potential breach is chosen by the user of the tool.
- Model Checking techniques are used to find a trace of events if possible leading to this specific breach.

By trace of events I mean communications between an intruder and the legitimate principals of the protocol. This trace of events can hence be represented by a traditional narration illustrating the interaction between the principals of the protocol and the intruder.

The static analysis performs a safe over-approximation of the potential breaches of a protocol. This means that we might encounter a false positive. In this case it is of cause not possible to reconstruct a trace of events.

The tool has a user interface, which enables a user to choose a certain potential breach revealed by the static analysis and carries out the model checking routine on this particular breach. See Section 5 for an overview of the tool and a user's guide.

The static analysis enables us to find two different types of breaches: Breach of authentication and breach of secrecy.

A breach of authentication occurs when the encryption and decryption of a given message can take place in a way which was not intended in the specification of the protocol. An example could be that the intruder, I, decrypts a message which was intended for principal B, or that a principal B decrypts a message which was encrypted by the intruder, but should have originated from a principal A.

A breach of secrecy is in most cases, (but not always), related to an authentication breach. A secrecy breach occurs when a message which should have been kept secret can end up in the knowledge of the intruder.

Motivation. The interesting aspect of this project is the combination of the two techniques: Static analysis and model checking. My motivation for this project has been my work with a tool for static analysis of authentication protocols: The LySa tool. The tool is developed by a group[2] of people from DTU and from Università di Pisa [1]. The LySa tool successfully reports all known flaws of a range of protocols from the famous Clark-Jakob library [11]. But the

[1] The tool has been given the name SAMC tool. SA for Static Analysis and MC for Model Checking.

[2] Università di Pisa: C. Bodei, P. Degano, DTU: M. Buchholtz, F. Nielson, H. Riis Nielson.

static analysis does not offer any information as to the exact nature of a given flaw. The techniques of model checking on the other hand might require more computational power and might not terminate on a given problem, but has the ability to give a more precise result as to the trace of events leading to a certain flaw in a protocol.

Besides from applying the two different techniques separately I also investigate in the possibilities of using results from static analysis to aid the model checking routines. I.e. I experiment with mixing the two techniques instead of just combining the results of each of the techniques applied to a given protocol.

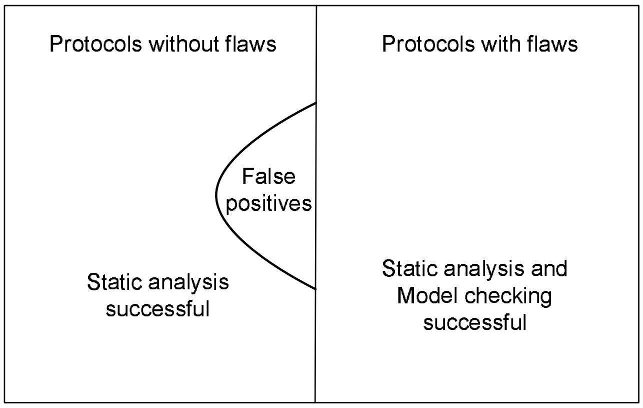

Figure 1.1: Model Checking and Static Analysis

Figure 1.1 shows the sets of protocols that can be dealt with by each of the two techniques. As we can see from the figure the static analysis can handle all protocols except a small set of flawless protocols where the static analysis returns a false positive due to the over-approximation performed. The size of this set depends on the precision of the analysis. The techniques of model checking can in general be successfully applied to protocols containing a flaw. This means that running the static analysis on protocols before considering model checking will remove a large part of the protocols that are flawless. This means that the problem of termination occurring when we use model checking on a flawless protocol is diminished. However it is not completely solved, since the static analysis still suffers from the false positives.

1.3 Prerequisites

The reader of this report should have some knowledge of the concepts of authentication protocols. It is also necessary to have some knowledge of the techniques of static control flow analysis as well as model checking. The reader should further more be familiar with process algebra and reduction semantics. Implementation of the techniques described in this project is done in Standard ML of New Jersey [6]. This report does not however require extensive knowledge in this area.

1.4 Structure of Report

The Sections 2, 3 and 4 deals with the three main concepts of this project: The specification of a protocol, static analysis and model checking.

Section 5 contains a description of some of the implementational details of SAMC tool. The section also contains a users guide.

Section 6 contains examples and results obtained by using the tool on various protocols.

Section 7 summarizes my work and discusses possible improvements and alternative approaches.

The source code is available on the web at the following address:
`http://www.student.dtu.dk/~s971701/masterthesis`

Appendix C contains instructions on how to install the tool.

Chapter 2

Specification of a Protocol

In this section I introduce the three ways of specifying a protocol used in this project. Because of the combination of the two techniques, we need two ways of representing a protocol: The *process algebra* LySa, which is used as the input to the static analysis, and the *graph representation*, which is used as a foundation for the model checking. The third way of specifying a protocol is by using the process algebra LySa$^{\mathrm{N}}$ that I have developed. LySa$^{\mathrm{N}}$ is a variant of LySa. It serves the purpose of tying the two other representations together. When a protocol is given as input to the SAMC tool it is specified in LySa$^{\mathrm{N}}$. This specification is transformed into a LySa specification and a graph representation respectively. LySa$^{\mathrm{N}}$ is a modified version of LySa, which in some ways limits the possible executions of a protocol and in other ways extend it.

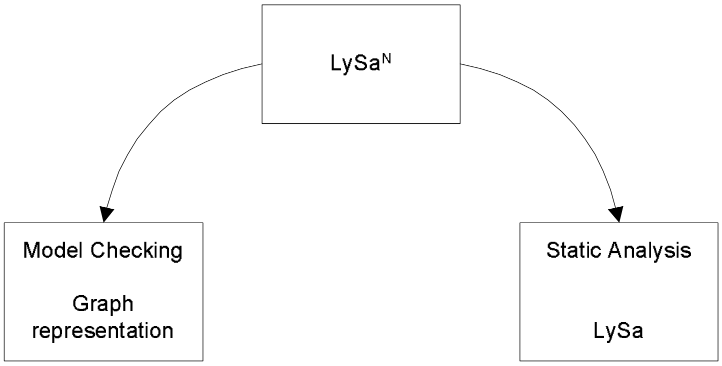

Figure 2.1: Specification of a protocol

2.1 LySa - The process algebra

The process algebra LySa [1] is used for specifying and analyzing authentication protocols. LySa makes it possible to explicitly specify many of the details of a protocol that are kept implicit in a protocol narration. In the following I will state the grammar and semantics of LySa and demonstrate the specification of a protocol by an example. In Section 3 I will explain the main concepts of the static analysis of a protocol specified in LySa. For further details on LySa and static analysis, see [1].

Syntax. LySa consists of terms and processes. Values correspond to terms without free variables. Values are used to code keys, nonces, messages etc. The syntax of terms E is as follows:

Terms

$$
\begin{array}{lll}
E ::= & n & \text{name } (n \in \mathcal{N}) \\
\mid & x & \text{variable } (x \in \mathcal{X}) \\
\mid & \{E_1, \cdots, E_k\}_{E_0} & \text{symmetric encryption } (k \geq 0)
\end{array}
$$

Where $\mathcal{N}$ and $\mathcal{X}$ denote sets of names and variables, respectively. Encryptions are tuples of terms $E_1, \cdots, E_k$ encrypted under a term E_0 representing an encryption key. We adopt an assumption of perfect cryptography.

The syntax of processes is specified by the following grammar:

Processes

$$
\begin{array}{lll}
P ::= & \mathsf{0} & \text{nil} \\
\mid & \langle E_1, \cdots, E_k \rangle.P & \text{output} \\
\mid & (E_1, \cdots, E_j; x_{j+1}, \cdots, x_k).P & \text{input (with matching)} \\
\mid & P_1 \mid P_2 & \text{parallel composition} \\
\mid & (\nu\, n)P & \text{restriction} \\
\mid & !\,P & \text{replication} \\
\mid & \mathsf{decrypt}\ E\ \mathsf{as}\ \{E_1, \cdots, E_j;\ x_{j+1}, \cdots, x_k\}_{E_0}\ \mathsf{in}\ P & \\
& & \text{symmetric decryption (with matching)}
\end{array}
$$

Pattern matching in LySa is restricted to patterns of the following simple form $(E_1, \cdots, E_j; x_{j+1}, \cdots, x_k)$, which is matched against a k-tuple of values $(E'_1, \cdots, E'_k)$. The matching succeeds when the first $1 \leq i \leq j$ values E'_i are pairwise equal to the values E_i, and the effect of the pattern match is to bind the remaining $k - j$ values to the variables $x_{j+1}, \cdots, x_k$. Syntactically, this is indicated by using a semicolon to separate the components where matching is performed from those where only binding takes place. For example, let

$$
\begin{array}{lll}
P & = & \mathsf{decrypt}\ \{y\}_K\ \mathsf{as}\ \{x;\ \}_K\ \mathsf{in}\ P' \text{ and} \\
Q & = & \mathsf{decrypt}\ \{y\}_K\ \mathsf{as}\ \{;\ x\}_K\ \mathsf{in}\ Q'
\end{array}
$$

While the decryption in P succeeds only if the value of x equals that of y, the decryption in Q always succeeds, binding the value of y to x in Q'.

An encryption term $\{t_1, \cdots, t_k\}_{t_0}$ can also be used for moddeling functions such as the successor function or a pair construction. This is done using a key, which is known to all principals of a protocol including the intruder. The expression $N + 1$ could be modelled as $\{N\}_{succ}$ in LySa.

As an example of a LySa specification of a protocol we take a look at the specification of the running example protocol, corresponding to the narration presented in Section 1.1.

$$
\begin{array}{ll}
& (\nu\, K_A)(\nu\, K_B) \\
& \quad !\,(\nu\, K)\langle A, B, \{K\}_{K_A}\rangle.(\nu\, Msg)\langle\{Msg\}_K\rangle.0 \\
| & \quad !\,(A, B; y_{Enc}).\mathsf{decrypt}\ y_{Enc}\ \mathsf{as}\ \{; y_K\}_{K_A}\ \mathsf{in}\ \langle\{y_K\}_{K_B}\rangle.0 \\
| & \quad !\,(; z_1).\mathsf{decrypt}\ z_1\ \mathsf{as}\ \{; z_K\}_{K_B}\ \mathsf{in}\ (; z_2).\mathsf{decrypt}\ z_2\ \mathsf{as}\ \{; z_{Msg}\}_{z_K}\ \mathsf{in}\ 0
\end{array}
$$

Note that small starting letters are used for variables and capital starting letters are used for names throughout this report.

The specification of the protocol is a basic instantiation, where we only consider three principals. One for each of the roles: Initiator (A), Server (S) and Responder (B). The first line specifies that the keys K_A and K_B are not known to the intruder. In fact the key K_A is only known to A and S, and the key K_B is only known to B and S, but this property can not be explicitly specified in LySa. However it can be seen from the use of the names in the specification of the process of each of the principals. The next three lines of the protocol specification corresponds to the process of each of the principals, A, S and B respectively. The restriction construction $(\nu\, K)$ indicates that K is a fresh key, initially only known to A. From the process of A, we see that A sends two messages, one to S and one to B. S receives a message from A and sends one to B. B receives two messages, one from S and one from A. We see the use of the decrypt construction in process S and B. S decrypts the first message from A in $\mathsf{decrypt}\ y_{Enc}\ \mathsf{as}\ \{; y_K\}_{K_A}\ \mathsf{in}\ \cdots$. The result is that y_K is bound to the value K. In the process of B the decrypt construction is first used to bind z_K to K, and afterwards to bind z_{Msg} to Msg.

Origin and destination annotations. Goals of authentication are described in LySa by adding annotations to encryption terms and decryption constructions. The annotations express the intended destination and the intended origin of the content of an encryption term. Origin and destination are specified using crypto points ℓ from some enumerable set $\mathcal{C}$. An encryption is represented by a LySa term of the form:

$$\{E_1, \cdots, E_k\}^{\ell}_{E_0}[\mathsf{dest}\ \mathcal{L}]$$

where ℓ defines the crypto point where the encryption is made and the assertion $[\mathsf{dest}\ \mathcal{L}]$ specifies the intended crypto points $\mathcal{L} \subseteq \mathcal{C}$ for decryption of the encrypted value.

Similarly, the construction representing the decryption of a term is of the form:

$$\mathsf{decrypt}\ E\ \mathsf{as}\ \{E'_1, \cdots, E'_j; x_{j+1}, \cdots, x_k\}^{\ell}_{E'_0}\ [\mathsf{orig}\ \mathcal{L}]\ \mathsf{in}\ P$$

where $[\mathsf{orig}\ \mathcal{L}]$, specifies the encryption points $\mathcal{L} \subseteq \mathcal{C}$ at which E is allowed to have been encrypted.

We often write $[\mathsf{dest}\ \ell]$ and $[\mathsf{orig}\ \ell]$ instead of the more inconvenient $[\mathsf{dest}\ \{\ell\}]$ and $[\mathsf{orig}\ \{\ell\}]$.

We shall write $\llbracket \cdot \rrbracket$ for a term with all annotations removed. In particular

$$\llbracket \{E_1, \cdots, E_k\}^{\ell}_{E_0}\ [\mathsf{dest}\ \mathcal{L}]\ \rrbracket = \{\llbracket E_1 \rrbracket, \cdots, \llbracket E_k \rrbracket\}_{\llbracket E_0 \rrbracket}$$

The running example protocol with annotations looks as follows:

$$
\begin{array}{ll}
& (\nu\, K_A)(\nu\, K_B) \\
& \quad (\nu\, K)\langle A, B, \{K\}^{A}_{K_A}\ [\mathsf{dest}\ S]\ \rangle.(\nu\, Msg)\langle \{Msg\}^{A}_{K}\ [\mathsf{dest}\ B]\ \rangle.0 \\
| & \quad (A, B; y_{Enc}).\mathsf{decrypt}\ y_{Enc}\ \mathsf{as}\ \{y_K\}^{S}_{K_A}\ [\mathsf{orig}\ S]\ \mathsf{in}\ \langle \{y_K\}^{S}_{K_B}\ [\mathsf{dest}\ B]\ \rangle.0 \\
| & \quad (; x_1).\mathsf{decrypt}\ x_1\ \mathsf{as}\ \{z_K\}^{B}_{K_B}\ [\mathsf{orig}\ S]\ \mathsf{in}\ (; x_2).\mathsf{decrypt}\ x_2\ \mathsf{as}\ \{z_{Msg}\}^{B}_{z_K}\ [\mathsf{orig}\ A]\ \mathsf{in}\ 0
\end{array}
$$

In this case we simply use the principal names as crypto points. In some cases it is necessary to use distinct names as crypto-points at different places within the same principal process. This is for instance necessary in the case where we want to be able to detect a breach of a protocol which relies on the interchange of two encryption terms send between the same two principals. (An example of this can be found in Section 3.1).

(Com)

$$
\frac{\wedge_{i=1}^{j}\ [\![E_i]\!] = [\![E'_i]\!]}{\langle E_1, \cdots, E_k\rangle.P \mid (E'_1, \cdots, E'_j; x_{j+1}, \cdots, x_k).Q \to P \mid Q[E_{j+1}/x_{j+1}, \cdots, E_k/x_k]}
$$

(Decr)

$$
\frac{\wedge_{i=0}^{j}\ [\![E_i]\!] = [\![E'_i]\!]}{\begin{array}{r}\mathsf{decrypt}\ \{E_1, \cdots, E_k\}^{\ell}_{E_0}\ [\mathsf{dest}\ \mathcal{L}]\ \ \mathsf{as}\ \{E'_1, \cdots, E'_j; x_{j+1}, \cdots, x_k\}^{\ell'}_{E'_0}\ [\mathsf{orig}\ \mathcal{L}']\ \mathsf{in}\ P \\ \to P[E_{j+1}/x_{j+1}, \cdots, E_k/x_k]\end{array}}
$$

(Par)
$$
\frac{P \to P'}{P \mid Q \to P' \mid Q}
$$

(Res)
$$
\frac{P \to P'}{(\nu\, n)P \to (\nu\, n)P'}
$$

(Congr)
$$
\frac{P \equiv Q\ \wedge\ Q \to Q'\ \wedge\ Q' \equiv P'}{P \to P'}
$$

Table 2.1: Operational semantics, $P \to P'$ for LySa.

Semantics. Table 2.1 gives a reduction semantic for LySa. We use the standard notion of substitution, $P[E/x]$ that replaces all free occurences of x in P with E.

The structural congruence, $\equiv$, is defined on processes to be the least congruence satisfying the conditions of Table 2.2.

The origin and destination annotations are ignored by the semantics. They are used in the judgments of the analysis in Section 3. The reduction relation is the least relation on closed processes, i.e. processes with no free variables, that satisfies the rules in Table 2.1.

The rule (Com) expresses that an output

$$\langle E_1, \cdots, E_j, E_{j+1}, \cdots, E_k\rangle.P$$

is matched by an input

$$(E'_1, \cdots, E'_j; x_{j+1}, \cdots, x_k).Q$$

a1	$P \equiv P$
a2	$P_1 \mid P_2 \equiv P_2 \mid P_1$
a3	$(P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3)$
a4	$(\nu\, n_1)(\nu\, n_2)P \equiv (\nu\, n_2)(\nu\, n_1)P$
a5	$(\nu\, n)0 \equiv 0$
a6	$!\,P \equiv P \mid !\,P$
r1	$P_1 \equiv P_2 \Rightarrow P_2 \equiv P_1$
r2	$P_1 \equiv P_2 \wedge P_2 \equiv P_3 \Rightarrow P_1 \equiv P_3$
r3	P_1 and P_2 are α-equivalent $\Rightarrow P_1 \equiv P_2$
r4	$P_1 \equiv P_2 \Rightarrow P_1 \vert P_3 \equiv P_2 \vert P_3$
r5	$P_1 \equiv P_2 \Rightarrow (\nu\, n)P_1 \equiv (\nu\, n)P_2$
r6	$n \notin \mathsf{fn}(P_2) \Rightarrow (\nu\, n)(P_1 \mid P_2) \equiv P_1 \mid (\nu\, n)P_2$

Table 2.2: Congruence of processes for LySa

in case the first j elements are pairwise the equal, (ignoring annotations). When these comparisons are successful each E_i is bound to each x_i for $i > j$ within Q.

The inference rule (Decr) expresses a similar match. The decrypt construction

$$\mathsf{decrypt}\ \{E_1, \cdots, E_j, E_{j+1}, \cdots, E_k\}_{E_0}\ \mathsf{as}\ \{E'_1, \cdots, E'_j; x_{j+1}, \cdots, x_k\}_{E'_0}\ \mathsf{in}\ P$$

can be executed if $[\![E_i]\!] = [\![E'_i]\!]$ for $i \leq j$. When the comparisons are successful the result of matching the encryption value, E with the corresponding pattern each E_i is bound to each x_i for $i > j$ within Q. Note that the keys must be equal, e.i. $[\![E_0]\!] = [\![E'_0]\!]$. This models perfect symmetric cryptography.

The rule (Par) describes the behavior of processes in parallel. If P can make a transition into P', then $P \mid Q$ can make the transition into $P' \mid Q$. The rule for restrictions, (Res), describes that if P can make a transition into P' then $(\nu\, n)P$ can make a transition into $(\nu\, n)P'$. The rule (Congr) introduces the use of the structural congruence, $\equiv$, defined above.

In the LySa tool a meta level is implemented, which enables a compact specification of protocols where several principals plays the same roles. This meta level comes with guidelines for naming and indexing legitimate principals, the trusted server and the intruder. This meta level is not considered in this project. The consequence is that it is simpler to specify a basic instance of a protocol but more troublesome to specify instances with more than one principal in each role.

2.2 LySa$^{\mathrm{N}}$

LySa$^{\mathrm{N}}$ is a variant of LySa. It has a very close resemblence to the graph representation which is introduced in Section 2.4. At the same time it has a fairly straight forward transformation into a LySa process that is *equivalent* with regard to the static analysis introduced in Section 3. As mentioned earlier the purpose of LySa$^{\mathrm{N}}$ is to tie the LySa representation and the graph representation together.

Syntax. Terms and patterns of LySa$^{\mathrm{N}}$ are defined by the following grammar:

Terms		
$t ::=$	n	name ($n \in \mathcal{N}$)
$\mid$	x	variable ($x \in \mathcal{X}$)
$\mid$	$\mathsf{key}[t]$	key table lookup
$\mid$	$\{t_1, \cdots, t_k\}_{t_0}^{t_{cp}}$ [$\mathsf{dest}\ t_{dest}$]	encryption term
Patterns		
$p ::=$	n	name
$\mid$	$\%x$	defining occurence of message variable
$\mid$	$\$x$	defining occurence of principal variable
$\mid$	x	applied occurence of variable
$\mid$	$\mathsf{key}[p]$	key table lookup
$\mid$	$\{p_1, \cdots, p_k\}_{p_0}^{p_{cp}}$ [$\mathsf{orig}\ p_{orig}$]	encryption pattern

$\%x$ denotes a defining occurence of a variable. LySa$^{\mathrm{N}}$ introduces a special type of variable, which can only be bound to principal names. $\$x$ denotes the defining occurence of such a *principal variable*.

The construction $\mathsf{key}[t]$ represents the ability of a principal to look up the key of t. The term t must be the name of a principal. This means that if t is a variable, the corresponding defining occurrence of that variable must be of the form $\$x$. The $\mathsf{key}[\cdot]$-construction is for instance used to simulate a database query made by a trusted server in order to retrieve the shared key of one of the principals.

The construction $\{t_1, \cdots, t_k\}_{t_0}$ represents the encryption of the sequence of terms $t_1, \cdots, t_k$ with the key t_0. Likewise for the pattern $\{p_1, \cdots, p_k\}_{p_0}$. We use annotations in the same way as in LySa except that the origin and destination annotations are represented by terms and patterns and not sets of crypto-points.

Pattern matching is defined in Table 2.3 by judgments on the form $\theta \vdash v \triangleright p : \theta'$. The semantics operates over values v of the form

$$v ::= n \ \mid \ \mathsf{key}[n] \ \mid \ \{v_1, \cdots, v_k\}_{v_0}$$

i.e. over terms with no free variables. When a pattern is matched against a value, it will be relative to an environment $\theta : Var \to Val$, mapping the applied occurrences of variables into their values. Furthermore the matching gives rise to an extension θ' of the environment, which additionally maps the defining occurrences of the variables of the pattern p into their values.

$\theta[y \mapsto v]$ denotes the environment θ extended with the mapping from y to v.

$$\theta \vdash n \rhd n : \theta \qquad\qquad \theta \vdash v \rhd x : \theta \;\; \text{if } v = \theta x$$

$$\frac{\theta \vdash v \rhd x : \theta}{\theta \vdash key[v] \rhd key[x] : \theta} \qquad\qquad \begin{array}{c} \theta \vdash v \rhd \%x : \theta[x \mapsto v] \\ \theta \vdash v \rhd \$x : \theta[x \mapsto v] \end{array}$$

$$\frac{\theta \vdash v_0 \rhd p_0 : \theta_0 \wedge \theta_0 \vdash v_1 \rhd p_1 : \theta_1 \wedge \cdots \wedge \theta_{k-1} \vdash v_k \rhd p_k : \theta_k}{\theta \vdash \{v_1, \cdots, v_k\}_{v_0} \rhd \{p_1, \cdots, p_k\}_{p_0} : \theta_k}$$

Table 2.3: Pattern matching in LySa$^{\mathrm{N}}$.

The grammar which defines processes has two levels, P and S.

$$\begin{array}{lll} P ::= & \langle t_1, \cdots, t_k\rangle.P & \text{output} \\ & \mid (p_1, \cdots, p_k).P & \text{input} \\ & \mid (\nu\, n)P & \text{restriction} \\ & \mid \mathsf{loop}(P) & \text{loop} \\ & \mid \textsc{ret} & \text{return} \end{array}$$

$$\begin{array}{lll} S ::= & (\nu\, n)S & \text{restriction} \\ & \mid S_1 \mid S_2 & \text{parallel composition} \\ & \mid [P] & \text{principal process} \end{array}$$

Processes on level P are sequential processes without any parallelism. Level S makes it possible to place processes of type P in parallel, ($[P_1] \mid [P_2] \mid \cdots$). A protocol is represented by an S-process. Each principal will in turn be represented by a P-process. Each process of type P is enclosed in a loop construction indicated by $\mathsf{loop}(\cdots .\textsc{ret})$, (see the semantical transition, loop, in Table 2.4). This means that we only consider protocols, where each P-process is executed sequentially an infinite number of times as opposed to LySa where the replication, $!P$, is used.

As in LySa the process $\langle t_1, \cdots, t_k\rangle.P$ denotes an output followed by the process P and $(p_1, \cdots, p_k).P$ denotes an input followed by the process P. Note that LySa$^{\mathrm{N}}$ does not contain the explicit decrypt construction

$$\mathsf{decrypt}\ E\ \mathsf{as}\ \{\cdots\}^{\ell}\ [\mathcal{L}]\ \mathsf{in}\ P$$

that we have in LySa.

The semantics of processes is defined as a reduction semantics in Table 2.4 using a structural congruence specified in Table 2.5.

The concept of principal variables motivates the introduction of the two types of terms and patterns: Mess and Prin. Terms and patterns of type Prin are either principal names or variables that can only be bound to principal names. All other terms and patterns are of type Mess. A wellformedness condition for terms, patterns and processes with regard to these types is defined in Table 2.6. In the conditions τ denotes a mapping from terms to types, i.e. $\tau : \mathcal{N} \cup \mathcal{X} \rightarrow \{\mathsf{Mess}, \mathsf{Prin}\}$. The wellformedness conditions has the following forms:

(Com)

$$\frac{\emptyset \vdash t_1 \rhd p_1 : \theta_1 \quad \theta_1 \vdash t_2 \rhd p_2 : \theta_2 \quad \theta_{k-1} \vdash t_k \rhd p_k : \theta_k}{[\langle t_1, \cdots, t_k\rangle.P_1] \mid [(p_1, \cdots, p_k).P_2] \to [P_1] \mid [\theta_k P_2]}$$

(Par)

$$\frac{S_1 \to S_1'}{S_1 \mid S_2 \to S_1' \mid S_2}$$

(Res S)

$$\frac{S \to S'}{(\nu\, n)S \to (\nu\, n)S'}$$

(Res P)

$$\frac{P \to P'}{(\nu\, n)P \to (\nu\, n)P'}$$

(Congr)

$$\frac{S \equiv R \wedge R \to R' \wedge R' \equiv S'}{S \to S'}$$

(Loop)

$$\mathsf{loop}(P) \to P[\mathsf{loop}(P)/\textsc{ret}]$$

Table 2.4: Operational semantics, $S \to S'$ and $P \to P'$ for LySA$^{\mathrm{N}}$.

$\mathsf{a1_N}$	$S \equiv_N S$
$\mathsf{a1'_N}$	$P \equiv_N P$
$\mathsf{a2_N}$	$S_1 \mid S_2 \equiv_N S_2 \mid S_1$
$\mathsf{a3_N}$	$(S_1 \mid S_2) \mid S_3 \equiv_N S_1 \mid (S_2 \mid S_3)$
$\mathsf{a4_N}$	$(\nu\, n_1)(\nu\, n_2)S \equiv_N (\nu\, n_2)(\nu\, n_1)S$
$\mathsf{a4'_N}$	$(\nu\, n_1)(\nu\, n_2)P \equiv_N (\nu\, n_2)(\nu\, n_1)P$
$\mathsf{r1_N}$	$S_1 \equiv_N S_2 \Rightarrow S_2 \equiv_N S_1$
$\mathsf{r1'_N}$	$P_1 \equiv_N P_2 \Rightarrow P_2 \equiv_N P_1$
$\mathsf{r2_N}$	$S_1 \equiv_N S_2 \wedge S_2 \equiv_N S_3 \Rightarrow S_1 \equiv_N S_3$
$\mathsf{r2'_N}$	$P_1 \equiv_N P_2 \wedge P_2 \equiv_N P_3 \Rightarrow P_1 \equiv_N P_3$
$\mathsf{r3_N}$	S_1 and S_2 are α-equivalent $\Rightarrow S_1 \equiv_N S_2$
$\mathsf{r3'_N}$	P_1 and P_2 are α-equivalent $\Rightarrow P_1 \equiv_N P_2$
$\mathsf{r4_N}$	$S_1 \equiv_N S_2 \Rightarrow S_1 \vert S_3 \equiv_N S_2 \vert S_3$
$\mathsf{r5_N}$	$S_1 \equiv_N S_2 \Rightarrow (\nu\, n)S_1 \equiv_N (\nu\, n)S_2$
$\mathsf{r5'_N}$	$P_1 \equiv_N P_2 \Rightarrow (\nu\, n)P_1 \equiv_N (\nu\, n)P_2$
$\mathsf{r6_N}$	$n \notin \mathsf{fn}(S_2) \Rightarrow (\nu\, n)(S_1 \mid S_2) \equiv_N S_1 \mid (\nu\, n)S_2$

Table 2.5: Congruence of processes for LySA$^{\mathrm{N}}$.

- $\tau \vdash t : type$, for terms, t. The condition expresses that τ is valid for the term t having the type *type*.
- $\tau \vdash p : type, \tau'$, for patterns, p. The condition expresses that τ is valid for the pattern p, with the type *type*. Furthermore the pattern gives rise to an extension, τ', of τ.
- $\tau \vdash P$, for processes on level P. τ is valid for the process P.
- $\tau \vdash S$, for processes on level S. τ is valid for the process S.

Definition: A pattern is *simple* if it has the form n, x or $\mathsf{key}[p]$ where p is simple.

Terms, $\tau \vdash t : type$

$$\tau \vdash n : \tau(n) \quad \text{if } n \in dom(\tau) \qquad\qquad \tau \vdash x : \tau(x) \quad \text{if } x \in dom(\tau)$$

$$\frac{\tau \vdash t : \mathsf{Prin}}{\tau \vdash \mathsf{key}[t] : \mathsf{Mess}} \qquad \frac{\tau \vdash t_0 : type_0 \wedge \cdots \wedge \tau \vdash t_k : type_k \wedge \quad \tau \vdash t_{cp} : \mathsf{Prin} \wedge \tau \vdash t_{dest} : \mathsf{Prin}}{\tau \vdash \{t_1, \cdots, t_k\}_{t_0}^{t_{cp}}\ [t_{dest}]\ : \mathsf{Mess}}$$

Patterns, $\tau \vdash p : type, \tau'$

$$\tau \vdash n : \tau(n), \tau \qquad \text{if } n \in dom(\tau)$$

$$\tau \vdash x : \tau(x), \tau \qquad \text{if } x \in dom(\tau)$$

$$\tau \vdash \%x : \mathsf{Mess}, \tau[x \mapsto \mathsf{Mess}] \qquad \text{if } x \notin dom(\tau)$$

$$\tau \vdash \$x : \mathsf{Prin}, \tau[x \mapsto \mathsf{Prin}] \qquad \text{if } x \notin dom(\tau)$$

$$\frac{\tau \vdash p : \mathsf{Prin}, \tau'}{\tau \vdash \mathsf{key}[p] : \mathsf{Mess}, \tau'}$$

$$\frac{\tau \vdash p_0 : \mathsf{Mess}, \tau_0 \wedge \tau_0 \vdash p_1 : type_1, \tau_1 \wedge \cdots \wedge \tau_{k-1} \vdash p_k : type_k, \tau_k \wedge \quad \tau_k \vdash p_{cp} : \mathsf{Prin}, \tau_{cp} \wedge \tau_{cp} \vdash p_{orig} : \mathsf{Prin}, \tau_{orig}}{\tau \vdash \{p_1, \cdots, p_k\}_{p_0}^{p_{cp}}\ [\mathsf{orig}\ p_{orig}]\ : \mathsf{Mess}, \tau_{orig}}$$

where $\exists j \geq 1 : \forall i \leq j : p_i$ is simple $\wedge\, \forall i > j : p_i$ is not simple
$\wedge$ subpatterns of p_0, p_{cp} and p_{orig} must not be of the form $\%x$ or $\$x$

Processes, $\tau \vdash P$ and $\tau \vdash S$

$$\tau \vdash \text{RET} \qquad\qquad \frac{\tau \vdash t_1 : type_1 \wedge \cdots \tau \vdash t_k : type_k \wedge \tau \vdash P}{\tau \vdash \langle t_1, \cdots, t_k \rangle.P}$$

$$\frac{\tau \vdash p_1 : type_1, \tau_1 \wedge \tau_1 \vdash p_2 : type_2, \tau_2 \wedge \cdots \wedge \tau_{k-1} \vdash t_k : type_k, \tau_k \wedge \quad \tau_k \vdash P}{\tau \vdash (p_1, \cdots, p_k).P}$$

where $\exists j \geq 1 : \forall i \leq j : p_i$ is simple $\wedge\, \forall i > j : p_i$ is not simple

$$\frac{\tau[n \mapsto \mathsf{Mess}] \vdash P}{\tau \vdash (\nu\, n)P} \qquad \frac{\tau \vdash P}{\tau \vdash \mathsf{loop}(P)} \qquad \frac{\tau \vdash P}{\tau \vdash [P]}$$

$$\frac{\tau[n \mapsto \mathsf{Prin}] \vdash S}{\tau \vdash (\nu\, n)S} \qquad\qquad \frac{\tau \vdash S_1 \wedge \tau \vdash S_2}{\tau \vdash S_1 \mid S_2}$$

Table 2.6: Wellformedness condition for terms, patterns and processes.

From Table 2.6 we see that the defining occurrences of message variables and principal variables give rise to extensions of τ in the judgments. The judgment for the encryption terms and encryption patterns demands that each of the sub-terms – and sub-patterns respectively – are wellformed. Note that the wellformedness condition also ensures that variables are bound before they occur in applied context and names are defined with the $(\nu\ \cdot)$-construction before they are used. Furthermore we see that variables cannot be redefined. In encryptions, input processes and output processes we have tuples of terms and patterns. From the wellformedness condition we see that these tuples are evaluated from left to right. From the judgments for processes we see that names defined on level S, $(\nu\, n)S$, are always names of type Prin and names defined on level P, $(\nu\, n)P$, are always names of type Mess. Note that since keys does not have to be represented by names, but can be represented using the $\mathsf{key}[\cdot]$-construction using a principal name, we do not explicitly have a restriction on each of the keys of the principals as in LySa.

The rules for input processes as well as encryption patterns demand that it must be possible to make a split somewhere in the (input or encryption) tuples such that we only have simple patterns on one side and and non-simple patterns on the other side. This is done in order to be able to make the conversion to LySa where a semicolon is introduced to separate matching terms from variables that are to be bound.

The condition that the key or annotations of an encryption pattern must not be of the form %x or $x is introduced to ensure that it is not possible to guess a key or an intended crypto-point.

Adding type checking in pattern match. As we shall see in the next section it is important to ensure that principal variables can only be bound to names, n, where $\tau(n) = \mathsf{Prin}$. This is done by parameterizing the judgments of the pattern match on τ: $\theta \vdash_\tau v \rhd p : \theta'$. Apart from adding the parameter to each of the rules, the only rule from Table 2.3 that is changed is the rule for matching a value with the defining occurrence of a principal variable. The rule is changed into

$$\theta \vdash_\tau v \rhd \$x : \theta[x \mapsto v] \text{ if } v = n \text{ and } \tau(n) = \mathsf{Prin}$$

Valid processes. A process, S, is a valid representation of a protocol if

- $\tau[I \mapsto \mathsf{Prin}] \vdash S$. I.e. the name I is always a principal of the protocol. Namely the intruder.
- Loops cannot be nested. I.e. there must be no occurrences of $\mathsf{loop}(\cdot)$ inside P in an expression $\mathsf{loop}(P)$.

An example of a LySa$^{\mathrm{N}}$ Specification. The following is the LySa$^{\mathrm{N}}$ specification of the running example protocol.

$$
\begin{array}{ll}
(\nu\, A)(\nu\, B)(\nu\, S) & \\
 & [\mathsf{loop}((\nu\, K)\langle A, B, \{K\}_{K_A}\rangle.(\nu\, Msg)\langle\{Msg\}_K\rangle.\textsc{ret})] \\
| & [\mathsf{loop}((\$y_A, \$y_B, \{\%y_K\}_{\mathsf{key}[y_A]}).\langle y_A, \{y_K\}_{\mathsf{key}[y_B]}\rangle.\textsc{ret})] \\
| & [\mathsf{loop}((\$z_A, \{\%z_K\}_{\mathsf{key}[B]}).(\{\%z_{Msg}\}_{z_K}).\textsc{ret})]
\end{array}
$$

2.3 Differences and transformation between LySa$^{\text{N}}$ and LySa

We have now defined the two process algebras, LySa$^{\text{N}}$ and LySa. In this section we shall take a closer look at the differences between the two and how to transform a LySa$^{\text{N}}$ process into a LySa process. Since our starting point is a LySa$^{\text{N}}$ specification of a protocol, we only need a transformation from LySa$^{\text{N}}$ to LySa. In Section 3.2 we look at how the analysis of a LySa process is related to the corresponding original LySa$^{\text{N}}$ process.

We have the following differences between LySa$^{\text{N}}$ and LySa. Each of the differences are explained in detail below.

- Principal variables and the $\mathsf{key}[\cdot]$-construction are used in LySa$^{\text{N}}$.
- Restriction $(\nu\, n)$ on S level are made on principal names instead of principal keys.
- LySa$^{\text{N}}$ uses a loop construction whereas LySa uses the replication operator, (!).
- The pattern matching of LySa$^{\text{N}}$ enables implicit decryption instead of the explicit decrypt construction of LySa.

LySa$^{\text{N}}$ has the construction, $\mathsf{key}[n]$. As explained this construction is used to specify the key that a certain principal with the name n shares with a trusted server. Of cause this construction can only be used when specifying protocols, that are actually based on a setup with a set of principals and a trusted server.

As an example we look at the process of the trusted server S from the running example protocol, specified in LySa$^{\text{N}}$ and LySa respectively:

LySa$^{\text{N}}$:

$$[\mathsf{loop}((\$y_A, \$y_B, \{\%y_K\}^S_{\mathsf{key}[y_A]}\ [\mathsf{orig}\ y_A]\ ).\langle y_A\{y_K\}^S_{\mathsf{key}[B]}\ [\mathsf{dest}\ y_B]\ \rangle.\textsc{ret})]$$

LySa:

$$!\,(A, B; y_{Enc}).\mathsf{decrypt}\ y_{Enc}\ \mathsf{as}\ \{; y_K\}^S_{K_A}\ [\mathsf{orig}\ A]\ \mathsf{in}\ \langle A, \{y_K\}^S_{K_B}\ [\mathsf{dest}\ B]\ \rangle.0$$

In the LySa$^{\text{N}}$ process the variables $\$y_A$ and $\$y_B$ can be bound to the names of any of the principals in the protocol including the intruder. The rest of the process works correctly, i.e. the keys and annotations corresponds to the principal names bound to the two variables. This dynamic behavior is not possible in the LySa process because variables are not allowed in annotations and because LySa does not contain the $\mathsf{key}[\cdot]$-construction. The consequence is that the LySa$^{\mathrm{N}}$ process can work as a server for any two principals in the roles of A and B, whereas the LySa process only decrypts the session key K if it is encrypted with the key of A and only re-encrypts K with the key of principal B.

Since the names of the principals does not necessarily has to be A and B in the LySa$^{\mathrm{N}}$ process we can consider the following example: The intruder can intercept the first message from A to S and insert the name I in the message such that y_B is bound to I. The consequence is that the session key K, which was intended for B now ends up on the network encrypted with $\mathsf{key}[I]$. This example corresponds to the following narration:

$$\begin{array}{llll} A & \rightarrow & I(S) & : & A, B, \{K\}_{\mathsf{key}[A]} \\ I(A) & \rightarrow & S & : & A, I, \{K\}_{\mathsf{key}[A]} \\ S & \rightarrow & I & : & A, \{K\}_{\mathsf{key}[I]} \end{array}$$

In LySa we need a set of parallel processes corresponding to one LySa$^{\mathrm{N}}$ process to obtain the same effect. In particular we need a server process of the form

$$!\,(A, I; y_{Enc}).\mathsf{decrypt}\ y_{Enc}\ \mathsf{as}\ \{y_K\}^{S}_{K_A}\ [\mathsf{orig}\ A]\ \mathsf{in}\ \langle A, \{y_K\}^{S}_{K_I}\ [\mathsf{dest}\ I]\ \rangle.0$$

to enable the attack above.

We refer to this part of the transformation as the principal unfolding and it is explained in details below.

Note that the LySa$^{\mathrm{N}}$ process not only contains variables in the $\mathsf{key}[\cdot]$-construction. The annotations has to reflect which principals are bound to the variables y_A and y_B and thereby reflect the intended source and destination of the encryptions. This is done by using the variables bound to the principal names as crypto-points[1].

To ensure the right conversion of these variables when transforming a process from LySa$^{\mathrm{N}}$ to LySa, we need to make certain that only names of principals can occur inside the $\mathsf{key}[\cdot]$-construction and as crypto-points. This is ensured by the wellformedness condition defining a certain type of variables as principal variables.

Since the key of a principal is not explicitly represented by a name but by using the construction $\mathsf{key}[n]$, where n is the name of the principal, we cannot use restrictions on S level as the are used in LySa. Instead we use the restriction on S level to specify the names of all principals, as seen from the wellformedness condition in Table 2.6.

The two process algebras also differ in the grammar of processes. A protocol specified in LySa$^{\mathrm{N}}$ consists of a set of sequential processes performing simple

[1] We encounter a problem when we are not satisfied with simply using principal names as crypto-points but have the need for distinct crypto-points within the same principal process as explained in Section 2.1. This problem could however simply be solved by allowing a construction where additional labels are assigned to each of the crypto-points within a principal.

loops in parallel. LySa allows a more advanced structure, where we can use parallel processes and replication (!) on any level.

LySa$^{\mathrm{N}}$ enables a more sophisticated pattern match. The reason for this is that we want to omit the decrypt construction of LySa. For instance the match of the following two tuples from the running example protocol also contains the implicit decryption of $\{K\}_{\mathsf{key}[A]}$:

$$\langle A, B, \{K\}_{\mathsf{key}[A]}\rangle \rhd (\$y_A, \$y_B, \{y_K\}_{\mathsf{key}[y_A]})$$

As we shall see in Section 2.4.1 omitting the explicit decrypt construction is an advantage when transforming a LySa$^{\mathrm{N}}$ specification of a protocol into a graph representation.

2.3.1 Transformation from LySa$^{\mathrm{N}}$ to LySa.

In the following we go through the transformation from LySa$^{\mathrm{N}}$ to LySa.

When we transform a LySa$^{\mathrm{N}}$ process we have to take care of the following aspects, cf. the differences explained above:

1. All processes on level P containing principal variables, $\$x$, must be unfolded and all occurrences of principal variables must be replaced. $\mathsf{key}[\cdot]$-constructions hereby becomes superfluous and is replaced as well.
2. Restrictions on names on level S, $(\nu\, n)$, are changed into restrictions on the corresponding key, $(\nu\, K_n)$.
3. We must use replication (!) instead of the sequential loop construction of LySa$^{\mathrm{N}}$.
4. The advanced pattern matching of LySa$^{\mathrm{N}}$ allowing patterns with defining occurrences of variables on any encryption level must be simplified or *flattened*. This procedure is implemented recursively to handle arbitrary deep encryptions.

1. Unfolding principal variables. Again we look at the LySa$^{\mathrm{N}}$ and LySa process representing the trusted server of the running example protocol. As we saw in the previous section, the two processes are not equivalent, since the LySa$^{\mathrm{N}}$ process establishes communication between any two principals.

To remove principal variables from a LySa$^{\mathrm{N}}$ process we perform an *unfolding* of the process into a set of LySa$^{\mathrm{N}}$ processes. The unfolding is based on the set of principals of the protocol *including* the intruder. In the running example protocol, we have the principals A, B, S, I. In the following example we unfold the LySa$^{\mathrm{N}}$ process (level P) to enable the server to serve any possible pairs of these principals. The LySa$^{\mathrm{N}}$ process

$$\mathsf{loop}((\$y_A, \$y_B, \{\%y_K\}^{S}_{\mathsf{key}[y_A]}\ [\mathsf{orig}\ y_A]\ ).\langle\{y_K\}^{S}_{\mathsf{key}[y_B]}\ [\mathsf{dest}\ y_B]\ \rangle.\mathrm{RET})$$

contains two principal variables and is unfolded to $4^2 = 16$ processes. 16 is the size of the set of possible substitutions:

$$\{[n_A/y_A, n_B/y_B] \mid (n_A, n_B) \in \{A, B, S, I\} \times \{A, B, S, I\}\}$$

The unfolded processes are placed in parallel:

$$\begin{array}{l} \ [\mathsf{loop}((A, A, \{\%y_K^{AA}\}^S_{\mathsf{key}[A]}[\mathsf{orig}\ A]).\langle\{y_K^{AA}\}^S_{\mathsf{key}[A]}[\mathsf{dest}\ A]\rangle.\mathrm{RET})] \\ \mid [\mathsf{loop}((A, B, \{\%y_K^{AB}\}^S_{\mathsf{key}[A]}[\mathsf{orig}\ A]).\langle\{y_K^{AB}\}^S_{\mathsf{key}[B]}[\mathsf{dest}\ B]\rangle.\mathrm{RET})] \\ \mid [\mathsf{loop}((A, I, \{\%y_K^{AI}\}^S_{\mathsf{key}[A]}[\mathsf{orig}\ A]).\langle\{y_K^{AI}\}^S_{\mathsf{key}[I]}[\mathsf{dest}\ I]\rangle.\mathrm{RET})] \\ \mid [\mathsf{loop}((A, S, \{\%y_K^{AS}\}^S_{\mathsf{key}[A]}[\mathsf{orig}\ A]).\langle\{y_K^{AS}\}^S_{\mathsf{key}[S]}[\mathsf{dest}\ S]\rangle.\mathrm{RET})] \\ \mid [\mathsf{loop}((B, A, \{\%y_K^{BA}\}^S_{\mathsf{key}[B]}[\mathsf{orig}\ B]).\langle\{y_K^{BA}\}^S_{\mathsf{key}[A]}[\mathsf{dest}\ A]\rangle.\mathrm{RET})] \\ \mid [\mathsf{loop}((B, B, \{\%y_K^{BB}\}^S_{\mathsf{key}[B]}[\mathsf{orig}\ B]).\langle\{y_K^{BB}\}^S_{\mathsf{key}[B]}[\mathsf{dest}\ B]\rangle.\mathrm{RET})] \\ \mid \cdots \end{array}$$

We have the substitution $[A/y_A, A/y_B]$ in the first line, $[A/y_A, B/y_B]$ in the second and so forth. To avoid name clashes between non-principal variables of different processes as well as new names $(\nu\, n)$ declared on P-level we annotate these with a unique identifier based on the substitution corresponding to the given instance of the process. As an example the variable y_K found in the process corresponding to the substitution $[A/y_A, I/y_B]$, (the third line), is renamed to y_K^{AI}. When we unfold a process according to the specified set of principal variables, we end up with process where all principal variable are substituted with names. This means that the $\mathsf{key}[\cdot]$-construction no longer is needed. Since it is not a part of LySa we exchange a term of the form $\mathsf{key}[n]$ with the name K_n.

Note that by substituting all principal variables, we have also ensured that there are no variables in the annotations.

The set of possible executions of the original LySa$^{\mathrm{N}}$ process is a subset of that of the new LySa$^{\mathrm{N}}$ process with principal variables removed. The wellformedness conditions and the rules for pattern matching ensures that the principal variables can only be bound to principal names. Since there exists a new LySa$^{\mathrm{N}}$ process corresponding to each of the possible combinations of substitutions we cannot have an execution path in the original process which cannot be simulated by the new set of unfolded processes. Because the static analysis performs an over-approximation we have that the analysis of the unfolded LySa$^{\mathrm{N}}$ process will also be valid for the original process.

2. Restriction on level S. $(\nu\, n)S$ is transformed into $(\nu\, K_n)S$. Instead of restricting principal names on S-level we now restrict the keys of the principals.

3. Replication instead of sequential return. A LySa$^{\mathrm{N}}$ process of the form

$$\begin{array}{l} (\nu\, A)(\nu\, B) \\ \quad [\mathsf{loop}(P_A)] \mid [\mathsf{loop}(P_B)] \mid [\mathsf{loop}(P_S)] \end{array}$$

is transformed into the LySa process

$$\begin{array}{l}(\nu\, K_A)(\nu\, K_B)\\ \quad !\, P'_A \mid !\, P'_B \mid !\, P'_S\end{array}$$

where P'_A, P'_B and P'_S are the transformed version of the LySa$^{\mathrm{N}}$ processes.

The structure on the S level is simply preserved and all sequential loop constructions are represented using replication instead. The semantics of the loop construction differs from the semantics of replication. However as we shall see in Section 3.2 the over-approximation performed by the static analysis is still safe.

4. Simplifying pattern matching. There are two aspects of the transformation of pattern matching from LySa$^{\mathrm{N}}$ to LySa. The simple part is that we have to determine where to position the semicolon that divides the matching terms from the variables that are to be bound. The wellformedness condition of LySa$^{\mathrm{N}}$ in Table 2.6 ensures that the patterns of LySa$^{\mathrm{N}}$ allows this division.

But before we can do this, we need to *flatten* the pattern. In LySa$^{\mathrm{N}}$ we can have arbitrary deep encryptions. These encryptions must first be bound to a new temporary variable which is introduced in the transformation. Afterwards the encryption must be decrypted using the decrypt-construction. As an example we look at the following LySa$^{\mathrm{N}}$ process:

$$(\{\%y\}_K).P$$

The process contains an implicit decryption as well as a binding of y. The process is transformed into the following LySa process:

$$(;x_1).\mathsf{decrypt}\ x_1\ \mathsf{as}\ \{;y\}_K\ \mathsf{in}\ P$$

This transformation is implemented recursively to be able to handle arbitrary depths of encryption as specified below.

Formal definition of the transformation. Table 2.8 contains a formal definition of the transformation. We introduce a variant of LySa$^{\mathrm{N}}$, LySa$^{\mathrm{N}}_{ext}$, as an intermediate stage in the transformation. This is convenient because we want to be able to use both the advanced pattern matching of LySa$^{\mathrm{N}}$ and the decrypt- construction of LySa in the same process, when iterating through levels of encryption and because we want to deal with the principal variables before simplifying the pattern matching. The extended grammar for terms, patterns and processes becomes:

$$\begin{array}{lcl}
t & ::= & n \mid x \mid \mathsf{key}[t] \mid \{t_1, \cdots, t_k\}^{\ell}_{t_0}\ [\mathsf{dest}\ \mathcal{L}] \\
p & ::= & n \mid \%x \mid x \mid \mathsf{key}[n] \mid \{p_1, \cdots, p_k\}^{\ell}_{p_0}\ [\mathsf{orig}\ \mathcal{L}] \\
\\
P & ::= & \langle t_1, \cdots, t_k\rangle.P \mid (p_1, \cdots, p_k).P \mid \mathsf{loop}(P) \mid \\
 & & \mathsf{decrypt}\ t\ \mathsf{as}\ \{p_1, \cdots, p_k\}^{p_{cp}}_{p_0}\ [\mathsf{orig}\ p_{orig}]\ \mathsf{in}\ P \mid (\nu\, n).P \mid \textsc{ret} \\
S & ::= & (\nu\, n)S \mid S_1 \mid S_2 \mid [P]
\end{array}$$

The only two differences are that principal variables are not allowed and that we introduce the $\mathsf{decrypt}$- construction. When principal variables are removed, the pattern $\mathsf{key}[p]$ in teh grammar above is changed to $\mathsf{key}[n]$ since it follows from the wellformedness condition in Table 2.6 that we can only have names in the $\mathsf{key}[\cdot]$-construction.

We define a transformation function, $\lfloor\cdot\rfloor : \textsc{LySa}^{\mathrm{N}}_{ext} \to \textsc{LySa}$, which denotes the transformation of a $\textsc{LySa}^{\mathrm{N}}_{ext}$ process to a $\textsc{LySa}$ process.

$$S_{\textsc{LySa}} = \lfloor S_{\textsc{LySa}^{\mathrm{N}}_{ext}} \rfloor$$

where $S_{\textsc{LySa}}$ and $S_{\textsc{LySa}^{\mathrm{N}}_{ext}}$ are equivalent with regard to the analysis result. The notation $\lfloor\cdot\rfloor$ is overloaded to work on processes of type P as well as S.

We have the following overview of the transformation:

$$\boxed{\textsc{LySa}^{\mathrm{N}}} \quad \xrightarrow{\substack{\text{unfold principal}\\ \text{variables}}} \quad \boxed{\textsc{LySa}^{\mathrm{N}}_{ext}} \quad \xrightarrow{\substack{\text{transform patternmatch}\\ \text{and replication}}} \quad \boxed{\textsc{LySa}}$$

Table 2.7: Transformation overview

Note that the languages $\textsc{LySa}^{\mathrm{N}}$ and $\textsc{LySa}^{\mathrm{N}}_{ext}$ intersects. In particular the result of unfolding principals in a process $S_{\textsc{LySa}^{\mathrm{N}}}$ is a process $S'_{\textsc{LySa}^{\mathrm{N}}}$ which conforms to both the $\textsc{LySa}^{\mathrm{N}}$ and the $\textsc{LySa}^{\mathrm{N}}_{ext}$ grammar. In other words, the transformation defined in Table 2.8 works on $\textsc{LySa}^{\mathrm{N}}$ process with the condition that there are no occurrences of principal variables.

Example. Consider the transformation of the following $\textsc{LySa}^{\mathrm{N}}_{ext}$ process. For readability origin and destination annotations are omitted. The notation "$\rightsquigarrow$" denotes a step in the recursive transformation function:

$$\begin{array}{ll}
 & \lfloor(\nu\, A)[\mathsf{loop}((A, \{B, \{\%y\}_{K_1}\}_{\mathsf{key}[A]}).P)]\rfloor \\
\rightsquigarrow & (\nu\, K_A)\ \lfloor[\mathsf{loop}((A, \{B, \{\%y\}_{K_1}\}_{\mathsf{key}[A]}).P)]\rfloor \\
\rightsquigarrow & (\nu\, K_A)\ \lfloor\mathsf{loop}((A, \{B, \{\%y\}_{K_1}\}_{\mathsf{key}[A]}).P)\rfloor \\
\rightsquigarrow & (\nu\, K_A)\ !\,\lfloor(A, \{B, \{\%y\}_{K_1}\}_{\mathsf{key}[A]}).P\rfloor \\
\rightsquigarrow & (\nu\, K_A)\ !\,(A; x_1).\lfloor\mathsf{decrypt}\ x_1\ \mathsf{as}\ \{B, \{\%y\}_{K_1}\}_{\mathsf{key}[A]}\ \mathsf{in}\ P\rfloor \\
\rightsquigarrow & (\nu\, K_A)\ !\,(A, x_1).\mathsf{decrypt}\ x_1\ \mathsf{as}\ \{B; x_2\}_{K_A}\ \mathsf{in}\ \lfloor\mathsf{decrypt}\ x_2\ \mathsf{as}\ \{\%y\}_{K_1}\ \mathsf{in}\ P\rfloor \\
\rightsquigarrow & (\nu\, K_A)\ !\,(A, x_1).\mathsf{decrypt}\ x_1\ \mathsf{as}\ \{B; x_2\}_{K_A}\ \mathsf{in}\ \mathsf{decrypt}\ x_2\ \mathsf{as}\ \{; y\}_{K_1}\ \mathsf{in}\ \lfloor P\rfloor \\
\rightsquigarrow & \cdots
\end{array}$$

where x_i are temporary variables introduced by the transformation. As we see the transformation deals with each of the aspects necessary to get from $\textsc{LySa}^{\mathrm{N}}_{ext}$ to $\textsc{LySa}$. The loop constructions is replaced by the ! operator. Restrictions on level S are transformed as explained. The pattern matching is simplified by sorting the patterns and introducing semicolon and by introducing the temporary variables x_i and the explicit decryptions. The result is a $\textsc{LySa}$ process which as we shall see is equivalent with regard to the static analysis introduced in Chapter 3.

$$\lfloor (p_1, \cdots, p_k).P \rfloor =$$
$$\quad \mathsf{let}\ (us, vs, Q) = check(p_1, check(p_2, \cdots, check(p_k, ([\,], [\,], P)) \cdots)$$
$$\quad \mathsf{in}\ (us; vs).\lfloor Q \rfloor$$

$$\lfloor \langle p_1, \cdots, p_k \rangle.P \rfloor = \langle p_1, \cdots, p_k \rangle.\lfloor P \rfloor$$

$$\lfloor \mathsf{decrypt}\ t\ \mathsf{as}\ \{p1, \cdots, p_k\}_{p_0}^{n_{cp}}\ [\mathsf{orig}\ n_{orig}]\ \mathsf{in}\ P \rfloor =$$
$$\quad \mathsf{let}\ (us, vs, Q) = check(p_1, check(p_2, \cdots, check(p_k, ([\,], [\,], P)) \cdots)$$
$$\quad \mathsf{in\ decrypt}\ t\ \mathsf{as}\ \{us; vs\}_{p_0}^{n_{cp}}\ [\mathsf{orig}\ n_{orig}]\ \mathsf{in}\ \lfloor Q \rfloor$$

$$\lfloor (\nu\, n)P \rfloor = (\nu\, n)\lfloor P \rfloor$$

$$\lfloor \mathsf{loop}(P) \rfloor = !\ \lfloor P \rfloor$$

$$\lfloor \mathrm{RET} \rfloor = 0$$

$$\lfloor (\nu\, n)S \rfloor = (\nu\, K_n)\lfloor S \rfloor$$

$$\lfloor S_1 \mid S_2 \rfloor = \lfloor S_1 \rfloor \mid \lfloor S_2 \rfloor$$

$$\lfloor [P] \rfloor = \lfloor P \rfloor$$

$$check(n, (us, vs, Q)) = (n :: us, vs, Q)$$
$$check(\mathsf{key}[n], (us, vs, Q)) = (K_n :: us, vs, Q)$$
$$check(x, (us, vs, Q)) = (x :: us, vs, Q)$$
$$check(\%x, (us, vs, Q)) = (us, x :: vs, Q)$$
$$check(\{p_1, \cdots, p_k\}_{p_0}^{n_{cp}}\ [\mathsf{orig}\ n_{orig}], (us, vs, Q)) =$$
$$\quad (us, x_i :: vs, \mathsf{decrypt}\ x_i\ \mathsf{as}\ \{p_1, \cdots, p_k\}_{p_0}^{n_{cp}}\ [\mathsf{orig}\ n_{orig}]\ \mathsf{in}\ Q)$$

Table 2.8: Transformation from LySa$^{\mathrm{N}}_{ext}$ to LySa. "::" denotes the prefix concatenation of lists. "[ ]" denotes the empty list. The i in x_i is incremented to ensure that a new variable is used in each decrypt construction.

Comments to the transformation. In LySa it is actually possible to perform an implicit decryption of a term as well as an explicit.

$$(\{A\}_K; x).\mathsf{decrypt}\ x\ \mathsf{as}\ \{; y\}_K\ \mathsf{in}\ P$$

The term $\{A\}_K$ is implicitly decrypted in the match and the term $\{B\}_K$ bound to x is explicitly decrypted afterwards. Note that variables cannot be bound in an implicit decryption. In LySa$^{\mathrm{N}}$ the same process would look as follows:

$$(\{A\}_K, \{B\}_{\%y}).P$$

And the transformation would be

$$(; x_1, x_2).\mathsf{decrypt}\ x_1\ \mathsf{as}\ \{A\}_K\ \mathsf{in\ decrypt}\ x_2\ \mathsf{as}\ \{; y\}_K\ \mathsf{in}\ P$$

This example shows that we cannot indicate in LySa$^{\mathrm{N}}$ that we only wish an implicit decryption of an encryption term in the corresponding LySa process. In the analysis the difference between the implicit and explicit decryption is that annotations are not checked in the implicit decryption as explained in Section 3.1. This is an advantage in the case where we use encryption constructions to model functions such as the successor function or a pair function as mentioned in Section 2.1. In these cases we are not interested in the annotations and can make the analysis ignore them by using implicit decryption. However this is not possible when converting from LySa$^{\mathrm{N}}$. Instead it is advisable to use crypto-points, which can be easily recognized and ignored when looking at the analysis result.

2.4 The graph representation

For the purpose of model checking we represent a protocol as a directed graph, where the nodes corresponds to a state of the protocol and each transition corresponds to communication between principal and intruder.

Corresponding to the grammar of LySa$^{\mathrm{N}}$ the protocols we consider are restricted to have the following properties:

1. A protocol consists of a set of parallel processes. One process for each of the participating principals.
2. Each of these processes is a simple sequence of inputs and outputs corresponding to the communication that the corresponding principal is willing to participate in as defined by the protocol specification. We use the term *communication step* to denote these inputs and outputs.

Encryption and decryption are implemented directly in the pattern matching of the input and output as explained.

States. A state in a protocol, (corresponding to a node in the graph), is defined as the "position" of each of the principals in their respective sequence of communication steps. A sequence of communication steps is decorated with labels, and each principal has a name. This means that we can represent a state by a set of pairs: $(name, label)$. Note that a state contains no information about the knowledge of each of the principals nor of the intruder. The graph is simply a static representation of the structure of the protocol.

Transitions. As mentioned the transitions of the graph corresponds to communication between principals and intruder. Each transition consists of either an input or an output. An input transition represents communication from the intruder to a principal of the protocol. An output transition represents communication from a principal to the intruder. We do not consider transitions representing communication between two principals, (for instance $A \rightarrow B$) since this is simply modeled by letting the message go unaltered through the intruder. $A \rightarrow I \rightarrow B$. The only difference is that the message becomes known to the intruder, which certainly does not exclude any attacks.

The information contained in an output or input transition is the corresponding output or input tuple of the LySA$^{\mathrm{N}}$ specification together with the name of the principal who performs the communication.

Figure 2.2 illustrates the graph representation of the running example protocol.

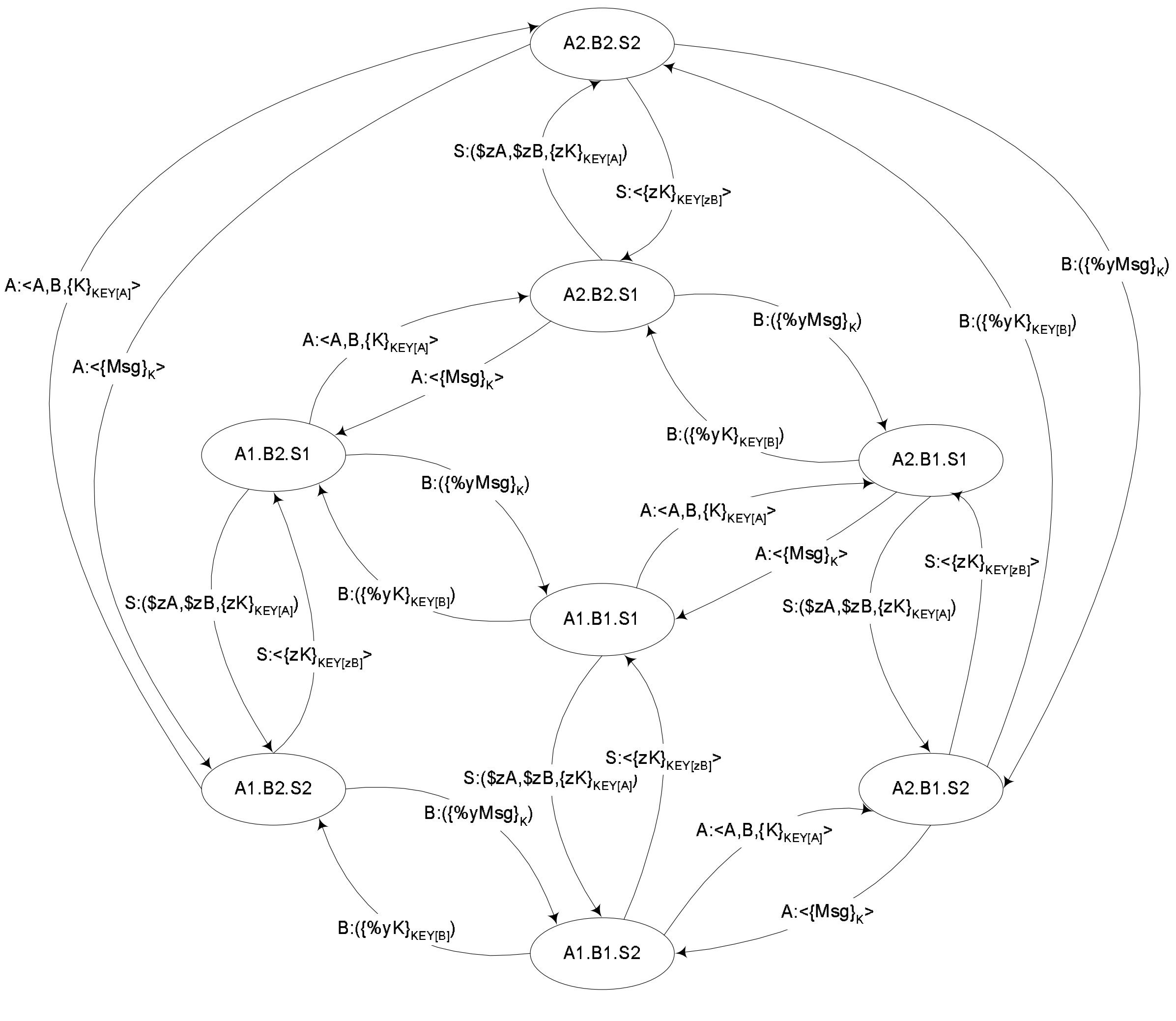

Figure 2.2: Graph representation of the running example protocol

The states are labeled according to the position of each of the principals, A, B and S. The initial state has the label $\{A_1, B_1, S_1\}$ (also written as $A_1.B_1.S_1$). The topology of the graph only depends on the number of principals and the number of steps in the process of each of the principals.

The first communication from principal A in the running example protocol is the following output: $\langle A, B, \{K\}_{\mathsf{key}[A]}\rangle$. Each input or output from the LySa$^{\mathrm{N}}$ representation can be found several places in the graph. The transition corresponding to the output $\langle A, B, \{K\}_{\mathsf{key}[A]}\rangle$ can be found between all pairs of states where we have the change of position of principal A from A_1 to A_2.

2.4.1 Creating the graph from LySa$^{\mathrm{N}}$

As we have seen the grammar and semantics of LySa$^{\mathrm{N}}$ closely resembles the graph representation of a protocol. We have restricted a protocol to consist of a set of parallel looping sequential processes and the actions of a process is communication steps. It is straight forward to convert a LySa$^{\mathrm{N}}$ specification of a protocol into the corresponding graph representation, which gives us the ability to simulate the communication steps of a protocol.

The conversion is implemented by the following steps:

- Labels are added to the LySa$^{\mathrm{N}}$ processes.
- The flow of the LySa$^{\mathrm{N}}$ process is generated. The flow consists of a list of steps. A step in turn consists of a pair of labels (l_1, l_2) together with the communication (input or output) corresponding to the transition of a principal from label l_1 to l_2.
- A graph is constructed from an initial state and the flow. The initial state is obtained from the labeled LySa$^{\mathrm{N}}$ process.

Adding labels. An implementational detail, which is not treated in the previous sections describing the semantics of LySa$^{\mathrm{N}}$, is that each looping process $[\mathsf{loop}(P)]$ is actually decorated with the name of the corresponding principal. These names are used when adding labels to the communication steps of a LySa$^{\mathrm{N}}$ process. As an example we take a look at a labeled LySa$^{\mathrm{N}}$ specification of the running example protocol:

$$
\begin{array}{ll}
\multicolumn{2}{l}{(\nu\, A)(\nu\, B)(\nu\, S)} \\
 & [(\nu\, K)\langle A, B, \{K\}_{\mathsf{key}[A]}\rangle^{A_1}.(\nu\, Msg)\langle\{Msg\}_K\rangle^{A_2}.\mathrm{RET}] \\
| & [(\$z_A, \$z_B, \{\%z_K\}_{\mathsf{key}[z_A]})^{S_1}.\langle\{z_K\}_{\mathsf{key}[z_B]}\rangle^{S_2}.\mathrm{RET}] \\
| & [(\{\%y_K\}_{\mathsf{key}[B]})^{B_1}.(\{\%y_{Msg}\}_{y_K})^{B_2}.\mathrm{RET}]
\end{array}
$$

We see that each transition is decorated with a label. A label consists of the name of the principal together with an index. If for instance principal S goes from position S_1 to S_2 it corresponds to executing the communication step $(\$y_A, \$y_B, \{\%y_K\}_{\mathsf{key}[A]})$.

Generating the flow. We generate the flow by traversing each of the parallel processes of the LySa$^{\mathrm{N}}$ process. The process corresponding to principal S in the labeled LySa$^{\mathrm{N}}$ process above generates the following contribution to the flow.

$$
\begin{array}{l}
((S_1, S_2), (\$z_A, \$z_B, \{\%z_K\}_{\mathsf{key}[z_A]})) \\
((S_2, S_1), \langle\{z_K\}_{\mathsf{key}[z_B]}\rangle)
\end{array}
$$

Constructing the graph. The initial state of the running example protocol is $A_1.B_1.S_1$. This is the first state which is added to the graph. All children to all states added to the graph so far are found by finding matching steps in the flow. A step has the form $((l_1, l_2), \mathit{communication})$. If l_1 is equal to one of the labels of a state already in the graph, we can create a new state, where l_1 is replaced by l_2. The transition between the two states is created to contain the communication information of the matching step. This procedure is repeated until no new states can be created.

2.5 LySa and a finite graph tied together by LySa$^{\mathrm{N}}$

We have seen how LySa$^{\mathrm{N}}$ is used to connect the two representations of a protocol. LySa$^{\mathrm{N}}$ imposes some restrictions compared to the specification of a protocol in LySa. These restrictions are necessary to be able to efficiently represent a protocol as a static graph, which can be used by the model checking routine.

The fact that we use sequential looping processes in parallel, $[P_1] \mid [P_2] \mid \cdots$, instead of the more uncontrollable replication, $!\,P_1 \mid !\,P_2 \mid \cdots$, makes it possible to represent a protocol as a finite cyclic graph.

Omitting the decrypt construction lets us focus on communication. Hence we get a graph where each transition is a communication step. In an early stage of the project I considered having transitions corresponding to the explicit decrypt action as well, but the result would be an unnecessary amount of different interleavings of the set of parallel processes. If B communicates with I it does not matter whether it is done before or after S decrypts a received term, but a graph which includes transitions corresponding to explicit decryption will allow the unnecessary simulation of both possibilities.

As we have seen the $\mathsf{key}[\cdot]$-construction makes it possible to have one LySa$^{\mathrm{N}}$ process do the job of several parallel LySa processes. Again the motivation for the construction was an efficient graph representation. An increase in the number of parallel processes drastically affects the number of states and possible paths in the graph.

The introduction of the $\mathsf{key}[\cdot]$-construction also implies the need for variables in annotations. If a server process can serve any two principals, the annotations has to change dynamically to correspond to the principals involved in the communication. Variables in the $\mathsf{key}[\cdot]$-construction and in annotations give a need for principal variables. It has to be explicitly specified which variables are to be substituted by principal names when unfolding the LySa$^{\mathrm{N}}$ process.

When using the SAMC tool protocols are specified in LySa$^{\mathrm{N}}$ and converted into LySa and the graph representation respectively. Note that all protocols specified in LySa$^{\mathrm{N}}$ can be transformed into an almost equivalent process in LySa. It is equivalent in the sense that the over-approximation of the static analysis is still safe. In Section 3.2 we take a closer look at this relation and how we relate the analysis result of a LySa specification to the original LySa$^{\mathrm{N}}$ specification.

Chapter 3

Static Analysis

LySa tool is a tool for performing static control flow analysis of protocols specified in LySa.

Section 3.1 describes the static analysis in details. Section 3.2 describes the relation between the results obtained by analyzing a LySa protocol and the corresponding LySa$^{\mathrm{N}}$ specification. In section 3.3 we consider some details regarding the interfacing between the LySa tool and the rest of the samc tool.

3.1 Analysis - the LySa tool

The following description of the static analysis is based on the description in [1].

The analysis takes a LySa process as input and creates a triple, (ρ, κ, ψ) as output.

- $\rho : \mathcal{X} \rightarrow \wp(\mathcal{V})$ is a mapping from variables to sets of values. It gives an over-approximation to the set of values corresponding to each of the variables of a given process, P.
- $\kappa \subset \wp(\mathcal{V}^*)$ is an over-approximation to the set of tuples, which can be communicated on the network, when running a given process in parallel with the intruder.
- $\psi \in \wp(\mathcal{C} \times \mathcal{C})$ is a set of pairs of crypto points, (ℓ, ℓ'), indicating that an encryption made at ℓ was unexpectedly decrypted at ℓ'.

The judgments of Table 3.1 gives us a logic formula corresponding to a given LySa process. This formula describes properties that a triple, (ρ, κ, ψ), must have to be a valid analysis result for the given process. When we solve this formula, we find the smallest sets, ρ, κ, and ψ, which satisfy the formula.

$$\frac{\lfloor n \rfloor \in \vartheta}{\rho \models n : \vartheta} \qquad \frac{\rho(\lfloor x \rfloor) \subseteq \vartheta}{\rho \models x : \vartheta}$$

$$\frac{\begin{array}{c} \wedge_{i=0}^{k}\, \rho \models E_i : \vartheta_i \;\wedge \\ \forall V_0, V_1, \cdots, V_k : \wedge_{i=0}^{k} V_i \in \vartheta_i \;\Rightarrow\; \{V_1, \cdots, V_k\}_{V_0}^{\ell}[\mathsf{dest}\ \mathcal{L}] \in \vartheta \end{array}}{\rho \models \{E_1, \cdots, E_k\}_{E_0}^{\ell}\ [\mathsf{dest}\ \mathcal{L}]\ : \vartheta}$$

$$(\rho,\kappa) \models \mathsf{0} : \psi \qquad \frac{(\rho,\kappa) \models P : \psi}{(\rho,\kappa) \models (\nu\, n)P : \psi}$$

$$\frac{\begin{array}{c} \wedge_{i=1}^{k}\, \rho \models E_i : \vartheta_i \;\wedge \\ \forall V_1, \cdots, V_k :\ \wedge_{i=1}^{k} V_i \in \vartheta_i \;\Rightarrow\; \langle V_1, \cdots, V_k \rangle \in \kappa \;\wedge \\ (\rho,\kappa) \models P : \psi \end{array}}{(\rho,\kappa) \models \langle E_1, \cdots, E_k \rangle.P : \psi}$$

$$\frac{\begin{array}{c} \wedge_{i=1}^{j}\, \rho \models E_i : \vartheta_i \;\wedge \\ \forall \langle V_1, \cdots, V_k \rangle \in \kappa :\ \wedge_{i=1}^{j} V_i\ \mathsf{E}\ \vartheta_i \Rightarrow\ \wedge_{i=j+1}^{k} V_i \in \rho(\lfloor x_i \rfloor) \;\wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad (\rho,\kappa) \models P : \psi \end{array}}{(\rho,\kappa) \models (E_1, \cdots, E_j; x_{j+1}, \cdots, x_k).P : \psi}$$

$$\frac{(\rho,\kappa) \models P_1 : \psi \;\wedge\; (\rho,\kappa) \models P_2 : \psi}{(\rho,\kappa) \models P_1 | P_2 : \psi} \qquad \frac{(\rho,\kappa) \models P : \psi}{(\rho,\kappa) \models\, !\, P : \psi}$$

$$\frac{\begin{array}{l} \qquad\qquad\qquad\qquad \rho \models E : \vartheta \;\;\wedge\;\; \wedge_{i=0}^{j}\, \rho \models E_i : \vartheta_i \;\wedge \\ \forall\ \{V_1, \cdots, V_k\}_{V_0}^{\ell}[\mathsf{dest}\ \mathcal{L}] \in \vartheta :\ \wedge_{i=0}^{j} V_i\ \mathsf{E}\ \vartheta_i \;\Rightarrow \wedge_{i=j+1}^{k} V_i \in \rho(\lfloor x_i \rfloor) \;\wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \neg(\ell \in \mathcal{L}' \wedge \ell' \in \mathcal{L} \Rightarrow (\ell, \ell') \in \psi) \;\wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\rho,\kappa) \models P : \psi \end{array}}{(\rho,\kappa) \models \mathsf{decrypt}\ E\ \mathsf{as}\ \{E_1, \cdots, E_j; x_{j+1}, \cdots, x_k\}_{E_0}^{\ell'}\ [\mathsf{orig}\ \mathcal{L}']\ \mathsf{in}\ P : \psi}$$

Table 3.1: Analysis of terms, $\rho \models E : \vartheta$, and analysis of processes, $(\rho,\kappa) \models P : \psi$.

Terms. For each term E, the analysis will determine a superset of the possible values that it may evaluate to. The judgment for expressions takes the form

$$\rho \models E : \vartheta$$

and expresses that $\vartheta \subseteq \mathcal{V}$ is an acceptable estimate of the set of values that E may evaluate to as specified by the mapping ρ. The judgment is defined by the three rules in the upper part of Table 3.1. The three rules relates to terms of the type name, variable and encryption respectively. The rule corresponding to a term of type name is trivial. ϑ is an acceptable estimate of the values of the term n if $n \in \vartheta$. Likewise with the rule corresponding to a term of type variable. ϑ is an acceptable estimate of the values of the term x with respect to ρ if $\rho(x) \in \vartheta$. The rule corresponding to an encryption term expresses that we find an acceptable estimate ϑ to the values of an encryption term, E, by finding all possible values for each of the terms $E_0, \cdots, E_k$ in E, and creating all possible encryption terms from all possible combinations of these values.

Processes. Judgements for processes has the form

$$(\rho, \kappa) \models P : \psi$$

and are defined by the rules and axioms in the lower part of Table 3.1. In these rules we shall use two kinds of membership tests: the usual $V \in \vartheta$ that simply tests whether V is in the set ϑ and the *faithful* test $V \mathrel{\mathsf{E}} \vartheta$ that holds if there is a value V' in ϑ that equals V when the annotations are ignored:

$$V \mathrel{\mathsf{E}} \vartheta \quad \text{iff} \quad \exists V' \in \vartheta : \llbracket V \rrbracket = \llbracket V' \rrbracket$$

In the analysis of processes we focus on which values can flow on the network:

- $\kappa \subseteq \wp(\mathcal{V}^*)$: the network environment that includes all the message sequences that may flow on the network.

To obtain this information we use the environment ρ, as done for terms. The mismatches between annotations describing the intended origin and destination for encryption terms are collected in the set ψ.

The rules and axioms defining the judgment for processes are explained below.

Since we want to find an over-approximation to ρ, κ and ψ it follows that all values are valid for the inactive process, $\mathsf{0}$.

The analysis of $(\nu\, n)P$ is equal to the analysis of P. The restriction only influence the initial knowledge of the intruder.

The rule corresponding to the analysis of a k-ary output followed by the process P, $\langle E_1, \cdots, E_k \rangle.P$, states that all tuples of the form $(V_1, \cdots, V_k)$ must be in κ, where $V_1, \cdots, V_k$ are taken from the sets $\vartheta_1, \cdots, \vartheta_k$ obtained by analyzing each of the terms $E_1, \cdots, E_k$. Furthermore the rule requires that (ρ, κ, ψ) are also valid analysis estimates of process P.

In the rule for *input* of the form $(E_1, \cdots, E_j; x_{j+1}, \cdots, x_k)$ we use the estimates of the value for each of the terms $E_1, \cdots, E_j$ to determine which of the messages predicted to flow on the network can potentially match the input. The environment ρ is updated to reflect the potential matches for each of the variables, $x_{j+1}, \cdots, x_k$. Note that annotations are ignored at this point by using the faithful membership predicate.

The rule for *decryption* handles the matching similarly to the rule for input but instead of trying to match the encryption with all tuples on the network we try to match it with all terms in ϑ, which is an estimate of the values of E. When matching, we check whether the values $V_0, \ldots . V_j$ are point wise included into the values in ϑ_i (including the key). Again we use the faithful membership tests for matching since the semantics ignores the annotations. If the check is successful then the values predicted for the variables x_i should point wise contain the values V_i. Finally the ψ-component of the analysis is updated to include any mismatch between the annotations.

The rules for *parallel composition*, and *replication* ensure that the analysis also holds for the immediate subprocesses.

The intruder. To analyze the security properties of a protocol we have to consider the protocol in parallel with a process corresponding to the intruder. The intruder could be represented by a LySa process but is in fact represented by a corresponding formula which is solved together with the formula generated by the rules of Table 3.1. Table 3.2 gives the formula representing the Dolev-Yao attacker.

$$
\begin{array}{ll}
(1) & \wedge_{k \in \mathcal{A}_\kappa} \forall \langle V_1, \cdots, V_k \rangle \in \kappa : \wedge_{i=1}^{k} V_i \in \rho(z_\bullet) \\
(2) & \wedge_{k \in \mathcal{A}^+_{\mathsf{Enc}}} \forall \{V_1, \cdots, V_k\}^{\ell}_{V_0} [\mathsf{dest}\ \mathcal{L}] \in \rho(z_\bullet) : \\
& \quad V_0 \mathrel{\mathsf{E}} \rho(z_\bullet) \Rightarrow (\wedge_{i=1}^{k} V_i \in \rho(z_\bullet) \wedge (\neg(\ell \in \mathcal{C} \wedge \ell_\bullet \in \mathcal{L}) \Rightarrow (\ell, \ell_\bullet) \in \psi)) \\
(3) & \wedge_{k \in \mathcal{A}^+_{\mathsf{Enc}}} \forall V_0, \cdots, V_k : \wedge_{i=0}^{k} V_i \in \rho(z_\bullet) \Rightarrow \{V_1, \cdots, V_k\}^{\ell_\bullet}_{V_0} [\mathsf{dest}\ \mathcal{C}] \in \rho(z_\bullet) \\
(4) & \wedge_{k \in \mathcal{A}_\kappa} \forall V_1, \cdots, V_k : \wedge_{i=1}^{k} V_i \in \rho(z_\bullet) \Rightarrow \langle V_1, \cdots, V_k \rangle \in \kappa \\
(5) & \{n_\bullet\} \cup \mathcal{N} \subseteq \rho(z_\bullet)
\end{array}
$$

Table 3.2: Dolev-Yao intruder.

$z_\bullet$ is a variable of the intruder. $\mathcal{N}$ is the set of names known to the intruder initially. $\mathcal{A}_\kappa$ and $\mathcal{A}^+_{\mathsf{Enc}}$ are sets of integers chosen by looking at the specific protocol to be analyzed. In [1] it is shown that we do not limit the capabilities of the intruder by limiting the arity of input and output tuples and the arity of encryption terms we consider as long as the limit is based on the given protocol. For details on how to choose $\mathcal{A}_\kappa$ and $\mathcal{A}^+_{\mathsf{Enc}}$, see [1].

The formula in Table 3.2 specifies that the attacker initially has some knowledge (5), that it may learn more by eavesdropping (1) or by decrypting messages with keys already known (2), that it may construct new encryptions using the keys known (3) and that it may actively forge new communications (4).

Analyzing an example. As an example we look at the analysis result for a simple LYSA process.

$$
\begin{array}{ll}
 & (\nu\, K_{AB}) \\
 & \quad (\nu\, M_S)(\nu\, M_P)\langle\{M_S\}^{a_1}_{K_{AB}}\ [\mathsf{dest}\ b]\ \rangle.\langle\{M_P\}^{a_2}_{K_{AB}}\ [\mathsf{dest}\ b]\ \rangle.0 \\
| & \quad (;x_1).(;x_2).\mathsf{decrypt}\ x_2\ \mathsf{as}\ \{x_P\}^{b}_{K_{AB}}\ [\mathsf{orig}\ a_2]\ \mathsf{in}\ \langle x_P\rangle
\end{array}
$$

corresponding to the narration:

$$
\begin{array}{ll}
A \to B & : \{M_S\}_{K_{AB}} \\
A \to B & : \{M_P\}_{K_{AB}} \\
B \to \mathit{any} & : P
\end{array}
$$

The idea of the process is that M_S should be kept secret and M_P should be known to anyone interested (in particular the intruder). However the following narration illustrates an attack where the attacker switches the two messages. This is possible because the form of the messages are identical.

$$
\begin{array}{lll}
A & \to I(B) & : \{M_S\}_{K_{AB}} \\
A & \to I(B) & : \{M_P\}_{K_{AB}} \\
I(A) & \to B & : \{M_P\}_{K_{AB}} \\
I(A) & \to B & : \{M_S\}_{K_{AB}} \\
B & \to I & : P
\end{array}
$$

The analysis result is as follows, (ρ is omitted):

$$
\begin{array}{lcl}
\kappa & = & \{\langle M_P\rangle, \langle M_S\rangle, \langle\{M_P\}_{K_{AB}}\rangle, \langle\{M_S\}_{K_{AB}}\rangle\} \\
\psi & = & \{(a_1, b)\}
\end{array}
$$

The analysis result confirms the attack. Both from the element in ψ and the fact that the term M_S is on the network (and consequently in the knowledge of the intruder).

3.2 Relating the analysis result to the LYSA$^{\text{N}}$ specification

We have defined a process algebra, LYSA$^{\text{N}}$ and a transformation, which transforms a LYSA$^{\text{N}}$ process into a similar LYSA process. The purpose of this transformation is to make it possible to use the existing LYSA analysis to determine properties of a LYSA$^{\text{N}}$ process.

In the following we consider a subset of LYSA$^{\text{N}}$ that does not allow principal variables. As we have seen there is a straight forward transformation which removes the principal variables of a LYSA$^{\text{N}}$ process. When principal variables are removed from a LYSA$^{\text{N}}$ process the process is also part of the LYSA$^{\text{N}}_{ext}$ language, which is why we can apply the transformation directly. For convenience we disregard these details in the following and proceed as if the transformation $\lfloor\cdot\rfloor$ simply has the type $\text{LySa}^{\text{N}} \rightarrow \text{LySa}$.

As we have seen, LySa and LySa$^{\text{N}}$ are two different languages but they are similar. The concepts of a network component, κ, a mapping of variables, ρ, and an error component, ψ, make sense for LySa as well as for LySa$^{\text{N}}$ even though the semantics of the processes are not identical.

We introduce the following definition:

$$(\kappa, \psi) \models_N S \Leftrightarrow (\rho, \kappa) \models \lfloor S \rfloor : \psi$$

expressing that κ and ψ are valid analysis estimates for the LySa$^{\text{N}}$ process S if and only if they are valid estimates for the transformed process, $\lfloor S \rfloor$. The mapping, ρ, is regarded as a secondary component, which is only used in the static analysis to calculate the two other components. We are therefore not interested in ρ in the analysis result for a LySa$^{\text{N}}$ process.

With the definition above we express that we directly use the values obtained by analyzing $\lfloor S \rfloor$ to express properties of the LySa$^{\text{N}}$ process S.

To capture the exact meaning of the components of the analysis in LySa and LySa$^{\text{N}}$ respectively we formulate the annotated version of the semantics in tables 3.3 and 3.4.

(Com)

$$\frac{\wedge_{i=1}^{j} \llbracket E_i \rrbracket = \llbracket E'_i \rrbracket}{\langle E_1, \cdots, E_k \rangle.P \mid (E'_1, \cdots, E'_j; x_{j+1}, \cdots, x_k).Q \xrightarrow{(\langle E_1, \cdots, E_k \rangle, \emptyset)} P \mid Q[E_{j+1}/x_{j+1}, \cdots, E_k/x_k]}$$

(Par)

$$\frac{P \xrightarrow{(m,A)} P'}{P \mid Q \xrightarrow{(m,A)} P' \mid Q}$$

(Decr)

$$\frac{\wedge_{i=0}^{j} \llbracket E_i \rrbracket = \llbracket E'_i \rrbracket}{\mathsf{decrypt}\ \{E_1, \cdots, E_k\}^{\ell}_{E_0}\ [\mathsf{dest}\ \mathcal{L}]\ \ \mathsf{as}\ \{E'_1, \cdots, E'_j; x_{j+1}, \cdots, x_k\}^{\ell'}_{E'_0}\ [\mathsf{orig}\ \mathcal{L}']\ \mathsf{in}\ P \xrightarrow{(\epsilon, \mathsf{AC}(\ell, \mathcal{L}, \ell', \mathcal{L}'))} P[E_{j+1}/x_{j+1}, \cdots, E_k/x_k]}$$

(Res)

$$\frac{P \xrightarrow{(m,A)} P'}{(\nu\, n)P \xrightarrow{(m,A)} (\nu\, n)P'}$$

(Congr)

$$\frac{P \equiv Q\ \wedge\ Q \xrightarrow{(m,A)} Q'\ \wedge\ Q' \equiv P'}{P \xrightarrow{(m,A)} P'}$$

Table 3.3: Operational semantics, $P \xrightarrow{(m,A)} P'$ for LySa.

Each of the transitions are annotated with a pair, (m, A). m is a message and A is a set of origin-destination annotation mismatches corresponding to a given transition. Table 3.4 contains an annotated version of the judgment of pattern matching. The judgment now has the form $\theta \vdash v \triangleright p : \theta', A$. θ' is still the extended environment and A is a set of origin-destination mismatches created by the match. The function

$$\mathsf{AC}(\ell, \mathcal{L}, \ell', \mathcal{L}') = \text{ if } \ell \notin \mathcal{L}' \vee \ell' \notin \mathcal{L} \text{ then } \{(\ell, \ell')\} \text{ else } \emptyset$$

$$\theta \vdash n \rhd n : \theta, \emptyset \qquad \theta \vdash v \rhd x : \theta, \emptyset \;\text{ if } v = \theta x$$

$$\frac{\theta \vdash v \rhd x : \theta, A}{\theta \vdash key[v] \rhd key[x] : \theta, A} \qquad \begin{array}{l} \theta \vdash v \rhd \%x : \theta[y \mapsto v], A \\ \theta \vdash v \rhd \$x : \theta[y \mapsto v], A \end{array}$$

$$\frac{\theta \vdash v_0 \rhd p_0 : \theta_0, A \wedge \theta_0 \vdash v_1 \rhd p_1 : \theta_1, A_1 \wedge \cdots \wedge \theta_{k-1} \vdash v_k \rhd p_k : \theta_k, A_k}{\begin{array}{ll} \theta \vdash & \{v_1, \cdots, v_k\}_{v_0}^{v_{cp}} \; [\mathsf{dest}\ v_{dest}] \rhd \\ & \{p_1, \cdots, p_k\}_{p_0}^{p_{cp}} \; [\mathsf{orig}\ p_{orig}] \; : \theta_k, \mathsf{AC_N}(v_{cp}, p_{orig}, p_{cp}, v_{dest}) \cup A_k \end{array}}$$

(Com)
$$\frac{\emptyset \vdash t_1 \rhd p_1 : \theta_1, A_1 \wedge \theta_1 \vdash t_2 \rhd p_2 : \theta_2, A_2 \wedge \cdots \wedge \theta_{k-1} \vdash t_k \rhd p_k : \theta_k, A_k}{[\langle t_1, \cdots, t_k\rangle.P_1] \mid [(p_1, \cdots, p_k).P_2] \xrightarrow{(\langle t_1, \cdots, t_k\rangle, A_1 \cup \cdots \cup A_k)} [P_1] \mid [\theta_k P_2]}$$

(Par)
$$\frac{S_1 \xrightarrow{(m,A)} S_1'}{S_1 \mid S_2 \xrightarrow{(m,A)} S_1' \mid S_2}$$

(Res)
$$\frac{S \xrightarrow{(m,A)} S'}{(\nu\, n)S \xrightarrow{(m,A)} (\nu\, n)S'}$$

(Congr)
$$\frac{S \equiv R \quad R \xrightarrow{(m,A)} R' \quad R' \equiv S'}{S \xrightarrow{(m,A)} S'}$$

(Loop)
$$\mathsf{loop}(P) \xrightarrow{(\epsilon,\emptyset)} P[\mathsf{loop}(P)/\textsc{ret}]$$

Table 3.4: Operational semantics, $S \xrightarrow{(m,A)} S'$ and $P \xrightarrow{(m,A)} P'$ for LySa$^{\text{N}}$.

compares the annotated crypto-points. A similar function is defined for the annotated semantics of LySa$^{\text{N}}$:

$$\mathsf{AC_N}(v_{cp}, p_{orig}, p_{cp}, v_{orig}) = \text{ if } v_{cp} \neq p_{orig} \vee p_{cp} \neq v_{orig} \text{ then } \{(v_{cp}, p_{cp})\} \text{ else } \emptyset$$

As an example consider the following Com-transition in LySa$^{\text{N}}$:

$$\langle \{Msg\}_K^S \; [\mathsf{dest}\ B] \; \rangle.P \mid (\{\%x_{Msg}\}_K^B \; [\mathsf{orig}\ S'] \;).Q \xrightarrow{(m,A)} P \mid Q$$

In this case, $m = \langle \{Msg\}_K^S \; [B] \; \rangle$ and $A = \{(S, B)\}$ (if $S \neq S'$)

The annotated semantics expresses the properties that we expect the analysis to fulfill.

To show that our definition of a valid analysis estimate for a LySa$^{\text{N}}$ process actually fulfills the properties we expect, we need to show the following theorem:

Theorem 1 (Analysis of LySa$^{\mathrm{N}}$) *The analysis of LySa$^{\mathrm{N}}$ fulfills the properties we expect as expressed by the annotated semantics.*

$$\textit{If } S \xrightarrow{(m,A)} S' \textit{ then } (\kappa,\psi) \models_N S \Rightarrow (\kappa,\psi) \models_N S' \wedge m \in \kappa \wedge A \subseteq \psi$$

Theorem 2 (Analysis of LySa) *The analysis of LySa fulfills the properties we expect as expressed by the annotated semantics.*

$$\textit{If } P \xrightarrow{(m,A)} P' \textit{ then } (\rho,\kappa) \models P : \psi \Rightarrow (\rho,\kappa) \models P'\psi \wedge m \in \kappa \wedge A \subseteq \psi$$

The theorems expresses that no matter what derivation sequence we choose in our protocol, we have a set, κ, containing all communicated messages and a set, ψ, containing pairs corresponding to all mismatches in origin-destination annotations.

The proof of Theorem 2 is done partly in [1]. It can be completed by using the definition of the static analysis in Table 3.1. We assume the correctness of Theorem 2 when proving Theorem 1.

We prove Theorem 1 by doing a case split corresponding to each of the transition rules in LySa$^{\mathrm{N}}$. For each possible transition

$$S \xrightarrow{(m,A)}_N S'$$

we show two properties:

1. There exists a sequence of transitions in LySa

   $$\lfloor S \rfloor \xrightarrow{(m_1,A_1)} \cdots \xrightarrow{(m_k,A_k)} Q'$$

   such that $m \in \{m_1, \cdots, m_k\}$ and $A \subseteq \bigcup_i A_i$.

2. The two derivation sequences, $S \xrightarrow{(m,A)} S'$ and $\lfloor S \rfloor \xrightarrow{(m_1,A_1)} \cdots \xrightarrow{m_k,A_k} Q'$ are also similar in the sense that they "end" in similar processes. I.e.

   $$(\rho,\kappa) \models Q' : \psi \Leftrightarrow (\rho,\kappa) \models \lfloor P' \rfloor : \psi$$

In other words, for any transition in LySa$^{\mathrm{N}}$, there exists a derivation sequence in LySa which is equivalent with regard to the analysis. Showing these two properties for each of the possible transitions in LySa$^{\mathrm{N}}$ is equivalent to showing Theorem 1.

Lemma 1 (Congruence, analysis)

$$P_1 \equiv P_2 \textit{ iff } (\rho,\kappa) \models P_1 : \psi \Leftrightarrow (\rho,\kappa) \models P_2 : \psi$$

Proof. See [1].

Lemma 2 (Congruence, transformation)

$$S_1 \equiv_N S_2 \Rightarrow \lfloor S_1 \rfloor \equiv \lfloor S_2 \rfloor \qquad \textit{and} \qquad P_1 \equiv_N P_2 \Rightarrow \lfloor P_1 \rfloor \equiv \lfloor P_2 \rfloor$$

Proof. See Appendix B.

The definition of the semantics for LySa and LySa$^{\mathrm{N}}$ uses two different but equivalent ways of representing a substitution. θP is equal to $P[v_1/x_1, \cdots, v_k/x_k]$ for $\theta = [x_1 \mapsto v_1, \cdots, x_k \mapsto v_k]$. Furthermore, we have that $\lfloor S \rfloor[v_1/x_1, \cdots, v_k/x_k] = \lfloor S[v_1/x_1, \cdots, v_k/x_k] \rfloor$, which can be shown by structural induction in S.

Proof of Theorem 1. Below we consider each of the five transition rules of LySa$^{\mathrm{N}}$ and show that any transition in LySa$^{\mathrm{N}}$ fulfills the properties 1 and 2 defined above. Some of the semantical rules of LySa$^{\mathrm{N}}$ are based on conditions. A structural induction on the size of the derivation tree is performed.

Case: Communication-rule. We simplify the input and output tuples of the communication transitions to contain only two terms. The proof can be extended to work for arbitrary matching tuples. We look at the transition

$$\underbrace{[\langle n_1, \{n_3\}_{n_2}^{n_{cp}}\ [\mathsf{dest}\ n_{dest}]\ \rangle.P_1] \mid [(n_1, \{\%y\}_{n_2}^{n'_{cp}}\ [\mathsf{orig}\ n_{orig}]\ ).P_2]}_{S} \xrightarrow{(m,A)}_N \underbrace{[P_1] \mid [\theta P_2]}_{S'}$$

where $\theta = [y \mapsto n_3]$. We have that $m = \langle n_1, \{n_3\}_{n_2}^{n_{cp}}\ [\mathsf{dest}\ n_{dest}]\ \rangle$ and $A = \mathsf{AC_N}(n_{cp}, n_{orig}, n'_{cp}, n_{dest})$.

$$\begin{array}{l} \lfloor S \rfloor = \\ \langle n_1, \{n_3\}_{n_2}^{n_{cp}}\ [\mathsf{dest}\ \{n_{dest}\}]\ \rangle.\lfloor P_1 \rfloor \\ \mid (n_1; x_1).\mathsf{decrypt}\ x_1\ \mathsf{as}\ \{;y\}_{n_2}^{n'_{cp}}\ [\mathsf{orig}\ \{n_{orig}\}]\ \mathsf{in}\ \lfloor P_2 \rfloor \end{array}$$

We construct the derivation sequence

$$\begin{array}{l} \langle n_1, \{n_3\}_{n_2}^{n_{cp}}\ [\mathsf{dest}\ \{n_{dest}\}]\ \rangle.\lfloor P_1 \rfloor \\ \mid (n_1; x_1).\mathsf{decrypt}\ x_1\ \mathsf{as}\ \{;y\}_{n_2}^{n'_{cp}}\ [\mathsf{orig}\ \{n_{orig}\}]\ \mathsf{in}\ \lfloor P_2 \rfloor \\ \xrightarrow{(m_1,A_1)} \lfloor P_1 \rfloor \mid \mathsf{decrypt}\ \{n_3\}_{n_2}^{n_{cp}}\ [\mathsf{dest}\ \{n_{dest}\}]\ \ \mathsf{as}\ \{;y\}_{n_2}^{n'_{cp}}\ [\mathsf{orig}\ \{n_{orig}\}]\ \mathsf{in}\ \lfloor P_2 \rfloor \\ \xrightarrow{(m_2,A_2)} \lfloor P_1 \rfloor \mid \lfloor P_2 \rfloor[n_3/y] \end{array}$$

where

$$\begin{array}{l} (m_1, A_1) = (\langle n_1, \{n_3\}_{n_2}^{n_{cp}}\ [\mathsf{dest}\ \{n_{dest}\}]\ \rangle, \emptyset) \text{ and} \\ (m_2, A_2) = (\epsilon, \mathsf{AC}(n_{cp}, \{n_{orig}\}, n'_{cp}, \{n_{dest}\})) \end{array}$$

since $m = m_1$ and $A = A_2$ we have shown property 1.

$$\begin{array}{l} Q' = \lfloor P_1 \rfloor \mid \lfloor P_2 \rfloor[n_3/y] = \lfloor P_1 \rfloor \mid \lfloor P_2[n_3/y] \rfloor \\ \lfloor S' \rfloor = \lfloor [P_1] \mid [\theta P_2] \rfloor = \lfloor P_1 \rfloor \mid \lfloor P_2[n_3/y] \rfloor \end{array}$$

We have shown that $Q' = \lfloor S' \rfloor$. This implies that property 2 holds. Namely that

$$(\rho, \kappa) \models Q' : \psi \Leftrightarrow (\rho, \kappa) \models \lfloor S' \rfloor : \psi$$

Both properties has been shown.

Case: Congruence-rule. In LySA$^{\mathrm{N}}$ we have the transition $S \xrightarrow{(m,A)} S'$ if $S \equiv R \wedge R \xrightarrow{(m,A)} R' \wedge R' \equiv S'$. Assuming as the inductive hypothesis that properties 1 and 2 holds for the transition $R \xrightarrow{(m,A)} R'$ and that $S \equiv R \wedge R' \equiv S'$ we show that properties 1 and 2 holds for the transition $S \xrightarrow{(m,A)} S'$.

From the inductive hypothesis we have that

$$\lfloor R \rfloor \xrightarrow{(m_1,A_1)} \dots \xrightarrow{(m_k,A_k)} Q' \text{ and } m \in \{m_1, \cdots, m_k\} \wedge A \subseteq \bigcup_i A_i$$

Furthermore,

$$(\rho, \kappa) \models Q' : \psi \Leftrightarrow (\rho, \kappa) \models \lfloor R' \rfloor : \psi \tag{3.1}$$

Since $S' \equiv R'$ we have from Lemma 2 that $\lfloor S' \rfloor \equiv \lfloor R' \rfloor$.

By applying the transition rule congr of LySA a number of times we can construct a sequence $\lfloor P \rfloor \xrightarrow{(m_1,A_1)} \dots \xrightarrow{(m_k,A_k)} Q'$

From (3.1) we have that $Q' \equiv \lfloor R' \rfloor \equiv \lfloor S' \rfloor$ which implies by Lemma 1 that $(\rho, \kappa) \models \lfloor S' \rfloor : \psi \Leftrightarrow (\rho, \kappa) \models Q' : \psi$

Hence we have shown both properties for the transition based on the congruence rule.

Case: Restriction-rule. We consider the transition

$$\underbrace{(\nu\, n)R}_{S} \xrightarrow{(m,A)} \underbrace{(\nu\, n)R'}_{S'}$$

As the inductive hypothesis we assume that $R \xrightarrow{(m,A)} R'$ and that the properties 1 and 2 holds for this transition.

From the inductive hypothesis we have that that there exists a derivation sequence

$$\lfloor R \rfloor \xrightarrow{(m_1,A_1)} \dots \xrightarrow{(m_k,A_k)} \overline{Q}'$$

such that

$$m \in \{m_1, \cdots, m_k\} \wedge A \subseteq \bigcup_i A_i$$

furthermore,

$$(\rho, \kappa) \models \lfloor R' \rfloor : \psi \Leftrightarrow (\rho, \kappa) \models \overline{Q}' : \psi \tag{3.2}$$

We have from the definition of $\lfloor \cdot \rfloor$ that

$$\begin{array}{ll} \lfloor S \rfloor = \lfloor (\nu\, n)R \rfloor = (\nu\, n)\lfloor R \rfloor & \text{and} \\ \lfloor S' \rfloor = \lfloor (\nu\, n)R' \rfloor = (\nu\, n)\lfloor R' \rfloor & \end{array}$$

The semantical rule (Res) of LySA in Table 3.3 implies that there exists a derivation sequence

$$\lfloor S \rfloor \xrightarrow{(m_1,A_1)} \dots \xrightarrow{(m_k,A_k)} Q'$$

such that $Q' = (\nu\, n)\overline{Q}'$. Property 1 is hereby shown.

From (3.2) and the analysis rule

$$\frac{(\rho,\kappa) \models P : \psi}{(\rho,\kappa) \models (\nu\, n)P : \psi}$$

in Table 3.1 we have that

$$(\rho,\kappa) \models \lfloor S' \rfloor : \psi \Leftrightarrow (\rho,\kappa) \models Q' : \psi$$

Property 2 is hereby shown.

Case: Parallel-rule. We consider the transition

$$\underbrace{S_1 \mid S_2}_{S} \xrightarrow{(m,A)} \underbrace{S_1' \mid S_2}_{S'}$$

As the inductive hypothesis we assume that $S_1 \xrightarrow{(m,A)} S_1'$ and that the properties 1 and 2 holds for this transition.

From the inductive hypothesis we have that that there exists a derivation sequence

$$\lfloor S_1 \rfloor \xrightarrow{(m_1,A_1)} \ldots \xrightarrow{(m_k,A_k)} Q_1'$$

such that

$$m \in \{m_1, \ldots, m_k\} \wedge A \subseteq \bigcup_i A_i$$

furthermore,

$$(\rho,\kappa) \models \lfloor S_1' \rfloor : \psi \Leftrightarrow (\rho,\kappa) \models Q_1' : \psi \tag{3.3}$$

We have from the definition of $\lfloor \cdot \rfloor$ that

$$\begin{array}{ll} \lfloor S \rfloor = \lfloor S_1 \mid S_2 \rfloor = \lfloor S_1 \rfloor \mid \lfloor S_2 \rfloor & \quad \text{and} \\ \lfloor S' \rfloor = \lfloor S_1' \mid S_2 \rfloor = \lfloor S_1' \rfloor \mid \lfloor S_2 \rfloor & \end{array}$$

The semantical rule (Par) of LySA in Table 3.3 implies that there exists a derivation sequence

$$\lfloor S \rfloor \xrightarrow{(m_1,A_1)} \ldots \xrightarrow{(m_k,A_k)} Q'$$

such that $Q' = Q_1' \mid \lfloor S_2 \rfloor$. Property 1 is hereby shown.

From (3.3) and the analysis rule

$$\frac{(\rho,\kappa) \models P_1 : \psi \ \wedge \ (\rho,\kappa) \models P_2 : \psi}{(\rho,\kappa) \models P_1 | P_2 : \psi}$$

in Table 3.1 we have that

$$(\rho,\kappa) \models \lfloor S' \rfloor : \psi \Leftrightarrow (\rho,\kappa) \models Q' : \psi$$

Property 2 is hereby shown.

Case: Loop-rule. We consider the transition

$$\underbrace{\mathsf{loop}(a_1.\cdots.a_k.\text{RET})}_{S} \xrightarrow{(m,A)} \underbrace{a_1.\cdots.a_k.\mathsf{loop}(a_1.\cdots.a_k.\text{RET})}_{S'}$$

where a_i are input or output steps (*actions*) as defined by the grammar of LySa$^{\text{N}}$. From the definition of $\lfloor\cdot\rfloor$ we have that

$$\begin{aligned}
\lfloor S\rfloor &= \lfloor \mathsf{loop}(a_1.\cdots.a_k.\text{RET})\rfloor =! \lfloor a_1.\cdots.a_k.\text{RET}\rfloor \\
&= !\,\overline{a}_{11}.\overline{a}_{12}.\cdots.\overline{a}_{1j_1}.\overline{a}_{21}.\overline{a}_{22}.\cdots.\overline{a}_{2j_2}.\cdots.\overline{a}_{kj_k}.\mathsf{0} \qquad \text{and} \\
\lfloor S'\rfloor &= \lfloor a_1.\cdots.a_k.\mathsf{loop}(a_1.\cdots.a_k.\text{RET})\rfloor \\
&= \overline{a}_{11}.\overline{a}_{12}.\cdots.\overline{a}_{1j_1}.\overline{a}_{21}.\overline{a}_{22}.\cdots.\overline{a}_{2j_2}.\cdots.\overline{a}_{kj_k}.\lfloor \mathsf{loop}(a_1.\cdots.a_k.\text{RET})\rfloor \\
&= \overline{a}_{11}.\overline{a}_{12}.\cdots.\overline{a}_{1j_1}.\overline{a}_{21}.\overline{a}_{22}.\cdots.\overline{a}_{2j_2}.\cdots.\overline{a}_{kj_k}. \\
&\quad\; !\,\overline{a}_{11}.\overline{a}_{12}.\cdots.\overline{a}_{1j_1}.\overline{a}_{21}.\overline{a}_{22}.\cdots.\overline{a}_{2j_2}.\cdots.\overline{a}_{kj_k}.\mathsf{0}
\end{aligned}$$

where $\lfloor a_i.\text{RET}\rfloor = \overline{a}_{i1}.\cdots.\overline{a}_{ij_i}.\mathsf{0}$. I.e. we transform each of the actions a_i into the equivalent LySa representation. The use of index reflects that the transformation of one LySa$^{\text{N}}$ action potentially results in more than one LySa action.

Since m $= \epsilon$ and $A = \emptyset$ as seen from Table 3.4, property 1 is trivially true for an empty derivation sequence.

Two show property two for an empty derivation sequence we simply show that

$$(\rho,\kappa) \models \lfloor S\rfloor : \psi \Leftrightarrow (\rho,\kappa) \models \lfloor S'\rfloor : \psi$$

To enhance readability we write $\vec{a}$ in place of the expression

$$\overline{a}_{11}.\overline{a}_{12}.\cdots.\overline{a}_{1j_1}.\overline{a}_{21}.\overline{a}_{22}.\cdots.\overline{a}_{2j_2}.\cdots.\overline{a}_{kj_k}$$

So we have that

$$\begin{aligned}
\lfloor S\rfloor &= !\,\vec{a}\,.\mathsf{0} \qquad \text{and} \\
\lfloor S'\rfloor &= \vec{a}\,.!\,\vec{a}\,.\mathsf{0}
\end{aligned}$$

Due to the fact that the analysis is flow insensitive we can conclude that

$$(\rho,\kappa) \models \lfloor S\rfloor : \psi \Leftrightarrow (\rho,\kappa) \models \lfloor S'\rfloor : \psi$$

Property 2 is hereby shown.

Summary. We have for each transition, $S \rightarrow S'$, in LySa$^{\text{N}}$ shown that there exists a sequence of transitions $\lfloor S\rfloor \rightarrow \cdots \rightarrow Q'$ in LySa, such that the expected contribution to the analysis from $S \rightarrow S'$ is a subset of that of $\lfloor S\rfloor \rightarrow \cdots \rightarrow Q'$. Furthermore the analysis result for the resulting processes $\lfloor S'\rfloor$ and Q' are identical. This concludes the proof of Theorem 1.

3.3 Interpreting and processing the LySa tool output

In the previous sections where I describe the theory behind the control flow analysis I do not look at the actual implementation of the analysis. For implementational details of the analysis I refer the reader to [3]. However I will take a closer look at one aspect of the implementation. Namely the representation of the output from the LySa tool. This aspect is interesting, since we have to process and interpret the output the right way in order to be able to interface the LySa tool with the rest of the SAMC tool.

The number of elements in kappa is infinite due to the definition of the intruder. The same is the case for the values $\rho(x)$ corresponding to a given variable x. Consider the following LySa process as an example:

$$\langle n\rangle.0 \mid !\,(;x).\langle\{x\}_K\rangle.0$$

The corresponding network component, κ, will be an infinite set of encryptions of arbitrary deep levels, $\{...\{V\}_K...\}_K$ where $V \in \rho(x)$.

In LySa tool these infinite sets are represented using a tree grammar. This is implemented by adding labels to the terms of the processes as follows:

$$\langle n^{l_1}\rangle.0 \mid !\,(;x).\langle\{x^{l_2}\}^{l_4}_{K^{l_3}}\rangle.0$$

These labels are considered as nonterminals of a tree grammar. We can now state that $\{l_l, l_4\} \subseteq \kappa$ and specify the infinite set by stating the following grammar:

$$\begin{array}{lcl} l_1 & \rightarrow & n \\ l_2 & \rightarrow & n \\ l_2 & \rightarrow & \{l_2\}_{l_3} \\ l_3 & \rightarrow & K \\ l_4 & \rightarrow & \{l_2\}_{l_3} \end{array}$$

We have a corresponding grammar for infinite set of terms produceable by the intruder. To enable the choice of an arbitrary term to be used in the model checking routine, I have implemented functions for unfolding the nonterminals according to the grammar to a level specified by the user. However the number of terms quickly explode when unfolding to a level of encryption deeper than one or two at the most, depending on the arity of encryptions. In most cases it is sufficient only to consider names in the intruder knowledge and ignore encryption terms.

Since the intruder has complete control over the network, the terms in the messages of the network component κ represents the knowledge of the intruder. We disregard the fact that κ contains sequences of terms. The knowledge of the intruder is simply defined as the set of terms, $\{t \mid \exists\langle t_1, \cdots, t_k\rangle \in \kappa, \exists i : t = t_i\}$. Note that the formula of Table 3.2 ensures that all terms that can be decrypted by the intruder are in fact decrypted and occurs on the network.

Chapter 4

Model Checking

The techniques of model checking rely on simulating the steps of a protocol. In general the model checking can be seen as a search for a breach in a search space consisting of all possible executions of a protocol in parallel with an intruder. This search space is infinite, which means that if we are searching for an error which does not exist, the search will not terminate. This weakness of the model checking technique is to some extend remedied by the fact that we do not even start the model checking routine if the analysis does not report a breach in the protocol. This greatly reduces the risk of searching for a breach which does not exist. However the analysis returns a safe over-approximation of the possible breaches of a protocol. This implies that we can still encounter a false positive in the analysis which in turn in general will cause the model checking routine to loop infinitely.

The basic concepts of the following sections are inspired by [8]. Section 4.7 describes how my work is related to these techniques.

4.1 Searching the graph

Recall that a protocol is presented as a directed graph, where nodes represent a given state of the protocol and each transition represents communication between the intruder and one of the principals. The model checking routine of the SAMC tool takes a protocol and a given breach as input. The breach can be a secrecy breach or an authentication breach. The task of the model checking routine is to simulate all possible executions of a protocol in parallel with the intruder until the given breach is encountered. This is done in a breadth first manner ensuring that we find the shortest path in the graph corresponding to a minimum sequence of communications.

Figure 4.1 shows the graph representation of the running example protocol. The bold transitions marked with numbers 1, 2, 3 and 4 potentially corresponds to

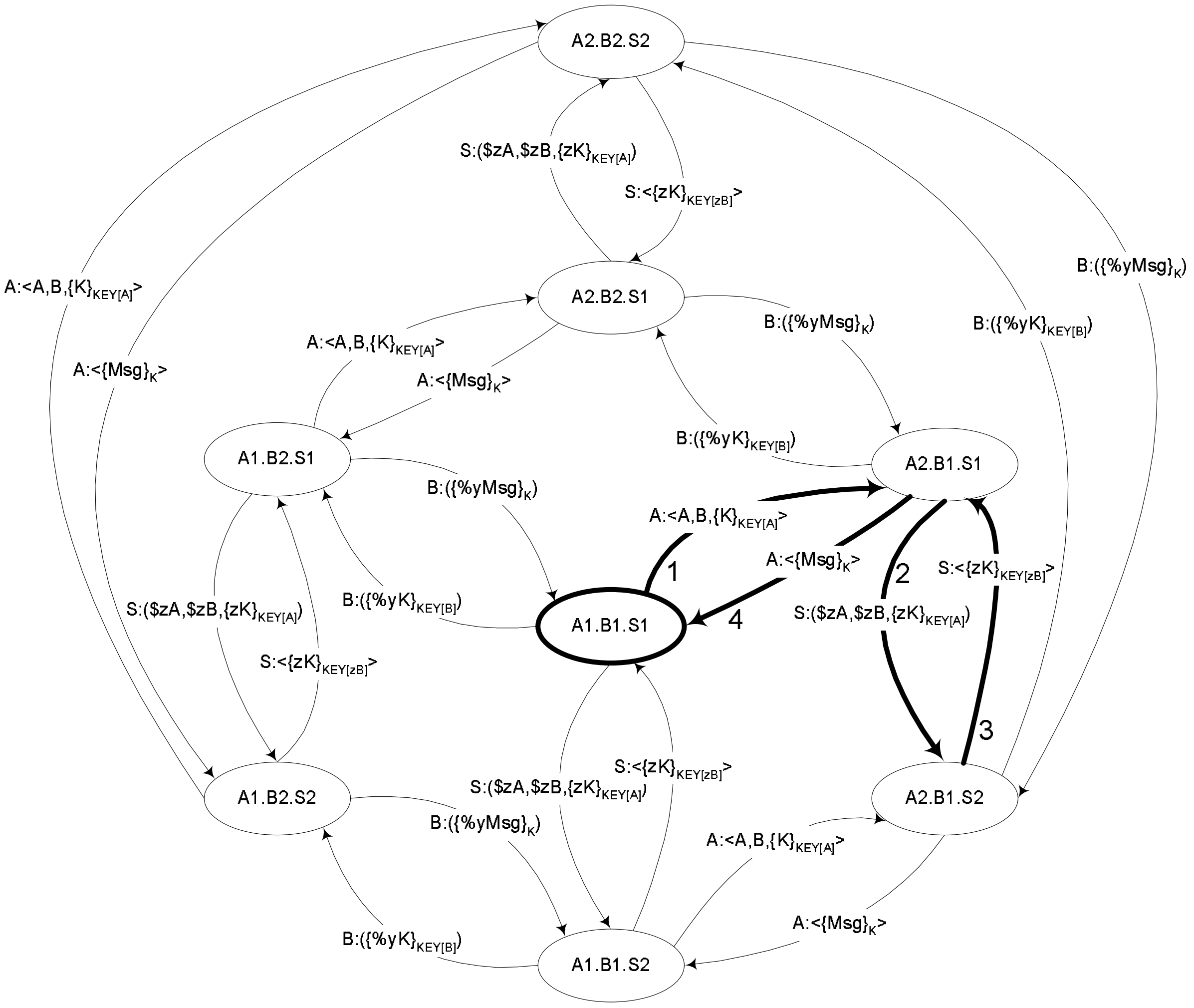

Figure 4.1: Running example protocol with a potential attack path indicatied by the bold transitions 1, 2, 3, 4.

the following attack that we have seen before:

$$\begin{array}{lllll} 1. & A & \rightarrow I(S) & : & A, B, \{K\}_{\mathsf{key}[A]} \\ 2. & I(A) & \rightarrow S & : & A, I, \{K\}_{\mathsf{key}[A]} \\ 3. & S & \rightarrow I(B) & : & A, \{K\}_{\mathsf{key}[I]} \\ 4. & A & \rightarrow I(B) & : & \{Msg\}_K \end{array}$$

During the simulation we have to keep track of three components corresponding to the components of the analysis:

- Values assigned to variables. (Corresponding to the mapping ρ in the analysis).

- Mismatch between origin and destination annotations. (Corresponding to the set ψ in the analysis).
- The knowledge of the intruder. (Corresponding to κ in the analysis).

The model checking routine gives us an exact value for each of the three components corresponding to a given path in the graph. The symbols $\widehat{\rho}$, $\widehat{\kappa}$ and $\widehat{\psi}$ denotes the exact values of the corresponding components of the analysis.

Simulating input and output. The model checking routine is based on a breadth first search in the graph. For now however, we consider the task of determining whether a given path leads to a certain breach.

As explained in Section 2.4 we do not simulate the semantics of LySA$^{\mathrm{N}}$ exactly. Since the intruder is able to resend a message unaltered to any principal, we only consider communication where the intruder is involved.

When we simulate communication from a principal to the intruder, (*output*), we simply add the terms of the output tuple to the knowledge of the intruder $\widehat{\kappa}$. The intruder knowledge grows as we follow a given path in the graph. When simulating a communication from the intruder to a principal, (*input*), we are faced with a more challenging problem. An input step is represented by the tuple $(p_1, \cdots, p_k)$. The task is to simulate all possible matches between tuples of terms from the intruder knowledge and the input tuple. I.e. we have to do a case split with a branch for each of the outputs from the intruder in the set

$$\begin{aligned}\{\langle t_1, \cdots, t_k\rangle \mid {} & \wedge_{i=1}^{k} t_i \in \mathcal{DY}(\widehat{\kappa}) \wedge \\ & \emptyset \vdash t_1 \rhd p_1 : \theta_1 \wedge \theta_1 \vdash t_2 \rhd p_2 : \theta_2 \wedge \cdots \wedge \theta_{k-1} \vdash t_k \rhd p_k : \theta_k \ \}\end{aligned}$$

according to the communication rule of Table 2.4. $\mathcal{DY}(\widehat{\kappa})$ denotes the set of terms that can be generated from a given knowledge, $\widehat{\kappa}$, according to the capabilities of the Dolev-Yao intruder as defined in Table 4.1. Recall that the Dolev-Yao intruder is capable of constructing new encryptions and decrypting encryption terms if he knows the key.

$$\frac{t \in \widehat{\kappa}}{t \in \mathcal{DY}(\widehat{\kappa})} \qquad \frac{\{t_0, t_1, \cdots, t_k\} \subseteq \mathcal{DY}(\widehat{\kappa})}{\{t_1, \cdots, t_k\}_{t_0} \in \mathcal{DY}(\widehat{\kappa})}$$

$$\frac{\{t_1, \cdots, t_k\}_{t_0} \in \mathcal{DY}(\widehat{\kappa}) \wedge t_0 \in \mathcal{DY}(\widehat{\kappa})}{\{t_1, \cdots, t_k\} \subseteq \mathcal{DY}(\widehat{\kappa})}$$

Table 4.1: Encryption and decryption capabilities of the intruder expressed by $\mathcal{DY}(\widehat{\kappa})$.

Since we potentially have infinitely deep levels of embedded encryptions and arbitrary lengths of encrypted tuples this set is of course infinite. Even though we limit the possible depth of encryptions by some constant number and limit the size of encrypted tuples cf. discussion of Table 3.2, we are still faced with a severe combinatorial explosion.

Path explosion. Another basic problem we encounter in the model checking is that the state space is infinite. Since we consider a set of looping processes in parallel represented by a cyclic graph, we can generate paths of infinite length. However in all experiments I have encountered no attacks are based on more than a few iterations of a protocol. But although we can expect that we do not need very long paths, we still encounter a problem of an explosion in the number of paths $P(l)$ of a given length l for fairly small values of l. The simple instance of the running example protocol, containing only the three principals A,B and S has $P(l) = 3^l$ paths of length l.

4.2 The lazy intruder – Collecting constraints

The concept of a lazy intruder is introduced to solve the problem of the infinite number of choices the intruder has when matching an input tuple of a principal. The basic idea is to postpone the decision about the exact values that are send to the principal, and instead collect a constraint which contains the information about the knowledge of the intruder at the point where the pattern matching occurs.

As an example we look at the input tuple of S in the running example protocol, $(\$z_A, \$z_B, \{z_K\}_{\mathsf{key}[z_A]})$. We simply remember that the terms z_A, z_B and $\{z_K\}_{\mathsf{key}[z_A]}$ are matched with terms generated from $\widehat{\kappa}$ as it looked at the time the input tuple was encountered. This information is collected in a list of constraints. Each constraint is a pair $(p, \widehat{\kappa})$ consisting of a pattern and a set representing the intruder knowledge.

$$\begin{array}{l}(z_A, \widehat{\kappa})\\(z_B, \widehat{\kappa})\\(\{z_K\}_{\mathsf{key}[z_A]}, \widehat{\kappa})\end{array}$$

The intruder knowledge is represented by a finite set consisting of the terms that are collected from the output tuples encountered so far in the path we consider. When we at some point start to solve the constraints, we have to consider the encryption and decryption of terms in $\widehat{\kappa}$.

Note that the symbols "%" and "\$" denoting defining occurrences of variables are removed.

When collecting constraints it is important to preserve the order of the constraints, as we shall see. Hence we consider constraint lists and not simply sets of constraints. In the following constraint list will be presented as tables whenever it is convenient. The following table represents the constraint list presented above:

1	z_A	$\widehat{\kappa}$
2	z_B	$\widehat{\kappa}$
3	$\{z_K\}_{\mathsf{key}[z_A]}$	$\widehat{\kappa}$

For a given path the constraints are generated according to the following algorithm presented in SML-like notation:

$$\begin{array}{ll} gc(\langle t_1, \cdots, t_k \rangle :: path, \widehat{\kappa}) & = gc(path, \{t_1, \cdots, t_k\} \cup \widehat{\kappa}) \\ gc((p_1, \cdots, p_k) :: path, \widehat{\kappa}) & = [(p_1, \widehat{\kappa}), ..., (p_k, \widehat{\kappa})]@gc(path, \widehat{\kappa}) \\ gc([\,], \widehat{\kappa}) & = [(final, \widehat{\kappa})] \end{array}$$

The symbol "::" denotes prefix concatenation of lists. $ls_1@ls_2$ denotes the appending of two lists.

For each output tuple in the path the intruder knowledge is updated to contain the terms of the output. For each pattern in each of the input tuples a constraint is generated. The final intruder knowledge is recorded in a "dummy" constraint in the last line. This is done to avoid loosing information if the last communication step(s) of a given path are output steps. $final$ is a variable not used elsewhere. This means that the "dummy" constraint does not imply any restrictions.

In the following the constraint generation is exemplified using the path, $[1, 2, 3, 4]$ from Figure 4.1. A path in the graph can be illustrated as a type of narration, where we introduce variables. The following *variable narration* corresponds to the path, $[1, 2, 3, 4]$. As we have seen, this path can potentially result in a secrecy breach, if the right substitution for each of the variables is chosen.

$$\begin{array}{ll} 1. & A \rightarrow I \ : A, B, \{K\}_{\mathsf{key}[A]} \\ 2. & I \ \rightarrow S : z_A, z_B, \{z_K\}_{\mathsf{key}[z_A]} \\ 3. & S \ \rightarrow I \ : z_A, \{z_K\}_{\mathsf{key}[z_B]} \\ 4. & A \rightarrow I \ : \{Msg\}_K \end{array}$$

The following table illustrates the corresponding constraint generation. $\widehat{\kappa}_0$ is the initial knowledge of the intruder. $\widehat{\kappa}_i$ for $i = 0, 1, 2, 3$ is introduced to keep track of how the intruder knowledge is increased corresponding to processing each of the output steps of the path.

$$\begin{array}{ll|l} 1. & A \rightarrow I \ : A, B, \{K\}_{\mathsf{key}[A]} & \widehat{\kappa}_1 = \widehat{\kappa}_0 \cup \{A, B, \{K\}_{\mathsf{key}[A]}\} \\ 2. & I \ \rightarrow S : z_A, z_B, \{z_K\}_{\mathsf{key}[z_A]} & (z_A, \widehat{\kappa}_1), (z_B, \widehat{\kappa}_1), (\{z_K\}_{\mathsf{key}[z_A]}, \widehat{\kappa}_1) \\ 3. & S \ \rightarrow I \ : z_A, \{z_K\}_{\mathsf{key}[z_B]} & \widehat{\kappa}_2 = \widehat{\kappa}_1 \cup \{z_A, \{z_K\}_{\mathsf{key}[z_B]}\} \\ 4. & A \rightarrow I \ : \{Msg\}_K & \widehat{\kappa}_3 = \widehat{\kappa}_2 \cup \{\{Msg\}_K\} \\ & & (final, \widehat{\kappa}_3) \end{array}$$

The constraint $(final, \widehat{\kappa}_3)$ is a constraint which is generated as explained to save the final knowledge of the intruder. If we choose the initial knowledge of the intruder to be the set consisting of the names of the principals and the intruder key, $A, B, S, I, \mathsf{key}[I]$, we end up with the following constraint list.

1	z_A	$\{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$
2	z_B	$\{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$
3	$\{z_K\}_{\mathsf{key}[z_A]}$	$\{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$
4	$final$	$\{Msg\}_K, z_A, \{z_K\}_{\mathsf{key}[z_B]}, \{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$

Table 4.2: The constraints are numbered to be able to refer to them in the following text. Do not confuse the constraint numbering with the numbering of path transitions.

Note that since we postpone the binding of variables, we can have occurrences of variables in the intruder knowledge. This fact gives rise to some problems that are addressed below.

To ensure that principal variables can only be bound to names of type Prin as specified in the rules for pattern matching in Table 2.3, we introduce a constraint corresponding to each of the principal variables of a LySA$^{\mathrm{N}}$ process:

$$(x, \mathcal{N}_P \cup \{I\})$$

where $\mathcal{N}_P \cup \{I\}$ is the set of all principal names including the intruder.

4.3 Constraint solving

So far we have been ignoring the origin and destination annotations of encryptions. We will do that a bit longer and focus on the secrecy breach. Recall that a secrecy breach is defined by the occurrence of a certain constant, (in our running example we look at the constant *Msg*), in the final intruder knowledge. This can be expressed by substituting the variable $final$ with the constant *Msg* in the last constraint. Constraint 4 is changed to

$$(Msg, \{\{Msg\}_K, z_A, \{z_K\}_{\mathsf{key}[z_B]}, \{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]\}$$

It is intuitively clear that the constraint list now contains all the information we need to determine whether a given path leads to a given secrecy breach. A constraint $(t, \widehat{\kappa})$ is satisfied when $t \in \mathcal{DY}(\widehat{\kappa})$. The task is to choose a substitution, σ, mapping variables into values, such that all constraints are satisfied. Note that $dom(\sigma)$ might not include all variables occurring in the constraints. This is due to the fact that the value of some of the variables might be irrelevant for the given attack, and are hence not assigned any values. A constraint, $(x, \widehat{\kappa})$, where x is a variable is always satisfiable, so in order to determine whether a given list of constraints is satisfiable, we do not necessarily need to find a substitution for every single variable in the constraint set. Note that since the intruder can generate new names, the set $\widehat{\kappa}$ can never be empty.

We can find a solution to the constraint list in Table 4.2 of the previous section corresponding to the attack by looking at the final knowledge of the intruder, which can be extracted from constraint 4. To be able to decrypt $\{Msg\}_K$ the intruder needs to know K. This can only be accomplished if z_K is bound to K and z_B is bound to I. These bindings would allow the intruder to decrypt the term $\{z_K\}_{\mathsf{key}[z_B]}$ and subsequently decrypt $\{Msg\}_K$. The substitution

$$[z_k \mapsto K, z_B \mapsto I]$$

also satisfies all of the constraints. In constraint 2 and 3 we have the two matches

$$z_B \leftrightarrow I \qquad \text{and} \qquad \{z_K\}_{\mathsf{key}[z_A]} \leftrightarrow \{K\}_{\mathsf{key}[A]}$$

The substitution together with the path corresponds to the narration of the attack.

But how do we find the right substitution in general? A brute force approach of enumerating all possible substitutions brings us back to the problem of the infinite intruder knowledge. The trick is not to assign values to variables before it is absolutely necessary. Hence the name *lazy* intruder. The technique is explained below.

We define four different constraint types depending on the left hand side (LHS) p of the constraint $(p, \widehat{\kappa})$.

- Constant, if p is a name
- Key, if p has the form $\mathsf{key}[p']$
- Variable, if p is a variable
- Encryption, if p has the form $\{p_1, \cdots, p_k\}_{p_0}$

We treat the constraints one at a time as they are ordered in the list generated from a given path.

Constant constraint. When we encounter a constraint of type *constant*, we simply check whether the constraint can be satisfied by testing whether the term on the LHS of the constraint is element in $\mathcal{DY}(\widehat{\kappa})$, where $\widehat{\kappa}$ is the RHS of the constraint. If the constraint is satisfiable it can be removed. If not the whole constraint list is unsatisfiable and the path is invalid in the sense that the intruder is incapable of generating terms that can match all of the input tuples of the path.

Key constraint. When encountering a *key* constraint, $(\mathsf{key}[p], \widehat{\kappa})$, we do a case split into all possible unifications between the LHS of the constraint and the terms in $\mathcal{DY}(\widehat{\kappa})$. These unifications might lead to a substitution, σ. E.g. the constraint $(\mathsf{key}[x], \{I, \mathsf{key}[I], \mathsf{key}[A]\})$ leads to a split into two cases where we obtain the two substitutions $[x \mapsto I]$ and $[x \mapsto A]$ respectively.

Variable constraint. When a *variable* constraint is encountered we move on to the next constraint, saving the variable constraint in a list of variable constraints. Every time the substitution is extended due to a key constraint, (or as we shall see, due to an encryption constraint), the list of considered variable constraints is reconsidered. This is done by applying the substitution to the constraints, which might imply that some of the constraints are no longer of the type variable constraint. In this case the constraints in question are reinserted in the list of constraints to be considered. Consider the following constraint list as an example:

x	$\{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$
$\mathsf{key}[x]$	$\{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$
$\cdots$	

The horizontal gap in the table indicates that the first constraint has already been considered and saved for later in a list of considered variable constraints. The second constraint results in the substitution $[x \mapsto I]$. When the substitution is applied the constraint $(x, \{\{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]\})$ changes into $(I, \{\{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]\})$. The constraint has changed type into a constant constraint and is reinserted into the list of constraints to be considered.

Encryption constraint. An *encryption* constraint, $(\{p_1, \cdots, p_k\}_{p_0}, \widehat{\kappa})$ gives rise to a case split into two main cases:

1. A direct unification between a term in $\widehat{\kappa}$ and the LHS of the constraint is performed. We do a split into sub-cases if there are more than one term in $\widehat{\kappa}$ that can be unified with the encryption term.
2. The encryption constraint, $(\{p_1, \cdots, p_k\}_{p_0}, \widehat{\kappa})$, is split up into $k+1$ new constraints: $(p_0, \widehat{\kappa}), (p_1, \widehat{\kappa}), \cdots, (p_k, \widehat{\kappa})$. The split into new constraints corresponds to the intruder generating the encryption from $\widehat{\kappa}$. The new constraints are inserted at the start of the list of constraints and are to be considered next.

4.3.1 Analyzing Intruder knowledge

When we look at the intruder knowledge as it is represented in the constraints we have to remember that it is only a finite representation of an infinite set. $\widehat{\kappa}$ simply contains the terms that are added when encountering an output in a given path. We also have to consider the decryption of terms and the construction of new encryptions from terms in $\widehat{\kappa}$. The construction of new encryption terms is taken care of by the decomposition of the LHS of encryption constraints as explained above. The decomposition of an encryption constraint into a constraint for each term in the encryption term, t, of the original constraint corresponds to the composition of t based on $\widehat{\kappa}$. E.g. for the constraint $(\{A\}_K, \{A, K\})$ we do a case split where the decomposition case results in the constraints

K	A, K
A	A, K

They are clearly both satisfied as they should be since the original encryption term $\{A\}_K$ could be generated from $\widehat{\kappa} = \{A, K\}$.

So decomposition of the LHS of encryption constraints takes care of the intruders capability of constructing encryptions. The decryption of terms in the intruder knowledge however is done explicitly every time a new constraint is considered. We simply decrypt all encryptions that can be decrypted as explained below.

Lemma 3 (Variables in $\widehat{\kappa}$) *Every variable occurring in $\widehat{\kappa}$ – possibly as part of an encryption – can only be bound to a term, $t \in \widehat{\kappa}$.*

Proof. Every variable originates from an input tuple and must hence occur for the first time – when considering the constraints in the order they are generated – on the LHS of a constraint. If the variable originally occurred as part of an encryption term, e.g. $(\{x\}_K, \widehat{\kappa})$, the encryption term has been dealt with and is either split up into new constraints or the variable has been substituted. Since the substitution is applied to all remaining constraints every time it is extended, the latter can not be the case. This means that when we encounter a variable in $\widehat{\kappa}$, the variable must also occur in an earlier variable constraint as illustrated by the following constraints.

$$\begin{array}{l} (x, \widehat{\kappa}_1) \\ \cdots \\ (\cdots, \widehat{\kappa}_1 \cup \{x, \cdots\}) \end{array}$$

From the first constraint we can conclude that if x at some point is assigned a value, v, it must hold that $v \in \widehat{\kappa}_1$. Since the intruder knowledge never decreases we know that a potential value of x is in $\widehat{\kappa}$ of all subsequent constraints. Note that the reason why constraints has to be treated in the order that they are generated is in order to make this argument valid.

Constant key. When the key of an encryption term is a constant, n, we decrypt if and only if $n \in \widehat{\kappa}$. A key on the form $\mathsf{key}[t]$, where t is a constant is treated the same way: We decrypt only if $\mathsf{key}[t] \in \widehat{\kappa}$. The interesting question is whether we can be sure that we cannot decrypt an encryption term with a constant key if the constant is not part of $\widehat{\kappa}$. Recall that the intruder knowledge also contains variables that are not assigned values *yet*. If $\widehat{\kappa} = \{\{t\}_K, x\}$ can we be sure that x cannot be bound to the value K at a later stage and hereby enabling the decryption of $\{t\}_K$? The answer is yes because of Lemma 3.

Variable key. Lemma 3 states that the potential values of a variable key, x, of an encryption term of the form $\{\cdots\}_x$ will be available in $\widehat{\kappa}$. This implies that encryption terms with variable keys are always decrypted.

Complex key. A complex key is a key that is either an encryption or has the form $\mathsf{key}[t]$ where t is a variable. Consider the constraint list

$$\begin{array}{c|l} x & I, \mathsf{key}[I] \\ \hline Msg & I, \mathsf{key}[I], \{Msg\}_{\mathsf{key}[x]} \end{array}$$

The first constraint is ignored (to begin with). The second constraint can only be satisfied if the term $\mathsf{key}[x]$ is in the set $\mathcal{DY}(\{I, \mathsf{key}[I], \{Msg\}_{\mathsf{key}[x]}\})$. It is easy to see that this is actually the case for the substitution $[x \mapsto I]$. This substitution also satisfies the first constraint. The way to find this solution is by generating a new constraint containing the key of the encryption term on the LHS. The strategy is that we for each encryption term in $\widehat{\kappa}$ with a complex key do a case split into two cases. One where we decrypt and one where we do not. If we decrypt we also generate a new constraint expressing that the key must be part of the intruder knowledge. The constraint list above is split into the following two constraint lists:

decrypt

$$\begin{array}{c|l} x & I, \mathsf{key}[I] \\ \hline \mathsf{key}[x] & I, \mathsf{key}[I] \\ \hline Msg & Msg, I, \mathsf{key}[I], \{Msg\}^{*}_{\mathsf{key}[x]} \end{array}$$

no decrypt

$$\begin{array}{c|l} x & I, \mathsf{key}[I] \\ \hline Msg & I, \mathsf{key}[I], \{Msg\}^{*}_{\mathsf{key}[x]} \end{array}$$

The LHS of the new constraint is the key in question. The RHS is the intruder knowledge without the encryption term that is considered. The encryption term is marked, $\{Msg\}^{*}_{\mathsf{key}[x]}$, in both cases to indicate that the term has now been considered.

From the above example it is clear to see that in this case decrypting the term is the right choice. The constraint list where we do not decrypt is clearly unsatisfiable, whereas the constraint list where decryption is chosen is satisfiable and yields the solution, $[x \mapsto I]$.

4.3.2 An example

We now use the strategy explained above on the running example protocol and the path corresponding to the attack we have looked at. The initial constraint list looks as follows:

1	z_A	$\{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$
2	z_B	$\{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$
3	$\{z_K\}_{\mathsf{key}[z_A]}$	$\{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$
4	Msg	$\{Msg\}_K, z_A, \{z_K\}_{\mathsf{key}[z_B]}, \{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$

The two first constraints are considered and saved for later, which is illustrated by the horizontal gap in the table.

1	z_A	$\{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$
2	z_B	$\{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$
3	$\{z_K\}_{\mathsf{key}[z_A]}$	$\{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$
4	Msg	$\{Msg\}_K, z_A, \{z_K\}_{\mathsf{key}[z_B]}, \{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$

When dealing with the third constraint, we split up into two cases. A case where we generate new constraints and a case where we make a unification. This split yields the following two new constraint lists.

(1)

1	z_A	$\{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$
2	z_B	$\{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$
3_a	z_K	$\{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$
3_b	$\mathsf{key}[z_A]$	$\{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$
4	Msg	$\{Msg\}_K, z_A, \{z_K\}_{\mathsf{key}[z_B]}, \{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$

(2)

2	z_B	$\{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$
$1'$	A	$\{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$
4	Msg	$\{Msg\}_K, \{K\}_{\mathsf{key}[z_B]}, \{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$

We consider case (2). The substitution $[z_K \mapsto K, z_A \mapsto A]$ from the unification of $\{z_K\}_{\mathsf{key}[z_A]}$ and $\{K\}_{\mathsf{key}[A]}$ has been applied and constraint 3 is removed. The variable constraint 1 has changed to a constant constraint, $1'$. The constant constraint is seen to be satisfied and is removed. We now deal with constraint 4.

2	z_B	$\{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$
4	Msg	$\{Msg\}_K, \{K\}_{\mathsf{key}[z_B]}, \{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$

First step is to decrypt whatever can be decrypted. The only term which could possibly be decrypted is the term $\{K\}_{\mathsf{key}[z_B]}$. In fact we split up into two cases: A case where we decrypt and a case where we do not. It is clear to see that the latter immediately will imply that the constraint is unsatisfiable. The interesting case is the one where we decrypt yielding the following constraint list.

2	z_B	$\{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$
4_a	$\mathsf{key}[z_B]$	$\{Msg\}_K, \{K\}_{\mathsf{key}[z_B]}, \{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$
$4'$	Msg	$K, Msg, \{Msg\}_K, \{K\}_{\mathsf{key}[z_B]}, \{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$

Note that the decryption of terms in $\widehat{\kappa}$ is iterated until no terms can be decrypted. Hence the decryption of $\{K\}_{\mathsf{key}[z_B]}$ is automatically followed by the decryption of $\{Msg\}_K$ which is now made possible.

The new key constraint yields the unification between $\mathsf{key}[z_B]$ and $\mathsf{key}[I]$ with the corresponding substitution $[z_B \mapsto I]$. This gives us the following constraint list.

$2'$	I	$\{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$
$4'$	Msg	$K, Msg, \{Msg\}_K, \{K\}_{\mathsf{key}[z_B]}, \{K\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$

Both constant constraints are satisfied, and the constraint list is solved. The substitution is collected as we go along to be able to reconstruct the exact narration of the attack from the path we started out with. We have generated the following substitution:

$$[z_K \mapsto K, z_A \mapsto A, z_B \mapsto B]$$

4.3.3 Adding annotations

As we have seen, finding a secrecy breach is directly implemented as a part of the constraint solving by using the breach term in a final constraint. To be able to search for authentication breaches as well we have to keep track of the annotations during the constraint solving.

Recall that an authentication breach occur when there is a mismatch between the actual and the intended places of encryption and decryption of an encryption term. A mismatch is represented by a pair (n_1, n_2), expressing that something encrypted at n_1 was wrongfully decrypted at n_2.

During the steps of the constraint solving we keep track of origin-destination mismatches. From the syntax of LySa$^{\text{N}}$ we recall that encryption patterns and encryption terms have the form

$$\{p_1, \cdots, p_k\}_{p_0}^{p_{cp}} \;[\mathsf{orig}\; t_{orig}]$$
$$\{t_1, \cdots, t_k\}_{t_0}^{t_{cp}} \;[\mathsf{dest}\; t_{dest}]$$

A mismatch between the two encryptions above correspond to the following mismatch in the static analysis: (t_{cp}, p_{cp}). In the model checking we record all four components yielding the following representation of the same mismatch:

$$((t_{cp}, p_{orig}), (t_{dest}, p_{cp}))$$

This is necessary because variables can occur in mismatches. As a consequence we need to collect a potential mismatch, e.g. $((A, A), (B, z_B))$, which for some substitutions, e.g. $[z_B \mapsto B]$, is in fact not a mismatch and is hence removed from the final result.

Types of mismatches. With I being the crypto-point of the intruder, we divide the origin-destination mismatches into three groups.

1. (n, I). An encryption term is wrongfully decrypted by the intruder.
2. (I, n). The intruder has generated an encryption term which is wrongfully decrypted by a principal.
3. (n_1, n_2). An encryption term generated at n_1 is wrongfully decrypted at n_2, n_1 and n_2 being crypto-points of two legitimate principals.

Each of these groups are handled differently in the constraint solving as explained below.

1. (n, I). If an encryption term $\{t_1, \cdots, t_k\}_{t_0}^{t_{cp}} \;[\mathsf{dest}\; t_{dest}] \in \widehat{\kappa}$ of a constraint, $(t, \widehat{\kappa})$, is decrypted, the intended destination, t_{dest} is checked. If $t_{dest} \neq I$ the origin-destination mismatch (t_{cp}, I) is collected. Note that since the intruder may be a legitimate principal, it is possible that the intended destination of an encryption term actually is I.
2. (I, n). The decomposition of an encryption constraint, c, into new constraints corresponding to each of the terms of the LHS of c corresponds to the case where the intruder generates a matching encryption term as explained above. Consider the following constraint solving step as an example: The constraint $(\{A\}_K^{n_{cp}} \;[\mathsf{orig}\; n_{orig}], \widehat{\kappa})$ is transformed into the two constraints $(A, \widehat{\kappa})$ and $(K, \widehat{\kappa})$. This transformation corresponds to the intruder generating a matching encryption, $\{A\}_K^{I} \;[\mathsf{dest}\; n_{cp}]$. If $n_{orig} \neq I$, we record the mismatch $((n_{cp}, n_{cp}), (n_{orig}, I))$.
3. (n_1, n_2). When a unification between an encryption term in $\widehat{\kappa}$ and a LHS of a constraint is made, the crypto-points are compared. If a mismatch occurs it is collected.

4.3.4 Adding session indices to variables and fresh names

When we consider a path that contains cycles for one or more of the involved principals, we have to keep track of the different sessions of a protocol. As an example we look at the LySa$^{\mathrm{N}}$ process of principal A in the running example protocol.

$$[\mathsf{loop}((\nu\, K)\langle A, B, \{K\}_{\mathsf{key}[A]}\rangle.(\nu\, Msg)\langle\{Msg\}_K\rangle.\mathrm{RET})]$$

K and Msg are session names that are generated in each iteration of the protocol. These names has to be identified in a way that makes it possible to distinguish between two different sessions of the same protocol. The names A and B on the other hand can be thought of as global. Whenever the process of principal A is restarted, it is still the same A and B but it is a new K and Msg. The same problem exists with variables. Just because a variable is bound to a value in a given session of the protocol it should not be the case that the variable has to be bound to the same value in the next session. In some sense, the variables are also reinstantiated in every iteration of a given process.

This problem motivates the introduction of indices of variables and fresh names. A global environment keeps track of the occurrence of variables and fresh names in a given path. Every time the defining occurrence of a variable, $\%x$, is encountered, the index of x is increased. The index is assigned to the defining occurence of x and all subsequent occurences of x until $\%x$ is encountered again.

Fresh names are specified in LySa$^{\mathrm{N}}$ by using the $(\nu\, n)$-construction. As seen from the illustration of the graph representation in figure 4.1 this information is not represented in the same way in the graph representation. Instead we introduce a way of marking the first occurrence of a fresh name, similar to the way of marking the defining occurrence of a variable. These marks are introduced in the conversion from the LySa$^{\mathrm{N}}$ representation to the graph representation. The process above is represented by the following sequence of input and output tuples:

$$\langle A, B, \{\mathsf{fresh}[K]\}_{\mathsf{key}[A]}\rangle, \langle\{\mathsf{fresh}[Msg]\}_K\rangle$$

Consider the following narration corresponding to two iterations of the path that we have seen before:

$$\begin{array}{llll}
i.1 & A & \to I(S) & : A, B, \{K\}_{\mathsf{key}[A]} \\
i.2 & I(A) & \to S & : z_A, z_B, \{z_K\}_{\mathsf{key}[z_A]} \\
i.3 & S & \to I(B) & : z_A, \{z_K\}_{\mathsf{key}[z_B]} \\
i.4 & A & \to I(B) & : \{Msg\}_K \\
\\
ii.1 & A & \to I(S) & : A, B, \{K\}_{\mathsf{key}[A]} \\
ii.2 & I(A) & \to S & : z_A, z_B, \{z_K\}_{\mathsf{key}[z_A]} \\
ii.3 & S & \to I(B) & : z_A, \{z_K\}_{\mathsf{key}[z_B]} \\
ii.4 & A & \to I(B) & : \{Msg\}_K
\end{array}$$

The corresponding constraint list with variables look as follows:

z_A^1	$\{K^1\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$
z_B^1	$\{K^1\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$
$\{z_K^1\}_{\mathsf{key}[z_A^1]}$	$\{K^1\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$
z_A^2	$\{K^2\}_{\mathsf{key}[A]}, \{Msg^1\}_{K^1}, z_A^1, \{z_K^1\}_{\mathsf{key}[z_B^1]}, \{K^1\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$
z_B^2	$\{K^2\}_{\mathsf{key}[A]}, \{Msg^1\}_{K^1}, z_A^1, \{z_K^1\}_{\mathsf{key}[z_B^1]}, \{K^1\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$
$\{z_K^2\}_{\mathsf{key}[z_A^2]}$	$\{K^2\}_{\mathsf{key}[A]}, \{Msg^1\}_{K^1}, z_A^1, \{z_K^1\}_{\mathsf{key}[z_B^1]}, \{K^1\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$
$final$	$\{Msg^2\}_{K^2}, z_A^2, \{z_K^2\}_{\mathsf{key}[z_B^2]}, \{K^2\}_{\mathsf{key}[A]}, \{Msg^1\}_{K^1}, z_A^1,$ $\{z_K^1\}_{\mathsf{key}[z_B^1]}, \{K^1\}_{\mathsf{key}[A]}, A, B, S, I, \mathsf{key}[I]$

The fact that we are able to separate variables and fresh names of each instance makes it possible to simulate attacks relying on more than one session of a protocol. When specifying the term of a secrecy breach the user has the choice of specifying an index as well. If no index is specified, e.g. Msg, the model checking routine will search for any instance of Msg. If an index is specified, it will only match the corresponding index. This makes it possible to search for any occurrence of a name, of for a specific occurrence e.g. the second occurrence, Msg^2.

4.3.5 The case splits and search states

Below we shall have a closer look at how to keep track of the case splits when solving a constraint list. Do not confuse the search for a path in the graph with the search for a solution to a constraint list corresponding to a given path.

In the strategy explained above we encounter a number of case splits in different situations. The following list sums up these case splits.

- Whenever the LHS of a constraint is unified with terms in $\widehat{\kappa}$ we split up into a branch corresponding to each of the possible unifications.
- When considering an encryption constraint we split up into two cases:
  1. Direct unification between the LHS of the constraint and a term in $\widehat{\kappa}$. (In this case there are sub-cases corresponding to each of the possible unifications).
  2. Decomposition of LHS into a set of new constraint.
- A case split is made every time we encounter an encryption term with a complex key in $\widehat{\kappa}$ of a constraint. The split corresponds to the choice of whether to decrypt or not.

Definition. *A* search states *consists of a triple, $(\sigma, C, C_v, \widehat{\psi})$. σ is a substitution, C is a list of constraints to be considered, C_v is a list of variable constraints that has been considered and $\widehat{\psi}$ is a set of origin-destination mismatches.*

To keep track of the case splits we implement the search for a solution to a given constrain list as a search in a search tree where each of the tree nodes are search states. An initial search state corresponding to a given path, p, in the graph consists of an empty substitution, a constraint list, C, containing the constraints corresponding to p, an empty list, C_v, of variable constraints, and an empty set $\widehat{\psi}$.

We can generate a set of new search states from a given search states using the strategy and case splits as explained above. Each time we generate a new search state, $(\sigma', C', C'_v, \widehat{\psi}')$, from a parent state, $(\sigma, C, C_v, \widehat{\psi})$, we know the following.

- C' is either simpler, (i.e. an encryption constraint is changed to number of simpler constraints), or smaller, (i.e. a constraint in C has been considered and is either moved to C_v or removed)
- $\sigma \subseteq \sigma'$. The substitution never decreases.

At some point a search state might contain a constraint list which is unsatisfiable. The corresponding branch of the search tree is simply discontinued.

Final search states are search states where $C = [\ ]$. When a constraint list is solved according to the case splits above an initial state is transformed into a set of final states. When C is empty we can ignore the variable constraints in since assigning values to the LHS of these constraints will not add additional elements to $\widehat{\psi}$. If we are searching for an authentication error, we simply have to determine whether there exists an origin-destination mismatch set $\widehat{\psi}$ in one of the final constraints, which contains the given authentication breach. The substitution together with the initial path are the elements needed to generate a narration for the corresponding attack. If we are searching for a secrecy breach, we can choose a substitution from any of the final states because the fact that C is empty implies that all constraints have been satisfied.

Redundancy in search states. It is clear that there are room for improvement in the representation of search states as well as the actual search. Many of the search states will contain each their copy of almost identical constraint lists. Two different branches in the search tree might lead to the same search state. This is not detected implying that large subtrees of a search tree can be identical.

The search is performed in a breadth first manner. I have considered implementing a depth first approach instead, but this approach is not compatible with a technique explained in the next section, where solutions to constraint lists corresponding to a subpaths of a given path are reused.

I have not investigated further in the optimization of the constraint solving, but it could be an interesting project for future work.

4.4 Implementing the constraints in the search

We have now seen how a constraint list can be solved. I have experimented with various ways of implementing the constraint solving in the search for a path leading to a given attack. Below I describe three approaches.

Method 1. Naive approach. My initial approach is to look at all possible paths of lengths 1, 2, 3, etc. and for each path create and solve a constraint list. It is obvious that this approach implies a lot of redundant work. A path containing the states (i.e. graph nodes), $s_1, \cdots, s_i, \cdots, s_n$ generates a constraint list which contains the constraint list already solved for the subpath $s_1, \cdots, s_i$. As we shall see, it is possible to reuse some information from the constraint lists already solved for subpaths in the graph.

Method 2. Eliminating paths with unsatisfiable subpaths. As explained a constraint list can be satisfiable even though the breach cannot be reconstructed from the path in case of an authentication breach. This is due to the fact that the constraint list does not explicitly contain the breach information as is the case for the secrecy breach, where the breach term is inserted as LHS of the final constraint.

A constraint list without explicit breach information is denoted a *basic constraint list*. When searching for an authentication breach, all constraint lists are basic. When searching for a secrecy breach we can obtain a basic constraint list by removing the final constraint.

If a basic constraint list is unsatisfiable, we know that not only does the path not generate the breach, but the path itself is not valid. I.e. there exists an input somewhere along the sequence of communication that cannot be matched with terms in the corresponding knowledge of the intruder.

An invalid path should of course not be extended. All extensions of an invalid path will generate unsatisfiable constraint lists.

The technique require that a basic constraint list is solved for a given path. If there are no final states, (i.e. states with $C = [\,]$), the path is invalid and we do not consider extensions to the path. If there are final states, the path is valid. To determine whether the path also generates a given secrecy breach we add the corresponding final constraint to the constraint list, C, of all the final search states and solves the resulting search states. In the case of an authentication breach, the component $\widehat{\psi}$ of each of the final states are checked as explained.

We now have an algorithm that for each path either gives a solution, (i.e. an attack has been found), or it returns whether or not a given path is valid.

Method 3. Reusing solved constraints. With the technique above we avoid extending invalid paths but there is still room for improvement. We can reuse the final search states obtained by solving a constraint list when we consider extensions to the corresponding path.

Definition. *A* path state *consists of a path and a set of search states,* (π, S)*. A path is represented by a list of graph nodes.*

In this approach the search for a path and a substitution generating a given attack is based on *path states*. The initial path state contains a path represented by a list of one node, the initial node, n_0. A corresponding constraint list consists of a single constraint, $(z, \widehat{\kappa}_0)$, where z is a dummy variable which is only used to be able to store $\widehat{\kappa}_0$, the initial knowledge of the intruder. The initial path state is the pair

$$\begin{array}{l}([n_0], \{(\sigma_0, C_0, C_{v0}, \widehat{\psi}_0)\}) \text{ where} \\ \sigma_0 = [\ ], C_0 = [\ ], C_{v0} = [(z, \widehat{\kappa}_0)], \text{ and } \widehat{\psi}_0 = \emptyset\end{array}$$

For each graph transition $n_0 \xrightarrow{com} n'$ going out of n_0, where *com* is an input tuple or an output tuple, we generate a new path state. This is done by extending the path and each of the final search states in S. A path is extended by adding n' to the list, p. Each of the final search states of a path state are extended in the following way depending on the type of transition:

- Case 1: *com* is an output transition, $\langle t_1, \cdots, t_k \rangle$. Each of the final search states, $(\sigma, C, C_v, \widehat{\psi})$, are extended by adding the constraint $(z, \widehat{\kappa} \cup \{t_1, \cdots t_k\})$ to the empty list, C. $\widehat{\kappa}$ is the final knowledge "so far" obtained from the RHS of the last constraint in C_v.
- Case 2: *com* is an input transition, $(p_1, \cdots, p_k)$. Each of the final search states are extended by adding the constraints $(p_1, \widehat{\kappa}), \cdots, (p_k, \widehat{\kappa})$ to the empty list, C. Again $\widehat{\kappa}$ is obtained from the RHS of the last constraint in C_v.

After extending the final search states of a path state, the search states are no longer final. The constraint solving algorithm is run on each of the search states obtaining a new set of final states. This results in a situation where the path state again consists of a path and a set of final constraints, (π, S).

If at some point we have that $S = \emptyset$ we can conclude that the path, π, is invalid.

For a given path state containing a set of final search states, it is simple to determine whether the path state can generate a given breach. In the case of a secrecy breach it is done by adding a constraint to each of the final search states in S, expressing that a given term should exist in the final knowledge of the intruder. (The new search states are again solved). In case of an authentication breach we simply check the set, $\widehat{\psi}$, of origin-destination mismatches in each of the final search states in S.

I have experimented with the three implementations. Section 4.6 contains some remarks on the running time of the implemented algorithms.

[9] contains descriptions of the possibilities of reusing parts of constraint lists from different paths. I have not gone further into these optimizations in this project.

4.5 Using analysis results

As mentioned in Section 1.2 one of the interesting aspects of my work has been experimenting with using results from the static analysis in the model checking routine. Recall that the static analysis result consists of the three components, κ, ρ and ψ. My idea is to use these results to reduce the search space and hence improve the efficiency of the model checking routine.

Breach states. One technique that I have implemented is an analysis of each of the nodes in the graph. A node has a number of ingoing transitions. By looking at these transitions it is possible to determine whether a given node represents a potential breach state. A node represent a potential breach state if any of the ingoing transitions are potential final transition in an attack. For a secrecy breach only output transitions are potential final transitions, since the final transition of a secrecy attack has to add information to the knowledge of the intruder. When looking at ingoing transitions, we use the mapping ρ from variables into values to determine the possible values of a given variable. A strategy used to determine whether a given term, t, potentially can be the last piece of information leading to a breach is outlined below. If t is a variable we consider each of the terms in the set $\rho(t)$ instead of t. The term t can lead to a secrecy breach if

- t is the the breach term.
- t is an encryption which can possibly be decrypted, and t contains one or more terms that can lead to a secrecy breach. (These steps are applied recursively). To determine whether an encryption can be decrypted, we use the analysis result to check whether the encryption key is in κ.
- t is the key of an encryption term in κ, which contains one or more terms that can lead to a secrecy breach.

This technique makes it possible to filter away some paths by only looking at the last transitions of the path. Similar heuristics can be formulated for authentication breaches. There is however a number of drawbacks. Firstly the decisions are based on the analysis results and involve a number of approximations. Secondly and more seriously the technique can only be applied in the case where we do not reuse the solved constraints from subpaths explained in the previous section. The fact that the constraint list of a subpath of p is part of the constraint list of p implies that we actually does not gain anything by filtering paths based on final transitions. Just because a path cannot potentially generate an attack, we cannot conclude anything about whether the path is a subpath of an attack.

Removing transitions. Another idea for reducing the search space which did not turn out to be possible is to not only rule out certain paths based on the final transition, but to rule out a transition completely. The idea was to remove input transitions entirely from the graph if κ does not contain terms that match the given transition. Unfortunately it is easy to see that this is never the case, since each transition is a part of a legitimate run of the protocol. Hence there exists matching terms on the network corresponding to each of the input tuples.

Termination. I have also considered using analysis results to help solve one of the basic problems of the model checking technique, namely the problem of termination. As explained it is a problem when to terminate a search when searching an infinite search space for a breach which does not exist. From the analysis we have an over approximation of the knowledge of the intruder, κ. If $\widehat{\kappa} = \kappa$ the intruder will not learn anything new from continuing the search and we can terminate if we are searching for a secrecy breach. But in the case where we search for a secrecy breach which does not exist, we will never end in a situation where $\widehat{\kappa} = \kappa$, so we do not gain anything from this idea.

My initial motivation for representing a protocol as a finite graph, was to be able to implement the techniques described above. However, since these techniques has not been as successful as I had hoped, one could pose the question whether the graph representation is a wise choice. It is fairly straight forward to generate the constraint lists from the graph representation, but the graph contains redundant information. From Figure 4.1 of the graph representing the running example protocol, it is clear to see that each input and output transition appear several times. This is due to the fact that the graph is an explicit representation of the possible interleavings of each of the principal processes. I had hoped that this explicit representation would reveal possibilities of reducing the search space in a manner described above, but this has not been the case. Hence a more compact representation of a protocol and the simulation would be preferable, such as the multi-set rewriting approach described in [8, 16]. It is beyond the scope of this project to investigate further in these considerations.

4.6 Running time

In this section we look at the efficiency of the model checking algorithm and each of the various techniques described.

In general the task of validating a protocol is undecidable, [16]. The model checking techniques rely on simulating the communication of the protocol. Since the processes of each of the principals loop infinitely, the algorithm will not terminate if a given error cannot be found. The efficiency of the various implementations is measured in terms of the maximum length of paths which can be examined. This depend on two things. The number of paths of a given length that has to be considered and the complexity of the constraint lists for each of these paths.

Figure 4.2 shows the number of paths of a given length for the two first approaches in Section 4.4 – the naive approach and the approach where unsatisfiable paths are not extended. It is clear to see a big improvement of the number of paths that has to be considered. However the work of solving the constraints does not differ as much between the two methods as the figure suggests. This is due to the fact that even though fewer paths are considered, the constraint lists that correspond to the unsatisfiable paths are faster to treat, since they are unsatisfiable. I.e. it is the easy part of the work that is avoided.

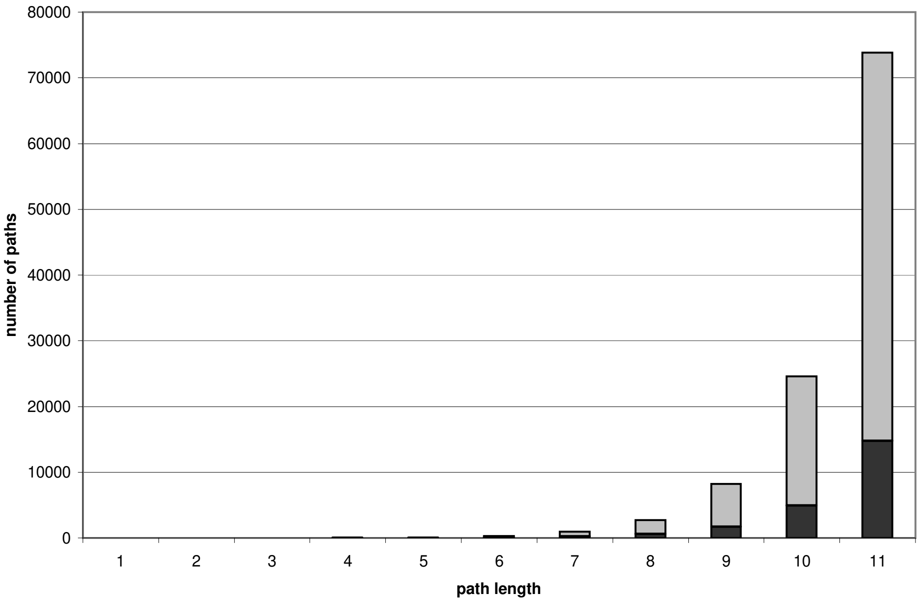

Figure 4.2: Number of paths of a given length that has to be considered for method 1 and 2.

Method 3, where solved constraints are reused has the same number of considered paths of a given length as method 2. The improvement from method 2 to method 3 is not as big as expected. The running time of the algorithm is dominated by the redundancy in the constraint solving as explained above.

The overall running time of the algorithm is exponential in the length of paths we wish to consider. However the current implementation makes it possible to reach a path length sufficient for reconstructing the breaches of interest in the Clark-Jacob library of protocols [11] as we shall see in chapter 6. And as we have seen there are still great room for improvement in the implementation of the constraint solving.

Decryption strategy. A possible improvement of the strategy of decrypting terms in the intruder knowledge is to decrypt on demand. I.e. only perform the decryption if the encrypted terms are actually needed. The problem is to determine when this is the case. I have limited my work not to include this feature, however, decryption of a term $\{t_1, \cdots, t_k\}_{t_0}$ is only considered if it adds new knowledge to $\widehat{\kappa}$. I.e. if $\{t_1, \cdots, t_k\} \nsubseteq \widehat{\kappa}$.

4.7 Inspiration from OFMC[1]

The basic concept of a lazy intruder is described several places in the literature. It it used in the techniques described in [8] but has also been used in earlier approaches. [8] contains a description of the basic principles of the lazy intruder as well as techniques for generating constraints. More specific details about implementing the constraint solving is however omitted in the article.

My contribution. I have implemented the techniques and adapted the constraints to include origin-destination annotations. I have also implemented the capability of searching for specific breaches as opposed to other model checking techniques. Furthermore, I have experimented with the three approaches of search described in Section 4.4.

[1] On-the-fly-Model-Checker described in [8]

Chapter 5

The SAMC tool – implementing the techniques

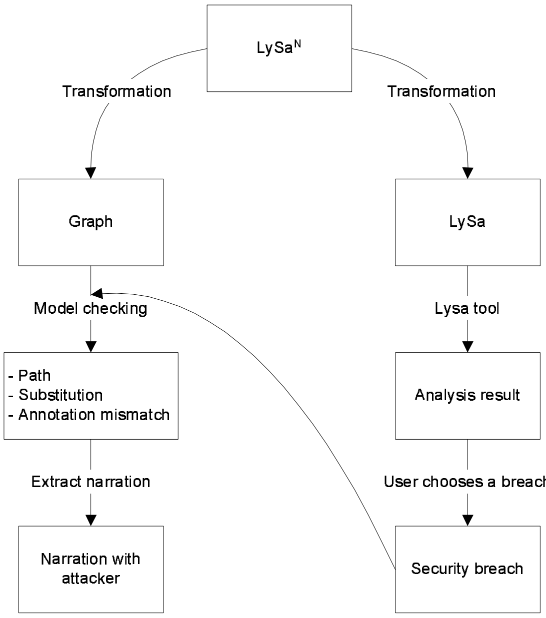

Figure 5.1: Overview of the SAMC tool.

The purpose of this section is to present the reader with a users guide to the SAMC tool as well as to treat some of the interesting implementational details and considerations. Figure 5.1 gives an overview of the tasks performed by the SAMC tool. The starting point is a LySa^{N} specification of a protocol, which is transformed and analyzed. Based on input from the user the model checking routine is executed on the graph representation of the LySA$^{\mathrm{N}}$ specification.

Section 5.1 contains a description of the SML data types corresponding to the grammar of LySA$^{\mathrm{N}}$. The section also treats some relevant aspects of the specification of a protocol. Section 5.2 describes the user interface of the tool.

Details about installing and running the tool can be found in Appendix C.

5.1 Specifying protocols in SML

Protocols are specified in SML using the following data type:

```
type principal = string
type Symbol    = string
type Name      = Symbol
type Var       = Symbol

datatype Form  = ORIG    of Term
               | DEST    of Term

and Term  =      NAME    of Name
               | VAR     of Var
               | DEFVARP of Var
               | DEFVAR  of Var
               | ENC     of Term list * Term * (Term * Form)
               | KEY     of Term
               | FRESH   of Term

datatype ProcP = OUT     of Term list * ProcP
               | INP     of Term list * ProcP
               | NEW     of Term * ProcP
               | NIL

datatype ProcS = NEWS    of Term * ProcS
               | PAR     of ProcS list
               | LOOP    of principal * ProcP
```

The data type corresponds to the grammar of LySA$^{\mathrm{N}}$. `ProcS` and `ProcP` corresponds to processes on level S and P respectively. `DEFVAR` and `DEFVARP` represents the defining occurrence of variables and principal variables respectively.

As an example consider the SML-representation of the running example protocol in Table 5.1. The value `wmf1` has the type `ProcS` and represents the protocol. Note how the principal names are specified for each of the parallel looping processes of type `ProcP`. This is done to be able to create the narration of an attack.

```
val A = NAME("A")
val I = NAME("I")
val B = NAME("B")
val S = NAME("S")
val K = NAME("K")
val Msg = NAME("Msg")

val yA = VAR("yA")
val yA' = DEFVARP("yA")
val yB = VAR("yB")
val yB' = DEFVARP("yB")
val zA = VAR("zA")
val zA' = DEFVARP("zA")
val zB = VAR("zB")
val zB' = DEFVARP("zB")
val yK = VAR("yK")
val yK' = DEFVAR("yK")
val yMsg = VAR("yMsg")
val yMsg' = DEFVAR("yMsg")
val zK = VAR("zK")
val zK' = DEFVAR("zK")

val Pa = NEW(K,
         OUT([A,B,ENC([K],KEY(A),(A,DEST S))],
         NEW(Msg,
         OUT([ENC([Msg],K,(A,DEST B))],
         NIL))))

val Pb = INP([yA',ENC([yK'],KEY(B),(B,ORIG S))],
         INP([ENC([yMsg'],yK,(B,ORIG yA))],
         NIL))

val Ps = INP([zA',zB',ENC([zK'],KEY(zA),(S,ORIG zA))],
         OUT([zA,ENC([zK],KEY(zB),(S,DEST zB))],
         NIL))

val wmf1 = NEWS(KEY(A),NEWS(KEY(B),
           PAR([LOOP("A",Pa),LOOP("B",Pb),LOOP("S",Ps)])))
```

Table 5.1: SML representation of the running example protocol.

The keys `KEY(A)` an `KEY(B)` are restricted om the S-level using the constructor `NEWS`. This does not conform to the wellformedness condition in Table 2.6. The reason to this is explained in Section 5.1.2 below.

5.1.1 Specifying leaks

As we shall see in Section 6 it is often interesting to test the vulnerability of a protocol when an old session key is leaked. Leaking certain information to the intruder is done by creating a separate process which consists of an output containing the relevant terms. To avoid an impact from the extra process on the running time of the model checking routine we use the principal name "leak" when specifying this process: `LOOP("leak",OUT(t1,···,tk))`. This is detected by the model checking routine. The terms are added to the initial knowledge of the intruder and the process is ignored in the model checking.

5.1.2 Restriction on keys

In one aspect the implementation of the transformation does not follow the definitions from section 2. To give a greater flexibility to the user, I have chosen to let the restriction on S-level differ from what is specified in the LySa$^{\mathrm{N}}$ wellformedness condition and the transformation in Tables 2.6 and 2.8 as explained below.

To be able to keep track of principal names and variables the wellformedness condition imposes the restriction that only principal names can be restricted on S-level, $(\nu\, n)S$. At the same time the transformation converts each of these restrictions into a corresponding restriction, $(\nu\, K_n)\lfloor S \rfloor$ in LySa.

The flexibility I introduce in the implementation is the possibility of using restrictions in LySa$^{\mathrm{N}}$ as they are used in LySa. So instead of restricting the principal names on level S, we restrict the keys of the principals as we would have done in LySa:

$$(\nu\, \mathsf{key}[A])(\nu\, \mathsf{key}[B])S$$

This way of restricting names on S-level makes it possible to specify protocols without using the concept of principal variables, and it makes it possible to have restrictions on names on S-level beside from the keys of the principals. We could for instance specify a protocol where each principal has more than one key.

However, it is a problem that we no longer have an implicit way of getting hold of the set of names of type Prin as defined in the wellformedness condition. The problem is solved by letting the user explicitly enumerate the set of principals that should be used when unfolding processes and removing principal variables. This solution also has the side effect that it is possible to experiment when unfolding the processes. The user might omit some of the principals to limit the combinatorial explosion of the unfolding.

5.2 User interface

The SAMC tool has a fairly simple command line user interface, which guides the user through the steps that are taken in order to analyze and reconstruct a breach in a protocol.

Predefined Protocols. The sml-file `protocols.sml` listed in Appendix D contains the SML representations of protocols and test cases used in this project. Some of the specifications are protocols from the Clark-Jacob library [11]. Others are simple examples that I have used to test various properties of the SAMC tool. For a protocol to be available to the SAMC tool it has to exist in the list `protocolList` defined in the end of the file `protocols.sml`. The list has SML type `(string * ProcS) list`, where `procS` is a LySA$^{\mathrm{N}}$ process. The string is the name of the protocol. By default the list is defined as:

```
val protocolList = [
  ("Test case Annot 1",SAnnot1),
  ("Test case Annot 2",SAnnot2),
  ("Test case Annot 3",SAnnot3),
  ("Test case Complex key",SComplexKey),
  ("Test case Intruder key and crypto-point",SIKey),
  ("WMF, variant 1, (Running example protocol)",wmfv1),
  ("WMF, variant 2, principal names encrypted",wmfv2),
  ("WMF, variant 2, principal names encrypted, leak",wmfv2Leak),
  ("needham schroeder with leak",nsLeak),
  ("andrew secure RPC ",andrew),
  ("andrew secure RPC with leak",andrewLeak),
  ("yahalom with leak",yahalom)
]
```

If a user wishes to run the SAMC tool on protocols other than the ones specified in this list, they have to be added to the list prior to running the tool.

Running the tool. The source files are compiled and the tool is started by executing the command

```
use "ui.sml";
```

from an SML/NJ command prompt. The working directory must be `/samc/test`. The protocols of the list described above are displayed on the screen as a numbered list and the user is asked choose a protocol:

```
1. Test case Annot 1
2. Test case Annot 2
3. Test case Annot 3
4. Test case Complex key
5. Test case Intruder key and crypto-point
6. WMF, variant 1, (Running example protocol)
7. WMF, variant 2, principal names encrypted
8. WMF, variant 2, principal names encrypted, leak
9. needham schroeder with leak
10. andrew secure RPC
```

```
11. andrew secure RPC with leak
12. yahalom with leak
Choose a protocol by entering number:
```

After a protocol has been chosen, a LySa^{N} specification of the protocol is displayed:

```
LySaN specification of wmf1
(ny Key[A])(ny Key[B])[
<A,B,{K*}Key[A]>.<{Msg*}K>.0,
($yA,{%yK}Key[B]).({%yMsg}yK).0,
($zA,$zB,{%zK}Key[zA]).<zA,{zK}Key[zB]>.0
]
Choose a set of principals:
```

A set of principals are chosen by entering principal names separated by comma. E.g. `A,B,I` for a setup containing the principals A and B together with the intruder, I. Note that the name I is reserved for the intruder. The protocol is unfolded with the specified principals as explained in Section 2.2. The static analysis is performed using functions from the LySa tool [1], and the result is displayed.

```
psi =
(S,S),(S,B),(I,B),(A,I),(A,B),
kappa =
[Msg,K,_NDY,I,K'I,B,A]
type a breach:
```

The user types a breach, e.g. `(A,I)` .The program parses the breach and detects whether it is a secrecy breach or an authentication breach. Note the conventions explained in the next section.

The model checking routine is run on the protocol and the given breach. During execution the program prints out the length l and number of paths of length l which are being processed at the moment.

```
searching 2 paths of length 2

searching 4 paths of length 3
..
searching 8 paths of length 4
......
searching 17 paths of length 5
..............................
searching 39 paths of length 6
.......................................................
```

To help illustrate the progress, a dot is printed for every 100 search states generated. This also helps the user to decide when to "give up". As explained the SAMC tool does not terminate if the breach cannot be found.

When a breach is found the narration and the list of variable constraints, the substitution and the annotation mismatch of the final search state is displayed. The following is the result from the running example protocol and the secrecy breach `Msg`:

```
narration:
A -> I: <A,B,{K_1}KEY[A](A,DEST S)>
I -> S:(A,I,{K_1}KEY[A](S,ORIG A))
A -> I: <{Msg_1}K_1(A,DEST B)>
S -> I: <A,{K_1}KEY[I](S,DEST I)>

final state:
vcs =
z:Msg_1,K_1,A,{K_1}KEY[I](S,DEST I)*,{Msg_1}K_1(A,DEST B)*,A,B,
  {K_1}KEY[A](A,DEST S),I,KEY[I]
z:{Msg_1}K_1(A,DEST B),A,B,{K_1}KEY[A](A,DEST S),I,KEY[I]
z:A,B,{K_1}KEY[A](A,DEST S),I,KEY[I]
z:I,KEY[I]

sub =
[zB_1->I,zA_1->A,zK_1->K_1]

psi =
((I,B)(A,A))
```

The program terminates after displaying the result. To use the tool on other examples the program must be restarted.

5.3 Name conventions

Below we look at the name conventions in the SAMC tool.

Reserved names. The variables x_i are reserved and are used when unfolding principal variables. The names K_n, where $\tau(n) = \mathsf{Prin}$ are also reserved and are used when transforming $\mathsf{key}[n]$.

Keys as secrecy breaches The construction $\mathsf{key}[n]$ is transformed into K_n and is displayed in the tool as `K'n`. When searching for the secrecy breach however, we have to use the original name for the key as specified in LySa$^{\mathrm{N}}$. In the SAMC tool we type `KEY(n)` when asked to input a breach, (representing the secrecy breach where $\mathsf{key}[n] \in \kappa$).

Fresh names and variables. Names restricted on level P and variables has to be renamed when principal variables are unfolded in order to avoid confusion between variables and fresh names of different LySa processes. The names and variables are annotated with the principal names corresponding to the substitution that the process is based on as explained in Section 2.3. A variable z_K from the unfolded server process of the running example protocol corresponding to the substitution $[z_A \mapsto A, z_B \mapsto B]$ is annotated z_K^{AB}. In the SAMC tool this becomes `zK''AB`. These annotations are introduced in the transformation and are hence not used in the model checking which is based on the original LySa$^{\mathrm{N}}$ process. If the analysis reports that session key `K''AB` is in κ, we reconstruct the trace by searching for the secrecy breach `K`, not `K''AB`.

Session indices. Recall that session indices are used in the model checking routine to be able to distinguish between names and variables from different sessions of a protocol. (Section 4.3.4). The analysis is flow insensitive. Hence the session indices are not part of the analysis result. If we for instance wish to search for the breach `Msg` in the running example protocol, where `Msg` is from the second session, we specify the breach as `Msg_2`. When no indices are specified we search for any instance – (always finding the shortest narration) – of Msg.

Chapter 6

Tests and Results

I have tested the SAMC tool on a number of small test cases illustrating the capabilities of the tool. The results are presented in Section 6.1. Furthermore, I have tested the tool on a number of protocols from the Clark-Jacob library. The analysis results as well as the reconstructed attacks are presented in Section 6.2.

The following sections contains a LySA$^{\mathrm{N}}$ representation of each of the test cases. The corresponding narration can be found in Appendix A.

6.1 Test cases

I have tested that the following features of the SAMC tool are functioning correctly. The tests are fairly simple and do not cover all cases but illustrate interesting aspects.

- Detecting mismatch of crypto-points
- Decryption with complex key
- Using session indices
- Using intruder crypto-point and key

6.1.1 Detecting mismatch of cryptopoints

The test is divided into three cases corresponding to the three types of mismatch as explained in Section 4.3.3:

1. (n, I). An encryption term is wrongfully decrypted by the intruder.

2. (I, n). The intruder has generated an encryption term, which is wrongfully decrypted by a principal.

3. (n_1, n_2). An encryption term generated at n_1 is wrongfully decrypted at n_2, n_1 and n_2 being crypto-points of two legitimate principals.

1. Mismatch of type (n, I)

LYSA$^{\mathrm{N}}$ specification of test process:

$$[(\%x_K).(\nu\, Msg)\langle\{Msg\}^{A}_{x_K}\ [\mathsf{dest}\ B]\ \rangle.\mathrm{RET}]$$

The process lets the intruder choose a key, x_K, generates a new message and sends out the message encrypted with x_K. The intended destination is B, hence we expect $(A, I) \in \psi$.

Analysis result:

κ	$Msg, I, \mathsf{key}[I]$
ψ	(A, I)

Narration for breach (A, I)

$$\begin{array}{lllll} 1. & I & \rightarrow & P & : & (x^1_K) \\ 2. & P & \rightarrow & I & : & \langle\{Msg^1\}^{A}_{x^1_K}\ [\mathsf{dest}\ B]\ \rangle \end{array}$$

The value of x^1_K is irrelevant as long as it comes from the knowledge of the intruder. The variable is therefore not assigned a value in the narration.

2. Mismatch of type (I, n)

LYSA$^{\mathrm{N}}$ specification of test process:

$$[(\nu\, K)\langle K\rangle.(\{\%y_{Msg}\}^{B}_{K}\ [\mathsf{orig}\ A]\ )]$$

The process creates a fresh key and outputs it on the network. The process then accepts an encryption term encrypted with K. The origin of the encryption is expected to be A, but since the key K has been send out on the network it is possible for the intruder to generate a matching encryption term.

κ	$K, I, \mathsf{key}[I]$
ψ	(I, B)

Narration for breach (I, B)

$$\begin{array}{llllll} 1. & P & \rightarrow & I & : & \langle K^1\rangle \\ 2. & I & \rightarrow & P & : & (\{y^1_{Msg}\}^{B}_{K^1}\ [\mathsf{orig}\ A]\ ) \end{array}$$

3. Mismatch of type (n_1, n_2)

LySA[N] specification of test protocol:

$$\begin{array}{l}(\nu\, K_{AB})\\ \quad [(\nu\, M_S)(\nu\, M_P)\langle\{M_S\}^{A^1}_{K_{AB}}\ [\mathsf{dest}\ B^1]\ \rangle.\langle\{M_P\}^{A^2}_{K_{AB}}\ [\mathsf{dest}\ B^2]\ \rangle]\\ \mid \quad [(\{\%x_S\}^{B^1}_{K_{AB}}\ [\mathsf{orig}\ A^1]\ ).(\{\%x_P\}^{B^2}_{K_{AB}}\ [\mathsf{orig}\ A^2]\ )\langle x_P\rangle]\end{array}$$

The protocol consists of two processes representing principals A and B. Principal A creates two messages M_S (intended to be kept secret) and M_P (intended to be public). A sends both messages to B encrypted with a secret key shared between A and B. B decrypts the second message, which is intended to be the public message, and sends it out on the network.

The analysis returns the following result:

κ	$M_S, M_P, I, \mathsf{key}[I]$
ψ	$(A^1, B^2), (A^2, B^1)$

Narration for breach (A^1, B^2):

$$\begin{array}{llcll}1. & A & \rightarrow & I & : \langle\{M_S^1\}^{A^1}_{K_{AB}}\ [\mathsf{dest}\ B^1]\ \rangle\\ 2. & I & \rightarrow & B & : (\{M_S^1\}^{B^1}_{K_{AB}}\ [\mathsf{orig}\ A^1]\ )\\ 3. & I & \rightarrow & B & : (\{M_S^1\}^{B^2}_{K_{AB}}\ [\mathsf{orig}\ A^2]\ )\end{array}$$

6.1.2 Decryption with complex keys

LySA[N] specification of test process:

$$[(\$x_A).(\nu\, Msg)\langle\{Msg\}^S_{\mathsf{key}[x_A]}\ [\mathsf{dest}\ A]\ \rangle]$$

The process accepts a principal name, x_A, and sends an encryption term which is encrypted with the key of principal x_A. The analysis returns the following result:

κ	$Msg, I, \mathsf{key}[I]$
ψ	(S, I)

Narration for breach Msg:

$$\begin{array}{llcll}1. & I & \rightarrow & S & : (I)\\ 2. & S & \rightarrow & I & : \langle\{Msg^1\}^S_{\mathsf{key}[I]}\ [\mathsf{dest}\ A]\ \rangle\end{array}$$

The intruder binds x_A to I and can hence decrypt the encryption term $\{Msg^1\}^S_{\mathsf{key}[I]}\ [\mathsf{dest}\ A]$.

6.1.3 Assigning session indices

We reuse the process from the previous test:

$$[(\$x_A).(\nu\, Msg)\langle\{Msg\}^S_{\mathsf{key}[x_A]}\ [\mathsf{dest}\ A]\ \rangle]$$

with the following analysis result:

κ	$Msg, I, \mathsf{key}[I]$
ψ	(S, I)

This time we search for the name Msg corresponding to the second session, Msg^2.

Narration for breach Msg^2:

$$\begin{array}{lllll} 1. & I & \rightarrow & S & : & (x_A^1) \\ 2. & S & \rightarrow & I & : & \langle \{Msg^1\}^S_{\mathsf{key}[x_A^1]} \ [\mathsf{dest}\ A] \ \rangle \\ 3. & I & \rightarrow & S & : & (I) \\ 4. & S & \rightarrow & I & : & \langle \{Msg^2\}^S_{\mathsf{key}[I]} \ [\mathsf{dest}\ A] \ \rangle \end{array}$$

The value of x_A^1 is irrelevant for the attack. x_A^2 however, is substituted with I enabling the breach.

6.1.4 Using intruder crypto-point and key

It should be possible to implement protocols where the intruder is a legitimate part of the protocol. The following process tests that the use of the intruder key, $\mathsf{key}[I]$, and the intruder name as crypto-point works as intended.

LySa$^{\mathrm{N}}$ specification of test process:

$$[(\nu\ Msg)\langle \{Msg\}^S_{\mathsf{key}[I]} \ [\mathsf{dest}\ I] \ \rangle]$$

Analysis result:

κ	$Msg, I, \mathsf{key}[I]$
ψ	$\emptyset$

As expected ψ is empty, since the intruder is the intended destination. The intruder does in fact decrypt the message, which can be seen from the fact that $Msg \in \kappa$.

Narration for breach Msg:

$$\begin{array}{llllll} 1. & S & \rightarrow & I & : & \langle \{Msg^1\}^S_{\mathsf{key}[I]} \ [\mathsf{dest}\ I] \ \rangle \end{array}$$

As expected the intruder simply receives the encryption $\{Msg^1\}^S_{\mathsf{key}[I]} \ [\mathsf{dest}\ I]$.

6.2 Testing SAMC tool on protocols

Table 6.1 gives an overview of the analyzed protocols and the errors found by the analysis. Some of the protocols have also been tested in a setup where an old session key has been leaked. In the subsequent sections we look at the reconstruction of each of the errors displayed in Table 6.1.

The crypto-point "∘" is used whenever we wish to ignore elements in ψ from the corresponding encryption. This is for instance the case when we intentionally leak terms to the intruder.

Protocol	ψ	ψ – Old session key leaked
WMF variant 1	(S,S) (S,B) (A,I) (I,B) (A,B)	*n/a*
WMF variant 2	(S,S)	(I,B)
Needham-Schroeder	(A^3,A^2)	(A^3,A^2) (B^2,I) (I,B^3) (I,B^4) (B^2,B^3)
Andrew Secure RPC	$\emptyset$	(A,I)
Yahalom	$\emptyset$	(I,B)

Table 6.1: Mismatch in origin-destination annotations.

6.2.1 WMF variant 1

LySA[N] specification:

$$
\begin{array}{l}
(\nu\, \mathsf{key}[A])(\nu\, \mathsf{key}[B]) \\
\quad [(\nu\, K) \\
\qquad \langle A, B, \{K\}^{A}_{\mathsf{key}[A]}\ [\mathsf{dest}\ S]\ \rangle. \\
\qquad (\nu\, Msg)\langle \{Msg\}^{A}_{K}\ [\mathsf{dest}\ B]\ \rangle \\
\quad] \\
\mid \quad [(\$y_A, \{\%y_K\}^{B}_{\mathsf{key}[B]}\ [\mathsf{orig}\ S]\ ). \\
\qquad (\{\%y_{Msg}\}^{B}_{y_K}\ [\mathsf{orig}\ y_A]\ ) \\
\quad] \\
\mid \quad [(\$z_A, \$z_B, \{\%z_K\}^{S}_{\mathsf{key}[z_A]}\ [\mathsf{orig}\ z_A]\ ). \\
\qquad \langle z_A, \{z_K\}^{S}_{\mathsf{key}[z_B]}\ [\mathsf{dest}\ z_B]\ \rangle \\
\quad]
\end{array}
$$

The SAMC tool gives the following analysis result:

κ	$Msg, K, I, \mathsf{key}[I], B, A$
ψ	$(S,S),(S,B),(I,B),(A,I),(A,B)$

Note that encryption terms in κ are not shown above.

The model checking routine is run on breaches from κ and ψ. The result is displayed in Table 6.2

Note that the attack narrations corresponding to the secrecy breach K does not imply any authentication breach as is often the case.

Breach	Narration					
K	1.	A	$\rightarrow$	I	:	$A, B, \{K^1\}_{\mathsf{key}[A]}$
	2.	I	$\rightarrow$	S	:	$A, I, \{K^1\}_{\mathsf{key}[A]}$
	3.	S	$\rightarrow$	I	:	$A, \{K^1\}_{\mathsf{key}[I]}$
Msg	1.	A	$\rightarrow$	I	:	$A, B, \{K^1\}_{\mathsf{key}[A]}$
	2.	I	$\rightarrow$	S	:	$A, I, \{K^1\}_{\mathsf{key}[A]}$
	3.	A	$\rightarrow$	I	:	$\{Msg^1\}_{K^1}$
	4.	S	$\rightarrow$	I	:	$A, \{K^1\}_{\mathsf{key}[I]}$
(S, S)	1.	I	$\rightarrow$	S	:	$I, z_B^1, \{z_K^1\}_{\mathsf{key}[I]}$
	2.	S	$\rightarrow$	I	:	$I, \{z_K^1\}_{\mathsf{key}[z_B^1]}$
	3.	I	$\rightarrow$	S	:	$z_B^1, z_B^2, \{z_K^1\}_{\mathsf{key}[z_B^1]}$
(S, B)	1.	I	$\rightarrow$	S	:	$I, B, \{\mathsf{key}[I]\}_{\mathsf{key}[I]}$
	2.	S	$\rightarrow$	I	:	$I, \{\mathsf{key}[I]\}_{\mathsf{key}[B]}$
	3.	I	$\rightarrow$	B	:	$y_A^1, \{\mathsf{key}[I]\}_{\mathsf{key}[B]}$
	4.	I	$\rightarrow$	S	:	$I, I, \{z_K^2\}_{\mathsf{key}[I]}$
	5.	S	$\rightarrow$	I	:	$I, \{z_K^2\}_{\mathsf{key}[I]}$
	6.	I	$\rightarrow$	B	:	$\{z_K^2\}_{\mathsf{key}[I]}$
(I, B)	1.	I	$\rightarrow$	S	:	$I, B, \{z_K^1\}_{\mathsf{key}[I]}$
	2.	S	$\rightarrow$	I	:	$I, \{z_K^1\}_{\mathsf{key}[B]}$
	3.	I	$\rightarrow$	B	:	$y_A^1, \{z_K^1\}_{\mathsf{key}[B]}$
	4.	I	$\rightarrow$	B	:	$\{y_{Msg}^1\}_{z_K^1}$
(A, I)	1.	A	$\rightarrow$	I	:	$A, B, \{K^1\}_{\mathsf{key}[A]}$
	2.	I	$\rightarrow$	S	:	$A, I, \{K^1\}_{\mathsf{key}[A]}$
	3.	A	$\rightarrow$	I	:	$\{Msg^1\}_{K^1}$
	4.	S	$\rightarrow$	I	:	$A, \{K^1\}_{\mathsf{key}[I]}$
(A, B)	1.	A	$\rightarrow$	I	:	$A, B, \{K^1\}_{\mathsf{key}[A]}$
	2.	I	$\rightarrow$	S	:	$A, B, \{K^1\}_{\mathsf{key}[A]}$
	3.	A	$\rightarrow$	I	:	$\{Msg^1\}_{K^1}$
	4.	S	$\rightarrow$	I	:	$A, \{K^1\}_{\mathsf{key}[B]}$
	5.	I	$\rightarrow$	B	:	$y_A^1, \{K^1\}_{\mathsf{key}[B]}$
	6.	I	$\rightarrow$	B	:	$\{Msg^1\}_{K^1}$

Table 6.2: Model checking results from WMF variant 1

6.2.2 WMF variant 2

LySa$^{\mathrm{N}}$ specification:

$$
\begin{array}{l}
(\nu\, \mathsf{key}[A])(\nu\, \mathsf{key}[B]) \\
\quad [(\nu\, K) \\
\qquad \langle A, \{B, K\}^{A}_{\mathsf{key}[A]} \ [\mathsf{dest}\ S]\ \rangle. \\
\qquad (\nu\, Msg)\langle \{Msg\}^{A}_{K}\ [\mathsf{dest}\ B]\ \rangle \\
\quad] \\
| \quad [(\{\$y_A, \%y_K\}^{B}_{\mathsf{key}[B]}\ [\mathsf{orig}\ S]\ ). \\
\qquad (\{\%y_{Msg}\}^{B}_{yK}\ [\mathsf{orig}\ yA]\ ) \\
\quad] \\
| \quad [(\$z_A, \{\$z_B, \%z_K\}^{S}_{\mathsf{key}[z_A]}\ [\mathsf{orig}\ z_A]\ ). \\
\qquad \langle \{z_A, z_K\}^{S}_{\mathsf{key}[z_B]}\ [\mathsf{dest}\ z_B]\ \rangle \\
\quad]
\end{array}
$$

The protocol is also tested with an old session key leaked. The following terms are added to the knowledge of the intruder:

$$K_{old}, \{B, K_{old}\}^{\circ}_{\mathsf{key}[A]}\ [\mathsf{dest}\ \circ]\ , \{A, K_{old}\}^{\circ}_{\mathsf{key}[B]}\ [\mathsf{dest}\ \circ]\ , \{Msg_{old}\}^{\circ}_{K_{old}}\ [\mathsf{dest}\ \circ]$$

resulting in the following analysis result.

	No leak	Leak
κ	I, K_I, B, A	$I, K_I, B, A, K_{old}, Msg_{old}$
ψ	(S, S)	$(S, S), (I, B)$

The model checking routine is run on the two breaches from ψ:

Breach	Narration
(S, S)	1. $I \rightarrow S : I, \{z^1_B, z^1_K\}_{\mathsf{key}[I]}$ 2. $S \rightarrow I : \{I, z^1_K\}_{\mathsf{key}[z^1_B]}$ 3. $I \rightarrow S : z^1_B, \{I, z^1_K\}_{\mathsf{key}[z^1_B]}$
(I, B)	1. $I \rightarrow B : \{A, K_{old}\}_{\mathsf{key}[B]}$ 2. $I \rightarrow B : \{y_M sg^1\}_{K_{old}}$

6.2.3 Needham-Schroeder.

LySa^N specification:

$$
\begin{array}{l}
(\nu\, \mathsf{key}[A])(\nu\, \mathsf{key}[B]) \\
\quad [(\nu\, N^A)(\nu\, Msg) \\
\quad\quad \langle A, B, N^A \rangle. \\
\quad\quad (\{B, N^A, \%x_K, \%x_{Enc}\}^{A^1}_{\mathsf{key}[A]}\ [\mathsf{orig}\ S]\ ). \\
\quad\quad \langle x_{Enc} \rangle. \\
\quad\quad (\{\%x_{N^B}\}^{A^2}_{x_K}\ [\mathsf{orig}\ B^2]\ ). \\
\quad\quad \langle \{\{x_{N^B}\}^{\circ}_{succ}\ [\mathsf{dest}\ \circ]\ \}^{A^3}_{x_K}\ [\mathsf{dest}\ B^3]\ \rangle. \\
\quad\quad \langle \{Q, Msg\}^{A^4}_{x_K}\ [\mathsf{dest}\ B^4]\ \rangle \\
\quad] \\
| \quad [(\nu\, N^B) \\
\quad\quad (\{A, \%y_K\}^{B^1}_{\mathsf{key}[B]}\ [\mathsf{orig}\ S]\ ). \\
\quad\quad \langle \{N_B\}^{B^2}_{y_K}\ [\mathsf{dest}\ A^2]\ \rangle. \\
\quad\quad (\{\{N_B\}^{\circ}_{succ}\ [\mathsf{orig}\ \circ]\ \}^{B^3}_{y_K}\ [\mathsf{orig}\ A^3]\ ). \\
\quad\quad (\{Q, \%y_{Msg}\}^{B^4}_{y_K}\ [\mathsf{orig}\ A^4]\ ) \\
\quad] \\
| \quad [(\nu\, K) \\
\quad\quad (A, B, \%z_{N^A}). \\
\quad\quad \langle \{B, z_{N^A}, K, \{A, K\}^{S}_{\mathsf{key}[B]}\ [\mathsf{dest}\ B^1]\ \}^{S}_{\mathsf{key}[A]}\ [\mathsf{dest}\ A^1]\ \rangle \\
\quad]
\end{array}
$$

The name Q is used to avoid a simple type flaw as mentioned in [1].

The analysis only reports of one breach in this protocol, namely the authentication breach (A^3, A^2). The model checking routine does not find this breach (within reasonable time). From these results it is of cause not possible to conclude with certainty that the breach is a false positive. However no matter whether the breach exists or not it is clear to see why it is (probably wrongfully) part of the analysis result. Due to the fact that the analysis is flow insensitive we have that the following term and pattern are matched:

$$
\begin{array}{ll}
 & \{\%x_{N^B}\}^{A^2}_{x_K}\ [\mathsf{orig}\ B^2] \\
\triangleright & \{\{x_{N^B}\}^{\circ}_{succ}\ [\mathsf{dest}\ \circ]\ \}^{A^3}_{x_K}\ [\mathsf{dest}\ B^3]
\end{array}
$$

But adding flow information we realize that the key x_K of the two encryptions comes from two different sessions. From a basic assumption (also made by the model checking routine) that session keys are always fresh, we can hence conclude that the match is not possible.

The protocol is also tested with an old session key leaked. The following terms are added to the knowledge of the intruder:

$$\begin{array}{l} K_{old}, N^A_{old}, N^B_{old}, \{A, K_{old}\}^{\circ}_{\mathsf{key}[B]}\ [\mathsf{dest}\ \circ]\ , \\ \{B, N^A_{old}, K_{old}, \{A, K_{old}\}^{\circ}_{\mathsf{key}[B]}\ [\mathsf{dest}\ \circ]\ \}^{\circ}_{\mathsf{key}[A]}\ [\mathsf{dest}\ \circ]\ , \\ \{Q, Msg_{old}\}^{\circ}_{K_{old}}\ [\mathsf{dest}\ \circ]\ , \{N^B_{old}\}^{\circ}_{K_{old}}\ [\mathsf{dest}\ \circ]\ , \\ \{\{N^B_{old}\}^{\circ}_{succ}\ [\mathsf{dest}\ \circ]\ \}^{\circ}_{K_{old}}\ [\mathsf{dest}\ \circ] \end{array}$$

The SAMC tool gives the following analysis result:

κ	$Msg_{old}, N^B_{old}, N^A_{old}, K_{old}, N^A, N^B, succ, Q, A, B, I, K_I$
ψ	$(B^2, I), (I, B^3), (I, B^4), (A^3, A^2)$

The model checking routine is run on breaches from ψ

Breach	Narration
(B^2, I)	1. $I \rightarrow B : \{A, K_{old}\}_{\mathsf{key}[B]}$ 2. $B \rightarrow I : \{N^1_B\}_{K_{old}}$ 3. $I \rightarrow S : A, B, z^1_{N_A}$
(I, B^3)	1. $I \rightarrow B : \{A, K_{old}\}_{\mathsf{key}[B]}$ 2. $B \rightarrow I : \{N^1_B\}_{K_{old}}$ 3. $I \rightarrow B : \{\{N^1_B\}_{succ}\}_{K_{old}}$
(I, B^4)	1. $I \rightarrow B : \{A, K_{old}\}_{\mathsf{key}[B]}$ 2. $B \rightarrow I : \{N^1_B\}_{K_{old}}$ 3. $I \rightarrow B : \{\{N^1_B\}_{succ}\}_{K_{old}}$ 4. $I \rightarrow B : \{Q, y^1_{Msg}\}_{K_{old}}$
(B^2, B^3)	*no reconstruction*

The breach (B^2, B^3) is a false positive introduced by the transformation. The error should occur by matching the following value and pattern:

$$\{N^B\}^{B^2}_{y_K}\ [\mathsf{dest}\ A^2]\ \rhd \{\{N^B\}^{\circ}_{succ}\ [\mathsf{orig}\ \circ]\ \}^{B^3}_{y_K}\ [\mathsf{orig}\ A^3]$$

This is clearly not a possible match. In the corresponding LySA process however we have the following fragment of the process of principal B:

$$\begin{array}{l} \langle \{N^B\}^{B^2}_{y_K}\ [\mathsf{dest}\ A^2]\ \rangle. \\ (; x_4). \\ \mathsf{decrypt}\ x_4\ \mathsf{as}\ \{; x_5\}^{B^3}_{y_K}\ [\mathsf{orig}\ A^3]\ \mathsf{in} \\ \mathsf{decrypt}\ x_5\ \mathsf{as}\ \{N^B;\}^{\circ}_{succ}\ [\mathsf{orig}\ \circ]\ \mathsf{in}\ \cdots \end{array}$$

If the intruder matches the input, $(; x_4)$, with the output, $\langle \{N^B\}^{B^2}_{y_K}\ [\mathsf{dest}\ A^2]\ \rangle$, the first decryption will succeed, binding x_5 to N^B and adding the element (B^2, B^3) to ψ. The second encryption, however will not succeed.

6.2.4 Andrew Secure RPC

LySA[N] specification:

$$
\begin{array}{l}
(\nu\, K)\\
\quad [(\nu\, N^A)\\
\qquad \langle A, \{u_1, N^A\}_K^{A^1} \ [\mathsf{dest}\ B^1]\ \rangle.\\
\qquad (\{u_2, \{N^A\}_{succ}^{\circ}\ [\mathsf{orig}\ \circ]\ , \%x_{N^B}\}_K^{A^2}\ [\mathsf{orig}\ B^2]\ ).\\
\qquad \langle \{u_3, \{x_{N^B}\}_{succ}^{\circ}\ [\mathsf{dest}\ \circ]\ \}_K^{A^3}\ [\mathsf{dest}\ B^3]\ \rangle.\\
\qquad (\{u_4, \%x_{K'}, \%x_{N'^B}\}_K^{A^4}\ [\mathsf{orig}\ B^4]\ )\\
\qquad (\nu\, Msg)\langle \{Msg\}_{x_{K'}}^{A}\ [\mathsf{dest}\ B]\ \rangle\\
\quad]\\
\mid \quad [(\nu\, N^B)\\
\qquad (A, \{u_1, \%y_{N^A}\}_K^{B^1}\ [\mathsf{orig}\ A^1]\ ).\\
\qquad \langle \{u_2, \{y_{N^A}\}_{succ}^{\circ}\ [\mathsf{dest}\ \circ]\ , N^B\}_K^{B^2}\ [\mathsf{dest}\ A^2]\ \rangle.\\
\qquad (\{u_3, \{N^B\}_{succ}^{\circ}\ [\mathsf{orig}\ \circ]\ \}_K^{B^3}\ [\mathsf{orig}\ A^3]\ ).\\
\qquad (\nu\, K')(\nu\, N'^B)\langle \{u_4, K', N'^B\}_K^{B^4}\ [\mathsf{dest}\ A^4]\ \rangle.\\
\qquad (\{\%y_{Msg}\}_{K'}^{B}\ [\mathsf{orig}\ A]\ )\\
\quad]
\end{array}
$$

The analysis does not report any breaches.

The protocol is tested with an old session key leaked. The following terms are added to the knowledge of the intruder:

$$K_{old}, \{K_{old}, N_{old}^B\}_K^{\circ}\ [\mathsf{dest}\ \circ]$$

The SAMC tool gives the following analysis result:

κ	$Msg, A, succ, K_{old}, N_{old}^B, I, K_I$
ψ	(A, I)

The model checking routine is run on breaches from κ and ψ

Breach	Narration
Msg	1. $A \rightarrow I : A, \{u_1, N^{A^1}\}_K$ 2. $I \rightarrow B : A, \{u_1, N^{A^1}\}_K$ 3. $B \rightarrow I : \{u_2, \{N^{A^1}\}_{succ}, N^{B^1}\}_K$ 4. $I \rightarrow A : \{u_2, \{N^{A^1}\}_{succ}, N^{B^1}\}_K$ 5. $A \rightarrow I : \{u_3, \{N^{B^1}\}_{succ}\}_K$ 6. $I \rightarrow A : \{u_4, K_{old}, N_{old}^B\}_K$ 7. $A \rightarrow I : \{Msg^1\}_{K_{old}}$
(A, I)	– identical –

6.2.5 Yahalom

LySa$^{\mathrm{N}}$ specification:

$$
\begin{array}{l}
(\nu\, \mathsf{key}[A])(\nu\, \mathsf{key}[B]) \\
\quad [(\nu\, N^A) \\
\quad\quad \langle A, N^A\rangle. \\
\quad\quad (\{B, N^A, \%x_K, \%x_{N^B}\}^A_{\mathsf{key}[A]}\ [\mathsf{orig}\ S]\ , \%x_{Enc}). \\
\quad\quad \langle x_{Enc}, \{x_{N^B}\}^A_{x_K}\ [\mathsf{dest}\ B]\ \rangle. \\
\quad\quad (\nu\, Msg)\langle\{Msg\}^A_{x_K}\ [\mathsf{dest}\ B]\ \rangle \\
\quad] \\
|\quad [(\$y_A, \%y_{N^A}). \\
\quad\quad (\nu\, N^B)\langle B, \{y_A, y_{N^A}, N^B\}^B_{\mathsf{key}[B]}\ [\mathsf{dest}\ S]\ \rangle. \\
\quad\quad (\{y_A, \%y_K\}^B_{\mathsf{key}[B]}\ [\mathsf{orig}\ S]\ , \{N^B\}^B_{y_K}\ [\mathsf{orig}\ y_A]\ ). \\
\quad\quad (\{\%y_{Msg}\}^B_{y_K}\ [\mathsf{orig}\ y_A]\ ) \\
\quad] \\
|\quad [(\$z_B, \{\$z_A, \%z_{N^A}, \%z_{N^B}\}^S_{\mathsf{key}[z_B]}\ [\mathsf{orig}\ z_B]\ ). \\
\quad\quad (\nu\, K)\langle\{z_B, z_{N^A}, K, z_{N^B}\}^S_{\mathsf{key}[z_A]}\ [\mathsf{dest}\ z_A]\ , \{z_A, K\}^S_{\mathsf{key}[z_B]}\ [\mathsf{dest}\ z_B]\ \rangle \\
\quad]
\end{array}
$$

The analysis does not report any breaches in the protocol. When the protocol is tested with an old session key leaked, there appears a false positive, $(I, B) \in \psi$. The model checking routine does not find the breach and it is also stated as being a false positive in [1].

Chapter 7

Evaluation

7.1 Improvements

Exploring analysis result. As described in Section 3.3 the infinite set κ from the output of the LySa tool is represented using a tree grammar. A problem arises when we unfold this tree grammar, because of a combinatorial explosion. The solution in this project is to consider a limited version of κ that only contains names. I.e. we ignore encryption terms. It would, however, be nice to be able to make it possible to explore κ via a user interface, which enabled "manual" unfolding of certain nonterminals of the grammars representing κ. I have implemented functions for this, but I have not included the feature in the user interface.

Narration. In the case where the intruder passes a message unaltered from one principal to another, the corresponding two steps of the attack narration could be combined into one. As an example consider the two first steps of the attack narration of the secrecy breach *Msg* in Andrew Secure RPC.

$$\begin{array}{llcll} 1. & A & \rightarrow & I & : & A, \{u_1, N^{A^1}\}_K \\ 2. & I & \rightarrow & B & : & A, \{u_1, N^{A^1}\}_K \end{array}$$

These two steps could be represented by the single step

$$\begin{array}{llcll} 1. & A & \rightarrow & B & : & A, \{u_1, N^{A^1}\}_K \end{array}$$

This compression would be fairly simple to implement.

Asymmetric encryption. The LySa tool includes facilities for asymmetric encryption. To limit my work I have chosen not to include these features in this project. It would however make it possible to experiment with a larger number of protocols.

Crypto-points with indices. As mentioned in Section 2.3 it is in many cases necessary to have distinct crypto-points within a process. This is not possible if principal variables are used. Hence in some of the tests I have had to specify protocols without the use of principal variables. The problem could be solved by introducing the possibility of using indices together with the principal name. A crypto-point would consist of a pair (n, i) of a name and an index.

Optimization of model checking. The implementation of the model checking routine leaves great room for improvement. However, the SAMC tool has been able to handle all test cases that I have considered. To be able to handle larger protocols and a larger number of parallel sessions some optimizations will be necessary.

Improving origin-destination annotations. Mismatch between origin and destination of the content of an encryption term is a straight forward and simple way to investigate whether a protocol contains a breach. However, the mismatch in it self does not contain any information about the severeness of the breach. It is often the case that a mismatch is completely harmless and will only lead to a deadlock. Since a deadlock can always occur in a network controlled by the intruder, this should not be considered a breach in it self. Other methods of analyzing authentication protocols, described in the literature, uses other ways of characterizing a breach. It would be interesting to consider improvements to the origin-destination annotations.

7.2 Summary

I have developed a tool that makes it possible to integrate static analysis and model checking of authentication protocols. To perform the static analysis I use the LySa tool. To perform the model checking I have implemented methods inspired by state of the art model checking techniques. The two approaches are linked successfully together by the process algebra LySa^{N}.

The SAMC tool has been tested on a number of protocols from the Clark-Jacob library [11]. The test results are as expected.

The user interface of the SAMC tool is very simple and easy to use. It provides an easy way to experiment with various protocols and breaches. I think that the concept of choosing a breach based on the analysis result is ideal for experimenting with and understanding interesting aspects of authentication protocols.

This Master Thesis has in particular been interesting to work with due to the fact that it contains many different disciplines of computer science. I have worked with semantics of process algebras, static analysis techniques, model checking techniques and a lot of basic algorithms to tie it all together. I think I have been able to cover all disciplines adequately, but in a future project it would be interesting to go into a greater level of details with part of the work.

The initial ideas I had about integrating the analysis results in the model checking has not proven successful. I still believe that it is possible to somehow integrate the two approaches. One idea could be to try to include flow sensitivity in the analysis and try to use such analysis results to reduce the problem of termination in the model checking.

The current implementation of the SAMC tool will probably not be efficient enough for analyzing industrial-strength protocols but I hope that other students might be able to use the tool for experimenting with the interesting field of authentication protocols.

Appendices

Appendix A

Narration of test protocols

A.1 Small test cases

A.1.1 Mismatch of crypto-points

1. Mismatch of type (n, I)

$$\begin{array}{lllll} 1. & any & \rightarrow & A & : & K \\ 2. & A & \rightarrow & any & : & \{Msg\}_K \end{array}$$

2. Mismatch of type (I, n)

$$\begin{array}{lllll} 1. & A & \rightarrow & any & : & K \\ 2. & any & \rightarrow & A & : & \{Msg\}_K \end{array}$$

3. Mismatch of type (n_1, n_2)

$$\begin{array}{lllll} 1. & A & \rightarrow & B & : & \{M_S\}_{K_{AB}} \\ 2. & A & \rightarrow & B & : & \{M_P\}_{K_{AB}} \\ 3. & B & \rightarrow & any & : & M_P \end{array}$$

A.1.2 Decryption with complex keys

$$\begin{array}{lllll} 1. & any & \rightarrow & A & : & I \\ 2. & A & \rightarrow & any & : & \{Msg\}_{\mathsf{key}[I]} \end{array}$$

A.1.3 Assigning session indices

$$\begin{array}{lllll} 1. & any & \rightarrow & A & : & I \\ 2. & A & \rightarrow & any & : & \{Msg\}_{\mathsf{key}[I]} \end{array}$$

A.1.4 Using intruder crypto-point and name

$$\begin{array}{llllll} 1. & A & \rightarrow & any & : & \{Msg\}_{\mathsf{key}[I]} \end{array}$$

A.2 Test protocols

A.2.1 Wide mouthed frog, variant 1

(Running example protocol)

$$\begin{array}{llllll} 1. & A & \rightarrow & S & : & A, B, \{K\}_{\mathsf{key}[A]} \\ 2. & S & \rightarrow & B & : & A, \{K\}_{\mathsf{key}[B]} \\ 3. & A & \rightarrow & B & : & \{Msg\}_K \end{array}$$

A.2.2 Wide mouthed frog, variant 2

$$\begin{array}{llllll} 1. & A & \rightarrow & S & : & A, \{B, K\}_{\mathsf{key}[A]} \\ 2. & S & \rightarrow & B & : & \{A, K\}_{\mathsf{key}[B]} \\ 3. & A & \rightarrow & B & : & \{Msg\}_K \end{array}$$

A.2.3 Needham-Schroeder

$$\begin{array}{llllll} 1. & A & \rightarrow & S & : & A, B, N^A \\ 2. & S & \rightarrow & A & : & \{B, N^A, K, \{K, A\}_{\mathsf{key}[B]}\}_{\mathsf{key}[A]} \\ 3. & A & \rightarrow & B & : & \{K, A\}_{\mathsf{key}[B]} \\ 4. & B & \rightarrow & A & : & \{N^B\}_K \\ 5. & A & \rightarrow & B & : & \{N^B + 1\}_K \\ 6. & A & \rightarrow & B & : & \{Msg\}_K \end{array}$$

A.2.4 Andrew Secure RPC

$$\begin{array}{llllll} 1. & A & \rightarrow & B & : & A, \{u_1, N^A\}_K \\ 2. & B & \rightarrow & A & : & \{u_2, N^A + 1, N^B\}_K \\ 3. & A & \rightarrow & B & : & \{u_3, N^B + 1\}_K \\ 4. & B & \rightarrow & A & : & \{u_4, K', N^{B'}\}_K \\ 5. & A & \rightarrow & B & : & \{Msg\}_{K'} \end{array}$$

A.2.5 Yahalom

1.	A	$\rightarrow$	B	:	A, N^A
2.	B	$\rightarrow$	S	:	$B, \{A, N^A, N^B\}_{\mathsf{key}[B]}$
3.	S	$\rightarrow$	A	:	$\{B, N^A, K, N^B\}_{\mathsf{key}[A]}, \{A, K\}_{\mathsf{key}[B]}$
4.	A	$\rightarrow$	B	:	$\{A, K,\}_{\mathsf{key}[B]}, \{N^B\}_K$
5.	A	$\rightarrow$	B	:	$\{Msg\}_K$

Appendix B

Proof of Lemma 2

Lemma 2. $S_1 \equiv_N S_2 \Rightarrow \lfloor S_1 \rfloor \equiv \lfloor S_2 \rfloor \qquad \text{and} \qquad P_1 \equiv_N P_2 \Rightarrow \lfloor P_1 \rfloor \equiv \lfloor P_2 \rfloor$

Table B.1 on the next page contains the definition of the structural congruences for LySa$^{\mathrm{N}}$ and LySa.

The proof is by induction on the proof of $S_1 \equiv_N S_2$ and proceed by a case analysis on each of the axioms and rules in the definition of $\equiv_N$.

By applying the transformation to both sides of each of the axioms we can easily see that a corresponding axiom exists in the definition of $\equiv$. Example:

From definition of $\equiv_N$ we have $\mathsf{a3_N} : (S_1|S_2)|S_3 \equiv_N S_1|(S_2|S_3)$
Show: $\lfloor (S_1|S_2)|S_3 \rfloor \equiv \lfloor S_1|(S_2|S_3) \rfloor$

$\Leftrightarrow \quad (\lfloor S_1|S_2 \rfloor)|\lfloor S_3 \rfloor \equiv \lfloor S_1 \rfloor|(\lfloor S_2|S_3 \rfloor)$ *(using definition of $\lfloor\cdot\rfloor$)*
$\Leftrightarrow \quad (\lfloor S_1 \rfloor|\lfloor S_2 \rfloor)|\lfloor S_3 \rfloor \equiv \lfloor S_1 \rfloor|(\lfloor S_2 \rfloor|\lfloor S_3 \rfloor)$ *(using definition of $\lfloor\cdot\rfloor$)*

which follows directly from axiom $\mathsf{a3}$ in the definition of $\equiv$.

For each of the rules we use the inductive hypothesis as illustrated by an example below:

From definition of $\equiv_N$ we have $\mathsf{r5_N}$:	$S_1 \equiv_N S_2 \Rightarrow (\nu\, n)S_1 \equiv_N (\nu\, n)S_2$
Assume:	$S_1 \equiv_N S_2$
From the inductive hypothesis we have:	$\lfloor S_1 \rfloor \equiv \lfloor S_2 \rfloor$
Using $\mathsf{r5}$ we have:	$(\nu\, K_n)\lfloor S_1 \rfloor \equiv (\nu\, K_n)\lfloor S_2 \rfloor$
and by definition of $\lfloor\cdot\rfloor$:	$\lfloor (\nu\, n)S_1 \rfloor \equiv \lfloor (\nu\, n)S_2 \rfloor$

Hence we have shown that $(\nu\, n)S_1 \equiv_N (\nu\, n)S_2 \Rightarrow \lfloor (\nu\, n)S_1 \rfloor \equiv \lfloor (\nu\, n)S_2 \rfloor$

LySa$^{\text{N}}$, $\equiv_N$		
	$\mathsf{a1_N}$	$S \equiv_N S$
	$\mathsf{a1'_N}$	$P \equiv_N P$
	$\mathsf{a2_N}$	$S_1 \mid S_2 \equiv_N S_2 \mid S_1$
	$\mathsf{a3_N}$	$(S_1 \mid S_2) \mid S_3 \equiv_N S_1 \mid (S_2 \mid S_3)$
	$\mathsf{a4_N}$	$(\nu\, n_1)(\nu\, n_2)S \equiv_N (\nu\, n_2)(\nu\, n_1)S$
	$\mathsf{a4'_N}$	$(\nu\, n_1)(\nu\, n_2)P \equiv_N (\nu\, n_2)(\nu\, n_1)P$
	$\mathsf{r1_N}$	$S_1 \equiv_N S_2 \Rightarrow S_2 \equiv_N S_1$
	$\mathsf{r1'_N}$	$P_1 \equiv_N P_2 \Rightarrow P_2 \equiv_N P_1$
	$\mathsf{r2_N}$	$S_1 \equiv_N S_2 \wedge S_2 \equiv_N S_3 \Rightarrow S_1 \equiv_N S_3$
	$\mathsf{r2'_N}$	$P_1 \equiv_N P_2 \wedge P_2 \equiv_N P_3 \Rightarrow P_1 \equiv_N P_3$
	$\mathsf{r3_N}$	S_1 and S_2 are α-equivalent $\Rightarrow S_1 \equiv_N S_2$
	$\mathsf{r3'_N}$	P_1 and P_2 are α-equivalent $\Rightarrow P_1 \equiv_N P_2$
	$\mathsf{r4_N}$	$S_1 \equiv_N S_2 \Rightarrow S_1 \vert S_3 \equiv_N S_2 \vert S_3$
	$\mathsf{r5_N}$	$S_1 \equiv_N S_2 \Rightarrow (\nu\, n)S_1 \equiv_N (\nu\, n)S_2$
	$\mathsf{r5'_N}$	$P_1 \equiv_N P_2 \Rightarrow (\nu\, n)P_1 \equiv_N (\nu\, n)P_2$
	$\mathsf{r6_N}$	$n \notin \mathsf{fn}(S_2) \Rightarrow (\nu\, n)(S_1 \mid S_2) \equiv_N S_1 \mid (\nu\, n)S_2$
LySa, $\equiv$		
	a1	$P \equiv P$
	a2	$P_1 \mid P_2 \equiv P_2 \mid P_1$
	a3	$(P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3)$
	a4	$(\nu\, n_1)(\nu\, n_2)P \equiv (\nu\, n_2)(\nu\, n_1)P$
	a5	$(\nu\, n)\mathsf{0} \equiv \mathsf{0}$
	a6	$!\,P \equiv P \mid !\,P$
	r1	$P_1 \equiv P_2 \Rightarrow P_2 \equiv P_1$
	r2	$P_1 \equiv P_2 \wedge P_2 \equiv P_3 \Rightarrow P_1 \equiv P_3$
	r3	P_1 and P_2 are α-equivalent $\Rightarrow P_1 \equiv P_2$
	r4	$P_1 \equiv P_2 \Rightarrow P_1 \vert P_3 \equiv P_2 \vert P_3$
	r5	$P_1 \equiv P_2 \Rightarrow (\nu\, n)P_1 \equiv (\nu\, n)P_2$
	r6	$n \notin \mathsf{fn}(P_2) \Rightarrow (\nu\, n)(P_1 \mid P_2) \equiv P_1 \mid (\nu\, n)P_2$

Table B.1: Congruence of processes for LySa$^{\text{N}}$ and LySa.

Appendix C

Installing the SAMC tool

All SML/NJ source files necessary to run the tool are collected in a zip-file which can be obtained from `http://www.student.dtu.dk/~s971701/masterthesis`. The zip-file contains the LySa tool and the SAMC tool. The files should be unpacked in a way that preserves the directory structure. SML/NJ can be obtained from `http://www.smlnj.org/`. The tool is run from an SML command prompt with the command

```
use "ui.sml";
```

Note that SML must be started in the directory `/samc/test`.

Section 5 contains a users guide to the SAMC tool.

Appendix D

Listing of protocols.sml

```
structure Prot =
struct

open Types
open LysaN

val A = NAME("A")
val B = NAME("B")
val S = NAME("S")
val I = NAME("I")
val Z = NAME("Z")

val K = NAME("K")
val Msg = NAME("Msg")

val A1 = NAME("A1")
val A2 = NAME("A2")
val A3 = NAME("A3")
val A4 = NAME("A4")

val B1 = NAME("B1")
val B2 = NAME("B2")
val B3 = NAME("B3")
val B4 = NAME("B4")

val u1 = NAME("u1")
val u2 = NAME("u2")
val u3 = NAME("u3")
val u4 = NAME("u4")

val xA = VAR("xA")
val xA' = DEFVARP("xA")
val xB = VAR("xB")
val xB' = DEFVARP("xB")
val yA = VAR("yA")
val yA' = DEFVARP("yA")
val yB = VAR("yB")
val yB' = DEFVARP("yB")
val zA = VAR("zA")
val zA' = DEFVARP("zA")
val zB = VAR("zB")
```

```
val zB' = DEFVARP("zB")
val xK = VAR("xK")
val xK' = DEFVAR("xK")
val yK = VAR("yK")
val yK' = DEFVAR("yK")
val zK = VAR("zK")
val zK' = DEFVAR("zK")
val yMsg = VAR("yMsg")
val yMsg' = DEFVAR("yMsg")

val xNb' = DEFVAR("xNb")
val xNb = VAR("xNb")
val yNa = VAR("yNa")
val yNa' = DEFVAR("yNa")
val zNb' = DEFVAR("zNb")
val zNb = VAR("zNb")
val zNa = VAR("zNa")
val zNa' = DEFVAR("zNa")

val xKNEW' = DEFVAR("xKNEW")
val xNbNEW' = DEFVAR("xNbNEW")
val xKNEW = VAR("xKNEW")
val xNbNEW = VAR("xNbNEW")

val succ = NAME("succ")
val oK = NAME("oK")
val oNb = NAME("oNb")
val oNa = NAME("oNa")
val oMsg = NAME("oMsg")
val Q = NAME("Q")

val KNEW = NAME("KNEW")
val NbNEW = NAME("NbNEW")

val Na = NAME("Na")
val Nb = NAME("Nb")

val xEnc = VAR("xEnc")
val xEnc' = DEFVAR("xEnc")

val y = VAR("y")
val y' = DEFVAR("y")
val MS = NAME "MS"
val MP = NAME "MP"
val xS = VAR "xS"
val xS' = DEFVAR "xS"
val xP = VAR "xP"
val xP' = DEFVAR "xP"
val Kab = NAME "Kab"

(*------------------------*)


(*origdest*)

(*
val S = NAME("S")
val P = NAME("P")
val K = NAME("K")

val xS = VAR("xS")
val xS' = DEFVAR("xS")
```

```
val xP = VAR("xP")
val xP' = DEFVAR("xP")

val a1 = NAME("a1")
val a2 = NAME("a2")
val b1 = NAME("b1")
val b2 = NAME("b2")

val Pa = NEW(S,
         OUT([ENC([S],K,(a1,DEST b1))],
         NEW(P,
         OUT([ENC([P],K,(a2,DEST b2))],
         NIL))))

val Pb = INP([ENC([xS'],K,(b1,ORIG a1))],
         INP([ENC([xP'],K,(b2,ORIG a2))],
         OUT([xP],
         NIL)))

val origdest = NEWS(K,PAR([LOOP("A",Pa),LOOP("B",Pb)]))
*)
(*------------------------*)


(*WMF variant 1. Running example protocol*)

val Pa = NEW(K,
         OUT([A,B,ENC([K],KEY(A),(A,DEST S))],
         NEW(Msg,
         OUT([ENC([Msg],K,(A,DEST B))],
         NIL))))

val Pb = INP([yA',ENC([yK'],KEY(B),(B,ORIG S))],
         INP([ENC([yMsg'],yK,(B,ORIG yA))],
         NIL))

val Ps = INP([zA',zB',ENC([zK'],KEY(zA),(S,ORIG zA))],
         OUT([zA,ENC([zK],KEY(zB),(S,DEST zB))],
         NIL))

val wmfv1 =
  NEWS(KEY(A),NEWS(KEY(B),
  PAR([LOOP("A",Pa),LOOP("B",Pb),LOOP("S",Ps)])))

(* WMF variant 2. Principal names encrypted *)


val Pa = (*INP([xA',xB'],*)
         NEW(K,
         OUT([A,ENC([B,K],KEY(A),(A,DEST S))],
         NEW(Msg,
         OUT([ENC([Msg],K,(A,DEST B))],
         NIL))))

val Pb = (*INP([yA',yB'],*)
         INP([ENC([yA',yK'],KEY(B),(B,ORIG S))],
         INP([ENC([yMsg'],yK,(B,ORIG yA))],
         NIL))

val Ps = INP([zA',ENC([zB',zK'],KEY(zA),(S,ORIG zA))],
         OUT([ENC([zA,zK],KEY(zB),(S,DEST zB))],
         NIL))
```

```
val leak = OUT([
oK,
ENC([B,oK],KEY(A),(Z,DEST Z)),
ENC([A,oK],KEY(B),(Z,DEST Z)),
ENC([oMsg],oK,(Z,DEST Z))
],NIL)

val wmfv2 =
  NEWS(KEY(A),NEWS(KEY(B),
  PAR([LOOP("A",Pa),LOOP("B",Pb),LOOP("S",Ps)])))

val wmfv2Leak =
  NEWS(KEY(A),NEWS(KEY(B),
  PAR([LOOP("A",Pa),
  LOOP("B",Pb),
  LOOP("S",Ps),
  LOOP("leak",leak)])))


(*----  needham schroeder ------*)

val Pleak = NEW(oK,NEW(oNa,NEW(oNb,NEW(oMsg,NEW(succ,
OUT([oK,oNa,oNb,succ,
ENC([A,oK],KEY(B),(Z,DEST Z)),
ENC([B,oNa,oK,ENC([A,oK],KEY(B),(Z,DEST Z))],KEY(A),(Z,DEST Z)),
ENC([Q,oMsg],oK,(Z,DEST Z)),
ENC([oNb],oK,(Z,DEST Z)),
ENC([ENC([oNb],succ,(Z,DEST Z))],oK,(Z,DEST Z))],
NIL))))))

val Pa = NEW(Na,NEW(Msg,
         OUT([A,B,Na],
         INP([ENC([B,Na,xK',xEnc'],KEY(A),(A1,ORIG S))],
         OUT([xEnc],
         INP([ENC([xNb'],xK,(A2,ORIG B2))],
         OUT([ENC([ENC([xNb],succ,(Z,DEST Z))],xK,(A3,DEST B3))],
         OUT([ENC([Q,Msg],xK,(A4,DEST B4))],
         NIL))))))))

val Pb = NEW(Nb,
         INP([ENC([A,yK'],KEY(B),(B1,ORIG S))],
         OUT([ENC([Nb],yK,(B2,DEST A2))],
         INP([ENC([ENC([Nb],succ,(Z,ORIG Z))],yK,(B3,ORIG A3))],
         INP([ENC([Q,yMsg'],yK,(B4,ORIG A4))],
         NIL)))))

val Ps = NEW(K,
         INP([A,B,zNa'],
         OUT([ENC([B,zNa,K,ENC([A,K],KEY(B),(S,DEST B1))],
                KEY(A),(S,DEST A1))],
         NIL)))

val nsLeak = NEWS(KEY(A),NEWS(KEY(B),
             PAR([LOOP("leak",Pleak),
                  LOOP("A",Pa),
                  LOOP("B",Pb),
                  LOOP("S",Ps)])))


(*------andrew secure RPC--------*)
```

```
val Pa = NEW(Na,
        OUT([A,ENC([u1,Na],K,(A1,DEST B1))],
          INP([ENC([u2,ENC([Na],succ,(Z,ORIG Z)),xNb'],
                 K,(A2,ORIG B2))],
          OUT([ENC([u3,ENC([xNb],succ,(Z,DEST Z))],
                 K,(A3,DEST B3))],
          INP([ENC([u4,xKNEW',xNbNEW'],K,(A4,ORIG B4))],
          NEW(Msg,
          OUT([ENC([Msg],xKNEW,(A, DEST B))],
          NIL)))))))

val Pb = INP([A,ENC([u1,yNa'],K,(B1,ORIG A1))],
          NEW(Nb,
          OUT([ENC([u2,ENC([yNa],succ,(Z, DEST Z)),Nb],
                 K,(B2,DEST A2))],
          INP([ENC([u3,ENC([Nb],succ,(Z,ORIG Z))],
                 K,(B3,ORIG A3))],
          NEW(KNEW,NEW(NbNEW,
          OUT([ENC([u4,KNEW,NbNEW],K,(B4,DEST A4))],
          INP([ENC([yMsg'],KNEW,(B, ORIG A))],
          NIL))))))))

val leak = OUT([oK,ENC([u4,oK,oNb],K,(Z,DEST Z))],NIL)

val andrew =
  NEWS(K,PAR([LOOP("A",Pa),LOOP("B",Pb)]))
val andrewLeak =
  NEWS(K,PAR([LOOP("A",Pa),LOOP("B",Pb),LOOP("leak",leak)]))

(*------ Yahalom -----------------*)

val Pa  = NEW(Na,
          OUT([A,Na],
          INP([ENC([B,Na,xK',xNb'],KEY(A),(A,ORIG S)),xEnc'],
          OUT([xEnc,ENC([xNb],xK,(A,DEST B))],
          NEW(Msg,
          OUT([ENC([Msg],xK,(A,DEST B))],
          NIL))))))

val Pb  = INP([yA',yNa'],
          NEW(Nb,
          OUT([B,ENC([yA,yNa,Nb],KEY(B),(B, DEST S))],
          INP([ENC([yA,yK'],KEY(B),(B,ORIG S)),
               ENC([Nb],yK,(B,ORIG yA))],
          INP([ENC([yMsg'],yK,(B,ORIG yA))],
          NIL)))))

val Ps  = INP([zB',
               ENC([zA',zNa',zNb'],KEY(zB),(S, ORIG zB))],
          NEW(K,
          OUT([ENC([zB,zNa,K,zNb],KEY(zA),(S,DEST zA)),
               ENC([zA,K],KEY(zB),(S,DEST zB))],
          NIL)))

val leak = OUT([
oK,oNa,oNb,
ENC([oNb],oK,(Z,DEST Z)),
ENC([oMsg],oK,(Z,DEST Z)),
ENC([A,oNa,oNb],KEY(B),(Z, DEST Z)),
ENC([B,oNa,oK,oNb],KEY(A),(Z,DEST Z)),
ENC([A,oK],KEY(B),(Z,DEST Z))
],NIL)
```

```
val yahalom =
  NEWS(KEY(A),NEWS(KEY(B),
  PAR([LOOP("A",Pa),
       LOOP("B",Pb),
       LOOP("S",Ps),
       LOOP("leak",leak)])))


(* ------test cases ------------*)

(*annotation 1*)
val P = INP([xK'],
        NEW(Msg,
        OUT([ENC([Msg],xK,(A,DEST B))],
        NIL)))
val SAnnot1 = LOOP("P",P)

(*annotation 2*)
val P = NEW(K,
        OUT([K],
        INP([ENC([yMsg'],K,(B,ORIG A))],
        NIL)))

val SAnnot2 = LOOP("P",P)

(*annotation 3*)

val Pa = NEW(MS,NEW(MP,
         OUT([ENC([MS],Kab,(A1,DEST B1))],
         OUT([ENC([MP],Kab,(A2,DEST B2))],
         NIL))))

val Pb = INP([ENC([xS'],Kab,(B1,ORIG A1))],
         INP([ENC([xP'],Kab,(B2,ORIG A2))],
         OUT([xP],
         NIL)))

val SAnnot3 = NEWS(Kab,PAR([LOOP("A",Pa),LOOP("B",Pb)]))

(*complex key*)
val P = INP([xA'],
        NEW(Msg,OUT([ENC([Msg],KEY(xA),(S,DEST A))],
        NIL)))

val SComplexKey = LOOP("S",P)

(*Intruder key and crypto-point*)
val P = NEW(Msg,OUT([ENC([Msg],KEY(I),(S,DEST I))],NIL))
val SIKey = LOOP("S",P)


(*-------------------------------------------------------*)
val protocolList = [
("Test case Annot 1",SAnnot1),
("Test case Annot 2",SAnnot2),
("Test case Annot 3",SAnnot3),
("Test case Complex key",SComplexKey),
("Test case Intruder key and crypto-point",SIKey),
("WMF, variant 1, (Running example protocol",wmfv1),
("WMF, variant 2, principal names encrypted",wmfv2),
("WMF, variant 2, principal names encrypted, leak",wmfv2Leak),
```

```
("needham schroeder with leak",nsLeak),
("andrew secure RPC ",andrew),
("andrew secure RPC with leak",andrewLeak),
("yahalom",yahalom)
]

end
```

Bibliography

[1] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, H. Riis Nielson: Static Validation of Security Protocols, Dep. of Informatics and Mathematical Modeling, Technical University of Denmark (2003)

[2] M. Buchholtz, F. Nielson, H. Riis Nielson: A Calculus for Control Flow Analysis of Security Protocols, Dep. of Informatics and Mathematical Modeling, Technical University of Denmark

[3] M. Buchholtz: Deliverable 19 - Static Analyzers, DEGAS (2003)

[4] M. Buchholtz: Implementing Control Flow Analysis, Dep. of Informatics and Mathematical Modeling, Technical University of Denmark (2003)

[5] N. Kaplan: Simulation of Authentication Protocols, Dep. of Informatics and Mathematical Modeling, Technical University of Denmark (2003)

[6] Web resources for Standard ML of New Jersey: `http://www.smlnj.org/`

[7] H. Sun: User's Guide for the Succinct Solver (V2.0), Dep. of Informatics and Mathematical Modeling, Technical University of Denmark (2003)

[8] D. Basin, S. Mödersheim, L. Viganò: An On-The-Fly Model-Checker for security Protocol Analysis. Technical Report 404, Dep. of Computer Science, ETH Zurich (2003)

[9] D. Basin, S. Mödersheim, L. Viganò: Constraint Differentiation: A New Reduction Technique for Constraint-Based Analysis of Security Protocols, Technical Report 405, Dep. of Computer Science, ETH Zurich (2003)

[10] A. Armando, D. Basin, M. Bouallagui, Y. Chevalier, L. Compagna, S. Mödersheim, M. Rusinowitch, M. Turuani, L. Viganò, L. Vigneron: The AVISS Security Protocol Analysis Tool, In E. Brinksma and K.G. Larsen, editors, Proceedings of CAV'02, LNCS 2404, pages 349–353. Springer-Verlag, 2002

[11] J. Clark, J. Jacob: A Survey of Authentication Protocols (1997)

[12] D. Engler, M. Musuvathi: Static analysis versus software model checking for finding bugs, Computer Systems Laboratory, Stanford

[13] G. Denker, J. Millen: CAPSL Intermediate Language. In Formal Methods and Security Protocols, 1999. FLOC'99 Workshop.

[14] B. Schneider: Applied Cryptography, 2nd edition (1996)

[15] I. Cervesato, N.A. Durgin, P.D. Lincoln, J.C. Mitchell A. Scedrov: A Meta-notation for Protocol Analysis, Computer Science Lab, SRI International. Computer Science Department, Stanford University. Mathematics Department, University of Pennsylvania

[16] N.A. Durgin, P.D. Lincoln, J.C. Mitchell, A. Scedrov: Undecidability of bounded security protocols, Computer Science Lab, SRI International. Computer Science Department, Stanford University. Mathematics Department, University of Pennsylvania