On the Automated Correction of Faulty Security Protocols

Ph.D in Computer Science
Thesis by

JUAN CARLOS LÓPEZ PIMENTEL

Adviser: DR. RAÚL MONROY BORJA
Co-adviser: DR. DIETER HUTTER
Thesis Committee: DR. GUILLERMO MORALES LUNA
DR. LUIS ANGEL TREJO RODRIGUEZ
DR. ALBERTO OLIART ROS

Atizapán de Zaragoza, Edo. Mex., April of 2008
Resumen

Un Protocolo de Seguridad se usa para establecer una comunicación segura entre dos o más agentes a pesar de condiciones adversas. Crímenes tales como la usurpación de identidades o accesos no autorizados a la información han provocado incontables pérdidas económicas especialmente cuando se realizan transacciones electrónicas. Por ese motivo, la comunidad de los métodos formales ha puesto un interés especial en la verificación de protocolos de seguridad, produciéndose en los últimos años una gran cantidad de técnicas y herramientas.

Otros autores han tomado un enfoque diferente, proponiendo principios de diseño para ayudar al desarrollo de mejores protocolos. Esta iniciativa se origina porque ellos han observado que los ataques de protocolos de seguridad son un resultado de malas prácticas de diseño. Algunos de los más importantes principios de diseño han sido publicados por Abadi y Needham. A pesar de ésto, el diseño de protocolos de seguridad es particularmente difícil y propenso a errores.

En esta tesis presentamos SHRIMP (a Smart metHod for Repairing IMperfect security Protocols). SHRIMP tiene como objetivo acelerar el ciclo de desarrollo de protocolos de seguridad, uniendo la brecha que hay entre diseño y análisis por medio de una tarea de diagnóstico y reparación. SHRIMP trabaja con herramientas de verificación existentes para el proceso de búsqueda de ataques en el protocolo (en particular hemos usado AVISPA ya que es considerado el estado del arte). SHRIMP primero analiza un protocolo y, si el protocolo tiene fallas, busca una o más corridas del protocolo que violan un requerimiento de seguridad, llamado ataque. Así, SHRIMP analiza el protocolo y el ataque para indicar qué pasos fallan en el protocolo y de esta forma sintetizar cambios apropiados para corregirlos. SHRIMP produce una versión mejorada del protocolo, el cuál es analizado y parchado nuevamente hasta que no sean detectados más errores.

Para organizar la aplicación de diagnóstico y reparación, hemos adoptado la metodología proof planning de Bundy. Así, SHRIMP es un conjunto de métodos de parche, cada uno de ellos es capaz de parchar una clase general de fallas. Para arreglar una falla, en cada método de parche hemos traducido algunos de los principios de Abadi y Needham en requerimientos formales, la cual llamamos precondiciones. Estas precondiciones son una colección de reglas que identifican la clase de ataque que el protocolo sufre y a partir de las cuales se propone un parche.

En esta tesis nos hemos enfocado en parchar protocolos vulnerables a los ataques de tipo replay. SHRIMP toma cuidado de este tipo de ataques cuando el mensaje siendo re-enviado es un texto cifrado. Si éste es el caso, SHRIMP incluye tres principios métodos de parche para enfrentar este tipo de ataques: message_encoding, agent_naming y session_binding. Hasta ahora hemos probado SHRIMP en 36 protocolos, 21 de ellos fueron obtenidos de la librería Clark-Jacob, obteniendo una tasa de reparación del 90%.
Abstract

A security protocol is usually used to achieve a secure network communication between two or more agents despite adversal conditions. Crimes such as user impersonation or unauthorized access to information have already resulted in countless losses especially when different entities have executed electronic transactions. Thereby, security protocol verification has attracted a lot of interest in the formal methods community yielding an abundance of tools and techniques in the last few years.

Other authors have taken a different approach, proposing design principles in order to help in the construction of better security protocols. The rational behind this initiative originates from the observations that successful attacks against security protocols are a result of bad practices in security protocol design. The most important design principles have been proposed by Abadi and Needham. Yet, security protocol design is particularly difficult and error-prone.

We introduce SHRIMP, a Smart method for Repairing IMPerfect security Protocols. SHRIMP aims at speeding up the formal software development cycle, bridging the gap between design and analysis by means of diagnosis and repair. SHRIMP relies on existing state-of-the-art tools (we particularly used AVISPA) both to analyse a protocol and, if the protocol is flawed, to find one or more of protocol runs violating a given security requirement, called an attack. SHRIMP then analyses the protocol and the attack to pinpoint the faulty steps of the protocol and synthesises appropriate changes to fix them. This yields an improved version of the protocol that should be analysed and potentially patched again until no further flaws can be detected.

As a general framework to organize the application of diagnostic and repair task, we have adopted Bundy’s proof planning methodology. So, SHRIMP is a set of patch methods, each of which is able to patch a general class of faults. To patch a protocol flaw, in each patch method we have translated some of Abadi and Needham’s informal principles into formal requirements, which we call preconditions. These preconditions are a collection of rules that identify the class of attack the protocol is suffering and from there a patch is suggested. However, the patches are not independent and the application of a rule requires preconditions to be applicable and should guarantee postconditions once it has been applied.

We have hitherto focused on automatically fixing protocols subject to a replay attack. SHRIMP takes care of replay attacks where the message being reused is a cypher-text and then it includes three main methods: message_encoding, agent_naming and session_binding. We have successfully tested SHRIMP on 36 protocols, 21 out of which were borrowed from the Clark and Jacob library, obtaining a repair rate of 90%.
Acknowledgments

First and foremost I would like to thank my adviser, Dr. Raúl Monroy Borja for his patient and expert guidance throughout the course of the research described in this thesis. His support, advice and encouragement has been invaluable. I would also like to thank my second supervisor, Dr. Dieter Hutter with his thorough reading of my first draft contributed greatly to the quality of the final thesis.

I am grateful to my examiners, Dr. Luis Angel Trejo Rodríguez, Dr. Guillermo Morales Luna and Dr. Alberto Oliart Ros, for their rigorous criticism and considered suggestions for improvements to the thesis.

I would like to thank my family for their support, particularly my mother by her economical and motivational help. She always believed in me and she always was a source of inspiration. Similarly to my father† who, from the heaven, has known to guide me to reach my objectives. I would like to thank my wife, Isela, by their great support during this large period of time. Without her help and understanding, I certainly would not have been able to finish this thesis.

I would also like to express my appreciation to my PhD students friends in the ITESM-CEM by their good company. I also gratefully acknowledge the financial support received by ITESM-CEM and CONACyT. Finally, I would also like to express my appreciation to Dr. Miguel González who being Postgraduate Director in Computer Science at ITESM-CEM help me in a lot of administrative procedures.
Declaration

I declare that this thesis has been composed by myself and that the work described in it is my own. Portions of the work described here have been published previously in [56] and [57].

Juan Carlos López Pimentel
Dedication

I want to dedicate this thesis to my precious wife, Isela. With her help and encouragement this thesis has been possible.

\[ JC \rightarrow BB : \{\text{Thanks}; BB; For; All\}_{JCBB} \]
# Table of Content

1 **Introduction**  
1.1 Current Support for Security Protocol Development ........................................ 1  
1.1.1 Hypothesis ...................................................................................................... 2  
1.1.2 Assumptions .................................................................................................... 3  
1.2 Overview ............................................................................................................. 3  
1.3 Contributions ..................................................................................................... 5  
1.4 Outline ............................................................................................................... 6  
1.5 Notation .............................................................................................................. 7

2 **Support for Security Protocol Development** .................................................. 8  
2.1 The Need for Support in Security Protocol Development ................................. 8  
2.2 Current Support for Security Protocol Development ........................................ 10  
2.2.1 The formal methods school ............................................................................. 10  
2.2.2 The computational complexity school ............................................................ 11  
2.2.3 Connections between both views .................................................................... 12  
2.3 Security Protocol Verification Using Formal Methods .................................... 12  
2.3.1 Formal Methods for Security Protocol Development .................................... 12  
2.3.2 Belief logic ..................................................................................................... 14  
2.3.3 State Exploration ............................................................................................ 14  
2.3.3.1 Early Attempts at Protocol Verification ...................................................... 15  
2.3.3.2 Model checking .......................................................................................... 15  
2.3.3.3 Strand Spaces approach .......................................................................... 17  
2.3.4 Theorem proving based approaches ................................................................ 19  
2.3.4.1 Methods Based on High Order Logic ......................................................... 19  
2.3.4.2 Methods Based on First Order Logic .......................................................... 20  
2.4 Fixing Faulty Security Protocols ........................................................................ 22  
2.4.1 A Faulty Security Protocol ............................................................................. 22  
2.4.2 Illustrating the Problem of Patching Faulty Security Protocols .................... 24  
2.4.2.1 Finding the flaw .......................................................................................... 24  
2.4.2.2 Fixing the protocol ..................................................................................... 25  
2.5 Conclusions ....................................................................................................... 26

3 **Principles to Guide the Design of Security Protocols** .................................... 28  
3.1 Informal Design Guidelines in the Literature ................................................. 29  
3.2 Abadi and Needham’s Principles to Guide the Design of Security Protocols ...... 29  
3.2.1 Trust relations ................................................................................................. 30  
3.2.2 On the Role of a Message .............................................................................. 30  
3.3 Encryption .......................................................................................................... 31  
3.3.1 Confidentiality ................................................................................................ 32  
3.3.2 Authentication ............................................................................................... 33  
3.4 Agent names ....................................................................................................... 34  
3.4.1 Name deduction in asymmetric cryptography .............................................. 35
3.4.2 Name deduction in symmetric cryptography ........................................ 35
3.5 Freshness ......................................................................................... 36
  3.5.1 Nonces ....................................................................................... 36
  3.5.2 Predictable numbers .................................................................... 37
  3.5.3 Timestamps ................................................................................ 38
3.6 Session keys ................................................................................... 39
3.7 Recognizing messages ....................................................................... 40
3.8 Conclusions ...................................................................................... 42

4 SHRIMP: Outline ............................................................................. 43
  4.1 A Large Class of Protocol Flaw ......................................................... 43
    4.1.1 Replay Attacks .......................................................................... 44
  4.2 SHRIMP for Security Protocol Development .................................... 45
  4.3 Patch Planning Faulty Security Protocols ........................................ 46
    4.3.1 Proof Planning Methodology for Patch Planning ......................... 46
    4.3.2 Patch Method Application ....................................................... 48
    4.3.3 Criteria to Evaluate Patch Methods ......................................... 49
    4.3.4 The Logic of the Patching Methods ......................................... 49
  4.4 Strand Spaces: The logic behind Shrimp ......................................... 49
    4.4.1 Message terms .......................................................................... 50
    4.4.2 Events, Strands and Strand Spaces ......................................... 50
    4.4.3 Bundles .................................................................................... 52
    4.4.4 Authentication Test ................................................................... 53
  4.5 Conclusions ..................................................................................... 54

5 SHRIMP’s Meta-logic ......................................................................... 56
  5.1 Operators to express Pre- and Post-conditions .................................. 56
    5.1.1 Message Similarity .................................................................... 57
    5.1.2 Explicit agent names in messages ............................................. 59
    5.1.3 Patch Production ...................................................................... 60
  5.2 Protocol Description ....................................................................... 61
    5.2.1 Variables, Constants and Types .............................................. 62
    5.2.2 Message Representation ......................................................... 62
    5.2.3 Protocol Specification .............................................................. 63
  5.3 Management of Change in a Protocol Description ............................. 64
    5.3.1 Substitution, Instantiation and Unification ................................ 65
    5.3.2 Protocol and Counterexample Definitions ................................. 65
    5.3.3 Change Propagations in a Symbolic Bundle .............................. 66
      5.3.3.1 Term positions ................................................................... 66
      5.3.3.2 Change operators for Messages ......................................... 67
      5.3.3.3 Change Operators for a Symbolic Bundle ......................... 68
    5.3.4 Changes in a Protocol Description ............................................ 69
  5.4 Conclusions .................................................................................... 72
6 SHRIMP: The Patching Framework  
6.1 The Replay Compound Method ........................................ 74  
6.2 Patching Protocols Violating Principle 10  
   6.2.1 The WMF Protocol Example ..................................... 76  
   6.2.2 The BAN-Yahalom Protocol Example ............................ 76  
   6.2.3 The message_encoding Method .................................. 77  
6.3 Patching Protocols Violating Principle 3  
   6.3.1 The NSPK Protocol Example .................................. 79  
   6.3.2 The BAN-ASRPC Protocol Example .......................... 80  
   6.3.3 The agent_naming Method ...................................... 81  
6.4 Patching Protocols Violating Principles 6—10  
   6.4.1 The Revisited WMF Protocol Example ......................... 83  
   6.4.2 The Denning-Sacco Shared Key (DSSK) Protocol Example ... 84  
   6.4.3 The session_binding Method ................................. 84  
      6.4.3.1 Nonce_flow .................................................. 85  
      6.4.3.2 Handshake .................................................. 87  
6.5 Conclusions .............................................................. 89  

7 Implementation and Results  90  
7.1 Implementation .......................................................... 90  
   7.1.1 AVISPA: The Verification Tool ................................ 90  
   7.1.2 SHRIMP Implementation ..................................... 91  
   7.1.3 Development Methodology ................................... 92  
7.2 The Protocol Set and Results ....................................... 93  
   7.2.1 The Protocol Set ............................................... 93  
   7.2.2 Results .......................................................... 95  
7.3 Conclusions .............................................................. 102  

8 Comparison and Directions for Future Work 104  
8.1 Comparison to Related Work ....................................... 104  
   8.1.1 On how to deal with Replay Attacks ......................... 104  
   8.1.2 On the Automated Design in Security Protocols .......... 105  
8.2 Three Protocol Examples to illustrate directions for Future Work 106  
   8.2.1 The IKEv2-DS protocol ....................................... 107  
      8.2.1.1 The IKEv2 Protocol Specification ....................... 107  
      8.2.1.2 Patching IKEv2 with SHRIMP .......................... 108  
   8.2.2 The Asokan and Ginzboorg Protocol ......................... 110  
      8.2.2.1 Specification of the Asokan and Ginzboorg Protocol . 110  
      8.2.2.2 Patching the Asokan and Ginzboorg Protocol with SHRIMP 111  
   8.2.3 Otway and Rees Protocol .................................... 112  
8.3 Future Work ............................................................. 113  
   8.3.1 On the Cryptographic Primitives ............................ 113  
   8.3.2 On the Global Requirements ................................ 114  
   8.3.3 On the Number of Participants .............................. 115  
   8.3.4 On Type Flaw Attacks ......................................... 116  
   8.3.5 On the linking to a Verification Tool ...................... 116  
8.4 Conclusions .............................................................. 117
9 Conclusions
  9.1 Final Summary ......................................................... 120

A AVISPA Tool
  A.1 An Overview of HLPSL Specifications .............................. 132
    A.1.1 Types and Messages ........................................... 133
    A.1.2 Basic Roles .................................................. 133
    A.1.3 Composed Roles .............................................. 135
    A.1.4 Security Goals .............................................. 136
  A.2 Verifying a protocol with the HLPSL specification ............ 136
    A.2.1 Output format .............................................. 138
  A.3 Authentication: AVISPA Hierarchy vs Lowe Hierarchy ........... 140
  A.4 Conclusions ....................................................... 141

B SHRIMP Implementation
  B.1 Software tools used in our meta-logic ......................... 142
    B.1.1 How to use SHRIMP-Implementation Input ................... 143
    B.1.2 How to use SHRIMP-Implementation ......................... 144
  B.2 On Technical Aspects of SHRIMP-Implementation ............... 146
    B.2.1 Representation of a protocol ............................. 146
    B.2.2 Communication events ..................................... 147
    B.2.3 Agents and Messages ....................................... 147
    B.2.4 Representation of a counterexample ....................... 149
  B.3 Illustrating the Full Verification Cycle for Security Protocols .... 149
    B.3.1 First Phase ............................................. 150
    B.3.2 Second Phase ............................................. 152
    B.3.3 Third Phase ............................................. 154
## List of Figures

1.1 An example message exchange ........................................... 7
2.1 Counterexample of the Otway and Rees protocol ....................... 23
2.2 Alternatives of Patching Otway and Rees Protocol ..................... 25
3.1 Counterexample to the WMF protocol .................................. 42
4.1 SHRIMP in the Security Protocols Development Process ............... 46
4.2 The skeleton of a patching method .................................... 47
4.3 The skeleton of a compound method ................................... 48
4.4 Strands of NSPK protocol ............................................. 51
4.5 A Regular Bundle of the NSPK Protocol ............................... 52
4.6 A Penetrator Bundle on the NSPK Protocol ............................. 53
5.1 The NSPK protocol description ....................................... 64
5.2 A message tree $A; |\{A; N_a\}_K$ with explicit $\pi$ positions ........... 67
5.3 A Regular Bundle of the Otway and Rees Protocol .................... 69
5.4 A Modified Regular Bundle of the Otway and Rees Protocol .......... 70
5.5 The Otway and Rees Protocol Description ............................. 71
6.1 The replay compound method ......................................... 75
6.2 The message_encoding method ....................................... 78
6.3 The agent_naming method ........................................... 82
6.4 The session_binding method ......................................... 85
A.1 HLPSL of the NSPK protocol ...................................... 137
A.2 The AVISPA tool’s output: Attack on the NSPK protocol ............. 139
B.1 SHRIMP in the Development Cycle for Security Protocols .......... 143
B.2 Syntax to write abn files ........................................... 144
List of Tables

4.1 Paul Syverson’s taxonomy of replay attacks ............................. 44
5.1 Types available in SHRIMP .................................................. 62
7.1 Protocol Examples Used for Development ................................. 93
7.2 Protocols Examples Used for Development and Testing .................. 94
7.3 Experimental results ............................................................ 96
7.4 Patched protocol descriptions, first part .................................... 97
7.5 Patched protocol descriptions, second part ................................. 98
7.6 Patched protocol descriptions, third part ................................... 99
7.7 Patched protocol descriptions, fourth part ................................ 100
7.8 Patched protocol descriptions, fifth part .................................. 101
7.9 The total elapsed verification time ........................................... 103
8.1 Security properties of the IKEv2 before and after the patch .......... 109
8.2 Security properties of the IKEv2 before and after applying handshake. 110
1. Introduction

1.1 Current Support for Security Protocol Development

Computer security is presently a strong requirement in everyday life. Crimes such as user impersonation or unauthorized access to information have already resulted in countless losses especially when different entities have executed electronic transactions. A security protocol is usually used to deal with these issues. A security protocol is a set of rules and conventions whereby one or more agents agree about each others’ identity, usually ending up in the possession of one or more secrets [74].

The goal of security protocols is to exchange sensitive data on insecure networks such as the Internet. Data must be protected from hackers on the network by using cryptographic primitives such as symmetric and asymmetric encryption, hash functions, and digital signatures. In some cases further devices like timestamps are also used to achieve security goals such as confidentiality, authentication, integrity and non-repudiation.

Although strong cryptographic algorithms exist, they do not guarantee the security of a communication system. In addition, if a security protocol is not designed carefully, it may contain flaws, which can be the ideal starting point for yielding an attack. Such flaws can be subtle and hard to find. A number of faulty protocols exists in the literature whose flaws were not found for a long time and that have received intensive analysis. One of these examples is the well known Needham and Schroeder public key protocol published in 1978. Even though it is a very simple protocol, a flaw was found 17 years after its publication.
Thus, experience has shown that the design of security protocols is particularly difficult and error-prone. Because of that, the verification of security protocols has attracted a lot of interest in the formal methods community yielding an abundance of tools and techniques in the last few years. Some methods are based on belief logics such as BAN [24] and GNY [21]; another methods are state exploration, like NRL [59], FDR [66], AVISPA [11]; another methods use the strand spaces model like [86, 80]; and finally, another methods are based on theorem proving, such as the Paulson’s Inductive Method [74], Coral [82], etc. Of all, Coral, AVISPA and Athena seem to be the most powerful: these tools are capable of determining whether or not a protocol is valid. In the case of unsatisfiability, a counterexample is output. A counterexample indicates a flaw on the protocol, in other words, it indicates a way to obtain an attack.

Complementing the use of formal methods, other authors have taken a different approach, proposing design principles in order to help in the construction of better security protocols. The rational behind this initiative originates from the observations that successful attacks against security protocols are a result of bad practices in security protocol design and that good design principles can lead to protocols that are significantly more robust.

The most important design principles for security protocols have been proposed by Abadi and Needham [1]. Abadi and Needham arrived at these principles by noticing some common errors in the design of security protocols. If these errors are avoided, protocols tend to become more readable and, more importantly, more correct.

1.1.1 Hypothesis

Although there exists formal development support, as well as informal design guidelines, a lot of protocols, whether recent or not, are faulty. Further aid for protocol development is thus required. This thesis is about a related but far less explored problem to protocol verification: the correction of faulty security protocols. A flawed protocol is a mal-formulation. Mal-formulations may become evident by the appearance of a failed proof attempt, possibly accompanied with a counterexample. Experience has shown that the analysis of these evidences often holds the key to the completion of proofs or to the correction of a faulty model [54].

So, this thesis has been arisen from the following hypothesis:

*Using Abadi and Needham’s principles in the analysis of both the description of a faulty protocol and of one of its counterexample holds the key for pinpointing the flaw in the protocol and for suggesting candidate patches to fix it.*

We consider that to patch a faulty protocol we must analyze the specification of the protocol and its attack. The attack is one or more security requirements that the protocol does not satisfy, which was verified using some state-of-the-art verification tool. Then, we must diagnose which
are the faulty protocol messages that make the protocol to be faulty and then, to propose candidate patches. It is important to clarify that finding attacks in the protocol is not the same with finding a protocol flaw. While the first one refers to finding a script evidencing that one or more security requirements are not satisfied by the protocol, the second one we take to mean finding the protocol messages that violate one or more design principles. Violations in design principles are the main cause of attacks in security protocols. Our work is precisely concerned with finding these kinds of violations.

1.1.2 Assumptions

The current support for security protocol analysis has two main approaches: the computational complexity and the formal methods, see section 2.2 for more details. Our research lies in the formal methods approach. Then, we do the following assumptions:

1. Formal methods adopt the perfect cryptography assumption, which states that the only way to decrypt a cypher-text is to have the appropriate key.

2. The intruder is modeled as a powerful agent that controls the network but cannot make cryptanalysis. The intruder follows the Dolev and Yao model [38].

3. We also follow the free algebra assumption, which specifies that two messages are equal if and only if they are syntactically equal.

Perfect encryption and free algebra assumption entail the type of attacks that can be found in this model and thereby the type of faulty protocols that we can patch. Roughly speaking, these attacks refer to the following deduction problem: given a state of a protocol run, it is possible to determine whether the intruder is able to construct a message of the form that some honest agent is expecting to receive (to break authentication), or whether he is able to obtain a message that is intended to be secret (to break secrecy), e.g. a key shared by two honest agents [16]. More details will be approached in section 2.3.1.

1.2 Overview

The result of our research was SHRIMP, a Smart metHod for Repairing IMperfect security Protocols. SHRIMP aims at speeding up the formal software development cycle, bridging the gap between design and analysis by means of diagnosis and repair. It offers benefits to practising security engineers, including getting a better insight into a protocol flaw and enabling incremental
protocol design. These features are all of interest because nowadays protocols are more complicated than just 3—5 steps (e.g. the SET protocol) and their various parts are intertwined, making it hard for a human to cope with all the subtle dependencies.

SHRIMP relies on existing state-of-the-art tools both to analyse an (intermediate) protocol\(^1\) and, if the protocol is flawed, to find one or more of protocol runs violating a given security requirement, called an *attack*. It then analyses the protocol and the attack to pinpoint the faulty steps of the protocol and synthesises appropriate changes to fix them. This yields an improved version of the protocol that should be analysed and potentially patched again until no further flaws can be detected. Our experiments show that it is not necessary to explicitly consider the property the protocol fails to satisfy; this might be attributable to that such a property is already implicit in the attack.

To identify and patch a protocol flaw, we have translated some of the informal principles for the design of security protocols of Abadi and Needham [1] into formal requirements. For each requirement, there is a collection of rules that determine if the attack shows common patterns to a specific flaw, with this, we can identify if the protocol violates one or more design principles. The correction of security protocols incorporates the use of several of these rules. However, the patches are not independent and the application of a rule requires preconditions to be applicable and should guarantee postconditions once it has been applied.

As a general framework to organise the application of diagnostic and repair task, we have adopted Bundy’s proof planning methodology [22]. So, SHRIMP is a set of patch methods, each of which is able to deal with a general class of faults. SHRIMP includes two classes of methods: *patching methods* and *compound methods*. A patching method consists of a structure with 5-tuple(name, input, preconditions, patch and effect): the name of the method; the input, which is often the description of a faulty protocol and an attack; the preconditions, a formula written in a meta-logic (which we have developed) that the input objects must satisfy; the patch, a procedure specifying how to mend the input protocol; and finally the effects, a formula specifying required properties of the newer version of the protocol. A compound method is a 4-tuple (name, input, preconditions and method where one can invoke other methods). The main difference between a patching method and a compound method is that the latter can invoke other methods, while the former cannot.

We have hitherto focused on automatically fixing protocols subject to a replay attack, since many known faulty protocols fail to resist it.\(^2\) A *replay attack* is a form of attack where a data transmission is repeated or delayed. SHRIMP takes care of replay attacks where the message be-

---

1\(^{1}\)We have used AVISPA throughout our experiments. AVISPA can be downloaded via [http://www.avispa-project.org/](http://www.avispa-project.org/).

2\(^{2}\)Most of the attacks reported in the Clark-Jacob library [30] are *replay attacks*. 
ing reused is a cypher-text and hence SHRIMP includes a compound method called *replay*. This compound method calls three patching methods (*message encoding, agent naming* and *session binding*).

With these patching methods, SHRIMP deals with the full class of replay attacks proposed by Syverson [84], within the context of *typed messages.*\(^3\) We have successfully tested SHRIMP on 36 protocols, 21 out of which were borrowed from the Clark and Jacob library, obtaining a repair rate of 90%.

### 1.3 Contributions

The automated correction of faulty security protocols is not a developed area in the formal support for security protocol development. While there are many approaches focused in the analysis of security protocols and a number of design principles, before SHRIMP there was not automated support for repairing imperfect security protocols. In assessing the contribution made by this research, we note that SHRIMP contributes in different ways. In particular:

- **SHRIMP** contributes in the formal support for security protocol development bridging the gap between design and analysis by means of a diagnosis and repair task. In this aspect, SHRIMP is the first approach to fix faulty security protocols automatically.

- As a result of Abadi and Needham’s principles formalization, we have identified three classes of replay attacks in terms of a mal-designed cypher-text. A cypher-text that has been used to build up a replay attack violates one or more of the following principles: i) the originator/recipient of a cypher-text in one message of the protocol cannot be distinguished; ii) two or more different cypher-texts of the same protocol cannot be distinguished from one another; the protocol portrays two or more cypher-texts that are different one another but have similar structure; and iii) two or more different runs of the same protocol cannot be distinguished from one another. The message cannot be associated with a particular protocol run. The protocol does not guarantee association or temporal succession.

- **SHRIMP** deals with the full class of replay attacks proposed by Syverson [84], within the context of typed messages.

- **SHRIMP** also contributes as a general knowledge for the design of sound security protocols. The general composed method, called *replay*, together with the three patching methods:

\(^3\)We emphasize that SHRIMP patches the full class of replay attacks within the context of typed messages because protocols that SHRIMP fails to fix are precisely susceptible to *type flaw attacks.*
agent_naming, message_encoding and session_binding, give a general knowledge about how to avoid violations to design security protocols.

1.4 Outline

We briefly outline here the following chapters with their contents:

Chapter 2 describes the current support in security protocol development comparing the most important formal verification approaches towards the analysis of security protocols, and shows that neither of them addresses the problem of finding the specific flaw, let alone patching a protocol. The chapter also illustrates that, using the counterexample alone, it is not trivial to identify the flaw causing the problem in a protocol.

Chapter 3 reviews informal guidelines for strengthening the design and the robustness of security protocols proposed in the literature and supports why we have selected Abadi and Needham’s principles. The chapter also shows common errors in the design of security protocols, stated precisely in Abadi and Needham’s principles, and how these principles can guide us to both find these errors and patch faulty security protocols.

Chapter 4 explains the class of protocol flaws that SHRIMP deals with and describes how this class exploits a violation of a good practice for protocol design, as stated by Abadi and Needham. Then, we give an overview of SHRIMP: describing where SHRIMP is located in the security protocol development process; the methodology used so that SHRIMP patches faulty protocols; how SHRIMP works and how new patching methods could be added to SHRIMP. Finally, we describe the logic on which SHRIMP has been constructed.

Chapter 5 introduces SHRIMP’s meta-logic, with which we can express formulae to capture mal-designed cypher-texts (properties of faulty protocols, which are then used to predict protocol flaws) and symbols about how to repair them.

Chapter 6 introduces the mechanism about how SHRIMP patches faulty security protocols. This mechanism is done through a patching framework, which is formed by one general compound method, called replay, and three patching methods: message_encoding, agent_naming and session_binding.

Chapter 7 describes implementation aspects and summarises the results produced by testing SHRIMP against a set of faulty security protocols. Both positive and negative results are clearly stated in order to characterise outstanding patches and limitations of the patching framework.
Chapter 8 aims to compare our work against rival approaches, and, second, it aims to give indications for further research work. To achieve the second aim we first introduce three protocol examples which allow us to explain the future work at more elaborate level of detail.

Chapter 9 concludes the thesis. The research presented throughout the dissertation is summarised and discussed.

1.5 Notation

In this thesis, the protocols are presented in Alice and Bob notation (A&B for short). For each protocol step (which an agent sends a message and another agent receives it), the step number, the sender and the intended recipient of the message, and the message itself are expressed as follows:

\[ 1. A \to B : M \]

Messages will be denoted by fat braces, using a notation due to Paulson [74], external braces being omitted. The cypher-text obtained by encrypting a plain-text \( M \) with a cryptographic key \( K \) is denoted as \( \{ |M| \} \).

\[ 1. A \to B : M \]
\[ 2. B \to A : \{ |M; M'| \} \]

Figure 1.1: An example message exchange

The example protocol in Figure 1.1 serves to demonstrate the notation. It consists of two steps. In the first step, agent \( A \) sends agent \( B \) plain-text \( M \), which \( B \) receives. In the second step, \( B \) replies to \( A \) with a cypher-text obtained by encrypting the concatenation of \( M \) and another message \( M' \) with a key \( K \). Throughout this thesis, we shall use the terms agent and participant indistinctly. Also we shall refer to the penetrator, spy and intruder indistinctly.
2. Support for Security Protocol Development

In this chapter, we describe two security protocol libraries that serve as our main source for faulty security protocols. The number of faulty protocols in these libraries evidences the need for support in the development of security protocols. The so-called formal approach to security protocol analysis has exclusively concentrated on three main approaches: belief logic, state exploration, and theorem proving based. We will compare the most important formal verification methods and we will show that although some of these methods can find attacks in a faulty security protocol, neither of them addresses the problem of finding the specific flaw, let alone patching it.

We will also illustrate that, using the counterexample alone, it is not trivial to identify the flaw causing the problem in a protocol. Thus, we conclude that further support for protocol development is required.

2.1 The Need for Support in Security Protocol Development

Although security protocols consist of only a few messages, designing a security protocol is an error prone task. Protocol flaws have gone unnoticed for quite a long time. For example, consider the Needham and Schroeder public key protocol (NSPK), introduced in 1978 by Roger Needham and Michael Schroeder. The protocol steps concerned with the authentication of the participants are the following:
1. \[ A \rightarrow B : \{N_a; A\}^{K_B^+}_{K_B} \]

2. \[ B \rightarrow A : \{N_a; N_b\}^{K_A^+}_{K_A} \]

3. \[ A \rightarrow B : \{N_b\}^{K_B^+}_{K_B} \]

Ideally, in step 1, agent \( A \) acting as an initiator begins a run with \( B \) acting as a responder by sending him a message encrypted under the public key of agent \( B \), denoted \( K_B^+ \). This message contains a nonce\(^1\) and the agent name \( A \). Upon reception agent \( B \) decrypts the package from \( A \), and sends back both the nonce \( N_a \) and a new secret (nonce \( N_b \)) he has generated, step 2. \( A \) decrypts the last message sent by \( B \), and checks that \( N_a \) is indeed the nonce generated to start the run. In that moment \( A \) authenticates \( B \), as only he could have read message 1. \( A \) responds by sending back nonce \( N_b \). \( B \) will receive this and authenticate \( A \), as only she could have decrypted message 2. \( A \) and \( B \) can now use nonce \( N_a \) and \( N_b \) to sign their messages to each other.

Even though the NSPK is a very simple protocol and it seems right at first glance, a flaw was found 17 years after its publication (in section 6.3.1 we detail this flaw). Another example protocol that was thought to be correct for a long time is Otway and Rees, introduced in 1987. In [24], the authors found this protocol faulty and then suggested changes to the original version. However, such a new protocol version was still faulty and so they amended it again but 7 years after [1]. We could go on mentioning other stories of faulty protocols in the same situation (e.g. the Denning and Sacco protocol, the Needham and Schroeder Symmetric key protocol and so on). Most protocols are documented in two main libraries: the Clark and Jacob library\(^3\), [30], and the AVISPA library\(^4\).

The Clark and Jacob’s library comprehends 50 protocols, 26 of which are known to be faulty and the rest of them are not. The AVISPA library comprehends 70 protocols, 17 of which are faulty and the rest of them are not, according to the AVISPA tool.

For every protocol that is faulty, both libraries include the protocol and an extended version of it that is not. This evidences that the design of security protocols is error-prone; flaws in security protocols have gone unnoticed for several years after protocol publication. In some cases, crimes such as user impersonation or unauthorized access to information have already resulted in countless losses. So, security protocols are considered safety critical applications.

In the next section, we briefly recall software development, focusing on security protocols, and the methods used for supporting it.

---

\(^1\) A nonce is a random, unguessable number that has not been used before. This newness is usually referred to as freshness in the literature.

\(^2\) Agent names are used to refer to the identity of a participant.

\(^3\) The Clark and Jacob library is available via http://www.lsv.ens-cachan.fr/spore/index.html

\(^4\) The AVISPA library is available via http://www.avispa-project.org/
2.2 Current Support for Security Protocol Development

The software development process is a structure imposed on the development of a software application. There are several models to software development, each describing approaches to a variety of tasks or activities that take place during the process. The best-known and oldest model is the waterfall model, which consists of the following levels: requirements analysis, specification, design, construction (implementation or coding), integration, testing, and maintenance.

In particular, when software is safety critical, the use of mathematical models is highly recommendable. These models solve software (and hardware) problems mainly at the specification, design and testing levels. Because security protocols are safety critical applications, researchers have focused their efforts in verifying security protocols in order to reduce risks after implementation. So, two distinct schools have arisen [2]: one of them relies on formal methods; and the other one on computational complexity and probability. In both communities the crux is cryptographic operations.

2.2.1 The formal methods school

In the formal methods school, both the program and its specification are modeled using some, not necessarily the same, formal language. Then mathematical, rigorous reasoning is used to demonstrate that the program and its specification are related, e.g. by some kind of equivalence or by logical implication. With respect to security protocols, formal methods are used to model the role of each protocol participant, together with the security properties to be verified. Then, determining whether a protocol is faulty or not amounts to proving that the intended behaviour of the protocol can be achieved regardless the capacities of an intruder. The intruder model usually is Dolev-Yaho but some extensions are also considered. We will have more to say about the Dolev-Yao model of the intruder in Section 2.3.1.

Crucially, as mentioned in Section 1.1.2, formal methods adopt the perfect cryptography assumption; under this consideration, there is no way to recover $M$ or $K$ from just knowing $\|M\|_K$. A protocol is considered free of attacks if the security properties being verified hold. In some techniques, a script describing a protocol attack is generated when the security properties do not hold.

In general, there are three main approaches for protocol verification: those based on belief logics such as BAN [24] and GNY [21]; another methods are based on state exploration, like NRL [59], FDR [66], AVISPA [11], strand spaces model like [86, 80]; and finally, another methods are based on theorem proving, such as the Paulson’s Inductive Method [74], Coral [82]. Like our research lies in the formal methods approach, in Section 2.3 we survey these approaches.
2.2.2 The computational complexity school

In this school cryptographic operations are seen as functions on strings of bits. Roughly speaking and following [2], a symmetric encryption scheme is defined as a triple of algorithms $\Pi = (K, E, D)$. Algorithm $K$ is a key generator which makes random choices, generating string $k$ (the key). $E$ is an encryption algorithm that maps strings $k$ and $m$ (the plain-text) into a string $E_k(m)$ (the cypher-text). Algorithm $D_k(c)$ is a decryption algorithm that maps strings $k$ and $c$ (the cypher-text) into a string that recuperates $m$. It is expected that $D_k(E_k(m)) = m$ for appropriate $k$ and $m$.

An adversary is essentially modeled as a Turing machine which has access to an oracle. The oracle aims to find the appropriate $k$ and $r$ that are able to decrypt a given cypher-text $E_k(m)$. To do this task, the oracle usually has some clues to facilitate its task such as knowledge of some components of $m$, knowledge of other messages encrypted under the same key $k$, etc. A protocol is considered good if the oracle cannot find $k$, or while consuming the computational power at hand the probability of finding $k$ is slow-growing under a determined threshold.

This computational school originated with the work of Blum and Micali [20], Yao [90], and Goldwasser and Micali [42]. In these works have been introduced a sophisticated body of definitions and theorems, which strengthen the scientific foundation of cryptography. These definitions are still applied, although for small protocols, in recent works like [19, 79].

This computational school has focused in three basic notions, [40]: i) the notion of computational indistinguishability; ii) the notion of zero-knowledge proofs; and iii) the notion of one-way functions. The first notion is about effective similarity, which was introduced by Goldwasser, Micali and Yao [42, 90]. The underlying idea is that they do not care whether or not messages are equal, all they care is whether there is a difference between the messages that can be observed by a feasible computation, otherwise, the two messages are equivalent.

The second notion is zero-knowledge, which was introduced by Goldwasser, Micali and Rackoff [41]. This refers to proofs that yield nothing beyond the validity of the assertion. That is, a verifier obtaining such a proof only gains conviction in the validity of the assertion. This is formulated by saying that anything that is feasible computable from zero-knowledge proof is also feasible computable from the (valid) assertion itself.

The third notion is about one-way functions and it is concerned with the construction of schemes which are easy to operate but hard to foil (modern cryptography). Modern cryptography rises or falls with the question of whether one-way functions exist. One-way functions are functions that are easy to evaluate but hard to invert [55].

Recent works are related with applications of pseudorandom generators under the notion of one-way functions; in [46] is stated that pseudorandom generators exist if and only if one-way functions exist.
2.2.3 Connections between both views

As mentioned in [36], connections between the formal view and the computational view are currently subject of investigation. These works have started independently with [78] and [2]. In particular, in their seminal paper, Abadi and Rogaway [2] have established a relation between syntactic equivalence of message terms, where encryption is treated as a formal operator, and secure encryption, where encryption is defined in terms of computational indistinguishably. The result of Abadi and Rogaway is applicable for symmetric encryption in the presence of a passive attacker. Some semantics connections between these views have been developed by the team at IBM Zurich, e.g. [13]. Herzog, [51, 50], shows that if a protocol attack exists in a formal model, there is an attack in the computational model. Recently, Baudet et. al. [18] introduced a reasoning framework for proving soundness of implementations of equational theories, which are used to specify cryptographic primitives. Recently, Kremer and Mazare [52] have extended Baudet et al.’s work to consider an adaptive user, rather than a purely passive one.

This thesis lies in the formal methods approach. So, in the next section, we describe in more level of detail the formal approach for security protocol verification and survey the most significant verification tools/methods.

2.3 Security Protocol Verification Using Formal Methods

There are three main approaches for protocol verification: belief logic, state exploration, and theorem proving. In this section, we will compare the most important verification methods of these approaches, and show their main contributions. While revising each work we pay special attention on the ability of finding attacks.

Before going further, though, we briefly describe the formal approach to software verification.

2.3.1 Formal Methods for Security Protocol Development

In general, formal methods can be applied at various points through the development process. From a high level perspective the formal methods community for security protocols development has concentrated its effort in the specification and verification processes. Roughly speaking, the task of verifying security protocols nowadays requires two main steps:

**Formal Specification:** First a protocol must be translated from an informal specification to its formal one, using some logic. In this step the properties to be verified are also formalized. This is a key step and requires human intervention.
**Verification:** Once formalized, the protocol and the property to be verified are then sent to an analyzer. If search-based, the analyzer may yield one of two outputs: OK, indicating that the protocol is free of errors, or a script, pinpointing that the protocol is flawed. This script is a counterexample: an interleaving of one or more protocol runs violating the security property. As we will see in the following section, most verification approaches include an automatic counterexample finder.

Then, determining whether a protocol is faulty or not depends on the security properties being verified.

**Security properties**  In general, the formal methods community has focused in two security properties: Authentication and Secrecy.

- **Authentication:** one agent should become sure of the identity of the other agent through a message exchanged between them. This property has various meanings, however, Lowe [62] has proposed a hierarchy of four security levels to interpret more formally this property.

- **Secrecy:** Certain parts of messages (so-called secrets) over the network are only readable by their intended recipients.

Formal methods approach designs an intruder model with some capabilities. If the protocol is faulty, then the intruder can break one or more of the security properties being verified.

**The intruder abilities**  Dolev and Yao, [38], formalized a model of a spy which has become the standard reference in the literature. In particular:

- He cannot do cryptanalysis.

- He is able to see all traffic in the network.

- He can delay messages.

- He can prevent that messages reach their intended recipient.

- He can split messages he has seen in the traffic.

- He can add fake messages.

- He can forward messages he cannot read (cypher-texts).
Some authors also assume that the spy knows all public keys, his own shared and private key, and all shared and private keys of compromised agents. In addition, other authors also consider that the spy knows some lost session keys. With these abilities the intruder is able to break with the security of many protocols.

In the following subsections, we survey the most prominent formal methods for protocol security verification.

2.3.2 Belief logic

It would not be fair to dismiss belief logic for being a model not much used currently. Belief logic was one of the first attempts to make the reasoning about the properties of security protocols more systematic. This logic formalizes what an agent may infer from the messages it receives. The BAN logic [24], due to Burrows, Abadi and Needham, is a modal logic that allows short, abstract proofs. To use BAN logic one must: i) idealize the protocol; ii) identify the initial security assumptions of a protocol; iii) use the rules of the logic to deduce new beliefs; iv) interpret the statements that have so far been proved and v) check whether the authentication goals have been reached.

BAN has been able to identify some protocol flaws but missed others. For example, BAN missed to identify Lowe’s attack upon the NSPK protocol [58]. This is because BAN does not consider an intruder on the network. Its main characteristic is its simplicity. Its main disadvantage is that it assumes that all of the agents are honest: it does not include a model of a spy. When authentication is not satisfied, it is not easy to see how to use the BAN deduction tree to generate a counter-example.

New belief logics were derived from BAN, e.g. BGNY [21] and AUTLOG [88]. They address some weaknesses of BAN but sacrifice its simplicity. BAN is still applied to analyze a new protocol, but only as a sort of first test. If a protocol passes the BAN test, then it is subject to a more thorough analysis, using one of the methods below.

2.3.3 State Exploration

In the state exploration approach, a protocol is characterized by the set of all its possible traces. Given an input security protocol, the verification method explores as many execution paths of the protocol as possible, checking each reachable state if some condition holds or, alternatively, if are violated. Since the search space generated from analyzing a cryptographic protocol may be infinite, this method can guarantee that a property holds (respectively an attack exists) only for a maximal number of finite runs. State exploration methods are powerful and automatic. When a property does not hold, they can easily exploit the traces that have been generated in order to build a counterexample. Yet, turning a state exploration technique into a decision procedure often requires
(finiteness) strong assumptions to bound the information that is analyzed. This is a limitation because there are infinitely many messages that attackers can possibly send.

2.3.3.1 Early Attempts at Protocol Verification

**Dolev and Yao’s method**  The first attempt at using state exploration technique for verifying security protocols is due to Dolev and Yao[38], 1983. Dolev and Yao developed a model that consists of a set of algorithms for determining whether or not a protocol is secure. The model is extremely limited, because they approach only secrecy issues and consider only encryption and decryption. One of Dolev and Yao’s main contributions is a model of the intruder, which we have already described in 2.3.1.

**Meadows’s method** In 1996, Meadows presented the NRL Protocol Analyzer (NPA) [66], based upon a version of the term-rewriting model of Dolev and Yao. NPA goes beyond of this model in two aspects. First, the goals of the *Spy* are more than just finding out secret messages. He may try, for instance, to convince an agent that a message has certain properties that it does not. For example, he may convince an agent that a message already known by the *Spy* is a session key. Second, instead of developing a set of algorithms as Dolev and Yao, she developed a general procedure for proving security properties of protocols by proving user-specified protocol states unreachable.

NPA found previously unknown flaws in the Simmon Selective Broadcast Protocol and the Burns-Mitchell Resource Sharing Protocol, and identified hidden assumptions in the Neuman-Stubblebine re-authentication protocol and the Aziz-Diffie wireless communication protocol. However, NPA has no systematic way of converting a protocol description into a set of transition rules. The tool also relies heavily on the user during the verification much in the same way as a theorem prover relies on the user to guide it during the search for a proof. Finally, the algorithms used in NPA are not guaranteed to terminate.

In the following, we describe two important families of state exploration methods: model checking and strand spaces.

2.3.3.2 Model checking

*Model checking* is a state exploration technique for verifying finite state concurrent systems. This method has been successfully used in practice to verify, for example, complex sequential circuits

---

5NPA is a special-purpose tool, written in prolog, that is used to authenticate principals, services and distribute keys in a network.
and communication protocols. The state space can then be traversed to check if some particular state is reached or if some trace is generated. The main challenge of model checking tools is dealing with the state explosion problem as any other state exploration technique.\(^6\)

We describe two families of model checking: process algebra and on the fly model checking.

**Lowe’s method**  Gavin Lowe\(^7\) has approached the verification of security protocols using Hoare’s calculus of Communicating Sequential Processes (CSP). To verify a security protocol, Lowe uses CSP to express a model of the whole communicating system: each agent taking part in the protocol, the communicating network and a Dolev and Yao spy. Then, he uses the standard CSP theory of traces to conduct protocol verification.

In 1996, Lowe used the Failures Divergences Refinement Checker (FDR), a model checker to verify CSP programs, to analyze a version modified of the NSPK protocol showing it is secure at least for a small system [59].

**Basin’s method**  In 1999, Basin [17] presented an approach to protocol analysis that combines complementary aspects of model checking and the formalism of Paulson (see section 2.3.4.1). This approach tries to deal with the state explosion problem and so its key idea is to use lazy data types to model the infinite state-space associated with a protocol. A lazy data type is one where data-type constructors build data types without evaluating their arguments; this allows to represent and compute with infinite data.

For modeling protocols and attackers Basin uses a trace-based interleaving semantic, motivated by, and closely following to Paulson’s logic. Rather than formalizing protocols as inductively defined sets as Paulson does, he formalizes them as infinite trees. The branches of the tree are traces and children correspond to trace extensions by a step of the protocol or an action by an attacker. Hence, a protocol, along with the Dolev and Yao spy, defines an infinite tree and a security property is a property of nodes in the tree. Violations of security properties are found by a kind of infinite-state model checking, which is performed by searching the infinite tree. So, if there is an attack, it will be present in a trace located at some node in the tree. However, finding

\(^{6}\)The state explosion problem occurs in systems with many components that can interact with each other or systems that have data structures that can assume many different values. The most successful techniques for dealing with the state explosion problem are based on a partial order reduction. These techniques exploit the independence of concurrently executed events. Two events are independent of each other when executing them in either order results in the same global state. A common model for representing concurrent software is interleaving.

\(^{7}\)Gavin Lowe is well-known for having found an attack on the NSPK protocol in 1995 [58], 17 years after it was introduced. Lowe’s attack allows an intruder to impersonate one agent in a session (we will have more to say about this in Chapter 6).
this node may not be easy. Not only may the tree be infinitely deep, but it also may have a large branching factor.

An exponential branching factor means that standard search algorithms are unlikely to succeed in finding even relatively simple attacks that might reside at shallow depths. Therefore, he uses two simple heuristics both to prune the search tree and to reorder the way it is searched. The first heuristic is to prune those traces that contain events that could not have consequences, such as traces where honest agents receive messages that do not follow the rules of the protocol. Second, he assigns priorities to events: he gives the highest priority to those events that could arise from the first step of the protocol and when the spy is involved.

In 2003, Basin et al. introduced the On-The-Fly Model-Checker OFMC [14], a tool that combines the technique above mentioned with the integration of symbolic techniques for modeling a Dolev and Yao intruder, whose actions are generated in a demand-driven way. This technique significantly reduces the search space without excluding any attack. Basin et al. have carried out a large number of experiments to validate their approach and have found attacks documented in the Clark and Jacob library and reported new ones in the AVISPA library (described in section 2.1).

The formal model that OFMC tool uses for protocol analysis is based on two specification languages, which they have been developing in the context of the AVISPA project: a high-level protocol specification language (HLPSL [26]) and a low-level one (the Intermediate Format IF). In appendix A we outline the AVISPA tool mechanism, since we have used this tool throughout our experiments.

Until now, the AVISPA tool [6] (specifically the OFMC backend) is considered to be the state of the art and the more powerful tool for protocol verification. It is capable of determining whether or not a protocol is valid, usually in a few milliseconds and, in the case of unsatisfiability, a counter-example is output. The downside of OFMC is that it cannot verify group security protocols.

### 2.3.3.3 Strand Spaces approach

**Fabrega et al.’s method** This method arises because the state exploration methods, due to the combinatorial explosion problem, sometimes cannot establish if a protocol is really correct specially when no attacks are found in the protocol. The Strand Space community has spent its efforts in demonstrating the correctness of a protocol when it really is. Fabrega, Herzog and Guttman introduced this method in 1998 [86]. A **strand** is a sequence of communications events (sending and receiving messages) that an agent may carry out in a protocol. A **strand space** is a set of strands.

---

8The AVISPA project (Automated Validation of Internet Security Protocols and Applications) supports the simple integration of four different back-end search engines: the On the Fly Model Checker (OFMC) [15], the Constraint-Logic-Based Attack Searcher (CL-AtSe) [28], the SAT-based Model-Checker (SATMC) [7] and the Tree Automata based on Automatic Approximations for the Analysis of Security Protocols (TA4SP) [47].
reflecting the activity of honest principals involved in a protocol, together with a number of strands of the intruder.

The strand space is a graph-based method. An event is represented by a node in the graph. When an agent sends a message $M$ the node is labeled with $+M$ and upon reception with $-M$. Every node belongs to a unique strand. A strand may have one or more nodes belonging to an agent. An edge in the graph is indicated by two events, $n_1 \rightarrow n_2$, meaning $n_1$ (node 1) sends a message, which is received by $n_2$ (node 2).

In order to prove the correctness of a protocol the concept of a bundle is introduced. A bundle is a finite acyclic sub-graph consisting of a number of strands (of honest agents or of the Spy) hooked together where one strand sends a message and another strand receives it.

In this method protocol correctness depends essentially on the freshness of data items such as nonces and session keys. These elements must not be known by the spy. Typically, for a protocol to be correct, each such a bundle must consist of one strand for each of the legitimate parties, all agreeing on the participants, nonces and session keys. Spy strands may also be entangled in a bundle, even in a correct protocol, but it does not prevent the legitimate parties from agreeing on the data values or from maintaining the secrecy of the chosen values.

For example, proving authentication of type non-injective agreement with $B$ acting as a responder and $A$ as an initiator (according to Lowe’s hierarchy of authentication [62]), it is necessary to establish that, whenever a bundle contains a responder strand using certain data items, $x$, then such a bundle must also contain an initiator strand using $x$. Now, proving authentication of type agreement amounts to showing that such a bundle contains an unique initiator strand using $x$.

The Strand Space method cannot report a counter-example and its verification process is not automated. However, its advantage is providing a simple way of expressing the protocol information flow in a graphic way. The model is appealing because runs can be illustrated in parallel. Later on, we will describe the strand spaces logic in more detail, since SHRIMP has been accommodated on this logic.

**Dawn Song’s method** In 2001, Song, Berezin and Perrig used techniques from both model checking and theorem proving in order to extend the strand space model so that it is both automatic and able to prove correctness of many security protocols with arbitrary number of concurrent runs. The new method has been implemented in Athena which exploits several techniques that enable it to efficiently analyze infinite sets of protocol runs. Athena includes some additional optimizations to improve efficiency, such as pruning theorems and developing an automatic procedure for evaluating well-formed formulas. Athena works as follows; first it transforms the security property to be verified into an initial sequent. It then applies a small set of inference rules with certain decision procedures to the states, building a proof tree, until it either completes the proof or refutes the
sequent. In the latter case Athena reports that the protocol is incorrect, and the state of the refuted sequent represents a counterexample.

One downside of Athena is that the verification procedure is not guaranteed to terminate. Termination can be enforced by bounding the number of concurrent protocol runs and the length of the messages. This naturally, reduces the search space.

2.3.4 Theorem proving based approaches

Theorem proving based approaches attempt to produce a formal proof, given a description of the protocol, a set of logical axioms, and a set of inference rules. These approaches have two branches: one is based on High Order Logic (HOL) and the other based on First Order Logic (FOL). Although, the former approach can inductively simulate an infinite number of agents running in an infinite number of sessions, it requires considerable interaction with the theorem proving framework. The latter approach is more automatic and more viable for generating counterexamples.

2.3.4.1 Methods Based on High Order Logic

Paulson’s method The inductive approach to the verification of security protocols was invented by Paulson in 1998 [74]. This method combines some characteristics from state exploration and belief logics. From the first approach, it borrows a concrete notion of events; from the second one, the idea of deriving guarantees from each message. It has been widely used for verifying protocols, but it requires a high level of skill to use. A user must guide the proof process, selecting the logical rules or tactics to be applied. To compound the problem, the inductive approach involves cumbersome and burdensome proofs. The analysis was done using the generic, interactive higher-order logic theorem prover Isabelle [73].

In this method, a protocol is inductively defined as a set of (all possible) traces. A trace is a list of (communication) events. Mostly, an event refers to either sending a message, or receiving one. Protocol descriptions include attacks and accidental losses of critical information, so-called Oops events.

The model involves four sorts of theories: message, event, shared and public. The message theory includes sorts of agents and sorts of messages. The event theory models the sending and receiving communication events, the knowledge of agents and a freshness operator. The Shared and Public theory involve shared-key and public-key cryptography respectively.

A protocol is described as a collection of inference rules. Each inference rule has zero or more hypotheses and one conclusion. The inference rules state the various forms in which a protocol trace can be possibly extended with new events. In this method for verifying a protocol, first, a
security property must be formulated, e.g. a key remaining secret, by some formula $P$. Then using the theories mentioned previously, it needs to be proven inductively that: $\forall t : \text{trace}.P(t)$.

One of the main downsides of Paulson’s approach is that there is no automated support for finding attacks on faulty protocols. Another disadvantage is that verifying protocols requires interactive theorem proving, which demands considerable effort.

2.3.4.2 Methods Based on First Order Logic

**Weidenbach’s method**  Weidenbach’s approach [89] tries to combine the benefits of finite state analysis and the inductive method. The idea is to use fragments of first-order logic that are expressive enough to have infinite, inductive models, but that are still subject to automated theorem proving.

For verifying a security protocol, Weidenbach translates a protocol together with the security property to be verified into first-order monadic Horn fragments.\(^9\) He analyzes the resulting formulas using SPASS.\(^10\) If SPASS terminates, then the protocol is free of flaws. Otherwise, the protocol contains an attack.

Weidenbach analyzed the Neumann-Stubblebine protocol (1993) and found flaws in it, giving a possible solution. Regardless of the success obtained in the verification of this protocol, he has suggested the accomplishment of a tool for the analysis of security protocols that should not limit itself to finite models. Unfortunately, in his method the abilities of the spy are weaker than the Dolev and Yao spy and he can model only a minimal number of agents. He restricts himself to a finite model and his formalism is not as expressive as Paulson’s approach. Another downside in Weidenbach’s approach is that it cannot generate counter-examples.

**Ernie Cohen’s method**  In 2002, Cohen [31] described a proof method that allows safety properties to be represented by ordinary first-order formulas. He has implemented the method in an automatic theorem proving, TAPS\(^11\), that proves safety properties roughly equivalent to those published in Isabelle by Paulson. TAPS generates these proofs quickly and with little or no guidance from the user. Cohen has used TAPS to analyze about 80 protocols, including most of the ones in the Clark and Jacob Survey [32].

For verifying a security protocol, Cohen models a protocol as a transition system. The state of the system is given by the set of transitions that have been executed and the set of messages that

\(^9\)It is called a Horn clause if it contains at most one positive literal. A monadic Horn theory is a set of Horn clauses where all occurring predicates are monadic (that is, they take only one argument).

\(^10\)SPASS is an automated theorem prover for first-order logic that uses a saturation methodology.

\(^11\)The Telcordia Authentication Protocol System.
have been sent in plain-text. A transition generates fresh values (to be used as nonces or keys), checks that some precondition holds, records that the transition has taken place, and sends a new message. Several implicit transitions model the actions of the Spy, and the states of the system can be further restricted by user-supplied axioms.

From the protocol description, TAPS generates a number of first-order invariants. Safety properties like authentication are proved from the invariants using ordinary first-order reasoning. All of this logical reasoning is performed by a resolution theorem prover. TAPS is usually able to generate a suitable invariant automatically, but the user has to provide sometimes formulas giving conditions necessary for certain nonces or keys. Finally, the user must provide a first-order formalization of the security properties to prove; TAPS tries to prove these goals from the invariants (using resolution theorem proving).

Cohen’s approach surpasses Paulson’s approach in that it generates proofs more quickly and almost without user guidance. The main downside compared to search-based systems is that TAPS does not generate counterexamples. TAPS searches for proofs, not for attacks.

**Graham Steel’s method** So far, most tools and techniques described above have focused on analyzing security protocols with a fixed number of participants. Very few approaches have appeared in the last few years to analyze protocols for group key agreement, where an arbitrary number of parties may be involved in a single round, [68]. Steel et al. developed Coral, a tool which gets around partially this problem [82].

Coral is a tool built on the top of theorem prover SPASS, it can not only detect an incorrect conjecture, but also supply counter-examples in order to allow the user to identify the attack of a protocol. The method uses the proof by consistency strategy to guide a search for a counterexample [81].

Proof by consistency is a technique for automating inductive proofs in first-order logic. It has also been called inductionless induction. Originally developed to prove theorems, this technique has the property of being refutation complete [33].

Coral is closely related to Weidenbach’s and Paulson’s method. Steel uses a first-order version of Paulson’s inductive formalism for protocol verification. The protocol is specified as an inductive datatype that represents the trace of all messages sent by the agents, the spy, the server, etc. Axioms specify the ways in which the trace may be extended by an agent sending a message (following the protocol), or a spy faking a message (using material he has seen in the trace). Like Paulson, his model allows an indeterminate and unbounded number of agents to participate, playing any role and using an arbitrary number of fresh nonces and keys, [83].

In Coral both the formal specification of a protocol and its security properties are specified in Horn clauses. The actions of the Spy follow the Dolev and Yao model and these are specified
in Horn Clauses too. To discover an attack on a protocol he formulates a security property such as authentication, by some formula $P$. In Paulson’s method, one would try to prove that: $\forall t : trace.P(t)$, to show the protocol is secure. However, when using Coral in this respect, one would try to find a counterexample $t1$ such that $P(t1)$ does not hold. This $t1$ is a trace of messages that breaks a security property $P$, i.e. an attack on the protocol.

CORAL has found two new attacks on the Asokan-Ginzboorg protocol for establishing a group key in an ad-hoc network of Bluetooth devices.

One downside of Coral is that verification is too time consuming, it may take even days. So, Coral does not compete for attack discovery, since model checking tools are better. It however automatically deals with group protocols, which are out of scope of other verification tools.

2.4 Fixing Faulty Security Protocols

As described in the previous section, when a protocol is faulty most verification tools include an automatic counterexample finder in order to display an attack on the protocol. With this result, the security protocol designer must fix the protocol manually and go on with the development process. For patching the protocol the designer must analyze the counterexample to identify the flaw causing the problem and then fix it. However, this may be a very hard task and in some cases more vulnerabilities may be introduced in the protocol. To illustrate this problem we present below the case of the Otway and Rees protocol.

2.4.1 A Faulty Security Protocol

The Otway and Rees protocol aims to authenticate two agents and to distribute a session key originated from a trusted server using symmetric cryptography\(^\text{12}\). Burrows, Abadi and Needham suggested [24] an “optimized version” of the Otway and Rees protocol, which was found to be faulty years later. The protocol in Alice and Bob notation is as follows:

1. $A \rightarrow B : M; A; B; \{\{N_a; M; A; B\}\}_B$

2. $B \rightarrow S : M; A; B; \{\{N_a; M; A; B\}\}_A; N_b; \{\{M; A; B\}\}_B$

3. $S \rightarrow B : M; \{\{N_a; K_{ab}\}_A\}; \{\{N_b; K_{ab}\}_B\}$

4. $B \rightarrow A : M; \{\{N_a; K_{ab}\}_B\}_A$

\(^{12}\)In symmetric cryptography two (or more) agents, $A$ and $B$, share a single key $K_{ab}$, used to encrypt and to decrypt each message.
A requests from B a new session, generating a plain-text and an authenticator (an authenticator is a cyphered component used to provide evidence of the message sender identity). Upon reception, B adds his own authenticator to the message received. When S receives message 2, it checks the integrity of the plain-text with authenticators of A and B. Then S sends B two key distributors (a key distributor is a cyphered component carrying a session key): one intended for A and the other intended for B. Next, B takes his key distributor and forwards the other part to A. A and B check their nonces. If they are the ones originated in steps 1 and 2 respectively, then they accept the session key.

A counter-example of the Otway and Rees protocol The above protocol seems right. However, four years after its appearance an attack was found [65]. The attack shows that the protocol cannot provide authentication of B with A, hence B accepts incorrectly a run of the protocol to share a session key with A, but A has never run the protocol. The attack is illustrated in Figure 2.1.

In this picture:

- Session step $sS : N$ means the $N$th message associated with session $S$.
- $B \rightarrow \text{Spy}(A) : m$ means that Spy intercepts $m$ which was originally intended for $A$;
- $\text{Spy}(A) \rightarrow B : m$ means that Spy impersonates $A$; and
- Dotted boxes have been included by illustrative issues. A dotted box with a $\uparrow$ denotes the origin of a cypher-text and a dotted box with a $\downarrow$ denotes a cypher-text being replayed.

---

13A session key is a key used for encrypting one message or a group of messages in a communication session.
Note that, in session steps $s_1 : 1$ and $s_1 : 2$ Spy’s authenticator is represented by a variable $X$ because $B$ cannot unveil its content. As you can see, to know how to repair the protocol by doing a simple inspection in the attack is not obvious. In the next subsection, we describe in more level of detail this problem.

### 2.4.2 Illustrating the Problem of Patching Faulty Security Protocols

For patching a faulty protocol we have to carry out two steps: firstly we must find the flaw from analyzing the protocol specification and its counterexample. Secondly we must modify the protocol specification in such a way that the attack is no longer possible.

#### 2.4.2.1 Finding the flaw

To do this, we analyze the attack and we must identify those messages that are being manipulated by the intruder. We will concentrate in the cypher messages, rather than the plain-texts, since plain-texts do not add much to establish security goals. The following analysis of the Otway and Rees attack will be concentrated in both its cause and effect.

The attack begins when $Spy$ takes possession (intercepts the message) of $\{M'; Spy; B\}_{K_B}$, which is formed by $B$ during a legitimate run of the protocol between $Spy$ and $B$. Such an authenticator is forwarded in session step $s_3 : 2$, where $Spy$ requests $S$ a session key to be shared between $Spy$ and $B$.

We can see that the effect of the attack is that $B$ accepts message $\{N_b; K_{spyb}\}_{K_B}$ in session step $s_2 : 3$, $B$ believes he shares session key $K_{spyb}$ with $A$ because of such a message also contains $N_b$ (which he sent in $s_2 : 2$ having received the request in session step $s_2 : 1$) whereas he in fact shares it with $Spy$. Because of that, he sent the message of $s_2 : 4$ apparently to $A$. Note that this message is a replay of session step $s_3 : 3$.

From this analysis we identify two candidate messages to be the cause of the problem: messages $\{M'; Spy; B\}_{K_B}$ and $\{N_b; K_{spyb}\}_{K_B}$. The first cypher-text is a candidate because the Spy can deceive the Server by requesting a false session. The second one is a candidate because the responder, when receiving the key distributor, accepts the session key and thereby he makes an incorrect authentication. In the following analysis we concentrate in the effect of the attack because we will try to avoid that the responder accepts an incorrect authentication. In addition, we will continue analyzing these messages, but now with their instances in the protocol description, which correspond to $\{M; A; B\}_{K_B}$ and $\{N_b; K_{ab}\}_{K_B}$.

---

14 The counterexample of a protocol is a script evidencing that the protocol does not satisfy one or more security requirements.

15 In this case, the cause of an attack consists in identifying a mal-designed cypher-text(s) that originated the attack. The effect consists in identifying where an honest agent, naively, accepts a message that he should not.
2.4.2.2 Fixing the protocol

To repair a protocol flaw, we must sometimes modify some protocol messages or maybe to introduce some protocol steps. This change sometimes includes to introduce some new elements in the protocol.

Returning again with our protocol example (Otway-Rees protocol), we need to identify if the receiver agent (of the replayed message) knows who agents are participating in the protocol from the message itself. Note that unlike authenticator message \( \{ M; A; B \} \), key distributor \( \{ N_b; K_{ab} \} \) itself does not state wholly which agents the session key is intended for: from the key \( K_B \) is obvious that it is directed to \( B \), but the other party is not stated. The omission of principal names is a violation to a design principle, which has been pointed out by Abadi and Needham [1].

To fix the protocol we concentrate precisely in the omission of agent names. So, a patch to the protocol would be to add agent name \( A \) in the key distributor as follows: \( \{ \boxed{A}; N_b; K_{ab} \} \) (see patch 1 in Figure 2.2, note that we enclose with a solid box changes in a protocol message). We could also include agent name \( A \) implicitly through including session identifier \( M \) in the key distributor as follows: \( \{ \boxed{M}; N_b; K_{ab} \} \) (see patch 2 in Figure 2.2). Another form of implicitly including agent name \( A \) in the key distributor (without doing changes in this message) is adding nonce \( N_b \) in authenticator \( \{ M; A; B \} \) as follows: \( \{ M; N_b; K_{ab} \} \) (see patch 3 in Figure 2.2). Doing this last change we arrive to the original version of the Otway and Rees protocol. Both patches above would provide weak authentication of \( B \) with \( A \) (see appendix A.3 for an authentication hierarchy).

---

**Figure 2.2: Alternatives of Patching Otway and Rees Protocol**

---

16 Although such a message could be also intended for the server, it is not confused because in the protocol specification the server never receives a key distributor.

17 We study in depth the subject of adding agent names explicitly and implicitly in 3.4 and 3.5.1.
As one can see, fixing faulty protocols is not an easy task especially because one has to do two time consuming activities. Firstly we must identify the flaw in the protocol, which is not trivial (in our example attack we concentrated in the omission of agent names, but it also could have been omission of temporal succession\(^{18}\)). Secondly, because there are a lot of possibilities of how the protocol could be patched (especially when the root of the flaw is unknown by the designer). It would be easier if the designer knows where the root of the flaw is located and thus he would try to patch the protocol in that specific part. Even though we could fix the protocol with some of the patches previously discussed, we would wonder whether the patch introduced is not violating any design principle.

So, we introduce SHRIMP, an automated tool that given the specification of a protocol and one of its attacks (a security requirement that the protocol does not satisfy) can not only detect the root of the flaw in a protocol but also give a patch. A tool like SHRIMP will help at the development process for security protocols and will benefit to designers in creating best security protocols, and thereby, reducing risks after implementation.

\section{Conclusions}

We have shown that there exists a number of security protocols that are known to be faulty. This is a clear demonstration that the design of security protocols is error-prone; flaws in security protocols have gone unnoticed for several years after protocol publication. Because of that, the formal methods community has put special interest in security protocols verification.

We have described a number of approaches for the analysis of cryptographic protocols such as belief logics, where intruders are not considered because a honest environment is assumed. Then we mentioned model checking approaches which are fast and automatic, but due to the state explosion problem they keep the state space small and some flaws can be overlooked. Next, we mentioned the inductive approach, which may accommodate arbitrary numbers of agents running in a parallel session. Unfortunately, this approach requires interactive theorem proving, which demands considerable effort. Furthermore, when a protocol does not satisfy a security property a counterexample is not given.

According to the advantages of model checking and the inductive approach, some characteristics of them have been used by several authors like Steel \cite{82}, Cohen \cite{31}, Basin \cite{17} and Weidenbach \cite{89}. Coral and AVISPA may be considered the state of the art. Coral stands out in that it can verify group security protocols, although it may take a long time to find an attack. AVISPA is fast and more automatic than whichever available tool.

\(^{18}\)We study in depth the subject of temporal succession in section 3.5.1
Even so, when one of these tools generates a counterexample the designer of the security protocol must fix the protocol manually. This task is burdensome and difficult because the designer has to find the specific flaw causing the problem in the protocol and patch it considering lots of possibilities. In addition, even experimented designers must take care because they could introduce more vulnerabilities. Therefore, we prompted the need of introducing an automated tool that given the specification of a protocol and one of its counterexamples can not only detect the root of the flaw in a protocol but also give a patch. To do this, we make use of some principles in order to guide the design in security protocols which will be studied in the next chapter.

Currently, there is a number of informal guidelines for strengthening the design and the robustness of security protocols. Abadi and Needham proposed the most important and well known principles in the literature, [1]. Abadi and Needham arrived at their principles by noticing common errors in the design of security protocols. Such errors make the protocols susceptible to an attack, as reported by some other researchers. If these errors are avoided, protocols tend to become more readable and, more importantly, correct.

This chapter has three major aims: first, to show common errors in the design of security protocols; second, to illustrate how the design principles of Abadi and Needham can guide us to find these errors; and third, to illustrate how these principles can guide us to patch faulty security protocols.

In the first section we review informal guidelines proposed in the literature. Then, in the second section we explain general aspects about Abadi and Needham’s principles and support why we have selected them. So, in the upcoming sections we analyze in detail such principles, namely: cypher-texts, agent names, nonces, timestamps, session keys and how message encoding should be carried out.
3.1 Informal Design Guidelines in the Literature

During the last few years, a number of protocol design principles have been proposed in the literature. These principles aim to guide the design of security protocols in order to make them simple and, hopefully, correct. The crux of the matter has been how to avoid replay attacks (a form of attack where a data transmission is repeated or delayed).

To avoid replay attacks, Carlsen [25] gave a full list of information that should be attached to cypher-texts (he called this list full information): protocol identifier, step identifier, message sub-components identifier, primitive type of data items and protocol run identifier. Protocol designers, however, find cumbersome including all this full information. Aura [10], considering performance issues, proposed to use a different cryptographic algorithm for each cyphered message in a protocol; to attach hash functions (including full information similar to Carlsen) in authenticator messages; to produce unique session keys using hash functions and to make no trust assumptions between principals. Using hash functions does not affect the performance in a protocol, but considering the use of a different cryptographic algorithm for each cyphered message is uncommon in the literature and it is not a design task.

A noteworthy work is Abadi and Needham’s principles [1], who suggested eleven design principles. These principles suggest good practices for strengthening the design of security protocols, which focus on two big issues: i) the messages involved in a protocol together with their content (principles 1, 3-10); and ii) the trust relations upon which the participants of the protocol depend (principles 2 and 11). Principles 3-9 recommend good practices in the meaning of a message such as agent names (principle 3), nonces (principles 6 and 7), timestamps (principle 8), session keys (principle 9) and cyphered messages (principles 4 and 5). Principle 10 emphasizes that the messages of a protocol should be distinguishable one another.

Anderson and Needham [5] extended Abadi and Needham’s principles 4 and 5 with respect to digital signatures and public key cryptography. Some of these new principles refer to cryptanalysis (which are not part of our research) and the remainder ones will be analyzed in the following sections together with those of Abadi and Needham, [1].

3.2 Abadi and Needham’s Principles to Guide the Design of Security Protocols

This thesis relies on good practices in the design of security protocols as proposed in Abadi and Needham’s principles. As described above these principles refer to two general aspects: those principles that are concerned with the content of a message and the other ones with external considerations to the formal description of the protocol. In the following two subsections we explain
precisely these two general aspects and describe what kind of principles can help us in order to patch faulty protocols and which ones cannot. Firstly we explain principles related to trust relations between the participants in a protocol and secondly we explain on the role of the protocol messages.

### 3.2.1 Trust relations

Principles related to trust relations are those about external considerations that the participants of the protocol should hold. Our work considers that the correctness of a protocol should be given only from its formal specification, without taking into account external considerations. Therefore, aspects like the ones mentioned below will be not considered:

- how session keys or nonces are generated; for example, whether it was generated by random numbers or large prime numbers, etc. In this context Aura, for example, proposed three strategies about how to generate session keys by hashing full information: strategies 3, 4 and 5, [10].

- how clocks are synchronized to establish message freshness. Abadi and Needham’s principle 8 is in part related to this issue.

- how trust is involved in the distribution of session keys. Abadi and Needham proposed in principles 2 and 11 that it does not suffice that the message content in a protocol says clearly what it means; also it is important that the protocol follows some norms, e.g. the Wide Mouthed Frog protocol, [24], has been criticized because a friendly agent is responsible from generating the session key to be distributed, rather than the server.

Although the design of security protocols sometimes depends on the aspects above mentioned, we chose not to take them into account because we make the following considerations: i) we work under the perfect cryptography assumption; ii) we assume that all computers’ clocks in the network are synchronized; and iii) trusted servers distribute session keys and friendly agents can forward secrets. Considering that a session key is a secret, so, friendly agents can also distribute session keys.

### 3.2.2 On the Role of a Message

In a protocol each message component should play a specific role. The role of a message component can sometimes be found via a syntactic analysis: a cyphered component carrying a session key is a key distributor. Also, a cyphered component used to provide evidence of the message
sender identity is an authenticator [70]. However, other times the role of some message components is hard to find out. A nonce, for example, can be used for one of several purposes, namely: to provide freshness, to provide authentication (by means of a challenge response), or to provide an abbreviation of the agents involved in the corresponding session (see section 3.5).

Abadi and Needham’s principle 1 emphasizes that, in a protocol message, each piece of information is important and plays a specific role:

Principle 1. *Every message should say what it means, the interpretation of the message should depend only on its content. It should be possible to write down a straightforward English sentence describing the content—though if there is a suitable formalism available that is good too.*

The role of a message, and of its components, should be inferable from the protocol only. Any agreement to be established between the participants should only be specified in the protocol messages. For example, if a protocol is intended for provide authentication, then the participants should conclude about each other’s identity using exclusively the information that they exchange together with their initial knowledge.

Principle 1 is a generalization of principles 3-9. So, in the following sections we describe the meaning of each message component, namely: cyphered texts, agent names, nonces, timestamps, session keys and recognizing messages.

### 3.3 Encryption

Perhaps *encryption* is the most important point in security protocols, hence, they are also named cryptographic protocols. In general, encryption may be used mainly for a variety of purposes: i) for the preservation of confidentiality; ii) for guaranteeing authenticity; iii) for producing random numbers [4], etc. Although in our work we make use of random numbers (e.g. nonces), it is not important how they are constructed. Therefore, we put special attention in the first two purposes of encoding: confidentiality and authentication. Before doing that, we recall a few results about message meaning under symmetric and asymmetric encryption, following the *free algebra assumption*. Under this assumption, two terms are equal if and only if they are syntactically equal.

**Symmetric cryptography** In symmetric cryptography, two (or more) agents, $A$ and $B$, share a single key, $K_{ab}$, used to encrypt and to decrypt each message. Then, proposition 1 and 2 hold.

**Proposition 1 (Authenticity)** *If only $A$ and $B$ know key $K_{ab}$, then anything that $A$ receives encrypted under $K_{ab}$ comes originally from $B$, as long as $A$ did not sent it.*
**Proposition 2 (Confidentiality)** The content of a message encrypted under a shared key remains secret if both it is not sent in clear and the shared key is not compromised.

**Asymmetric cryptography** Unlike symmetry cryptography, asymmetric cryptography requires two encryption keys, instead of one. Each agent is associated with two unique keys: one key is public, known by all agents in the network; and the other one is private, known only by the agent itself. Any message encrypted with one of these keys can be decrypted with the other one, and vice versa. In this kind of cryptography, two propositions hold:

**Proposition 3 (Authencity)** Any message encrypted under $K_b^-$ comes originally from $B$, unless $B$ is compromised.

**Proposition 4 (Confidentiality)** Any message encrypted under $K_a^+$ can be decrypted only by $A$ (if not compromised) using his private key $K_a^-$, therefore the message is intended for $A$.

Note that, these propositions hold iff all agents in the network initially know whom each public key belongs to. Anderson and Needham [5] proposed a principle: *Do not assume the secrecy of anybody else’s “secrets”* (except possibly those of a certification authority).

### 3.3.1 Confidentiality

Confidentiality is a property that a protocol must hold to ensure that information is not accessed by unauthorized agents. In asymmetric cryptography confidentiality may usually be obtained by public key encryption, see proposition 4, whereas in symmetric cryptography by simply encrypting a message with the associated key, see proposition 2.

Sometimes protocol designers believe that using only propositions 2 and 4 is enough to provide automatically guarantees of confidentiality, causing that sometimes messages are unnecessarily encrypted and some other times pieces of messages are omitted. Abadi and Needham noted that encryption sometimes is used incorrectly and thus proposed:

**Principle 4:** Be clear about why encryption is being done. Encryption is not wholly cheap, and not asking precisely why it is being done can lead to redundancy. Encryption is not synonymous with security, and its improper use can lead to errors.

For example, consider the Needham and Schroeder Shared Key (NSSK) protocol:

1. $A \rightarrow S : A; B; N_a$

2. $S \rightarrow A : \{\{N_a; B; K_{ab}; \{K_{ab}; A\}_K_B\}_K_B\}_K_A$
3. \(A \rightarrow B: \{K_{ab}; A\}_{K_B}\)

4. \(B \rightarrow A: \{N_b\}_{K_{ab}}\)

5. \(A \rightarrow B: \{N_b + 1\}_{K_{ab}}\)

This protocol has two aims: i) to distribute a session key \(K_{ab}\) to agents \(A\) and \(B\) through a trusted server and ii) to establish authentication of the participants. One may question why this protocol uses double encryption in message 2. Double encryption does not add anything else from the point of view of confidentiality. However, such a double encryption guarantees to \(B\) the participation of \(A\) (perhaps not recently), as only \(A\) could have decrypted the key distributor intended for \(B\) (in step 3). In this case, double encryption has been done with authentication purposes.

### 3.3.2 Authentication

A property very difficult to obtain in security protocols is authentication. There exist a lot of strategies to obtain this property such as challenge-response as suggested by Lowe [61] (e.g. the challenge-response of the protocol above, steps 4 and 5); signatures; or a collection of different tricks (e.g. double encryption as in the protocol above), etc. We study with more detail challenge-response in sections 3.5.3 and 3.6. In this subsection we concentrate exclusively on signatures.

Signatures are used, as the name suggest, to indicate which principal last encrypted a message. Signatures depend definitely on the kind of encryption key that the protocol uses. In symmetric cryptography, for example, it is not trivial to know which agent was the originator of a cyphertext because two or more agents are able to encrypt and decrypt a message with the same key, see proposition 1. In asymmetric cryptography it is apparently easier to know who has signed a message than in symmetric cryptography because it is obtained by encrypting a message using the agent’s private key, see proposition 3. The concept of digital signature usually is given for asymmetric cryptography, therefore, we give some assumptions in the use of this kind of cryptography.

It is frequently considered that the signer of a message knows its content, however, that may not be always the case (see below). Another incorrect assumption is to assume that an agent originates a message, only because a message that was encrypted for confidentiality purposes carries his signature, although such a message could be also a replay (see an example below). So, Abadi and Needham proposed the following principle:

**Principle 5:** *When a principal signs material that has already been encrypted, it should not be inferred that the principal knows the content of the message. On the other hand, it is proper to infer that the principal that signs a message and then encrypts it for privacy knows the content of the message.*
For example, in the following message:

\[ A \rightarrow B : \left\{ N_a; A; B; \{ M \}^{K_B^+}_{K_A^-} \right\}^{K_B^-}_{K_A^+} \]  \hspace{1cm} (3.1)

\( B \) knows from the signature that \( A \) is the originator one and that he knows the components \( N_a, A, B \) and \( \{ M \}^{K_B^+}_{K_B^-} \); but agent \( B \) may not be sure that agent \( A \) is actually aware of \( M \). This corresponds to a scenario where \( \{ M \}^{K_B^+}_{K_B^-} \) could have been a replay, as illustrated in the following example. Let us encrypt (3.1) for confidentiality purposes:

\[ A \rightarrow B : \left\{ \left\{ N_a; A; B; \{ M \}^{K_B^+}_{K_B^-} \right\}^{K_B^-}_{K_B^+} \right\}^{K_B^-}_{K_B^+} \]  \hspace{1cm} (3.2)

Upon reception, \( B \) knows that this message is intended for him since only he can decode it. Yet, \( B \) cannot assume that message \( \left\{ \left\{ N_a; A; B; \{ M \}^{K_B^+}_{K_B^-} \right\}^{K_B^-}_{K_B^+} \right\}^{K_B^-}_{K_B^+} \) was encrypted by \( A \), this issue requires another consideration, e.g. that \( N_a \) is part of a challenge response.

Many flaws have arisen for misunderstanding principle 5, see e.g. the CCITT.X.509 standard. The first message of one of these protocols contains an error similar to the message described in (3.1).

In general, to obtain confidentiality or authentication in a protocol it is necessary to apply a combination of encryption, fresh nonces and proper agent naming.

### 3.4 Agent names

An agent name is used for authentication purposes. Put differently, an agent name is used to pinpoint either who has sent a message or whom a message is intended for. For example, an authenticator should include which agent(s) is authenticating and a key distributor should include which agent(s) the key is intended for.

Protocol designers use basically two means to convey the name of an agent: i) directly, including the agent identity in plain text, e.g. \( A, B \); and ii) indirectly, using either encryption (see Propositions 1 and 3) or other piece of data (e.g. a nonce acting as a substitute for agent names). Substituting agent names with nonces may become a source of vulnerability (especially if it is usually used incorrectly (see section 3.5.1).

A common mistake of protocol designers is to consider that an agent name can be excluded from an encrypted message because it can be derived indirectly, when it cannot. Therefore, Abadi and Needham suggested principle 3 to avoid these kinds of errors:

Principle 3: *if the identity of a principal is essential to the meaning of a message, it is prudent to mention the principal’s name explicitly in the message.*
3.4.1 Name deduction in asymmetric cryptography

An example of incorrectly omitting an agent name in a key distributor is the following:

\[ A \rightarrow B : \left\{ T_a, A; \| K_{ab} \| K_A^{-} \right\}_{K_B^{+}} \] (3.3)

In this step, \( A \) sends a session key \( K_{ab} \) to \( B \). Upon reception, \( B \) notes that the message has two interesting bits: an outer cypher-text and an inner one. The outer cypher-text is encrypted under \( K_B^{+} \) and it is intended for provide confidentiality from its content (see proposition 4). Similarly, the inner cypher text, encrypted under \( K_A^{-} \), is intended for provide authentication of \( A \) (see proposition 3). Although \( B \) could trust the session key, no-one guarantees that the key distributor is not a forward. In fact, the key could be compromised as shown below.

Let \( B \) remove the outer encryption of (5.6), re-encode the message using \( K_C^{+} \) and then, impersonating \( A \), re-send it to some other agent, \( C \):

\[ B(A) \rightarrow C : \left\{ T_a, A; \| K_{ab} \| K_A^{-} \right\}_{K_C^{+}} \] (3.4)

where \( B(A) \) means that \( B \) impersonates \( A \).

\( C \) could assume that the message is intended for him and that it comes from \( A \), when in fact, it comes really from \( B \). So, as principle three recommends, in this case, it is important to include the name of \( B \) in the key distributor (inner cypher-text). Notice that it is not necessary to include the agent name \( A \) in the key distributor, since it can be obtained from the kind of cryptography used (see proposition 3):

\[ A \rightarrow B : \left\{ T_a, A, \{ B, K_{ab} \} \right\}_{K_B^{+}} \] (3.5)

We can deduce agent names in asymmetric cryptography but we must take into account principle 1 “the interpretation of the message should depend only on its content”. So, for deducing agent names in messages encrypted using public or private keys we must concentrate exclusively in the message under analysis without considering other ones, e.g. agent name \( B \) in the outer encryption and agent name \( A \) in the inner encryption.

3.4.2 Name deduction in symmetric cryptography

Another example of incorrectly omitting agent names in cypher-texts, but now for symmetric cryptography, is the following:

\[ A \rightarrow B : \{ N_a \}_{K_{ab}} \] (3.6)

Let \( A \) and \( B \) be non-compromised agents and \( K_{ab} \) be a non-compromised key shared between \( A \) and \( B \). In this step, \( A \) sends a secret \( N_a \) to \( B \). Upon reception, \( B \) notes that the message comes from \( A \), as long as he did not sent it (see proposition 1). By performance issues, it is considered
that agents in security protocols do not have memory, except for the current run of the protocol. So, for B it is very difficult to know if such a message was (or not) sent by him in any run before, as shown below.

Let C be a dishonest agent and resend the previous message to A as follows:

\[
C(B) \rightarrow A : \{N_a\}_{K_{ab}}
\] (3.7)

As you can see, A is now in the same position as B was previously because the flow of the message cannot be determined. Therefore, when a message is encrypted using symmetric key it is very important to include the agent names explicitly; an example would be to modify the cypher-text as follows:

\[
A \rightarrow B : \{[A; B; N_a]\}_{K_{ab}}
\] (3.8)

Note that agent names are important pieces to obtain confidentiality and authentication. Yet, designers ignore this principle. It is important to know under what conditions agent names can be deduced and when they cannot.

3.5 Freshness

Freshness is an important property that protocol messages must meet so that protocol runs can be distinguished one another. Many protocols are flawed because they lack this property. Nonces and timestamps are often used to provide this property.

3.5.1 Nonces

A nonce serves almost always as a fresh identifier. It is used, among others things, to express that the message to which it is attached should be considered fresh too. A nonce is also used to identify runs, so no agent should re-use an old nonce for a new run.

In general, a nonce may be used for three purposes: i) to say that a message is fresh or that a run of a protocol is fresh (session identifier); ii) to elaborate a challenge response; and iii) to substitute agent names. Sometimes the third purpose involves the second one and the second purpose involves the first one. Often, a protocol flaw has to do with a designer mixing these purposes. For example, in a protocol step a nonce could originally be used in a challenge-response and in a subsequent step it could be erroneously used as a substitute of agent names. Sometimes designers assume that a nonce provides more roles than it really does. For instance, consider the Otway and Rees protocol:

1. \[
A \rightarrow B : \{N_a; M; A; B\}_{K_A}
\]

2. \[
B \rightarrow S : \{N_a; M; A; B\}_{K_A} : \{N_b; M; A; B\}_{K_B}
\]
3. $S \rightarrow B : M; \{N_a; K_{ab}\}_{K_A}; \{N_b; K_{ab}\}_{K_B}$

4. $B \rightarrow A : M; \{N_a; K_{ab}\}_{K_A}$

Burrows, Abadi and Needham incorrectly considered unnecessary the encryption of nonce $N_b$ in step 2. Thus they proposed to take it out arriving at the protocol presented in page 22. As we already showed there, that protocol is faulty: the encryption of $N_a$ and $N_b$ in step 2 are essential. $N_a$ associates agent names $A$ and $B$, and thus, $N_a$ may serve as a secure reference to $A$ and $B$ in steps 3 and 4. However, $N_b$, being in plain-text at step 2, cannot substitute to $A$ and $B$. Now, the flaw presented in section 2.4.2 should be clearer. Abadi and Needham overlooked this consideration in [24] but suggested:

**Principle 6:** Be clear what properties you are assuming about nonces. What may do for ensuring temporal succession may not do for ensuring association and perhaps association is best established by other means.

Temporal succession refers to assure that a message is sent after some other. For example, if a principal generates a nonce for the current protocol run and receives messages that contain it, this principal may deduce that these messages have been created after the nonce was generated. Nonce association refers to a nonce acting as a substitute of one or more agent names. Deducing agent names from a nonce acting as a substitute of agent names involves an inductive analysis of the protocol, rather than a syntactic analysis. Furthermore, this inductive analysis would be a verification task.

### 3.5.2 Predictable numbers

Principle 7 is about the use of a predictable number such as the value of a counter. Many attacks can arise when a number (used like a nonce) is predictable because a spy could guess the number that an agent sends. Thus, this principle suggests:

**Principle 7:** The use of a predictable quantity (such as the value of a counter) can serve in guaranteeing newness, through a challenge-response exchange. But if a predictable quantity is to be effective, it should be protected so that an intruder cannot simulate a challenge and later replay a response.

A counter, in spite of being predictable, can be used as part of a challenge-response. For example, in steps 4 and 5 of the Needham and Schroeder Shared Key distribution protocol agents $A$ and $B$ carry out a challenge-response (see section 3.3.1). In step 4, agent $B$ generates $N_b$ and sends a challenge composed by a cypher-text $\{N_b\}_{K_{ab}}$, where $K_{ab}$ is a session key distributed in previous steps to be shared between $A$ and $B$. Upon reception, $A$ responds the challenge with the
same ingredients besides of using a counter. The counter aims to break the ambiguity between both messages. Note that, for establishing successfully the challenge-response it is necessary that some conditions hold. Firstly, the session key must be secure between the participants; secondly, $N_a$ must be fresh; and thirdly, the messages involved in the challenge-response must be unambiguous.

3.5.3 Timestamps

Time plays an important role in many security protocols for guaranteeing freshness. A time-stamp might be used for this purpose and it might consist of the time of the day recorded in a transaction. The current time is usually maintained by a host and is used for a variety of synchronization purposes. Instead of the time of the day, a timestamp may also be a time relative to a starting point.

Often a timestamp is used as a kind of a nonce or as a lifetime. When a timestamp is used as a kind of a nonce it does not depend on clock synchronization at all (see subsection 3.5.1), but we need to take care because timestamps may be predictable. When a timestamp is used like a lifetime it really depends on clock synchronization. For example, an agent that sees a message containing a timestamp, can accept the message only if the timestamp is within a reasonable interval of the agent’s local time. If the agent’s local time is not synchronized with the rest of the participants of the protocol, then the protocol may be subject of various kinds of attacks. Hence, Abadi and Needham propose:

Principle 8: If timestamps are used as freshness guarantees by reference to absolute time, then the difference between local clocks at various machines must be much less than the allowable age of a message deemed to be valid. Furthermore, the time maintenance mechanism everywhere becomes part of the trusted computing base.

Abadi and Needham point out that many flaws have arisen because local time of computers is not updated. For example, the Kerberos protocol uses timestamps as freshness. The protocol is flawless if the network has synchronized clocks. Gong shows a scenario where this protocol is attacked when clocks are not synchronized [44].

Following the previous argument, a message containing a timestamp will be considered like recent, but not fresh precisely. For example, an agent $B$ receiving message, $A \rightarrow B : \{T_a, M\}^{K_{ab}}$, can prove that such cypher-text is formed recently if $|T_b - T_a| < \Delta$, where $T_b$ is a timestamp generated by $B$ in the time he receives the message and $\Delta$ is an allowed interval. However, to prove that the message is fresh requires other means since the message could have been forwarded as many times as $\Delta$ had not expired. For example, consider the following Denning and Sacco protocol [37]:

1. $A \rightarrow S : A; B$
2. \( S \rightarrow A : \{T_s; B; K_{ab}; \{T_s, K_{ab}; A\}_K_B\}_K_A \)

3. \( A \rightarrow B : \{T_s, K_{ab}; A\}_K_B \)

This protocol contains a number of attacks. The one corresponding to our study consists in that the last message could be forwarded to \( B \) as many times as \( T_s \) does not expire, whereas \( B \) could accept the same number of sessions as messages received.

When using timestamps designers frequently overlook other properties that should be taken into account like temporal succession. Another example of mal-use of timestamps, among other things, is the Wide Mouthed Frog (WMF) Protocol causing temporal succession problems, shown in section 3.7.

### 3.6 Session keys

A session key is usually distributed by a trusted server through a key distributor in order to be shared between two or more agents. The use of a session key is temporal, this is to avoid that a spy can guess the key in a sufficiently long period of time. So, a new session key is distributed in each run of a protocol. Abadi and Needham noted that some flaws in protocols originate because key distributors sometimes do not provide freshness. An example of this is the Needham and Schroeder key distribution protocol:

1. \( A \rightarrow S : A; B; N_a \)

2. \( S \rightarrow A : \{N_a; B; K_A; \{K_{ab}; A\}_K_B\}_K_A \)

3. \( A \rightarrow B : \{K_{ab}; A\}_K_B \)

4. \( B \rightarrow A : \{N_b\}_K_{ab} \)

5. \( A \rightarrow B : \{N_b + 1\}_K_{ab} \)

Note that, when \( A \) receives key distributor (step 2) she may consider that \( K_{ab} \) is fresh due to \( N_a \). However, \( B \) cannot make the same consideration because in key distributor there is no evidence of freshness. Therefore, \( B \) establishes, in the remainder steps, a challenge-response with \( A \) to assure her recent participation and so trust in \( K_{ab} \). Yet, \( K_{ab} \) could be compromised and \( \{K_{ab}, A\}_K_B \) could have been a replay of an old run of the protocol and then \( B \) could erroneously think that \( A \) wishes communication with him and to accept a session key that could really come from an intruder. In
Alice and Bob notation:

\[ K_{ab} \text{ is compromised.} \]

3. \( \text{Spy}(A) \rightarrow B : \{K_{ab}, A\}_B \)
4. \( B \rightarrow \text{Spy}(A) : \{N_b\}_K_{ab} \)
5. \( \text{Spy}(A) \rightarrow B : \{N_b + 1\}_K_{ab} \)

To fix this flaw Lowe proposed to include a time-stamp in the key distributor intended for \( B \) [61], steps 2 and 3, as follows:

2. \( S \rightarrow A : \{N_a, B, K_{ab}, \{K_{ab}, A, [T_s]\}_K_B\}_K_A \)
3. \( A \rightarrow B : \{K_{ab}, A, [T_s]\}_K_B \)

For these kinds of flaws Abadi and Needham suggested:

**Principle 9:** A key may have been used recently, for example to encrypt a nonce, yet be quite old, and possibly compromised. Recent use does not make the key look any better than it would otherwise.

From this principle it follows that we should be careful on the use of session keys and avoid confusing recent use with freshness; before using a session key as a part of a challenge-response we must guarantee freshness in key distributions. Freshness usually is obtained by including either timestamps (see subsection 3.5.3) or a nonce challenge response (see subsection 3.5.1). Many protocols establish the challenge-response after receiving a session key (as the protocol above). However, sometimes it is better to establish the challenge before distributing the session key [72].

### 3.7 Recognizing messages

Sometimes, in computer networks there exists a lot of security protocols running in parallel with different purposes. Often an agent may take part of various runs, in this case the agent must recognize one protocol message from other different message. When this circumstance is not possible to get, it is possible that the meaning of the message is not sufficient and then it should be either completed or modified.

Abadi and Needham described that two possible forms of confusion may arise when a protocol is poorly defined: i) between the current message and a message of similar purpose from a previous run of the protocol (like the Wide Mouthed Frog protocol shown below); and ii) between the current message and a message belonging either elsewhere in the protocol or to another protocol. For example, the kerberos protocol and the Needham and Shroeder symmetric key protocol are similar in structure and aim to achieve similar security goals, thus some messages could be confused.
Other kinds of flaws have arisen due to type flaw attacks. Catherine Meadows, for example, has shown that type confusions are possible between two or more messages with different purposes. A type flaw attack is an attack where a principal accepts a message component of one type as a message of another. It relies on a principal’s inability to separate two messages of different steps [69].

Abadi and Needham propose that agents must recognize messages and the kind of encoding being used:

\[ \text{Principle 10: If an encoding is used to present the meaning of a message, then it should be possible to tell which is being used. In the common case where the encoding is protocol dependent, it should be possible to deduce that the message belongs to this protocol, and in fact to a particular run of the protocol, and to know its number in the protocol.} \]

To illustrate this principle we present the Wide Mouthed Frog (WMF) protocol [24]:

1. \( A \to S : A; \{B; T_a; K_{ab}\}_K \)

2. \( S \to B : \{A; T_s; K_{ab}\}_K \)

This protocol aims to distribute a session key between two agents. In step 1, when server \( S \) receives the message, he sees something like: “\( A, X \)” and assumes that \( X \) has been cyphered using the key shared with agent \( A \) because of the joined message in plain-text, \( A \). When the server decodes \( X \), he will see whether the result is consistent (integrity). Now, let us assume the message of step 1 without the agent name \( A \), in such a case, the server could not know which decoding should be used. By contrast in step 2 a plain-text \( B \) is not necessary, because \( B \) acting as a responder knows that the message received, step 2, could only have been sent by the server.

Now, key distributors \( \{B; T_a; K_{ab}\}_K \) and \( \{A; T_s; K_{ab}\}_K \) (steps 1 and 2) violate principle 10 because they are identical in structure. The Server, for example, could not identify if \( \{A; T_s; K_{ab}\}_K \) is used concurrently as the first step in a new run of the protocol and to be subject of a replay as shown in Figure 3.1. As can be seen, instances of the key distributors are forwarded in different sessions.

Many protocols are faulty because an agent cannot recognize whether a message belongs to a specific step or a specific run of a protocol. Because protocol verification researchers have not focused their efforts to the verification of two or more protocols in a parallel way, in our work we have not considered attacks between two or more protocols running in parallel. So we leave it as a further work.
3.8 Conclusions

As we have seen in this chapter, many informal design guidelines have been proposed in the literature in order to strengthen the design of security protocols. Designers find some of these guidelines cumbersome; others are rather directed to implementation issues; and finally those that rely on good practices on the meaning of each message in the protocol such as some of the Abadi and Needham’s principles.

Abadi and Needham noted some common features in faulty protocols that designers frequently carry out. So, they proposed eleven design principles with the hope of improving the practice of designing cryptographic protocols. Some of these principles focus on the meaning of the messages in a protocol and the remainder ones are related with trust relations. These principles are usually presented in form of warning examples of how protocols should not be built. Unfortunately, there are not specific (mechanized) instructions for fixing protocols once we know that the protocol is faulty. For a protocol designer it is very difficult to have in mind all of these principles, because of that, designers still make mistakes.

To automate the correction of faulty security protocols we rely on Abadi and Needham’s principles focused on the meaning of the messages in a protocol because these principles let us determine when a message could be modeled incorrectly. In the next chapter we describe how SHRIMP works and how it will help in the development process of security protocols.
4. **SHRIMP: Outline**

Although there exist informal design guidelines and formal development support, the development of security protocols is still error prone and so is very time-consuming.

In this chapter, we give an overview of SHRIMP, a mechanism that aims to speed up the development cycle of security protocols by adding automated aid for protocol diagnosis and repair. These features are all of interest because nowadays protocols are more complicated than just 3—5 steps (e.g. the SET protocol) and their various parts are intertwined, making it hard for a human to cope with all the subtle dependencies.

For conveying to the solution of our research, we first describe the class of protocol flaw that SHRIMP may deal with. We will also describe how this class exploits a violation of a good practice for protocol design, as stated by Abadi and Needham. Next, we present the role of SHRIMP in the development process for security protocols. Then, we elaborate our framework for patching security protocols: how it works, how new patching methods could be added, etc. Finally, we describe the logic upon which SHRIMP has been constructed.

4.1 **A Large Class of Protocol Flaw**

This thesis aims at proposing a general framework for the correction of security protocols capable of dealing with a general class of flaws. To this end, we have chosen to address protocols that are susceptible to a replay attack. A *replay attack* is a form of network attack in which a valid data
transmission is maliciously or fraudulently repeated or delayed.\textsuperscript{1} The rational behind this decision originates from the fact that many faulty protocols, as output by the experiments documented in both libraries Clark and Jacob’s and AVISPA’s (see Chapter 2), are actually susceptible to this kind of attack. In what follows, we describe replay attacks in a more elaborate level of detail using Syverson’s taxonomy. We shall see that any kind of replay attack exploits a violation of a good practice for protocol design, as stated in particular by Abadi and Needham.

4.1.1 Replay Attacks

Paul Syverson’s taxonomy to replay attacks is a simple means to understand how an intruder is able to reuse a message, usually a cypher-text, in order to make the protocol fail to provide a security requirement. Syverson suggests to classify a replay attack mainly in terms of two features (as illustrated in table 4.1):

- the origin of the replayed message (the replay for short): is it originated within the same protocol run (run internal attack) or from another one (run external attack)?; and

- the recipient of the replay: is it the original, intended recipient (straight replay), has the replay been deflected to some other party (deflected attack) or has the replay been sent back to sender (reflected attack)?.

Syverson’s taxonomy is also of interest to us as it has been argued to be complete [84], in that it can capture any replay attack.

<table>
<thead>
<tr>
<th></th>
<th>External</th>
<th>Internal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Interleaving</td>
<td>Classic</td>
</tr>
<tr>
<td>Reflexion</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Deflexion</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Straight</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4.1: Paul Syverson’s taxonomy of replay attacks

Note that an interleaving external replay attack requires contemporaneous protocol runs while a classic external replay attack does not.

From Syverson’s taxonomy, we will demonstrate that whenever a protocol is susceptible to a replay attack, then the protocol does not comply to one or more of Abadi and Needham’s design

\textsuperscript{1}http://en.wikipedia.org/wiki/Replay_attack
principles. In particular, the protocol violates one or more of principles 3—10, related to message meaning. In what follows, we describe these violations according to Syverson’s taxonomy:

- **Reflected and Deflected replay attack**: implies that the originator/recipient of a cypher-text in one message of the protocol cannot be distinguished. This flaw violates principle 3, *agent naming*, which prescribes that the agent names relevant for a message should all be derivable either from the format of a message or from its content;

- **Deflected replay attack**: implies that two or more different cypher-texts of the same protocol cannot be distinguished from each other. This flaw violates principle 10, *recognizing messages and encodings*, which prescribes to be careful about the format of a message: principals should be able to associate which step or which run a message corresponds to, regardless of whatever protocol they are running;

- **straight replay**: implies that two or more different runs of the same protocol cannot be distinguished from one another. Upon reception, a participant cannot separate which run the message belongs to. This flaw violates principle 10, *the message cannot be bound to a particular run of the protocol*. It also violates principles 6—8, *the protocol does not guarantee association or temporal succession*.

### 4.2 SHRIMP for Security Protocol Development

As discussed in Chapter 2, the software development cycle involves requirement analysis, specification, design, construction, testing and maintenance. Formal methods have been extensively used for software specification and verification. Yet, the application of formal methods to the maintenance of software had been largely ignored, security protocols not being an exception.

In the formal methods approach to software development, both the protocol and its specification are first formalized using a suitable, but not necessarily the same, logic. Then, mathematical reasoning is used to prove that the security protocol satisfies the specification, the so called *verification theorem*. Security protocol verification is tool supported. A verification tool would either prove or disprove the verification theorem. If the protocol is faulty, a state-of-the-art tool would output a counterexample, a script denoting an interleaving of protocol runs that illustrate that the protocol violates the specification.

This thesis introduces SHRIMP, a mechanism that aims at identifying and patching faulty security protocols. SHRIMP is a two-step approach. In a first step, SHRIMP finds the flaw in the protocol. Then, in a second step it suggests changes to the specification of the protocol. These changes yield an improved version of the protocol, which should be verified again (using the verification tool) in order to know whether the security protocol is free of flaws, or in the worst case,
to be no longer susceptible to the same security property. If the protocol is still faulty, then we will get another attack from the verification tool and we would repeat the process as many times as necessary. Figure 4.1 illustrates the inclusion of SHRIMP in the development process for security protocols.

To identify a protocol flaw, we analyze the protocol specification and its attack to determine if the attack shows common patterns to a specific flaw. We check what design principles are being violated and we follow the recommendations of such design principles in order to patch the faulty protocol. To do that, we translated some of the informal principles of Abadi and Needham [1] into formal requirements, see chapter 6. For each requirement, there is a collection of rules that represent violations of one or more design principles. If the faulty protocol and its attack satisfy a requirement, then the rules identify the ciphertext(s) that causes the problem in the protocol and using also several of these rules we must include some elements in the protocol to avoid the flaw (patching the protocol).

As a general framework to organize the application of such rules, we have adopted the proof planning methodology [22], developed to automate inductive theorem proving. We explain this methodology in the following section.

4.3 Patch Planning Faulty Security Protocols

4.3.1 Proof Planning Methodology for Patch Planning

Proof planning [22] is a meta-level reasoning technique, developed especially as a search control engine to automate theorem proving. A proof plan expresses the patterns of reasoning that must be
followed to search proof in theorems. We use proof planning methodology to search for suitable patches in faulty security protocols. In proof planning, *methods* are the basic building blocks that make up proof-plans. Patch planning is not the exception and so, SHRIMP is given as a set of patch methods capable of dealing with a general class of faults.

**Patch methods** A *patch method* consists of a 5-tuple, involving name, input, preconditions, patch and effect. The first element is the *name* of the method; the second component is the *input*, often the description of a faulty protocol and an attack; the third component is the *preconditions*, a formula written in a meta-logic that the input objects must satisfy; the fourth component is the *patch*, a procedure specifying how to mend the input protocol. The fifth component is the *effects*, a formula specifying required properties of the newer version of the protocol. SHRIMP uses the preconditions to predict whether the associated patch will make the protocol no longer susceptible to the attack; Figure 4.2 shows the skeleton of a patching method.

<table>
<thead>
<tr>
<th>Name:</th>
<th>% Name of the method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>% Usually a protocol description and an attack</td>
</tr>
<tr>
<td>Preconditions:</td>
<td></td>
</tr>
<tr>
<td>% Set of clauses characterizing a protocol flaw in terms of properties of the input objects.</td>
<td></td>
</tr>
<tr>
<td>Patch:</td>
<td></td>
</tr>
<tr>
<td>% A procedure that modifies the structure of the input protocol to eliminate the flaw captured by the preconditions.</td>
<td></td>
</tr>
<tr>
<td>Effect:</td>
<td></td>
</tr>
<tr>
<td>% A formula specifying new properties that the modified protocol must satisfy.</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.2: The skeleton of a patching method

Both preconditions and patch rest upon Abadi and Needham’s principles, used as a meta-level guidance for fault location, identification and repair. Effects involve two tests: one verifying that the attack is no longer possible and the other that the message formats in the new protocol agree from one step to the next.

**Compound methods** Methods can be compound by invoking other methods using methods (functions that link methods together to control search). A *compound method* is a 4-tuple
(name, input, preconditions, method). It involves the name of the compound method, the input, the preconditions, and then the method(s) that can be called, linked by methodicals. Methods are usually attempted in a sequential way using the `orelse meth` methodical. The statement `orelse meth meth₁ meth₂` attempts the application of method `meth₁` and if that fails then attempts the application of `meth₂`. Notice that if `meth₁` is always applicable, `meth₂` will never be tried. Figure 4.3 shows the skeleton of a compound method.

<table>
<thead>
<tr>
<th>Name: % Name of the method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: % Usually a protocol description and its attack</td>
</tr>
<tr>
<td>Preconditions: % Set of clauses characterizing a protocol flaw in terms of properties of the input objects.</td>
</tr>
<tr>
<td>Method: % Method invocation</td>
</tr>
<tr>
<td><code>orelse meth meth₁(Parameters) meth₂(Parameters)</code></td>
</tr>
</tbody>
</table>

Figure 4.3: The skeleton of a compound method

### 4.3.2 Patch Method Application

A patch method is said to be *applicable* if the method preconditions hold. A method is attempted as follows:

1. First the input objects are unified with the method parameters;
2. The *preconditions* are evaluated; and if the preconditions hold
3. The *patch* is applied to the input protocol description.

Thus, the preconditions will never be evaluated without first unifying the input parameters. Similarly, the patch will never be applied if the preconditions do not hold.

**Patch Planning** Often more than one method is applicable to given faulty security protocol. This results in different patching suggestions, some might not be the more adequate ones. Because of that, the order how methods are attempted is very important. As might have been noticed by now, `orelse meth` naturally implements a *depth-first* search strategy. Depth-first search pursues choices in the order of appearance. If no method is applicable, SHRIMP will terminate, reporting failure.
So far, our experiments have been conducted using only depth-first. However, the more patch methods are added, the more it becomes necessary to explore other strategies, in particular best-fit search.

### 4.3.3 Criteria to Evaluate Patch Methods

While developing patch methods, one has to be careful not to create specialized methods, capable of dealing only with a limited class of flaws. This suggests criteria to assess a patch method, which in our case are as follows:

- **Usefulness**: the methods must find the flaws in a protocol and patch it successfully.
- **Generality**: the methods must succeed in a large number of cases.
- **Predictability**: the methods must patch all faulty protocols captured by the preconditions of each method.

Other criteria might be considered. Some might be more arguable, such as protocol readability, protocol complexity, and so on. Here, however, we will use usefulness, generality and predictability only.

### 4.3.4 The Logic of the Patching Methods

SHRIMP comes with a set of functions which allow us to identify flaws in a protocol (which are evidenced in the attack and in the protocol). Many of these functions have to do with message similarity, missing agent names identifiers, lacks of freshness or uniqueness in the messages of a protocol. These functions also allow us to change the structure of a protocol message and to propagate such changes in all messages of the protocol where the message appears in such a way that the protocol is executable after changes.

Although SHRIMP’s constructs might be accommodated within any logic (e.g. Strand-Space model [86] or Paulson’s logic [74]) for the formulation of the preconditions, patch and effect, we have chosen Strand-spaces because this logic let us express suitably SHRIMP’s formalizations. In addition, many formalizations that we will see throughout this work can theoretically be justified in this logic.

### 4.4 Strand Spaces: The logic behind Shrimp

This section is a review of the standard notations developed by Thayer, Gerzog and Guttman [86]. Firstly, we describe the concept of *terms* that are used to represent the messages of a protocol. Next,
we describe the concepts of strand, strand-spaces and bundles, which will permit us to represent runs of a protocol and its counterexamples.

4.4.1 Message terms

**Notation** We denote meta-variables (i.e. variables ranging over message terms) with lower-case sans serif, e.g. \(m_1; m_2\), \(\{m\}_k\), etc., and sets of terms with upper-case sans serif, e.g. \(K, M, \ldots\).

Let \(M\) be a set of terms that are the possible messages that can be exchanged between principals in a protocol. We will refer to the elements of \(M\) as terms, ranged over \(m_1, m_2, \ldots\). The set of terms, \(M\), is freely generated from two disjoint sets, the set of texts \((A)\) and the set of keys \((K)\), by means of concatenation, \(m_1; m_2\), and encryption, \(\{m\}_k\), where \(k \in K\). \(A\) contains agent names \(A, B, \ldots\), nonces \(N_a, N_b, \ldots\), time-stamps \(T_a, T_b, \ldots\) and session keys \(K_{ab}, K_{bc}, \ldots\). In the set of keys \((K)\) there are two functions, one maps principals, \(A, B, \ldots\), to their public keys, \(K^+_A, K^+_B, \ldots\), and the other a pair of principals, \((A, B)\), to their symmetric shared key, \(K_{ab}\). \(K\) comes with an inverse operator mapping each member of a key pair for an asymmetric cryptosystem to the other, \((K^+_A)^{-1} = K^-_A\), and each symmetric key to itself, \((K_{ab})^{-1} = K_{ab}\). Let \(\text{Safe}\) denote the set of keys that are safe and \(\text{Safe}_A\) denote the set of keys known by a regular, non-compromised agent \(A\).

**Sub-term relation** A term \(m_1\) is a sub-term of term \(m_2\) if \(m_1\) appears in \(m_2\). The sub-term relation, \(\sqsubseteq\), is the least relation such that:

- \(m \sqsubseteq m\), if \(m \in M\);
- \(m \sqsubseteq \{m_1\}_k\), if \(m \sqsubseteq m_1\)
- \(m \sqsubseteq m_1; m_2\) if \(m \sqsubseteq m_1 \lor m \sqsubseteq m_2\)

Notice that, for any \(k \in K\), \(k \sqsubseteq \{m\}_k\) only if \(k \sqsubseteq m\).

A message is a component of another one, denoted as \(m_1 \sqsubseteq m_2\), if \(m_1 \sqsubseteq m_2\), \(m_1\) is not a concatenated term, and every \(m' \neq m_1\) such that \(m_1 \sqsubseteq m' \sqsubseteq m_2\) is a concatenated term. Components are either atomic values or cypher-texts.

This model uses the free encryption assumption which states that: \(\{m\}_k = \{m'\}_{k'}\) iff \(m = m'\) and \(k = k'\).

4.4.2 Events, Strands and Strand Spaces

**Events** An event is a communicating action where the transmission of a term \(m\) is denoted as \(+m\) and the reception of the same term is denoted as \(-m\). Events are denoted as signed terms...
+m and −m. The set of signed terms is denoted as ±M, and the set of finite sequences of signed terms is (±M)*.

**Strands**  A strand is a sequence of events carried out by a participant. In terms of graph a strand is a sequence of nodes, each one denotes the step of a protocol and a communicating event (so a node is either positive (+) or negative (−)). Figure 4.4 illustrates initiator and responder strands of the NSPK protocol.

\[
\text{Initiator} = \langle +, \{N_a; A\}_k^+, \rangle, \quad \text{Responder} = \langle -, \{N_a; A\}_k^+, \rangle,
\]
\[
\langle -, \{N_a; N_b\}_k^+ \rangle, \quad \langle +, \{N_a; N_b\}_k^+ \rangle,
\]
\[
\langle +, \{N_b\}_k^+ \rangle \quad \langle -, \{N_b\}_k^+ \rangle
\]

Figure 4.4: Strands of NSPK protocol

**Strand Spaces**  A strand space is a graph formed by a set of strands \( \Sigma \) with a trace mapping \( \text{tr} : \Sigma \rightarrow (\pm M)^* \):

1. A node is a pair \( (s, i) \), with \( s \in \Sigma \) and \( i \) an integer satisfying \( 1 \leq i \leq \text{length}(\text{tr}(s)) \). The set of nodes is denoted by \( \mathcal{N} \). The node \( (s, i) \) belongs to strand \( s \). Every node belongs to a unique strand.

2. If \( n = (\alpha m) = (s, i) \in \mathcal{N} \), where \( \alpha \in \{+,-\} \), then \( \text{index}(n) = i; \; \text{strand}(n) = s; \; \text{msg}(n) = m \) and \( \text{sign}(n) = \alpha \).

3. If \( n_1, n_2 \in \mathcal{N} \), then \( n_1 \rightarrow n_2 \) means that \( n_1 = +m \) and \( n_2 = -m \) for some message \( m \). This represents sending a message \( m \) from \( n_1 \) to \( n_2 \).

4. If \( n_1, n_2 \in \mathcal{N} \), then \( n_1 \Rightarrow n_2 \) means that \( n_1 \) and \( n_2 \) occur on the same strand with \( \text{index}(n_2) = \text{index}(n_1) + 1 \). This represents an event \( n_1 \) immediately followed by \( n_2 \). \( \Rightarrow^+ \) and \( \Rightarrow^\ast \) are used to respectively denote the transitive and the transitive-reflexive closure of \( \Rightarrow \).

5. A term \( m \) originates at a node \( n \in \mathcal{N} \) iff \( \text{sign}(n) = +; \; m \sqsubseteq \text{msg}(n); \) and \( m \not\sqsubseteq \text{msg}(n') \), for all \( n' \Rightarrow^+ n \).

6. \( m \) is said to be uniquely originating if it originates on only one node in the strand space. \( \text{unique}_s \) is the set of terms uniquely generated at strand \( s \).
A strand space $\Sigma$ is a set of strands, where $\Rightarrow$ and $\rightarrow$ impose a graph structure on the nodes of $\Sigma$. Each strand represents a protocol run from the local perspective of a participant. If the participant is honest, the strand, as well as the strand nodes, is said to be regular and penetrator otherwise.

Causal Precedence Let $\Sigma$ be a strand space. For nodes $n_1, n_2 \in \Sigma$, define $n_1 \preceq \Sigma n_2$ iff there is a sequence of zero or more edges of $\Rightarrow$ and $\rightarrow$ leading from $n_1$ to $n_2$ in $\Sigma$. The relation $\preceq \Sigma$ expresses a causal precedence. In other words, $\prec \Sigma$ and $\preceq \Sigma$ denote respectively the transitive and the transitive-reflexive closure of $(\Rightarrow \cup \rightarrow)$.

4.4.3 Bundles

Bundles A bundle represents a protocol run under some configuration. Let $\beta = \langle \mathcal{N}, (\rightarrow \cup \Rightarrow) \rangle$. $\beta$ is a bundle if:

1. $\beta$ is non-empty, finite and acyclic graph;
2. For every $n_2 \in \mathcal{N}$, if $\text{sign}(n_2) = -$ , then there is a unique $n_1 \in \mathcal{N}$ with $n_1 \rightarrow n_2$;
3. If $n_1 \Rightarrow n_2$ then $n_1 \in \mathcal{N}$, $n_2 \in \mathcal{N}$ and $n_1 = \langle s, i \rangle$ and $n_2 = \langle s, i + 1 \rangle$.

Let $\mathcal{B}$ be the set of all bundles and $\beta \in \mathcal{B}$, then $\prec$ and $\preceq$ are partial ordering. A legal run of a protocol forms a regular bundle (a run with only honest participants) and a penetrator bundle otherwise.

Figure 4.5 illustrates a regular bundle of the NSPK protocol and Figure 4.6 illustrates a penetrator bundle (a counterexample).

---

Figure 4.5: A Regular Bundle of the NSPK Protocol
4.4.4 Authentication Test

Sometimes when a protocol lacks some security properties such as secrecy or authentication, often its original problem corresponds to authentication issues. To fix this problem, sometimes we will need to change the structure of a protocol to repair the flaw causing the problem. In such a case, we will need to prove that our changes are valid. Particularly, when our changes fix a time reference problem, we will use an authentication test [45], which was introduced initially to verify security protocols, but also has been used to guide protocol development. Authentication test is a mechanism that allows one to infer that some principal possessing relevant key material has received and transformed a message carrying a distinguished value.

The crux of an authentication test is as follows: suppose an agent in a security protocol generates and sends a message containing a new nonce $N$, and later receives $N$ again in a different cryptographic context. The agent can conclude that some agent possessing the specific key has received and transformed the message under which $N$ was emitted. If the key is safe, then this agent cannot be the penetrator, but instead must be a regular agent. Mainly there are two kinds of authentication test: i) an outgoing test is one where the new value $N$ is transmitted in encrypted form, and only a regular participant can extract it from that form; ii) an incoming test is one in which $N$ is received back in encrypted form, and only a regular participant can put it in that form. Next, we explain these concepts in detail:

Figure 4.6: A Penetrator Bundle on the NSPK Protocol
**Outgoing test:** Suppose that $A$ is a participant in a protocol. Suppose that at node $n_0$ $A$ creates a new term $t$, builds $m = \{m'\}^k$, such that $t \sqsupseteq m$ and $m$ is a component of $\text{msg}(n_0)$, and then transmits $\text{msg}(n_0)$. Suppose that $t$ is uniquely generated, that $m$ is not a subterm of a component of any regular node in the protocol and that the decryption key is safe, $k^{-1} \in \text{Safe}$. If $t$ is later received, at node $n_1$, outside the form $\{m'\}^k$, then only a honest participant, not the penetrator, must have been responsible for $t$ to have gone out of this form. The edge $n_0 \Rightarrow^+ n_1$ is an outgoing test for $t$ in $m$.

**Incoming test:** If, instead, $t$ is sent possibly in clear and it later is received in encrypted form $\{\ldots; t; \ldots\}^k'$, where $k' \in \text{Safe}$, then only a honest participant, not the penetrator, must have been responsible for $t$ to have entered to this form. The edge $n_0 \Rightarrow^+ n_1$ is an incoming test for $t$ in $\{\ldots; t; \ldots\}^k'$.

In what follows we define formally the two authentication tests introduced in [45]. First, we introduce more auxiliary notions, also taken from [45].

Let $\Sigma$ be a strand space. The message term $m = \{t\}^k$ is a test component for a term $a$ in a node $n$ if:

- $a \subseteq m$ and $m$ is a component of $n$; and
- $m$ is not a proper subterm of a component of any regular node $n' \in \Sigma$.

The edge $n \Rightarrow^+ n'$ is a transformed edge for a term $a$ if $n$ is positive and $n'$ is negative (respectively, if $n$ is negative and $n$ is positive), $a \subseteq \text{msg}(n)$, and there is a new component $m'$ of $n'$ such that $a \subseteq m'$. The edge $n \Rightarrow^+ n'$ is a test for $a$ if $a$ uniquely originates at $n$ and $n \Rightarrow^+ n'$ is a transformed edge for $a$.

**Definition 1 (Outgoing test)** The edge $n_0 \Rightarrow^+ n_1$ is an outgoing test for $a$ in $t = \{m\}^k$ if it is a test component for $a$ in which: $t$ is a component of $\text{msg}(n_0)$, $k^{-1} \in \text{Safe}$; $a$ does not occur in any component of $n_0$ other than $t$; and $t$ is received at node $n_1$, outside the form $\{m\}^k$.

**Definition 2 (Incoming test)** The edge $n_0 \Rightarrow^+ n_1$ is an incoming test for $a$ in $t_1 = \{m\}^k$ if it is a test for $a$ in which $k \in \text{Safe}$ and $t_1$ is a test component for $a$ in $n_1$.

### 4.5 Conclusions

In this chapter we have described the class of protocol flaws that SHRIMP is intended for fix, the one of replay attack. The rational behind our selection is the fact that most faulty protocols in the literature are precisely vulnerable to this class of attack. We have identified that Syverson’s
taxonomy of replay attacks corresponds to one or more violations of Abadi and Needham’s design principles. Thus, these principles give evidently a clue to develop our patching framework.

We have also explained how SHRIMP contributes to the development process for security protocols. Roughly speaking, SHRIMP is aimed to bridge the gap between design and analysis by adding automated aid for protocol diagnosis and repair. In addition, we have also explained that our patching framework (SHRIMP) is based on proof planning methodology. We have described how each patching method is structured; when they are applicable and how new patching methods could be added to SHRIMP.

Finally, we described the Strand-spaces logic, since SHRIMP has been constructed on it. In the next chapter we describe SHRIMP’s meta-logic, which allow us to express the patching methods that will be introduced later on.
5. **SHRIMP’s Meta-logic**

In this chapter we introduce SHRIMP’s meta-logic, with which we express pre- and post-conditions of our patching methods. SHRIMP’s meta-logic is based on the strand space logic, [86]. Since the strand space logic was designed for protocol verification, its operators are not sufficient to capture all aspects of protocol repair. So, in this chapter, we introduce functions that SHRIMP uses to capture mal-designed cypher-texts (properties of faulty protocols, which are then used to predict protocol flaws) and functions about how to repair them. Protocol repair usually involves changing the structure of a message and so we introduce functions to propagate such changes to all protocol nodes where the changed message appears. We use a *regular bundle* (a regular protocol run) and a *penetrator bundle* (the attack) in our flaw detection process to identify a mal-designed cypher-text, however in order to patch a protocol, we must modify the protocol description rather than the protocol run. To do this, we also introduce a general protocol description language and add some functions in order to link regular bundles with our protocol description (through instantiation).

5.1 **Operators to express Pre- and Post-conditions**

Considering that we are working under the *perfect cryptography assumption*, where the intruder does not practice cryptanalysis, then replaying messages is one of the main strategies that is used by the intruder if he does not know the key that decrypts a cypher-text. The intruder carries out this strategy with the aim that an honest agent, naively, reveals important information such as session keys, nonces, etc. or maybe to make an honest agent authenticate wrongly another agent, who has
never participated in a run of the protocol.

The intruder usually can replay any cypher-text of the protocol under attack, but the replay really has effect if the cypher-text has been designed incorrectly; sometimes the cypher-text does not contain enough information or it is very similar with respect to another in the protocol. Most functions introduced in this section have to do with a syntactic analysis on protocol messages, especially on cypher-texts. We implement some functions that identify situations in which it is necessary to modify a cypher-text that has been used to build up a replay and how it should be changed to prevent the replay.

In what follows, we introduce functions to capture when two messages can be regarded as similar and how to build messages that break this similarity.

5.1.1 Message Similarity

As already explained, in Chapter 4.1.1, a subclass of replay attack makes use of a violation of Abadi and Needham’s principle 10: two or more different cypher-texts of the same protocol cannot be distinguished from one another. So, we must be careful about the format of a message, especially with cypher-texts. To this end, we use the concept of message similarity which we introduce after describing pattern and equivalence.

**Pattern** To capture similarity in the structure of a cypher-text, we use Abadi and Rogaway’s notion of patterns. A pattern is an expression that shows explicitly parts that an agent can decrypt, otherwise they are explicitly denoted by the box symbol, □. Originally the notion of patterns was introduced to capture what an intruder sees on the network considering a finite set of keys (compromised or not) that he knows. The pattern $p$ of a message $m$ that is visible with respect to a set of keys $K$ is defined by:

$$
p(m, K) \overset{\text{def}}{=} m \quad \text{if } m \text{ is atomic or hash}
$$

$$
p(m_1; m_2, K) \overset{\text{def}}{=} p(m_1, K); p(m_2, K)
$$

$$
p(|m|_k, K) \overset{\text{def}}{=} \begin{cases} 
  p(m, K)_k & \text{if } k^{-1} \in K \\
  □ & \text{otherwise}
\end{cases}
$$

For example, let $K = \{K_A^{-1}, K_B^{-1}\}$ be a set of keys, $m_1 = \{|A; N_a; \{A; N_a\}_{K_C}\}_{K_A}$ and $m_2 = \{|A; N_a; \{A; N_a\}_{K_B}\}_{K_A}$ be two messages, then the pattern of such messages is $p(m_1, K) = \{|A; N_a; □\}_{K_A}$ and $p(m_2, K) = \{|A; N_a; □\}_{K_A}$.  

\footnote{Patterns are similar to the notion of skeletons used in [85].}
Equivalence of Messages  Even though two or more different messages have been created to play different roles, they could be confused by an agent because the messages are not distinguishable for an observer; in other words the messages are equivalent and the agent could assume that such messages are the same. Two message terms $m$ and $m'$ are equivalent with respect to a set of keys $K$ iff they yield the same pattern:

$$m \cong m' \text{ iff } p(m, K) = p(m', K)$$

For example, let $K = \{ K_B^{-1} \}$ be a set of keys known by $B$, $m_1 = \{ A; N_a; \{ A; N_a \}_K \}_K$ be a message for $B$ and $m_2 = \{ A; N_a; \{ A; N_a \}_K \}_K$ be a message known by the spy. If the spy sends $m_2$, $B$ could wrongly accept $m_2$ as $m_1$ because $m_1 \cong m_2$, put differently, $p(m_1, K) = p(m_2, K) = \{ A; N_a; \square \}_K$. Notice that $B$, not knowing $K_C$, could not observe the difference between $m_1$ and $m_2$.

Similarity  Sometimes two or more messages that are not equivalent can be similar and under instantiation (see section 5.3.1) they become equal. The intruder could also use notion of similarity in order to confuse a honest agent. So, we define similarity to capture this kind of vulnerability as follows:

Definition 3 (similarity) Two terms $m$ and $m'$ are similar with respect to a set of keys $K$, written $m \sim_K m'$, iff $m \neq m'$ and there is a bijective replacement $\sigma$ on patterns (with $\square \sigma = \square$) mapping symbols from $m$ to $m'$, such that $p(m, K) = p(m', K)\sigma$ holds.

For instance, if $K_1 \neq K_2$ and $K = \{ K_1^{-1}, K_2^{-1} \}$ the following relations are held:

$$\{ A; B; \{ A; N_a \}_K \}_K \sim_{K_1} \{ C; D; \{ C; N_a \}_K \}_K$$  \hspace{1cm} (5.1)

$$\{ A; B; \{ m_1 \}_K \}_K \sim_{K_1} \{ C; D; \{ m_2 \}_K \}_K$$  \hspace{1cm} (5.2)

$$\{ A; B; \{ m_1 \}_K \}_K \not\sim_{K_1} \{ A; B; \{ m_1 \}_K \}_K$$  \hspace{1cm} (5.3)

$$\{ A; B; \{ m_1 \}_K \}_K \not\sim_{K_1} \{ C; D; \{ m_2 \}_K \}_K$$  \hspace{1cm} (5.4)

$$\{ A; B; \{ A; T_a \}_K \}_K \not\sim_{K_1} \{ C; D; \{ C; N_a \}_K \}_K$$  \hspace{1cm} (5.5)

Notice that as described in Section 4.4.1, we are using type terms whose ranges are disjoint. It follows that the various kinds of messages are distinct, where $A \neq N$, $N \neq K$ and so forth. However, $A$ is equal under substitution to $B$ ($A \sim B$) because of that, relations (5.1) and (5.2) are held. On the other hand, relations (5.4) and (5.5) do not hold because $K_1 \not\sim K_2$ and $T_a \not\sim N_a$ respectively. Obviously relation (5.3) does not hold because the messages are equal (according to definition 3).
We introduce the function \textit{visible content} \( ct(m, K) \) of a message \( m \) wrt. a set of keys \( K \) like the set of messages that can be obtained by adding the components of compound messages and by decrypting messages whose keys are in \( K \). This operator is similar to operator \textit{analz} \( (M) \) of Paulson [74], except that Paulson’s operator can also add agent names (not included in \( M \)) because they are publicly known. Function visible content is given by:

\[
ct(m, K) \overset{\text{def}}{=} \{ m \} \quad \text{if } m \text{ is Atomic or hash}
\]
\[
ct(m_1; m_2, K) \overset{\text{def}}{=} \ct(m_1, K) \cup \ct(m_2, K)
\]
\[
ct([m]_k, K) \overset{\text{def}}{=} \begin{cases} 
\ct(m, K) & \text{if } k^{-1} \in K \\
\{ [m]_k \} & \text{otherwise}
\end{cases}
\]

\subsection{Explicit agent names in messages}

Another subclass of replay attack exploits a violation to Abadi and Needham’s principle 3: the originator/recipient of a cypher-text in a message of the protocol cannot be distinguished. This principle prescribes that the agent names relevant for a message should all be derivable either from the format of a message or from its content. To determine when a cypher-text lacks of this property, SHRIMP includes functions that allow us to compute the name of the agents involved in the exchange of a message and to reason about the name of the agents that can be inferred from the encoding of a message.

\textbf{Witnesses of a Message} It is evident that agents involved in the exchange of a message are its initiator and its final receiver. However, sometimes there are some additional agents involved in its exchange. So, we must precisely know who are such agents and to register them. There is a restriction in this activity, sometimes agents are only used like an intermediary oracle (they can only resend a message, e.g. a cypher-text, without knowing its content), thus we should not consider these agents because they are not intervening in reality. In symbols:

Given a bundle \( \beta \), \( A \) is the \textit{originator} (respectively \textit{recipient}) of a term \( m \), \( A \beta m \), (respectively \( A \beta m \)) iff there is a positive (respectively negative) node \( n \) in a strand played by \( A \) in \( \beta \) such that \( m \) \textit{originates} at \( n \) (respectively \( m \in \ct(msg(n), K) \), \( K \) being the keys known to \( A \), \textit{Safe}.)

\textbf{Definition 4 (Correspondents)} Let \( \beta \) be a bundle. The participants involved in the exchange of \( m \) in \( \beta \), called the \textit{correspondents} of \( m \), \( \text{Partners}(m)_\beta \), is given by \( \{ A : A \beta m \lor A \beta m \} \).

\footnote{See 4.4.2 for a description of \textit{originates}}
Agent Name Deduction  Sometimes the agent names that are obtained by definition 4 are omitted in a cypher-text, resulting most of the time in a protocol flaw. We implement a function that uses basically two means to deduce the name of an agent: i) directly, from the agent identity in plain text, e.g. $A, B$; and ii) indirectly, from the encryption key (see Propositions 1 and 3 in chapter 3).

**Definition 5 (Agent name deduction)** The name of the agents that can be (safely) deduced from the encoding of a message is given by:

$$
\text{Agents}(\|m\|_k) \overset{\text{def}}{=} \begin{cases} 
\{A\} \cup \text{Agents}(m) & \text{if } k = K_A^+ \lor k = K_A^- \\
\text{Agents}(m) & \text{otherwise}
\end{cases}
$$

$$
\text{Agents}(m_1; m_2) \overset{\text{def}}{=} \text{Agents}(m_1) \cup \text{Agents}(m_2)
$$

$$
\text{Agents}(m) \overset{\text{def}}{=} \begin{cases} 
\{m\} & \text{if } m \text{ is an agent name} \\
\{\} & \text{otherwise}
\end{cases}
$$

Notice that we do not infer names from symmetric encryption, because the flow of the message cannot be determined. Since, more of one agent knows the encryption key, then it cannot be inferred who encrypted a message from the message itself. We retake the example described in section 3.4.2 to illustrate our previous argument:

$$
A \rightarrow B : \|N_a\|_{K_{ab}} \quad (5.6)
$$

Upon message reception of this protocol step, it is very difficult for $B$ to know if such a message was sent by $A$ because it could also have been sent by him in any run before, as illustrated in the following replay:

$$
C(B) \rightarrow A : \|N_a\|_{K_{ab}} \quad (5.7)
$$

Let $C$ be a dishonest agent, who resends $A$ the message of (5.6). As one can see, $A$ is in the same position as $B$ was previously because the flow of the message cannot be determined. Therefore, when a message is encrypted using symmetric key we cannot infer names.

### 5.1.3 Patch Production

**Breaking similarity** Sometimes the similarity of two or more messages can be broken by re-arranging their content. This operation implies to know whether the message rearranged preserves the same meaning as before to preserve its purpose. Put differently, we must know whether a message (the new one) is equal to another (the older one) after applying the rearrangement operation. To do this, we introduce the concept of equivalence under rearrangement as follows:
\textbf{Definition 6 (equivalent under rearrangement)} Two messages $m$ and $m'$ are equivalent under rearrangement, $m \equiv m'$ for short, iff $ct(m, K) = ct(m', K)$ for all sets of keys $K$.

For example, message $\{A; Na; \{A; Na\} _{KC}\} _{KA}$ and message $\{\{A; Na\} _{KC} ; Na; A\} _{KA}$ are equivalent under rearrangement, then it holds that:

$$ct(\{A; Na; \{A; Na\} _{KC}\} _{KA} , \{(K_A)^{-1}\}) = ct(\{\{A; Na\} _{KC} ; Na; A\} _{KA} , \{(K_A)^{-1}\})$$

In order to fix a mal-designed cypher-text sometimes we have to do changes in the structure of a message such as term rearrangement (as described above ) or term insertion (as described in 5.1.2). For example, sometimes we have to add agent names in cypher-text that are necessary; other times we have to add additional information (like tags) for breaking the similarity of two or more messages when it cannot be done under rearrangement. These changes will simply enrich the messages exchanged by the principals with additional information. To capture this, we use the following property:

\textbf{Definition 7 (Monotonicity)} Let $m, m'$ be messages and let $A_0 \subseteq T(\Omega, A)$. $m \leq A_0 m'$ iff $ct(m, K') \subseteq ct(m', K')$ and $ct(m', K') \subseteq ct(m, K')$ and $ct(A_0, K')$ for all $K' \subseteq K$.

For example, let $m_1$ be a message term and $(K_B)^{-1} \in K'$, then $\{Na\} _{KB} \leq \{m_1\} \{Na, m_1\} _{KB}$ because

$$ct(\{Na\} _{KB} , K') \subseteq ct(\{Na, m_1\} _{KB} , K')$$

then $\{Na\} \subseteq \{Na, m_1\}$ and;

$$ct(\{Na, m_1\} _{KB} , K') \subseteq ct(\{Na\} _{KB} , K') \cup ct(\{m_1\}, K')$$

then $\{Na, m_1\} \subseteq \{Na, m_1\}$.

\textbf{Avoiding Message Confusion} While breakin similarity of two messages adding new information to the protocol, we must be careful in avoiding message confusion. We need ensuring that the new message component or even the entire new message does not have a structure that is similar to another one in the protocol, so-called \textit{collision freeness}:

\textbf{Definition 8 (Collision Freeness)} Let $K$ be a set of keys, $m$ a message and let $A_1 \subseteq T(\Omega, A)$ be a set of messages. $m$ is collision free to $A_1$ iff $\forall m' \in ct(m, K), \forall m'' \in A_1, (m' \not\sim_K m'')$.

Attention is now given to functions that enable us to carry out general transformations to a protocol run.

\section{5.2 Protocol Description}

The strand space logic represents protocol executions rather than protocol descriptions. So, we have to develop our own protocol specification language, which should be general, simple and specially so that the protocol steps can be instantiated with a regular bundle.
So in what follows, we introduce the syntax of our protocol specification language, which contains variables, constants and types used to represent the protocol messages.

### 5.2.1 Variables, Constants and Types

The specification of a protocol in SHRIMP is typed: each variable and constant must have a unique type. We do not use explicit constructors to define the type of a message (as Paulson does [74]), instead we establish a type declaration in the variables and so, each variable has implicitly its intended type. We use typed variables and constants to form messages. Table 5.1 summarizes the types available in SHRIMP.

<table>
<thead>
<tr>
<th>agent: for agent names. The server and the intruder always have a special identifier, S and Spy respectively.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ltKey: for long-term keys (also known as shared keys).</td>
</tr>
<tr>
<td>stKey: for short-term keys (also known as session keys).</td>
</tr>
<tr>
<td>nat: for timestamps, credit card numbers, protocol steps, etc.</td>
</tr>
<tr>
<td>nonce: for nonces (unique values, unguessable by the intruder).</td>
</tr>
<tr>
<td>pubKey: for public keys.</td>
</tr>
<tr>
<td>privKey: for private keys.</td>
</tr>
</tbody>
</table>

Table 5.1: Types available in SHRIMP

### 5.2.2 Message Representation

In the Strand-spaces model the set of messages \( M \) that can be exchanged between principals in a protocol is freely generated from two disjoint sets, the set of keys \( K \) and the set of messages \( A \). We represent the set of texts \( M \) in two groups: Atomic and Compound. Atomic are constant messages or variable messages denoting agent names, time-stamps, nonces and keys and Compound messages are concatenated messages, ciphered messages and one-way function (hash) messages. Let \( C, V \) and \( K \) be disjoint countable sets of constants, variables and keys respectively. The syntax of protocol messages is defined by the following grammar:

\[
\begin{align*}
Msg & \equiv \text{Atomic} \mid \text{Compound} \\
\text{Atomic} & \equiv C \mid V \mid K \\
\text{Compound} & \equiv Msg; Msg \mid \text{Hash}(Msg)
\end{align*}
\]

The set of keys \( K \) is defined by two groups: symmetric and asymmetric keys. Symmetric keys include long-term keys and short-term keys. Asymmetric keys include public keys and private keys.
keys. Each agent contains a key-ring where he can either store keys or recover keys already known. For that and following what we have introduced in the strand space logic, section 4.4.1, there are three functions: the first one maps principals, A, B, . . . to their symmetric shared keys $K_A = \text{sym}(A), K_B = \text{sym}(B), . . .$ from keyring, the second one principals to their public keys $K_A^+ = \text{pub}(A), K_B^+ = \text{pub}(B), . . .$ and the third one to their private keys $K_A^- = \text{priv}(A), K_B^- = \text{priv}(B)$.

5.2.3 Protocol Specification

We represent a protocol in three main parts: i) the type declaration, ii) the init-state, and iii) a set of roles. In the first part we declare the type of each variable which indicates the intended kind of constant with which the variable should be instantiated. The syntax of a type declaration is given by the following context-free grammar:

\[
\begin{align*}
\text{TypeDeclaration} & \equiv \text{typeDec} ::= \text{Declaration} \\
\text{Declaration} & \equiv \text{Type Vars; Declaration | Type Vars} \\
\text{Type} & \equiv \text{agent | nonce | nat | pubKey | privKey | ltKey | stKey} \\
\text{Vars} & \equiv \text{Var, Vars | Var} \\
\text{Var} & \equiv [A-Z][a-z][0-9]^* \\
\end{align*}
\]

where typeDec is a reserved word that initiates the type declaration. By convention, we follow Alice and Bob notation and we reserve A, B, . . . , to denote variables of type agent; T_a, T_b, . . . , to denote timestamps of type nat; N_a, N_b, . . . to denote variables of type nonce; K_{ab}, K_{ac}, . . . to denote session keys of type stKey. K_A to denote the long-term key of agent A, of type ltKey; K_{pubA} to denote the public key of agent A, of type pubKey; and K_{privA} to denote the private key of agent A, of type privKey. We will write $K_A^+$ and $K_A^-$ for short of $K_{pubA}$ and $K_{privA}$ respectively.

In the second part we declare the initial knowledge of each participant role, so-called init-state. The initial knowledge varies from one protocol to another. For example, in some cases public keys are distributed by a server, but in other cases they must be specified in the initial knowledge of an agent. The syntax of init-state is given as follows:

\[
\begin{align*}
\text{IS} & \equiv \text{initState ::= Knowledge} \\
\text{Knowledge} & \equiv \text{RoleName : Vars; Knowledge | RoleName : Vars} \\
\text{RoleName} & \equiv \text{initiator | responder N | Server} \\
\text{N} & \equiv 1 | 2 | . . . \\
\end{align*}
\]

In the third part we declare the role that each participant plays in a protocol: server (if any), initiator and (one or more) responder(s). A role is given by a symbolic strand representing the
sequence of actions prescribed by the protocol to an agent. A symbolic strand is a sequence of
nodes, each of which is denoted by an id-step and a communication event. The syntax of a role is
given by the following context-free grammar:

\[
\begin{align*}
\text{Role} & \overset{\text{def}}{=} \text{RoleName} ::= \text{Strand} \\
\text{Strand} & \overset{\text{def}}{=} \langle \text{Node} \rangle \mid \text{Strand, Strand}^* \\
\text{Node} & \overset{\text{def}}{=} \text{N.Event} \\
\text{Event} & \overset{\text{def}}{=} \alpha.\text{Msg} \\
\alpha & \overset{\text{def}}{=} + \mid -
\end{align*}
\]

Figure 5.1 illustrates the protocol description for the NSPK protocol.

\[
\begin{align*}
typeDec & ::= \text{agent A, B}; \text{nonce N}_a, N_b; \text{pubKey}K^+_A, K^+_B \\
\text{initState} & ::= \text{initiator} : A, B, K^-_A, K^+_A, K^+_B; \\
& \quad \text{responder} : A, B, K^-_B, K^+_B, K^+_A \\
\text{initiator} & ::= \langle 1. + . \langle \{N_a; A\}K^-_B \rangle, \langle 2. - . \langle \{N_a; N_b\}K^+_B \rangle, \rangle, \\
& \quad \langle 3. + . \langle \{N_b\}K^+_B \rangle \rangle \\
\text{responder} & ::= \langle 1. - . \langle \{N_a; A\}K^+_B \rangle, \langle 2. + . \langle \{N_a; N_b\}K^-_B \rangle, \rangle, \\
& \quad \langle 3. - . \langle \{N_b\}K^-_B \rangle \rangle
\end{align*}
\]

Figure 5.1: The NSPK protocol description

5.3 Management of Change in a Protocol Description

SHRIMP finds bugs in protocols by analyzing a penetrator bundle (the attack) and a regular bundle
(an intended protocol run). Often this analysis suggests a change in the structure of a message and
in that case SHRIMP identifies such a message in the regular bundle and by matching SHRIMP
computes the changes to be done in the protocol description. Notice that when a message is
modified we must consider if the agent who has originally created the message can do the change.
We take care of these capabilities when we analyze the regular bundle, because of that we do not
take into account these capabilities in the operations of the protocol description.

\footnote{A symbolic strand is as a particular strand, except that all message terms of all nodes are variables.}
To do the matching previously explained we firstly describe substitution, instantiation and unification.

5.3.1 Substitution, Instantiation and Unification

We use \( \text{vars}(m) \) to denote the set of variables occurring in a message \( m \). We say that \( m \) is ground (a constant message) whenever \( \text{vars}(m) = \emptyset \) and write \( \text{ground}(m) \).

**Definition 9 (Substitution)** A substitution \( \sigma = [m/V] \) is a mapping of variables \( V \) to message terms \( m \). The substituted message terms do not have to be closed and may contain free variables that can later be replaced by other substitutions.

Applying a substitution \( \sigma \) to a particular message term \( m \) yields a new one \( \sigma(m) \) obtained from \( m \) by simultaneously replacing all free variables \( V \) in \( m \) with message term \( \sigma(m) \).

**Definition 10 (Instantiation)** A message term \( m \) is an instance of a variable \( V \) iff there exists a substitution \( \sigma \) such that \( m = V\sigma \).

**Definition 11 (Unification)** Two message terms, \( m_1 \) and \( m_2 \), unify whenever there exists a substitution \( \sigma \), called their unifier, under which they are equal; in symbols \( m_1 = \sigma m_2 \).

5.3.2 Protocol and Counterexample Definitions

We introduce the symbolic bundle term, which is like a particular bundle, except that the messages of all nodes in the symbolic bundle are variables. We extract the symbolic bundle of a protocol specification \( P \) using function \( \text{bundleOf}(P) \).

**Definition 12 (Protocol specification)** We extend \( \text{ground}(m) \), \( \text{vars}(m) \) and substitution application to homomorphisms over strand nodes, roles and sets of roles in the expected manner. Then, the specification of a protocol \( P \) consists of a type declaration \( D \), init-state \( IS \) and a set of roles \( R \), such that \( \text{vars}(IS) \subseteq \text{vars}(R) \subseteq \text{vars}(D) \) and the symbolic strands of all roles of \( R \) form a symbolic bundle, \( \text{bundleOf}(P) \).

If we instantiate a role (a symbolic strand) we get a particular strand representing a trace of a principal in a protocol run. When all roles are instantiated the strands form a legal execution of a protocol, a bundle.

**Definition 13 (Protocol instantiation)** An instantiation of a protocol specification with roles \( R \), type declaration \( D \), init-state \( IS \) and substitution \( \sigma \) is valid if i) \( \text{ground}(R\sigma) \), \( \text{ground}(D\sigma) \) and \( \text{ground}(IS\sigma) \) hold; and \( R\sigma \) yields a bundle, built out of each ground strand representing a trace of an agent.
A counterexample, \( CE \), is a bundle containing a \textit{penetrator strand} (the intruder participation) and various executions of a protocol, which states that a security protocol violates a security property. Note that in a counterexample the intruder may make an agent play interchangeably the role of initiator and responder as many times as necessary for the intruder to find an attack.

Let \( \mathcal{B} \) be the set of bundles (regular runs or not) that can be executed from protocol \( P \) then \( \beta_P = \text{bundleOf}(P) \) denotes the symbolic bundle extracted directly from \( P \), and \( CE \) be a particular bundle such that \( CE \in \mathcal{B} \), then it holds that for every node \( \langle n \rangle \) in \( CE \) there exists a node \( \langle N \rangle \) in \( \beta_P \) that can be instantiated, \( N\sigma = n \).

### 5.3.3 Change Propagations in a Symbolic Bundle

When SHRIMP detects bugs in a faulty protocol, often it suggests a change in the structure of a message. In this case SHRIMP identifies the node originating such a message considering a regular bundle and by matching with the symbolic bundle of the faulty protocol, SHRIMP computes the changes to be done in the symbolic one and in the rest of the protocol description. The following definitions aim at a meta-theory to allow for tracing the consequences of changing all the nodes in the set of symbolic strands depending on changes to a particular node. Before doing that, we introduce term positions theory because we use this theory like base to represent changes.

#### 5.3.3.1 Term positions

In the strand space logic the set of messages \( M \) that can be exchanged between principals in a protocol is freely generated from two disjoint sets, the set of keys (\( K \)) and the set of atomic messages (\( A \)) by means of concatenation, \( m_1; m_2 \), and encryption, \( \{m\}_K \). Here we extend this consideration to use hash functions, \( \text{Hash}(m) \).

Compound message operators can be seen as \textit{function symbols} \( f \) that receive messages as parameters (the number of parameters is the \textit{arity} of the function symbol) and return a new compound message. For example, the concatenated message function \( ';' \) has arity 2; the encrypted message function \( \{m\}_K \) has arity 2: the first parameter is the encryption key and the second one is the message to be encrypted; and the hash function \( \text{Hash}(m) \) has arity 1.\footnote{Although a hashed message is a compound message, most operations in this chapter will interpret this kind of compound message like an atomic one, since messages in a hashing operation are unrecoverable.} Let \( \Omega \) be the set of function symbols on messages and \( A \) be a set of atomic messages, then \( M = T(\Omega, A) \) denotes the set of all message terms that can be formed from \( \Omega \) and \( A \).

The structure of a message can be nicely illustrated by representing it as a tree, where atomic messages are leaves and compound messages are intermediate nodes. Figure 5.2 depicts a message tree example for term \( m = A; \{A; N_a\}_K \). Note that \( \pi_i \) denotes the positions of message \( m \). In such...
an example, position $\epsilon$ refers to the top construction on $m$, and for each term of $m$, a string of integers is assigned. We adopt definitions of [39] on standard notations of $\text{Pos}(t)$ to denote the set of all positions $\pi$ in $t$, such that $t \in T(\Omega, A)$, then $\text{Pos}(t)$ is inductively defined as follows:

- If $t \in A$ then: $\text{Pos}(t) \overset{\text{def}}{=} \{\epsilon\}$, where $\epsilon$ denotes the empty string;

- If $t = f(t_1, \ldots, t_n)$, where $f$ represents a Compound message, then

$$\text{Pos}(t) \overset{\text{def}}{=} \{\epsilon\} \cup \bigcup_{i=1}^{n} \{i.\pi \mid \pi \in \text{Pos}(t_i)\}.$$

$t_{|\pi}$ denotes the sub-term of $t$ at position $\pi$ and $t[|\pi \leftarrow s]$ denotes the term that we obtain by replacing the term $t_{|\pi}$ in $t$ by $s$. Note that by induction:

$$t_{|\pi} \overset{\text{def}}{=} t$$

$$f(t_1, \ldots, t_n)_{|\pi, n'} \overset{\text{def}}{=} (t_{|\pi})_{|n'}$$

**Definition 14 (Subterm positions)** Let $t, s \in T(\Omega, A) \land s \subseteq t$. A finite set $\Pi_s^t = \{\pi_1, \ldots, \pi_n\}$ denotes the set of all occurrences of sub-term $s$ in $t$ iff for all $1 \leq i \leq n : \pi_i \in \text{Pos}(t) \land t_{|\pi_i} = s$.

For example, the set of all positions of $m = A; \{A; N_a\}_K$ (message of Figure 5.2), is defined by $\text{Pos}(m) = \{\epsilon, 1, 2, 2.1, 2.2, 2.1.1, 2.1.2\}$, $m_{|2} = \{A; N_a\}_K$ and $m_{|2.1.2} = N_a$. The positions of term $A$ in term $m$ are $\Pi^m_A = \{1, 2.1.1\}$.

### 5.3.3.2 Change operators for Messages

The following definitions capture changes in the structure of a message.

---

**Figure 5.2:** A message tree $A; \{A; N_a\}_K$, with explicit $\pi$ positions
Definition 15 (Term replacement) Let \( t \) be a message term, \( m_1 \) be a message subterm such that \( m_1 \sqsubseteq t \) which we want to replace by \( m_2 \), written \( \Theta^t[m_1/m_2] \). Then,

\[
\Theta^t[m_1/m_2] \overset{\text{def}}{=} \forall \pi_i \in \Pi^t_{m_1}, t = t[\pi_i \leftarrow m_2]
\]

Notice that \( m_1 \) and \( m_2 \) may have similar subterms. In this case, term replacement operation could not get in an infinite operation because it is done only in first order, it means that \( \Pi^t_{m_1} \) is only evaluated once.

For example, considering message \( A; \{|A; Na|\}^K_A \), if we want to replace agent name \( A \) by \( B \) in such a message, we apply the term replacement rule \( \Theta^A;\{|A; Na|\}^K_A[A/B] = A; \{|A; Na|\}^K_A[\pi_i \leftarrow B], \) for all \( \pi_i \in \Pi^A;\{|A; Na|\}^K_A = \{1, 2.2.1\}, \) then \( A; \{|A; Na|\}^K_A[1 \leftarrow B, 2.2.1 \leftarrow B] = B; \{|B; Na|\}^K_A \).

Another example is again to consider message \( A; \{|A; Na|\}^K_A \), if we want to replace agent name \( A \) by \( \text{Hash}(A) \) in such a message, we apply the term replacement rule \( \Theta^A;\{|A; Na|\}^K_A[A/\text{Hash}(A)] = A; \{|A; Na|\}^K_A[\pi_i \leftarrow \text{Hash}(A)], \) for all \( \pi_i \in \Pi^A;\{|A; Na|\}^K_A = \{1, 2.2.1\}, \) then

\[
A; \{|A; Na|\}^K_A[1 \leftarrow \text{Hash}(A), 2.2.1 \leftarrow \text{Hash}(A)] = \text{Hash}(A); \{|\text{Hash}(A); Na|\}^K_A
\]

5.3.3.3 Change Operators for a Symbolic Bundle

Changing a message that is sent from \( A \) to \( B \) in the bundle changes the knowledge of \( B \) and thus its abilities to construct consecutive messages. If \( B \) should forward an encrypted message coming from \( A \) and this latter message is changed in the bundle, then we also have to change the message forwarded by \( B \) and so we propagate changes. To this end, we introduce the following definition:

Definition 16 (term replacements in a bundle) Let \( \beta \) be a bundle such that \( \beta = \langle N, (\rightarrow \cup \Rightarrow) \rangle \). The replacement of a term \( t \) by \( t' \) in \( \beta \), written as \( \beta[t/t'] \) is given by:

\[
\beta[t/t'] = \langle N[t/t'], (\rightarrow \cup \Rightarrow) \rangle \overset{\text{def}}{=} \forall n \in N. \Theta^{\text{msg}(n)}[t/t']
\]

For example, if we want to change message \( \{|Na; M; A; B|\}^K_A \) sent by initiator \( A \) in node 1 of Figure 5.3 by this new one \( \{|Na; M; B|\}^K_A \), first of all, we need to modify the message in a node level and then to propagate changes in all nodes where the old one appears.
Changes in a node level: Let \( \pi \in \text{Pos}(\text{msg}(\{1. + . M; A; B; \{\{N_a; M; A; B\}\}_A^k)) \), such that:

\[
\Theta^{\{N_a; M; A; B\}_A^k}_A[\pi; B] = \{\{N_a; M; A; B\}\}_A^k, \quad \text{then}
\]

\[
\Theta^{\{N_a; M; A; B\}_A^k}_A[\pi_i; B] = \{\{N_a; M; A; B\}\}_A^k, \quad \text{for all } \pi_i \in \Pi_{\{A; B\}_A^k}^{\{N_a; M; A; B\}_A^k}, \text{then}
\]

Changes in a bundle level: So far, with \( \Theta^{\{N_a; M; A; B\}_A^k}_A[A; B] \) we have changed the message at a node level, now we have to do the same operation with each node in the bundle. Notice that, changes will be only reflected where message \( \{\{N_a; M; A; B\}\}_A^k \) appears. Therefore, applying \( \beta^\star[\{\{N_a; M; A; B\}\}_A^k / \{\{N_a; M; B\}\}_B^k] \) the new bundle of Figure 5.3 would be like the one in Figure 5.4.

5.3.4 Changes in a Protocol Description

In this subsection we introduce how to do changes in our protocol description. These changes are firstly carried out in a symbolic bundle and then are propagated to the rest of the protocol. We must be careful that changes in a protocol description need to be consistent in the sense that a new message term added in the symbolic bundle must also be:

- added in the type declaration;
- verified whether the new message term can be synthesized correctly by the role that constructs such a message; and
- added in its corresponding (sending or receiving) node.
Let `bundleOf vars`.

To do that, we firstly introduce either the notion of a well formed protocol and the operator `synthAg(M)`. This last operator models the messages that can recursively be built up from all messages contained in `M`. This operator is similar to operator `synth(M)` of Paulson [74], except that Paulson’s operator can also add agent names (not included in `M`) because they are publicly known. We use a similar definition of [74] to formalize `synthAg(M)` as follows:

\[
\begin{align*}
\text{m} & \in M & \text{m} & \in \text{synthAg}(M) \\
\text{Hash}(m) & \in \text{synthAg}(M) \\
\text{m}_1 & \in \text{synthAg}(M) & \text{m}_2 & \in \text{synthAg}(M) & \text{m} & \in \text{synthAg}(M) & k & \in M \\
\{\text{m}_1, \text{m}_2\} & \in \text{synthAg}(M) & \{\text{m}\}_k & \in \text{synthAg}(M)
\end{align*}
\]

**Definition 17 (A well formed protocol)** Let \(\mathcal{P}\) be the specification of a protocol with type declaration \(D\), init-state \(IS\) and a set of roles \(\mathcal{R}\). We say that \(\mathcal{P}\) is a well formed protocol if:

1. \(\text{vars}(IS) \subseteq \text{vars}(\mathcal{R}) \subseteq \text{vars}(D)\)

2. \(\text{bundleOf}(P)\) is a well formed symbolic bundle, such that: let \(n\) be a node in strand \(s\) such that \(s = \text{strand}(n)\), \(M_{n-1}\) be the visible content of messages extracted from all nodes in \(s\) up to \(n - 1\) (including the init-state of \(s\)) and \(U_n\) be a set of uniquely originating messages in node \(n\), then for all \(n \in \text{bundleOf}(P)\) and \(\text{sign}(n) = +\), \(\text{msg}(n) \in \text{synthAg}(M_{n-1} \cup U_n)\) holds.

3. Let \(n_1\) and \(n_2\) be nodes in \(\text{bundleOf}(P)\), then for every \(n_1 \rightarrow n_2\), there exists an evaluation such that \(\text{msg}(n_1) = \text{msg}(n_2)\) holds.

For example, consider again the Otway and Rees protocol (Figure 5.5), but now in our formal representation. This protocol is a well formed protocol because the three points described in def-
\[
\text{typeDec} ::= \text{agent } A, B, S; \text{nonce } M, N_a, N_b;
\]
\[
\text{ltKey}_{K_A}, K_B; \text{stKey}_{K_{ab}}
\]
\[
\text{initState} ::= \text{initiator} : A, B, S, K_A
\]
\[
\text{responder} : A, B, S, K_B
\]
\[
\text{Server} : A, B, S, K_A, K_B
\]
\[
\text{initiator} ::= \langle 1. + . M; A; B; \{N_a; M; A; B\}_{K_A} \rangle,
\]
\[
\langle 4. - . M; \{N_a; K_{ab}\}_{K_A} \rangle,
\]
\[
\text{responder} ::= \langle 1. - . M; A; B; \{N_a; M; A; B\}_{K_A} \rangle,
\]
\[
\langle 2. + . M; A; B; \{N_a; M; A; B\}_{K_A}; \{N_b; M; A; B\}_{K_B} \rangle,
\]
\[
\langle 3. - . M; \{N_a; K_{ab}\}_{K_A}; \{N_b; K_{ab}\}_{K_B} \rangle,
\]
\[
\langle 4. + . M; \{N_a; K_{ab}\}_{K_A} \rangle
\]
\[
\text{Server} ::= \langle 2. - . M; A; B; \{N_a; M; A; B\}_{K_A}; \{N_b; M; A; B\}_{K_B} \rangle,
\]
\[
\langle 3. + . M; \{N_a; K_{ab}\}_{K_A}; \{N_b; K_{ab}\}_{K_B} \rangle
\]

Figure 5.5: The Otway and Rees Protocol Description

inition 17 hold. Now, If we want to change message \{N_a; M; A; B\}_{K_A} sent by the initiator in node 1 by this new message \{N_a; M; B\}_{K_A}, we need to propagate the change in all nodes where the old one appears. To do this, we follow a similar operation with bundle operation, described in 5.3.3.3, and introduce the following definition:

**Definition 18 (term replacements in a protocol)** Let \( \mathcal{P} \) be the specification of a protocol with type declaration \( D \), init-state \( IS \) and a set of roles \( \mathcal{R} \). The replacement of a term \( t \) by \( t' \) in \( \mathcal{P} \), written as \( \mathcal{P}[t/t'] \), holds iff:

- \( \beta = \text{bundleOf}(\mathcal{P}) \), \( \mathcal{P}' = \mathcal{P}[t/t'] \);
- \( \beta[t/t'] \) is equal to \( \text{bundleOf}(\mathcal{P}') \);
- \( \mathcal{P}' \) is a well formed protocol.

Using \( \mathcal{P}[t/t'] \), SHRIMP is able to modify a protocol description, starting from the node originating the message that suggested the enhancement to all the successor nodes where the changes endure. This yields a path, we call a *change enduring path*. A change enduring path is guaranteed to be finite and acyclic both because bundles are also finite and acyclic and because changes are propagated considering only inter-strand transitions.
5.4 Conclusions

In this chapter we have introduced SHRIMP’s meta-logic. This meta-logic has been developed on the strand spaces logic, however, any logic able to represent counterexamples could have been used. SHRIMP’s meta-logic will allow us to represent pre- and post-conditions of the patching methods that will be introduced in the next chapter.

To build up part of this meta-logic we have used terms theory and thus we can describe some operators on messages because messages can be represented as terms. Terms theory allows us to define operations in an easier way such as: term positions, insertion, subterm repetition, term replacement, etc.

SHRIMP’s meta-logic includes operators on messages, protocols and counterexamples. Operators in a message level allow us to instantiate messages; to deduce agent names; to know when two or more messages are similar and to do changes in the structure of a message to break its ambiguity, etc. Operators in a protocol level allow us to propagate changes in all nodes of a protocol. Operators in a counterexample level allow us to instantiate counterexample messages with the protocol description.

Once introduced all basic ingredients, we are ready to represent the patching methods in the following chapter.
6. **SHRIMP: The Patching Framework**

In this chapter we introduce how SHRIMP patches faulty security protocols, the *patching framework*. The patching framework that we have developed deals with faulty protocols susceptible to a replay attack, since most faulty protocols found in the literature are *replay attacks* (as already explained in Chapter 4.1).

While developing our patching methods, we identified that the class of replay attack has different subclasses. Therefore, our patching framework is formed by one *composed method* and three *patching methods*.\(^1\) The general composed method of SHRIMP is *replay*, section 6.1. This compound method calls three patching methods: *message_encoding*, *agent_naming* and *session_binding*. *message_encoding* repairs faulty protocols where two or more cypher-texts that are different one another have similar structure, section 6.2. The *agent_naming* patching method repairs faulty protocols containing a message without proper naming, section 6.3. Finally, *session_binding* patching method deals with faulty protocols that contain a message which cannot be associated with a particular protocol run, section 6.4.

To illustrate the use of SHRIMP’s patching methods, for each patching method we present two faulty protocols and we show how these protocols are patched. Each of the attacks presented in this chapter has been found using the AVISPA tool. So, we use the AVISPA hierarchy of authentication (see appendix A.3 for more details).

---

\(^1\)See section 4.3 for more details about *composed methods* and *patching methods*
6.1 The Replay Compound Method

As we have already explained in section 2.2 the capabilities of the intruder in the formal methods school follow the Dolev and Yao model. In this model although the intruder cannot break cryptography, he may take advantage of mal-designed cypher-texts to exploit a vulnerability in a protocol. So, his main strategy is to reuse messages. For example, most faulty security protocols of the Clark and Jacob library and AVISPA library, described in 2.1, are susceptible to replay attacks.

The chief method of SHRIMP is the replay compound method, see Fig. 6.1. A replay attack involves the replay of a cyphered message from outside the current run of a protocol. So, we formulate in the compound message’s preconditions a general condition that spots any cypher-text being reused in an attack.

The input of the method is a protocol specification \( P \), an attack of the protocol \( \beta_A \), and a regular bundle \( \beta_R \). We find the protocol flaw by analyzing both its attack and intended run, then we modify \( P \). The preconditions state that there exists, in the attack, a regular agent \( s_r \) (honest agent) playing a normal run of the protocol which sent a cipher-text \( \{m\}_k \) in node \( \langle s_r, i \rangle \). Then the penetrator agent \( s_p \) (the intruder agent) intercepts the cypher-text and then resends it in node \( \langle s_p, j \rangle \) to other honest agent \( s_q \) (maybe the same agent).

The replay compound method invokes three sub-methods: message_encoding, agent_naming and session_binding. The order in which methods are attempted is important as it imposes a hierarchy in terms of the complexity of implementing the patch. message_encoding is more viable because when modifying a protocol message it may not introduce additional components. agent_naming is more viable than session_binding because it only modifies protocol messages. By contrast, session_binding involves the insertion of protocol steps.

6.2 Patching Protocols Violating Principle 10

The message_encoding patching method repairs a faulty protocol that portrays two or more cypher-texts that are different one another but have similar structure. These kinds of protocols violate principle 10 of Abadi and Needham. The intruder may exploit this vulnerability by making one cypher-text play the role of a message like the other in a different run. According to Syverson’s taxonomy, in this class of attack, so-called deflection attack, one message is forwarded to some other agent than the intended recipient.

To illustrate the rationale behind our message_encoding patching method, we shall show two faulty security protocols, namely: the WMF and BAN-Yahalom protocol.
**Name:** replay

**Input:** $P \in \Sigma$ % Protocol specification
- $\beta_A$ % A bundle denoting an attack of the protocol
- $\beta_R$ % A bundle denoting a regular run of the protocol

** Preconditions:**
- % Spy reuses cypher-text $\| m \|_k$:
  $$\exists i, j, k, m, k.\ (s_r, i) \leq_{\beta_A} (s_p, j) \prec_{\beta_A} (s_q, k) \land \| m \|_k \subseteq \text{msg}((s_r, i)) \land \| m \|_k \subseteq \text{msg}((s_p, j))$$
  $$\land \| m \|_k \subseteq \text{msg}((s_q, k))$$
  % $s_p$ is the penetrator while $s_q$ the principal being deceived
  $\land (s_r, i)$ and $(s_q, k)$ are regular but $(s_p, j)$ is not
  $\land \text{sign}((s_r, i)) = + = \text{sign}((s_p, j))$ but $\text{sign}((s_q, k)) = -$ 

**Method:**
- % where $\pi \in \text{Pos}(\text{msg}(n))$ and $\text{msg}((s_q, k))|_{\pi} = \| m \|_k$
- $\text{orelse\_meth\_message\_encoding}(P, \beta_A, \beta_R, (s_q, k), \pi)$
- $\text{orelse\_meth\_agent\_naming}(P, \beta_A, \beta_R, (s_q, k), \pi)$
- $\text{session\_binding}(P, \beta_A, \beta_R, (s_q, k), \pi)$

Figure 6.1: The replay compound method
6.2.1 The WMF Protocol Example

An example faulty protocol violating principle 10 of Abadi and Needham is Wide-Mouth Frog (WMF): in a run of the protocol (part of) the initiator’s first message, $\langle B; T_a; K_{ab}\rangle_{K_a} \uparrow$, can be reused to mimic the result of another request that the server has acted upon, $\langle B; T_a; K_{ab}\rangle_{K_a} \downarrow$. AVISPA proves WMF fails to guarantee weak authentication of $A$ to $B$, yielding the following attack:

<table>
<thead>
<tr>
<th>WMF</th>
<th>Attack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $A \rightarrow S$ : $A; \langle B; T_a; K_{ab}\rangle_{K_A}$</td>
<td>s1:1. $A \rightarrow \text{Spy}(S)$ : $A; \langle B; T_a; K_{ab}\rangle_{K_a} \uparrow$</td>
</tr>
<tr>
<td>2. $S \rightarrow B$ : $\langle A; T_{a+d}; K_{ab}\rangle_{K_B}$</td>
<td>s2:2. $\text{Spy}(S) \rightarrow A$ : $\langle B; T_a; K_{ab}\rangle_{K_a} \downarrow$</td>
</tr>
</tbody>
</table>

Notice that part of the first message of the protocol has a similar structure with the second message, since they may be equal under instantiation. The intruder precisely takes advantage of this mal-design and thus the attack is successful. To remove the protocol flaw, it suffices to break this similarity, as shown in 6.2.3.

6.2.2 The BAN-Yahalom Protocol Example

Another example faulty protocol violating principle 10 of Abadi and Needham is the BAN-Yahalom protocol:

1 $A \rightarrow B$ : $A; N_a$
2 $B \rightarrow S$ : $B; N_b; \langle A; N_a\rangle_{K_B}$
3 $S \rightarrow A$ : $N_b; \langle A; B; K_{ab}; N_a\rangle_{K_A}; \langle B; A; K_{ab}; N_b\rangle_{K_B}$
4 $A \rightarrow B$ : $\langle A; K_{ab}; N_b; B\rangle_{K_B}; \langle N_b\rangle_{K_{ab}}$

In a run of the protocol between the initiator and the responder (part of) the server’s third message, $\langle A; B; K_{ab}; N_a\rangle_{K_a} \uparrow$, can be reused to mimic the answer of the server as if it was other run of the protocol between the responder and the initiator, $\langle A; B; K_{ab}; N_a\rangle_{K_a} \downarrow$. AVISPA proves that

---

2Recalling again the meaning of box-arrow described in Section 2.4.1, a dotted box with a $\uparrow$ denotes the origin of a cypher-text and a dotted box with a $\downarrow$ denotes a cypher-text being replayed.
BAN-Yahalom fails to guarantee weak authentication of $B$ to $A$, yielding the following attack:

\begin{align*}
  s_1 : 1 & \quad A \rightarrow \text{Spy}(B) : A; N_a \\
  s_2 : 1 & \quad \text{Spy} \rightarrow A : B; N'_a \\
  s_2 : 2 & \quad A \rightarrow \text{Spy}(S) : A; N_b; \{B; N'_a\}_K \\
  s_2 : 2 & \quad \text{Spy} \rightarrow S : A; N_a; \{B; N'_a\}_K \\
  s_2 : 3 & \quad S \rightarrow \text{Spy}(B) : \{B; A; K_{ab}; N'_a\}_K; \{A; B; R_{ab}; N_a\}_K \\
  s_1 : 3 & \quad \text{Spy} \rightarrow A : N_{bx}; \{B; K_{ab}; N'_a\}_K; N_{bx}_x \downarrow; M_x \\
  s_1 : 4 & \quad A \rightarrow \text{Spy}(B) : M_x; \{N_{bx}\}_K_{ab}
\end{align*}

Again, notice that in the third step of the protocol description, key distributor $\{A; B; K_{ab}; N_a\}_K$, which is intended for $A$ has a similar structure with key distributor $\{B; A; K_{ab}; N'_a\}_K$, which is intended for $B$. To remove the protocol flaw, it suffices to break this similarity, as shown below. Notice that $M_x$ in $s_1 : 3$ may be anything because the message in such a position is not intended to be recognized by $A$, then $A$ can accept any message.

### 6.2.3 The message-encoding Method

The message-encoding method receives the protocol description $P$, an attack $\beta_A$, a regular run of the protocol $\beta_R$, a node where the attack was exploited that matches the regular run of the protocol $n \in \beta_A \cap \beta_R$, and a position of the replayed message $\pi \in Pos(msg(n))$. For the message encoding method to be applicable, it is necessary that the replayed message $\{|m|\}_K$ be similar with respect to another $\{|m'|\}_K'$ in the protocol. To patch a protocol that suffers from improper message encoding we must break the similarity. For that, we use the concept of collision freeness, introduced in Definition 8 Section 5.1.1. We make that the new component is collision free with respect to all messages in the protocol.

The method is shown in Figure 6.2. The following proposition shows that when applied the method makes the cypher-text no longer susceptible to be re-used.

**Proposition 1** Let $P' = P_{\{|m|_K; |m'|_K\}}'$ be the corresponding revised protocol. Then $\{|m'|\}_K$ cannot be used to arm a message encoding replay attack on $P'$.

**Proof 1** $\{|m'|\}_K$ is not similar to any other component. Then, only the execution of a specific step in $P'$ may cause $\{|m'|\}_K$ to appear on the traffic, if ever. If $P$ satisfies security property $\phi$ then so will $P'$, because extra elements in $\{|m'|\}_K$, if any, are all innocuous tags.
Name: message_encoding

Input: $P \in \Sigma, \beta_A, \beta_R, n \in \beta_A \cap \beta_R, \pi \in \text{Pos}(\text{msg}(n))$

$% n$ lies on strand of deceived agent where $\text{msg}(n)|\pi = \{|m|\}_k$

$% is the message used to elaborate replay

Preconditions:

$%$ different cypher-texts cannot be distinguished:

Let $S = \text{Safe}_r$

$\exists n' \in \beta_R, \exists m', k'$. 

$(\{|m'|\}_k' \in \text{msg}(n') \land \{|m'|\}_k' \sim_S \{|m|\}_K \land \{|m'|\}_k' \neq \{|m|\}_k)$

Patch:

$%$ Break similarity of $\{|m|\}_k$:

select $m''$ such that $m \leq_A m''$, $A_0$ is a minimal set of tags, and $\{|m''|\}_k$ is collision free with $L = \{|m'|\}_K' \subseteq \text{msg}(n), n \in \beta_R$

let $P' = P_{\{|m|\}_K'\sigma/\{|m''|\}_K'\sigma}$. $%$ where $\sigma$ is a substitution $%$ to the protocol description

Effects:

$%$ The replayed message is unambiguous in the protocol

Let $S = \text{Safe}_r$

$\exists n' \in \text{bundleOf}(P'), \exists m', k'$.

$\forall \{|m'|\}_k' \subseteq \text{msg}(n') \land \{|m'|\}_k' \neq \{|m''|\}_k \sigma \rightarrow \{|m'|\}_K' \neq \{|m''|\}_K' \sigma$

---

Figure 6.2: The message_encoding method
Patching the WMF Protocol. When input WMF and the attack above, \textit{message\_encoding} method successfully repairs it returning:

\begin{align}
1. & \quad A \rightarrow S \quad : \quad A ; \{ T_a ; B \}_{K_{ab}} \\
2. & \quad S \rightarrow B \quad : \quad \{ A ; T_{a+d} ; K_{ab} \}_{K_b}
\end{align}

It is necessary to say that message concatenation, \textit{“;”}, associates to the right. That is, message \(T_a ; B ; K_{ab}\) is equal to \(T_a ; (B ; K_{ab})\). Protocol changes are enclosed within a solid box to ease reference. It is worth noting that with this modification the WMF protocol already provides weak authentication of the initiator.

Patching the BAN-Yahalom Protocol. Again, when input BAN-Yahalom and the attack above, \textit{message\_encoding} method successfully repairs it returning:

\begin{align}
1. & \quad A \rightarrow B \quad : \quad A ; N_a \\
2. & \quad B \rightarrow S \quad : \quad B ; N_b ; \{ A ; N_a \}_{K_b} \\
3. & \quad S \rightarrow A \quad : \quad N_b ; \{ A ; B ; K_{ab} ; N_a \}_{K_a} ; \{ A ; K_{ab} ; N_b ; B \}_{K_B} \\
4. & \quad A \rightarrow B \quad : \quad \{ A ; K_{ab} ; N_b ; B \}_{K_{ab}} ; \{ N_b \}_{K_{ab}}
\end{align}

With this modification the BAN-Yahalom protocol not only already provides weak authentication of the responder, but also strong authentication of the responder. Hence, the patch was so successful in that the protocol now provides all levels of authentication.

### 6.3 Patching Protocols Violating Principle 3

The \textit{agent\_naming} patching method repairs a faulty protocol containing a message without proper naming. In this class of faulty protocols the originator or the recipients of a cypher-text in one message of the protocol cannot be distinguished. This flaw violates principle 3 of Abadi and Needham, \textit{agent naming}. The adversary may exploit this vulnerability by making one cypher-text play the role of a message like the other in a different run. Hence, the agent being deceived does not have any mechanism to distinguish who originated an inward message or whom such a message is intended for. According to Syverson’s taxonomy, in this class of attack, so-called \textit{reflection attack} and \textit{deflection attack}, one (or more messages) does not indicate who a message is from, who it is intended for, or both.

To illustrate the rationale behind our \textit{agent\_naming} patching method, we shall show two faulty security protocols, namely: the NSPK and the BAN-ASRPC protocols.
6.3.1 The NSPK Protocol Example

The Needham and Schroeder public key protocol (NSPK) consists of seven steps, but four of them are concerned with the exchange of the agents public keys, whereas the remaining steps are concerned with the authentication of the participants, A and B. Thus our description of the NSPK protocol in Alice and Bob notation is restricted to the following 3 steps:

1. \( A \rightarrow B : \{N_a; A\}^{+}_{K_B} \)
2. \( B \rightarrow A : \{N_a; N_b\}^{-}_{K_A} \)
3. \( A \rightarrow B : \{N_b\}^{+}_{K_B} \)

The NSPK protocol seems right at first glance, but it is faulty. Lowe found that an intruder could impersonate one agent holding concurrently a session with another agent \[58\]. AVISPA proves that NSPK fails to guarantee secrecy of \(N_b\) and weak authentication of \(B\) to \(A\), yielding the following attack:

\[
\begin{align*}
\text{s1 : 1. } & A \rightarrow \text{Spy} : \{N_a; A\}^{+}_{K_{Spy}} \\
\text{s2 : 1. } & \text{Spy}(A) \rightarrow B : \{N_a; A\}^{+}_{K_B} \\
\text{s2 : 2. } & B \rightarrow \text{Spy}(A) : \{N_a; N_b\}^{-}_{K_B} \\
\text{s1 : 2. } & \text{Spy} \rightarrow A : \{N_a; N_b\}^{-}_{K_A} \\
\text{s1 : 3. } & A \rightarrow \text{Spy} : \{N_b\}^{+}_{K_{Spy}} \\
\text{s2 : 3. } & \text{Spy}(A) \rightarrow B : \{N_b\}^{+}_{K_B}
\end{align*}
\]

In Lowe’s attack, an identical instance of message 2, \(\{N_a; N_b\}^{+}_{K_B}\), is used in two independent runs (s1 and s2). The deceived agent, \(A\), the initiator of the first run, is the intended recipient of both instances of message 2, but she cannot distinguish who built it or sent it. Thus, while \(B\) knows that \(A\) has recently participated in a run of the protocol, he cannot tell whether \(A\) is running it apparently with him. As one can see, message 2 of the protocol description does not include the the originator agent name of the message.

6.3.2 The BAN-ASRPC Protocol Example

The BAN Andrew Secure RPC (BAN-ASRPC) protocol consists of 4 steps that aims to distribute a session key and to establish authentication between the two participants. The Alice and Bob notation of this protocol is as follows:

1. \( A \rightarrow B : A; N_a \)
2. \( B \rightarrow A : \{N_a; K_{ab}'\}^{+}_{K_{ab}} \)
3. \( A \rightarrow B : \{N_a\}^{+}_{K_{ab}} \)
AVISPA proves that BAN-ASRPC protocol fails to guarantee weak authentication of A to B, yielding the following attack:

\[
\begin{align*}
& s_1 : 1 \quad A \rightarrow \text{Spy}(B) : A; N_a \\
& s_2 : 1 \quad \text{Spy}(B) \rightarrow A : B; N_a \\
& s_2 : 2 \quad A \rightarrow \text{Spy}(B) : \{\{N_a; K'_{ab}\}_{K_{ab}}\} \uparrow \\
& s_1 : 2 \quad \text{Spy}(B) \rightarrow A : \{\{N_a; K'_{ab}\}_{K_{ab}}\} \downarrow \\
& s_1 : 3 \quad A \rightarrow \text{Spy}(B) : \{\{N_a\}_{K'_{ab}}\} \uparrow \\
& s_2 : 3 \quad \text{Spy}(B) \rightarrow A : \{\{N_a\}_{K'_{ab}}\} \downarrow
\end{align*}
\]

As we can see in this protocol two messages are replayed in two different runs, \(\{\{N_a; K'_{ab}\}_{K_{ab}}\} \downarrow\) and \(\{\{N_a\}_{K'_{ab}}\} \downarrow\). The deceived agent, A, the initiator of the first run \(s_1\), is the responder in the second run (playing as the responder), getting the same message that he created in the first run. Agent A cannot distinguish who built it or sent it. Thus, A does not know that B is not participating in the run and besides that, it is the spy who is playing with the messages because such messages lack of proper naming.

### 6.3.3 The agent naming Method

Similar to the session binding method, the agent naming patching method receives the protocol description \(P\), an attack \(\beta_A\), a regular run of the protocol \(\beta_R\), a node where the attack was exploited that matches the regular run of the protocol \(n \in \beta_A \cap \beta_R\), and the position of the replayed message \(\pi \in Pos(msg(n))\).

For the agent naming method to be applicable, it is necessary that one or more participant names involved in the exchange of the replayed message, \(\text{Partners}(\{m\}_k)\), cannot be inferred from the encoding itself using function \(\text{Agents}\), introduced in section 5.1.2.

To patch a protocol containing a message without proper naming we must include those agent names that have been omitted in the design. This addition process, however, attempts to avoid the introduction of redundancies and the introduction of erroneous message encoding that could yield a similarity. To avoid the introduction of message similarity, we look back at the protocol description. We need to ensure that the new message component or even the entire new message does not have a structure that is similar to another one in the protocol. In case of the similarity of the messages cannot be broken we introduce tags, as the last resource. To do that, we make use of the concept of collision freeness, which is the base of the message encoding patching method. We work so that the new component is collision free with respect to all messages in the protocol.

The method is shown in Figure 6.3. The following proposition shows that when applied the method makes the cypher-text no longer susceptible to be re-used.
**Proposition 2** Let $P'$ and $\{m''\}_k$ be the revised protocol and cypher-text. Then, $\{m''\}_k$ can not be used to arm a naming replay attack.

**Proof 2** Let $P(R, \bar{x})$ denote the set of strands of role $R$ in $P$ instantiated with parameters $\bar{x}$. Effects guarantee that $\forall R \in \text{Partners}(\{m''\}_k), P'(R, \bar{x}) \subseteq P(R, \bar{x})$, because the parameters agree at least on the associated names of the correspondents of $\{m''\}_k$. It follows that the cypher-text cannot be used to arm a naming replay. \qed

---

**Name:** agent_naming

**Input:** $P, \beta_A, \beta_R, n \in \beta_A \cap \beta_R, \pi \in \text{Pos}(\text{msg}(n)) \% \text{ with } \text{msg}(n)|_\pi = \{m\}_K$

**Preconditions:**

$$\bigvee_{A \in \text{Partners}(\{m\}_k)} A \notin \text{Agents}(\{m\}_k)$$

**Patch:**

Select $m''$ such that:

$m \leq_I m''$ with $I = \text{Partners}(\{m\}_k) \setminus \text{Agents}(\{m\}_k)$

$\{m''\}_k$ is collision free with $\{m'\}_K$ wrt. Safe, $\leq \text{msg}(n), n \in \beta_R$)

let $P' = P |_{\{m\}_k/\text{subst}}$, $\%$ where $\sigma$ is a substitution factor $\%$ to the protocol description

**Effects:**

$$\text{Partners}(\{m''\}_k) = \text{Agents}(\{m''\}_k)$$

Figure 6.3: The *agent_naming* method

---

**Patching the NSPK protocol.** When input the NSPK protocol described above, *agent_naming* patching method, Figure 6.3, adds the name of the agent originating message 2, $B$ in this case, arriving at Lowe’s fix:

1. $A \rightarrow B : \{Na, A\}_{K_B^+}$
2. $B \rightarrow A : \{[B], Na, Nb\}_{K_A^+}$
3. $A \rightarrow B : \{Nb\}_{K_B^+}$

With this modification the NSPK protocol not only already provides weak authentication to the initiator, but also strong authentication. Hence, the patch was also so successful in that the protocol now provides all levels of authentication.
**Patching the BAN-ASRPC protocol.** Again, when input the BAN-ASRPC protocol, *agent_naming* patching method adds the corresponding agent names in messages 2 and 3, pinpointing the agent originating the message and whom it is intended for:

1. $A \rightarrow B : A; Na$
2. $B \rightarrow A : \{[A; B; N_a; K'_{ab}]\}_{K_{ab}}$
3. $A \rightarrow B : \{[A; B; N_a]\}_{K'_{ab}}$

With this modification the BAN-ASRPC protocol not only already provides weak authentication to the initiator, but also strong authentication. Hence, the patch was also so successful in that the now provides all levels of authentication.

### 6.4 Patching Protocols Violating Principles 6—10

The *session_binding* method deals with faulty protocols that contain a message which cannot be associated with a particular protocol run. This flaw violates principle 10 of Abadi and Needham. It also violates principles 6—8, the protocol does not guarantee association or temporal succession. An attack exploiting this flaw, called *replay protection*, causes an agent to consider that another is trying to set up a simultaneous session, when he is not [61]. According to Syverson’s taxonomy of replay attacks, this kind of attack corresponds to *classic replays*, a subclass of external attacks where protocol runs need not be contemporaneous.

Two example protocols subject to this type of attack, because none satisfies *strong authentication* of $B$ to $A$, are the revisited WMF protocol of section 6.2.1 and the Denning-Sacco Shared Key (DSSK) protocol, which we study in the next two sections.

#### 6.4.1 The Revisited WMF Protocol Example

The (revised version of) WMF protocol (6.1) is subject to classic replay attack, if demanded to fulfill *strong authentication* of $B$ to $A$. AVISPA finds the *replay protection* attack. The attack is as follows:

\[ s1 : 1. \quad A \rightarrow S : A, [\{T_{a+d}; B, K_{ab}\}\}_{K_a} \]
\[ s1 : 2. \quad S \rightarrow B : \{[A, T_{a+d}, K_{ab}]\}_{K_b} \uparrow \quad (6.2) \]
\[ s2 : 2. \quad \text{Spy}(S) \rightarrow B : \{[A, T_{a+d}, K_{ab}]\}_{K_b} \downarrow \]

This attack succeeds because $T_{a+d}$ provides only partial authentication of $A$. The origin of the error has nothing to do with message encoding or agent naming, but with a time reference. As you can see, the attack is not interleaving because sessions 1 and 2 ($s1$ and $s2$) do not overlap in
execution, as required for attacks in message encoding and agent naming methods. In this case, the resending of message $\{A, T_{a+d}, K_{ab}\}_{K_b}$ needs not be contemporaneous.

### 6.4.2 The Denning-Sacco Shared Key (DSSK) Protocol Example

The DSSK protocol aims to establish a session key $K_{ab}$ and to provide authentication between $A$ and $B$. The protocol is as follows:

1. $A \rightarrow S : A; B$
2. $S \rightarrow A : \{\{B; K_{ab}; T_s; \{\{B; K_{ab}; A; T_s\}_{K_b}\}_{K_a}\}\}_{K_a}$
3. $A \rightarrow B : \{\{B; K_{ab}; A; T_s\}_{K_b}\}_{K_a}$

This protocol also fails to fulfill strong authentication of $B$ to $A$. The attack that AVISPA finds on this protocol is as follows:

$$s_1 : 1. \quad A \rightarrow S : A; B$$
$$s_1 : 2. \quad S \rightarrow A : \{\{B; K_{ab}; T_s; \{\{B; K_{ab}; A; T_s\}_{K_b}\}_{K_a}\}\}_{K_a}$$
$$s_1 : 3. \quad A \rightarrow B : \{\{B; K_{ab}; A; T_s\}_{K_b}\}_{K_a}$$
$$s_2 : 3. \quad \text{Spy}(A) \rightarrow B : \{\{B; K_{ab}; A; T_s\}_{K_b}\}_{K_a}$$

Upon reception of message 3, $B$ knows that $A$ has participated in the protocol because she is the only one that could have sent such a message after having decrypted message 2. However, $B$ does not know whether this was done recently.

Notice that both the DSSK and the WMF protocol prescribe the responder, $B$, to react upon an unsolicited test. An unsolicited test is a weaker type of authentication test, where if a term $\{m\}_k$ is received, and $k$ is in Safe, then $\{m\}_k$ originated on some regular strand. After all, it originated somewhere, and that can not have been a penetrator strand if $k \in \text{Safe}$. We know only that the regular node originating $\{m\}_k$ is before the node on which it is received [45]. In other words, upon reception of term $\{m\}_k$, it does not contain neither kind of information that the receiver can verify like an authentication test.

### 6.4.3 The session_binding Method

SHRIMP is equipped with the session_binding method which introduces a nonce-flow requirement to fix this kind of flaw [1, 77, 61, 45] (c.f. principle 7.) This requirement is carried out by transforming the unsolicited test (explained above) into a solicited one. We name solicited test to the two main types of authentication test: outgoing test and incoming test (see section 4.4.4 of Chapter 4 for details about these terms). So, session_binding method, shown in Figure 6.4, describes in the preconditions the requirements of an unsolicited test.
In order to transform the unsolicited test into a solicited one the session-binding method calls two patching strategies: handshake and nonce_flow. As already explained in section 4.3.1, orelse_meth attempts the application of handshake and if that fails then attempts the application of nonce_flow. In what follows, we explain in detail these strategies and we support their application order. Let us consider first nonce-flow.

### 6.4.3.1 Nonce_flow

The nonce-flow requirement is introduced via the following transformation rules, called nonce_flow and tried to be applied in the order of appearance:

\[
A \xrightarrow{m_1} B \xrightarrow{\[m_2\]} C \quad \Leftrightarrow \quad B \xrightarrow{\[m_1\]} C \quad \Leftrightarrow \quad A \xrightarrow{m_1} C \xrightarrow{\{C; N_e; h(m_1)\}_{K_b}^+} B \xrightarrow{\{C; N_e; h(m_2)\}_{K_b}^-} C
\]

where:

- \([M]\) marks the message at which the replay is carried out;
- \([M]\) marks changes in the protocol message or in the protocol step;
- \(A \xrightarrow{[m]} B\) denotes the step at which the replay is carried out.
- \(A \xrightarrow{m_1} B \xrightarrow{m_2} C\) abbreviates two consecutive protocol steps:

\[
q. A \rightarrow B : m_1 \quad q + 1. B \rightarrow C : m_2
\]
\begin{itemize}
  \item \( A \overset{m}{\rightarrow} B \leadsto A \overset{m'}{\rightarrow} \overline{C} \overset{m''}{\rightarrow} B \) denotes the insertion of a protocol step, demanding the participation of \( C \) as intermediary of \( A \) and \( B \). Notice that the strands ought to be modified as follows:

  1. for \( A \) and \( B \), \( \exists n_a \in s_a, n_b \in s_b. \text{msg}(n_a) = \text{msg}(n_b) = m \), so we shall have \( \text{msg}(n_a) = m' \) and \( \text{msg}(n_b) = m'' \); and

  2. for \( C \), two nodes, \( n_{c_1} \) and \( n_{c_2} \), are inserted such that \( \text{msg}(n_{c_1}) = m' \) and \( \text{msg}(n_{c_2}) = m'' \) with \( \text{sign}(n_{c_1}) = - \neq \text{sign}(n_{c_2}) \).

  The rule identifies where these nodes are to be inserted: c.f. \( C \)'s participation, previous to the attack. Any other transformation is handled similarly.

  \item \( h(m) \) denotes a one-way, collision-resistant hash function, which is used to tie each test component to the current run.

  Notice that if \( A = C \) then the first two steps of the right-hand side of the first rule merge. Also notice that applying \texttt{nonce\_flow} without considering the structure of \( m_1 \) or \( m_2 \) may add unnecessary components. For that, we introduce function \( \text{shrnk}(m, m') \), which reduces in size the compound message \( m; m' \) removing those unnecessary components. So, we shrink the transformation messages \( m_1; \{C; N_c; h(m_1)\}_{K_b^+} \) (\( \text{shrnk}(m_1, \{C; N_c; h(m_1)\}_{K_b^+}) \)) and \( m_2; \{C; N_c; h(m_2)\}_{K_b^-} \) (\( \text{shrnk}(m_2, \{C; N_c; h(m_2)\}_{K_b^-}) \)) as follows:

  \[
  \text{shrnk}(m, m') \overset{\text{def}}{=} \begin{cases} 
    m; m' & \text{if } \text{shrnk}_0(m, m') = m \\
    \text{shrnk}_0(m, m') & \text{otherwise}
  \end{cases}
  \]

  where

  \[
  \begin{align*}
  \text{shrnk}_0(m_1, m) & \overset{\text{def}}{=} m_1 \quad \text{if } m_1 \text{ is atomic} \\
  \text{shrnk}_0(m_1; m_2, m') & \overset{\text{def}}{=} \text{shrnk}_0(m_1, m'); \text{shrnk}_0(m_2, m') \\
  \text{shrnk}_0(\{m\}_{K_b}, \{C; N_c; m'\}_{K_b}) & \overset{\text{def}}{=} \{C; N_c; m\}_{K_b} \\
  \text{shrnk}_0(\{m\}_{K_b^+}, \{C; N_c; m'\}_{K_b^+}) & \overset{\text{def}}{=} \{\text{shrnk}(m, \{C; N_c; h(m)\}_{K_b^+})\}_{K_b^+} \\
  \text{shrnk}_0(\{m\}_{K_b^-}, \{C; N_c; m'\}_{K_b^-}) & \overset{\text{def}}{=} \{\text{shrnk}(m, \{C; N_c; h(m)\}_{K_b^-})\}_{K_b^-}
  \end{align*}
  \]

  where we assume that \( C \) originates the message \( m_1; \{C; N_c; h(m_1)\}_{K_b^+} \) (respectively \( m_2; \{C; N_c; h(m_2)\}_{K_b^-} \), the challenger, and \( B \) is the recipient, the champion.

  \textbf{Proposition 3} Let \( P' \) be the revised version of the protocol and let \( N_c \) be the nonce introduced by an application of \texttt{nonce\_flow} on the replay of \( m \). If \( N_c \in \text{unique} \), then \( m \) cannot be used to elaborate a replay.
Proof 3 Take the first rule, so \( m = s_1; \). Let \( \text{node}_a^+(m') \) (respectively \( \text{node}_a^-(m') \)) denote the positive (respectively negative) node of strand \( A \) at which message \( m' \) is sent (respectively received), then if \( K_b^- \) is safe:

\[
\text{node}_c^+(m_1; \{C; N_c; h(m_1)\} \_K^+_b) \Rightarrow + \text{node}_c^-(m_2; \{C; N_c; h(m_2)\} \_K^-_b)
\]

is an outgoing test for \( N_c \) in \( \{C; N_c, h(m_1)\} \_K^+_b \). Then by definition 1 of page ?? only a regular participant must have been responsible for \( N_c \) to exit the cipher-text \( \{C; N_c; h(m_1)\} \_K^+_b \) and then enter to \( \{C; N_c; h(m_2)\} \_K^-_b \). Any occurrence of \( m_i \) \((i = 1, 2)\) is thus tied via \( h(m_i) \) to a unique test and so the result follows.

6.4.3.2 Handshake

Notice that applying nonce_flow when the server is involved in the replay may yield a clumsy protocol. This is because the protocol would involve too many server participations and provide guarantees to the server rather than to the participants. We get around this situation by applying a very specific patching strategy, due to Lowe [61], which consists of making the participants handshake. The handshake messages are cyphered using the session key, similar to a key confirmation step. A key confirmation step is adding one or more protocol steps in order to assure to the rightful participants, in a key-establishment protocol, with the aim that the intended recipient(s) possess the same (correct) shared key. nonce_flow is thus attempted only if the following rules, called handshake, are not applicable:

\[
S \xrightarrow{m(k)} X_i \quad \Leftrightarrow \quad S \xrightarrow{m(k)} X_i \quad \Leftrightarrow \quad X_k \xrightarrow{m^'} S \xrightarrow{m(k)} X_i
\]

where:

- \( T = \{T'; h(m(k))\} \_K_i \) and where the last two steps of the protocol structure on the right-hand side are applied for all \( j \neq i \) and so are actually rounds of messages.

- Notice that for handshake to be applicable the message \( m \) should carry a session key, \( m(k) \).

This patching strategy is known to be susceptible to a known-key attack, [37], but the insertion of the timestamp, \( T' \), makes it very difficult for an adversary to timely carry out the replay. A known-key attack is an attack whereby, once getting knowledge of a session key, the adversary is able, if passive, to compromise keys of other sessions or, if active, to impersonate one of the (honest) protocol parties. The following propositions show that when applied the method makes the cypher-text no longer susceptible to be re-used.
**Proposition 4** Let $P'$ be the revised version of the protocol and let $N_j$ be the nonce introduced by an application of handshake on the replay of $m$. If $N_j \in \text{unique}_s$ then $m$ cannot be used to elaborate a replay.

**Proof 4 (outline)** Take the second rule, so $m = \overline{m(K)}$. Let $\text{node}^+_a(m')$ (respectively $\text{node}^-_a(m')$) denote the positive (respectively negative) node of strand $A$ at which message $m'$ is sent (respectively received), then if $K$ is safe:

$$\text{node}^+_c(\{X_i; X_j; N_j\}_K) \Rightarrow \text{node}^-_c(\{X_j; N_j\}_K)$$

is an outgoing test for $N_j$ in $\{X_i; X_j; N_j\}_k$. Then by proposition 19 of [45] only a regular participant must have been responsible for $N_j$ to exit the cypher-text $N_j$ in $\{X_i; X_j; N_j\}_k$ and then enter to $\{X_j; N_j\}_k$.

**Patching the WMF protocol.** When input the WMF protocol (revisited version) (6.1) and its attack (6.3), our method yields:

1. $A \rightarrow S : A; \{T_a, B, K_{ab}\}_K$  
2. $S \rightarrow B : \{A, T_{a+d}, K_{ab}\}_K$  
3. $B \rightarrow A : \{A, B, N_b\}_K$  
4. $A \rightarrow B : \{A, B, f(N_b)\}_K$

This protocol is the same as that one proposed by Lowe in [61].

**Patching the DSSK protocol.** Two consecutive applications of session binding method, on input DSSK yields:

1. $A \rightarrow B : A; B$  
2. $B \rightarrow S : A; B; \overline{\{N_b; A\}_K}$  
3. $S \rightarrow A : \{B; K_{ab}; T_{s}; \overline{\{N_b; B; K_{ab}; A; T_{s}\}_K}\}_K$  
4. $A \rightarrow B : \{N_b; B; K_{ab}; A; T_{s}\}_K; \{A; B; N_b\}_K$  
5. $B \rightarrow A : \overline{\{B; N_b\}_K}$

Step 2, together with $N_b$, is inserted in the first application, preventing a replay protection attack on $B$, while the handshake à la Lowe is inserted in the second one, preventing a replay protection attack on $S$. 
6.5 Conclusions

In this chapter we have presented SHRIMP, which is formed by a general compound method replay and three patching methods message_encoding, agent_naming and session_binding. Our general method is about replay attack because we have found in the literature that the main source of attacks in security protocols is precisely this class.

The replay compound method, as already described, calls three patching methods. Each of them deals with a general subclass of replay attacks. The interesting part of our methods is that they only modify the part(s) of the protocol where the flaw has been originated, patching the security property being violated. However, sometimes we could require that the protocol provides more security properties, in that case, it is necessary to carry out more than once the development cycle, which was explained in section 4.2.

With the presentation of SHRIMP’s methods, we finish our discussion on the entire patching faulty protocols plan. It now remains to present experimental results, compare our method with rival techniques and provide directions for further work.
7. Implementation and Results

This chapter describes implementation aspects and summarizes the results produced by running SHRIMP on a test set of faulty security protocols. Both positive and negative results are clearly stated in order to point out outstanding performance and limitations of the patching framework.

7.1 Implementation

7.1.1 AVISPA: The Verification Tool

As explained in section 4.2, SHRIMP relies on existing state-of-the-art tools both to analyze an (intermediate) protocol and, if the protocol is flawed, to find a counterexample. It then analyzes the protocol and the counterexample to pinpoint the faulty steps of the protocol and synthesizes appropriate changes to fix them. This yields an improved version of the protocol that should be analyzed and potentially patched again until no further flaws can be detected. As we can see, we use a verification tool twice: to discover that the protocol is faulty and to verify the new version of the protocol after patching it.

We have decided to use AVISPA because it is considered to be the state of the art in security protocol verification. This tool has been able to find all attacks in the Clark and Jacob library [30] in just a few seconds of CPU time. AVISPA also has been applied to industrial protocols, documented in its own library (AVISPA library), such as IKE and H.530 protocols.

Roughly speaking, for verifying a protocol in this tool, one must formulate the protocol and the
properties to be verified (e.g. secrecy or/and authentication) in a high-level protocol specification language (HLPSL). Avispa tool contains a translator called \textit{hlpsl2if} that automatically translates a HLPSL protocol specification into a low-level intermediate format \textit{IF}. The IF format is the verification language that Avispa uses in order to prove the security properties of a protocol. Avispa find attacks using one of the four different back-ends search engines: OFMC, CL-AtSe, SATMC and TA4SP.

In this thesis we have used mainly OFMC [15] and CL-AtSe [87] because they have shown to timely provide verification results.

Appendix A gives a technical description of Avispa, including its specification language, HLPSL, and its verification language, IF.

\section*{7.1.2 SHRIMP Implementation}

To implement SHRIMP we have constructed two specification languages: an informal Alice and Bob language and a formal language based in part in Paulson’s inductive approach [74]. The Alice and Bob language provides a convenient, human readable, an easy to use notation. Yet, it is enough to express a regular run of a protocol and a counterexample as specified and output by AVISPA respectively. For more details see appendix B.

Although Alice and Bob language is convenient and human readable, it is insufficient to reason about more complicated symbols introduced in chapter 5. To this end, we have developed a script \texttt{abn2fn.py} which was written in Python\textsuperscript{1} which translates a script containing the specification of a protocol and one of its counterexamples (in Alice and Bob language) into the formal language that SHRIMP’s meta-logic implementation understands.

SML was used in the construction of SHRIMP’s meta-logic.\textsuperscript{2} This implementation consists of a list of sml files where each of them represents a list of symbols (the meta-logic) in order to represent \textit{roles}, \textit{protocols}, \textit{counterexamples} and symbols to represent the preconditions and the suitable patches to the faulty protocols according to the preconditions. For more details about SHRIMP’s meta-logic implementation see appendix B. These all program files ought to be available at the tool homepage, \url{http://homepage.cem.itesm.mx/raulm/shrimp/}.

\textsuperscript{1}We have chosen Python because it is an interpreted programming language fully dynamically typed and uses automatic memory management; it is developed as an open source project, managed by the non-profit Python Software Foundation, and, the most important thing, is available for free from the project website.

\textsuperscript{2}We have used Standard ML – sometimes called SML, or just ML, because it supports functional programming, where programs consist of functions operating on simple data structures. Functional programming is ideal for many aspects of problem solving like verification, optimization, and parallel programs. In addition, its distribution is also free.
7.1.3 Development Methodology

The main goal of this thesis is to automate the correction of faulty security protocols. Determining whether or not this research has been achieved raises the question: how the performance of SHRIMP can be evaluated. To determine this, we have used the same methodology used in the PhD Monroy’s thesis [71].

There are different bases upon which the performance of SHRIMP can be evaluated. As we use patch planning we will use the same strategies used to assess proof planning; according to [23] and clearly explained in [71], the most important evaluation criteria with respect to proof planning are generality and expectancy. Generality means usefulness in a large number of examples amongst the intended proof family. Expectancy consists in the ability of predicting the successful outcome of a plan: there should be a story about why the technique works. Once explained what generality and expectancy means, we relate the SHRIMP’s performance using these concepts.

Chapter 3 and 6 have provided, implicitly, a discussion of expectancy: they include an account as to why the patch planning should work. Moreover, they have initiated already our attempt to evidence the generality of the patch planning, by showing its behavior on a few worked examples. As far as generality is concerned, this chapter takes a step further: it provides empirical evidence in the form of results obtained from testing the plan on a set of examples. Naturally, for this to be meaningful, the test set must be representative. Now, in order to show the representativeness of the test set, we now outline the methodology used in the invention of our patching methods.

Firstly, we distinguished two sets of example faulty security protocols, similarly as in [71]:

Development: this class consists of a few protocol examples that were used for designing the patching method. For this to make sense, the development faulty protocol must have a similar flaw (at least be subject to the same kind of attack) and they should be different in size, type of cryptography used, participants (involving a trusted server or not), etc. The method was tested by hand on the development set before implementation;

Testing: this class contains example protocols used for testing the robustness of the method, and was considered only when the development was complete. The testing set includes the development set but also contains examples that were not used during development.

Secondly, we attempted to keep the development protocol examples as dissimilar as possible.

Thirdly, we gathered examples from different sources, e.g., books, research reports, etc., and from the Clark and Jacob library, which turned out to be the definite source.

The development set included 8 protocols, as shown in the following table:

3It is, to some extent, unrealistic to test a method using examples outside the domain which it was especially designed to handle. On the one hand, failure could be easily explained by appealing to specialization; on the other hand, success could be potentially misused to claim generality.
Table 7.1: Protocol Examples Used for Development

<table>
<thead>
<tr>
<th>Protocol Name</th>
<th>Cryptography</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAN ASRPC</td>
<td>Symmetric</td>
<td>C&amp;J</td>
</tr>
<tr>
<td>DSSK</td>
<td>Symmetric</td>
<td>C&amp;J</td>
</tr>
<tr>
<td>NSPK</td>
<td>Asymmetric</td>
<td>C&amp;J</td>
</tr>
<tr>
<td>WMF</td>
<td>Symmetric</td>
<td>C&amp;J</td>
</tr>
<tr>
<td>WMF Revisited</td>
<td>Symmetric</td>
<td>C&amp;J</td>
</tr>
<tr>
<td>BAN Yahalom</td>
<td>Symmetric</td>
<td>C&amp;J</td>
</tr>
<tr>
<td>A. DH</td>
<td>Asymmetric</td>
<td>Invented</td>
</tr>
<tr>
<td>2steps SK</td>
<td>Symmetric</td>
<td>Invented</td>
</tr>
</tbody>
</table>

In the next section, we recapitulate both the experimental work and the results obtained throughout our investigations.

7.2 The Protocol Set and Results

SHRIMP is available upon request, by sending e-mail to (juan.pimentel@itesm.mx) and online via www.homepage.cem.itesm.mx/juan.pimentel/pub/shrimp.htm. This section contains a compendium of the patching examples used for testing SHRIMP.

7.2.1 The Protocol Set

A number of example faulty protocols were used as a basis for developing and testing SHRIMP. Table 7.2 summarizes our set of protocols used for development and testing SHRIMP. In the table protocols marked with “*” are part of our development set, the remainder ones are part of our testing set. Protocols annotated with (+) aim to provide authentication; the remainder protocols aim to achieve both authentication and session key distribution. In order to keep that our testing set as dissimilar as possible we took into account the following characteristics: 17 protocols use symmetric cryptography and the remainder ones use asymmetric one; 13 protocols include the participation of a trusted server and the remainder ones do not.

As you can see most protocols were obtained from Clark and Jacob (C&J) library, only 4 were invented by us with the aim to have a bigger universe of protocols. The Clark and Jacob (C&J) library, [30], is available online via http://www.lsv.ens-cachan.fr/spore/.
<table>
<thead>
<tr>
<th>Protocol Name</th>
<th>Cryptography</th>
<th>S</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAN ASRPC*</td>
<td>Symmetric</td>
<td></td>
<td>C&amp;J</td>
</tr>
<tr>
<td>DSSK*</td>
<td>Symmetric</td>
<td>√</td>
<td>C&amp;J</td>
</tr>
<tr>
<td>NSPK* (+)</td>
<td>Asymmetric</td>
<td></td>
<td>C&amp;J</td>
</tr>
<tr>
<td>WMF*</td>
<td>Symmetric</td>
<td>√</td>
<td>C&amp;J</td>
</tr>
<tr>
<td>WMF Revisited*</td>
<td>Symmetric</td>
<td>√</td>
<td>C&amp;J</td>
</tr>
<tr>
<td>BAN Yahalom*</td>
<td>Symmetric</td>
<td>√</td>
<td>C&amp;J</td>
</tr>
<tr>
<td>ASRPC</td>
<td>Symmetric</td>
<td></td>
<td>C&amp;J</td>
</tr>
<tr>
<td>CCITTX.509(1)(+)</td>
<td>Asymmetric</td>
<td></td>
<td>C&amp;J</td>
</tr>
<tr>
<td>CCITTX.509(3)(+)</td>
<td>Asymmetric</td>
<td></td>
<td>C&amp;J</td>
</tr>
<tr>
<td>NSSK</td>
<td>Symmetric</td>
<td>√</td>
<td>C&amp;J</td>
</tr>
<tr>
<td>DSPK</td>
<td>Asymmetric</td>
<td>√</td>
<td>C&amp;J</td>
</tr>
<tr>
<td>Kao Chow A. v1</td>
<td>Symmetric</td>
<td>√</td>
<td>C&amp;J</td>
</tr>
<tr>
<td>Neumann-Stubblebine</td>
<td>Symmetric</td>
<td>√</td>
<td>C&amp;J</td>
</tr>
<tr>
<td>KSL</td>
<td>Symmetric</td>
<td>√</td>
<td>C&amp;J</td>
</tr>
<tr>
<td>Otway-Rees</td>
<td>Symmetric</td>
<td>√</td>
<td>C&amp;J</td>
</tr>
<tr>
<td>BAN OR</td>
<td>Symmetric</td>
<td>√</td>
<td>C&amp;J</td>
</tr>
<tr>
<td>Splice/AS</td>
<td>Asymmetric</td>
<td></td>
<td>C&amp;J</td>
</tr>
<tr>
<td>CJ Splice</td>
<td>Asymmetric</td>
<td></td>
<td>C&amp;J</td>
</tr>
<tr>
<td>HC Splice</td>
<td>Asymmetric</td>
<td></td>
<td>C&amp;J</td>
</tr>
<tr>
<td>WMF++</td>
<td>Symmetric</td>
<td>√</td>
<td>Invented</td>
</tr>
<tr>
<td>ASRPC prune</td>
<td>Symmetric</td>
<td></td>
<td>Invented</td>
</tr>
<tr>
<td>WLM</td>
<td>Symmetric</td>
<td>√</td>
<td>C&amp;J</td>
</tr>
<tr>
<td>Woo-Lam Pi (+)</td>
<td>Symmetric</td>
<td></td>
<td>C&amp;J</td>
</tr>
<tr>
<td>A. DH*</td>
<td>Asymmetric</td>
<td></td>
<td>Invented</td>
</tr>
<tr>
<td>2steps SK*</td>
<td>Symmetric</td>
<td></td>
<td>Invented</td>
</tr>
</tbody>
</table>

Table 7.2: Protocols Examples Used for Development and Testing
7.2.2 Results

Table 7.3 summarizes our results. We considered 36 experiments, of which 20 involve protocols borrowed from the Clark-Jacob library; 4 are variants of some of these protocols (annotated with \(\star\)); and 12 are protocols output by SHRIMP, a next-generation of an input protocol. Next-generation protocols are shown in a separate row within the associated entry.

Each row displays the result of testing a protocol against a (hierarchical) collection of properties: secrecy, \(s\), weak authentication of the initiator, \(wa_i\) (respectively responder, \(wa_r\)) and strong authentication of the initiator, \(sa_i\) (respectively responder \(sa_r\)), where \(wa_i < sa_i\) (respectively \(wa_r < sa_r\)). The table separates the verification results for the original protocol, before, and the mended protocol, after, as output by SHRIMP. The value of the field that exists at the intersection between a protocol \(P\) and a property \(\phi\) might be either T, meaning \(P\) satisfies \(\phi\), F, meaning \(P\) does not satisfy \(\phi\), or X, meaning this property was not tested (because \(P\) was not expected to satisfy it.) Column \(M\) specifies the patch method that was applied to modify each faulty protocol: message (E)ncoding, agent (N)aming or session (B)inding. For the latter method, \(B_1\) refers to rule nonce_flow and \(B_2\) to rules handshake. In all our experiments, the application of a patch method yielded a revised protocol able to satisfy the security property that the original one did not. Whenever applicable, each mended protocol was then further requested to satisfy the remaining, stronger properties in the hierarchy, thus explaining why some entries have several runs. Note that in the discovery of some attacks we had to specify in AVISPA the possibility of losing a session key (annotated with \(\dagger\)) in order to have a bigger universe of protocol attacks.

SHRIMP is thus able to identify a flaw and a successful candidate patch in 33 faulty protocols out of 36. Of these experiments, it applied 12 times agent_naming, 4 times message_encoding, 9 times rules nonce_flow and 8 times handshake. The protocols SHRIMP fails to fix, namely: Neumann-Stubblebine, Otway-Rees and Woo-Lam Pi, are all susceptible to the type flaw subclass of replay attack. In this case, the problem is not SHRIMP, but the logic we chose (strand spaces, which considers typed messages). It would be misleading to dismiss message encoding on account of the few protocols it patched. This is because while applying the other methods we use it to ensure the patch did not incur in an infringement of principle 10.

Tables 7.4, 7.5, 7.6 and 7.7 show the output of SHRIMP. For each protocol, they describe the new specification. Changes are enclosed in the new description of the protocol.

Table 7.9 shows the total elapsed verification time (TEVT) and the back-end (OFMC or CL-Atse) used to verify the new protocol description. Our research also reports the HLPSL formulation of 21 patched security protocols not available in the AVISPA library. Our experiments were carried

\footnote{As mentioned in section 2.1, the Clark-Jacob library comprehends 50 protocols, 26 out of which are known to be faulty. So our validation test set contains all but 6 of these security protocols, which are not susceptible to a replay attack.}
<table>
<thead>
<tr>
<th>Protocol</th>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>s  wa_i  sa_i  wa_r  sa_r</td>
<td>M  s  wa_i  sa_i  wa_r  sa_r</td>
</tr>
<tr>
<td>ASRPC</td>
<td>T  T  T  T  F</td>
<td>B_1  T  T  T  T  T</td>
</tr>
<tr>
<td>BAN ASRPC</td>
<td>T  F  X  X  X</td>
<td>N  T  T  T  X  X</td>
</tr>
<tr>
<td>CCITT X.509(1)</td>
<td>T  F  X  X  X</td>
<td>N  T  T  T  X  X</td>
</tr>
<tr>
<td></td>
<td>T  T  F  X  X</td>
<td>B_1  T  T  T  T  T</td>
</tr>
<tr>
<td>CCITT X.509(3)</td>
<td>T  F  X  T  T</td>
<td>N  T  T  T  T  T</td>
</tr>
<tr>
<td>DSSK</td>
<td>T  T  F  X  X</td>
<td>B_1  T  T  T  T  T</td>
</tr>
<tr>
<td></td>
<td>T  T  T  T  F</td>
<td>B_2  T  T  T  T  T</td>
</tr>
<tr>
<td>NSSK</td>
<td>T  T  X  T  F</td>
<td>B_1  T  T  X  T  T</td>
</tr>
<tr>
<td></td>
<td>T  T  F  T  T</td>
<td>B_2  T  T  T  T  T</td>
</tr>
<tr>
<td>DSPK</td>
<td>T  F  X  X  X</td>
<td>N  T  T  X  X  X</td>
</tr>
<tr>
<td></td>
<td>T  T  F  X  X</td>
<td>B_2  T  T  T  X  X</td>
</tr>
<tr>
<td>Kao Chow A.  v1</td>
<td>T  T  F  T  T</td>
<td>B_2  T  T  T  T  T</td>
</tr>
<tr>
<td>KSL</td>
<td>T  F  X  X  X</td>
<td>E  T  T  T  T  T</td>
</tr>
<tr>
<td>NSPK</td>
<td>F  X  X  T  T</td>
<td>N  T  T  T  T  T</td>
</tr>
<tr>
<td>BAN OR</td>
<td>T  F  X  X  X</td>
<td>N  T  T  X  X  X</td>
</tr>
<tr>
<td>Splice/AS</td>
<td>T  X  X  F  X</td>
<td>N  T  T  T  X  X</td>
</tr>
<tr>
<td></td>
<td>T  T  F  T  X</td>
<td>B_2  T  T  T  T  T</td>
</tr>
<tr>
<td>CJ Splice</td>
<td>T  F  X  T  T</td>
<td>B_2  T  T  T  T  T</td>
</tr>
<tr>
<td>HC Splice</td>
<td>T  X  X  F  X</td>
<td>N  T  X  X  T  X</td>
</tr>
<tr>
<td>WMF</td>
<td>T  F  X  X  X</td>
<td>E  T  T  X  X  X</td>
</tr>
<tr>
<td></td>
<td>T  T  F  X  X</td>
<td>B_2  T  T  T  T  T</td>
</tr>
<tr>
<td>WMF++*</td>
<td>T  T  F  X  X</td>
<td>B_2  T  T  T  T  T</td>
</tr>
<tr>
<td>ASRPC prune*</td>
<td>T  F  X  X  X</td>
<td>N  T  T  X  X  X</td>
</tr>
<tr>
<td></td>
<td>T  T  F  X  X</td>
<td>N  T  T  X  X  X</td>
</tr>
<tr>
<td></td>
<td>T  T  X  T  F</td>
<td>B_1  T  T  X  T  T</td>
</tr>
<tr>
<td></td>
<td>T  T  F  T  T</td>
<td>B_1  T  T  T  T  T</td>
</tr>
<tr>
<td>WLM</td>
<td>T  F  X  X  X</td>
<td>E  T  T  T  T  T</td>
</tr>
<tr>
<td>BAN Yahalom</td>
<td>T  T  T  F  X</td>
<td>E  T  T  T  T  T</td>
</tr>
<tr>
<td>A. DH*</td>
<td>T  X  X  F  X</td>
<td>N  T  X  X  T  X</td>
</tr>
<tr>
<td></td>
<td>T  X  X  T  F</td>
<td>B_1  T  T  X  T  T</td>
</tr>
<tr>
<td>2steps SK*</td>
<td>T  X  X  F  X</td>
<td>N  T  X  X  T  X</td>
</tr>
<tr>
<td></td>
<td>T  X  X  T  F</td>
<td>B_1  T  T  X  T  T</td>
</tr>
<tr>
<td></td>
<td>T  T  F  T  T</td>
<td>B_1  T  T  T  T  T</td>
</tr>
</tbody>
</table>

Table 7.3: Experimental results
<table>
<thead>
<tr>
<th>Protocol Name</th>
<th>New Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ASRPC</strong></td>
<td>1. $A \rightarrow B : A; {N_a}<em>{K</em>{ab}}$&lt;br&gt;2. $B \rightarrow A : {f_2(N_a); N_b}<em>{K</em>{ab}}$&lt;br&gt;3. $A \rightarrow B : {f_2(N_b)}<em>{K</em>{ab}}$&lt;br&gt;4. $B \rightarrow A : {[N_a; A; B; K'<em>{ab}; N'<em>b]}</em>{K</em>{ab}}$</td>
<td>$N_a$ in step 4 does the nonce-challenge response</td>
</tr>
<tr>
<td><strong>BAN concrete ASRPC</strong></td>
<td>1. $A \rightarrow B : A; N_a$&lt;br&gt;2. $B \rightarrow A : {A; B; N_a; K'<em>{ab}}</em>{K_{ab}}$&lt;br&gt;3. $A \rightarrow B : {A; B; N_a}<em>{K</em>{ab}}$&lt;br&gt;4. $B \rightarrow A : N_b$</td>
<td>This patch is similar to that proposed by Lowe in [60].</td>
</tr>
<tr>
<td><strong>CCITT X.509 (1)</strong></td>
<td>1. $A \rightarrow B : A; {T_a; N_a; B; N_{xa}; {A; N_{ya}}<em>{K+B}}</em>{K_A}$&lt;br&gt;2. $B \rightarrow A : {A; B; N_p; A; B}<em>{K_A}$&lt;br&gt;3. $A \rightarrow B : B; A; {N_p; T_a; N_a; B; N</em>{xa}; A; N_{ya}}_{K_B}$&lt;br&gt;4. $B \rightarrow A : N_b$</td>
<td>This patch is similar to that proposed by Abadi and Needham in [1].</td>
</tr>
<tr>
<td><strong>CCITT X.509 (3)</strong></td>
<td>1. $A \rightarrow B : A; {T_a; N_a; B; N_{xa}; {A; N_{ya}}<em>{K_B}}</em>{K_A}$&lt;br&gt;2. $B \rightarrow A : B; A; {N_p; T_a; N_a; B; N_{xa}; A; N_{ya}}<em>{K_B}$&lt;br&gt;3. $A \rightarrow B : A; {B; N_b}</em>{K_A}$</td>
<td>To avoid classic-replay attack, SHRIMP establishes the nonce-challenge response with $N_p$.</td>
</tr>
<tr>
<td><strong>CCITT X.509 (3)</strong></td>
<td>1. $A \rightarrow B : A; {T_a; N_a; B; N_{xa}; {A; N_{ya}}<em>{K_B}}</em>{K_A}$&lt;br&gt;2. $B \rightarrow A : B; A; {N_p; T_a; N_a; B; N_{xa}; A; N_{ya}}<em>{K_B}$&lt;br&gt;3. $A \rightarrow B : A; {B; N_b}</em>{K_A}$</td>
<td>This patch is similar to that proposed by Burrows, Abadi and Needham in [24].</td>
</tr>
</tbody>
</table>

Table 7.4: Patched protocol descriptions, first part
<table>
<thead>
<tr>
<th>Protocol Name</th>
<th>New Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DSSK</strong></td>
<td>1. $A \rightarrow B : A; B$</td>
<td>With these changes, $B$ authenticates $A$.</td>
</tr>
<tr>
<td></td>
<td>2. $B \rightarrow S : A; B; { N_p; A; B }_{K_B}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. $S \rightarrow A : { B; K_{ab}; T_s; { N_p; B; K_{ab}; A; T_s }<em>{K_B} }</em>{K_A}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. $A \rightarrow B : { N_p; B; K_{ab}; A; T_s }_{K_B}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1. $A \rightarrow B : A; B$</td>
<td>With this change, $A$ authenticates $B$.</td>
</tr>
<tr>
<td></td>
<td>2. $B \rightarrow S : A; B; { N_p; A; B }_{K_B}$</td>
<td>This version is similar to the NSSK protocol.</td>
</tr>
<tr>
<td></td>
<td>3. $S \rightarrow A : { B; K_{ab}; T_s; { N_p; B; K_{ab}; A; T_s }<em>{K_B} }</em>{K_A}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. $A \rightarrow B : { N_p; B; K_{ab}; A; T_s }_{K_B}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5. $B \rightarrow A : { B; N_a }<em>{K</em>{ab}}$</td>
<td></td>
</tr>
<tr>
<td><strong>NSSK</strong></td>
<td>1. $A \rightarrow S : A; B; { N_p; A; B }_{K_A}$</td>
<td>These changes establish temporal secession and guarantee that $A$ knows the specific run of the protocol.</td>
</tr>
<tr>
<td></td>
<td>2. $S \rightarrow A : { N_p; B; K_{ab}; { K_{ab}; A }<em>{K_B} }</em>{K_A}$</td>
<td>To avoid a classic replay attack, SHRIMP adds $T_s$.</td>
</tr>
<tr>
<td></td>
<td>3. $A \rightarrow B : { K_{ab}; A; T_s }_{K_B}$</td>
<td>This patch is similar to that proposed by Lowe in [61].</td>
</tr>
<tr>
<td></td>
<td>4. $B \rightarrow A : { A; B; N_b }<em>{K</em>{ab}}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5. $A \rightarrow B : { A; B; f_2(N_b) }<em>{K</em>{ab}}$</td>
<td></td>
</tr>
<tr>
<td><strong>Denning-Sacco</strong></td>
<td>1. $A \rightarrow S : A; B$</td>
<td>This patch is the same to that proposed by Abadi-Needham in [1].</td>
</tr>
<tr>
<td><strong>PK</strong></td>
<td>2. $S \rightarrow A : { A; K_A^+; T_s }<em>{K_s}; { B; K_B^+; T_s }</em>{K_s}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. $A \rightarrow B : { B; K_B^+; T_s }<em>{K_s}; { { B; K</em>{ab}; T_a }<em>{K_B} }</em>{K_A}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. $B \rightarrow A : { A; B; N_b }<em>{K</em>{ab}}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5. $A \rightarrow B : { A; B; f_2(N_b) }<em>{K</em>{ab}}$</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.5: Patched protocol descriptions, second part
<table>
<thead>
<tr>
<th>Protocol Name</th>
<th>New Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kao Chow A. v1</td>
<td>$1. A \to B : { A; B; N_a }$</td>
<td>To establish the nonce-challenge response with $N_p$, SHRIMP adds a step between $A$ and $S$ among steps 1 and 2.</td>
</tr>
<tr>
<td></td>
<td>$2. B \to S : A; B; N_a; { N_p; A; B }^g_{KB}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$3. S \to B : { A; B; N_a; K_{ab} }^g_{KA}; { N_p; A; B; N_a; K_{ab} }^g_{KB}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$4. B \to A : { A; B; N_a; K_{ab} }^g_{KA}; { A; B; N_a; K_{ab} }^g_{KB}; N_b$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$5. A \to B : { N_b }^g_{Kab}$</td>
<td></td>
</tr>
<tr>
<td>KSL</td>
<td>$1. A \to B : N_a; A$</td>
<td>This patch is the same to that proposed by Lowe in [60].</td>
</tr>
<tr>
<td></td>
<td>$2. B \to S : N_a; A; N_b; B$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$3. S \to B : { A; N_b; K_{ab} }^g_{KB}; { N_a; B; K_{ab} }^g_{KA}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$4. B \to A : { N_a; B; K_{ab} }^g_{KA}; { T_b; A; K_{ab} }^g_{Kbb}; N_c; { N_a }^g_{Kab}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$5. A \to B : { N_c }^g_{Kab}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$6. A \to B : N_{ma}; { T_b; A; K_{ab} }^g_{Kbb}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$7. B \to A : N_{mb}; { N_{ma} }^g_{Kab}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$8. A \to B : { N_{mb} }^g_{Kab}$</td>
<td></td>
</tr>
<tr>
<td>NSPK</td>
<td>$1. A \to B : { N_a; A }^g_{K_B}$</td>
<td>This patch is the same to that proposed by Lowe in [58].</td>
</tr>
<tr>
<td></td>
<td>$2. B \to A : { B; N_a; N_b }^g_{KA}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$3. A \to B : { N_b }^g_{K_B}$</td>
<td></td>
</tr>
<tr>
<td>O&amp;R BAN</td>
<td>$1. A \to B : M; A; B; { N_a; M; A; B }^g_{KA}$</td>
<td>Adding agent name $A$, the protocol now provides weak authentication of $B$ with $A$.</td>
</tr>
<tr>
<td></td>
<td>$2. B \to S : M; A; B; { N_a; M; A; B }^g_{KA}; N_b; { M; A; B }^g_{KB}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$3. S \to B : M; { N_a; K_{ab} }^g_{KA}; { A; N_b; K_{ab} }^g_{KB}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$4. B \to A : M; { N_a; K_{ab} }^g_{KA}$</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.6: Patched protocol descriptions, third part
### Table 7.7: Patched protocol descriptions, fourth part

<table>
<thead>
<tr>
<th>Protocol Name</th>
<th>New Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPLICE/AS</td>
<td><strong>This new protocol version (result of SHRIMP’s patch) is precisely the same version as Hwang-Chen Splice/as protocol. Although such a version, when demanded to provide higher authentication level, resulted to be faulty, as shown below.</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>In this patch SHRIMP implements a nonce-challenge response of (N_p) to guarantee to Bs that C is alive.</strong></td>
<td></td>
</tr>
<tr>
<td>CJ Splice</td>
<td><strong>In this patch SHRIMP not only adds a nonce-challenge response, but also adds agent name C.</strong></td>
<td></td>
</tr>
<tr>
<td>Hwang-Chen</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPLICE/AS</td>
<td><strong>With this change SHRIMP breaks the similarity between messages.</strong></td>
<td></td>
</tr>
<tr>
<td>WMF</td>
<td><strong>This new version resulted to be the same version proposed by Lowe in [61].</strong></td>
<td></td>
</tr>
<tr>
<td>WMF++</td>
<td><strong>We designed a new version of WMF following ideas of [72]. Such a version is faulty and SHRIMP patches it by including a nonce-challenge response.</strong></td>
<td></td>
</tr>
<tr>
<td>Protocol Name</td>
<td>New Description</td>
<td>Comment</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------</td>
<td>---------</td>
</tr>
<tr>
<td>ASRPC Prune</td>
<td>(1. A \rightarrow B : { [A; B; N_a] }<em>K</em>{ab} ) (2. B \rightarrow A : { [A; B; K'_{ab}] }<em>K</em>{ab} )</td>
<td>With this protocol example we can see how SHRIMP repairs a faulty protocol using methods agent_naming (twice) and session_binding (twice).</td>
</tr>
<tr>
<td></td>
<td>(1. A \rightarrow B : { [A; B; N_a] }<em>K</em>{ab} ) (2. B \rightarrow A : { [N_a; A; B; K'_{ab}] }<em>K</em>{ab} )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(1. A \rightarrow B : { [N_p; N_a; A; B; K'_{ab}] }<em>K</em>{ab} ) (2. A \rightarrow B : { [A; B; N_p; A] }<em>K</em>{ab} )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(1. P \rightarrow Q : P; N_1 ) (2. Q \rightarrow P : Q; N_2 ) (3. P \rightarrow Q : { [P; Q; N_1; N_2] }_K_P ) (4. Q \rightarrow S : { [P; Q; N_1; N_2] }_K_P ) ({ [P; Q; N_1; N_2] }<em>K_Q ) (5. S \rightarrow Q : { [N_1; Q; N_2; K</em>{pq}] }<em>K_P ) ({ [P; N_1; N_2; K</em>{pq}] }<em>K_Q ) (6. Q \rightarrow P : { [N_1; Q; N_2; K</em>{pq}] }<em>K_P ) ({ [P; N_1; N_2; K</em>{pq}] }_K_Q )</td>
<td>With this change SHRIMP breaks the similarity between messages</td>
</tr>
<tr>
<td></td>
<td>(1. A \rightarrow B : A; N_a ) (2. B \rightarrow S : B; N_b; { [A; N_a] }<em>K_B ) (3. S \rightarrow A : N_b; { [K</em>{ab}; B; N_a] }_K_A ); { [A; N_a] }<em>K_B ) (4. A \rightarrow B : { [A; K</em>{ab}; N_b] }_K_B ); { [N_b] }<em>K</em>{ab} )</td>
<td>The patch is similar to that proposed by Paulson in [75].</td>
</tr>
<tr>
<td></td>
<td>(1. A \rightarrow B : N_a ) (2. B \rightarrow A : { [A; N_a] }_K_B )</td>
<td>With this protocol we illustrate the base case of applying session_binding for asymmetric cryptography.</td>
</tr>
<tr>
<td></td>
<td>(1. A \rightarrow B : N_a ) (2. B \rightarrow A : { [N_a; A; B] }_K_B )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(1. A \rightarrow B : { [N_a; A; B] }<em>K</em>{ab} ) (2. B \rightarrow A : { [N_a; A; B; N_b] }<em>K</em>{ab} ) (3. A \rightarrow B : { [A; B; N_b] }<em>K</em>{ab} )</td>
<td>With this protocol we illustrate the base case of applying session_binding for symmetric cryptography.</td>
</tr>
</tbody>
</table>

Table 7.8: Patched protocol descriptions, fifth part
out on a PC with 1.6 GHz Pentium IV processor and 512Mb RAM. The operating system, Fedora C7.

### 7.3 Conclusions

We have carried out a large number of experiments to validate SHRIMP, finding that it successfully deals with almost all the class of replay attacks, except for the subclass *type flaw*. SHRIMP is thus able to identify a flaw and a successful candidate patch in 33 faulty protocols out of 36. The protocols SHRIMP fails to fix, namely: Neumann-Stubblebine, Otway-Rees and Woo-Lam Pi, are all susceptible to the *type flaw* subclass of replay attack.

In the next chapter we explain what a *type flaw* attack consists in and how SHRIMP could deal with this class of attack. In addition, we give an analysis detailed on how SHRIMP could patch an abstraction of the faulty IKEv2-DS protocol, the Asokan and Ginzboorg protocol and thus, directing SHRIMP to deal with other class of security protocols.
<table>
<thead>
<tr>
<th>Protocol Name</th>
<th>Tool</th>
<th>TEVT (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASRPC</td>
<td>OFMC</td>
<td>1.02</td>
</tr>
<tr>
<td>BAN concrete ASRPC</td>
<td>OFMC</td>
<td>34.34</td>
</tr>
<tr>
<td>CCITT X.509 (1)</td>
<td>OFMC</td>
<td>0.10</td>
</tr>
<tr>
<td>CCITT X.509 (3)</td>
<td>OFMC</td>
<td>2.27</td>
</tr>
<tr>
<td>DSSK</td>
<td>CL-AtSe</td>
<td>0.14</td>
</tr>
<tr>
<td></td>
<td>CL-AtSe</td>
<td>1.38</td>
</tr>
<tr>
<td>NSSK</td>
<td>OFMC</td>
<td>0.81</td>
</tr>
<tr>
<td></td>
<td>OFMC</td>
<td>0.75</td>
</tr>
<tr>
<td>Denning-Sacco PK</td>
<td>OFMC</td>
<td>0.33</td>
</tr>
<tr>
<td></td>
<td>CL-AtSe</td>
<td>334.30</td>
</tr>
<tr>
<td>Kao Chow A. v1</td>
<td>OFMC</td>
<td>2.59</td>
</tr>
<tr>
<td>KSL</td>
<td>OFMC</td>
<td>17.26</td>
</tr>
<tr>
<td>NSPK</td>
<td>CL-AtSe</td>
<td>28.8</td>
</tr>
<tr>
<td>O&amp;R BAN version</td>
<td>OFMC</td>
<td>4.15</td>
</tr>
<tr>
<td>SPLICE/AS</td>
<td>CL-AtSe</td>
<td>8.43</td>
</tr>
<tr>
<td></td>
<td>OFMC</td>
<td>0.50</td>
</tr>
<tr>
<td>CJ Splice</td>
<td>OFMC</td>
<td>0.51</td>
</tr>
<tr>
<td>Hwang-Chen SPLICE/AS</td>
<td>CL-AtSe</td>
<td>35.15</td>
</tr>
<tr>
<td>WMF</td>
<td>CL-AtSe</td>
<td>0.70</td>
</tr>
<tr>
<td></td>
<td>CL-AtSe</td>
<td>1.82</td>
</tr>
<tr>
<td>WMF revisited</td>
<td>OFMC</td>
<td>9.15</td>
</tr>
<tr>
<td>ASRPC prune</td>
<td>OFMC</td>
<td>0.85</td>
</tr>
<tr>
<td></td>
<td>OFMC</td>
<td>0.52</td>
</tr>
<tr>
<td></td>
<td>OFMC</td>
<td>0.33</td>
</tr>
<tr>
<td></td>
<td>OFMC</td>
<td>1.02</td>
</tr>
<tr>
<td></td>
<td>OFMC</td>
<td>0.35</td>
</tr>
<tr>
<td>Woo and Lam Mutual</td>
<td>OFMC</td>
<td>0.74</td>
</tr>
<tr>
<td>BAN modified Yahalom</td>
<td>OFMC</td>
<td>0.92</td>
</tr>
<tr>
<td></td>
<td>OFMC</td>
<td>2.09</td>
</tr>
<tr>
<td>A.DH</td>
<td>OFMC</td>
<td>0.78</td>
</tr>
<tr>
<td></td>
<td>OFMC</td>
<td>0.36</td>
</tr>
<tr>
<td>2Steps SK</td>
<td>OFMC</td>
<td>1.12</td>
</tr>
</tbody>
</table>

Table 7.9: The total elapsed verification time
8. Comparison and Directions for Future Work

The purpose of this chapter is twofold. First, it aims to compare our work against rival methods, and, second, it aims to give indications for further research work. To achieve the second aim we first introduce three protocol examples which allow us to explain future work at a more elaborate level of detail.

8.1 Comparison to Related Work

8.1.1 On how to deal with Replay Attacks

For years the main approach to security protocols has been their formal verification. Many researchers have taken a new look at the support of security protocols development from the perspective of proposing design principles. It is commonly suggested in the literature that successful attacks against security protocols are a result of not adopting good design principles, and that good principles can lead to protocols to be significantly more robust.

Our work is supported by using design principles for security protocols. However, when considering an automatic protocol repair mechanism like ours, one wonders whether there is an upper bound as to the information that every message should include to avoid a replay. If there is one, we could simply ensure that every message conforms it previous to any verification attempt.
Carlsen [25] has looked into this upper bound. He suggested that in order to avoid replays every message should include five pieces of information: protocol-id, session-id, step-id, message subcomponent-id and primitive type of data items. Protocol designers, however, find including all these elements cumbersome.

In a similar vein, Aura [10], considering performance issues, proposed some strategies against replay attacks like to use several crypt-algorithms for each cipher-text in a protocol; to attach hash functions (including full information similar to Carlsen) in authenticator messages; to produce unique session keys using hash functions and without assuming trust assumptions between the principals. In effect, using hash function does not affect the performance in a protocol, but replay protection attack is not avoided.

Malladi et al., [64], in order to avoid replay attacks, proposed to identify every protocol run using session-ids generated by all participants in the protocol. They showed that if all cipher-texts in the protocol include session-ids along with another data items, then the protocol is not susceptible to replays. The problem now is how the session-id between the participants should be agreed upon, especially when one of the main difficult problems in key-agreement protocols is to establish a secret between the participants.

For a protocol designer it is very difficult to have in mind all of these principles, because of that, designers still make mistakes. Unlike all works presented, SHRIMP has been automated. SHRIMP not only identifies when selected pieces of information are necessary in the protocol, but also when it is necessary to add steps to the protocol to fulfill some security property.

### 8.1.2 On the Automated Design in Security Protocols

R. Choo [29] has also looked at the problem of automated protocol repair. His development framework applies a model checker to perform state-space analysis on an (indistinguishable-based) model of a protocol, encoded using asynchronous product automated. If the protocol is faulty, Choo’s approach automatically repairs it. A fair comparison between Choo’s approach and ours cannot be conducted mainly because so far there is not an archival publication of [29].

Complementary to ours is the work of Gong and Syverson [43], who have developed a methodology to facilitate the design and analysis of security protocols. They proposed to restrict protocol designs to well-defined practices such as to develop firstly a fail-stop protocol and then verifying it using a verification tool such as BAN logic. A protocol is fail-stop if: i) the content of each message contains full information like the one prescribed by Carlsen; ii) each message contains the sender and receiver identities iii) each message is encrypted under the key shared between its sender and intended recipient; iv) a honest agent follows the rules of the protocol; and v) an agent halts any protocol run in which an expected message does not arrive within a specified timeout
period. Gong and Syverson suggest types of protocols that are fail-stop and whether the protocol is not they make changes to turn it into one that is. They mention that whenever a protocol is fail-stop, then the protocol satisfies the secrecy assumption. So, having done that, they use BAN logic in order to demonstrate formally whether the protocol satisfies such a security property or more. Using BAN logic has a lot of limitations, as explained in Chapter 2. Therefore, this methodology suffers from the same limitations of BAN logic. For example, Gong and Syverson overlooked the WMF protocol flaw and so, they could not fix such a protocol either.

A similar work is that of Perrig and Song’s [77], who have developed a system, called APG, for the synthesis of security protocols. The synthesis process, though automated, is generate and test: APG generates (extends) a protocol step by step, taking into account the security properties that are introduced as part of the system requirements. APG analyzes the resulting protocol using Athena and then it discards those protocols that do not satisfy the requirements. In case neither of the protocols satisfies such requirements, APG adds a new protocol step (introducing suitable messages in such a step) and the resulting protocols are analyzed again. The cycle is repeated as many times as necessary until the requirements are satisfied. APG is limited to generate only 3-party protocols (two principals and one server). The main problem to this tool is the combinatorial explosion (the search space is of the order $10^{12}$ according to the authors). By comparison, SHRIMP only inserts selected pieces of information considering the attack at hand but may add steps to the protocol if necessary to fulfill a stronger security property. SHRIMP does not consider the security property that the protocol violates, since our experiments show that it is not necessary to explicitly consider the property the protocol fails to satisfy; this might be attributable to that such a property is already implicit in the attack. In addition, SHRIMP has not problems with respect to the combinatorial explosion problem, because SHRIMP only analyzes the attack and the protocol description and this analysis consumes very little computational resources.

We can say that before SHRIMP the support for security protocols development had exclusively been concentrated to formal verification. SHRIMP introduces a new paradigm in the formal development of security protocols. Faulty protocols can now be diagnosed by SHRIMP in order to propose a patch.

Being SHRIMP a new method, it might be improved in several ways. So, the following section describes how SHRIMP could be extended in order to deal with other classes of faulty protocols.

### 8.2 Three Protocol Examples to illustrate directions for Future Work

This section gives indications to further works using three protocols as motivating examples. Roughly, we use the first two example protocols to motivate the need for strengthening shrimp’s
language and we use the last example protocol to indicate what would take for SHRIMP to deal with type flaw attacks.

8.2.1 The IKEv2-DS protocol

We have recently made SHRIMP try to patch the IKEv2-DS protocol, which is part of AVISPA’s library and an abstraction of IKEv2. We found that if we abstract out the equational issues inherent to the AVISPA attack, SHRIMP successfully identifies a violation to a good practice for protocol design: the omission of principal names.

We have reviewed IKEv2 protocol version (reviewed by Sebastian Mödersheim and Paul Hankes on December of 2003 using the AVISPA tool and which can be consulted in http://www.avispa-project.org/library/IKEv2-DS.html).1 In what follows, we report the analysis on this version of the protocol.

8.2.1.1 The IKEv2 Protocol Specification

This protocol proceeds in two phases. In the first phase, the users exchange nonces and perform a Diffie-Hellman exchange, establishing an initial security association called the IKE_SA. In the second phase, the users authenticate the previous messages, they exchange their identities and establish the first so-called child security association or CHILD_SA which will be used to secure the subsequent IPsec tunnel.

First phase

1 \[ A \rightarrow B : N_{s1}, \alpha^{r_a}, N_a \]

2 \[ B \rightarrow A : N_{s1}, \alpha^{r_b}, N_b \]

Second phase

3 \[ A \rightarrow B : \left\{ A, \left\{ N_{s1}, \alpha^{r_b}, N_a, N_b \right\}_{K_A}, N_{s2} \right\}_{Hash(N_a, N_b, N_{s1}, \alpha^{r_b} r_a)} \]

4 \[ B \rightarrow A : \left\{ B, \left\{ N_{s1}, \alpha^{r_b}, N_a, N_b \right\}_{K_B}, N_{s2} \right\}_{Hash(N_a, N_b, N_{s1}, \alpha^{r_b} r_a)} \]

In this protocol, \( \alpha^r \) denotes the Diffie-Hellman operation, where \( \alpha^{r_1} r_2 = \alpha^{r_2} r_1 \) holds. The Diffie-Hellman operation relies on the decision problem of discrete logarithm. Roughly speaking, it states that given \( \alpha^r \) and \( \alpha^s \), it is hard to distinguish \( r \) from a random number. This protocol version

---

1This version of the protocols is based on IKE protocol of [67]
In the SHRIMP implementation we have a type of message which models a message that is unknown, so we have used such a type in order to model the Diffie-Hellman operation. So, \( M_{ra} \) is an unknown message and it is an abstraction that we did to model the Diffie-Hellman operation. Another abstraction was the short-term key used in the IKEv2 protocol. In the IKEv2 the short-term key is derived by means of hash function. To do this, we implemented a function which translates a message to a short-term key, \( \text{msg2Key}(m) \).

Sebastian et al. mentioned that this attack is of questionable validity because the intruder never learns the key that \( B \) believes to have established with \( A \). Thus, the intruder cannot exploit the authentication flaw to further purposes. They suggested that the attack could be precluded adding a key confirmation, as demonstrated by a newer version (IKEv2-DSx).

**8.2.1.2 Patching IKEv2 with SHRIMP**

The following is an abstraction of the IKEv2 protocol suitable to SHRIMP:

1. \( A \rightarrow B : N_{sa1}; M_{ra}; N_a \)
2. \( B \rightarrow A : N_{sa1}; M_{rb}; N_b \)
3. \( A \rightarrow B : \{ A; \{ N_{sa1}; M_{ra}; N_a; N_b \} K_A^{-1}; N_{sa2} \} \text{msg2Key}(\text{hash}(X)) \)
4. \( B \rightarrow A : \{ B; \{ N_{sa1}; M_{rb}; N_a; N_b \} K_B^{-1}; N_{sa2} \} \text{msg2Key}(\text{hash}(X)) \)

In the SHRIMP implementation we have a type of message which models a message that is unknown, so we have used such a type in order to model the Diffie-Hellman operation. So, \( M_{ra} \) is an unknown message and it is an abstraction that we did to model the Diffie-Hellman operation. Another abstraction was the short-term key used in the IKEv2 protocol. In the IKEv2 the short-term key is derived by means of hash function. To do this, we implemented a function which translates a message to a short-term key, \( \text{msg2Key}(m) \).

**First experiment** After doing the experiments SHRIMP detects, using the `agent.naming` method, that an agent name is being omitted in message of step 3. The patch suggested by SHRIMP...
is the following:

\[ \begin{array}{l}
1 \ A \rightarrow B : N_{sa1}; M_{ra}; N_a \\
2 \ B \rightarrow A : N_{sa1}; M_{rb}; N_b \\
3 \ A \rightarrow B : \left\{ A; \left\{ N_{sa1}; M_{ra}; N_a; N_b \right\}_K^A; N_{sa2}\right\}_m g 2 K e y(h a s h(X)) \\
4 \ B \rightarrow A : \left\{ B; \left\{ N_{sa1}; M_{rb}; N_a; N_b \right\}_K^B; N_{sa2}\right\}_m g 2 K e y(h a s h(X))
\end{array} \]

<table>
<thead>
<tr>
<th>Protocol</th>
<th>s</th>
<th>wa_i</th>
<th>sa_i</th>
<th>wa_r</th>
<th>sa_r</th>
<th>M</th>
<th>s</th>
<th>wa_i</th>
<th>sa_i</th>
<th>wa_r</th>
<th>sa_r</th>
</tr>
</thead>
<tbody>
<tr>
<td>IKEv2</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>N</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.1: Security properties of the IKEv2 before and after the patch

With this patch, the protocol satisfies the same security properties as IKEv2-DSx (proposed in the AVISPA library). IKEv2-DSx however is different, it is as IKEv2 except that adds a key confirmation, which adds two steps to the protocol. Although the patch proposed by SHRIMP involves less overhead than IKEv2-DSx, we wondered why in the AVISPA library was not proposed to include such an agent name in step 3. The answer is because the omission of the identities in various parts of the protocol were no accident: the IKE protocol was deliberately designed so that the user has the option of not giving out identification information until it was absolutely necessary. However, this feature is somewhat contrary to the requirements of authentication [67].

**Second experiment**  Considering the argument stated before, we proceeded to do another experiment, which consisted in eliminating (by hand) the *agent naming* method (because this method precisely adds agent names) in order to see if SHRIMP could apply the session binding method and, to check if the proposed patch was similar to IKEv2-DSx.

Doing that, SHRIMP effectively patches the protocol with *session binding* method (*handshake*), the resulting protocol is as follows:

\[ \begin{array}{l}
1 \ A \rightarrow B : N_{sa1}; M_{ra}; N_a \\
2 \ B \rightarrow A : N_{sa1}; M_{rb}; N_b \\
3 \ A \rightarrow B : \left\{ A; \left\{ N_{sa1}; M_{ra}; N_a; N_b \right\}_K^A; N_{sa2}\right\}_m g 2 K e y(h a s h(X)) \\
4 \ B \rightarrow A : \left\{ B; \left\{ N_{sa1}; M_{rb}; N_a; N_b \right\}_K^B; N_{sa2}\right\}_m g 2 K e y(h a s h(X)) \\
3 \ A \rightarrow B : \left\{ A; B; N_p\right\}_K^x \\
4 \ B \rightarrow A : \left\{ B; A; f 2(N_p)\right\}_K^x
\end{array} \]
This version of the protocol is similar to IKEv2-DSx, which also is part of AVISPA’s library and it is attack-free. As shown in table 8.2. Notice that due to the abstractions done in the protocol, SHRIMP added session key $K_x$.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>$s$</th>
<th>$wa_i$</th>
<th>$sa_i$</th>
<th>$wa_r$</th>
<th>$sa_r$</th>
<th>$M$</th>
<th>$s$</th>
<th>$wa_i$</th>
<th>$sa_i$</th>
<th>$wa_r$</th>
<th>$sa_r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>IKEv2</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>N</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Table 8.2: Security properties of the IKEv2 before and after applying handshake.

As seen in this section, SHRIMP currently does not consider cryptographic primitives like xor, Diffie-Hellman operator, commutativity, etc. These cryptographic primitives are very important because newer protocols such as key agreement protocols, group protocols, etc., are designed using these kinds of cryptosystems.

In addition, sometimes a protocol is deliberately designed according to the conditions of the network, as the IKE protocol, which was deliberately designed so that the user has the option of not giving out identification information until it is absolutely necessary. This condition in SHRIMP is still not implemented, so we manipulated (by hand) the patching methods so that SHRIMP could apply a specific patching method. This requirement evidently requires to implement other conditions. These conditions could be parametrized in the patching methods, in such a way that SHRIMP could evaluate the preconditions of each patching method taking into account the condition being parametrized. These conditions could be very important for external considerations, as those described in section 3.2.1.

8.2.2 The Asokan and Ginzboorg Protocol

Asokan and Ginzboorg proposed a protocol to use with Bluetooth devices [9]. They described two protocols for distributing a session key. Our case study corresponds to the first of these.

8.2.2.1 Specification of the Asokan and Ginzboorg Protocol

The scenario under consideration is the following: a group of people are in a meeting room and want to set up a secure session amongst their Bluetooth-enabled laptops. They know and trust each other, but their computers have no shared prior knowledge and there is not trusted third party or public key infrastructure available. The protocol proceeds by assuming a short group password, $P$, which is chosen and displayed, e.g. on a whiteboard. The group, $M$, use the password, $P$, to establish a secure secret key. The Alice and Bob description of the Asokan-Ginzboorg protocol is as follows [9, 82]:
1. $M_n \rightarrow ALL : M_n; \{E\}_P$

2. $M_i \rightarrow M_n : M_i; \{R_i; S_i\}_E \quad i = 1, \ldots, n - 1$

3. $M_n \rightarrow M_i : \{\{S_j; j = 1, \ldots, n\}\}_R_i \quad i = 1, \ldots, n - 1$

4. $M_i \rightarrow M_n : M_i; \{S_i, h(S_1, \ldots, S_n)\}_K \quad$ some $i$

In words:

1. $M_n$ broadcasts a message containing a fresh public key, $E$, encrypted under $P$.

2. Every other participant $M_i$, for $i = 1, \ldots, n - 1$, sends $M_n$ a contribution to the final key, $S_i$ and a fresh symmetric key, $R_i$, encrypted under $E$.

3. Once $M_n$ has a response from everyone in the room, she collects together the $S_i$ in a package along with a contribution of her own ($S_n$) and sends out one message to each participant, containing this package $S_1, \ldots, S_n$ encrypted under the respective symmetric key $R_i$.

4. Some participant $M_i$ responds to $M_n$ with the package he just received passed through a one way hash function $h()$ and encrypted under the new group key $K = f(S_1, \ldots, S_n)$, with $f$ a commonly known function (maybe a hash function).

This protocol is faulty and Graham Steel, using CORAL [82], found the following attack for a group of size 2:

\[s1 : 1 \quad \begin{array}{c}
M_2 \quad \rightarrow \quad ALL \quad : \quad M_2; \{E\}_p \\
\end{array}\]

\[s2 : 1 \quad \begin{array}{c}
\text{Spy}(M_1) \quad \rightarrow \quad ALL \quad : \quad M_1; \{E\}_p \\
\end{array}\]

\[s2 : 2 \quad \begin{array}{c}
M_2 \quad \rightarrow \quad \text{Spy}(M_1) \quad : \quad M_2; \{\{R_2; S_2\}\}_E \uparrow \\
\end{array}\]

\[s1 : 2 \quad \begin{array}{c}
\text{Spy}(M_1) \quad \rightarrow \quad M_2 \quad : \quad M_1; \{\{R_2; S_2\}\}_E \downarrow \\
\end{array}\]

\[s1 : 3 \quad \begin{array}{c}
M_2 \quad \rightarrow \quad \text{Spy}(M_1) \quad : \quad \{\{S'_2; S_2\}\}_{R_2} \\
\end{array}\]

\[s2 : 3 \quad \begin{array}{c}
\text{Spy}(M_1) \quad \rightarrow \quad M_2 \quad : \quad \{\{S'_2; S_2\}\}_{R_2} \\
\end{array}\]

\[s2 : 4 \quad \begin{array}{c}
M_2 \quad \rightarrow \quad M_1 \quad : \quad M_2; \{S_2; h(S'_2; S_2)\}_{f(S'_2; S_2)} \\
\end{array}\]

At the end of the run, $M_2$ accepts the key $f(S'_2, S_2)$ as a valid group key, but it contains numbers known only by $M_2$ and not by $M_1$.

### 8.2.2.2 Patching the Asokan and Ginzboorg Protocol with SHRIMP

To patch the Asokan and Ginzboorg Protocol with SHRIMP we need to do some changes to the original specification of the protocol. We need to fix the number of participants in the protocol description. In this case a group of size 2. So, after we input this new description to SHRIMP together with its counterexample, SHRIMP suggested to modify the protocol as follows:
1. $M_2 \rightarrow M_1 : M_2; \left\{ \left[ M_1; M_2 \right]; E \right\}_P$

2. $M_1 \rightarrow M_2 : M_1; \left\{ \left[ M_1 \right]; R_1; S_1 \right\}_E$

3. $M_2 \rightarrow M_1 : \left\{ \left[ M_1; M_2 \right]; \left\{ S_1, S_2 \right\} \right\}_{R_1}$

4. $M_1 \rightarrow M_2 : M_1; \left\{ S_1, h(S_1, S_1) \right\}_K$

If we instantiate this protocol of size 2 to the Asokan and Ginzboorg Protocol of size n, the protocol would be as follows:

1. $M_n \rightarrow ALL : M_n; \left\{ \left[ M_n \right]; E \right\}_P$

2. $M_i \rightarrow M_n : M_i; \left\{ \left[ M_n/M_i \right]; R_i; S_i \right\}_E \quad i = 1, \ldots, n - 1$

3. $M_n \rightarrow M_i : \left\{ \left[ M_n \right]; \left\{ S_j, j = 1, \ldots, n \right\} \right\}_{R_i} \quad i = 1, \ldots, n - 1$

4. $M_i \rightarrow M_n : M_i; \left\{ S_i, h(S_1, \ldots, S_n) \right\}_K \text{ some } i$

This protocol corresponds to that one proposed by Graham in [82], for which CORAL has not found an attack. As we can see, SHRIMP patches this protocol using agent-naming method, however, we had to fix the participants of the protocol. Clearly, this is an area of opportunity where SHRIMP could be extended. However, we have some limitations with respect to the verification tools, which SHRIMP depends on, and the logic used to model SHRIMP. Currently, automated tools to security protocol verification are focused only for fixed participants (two or three). With respect to the logic, SHRIMP uses strand-spaces, this logic was not originally designed for group protocols. More details will be explained in section 8.3.3.

### 8.2.3 Otway and Rees Protocol

This protocol already presented in Chapter 3 aims to distribute a session key through a trusted server and to authenticate two friendly agents ($A$ and $B$). Here we present it again:

1. $A \rightarrow B : N'_a; A; B; \left\{ N_a; N'_a A, B \right\}_{K_A}$

2. $B \rightarrow S : N'_a; A; B; \left\{ N_a; N'_a A, B \right\}_{K_A}; \left\{ N_b; N'_a A, B \right\}_{K_B}$

3. $S \rightarrow B : N'_a; \left\{ N_a; K_{ab} \right\}_{K_A}; \left\{ N_b; K_{ab} \right\}_{K_B}$

4. $B \rightarrow A : N'_a; \left\{ N_a; K_{ab} \right\}_{K_A}$

This protocol, as seen throughout the thesis, suffers from a number of attacks. Now, we explain a new one so-called *type flaw* attack. A *type flaw attack* is an attack where a principal accepts
a message component of one type as a message of another. It relies on a principal’s inability to separate two messages of different steps [69]. The attack is as follows:

\[
\begin{align*}
    s1 : 1 & \quad A \rightarrow B : N'_a; A; B; \{\{N_a; N'_a; A; B\}\}_K_a \uparrow \\
    s1 : 2 & \quad B \rightarrow S : N'_a; A; B; \{\{N_a; N'_a; A; B\}\}_K_a; \{\{N_b; N'_a; A; B\}\}_K_b \\
    s1 : 3 & \quad S \rightarrow \text{Spy}(B) : N'_a; \{\{N_a; K_{ab}\}\}_K_a; \{\{N_b; K_{ab}\}\}_K_b \\
    s1 : 4 & \quad \text{Spy}(B) \rightarrow A : N_{ap}; \{\{N_a; N'_a; A; B\}\}_K_a \downarrow
\end{align*}
\]

Note that, \(\{\{N_a; N'_a; A; B\}\}_K_a\) is the cypher-text being replayed. \(s1 : 1\) is the session-step where \(\{\{N_a; N'_a; A; B\}\}_K_a\) was originated and \(s1 : 4\) is the session-step where such a message is being exploited. Therefore, a protocol flaw is found in step 4 because the responder accepts the message \(\{\{N_a; N'_a; A; B\}\}_K_a\) as being the session key \(K_{ab}\).

So far, SHRIMP cannot patch this class of protocols due mainly to the following reason: it uses strand-spaces model and this model considers typed messages. Put differently, the kinds of messages are disjoint. This consideration evidently overlooks type flaw attacks. In order that SHRIMP can deal with type flaw attacks, firstly, we have to either extend strand-spaces logic or change of logic so that this kind of attacks can be represented. We have explored two possibilities about how to deal with this class of attacks, which will be detailed in section 8.3.4.

### 8.3 Future Work

As demonstrated in this thesis, SHRIMP is a new method in the formal methods community and as a pioneer method it could be extended in a number of ways such as on the kind of cryptographic primitives, on the global requirements, on the number of participants, on type flaw attacks, and on the linking to a verification tool.

#### 8.3.1 On the Cryptographic Primitives

As described in section 8.2.1, SHRIMP currently does not consider cryptographic primitives like xor, Diffie-Hellman operator, commutativity, etc. It would be very important to add these cryptographic primitives in the future because if SHRIMP were equipped with these primitives, new protocols such as the complete version of IKE protocol, key agreement protocols, group protocols, etc., could be also studied. Of course, new verification tools should arise in the literature, as explained below.

Before including new patching methods to SHRIMP tackling cryptographic primitives like the ones mentioned previously, it is necessary that new design principles considering such primitives
are proposed in the literature. Since, (and following our methodology) we have used design principles in order to develop preconditions and patches in our patching methods.

However, before proposing new design principles, many researchers must solve firstly if the so-called intruder deduction problem is decidable: given a state of the protocol run, it is possible to determine whether the intruder is able to construct a message of the form that some honest agent is expecting to receive, or whether he is able to obtain a message that is intended to be secret, e.g. a key shared by two honest agents [16].

The intruder deduction problem is one of the core problems for formally analyzing security protocols because by means of this problem a lot of researchers are focused in this problem with the presence of algebraic equations (cryptographic primitives like the ones mentioned above) and others on type flaw attacks (explained below). Solutions of this context have been given by individual algebraic theories, such as the ones described below.

Common-Lundh and Shmatikov, [34], proposed to develop decision results for faulty protocols in the presence of relevant equational theories like the bitwise exclusive or.

Chevalier and Rusinowitch, [27], proposed a general procedure for deciding security of protocols in presence of exponentiation operations, group protocols, xor functions or a combination of them. However, their procedure have not been mechanized.

David Basin et. al., [16], proposed a framework that can handle algebraic properties of cryptographic operators. Their framework is based on two ideas: i) to use modular rewriting to formalize a generalized equational deduction problem for the Dolev-Yao intruder; and ii) to introduce two depth parameters that bound the depth of message terms and the operations that the intruder can use to analyze messages. They have begun integrating their framework into OFMC.

Abadi and Cortier, [3], have proved that message deducibility and indistinguishability are both decidable in polynomial time for a large class of equational theories. This class of equational theories is defined syntactically and includes, for example, theories for encryption, decryption, and digital signatures. An extension of this work is presented in [35], which extends the class of equational theories.

Lafourcade et. al., [53] have proved decidability of the intruder deduction problem in Abelian group and exclusive or operators. They have obtained a polynomial time decision procedure in a restricted case. It means that they have bounded the number of parallel protocol sessions.

### 8.3.2 On the Global Requirements

Sometimes a protocol is deliberately designed according to the conditions of the network. An example of this is the IKE protocol already analyzed, which was deliberately designed so that the user has the option of not giving out identification information until it is absolutely necessary. This
condition in SHRIMP is still not implemented, so as illustrated in section 8.2.1.1, we manipulated (by hand) the patching methods so that SHRIMP could apply a specific patching method and then to patch the protocol in a desired way.

The requirement previously explained evidently requires to implement other kind of conditions. We think that these conditions could be parametrized in the patching methods, in such a way that SHRIMP could evaluate the preconditions of each patching method taking into account the condition being parametrized, and then to skip the method not desired to be applied. We think that these conditions could be very important for external considerations, as those described in section 3.2.1. It could also be applicable to protocols that are designed to run under some network limitations.

8.3.3 On the Number of Participants

SHRIMP is a method that depends on a state-of-the-art verification tool. To this end, we have used AVISPA tool. The problem of the AVISPA tool, among most automated tools, is that it does not deal with group protocols. This is because the abstractions made to improve performance on fixed 2 or 3 party protocols either preclude the modeling of group protocols all together, or permit modeling only in a fixed scenario, which can prevent attacks from being discovered. With these restrictions, our experiment (with the Asokan and Ginzboorg Protocol) was conducted for a fixed number of participants.

A noteworthy tool to avoid doing these kinds of abstractions is Coral, a tool for finding counterexamples in both group key agreement and group key management protocols. Unfortunately, in Coral it is not possible to analyze group security protocols that include algebraic equations such as those described in [76] and the second protocol example shown in [9]. Even so, some researchers have intended proposing new approaches intending to demonstrate that it is possible to verify group security protocols (including algebraic equations) [76]. However, this is still an ongoing research. An additional problem of Coral is that it requires considerable effort in the formulation of a protocol to be verified and sometimes finding an attack is time-consuming. Yet, it would be interesting to link SHRIMP to Coral in order to extend the class of faulty protocols to be analyzed and patched.

On the other hand, the logic behind SHRIMP is strand-spaces. This logic was not originally designed for group protocols. Although, SHRIMP’s meta-logic was developed to be independent of any logic, it would be necessary either to change for a suitable logic or to do some extensions to the strand space logic to deal with group protocols.
8.3.4 On Type Flaw Attacks

In order that SHRIMP can deal with type flaw attacks, firstly, we have to either extend strand-spaces logic or change of logic so that this kind of attacks can be represented.

Heather et al. [48] have shown that tagging every protocol message, and each element thereof, with a string indicating its intended type prevents all type flaw attacks. However, as noticed by Malladi and Alves-Foss [63], message tagging makes it easier to elaborate a password guessing attack: from \{\|N\|_K\}, one needs to know \(N\) to attempt to guess \(K\); by contrast, from \{\|\text{Nonce} \; ; \; N\|_K\} one does not, since one needs only to check that any guess attempt yields a message starting with the tag \text{Nonce}. Later on, Meadows [69] argued that even when messages are tagged type flaw attacks can still arise, since there is always a chance that two messages can be confused. She developed a method that enables the intruder to identify a strategy, if any, to raise the probability of two messages are confused above a preset threshold. On the other hand, Aura [10] states that the most common integrity-protecting functions are encryption or secure hash. So, she suggested to use a unique cryptographic function for each submessage in order to tag the submessages with their static data types. In practice, however, such a suggestion is avoided for workload reasons.

Considering the arguments above, we have already explored two possibilities about how to deal with this class of attacks, but neither of them have been concreted:

1. By using hash functions and message tags. These ideas are (in part) obtained of Aura and Heather [10, 49]. Hash functions let us check the integrity of a message (type flaw attacks is precisely an integrity problem). Beside, tags let us check that the message we are receiving contains the same intention which it has been sent.

2. By extending message encoding patching method and putting enough information under the control of the receiver in such a way that the agent can associate the message to a particular step. This proposal aims to show that message tagging is not necessary: following good practices, such as the proposed by Abadi and Needham [1].

So, we think that SHRIMP could be extended using either of these considerations to deal with type flaw attacks.

8.3.5 On the linking to a Verification Tool

SHRIMP uses AVISPA tool like the state-of-the-art verification tool. This tool contains 4 different back-ends search engines: OFMC, CL-AtSe, SATMC and TA4SP. We have very familiarized with OFMC backend and we think that we could link SHRIMP directly with this backed.

In general, we think that SHRIMP could be linked to AVISPA tool because SHRIMP has some similarities in the input and output of this tool, hence we have thought in that precisely. To do this,
firstly we give a general explanation about how it should be carried out. We need to automate two parts: to link the AVISPA input with the SHRIMP output and to link the AVISPA output with the SHRIMP input.

With respect to the first link, SHRIMP has much similarities with HLPSL specification (AVISPA input). The only setback would be to automate the control predicates such as witness, request, wrequest used in HLPSL description to carry out the verification process. These predicates are not easy to configure (by hand) even for experimented users. It is because in the HLPSL specification one must specify which are the protocol submessages by means the verification will be carried out. Sometimes a nonce is used to authenticate two or more agents, sometimes a session key is used, etc. If these control predicates are not configured correctly the protocol verification could result erroneous.

With respect to the second link, it is not a problem because we have done a script with such characteristics, see Section B.1.

### 8.4 Conclusions

We have stated in this chapter that SHRIMP is a pioneer method with respect to automated correction of faulty security protocols. There exist some related works such as those of R. Choo [29], Carlsen [25], Aura [10], Malladi [64], R. Choo [29], Gong-Syverson [43] and Perrig-Song [77]. However, SHRIMP is better because our method only makes selected changes (maybe message or/and step insertion) in messages where the protocol lacks of a well design.

SHRIMP is open-ended; it will be never complete but can always be improved by adding new methods. New methods must follow the structure proposed in Chapter 4. Our model is, like all mathematical models, only ever approximations to reality. We are usually assuming that the intruder’s only source of information is through the Dolev and Yao model (monitoring traffic, among other things). In reality the intruder may have other sources such as cryptanalysis, etc. Although some attempts of bridging these two views [2] have been proposed, it seems best to deal with such vulnerabilities separately. Thus we must bear in mind that the validity of our analysis will always be relative to the threat model and to the faithfulness of our methods, so proofs of security will never be absolute. Of course, as and when new modes of attack are discovered, they can be incorporated to SHRIMP. For example, as we have already described currently the patching framework is composed by three patching methods: message_encoding, agent_naming and session_binding. With these patch methods SHRIMP can deal with the full class of replay attacks proposed by Syverson [84], within the context of typed messages. Even so, SHRIMP can be extended of a number of ways. Further work thus is concerned with extending SHRIMP to cope with type flaw attacks. Further work is also concerned with extending SHRIMP to account for more
cryptographic primitives, including the equational properties thereof.

So far, Patch methods are independent of the global security requirements required from the security protocol under analysis. Therefore, further work involves considering global rules (maybe like conditions) that may change a protocol patch depending on these global requirements.

In this thesis we have focused with fixed faulty protocols, however, new protocols are currently being designed, for example group protocols. These new protocols also demands the use of more cryptographic primitives and thus, the protocol development process requires more sophisticated tools in order to be analyzed. So, while better verification tools are developed, SHRIMP also may be extended.

So, taking into account all considerations above described, SHRIMP could deal with greater variety of protocols.
9. Conclusions

In this chapter, we discuss at more elaborate level of detail the research contributions of the thesis that were outlined in Chapter 1. Having described the background, the theory, the implementation, the experiments we have carried out and related work in more detail, we are now in a position to discuss the main contributions fully:

**SHRIMP contributes in the formal support for security protocol development bridging the gap between design and analysis by means of a diagnosis and repair task. In this aspect, SHRIMP is the first approach to fix faulty security protocols automatically.**

In Chapter 2 we demonstrated how the Automated Support for Security Protocols Development has been exclusively concentrated in demonstrating that protocols satisfy some security requirements or do not. We also presented in Chapter 3 research that aims to postulate good practices in design of security protocols. Finally in Chapter 8 we presented related work and concluded that SHRIMP is the first approach that bridges the gap between design and analysis of protocols. So, we presented our method in Chapters 4, 5, 6 and we validated it in Chapter 7.

As a result of Abadi and Needham’s principles formalization, we have identified three classes of replay attacks in terms of a mal-designed cypher-text. A cypher-text that has been used to build up a replay attack violates one or more of the following sentences: i) the originator/recipients of a cypher-text in one message of the protocol cannot be distinguished; ii) two or more different cypher-texts of the same protocol cannot be distinguished from one another: the protocol portrays two or more cypher-texts that are different one another but have similar structure; and iii) two or more
different runs of the same protocol cannot be distinguished from one another. The message cannot be associated with a particular protocol run. The protocol does not guarantee association or temporal succession.

In Chapter 4 we showed how the taxonomy of replay attacks proposed by Syverson also can be illustrated as violations to Abadi and Needham’s principles. Particularly a replay attack violates principle 3 (about agent naming), principle 10 (recognizing messages and encoding, the message cannot be bound to a particular run of the protocol), principles 6—8 (the protocol does not guarantee association or temporal succession).

SRIMP deals with the full class of replay attacks proposed by Syverson [84], within the context of typed messages.

In Chapter 7 we presented a large number of experiments to validate SRIMP, finding that it successfully deals with the class of replay attacks, except for type flaw subclass. The protocols SRIMP fails to fix are precisely protocols susceptible to type flaw attacks, because of that we emphasize that SRIMP deals with the full class of replay attacks within the context of typed messages.

SRIMP also contributes as a general knowledge for the design of sound security protocols. The general composed method, called replay, together with the three patching methods: agent_naming, message_encoding and session_binding, give a general knowledge about how to avoid violations to design security protocols.

The theory (SRIMP’s meta-logic) explained in Chapter 5, together with the patching framework described in Chapter 6, could be used by the designer to give himself a general idea about the main errors that he could make and thus try to avoid them.

9.1 Final Summary

Before SRIMP the automated support for security protocols development had exclusively concentrated in formal verification. Because of that, the formal methods community have proposed several techniques to better the design of security protocols such as verifying protocols or proposing informal guidelines. However, neither of them have addressed the problem of finding the specific flaw, let alone patching the faulty protocol. Now, faulty protocols can be diagnosed and patched with SRIMP.

With the aim that protocol designers can use SRIMP freely and not to be subject to a determined protocol verification tool, we made SRIMP independent of any logic. So, SRIMP can
be used in the development process for a security protocol like an additional tool. We construct a meta-logic that may be implemented on any logic, but we selected strand-space because this logic let us express suitably SHRIMP’s formalizations.

To construct SHRIMP we used proof planning methodology. This methodology is a reasoning technique developed especially as a search control engine to automate theorem proving. A proof plan expresses the patterns of reasoning that must be followed to search proof in theorems. We use proof planning methodology to search for suitable patches in faulty security protocols, which we call patch planning. In addition, the concepts used by this methodology (preconditions and postconditions) were very suitable with the requirements of patching protocols. So, using patch planning we explain how SHRIMP works, the kinds of attacks that it can patch and how new patching methods could be added to SHRIMP.

Currently SHRIMP is formed by one general composed method, called replay, and three patching methods: message_encoding, agent_naming and session_binding. Our general method is about replay attack because we have found in the literature that the main source of attacks in security protocols is precisely this class.

A comparison of SHRIMP with other works is difficult because SHRIMP is a new method. Currently there exists a lot of principles proposed in the literature that protocol designers must have in mind at the time they are developing a protocol. This activity is so difficult that designers still make mistakes because for a protocol designer it is very difficult to have in mind all principles proposed. SHRIMP hence can help designers as it is automated. However, by comparison with other works consisting of synthesizing protocols, SHRIMP is better because it identifies not only when selected pieces of information (that violates a design principle) are necessary in the protocol, but also when it is necessary to add steps to the protocol when it does not satisfy some security property.

The successful of SHRIMP is evidenced with the amount of faulty protocols that it can patch, 33 faulty protocols out of 36. The remainder 3 is not a SHRIMP problem, but the logic we selected (strand spaces). Even so, SHRIMP can be extended in a number of ways. In Chapter 8 we gave three protocol examples to explain in detail how SHRIMP could be extended. Here, we summarize this future work. Further research is concerned with extending SHRIMP to cope with type flaw attacks. Further research also is concerned with extending SHRIMP to account for more cryptographic primitives, including the equational properties thereof.

Patch methods are independent of the global security requirements required from the security protocol under investigation and the local change of one or two protocol steps may cause flaws in combination with other protocol steps which have been not considered at that point. Therefore, further research involves considering global rules that may change a protocol depending on these global requirements to be achieved, controlling the local patch process.
In this thesis we have focused with fixed faulty protocols, however, new protocols are currently being designed, for example *group protocols*. These new protocols also demand the use of more cryptographic primitives and thus, the protocol development process requires more sophisticated tools in order to be analyzed. So, while better verification tools are developed, SHRIMP also may be extended.
References


REFERENCES


Appendix A. AVISPA Tool

This chapter introduces AVISPA tool, the state-of-the-art verification tool, that we have used to verify security protocols. It takes as input a HLPSL specification (a high-level protocol specification language for modelling communication and security protocols) and outputs a script describing the outcome of such a verification. While we describe both the main parts of AVISPA’s input (HLPSL) and the AVISPA’s output, we also explain the main characteristics and difficulties about how to link SHRIMP with AVISPA. Finally, we give an authentication hierarchy of AVISPA compared with Lowe’s Hierarchy. Since, the former is widely known in the literature but we have used the first one.

A.1 An Overview of HLPSL Specifications

Protocol specifications in HLPSL comes in two parts: one describing the protocol and the other one describing the properties to be verified. The first part is divided into roles, some roles (so-called basic) describe the actions of the participating agents in a run of a protocol. Other roles (composed roles) instantiate these basic roles to model an entire protocol run. Once given all roles, we must then define the security goals that the protocol is aimed to satisfy.

Before to introduce basic roles, composed roles, and security goals we describe types and messages in HLPSL.
A.1.1 Types and Messages

HLPSL is a typed language: each variable and constant must have a unique type. We summarize the basic types available in HLPSL:

- **agent**: represents agent names. The spy (intruder) is always assumed to have the special identifier i.

- **text**: often used as nonces. When used for this purpose, function new() is used to indicate freshness.

- **nat**: represents the natural numbers in non-message contexts.

- **function**: represents one-way functions such as hash.

- **public_key**: represents agents’ public keys for asymmetric cryptography. For a given public (respectively private) key $k$, its inverse private (respectively public) key is obtained by $\text{inv}(k)$.

- **symmetric_key**: represents keys for symmetric encryption.

- **channel**: connects communicating agents. This type takes an attribute which specifies the spy model to be used for communication over the channel. Currently, the Dolev-Yao (dy) model is the only supported.

Concatenation operation on messages is given under “.” operator and a message $m$ encrypted under key $k$ is written as $\{m\}_{k}$. If we have an agent name $A$ of type **agent**, a nonce $Na$ of type **text** (new()), and a key $K$ of type **symmetric_key**, then the following is a valid message:

$$\{A.Na\}_{K} \quad \% \text{agent name and nonce encrypted under key } K$$

Note that comments in HLPSL begin with the % symbol and continue until the end of the line.

As you can see, the protocol specification that we use in our protocol description language is formed by almost the same types as HLPSL does, see Section 5.2. The only extension in this way would be add the type **channel**.

A.1.2 Basic Roles

Figure A.1 illustrates two basic roles (**role alice** and **role bob**). The definition of a basic role generally consists of the following elements:

1. role declaration: name, list of arguments and a player declaration. It is declared using keyword **role**;
2. declaration of local variables using keyword *local*;
3. initialisation of variables using keyword *init*;
4. declaration of accepting states;
5. knowledge declarations: usually the arguments (taken as initial knowledge) and all of the local variables acquired during the transitions; and
6. transition section, where the steps of the protocol are specified.

**Transitions:** In Figure A.1, arrow \( X = | > Y \) represents an immediate reaction transition and relates an event \( X \) and an action \( Y \). This expresses that, whenever in the execution of such a transition if in the left-hand side (LHS) event predicate \( X \) is true, then immediately the execution of action \( Y \) must hold in the right-hand side (RHS).

**Old and new values of variables:** A primed variable (e.g. \( Na' \)) denotes the new value of a variable in a transition: this new value has been either learned in the LHS of the transition (received on a channel, among other things), or assigned in the RHS of the transition. Function \( \text{new}() \) means assigning the new value of a variable with a fresh value (i.e. a nonce). A predicate like \( \text{Rcv}(Nb') \) means to assign \( Nb' \) to the value sent on the channel \( \text{Rcv} \).

**Goal specification:** Goals are specified as macros representing pre-defined safety temporal formulae built on the goal predicates *witness*, *wrequest* (for weak authentication), *request* (for strong authentication), and secret. These goal predicates are explicitly declared in the RHS of HLPSL transitions and they are used to specify secrecy and different forms of authentication.

For clarifying the concepts previously described we explain role *alice* in detail, Figure A.1. Role *alice* receives, from composed role *nspk*, the following arguments: agent names (\( A \) and \( B \)), public keys (\( Ka \) and \( Kb \)), and the channels whereby the messages will be sent (\( \text{Snd} \)) and received (\( \text{Rcv} \)). These arguments can be taken to be her initial knowledge.

We can see that the first transition begins upon reception of a start message (\( \text{Rcv}(\text{start}) \)). This message is sent by the intruder through the channel in order to signal who is the initiator. Once it has occurred variable \( \text{state}^1 \) changes to 1, nonce \( Na \) is created (note that \( Na' \) means the new value of \( Na \)) and the first message of the protocol is sent through the channel \( \text{Snd}.^2 \) Predicate \( \text{witness}(A,B,bob,alice,na,Na') \) means that agent \( A \) asserts that she wants to be the peer of agent

---

1 Variable \( \text{state} \) is used like an identifier in order to know the transition step and it must be declared in *local* section.
2 Note that \( Na \) and \( Nb \) must be declared of type *text* in *local* section.
REFERENCES

135

B, agreeing on value Na’ in an authentication effort, identified by the protocol id bob_alice_na (Note that this witness is linked with the wrequest(B,A,bob_alice_na,Na) of role bob for weak authentication). Predicate secret(Na’,na,\{A,B\}) means that value Na’ is shared between (only) the agent set A and B, identified by the protocol id na.3

In the second transition step2and3, upon reception of message Rcv(\{NaNb’\},Ka), alice responds the respective message to B through the channel, Snd(\{Nb’\},Kb). Predicate request(A,B,alice_bob_nb,Nb’) means that A accepts value Nb’ and now relies on the guarantee that agent B exists and agrees with him on this value, identified by the protocol id alice_bob_nb (Note that this request is linked with the witness(B,A,alice_bob_nb,Nb’) of role bob for strong authentication).

Each of the parts in this subsection can be automated in the SHRIMP output in order to represent the AVISPA input. However, as explained in section 8.3.5, the more difficult part would be to add the control predicates such as witness, request, wrequest used precisely in HLPSL description to carry out the verification process. These predicates are not easy to configure specially because it depend on the security properties that the designer wants to verify. For example, if the designer wants to verify strong authentication, it is necessary to include the request control, but if the designer wants to verify weak authentication, then it is necessary to include wrequest control. On the other hand, it is very important to know what is the protocol message that will be used to verify authentication or secrecy.

A.1.3 Composed Roles

The definition of a composed role is similar to a basic role. It differs in that there are not transition section. Rather, there is a section entitled composition in which other roles (both basic and composed) are instantiated. Composition can be sequential (using ; operator) or parallel (using /\ operator). HLPSL does not allow compositional instantiation outside a role definition. That is, any composition must be inside a role. For example, role nspk, Figure A.1, instantiates in parallel form basic roles alice and bob passing their initial knowledge. This knowledge is received in turn of the environment role.

The environment role is the top-level role where the sessions are described. For instance Figure A.1 describes three concurrent sessions (of nspk). The first is a typical session with the legitimate agents a and b. Note that all of the arguments are in lower-case because are constants rather than variables (declared in const section). The second and third sessions are ones in which

---

3The protocol id must be declared of type protocol_id in the cont section
the intruder is impersonating either Alice or Bob. In addition, in the environment role the intruder’s initial knowledge is declared as a set of messages.

A.1.4 Security Goals

Currently, HLPSL supports only secrecy and authentication. Goals are given in their own section, which generally comes at the end of a HLPSL specification. It begins with the keyword goal and ends with end goal. Between the two, multiple security goals may be listed.

To express the secrecy goal secrecy_of is used (linked with goal predicate secret), for weak authentication goal weak_authentication_on is used (linked with goal predicates witness and wre-quest) and for strong authentication goal authentication_on is used (linked with goal predicate witness and request).

For example, in goal section of Figure A.1, authentication_on alice_bob_nb indicates that identifier alice_bob_nb, which has been used in witness(B,A,alice_bob_nb,Nb’) (role bob) and in request(A,B,alice_bob_nb,Nb’) (role alice), must verify that a principal is right in believing that its intended peer is present in the current session, has reached a certain state, and agrees on certain values, which typically are fresh. In addition, secrecy_of na states if the intruder learns the value in the first parameter of secret(Na’,na,{A,B}) in role alice and he is not in the set given in the third position, then this represents an attack. For more details in basic roles, composed roles and security goals refer to [8, 11, 12].

This part could be mechanized from the SHRIMP output. However, this part depends on the control predicates, then in order to do a correct verification task, it is necessary to include such control predicates correctly.

A.2 Verifying a protocol with the HLPSL specification

Once HLPSL specification has been written, the script avispa can be used to evaluate it and will print the result of the analysis. The AVISPA tool is composed mainly of two modules:

1. a translator for transforming HLPSL specifications to IF specifications⁴, called hlpsl2if; and

2. four different verification tools (back-ends) that can analyze IF specifications: CL-AtSe, OFMC, SATMC and TA4SP. A detailed study of these back-ends is given in [11].

An example of running AVISPA tool on the NSPK protocol, with the back-end CL-AtSe is the following command:

⁴IF is an abbreviation of intermediate format
%% Role declaration, arguments are the initial knowledge
role alice(A,B: agent, Ka, Kb: public_key, Snd, Rcv: channel (dy)) played by A
def=  
  local State: nat, % declaration of local variables
    Na, Nb: text
const na, alice_bob_nb, bob_alice_na: protocol_id
init State := 0 % initialisation of variables
transition
% Start of the protocol, step 1. 1. A --> B : {Na, A}_Kb
step1. State = 0 \{ Rcv(start) \} =>
  State' := 1 \{ Na := new() \{ Snd({Na'.A}_Kb) \}
  \{ witness(A, B, bob_alice_na, Na') \}
  \{ secret(Na', na, {A, B}) \}
% Receiving the 2nd message, step 2. 2. B --> A : {Na, Nb}_Ka
step2and3. State = 1 \{ Rcv({Na.Nb'}_Ka) \} =>
% Sending the 3rd message, step 3. 3. A --> B : {Nb}_Kb
   State := 2 \{ Snd({Nb'}_Kb) \}
   \{ request(A, B, alice_bob_nb, Nb') \}
end role
role bob(A, B: agent, Kb, Ka: public_key, Snd, Rcv: channel (dy)) played by B
def=  
  local State: nat, Na, Nb: text
const nb, alice_bob_nb, bob_alice_na: protocol_id
init State := 0
transition
step1and2. State = 0 \{ Rcv({Na'.A}_Kb) \} =>
  % Bob discards whatever received message not holding the step's format
  State' := 1 \{ Nb := new() \{ Snd({Nb'.Nb'}_Ka) \}
  \{ witness(B, A, alice_bob_nb, Nb') \}
  \{ secret(Nb', nb, {A, B}) \}
% Last step of the protocol n. JC --> K : {Th4nks Fl4k7t4 f8r 4ll}_JK
step3. State = 1 \{ Rcv({Nb}_Kb) \} =>
  State' := 2 \{ wrequest(B, A, bob_alice_na, Na) \}
end role
% The role representing a partial session between alice and bob
role nspk(A, B: agent, Ka, Kb: public_key, SC, RC: channel (dy))
def=  
  composition
    alice(A, B, Ka, Kb, SC, RC) \bob(A, B, Kb, Ka, SC, RC)
end role
role environment() % The main role
def=  
  local Snd, Rcv: channel(dy)
const a, b, i: agent, ka, kb, ki: public_key
intruder_knowledge = {ki, inv(ki), a, b, ka, kb}
composition
  nspk(a, b, ka, kb, Snd, Rcv)
  \nspk(a, i, ka, ki, Snd, Rcv)
  \nspk(i, b, ki, kb, Snd, Rcv)
end role
goal % Properties to verify
  authentication_on alice_bob_nb
  weak_authentication_on bob_alice_na
  secrecy_of na
  secrecy_of nb
end goal
environment() % Call of the main role

Figure A.1: HLPSL of the NSPK protocol
avispa NSPK.hlpsl --cl-atse -ns -lr

Note that -ns and -lr are two options in CL-AtSe: the first one for no simplifications and the second one for getting one of the shortest attacks. The output format of this verification can be seen in Figure A.2.

A.2.1 Output format

All back-ends of the AVISPA tool have the same output format. Roughly speaking, the output format is formed by the following parts: SUMMARY, DETAILS, PROTOCOL, GOAL, BACKEND, STATISTICS and ATTACK TRACE (if any). We concentrate only in the ATTACK TRACE part.

Figure A.2 shows a secrecy attack on the NSPK protocol. Note that the numbers to the left (in ATTACK TRACE part) were annotated only as reference (in this case, it is not part of the output format). In what follows, we describe all of the lines of ATTACK TRACE part:

Line 1: spy indicates to agent $A$ (instantiated by $(a,6)$) that she is the initiator of the protocol.

For reasons related to the internal workings of the hlpsl2if translator and the AVISPA back-ends, each role instance is assigned a session number: in this case, “6”.

Lines 2, 4, 5, 8 and 9 follow the steps of the protocol. Note that terms generated freshly (nonces, session keys) are denoted with their variable names and a number which is used to identify them uniquely, in this case $Na$ is denoted by $n9(Na)$ and $Nb$ is denoted by $n5(Nb)$.

Lines 3, 6 and 7: For reasons internal to the AVISPA tool, the set of agents sharing the secret $n9(Na)$ is stored in an automatically generated variable called set_74, respectively $n5(Nb)$ to set_70. The two events Add a to set_74 and Add i to set_74 in this protocol step (respectively to set_70) pinpoint that both a and the intruder (i) are members of the set: that is, the intruder is allowed to know this “secret”. Witness, line 6, means that agent $b$ asserts he wants to be the peer of gent $a$, on value $n5(Nb)$.

Since line 7 states that nonce $n5(Nb)$, stored in set 70 (line 6), is only permitted to be known by agents $a$ and $b$, the violation in the protocol is given when the intruder knows such a nonce, line 9. Note that AVISPA tool searches attacks according to the security goals given in section goals and the first attack found is the first output.

As you can see the notation of the AVISPA output is very similar with respect to Alice and Bob notation, and we have already done a script that translates Alice and Bob notation into SHRIMP notation, see Section B.1 and Appendix B.1. So, AVISPA output it is not difficult to mechanize to become the SHRIMP input.
REFERENCES

SUMMARY

UNSAFE
DETAILS
ATTACK_FOUND
TYPED_MODEL

PROTOCOL

/gousr/local/avispa-1.0/testsuite/results/NSPK-withoutServer.if

GOAL
Secrecy attack on (n5(Nb))

BACKEND
CL-AtSe

STATISTICS

Analysed : 40 states
Reachable : 35 states
Translation: 0.01 seconds
Computation: 0.00 seconds

ATTACK TRACE

1 i -> (a,6): start
2 (a,6) -> i: {n9(Na).a}_ki
3 & Secret(n9(Na),set_74); Add a to set_74; Add i to set_74;
4 i -> (b,4): {n9(Na).a}_kb
5 (b,4) -> i: {n9(Na).n5(Nb)}_ka
6 & Secret(n5(Nb),set_70); Witness(b,a,alice_bob_nb,n5(Nb));
7 & Add a to set_70; Add b to set_70;
8 i -> (a,6): {n9(Na).n5(Nb)}_ka
9 (a,6) -> i: {n5(Nb)}_ki

Figure A.2: The AVISPA tool’s output: Attack on the NSPK protocol
A.3 Authentication: AVISPA Hierarchy vs Lowe Hierarchy

Authentication: one agent should become sure of the identity of the other agent through a message established between them. This property has various meanings, however, the four levels introduced by Lowe [62] has become the standard reference in the literature: Aliveness, Weak agreement, Non-injective agreement and Agreement.

- **Aliveness:** It is the weakest reasonable definition of authentication. A protocol guarantees to an initiator $A$ aliveness of another agent $B$ if, whenever $A$ (acting as initiator) completes a run of the protocol, apparently with responder $B$, then $B$ has previously been running the protocol.\(^5\)

- **Weak agreement:** A protocol guarantees to an initiator $A$ weak agreement with another agent $B$ if, whenever $A$ (acting as initiator) completes a run of the protocol, apparently with responder $B$, then $B$ has previously been running the protocol, apparently with $A$.\(^6\)

- **Non-injective agreement:** A protocol guarantees to an initiator $A$ Non-injective agreement with a responder $B$ on a set of data items $ds$ (where $ds$ is a set of free variables appearing in the protocol description) if, whenever $A$ (acting as initiator) completes a run of the protocol, apparently with responder $B$, then $B$ has previously been running the protocol, apparently with $A$, and $B$ was acting as responder in his run, and the two agents agreed on the data values corresponding to all the variables in $ds$.\(^7\)

- **Agreement:** A protocol guarantees to an initiator $A$ agreement with a responder $B$ on a set of data items $ds$ if, whenever $A$ (acting as initiator) completes a run of the protocol, apparently with responder $B$, then $B$ has previously been running the protocol, apparently with $A$, and $B$ was acting as responder in his run, and the two agents agreed on the data values corresponding to all the variables in $ds$, and each such run of $A$ corresponds to a unique run of $B$.\(^8\) or simply agreement means that there is a one-one relationship between the two agents’ runs.

As we can see authentication is composed by different levels. Each of them offers a different security level. For example, if a protocol is designed to offer agreement but it only provides weak-agreement, then it is a flaw of authentication. A lot of protocols are faulty by not providing the level of security which they were originated to provide.

\(^5\)Note that $B$ may not necessarily have believed that he was running the protocol with $A$. Also, $B$ may not have been running the protocol recently

\(^6\)Note that $B$ may not necessarily have been acting as responder

\(^7\)Note that this does not guarantee that there is a one-one relationship between the runs of $A$ and the runs of $B$. 

\(^8\)Injective agreement
**AVISPA Hierarchy of Authentication**  AVISPA Hierarchy of Authentication was inspired with the Lowe’s hierarchy of authentication. Thus, this property is annotated with $A_{Auth_B,x}$ and $A_{WAuth_B,x}$. $A_{Auth_B,x}$ abbreviates that $A$ authenticates strongly to $B$ on $x$ and $A_{WAuth_B,x}$ abbreviates that $A$ authenticates weakly to $B$ on $x$.

**A comparison of both Hierarchies**  $A_{Auth_B,x}$ is similar to non-injective-agreement and $A_{WAuth_B,x}$ is similar to weak-agreement according to the Lowe’s hierarchy. In AVISPA Hierarchy to obtain injective-agreement it is necessary that the protocol provides $A_{Auth_B,x_1}$ and $B_{Auth_A,x_1}$ on the same variable.

### A.4 Conclusions

In this thesis we used AVISPA tool in our security protocol verification process. AVISPA tool is a state-of-the-art verification tool for affix agents. A protocol specification in AVISPA tool is declared in a convenient, human readable, and lightly easy to use language so-called HLPSL. We think that SHRIMP could be added to AVISPA tool, which throughout this appendix we have described the main characteristics and difficulties to do this activity.

We also related AVISPA hierarchy and Lowe hierarchy of authentication because the latter is widely known in the literature and we used the former one.
Appendix B. SHRIMP Implementation

In this chapter we present how we have implemented SHRIMP, the tools that we have used and how one can execute SHRIMP implementation.

B.1 Software tools used in our meta-logic

In this section we give an overview about the software tools that were used to build SHRIMP. Figure B.1 illustrates in a dotted box where SHRIMP is located within the development process for security protocols. As you can see, SHRIMP receives like input a faulty security protocol description and a counterexample, returning a new description of the protocol when SHRIMP patch the protocol successfully.

To implement SHRIMP we have constructed two specification languages: an informal Alice and Bob language and a formal language based in part to Paulson inductive approach [74]. The Alice and Bob language must provide a convenient, human readable, an easy to use. Yet, it must be enough to support a regular run of a protocol and a counterexample as specified and output by AVISPA respectively.

Although Alice and Bob language is convenient and human readable, it is insufficient to reason with more complicated symbols introduced in Chapter 5. To this end, we have developed a script abn2fn.py written in Python, which translates a script containing the specification of a protocol and one of its counterexamples (in Alice and Bob language) into the formal language that SHRIMP’s meta-logic implementation understands.
Sml was used in the construction of SHRIMP’s meta-logic. This implementation consists of a list of sml files where each of them represents a list of symbols (the meta-logic) in order to represent roles, protocols, counterexamples and symbols that allows to represent the preconditions and the suitable patches to the faulty protocols.

We focus the following two subsections to describe how to use both script `abn2fn.py` and SHRIMP meta-logic implementation.

**B.1.1 How to use SHRIMP-Implementation Input**

Script `abn2fn.py` translates a security protocol description and a counterexample, which are written in Alice and Bob notation, to SHRIM notation. This script receives like input a file with `abn` extension, e.g. `protocol.abn` and the traduction is placed in other file denoted with `jka` extension, which is a file in ML notation.

A file `abn` is composed by three parts and clearly splitted by the following reserved words: `initState`, `protocol`, and `counterExample`. Figure B.2 illustrates the syntax to write files `< filename.abn >`.  

---

**Figure B.1: SHRIMP in the Development Cycle for Security Protocols**

---
initState

\[ agent = \text{msg} \mid (\text{msg}, \text{msg})^* \]

protocol

\[ \text{nstep agent} \rightarrow \text{agent} : \text{msg} \]

counterExample

\[ s \text{nstep} : \text{nstep sender} \rightarrow \text{receiver} : \text{msg} \]

---

Figure B.2: Syntax to write abn files

where:

\[
\begin{align*}
\text{agent} & \equiv [A - E][id]^* \mid S \\
\text{nstep} & \equiv [0 - 9]^+ \\
\text{sender} & \equiv \text{agent} \mid \text{Spy} \\
\text{receiver} & \equiv \text{agent} \mid \text{Spy} (\text{agent}) \\
\text{msg} & \equiv \text{msgsimplify} \mid \text{msgcompose} \\
\text{msgsimplify} & \equiv \text{agent} \mid T[id] \mid N[id] \mid M[id] \mid \text{key} \\
\text{msgcompose} & \equiv F1(\text{msg}) \mid F2(\text{msg}) \mid \text{msg} \_ \text{key} \mid \text{msg, msg} \\
\text{id} & \equiv [a - z, 0 - 9]^+ \\
\text{key} & \equiv K + [id] \mid K - [id] \mid K[\text{agent}] \mid K[id]
\end{align*}
\]

To execute script abn2fn.py in the shell, just do the following:

\[
\$\text{abn2fn.py} \ < \ \text{protocol.abn}
\]

If the protocol description is printed in the shell means that the script has successfully translated the protocol, otherwise a line error will be displayed explaining the cause of the problem.

**B.1.2 How to use SHRIMP-Implementation**

SHRIMP meta-logic has been developed on the Standard ML ’97 (SML). SHRIMP makes use of the Paulson’s logic of agent population and message structure. In what follows, we explain how to both install and run SHRIMP implementation.
RUNNING

1. To install SML/NJ 110.

2. To run SML. For example:

   $sml
   Standard ML of New Jersey, Version 110.0.7, September 28, 2000 [CM; autoload enabled]

3. To load shrimp.sml library as follows:

   - use "shrimp.sml";

4. The system lists the examples that come pre-loaded with SHRIMP.

LIBRARY EXPLANATION

shrimp.sml is the main file which calls all libraries pre-loaded. The libraries are composed by 3 parts: basics, meta-language and input. Basics are libraries that are not precisely part of the meta-logic but that are used by it, these libraries include strings.sml and sets.sml. Meta-language are libraries that are central to construct the meta-logic. In these libraries are defined the types of messages, roles of agent, protocol definition, counterexample definition and the patching methods; these libraries include definition.sml, agentRoles.sml, messages.sml, print.sml, replay.sml and latexPrint.sml. Input are libraries containing the relation of faulty protocols that will be input to SHRIMP to be patched, in this case, prot_similarity.sml, prot_naming.sml and prot_handshake.sml have been pre-loaded.

strings.sml contains all comments output by SHRIMP.

sets.sml contains functions of the set theory which we have developed to be used by SHRIMP.

definition.sml contains declarations and definitions of messages, communication events, protocol and counterexample.

agentRoles.sml contains the definition of types of agents and operators to find the initiator, the responder(s) and the server in a protocol.
messages.sml contains functions that can operate on those messages that are sent or received in
the traffic. This file is similar to the Paulson’s message theory.

print.sml contains the most important functions that are used to print on the shell of SML.

replay.sml contains the main method, so-called replay, and the others submethods: message en-
coding, agent naming and session binding.

latexPrint.sml translates a protocol and a counterexample in latex form.

prot_similarity.sml, prot_naming.sml, and prot_handshake.sml contain the pre-loaded faulty pro-
tocols that violate message encoding, agent naming and session binding methods respec-
tively.

B.2 On Technical Aspects of SHRIMP-Implementation

SHRIMP meta-logic makes use of the Paulson’s logic like agent population, message structure and
(partially) communication events. SHRIMP adds some functions and some definitions to complete
the reasoning in the analysis of the description of a security protocol and one of its counterexam-
ples. We have used this logic because it is enoughly known by researchers focused in the formal
verification of security protocols, besides this logic holds the most of the cryptographic primitives
used in the design of this protocols.

B.2.1 Representation of a protocol

Formalizing a protocol involves defining the initial knowledge of each agent and each step of the
protocol, communication events, agents and messages. We specify a security protocol \( P(\text{IK}, \text{steps}) \)
like a tuple consisting of the initial knowledge and a list of steps:

\[
P \overset{\text{def}}{=} (\text{initKnow}, \text{step} :: \text{steps})
\]

Recall that \( l :: tl \) is the list consisting of \( l \) prefixed to the list \( tl \).

The initial knowledge is a list of messages that an agent knows before of the run of a protocol.
It is defined as follows:

\[
datatype \text{initKnow} \overset{\text{def}}{=} \text{Know of agent} * \text{msg list};
\]

A step of a protocol is specified as an integer and a tuple of communication events. The integer
denotes the step identifier of a protocol and the tuple denotes two communicating events: the
sending and the reception of the associated message:

\[ \text{step} \overset{\text{def}}{=} \text{int } (ev * ev) \]

### B.2.2 Communication events

We model two communication events: *Says* and *Gets*. *Says A B M*, denotes agent A sending message M to agent B through the traffic. *Gets B M* denotes agent B receiving message M from the traffic. The representation is as follows:

\[
\text{datatype event } \overset{\text{def}}{=} \text{Says of agent } * \text{agent } * \text{msg} \\
| \text{Gets of agent } * \text{msg}
\]

The data-type event could, of course, be extended in many ways. For example, we could include other kinds of communication events in order to model security protocol groups (e.g. *Saysbroadcast A M*, where an agent A could send a broadcast through the network, etc.). This kind of extensions are future work.

### B.2.3 Agents and Messages

There are three kinds of agents: the server S, the friendly agents and a Dolev-Yao spy, Spy [38]. The server is considered absolutely a trusted agent and friendly agents have the form *Friend nat*. The following declaration specifies type *agent*:

\[
\text{datatype agent } \overset{\text{def}}{=} \text{Server } | \text{Friend of nat } | \text{Spy}
\]

As in [74] is mentioned, a data-type declaration creates a union type, with injections whose ranges are disjoint. In SML it follows that the various kinds of agent are distinct, with Server \(\neq\) Friend \(i\), Server \(\neq\) Spy, Spy \(\neq\) Friend \(i\), and moreover Friend \(i =\) Friend \(j\) if \(i = j\).

A message also is a data-type declaration. A message comprises agent names, time-stamps (guessable), nonces (non-guessable), keys (public, shared and session keys), one-way (Hash), two-
way (Succ), xor functions (considered for the future), ciphered messages and compound messages:

\[
\text{datatype } \text{msg} \overset{\text{def}}{=} \text{Agent of agent} \\
| \text{Number of nat (guessable)} \\
| \text{Nonce of nat (non \textendash guessable)} \\
| \text{Key of key} \\
| \text{OWFun of msg} \\
| \text{TWFun of msg} \\
| \text{Crypt of key \boldsymbol{\ast} msg} \\
| \text{MPair of msg \boldsymbol{\ast} msg}
\]

Again, the kinds of message are distinct, with \text{Agent} A \neq \text{Nonce} N and so forth.

The data-type key has four different types: the first two for asymmetric cryptography: public key and private key; the last two for symmetric cryptography: long-term key (usually keys shared with \text{S}) and short-term key (session keys). The representation is as follows:

\[
\text{datatype } \text{key} \overset{\text{def}}{=} \text{Pub of Friend of nat} \\
| \text{Priv of Friend of nat} \\
| \text{Ltk of Friend of nat} \\
| \text{Stk of nat}
\]

Note that the short-term key, \text{Stk}, does not have constructor \text{Friend} because a session key does not link with an agent in particular.

Given the basic ingredients we can represent the NSPK protocol formally as follows:
// Initial Knowledge
nspk  =  ( [Know(Friend(A), msg2Set(MPair(Key(Pub(Friend(A))), MPair(Key(Priv(Friend(A))), MPair(Key(Pub(Friend(B))), Key(Pub(Spy))))))),
 Know(Friend(B), msg2Set(MPair(Key(Pub(Friend(B))), MPair(Key(Priv(Friend(B))), MPair(Key(Pub(Friend(A))), Key(Pub(Spy))))))),
 Know(Spy, msg2Set(MPair(Key(Pub(Spy))), MPair(Key(Priv(Spy))), MPair(Key(Pub(Friend(A))), Key(Pub(Friend(B))))))])

// Protocol Steps
[[((1, (Says(Friend(A), Friend(B), Crypt(Pub(Friend(B))), MPair(Nonce(Na), Agent(Friend(A))))), Gets(Friend(B), Crypt(Pub(Friend(B))), MPair(Nonce(Na), Agent(Friend(A)))))),
 (2, (Says(Friend(B), Friend(A), Crypt(Pub(Friend(A))), MPair(Nonce(Na), Nonce(Nb)))),
 Gets(Friend(A), Crypt(Pub(Friend(A))), MPair(Nonce(Na), Nonce(Nb))))),
 (3, (Says(Friend(A), Friend(B), Crypt(Pub(Friend(B))), Nonce(Nb)),
 Gets(Friend(B), Crypt(Pub(Friend(B)), Nonce(Nb))))))]

B.2.4 Representation of a counterexample

The definition of a counterexample is similar to the protocol specification. It differs from a protocol in the session identifier solely. Thereby, we represent a counterexample similarly to the specification of the protocol as follows:

\[ C \overset{\text{def}}{=} \text{sstep} :: \text{tl}C \]

A step of a counterexample will be called session-step, \text{sstep}, and be defined as a tuple: an integer (to represent the session) and a step of a protocol.

\[ \text{sstep} \overset{\text{def}}{=} (\text{int} \ast \text{step}) \]

An example of a counterexample is illustrated below.

B.3 Illustrating the Full Verification Cycle for Security Protocols

In this section we illustrate the development cycle for security protocols using SHRIMP, which we have called Full Verification Cycle. To do this task, we show an example with the NSPK protocol.
B.3.1 First Phase

Following Figure B.1 we must translate the protocol given in Alice and Bob notation into the formal representation of the verification tool we are using. As we have used AVISPA tool we must translate it in HLPSL specification. So, the NSPK protocol in HLPSL is as follows:

```plaintext
%% NSPK protocol
role alice(A,B:agent,
    Ka,Ks: public_key,
    Snd, Rcv: channel (dy))
played_by A
def=
    local State: nat,
    Na, Nb: text,
    Kb: public_key
    const na, alice_bob_nb, bob_alice_na: protocol_id
init State := 0
transition
%% Start of the protocol
step1. State = 0 /
    Rcv(start) =>
    State' := 2 /
    Na' := new() /
    Snd({Na'.A}_Kb')
    \ witness(A,B,bob_alice_na,Na')
    \ secret(Na',na,{A,B})

%% Receiving the second message of the protocol and sending the third
step3. State = 2 /
    Rcv({Na.Nb'}_Ka) =>
    State' := 3 /
    Snd({Nb'}_Kb)
    \ request(A,B,alice_bob_nb,Nb')
end role

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
role bob(A,B:agent,
    Kb,Ks: public_key,
    Snd, Rcv: channel (dy))
played_by B
def=
    local State: nat,
    Na, Nb: text,
    Ka: public_key
    const nb, alice_bob_nb, bob_alice_na: protocol_id
init State := 0
transition
%% Start of the protocol by part of the responder
step1a. State = 0 /
    Rcv({Na'.A}_Kb) =>
    State' := 2 /
    Nb' := new() /
    Snd({Na'.Nb'}_Kb')
```

```plaintext
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
role bob(A,B:agent,
    Kb,Ks: public_key,
    Snd, Rcv: channel (dy))
played_by B
def=
    local State: nat,
    Na, Nb: text,
    Ka: public_key
    const nb, alice_bob_nb, bob_alice_na: protocol_id
init State := 0
transition
%% Start of the protocol by part of the responder
step1a. State = 0 /
    Rcv({Na'.A}_Kb) =>
    State' := 2 /
    Nb' := new() /
    Snd({Na'.Nb'}_Kb')
```
\[ \text{witness}(B,A,alice\_bob\_nb,Nb') \]
\[ \text{secret}(Nb',nb,{A,B}) \]

%% Last step of the protocol
step3. State = 2 \text{Rcv}(\{Nb\}_Kb) =|>
State' := 3 \text{wrequest}(B,A,bob\_alice\_na,Na)
end role

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% The role representing a partial session between alice and bob
role nspk(SC, RC: channel (dy),
    Ks: public_key,)
def=
    local A, B :agent,
    Ka, Kb: public_key
    composition
        alice(A,B,Ka,Ks,SC,RC)
    end role

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% The main role
role environment()
def=
    local Snd, Rcv: channel(dy)
    const a, b, s, i : agent,
    ka, kb, ki, ks : public_key
    intruder\_knowledge = \{ki, inv(ki), a, b, ks, ka, kb\}
    composition
        nspk(Snd,Rcv,ks)
end role

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Properties to verify
goal
    authentication\_on alice\_bob\_nb
    secrecy\_of na, nb
end goal

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Call of the main role
environment()
Once we have specified the protocol in HLPSL language we must verify the protocol using AVISPA, as illustrated in appendix A.2.

B.3.2 Second Phase

Once verified the protocol we obtain a counterexample. Below you can see the specification of the protocol together with its counterexample in *abn* notation:

**initState**

B: K+B, K-B, K+A, K+Spy
Spy: K+Spy, K-Spy, K+A, K+B

**protocol**

1 A --> B : \{Na, A\}_|K+B|
2 B --> A : \{Na, Nb\}_|K+A|
3 A --> B : \{Nb\}_|K+B|

**counterExample**

s1:1 A --> Spy : \{Na, A\}_|K+Spy|

s2:1 Spy --> B : \{Na, A\}_|K+B|

s2:2 B --> Spy(A) : \{Na, Nb\}_|K+A|

s1:2 Spy --> A : \{Na, Nb\}_|K+A|

s1:3 A --> Spy : \{Nb\}_|K+Spy|

s2:3 Spy --> B : \{Nb\}_|K+B|

Using script *abn2fn.py* we translate this protocol in *jka* notation as follows:

```plaintext
val name_nspk = "nspk\n";
val is_nspk = [Know(Friend(Ins"A"), msg2Set(MPair(Key(Pub(Friend(Ins"A")))))
               MPair(Key(Pub(Friend(Ins"A"))))),MPair(Key(Pub(Friend(Ins"B")))))
               Key(Pub(Spy))))),Know(Friend(Ins"B"), msg2Set(MPair(Key
               (Pub(Friend(Ins"B")))))MPair(Key(Pub(Friend(Ins"B")))))
               Key(Pub(Spy))))),Know(Spy,
               msg2Set(MPair(Key(Pub(Spy)))MPair(Key(Pub(Spy)))
               MPair(Key(Pub(Friend(Ins"A")))))
               Key(Pub(Friend(Ins"B"))))));
val nspk = [(1, [Says(Friend(Var"A"), Friend(Var"B"),
               Crypt(Pub(Friend(Var"B")))MPair(Nonce(Var"Na"),Agent(Friend(Var"A")))))
               Gets(Friend(Var"B"), Crypt(Pub(Friend(Var"B")))MPair(Nonce(Var"Na"),
               Agent(Friend(Var"A"))))))),(2, [Says(Friend(Var"B"), Friend(Var"A"),
               Crypt(Pub(Friend(Var"A")))MPair(Nonce(Var"Na"),Nonce(Var"Nb")))
               Gets(Friend(Var"A"),Crypt(Pub(Friend(Var"A")))MPair(Nonce(Var"Na"),
               Nonce(Var"Nb")))))),(3, [Says(Friend(Var"A"), Friend(Var"B"),
```
This protocol representation is input to SHRIMP together with all its libraries. To patch this protocol we introduce the following line in the shell of SML:

```sml
val (rNSPK, nspk, newNSPK) = replayAttack(is_nspk, nspk, CE_nspk);
```

To show the results we put the following line:

```sml
string2Screen(showResultMeth(rNSPK, nspk, CE_nspk, newNSPK, name_nspk));
```

**SHRIMP displays the following:**

*************************************************************
Method applied in
PROTOCOL Name : nspk
the result is:
This is an attack violating Principle 3
Precondition 2 of naming method (P3 ) holds

Faulty protocol:
1 A --> B : {Na, A}K+B
2 B --> A : {Na, Nb}K+A
3 A --> B : {Nb}K+B

CounterExample:
s1:1 A --&gt; Spy : \{Na, A\}K+Spy
s2:1 Spy --&gt; B : \{Na, A\}K+B
s2:2 B --&gt; Spy(A) : \{Na, Nb\}K+A
s1:2 Spy --&gt; A : \{Na, Nb\}K+A
s1:3 A --&gt; Spy : \{Nb\}K+Spy
s2:3 Spy --&gt; B : \{Nb\}K+B

Fixed protocol:
1 A --&gt; B : \{Na, A\}K+B
2 B --&gt; A : \{B, Na, Nb\}K+A
3 A --&gt; B : \{Nb\}K+B

**************************************************************

B.3.3 Third Phase

As you can see the output of SHRIMP proposes a new description of the protocol. This new
description must be verified again using the verification tool used in the first phase. In case the
protocol have not been patched we must repeat the process as many time as necessary.