Game Semantics for the Specification and Analysis of Security Protocols

Mohamed Saleh

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy at

Concordia University

Montréal, Québec, Canada

November 2008

Your file Votre référence
ISBN: 978-0-494-45677-4
Our file Notre référence
ISBN: 978-0-494-45677-4

ABSTRACT

Game Semantics for the Specification and Analysis of Security Protocols

Mohamed Saleh, Ph. D.
Concordia University, 2008

Security protocols are communication protocols that are used when agents communicate sensitive information in hostile environments. They are meant to achieve security goals such as the secrecy of a piece of communicated information or the authenticity of an agent's identity. Their two main characteristics are the use of cryptographic operations such as encryption or digital signatures and the assumption that communication takes place in the presence of a malicious intruder. It is therefore necessary to make sure that the protocol design is correct and will thus achieve its security goals even when under attack by the intruder. Design verification for security protocols is no easy task; a successful attack on the Needham-Shroeder authentication protocol was discovered 17 years after the protocol had been published.

We present a framework for the specification and analysis of security protocols. The specification language is close to the standard "arrow" notation used by protocol designers and practitioners, however, we add some constructs to declare persistent and fresh knowledge for agents. The analysis that we conduct consists of two stages: Modeling and verification. The model we use for protocols is based on game-semantics, in which the emphasis is put on interaction. The protocol is modeled as a game between the intruder and agents. Verification amounts to finding successful strategies for either the agent or the intruder. For instance, if the protocol goal is to achieve fairness in exchanges between possibly cheating agents, then the verification algorithm searches the game tree to insure that each non-cheating agent is not put at a disadvantage with respect to other agents. In order to be able to specify a wide range of security properties of strategies, we propose a logic having modal, temporal and linear characteristics. The logic is also equipped with a tableau-based proof system that serves as a basis for a model checking algorithm. To validate our approach, we designed and implemented a software environment that verifies protocol specifications against required properties. We use this environment to conduct case studies.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my supervisor, Dr. Mourad Debbabi, for introducing me to several research topics and guiding me throughout the preparation of this thesis. I would also like to thank Dr. Rachida Dssouli for accepting me to the Ph.D. program and helping me pass its initial steps. The experience and coding skills of Dr. Rachid Hadjidj were invaluable for the completion of the software environment that we jointly implemented. Also, the discussions I had with Dr. Chamseddine Talhi and his valuable suggestions greatly improved the style and presentation of the thesis. I deeply appreciate the effort and time all the examination committee members have sacrificed to review this thesis. The internal committee members are Dr. Otmane Ait Mohamed, Dr. Walaa Hamouda, and Dr. Juergen Rilling. I am thankful to Dr. Jean Goubault-Larrecq, the external committee member, for his effort and comments. My father, mother, and brother are, as they have always been, my solid base of support. Finally, to all my teachers, I thank you for your dedication.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

ATL	Alternating-time Temporal Logic
ATS	Alternating Transition System
AVISPA	Automated Validation of Internet Security Protocols and Applications
CAPSL	Common Authentication Protocol Specification Language
CCC	Cartesian Closed Category
CSP	Communicating Sequential Processes
CTL	Computational Tree Logic
FDR	Failure Divergence Refinement
FSM	Finite State Machine
HLPSL	High Level Protocol Specification Language
ISL	Interface Specification Language
LEM	Law of Excluded Middle
PCF	Programmable Computable Functions
PPT	Probabilistic Polynomial Time
TTP	Trusted Third Party

Chapter 1

Introduction

A communication protocol specifies a set of rules that have to be followed by principals (agents) wishing to communicate on a certain communication channel or network. More specifically, a protocol is a set of predefined communication steps between the agents, aiming to ensure a common language between them. A protocol can also specify a set of internal steps that each agent has to execute in order to update its internal state. Security protocols, on the other hand, are a subclass of communication protocols. They are used when agents want to share "secret" information while communicating over an insecure network. They serve a variety of security objectives, such as authenticating agents to each other, exchanging cryptographic keys, or ensuring that the data sent is not changed or replicated. Security objectives are numerous and so are the applications of cryptographic protocols such as e-commerce, online banking, electronic voting, etc. The following section will give an overview of security protocols and their properties.

1.1 Security Protocols

In security protocol terminology, *principals* or *agents* are the entities wishing to communicate. in this thesis these two terms are used interchangeably. We also treat them as hardware/software entities and hence use neuter pronouns when referring to them. Agents use security protocols in order to achieve certain security objectives when communicating over an insecure channel. These security objectives are regarded as properties that the security protocol must satisfy in order to successfully serve its purpose. The security properties of protocols include [2]:

- Authentication: Simply put, authentication means making sure that each agent is the one it claims to be, i.e. no agent should be able to impersonate another agent.
- Confidentiality: The protection of transmitted data from malicious attacks, which means that the message content should only be disclosed to the intended receiver of the message.
- Integrity: This means that messages should be received as sent without modification, duplication, reordering or replay.
- Non-repudiation: This is to ensure that neither the sender (nonrepudiation of origin) nor the receiver (nonrepudiation of receipt) should be able to deny sending (or receiving) a message.

Security protocols are distinguished by two factors: The use of cryptographic functions, and the assumption of the presence of a malicious intruder that tries to break the protocol security. Cryptographic functions such as encryption, decryption, or hashing are necessary for the protocol to achieve its security objectives. They are the basic "tools" that security protocols use. However, even if we assume that the encryption is unbreakable (the perfect cryptography assumption), there remains a number of very important issues such as what messages to encrypt and with which keys, how do we ensure a safe exchange of keys, how to prevent replay attacks, etc. All of these are issues can compromise security *even* under the assumption of perfect cryptography [19, 20]. They are dealt with in the framework of security protocols, and it is therefore of utmost importance to make sure that they are dealt with *correctly*. This turns out to be a difficult task, even for very simple protocols. Needham and Shroeder published their cryptographic protocol in 1978 [87]. It was discovered to be flawed after 17 years of service, the attack was discovered by Gavin Lowe first in 1995, and was explained in several papers [70].

As for the malicious intruder, its presence is always assumed and its purpose is to make the security protocol fail in its intended goals. In this regard, security protocols can be classified according to which goals they aim to achieve. For instance, one of the first and most widely used applications of security protocols, as mentioned above, is authentication. Authentication protocols aim to authenticate users to each other before beginning to exchange secret data. An example is the Needham-Shroeder protocol [87] discussed earlier. Another type is the key exchange protocols whose purpose is to securely and correctly handle the distribution of cryptographic keys between different agents. An example of these is the Neuman-Stubblebine protocol [88]. With the rapid growth of electronic commerce, several protocols were designed to deal with the issues involved. For instance, the ASW protocol [21], developed by Asokan, Shoup, and Waidner, is designed to ensure the fair exchange of information between two parties. It is used in contract signing, where each party sends to the other a non repudiable commitment to a certain contract. Here, fairness requires that if the protocol was executed successfully, each party will end up with a copy of the contract including the commitment of the other party. If, on the other hand, the protocol was ended prematurely, no party should have advantage over the other given the information already exchanged thus far. Moreover, the protocol supports non-repudiation of receipt and of origin, while guaranteeing the termination of the information exchange in a finite time. A trusted third party is used during the protocol *only* in the case of exceptions.

Making *sure* that a security protocol actually can resist intruder attacks and achieve its goals is not an easy task since it involves the analysis of distributed systems with issues of concurrency in addition to security issues. Several methods have been developed to specify and analyze security protocols. This is the subject of the next two sections.

1.2 Specification and Analysis

There are various methods [1], that differ in scope and purpose, for specifying and analyzing security protocols. For example, some specifications are informal narrations that mix some kind of defined syntax with natural language descriptions. In fact, many specifications are written in what we call in this thesis "standard notation" or "arrow-notation" of security protocols. An example is given below:

$$\begin{array}{ll} \text{Step 1.} & A \rightarrow B: \ \{N_A\}_{K_{AB}} \\ \text{Step 2.} & B \rightarrow A: \ \{N_A, N_B\}_{K_{AB}} \end{array}$$

The notation above means that, in step 1, agent A sends to agent B a *nonce* encrypted by the private key between A and B (K_{AB}). A nonce is short for "number used once", which is used to guarantee freshness and hence prevent replay attacks. In step 2, agent B replies by concatenating its own nonce to the one first received from A, encrypting the whole with the private key between A and B and sending it to A. Of course, the notation above is accompanied by a natural language description that helps to clarify several issues. For instance, it should be made clear what information is known to each agent prior to communication —in the case above it is the encryption key—, what values are freshly generated, and what are the checks the agents are supposed to do over the messages they receive. These checks are very important to mention correctly as they may be the sources of confusion, which may lead to malicious attacks.

On the other hand, there exist other specifications for security protocols that are based on formal specification languages [77], logics [29] and process calculi [5].

They enable more precise and verifiable descriptions of the protocols. Several of these methods are surveyed in Chapter 3.

The analysis of security protocols aims to provide some level of assurance that the protocol actually does what it is supposed to do. Methods used for the analysis generally can be categorized as: Methods used in cryptanalysis [53], the use of software/hardware testing tools [103], logic-based methods [29], theorem proving techniques [92] and model checkers [24, 75]. Other methods also do exist, such the ones based on process calculi [70] and type theory [41]. The approach used in these methods together with some examples are presented in Chapter 3.

1.3 Security Protocols and Formal Methods

In order to obtain a credible assessment of a security protocol, a rigorous verification method has to be applied to it. The aim of the verification is to provide assurance that the protocol actually achieves its intended goals. In this regard, there are two broad approaches: The computational approach and the formal approach. Computational approaches [53] are developed by cryptanalysts in the cryptographic community. They deal with messages as bit strings and with cryptographic operations as functions over these strings. They use probabilistic concepts and deal with numerical data using ideas from cryptanalysis in order to discover attacks and evaluate their likelihood. Usually, these ideas involve number theory and known practices developed over the years. Formal approaches [78], on the other hand, are developed in the programming languages and formal verification communities. They apply experience from these fields to formulate security properties and verify them on models of security protocols. These models, however, treat cryptographic operations symbolically; operations are considered black boxes with ideal properties that we discuss below. Some research work [6] tries to bridge the gap between the two approaches by introducing more practical models of security protocols that can express some algebraic properties of cryptographic operations. In this thesis, we follow this third approach when introducing our model.

When analyzing security protocols in a formal framework [78] several assumptions are being made [107]:

- Perfect cryptography: This means that all cryptographic functions are unbreakable. Usually this assumption is made to concentrate the effort on the "logic" of the protocol and not the "cryptographic tools" it uses, which can be studied separately.
- The intruder: As we mentioned earlier, security protocols are characterized by the presence of a malicious intruder trying to break protocol security. To express the intruder's capabilities, the Dolev-Yao model [44] is adopted. It can be summarized as follows:
  - The intruder is a legitimate user of the network, which means it can send messages through the network.
  - The intruder can obtain any message sent through the network (eavesdropping).
  - The intruder can be a receiver to any agent in the network.
  - The intruder can prevent any message in the network from reaching its intended receiver.

The underlying assumption is that the intruder has perfect control over the network and will carefully design attacks to compromise communication security. The attacks may include impersonating legitimate users of the networks, replaying messages, etc.

Formal methods have been successfully used in the specification and verification of hardware systems and communication protocols. This constitutes an incentive to their use for security protocols. To be precise, by "formal methods" we mean that the specification and verification processes are based on a mathematical or a logical model of the system. This model provides the tools necessary to describe the system and analyze it in order to prove that it satisfies its requirements. One of the first examples of the use of formal methods with security protocols was the work done by Dolev and Yao [44] to analyze public key encryption protocols. Ever since, new techniques and models have emerged [1, 78], and a survey will be presented in Chapter 3 of this thesis.

1.4 Security Protocols and Game Semantics

Several formal methods have been proposed for the specification and verification of security protocols. Most of these methods do not capture some features of security protocols in order to simplify the analysis. Game semantics [8, 9, 10, 11, 59, 60] is an approach to the semantics of programming languages that makes explicit the interaction of the system with the environment in each computation step. This interaction is modeled by a game in which the players are the environment (the opponent) and the system (the proponent). Any computational step done by the system can be modeled by a possible sequence of moves in the game that describes the system. In this view, we make the distinction between the program type (e.g., function signature) and the algorithm (e.g., function definition) that this program implements. A program of a certain type is modeled by a certain game, whereas the algorithm specifies the rules according to which the game should be played. So, any particular run of the program (execution of the algorithm) represents a certain sequence of moves over the game that describes the program. In this case, the game-semantics specification of the program is the set of all such sequences, i.e., the set of all possible runs of the program.

The use game semantics as a model for security protocols is motivated by the following reasons:

- Game semantics can model the interaction between agents and the intruder in a natural way. The intruder is the opponent in the protocol game, while honest agents are the proponents. Each interaction in the protocol can be modeled as a sequence of moves in the game between the intruder and an honest agent.

- Having the concept of agents in the protocol model, enables us to express security properties that involve exchanging items between many parties such as in contract signing and fair exchange protocols. These properties cannot be expressed as trace properties [74], i.e., properties of a single trace in the execution tree of the protocol.

- Game strategies can be used to express protocol execution among agents, and verification amounts to finding winning strategies. This approach to verification covers a wide range of both trace-based and non-trace-based properties. Moreover, quantification over traces can then be done existentially, universally and by agent.

- Rules that govern the play of the game can be easily set to express concepts of security protocols such as freshness, the growth of knowledge of each agent with each communication step, etc.

- Games give a dynamic view of how the protocol communication would proceed under various manipulations by the intruder. They can also model the computational steps that are done by agents to generate messages, this is achieved by choosing appropriate definitions for agents strategies.

Similar to the discussion above about game semantics and programs, in order to build the game that represents the protocol, we have to assign a certain type to the protocol. This type will determine the game to be played. The specific syntax of the protocol (protocol steps specifying messages), on the other hand, will determine how this game should be played. This means that two protocols having the same type will be represented by the same game, however the game is played differently in case the two specifications differ.

1.5 Problem Statement and Objectives

The research problem we are concerned with is the development of a dedicated theoretical framework, backed with a software tool for the development of correct and secure cryptographic protocols. More specifically, we consider:

- **Protocol Specification**: We aim to devise a language for the description of security protocols that is close to the standard notation already used.
- **Protocol Analysis**: The developed language should have formal syntax and semantics. The semantic model, can then be used to verify security properties which should be stated in a suitable logic.

The two general objectives stated above can be further broken down to the more detailed tasks stated below:

- Define a syntax for security protocols that is close to the standard notation and yet is detailed enough to avoid ambiguities. It therefore must have explicit constructs to declare, for each agent, persistent and fresh knowledge. The former does not change across sessions and the latter is different for each session.
- Develop a semantic model for the defined language that should be able to cover features of security protocols, namely cryptographic operations and an intruder model.
- Extend the semantic model with some concepts from the computational approach to the analysis of security protocols. As an example, it should be able to handle equational theories that express algebraic properties of messages.

- Define a logic to express a wide range of security properties which are both trace-based and non-trace-based. It should also be able to express message patterns since we deal with messages as symbolic algebraic terms.

- Provide a prototype software environment that validates the theoretical ideas presented in the thesis and enables us to conduct case studies on different protocols. This software environment should be preferably written in a popular programming language so that the source code can be easily understood by a large number of programmers who can then contribute to the tool.

In the following section, we present the work done in this thesis in pursuit of our goals. We point out our contributions to the state-of-the-art.

1.6 Contribution

In order to reach the goals listed in the previous section, we developed a game semantics model for security protocols. We also provided a logic to express security properties over the model. Moreover, in order to be able to consider case studies, we designed and implemented a prototype software environment that we used to specify security protocols and verify their properties. In more detail, we can list our contributions in the following:

- A model for security protocols, based on game semantics, with a syntax close to the standard notation to facilitate its use by protocol designers and practitioners. The model can handle algebraic properties of messages in the form of equational theories.

- A logic that expresses security properties of game strategies. The logic enables the formulation of linear-time and branching-time properties and is equipped with an agent restriction operator that enables quantification by agent identities. The logic can also express patterns of messages since messages exchanged in the games have an algebraic structure.

- A software environment written in Java programming language that implements a model-checking algorithm for the logic and that is used to conduct case studies on several protocols.

1.7 Structure of Dissertation

This thesis is organized in eight chapters starting by the introduction. In Chapter 2, we introduce some concepts of cryptography that we are going to use in the following chapters. In Chapter 3, we present the state-of-the-art in the specification and verification of security protocols. We list major methodologies using a common notation to make the comparison easier, and we provide examples of the use of each methodology. Chapter 4 presents our treatment of protocol messages as symbolic algebraic structures. We also show how to design abstract computation procedures that operate on symbolic messages to produce new ones. In Chapter 5, we begin by defining the games that we use to represent interactions in a protocol. Games are then used to give semantics to security protocols in two cases: Functional semantics and security semantics. In functional semantics, we consider all communicating parties to be honest and thus describe how the protocol would *normally* proceed. In security semantics, we take into account the possible manoeuvres done by the intruder in order to attack protocol security. We also show how to extract abstract computation procedures from protocol specifications using the idea of frames. In Chapter 6, we define a logic for expressing security properties of the model. This logic is equipped with a tableau-based proof system from which we design a model-checking algorithm. In Chapter 7, we present our software environment and use it to conduct case studies. Finally, Chapter 8 concludes the thesis.

Chapter 2

Security, Cryptography and Game Semantics

In this chapter, we present some background material on topics related to the thesis subject. We start by providing a general overview of information security and the role that cryptography plays in it. Then, we present general concepts of game semantics and its use in the theory of programming languages. Concepts and terminology introduced in this chapter will be used throughout this dissertation.

2.1 Security and Cryptography

The secrecy of sensitive information has long been a goal of individuals, governments and organizations throughout history. Secret information that is communicated or stored needs to be concealed from any malicious intruders. This concealment is mainly carried out in three different ways:

- Information hiding [93]: The sensitive information is hidden in other information that is easily accessible, for instance, the use of secret ink to conceal messages in a letter or the use of digital watermarking techniques [49] to hide data in a digital photograph.

- Signal conversion [52]: Special signal processing techniques are applied to analog signals to make them unrecognizable such as the scrambling of video or audio signals.

- Cryptography [104]: Discrete messages, called the plain text, are converted to a text having some security features. For instance, in encryption, plain text is transformed into a cipher text or cryptogram. The algorithm that carries out this conversion uses a key in order to produce the cipher text which should not reveal any information about the plain text. Knowing the algorithm and the key, the cipher text can be converted back to the corresponding plain text.

We are concerned with cryptography as in the last item above. We distinguish here between cryptographic *operations*, cryptographic *systems*, cryptographic or security *protocols* and finally cryptographic *mechanisms*. Cryptographic operations are the individual algorithms available to cryptographers in order to operate on plain text. Often, one or more of these operations are combined to produce a cryptographic system whose purpose is to achieve a certain security objective. A cryptographic system is thus an entity that produces information, e.g., cipher text, that has cryptographic properties. In a networked environment, this information will often be communicated to other systems or entities. Security protocols are the communication protocols used whenever two or more systems need to securely exchange data. The protocols define computation steps, during which cryptographic operations are often used, and communication steps that involve sending messages between systems. A cryptographic mechanism has a larger scope and it involves all the necessary operations and protocols needed to achieve a certain security objective. In the next four sections, we will briefly discuss security objectives, cryptographic operations, security protocols and analysis of cryptographic mechanisms.

2.1.1 Security Objectives

Security objectives represent the ultimate goal for which a cryptographic mechanism is developed. In other words, they are propeties that the mechanism should satisfy. The mechanism may involve more than one security protocol and within each protocol, cryptographic operations are used to produce messages. Some security objectives are listed below along with their descriptions [99, 104]:

- Secrecy: To keep the content of a message secret except for those who are authorized to know it.
- Data integrity: To ensure that a certain message was not tampered with during its transmission.
- Authentication: To ensure that no agent is lying about its identity.
- Non-repudiation: To ensure that an agent cannot deny sending or receiving a message.
- Fairness: To ensure that no agent is put in a disadvantage with respect to other agents when exchanging one item for another.
- Anonymity: To hide the identity of some agents involved in a transaction in a certain system.

Usually a security mechanism is designed to achieve one or more of the objectives listed above. Within the mechanism, cryptographic operations and protocols are the tools used to reach these security goals. Various analysis techniques have been developed to make sure that security objectives are reached taking into account the presence of a malicious intruder or attacker trying the break the security of the mechanism. We discuss these techniques after introducing cryptographic operations and protocols.

2.1.2 Cryptographic operations

Cryptographic operations, which are also called cryptographic tools or primitives, are the basic building blocks of a cryptographic system. The purpose of such a system is to produce data or messages that have certain security features. We will briefly define some operations, a more comprehensive presentation can be found in almost all standard texts on cryptography such as [79] and [99].

One-way functions: A function $f : \mathbb{X} \rightarrow \mathbb{Y}$ is called one way if it is much "easier" to compute $f(x)$ for an element $x \in \mathbb{X}$ than it is to compute $f^{-1}(y)$ for most elements $y \in \mathbb{Y}$.

Trapdoor one-way functions: It is a one-way function such that with the knowledge of a piece of information (the trapdoor), it becomes "easy" to compute $f^{-1}(y)$ for any $y \in \mathbb{Y}$.

Cryptographic hash functions: A hash function $f : \mathbb{X} \rightarrow \mathbb{Y}$ is one that produces a fixed length output $y \in \mathbb{Y}$ for any input $x \in \mathbb{X}$. A cryptographic hash function, informally called one-way hash function, is also one-way and collision-free, i.e., it is "unlikely" to find two values x and x' such that $f(x) = f(x')$.

Random number generation: A number is generated randomly if it is generated (in binary form) by flipping a fair coin successively and considering the head landing up as 0 and tails as 1. Any other process similar to flipping a coin can also be used. Other processes not satisfying the criteria of a fair coin flip generate a pseudorandom number. Details and statistical characteristics of random numbers and sequences can be found in [23].

Symmetric cryptographic system: Let $\mathbb{K}$, $\mathbb{P}$ and $\mathbb{C}$ be the sets of keys, plain text and cipher text respectively, a symmetric encryption system defines two functions: $e : \mathbb{K} \times \mathbb{P} \rightarrow \mathbb{C}$ and $d : \mathbb{K} \times \mathbb{C} \rightarrow \mathbb{P}$. The functions e and d are called encryption and decryption, respectively. For a certain key K, the functions $e(K, .)$ and $d(K, .)$ are both bijective. Given a key $K \in \mathbb{K}$ and a plain text $P \in \mathbb{P}$ we have $e(K, P) = C$, where C is a cipher text in $\mathbb{C}$. The decryption function should have the property that if it is applied to C using the same key K, we get $d(K, C) = P$, i.e., knowing a key and a plain text we can get a cipher text and if we decrypt this cipher text using the same key, we obtain the plain text again. A cryptographic system is considered secure and hence successful at its goal if it is "difficult" to obtain information about the plain text using only the cipher text.

Asymmetric cryptographic system: In an asymmetric cryptographic system keys come in pairs, an encryption or public key $K \in \mathbb{K}$ and a decryption or private key K^{-1}. To encrypt the plain text we use the encryption function $e(K, P) = C$ and to obtain the plain text P from C, we have to use K^{-1} where $d(K^{-1}, C) = P$. We can therefore regard e as a one way trapdoor function where the trapdoor information is the decryption key. The public key is called as such since it can be made public for anyone to use and produce cipher text, however only the person who has the private key will be able to get the plain text by applying the decryption function.

Signatures: Signatures are used to associate a certain message or piece of information with a certain agent or entity. A digital signature system comprises a method for signing and another one for the verification of signatures. More formally, let $\mathbb{A}$, $\mathbb{M}$ and $\mathbb{S}$ be the sets of agents, messages and signatures respectively. A digital signature system assigns to each agent $A \in \mathbb{A}$ a signing function $g_A : \mathbb{M} \to \mathbb{S}$, which is usually a one-way hash function and a verification function $v_A : \mathbb{M} \times \mathbb{S} \to \{\texttt{true}, \texttt{false}\}$. Agent A is the only one knowing g_A so that it can sign any message m and produce a signature $s = g_A(m)$, the verification function v_A however is made public so that anyone can verify if a certain message m whose signature is s was actually signed by A, i.e., $v_A(m, s) = \texttt{true}$.

In the definitions above, the words "easy" and "difficult" were used in the context of determining how "good" a cryptographic tool is. For instance, a successful digital signature system is one where it is difficult for any agent except A to produce a signature s from a message m such that $s = g_A(m)$. However, cryptographic operations are often used in hostile situations and it is almost certain that malicious intruders will try to "break" their security, i.e., defeat the purpose of their use. Cryptographers design cryptographic systems in order to achieve a certain security objective, while cryptanalysts try to develop methods to break these systems and obtain some information they are not entitled to. In Section 2.1.4, we try to formalize the notion of how "good" a cryptographic system is, but first, in the next section, we introduce security protocols.

2.1.3 Security Protocols

A communication protocol specifies a set of rules that have to be followed by agents wishing to communicate on a certain communication channel or network. More specifically, a protocol is a set of predefined communication steps between the agents, aiming to ensure a common language between them. A protocol also specifies a set of internal computation steps that each agent has to execute in order to update its internal state. Security protocols, on the other hand, are a subclass of communication protocols. They are used when agents communicate with the purpose of achieving some security objectives. For instance, they are used to authenticate agents to each other, to exchange cryptographic keys, or to guarantee that the data sent is not changed or replicated. Applications of cryptographic protocols are numerous such as e-commerce, online banking, electronic voting, etc.

Security protocols specify what operations (cryptographic or not) are used and by which agents in order to produce messages that agents exchange. Protocol design is a major issue in network security since that, even if we assume that all the cryptographic operations are unbreakable, i.e., the perfect cryptography assumption, there remain a number of very important issues to be considered. For instance, we need to decide what messages to encrypt and with which keys, how do we ensure a safe exchange of keys between agents, how to prevent replay attacks, etc. All of these are issues can compromise security *even* under the assumption of perfect cryptography.

In security protocol terminology, *principals* are the actual *agents* wishing to communicate. We use these two terms interchangeably. Sometimes, the names Alice and Bob, abbreviated as A and B, are also used to denote agents. In contrast, *roles* are specifications of agents. In other words, agents are instantiations of roles, which means that the same role can be played by more than one agent and one agent can play more than one role, in parallel or sequentially. As an example, in an actual network, there could be three agents playing the role R_1 and two agents playing the role R_2. Intuitively, to borrow from the object-oriented programming terminology, a role represents a class while an agent represents an instance of a class. To clarify the idea, an example is given below, which is taken from the Needham-Schroeder public key authentication protocol [87]:

$$\begin{array}{lll} \text{Step 1.} & A \rightarrow B: & \{A, N_A\}_{K_B} \\ \text{Step 2.} & B \rightarrow A: & \{N_A, N_B\}_{K_A} \end{array} \tag{2.1}$$

The notation used above is the standard notation used to describe protocols. It specifies what messages should be produced, by which agent, and to which agent they should be sent. The protocol specifies two roles: That of an initiator, which we call *init*, and the role of a responder, which we call *resp*. In the protocol description in (2.1), the role *init* is played by A and *resp* is played by B. In order to instantiate the role *init* into an agent, we must have the identities of the agents that will play the roles of initiator and responder, hence we can write the role *init* as $init(Agent_i, Agent_r)$. In (2.1), *init* is instantiated by the values A and B, i.e, $init(A, B)$. We did not include the values of N_A and K_B in the parameter list of *init* since they depend on the parameters $Agent_i$ and $Agent_r$ (A and B in (2.1)). Moreover, N_A depends also on the specific session in which $Agent_i$ will be involved, as explained below. The role *resp* has only one parameter however, which is the identity of the agent playing the role of responder. We can therefore write the role as $resp(Agent_{r'})$. We do not include the identity of the agent palying the initiator role as a prameter in *resp* since the responder agent can be instantiated without knowing who the initiator will be. A session of the protocol is an execution of protocol steps; it is started when both roles *init* and *resp* are instantiated such that $Agent_r$ in *init* is the same as $Agent_{r'}$ in *resp*, e.g., the roles are instantiated as $init(A, B)$ and $resp(B)$. In one session, communication steps in the protocol specification are executed sequentially. In step 1, A concatenates its identity to a nonce N_A, encrypts the whole with the public key of B and sends it as a message to B, where the notation $\{m\}_K$ means message m encrypted by key K (this should not be confused with the set notation). Nonces are produced by pseudorandom number generators that are implemented in agents and that create a new number whenever the agent contributes in a new session. They are used to guarantee freshness and hence prevent replay attacks, since a different value for N_A will be generated for each session in which agent A will contribute. In step 2, B replies by concatenating its own nonce to the nonce received from A, encrypting the whole message with the public key of A and sending the result to A.

From the discussion above we can deduce some of the following notations commonly used in standard protocol descriptions:

- The symbols $A, B, \ldots$ are used to denote agent identities and S is reserved for secure trusted servers.
- The symbol N_A is used to denote a nonce produced by A.
- The symbols K_{AB}, K_A, K_A^{-1} are used to denote a secret key between A and B, the public key of A and the private key of A, respectively.
- The symbol m_1, m_2 is used to denote the concatenation of messages m_1 and m_2.
- The symbol $\{m\}_K$ is used to denote the encryption of message m with key K.

Of course, the protocol specification above should be accompanied by natural language descriptions that helps to clarify several issues. For instance, it should be clear what are the nonces being produced, what is known to each agent prior to communication —in the case above it is the encryption key.

Making *sure* that a security protocol actually achieves its objectives is not an easy task. The analysis of security protocols aims at providing some level of assurance that the protocol actually does what is supposed to do. A large number of methods have been developed to this purpose. In the next section, we introduce the problem of analyzing cryptographic operations and protocols in order to verify that they are successful at reaching their desired security goals.

2.1.4 Analysis of Cryptographic Mechanisms

Several techniques have been developed in order to verify whether security mechanisms achieve their security objectives or not. As a simple example, the secrecy objective mentioned earlier in Section 2.1.1, may be formulated as "no information about the plain text should be easily obtained from the corresponding encrypted text unless the encryption key is known". Of course, we then have to clarify what we mean by "information" and "easily" and devise a measure that will indicate how "successful" a security mechanism is. We also have to take into account the presence and capabilities of a malicious attacker whose interest is to break the security of the cryptographic mechanism by preventing the achievement of its security objectives. Such issues constitute challenges in the domain of cryptographic analysis that comprises a number of approaches. They can be broadly classified into the computational approach [53, 99] and the formal verification approach [78].

In the computational approach, cryptographic operations are seen as functions operating on numbers or sequences of bits. Their success against attacks is assessed in probabilistic measures and depends on the computational complexity of the operations needed to mount a successful attack. For instance, using a brute force attack, which tries to decrypt a cipher text with every possible value of the key, it is computationally more demanding to decrypt a message that was encrypted with a 128-bit key than a message that was encrypted with an 8-bit key. Moreover, in this approach, it is possible to define partial information as a property of a sequence of bits. As an example, the attacker may not be able to know the value of a message but the value of one bit or if the message is odd or even.

In the formal verification approach, cryptographic operations are treated symbolically. Messages are just symbols or terms which can be manipulated by cryptographic functions that transform terms to other terms. Properties are also expressed symbolically as formulas of logics or formal specifications of systems. Formal methods are used to prove these properties. This approach has the advantage of abstracting away from the implementation details of complex systems and hence provide the opportunity for a more global reasoning focusing on the logic behind the design of mechanisms. Of course, both approaches are complementary and represent two different point of views to the same problem, which helps in discovering different types of attacks. In this section, we introduce some concepts of cryptanalysis, whereas formal methods are treated more thoroughly in the next chapter. A thorough presentation of cryptanalysis and pointers to references can be found in standard texts such as [79].

Cryptanalysis is the study of cryptographic operations in order to defeat the purpose of using cryptographic tools. In other words, it is the mathematical study of cryptographic properties of operations in order to be able to break the security of systems and/or mechanisms. For instance, a cryptanalyst may try to know the plain text of a cipher text without having the decryption key, or a malicious intruder may want to forge an agent's signature of a certain message. In this setting, there should be some criteria that a cryptographic system must satisfy for it to be as secure as possible. Finding such criteria is by no means an easy task. Several attempts have been made throughout history where the oldest techniques were focused on keeping the details of the cryptographic system secret, a practice that later became known as "security through obscurity". In 1883, Kerckhoff published his paper "La Cryptographie Militaire" [63] in which he mentions six criteria that became widely accepted as design principles for cryptographic systems. Perhaps the most important of which is that system details should not be kept secret, but secrecy should be reserved only for some system input. The logic behind this is that it is normally easier to change this secret input if it became known by an intruder as opposed to changing the whole system. In other words, security should not reside in the system (e.g., the algorithms) but in a piece of information that can be easily changed if compromised (e.g., the key). It then became standard to assume that the cryptanalyst trying to break the system's security knows all system details and can easily intercept the system's output (e.g., the cipher text). This assumption was used by Shannon [100] when he formulated the notions of perfect secrecy and practical secrecy. At that time asymmetric cryptographic systems where not invented yet and so his work focused on symmetric encryption.

Shannon's work wanted to address the notion of "security" of an encryption system. He formalized the notions of perfect secrecy and practical secrecy. Perfect secrecy means that intruders that intercept the cipher text should not be able to gain more knowledge about the plain text than they already had before obtaining the cipher text, even assuming no limits on their computational powers. This means that the system is unbreakable. The only system that fits this definition is the one using one time pads with totally random keys that are at least as long as the plain text [100]. All other systems are only practically secure, which means it is really "hard" for an intruder to break them. The degree of "hardness" is assessed using methods and terminology from the field of computational complexity [91]. Another approach is that of provable security [53] where statements about security are reduced to statements about known complex problems such as quadratic residuosity modulo composite integers. In this thesis, however, we focus on the formal analysis of security protocols.

2.2 Game Semantics

The idea of using games in logic specifications dates back to Lorenzen [69] who viewed the attempt to prove a logic fomula as a game. More specifically, in order to prove a logical proposition, a game is palyed between two players; one trying to assert it (the proponent) and the other trying to attack it (the opponent). The formula is proved if the proponent wins. In this case for instance, a disjunction is asserted if any of its constituents is asserted, whereas a conjunction is asserted only if all of its constituents are asserted. In the diagrams below, we show the games representing the logical OR and logical NOT, and how to prove the Law of Excluded Middle (LEM) using games. For the logical OR game, the opponent attacks the proposition, then the proponent chooses the constituent to defend, which is then attacked by the opponent. The logical NOT game begins by the opponent attacking the negation, while the proponent responds by attacking the proposition itself. A game is won by the player (opponent or proponent) who made the last move as long as the other player has no moves to play. This principle is used to prove the law of excluded middle as can be shown from the diagrams.

	Logical OR		Logical NOT		Law of Excluded Middle
	$A \vee B$		$\neg A$		$A \vee \neg A$
O	?	O	?	O	?
P	Choose to defend A	P	Attack A	P	Choose to defend $\neg A$
O	Attack A			O	Attack $\neg A$
				P	Attack A
				O	Assert A
				P	Assert A

The idea of using games to give semantics to logic propositions was further developed by Andreas Blass [26], who used it to give semantics to linear logic. Abramsky, Jagadeesan, and Malacaria [10] and Hyland and Ong [59] then, both independently, used game semantics to prove the full abstractness of the Programming language for Computable Functions (PCF) [95], which was a long standing problem in the theory of programming languages. The use of game semantics in programming languages has ever since been investigated and efforts done to further develop it [60]. A lot of work has been done for the use of game semantics in various programming languages [8, 9, 68]. The use of games in security protocols was investigated by Kremer and Raskin [66], where they followed a different approach than that taken by Abramsky and Hyland. More specifically they used Alternating Transition Systems (ATS) [17] as their model for the verification of fair exchange and non-repudiation protocols. However, to the best of our knowledge, there are not yet any publications that make full use of the power of game semantics to define a formal semantics for security protocols. In the next sections, we introduce the main concepts of game semantics, which will be later used throughout this dissertation.

2.2.1 Games

Intuitively, a game is a sequence of plays (moves) between two parties (*players*): A *proponent* P and an *opponent* O. There can also be multi-party games, but these can be expressed as combined two-party games. So we will focus our interest on two-party games and operations on these games. In the context of game semantics, the proponent represents the system, while the opponent represents the environment. A sequence of moves in the game represents a certain interaction between the system and its environment. Each move in this interaction takes the form of a question Q or an answer A. For instance, the environment can ask for a value (question), and the system supplies this value (answer) directly, *or* asks the environment for more detail (question), and so on. We adopt the convention that the opponent always makes the first move (i.e. the environment starts the interaction) then the game proceeds as alternating moves between proponent and opponent. Formally a game G is a structure (M_G, λ_G, P_G) where [9]:

$$\begin{array}{ll}
M_G & \text{Set of moves of the game.} \\
\lambda_G : M_G \to \{P, O\} \times \{Q, A\} & \text{Labeling function for each move} \\
\lambda_G = \langle \lambda_G^{PO}, \lambda_G^{QA} \rangle & \\
\lambda_G^{PO} : M_G \to \{P, O\} & \text{Labeling function} \\
\lambda_G^{QA} : M_G \to \{Q, A\} & \text{Labeling function} \\
P_G \subseteq^{nepref} M_G^{alt} & \text{Non-empty, prefix closed set of sequences}
\end{array} \tag{2.2}$$

The domain M_G is the set of moves. The three labeling functions are used to classify moves as questions Q or answers A and as moves of proponent P or opponent O:

$$\forall m \in M_G,\ \lambda_G(m) = x, \text{ where } x \in \{PQ, PA, OQ, OA\}$$
$$\forall m \in M_G,\ \lambda_G^{PO}(m) = x, \text{ where } x \in \{P, O\}$$
$$\forall m \in M_G,\ \lambda_G^{QA}(m) = x, \text{ where } x \in \{Q, A\}$$

For instance, PA means the move is an answer by the proponent. We write M_G^* for the set of finite sequences over M_G. A sequence $s = s_1.s_2\ldots s_n$ has length $|s| = n$. Then, M_G^{alt} is defined as follows:

$$\begin{aligned} M_G^{alt} = \{s \mid s \in M_G^*,\ \ \forall i\,.\,1 \leqslant i \leqslant |s|, (even(i) \Rightarrow \lambda_G(s_i) = P)\wedge \\ (odd(i) \Rightarrow \lambda_G(s_i) = O)\} \end{aligned} \tag{2.3}$$

Informally, M_G^{alt} is the set of finite-length sequences of moves such that the first move belongs to the opponent, the second to the proponent, and so on. The domain P_G is a non-empty, prefix closed set of sequences of moves, each of these sequences represents a possible interaction during the play of the game. The domains P_G^{even} and P_G^{odd} are the sets of even- and odd- length sequences respectively. Prefix closure means that $\mathbf{Pref}(P_G) = P_G$; $\mathbf{Pref}(P_G)$ is defined below, where s and t are any two sequences:

$$\begin{aligned} \mathbf{pref}(st) &= s \\ \mathbf{Pref}(P_G) &= \{\mathbf{pref}(s) \mid s \in P_G\} \end{aligned}$$

It is worth noting that the first equation can also be written, in another notation, as $s \sqsubseteq st$, meaning that s is a prefix of st. Considering the previous definitions, the set P_G can be graphically represented as a tree, where each sequence $s \in P_G$ represents a path in the tree. The moves of the sequence will be labels on edges of the path and, for any two sequences s and s' in P_G, if $s \sqsubseteq s'$ then the path of s' will contain the path of s. We can therefore call P_G the game tree.

A deterministic *strategy* for the proponent (the system) in a game G is a set

σ_G of sequences of moves, where:

$$\begin{array}{rcl} \sigma_G & \subseteq & P_G^{even} \\ \epsilon & \in & \sigma \\ sab \in \sigma & \Rightarrow & s \in \sigma \\ sab, sac \in \sigma & \Rightarrow & b = c \end{array} \tag{2.4}$$

Here $s, t, u, \ldots$ represent sequences, and $a, b, c, \ldots$ represent single moves. Intuitively, a strategy is a subtree of the game tree that has the following properties:

- Each path in the strategy contains an even number of moves (this is stated by the third condition in (2.4) above.
- At any position in a path that ends with an opponent move, there at most one move for the proponent to make (last condition above). This is why the strategy is called deterministic

2.2.2 Operations on Games

Before defining operations on games, we give the following definitions:

For any two sets X and Y, the set $Z = X \uplus Y$ is their disjoint union. If the sequence $s \in Z^*$, then $s \upharpoonright X \in X^*$, which means $s \upharpoonright X$ is the sequence obtained by removing, from s, all the elements not in X.

Tensor Product

For any two games G and H, the tensor product $G \otimes H$ is defined as:

$$\begin{array}{rcl} M_{G\otimes H} & = & M_G \uplus M_H \\ \lambda_{G\otimes H} & = & [\lambda_G, \lambda_H] \\ P_{G\otimes H} & = & \{s \in M_{G\otimes H}^{alt} \mid (s \upharpoonright M_G \in P_G) \wedge (s \upharpoonright M_H \in P_H)\} \end{array} \tag{2.5}$$

As previously mentioned, all games start by O making a move. Here O can decide to make a move in G or H. The third condition above means that O can decide to make a move in either game (G or H), whereas P has to play in the game that O chose. This means that for any two consecutive moves s_i and s_{i+1} if s_{i+1} is a move of a subgame different than that of s_i, then $\lambda^{PO}_{G\otimes H}(s_i) = P$ and $\lambda^{PO}_{G\otimes H}(s_{i+1}) = O$. This is called the *switching condition* [9]. Intuitively, the tensor products represents the interleaved play of two games, subject to the switching condition.

Duality of Games

For any game G, its dual $G^{\perp}$ is obtained by interchanging the roles of the two players (P and O). More formally, we can define:

$$\begin{aligned} M_{G^{\perp}} &= M_G. \\ \forall m \in M_{G^{\perp}} \,.\, \lambda_{G^{\perp}}(m) &= \overline{\lambda_G}(m) \end{aligned} \tag{2.6}$$

where:

$$\overline{\lambda_G} = \langle \overline{\lambda^{PO}_G}, \lambda^{QA}_G \rangle$$

$$\overline{\lambda^{PO}_G}(a) = \begin{cases} P \text{ when } \lambda^{PO}_G(a) & = O \\ O \text{ when } \lambda^{PO}_G(a) & = P \end{cases}$$

Linear Implication

For any two games G and H, the game $G \multimap H$ is defined as:

$$\begin{aligned} M_{G\multimap H} &= M_G \uplus M_H \\ \lambda_{G\multimap H} &= [\overline{\lambda_G}, \lambda_H] \\ P_{G\multimap H} &= \{s \in M^{alt}_{G\multimap H} \mid (s \upharpoonright M_G \in P_G) \wedge (s \upharpoonright M_H \in P_H)\} \end{aligned} \tag{2.7}$$

This is equivalent to playing the game $G^{\perp} \otimes H$. In this case, since we assume that the first move is always done by O, then the first move will be in H because in G, the opening move is now labeled with P not O. It is worth noting that the switching condition in $G \multimap H$ now means that P is free to make a move in any of the subgames (G or H), while O has to play in the subgame chosen by P. Equivalently, for any two consecutive moves s_i and s_{i+1} if s_{i+1} is a move of a subgame different than that of s_i, then $\lambda^{PO}_{G\otimes H}(s_i) = O$ and $\lambda^{PO}_{G\otimes H}(s_{i+1}) = P$.

2.2.3 Enabling Relation

An *enabling* relation is defined over the set $M_G \cup \{\star\}$, where $\star$ is just a dummy symbol. The enabling relation means that a move cannot be played unless it was enabled (justified) by another move. The first move in the game is justified by the dummy symbol. The enabling relation is defined as:

$$\begin{array}{lcl} m \rightsquigarrow_G m' & & m' \text{ cannot be played unless } m \text{ was played} \\ \star \rightsquigarrow_G m & \Rightarrow & \lambda_G(m) = OQ \wedge (n \rightsquigarrow_G m \Leftrightarrow n = \star) \end{array} \qquad (2.8)$$

The idea behind the enabling relation is that a move cannot be made unless it was preceded with another move enabling it. This is what is stated in the first relation above. The $\star$ is special in that it enables the first move of the game, called the *opening* or *initial* move, i.e. a move m such that $\star \rightsquigarrow_G m$. This is what is stated in the second form of the enabling relation in (2.8) above, which states also that the initial move of the game is always a question by the opponent. It is important to note that this relation is *not* transitive. So, if $m_1 \rightsquigarrow_G m_2$ and $m_2 \rightsquigarrow_G m_3$, this means that *both* m_1 and m_2 have to be played before m_3 can be played.

2.2.4 Equivalence Relation

An equivalence relation $\sim_G$ is defined on P_G, which satisfies the three conditions below, where s and t denote sequences of moves, while a and b denote single moves:

$$\begin{aligned} s \sim_G t \;&\Rightarrow\; \lambda_G^*(s) = \lambda_G^*(t) \\ (s \sim_G t) \wedge (s' \sqsubseteq s) \wedge (t' \sqsubseteq t) \wedge (|s'| = |t'|) \;&\Rightarrow\; s' \sim_G t' \\ (s \sim_G t) \wedge (sa \in P_G) \;&\Rightarrow\; \exists b \,.\, sa \sim_G tb \end{aligned} \tag{2.9}$$

The first condition means that if two sequences are equivalent, then each of their moves will have the same label where a label can be one of the four cases (PQ, PA, OQ, OA) as previously mentioned. This also implies that the two sequences will have the same length. Here we denote the labeling function by λ^* instead of λ which was defined for single moves and not sequences. The second condition simply states that if two sequences are equivalent then any two prefixes of these sequences will also be equivalent, provided they have the same length. The third condition, on the other hand, implies that it is always possible to extend two equivalent sequences while maintaining their equivalence.

2.2.5 Category of Games

In the context of game semantics, games are used as the semantic domain when assigining denotations to programs. A program is interpreted as a strategy over a game. In order to define a mathematical structure for the domain of games and to investigate its properties, category theory is used [9]. To define a category, we need to define the following [94]:

- Objects.
- Arrows (morphisms).

- Relationships of arrows to objects: Each arrow f relates two objects, its domain *dom* f and its codomain *cod* f. For instance, in $f : A \longrightarrow B$, *dom* $f = A$, and *cod* $f = B$. We can also use the notation $A \xrightarrow{f} B$.

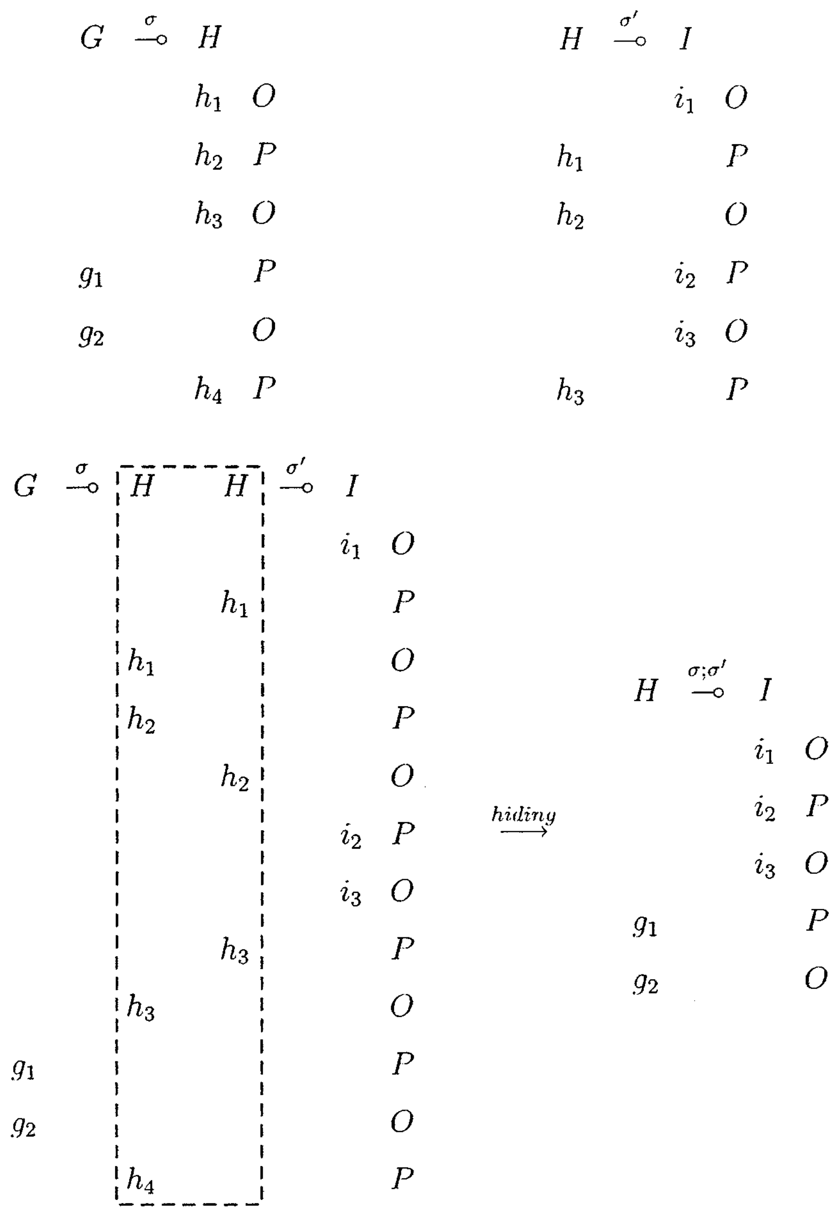
- Composition of morphisms. For any two morphisms f and g, where $A \xrightarrow{f} B$, and $B \xrightarrow{g} C$, the composed morphism $g \circ f$ means that $A \xrightarrow{g \circ f} C$. Composition should be associative, i.e., $h \circ (g \circ f) = (h \circ g) \circ f$.

- An identity arrow id_A, where $A \xrightarrow{id_A} A$. For any arrow $A \xrightarrow{f} B$, we have $id_B \circ f = f$ and $f \circ id_A = f$.

A category of games is defined where objects are games. A morphism between two objects (games) G and H is a strategy σ on $G \multimap H$ ($\sigma : G \multimap H$), which is sometimes written as $G \overset{\sigma}{\multimap} H$. The composition of morphisms is defined to be *interaction between strategies*; if we have $\sigma : G \multimap H$, and $\tau : H \multimap I$, the composition of these two strategies is $\sigma;\tau : G \multimap I$. The composed strategy $\sigma;\tau$ is formally defined as:

$$\begin{aligned}
\sigma;\tau &= (\sigma\|\tau)/H = \{s \upharpoonright (G, I) \mid s \in \sigma\|\tau\} \\
\sigma\|\tau &= \{s \in (M_G + M_H + M_I)^* \mid s \upharpoonright (G, H) \in \sigma \wedge s \upharpoonright (H, I) \in \tau\} \qquad (2.10) \\
& s \upharpoonright (G, I) \text{ is shorthand for } (s \upharpoonright G) \upharpoonright I
\end{aligned}$$

This is similar to the process algebra concept of "parallel composition plus hiding" [9]. An example is shown below, first by giving the individual strategies σ and σ', then $\sigma\|\sigma'$ and finally deducing the strategy $\sigma;\sigma'$ by hiding moves in the game H:


$G \overset{\sigma}{\multimap} H$
$H \overset{\sigma'}{\multimap} I$
$G \overset{\sigma}{\multimap} H \quad H \overset{\sigma'}{\multimap} I$
hiding
$H \overset{\sigma;\sigma'}{\multimap} I$

The identity morphism is the copy cat strategy id_G over the game $G \multimap G$ ($G \overset{id_g}{\multimap} G$), which is denoted by $\widehat{G \multimap G}$. Sometimes, the strategy is written $G_1 \overset{id_g}{\multimap} G_2$ to differentiate between the two instances of game G. An example of a copy cat strategy is given below, where moves of the game G are denoted by a_i for moves of the opponent and b_i for moves of the proponent, $i = 1, 2, 3, \ldots$ Here, we notice that moves of the opponent in G_2 are mapped by the proponent in G_1.

$$\begin{array}{cccc}
G_1 & \overset{id_G}{\multimap} & G_2 & \\
 & & a_1 & O \\
a_1 & & & P \\
b_1 & & & O \\
 & & b_1 & P \\
 & & a_2 & O \\
a_2 & & & P \\
\vdots & & &
\end{array}$$

The formal definition of a copy cat strategy is given in (2.11) below:

$$id_G = \{s \in P^{even}_{G \multimap G} \mid \forall t \text{ even-length prefix of } s : t \upharpoonright G_1 = t \upharpoonright G_2\} \tag{2.11}$$

It is shown in [9] that the previous definition of the category of games gives rise to a Cartesian Closed Category (CCC).

In this chapter we have introduced basic concepts of security protocols and game semantics. These concepts will be used througout this dissertation with the same notations. In the next chapter, we survey the state-of the-art in the specification and verification of security protocols.

Chapter 3

Specification and Verification of Security Protocols

Formal analysis of security protocols starts by choosing, or designing, a model that formally expresses the main protocol features, namely: Message construction, communication steps and the intruder. In order for the model to be used for actual analysis of protocols, it should be equipped with a certain language that will enable the formal specification of protocols. This language allows the "coding" of a protocol into model components. At this stage, formal verification can start; it uses these components in order to reason about protocol behavior and ascertain that it satisfies its required properties. The properties, of course, should be expressed in a language related to the model. This general view of the formal analysis of security protocols encompasses both model checking and logic-based techniques.

This thesis is based on the idea of using game semantics as the underlying model for the specification and analysis of security protocols. In this chapter, we survey the state-of-the-art in formal specification and verification methods as applied to security protocols. First, we discuss different modeling techniques, we then present various formal verification methods along with the most prominent software tools in security protocol analysis.

3.1 Modeling Security Protocols

The primary and crucial step in the application of formal methods to the analysis of security protocols is the choice of an execution model. This choice will determine the expressive and analytical power with which the protocol can be analyzed. The protocol model should take into account the following features:

- The computation steps done by communicating agents in order to compute messages before sending them.
- The communication steps between agents.
- The presence of a malicious intruder that can manipulate messages sent over the network.
- The cryptographic operations used in protocol messages, e.g., encryption, hashing, generation of random numbers, etc.

Moreover, some protocol models may also consider the possible presence of dishonest agents or some real-time aspects such as timeouts and timestamps. Dishonest communicating agents try to "cheat", and in some protocols such as fair exchange protocols [74], their behavior is considered. Real-time aspects, on the other hand, can also be analyzed [89] in order to verify some protocols such as the Wide-Mouthed Frog protocol [29] that uses timestamps.

In the next subsections, we will describe different published models of security protocols. We begin by introducing intruder models since they are a vital part of the analysis. Then we present protocol models classified into into two categories: Behavior description models and logic-based models.

3.1.1 Intruder Models

An intruder model addresses the question of intruder capabilities. Namely, what are the messages the intruder is able to collect and generate. More specifically, what is the initial knowledge of the intruder before protocol execution, and how does its knowledge increase during protocol execution. This increase in knowledge is due to messages that the intruder intercepts and to computations it is able to do. With respect to computations, two different approaches exist: The formal approach and the computational approach. In the formal or Dolev-Yao approach, which originated in the work of D. Dolev and A.C. Yao [44], messages are formal, i.e., symbolic, terms. Computations with messages are also symbolic and follow a defined set of rules as will be explained below. In the computational approach [53], messages are bit-strings and the intruder is a probabilistic polynomial-time (PPT) Turing machine that performs computations on messages. The methods and techniques of cryptanalysis are used, they involve the notions of probability and partial information all of which are absent in the formal model. It is established that a computational intruder is more realistic than a formal one since it deals with bit-strings and can carry out numerical calculations. The formal approach, however, is more suitable for automatic verification [13, 25, 111]. Some research work was done in order to bridge the gap between the two approaches [6, 38].

Original Dolev-Yao Model

This Model is due to D. Dolev and A. C. Yao [44] and is the first formulation of a formal model of the intruder. It has been extensively used ever since. The model assumes that the intruder has total control over the network and has the following capabilities:

- It can monitor all messages sent through the network.
- It can intercept (i.e., block) messages.
- It can generate new messages from its initial knowledge before protocol execution and the messages it collects during the execution.

- It is a legitimate user of the network and therefore has its own public key and can send messages to agents, receive messages from agents, initiate a protocol with any other agent, etc.

It is assumed that cryptographic operations are unbreakable, i.e., the perfect cryptography assumption, and that the cipher text reveals no information about its plain text. The intruder knowledge is defined by a set of formal rules that specify which messages the intruder can deduce given the set $\mathbb{M}$ of messages it already obtained. The intruder starts out with the set $\mathbb{M}$ containing its initial knowledge such as the public keys of all agents. It then adds to $\mathbb{M}$ all messages exchanged on communication channels, since it has total control over the network. In the original Dolev-Yao model, the set of rules includes only encrypting, decrypting, pairing and unpairing of messages. The assumption of perfect cryptography implies that decryption is impossible without knowing the key. Below, we formalize these rules in a deductive style where $\mathbb{M} \vdash m$ means that message m can be deduced from the set $\mathbb{M}$ of messages.

$$\begin{array}{cc}
\multicolumn{2}{c}{(\mathsf{know})\ \dfrac{\cdot}{\mathbb{M} \vdash m}\ m \in \mathbb{M}} \\
(\mathsf{encrypt})\ \dfrac{\mathbb{M} \vdash m \quad \mathbb{M} \vdash K}{\mathbb{M} \vdash m_K} & (\mathsf{decrypt})\ \dfrac{\mathbb{M} \vdash m_K \quad \mathbb{M} \vdash K}{\mathbb{M} \vdash m} \\
(\mathsf{pair})\ \dfrac{\mathbb{M} \vdash m_1 \quad \mathbb{M} \vdash m_2}{\mathbb{M} \vdash m_1, m_2} & (\mathsf{unpair})\ \dfrac{\mathbb{M} \vdash m_1, m_2}{\mathbb{M} \vdash m_1 \quad \mathbb{M} \vdash m_2}
\end{array} \tag{3.1}$$

The advantages of the Dolev-Yao model is that it enables automatic verification of protocols. However, we can summarize its limitations to be [83]:

- The absence of cryptanalysis which means that we cannot incorporate any known weaknesses of cryptographic operations into the verification process.

- The absence of probabilistic information, for instance the properties of any used pseudorandom number generation or any statistical analysis that can be made by the intruder.

- The absence of the notion of partial information, for instance the intruder can know some information about the plain text from its corresponding cipher text such as parity check.

Extended Dolev-Yao Model

Some research papers [3, 37] investigate increasing the power of the intruder by expanding its capability to generate messages. This is done by incorporating the characteristics of the cryptographic system in use into the analysis. In this case, messages are terms of a term algebra [112] that has a signature Σ and that is equipped with a set E of equations. This set expresses the algebraic properties of the cryptographic system, it also generates a congruence relation $=_E$ between the terms (messages). To determine if two messages m_1 and m_2 are congruent, i.e., $m_1 =_E m_2$, term rewriting techniques are used. In order to take congruent terms into account when analyzing protocols, the set of deductive rules of the Dolev-Yao system in (3.1) above is extended by the rule:

$$\text{(equation)}\ \frac{\mathbb{M} \vdash m_1}{\mathbb{M} \vdash m_2}\ m_1 =_E m_2 \tag{3.2}$$

The rule above basically means that of the intruder can generate m_1, then it can generate m_2, if m_1 and m_2 are congruent under the set E of equations.

Computational Model

The computational model of the intruder is one that involves cryptanalysis. As mentioned earlier, the intruder is assumed to have certain computational capabilities, and is generally modeled as a Probabilistic Polynomial Time (PPT) Turing machine having messages as input. These messages are sequences of bits upon which the Turing machine can perform computations. This model is used in cryptographic analysis of protocols the most famous of which is the concept of provable security [53]. In this case, security depends on the computational complexity of a known problem, such as quadratic residuosity modulo composite integers [65]. This means that if an intruder is able to break the security of the cryptographic system, then it is able to develop an efficient algorithm for this problem.

3.1.2 Behavior Description Models of Protocols

Models that describe protocol behavior are basically transition systems in which each agent possesses a countable set of states and communicates with other agents through the network. The global state of the system depends on the state of each agent and the messages in transit in the network. In the following, we present the main behavior models used for security protocols, namely, finite state machines, strand spaces, the model of the inductive approach, process algebra, and, finally, a game-theoretic model.

Finite State Machines

In this model, communicating agents are finite state machines. The intruder is modeled by the set of messages that it knows and the set of actions that it can take, which gives rise to a nondeterministic finite state machine. The nondeterminism stems from the fact that we cannot predict what exact message the intruder will send, since it can send any message that it can deduce from its knowledge. Message computation is implicitly modeled by transition rules that specify the structure of output messages given the reception of expected input messages. Communication takes place via shared variables or queues. Several examples exist in the literature [83, 85, 101]. Also, several techniques are presented [102] to reduce the state space and render the model more efficient to verify. We give below an example using the

Needham-Schroeder public key protocol [87].

The protocol is described using standard notation as:

$$\begin{array}{lll} \text{Step 1.} & A \rightarrow B: & \{N_A, A\}_{K_B} \\ \text{Step 2.} & B \rightarrow A: & \{N_A, N_B\}_{K_A} \\ \text{Step 3.} & A \rightarrow B: & \{N_B\}_{K_B} \end{array} \tag{3.3}$$

The system is modeled as three communicating finite state machines [85], and communication takes place asynchronously through the global variable `net`. The first machine is that of the initiator, playing the role of A above, the second is the responder, playing the role of B and finally the last is the intruder. As an example, the initiator has three states: `I_SLEEP`, `I_WAIT`, and `I_COMMIT` and two transitions. The first transition fires from the `I_SLEEP` state and so the output, i.e., the first message of the protocol, is assigned to the variable `net` and the state is changed to `I_WAIT`. The second transition fires from the state `I_WAIT` upon receiving the message of step 2 of the protocol, which means reading the variable `net` and checking that it has the right structure and that it is destined to the correct initiator since the system may have more than one. The state is then changed to `I_WAIT`. The responder, on the other hand has two states: `R_WAIT` and `R_COMMIT` and one transition. The tricky part of the system is modeling the intruder as a finite state machine. The intruder model used is that of Dolev-Yao. However, some restrictions are imposed in order to reduce the state space. For instance, the intruder is not modeled as a non-deterministic finite state machine that can send any message at any time. Instead, the messages sent are restricted to only those that a communicating agent will accept according to the checks it makes on the message format. This does not undermine the capabilities of the intruder since it makes no sense to allow it to send messages that will later get refused by other agents. The system can then be run using various configurations and the reported results [85] show that when having two initiators, two responders and an intruder, the number of generated states is 514550. This data exemplifies the well-known problem of state explosion when specifying systems as finite state machines. To reduce the number of states, several techniques are used [102], they include:

- The intruder always intercepts all messages and then it can send the same message it intercepted or another one to the honest agent waiting to receive a message.
- When there is a choice in the state graph between the intruder sending a message and an honest agent sending a message, we choose the path where the agent sends the message.

The soundness of the above techniques is proved by showing that any attack that is discoverable in the large state space will also be discoverable in the reduced state space.

Aside from the state explosion problem, a criticism to this approach is that it is not usually straightforward to model agents, and particularly the intruder as FSMs. The same applies to the algebraic properties of messages. Finally, there is no unified approach to the specification of security properties that have to be satisfied by the system of communicating FSMs.

Strand Spaces

A strand represents a sequence of actions performed by a certain agent in a protocol session. Each action can be either sending or receiving a message. A strand space [46, 47] is a set of strands and a mapping between each strand and a sequence of messages. The mapping need not be injective so that two strands may map to the same sequence of messages. In this way, a strand can be thought of as an actual execution of a sequence $m_1.m_2.m_3\ldots$ of messages by an agent. This sequence represents one session of the protocol and if the agent is involved in more than one session at the same time, then each session is represented by its own strand. The messages themselves are signed terms of an algebra where the + sign means that the message is sent and the $-$ sign means the message is received. Messages are defined by the following grammar:

$$\begin{aligned}\mathsf{Msg} &::= +\mathsf{Trm} \mid -\mathsf{Trm}\\ \mathsf{Trm} &::= \mathsf{Txt} \mid \mathsf{Key} \mid \mathsf{Trm}_{\mathsf{Key}} \mid [\mathsf{Trm}, \mathsf{Trm}]\end{aligned} \tag{3.4}$$

In (3.4) above, Txt represents atomic terms (i.e., agent names and numbers), Key represent keys and the last two terms represent encryption of a term with a key and the concatenation of two terms, respectively. Each encryption key K is assumed to have a decryption key K^{-1}. It is also assumed that typing rules will allow agents to differentiate between different types of terms, for instance, an encrypted term will not be confused with a key.

Each communicating agent is represented by a certain strand that indicates which messages this agent receives and which messages it sends and in what order. The order is imposed by the actual sequence of messages in the strand. In graphical terms, a strand s that is N messages long is a graph with N nodes (vertices). Each node corresponds to a certain message on the strand. For a given strand s and a node n, $\mathbf{term}(n)$ is the message in s that corresponds to n, and $\mathbf{index}(n)$ is the index of this message on the strand, where the index of the first message equals 1. There are two types of directed edges: Edges that connect two nodes of the same strand and edges that connect two nodes on different strands. An edge of the first type is directed from a node n_1 to a node n_2 if and only if $\mathbf{index}(n_2) = \mathbf{index}(n_1) + 1$, whereas an edge of the second type is directed from a node n_1 to a node n_2 if and only if $\mathbf{term}(n_1) = +a$ and $\mathbf{term}(n_2) = -a$. The set of nodes together with the edges define a graph that represents the strand space. A bundle is an acyclic finite subgraph of this graph such that if any node in the bundle is at the destination of an edge, then the bundle also contains the node from which the edge originates. Intuitively, strands represent communicating agents and bundles represent one or more complete protocol sessions between the agents.

Therefore, message computation is implicitly modeled by the structure of messages as terms of an algebra and communication steps are modeled by the edges that connect two nodes having the same message but with different signs. The intruder, on the other hand, is Dolev-Yao and is modeled by special strands called penetrator strands. They represent the possible manipulations that can be done by the intruder to messages, namely: (1) creating an atomic message (the strand will have a single message, i.e., $+t$) (2) blocking a message ($-t$), (3) forwarding a message (the intruder will receive the message then send it without change ,i.e., $-t.+t$), (4) concatenating messages ($-g.-h.+[g,h]$), (5) separating concatenated messages to their components ($-[g,h].+g.+h$), (6) generating keys ($+K$), (7) encrypting messages with keys ($-K.-t.+t_K$), and finally (8) decrypting messages with known keys ($-K^{-1}.-t_K.+t$). We can also combine the previous strands to have longer strands since the intruder can be engaged in successive interactions with other agents.

Strand spaces are called infiltrated if they contain penetrator strands. Figure 3.1 shows the infiltrated strand space of the Needham-Schroeder protocol. It has five columns, the leftmost and the rightmost ones show the strands for agent A and agent B respectively, whereas the three middle columns represent a bundle in the infiltrated strand space of the protocol.

It was pointed out [57] that the strand space model does not make a clear distinction between an agent and a strand. We share this view, and think that this will limit the ability to analyze multiple runs of one protocol executed by the same agent, or multiple runs of different protocols for that matter.

Model of the Inductive Approach

In the inductive approach [92], a protocol is a set of traces, where each trace is a sequence of events. Events represent different operations such as sending a message from one agent to the other or storing a message. Messages used in an event are terms

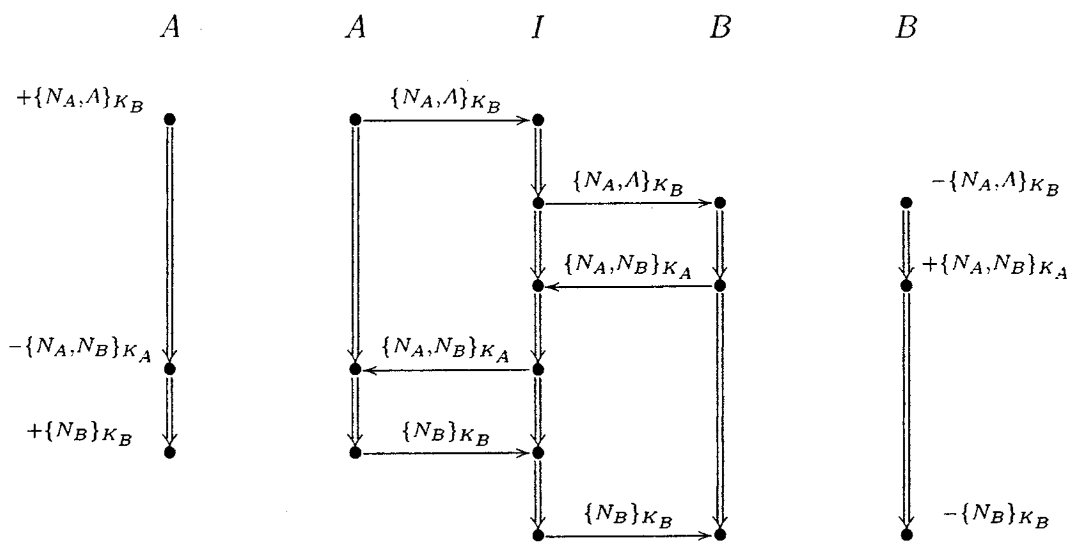


Figure 3.1: Attack on the Needham-Schroeder protocol.

of a term algebra with variables, where variables are used to represent messages that agents are willing to accept without verification because they do not know the message beforehand, e.g., the message is encrypted with a key that they do not have. Messages are defined by the following BNF grammar:

$$\mathsf{Msg} ::= \mathsf{Agt} \mid \mathsf{Key} \mid \mathsf{Nat} \mid \mathsf{Var} \mid [\mathsf{Msg}, \mathsf{Msg}] \mid \mathbf{Crypt}\mathsf{KeyMsg} \tag{3.5}$$

In the production rule above, a message can be an agent name, a key, a natural number (to represent nonces), a variable, the concatenation of two messages or the encryption of a message with a key. The symbol m is used to range over messages, i.e., sentences derived from Msg. Agents, keys, nonces, and variables are specified by:

$$\begin{array}{ll} \mathsf{Agt} ::= A \mid B \mid \ldots & \mathsf{Key} ::= K_A \mid K_A^{-1} \mid K_{AB} \mid K_{AB}^{-1} \mid \ldots \\ \mathsf{Nat} ::= N_A \mid N_B \mid \ldots & \mathsf{Var} ::= X \mid Y \mid \ldots \end{array} \tag{3.6}$$

It is clear that both symmetric and asymmetric encryption are allowed, the decryption key of a given key K is denoted by K^{-1} where, for symmetric encryption, $K_{AB}^{-1} = K_{AB}$.

In the paper by Paulson [92], only one type of events is presented and has the form **Says**$A\ B\ m$ where A, B are agent names, and m is a message. This event is meant to represent the transmission of message m from A to B. Traces of events, which represent possible protocol executions, are defined inductively by a set of rules that determine how a trace can be extended by an event. For any protocol, there exist two sets of rules: Those that are protocol-dependent, and those that are valid for any protocol. Before giving an example, we discuss the intruder model. The intruder is Dolev-Yao and therefore have access to all messages sent to the network. Blocking messages is implicitly modeled by the fact that there will be valid traces of the protocol in which an agent does not receive the message it expects. This is the case since agents are not required to respond to every message they receive, even if it is valid. In order to model the intruder capabilities to generate messages, let $\mathbb{H}$ be the set of messages in a certain trace evs, i.e, they appear in the events of the trace, $\mathbb{H}$ also contains the messages that are publicly known such as agent names and public keys. Two sets of messages are defined inductively from $\mathbb{H}$, these are **analz**$\mathbb{H}$ and **synth**$\mathbb{H}$, they are defined by the following rules:

$$\begin{array}{rcl}
m \in \mathbb{H} & \Rightarrow & m \in \mathbf{analz}\mathbb{H} \\
[m_1, m_2] \in \mathbf{analz}\mathbb{H} & \Rightarrow & m_1 \in \mathbf{analz}\mathbb{H} \ \wedge \ m_2 \in \mathbf{analz}\mathbb{H} \\
\mathbf{Crypt}K\ m \in \mathbf{analz}\mathbb{H} \ \wedge \ K^{-1} \in \mathbf{analz}\mathbb{H} & \Rightarrow & m \in \mathbf{analz}\mathbb{H} \\
m \in \mathbb{H} & \Rightarrow & m \in \mathbf{synth}\mathbb{H} \\
m_1 \in \mathbf{synth}\mathbb{H} \ \wedge \ m_2 \in \mathbf{synth}\mathbb{H} & \Rightarrow & [m_1, m_2] \in \mathbf{synth}\mathbb{H} \\
m \in \mathbf{synth}\mathbb{H} \ \wedge \ K \in \mathbf{synth}\mathbb{H} & \Rightarrow & \mathbf{Crypt}K\ m \in \mathbf{synth}\mathbb{H}
\end{array} \tag{3.7}$$

An intruder can generate a message m from a trace evs if and only if $m \in$ **synth**(**analz**$\mathbb{H}$). The intruder can then send m, this is modeled by the event

Says$Spy\ A\ m$, where Spy denotes the intruder.

We note here that, in the inductive model, message computation is modeled by the structure of messages using algebraic operators, and communication steps are expressed using events of the form **Says**$A\ B\ m$. The intruder's knowledge is defined using the two operators: **synth** and **analyz** that model its capabilities to analyze and construct messages.

We now give an example using the Needham-Schroeder (NS) protocol defined in (3.3). The protocol is defined as a set $\mathbb{P}_{NS}$ of traces (i.e., sequences of events) by the following set of rules, where α, β, γ are traces and for a trace evs and event v, we denote the concatenation of v to evs by $evs.v$. The rules define the set $\mathbb{P}_{NS}$ inductively:

$$\frac{evs \quad \in \mathbb{P}_{NS}}{evs.\mathbf{Says}A\ B\ \mathbf{Crypt}K_B\ [N_A, A] \quad \in \mathbb{P}_{NS}}$$

Conditions: $A \neq B$, N_A is a fresh nonce

$$\frac{\alpha.\mathbf{Says}A'\ B\ \mathbf{Crypt}K_B\ [N_A, A].\beta \quad \in \mathbb{P}_{NS}}{\alpha.\mathbf{Says}A'\ B\ \mathbf{Crypt}K_B\ [N_A, A].\beta.\mathbf{Says}B\ A\ \mathbf{Crypt}K_A\ [N_A, N_B] \quad \in \mathbb{P}_{NS}}$$

Condition: N_B is a fresh nonce

$$\frac{\alpha.\mathbf{Says}A\ B\ \mathbf{Crypt}K_B\ [N_A, A].\beta.\mathbf{Says}B'\ A\ \mathbf{Crypt}K_A\ [N_A, N_B].\gamma \quad \in \mathbb{P}_{NS}}{\alpha'.\beta.\mathbf{Says}B'\ A\ \mathbf{Crypt}K_A\ [N_A, N_B].\gamma.\mathbf{Says}A\ B\ \mathbf{Crypt}K_B\ N_B \quad \in \mathbb{P}_{NS}}$$

$\alpha' = \alpha.\mathbf{Says}A\ B\ \mathbf{Crypt}K_B\ [N_A, A]$

(3.8)

The rules in (3.8) are derived from the protocol, we note the following:

- In the premise of the second and third rule, the use of B' indicates the fact that an agent cannot be sure of the origin of the message relying on the source address since it can be spoofed.

- The premise of the third rule contains two events in the sequence to clearly indicate which agents are communicating in each step. So, we notice that in the last event in the conclusion of the rule, agent A sends a message to agent B although it may have received a message from another agent B'. In fact, this agent B is the same to which agent A sent the first message of the protocol, i.e., $\textbf{Says}A\ B\ \textbf{Crypt}K_B\ [N_A, A]$.

In rules (3.8), we did not use variables since all keys are public, however, we illustrate their use using a toy example, where we have agents A, B and a trusted server S with which A and B share secret keys K_{AS} and K_{BS} respectively. Consider the following two steps: $A \longrightarrow B : \{A, N_A\}_{K_{AS}}$ and then $B \longrightarrow S : \{B, N_B, \{A, N_A\}_{K_{AS}}\}_{K_{BS}}$, the rule corresponding to the second step would be:

$$\frac{\alpha.\textbf{Says}A'\ B\ X.\beta \quad \in \mathbb{P}}{\alpha.\textbf{Says}A'\ B\ X.\beta.\textbf{Says}B\ S\ \textbf{Crypt}K_{BS}[B, N_B, X] \quad \in \mathbb{P}} \tag{3.9}$$

Here, we use X since B has no way of knowing the contents of the message since it does not have the key K_{AS}. In this case B can accept any message and just sends it inside the message that it is sending to the server.

In addition to protocol dependent rules, there are rules that apply to any protocol. The first rule is that the empty sequence is a valid trace. The other two rules concern the intruder. Assume *evs* is a valid trace of any protocol, then *evs* can be extended by any of the following events:

- $\textbf{Says}Spy\ A\ X$, where A is an agent and $X \in \textbf{synth}(\textbf{analz}\mathbb{H})$ as mentioned above.

- $\textbf{Says}A\ Spy\ [N_A, N_B, K]$, where A is an agent, N_A, N_B are nonces and K is a key.

The first rule indicates that the intruder can send any message to an agent, provided that it can create this message out of its initial knowledge and the messages that have been sent to the network. The second rule models the possible accidental loss of a key, where the nonces are used to indicate the session in which the key was lost. Given a certain protocol, rules that are protocol-dependent have to be formulated. Then, these rules, together with the general rule will define the set of the protocol traces, which can be used for analysis.

The main characteristic of the inductive approach is the ability to prove properties about security protocols without enumerating all possible traces. Therefore, the analysis is not limited to the finite case. However, coding a protocol as inference rules is not a straightforward task and requires experience. The same also applies to the formulation of security properties which are dependent on the particular protocol under analysis as can be seen from the work by Paulson [92]. This dependence of the properties on protocols calls for experience in using the software tools and reduces the code that can be reused from one protocol to another.

Process Algebra

In this approach, the behavior of each agent is described by process terms that can have constructs for both computation of messages and communication steps. A protocol session is expressed as several instances of processes running in parallel. Examples of research works using these concepts are the papers by Lowe [70] and Abadi and Gordon [5]. The intruder in some approaches is modeled by the process algebra notion of "environment" [5]. In other approaches, however, it is modeled as another process in the system [70]. In both cases, the model used is that of Dolev-Yao. As for security properties, secrecy of a message m, for instance, is expressed by the notion of process equivalence [5], so that m is secret if the processes $P(m)$ and $P(m')$ appear equivalent to their environments. This notion of secrecy will be discussed in more detail in Section 3.2.3 about protocol verification. In the following, we will discuss the Spi calculus [5] as a prominent one developed specifically for cryptographic protocols. It is worth mentioning that other process calculi, such as

CSP, which were developed for general concurrent processes, were used to analyze security protocols [70].

The Spi calculus [5] is based on π-calculus [82]. There are several reasons for this choice as mentioned in [5]: First, the π-calculus has an operator that allows the creation of new names (constants or channels) which can model the generation of keys or random values, second, this operation allows the "hiding" of the newly created name in a certain scope where only the processes in the scope can see the new name, third, the scope of the new name can be extended (scope extrusion) when the name is sent to a process outside the original scope. All of these features fit quite well for use with security protocols. However, since the π-calculus was originally designed for use with mobile concurrent processes it lacks expressions needed to specify security protocols, such as encrypted terms or the decryption process. Abadi and Gordon [5] extended the π-calculus with such operations to produce the Spi calculus, the syntax of which is given below:

$$\mathsf{Term} ::= \mathsf{Name} \mid \mathsf{Var} \mid 0 \mid \mathbf{suc}(\mathsf{Term}) \mid (\mathsf{Term}, \mathsf{Term}) \mid \{\mathsf{Term}\}_{\mathsf{Term}} \tag{3.10}$$

In the syntax above, a term can be a name (usually used for communication channels), a variable, the character 0 (zero), a successor of another term, a pair of terms, or a term encrypted by another term. It is clear that terms represent most of the data that a security protocol may deal with (e.g., there is no mention of hashing or signatures). We use L, M, N for terms, n for names, $x, y, \ldots$ for variables. Agents, on the other hand, are specified by processes, for which we use the letters P, Q, R.

Their syntax is presented below.

$P, Q, R ::=$	
$\overline{M}\langle N\rangle.P$	Outputs term N on channel M, then behave as P.
$M(x).P$	Receives an input on channel M, then behave as P, binds x in P.
$P\vert Q$	Parallel composition of P and Q.
$(\nu n)P$	Creates a new name n, binds n in P.
$!P$	Creates parallel replications of P.
$[M\ is\ N]P$	Behaves as P provided that M and N are the same, otherwise it is stuck.
$\mathbf{0}$	Nil process, does nothing.
$let(x, y) = M\ in\ P$	If $M = (L, N)$ it behaves as $P[L/x, N/y]$ (i.e., L and N replace x and y), otherwise it is stuck.
$case\ M\ of\ 0 : P\ \mathbf{suc}(x) : Q$	Behaves as P if M is 0, and as $Q[N/x]$ if M is $\mathbf{suc}(N)$, otherwise it is stuck
$case\ L\ of\ \{x\}_N\ in\ P$	Behaves as $P[M/x]$ if L is $\{M\}_N$, otherwise it is stuck.

(3.11)

Referring to the discussion in Section 2.1.3 about roles and agents, the Spi calculus provides the tools to specify roles and agents. This is by defining the notions of abstraction and concretion which are close to, but different than processes. An abstraction has the form $(x)P$, where P is a process and the abstraction binds x in P. Intuitively, it can be thought of as a process having the form $n(x).P$ where n is the name of a channel that we omit in the abstraction. A concretion has the form $(\nu m_1, \ldots, m_k)\langle M\rangle P$, it is as the process $(\nu m_1)\ldots(\nu m_k)\overline{n}\langle M\rangle P$, a shorthand for concretions is $(\nu\overrightarrow{m})\langle M\rangle P$. An interaction between an abstraction $F = (x)P$ and a concretion $C = (\nu\vec{m})\langle M\rangle Q$ is written as $F@C$ and defined to be:

$$F@C = (\nu\vec{m})(P[M/x] \mid Q) \qquad C@F = (\nu\vec{m})(Q \mid P[M/x]) \tag{3.12}$$

As an example, we consider the first step of the Needham-Schroeder protocol: $A \rightarrow B : \{A, N_A\}_{K_B}$. We specify the role of initiator in Spi notation to be $(i, j)P$, i.e., an abstraction where i and j are variables representing identities of agents playing the roles of initiator and responder, respectively. We note here that $(i, j)P$ is shorthand for (x) $let(i, j) = x$ in P. The process P is $(\nu N_A)\overline{c_{ij}}\langle\{(i, N_A)\}_{K_j}\rangle.Q$, where Q should specify the next steps that the initiator has to do. We notice here the creation of the nonce and the use of j to know the key K_j and the channel c_{ij}. Other roles can be specified similarly and protocol sessions are interactions between different instantiation of roles.

Although process calculi successfully model distributed systems, we find that their use with cryptographic protocols can complicate the analysis with concepts not directly related to security properties. An example of these concepts is their use of channels which complicates the treatment of the intruder. It is easier to consider a single channel under total control of the intruder.

Game-Theoretic Model

A Game theoretic model for security protocols was introduced by Kremer and Raskin [66, 67] where a protocol is regarded as a game between different agents that aim at achieving their own benefit even if it means using cheating, e.g., not following protocol rules. This setting is suitable for fair exchange protocols whose purpose is to guarantee fair treatment for all players who abide by the rules. Thus, a protocol is fair if it ensures that an agent playing with an honest strategy will eventually get their exchanged item.

The model used by Kremer and Raskin to analyze fair exchange protocols is based on the ideas of Alternating Transition Systems (ATS) [17]. An ATS adds the idea of agents to transition systems. A system of several agents makes a transition from one state to the other with the participation of all agents. Each agent possesses a strategy and it is the interaction of these strategies that determines the state of the system. Formally an ATS S is a tuple $\langle \Pi, \mathbb{A}, \mathbb{Q}, \pi, \delta \rangle$, where Π is a set of propositions, $\mathbb{A}$ is a set of agents, $\mathbb{Q}$ is a set of states, $\pi : \mathbb{Q} \rightarrow 2^{\Pi}$ is a function that maps a sate to the set of propositions that are true in this state, and finally $\delta : \mathbb{Q} \times \mathbb{A} \rightarrow 2^{2^{\mathbb{Q}}}$ is a transition function that assigns to each agent, at a certain system state, a set of choices. A choice is a set of states, which means that if the system S is in state q, and a is an agent, $\delta(q, a) = \{\mathbb{C}_1, \ldots, \mathbb{C}_n\}$, where each $\mathbb{C}_i$ is a set of states. Let λ be a string of states, i.e., $\lambda \in \mathbb{Q}^*$, we call λ an S-computation following the notation of transition systems. We denote by $\lambda[i]$, $\lambda[0, i]$, $\lambda[0, \infty]$ the i-th state of λ, the string from q_0 to q_i, and the infinite string $q_0 q_1 \ldots$ respectively. A strategy f_a of an agent a is a function that maps a non-empty system computation to a single choice. In other words for a computation $\lambda.q$ and agent a, we have $f_a(\lambda.q) \in \delta(q, a)$.

In order to see how an ATS S proceeds through transitions we assume an S-computation $\lambda.q$, i.e., the system is currently in state q. To determine the next state q' each agent a will have a set of choices, i.e., $\delta(q, a)$, and out of this set, a will pick up one choice $\mathbb{C}_a$ according to its strategy f_a. The next state q' is determined by $\{q'\} = \bigcap_{a \in A} \mathbb{C}_a$, or, in other form, $\{q'\} = \bigcap_{a \in A} f_a(\lambda.q)$. An essential assumption here is that $\bigcap_{a \in A} \mathbb{C}_a$ is a singleton set. As an example, we consider the famous prisoner's dilemma [90], where the two players of the game are two criminals A and B who committed a crime and got arrested by the police. During investigations, they are interrogated separately and each one of the them can either confess to the crime or deny doing it. Therefore, the system has four possible states $\mathbb{Q} = \{A-, -B, AB, --\}$, where $A-$ means that A confesses while B does not, $-B$ is the inverse situation, in AB both A and B confess and finally $--$ means neither of them confess. The set Π of propositions contains only two propositions:

`A confesses` and `B confesses` and we define a function π that maps states to propositions, e.g., $\pi(A-) =$ `A confesses`. We assume here that the interrogation is done in rounds where, in each round, criminals are allowed to change their plea according to the following rule: If a player confesses in one round they cannot change their situation, however if a player denied doing the crime, they can either continue denying or confess. In each round, the system will thus change state and we can define the transition function to be:

$$\begin{array}{ll} \delta(A-, A) = \{\{A-, AB\}\} & \delta(-B, A) = \{\{AB, A-\}, \{-B, --\}\} \\ \delta(AB, A) = \{\{AB, A-\}\} & \delta(--, A) = \{\{A-, AB\}, \{--, -B\}\} \\ \delta(A-, B) = \{\{AB, -B\}, \{A-, --\}\} & \delta(-B, B) = \{\{-B, AB\}\} \\ \delta(AB, B) = \{\{AB, -B\}\} & \delta(--, B) = \{\{-B, AB\}, \{--, A-\}\} \end{array} \tag{3.13}$$

To clarify the basic idea, we compare between $\delta(A-, A)$ and $\delta(A-, B)$ and recall that the two players are interrogated separately. In $\delta(A-, A)$, the system is seen through the eyes of A, in this case A knows that he confessed but he does not know about B so, *according to* A, the next state will be either AB, which means B confesses, or $A-$ in case B does not confess. We can now compare this to the situation of B in the same state $A-$. Using the same argument, B does not know about A, yet he knows he has one of two choices: Either to confess or to continue denying. In the first choice, the next sate, *according to* B, will either be AB or $-B$ and in the second choice it will be either $A-$ or $--$. An external observer knows however that the next state after $A-$ cannot be $-B$ or $--$ since A is not allowed to withdraw his confession. In other words, the *interaction* of all players in the game cannot move the system from $A-$ to $-B$ or $--$.

As argued by Kremer and Raskin [67], modeling a security protocol directly as an ATS would be unnatural and cumbersome. Instead, each agent is modeled as a reactive module like the ones used by the model checking tool Mocha [18], which has its own specification language based on Dijkstra's guarded command language [43]. A reactive module represents a system with constant interaction with its environment. It is comprised of a set of guarded commands that manipulate data. The syntax of guarded commands is specified by:

$$\begin{aligned} \langle guarded\_command \rangle &::= \langle guard \rangle \rightarrow \langle assignments \rangle \\ \langle assignments \rangle &::= \langle assign \rangle; (\langle assign \rangle;)^* \\ \langle assign \rangle &::= \langle var \rangle := \langle expression \rangle \end{aligned} \tag{3.14}$$

In the grammar above, a guard is a predicate that evaluates to `True` or `False` and an assignment list consists of at least one assignment where each assignment assigns the value of an expression to a variable, e.g., expressions can be the incremented value of another variable. Each agent (module) a has control over a set of variables $\mathbb{X}_a$ and we define $\mathbb{X}$ to be $\bigcup_{a \in \mathbb{A}} \mathbb{X}_a$. The interaction between agents (modules) takes place in successive rounds, where the value of variables x is updated to be $x' \in \mathbb{X}'$, x' is understood to be "the value of x in the new state". For all variables y whose values are not changed by any command, we have $y' = y$. Updates for variables are done according to the following rule: In a guarded command, if the guard evaluates to `True`, the assignments are carried out in succession and the values of variables are updated. An agent can update only those variables over which it has control. Expressions, however, can involve any variable that the agent can see.

In order to map an ATS $S = \langle \Pi, \mathbb{A}, \mathbb{Q}, \pi, \delta \rangle$ to a set of modules, we follow the following steps:

- Each agent $a \in \mathbb{A}$ is represented as a reactive module that has control over a set of variables $\mathbb{X}_a$.
- The set Π is a set of predicates over the variables in the set $\mathbb{X} = \bigcup_{a \in \mathbb{A}} \mathbb{X}_a$
- Each state $q \in \mathbb{Q}$ is a certain valuation of the variables in $\mathbb{X}$, i.e., a mapping $q : \mathbb{X} \rightarrow \mathbb{V}$ where $\mathbb{V}$ is a set of values (symbols or constants). Hence, the state is determined by the values of all variables, i.e., the state is q if and only if the

value of each variable x is equal to $q(x)$.

- As a convention, primed states q' are understood to be valuations of variables in the new state. If the old state is q, then q and q' agree on the value of all variables that were not updated, i.e., $\forall x \centerdot x = x' \Rightarrow q(x) = q'(x')$.

- The mapping π maps a state (valuation) to predicates in Π that are true in this state.

- Consider a guarded command ξ, with a guard and assignments, the two predicates $\mathbf{grd}_\xi(q)$ and $\mathbf{asn}_\xi(q, q')$ are defined. The guarded command ξ is said to be *enabled* if the guard $\mathbf{grd}_\xi(q)$ evaluates to `True`. In this case, the assignments can be executed, which will update the values of variables. The predicate $\mathbf{asn}_\xi(q, q')$ evaluates to `True`, in the guarded command ξ, if the value of each variable x' in the new state, i.e., after execution of ξ, is equal to $q'(x')$ and the old value for each variable x is equal to $q(x)$.

- The transition function $\delta : \mathbb{Q} \times \mathbb{A} \rightarrow 2^{2^{\mathbb{Q}}}$ maps a state and an agent into a set of sets of states such that:
$\delta(q_{old}, a) = \{\{q \in \mathbb{Q} \mid \forall x_a \centerdot q(x_a) = q'_{new}(x_a)\} \mid \xi \in \Phi_a \wedge \mathbf{grd}_\xi(q_{old}) \wedge \mathbf{asn}_\xi(q_{old}, q'_{new})\}$
where Φ_a is the set of guarded commands of a. Intuitively, the transition function means that by updating the value of some variables, the reactive module (player a) is actually choosing a set of states. These are all the states q such that the value of each variable x_a, after the execution of some enabled command ξ, is equal to $q(x_a)$. Therefore, changing the values of different variables chooses a different set of states.

From the previous discussion, we can see that each agent (a reactive module) contributes to the global state of the system by controlling a subset of variables, where a state is determined by the values of all variables. To illustrate the general idea, we assume an agent a has control over the variable x_a, a certain guarded command ξ_a in Φ_a may increment or decrement the value of x_a by one depending on some condition (guarded commands may contain conditionals). Now, let's suppose under some state q, we have $q(x_a) = 1$, then $\delta(q, a) = \{\mathbb{Q}_{inc}, \mathbb{Q}_{dec}\}$, where $\mathbb{Q}_{inc} = \{q_i \in \mathbb{Q} \mid q_i(x_a) = 2\}$ and $\mathbb{Q}_{dec} = \{q_i \in \mathbb{Q} \mid q_i(x_a) = 0\}$ the first set corresponds to a incrementing the value of x_a and the second corresponds to a decrementing x_a. The example can be easily generalized to the case where a controls more than one variable.

An agent A in a security protocol that encrypts a message m_A with key K_A to produce cipher text c_A can be modeled by:

$$A : m_A \wedge K_A \rightarrow c'_A := \texttt{True} \tag{3.15}$$

All variables in the description above were considered boolean since they indicate the knowledge of A, i.e., if A knew m_A and K_A it can know c_A. Each agent is described as a reactive module following the lines explained above and the protocol proceeds as an interaction between agents.

The previous model, based on game theory, was used for the analysis of fair exchange protocols. To the best of our knowledge, no attempt has been made to apply it in the verification of other protocols, e.g., authentication protocols. It seems this would not be an easy task since the model has no explicit mention of an intruder controlling the communication network. Instead, the only malicious behavior is that of the cheating agents.

Game Semantics and SPC

Loosely based on game theoretic concepts, the idea of using game semantics in security protocol analysis was investigated in the Security Protocol Calculus known as SPC [14]. The main characteristic of SPC is its emphasis on the interactions between communicating parties and the intruder as moves in a game. However, no verification methodology is developed that makes use of the calculus in order to express security properties and verify protocols. Moreover, no attempt is made to express algebraic properties of messages. More specifically, the features missing from SPC are:

- Expressing the notion of freshness in the syntax.
- Having a complete denotational semantics.
- Modeling computational steps in security protocols.
- Having a dedicated logic to specify properties of the model.
- Expressing branching-time properties.
- Having a software implementation or a model checking algorithm.

3.1.3 Logic-Based Models of Protocols

In Section 3.1.2, we presented several models of security protocols. They rely on describing protocol behavior as a distributed system using different formalisms, such as FSMs or process algebra. The described subsystems can then be executed and their interaction analyzed for certain properties. In this section, a somewhat more abstract view of protocols is introduced. This view is more concerned about what happens in protocol interactions than how it happens. More precisely, it is concerned about what logical properties are satisfied by protocol participants and how these properties change during protocol interactions. The advantage of this approach is to simplify the analysis by disregarding irrelevant details. However, a highly abstract view of protocols could lead to a coarse analysis that misses some security weaknesses.

Multiset Rewriting

Although multiset rewriting [30, 45, 84] deals with states and transitions, we chose to consider it as a logic-based approach and not a behavior description approach.

The reason is that a state is expressed by a set of logic formulas and a transition is a change in the formulas. A state of the system is expressed as a multiset of facts, where each fact is a formula of first order logic. A state transition, on the other hand is expressed by a rule of the form $l \longrightarrow r$, where both l and r are multisets of facts. The choice of first order logic has two main advantages: First, it can describe the structure of messages since they are expressed as terms over a signature that contains the necessary operation symbols such as pairing and encryption, second, the use of the existential quantifier can be used to model the generation of new values such as nonces. This becomes apparent if we take a look at the general form of rules:

$$F_1, \ldots, F_m \longrightarrow \exists x_1, \ldots, \exists x_n \,.\, G_1, \ldots, G_l \tag{3.16}$$

In the rule above, F_1 to F_m and G_1 to G_l are facts, where each fact has the form $P_i(t_1, \ldots t_k)$, i.e, a k-ary predicate symbol applied to k terms, which is sometimes written $P_i(\overrightarrow{t})$. A rule means that if a state S contains all the facts in the left hand side of the rule, the firing of a state transition takes place and the state changes to S'. In the new state S', all facts of the left hand side of the rule are destroyed and the facts of the right hand side are created with all variables x_1 to x_n substituted by "new" symbols. The "new" symbol condition is necessitated by proof rules of the existential quantifier, since in order to prove ψ from $\exists x \,.\, \varphi$ we need to derive ψ from $\varphi[y/x]$ under the condition that y does not appear free anywhere in the derivation, to avoid conflicts. The destruction and creation of facts is related to linear logic which introduced the idea of a logic that consumes its resources. In other words, an atomic formula A cannot be used twice in the same derivation and in case we need A more than once, then we must have multiple copies of A. This is the idea behind using multisets of facts instead of just sets. An example of a system is given below:

$$\begin{array}{lll} \text{Signature:} & \Sigma = \{a, b, c, f_1(.), f_2(., ., .)\} & \\ \text{Variables:} & \mathbb{X} = \{x, y, z, \dots\} & \\ \text{Predicates:} & \mathbb{P} = \{P_1(., .), P_2(.), P_3(., .)\} & \\ \text{Rules:} & \begin{array}{llll} R_1 : & & \longrightarrow & P_1(a, f_1(b)), P_2(f_2(a, b, c)) \\ R_2 : & P_1(a, x) & \longrightarrow & \exists y \,.\, P_3(x, y) \end{array} & \end{array} \tag{3.17}$$

The system specification above can be instantiated to a number of different traces that represent its transitions from one state to the other. In all traces, we start in a state $S_0 = \{P_1(a, f_1(b)), P_2(f_2(a, b, c))\}$ this is because the left hand side of R_1 is empty. Since S_0 contains $P_1(a, f_1(b))$, then the rule R_2 can be applied with $x = f_1(b)$, the application of the rule will result in the destruction of $P_1(a, f_1(b))$ and the creation of $P_3(x, y)$, the value of x is already determined to be $f_1(b)$ and the value of y can be substituted by any new term. The term we give for y should be noted so that we do not use it again for another variable that is bound by an existential qualifier. Different substitutions of the variable y will lead to different traces of the system. Let the variable y be substituted by c, the new state will then be $S_1 = \{P_2(f_2(a, b, c)), P_3(f_1(b), c)\}$. Let σ be a substitution that satisfies this condition of choosing new symbols, and $R = l \longrightarrow r$ be a rewrite rule, we denote by σr the multiset of facts obtained by applying the substitution σ to the right hand side predicates, the same applies to the left hand side. As an example in rule $R_2 = l_2 \longrightarrow r_2$ above, $\sigma = [x \mapsto f_1(b), y \mapsto c]$, and so $\sigma l_2 = \{P_1(a, f_1(b))\}$ and $\sigma r_2 = \{P_3(f_1(b), c)\}$. We say that a rule $R = l \longrightarrow r$ enables a rule $R' = l' \longrightarrow r'$ if there are substitutions σ, σ' such that $\sigma r \cap \sigma' l' \neq \emptyset$. A theory is a set of rules and a theory τ precedes a theory τ' if no rule in τ' enables a rule in τ.

With respect to security protocols, a protocol is modeled by a set of rewrite rules, i.e., a theory. In general, a protocol theory consists of an initialization theory and a set of principal theories. The initialization part takes care of the initial knowledge of protocol participants and the principal theories describe the change of states of the protocol due to communication steps. Moreover, the intruder is modeled by an intruder theory that is protocol-independent. In order for a theory to be a valid protocol theory, it must have some properties. This is the subject of the next discussion.

Let $R = l \rightarrow r$ be a rule in a theory τ, P a predicate and $P(\overrightarrow{t})$ a fact. We note here that a fact is determined by the predicate symbol *and* the arguments so $P(t_1, \ldots, t_n)$ and $P(s_1, \ldots, s_n)$ are the same fact if and only if $\forall i \,.\, t_i = s_i$. We say that R creates (consumes) P facts if some fact $P(t_1, \ldots, t_n)$ appears more (less) times in r than in l. The rule R preserves P facts if all $P(t_1, \ldots, t_n)$ occur the same number of times in l and in r. A predicate P is persistent in τ if no rule in τ consumes P facts. Principal theories are theories that describe a certain role of the protocol. They must have the following properties:

- They must have an ordered set of predicates, A_0 to A_k, where A_0 is called the initial roles state.
- Each rule $l \rightarrow r$ should contain exactly one A_i predicate in l and exactly one A_j in r where $i < j$.

A theory with the previous properties is called a well-founded principal theory and can describe the behavior of a certain protocol role going through the states A_0 to A_k. A theory $\tau \subset \tau'$ is a bounded subtheory of τ' if every rule R in τ that creates a fact F, then F either (1) contains existential quantifiers, or (2) is an initial role state in τ', or (3) is persistent in τ'. A theory τ' is a well-founded protocol theory if it is the disjoint union of well-founded principal theories and a bounded subtheory τ called the initialization theory. To sum up, a protocol theory consists of an initialization theory and principal theories. Properties of principal theories make sure that they describe progression through states of a protocol role, while properties of initialization theories make sure that they create data (this creation is expressed in the form of facts) that initializes the states for protocol roles.

A well-founded protocol theory is protocol-dependent, i.e., rules will depend on messages and role states. The intruder however is described by a set of rules called intruder theory, these rules are independent of any protocol and characterize a Dolev-Yao intruder. They contain three unary predicates, namely the decomposition, composition and memory predicates, which are denoted D, C, and M respectively. The decomposition predicate represents the intruder's capacity to decompose terms into smaller ones (smaller here means in terms of number of symbols). The composition predicate, on the other hand, represents the intruder's capacity to construct terms, while the memory predicate is for the storage of terms. Another predicate N is used where $N(x)$ means the message x is sent to the network and therefore can be known by the intruder. As an example consider the following intruder theory:

$$\begin{array}{ll}
N(x) \rightarrow M(x) & \\
D([x,y]) \rightarrow D(x), D(y) & \text{Unpairing of terms (the term } [x,y] \text{ is a pair)} \\
D(z) \rightarrow M(z) & \text{Storing messages} \\
D(x_K), M(K) \rightarrow D(x), M(K) & \text{Decryption with a known key.} \\
C(x), C(y) \rightarrow C([x,y]) & \text{Pairing of terms} \\
C(x), M(K) \rightarrow C(x_K), M(K) & \text{Encryption with a known key}
\end{array} \tag{3.18}$$

As an example, we model just the first step of the Needham-Schroeder protocol ($A \rightarrow B : \{A, N_A\}_{K_B}$), the full detailed specification is given in [30] with slightly different notations:

$$\begin{array}{lcl}
 & \longrightarrow & \exists K, \exists K^{-1} \centerdot Agent(K, K^{-1}), KeyPair(K, K^{-1}) \\
Agent(K, K^{-1}) & \longrightarrow & Agent(K, K^{-1}), A_0(K) \\
Agent(K, K^{-1}) & \longrightarrow & Agent(K, K^{-1}), B_0(K) \\
Agent(K, K^{-1}) & \longrightarrow & Agent(K, K^{-1}), MadePublic(K) \\
MadePublic(K_B), A_0(K_A) & \longrightarrow & \exists N_A \centerdot A_1(K_A, K_B, N_A), Send1(\{[K_A, N_A]\}_{K_B}), \\
 & & MadePublic(K_B)
\end{array} \tag{3.19}$$

The rules in (3.19) represent a part of the protocol theory of the Needham-Schroeder protocol, it includes an initialization theory (the first four rules) and one rule from the theory of role A, i.e., the principal theory for A. The initialization rules state that an agent is characterized by a *freshly* created pair of keys, and that the encryption key is made public. The last rule states that agent A creates a fresh nonce and pairs the nonce with its public key, which represents its identity. It then encrypts the pair with B's public key and sends the encrypted message to the network. In order to analyze a protocol, we combine its theory with the intruder theory and step through the transitions of the whole system.

Finally, we notice that message computation is modeled implicitly by the signature of the term algebra. Transmission of messages is implicitly modeled by the change of knowledge of each agent, i.e., knowledge increases with the reception of messages and facts express the local state as a function of knowledge. Intruder capabilities are modeled as predicates and transitions, predicates represent the intruder knowledge and transitions describe the change of this knowledge according to the intruder capabilities to analyze and synthesize messages.

Multiset rewriting has been used as the underlying model for some research efforts in security protocol verification [24, 31], and also for studying the complexity of the verification problem [45]. We believe it has the advantage of being abstract enough to formalize symbolic computations of security protocols in an intuitive manner. However, it does not in itself provide a verification methodology or a formulation of security properties that can be verified using the model. Moreover, writing a certain protocol's specification as a set of rewrite rules is not always an easy task.

Modal Logics

A modal logic, in the context of security protocols, consists of various statements about an agent's belief in certain messages or knowledge about these messages. The logic also specifies inference rules for deriving new beliefs from available beliefs and/or new knowledge from available knowledge. The most prominent amongst modal logics is the BAN logic, proposed by Burrows, Abadi, and Needham [29]. It was devised to specify and reason about authentication protocols. BAN deals with time by ensuring that beliefs held in the past will not extend to the present. In this regard, time is split into two epochs: Past and present, where present means the current run of the protocol. Abadi and Tuttle [7] expand the BAN logic by increasing its expressiveness and providing syntax and semantics rules. Moreover, Syverson [105] further extends BAN by adding temporal logic operators, which enabled the discovery of some attacks related to causal consistency. The GNY logic [55] developed by Gong, Needham, and Yahalom extends the BAN logic by adding more concepts like "possession" which allows a principal to send a message in which it does not believe but that it simply possesses. Another concept in GNY also is "recognizability" that allows principals to recognize the messages they expect. In the following, we provide a brief presentation of BAN.

The BAN logic deals with three sorts: Principals (A, B, S, etc.), keys (K_A, K_A^{-1}, etc.) and statements (N_A, N_B, etc.). Statements are also called formulas and are ranged over by the variables $X, Y, \ldots$, the variables $P, Q, \ldots$ range over principals and K ranges over keys. Any sentence in the logic will have one of the following forms:

$$\begin{array}{ll}
P\ \textbf{believes}\ X & P \text{ thinks } X \text{ is true.} \\
P\ \textbf{sees}\ X & P \text{ received } X \text{ and can store it.} \\
P\ \textbf{said}\ X & P \text{ once sent } X \\
P\ \textbf{controls}\ X & P \text{ is authorized to see or produce } X. \\
\textbf{fresh}(X) & X \text{ has not been sent before.} \\
P \overset{K}{\leftrightarrow} Q & P \text{ and } Q \text{ share } K. \\
\overset{K}{\mapsto} P & K \text{ is } P\text{'s public key.} \\
P \overset{X}{\rightleftharpoons} Q & X \text{ is a secret between } P \text{ and } Q \\
\{X\}_K & X \text{ is encrypted by } K \\
\langle X\rangle_Y & Y \text{ is } X\text{'s proof of origin}
\end{array} \tag{3.20}$$

A protocol is modeled as a set of logic sentences that follow the syntax above. These sentences are supposed to express what each agent thinks about other agents or messages. For instance, some of the beliefs of agents in the Needham-Schroeder protocol are expressed as:

$$\begin{array}{l}
A\ \textbf{believes}\ \textbf{fresh}(N_A) \\
B\ \textbf{believes}\ \textbf{fresh}(N_B) \\
A\ \textbf{believes}\ A \overset{N_A}{\rightleftharpoons} B
\end{array} \tag{3.21}$$

The full protocol description is given in [29]. Message construction and transmission are expressed implicitly as parts of the logic sentences. The intruder model is that of Dolev-Yao, and its capabilities can be expressed by inference rules such as:

$$\frac{P\ \textbf{believes}\ \overset{K}{\mapsto} Q \qquad P\ \textbf{sees}\ \{X\}_{K^{-1}}}{P\ \textbf{sees}\ X} \tag{3.22}$$

The rule above is meant to express decryption capabilities when the key is known. Given a protocol description as logical statements, and using the proof rules of BAN, verification is done by formulating security properties and proving them.

The main criticism to the BAN logic [27, 110] is that the translation from protocol specifications to logical statements is an error-prone process. This "idealization" of protocols may lead to proving the correctness of the ideal version when, in fact, the actual protocol is flawed.

3.2 Formal Verification of Security Protocols

In Section 3.1, we presented various models of security protocols. A model is usually developed in order to be able to rigorously study a protocol's characteristics and verify that it satisfies its intended properties. Several verification methodologies can exist that make use of the same model. In this section, we survey the most prominent methods used for formal verification of security properties. Intuitively, the main purpose of the verification is to ascertain that a protocol will actually achieve the objective for which it was designed, even in the existence of a malicious intruder trying to attack it. Hence, the verification process has two main purposes: To prove correctness of protocol designs and to unveil possible attacks. Any verification method aims to achieve at least one of these two goals. We begin by briefly presenting a categorization of security protocols according to their participants and design objectives, followed by a listing of the security properties that they should satisfy. We then give a classification of protocol attacks that result when a protocol fails to satisfy a required property. This is followed by an explanation of formal verification methods which comprise both model-based and proof-based methods.

3.2.1 Classification and Properties of Security Protocols

The properties that a protocol should satisfy depend on the objective that it aims to achieve. The objective is also related to protocol participants and may necessitate the presence of special participants such as a trusted server. In this section, we survey protocols according to their participants, objectives and the properties they should satisfy.

Participant-Based Classification

With respect to protocol participants, security protocols can be classified into [99, 79]: Arbitrated, adjudicated and self-enforcing protocols. Arbitrated protocols are the ones involving two or more protocol parties and another entity trusted by all parties, sometimes called Trusted Third Party (TTP). The TTP is assumed to be immune against intruder attacks and is involved on some protocol steps that require mutual trust between the TTP and each of the protocol participants. Protocols in which the TTP distributes cryptographic keys to participants are an example. In such a case, each participant should trust the TTP not to reveal their key to any other agent. A somehow similar category of protocols are adjudicated protocols where protocol execution has a normal path in which the TTP does not interfere and an exceptional path in which the help of a TTP is needed. An example is fair exchange protocols in which participants swap items. In a normal protocol run, no participant tries to cheat and everyone end up having the item they want. In case a participant does not follow protocol rules and tries for instance to receive the item it requested without sending an item in exchange, the interference of a TTP is needed which acts as an arbitrator between complaining participants. Adjudicated protocols represent a compromise when a TTP is needed since in the case of arbitrated protocols the TTP can become a bottleneck in the communication network. This is the case since a single TTP is usually involved in several protocol sessions and with different participants as it is costly to set up multiple TTPs that everyone on the network would trust. In order to avoid problems associated with TTPs completely, self-enforcing protocols are designed. They are protocols whose design enforces correct behavior on all participants. In other words, if some participants try to cheat, they will be detected and the protocol will stop. Of course it is not always possible to design self-enforcing protocols as in case of fair exchange protocols for instance.

Objective-Based Classification

Another categorization of security protocols can be done according to the objective they try to achieve such as: Secret messaging protocols, key exchange protocols, authentication protocols, fair exchange protocols, e-commerce protocols, zero-knowledge proofs protocols, and non-repudiation protocols. Of course, some protocols may achieve more than one of these objectives such as key exchange and authentication. In secret sharing, two or more participants share a secret message and the message cannot be known without participation of all parties or some of them. For instance, the secret can be an encrypted message and the plain text cannot be known unless both the cipher text and the key are known. An example is the Rivest-Shamir-Adleman three pass protocol [99]:

$$\begin{aligned}
&A \rightarrow B : \{m\}_{K_A} \\
&B \rightarrow A : \{\{m\}_{K_A}\}_{K_B} \\
&A \rightarrow B : \{m\}_{K_B}
\end{aligned} \tag{3.23}$$

In this protocol, A wants to send to B an encrypted message without having to share keys. The encryption system used must have the property that $\{\{m\}_{K_A}\}_{K_B} = \{\{m\}_{K_B}\}_{K_A}$ so that, in the second step, A can decrypt the message that it received from B and get $\{m\}_{K_B}$. Of course sending a secret message can be part of a larger protocol in which A and B begin by exchanging keys. An example of a key exchange protocol is the Diffie-Hellman key exchange [99]:

$$\begin{aligned}
&A \text{ and } B \text{ generate secret nonces } N_A \text{ and } N_B\text{, both are between 1 and } q \\
&A \rightarrow B : \alpha, q, \alpha^{N_A} \bmod q \qquad 1 < \alpha < q \\
&B \rightarrow A : \alpha^{N_B} \bmod q \\
&A \text{ computes key: } K = (\alpha^{N_B} \bmod q)^{N_A} = (\alpha^{N_B})^{N_A} \bmod q \\
&B \text{ computes key: } K' = (\alpha^{N_A} \bmod q)^{N_B} = (\alpha^{N_A})^{N_B} \bmod q = K
\end{aligned} \tag{3.24}$$

It is obvious that computations are done in a finite field, i.e., the Galois field $GF(q)$ (where q is a prime power), and the security of the computed key stems from the fact that it is difficult to compute logarithms in such fields.

Authentication protocols are meant to ascertain agents about the identities of other agents communicating with them. This is crucial in communication networks since the source address of a message cannot be relied on as an indication of the actual message source. An example of a famous authentication protocol is the public key Needham-Schroeder protocol presented in the standard notation in (2.1). A hierarchy of authentication definitions is developed by Lowe [71].

Fair exchange protocols are used when agents would like to exchange items they possess against items in the possession of other agents. The situation can arise, for instance, when two contracts are exchanged. The "fairness" condition here means that no agent should be put in a disadvantage with respect to other agents during any step in the protocol. In other words, an agent that follows the exact protocol rules should not be vulnerable to cheating from malicious agents. Such a condition is not satisfiable without the existence of a TTP [74, 114]. However, there exist designs of "optimistic" fair exchange protocols [21], which are adjudicated and hence aim to solve communication bottlenecks associated with arbitrated protocols. A practical and popular example of fair exchange protocols is e-commerce. An e-commerce protocol such as SET [73, 99] is aimed at securing credit card transactions via the Internet by providing both customers and merchants with a way to register with a "payment gate" such that the identity of a customer is verified before their credit card is charged.

Interactive proofs [54] are proofs in which a prover tries to convince a prover with the validity of some assertion. The proof proceeds as a series of challenges from the prover followed by responses from the prover. At the end, the verifier declares whether or not he is convinced. A question arises here about how much "knowledge" the prover must convey to the verifier in order to convince him. The minimum knowledge is whether the assertion is valid or not, i.e., one bit. Interactive proofs are called zero knowledge interactive proofs if the only knowledge that is conveyed to the verifier is this minimum knowledge. Interactive proofs of knowledge [48, 16], on the other hand, are interactive proofs about the "knowledge" of the verifier. In other words, the verifier does not aim to prove the validity of the assertion, i.e., whether the assertion is true or not, but to prove to the verifier that he knows whether the assertion is valid or not. In this case, the knowledge that is conveyed to the verifier has a minimum of zero bits since the verifier is not even told whether the assertion is true or not. Zero knowledge interactive proofs of knowledge are the ones that achieve this minimum of conveyed knowledge, i.e., zero bits. Zero knowledge interactive proofs of identity [48] are a special case in which the prover is an agent trying to authenticate itself to a server (verifier) by proving to the server that it knows a password (identity) without actually revealing the password. The Feige-Fiat-Shamir protocol [48] uses this idea to authenticate agents to each other.

Non-repudiation protocols are used in cases such as certified e-mail [99]. The objective of these protocols is that the sender of a message cannot deny sending it (non-repudiation of origin) and that the receiver cannot deny receiving it (non-repudiation of receipt).

Security Properties

A security protocol satisfies a security property if the protocol is successful in achieving its design goal. Security properties are therefore related to design objectives and are listed below:

Secrecy: A message m is secret in a protocol, if the intruder is not able to deduce m from its knowledge. The previous definition implicitly assumes that the intruder is either capable of knowing m or cannot know any information about m. Therefore a more strict definition of this property is that the intruder should not be able to distinguish between a protocol run (session) containing m and another run containing a different message m'

Authentication: Informally speaking, authentication of agent A to agent B means that agent B is certain that A is not lying about its identity. It is not straightforward to formalize a definition of authentication and in [71], several definitions are given that vary in their strictness. The definition of "agreement" is that two agents are authenticated to each other if, at the end of a protocol run, they both agree on some facts. The simplest fact is that they were both running the protocol one agent as an initiator and the other as a responder, this is called weak agreement. In addition to this fact, full agreement requires that both agents A and B agree on the value of a set of data and each run of A corresponds to a unique run of B.

Integrity: In the case of message secrecy, an intruder may not be able to gain any knowledge about a message m, however it can corrupt some protocol messages in a way that will prevent an honest agent from receiving the correct value of m. Message integrity means that the intruder is not even capable of corrupting the message. If it were corrupted, then an honest agent should be able to know this.

Non-repudiation: As mentioned earlier, non-repudiation means the incapability of an agent to deny sending or receiving a message.

Fairness: This property is related to fair exchange protocols and means that no agent is put in disadvantage with respect to other agents when exchanging items. In [67], this property is formalized in a game-theoretic setting since adversarial behavior between agents is assumed. Fairness is then defined as the existence of a strategy for each agent A that guarantees the possibility of safe termination for A *regardless* of the strategies of other agents. Safe termination here means that A will either possess its own item or the item it wanted from the exchange.

Anonymity: Anonymity is a property allowing to hide the identity of some agents while allowing them to send certain messages. A good example is a voting protocol, where it is necessary that agents are allowed to vote without knowing the actual vote of a certain agent.

3.2.2 Attacks on Security Protocols

In the previous section, we provided various examples of security protocols design objectives. An attack on a protocol executed by a malicious intruder aims at making the protocol fail to achieve its objective(s). For instance, an attack on a protocol that is designed to convey a secret message is successful if the intruder is able to know some information about the message or the whole message. With respect to intruder actions, attacks can be active or passive. Active attacks (man-in-the-middle attacks) are the ones where the intruder interrupts the flow of information in the network by intercepting, modifying, sending or receiving messages. In passive attacks (eavesdropping) the intruder just monitors the flow of messages in the network without trying to intervene in the protocol steps. In such case, the intruder tries to deduce knowledge to which it is not entitled from the collected messages.

Attacks on security protocols may be in one of the following categories depending on the way they are executed:

Guessing attacks: These are attacks where the intruder is able to guess the value of supposedly randomly created information such as nonces and some keys. Guessing attacks are usually based on exploiting weaknesses of pseudorandom number generators. Moreover, the intruder maybe able to monitor several protocol runs and compare them for similarities.

Replay attacks: Replay attacks are common in the literature and in [106] a taxonomy is given for them. The intruder records messages and replays them later with or without modification to some message components. In order to counter such attacks, nonces are used to guarantee freshness of messages. This method is not always successful as in the case of the attack [70] on the Needham-Schroeder (NS) protocol shown below. It is important to mention here that the replay of messages can take place in the same protocol run during which the messages were recorded or in different protocol runs where the runs can happen concurrently (in an interleaved manner) or not. The following attack on the NS protocol takes place in two interleaved runs α and β:

$$\begin{array}{lll} \alpha.1 & & A \rightarrow I : \{N_A, A\}_{K_I} \\ & \beta.1 & I(A) \rightarrow B : \{N_A, A\}_{K_B} \\ & \beta.2 & B \rightarrow I(A) : \{N_A, N_B\}_{K_A} \\ \alpha.2 & & I \rightarrow A : \{N_A, N_B\}_{K_A} \\ \alpha.3 & & A \rightarrow I : \{N_B\}_{K_B} \\ & \beta.3 & I(A) \rightarrow B : \{N_B\}_{K_B} \end{array} \tag{3.25}$$

We notice that by the end of the β run, I impersonating A was able to authenticate itself to B as A. It did this by replaying messages between A and B.

Type flaw attacks: Type flaw attacks are usually due to implementation where an agent can accept a value of a certain type as being of another type. For instance an agent can accept a key as a nonce or vice versa since both are numbers. At the bit level all messages are similar, i.e., a string of bits, and the worst type flaw attacks are where an agent can accept any message instead of any other. Another situation is where an agent can distinguish compound messages (pairs and encrypted messages) from basic messages (agent names, nonces, keys, etc.) but can confuse basic messages. In all cases, type flaw attacks are avoidable with careful protocol implementations.

Dictionary attacks: These attacks are brute force attacks in which the intruder tries every value of a secret until it gets a successful protocol run. Attacks like this are known against passwords or keys and can be made increasingly difficult by using longer passwords or keys.

Multi-protocol attacks: Two or more protocols may have similar structure of messages and an intruder monitoring protocol runs of these protocols may be able to replay messages between them. In [40], a survey is given about a number of protocols and the possible attacks that can be mounted against them.

Cryptographic system attacks: These are attacks that stem from vulnerabilities in the cryptographic systems used by the protocols and are studied in the field of cryptanalysis. It is possible that a protocol, that is secure under the perfect cryptography assumption, be vulnerable to a number of attacks once the properties of its cryptographic operations are taken into consideration.

3.2.3 Model-Based Analysis

Cryptographic protocols are distributed systems with the added characteristic that they use cryptographic operations and that an intruder is present. Therefore, general methods and software tools of formal verification can be used to analyze them, provided that careful thought is given to the modeling of their special characteristics. The most prominent automatic formal verification technique for distributed systems is model checking [75]. It is basically an exhaustive search of the state space of a system in order to make sure that a certain property is satisfied. Of course, when the state space is very large or even infinite, efficient search algorithms have to be developed. Some of these algorithms may even be semi-decidable, this will be discussed further in the following.

Several general-purpose verification tools use model checking and are based on various models. The models include, for instance, Petri nets, finite state machines, and the calculus of Communicating Sequential Processes (CSP). The verification process is automated or semi-automated using various tools such as Ina test [61], Murφ [85], Failure Divergences Refinement (FDR) checker [70], etc. Moreover, several other verification techniques are specifically developed for security protocols. Their advantage is that they provide an "easier" and more natural way of specifying protocols and verifying them since, for instance, a model of the intruder is already defined. In the following, we present both the general-purpose and the security-specific methods based on automata analysis, process algebra, and game theory.

Moreover, we explain methods used to deal with infinite state spaces of models.

General-Purpose Automata Analysis

Kemmerer [61] was one of the first to use tools developed for the verification of FSMs in order to verify security protocols. The InaJo language and Inatest tool were used to model and analyze protocols. The protocol is modeled as a FSM where states are sets of state variables and state constants. State constants do not change from one state to the other, whereas state variables do. A transition from one state to the other represents some protocol operation that manipulates variables. The intruder knowledge is modeled as a state variable that tabulates all the information the intruder has access to. State invariants are expressed in the form of criteria which are first order logic formulas. A reachability analysis determines if all invariants are satisfied. Another attempt was done in [85], where the general purpose model checker Murφ was used to analyze some security protocols. We already described the modeling process in Section 3.1.2. Invariants can then be expressed and, when running the tool, a reachability analysis is done.

In the Multiset rewriting model described in 3.1.3, a trace is a sequence of logic formulas where existentially quantified variables are replaced by new values. The general-purpose LLF tool of linear logic was used for verification [30]. The translation of transition rules from first order formulas to formulas of linear logic is straightforward and shown in [30]. The verification algorithm searches through all possible traces to check if a formula is true. Formulas can be written to represent security flaws, examples are given in [30].

Security-Specific Automata Analysis

In this section, we present some state-analysis software tools that were developed specifically for the verification of security protocols. Some of the earliest tools are the NRL protocol analyzer [76] and the Interrogator [81]. NRL is written in Prolog for the analysis of authentication and key distribution protocols. It uses a Dolev-Yao model of the intruder and the protocol is specified as a set of FSMs, each representing a communicating agent. Each FSM has a set of local variables and can receive input. When a condition defined over the input and the local state variables is satisfied, a transition fires which produces an output and changes the state. The input and output are words over an alphabet and the global state of the protocol is determined by the words known by the intruder and the values of all local state variables. The intruder states and transitions are not specified explicitly but are constructed from the description of intruder abilities, and hence it is possible to query the analyzer about the possibility that the intruder performs a certain transition. Moreover, algebraic properties of messages (exchanged words) can be expressed like for instance the fact that encryption followed by a decryption with the same key will cancel out. The intruder, trying to know specific words, is actually trying to solve a word problem in a term rewrite system. Verification is done by backward reachability analysis starting from an unsafe state described as a set of words known by the intruder and values for local state variables.

The Interrogator is also written in Prolog and described in [81]. A protocol is a set of FSMs where each machine P represents a protocol role and exchanges messages with other machines. A message $m \in X^*$ is a sequence of elements from the set X of symbols and given a message m, $|m|$ denotes the components of m, i.e., $|m| \subseteq X$. The state q of each FSM is a set of messages $q \subseteq X^*$ and the transition relation of a machine P is defined to be $t_P : \mathbb{Q} \times X^* \rightarrow \mathbb{Q} \times X^*$, where $\mathbb{Q}$ is the set of states. The global state of the system is a triple (N, m, K), where N is a function such that $N(p)$ is the local state of machine P, m is a transient message (in the network) and K is a set of messages that represents the intruder knowledge. A network transition $(N, m, K) \rightarrow (N', m', K')$ can result from a transition done by a FSM or from an operation done by the intruder. Verification is done by defining a predicate that is true in an unsafe situation, for instance that a secret message

m is in K, and then querying the tool about the truth value of such a predicate. The tools then performs a backward reachability analysis in order to provide the answer. It is worth noting that a study [62] was performed to compare NRL and the Interrogator along with the general-purpose tool Inatest.

In addition to using Prolog, there exist several formal specification languages dedicated to security protocols, one of the first languages was the Interface Specification Language (ISL) developed by Brackin [28]. ISL specifies the protocol by listing a set of definitions, initial conditions, communication steps (of the protocol), and finally the goals that are required to be reached by the execution of the protocol. Definitions contain for instance the identifiers of the communicating parties and the exchanged keys, while initial conditions state the initial knowledge (e.g., encryption keys) and beliefs of each principal in the protocol. The protocol part of the description just lists the communication steps in standard notation, while the "goals" part contains the final result required when the protocol is executed (e.g., which principal gets which key). The language is simple, intuitive and represents the steps of evolution of the knowledge and beliefs of each principal in the protocol.

Another specification language is the Common Authentication Protocol Specification Language (CAPSL) developed by Millen [80] and based on ISL. This language was further expanded (MuCAPSL) to be able to specify group multicast protocols.

The CASPER language [72] was developed in order to transform a protocol description into the corresponding specification in the calculus of Communicating Sequential Processes (CSP). This is done for the purpose of using the Failure Divergences Refinement (FDR) checker, which is a general purpose tool that can be used to determine whether an implementation refines a specification.

In the course of the project for the Automated Validation of Internet Security Protocols and Applications (AVISPA) [96], the High Level Protocol Specification

Language (HLPSL) was created to describe security protocols and formalize security properties. The AVISPA software tool then translates this language into an intermediate format based on the model of multiset rewriting [30]. The new format serves as an input for several model checkers which analyze the protocol [108]. Since AVISPA is based on multiset rewriting, it suffers from the same drawbacks, namely the lack of a formally defined language for the specification of security properties. Only some statements of security objectives can be written and, at the time of writing of this thesis, they are limited to secrecy and authentication [108].

Process Algebra

As previously presented in Section 3.1.2, the process algebra approach models the protocol as a number of processes running in parallel in a hostile environment. We presented the SPI calculus which, for verification, relies on the notion of process equivalence. More specifically, a message m is kept secret if the processes $F(m)$ and $F(m')$ appear indistinguishable to any other process they run in parallel with. In [70], verification of security protocols was also done using CSP and the Failure Divergences Refinement (FDR) tool. In this case the notion of refinement was used instead of equivalence. The security property was formulated as a specification process and verification in this case consisted of checking whether the protocol process refined the specification process.

Game-Theoretic Verification

The game-theoretic model, used for the verification of fair exchange protocols was presented in Section 3.1.2 and the Alternating Transition System (ATS) explained. Verification in this case is done by model checking an ATS representing the protocol against formulas of the Alternating-time Temporal Logic (ATL). ATL formulas φ are interpreted over an ATS and have the following syntax, where p is a proposition and A is a set of agent names:

$$\varphi ::= p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \langle\langle A\rangle\rangle \bigcirc \varphi \mid \langle\langle A\rangle\rangle \varphi_1 \mathcal{U} \varphi_2 \tag{3.26}$$

In order to interpret a formula of ATL we assume a set $\mathbb{A}$ of agents, where $A \subseteq \mathbb{A}$ and a function Π, where $\Pi(q)$ is the set of propositions that are true in q. A state q in an ATS satisfies a formula φ of ATL, written $q \models \varphi$, in the following cases:

- $q \models p$ iff $p \in \Pi(q)$
- $q \models \neg\varphi$ iff $\neg(q \models \varphi)$
- $q \models \varphi_1 \vee \varphi_2$ iff $q \models \varphi_1$ or $q \models \varphi_2$
- $q \models \langle\langle A\rangle\rangle \bigcirc \varphi$ iff there exists a set of strategies, one strategy f_a per player $a \in A$, such that if each player a followed f_a, then (for all strategies of other palyers that are not in A) in all computations out of q, the next state q' will be such that $q' \models \varphi$.
- $q \models \langle\langle A\rangle\rangle \varphi_1 \mathcal{U} \varphi_2$ iff there exists a set of strategies, one strategy f_a per player $a \in A$, such that if each player a followed f_a, then for all computations out of q, after i transitions, we will encounter a state q' such that $q' \models \varphi_2$ and for all states q'' between q and q' (including q) we have $q'' \models \varphi_1$.

We notice that ATL is a generalization of CTL by adding the concept of agents. The idea is that in order to get the system to do a certain computation, the interaction between agents is necessary. Fairness for a single agent a can thus be formulated as the existence of a strategy for a that can force the computation of the system in a direction that will guarantee that a gets what it wants, regardless of the strategies of other agents.

Infinite State Spaces

As mentioned earlier, in model-based approaches, agents are modeled as transition systems that communicate through a network totally controlled by the intruder. Each agent at a certain state, upon receiving a message, will change its state and send a message to the network. All messages in the network are added to the intruder's knowledge. In this setting, sources of infiniteness can be [58]:

- Some agents have an infinite number of states.
- The number of agents is infinite.
- The intruder, which is also the network, can send an infinite number of messages to an agent.

The first two conditions are not usually considered since they do not apply to most situations [58]. The third condition however is dealt with using two main approaches [34]: Limiting the set of messages the intruder generates, and using abstractions to represent (possibly infinite) sets of messages that the intruder can generate. The first approach limits the intruder deductive capabilities [34] or limits the messages that it produces to only those that can pass checks done by the agent receiving the message [85]. The second approach involves several techniques, for instance, one technique [58] replaces parts of the messages produced by the intruder by variables and then solves a set of constraints to find values for these variables whenever a value is needed. The set of constraints represents the knowledge of the intruder at the time it generated the message. This approach is also used in the AVISPA tool [24]. Another technique [86] uses an abstract interpretation [39] framework by representing the set of messages that the intruder can generate as a tree automaton. An abstract state space is built and analyzed for security properties. This model was further expanded [56] to include the analysis of multi-session protocol agents. A survey is presented in [36] about approaches to deal with infinite state spaces.

3.2.4 Proof-Based Analysis

As opposed to model-based analysis, proof based analysis formalizes protocol descriptions in a manner suitable for the application of proof methods. Proof can be done for instance by induction or by using the proof system of a certain logic. This method of analysis has the advantage that verification can be potentially done over infinite domains since we do not need to enumerate states as in the case of model-based analysis. The drawback, however, is the lack of fully automatic tools that can carry out the proofs autonomously. In the following, we briefly present the main ideas behind proofs carried out using three approaches that we already introduced, namely, strand spaces, modal logics, and the inductive approach.

Strand Spaces

Properties are formulated as theorems that have to be proved using the properties of strands and bundles. For instance in order to specify that a certain message m is secret in a bundle $\mathcal{C}$ we have to prove that for all nodes n in the bundle, m is not a clear term of n, i.e., we have to prove the theorem $\forall n \in \mathcal{C} \,.\, term(n) \neq m$. Steps of the proof vary according to protocol since each protocol will have its own strands.

Modal Logics

The general idea here is that the protocol is described by a set of formulas, and the logic itself has a proof system. Verification amounts to the following: Given the protocol formulas as premises and using the proof system of the logic, can we prove a certain formula φ that represents a desired property of the protocol? In other words, verification means proving that starting from an initial set of beliefs, communicating parties will reach a certain belief (the logical description of the desired property). As an example, consider the following inference rule, where meanings of terms were explained in Section 3.1.3:

$$\frac{P \textbf{ believes } P \overset{K}{\leftrightarrow} Q \qquad P \textbf{ sees } \{X\}_K}{P \textbf{ believes } Q \textbf{ said } X} \tag{3.27}$$

The rule above simply states that if P shares a secret key with Q and then sees a message encrypted by this key, he can assume that the sender of the message was Q. A number of rules are given in [29], and they are used to prove conclusions out of premises that are protocol dependent.

Inductive Approach

In this approach, the set of traces of a protocol was defined inductively. Therefore, properties of traces can be proved by induction. To prove a property $P(evs)$ over the trace evs, we have to prove that:

- $P([\,])$ holds, where $[\,]$ is the empty trace.
- For all rules of the protocol that extend the trace evs with an event ev if $P(evs)$ holds then $P(evs.ev)$ holds.

The first item above is the base case and the second is the induction step. Since it is possible to extend a protocol trace using any one of a number of rules as indicated in Section 3.1.2, so in order to prove a property P, it is necessary to prove that all rules preserve the property P.

Chapter 4

Protocol Messages

Messages are the data structures exchanged between communicating agents during protocol runs. A protocol specification defines what messages should be exchanged, in what order, and between which agents. In order to be able to analyze security protocols, a computation model for messages has to be provided. In this chapter, we present our model of protocol messages which treats messages symbolically. The model deals with the following issues:

- The representation of messages such as keys, nonces and encrypted messages.
- The representation of computations on messages such as the encryption or decryption operations.

The model follows an algebraic framework, in other words, messages are not considered as numbers or sequences of bits but as symbols manipulated algebraically. We thus provide a general approach that distinguishes between messages syntax and semantics. Syntactically, messages are presented as terms of a term algebra, where atomic messages are the constants, and composed messages are formed from constants and variables using function symbols. Semantically, using concepts of game semantics, the generation of a message by an agent is an interaction between the agent and its environment in which the agent applies an algorithm to a set of messages it already knows in order to obtain the message requested by the environment. Such an approach has its roots in the Dolev-Yao model for the analysis of security protocols [44]. However, we make use of equational theories to extend the model capabilities by generating congruences over terms. In this case, we deal with classes of terms which are computationally equivalent yet syntactically different. We also present a model of abstract computation [109] in order to be able to define algorithms over messages. These algorithms are written in a language that makes use of the operator symbols of the algebra to represent computational steps.

We begin by introducing our message syntax, message algebra and abstract computation procedures. Then we present our treatment of the addition of equational theories to the algebra. Finally we give a game semantics interpretation to message terms.

4.1 Message Syntax

In the standard notation of security protocols, messages are written in a way to specify the structure of each message. For instance, a message could be atomic such as an agent's name or a nonce, or it can be composed from other messages using concatenation or encryption. Moreover, the notation also specifies which messages, or parts of messages, are encrypted and with which keys. We give below the BNF grammar that is used when specifying messages:

$$\begin{array}{rcl} m & ::= & a \\ & | & c \\ & | & n \\ & | & k \\ & | & m, m' \\ & | & \{m\}_k \\ & | & m\ op\ m' \end{array}$$

Where $op \in \{+, -, *, /\}$

In the grammar above, a ranges over agent names, similarly c represents constant (text) messages, n represents natural numbers or nonces, and k represents cryptographic keys. The term m, m' represents concatenated messages. In some cases, however, concatenation is written $m.m'$ when it might be confused with elements in a set. The term $\{m\}_k$ denotes encrypted messages, where k is the encryption key. Finally, the last expression above, i.e., $m\ op\ m'$, denotes arithmetic operations. We treat messages whose syntax follows the grammar above as a subset $\mathbb{M}$ of the set of terms of a term algebra [112]. We first need some definitions, which we list in the next subsection.

4.1.1 Definitions

We use the usual notation for sequences over sets such that, for a set $\mathbb{S}$, we have $\mathbb{S}^* = \{\epsilon\} \cup \{S_1.S_2 \ldots S_l \mid l \in \mathbb{N} \setminus \{0\} \wedge \forall i \in \{1, \ldots, l\} \,.\, S_i \in \mathbb{S}\}$, where $\mathbb{N}$ is the set of natural numbers. The set $\mathbb{S}^*$ is the set of all sequences of elements of $\mathbb{S}$ that have finite length l, including the empty sequence ϵ of length 0. For any sequence $w \in \mathbb{S}^*$, $\mathbf{len}(w)$ is a natural number that is the length of w. Also, given a set $\mathbb{I}$, a family of sets indexed by the elements of $\mathbb{I}$ is a collection of sets, one for each element of $\mathbb{I}$, it is written as $\{\Lambda_i \mid i \in \mathbb{I}\}$, each set Λ_i is called a member of the family.

A multi-sorted signature Σ [51] is a pair $\langle \mathbb{S}, \mathbb{F} \rangle$, where $\mathbb{S}$ is a set of sorts and $\mathbb{F} = \{\mathbb{F}_{w,S} \mid w \in \mathbb{S}^* \wedge S \in \mathbb{S}\}$ is a family whose members $\mathbb{F}_{w,S}$ are sets that are indexed by elements of $\mathbb{S}^* \times \mathbb{S}$ and that contain operator symbols. For an operator symbol $\mathtt{f} \in \mathbb{F}_{w,S}$, the sequence w is called the arity of $\mathtt{f}$ while S is called the coarity of $\mathtt{f}$, the rank of $\mathtt{f}$ is the pair $\langle w, S\rangle$. Operator symbols having rank $\langle \epsilon, S\rangle$ are called constants of sort S. Moreover, we define the family $\{\mathbb{X}_S \mid S \in \mathbb{S}\}$, where an element $x \in \mathbb{X}_S$ is called a variable of sort S and the set $\mathbb{X} = \bigcup_{S\in\mathbb{S}} \mathbb{X}_S$ is the set of all variables. Variables differ from constants of the same sort by the fact that they can be mapped by substitutions as will be defined below. We can now inductively define the set of well-sorted terms of sort S, which contains:

- All variables of sort S.
- All constants of sort S.
- All strings of the form "$\mathtt{f(}t_1\mathtt{,}\dots\mathtt{,}t_n\mathtt{)}$", for each operator symbol $\mathtt{f} \in \mathbb{F}_{w,S}$ whose arity w is $S_1 \dots S_n$, coarity is S, and $t_1, \dots, t_n$ are terms of sorts $S_1, \dots, S_n$ respectively.

The set $\mathbb{T}_{\Sigma,S}(\mathbb{X})$ of terms of sort S over the signature Σ is the smallest set that contains all the elements listed above. The set $\mathbb{T}_\Sigma(\mathbb{X})$ of terms over the signature Σ is just the union of all sets of terms of different sorts, i.e., $\mathbb{T}_\Sigma(\mathbb{X}) = \bigcup_{S\in\mathbb{S}} \mathbb{T}_{\Sigma,S}(\mathbb{X})$. Terms that do not contain any variables are called ground terms. The set of all ground terms of sort S is denoted by $\mathbb{T}_{\Sigma,S}$, and the set of all gound terms over Σ is denoted by $\mathbb{T}_\Sigma$. It is important to note that terms are just syntactic entities, i.e., words, formed from operator symbols, variables, the symbol "`(`", the symbol "`)`" and the symbol "`,`" we do not attach any meaning to them. The double quotes and the use of typewriter font in "$\mathtt{f(}t_1\mathtt{,}\dots\mathtt{,}t_n\mathtt{)}$" are used to emphasize this fact.

An Algebra $\mathcal{A}$ with signature $\Sigma = \langle \mathbb{S}, \mathbb{F}\rangle$, called a Σ-Algebra, is a pair $\langle \mathbb{A}, \mathbb{F}^{\mathcal{A}}\rangle$, such that:

- $\mathbb{A} = \{\mathbb{A}_S \mid S \in \mathbb{S}\}$ is a family of sets where, for each sort $S \in \mathbb{S}$, the set $\mathbb{A}_S$ is called the carrier of sort S and for each constant of sort S, there exists a corresponding element $a \in \mathbb{A}_S$.

- $\mathbb{F}^{\mathcal{A}} = \{\mathbb{F}^{\mathcal{A}}_{d,c}\}$ is a family of sets of functions such that for each operator symbol `f` $\in \mathbb{F}_{w,S}$, with arity $w = S_1 \ldots S_n$ and coarity S, there exists a function $f^{\mathcal{A}} \in \mathbb{F}^{\mathcal{A}}_{d,c}$, with domain $d = \mathbb{A}_{S_1} \times \ldots \times \mathbb{A}_{S_n}$ and codomain $c = \mathbb{A}_S$.

From the definition of the Σ-algebra above, we note that the algebra assigns, to each sort, a set called the carrier of the sort, to each constant an element of the carrier of its sort, and to each operator symbol a function (sometimes also called operator) whose domain and codomain contain carrier sets of the arity and coarity of the operator symbol, respectively. In algebraic denotational semantics [51], a Σ-algebra $\mathcal{A}$ with a family of carriers $\mathbb{A}_S$, together with a family of substitutions $\theta_S : \mathbb{X}_S \rightarrow \mathbb{A}_S$ are used to assign semantics to terms over a certain signature Σ, i.e., terms $t \in \mathbb{T}_\Sigma(\mathbb{X})$. The interpretation of a term t of sort S is an element of the carrier set $\mathbb{A}_S$ of the Σ-algebra. If the term t is a variable x, then the interpretation is $\theta_S(x)$, if it is a constant, then the interpretation is the element in $\mathbb{A}_S$ assigned to the constant by the algebra $\mathcal{A}$. If the term is of the form `f(`t_1`, ... ,`t_n`)`, the interpretation is the element $f^{\mathcal{A}}(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n))$, where for each i, $\mathcal{I}(t_i)$ is the interpretation of the term t_i. A special kind of Σ-algebras is the Σ-term algebra $\mathcal{T}$, in which:

- The family $\mathbb{T}$ of carrier sets is $\{\mathbb{T}_{\Sigma,S}(\mathbb{X})\}$, i.e., the carrier set of sort S is the set of terms of sort S. We omit the subscript Σ when the signature is obvious from the context.

- For each function $f^{\mathcal{T}} : \mathbb{T}_{S_1}(\mathbb{X}) \times \ldots \times \mathbb{T}_{S_n}(\mathbb{X}) \rightarrow \mathbb{T}_S(\mathbb{X})$ in $\mathbb{F}^{\mathcal{T}}_{d,c}$ and terms $t_1 \in \mathbb{T}_{S_1}(\mathbb{X}), \ldots, t_n \in \mathbb{T}_{S_n}(\mathbb{X})$, we have $f^{\mathcal{T}}(t_1, \ldots, t_n) =$ `"f(`t_1`, ... ,`t_n`)"`, i.e., functions just map a tuple of terms to a term, where $f^{\mathcal{T}}$ is the function that corresponds to the operator symbol `f`.

In other words, a Σ-term algebra is an algebra that interprets terms by themselves. A Σ-homomorphism is a function between Σ-Algebras that preserves their structure. If $\mathcal{A}$ and $\mathcal{B}$ are two Σ-Algebras, i.e., having the same signature Σ,

$h : \mathcal{A} \rightarrow \mathcal{B}$ is a called a Σ-homomorphism iff:

$$\forall \mathtt{f} \in \mathbb{F} \centerdot h(f^{\mathcal{A}}(a_1, \ldots, a_n)) = f^{\mathcal{B}}(h(a_1), \ldots, h(a_n))$$

A substitution $\theta : \mathbb{X} \longrightarrow \mathbb{T}_\Sigma(\mathbb{X})$ is a mapping from variables to terms of the same sort. A ground substitution maps variables to ground terms. A substitution is generally extended to a homomorphism in the following way:

$$\begin{aligned}
&\theta : \mathbb{X} \rightarrow \mathbb{T}_\Sigma(\mathbb{X}) \text{ is extended to} \\
&\tilde{\theta} : \mathbb{T}_\Sigma(\mathbb{X}) \rightarrow \mathbb{T}_\Sigma(\mathbb{X}) \\
&\tilde{\theta}(c) = c \quad \text{for all constants } c \\
&\tilde{\theta}(x) = x \quad \text{for all variables } x \\
&\tilde{\theta}(\mathtt{f}\,(t_1, \ldots, t_n)) = \mathtt{f}\,(\tilde{\theta}(t_1), \ldots, \tilde{\theta}(t_n))
\end{aligned}$$

Usually, we write θ for $\tilde{\theta}$. As a conventional notation, a substitution θ that maps a variable x to a term t is written $[x \mapsto t]$.

To illustrate the previous concepts, we consider an example where the signature Σ contains a set of sorts $\mathbb{S} = \{\mathsf{bool}, \mathsf{int}\}$ and a family $\mathbb{F}$ whose members are $\mathbb{F}_{\epsilon,\mathsf{bool}} = \{\mathtt{true}, \mathtt{false}\}$, $\mathbb{F}_{\mathsf{bool},\mathsf{bool}} = \{\mathtt{not}\}$, $\mathbb{F}_{\epsilon,\mathsf{int}} = \{\mathtt{zero}\}$, $\mathbb{F}_{\mathsf{int},\mathsf{int}} = \{\mathtt{succ}\}$. We write $x_1, \ldots, x_n$ for variables of sort bool and $y_1, \ldots, y_n$ for variables of sort int. The following are all terms of sort bool: $\mathtt{false}, \mathtt{not(not(true))}, \mathtt{not}(x_1)$. Examples of terms of sort int are: $\mathtt{zero}, \mathtt{succ(succ(succ}(y_1)))$. The term algebra $\mathcal{T}$ of this signature has carrier sets $\mathbb{T}_{\mathsf{bool}}$ and $\mathbb{T}_{\mathsf{int}}$ that contain all terms of sort bool and int, respectively. The algebra assigns the constants $\mathtt{true}, \mathtt{false}, \mathtt{zero}$ to themselves and has two functions: *not* and *succ*, where each function corresponds to the respective operator symbol. For instance, for a term $t_1 \in \mathbb{T}$ of sort bool, we have: $\mathit{not}(t_1) = \mathtt{not}(t_1)$. We distinguish here between $\mathit{not}(t_1)$ which means the application of the function *not* to the term t_1 and $\mathtt{not}(t_1)$, which is the literal formed from the symbols "not", "(", "t_1", and ")". It is worth noting that the set of terms is countably infinite and that we do not have any rule for equating terms, so $\texttt{not(true)} \neq \texttt{false}$, this is due to the purely syntactic nature of terms.

Using the same signature Σ defined in the example of the previous paragraph, we may define the following Σ-algebra:

- Carrier of sort bool is $\{\top, \bot\}$.
- Carrier of sort int is the set $\mathbb{N}$ of natural numbers.
- Operator symbols are assigned the following functions and constants:
  The symbol $\texttt{true}$ is assigned $\top$
  The symbol $\texttt{false}$ is assigned $\bot$
  The symbol $\texttt{not}$ is assigned the function $\neg$, where $\neg(\top) = \bot$ and $\neg(\bot) = \top$
  The symbol $\texttt{zero}$ is assigned the number $0 \in \mathbb{N}$
  The symbol $\texttt{succ}$ is assigned the function $\mathbf{i} : \mathbb{N} \rightarrow \mathbb{N}$, where $\mathbf{i} = \{(n_1, n_2) \mid n_2 = n_1 + 1\}$

In the framework of algebraic semantics, each ground term of a signature Σ is interpreted as a member of the carrier set of a Σ-algebra. For instance, in the example above, $\texttt{false}$ is interpreted as $\bot$ and $\texttt{not(not(true))}$ is interpreted as $\neg(\neg(\top)) = \top$. Similarly, $\texttt{succ(succ(succ(zero)))}$ is interpreted as the number 3. If the Σ-Algebra is the term algebra $\mathcal{T}$, then each ground term is interpreted as itself, i.e., just a syntactic sequence of symbols. Interpretations of terms that contain variables will depend on the substitution applied to variables. For instance, in the example above, the interpretation of the term $t = \texttt{succ(succ(succ(}y_1\texttt{)))}$ will depend on the value assigned to y_1 by a certain substitution, e.g., if we have a substitution $\theta[y_1 \mapsto 0]$, then the term t will be interpreted as the number 3.

4.1.2 Message Algebra

We define protocol messages as a set $\mathbb{M}$ of ground terms of a Σ-term algebra. In other words, the set $\mathbb{M}$ of messages is a subset of the set $\mathbb{T}_\Sigma$ of well-sorted ground terms. The signature Σ is order-sorted [50, 98], which means that the set of sorts is a partially ordered set $(\mathbb{S}, \leq_{\mathbb{S}})$. The same order relation can be extended to arities, where for any two arities $w = S_1 \ldots S_n$ and $w' = S'_1 \ldots S'_n$, we write $w \leq_{\mathbb{S}} w'$ if $\forall i \in \{1, \ldots, n\} \centerdot S_i \leq_{\mathbb{S}} S'_i$. We also order ranks $\langle w, S \rangle$ in a similar manner. The definitions are the same as in the case of multi-sorted signatures, except for the following restrictions [50]:

- For any two sorts S and S', $S \leq_{\mathbb{S}} S'$ means that $\mathbb{A}_S \subseteq \mathbb{A}_{S'}$, i.e., the carrier of sort S is a subset of the carrier of sort S'.

- For any operator symbol $\mathtt{f} \in \mathbb{F}_{w,S} \cap \mathbb{F}_{w',S'}$ if $w \leq_{\mathbb{S}} w'$ then $S \leq_{\mathbb{S}} S'$.

The second condition above is called the monotonicity condition. To clarify it, assume two arities: $w = S_1 \ldots S_n$ and $w' = S'_1 \ldots S'_n$. In an algebra $\mathcal{A}$, the operator symbol $\mathtt{f} \in \mathbb{F}_{w,S}$ will be assigned a function $f^{\mathcal{A}} : \mathbb{A}_{S_1} \times \ldots \times \mathbb{A}_{S_n} \rightarrow \mathbb{A}_S$ and the operator symbol $\mathtt{f} \in \mathbb{F}_{w',S'}$ will be assigned a function $f^{\mathcal{A}} : \mathbb{A}_{S'_1} \times \ldots \times \mathbb{A}_{S'_n} \rightarrow \mathbb{A}'_S$. If we have $w \leq_{\mathbb{S}} w'$, the domains of both functions will be related by a subset relation, i.e., $\mathbb{A}_{S_1} \times \ldots \times \mathbb{A}_{S_n} \subseteq \mathbb{A}_{S'_1} \times \ldots \times \mathbb{A}_{S'_n}$. The monotonicity conditions means that functions related this way will have $\mathbb{A}_S \subseteq \mathbb{A}'_S$. This is because the functions should have the same value for the same argument, which implies that $S \leq_{\mathbb{S}} S'$.

We now define the order-sorted signature Σ of our message algebra:

Sorts: $\mathbb{S} = \{\mathsf{Msg}, \mathsf{Sig}, \mathsf{Agt}, \mathsf{Txt}, \mathsf{Nat}, \mathsf{Key}, \mathsf{Pair}, \mathsf{Cphr}\}$

Subsorting: $\mathsf{Agt} <_{\mathbb{S}} \mathsf{Msg},\ \mathsf{Txt} <_{\mathbb{S}} \mathsf{Msg},\ \mathsf{Nat} <_{\mathbb{S}} \mathsf{Msg},$
$\mathsf{Key} <_{\mathbb{S}} \mathsf{Msg},\ \mathsf{Pair} <_{\mathbb{S}} \mathsf{Msg},\ \mathsf{Cphr} <_{\mathbb{S}} \mathsf{Msg}$

Constants: A number of disjoint sets, one set per sort.
Sig has only two constants: `start` and `terminate`.

Operator Symbols: (4.1)

arity (w)	**coarity** (S)	$\mathbb{F}_{w,S}$
Msg.Msg	Pair	{`conc`}
Msg.Key	Cphr	{`encrypt`}
Cphr.Key	Msg	{`decrypt`}
Pair	Msg	{`fst`}
Pair	Msg	{`snd`}
Key	Key	{`inverse`}

The sorts Msg, Sig, Agt, Txt, Nat, Key, Pair, Cphr, represent messages, start and terminate signals, agent names, text, natural numbers (nonces), cryptographic keys, paired messages and encrypted messages (cipher text), respectively. The subsorting relation $\leq_{\mathbb{S}}$ is the reflexive and transitive closure of the relation $<_{\mathbb{S}}$ defined in (4.1). We assume each sort has a countably infinite set of constants, except for the sort Sig which has only the two constants defined in (4.1). Operator symbols, in the order they are listed in (4.1), represent pairing of messages, encryption, decryption, pair-selectors to select the first or second element of a pair, and, finally, the inverse operation for keys that assigns a decryption key for a certain encryption key. Moreover, we assume a set $\mathbb{X}$ of sorted variables, where $\mathbb{X} = \bigcup_{S \in \mathbb{S}} \mathbb{X}_S$. The definition of well-sorted terms for an order-sorted signature differs from the one mentioned for the multi-sorted case. This is because a term with sort S can take the place of a term with sort S' if $S \leq_{\mathbb{S}} S'$, the inverse is not true however. We formalize this idea by defining the set $\mathbb{T}_{\Sigma,S}(\mathbb{X})$ of well-sorted terms of sort S over the signature Σ to be the least set that contains:

- All constants and variables of sort S.
- All terms in all sets $\mathbb{T}_{\Sigma,S'}(\mathbb{X})$, where $S' \leq_{\mathbb{S}} S$.
- All strings of the form "$\mathtt{f}(t_1, \ldots, t_n)$", for each operator symbol $\mathtt{f} \in \mathbb{F}_{w,S}$ whose arity is $w = S_1 \ldots S_n \neq \epsilon$, coarity is S, and $t_1, \ldots, t_n$ are terms of sorts $S_1, \ldots, S_n$, respectively.

The definition above means that if a substitution maps variables of sort S to terms of the same sort, then it is valid to have a substitution $\theta[x \mapsto t]$, where x is of sort S, t is of sort S' and $S' \leq_{\mathbb{S}} S$. The Σ-term algebra $\mathcal{T}$ associated with the order-sorted signature Σ is defined as:

- For each sort $S \in \mathbb{S}$, the carrier of S is $\mathbb{T}_{\Sigma,S}(\mathbb{X})$, i.e., the set of terms of sort S.
- For each operator symbol $\mathtt{f} \in \mathbb{F}_{w,S}$ with arity $w = S_1 \ldots S_n \neq \epsilon$ and coarity S, there exists a function $f^{\mathcal{T}} : T_{\Sigma,S_1}(\mathbb{X}) \times \ldots \times T_{\Sigma,S_n}(\mathbb{X}) \rightarrow T_{\Sigma,S}(\mathbb{X})$, where $f^{\mathcal{T}}(t_1, \ldots t_n) =$ "$\mathtt{f}(t_1, \ldots, t_n)$".

Again, we distinguish between $f^{\mathcal{T}}(t_1, \ldots t_n)$ where the parentheses represent the application of the function $f^{\mathcal{T}}$ to its arguments and the string "$\mathtt{f}(t_1, \ldots, t_n)$" which is just a sequence of symbols, hence the use of the double quotes and the different font.

The Σ-term algebra of the order-sorted signature Σ defined in 4.1 is our message algebra in the following manner:

- The set $\mathbb{M}$ of messages that are exchanged between communicating agents is defined as $\mathbb{M} \subseteq \bigcup_{S \in \mathbb{S}} \mathbb{T}_{\Sigma,S}$, i.e., a subset of the set of all ground terms of the algebra.

- The functions defined by the algebra will be used to perform abstract computations on messages. In our algebra, functions have the same names as operator symbols but written in an italic font, e.g., the function *encrypt* is assigned to the operator symbol `encrypt`.

The reason for which we defined the set $\mathbb{M}$ as a subset of the set of ground terms is that there are terms that are not messages, e.g., `decrypt(encrypt(m,k),k)` or `decrypt(encrypt(m,k),k')`, since, if we take into account the semantics of operators, the first term should transform to m and the second term is a meaningless message. Cases like the first one would be solved by using a term rewriting system and this is what we are going to deal with when we discuss equational theories. Cases like the second one, however, will be avoided by defining a grammar that specifies message syntax, this way these terms will not be considered syntactically valid messages. This is why we define the following grammar for messages:

$$m ::= a \mid c \mid n \mid k \mid k^{-1} \mid \langle m, m \rangle \mid \{m\}_k \mid \texttt{start} \mid \texttt{terminate} \tag{4.2}$$

The grammar rule in (4.2) indicates that a message can be an agent name "a", text (a string) "c", a nonce "n", the symbol `start`, or the symbol `terminate`. All these messages are called atomic. The symbols `start` and `terminate` will be used in the protocol model to indicate the beginning and the end of protocol sessions. Moreover, a message can be the inverse of a key "k^{-1}", the pairing of two messages "$\langle m_1, m_2 \rangle$" or a message encrypted by a key "$\{m\}_k$". The notations $\_^{-1}$, $\langle \_, \_ \rangle$ and $\{\_\}_\_$ are used instead of `inverse(_)`, `conc(_,_)`, `encrypt(_,_)` which were defined in the signature Σ to comply with the standard notation used in security protocols. Also, sometimes we write m_1, m_2 for $\langle m_1, m_2 \rangle$, when no confusion could arise.

4.1.3 Abstract Computation Procedures

During the execution of protocol steps, communicating agents often need to compute new messages in order to send them to other agents or to update their knowledge. In our model, the basic tools used to perform computations on messages are the functions defined in the message algebra, i.e., the Σ-term algebra. In general, however, the computations required to be performed on messages can seldom be expressed as one single function of the algebra. Instead, we need to describe *algorithms* that combine a finite number of the algebraic functions in order to compute more complex ones. A language to describe such algorithms was developed in the framework of abstract computation [109] whose concern is to investigate computable functions on algebras. We will denote this language by $\mathcal{W}$ as it is called the "while" language in [109]. The language constructs are: Basic functions of the algebra, sequencing of functions, conditional branching and iteration. An algorithm description written in $\mathcal{W}$ will be called a $\mathcal{W}$-procedure. The syntax and semantics of $\mathcal{W}$ are detailed in [109], we briefly recall the syntax:

$$\begin{aligned} &\textit{procedure} ::= \texttt{proc in } l_1 \texttt{ out } l_2 \texttt{ aux } l_3 \texttt{ begin } S \texttt{ end} \\ &S ::= \texttt{skip} \mid x := t \mid S_1 ; S_2 \mid \texttt{while } b \texttt{ do } S \texttt{ od} \mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi} \end{aligned} \tag{4.3}$$

In (4.3) above, l_1, l_2, and l_3 are lists of variables that represent inputs, outputs and local variables respectively. The syntactic category S represents statements, which can be (in the order they are listed in (4.3)): No action, an assignment of a term t of the algebra to a variable of the same sort, the sequencing of two statements, `while` loops with a condition b, and, finally, conditional branching with condition b. It is assumed that the algebra either contains the boolean sort or can be extended to include it. In either case the condition b is a boolean term of the algebra which is also assumed to be equipped with an equality predicate between pairs of terms. In the next chapter, procedures will be used to specify computations performed by communicating agents to produce messages.

4.2 Messages and Knowledge

In this section, we investigate how agents can deduce knowledge from a set of messages they already possess. We begin by introducing equation systems to the message algebra, then we present a system of deductive rules for equational theories.

4.2.1 Equational Theories

A term $t = \mathtt{f}(t_1, \ldots, t_n)$ in the set $\mathbb{T}_\Sigma(\mathbb{X})$ of terms can be regarded as an ordered tree whose nodes are labeled by operator symbols (including constants) or variables in the following manner:

- If the term is a constant or a variable, the term tree is just a node without children, i.e., a leaf.
- If the term has the form $\mathtt{f}(t_1, \ldots, t_n)$, the term tree has a root labeled by $\mathtt{f}$ and n children that are subtrees which represent the terms t_1 to t_n in order.

A term t' is a subterm of $t = \mathtt{f}(t_1, \ldots, t_n)$ if $t' = t$ or t' is a subterm of any $t_i, i \in \{1, \ldots, n\}$. We use the Dewey decimal notation [64] to order nodes in trees. A position π in a term t is the Dewey number of a certain node in the tree of t. The subtree whose root is π represents a subterm of t, which we denote by $t|_\pi$. A context is a term t over the signature $\Sigma \cup \{\Box\}$, i.e., $t \in \mathbb{T}_{\Sigma\cup\Box}(\mathbb{X})$, such that t has exactly one appearance of $\Box$ at a certain position π, contexts are denoted by $t[\,]_\pi$. The symbol $\Box$ in a context may be replaced by a term t' in which case we write $t[t']_\pi$, the same notation is also used for replacing the subterm at position π with t'.

An equation over the set $\mathbb{T}_\Sigma(\mathbb{X})$ of terms, is a pair (u, v) such that u and v are both in $\mathbb{T}_\Sigma(\mathbb{X})$ and have the same sort. We write $u = v$ to designate the equation

(u, v). A set E of equations over $\mathbb{T}_\Sigma(\mathbb{X})$ is called an equation system. Moreover, E induces a relation $\rightarrow_E \subseteq \mathbb{T}_\Sigma(\mathbb{X}) \times \mathbb{T}_\Sigma(\mathbb{X})$, defined inductively as follows:

- $(u = v \in E) \Rightarrow (u \rightarrow_E v)$
- $(u = v \in E) \Rightarrow (\forall \theta, t, \pi \,.\, t|_\pi = \theta(u) \Rightarrow t[\theta(u)]_\pi \rightarrow_E t[\theta(v)]_\pi)$

We note here that the definition of the relation $\rightarrow_E$ is based on the fact that, in a context $t[\,]_\pi$, we can replace an *instantiation* $\theta(u)$ of the left-hand side u of an equation by an instantiation $\theta(v)$ of the right-hand side. In this case, we say that the the term $t[\theta(v)]_\pi$ is derived from $t[\theta(u)]_\pi$. However, equations also mean that we can replace right-hand side instantiations by left-hand side ones. We therefore define the relation $\leftrightarrow_E$ to be the symmetric closure of $\rightarrow_E$. Moreover, using the usual notation, $\rightarrow^+$ and $\leftrightarrow^+_E$ are the transitive closures of $\rightarrow_E$ and $\leftrightarrow_E$, respectively, while $\rightarrow^*_E$ and $\leftrightarrow^*_E$ are their reflexive and transitive closures. A term s is said to be derived from a term t if $t \rightarrow^*_E s$. We also say that s is a syntactic or equational consequence of t if $t \leftrightarrow^*_E s$. The definition of the relation $\leftrightarrow^*_E$ implies that it is an equivalence relation (reflexive, transitive, and symmetric) and that it is closed under substitutions and term formation with function symbols, i.e., $t_1 \leftrightarrow^*_E t_2$ implies $\forall \theta \,.\, \theta(t_1) = \theta(t_2)$ and $t_1 \leftrightarrow^*_E t'_1 \wedge \ldots \wedge t_n \leftrightarrow^*_E t'_n$ implies $\forall \mathtt{f} \in \mathbb{F}_{w,S} \,.\, \mathtt{f}(t_1, \ldots, t_n) \leftrightarrow^*_E \mathtt{f}(t'_1, \ldots, t'_n)$, respectively. Therefore, the relation $\leftrightarrow^*_E$ is a congruence. To determine if s is a syntactic consequence of t, we have to check if (t, s) is in $\leftrightarrow^*_E$. As another option, we may also use deductive proof rules of equational logic to prove whether s is a syntactic consequence of t under the equation system E, written $E \vdash t = s$. The rules are:

$$\frac{t = s \in E}{E \vdash t = s} \qquad \frac{E \vdash t = s}{E \vdash \theta(t) = \theta(s)}$$
$$\frac{E \vdash t = s}{E \vdash s = t} \qquad \frac{E \vdash t_1 = t_2 \quad E \vdash t_2 = t_3}{E \vdash t_1 = t_3} \tag{4.4}$$
$$\frac{E \vdash t_1 = t'_1 \quad E \vdash t_2 = t'_2 \ \ldots \ E \vdash t_n = t'_n}{E \vdash \mathtt{f}(t_1, \ldots, t_n) = \mathtt{f}(t'_1, \ldots, t'_n)}$$

The set of all equations $t = s$ such that $E \vdash t = s$ is called the equational theory of E, where the equations $u = v$ in the set E are considered the axioms of the theory. The role of the proof rules of equational logic is then to decide if an equation $t = s$ is in the equational theory of E.

A term rewrite system R, defined over the set $\mathbb{T}_\Sigma(\mathbb{X})$ of terms, is a set of rules of the form $l \rightarrow r$, under the conditions that [22]: (1) l is not a single variable and (2) all variables of r appear in l. Each rule $l \rightarrow r$ in R can be regarded as an oriented equation, i.e., left-hand side instantiations can be replaced by right-hand side ones and not the other way around. As with an equational system E, R induces a relation $\rightarrow_R$ over $\mathbb{T}_\Sigma(\mathbb{X})$, called the rewrite relation, whose definition is the same as $\rightarrow_E$, the only difference is that we have to replace the condition $u = v \in E$ by $u \rightarrow v \in R$ and $\rightarrow_E$ by $\rightarrow_R$. We say that a term t rewrites to t' if $t \rightarrow_R^* t'$. A relationship between an equational system E and a rewrite system R can be established if R is sound and adequate for E [42]. Soundness means that all the rules $l \rightarrow r$ of R are in $\leftrightarrow_E^*$, i.e., $l \rightarrow r \in R$ implies $l \leftrightarrow_E^* r$ and adequacy means that $u = v \in E$ implies $u \leftrightarrow_R^* v$, in such a case we have $\leftrightarrow_E^* = \leftrightarrow_R^*$. Hence, a term rewrite system R that is sound and adequate for a certain equation system E, can be used to decide the syntactic consequence relationship. In order to do so this, however, R has to satisfy some properties, namely termination and confluence [22]. A terminating and confluent term rewrite system R is called convergent or complete and in this case each term has a unique normal form, which is a term that cannot be rewritten any further. With such a system it is possible to determine whether a term s is a syntactic consequence of t, i.e., whether $t \leftrightarrow_E^* s$, by rewriting both t and s to their normal forms and then comparing them.

Proof rules of equational logic and term rewrite systems both deal with syntactic terms, i.e., strings of symbols, which may be operator symbols or variables. Therefore, equality was called syntactic consequence or equational consequence, and relates two terms that may have different syntactic forms. As described in Section

4.1.1, algebras are the semantic models for terms in the sense that an algebra $\mathcal{A}$, with a family of substitutions $\theta_S : \mathbb{X}_S \to \mathbb{A}_S$ assigns to each term t, of sort S, an element a in $\mathbb{A}_S$, the carrier of sort S. An equation $u = v$ (where both terms u and v are of sort S) is valid in an algebra $\mathcal{A}$ if for all substitutions θ_S, the terms u and v are assigned the same element in the carrier $\mathbb{A}_S$ of the algebra. In this case, we write $\mathcal{A} \models u = v$, meaning that $\mathcal{A}$ models the equation $u = v$. On the other hand, an equation $u = v$ is satisfiable in $\mathcal{A}$, if there exists a substitution θ_S such that u and v are assigned the same element in $\mathbb{A}_S$. An algebra $\mathcal{A}$ is called a model for an equational system E if all equations $u = v$ in E are valid in $\mathcal{A}$. Moreover, we define the relation $=_E$ on $\mathbb{T}_\Sigma(\mathbb{X})$ such that $u =_E v$ means that the equation $u = v$ is valid in all models of E.

Assume an equational system E and an algebra $\mathcal{A}$ that models it, we would like to know all the equations (other than those in E) that are valid in $\mathcal{A}$, i.e., all pairs (t, s) of terms such that $t =_E s$. By Birkhoff's Theorem [22], we only need to construct the relation $\leftrightarrow_E^*$, since the theorem states that $t =_E s$ if and only if $t \leftrightarrow_E^* s$, it therefore links the relation $\leftrightarrow_E^*$ defined syntactically to $=_E$ which is defined semantically with respect to models. This theorem basically states that the proof system of equational logic is sound and complete.

4.2.2 Equations for the Message Algebra

In this section, we expand the capabilities of the Dolev-Yao model by adding an equational theory to the algebra of messages. Adding the equational theory has several advantages:

- The ability of the intruder to deduce and recognize messages can be expanded.
- By providing an equational theory tailored for a specific protocol, which will depend on the cryptographic system that the protocol uses, the security analysis is more rigorous and the ability to find security flaws is increased.

We equip the algebra of messages with an equational system E that is chosen according to the specific properties of the protocol and its underlying cryptosystem. In this case, the congruence induced by E is used to define equivalence classes between messages. We write $m =_E n$ to say that m is congruent to n under E. We focus our attention to equational theories for which there exists a convergent (complete) term rewriting system. For these systems, each term has a unique normal form. Therefore, the problem of determining whether $m =_E n$ for any two closed terms m and n is decidable by rewriting both m and n and checking if they have the same normal form. As an example, in the original Dolev Yao model [44], E will contain:

$$\begin{array}{lcl} \texttt{decrypt}(\{x\}_y, y) & = & x \\ \texttt{fst}(\langle x, y\rangle) & = & x \\ \texttt{snd}(\langle x, y\rangle) & = & y \end{array}$$

4.2.3 Building Knowledge

During the communication steps of a certain protocol, principals exchange messages that are meant to communicate knowledge between them. Each message can contain components that are already known to the recipient of the message and other components that represent new knowledge. Upon the reception of a message, the agent knowledge is increased in two ways: The message itself is added to the set of messages that the agent knows, and the message enables the deduction of further messages. For instance, if an agent receives an encrypted message while it knows the encryption key, then it can deduce the plain-text message. To express this, we define the function **know** that maps an agent's name into the set of messages known to the agent. For any agent A, **know**(A) is the smallest set $\mathcal{M}$, which satisfies the following conditions:

$$\begin{array}{rcll}
(m \in \mathcal{M}) \wedge (m' =_E \mathcal{M}) & \Rightarrow & m' \in \mathcal{M} & \text{Equational deduction.} \\
(m \in \mathcal{M}) \wedge (m' \in \mathcal{M}) & \Rightarrow & m \; op \; m' \in \mathcal{M} & \text{Arithmetic operation.} \\
(m \in \mathcal{M}) \wedge (k \in \mathcal{M}) & \Rightarrow & \{m\}_k \in \mathcal{M} & \text{Encryption.} \\
(m \in \mathcal{M}) \wedge (m' \in \mathcal{M}) & \Rightarrow & m.m' \in \mathcal{M} & \text{Concatenation.} \\
(\{m\}_k \in \mathcal{M}) \wedge (k' \in \mathcal{M}) \wedge (k' = k^{-1}) & \Rightarrow & \{m\} \in \mathcal{M} & \text{Decryption.} \\
m.m' \in \mathcal{M} & \Rightarrow & \{m, m'\} \in \mathcal{M} & \text{Deconcatenation.}
\end{array}$$

The set $\mathcal{M}$ initially contains all messages known to the agent prior to executing any communication steps such as names of other agents, their public key and the agent's own public and private keys. Whenever, the agent produces a fresh message during a protocol run or receives a message in a protocol step, these messages are added to $\mathcal{M}$ whose size increases by the number of messages that can be deduced after the addition of m.

4.3 Game Semantics for Messages

We focussed thus far on the algebraic semantics of messages, in which each message is interpreted as an element of a carrier set of an algebra. In this section, we assign game semantics to messages in the set $\mathbb{M}$ which we defined by a grammar as a subset of the terms of the algebra. Our purpose is to provide a unified approach to the semantics of protocols, in which both messages and communications steps are expressed as strategies over games.

4.3.1 Atomic Messages

As explained earlier, atomic messages are messages that are either agent names, cryptographic keys, natural numbers, or constants (text or symbols). In algebraic semantics these were the basic sorts of the algebra. in game semantics however, they are regarded as types. Each one of these types is represented by a game, so the types A, K, N, T are represented by the games Agt, Key, Nat, Txt respectively. In addition, we define the game Msg to represent the type M, which corresponds to the sort Msg of the algebra. From the subsort relation in the rules of (4.1), we deduce that all the types are subtypes of M. In game semantics, this means that any valid strategy over any of the games Agt, Key, Nat, Txt is also a valid strategy over the game Msg [33]. The idea is that games represent types, whereas strategies represent algorithms. An example is given below on how to represent constants and variables of the type N:

$$\begin{array}{cc} \mathsf{Nat} & \\ q & O \\ n & P \\ & \end{array} \qquad \begin{array}{cccc} \mathsf{Nat} & \multimap & \mathsf{Nat} & \\ & & q & O \\ q & & & P \\ n' & & & O \\ & & n' & P \end{array}$$

The game to the left represents constants of natural numbers, a certain strategy over this game represents a particular constant, for instance the constant 3 is represented by the strategy $\sigma = q.3$. This means that the environment (O) asks for a constant and the system (P) replies with the number 3. The game to the right represents variables of natural numbers, i.e., the lambda term $\lambda x : \mathsf{N}.x$, which has type $\mathsf{N} \longrightarrow \mathsf{N}$ (corresponding to the game $\mathsf{Nat} \multimap \mathsf{Nat}$). Any strategy τ over this game represents a variable and has the form $\tau = q.q.n'.n'$, where n' is any natural number. If we apply the value 3 to the lambda term above we get $(\lambda x : \mathsf{N}.x)3 = 3$, the equivalence of this operation in game semantics is interaction between strategies (parallel composition plus hiding) defined in (2.10). In order to conform with the definition of interaction between strategies, we redefine the game of constant naturals to be $\mathsf{Emp} \multimap \mathsf{Nat}$, where Emp is the empty game (has no moves). We do this so that we can compose the two strategies $\sigma : \mathsf{Emp} \multimap \mathsf{Nat}$ $(\sigma = q.3)$ and $\tau : \mathsf{Nat} \multimap \mathsf{Nat}$ $(\tau = q.q.n'.n')$. Now, hiding the common Nat game will leave us with a $\mathsf{Emp} \multimap \mathsf{Nat}$ game which represents the constant 3 as expected, i.e., $\sigma;\tau = q.3$. This is shown in

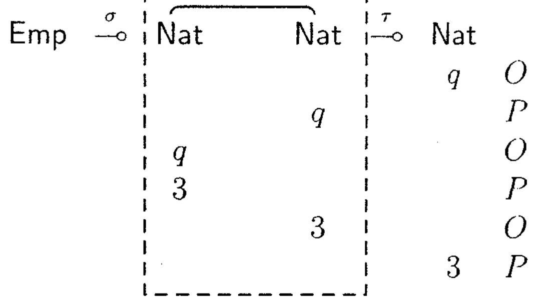


Figure 4.1: Interaction between startegies.

Figure 4.1. We notice here how the *copy cat* strategy between the two Nat games served as a link between σ and τ. Of course, *hiding* is done by removing whatever is included in the dashed rectangle. The same example could be generalized to all types of atomic messages mentioned above.

4.3.2 Composed Messages

The composed messages that we define here result from the outcome of two operations: Concatenation, and encryption. It is worth noting that composed messages are also of type M. The game used to express the *operation* of concatenation is the game Msg $\multimap$ Msg $\multimap$ Msg (representing type M $\rightarrow$ M $\rightarrow$ M). We note that this is the representation of a function type, the return type of the function is M (represented by the game Msg) as we mentioned above. The game used to represent the encryption operation is the game Msg $\multimap$ Key $\multimap$ Msg. The specific operations of concatenation and encryption are expressed by strategies over the respective games. For instance the concatenation operation is represented by the strategy $q.q.m_1.q.m_2.\mathsf{conc}(m_1, m_2)$. The encryption operation, on the other hand, is represented by the strategy $q.q.m.q.K.\mathsf{encrypt}(m, k)$. The functions conc : M $\rightarrow$ M $\rightarrow$ M and encr : M $\rightarrow$ K $\rightarrow$ M represent the concatenation and encryption operations respectively. An example is given below for the encryption operation:

$$\begin{array}{ccccccc}
\mathsf{Msg} & \multimap & (\mathsf{Key} & \multimap & \mathsf{Msg}) & \\
 & & & & q & O \\
q & & & & & P \\
m_i & & & & & O \\
 & & q & & & P \\
 & & K & & & O \\
 & & & & \mathsf{encr}(m_i, K) & P
\end{array}$$

The last move by the player in the game above is $\mathsf{encr}(m_i, K)$, which means that we assume the existence of a predefined function $\mathsf{encr} : \mathsf{M} \rightarrow \mathsf{K} \rightarrow \mathsf{M}$ that the player is able to compute. In genral, we assume that players are able to compute any function that can be expressed as a procedure of the "while"-language presented in 4.1.3.

4.3.3 Category of Message Games

We define the category $\mathcal{MGC}$ that contains the games defined for messages above. In this category:

- Objects are games, e.g., Agt, Key, Nat, Txt, and Emp.
- A morphism between any pair of games G and H ($f : G \rightarrow H$) is a strategy σ over the game $G \multimap H$.
- The identity morphism for any game G is the copy cat strategy over the game $G \multimap G$.
- Composition of morphisms is interaction between strategies.

To prove that $\mathcal{MGC}$ is a category we have to prove the *associativity law* of morphism composition, i.e., for any three morphisms f, f', and f'', we have $(f \circ f') \circ f'' = f \circ (f' \circ f'')$. We also have to prove the *identity law*, i.e., for any morphism f and an identity morphism id_G over any game G, we have $f \circ id_G = id_G \circ f = f$.

Proof of Associativity of Composition. This amounts to proving that interaction of strategies is associative.

For any 4 games G, H, I, and J, suppose we have the strategies: $\sigma : G \multimap H$, $\tau : H \multimap I$, and $\iota : I \multimap J$. We have the following definitions:

$$\begin{aligned}
(\sigma;\tau);\iota &= (((\sigma\|\tau)/H)\|\iota)/I \\
(\sigma\|\tau)/H &= \{s \upharpoonright G, I \mid s \in \sigma\|\tau\} \\
\sigma\|\tau &= \{s \in (M_G + M_H + M_I)^* \mid s \upharpoonright G, H \in \sigma \wedge s \upharpoonright H, I \in \tau\}
\end{aligned}$$

From the definitions above we can write:

$$\begin{aligned}
(\sigma;\tau);\iota &= ((\sigma\|\tau)\|\iota)/H, I \\
&= (\sigma\|(\tau\|\iota))/H, I \\
&= (\sigma\|((\tau\|\iota)/I))/H \\
&= \sigma;(\tau;\iota) \quad \text{which proves associativity.}
\end{aligned}$$

Proof of Identity law. This amounts to proving that the interaction of any strategy with the copy cat strategy yields the same strategy.

Assume any two games G and H such that we have the strategy $\sigma : G \multimap H_1$ and the copy cat strategy $id_H : H_1 \multimap H_2$. Here the subscripts 1 and 2 are just used to denote different copies of the game H.

$$\begin{aligned}
id_H &= \{s \in P^{even}_{H_1 \multimap H_2} \mid \forall t \text{ even-length prefix of } s \,.\, t \upharpoonright H_1 = t \upharpoonright H_2\} \\
(\sigma; id_H) &= (\sigma \| id_H)/H_1) \\
(\sigma \| id_H)/H_1 &= \{s \upharpoonright G, H_2 \mid s \in \sigma \| id_H\} \\
\sigma \| id_H &= \{s \in (M_G + M_{H_1} + M_{H_2})^* \mid s \upharpoonright G, H_1 \in \sigma \wedge s \upharpoonright H_1, H_2 \in id_H\}
\end{aligned}$$

From the definitions above and assuming even-length sequences (strategies):

$$s \upharpoonright H_2 = s \upharpoonright H_1$$

In $\sigma \| id_H$:

$$s \upharpoonright G, H_1 = s \upharpoonright G, H_2 \in \sigma$$

By hiding moves in H_1 to obtain $\sigma; id_H$ we end up with:

$s \upharpoonright G, H_2 \in \sigma$

Therefore, all sequences of $\sigma; id_H$ are sequences of σ, which proves the identity law.

4.3.4 Semantics of Messages

Before defining a semantic function for messages we need a few definitions:

If σ is a strategy over the game $\mathsf{Msg} \multimap \mathsf{Msg} \multimap \mathsf{Msg}$, we define the operator $\mathfrak{U}(.)$ such that $\mathfrak{U}(\sigma)$ is the strategy over the game $\mathsf{Msg} \otimes \mathsf{Msg} \multimap \mathsf{Msg}$. This is similar to the operation of *un-currying* in functional programming [11].

For any three Games G, H and F such that we have two strategies: The strategy $\sigma : G \multimap H$, and the strategy $\tau : G \multimap F$, we define the strategy $\langle \sigma, \tau \rangle$ [12] as the strategy over the game $G \multimap H \otimes F$, where:

$$\begin{aligned}
\langle \sigma, \tau \rangle = \; & \{s \in P_{G \multimap H \otimes F} \mid s \upharpoonright G, H \in \sigma \wedge s \upharpoonright F = \epsilon\} \cup \\
& \{s \in P_{G \multimap H \otimes F} \mid s \upharpoonright G, F \in \tau \wedge s \upharpoonright H = \epsilon\}
\end{aligned}$$

For any three Games G, H and F such that we have two strategies: The strategy $\sigma : G \multimap H$, and the strategy $\tau : H \multimap F$. We define interaction between strategies as $\sigma;\tau$ which yields the strategy over the game $G \multimap F$. This was defined in Section 2.2.5 and related to the concept of parallel composition plus hiding in CSP.

The semantic function for messages is defined in Table 4.1:

Table 4.1: Semantic definitions of messages.

$\mathsf{A}, \mathsf{K}, \mathsf{N}, \mathsf{T}$ are types for agent names, keys, natural numbers and constant messages (text) respectively.
All above types are subtypes of M; the type of messages.

$\mathsf{encr} : \mathsf{M} \rightarrow \mathsf{K} \rightarrow \mathsf{M}$	Encryption function
$\mathsf{conc} : \mathsf{M} \rightarrow \mathsf{M} \rightarrow \mathsf{M}$	Concatenation function

$$\mathfrak{S} : \mathsf{M} \rightarrow \textstyle\prod \tau \in \mathcal{T} \,.\, \mathcal{S}_\tau$$

$$\begin{aligned}
\mathfrak{S}[\![a]\!](\mathsf{A}) &= \sigma : \mathsf{Emp} \multimap \mathsf{Agt}, \text{ where } \sigma = \{\epsilon, q.a\} \\
\mathfrak{S}[\![c]\!](\mathsf{T}) &= \sigma : \mathsf{Emp} \multimap \mathsf{Txt}, \text{ where } \sigma = \{\epsilon, q.c\} \\
\mathfrak{S}[\![k]\!](\mathsf{K}) &= \sigma : \mathsf{Emp} \multimap \mathsf{Key}, \text{ where } \sigma = \{\epsilon, q.k\} \\
\mathfrak{S}[\![n]\!](\mathsf{N}) &= \sigma : \mathsf{Emp} \multimap \mathsf{Nat}, \text{ where } \sigma = \{\epsilon, q.n\} \\
\mathfrak{S}[\![m_1, m_2]\!](\mathsf{M}) &= \sigma : \mathsf{Msg}, \text{ where } \sigma = \langle \mathfrak{S}[\![m_1]\!], \mathfrak{S}[\![m_2]\!] \rangle ; \mathfrak{U}(\sigma') \\
&\quad\ \sigma' \text{ is a strategy over } \mathsf{Msg} \multimap \mathsf{Msg} \multimap \mathsf{Msg} \text{ such that:} \\
&\quad\ \sigma' = \{\epsilon, q.q\} \cup \\
&\qquad\quad \{q.q.m.q \mid m : \mathsf{M}\} \cup \\
&\qquad\quad \{q.q.m.q.m'.\mathsf{conc}(m, m') \mid m, m' : \mathsf{M}\} \\
\mathfrak{S}[\![\{m\}_k]\!](\mathsf{M}) &= \sigma : \mathsf{Msg}, \text{ where } \sigma = \langle \mathfrak{S}[\![m]\!], \mathfrak{S}[\![k]\!] \rangle ; \mathfrak{U}(\sigma') \\
&\quad\ \sigma' \text{ is a strategy over } \mathsf{Msg} \multimap \mathsf{Key} \multimap \mathsf{Msg} \text{ such that:} \\
&\quad\ \sigma' = \{\epsilon, q.q\} \cup \\
&\qquad\quad \{q.q.m.q \mid m : \mathsf{M}\} \cup \\
&\qquad\quad \{q.q.m.q.k.\mathsf{encr}(m, k) \mid m : \mathsf{M}, k : \mathsf{K}\}
\end{aligned}$$

Chapter 5

Model for Security Protocols

In the previous chapter, we demonstrated our basic ideas about messages, in which we use an algebraic framework to model the structure of messages and the computations we can perform on them. In this chapter, we incorporate those ideas into our model of security protocols. In this model, protocols are games played between communicating agents, where the game tree represents all possible interactions that can take place between agents according to protocol specifications. In order to be able to specify protocols, we present a simple specification language that is close to the standard arrow-notation used in practice. Then, starting from a protocol specification, we show how to build the game semantics model of the protocol. The model that we adopt for the intruder is the extended Dolev-Yao model discussed in Chapter 3.

In the following sections, we begin by presenting the syntax that we propose for specifying security protocols, we then demonstrate how we derive computation procedures on messages from protocol specifications. This involves introducing the notion of frames. We follow this by definitions of game semantics for protocol specifications. This is necessary in order to construct the game tree. In other words, given a protocol specification written according to our defined syntax, we define a semantic interpretation function that will map the specification to a game tree which is the model we are going to use for the analysis in the next chapter.

5.1 Protocol Specifications

Intuitively, a security protocol specification describes a number of steps in which agents are exchanging messages. We regard the syntactic structure of a message as an indication of the computation that must be done by the agent sending the message. We therefore develop an algorithm that, given a certain message in a protocol specification, produces a procedure whose output is the required message and which is written in the $\mathcal{W}$ language presented in Section 4.1.3. The input to the procedure will be a collection of messages that represents the knowledge used to produce the output. In this section, we first present the syntax we use for protocol specifications, then we show how to develop algorithms for the computation of messages.

5.1.1 Syntax

The syntax that we propose for protocol specifications is close to the arrow notation (the Alice and Bob notation) traditionally used in cryptography to describe security protocols. We add more constructs that are aimed to resolve some of the ambiguities in this notation, which were usually dealt with by providing natural language descriptions of the protocol specifications. We therefore divide a protocol specification into a declaration part and a communication part. The communication part contains the arrow notation that specifies message exchanges, whereas the declaration part is composed of:

- *Knowledge declaration*: To declare messages that are stored in the agent knowledge and are persistent in all protocol executions.

- *Freshness declaration*: To declare messages that are freshly created by the agent in each protocol run, e.g., nonces or secrets.

The grammar for protocol specifications is shown below:

$$\begin{array}{rcl} \mathcal{P}rot & ::= & \mathcal{D}ecl\,.\,\mathcal{C}omm \;\mid\; \epsilon \\ \mathcal{D}ecl & ::= & \kappa_A \rhd m\,.\,\mathcal{D}ecl \;\mid\; \nu_A \rhd m\,.\,\mathcal{D}ecl \;\mid\; \epsilon \\ \mathcal{C}omm & ::= & \texttt{step}\, i \rhd A \rightarrow B :\; m\,.\,\mathcal{C}omm \;\mid\; \epsilon \end{array} \tag{5.1}$$

In the grammar above, A and B range over a set of names that are the protocol roles, m ranges over the set $\mathbb{M}$ of possible messages discussed in Section 4.1.2, and i is a natural number greater than zero. The declaration $\kappa_A \rhd m$ means that m is part of the initial persistent knowledge of A, and $\nu_A \rhd m$ means that m is freshly produced by A for each protocol session. As for $\texttt{step}\, i \rhd A \rightarrow B :\; m$, it means that message m is sent to the network by agent A at the i-th communication step and that it is intended to be received by agent B.

As mentioned in Chapter 2, protocol specifications of knowledge, freshness and behavior describe roles, whereas the actual implementations of these specifications are incorporated into agents that will interact to execute the protocol. For instance, a protocol $\wp$ with two roles A and B implies that a session (a run) of the protocol involves two agents, one playing the role of A and the other the role of B. In an actual environment, we may have all sorts of combinations where an agent may play multiple roles and a role may be played by many agents.

5.1.2 Frames and Computation

We mentioned in Chapter 3 that any model for security protocols tries to answer three questions: How to model computation, how to model communication and what are the intruder capabilities. As mentioned earlier, we adopt a Dolev-Yao intruder with extended capabilities. We deal with message communication when we begin defining games for protocols. In this section, we deal with message computation, namely, how to represent the generation of a message by an agent, knowing that this message may depend on previously known or received messages. Moreover, we need to differentiate between fresh messages that are generated for each new session and messages that constitute the persistent knowledge of an agent. To handle these issues we define frames and procedures.

During one session of a protocol, an agent A receives a number of messages, we follow the same notation as in [3, 4] (with different interpretation, however) and organize these messages into a frame $\nu\mathbb{C}\,.\,\theta_f$, where $\mathbb{C}$ is a set of names (constants), ν binds the names in $\mathbb{C}$ to the frame, and θ_f is a partial mapping from a finite set $\{0,\ldots,N\}$ to sets of messages. In the frame, $\theta_f(0)$ is a set containing atomic messages that are initially known to A and $\theta_f(i)$ such that $0 < i \leq N$ is the set containing the message expected to be received by A at step i of the protocol. A frame of a certain agent A encapsulates three pieces of information: (1) The set $\mathbb{C}$ which contains names that are newly generated for each protocol session such as nonces, (2) the order in which messages are received in the protocol, and (3) the syntactic structure of these messages. We define a function ϕ (of type Frm) that maps agents names to frames, i.e., $\phi(A)$ is the frame of messages of agent A. For a frame $\phi(A) = \nu\mathbb{C}\,.\,\theta_f$, the function $\mathbf{size}(\phi(A))$ is defined to be $|\mathbf{dom}(\theta_f)|$; the cardinality of the domain of the substitution θ_f. Moreover for any positive integer i, such that $i \leq \mathbf{size}(\phi(A))$, $\mathbf{trunc}(\phi(A), i), i \in \mathbf{dom}(\theta_f)$ is the new frame obtained by restricting the domain of θ_f to the set $\{0,\ldots,i\}$, i.e., it is the frame of A just after the i-th step of the protocol. The function **frame** that parses the protocol and constructs the frames is defined in Table 5.1.

Intuitively, the frame $\phi(A) = \nu\mathbb{C}\,.\,\theta_f$ is constructed as follows:(1) The set of all fresh names for an agent A is added to $\mathbb{C}$, (2) all atomic terms that are initially known to agent A are added to the set $\theta_f(0)$, and (3) for a message m_i received at step i of the protocol run, θ_f is augmented by the mapping $i \mapsto \{m_i\}$. As a consequence, the sets $\mathbb{C}$ and $\theta_f(0)$ are finite, whereas the sets $\theta_f(i)$, $i > 0$ are all singleton. We note

$$
\begin{array}{l}
\mathbf{frame} : \mathcal{P}rot \to \mathsf{Frm} \to \mathsf{Frm} \\
\mathbf{frame}[\![\ \nu A \rhd m.\mathcal{D}.\mathcal{C}\ ]\!](\phi) = \mathbf{frame}[\![\ \mathcal{D}.\mathcal{C}\ ]\!](\phi[A \mapsto \nu\mathbb{C} \cup m \,.\, \theta_f]) \\
\mathbf{frame}[\![\ \kappa A \rhd m.\mathcal{D}.\mathcal{C}\ ]\!](\phi) = \mathbf{frame}[\![\ \mathcal{D}.\mathcal{C}\ ]\!](\phi[A \mapsto \nu\mathbb{C} \,.\, \theta_f[0 \mapsto \theta_f(0) \cup \{m\}]]) \\
\mathbf{frame}[\![\ \epsilon.\mathcal{C}\ ]\!](\phi) = \mathbf{frame}[\![\ \mathcal{C}\ ]\!](\phi) \\
\mathbf{frame}[\![\ \mathtt{step}\ i \rhd B \to A :\ m.\mathcal{C}\ ]\!](\phi) = \mathbf{frame}[\![\ \mathcal{C}\ ]\!](\phi[A \mapsto \nu\mathbb{C} \,.\, \theta_f[i \mapsto \{m\}]]) \\
\mathbf{frame}[\![\ \epsilon\ ]\!](\phi) = \phi
\end{array}
$$

Table 5.1: Construction of a frame.

here that the function **frame** constructs frames from the protocol specification, i.e., we assume that at step i of the protocol the message m_i travels through the network unaltered. This is why we call these frames *specification frames.* To simplify the notation, the message received by an agent A at step l of the protocol is denoted by $r_A(l)$. So, if we assume that messages are unaltered in the network, then, for a frame $\phi(A) = \nu\mathbb{C} \,.\, \theta_f$, we have $\forall l \in \mathbf{dom}(\theta_f) \,.\, l > 0 \Rightarrow r_A(l) \in \theta_f(l)$. This is generally not the case, since messages may be altered by the intruder and we may have $r_A(l) \notin \theta_f(l)$.

We use the Rivest-Shamir-Adleman three pass protocol [99], which we will henceforth call RSA3P, as a running example to illustrate the introduced concepts. The specification of this protocol, in our syntax, is written as:

$$
\begin{array}{l}
\kappa_A \rhd K_A \\
\nu_A \rhd m \\
\kappa_B \rhd K_B \\
\mathtt{step}\ 1 \rhd A \to B : \{m\}_{K_A} \\
\mathtt{step}\ 2 \rhd B \to A : \{\{m\}_{K_A}\}_{K_B} \\
\mathtt{step}\ 3 \rhd A \to B : \{m\}_{K_B}
\end{array}
\tag{5.2}
$$

This protocol uses the commutative property of RSA encryption, which implies that $\{\{m\}_{K_A}\}_{K_B} = \{\{m\}_{K_B}\}_{K_A}$. As it is shown from the knowledge of each agent, A and B share no information about their keys. The signature Σ contains the operation symbols we defined in the previous chapter, i.e., $\{\_\}_\_$, decrypt(_, _) and $(\_)^{-1}$ which represent encryption, decryption and key inverses, respectively. To express properties of the used cryptographic operations, we add the equations: $\{\{x\}_y\}_z = \{\{x\}_z\}_y$ and $\texttt{decrypt}(\{x\}_y, y^{-1}) = x$. The frames of both agents are:

$$\begin{aligned}
&\phi(A) = \nu\{m\}\,.\,[0 \mapsto \{K_A\}, 2 \mapsto \{\{\{m\}_{K_A}\}_{K_B}\}] \\
&\phi(B) = \nu\emptyset\,.\,[0 \mapsto \{K_B\}, 1 \mapsto \{\{m\}_{K_A}\}, 3 \mapsto \{\{m\}_{K_B}\}] \\
&\mathbf{trunc}(\phi(A), 0) = \nu\{m\}\,.\,[0 \mapsto \{K_A\}] \\
&\mathbf{trunc}(\phi(B), 1) = \phi(B) = \nu\emptyset\,.\,[0 \mapsto \{K_B\}, 1 \mapsto \{\{m\}_{K_A}\}]
\end{aligned}$$

Using the simplified notation introduced above, we can write $r_B(3) = \{m\}_{K_B}$. A term $m \in \mathbb{T}_\Sigma$ is deducible by agent A from a frame $\phi(A) = \nu\mathbb{C}\,.\,\theta_f$, written $\phi(A) \vdash m$, if it can be obtained by the application of one or more of the following rules:

$$\begin{array}{lll}
(\mathsf{rcv}) & \dfrac{\cdot}{\phi(A) \vdash m} & \exists i \in dom(\theta_f)\,.\,m \in \theta_f(i) \\[2ex]
(\mathsf{new}) & \dfrac{\cdot}{\phi(A) \vdash m} & m \in \mathbb{C} \\[2ex]
(\mathsf{apl}) & \dfrac{\phi(A) \vdash m_1 \ldots \phi(A) \vdash m_n}{\phi(A) \vdash f(m_1, \ldots m_n)} & \\[2ex]
(\mathsf{eqn}) & \dfrac{\phi(A) \vdash m \quad m =_E m'}{\phi(A) \vdash m'} &
\end{array} \tag{5.3}$$

We define the set $\mathbf{deduce}(\phi(A)) = \{m \in \mathbb{M} \mid \phi(A) \vdash m\}$, this definition is necessary to avoid terms that are not messages, e.g., $\texttt{decrypt(encrypt(}m\texttt{,}K_1\texttt{),}K_2\texttt{)}$. Moreover, the set $\mathbf{deduce}(\phi(A))$ can be expressed as $\bigcup_{d\geq 0} \mathbf{deduce}^d(\phi(A))$, where $\mathbf{deduce}^0(\phi(A))$ will contain all messages in the frame $\phi(A)$. For $d > 0$, the set $\mathbf{deduce}^d(\phi(A))$ is obtained by applying the rules (apl) and (eqn) only once to the messages of $\mathbf{deduce}^{d-1}(\phi(A))$ and adding the generated messages to those already in $\mathbf{deduce}^{d-1}(\phi(A))$. A formal definition is given in (5.4).

$$\begin{aligned}
\textbf{deduce}^0(\phi(A)) &= \textstyle\bigcup_{i\in dom(\theta_f)} \theta_f(i) \cup \mathbb{C} \\
\textbf{deduce}^d(\phi(A)) &= \textbf{deduce}^{d-1}(\phi(A)) \cup \mathbb{X}^d_{\mathbf{apl}} \cup \mathbb{X}^d_{\mathbf{eqn}} \\
\mathbb{X}^d_{\mathbf{apl}} &= \{f(t_1,\ldots,t_n) \mid f \in \Sigma \wedge t_1,\ldots,t_n \in \textbf{deduce}^{d-1}(\phi(A))\} \\
\mathbb{X}^d_{\mathbf{eqn}} &= \{m \mid m =_E m' \wedge m' \in \textbf{deduce}^{d-1}(\phi(A)\}
\end{aligned} \tag{5.4}$$

Constructing Procedures

Now that we have defined frames that represent the history of messages as seen by a certain agent, we can investigate their use in the computation of new messages. For a protocol step "`step` $i \triangleright A \longrightarrow B : m_i$", and given a frame $\phi(A) = \nu\mathbb{C}\,.\,\theta_f$ that belongs to an agent A, we would like to find the abstract computation procedure that describes how A should compute m_i from the frame. In other words, we would like to know the operations that agent A should perform on messages of the frame in order to get m_i. We call this procedure $\mathbf{comp}_i$, and it is written in the $\mathcal{W}$ language described in Section 4.1.3. So, we define $\mathbf{comp}_i$ to be `proc in` α `out` β `aux` γ. As described in the syntax of $\mathcal{W}$, α, β, and γ are three sequences of variables that are the input, output, and auxiliary sequences, respectively. In our case, α depends on the number of messages received by agent A from the network, β contains a single variable o that represents the message to be produced, and γ will generally be the empty sequence ϵ. The reason why α depends on received messages and not on the whole frame (the frame contains fresh messages and initial knowledge in addition to received messages) is that we treat fresh messages and initial knowledge as constants, since they are not manipulated by the intruder like received messages.

Assuming a protocol step "`step` $i \triangleright A \longrightarrow B : m_i$" and a frame $\phi(A) = \nu\mathbb{C}\,.\,\theta_f$, the sequences α, β, and γ of the procedure $\mathbf{comp}_i$ are constructed as follows:

- $\alpha = \alpha_1.\alpha_2\ldots\alpha_n$, each α_i is a variable of sort Msg (defined in equation (4.1)) and the sequence length $n = \mathbf{size}(\phi(A)) - 1$, i.e., the number of received messages in the frame. We recall that $\mathbf{size}(\nu\mathbb{C}.\theta_f) = |\mathbf{dom}(\theta_f)|$, and that $\theta_f(0)$ contains the initial knowledge of the agent. When the procedure is executed each variable α_i will be replaced by the corresponding received message.

- $\beta = o$, o is a variable of sort Msg.
- $\gamma = \epsilon$

The sequence γ is generally empty except in the case where the definition of the procedure $\mathbf{comp}_i$ contains `while` statements that need temporary storage. The procedure $\mathbf{comp}_i$ will have the general form:

```
proc in $\alpha$ out $o$ aux $\epsilon$
begin
$o := t$;
if( not( $t \in \mathbb{M}$)) then $o :=$ terminate else skip;
end
```

In the procedure above, the statements between `begin` and `end` are called the body of the procedure and they may include variables from the sequences α, β, and γ. A *procedure call* for $\mathbf{comp}_i$ is denoted by $\mathbf{comp}_i(\delta)$, where δ is a sequence of messages $\delta_1.\delta_2\ldots\delta_n$ that has the same length as α. Calling the procedure means that each variable α_i in the procedure body (in the term t) will be replaced by a message $\delta_i \in \mathbb{M}$ and the term t assigned to o will be taken as the output of the procedure. When calling the procedure $\mathbf{comp}_i$ the sequence δ of messages is constructed such that δ_i is the i-th message received by A. In other words, δ_1 will be the first message received by A and so on until we have δ_n which is the last message received by A. This is why the number of variables in the sequence α was set to $\mathbf{size}(\phi(A)) - 1$, which is the number of messages received by A, since $\theta_f(0)$ represents initial knowledge and not a received message.

Consider a certain frame $\phi(B) = \nu\emptyset \,.\, [0 \mapsto \{K_B, K_A\}, 1 \mapsto \{\{m\}_{K_A}\}, 3 \mapsto \{\{m\}_{K_B}\}]$, we have $\delta_1 = \{m\}_{K_A}$, and $\delta_2 = \{m\}_{K_B}$, and when calling a procedure where agent B produces a message, each variable α_i in the procedure body, e.g., in the term t, will be replaced by δ_i, and the term t will then be the output of the procedure. Formally, we define the sequence δ to be $\delta = \mathbf{rcvd}(\phi(B))$, where the function **rcvd** maps a frame to a sequence of messages $r_B(l_1).r_B(l_2)\ldots r_B(l_n)$, where $r_B(l_i)$ is the message received by B at step l_i. Also for any sequence δ, we define $\delta|_i$ to be the symbol at position i of the sequence, e.g., $a.b.c|_2 = b$.

As mentioned earlier, having a protocol step "`step` $i \triangleright A \longrightarrow B : m_i$" and a frame $\phi(A) = \nu\mathbb{C} \,.\, \theta_f$, we need to construct the procedure $\mathbf{comp}_i$ that the agent A will use to compute m_i. More specifically, we need to know how to construct the term t. Moreover, in the last statement of the procedure body we check whether o is a valid message or not. The fact that o may not be a message is explained in the next section, when we discuss real frames. In (5.5) below, we provide an algorithm, which we call **prc**, that constructs the term $t \in \mathbb{T}_\Sigma(\mathbb{X})$ of the procedure from the protocol specification of a message m_i.

We assume that the message m_i in the specification `step` $i \triangleright A \longrightarrow B : m_i$ is in the set $\mathbf{deduce}^d(\phi(A))$, the algorithm **prc** that constructs the term t is then defined in pseudocode as:

```
case $d \geq 1$
$\mathbf{prc}(\phi(A), \alpha, m_i, d) =$
  case $m_i \in \mathbb{X}^d_{\mathbf{apl}}$
    return $f^T(\mathbf{prc}(\phi(A), \alpha, t_1, d-1), \ldots, \mathbf{prc}(\phi(A), t_n, d-1))$
    where $m_i = \mathtt{f}(t_1, \ldots, t_n)$
  case $m_i \in \mathbb{X}^d_{\mathbf{eqn}}$
    return $\mathbf{prc}(\phi(A), \alpha, m', d-1)$
    where $m' =_E m_i$
  case $m_i \in \mathbf{deduce}^{d-1}(\phi(A))$                                   (5.5)
    return $\mathbf{prc}(\phi(A), \alpha, m_i, d-1)$
$\mathbf{prc}(\phi(A), \alpha, m_i, 0) =$
  case $m_i = n, n \in \theta_f(0)$
    return $n$
  case $m_i \in \theta_f(l), l > 0$
    return $\alpha_j$
    where $m_i = \mathbf{rcvd}(\phi(A))|_j$
  case $m_i = c, c \in \mathbb{C}$
    return $c$
```

The algorithm has four inputs: The frame $\phi(A)$, the sequence α of variables of $\mathbf{comp}_i$, the message m_i that we would like to compute, and the index d. It is defined recursively where we have three base cases (the bottom three case statements) and three recursive calls (the top three case statements). We notice that in the recursive cases the value of d is decreased by 1 in the recursive calls, and if the value of d is 0, we will be in one of the base cases, which ensures termination. Hereafter, we list the cases with the same order in which they appear in the algorithm definition:

- In the first case, the message m_i that we would like to compute has the form

$\texttt{f}(t_1, \ldots, t_n)$. So, in order to compute it, we have to know how to compute each message (term) t_i from the frame and then apply the function $f^{\mathcal{T}}$ of the term algebra. We note here the difference between $\texttt{f}$ as an operation symbol, i.e., syntax, and $f^{\mathcal{T}}$ which is the function assigned to $\texttt{f}$ in the term algebra $\mathcal{T}$.

- In the second case, we cannot find a message with the exact syntax of m_i, but we find another one m' such that $m' =_E m_i$. So now we have to know how to compute m', which justifies the recursive call.

- In the third case, we discover that $m_i \in \mathbf{deduce}^{d-1}$, where $\mathbf{deduce}^{d}(\phi(A)) = \mathbf{deduce}^{d-1}(\phi(A)) \cup \mathbb{X}^{d}_{\mathbf{apl}} \cup \mathbb{X}^{d}_{\mathbf{eqn}}$. So the message in the argument of the recursive call will still be m_i, whereas the value of d is decreased by 1.

- In the fourth case, the value of d has reached 0, where $\mathbf{deduce}^{0}(\phi(A)) = \bigcup_{i \in dom(\theta_f)} \theta_f(i) \cup \mathbb{C}$, which means that the message we are looking for is a fresh message, a message in the initial knowledge of the agent, or a received message. In this particular case, the message is in the initial knowledge, i.e, the set $\theta_f(0)$, so we just return the message.

- In the fifth case, like the fourth, the message is in the frame. However, it has been received at step l, i.e., it is in the singleton set $\theta_f(l)$. So we return the variable α_j, where the message m_i is the j-th message received by the agent. The variable α_j is returned instead of the message itself because this is a received message which could have been altered by the intruder. When we call the procedure $\mathbf{comp}_i$, this variable will be replaced by the message $\mathbf{rcvd}(\phi(A))|_j$, which depends on the frame. We will discuss this point further in the next section, when we discuss real frames.

- In the sixth case, the message is in the frame and it is a member in the set $\mathbb{C}$ of fresh messages so we just return it.

For a frame $\phi(A) = \nu\mathbb{C} \centerdot \theta_f$, we say that the output $m_i = \mathbf{comp}_i(\phi(A))$ is feasible if $\forall m \preccurlyeq m_i \centerdot \mathbf{trunc}(\phi, i) \vdash m$, where $\preccurlyeq$ is the subterm relation defined over $\mathbb{T}_\Sigma(\mathbb{X})$, i.e., $x \preccurlyeq y$ if $x = y$ or $y = f(t_1, \ldots t_n) \wedge \exists i \centerdot x \preccurlyeq t_i$. In other words, the algorithm constructing the output message is feasible if it constructs the message that is supposed to be sent by A at step i of the protocol using only the initial knowledge of A and the messages received by A up to step i. This condition of feasibility is a simple and preliminary check on the well-formedness of protocol specifications.

As an example, we use the RSA3P protocol introduced in Equation (5.2). We recall the protocol steps below.

$$\begin{aligned}
&\texttt{step } 1 \rhd A \rightarrow B : \{m\}_{K_A}\\
&\texttt{step } 2 \rhd B \rightarrow A : \{\{m\}_{K_A}\}_{K_B}\\
&\texttt{step } 3 \rhd A \rightarrow B : \{m\}_{K_B}
\end{aligned}$$

We would like to construct the abstract computation procedure that agent A follows at step 3 of the protocol in order to produce the message $\{m\}_{K_B}$ that is then sent to the network to be delivered to B. We begin by constructing the sets $\mathbf{deduce}^i(\phi(A))$:

$$\begin{aligned}
\mathbf{deduce}^0(\phi(A)) = \;& \{m, K_A, \{\{m\}_{K_A}\}_{K_B}\}\\
\mathbf{deduce}^1(\phi(A)) = \;& \mathbf{deduce}^0(\phi(A)) \cup \{K_A^{-1}, \texttt{decrypt}(\{\{m\}_{K_A}\}_{K_B}, K_A), \ldots\}\\
& \cup\{\{\{m\}_{K_B}\}_{K_A}\}\\
\mathbf{deduce}^2(\phi(A)) = \;& \mathbf{deduce}^1(\phi(A)) \cup \{\texttt{decrypt}(\{\{m\}_{K_B}\}_{K_A}, K_A^{-1}), \ldots\}\\
& \cup\{\ldots\}\\
\mathbf{deduce}^3(\phi(A)) = \;& \mathbf{deduce}^2(\phi(A)) \cup \{\{m\}_{K_B}, \ldots\} \cup \{\ldots\}
\end{aligned}$$

We note that the set $\mathbf{deduce}^0(\phi(A))$ contains all messages in the frame. The set $\mathbf{deduce}^1(\phi(A))$ contains all messages of $\mathbf{deduce}^0(\phi(A))$ in addition to two sets of messages. The first one is $\mathbb{X}^1_{\mathbf{apl}}$ and the second one is $\mathbb{X}^1_{\mathbf{eqn}}$, as defined in (5.4). We construct the rest of the sets similarly.

Now we want to know how agent A is able to compute the message he is sending in step $i = 3$ as a function from the knowledge he gathered thus far. In other words, we want to get $\mathbf{comp}_3 = \texttt{proc in } \alpha_1 \texttt{ out } o \texttt{ aux } \epsilon$, where the sequence α is composed of only one variable α_1 since A receives only one message before step 3. The body of the procedure $\mathbf{comp}_3$ will contain the statement $o := t$ as mentioned earlier and to know how to compute t we apply the algorithm **prc**. The message sent by A at step 3 is $\{m\}_{K_B}$, and $\{m\}_{K_B} \in \mathbf{deduce}^3(\phi(A))$, i.e., $d = 3$, so we have:

$$\begin{aligned}
&\mathbf{prc}(\phi(A), \alpha_1, \{m\}_{K_B}, 3)\\
&\quad= \mathbf{prc}(\phi(A), \alpha_1, dec(\{\{m\}_{K_B}\}_{K_A}, K_A^{-1}, 2)\\
&\quad= \texttt{decrypt}(\mathbf{prc}(\phi(A), \alpha_1, \{\{m\}_{K_B}\}_{K_A}, 1), \mathbf{prc}(\phi(A), \alpha_1, K_A^{-1}, 1))\\
&\quad= \texttt{decrypt}(\mathbf{prc}(\phi(A), \alpha_1, \{\{m\}_{K_A}\}_{K_B}, 0), (\mathbf{prc}(\phi(A), \alpha_1, K_A, 0))^{-1})\\
&\quad= \texttt{decrypt}(\alpha_1, K_A^{-1})
\end{aligned}$$

The last line means that in order to obtain the message that should be sent at step 3 by agent A, then A must decrypt the message he received at step 2 by the inverse of his key K_A. Therefore, the intruder can intercept the message $\{m\}_{K_A}$ at step 1, send it back to A at step 3 (masquerading as B) and obtain the secret message m. The intruder may skip the execution of step 2 by not sending $\{m\}_{K_A}$ to B since it does not contribute to the success of the attack.

The algorithm **prc** enables us to automatically deduce, from the protocol syntax, the computation procedure that should be performed to compute each sent message. This enables us to have a simple syntax that is familiar to cryptographers. Moreover, since the procedure uses abstract computation in an algebraic framework, it is easily integrated with the algebraic model of messages that we adopt.

Real Frames and Agents Responses

The intruder's behavior is considered non-deterministic in the sense that, at any point in time, we cannot tell exactly which message it is going to send. Agents on the other hand have a deterministic behavior; the message an agent A sends at any step i is determined by $\mathbf{comp}_i(\mathbf{rcvd}\phi(A))$, i.e., the message is dependent on the frame of messages seen by A and the step number. The specification frame $\phi(A)$ is constructed based on the protocol specification, and the underlying assumption is that the protocol is executed exactly as specified. This is not the actual case however, since we assume the presence of an intruder and/or dishonest agents. We, therefore, define a real frame $\rho(s, A) = \nu\mathbb{C}\,.\,\theta_f$ of type RFrm, where s is the sequence of messages that were exchanged over the network up to current time. The set $\mathbb{C}$, as in $\phi(A)$, contains all fresh values generated by A, whereas θ_f describes s as seen by A. For instance, assuming m appears in s then $\theta_f(i) = \{m\}$ means that agent A received message m at step i of the protocol session in which A is participating. We write this as $\rho(s, A) = \nu\mathbb{C}\,.\,\theta_f[i \mapsto \{m\}]$, we may also abbreviate it by writing $r_A(s, i) = m$. It is worth noting that, for agents, each protocol session has its own real frame.

The difference between the real frame $\rho(s, A)$ and the specification frame $\phi(A)$ of an agent A lies in the substitution θ_f. Both frames agree at $\theta_f(0)$ since it contains the initial knowledge. At any other $i > 0$, $\theta_f(i)$ of the specification frame is the singleton set that contains the message specified to be received by the agent at step i. However, for an actual frame, the set $\theta_f(i)$ contains the message that was actually received from the network at step i during an actual session of the protocol execution. This message could have been manipulated by the intruder. Consequently, for a protocol step specification $\mathtt{step}\ i \triangleright A \rightarrow B\ :\ m_i$, the output of a procedure call $\mathbf{comp}_i(\mathbf{rcvd}(\phi(A))$ will be the message m_i. However, the output of the procedure call $\mathbf{comp}_i(\mathbf{rcvd}(\rho(s, A))$, will be either the message m_i, another message m, or the special message `terminate`. We note here that although the procedure $\mathbf{comp}_i$ was derived using $\phi(A)$ (through the use of the algorithm $\mathbf{prc}$), we were able to use $\rho(s, A)$ in the place of $\phi(A)$ since both of them have the same structure $\nu\mathbb{C}\,.\,\theta_f$. The case where the output of the procedure $\mathbf{comp}_i$ is `terminate`, i.e., the output is not a valid message, represents situations where the received message is corrupt or where the intruder is mounting an unsuccessful attack. Furthermore, our model can handle the case of a dishonest agent D who does not follow protocol rules and the messages it sends can then be any message in $\mathbf{deduce}(\rho(s, D))$.

Intruder Frames

An intruder frame is always an actual frame, where $\rho(s, I) = \nu\mathbb{C} \,.\, \theta_f$. In this case, $\mathbb{C}$ is a set containing fresh values created by the intruder up to the current moment. The substitution θ_f in this case maps the set $\{0, 1, \ldots, N\}$ to messages in $\mathbb{M}$, where $\theta_f(0)$ contains all messages initially known to I such as public keys. Moreover, the statement $m \in \theta_f(i)$ means that message m was received by the intruder at step i of a certain protocol session. Since we consider multi-session attacks, the intruder has only one frame which spans over multiple sessions and therefore can represent the whole knowledge of the intruder. Inference rules for the intruder are similar to those defined in (5.3) above for agents, the difference is that $\phi(A)$ should be replaced by $\rho(s, I)$. Also, we write $\rho(s, I) \vdash m$ to indicate that message m can be deduced from the frame $\rho(s, I)$.

5.2 Games for Security Protocols

In this section, we present game definitions for security protocols. A protocol is represented as a game between the intruder and agents, where the type of the game depends on the number of sessions and the assumption about intruder behavior which can be passive or active. A passive intruder just forwards messages between agents, whereas an active one may alter or block messages. A play of the game is a sequence of moves that begins by a move of the intruder then an agent and then moves alternate between the intruder and agents. All possible plays of the game form the game tree. When defining games, we define the set of moves, the labeling functions, the enabling relation and finally the game tree. We begin by defining simple games, and show how to compose them into more complicated ones.

Namely, we define four games:

- A one-session game with a passive intruder.
- A multiple-session game with a passive intruder.
- A one-session game with an active intruder.
- A multiple-session game with an active intruder.

We call the first two cases the functional view of protocols and the latter two the security view.

5.2.1 Definition of Games

When defining games for security protocols, each move m in the game represents a message sent from an agent to another. As mentioned earlier the set of messages is $\mathbb{M} \subset \mathbb{T}_\Sigma$. The generation of a single message by an honest agent is represented as a strategy over the game Msg which is defined as follows:

$$\begin{array}{rcllccl}
M_{\mathsf{Msg}} & = & \{q\} \cup \mathbb{M} & & & & \\
\lambda(q) & = & OQ & & \star & \rightsquigarrow & q \\
\lambda(m) & = & PA & m \in \mathbb{M} & q & \rightsquigarrow & m \quad m \in \mathbb{M} \\
P_{\mathsf{Msg}} & = & \{q.m \mid m \in \mathbb{M}\} & & & &
\end{array}$$

Here, we assume that any $m \in \mathbb{M}$ can be played at any time. This is not an accurate assumption, as any principal in the protocol can play only those messages m that they are able to construct, i.e., $\phi(A) \vdash m$. A single communication step, on the other hand, is represented by a strategy over the game $\mathsf{Csg} = \mathsf{Msg} \multimap \mathsf{Msg}$. The formal definition of the game is given hereafter:

$$
\begin{aligned}
M_{\mathsf{Csg}} &= \{q^1\} \cup \{q^2\} \cup \{m^i \mid m \in \mathbb{M}, i \in \{1,2\}\} \\
\lambda_{\mathsf{Csg}}(q^i) &= \begin{cases} PQ & i = 1 \\ OQ & i = 2 \end{cases} \\
\lambda_{\mathsf{Csg}}(m^i) &= \begin{cases} OA & m \in \mathbb{M} \wedge i = 1 \\ PA & m \in \mathbb{M} \wedge i = 2 \end{cases}
\end{aligned}
$$

The following enabling relation is defined over M_{Csg}:

$$
\begin{array}{rcll}
\star & \rightsquigarrow_{\mathsf{Csg}} & q^2 & \\
q^2 & \rightsquigarrow_{\mathsf{Csg}} & q^1 & \\
q^1 & \rightsquigarrow_{\mathsf{Csg}} & m^1 & m \in \mathbb{M} \\
m^1 & \rightsquigarrow_{\mathsf{Csg}} & n^2 & m, n \in \mathbb{M}
\end{array} \tag{5.6}
$$

The enabling relation affects the game tree:
$P_{\mathsf{Csg}} = \{\epsilon\} \cup \{q^2, q^2.q^1\} \cup \{q^2.q^1.m^1 \mid m \in \mathbb{M}\} \cup \{q^2.q^1.m^1.n^2 \mid m, n \in \mathbb{M}\}$

We notice here that we used a superscript to differentiate between moves in each game, since the set of moves is the *disjoint* union of the sets of the individual games. We used the superscript 1 to denote moves of the game to the left of $\multimap$, and the superscript 2 for the other game. This is equivalent to denoting the games as $\mathsf{Msg}^1 \multimap \mathsf{Msg}^2$. The definition of the enabling relation in (5.6) makes sure that n^2 cannot be played unless m^1 is played first (any sequence in the game tree will be the prefix of a sequence in the form $q^2.q^1.m^1.n^2$). This results from the fact that we assumed that in any communication step (Csg game) an agent (the proponent) only sends a message in response to a message that it received from the intruder (the opponent). We clarify these ideas by taking, as example, the RSA3P protocol:

$$\begin{array}{lrcl} \text{step 1.} & A \rightarrow B & : & \{m\}_{K_A} \\ \text{step 2.} & B \rightarrow A & : & \{\{m\}_{K_A}\}_{K_B} \\ \text{step 3.} & A \rightarrow B & : & \{m\}_{K_B} \end{array}$$

Examining step 1, we notice that A initiates the protocol. Since, in game semantics, O (the channel) always plays first, we assume A gets a start message "`start`" from the channel and replies with $\{m\}_{K_A}$. The "`start`" message serves as an action to begin the execution of the protocol. In step 2, B receives the message $\{m\}_{K_A}$ from the channel and replies with $\{\{m\}_{K_A}\}_{K_B}$, and the last step follows. We rewrite the steps as:

$$\begin{array}{lrcl} \text{step 1.} & I \rightarrow A & : & \texttt{start} \\ & A \rightarrow I & : & \{m\}_{K_A} \\ \text{step 2.} & I \rightarrow B & : & \{m\}_{K_A} \\ & B \rightarrow I & : & \{\{m\}_{K_A}\}_{K_B} \\ \text{step 3.} & I \rightarrow A & : & \{\{m\}_{K_A}\}_{K_B} \\ & A \rightarrow I & : & \{m\}_{K_B} \\ \text{step 4.} & I \rightarrow B & : & \{m\}_{K_B} \\ & B \rightarrow I & : & \texttt{terminate} \end{array} \tag{5.7}$$

The protocol description in (5.7) makes clear the interaction with the intruder I; an agent sends a message only in response to a message received from the intruder. Each communication step has the form: $I \rightarrow X : m_i$ followed by $X \rightarrow I : m_j$ where X is an honest agent. To respect the notation, a protocol will always end by an agent X sending a `terminate` message to the intruder. The `terminate` message marks the end of the execution of one protocol session. A certain communication step in a protocol can then be captured as a strategy σ over the game Csg. In this strategy, O starts by asking for a message, P replies by asking for another message.

Then, once P receives a message from O it will reply by its own message, which will depend on P's frame.

An example is given below in Figure 5.1, for a communication step i, where the intruder sends message m to an agent A. The agent then executes the procedure $\mathbf{comp}_i(\mathbf{rcvd}(\rho(s,A)))$ and sends its output to the intruder (the channel). So, the semantics of communication step i of a protocol is a strategy σ over the game Csg. The sequences of σ all have the form $q^2.q^1.m^1.n^2$. The intruder message m is any message such that $\rho(s,I) \vdash m$, and the agent's message $n = \mathbf{comp}_i(\mathbf{rcvd}(\rho(s,A)))$, where s is the sequence of messages exchanged during protocol execution up to the current moment. More formally, we write the strategy as $\sigma = \{q^2.q^1.m^1.n^2 \mid m,n \in \mathbb{M} \wedge \rho(s,I) \vdash m \wedge n = \mathbf{comp}_i(\mathbf{rcvd}(\rho(s,A)))\}$. We note again that the superscripts are not meant as exponents but as a identifiers to know the game in which the move is played.

$$\begin{array}{ccccc}
\mathsf{Msg}^1 & \multimap & \mathsf{Msg}^2 & & \\
 & & q^2 & & O \\
q^1 & & & & A \\
m^1 & & & & O \\
 & & n^2 & & A \\
\multicolumn{3}{l}{n = \mathbf{comp}_i(\mathbf{rcvd}\rho(s,A))} & &
\end{array}$$

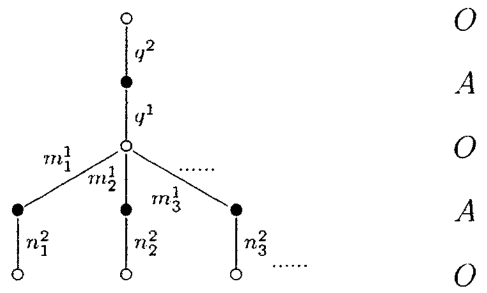


Figure 5.1: Game for single protocol step.

The execution of a number of steps in succession can be represented by a strategy over the tensor product of a number of Csg games. As an example, Step

1 and Step 2 in (5.7) can be represented as a strategy over the game $(\mathsf{Msg}^{1,1} \multimap \mathsf{Msg}^{1,2}) \otimes (\mathsf{Msg}^{2,1} \multimap \mathsf{Msg}^{2,2})$. Here, we modified the superscripts to identify different copies of the Msg game. The superscript is now a tuple (j, k), where j indicates the step and k indicates a specific Msg game within a step. The tensor product, however, does not specify which game is played first (i.e., we can start by playing the game of Step 2). This is why we need the enabling relation to specify the order of moves. Actually, this is what we are going to use when we begin the discussion about Protocol Session Games (Psg games).

Single Protocol Session (Functional View)

The functional view of a security protocol describes how the protocol executes with the existence of a passive intruder. For any protocol with $N-1$ communication steps, the Protocol Session Game $\mathsf{Psg}_{[\mathsf{N}]}$ is defined by: $\mathsf{Psg}_{[\mathsf{N}]} \overset{def}{=} !_N\mathsf{Csg}$, which is the tensor product of N copies of the Csg game under the condition that no play can take place in the i^{th} copy unless a play was made in the $(i-1)^{th}$ copy. This insures that the protocol steps are executed in order. We have N copies of the Csg game for a protocol with $N-1$ steps because of the addition of the `start` and `terminate` messages as shown in (5.7). The previous discussion is summarized by the following equations:

$$\begin{aligned} \mathsf{Psg}_{[\mathsf{N}]} &= !_N\mathsf{Csg} \\ !_N\mathsf{Csg} &= \mathsf{Csg}^1 \otimes \mathsf{Csg}^2 \ldots \otimes \mathsf{Csg}^n \qquad \forall i' > i \,.\, \mathsf{Csg}^{i'} \text{ is started after } \mathsf{Csg}^i \\ &= (\mathsf{Msg}^{1,1} \multimap \mathsf{Msg}^{1,2}) \otimes (\mathsf{Msg}^{2,1} \multimap \mathsf{Msg}^{2,2}) \ldots \otimes (\mathsf{Msg}^{N,1} \multimap \mathsf{Msg}^{N,2}) \end{aligned} \tag{5.8}$$

The symbol P in game semantics represents the system, while O represents the environment. In the Csg game, P represents an honest agent, while O represents the intruder (the network). When multiple Csg games are being played such as in a Psg game, O is the same in all of these games, whereas P is a certain agent. To clarify this ambiguity we define the mapping $\mathbf{Id} : \mathrm{copy} \to \mathbb{P}$, where $\mathbb{P}$ is a countable set of symbols representing agents identities and "copy" is an index, e.g., a tuple of natural numbers, that is the superscript identifying the copy of the Csg game being played. For instance in the RSA3P protocol in (5.7) above, $\mathbf{Id}(2) = B$ since the game Csg^2 is played between I and B. By abuse of notation, we will write $m^{i,r} \in \mathsf{Msg}^{i,r}$ instead of $m^{i,r} \in M_{\mathsf{Msg}^{i,r}}$to indicate that the move m is played in copy i, r of Msg. Moreover, we define $\vartheta(m^{i,r})$ to be the move without the "copy"superscript; $\vartheta(m^{\mathrm{copy}}) = m$. The definitions for the $\mathsf{Psg}_{[\mathsf{N}]}$ game are shown in (5.9)

$$\begin{aligned} M_{\mathsf{Psg}_{[\mathsf{N}]}} &= \{q^{i,1} \mid i \in \{1, \ldots, N\}\} \cup \\ &\quad \{q^{i,2} \mid i \in \{1, \ldots, N\}\} \cup \\ &\quad \{m^{i,1} \mid m \in \mathbb{M}, i \in \{1, \ldots, N\}\} \cup \\ &\quad \{m^{i,2} \mid m \in \mathbb{M}, i \in \{1, \ldots, N\}\} \\ \lambda_{\mathsf{Psg}_{[\mathsf{N}]}}(q^{i,r}) &= \begin{cases} PQ & r = 1 \\ OQ & r = 2 \end{cases} \\ \lambda_{\mathsf{Psg}_{[\mathsf{N}]}}(m^{i,r}) &= \begin{cases} OA & m \in \mathbb{M} \wedge r = 1 \\ PA & m \in \mathbb{M} \wedge r = 2 \end{cases} \end{aligned} \tag{5.9}$$

The enabling relation is defined in (5.10).

$$\begin{array}{rlll} \star & \rightsquigarrow_{\mathsf{Psg}_{[\mathsf{N}]}} & q^{1,2} & \\ m^{i-1,2} & \rightsquigarrow_{\mathsf{Psg}_{[\mathsf{N}]}} & q^{i,2} & m \in \mathbb{M} \quad i \in \{2, \ldots, N\} \\ q^{i,2} & \rightsquigarrow_{\mathsf{Psg}_{[\mathsf{N}]}} & q^{i,1} & i \in \{1, \ldots, N\} \\ q^{1,1} & \rightsquigarrow_{\mathsf{Psg}_{[\mathsf{N}]}} & \mathtt{start} & \\ m^{N,1} & \rightsquigarrow_{\mathsf{Psg}_{[\mathsf{N}]}} & \mathtt{terminate} & m \in \mathbb{M} \\ q^{i,1} & \rightsquigarrow_{\mathsf{Psg}_{[\mathsf{N}]}} & m^{i,1} & i \in \{2, \ldots N\} \wedge \\ & & & m \in \mathbb{M} \wedge m = \vartheta(n^{i-1,2}) \\ m^{i,1} & \rightsquigarrow_{\mathsf{Psg}_{[\mathsf{N}]}} & n^{i,2} & m, n \in \mathbb{M} \wedge i \in \{1, \ldots, N-1\} \end{array} \tag{5.10}$$

The game tree $P_{\mathsf{Psg}_{[N]}}$ is the set of sequences, that is a subset of $M^{alt}_{\mathsf{Psg}_{[N]}}$ (defined in Section 2.2.1), each sequence satisfies the switching condition, and the enabling relation. In the enabling relation, the first rule states that the game opens with a move by the opponent in Csg^1. Since Csg ends with a move by the proponent in Msg^2 ($m^{i,2}$), the second rule results in that the Csg games are played one after the other (according to the order of the communication steps). The third rule implies the switching conditions between Msg^2 and Msg^1, while the fourth and fifth rules are special for the `start` and `terminate` messages respectively. The sixth rule states that questions enable answers in $\mathsf{Msg}^{i,1}$, in any step, and that the message in these intruder answers, i.e., m equals the message received by the intruder in the previous communication step, i.e., the message $\vartheta(n^{i-1,2})$. This is because in the functional description of the protocol, we assume a passive intruder that just forwards messages between different agents. The seventh rule states that $n^{i,2}$ cannot be played unless $m^{i,1}$ was played first (an agent does not send a message to the intruder unless it received a message from the intruder in return).

The way the $\mathsf{Psg}_{[N]}$ game is defined means that the play proceeds in Csg^1, followed by Csg^2 and so on till we reach Csg^N where the play ends with the terminating move. An illustration of the $\mathsf{Psg}_{[N]}$ game is shown in Figure 5.2 to clarify the notation. In this figure, the intruder forwards the message m_2 (that he obtained in Step 1) to the player in Step 2, and the game continues. We notice that the move is the same, the change in the superscript is to denote that this move is played in two different copies of the the Msg game.

A protocol, as we describe it, is a number of interactions between the intruder (opponent) and honest entities (proponents). For instance, the RSA3P protocol in (5.7) shows two honest entities A, and B. Each of these honest entities describes a certain *role*. An agent is an instance of a certain role. For example, role A can be played by agents M and Q running on any computers in the network. To differentiate between roles and agents, roles are written as A, B, S, etc. while agents

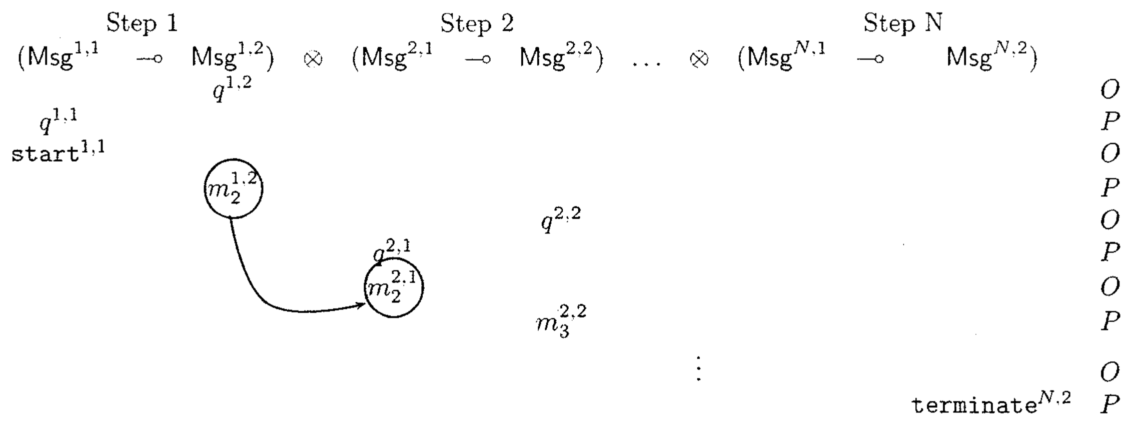


Figure 5.2: Game representation of one protocol session.

are given subscripts A_1, A_2, etc. and B_1, B_2, etc. For any communication step Csg^i, the opponent O is always the intruder, while the player P is an agent playing a certain role.

Multiple Protocol Sessions (Functional View)

Running multiple sessions of the protocol can be represented by the game $!\mathsf{Psg}_{[N]}$, which is the tensor product of an unbounded number of $\mathsf{Psg}_{[N]}$ games. In this case, a certain run of a protocol $\wp$ can be represented by a sequence of moves over the game $\mathsf{Prt}_{[N]}$, where $\mathsf{Prt}_{[N]}$ is defined as follows:

$$\begin{aligned}
\mathsf{Prt}_{[N]} &= !\mathsf{Psg}_{[N]} \\
!\mathsf{Psg}_{[N]} &= \mathsf{Psg}^1_{[N]} \otimes \mathsf{Psg}^2_{[N]} \ldots \quad \forall j' > j \,\bullet\, \mathsf{Psg}^{j'}_{[N]} \text{ is started after } \mathsf{Psg}^j_{[N]} \\
&= \mathsf{Csg}^{1,1} \ldots \otimes \mathsf{Csg}^{1,N} \otimes \mathsf{Csg}^{2,1} \ldots \otimes \mathsf{Csg}^{2,N} \ldots \\
&= (\mathsf{Msg}^{1,1,1} \multimap \mathsf{Msg}^{1,1,2}) \ldots \qquad \otimes (\mathsf{Msg}^{1,N,1} \multimap \mathsf{Msg}^{1,N,2}) \otimes \\
&\quad\; (\mathsf{Msg}^{2,1,1} \multimap \mathsf{Msg}^{2,1,2}) \ldots \qquad \otimes (\mathsf{Msg}^{2,N,1} \multimap \mathsf{Msg}^{2,N,2}) \otimes \\
&\quad\; \ldots
\end{aligned} \tag{5.11}$$

The symbols are read as follows: $\mathsf{Psg}^i_{[N]}$ is the i^{th} copy of the $\mathsf{Psg}_{[N]}$ game (played in the i^{th} session of the protocol), $\mathsf{Csg}^{i,j}$ is the game played in the j^{th} communication step of the i^{th} session, $\mathsf{Msg}^{i,j,r}$ is the r^{th} copy of Msg played in the j^{th} communication step of the i^{th} session, and finally $m^{i,j,r}$ is a certain move m played in $\mathsf{Msg}^{i,j,r}$. For a certain protocol of $N-1$ communication steps, running M sessions of this protocol will result in the variables i, j, and r ranging over the values $\{1,\ldots,M\}$, $\{1,\ldots,N\}$, and $\{1,2\}$ respectively.

$$\begin{aligned}
M_{\mathsf{Prt}_{[N]}} &= \{q^{i,j,1} \mid i \in \mathbb{N}\setminus\{0\}, j \in \{1,\ldots,N\}\}\cup \\
&\quad \{q^{i,j,2} \mid i \in \mathbb{N}\setminus\{0\}, j \in \{1,\ldots,N\}\}\cup \\
&\quad \{m^{i,j,1} \mid m \in \mathbb{M}, i \in \mathbb{N}\setminus\{0\}, j \in \{1,\ldots,N\}\}\cup \\
&\quad \{m^{i,j,2} \mid m \in \mathbb{M}, i \in \mathbb{N}\setminus\{0\}, j \in \{1,\ldots,N\}\} \\
\lambda_{\mathsf{Prt}_{[N]}}(q^{i,j,r}) &= \begin{cases} PQ & r=1 \\ OQ & r=2 \end{cases} \\
\lambda_{\mathsf{Prt}_{[N]}}(m^{i,j,r}) &= \begin{cases} OA & m \in \mathbb{M} \wedge r=1 \\ PA & m \in \mathbb{M} \wedge r=2 \end{cases}
\end{aligned} \tag{5.12}$$

In the definition above, $\mathbb{N}\setminus\{0\}$ is the set of positive integers. The enabling relation is defined as follows:

$$
\begin{array}{rlll}
\star & \rightsquigarrow_{\mathsf{Prt}_{[N]}} & q^{1,1,2} & \\
m^{i,(j-1),2} & \rightsquigarrow_{\mathsf{Prt}_{[N]}} & q^{i,j,2} & m \in \mathbb{M} \\
q^{i,j,2} & \rightsquigarrow_{\mathsf{Prt}_{[N]}} & q^{i,j,1} & i \in \mathbb{N} \setminus \{0\} \wedge j \in \{1, \dots N\} \\
q^{i,1,1} & \rightsquigarrow_{\mathsf{Prt}_{[N]}} & \texttt{start} & i \in \mathbb{N} \setminus \{0\} \\
m^{i,N,1} & \rightsquigarrow_{\mathsf{Prt}_{[N]}} & \texttt{terminate} & m \in \mathbb{M} \wedge i \in \mathbb{N} \setminus \{0\} \\
q^{i,j,1} & \rightsquigarrow_{\mathsf{Prt}_{[N]}} & m^{i,j,1} & i \in \mathbb{N} \setminus \{0\} \wedge j \in \{2, \dots N\} \wedge \\
 & & & m \in \mathbb{M} \wedge m = \vartheta(n^{i,j-1,2}) \\
m^{i,j,1} & \rightsquigarrow_{\mathsf{Prt}_{[N]}} & n^{i,j,2} & m, n \in \mathbb{M} \wedge i \in \mathbb{N} \setminus \{0\} \wedge j \in \{1, \dots N-1\} \\
m^{i-1,1,2} & \rightsquigarrow_{\mathsf{Prt}_{[N]}} & q^{i,1,2} & m \in \mathbb{M} \wedge i \in \{2, \dots M\}
\end{array} \tag{5.13}
$$

In the enabling relation above, the first rule states that the initial move of the $\mathsf{Prt}_{[N]}$ game is by the opponent in Csg^1 (communication step 1) of session 1. The second to the seventh rules control moves within a certain protocol session i and were explained for the game $\mathsf{Psg}_{[N]}$. The eighth rule states that finishing the first communication step of a certain session enables the first communication step of the next session. The game tree $P_{\mathsf{Prt}_{[N]}}$ is defined similarly to the game tree of the $\mathsf{Psg}_{[N]}$ game.

Single Protocol Session (Security View)

In the security view of protocols, we consider different manipulations that can be done by the intruder to messages in order to execute an attack. To describe the security semantics of a protocol we define the the game $\mathsf{Ssg}_{[N]}$, which describes a single session of a protocol with $N - 1$ communication steps assuming intruder manipulations.

$$\begin{aligned}\mathsf{Ssg}_{[\mathsf{N}]} &= \otimes_N \mathsf{Csg}\\ &= \mathsf{Csg}^1 \otimes \mathsf{Csg}^2 \ldots \otimes \mathsf{Csg}^N\\ &= (\mathsf{Msg}^{1,1} \multimap \mathsf{Msg}^{1,2}) \otimes (\mathsf{Msg}^{2,1} \multimap \mathsf{Msg}^{2,2}) \ldots \otimes (\mathsf{Msg}^{N,1} \multimap \mathsf{Msg}^{N,2})\end{aligned} \tag{5.14}$$

The difference between $\otimes_N\mathsf{Csg}$ and $!_N\mathsf{Csg}$ is that in the former we drop the condition that the play in Csg^i has to be started directly after Csg^{i-1}. This is because the intruder might skip steps of the protocol while executing an attack. The only restriction in the order of the moves of the intruder is that communication steps played between the intruder and a certain agent have to be in the order expected by that agent. In (5.15), the enabling relation is changed to reflect these facts about the order of moves. The game $\mathsf{Ssg}_{[\mathsf{N}]}$ has the following enabling relation:

$$\begin{array}{rlll} \star & \rightsquigarrow_{\mathsf{Ssg}_{[\mathsf{N}]}} & q^{t,2} & \forall t' \in \{1,\ldots,N\}\,.\,\mathbf{Id}(t) = \mathbf{Id}(t') \Rightarrow t \leq t' \\ q^{i,2} & \rightsquigarrow_{\mathsf{Ssg}_{[\mathsf{N}]}} & q^{i,1} & i \in \{1,\ldots N\} \\ q^{1,1} & \rightsquigarrow_{\mathsf{Ssg}_{[\mathsf{N}]}} & \mathtt{start} & \\ m^{i,2} & \rightsquigarrow_{\mathsf{Ssg}_{[\mathsf{N}]}} & q^{j,2} & m \in \mathbb{M} \wedge i < j \wedge \mathbf{Id}(i) = \mathbf{Id}(j) \\ m^{N,1} & \rightsquigarrow_{\mathsf{Ssg}_{[\mathsf{N}]}} & \mathtt{terminate} & m \in \mathbb{M} \\ q^{i,1} & \rightsquigarrow_{\mathsf{Ssg}_{[\mathsf{N}]}} & m^{i,1} & m \in \mathbb{M} \wedge i \in \{2,\ldots N\} \\ m^{i,1} & \rightsquigarrow_{\mathsf{Ssg}_{[\mathsf{N}]}} & n^{i,2} & m,n \in \mathbb{M} \wedge i \in \{1,\ldots N-1\} \end{array} \tag{5.15}$$

The modifications to the enabling relation affects the game tree of $\mathsf{Ssg}_{[\mathsf{N}]}$. Another major difference from the definition of the enabling relation of $\mathsf{Psg}_{[\mathsf{N}]}$ is that we drop the condition that the intruder is restricted to forwarding messages between agents. The only restriction will be that messages sent by the intruder can be accepted by the honest agent at step i of the protocol. This is protocol-specific and should be determined by the semantic functions examining the protocol.

Multiple Protocol Sessions (Security View)

In this case, the game that represents protocol interactions is denoted by $\mathsf{Spr}_{[\mathsf{N}]}$:

$$\begin{aligned}
\mathsf{Spr}_{[\mathsf{N}]} &= !\mathsf{Psg}_{[\mathsf{N}]} \\
!\mathsf{Psg}_{[\mathsf{N}]} &= \mathsf{Psg}^1_{[\mathsf{N}]} \otimes \mathsf{Psg}^2_{[\mathsf{N}]} \dots \quad \forall j' > j \,\textbf{.}\, \mathsf{Psg}^{j'}_{[\mathsf{N}]} \text{ is started after } \mathsf{Psg}^{j}_{[\mathsf{N}]} \\
&= \mathsf{Csg}^{1,1} \dots \otimes \mathsf{Csg}^{1,N} \otimes \mathsf{Csg}^{2,1} \dots \otimes \mathsf{Csg}^{2,N} \dots \\
&= (\mathsf{Msg}^{1,1,1} \multimap \mathsf{Msg}^{1,1,2}) \dots \qquad \otimes(\mathsf{Msg}^{1,N,1} \multimap \mathsf{Msg}^{1,N,2})\otimes \\
&\quad\ (\mathsf{Msg}^{2,1,1} \multimap \mathsf{Msg}^{2,1,2}) \dots \qquad \otimes(\mathsf{Msg}^{2,N,1} \multimap \mathsf{Msg}^{2,N,2})\otimes \\
&\quad\ \dots
\end{aligned} \tag{5.16}$$

Definitions of moves and labeling functions of $\mathsf{Spr}_{[\mathsf{N}]}$ are the same as those of $\mathsf{Prt}_{[\mathsf{N}]}$, but the enabling relation is changed to be:

$$\begin{array}{rcll}
\star & \rightsquigarrow_{\mathsf{Spr}_{[\mathsf{N}]}} & q^{1,j,2} & \forall j' \in \{1,\dots,N\} \,\textbf{.}\, \mathbf{Id}(1,j) = \mathbf{Id}(1,j') \Rightarrow j \le j' \\
m^{i-1,j,2} & \rightsquigarrow_{\mathsf{Spr}_{[\mathsf{N}]}} & q^{i,j',2} & m \in \mathbb{M} \wedge i \in \{2,\dots M\} \wedge \\
 & & & \forall k \,\textbf{.}\, \mathbf{Id}(i,j') = \mathbf{Id}(i,k) \Rightarrow j' \le k \\
q^{i,1,1} & \rightsquigarrow_{\mathsf{Spr}_{[\mathsf{N}]}} & \texttt{start} & \\
m^{i,N,1} & \rightsquigarrow_{\mathsf{Spr}_{[\mathsf{N}]}} & \texttt{terminate} & m \in \mathbb{M} \wedge j \in \mathbb{N} \setminus \{0\} \\
q^{i,j,2} & \rightsquigarrow_{\mathsf{Spr}_{[\mathsf{N}]}} & q^{i,j,1} & i \in \mathbb{N} \setminus \{0\} \wedge j \in \{1,\dots N\} \\
q^{i,j,1} & \rightsquigarrow_{\mathsf{Spr}_{[\mathsf{N}]}} & m^{i,j,1} & m \in \mathbb{M} \wedge i \in \{1,\dots M\} \wedge j \in \{2,\dots N\} \\
m^{i,j,1} & \rightsquigarrow_{\mathsf{Spr}_{[\mathsf{N}]}} & n^{i,j,2} & m, n \in \mathbb{M} \wedge i \in \{1,\dots M\} \wedge j \in \{1,\dots N\} \\
m^{i,j,2} & \rightsquigarrow_{\mathsf{Ssg}_{[\mathsf{N}]}} & q^{i,k,2} & j < k \wedge \mathbf{Id}(i,j) = \mathbf{Id}(i,k)
\end{array} \tag{5.17}$$

The first condition above states that the play begins in Session 1 in any step where an agent expects to receive his first message in the protocol. Notice that the first move of this step has to be a question by the opponent, i.e., $q^{1,j,2}$. Once the game has started in a step in a certain session, *and* the intruder has received a message in this step ($m^{i-1,j,2}$), he can start the play in any step in the next session ($q^{i,j',2}$) provided this is also the first step in the interaction with a certain agent. This is stated by the second condition. The third condition is special for the start message, i.e., the start message is always enabled in communication step 1 in any session. The fourth condition states the condition for the termination of one session of the protocol, i.e., the reception of $m^{i,N,1}$ ($N-1$ is the number of communication steps of the protocol). The fifth, sixth and seventh rules put a condition on the sequence of moves in any communication step in a certain session. They simply state that in any communication step we cannot have a sequence $q^{i,j,2}.m^{i,j,2}$, this sequence means that an agent sends a message to the intruder without first getting a message form the intruder. This is to emphasize the rule that we established before that each communication step is an exchange between the intruder and an agent, where the intruder has to supply a message in order to get a message in return. The eighth rule imposes order on the messages of the intruder in the same session. Basically, when playing with a certain agent, the intruder has to supply messages in the order expected by this agent. It is worth noting that games defined this way give rise to a category where objects are games and a morphism between any two games G and H is a strategy over the game $G \multimap H$, details about the structure of the category can be found in [9].

Quantification over Strategies

The game Spr represents an interleaving of actions from different Csg games. Each Csg game is played between an agent and the intruder. A strategy over Spr may involve several agents. The set of all strategies over Spr is denoted $\mathcal{S}_{\mathsf{Spr}}$. For any strategy $\sigma \in \mathcal{S}_{\mathsf{Spr}}$ and a set $\mathbb{A}$ of honest agents identities, we define $\sigma^{\mathbb{A}}$ as:

$$\sigma^{\mathbb{A}} = \{ s \upharpoonright M \mid s \in \sigma \}$$

where M is defined as:

$$M = \{m \in \mathsf{Csg}^{\text{copy}} \mid \mathbf{Id}(\text{copy}) \in \mathbb{A}\}$$

Intuitively, $\sigma^{\mathbb{A}}$ is obtained from σ by eliminating all moves from any copy of the Csg game whose player P is not a member of $\mathbb{A}$. The set of all strategies involving a set $\mathbb{A}$ of players is denoted $\mathcal{S}^{\mathbb{A}}_{\mathsf{Spr}}$ and is defined as:

$$\mathcal{S}^{\mathbb{A}}_{\mathsf{Spr}} = \{\sigma^{\mathbb{A}} \mid \sigma \in \mathcal{S}_{\mathsf{Spr}}\}$$

5.2.2 Example

We take the example of the RSA3P protocol:

$$\begin{aligned} A \rightarrow B &: \{m\}_{K_A} \\ B \rightarrow A &: \{\{m\}_{K_A}\}_{K_B} \\ A \rightarrow B &: \{m\}_{K_B} \end{aligned} \tag{5.18}$$

A single session tree is depicted in Figure 5.3, where for each sequence of moves the game in which these moves were played is shown. Since a game $\mathsf{Csg}^{i,j}$ is defined to be $\mathsf{Msg}^{i,j,1} \multimap \mathsf{Msg}^{i,j,2}$, sequences of moves should have been written $q^{i,j,2}.q^{i,j,1}.m^{i,j,1}.n^{i,j,2}$ to indicate the game to which the move belongs. In Figure 5.3, we omitted superscripts from moves to improve readability.

5.3 Semantics

A strategy, for a certain agent, is its response to messages received from the network, and it is extracted from protocol specifications. Given a certain game, the set of all possible sequences of moves during the play of the game is the game tree. However, some of these sequences will be invalid according to agents strategies, e.g., a sequence

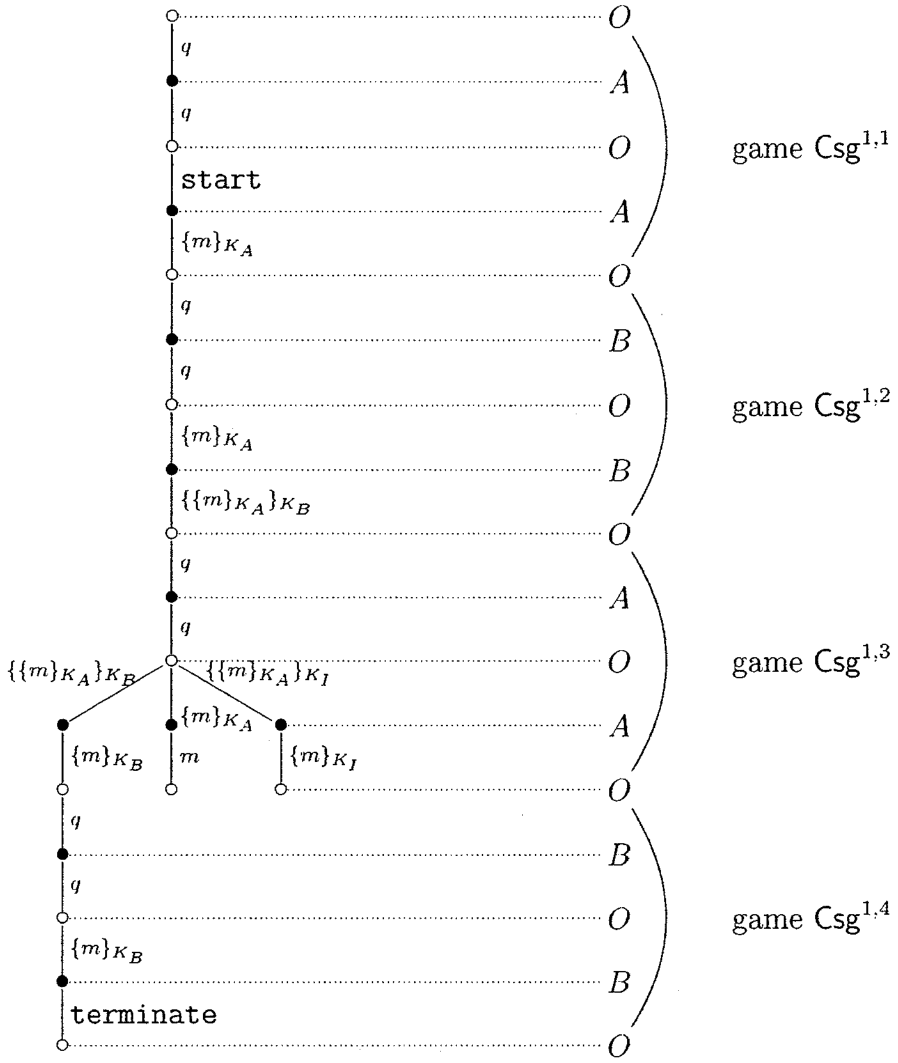


Figure 5.3: Game tree for the RSA three-pass protocol.

where agent A sends message m where his strategy is to send m'. All sequences where each agent follow their strategy are valid sequences according to the protocol specification and are taken to be the protocol semantics. In this section, we give the semantics of security protocols over the game of multiple sessions and a malicious intruder (the security view). First we recall the grammar rules that we use to specify protocols:

$$\begin{array}{rcl} \mathcal{P}rot & ::= & \mathcal{D}ecl\,.\,\mathcal{C}omm \mid \epsilon \\ \mathcal{D}ecl & ::= & \kappa_A \triangleright m\,.\,\mathcal{D}ecl \mid \nu_A \triangleright m\,.\,\mathcal{D}ecl \mid \epsilon \\ \mathcal{C}omm & ::= & \mathtt{step}\, i \triangleright A \longrightarrow B : m\,.\,\mathcal{C}omm \mid \epsilon \end{array}$$

As we mentioned earlier, the game tree of the game $\mathsf{Spr}_{[\mathsf{N}]}$ represents all possible interactions of agents in any protocol with $\mathsf{N}-1$ steps. A certain protocol $\wp$ (with $\mathsf{N}-1$ steps) is represented by a set of strategies over the game tree of $\mathsf{Spr}_{[\mathsf{N}]}$. This set of strategies forms a subtree of the game tree of $\mathsf{Spr}_{[\mathsf{N}]}$ which represents all possible interactions of agents according to the rules of $\wp$, with the assumption of a malicious intruder.

5.3.1 Protocol Semantics

We begin by defining a well-formedness condition for protocol specifications. A protocol specification is *well-formed* if steps are numbered consecutively starting at 1 and no message is sent by an agent A at step i unless it can be deduced from $\mathbf{trunc}(\phi(A), i)$ (the specification frame). Hereafter, we define the predicate Wf_Prot that checks a protocol for well-formedness, given a sequence of communication steps, a natural number and a function of type Frm mapping agent names to frames. First we need to define the function $\mathfrak{I} : i \longrightarrow \mathsf{Nat}$ that maps the syntactic symbol i to its corresponding natural number.

Table 5.2: Definitions needed for the semantic interpretation function.

name:	Type	Description
ϕ :	Frm	A function that maps agents to their specification frames, i.e., $\phi(A)$ is the specification frame of A.
ρ :	RFrm	A function that maps agents and the intruder to their real frames, i.e., $\rho(s, A)$ is the real frame of A.
comp :	Output	A procedure such that $\mathbf{comp}(i, \mathbf{rcvd}(\phi(A)))$ is the message that is sent to the network by agent A at step i according to the specification. Similarly, $\mathbf{comp}(i, \mathbf{rcvd}(\rho(s, A)))$ is the message that is actually sent to the network by agent A at step i where $\rho(s, A)$ is the actual frame seen by A during a protocol run.
P_G^{sub} :	GameTree	$P_G^{sub} \subseteq P_G$ a subset of the game tree of the game G (subtree).

$$\mathbf{Wf\_Prot} : \mathcal{C}omm \rightarrow \mathsf{Frm} \rightarrow \mathsf{Bool}$$

$$\mathbf{Wf\_Prot}[\![\mathcal{C}]\!](\phi)$$

$\mathcal{C}$ is the communication part of the protocol

ϕ is constructed by the function **frame** (Section 5.1.2)

$$\begin{aligned}
&= \mathbf{Wf\_Prot}[\![\mathtt{step}\ i \triangleright A \rightarrow B : m.\mathcal{C}']\!](\phi) \\
&= (\mathbf{trunc}(\phi(A), \mathfrak{I}[\![i]\!]) \vdash m) \wedge \\
&\quad \mathsf{Wf\_Prot}[\![\mathcal{C}']\!](\phi) \\
\mathbf{Wf\_Prot}[\![\epsilon]\!](\phi) &= \text{true}
\end{aligned}$$

In security semantics, we investigate possible manoeuvres that can be performed by the intruder in order to break the protocol's security. The security semantics of a security protocol with $\mathsf{N} - 1$ communication steps is a set of strategies over the game $\mathsf{Spr}_{[\mathsf{N}]}$. In order to be able to define these strategies we need the definitions in Table 5.2, where the first three definitions were introduced in Section 5.1.2. The only difference is that the procedure **comp** was written $\mathbf{comp}_i(\mathbf{rcvd}(\phi(A)))$, with i as a subscript not a parameter.

The semantic interpretation function $\mathfrak{P}$ assigns a set of strategies over the game $\mathsf{Spr}_{[\mathsf{N}]}$ to a protocol specification. In order to do this, we must first construct a specification frame $\phi(A)$ for each agent A and define the procedure $\mathbf{comp}(i)$ for each communication step i. Both of these tasks were demonstrated in Section 5.1.2 using the functions **frame** and **prc**, respectively. In the following, we define the semantic interpretation function $\mathfrak{P}$ that scans protocol specifications and returns a set of strategies over the game $\mathsf{Spr}_{[\mathsf{N}]}$. We assume that the frames and the **comp** procedures are already constructed.

$$\begin{aligned} \mathfrak{P} : \mathcal{P}rot \longrightarrow \mathsf{RFrm} \longrightarrow \mathsf{Output} \longrightarrow \textstyle\prod_{\mathsf{N} \in \mathsf{Nat}} P^{sub}_{\mathsf{Spr}_{[\mathsf{N}]}} \\ \mathfrak{P}[\![\, \wp \,]\!](\rho)(\mathbf{comp})(N) \;=\; S \end{aligned} \tag{5.19}$$

where,

$$\begin{aligned} S = \quad & \{s.m_1^{i,j,1}.m_2^{i,j,2} \in P_{\mathsf{Spr}_{[\mathsf{N}]}} \mid i \in \mathbb{N} \wedge j \leq N \wedge \\ & \rho(s, I) \vdash m_1 \wedge \mathbf{comp}(j, \mathbf{rcvd}(\rho(s, \mathbf{Id}(i, j)))) = m_2\} \end{aligned}$$

In (5.19), $\mathbb{N}$ is the set of natural numbers and $N - 1$ is the number of communication steps of the protocol. The semantic function $\mathfrak{P}$ collects, in the set S, all those sequences of the game tree $P_{\mathsf{Spr}_{[\mathsf{N}]}}$ that "conform" to the specification of the protocol $\wp$. A sequence $s.m^{i,j,1}.m^{i,j,2}$ "conforms" to the specification of $\wp$ if: (1) the intruder can deduce the message $m^{i,j,1}$ from its previous communication history s and (2) every message $m^{i,j,2}$ sent by an agent $A = \mathbf{Id}(i, j)$ in a step j equals the output of the computation procedure of A at step j. This output is equal to $\mathbf{comp}(j, \mathbf{rcvd}(\rho(s, A)))$ as described in Section 5.1.2, in the discussion about real and specification frames.

Chapter 6

Logic for Protocol Verification

In Chapter 5, we presented our game-based model for security protocols and defined the semantic function that maps a protocol specification to a game tree. We now need to express properties of this tree, these properties should be interesting from the security point of view. Moreover, we should be able to verify these properties against the game tree. In this chapter, we present a new logic for the expression and verification of security protocols. It can be used to specify and verify the security properties that should be satisfied by the protocol. Verification is carried out using a tableau-based proof system which is implemented as a model checking algorithm. The model is the game tree that represents protocol interactions. The logic is based two previous logics [15] and [17]. The main differences are that in [17], there is no explicit mention of the intruder, only adversarial behavior between players is considered. The logic was not specifically designed to be used for security properties, although its ideas were used later in [66] to verify non-repudiation protocols. Also, interaction between players is not explicitly modeled, i.e., no messages are exchanged between players. This complicates the specification of properties based on traces of messages. As for the logic presented in [15], the model considered is a single trace, verification of a protocol amounts to verification of a property over all traces which limits the analysis. Also, the logic considers only traces with atomic actions, it cannot specify a certain structure for the exchanged messages. In our model, the logic is based on the idea of interaction, where the game tree is built from possible interactions between players and intruder. Logic formulas specify properties over the game tree. Moves of the game are actual messages that are exchanged in a protocol run. We can therefore specify properties on the structure of messages, and on traces of messages. We can quantify existentially, universally or by players strategies, i.e., all traces in which certain players are interacting.

We begin by introducing the syntax of the logic and its semantics. We then present the tableau-based proof system and give examples for a number of security properties. Finally, we prove the properties of the proof tableaux, namely, finiteness, soundness, and completeness.

6.1 Syntax of Formulas

Before presenting the syntax of formulas, we present the concept of a sequence pattern r. A sequence pattern represents a set of game strategies; namely, all those strategies that "satisfy" the pattern. The definition of a strategy satisfying a pattern will be presented below. The syntax of patterns is specified by the following grammatical rules:

$$r ::= \epsilon \mid a^{i,j,n}.r \mid x_r.r \qquad a ::= m \mid \lceil m \rceil \tag{6.1}$$

The description of each syntactic symbol is as follows:

- The symbol x_r is called a pattern variable and is used to represent a sequence of game moves of zero or any finite length, the subscript r is added to avoid confusion with variables in moves $a^{i,j,n}$.

- The symbol $a^{i,j,n}$ is used to represent moves in the game tree $P_{\mathsf{Spr}_{[\mathsf{N}]}}$ where "a" has the form m or $\lceil m \rceil$, i and j can be variables or positive numerals and n

can be a variable or one of the constants 1 and 2.

- The symbol m represents a term in $\mathbb{T}_\Sigma(\mathbb{X})$, the message algebra with variables, and is used to represent messages. Variables in m are given the symbols $x, y, z, \ldots$ with no subscripts.

- The symbol $\lceil m \rceil$ means a term $t \in \mathbb{T}_\Sigma(X)$ that contains m, i.e., $t = m$ or $t = f(t_1, t_2, \ldots, t_n) \wedge \exists t_i \bullet t_i = \lceil m \rceil$. We can express the same meaning using the subterm relation defined in Section 5.1.2, in this case we have $m \preccurlyeq \lceil m \rceil$.

Intuitively, "$a^{i,j,n}$" represents a move played in the game $\mathsf{Spr}_{[\mathsf{N}]}$ in one copy of the Msg game, i.e., the game $\mathsf{Msg}^{i,j,n}$. This explains the limitations we have on i, j, and n since the superscript i, j, n should be a valid one according to the definition of the game $\mathsf{Spr}_{[\mathsf{N}]}$ presented in (5.16). It is important to note here that a move is a message with a superscript that designates the game in which the message was exchanged.

The sets of variables and moves in a pattern r are written $\mathbf{var}(r)$ and $\mathbf{mov}(r)$ respectively. Moreover, for any pattern r, the symbol $r|_i$ represents the variable or move at position i of r. We define the same operation for sequences of moves such that for a sequence s, $s|_i$ is the move at position i in s. As an example, we consider the pattern $r = x_r.\lceil K_A \rceil^{1,2,2}.y_r.(\{x\}_{K_A})^{1,v,1}$, in this case we have: $\mathbf{mov}(r) = \{\lceil K_A \rceil^{1,2,2}, (\{x\}_{K_A})^{1,v,1}\}$, $\mathbf{var}(r) = \{x_r, y_r\}$, $r|_1 = x_r$, $r|_3 = y_r$ and we note the use of the subscript r in pattern variables (x_r and y_r in the example) to differentiate between them and move variables (x and v in the example). We define the substitution $\theta_r : \mathbf{var}(r) \longrightarrow M^*_{\mathsf{Spr}_{[\mathsf{N}]}}$ that maps pattern variables in a sequence pattern to sequences of moves of the game $\mathsf{Spr}_{[\mathsf{N}]}$ and the substitution θ_m that maps variables in terms of the form $m^{i,j,n}$ to messages. The substitution θ_r maps patterns variables to sequences of moves in a straightforward manner, the situation is a little more complex with θ_m, since both m and the superscript i, j, n may contain variables. We overload the substitution θ_m to deal with both cases. Thus, $\theta_m(m)$ will be a ground term t in the message algebra, i.e., $t \in \mathbb{T}_\Sigma$, and $\theta_m(i, j, n)$ will be a tuple of numbers indicating a certain copy of the Msg game.

We also define the predicate $\mathbf{satisfy}(\sigma, r, \theta_m, \theta_r)$, which is true when a strategy σ in the game tree *satisfies* a pattern r. Intuitively, a strategy σ satisfies a pattern r if there is a sequence $s \in \sigma$ such that s matches r. Formally:

$$\mathbf{satisfy}(\sigma, r, \theta_m, \theta_r) = \exists s \in \sigma \,\bullet\, \mathbf{match}(s, r, \theta_m, \theta_r)$$

Let $s = s_1 s_2 \ldots s_l$ be a sequence of moves in the game tree $P_{\mathsf{Spr}_{[N]}}$, where each move in s has the form $t^{\alpha,\beta,\gamma}$. The predicate $\mathbf{match}(s, r, \theta_m, \theta_r)$, is defined as follows:

$$\begin{array}{lcl}
\mathbf{match}(\epsilon, \epsilon, \theta_m, \theta_r) & = & \text{true} \\
\mathbf{match}(s, \epsilon, \theta_m, \theta_r) & = & \text{false} \qquad \text{if } s \neq \epsilon \\
\mathbf{match}(t^{\alpha,\beta,\gamma}.s', m^{i,j,n}.r, \theta_m, \theta_r) & = & (\theta_m(m) = t) \wedge (\theta_m(i, j, n) = \alpha, \beta, \gamma) \\
 & & \wedge \mathbf{match}(s', r, \theta_m, \theta_r) \\
\mathbf{match}(t^{\alpha,\beta\gamma}.s', \lceil m \rceil^{j,in}.r, \theta_m, \theta_r) & = & (\theta_m(m) \preccurlyeq t) \wedge (\theta_m(j, in) = \alpha, \beta\gamma) \\
 & & \wedge \mathbf{match}(s', r, \theta_m, \theta_r) \\
\mathbf{match}(s, x_r.r, \theta_m, \theta_r) & = & \exists \iota \leq l \,\bullet\, \theta_r(x_r) = s_1 \ldots s_\iota \\
 & & \wedge \mathbf{match}(s_{\iota+1} \ldots s_l, r, \theta_m, \theta_r)
\end{array}$$

In order to simplify the notation, we combine θ_r and θ_m into a single substitution θ that acts on patterns in the following way:

$$\begin{array}{lcl}
\theta(\epsilon) & = & \epsilon \\
\theta(a^{i,j,n}.r) & = & \theta_m(a^{i,j,n}).\theta(r) \\
\theta(x_r.r) & = & \theta_r(x_r).\theta(r)
\end{array}$$

We follow the usual notation for substitutions and write $r\theta$ for $\theta(r)$. From the definitions of the predicate **match** and the substitution θ above, we notice that the condition for a match between a pattern and a sequence is the existence of one or more substitutions θ, we can therefore write the predicates above as **satisfy**(σ, r, θ) and **match**(s, r, θ). As an example, we consider the following:

- The pattern $r = x_r.\lceil K_A \rceil^{1,1,2}.y_r.(\{x\}_{K_A})^{1,v,2}$.
- The sequence $s = q^{1,1,2}.q^{1,1,1}.\texttt{start}^{1,1,1}.(\{m\}_{K_A})^{1,1,2}.$ $q^{1,2,2}.q^{1,2,1}.(\{m\}_{K_A})^{1,2,1}.(\{\{m\}_{K_B}\}_{K_A})^{1,2,2}$

The sequence s can match r using the substitution θ where:

- $\theta_r = [x_r \mapsto q^{1,1,2}.q^{1,1,1}.\texttt{start}^{1,1,1}, y_r \mapsto q^{1,2,2}.q^{1,2,1}.(\{m\}_{K_A})^{1,2,1}]$
- $\theta_m = [x \mapsto \{m\}_{K_B}, v \mapsto 2]$

It is easy to check that the predicate **match**(s, r, θ) will evaluate to "true" taking into account that $K_A \preccurlyeq \{m\}_{K_A}$.

Having defined patterns, and the conditions that make a pattern match a strategy, we need to be able to reason about patterns. In other words, we need to express properties of strategies through the use of patterns. These properties should be relevant as security features of protocols. We, therefore, define a logic that is able to express modalities such as "possible" and "necessary", it can also express temporal properties through the use of a recursion operator. Furthermore, a defining feature of our logic is its ability to modify the model, i.e., the game tree, by removing moves and replacing them by dummy symbols. This feature, along with the use of the recursion operator, allows us to express some counting properties, such as "for every message m there is a corresponding m'".

A formula in our logic can be:

- A variable, for use with the recursion operator.
- The negation on another formula.
- The conjunction of two formulas.

- A pattern matching formula where we match a pattern to a strategy in the game tree, modify the strategy (by replacing some moves with dummy symbols) and try to match another pattern to the modified strategy.
- A recursive formula that is the greatest fixed point of a function over sets of strategies.
- A formula that checks only those strategies that belong to certain agents.

The syntax of a formula ϕ is expressed by the following grammar:

$$\varphi ::= Z \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid [r_1 \looparrowright r_2]\varphi \mid \nu Z.\varphi \mid \langle\!\langle \mathbb{A} \rangle\!\rangle \varphi \tag{6.2}$$

We require the following two syntactic conditions:

- In $[r_1 \looparrowright r_2]$, $\mathbf{var}(r_1) = \mathbf{var}(r_2)$ and $\forall i\,.\,(r_1|_i \in \mathbf{var}(r_1) \Leftrightarrow r_1|_i = r_2|_i) \wedge (r_1|_i \in \mathbf{mov}(r_1) \Leftrightarrow r_1|_i = r_2|_i \vee r_2|_i = \circledast)$.
- In $\nu Z.\varphi$, any free Z in φ appears under the scope of an even number of negations.

The first condition above means that r_2 is obtained from r_1 by replacing some of the moves of r_1 by the dummy symbol $\circledast$, where $\forall\theta\,.\,\theta(\circledast) = \circledast$, and changing nothing else, i.e., all the pattern and move variables of r_2 are those of r_1. This is to ensure that, in a strategy σ, if σ contains a sequence s and there exists a substitution θ such that $\mathbf{match}(s, r_1, \theta)$ = true, then we can obtain a sequence t from s such that $\mathbf{match}(t, r_2, \theta)$ = true. We were able to use the same substitution θ with r_2 since r_2 does not introduce any new variables, hence $r_2\theta$ contains only ground terms. The sequence t has the same length as s ($\mathbf{len}(t) = \mathbf{len}(s)$) and it is constructed such that $(r_2\theta|_i = \circledast) \Rightarrow (t|_i = \circledast)$, and in all other positions $t|_i = s|_i$. It is then possible to replace s by t in σ, and get a new strategy σ', which we denote by writing $\sigma' = \sigma[t/s]$. The second condition is necessary for the monoticity of the semantic interpretation function as will be explained in the semantics section.

Intuitively, the formula $[r_1 \looparrowright r_2]\varphi$ is true, for a strategy σ, if for all substitutions θ such that $\mathbf{satisfy}(\sigma, r_1, \theta) = \text{true}$, then σ' satisfies φ, where σ' is obtained from σ such that $\mathbf{satisfy}(\sigma', r_2, \theta) = \text{true}$. In other words, $\exists s \in \sigma \,.\, \mathbf{match}(s, r, \theta) = \text{true}$ and $\sigma' = \sigma[t/s]$ where t is obtained from s as mentioned earlier (by substituting some moves in s with $\circledast$). The quantifier $\langle\langle \mathbb{A} \rangle\rangle$ chooses the subtree of the game tree where only the agents in the set $\mathbb{A}$ interact together (and the intruder) through their strategies. The formula to the right of $\langle\langle \mathbb{A} \rangle\rangle$ operates on this subtree. The rest of the formulas have their usual meaning in modal μ-calculus [35].

6.2 Semantics

A formula in the logic is interpreted over a game tree. Given a certain game tree $\mathcal{G}$, e.g. $P_{\mathsf{Spr}_{[\mathsf{N}]}}$, and an environment e that maps formulae variables to strategies in $\mathcal{G}$, the semantic function $[\![\varphi]\!]_e^{\mathcal{G}}$ maps a formula φ to a set of strategies $S \subseteq \mathcal{S}_{\mathcal{G}}$ that satisfy ϕ, where $\mathcal{S}_{\mathcal{G}}$ is the set of all strategies in $\mathcal{G}$. The semantic function is defined by the following rules:

$$\begin{aligned}
[\![Z]\!]_e^{\mathcal{G}} &= e(Z) \\
[\![\neg\varphi]\!]_e^{\mathcal{G}} &= \mathcal{S}_{\mathcal{G}} \setminus [\![\varphi]\!]_e^{\mathcal{G}} \\
[\![\varphi_1 \wedge \varphi_2]\!]_e^{\mathcal{G}} &= [\![\varphi_1]\!]_e^{\mathcal{G}} \cap [\![\varphi_2]\!]_e^{\mathcal{G}} \\
[\![[r_1 \looparrowright r_2]\varphi]\!]_e^{\mathcal{G}} &= \{\sigma \in \mathcal{S}_{\mathcal{G}} \mid \forall\theta \,.\, \mathbf{satisfy}(\sigma, r_1, \theta) \Rightarrow \sigma' \in [\![\varphi]\!]_e^{\mathcal{G}'}\} \\
[\![\nu Z.\varphi]\!]_e^{\mathcal{G}} &= \bigcup\{S \subseteq \mathcal{S}_{\mathcal{G}} \mid S \subseteq [\![\varphi]\!]_{e[Z \mapsto S]}^{\mathcal{G}}\} \\
[\![\langle\langle \mathbb{A} \rangle\rangle\varphi]\!]_e^{\mathcal{G}} &= \mathcal{S}_{\mathcal{G}}^{\mathbb{A}} \cap [\![\varphi]\!]_e^{\mathcal{G}}
\end{aligned} \tag{6.3}$$

In the semantic rules above, the first three concern formula variables, negation and conjunction, respectively. The fourth rule is for formulas matching strategies to patterns, where, in $\sigma' \in [\![\varphi]\!]_e^{\mathcal{G}'}$, $\mathcal{G}' = \mathcal{G}[t/s]$ and $\sigma' = \sigma[t/s]$. The sequence s is in σ and matches r_1, i.e., $s \in \sigma \wedge \mathbf{match}(s, r_1, \theta) = \text{true}$. The sequence t is obtained from s by the following rules:

$$\mathbf{len}(t) = \mathbf{len}(s), \forall i \leq \mathbf{len}(t) \,.\, t|_i = \begin{cases} ⊛ & r_2\theta = ⊛ \\ s|_i & \text{otherwise} \end{cases}$$

The fifth rule describes the meaning of the recursive formula $\nu Z.\varphi$ to be the greatest fixpoint of a function $f : 2^{S_\mathcal{G}} \to 2^{S_\mathcal{G}}$, where $f(S) = [\![\varphi]\!]^{\mathcal{G}}_{e[Z \mapsto S]}$. The function f is defined over the lattice $(2^{S_\mathcal{G}}, \subseteq, \cup, \cap)$, the syntactic condition on φ (X appears under the scope of an even number of negations) ensures that $f(S)$ is monotone [35] and hence has a greatest fixpoint (by Knaster-Tarski fixed point theorem). The last rule is for quantification by agent identities, where $\mathcal{S}^{\mathbb{A}}_{\mathcal{G}}$ was defined in Section 5.2.1 to be the set of all strategies of the game $\mathcal{G}$ which involve a set $\mathbb{A}$ of players.

We use the following shorthand notations:

$$\begin{aligned} \neg(\neg\varphi_1 \wedge \neg\varphi_2) &\equiv \varphi_1 \vee \varphi_2 \\ \neg\varphi_1 \vee \varphi_2 &\equiv \varphi_1 \Rightarrow \varphi_2 \\ \neg[r_1 \looparrowright r_2]\neg\varphi &\equiv \langle r_1 \looparrowright r_2 \rangle \varphi \\ \neg\nu Z.\neg\varphi[\neg Z/Z] &\equiv \mu Z.\varphi \end{aligned} \tag{6.4}$$

Moreover, we use the following two notations $\mathtt{tt} \equiv \nu Z.Z$ and $\mathtt{ff} \equiv \mu Z.Z$. Examples of logic formulas and the properties they express are given in Section 6.3.2. Here, we just give a simple example where we have:

- A strategy $\sigma = \{q^{1,1,2}.q^{1,1,1}.\mathtt{start}^{1,1,1}.m_1^{1,1,2},$
  $q^{1,1,2}.q^{1,1,1}.\mathtt{start}^{1,1,1}.m_1^{1,1,2}.q^{1,2,2}.q^{1,2,1}.m_2^{1,2,1}.m_3^{1,2,2}\}$

- The property $\langle x.m_1^{j,i,n}.y \looparrowright x.⊛.y\rangle\langle x.m_2^{j,2,n}.y \looparrowright x.⊛.y\rangle\mathtt{tt}$, where x, y, j, i and n are variables.

This property states that, in σ, there exists a sequence s that contains m_1 with any superscript (hence the variables j, i, n). Then, in s, if we replace $m_1^{j,i,n}$ by ⊛, the resulting sequence t will satisfy the rest of the formula which is the part $\langle x.m_2^{j,2,n}.y \looparrowright x.⊛.y\rangle\mathtt{tt}$. This part says that, in the *new* sequence t, we can find a move with message m_2 and this move was played in any session j in step 2, i.e., in the game $\mathsf{Csg}^{j,2}$, this can be in the game $\mathsf{Msg}^{j,2,1}$ or $\mathsf{Msg}^{j,2,2}$. Then, in t, if we replace $m_2^{j,2,n}$ by $\circledast$, the resulting sequence t' will satisfy $\mathtt{tt}$, which is satisfied by any sequence. Therefore, the strategy σ satisfies the property:

- For the part: $\langle x.m_1^{j,i,n}.y \looparrowright x.\circledast.y\rangle$, we have:
  $s = q^{1,1,2}.q^{1,1,1}.\mathtt{start}^{1,1,1}.m_1^{1,1,2}.q^{1,2,2}.q^{1,2,1}.m_2^{1,2,1}.m_3^{1,2,2}$ and
  $\theta = [x \mapsto q^{1,1,2}.q^{1,1,1}.\mathtt{start}^{1,1,1}, y \mapsto q^{1,2,2}.q^{1,2,1}.m_2^{1,2,1}.m_3^{1,2,2}, j \mapsto 1, i \mapsto 1, n \mapsto 2]$
  $\theta(x.\circledast.y) = q^{1,1,2}.q^{1,1,1}.\mathtt{start}^{1,1,1}.\circledast.q^{1,2,2}.q^{1,2,1}.m_2^{1,2,1}.m_3^{1,2,2}$ and thus we get t:
  $t = q^{1,1,2}.q^{1,1,1}.\mathtt{start}^{1,1,1}.\circledast.q^{1,2,2}.q^{1,2,1}.m_2^{1,2,1}.m_3^{1,2,2}$

- The part: $\langle x.m_2^{j,2,n}.y \looparrowright x.\circledast.y\rangle\mathtt{tt}$ can be checked similarly starting by the sequence t.

From the previous example we note that the operators $\langle . \looparrowright . \rangle$ and $[. \looparrowright .]$ bind variables. Therefore, the variables $x, y, \ldots$ of the first part of the formula are not the same as those of the second part. Moreover, the ability to remove moves from sequences combined with the recursive formulas allows us to specify properties such as "for every message m there is a corresponding n", which provides a sort of "counting". In the following, we investigate some properties of the the recursion operator.

Lemma 6.2.1 $[\![\varphi[\psi/Z]]\!]_e^{\mathcal{G}} = [\![\varphi]\!]_{e[Z \mapsto [\![\psi]\!]_e^{\mathcal{G}}]}^{\mathcal{G}}$

The proof is done by structural induction over φ.
Base case: $\varphi = Z$
$[\![\psi/Z]\!]_e^{\mathcal{G}} = [\![\psi]\!]_e^{\mathcal{G}}$
But: $[\![Z]\!]_e^{\mathcal{G}} = e(Z)$, so $[\![\psi/Z]\!]_e^{\mathcal{G}} = [\![Z]\!]_{e[Z \mapsto [\![\psi]\!]_e^{\mathcal{G}}]}^{\mathcal{G}}$
We demonstrate the case $\varphi = [r_1 \looparrowright r_2]\varphi'$ and the other cases can be easily proved:
$[\![\varphi[\psi/Z]]\!]_e^{\mathcal{G}} = \{\sigma \in \mathcal{S}_{\mathcal{G}} \mid \forall\theta \,.\, \mathbf{satisfy}(\sigma, r_1, \theta) \Rightarrow \sigma' \in [\![\varphi'[\psi/Z]]\!]_e^{\mathcal{G}'}\}$
By induction hypothesis:

$[\![\varphi[\psi/Z]]\!]^{\mathcal{G}}_e = \{\sigma \in \mathcal{S}_{\mathcal{G}} \mid \forall\theta\,.\,\mathbf{satisfy}(\sigma, r_1, \theta) \Rightarrow \sigma' \in [\![\varphi']\!]^{\mathcal{G}'}_{e[Z\mapsto [\![\psi]\!]^{\mathcal{G}}_e]}\}$
$[\![\varphi[\psi/Z]]\!]^{\mathcal{G}}_e = [\![\varphi]\!]^{\mathcal{G}}_{e[Z\mapsto [\![\psi]\!]^{\mathcal{G}}_e]}\}$
$\square$

As a result, we have $[\![\nu Z.\varphi]\!]^{\mathcal{G}}_e = [\![\varphi[\nu Z.\varphi/Z]]\!]^{\mathcal{G}}_e$. This follows from the fact that $[\![\nu Z.\varphi]\!]^{\mathcal{G}}_e = [\![\varphi]\!]^{\mathcal{G}}_{e[Z\mapsto T]}$, where $T = \bigcup\{S \subseteq \mathcal{S}_{\mathcal{G}} \mid S \subseteq [\![\varphi]\!]^{\mathcal{G}}_{e[Z\mapsto S]}\} = [\![\nu Z.\varphi]\!]^{\mathcal{G}}_e$.

We can now prove that the semantics of the expression $\mu Z.\varphi$ defined earlier as $\neg\nu Z.\neg\varphi[\neg Z/Z]$ is the least fixpoint of the function $f(S) = [\![\varphi]\!]^{\mathcal{G}}_{e[Z\mapsto S]}$.

$$\begin{aligned}
&[\![\neg\nu Z.\neg\varphi[\neg Z/Z]]\!]^{\mathcal{G}}_e\\
&= \mathcal{S}_{\mathcal{G}} \setminus \bigcup\{S \subseteq \mathcal{S}_{\mathcal{G}} \mid S \subseteq [\![\neg\varphi[\neg Z/Z]]\!]^{\mathcal{G}}_{e[Z\mapsto S]}\}\\
&= \mathcal{S}_{\mathcal{G}} \setminus \bigcup\{S \subseteq \mathcal{S}_{\mathcal{G}} \mid S \subseteq \mathcal{S}_{\mathcal{G}} \setminus [\![\varphi]\!]^{\mathcal{G}}_{e[Z\mapsto [\![\neg Z]\!]^{\mathcal{G}}_{e[Z\mapsto S]}]}\}\\
&= \mathcal{S}_{\mathcal{G}} \setminus \bigcup\{S \subseteq \mathcal{S}_{\mathcal{G}} \mid S \subseteq \mathcal{S}_{\mathcal{G}} \setminus [\![\varphi]\!]^{\mathcal{G}}_{e[Z\mapsto \mathcal{S}_{\mathcal{G}}\setminus S]}\}
\end{aligned}$$

For any set of strategies $S \subseteq \mathcal{S}_{\mathcal{G}}$, let $S^c = \mathcal{S}_{\mathcal{G}} \setminus S$. By De Morgan laws, for any two sets A and B, we have:
$(A \cap B)^c = A^c \cup B^c$, $(A \cup B)^c = A^c \cap B^c$, $A \subseteq B \Rightarrow B^c \subseteq A^c$.

$$\begin{aligned}
&[\![\neg\nu Z.\neg\varphi[\neg Z/Z]]\!]^{\mathcal{G}}_e\\
&= (\bigcup\{\mathcal{S}_{\mathcal{G}} \setminus S^c \subseteq \mathcal{S}_{\mathcal{G}} \mid S \subseteq ([\![\varphi]\!]^{\mathcal{G}}_{e[Z\mapsto S^c]})^c\})^c\\
&= (\bigcup\{\mathcal{S}_{\mathcal{G}} \setminus S^c \subseteq \mathcal{S}_{\mathcal{G}} \mid [\![\varphi]\!]^{\mathcal{G}}_{e[Z\mapsto S^c]} \subseteq S^c\})^c\\
&= \bigcap(\{\mathcal{S}_{\mathcal{G}} \setminus S^c \subseteq \mathcal{S}_{\mathcal{G}} \mid [\![\varphi]\!]^{\mathcal{G}}_{e[Z\mapsto S^c]} \subseteq S^c\})^c\\
&= \bigcap\{S^c \subseteq \mathcal{S}_{\mathcal{G}} \mid [\![\varphi]\!]^{\mathcal{G}}_{e[Z\mapsto S^c]} \subseteq S^c\}
\end{aligned}$$

Moreover, we investigate the semantics of the the expression $\langle r_1 \nrightarrow r_2\rangle\varphi$ as defined above:

$$\begin{aligned}
&[\![\langle r_1 \looparrowright r_2\rangle\varphi]\!]^{\mathcal{G}}_e = [\![\neg[r_1 \looparrowright r_2]\neg\varphi]\!]^{\mathcal{G}}_e \\
&= \{\sigma \in \mathcal{S}_{\mathcal{G}} \mid \neg\forall\theta\, \textbf{.}\, \textbf{satisfy}(\sigma, r_1, \theta) \Rightarrow \sigma' \in \mathcal{S}_{\mathcal{G}} \setminus [\![\varphi]\!]^{\mathcal{G}'}_e\} \\
&= \{\sigma \in \mathcal{S}_{\mathcal{G}} \mid \neg\forall\theta\, \textbf{.}\, \textbf{satisfy}(\sigma, r_1, \theta) \Rightarrow \neg\sigma' \in [\![\varphi]\!]^{\mathcal{G}'}_e)\} \\
&= \{\sigma \in \mathcal{S}_{\mathcal{G}} \mid \neg\forall\theta\, \textbf{.}\, \neg(\textbf{satisfy}(\sigma, r_1, \theta) \wedge \sigma' \in [\![\varphi]\!]^{\mathcal{G}'}_e)\} \\
&= \{\sigma \in \mathcal{S}_{\mathcal{G}} \mid \exists\theta\, \textbf{.}\, (\textbf{satisfy}(\sigma, r_1, \theta) \wedge \sigma' \in [\![\varphi]\!]^{\mathcal{G}'}_e)\}
\end{aligned}$$

In the derivation above, we used the fact that $\psi \Rightarrow \neg\varphi$ implies $\neg(\psi \wedge \varphi)$, and the fact that for any set of strategies S, $S \cap (\mathcal{S}_{\mathcal{G}} \setminus S) = \emptyset$.

6.3 Tableau-Based Proof System

In general terms, a tableau-based proof system [97] starts by the formula to be proved and works its way backwards by trying to prove the formula's components. For instance, in order to prove a conjunction, we have to prove all its conjuncts. A proof for a formula is therefore a tree with the formula at the root, and a branch is created for each component of the formula, e.g., a branch for each conjunct. A formula is true if it has a "successful" tree, the definition of a successful tree will be presented below. Defining a tableau-based proof system for a logic means defining a method to construct proof trees starting by the formula to be proved. This definition, therefore, directly produces an algorithm to prove formulas. In our case, this will be our model checking algorithm.

Before we present the rules of the tableau, we define the immediate subformula relation [35] $\prec_I$ as:

$$\begin{array}{lll}
\varphi & \prec_I & \neg\varphi \\
\varphi & \prec_I & [r_1 \looparrowright r_2]\varphi \\
\varphi_i & \prec_I & \varphi_1 \wedge \varphi_2 \quad i \in \{1, 2\} \\
\varphi & \prec_I & \nu Z.\varphi
\end{array}$$

We define the proper subformula relation $\prec$ to be the transitive closure of $\prec_I$ and the subformula relation $\preceq$ to be its transitive and reflexive closure. A tableau based proof system starts from the formula to be proved as the root of a proof tree and proceeds in a top down fashion. In every rule of the tableau, the conclusion is above the premises. Each conclusion of a certain rule represents a node in the proof tree, whereas the premises represent the children to this node. In our case, the proof system proves sequents of the form $H, b \vdash \sigma \in \varphi$, which means that under a set H of hypotheses and the symbol $b \in \{\epsilon, \neg\}$, the strategy σ satisfies the property φ. The set H contains elements of the form $\sigma : \nu Z.\varphi$ and is needed for recursive formulas. Generally speaking, the use of H means that in order to prove that a strategy σ satisfies a recursive formula φ_{rec}, we must prove the following: Under the hypothesis that σ satisfies φ_{rec}, then σ also satisfies the unfolding of φ_{rec}. We also define the set $H \upharpoonright \nu Z.\varphi = \{\sigma \in \mathcal{S}_{\mathcal{G}} \mid \sigma : \nu Z.\varphi \in H\}$. The use of H, and b will be apparent after we state the rules of the proof system:

$$(\mathrm{R}_{\neg}) \qquad \frac{H, b \vdash \sigma \in \neg\varphi}{H, \neg b \vdash \sigma \in \varphi}$$

$$(\mathrm{R}_{\wedge}) \qquad \frac{H, \epsilon \vdash \sigma \in \varphi_1 \wedge \varphi_2}{H, \epsilon \vdash \sigma \in \varphi_1 \qquad H, \epsilon \vdash \sigma \in \varphi_2}$$

$$(\mathrm{R}_{\neg\wedge_i}) \qquad \frac{H, \neg \vdash \sigma \in \varphi_1 \wedge \varphi_2}{H, \neg \vdash \sigma \in \varphi_i} \qquad i \in \{1, 2\}$$

$$(\mathrm{R}_{\nu}) \qquad \frac{H, b \vdash \sigma \in \nu Z.\varphi}{H' \cup \{\sigma : \nu Z.\varphi\}, b \vdash \sigma \in \varphi[\nu Z.\varphi / Z]} \qquad \sigma : \nu Z.\varphi \notin H$$

$$(\mathrm{R}_{[]}) \qquad \frac{H, \epsilon \vdash \sigma \in [r_1 \looparrowright r_2]\varphi}{\xi_1 \quad \xi_2 \quad \ldots \xi_n} \qquad \xi_i = H, \epsilon \vdash \sigma_i \in \varphi$$

$$(\mathrm{R}_{\neg[]_i}) \qquad \frac{H, \neg \vdash \sigma \in [r_1 \looparrowright r_2]\varphi}{H, \neg \vdash \sigma_i \in \varphi} \qquad i \in \{1, \ldots, n\}$$

$$(\mathrm{R}_{\langle\langle\rangle\rangle}) \qquad \frac{H, b \vdash \sigma \in \langle\langle \mathbb{A} \rangle\rangle\varphi}{H, b \vdash \sigma^{\mathbb{A}} \in \varphi}$$

where,

in (R_ν) :

$$H' = H \setminus \{\sigma' : \Gamma \mid \nu Z.\varphi \prec \Gamma\}$$

in $(\mathrm{R}_{[]})$

$\sigma_i = \sigma[t_i/s]$, t_i and s satisfy the formula:

$$\forall\theta\,.\,((\exists s \in \sigma\,.\,\mathbf{match}(s, r_1, \theta) = \text{true}) \Rightarrow \exists t_i\,.\,t_i|_j = \begin{cases} r_2\theta|_j & r_2\theta = \circledast \\ s|_j & \text{otherwise} \end{cases})$$

in and $(\mathrm{R}_{\neg[]_i})$:

$\sigma_i = \sigma[t_i/s]$, t_i and s satisfy the formula:

$$\exists\theta\,.\,((\exists s \in \sigma\,.\,\mathbf{match}(s, r_1, \theta) = \text{true}) \Rightarrow \exists t_i\,.\,t_i|_j = \begin{cases} r_2\theta|_j & r_2\theta = \circledast \\ s|_j & \text{otherwise} \end{cases})$$

The first rule $(\mathrm{R}_\neg)$ concerns negation of formulas where $b \in \{\epsilon, \neg\}$ serves as a "memory" to remember negations, in this case $\epsilon\varphi = \varphi$. We define $\epsilon\epsilon = \epsilon$, $\epsilon\neg = \neg\epsilon = \neg$, and $\neg\neg = \epsilon$. The second rule $(\mathrm{R}_\wedge)$ states that in order to prove a conjunction, we have to prove both conjuncts. The third rules is actually two rules: $(\mathrm{R}_{\neg\wedge_1})$ and $(\mathrm{R}_{\neg\wedge_2})$. It states that in order to prove the negation of a conjunction, we have to prove the negation of the first conjunct or the negation of the second conjunct. This is due to the identity $\neg(\varphi_1 \wedge \varphi_2) = \neg\varphi \vee \neg\varphi_2$. The fourth rule (R_ν) concerns proving a recursive formulas, where the construction of the set H, via H', ensures that the validity of the sequent $H, b \vdash \sigma \in \nu Z.\varphi$ is determined only by subformulas of φ [35]. The fifth rule $(\mathrm{R}_{[]})$ takes care of formulas matching strategies to patterns. Intuitively, it states that in order to prove that a strategy σ satisfies the formula $[r_1 \looparrowright r_2]\varphi$, we have to prove that for *all* substitutions θ_1 to θ_n that make σ match the pattern r_1, if we replace the sequence $s = r_1\theta_i$ (which is one of sequences of σ) by $r_2\theta_i$ to obtain σ_i then σ_i satisfies φ. This is why this rule will produce n branches in the tableau, i.e., one branch for each substitution θ_i. The sixth rule deals with the formula $\neg[r_1 \looparrowright r_2]\varphi$, it is actually a collection of n rules: $(\mathrm{R}_{\neg[]_1})$ to $(\mathrm{R}_{\neg[]_n})$. It is based on the identity: $\neg[r_1 \looparrowright r_2]\neg\varphi = \langle r_1 \looparrowright r_2\rangle\varphi$, which was introduced in (6.4). So, considering that $\neg\neg\varphi = \varphi$, we have $\neg[.]\varphi = \neg[.]\neg\neg\varphi = \langle.\rangle\neg\varphi$. So, proving $\neg[r_1 \looparrowright r_2]\varphi$ amounts to proving $\langle r_1 \looparrowright r_2\rangle\neg\varphi$. The semantics of $\langle r_1 \looparrowright r_2\rangle\varphi$ was investigated in Section 6.2 on Page 149. According to the semantics, in order to prove that a strategy σ satisfies $\langle r_1 \looparrowright r_2\rangle\varphi$, we have to find *at least one* substitution θ_i that makes σ match r_1, and if we replace $r_1\theta_i$ by $r_2\theta_i$ to obtain σ_i, then σ_i will satisfy φ. This is why this rule is a collection of n rules since if we can find a single substitution out of any n possible substitutions, the formula is proved. Finally, the seventh rule ($R_{\langle\langle\rangle\rangle}$) is for formulas dealing with subtrees of the game tree.

Starting from the formula to be proved at the root of the proof tree, the tree grows downwards until we hit a node where the tree cannot be extended anymore, i.e., a leaf node. A formula is proved if it has a successful tableau, where a successful tableau is one whose all leaves are successful. A successful leaf meets one of the following conditions:

1. $H, \epsilon \vdash \sigma \in Z$ and $\sigma \in [\![Z]\!]_e^{\mathcal{G}}$
2. $H, \neg \vdash \sigma \in Z$ and $\sigma \notin [\![Z]\!]_e^{\mathcal{G}}$
3. $H, \epsilon \vdash \sigma \in \nu Z.\varphi$ and $\sigma : \nu Z.\varphi \in H$
4. $H, \epsilon \vdash \sigma \in [r_1 \looparrowright r_2]\varphi$ and $\{\sigma \in \mathcal{S}_{\mathcal{G}} \mid \exists\theta\,.\,\mathbf{match}(\sigma, r_1, \theta)\} = \emptyset$.

6.3.1 Properties of Tableau System

We would like to prove three main properties, namely the finiteness of the tableau for finite models, the soundness, and the completeness. Soundness and completeness are proved with respect to a relativized semantics, in which the semantic function considers the set of hypotheses (the set H) as a parameter. The new semantics is the same as the one provided in Section 6.2 for all formulas except for recursive formulas where it is defined as:

$$[\![\nu Z.\varphi]\!]_e^{\mathcal{G},\Pi} = (\nu [\![\varphi]\!]_{e[Z \mapsto S \cup S']}^{\mathcal{G},\Pi}) \cup S'$$
$$\text{where, } S' = H \upharpoonright \nu Z.\varphi$$

In the equation, the greatest fixpoint operator is applied to a function $f(S) = [\![\varphi]\!]_{e[Z \mapsto S]}^{\mathcal{G},\Pi}$ whose argument is $S \cup S'$. Since the function is monotone over a complete lattice, as mentioned earlier, then the existence of a greatest fixpoint is guaranteed. We now list the theorems regarding the properties of the proof system. These properties are: Finiteness, soundness, and completeness, the detailed proofs are in Section 6.4.

Theorem 6.3.1 Finiteness. *For any sequent $H, b \vdash \sigma \in \varphi$ there exists a finite number of finite tableaux.*

The idea of the proof is that for any formula at the root of the proof tree we begin applying the rules $R_\neg$, $R_\wedge$, $R_{\neg\wedge_i}$, $R_{[]}$, $R_{\neg[]_i}$, $R_{\langle\!\langle\rangle\!\rangle}$, and R_ν. The application of the first six rules results in shorter formulas, while the application of the R_ν rule results in larger hypothesis sets H. The proof shows that shortening a formula and increasing the size of H cannot continue infinitely. Hence no path in the tree will have infinite length. Branching happens in the proof tree whenever we have an expression of the form $\varphi_1 \wedge \varphi_2$ or $[r_1 \looparrowright r_2]\varphi$. Finite branching is guaranteed in the first case by the finite length of any expression and in the second case by the finiteness of the model.

Theorem 6.3.2 Soundness. *For any sequent $H, b \vdash \sigma \in \varphi$ with a successful tableau, $\sigma \in [\![\varphi]\!]_e^{\mathcal{G},\Pi}$.*

The idea behind the proof is to show that all the successful leaves described above are valid and that the application of the rules of the tableau preserves semantic validity.

Theorem 6.3.3 Completeness. *If for a strategy $\sigma \in \mathcal{S}_\mathcal{G}$, $\sigma \in [\![\varphi]\!]_e^{\mathcal{G},\Pi}$, then the sequent $H, b \vdash \sigma \in \varphi$ has a successful tableau.*

The proof relies on showing that we cannot have two successful tableaux for the sequents $H, b \vdash \sigma \in \varphi$ and $H, b \vdash \sigma \in \neg\varphi$.

6.3.2 Examples of Security Properties

The following are some security properties expressed in our logic:

- Fairness: A protocol is fair for a player L if:
  $\langle\!\langle \{L\} \rangle\!\rangle (\langle x.\lceil m \rceil^{i,j,2}.y \looparrowright x. \circledast .y \rangle tt) \Rightarrow (\langle z.\lceil n \rceil^{i,j',1}.q \looparrowright z. \circledast .q \rangle tt)$
  where player L exchanges item m for item n. This formula means that player L will always have a strategy to obtain item n whenever he sent item m, in the same session j. This matches the definition in [32].

- Authentication: We use the definition called "agreement" in [71]:
  $\langle\!\langle \{A, B\} \rangle\!\rangle \nu X.[x.\lceil m \rceil^{i,j,2}.y.\lceil m \rceil^{i,j'1}.z \looparrowright x. \circledast .y. \circledast z]X$
  Here we assume agents A and B authenticate each other over one single message m, this could be easily extended to the case where authentication takes place over a series of messages. We note that one session of A corresponds to one session of B (the variable j) and that the message m is exchanged between A and B without being changed.

6.4 Proofs

6.4.1 Proof of Finiteness

The following definitions are needed for the proofs:

Definition 6.4.1 Size. *The size $|\varphi|$ of a formula is a positive integer, which can be computed using the following rules:*
$|Z| = 1$
$|\neg\varphi| = 1 + |\varphi|$

$|\varphi_1 \wedge \varphi_2| = 1 + |\varphi_1| + |\varphi_2|$

$|[r_1 \looparrowright r_2]\varphi| = 1 + |\varphi|$

$|\nu Z.\varphi| = 1 + |\varphi|$

Definition 6.4.2 Fischer-Ladner Closure. *The Fischer-Ladner closure $CL(\varphi)$ of a formula φ is a set of formulas that is defined as:*

$CL(Z) = \{Z\}$

$CL(\neg\varphi) = \{\neg\varphi\} \cup CL(\varphi)$

$CL(\varphi_1 \wedge \varphi_2) = \{\varphi_1 \wedge \varphi_2\} \cup CL(\varphi_1) \cup CL(\varphi_2)$

$CL([r_1 \looparrowright r_2]\varphi = \{[r_1 \looparrowright r_2]\varphi\} \cup CL(\varphi)$

$CL(\nu Z.\varphi) = \{\nu Z.\varphi\} \cup CL(\varphi[\nu Z.\varphi/Z])$

In other words the closure of a formula φ is a set that contains φ and all of its subformulas $\varphi_1, \ldots, \varphi_n$, where formula variables are replaced by the smallest formula in which they are bound. Formally:

$$CL(\varphi) = \{\varphi\} \cup \{\text{rep}(\varphi_1)\} \cup \ldots \cup \{\text{rep}(\varphi_n)\}$$

where,

$$\forall i \in \{1, \ldots, n\} \,.\, \varphi_i \preceq \varphi$$

$$\text{rep}(\varphi_i) = \begin{cases} \nu Z.\varphi' & \varphi_i = Z \\ \varphi_i & \text{otherwise} \end{cases}$$

Since all formula variables Z_i appear in φ, so $e(Z_i) = \nu Z_i.\varphi'$ will be some formula $\varphi_i \preceq \varphi$. Hence, $CL(\varphi)$ will be finite and bounded by $|\varphi|$.

Definition 6.4.3 H-Ordering. *Between hypothesis sets, we define some ordering relations relative to a formula φ, where $\mathcal{H}$ is the set of all hypothesis sets:*

$\sqsubseteq_Z = \mathcal{H} \times \mathcal{H}$

$\sqsubseteq_{\neg\varphi'} = \sqsubseteq_{\varphi'}$

$\sqsubseteq_{\varphi_1 \wedge \varphi_2} = \sqsubseteq_{\varphi_1} \cap \sqsubseteq_{\varphi_2}$

$\sqsubseteq_{[r_1 \looparrowright r_2]\varphi'} = \sqsubseteq_{\varphi'}$

$\sqsubseteq_{\nu Z.\varphi'} = \{(H_1, H_2) \in \sqsubseteq_{\varphi'} \mid (H_2, H_1) \in \sqsubseteq_{\varphi'} \Rightarrow H_1 \upharpoonright \nu Z.\varphi' \subseteq H_2 \upharpoonright \nu Z.\varphi'\}$
$H_1 =_\varphi H_2 \Leftrightarrow H_1 \sqsubseteq_\varphi H_2 \wedge H_2 \sqsubseteq_\varphi H_1$
$H_1 \sqsubset_\varphi H_2 \Leftrightarrow H_1 \sqsubseteq_\varphi H_2 \wedge H_1 \neq_\varphi H_2$
$(\forall \varphi' \in CL(\varphi) \,.\, H_1 \sqsubseteq_{\varphi'} H_2) \Rightarrow H_1 \trianglelefteq_\varphi H_2$
$H_1 \equiv_\varphi H_2 \Leftrightarrow H_1 \trianglelefteq_\varphi H_2 \wedge H_2 \trianglelefteq_\varphi H_1$
$H_1 \triangleleft_\varphi H_2 \Leftrightarrow H_1 \trianglelefteq_\varphi H_2 \wedge H_2 \ntrianglelefteq_\varphi H_1$

From the definitions above, it is clear that $\forall \varphi, \varphi' \,.\, \varphi' \prec_I \varphi \Rightarrow \sqsubseteq_\varphi \subseteq \sqsubseteq_{\varphi'}$. The following results about $\sqsubseteq_\varphi$ are proved in [35]:

- $\sqsubseteq_\varphi$ is reflexive and transitive, i.e., a preorder.
- $H \sqsubset_{\nu Z.\varphi} H' \cup \{\sigma : \nu Z.\varphi\}$, where $H' = H \backslash \{\sigma' : \Gamma \mid \nu Z.\varphi \prec \Gamma\}$ and $\sigma : \nu Z.\varphi \notin H$.
- $\sqsubset_\varphi$ has no infinite ascending chains.

Definition 6.4.4 Sequent ordering *For any two sequents $\tau_1 = H_1, b_1 \vdash \sigma_1 \in \varphi_1$ and $\tau_2 = H_2, b_2 \vdash \sigma_2 \in \varphi_2$, such that $\varphi_2 \in CL(\varphi_1)$, the relation $\tau_1 < \tau_2$ holds, whenever one of the following holds:*
- $H_1 \triangleleft_{\varphi_2} H_2$
- $H_1 \equiv_{\varphi_2} H_2 \wedge |\varphi_2| < |\varphi_1|$

Lemma 6.4.1 *The sequent ordering relation $<$ has no infinite ascending chain.*

Proof. Suppose we start by $\tau_0 = H_0, b_0 \vdash \sigma \in \varphi_0$, the chain $\tau_0 < \tau_1 < \tau_2 \ldots$ cannot ascend infinitely, the proof follows from the facts that $|CL(\varphi_0)|$ is finite, $\triangleleft_{\varphi_0}$ has no infinite ascending chains, and $\forall \varphi' \in CL(\varphi) \,.\, |\varphi'| < |\varphi|$. □

Theorem 6.4.1 Finiteness. *For any sequent $\tau = H.b \vdash \sigma \in \varphi$ there exists a finite number of finite tableaux.*

Proof. The proof is by well-founded induction on the inverse of the sequent ordering relation, since $<$ has no infinite ascending chain, then $<^{-1}$ has no infinite descending chain.

Induction hypothesis: for all τ' such that $\tau < \tau'$, τ' has a finite number of finite tableaux.

Required: for any τ, the applicable rule of the tableau will produce a finite number of τ', where $\tau < \tau'$.

We consider here the cases of matching and recursive formulas, i.e., $\tau = H, b \vdash \sigma \in [r_1 \looparrowright r_2]\varphi$ and $\tau = H, b \vdash \sigma \in \nu Z.\varphi$, since the other cases are straightforward.

Case 1: $\tau = H, b \vdash \sigma \in [r_1 \looparrowright r_2]\varphi$

The only applicable rules are $R_{[]}$ and $R_{\neg[]_i}$. The rule $R_{[]}$ generates sequents of the form $\tau_i = H_i, \epsilon \vdash \sigma_i \in \varphi$, where $i \in \mathbb{I} \subseteq \mathbb{N}$, $\mathbb{N}$ is the set of natural numbers. For all i, $H_i = H$, moreover $\varphi \in CL([r_1 \looparrowright r_2]\varphi)$ and $|\varphi| < |[r_1 \looparrowright r_2]\varphi|$, hence $\tau < \tau_i$. The set $\mathbb{I}$ is determined by the set $\Theta = \{\theta_i \mid \textbf{satisfy}(\sigma, r_1, \theta_i) = \text{true}\}$, where $|\mathbb{I}| = |\Theta|$. Therefore we need to prove that Θ is finite, which follows from the fact that σ is finite since we are dealing with finite models. So we can write $i \in \{0, 1, \ldots n\}$. The same argument is valid for the collection of rules $R_{\neg[]_i}$, since each rule generates only one sequent $H, \neg \vdash \sigma_i \in \varphi$. However, the number of rules is finite, since each rule picks one substituion from the finite set Θ.

Case 2: $\tau = H, b \vdash \nu Z.\varphi$

The only applicable rule is R_ν, which produces only one sequent $\tau' = H', b \vdash \varphi[\nu Z.\varphi/Z]$, using the results about $\sqsubseteq_\varphi$ above and the definition of the sequent order relation, it is easy to prove that $\tau < \tau'$. $\square$

6.4.2 Proof of Soundness

To prove the soundness, we have to prove that all successful leaves are semantically sound and that all rules of the tableau preserve soundness. We consider two cases here and the rest of the cases can be easily proved.

Theorem 6.4.2 Soundness. *For any sequent $H, b \vdash \sigma \in \varphi$ with a successful tableau, $\sigma \in [\![\varphi]\!]_e^{\mathcal{G},\Pi}$.*

Proof. We consider the two following cases of successful leaves and rules:

Case 1: The sequent $H, \epsilon \vdash \sigma \in [r_1 \looparrowright r_2]\varphi$, is a successful leaf when $\forall \theta \,.$ **satisfy**$(\sigma, r_1, \theta) =$ false. This agrees with the semantics $[\![[r_1 \looparrowright r_2]\varphi]\!]_e^{\mathcal{G},\Pi} = \{\sigma \in \mathcal{S}_\mathcal{G} \mid \forall\theta \,.\, \textbf{satisfy}(\sigma, r_1, \theta) \Rightarrow \sigma' \in [\![\varphi]\!]_e^{\mathcal{G}',\Pi}\}$, since the implication ($\Rightarrow$) will evaluate to "true". Moreover, the rule $R_{[]}$ preserves soundness since it is just an expression of the semantics of $\forall$ from first order logic.

Case 2: The sequent $H, \epsilon \vdash \sigma \in \nu Z.\varphi$ is a successful leaf when $\sigma : \nu Z.\varphi \in H$. We recall the definition of R_ν and the relativized semantics of $\nu Z.\varphi$:

$$\frac{H, b \vdash \sigma \in \nu Z.\varphi}{H' \cup \{\sigma : \nu Z.\varphi\}, b \vdash \sigma \in \varphi[\nu Z.\varphi / Z]} \qquad \sigma : \nu Z.\varphi \notin H$$

$$[\![\nu Z.\varphi]\!]_e^{\mathcal{G},\Pi} = (\nu [\![\varphi]\!]_{e[Z \mapsto S \cup S']}^{\mathcal{G},\Pi}) \cup S'$$

where, $H' = H \setminus \{\sigma' : \Gamma \mid \nu Z.\varphi \prec \Gamma\}$ and $S' = H \restriction \nu Z.\varphi$. So, what we would like to prove is that $[\![\varphi[\nu Z.\varphi / Z]]\!]_e^{\mathcal{G},\Pi' \cup \{\sigma : \nu Z.\varphi\}} = (\nu [\![\varphi]\!]_{e[Z \mapsto S \cup \{\sigma\}]}^{\mathcal{G},\Pi}) \cup \{\sigma\}$. The proof relies on lemma 6.2.1 and on the properties of fixpoints. It is detailed in [35].

6.4.3 Proof of Completeness

The proof depends on the following theorem:

Theorem 6.4.3 *The sequent $\tau = H, b \vdash \sigma \in \varphi$ has a successful tableau if and only if $\tau' = H, b \vdash \sigma \in \neg\varphi$ has no successful tableau.*

Proof.

Step 1: $\Rightarrow$

Suppose that both τ and τ' have successful tableaux, then by the soundness theorem $\sigma \in [\![b\varphi]\!]_e^{\mathcal{G},\Pi}$ and $\sigma \in [\![b\neg\varphi]\!]_e^{\mathcal{G},\Pi}$, which implies $\sigma \in [\![\neg b\varphi]\!]_e^{\mathcal{G},\Pi}$. From the definition of the semantics, we will have $\sigma \in [\![b\varphi]\!]_e^{\mathcal{G},\Pi}$ and $\sigma \notin [\![b\varphi]\!]_e^{\mathcal{G},\Pi}$, which is a contradiction.

Step 2: $\Leftarrow$

We would like to prove that if τ has no successful tableau then τ' has a successful tableau. We do this by induction on the height of the proof tree, starting from the leaves, i.e., prove that unsuccessful leaves imply $\sigma \in \neg\varphi$ and whenever a node in the proof tree implies $\sigma \in \neg\varphi$ its parent implies the same thing. We consider here the case of the unsuccessful leaf $H, \neg \vdash \sigma \in [r_1 \looparrowright r_2]\varphi$ and $\{\sigma \in \mathcal{S}_\mathcal{G} \mid \exists\theta \centerdot \textbf{satisfy}(\sigma, r_1, \theta)\} = \emptyset$. By definition, we have $H, \neg \vdash \sigma \in [r_1 \looparrowright r_2]\varphi \Rightarrow H, \epsilon \vdash \sigma \in \neg[r_1 \looparrowright r_2]\varphi$ which is a successful leaf proving that $\sigma \in \neg[r_1 \looparrowright r_2]\varphi$. Now, we consider the rules $R_{[]}$ and $R_{\neg[]_i}$:

$$\frac{H, \epsilon \vdash \sigma \in [r_1 \looparrowright r_2]\varphi}{\xi_1 \quad \xi_2 \quad \ldots \xi_n} \qquad \text{and} \qquad \frac{H, \neg \vdash \sigma \in [r_1 \looparrowright r_2]\varphi}{H, \neg \vdash \sigma_i \in \varphi}$$

By the rule $R_{[]}$, if at least one $\xi_i = H, \epsilon \vdash \sigma_i \in \varphi$ does not have a successful tableau, then $\tau = H, \epsilon \vdash \sigma \in [r_1 \looparrowright r_2]\varphi$ does not have a successful tableau. In this case, by induction hypothesis, $\xi_i' = H, \epsilon \vdash \sigma_i' \in \neg\varphi$ has a successful tableau. By definition, we have that $H, \neg \vdash \sigma_i' \in \varphi$ has a successful tableau. By the rule $R_{\neg[]_i}$, this means that $H, \neg \vdash \sigma \in [r_1 \looparrowright r_2]\varphi$, will have a successful tableau, or in other words $H, \epsilon \vdash \sigma \in \neg[r_1 \looparrowright r_2]\varphi$ will have a successful tableau. $\square$

Theorem 6.4.4 Completeness. *If for a strategy $\sigma \in \mathcal{S}_\mathcal{G}$, $\sigma \in [\![\ \varphi\ ]\!]_e^{\mathcal{G},II}$, then the sequent $H, b \vdash \sigma \in \varphi$ has a successful tableau.*

Proof. The proof follows from theorems 6.4.2 and 6.4.3 by contradiction. $\square$

Chapter 7

Implementation and Case Studies

In this chapter, we present a software tool that we developed, in Java, to validate our approach by experimenting with test cases. The tool allows the specification of security protocols using the notation that we presented in Chapter 5 and that is very close to the standard notation used by protocol designers and practitioners. Once a protocol is specified, its game tree model is constructed. Security properties can then be specified as logic formulas of the logic that we developed and verified against the protocol model. In the following sections, we describe the tool and provide details on its innerworkings and the design choices that we made. We begin by presenting the syntax of protocol specifications and logic formulas, followed by a description of the algorithms used to generate game tree models of protocols. We then provide the model checking algorithm used to verify security properties over the game tree. Finally, we consider several test cases that were analyzed using the tool.

7.1 Specification Language

In this section, we describe the specification languages for both protocols and logic formulas. We give the syntax of the specification languages in Extended Backus

Naur Form (EBNF) accompanied by natural language clarifications.

7.1.1 Specifying Protocols

The syntax for the specification language in EBNF is:

```
<Program> ::= ("Declaration" <Declaration>
              | "Communication" <Communication>)+
<Declaration>::= (<Agent> "Knows" <Message> ";"
                 | <Agent> "Produces" <Message> ";")*
<Communication>::= (<Step>)+ | "if" "(" <Condition> ")" (<Step>)+ "endif"
<Condition>::= <Agent> "Receives" <Message>
<Step> ::= (<Agent> "->" <Agent> ":" <Message> ";")*
<Message>::= <SimpleMessage>("." <SimpleMessage>)*
<SimpleMessage>::= <PrimitveMessage> | <Nonce> | <Key> | <AgentID> |  <Encrypted>
<Encrypted>::= "{" <Message> "}" <Key>
<PrimitveMessage>::= <Message> <Rest>
Nonce::= "Nonce" <Rest>
Key::="SmKey" <Rest> | "PrKey" <Rest> | "PbKey" <Rest>
AgentID::= "IDofAgent" <Anything>
Agent::= "Agent" <Rest>
Rest::= "_" <Anything>
<Anything> ::= <AnyValidJavaIdentifier>
```

A protocol specification consists mainly of two parts: Declaration and communication, they start by the keywords `"Declaration"` and `"Communication"` respectively. The declaration part consists of a number of statements, each ending by a semicolon, that declare persistent knowledge and fresh knowledge. Persistent knowledge of an agent is any data that the agent possesses in all protocol sessions without change, e.g., the agent's public key. Fresh knowledge, on the other hand, is any data that an agent freshly generates for each protocol session, e.g., session keys. A persistent knowledge statement contains the keyword `"Knows"`, whereas a fresh knowledge statement contains the keyword `"Produces"` as listed in the grammar.

Both statements must specify the intended agent and the message that it knows or freshly produces. In all cases, "Agent_Intruder" is reserved for the intruder.

The communication part of the protocol consists of a number of communication steps specifications. Steps can also be grouped into a conditional block that starts with the keyword "if" and ends with "endif". The conditional block is executed only if its condition is true, where the condition states that a certain agent receives a specific message. Each step specifies the agent that sends the message, the agent that is supposed to receive it and the message itself. The message can be a concatenation, a text message (<PrimitiveMessage> in the grammar), a nonce, a key, an agent's identification, or an encrypted message. Identifiers for text messages, nonces, and agent names start with the keywords "Message_", "Nonce_", and "Agent_" followed by any valid Java identifier. We can also specify symmetric keys, private keys and public keys, which start by "SmKey_", "PrKey_", "PbKey_", respectively followed by any valid Java identifier. Concatenated messages are denoted by a sequence of messages separated by dots and encrypted messages have the form "{" <Message> "}" <Key> which designates the message to be encrypted and the encryption key. As an example we give the specification of the public key Needham-Scroeder protocol [87] below:

```
Declaration
Agent_A Knows PrKey_A.PbKey_A.PbKey_B.IDofAgent_A;
Agent_B Knows PbKey_A.PrKey_B.PbKey_B;
Agent_A Produces Nonce_NA;
Agent_B Produces Nonce_NB;
Communication
Agent_A -> Agent_B : {Nonce_NA.IDofAgent_A}PbKey_B;
Agent_B -> Agent_A : {Nonce_NA.Nonce_NB}PbKey_A;
Agent_A -> Agent_B : {Nonce_NB}PbKey_B;
```

The tool analyzes the protocol specification statically, first by checking the protocol syntax for any errors, then by doing a static semantic analysis. The static semantic analysis ensures the following:

- No message is specified to be sent by an agent unless it is possible for the agent to deduce it at this point of the protocol. In order to be able to deduce a message, the agent either knows or produces it in a declaration or can deduce it from its knowledge by applying either decryption or deconcatenation or a combination thereof, e.g., the agent knows the message encrypted by a key and knows the key. Rules for deducing messages were explained in Chapter 4, Section 4.2.

- For each private key there corresponds a public key and vice versa. This correspondence is syntactically specified by having both keys end with the same string after the prefixes `"PrKey"` and `"PbKey"`. For instance, in the specification above, `"PrKey_A"` and `"PbKey_A"` are a private-public key pair.

- Each agent ID corresponds to an agent that ends with the same suffix after the underscore. For instance, in the specification above, `"IDofAgent_A"` corresponds to `"Agent_A"`.

If the protocol specification passes the static analysis successfully, the game tree is generated as will be discussed below.

7.1.2 Specifying Logic Formulas

The syntax for specifying logic formulas is close to the one used in Chapter 6, Section 6.1; grammar rules are given below in EBNF:

```
<Formula>::= <Basicformula> | <Conjunctionformula> | <Transformationformula> |
             <Agentrestrictionformula> | <Recursiveformula>
<Basic> ::= <Primitiveboolean> | <Formulavariable>
```

```
          | <Negationformula> | "(" <Formula> ")"
<Primitiveboolean> ::= "True" | "False"
<Formulavariable> ::= ":" <AnyValidJavaIdentifier> ":"
<Negationformula> ::= "!" <Basic> | "!" <Transformationformula> |
                      "!" <Agentrestrictionformula> | "!" <Recursiveformula>
<Conjunctionformula> ::= <Basicformula> ("&&" <Formula>)+
<Transformationformula> ::= "["<Sequencepattern>"->"<Sequencepattern>"]" <Formula>
<Sequencepattern> ::= (<Sequencepatterunit>)+
<Sequencepatternunit> ::= <Sequencevariable> | "'" <Messagepattern> "'"
<Sequencevariable> ::= "SeqVar_" <AnyValidJavaIdentifier>
<Messagepattern> ::= <Simplemessagepattern> ( "." <Simplemessagepatttern>)*
<Simplemessagepattern> ::= <Messagevariable> | <Primitvemessage> | <Nonce> |
                           <Key> | <Agent> |  <Encryptedmessagepattern>
<Messagevariable> ::= "MesVar_" <AnyValidJavaIdentifier>
<Encryptedmessagepattern> ::= "{" <messagepattern> "}" (<key> | <variable>)
<agentrestriction> ::= "<<" <Agent> (, <Agent>)* ">>" <Formula>
<Recursive> ::= "^" <Formulavariable> "." <Formula>
```

The grammar ensures the following operator precedence rules: `"!"` binds most tightly, then `"&&"`, then `"[..]"`, then `"<<..>>"` and finally `"^"` which is the greatest fixpoint operator. For instance `!A && B` means `(!A) && B` and `^X. A && B` means `^X. (A && B)`

Before starting the model checking algorithm, formulas are analyzed to be sure they follow the syntactic rules explained in Chapter 6. To be more precise, the following rules will be checked:

- All formula variables are bound to fixpoint operators.
- All formula variables, inside formulas, appear under the scope of an even number of negation operators.
- In transformation formulas of the form `[r_1 -> r_2]`φ, the patterns `r_1` and `r_2` have the same length, all sequence variables of `r_2` are those of `r_1` and,

finally, all message patterns that appear in `r_1` appear in `r_2` or are replaced by the special message `Message_*`.

For instance, under the rules above, the formula `^:X:.:X:&&:Y:` is rejected because the formula variable `Y` appears freely. Similarly, the formula `[x.y -> x.z]True` is also rejected because the sequence variable `z` does not appear to the left of the arrow. The formulas `[x.'x'.y -> x.*.y]True` and `!^:x:.:x:` on the other hand are all well-formed. We note the use of single quotes for message patterns and colons for formula variables. As an example, `'x'` denotes a message pattern whereas `x` denotes a sequence variable and `:x:` denotes a formula variable.

7.2 Generating the Game Tree

In order to generate the game tree, we need to create roles, sessions, agents, and the intruder. The information necessary to create each of them is collected when protocol specifications are scanned according to the grammar rules described above. In the following sections, we discuss our implementation of each of these entities.

7.2.1 Roles, Sessions and Agents

Messages are implemented as abstract data types; each message type is a class. We deal with messages as symbolic terms that are manipulated by operations defined for each class of messages. This corresponds to the formal or symbolic analysis of security protocols. Roles are the most fundamental units in our tool, whereas a session links roles together since a session is composed of two or more roles that agents fill when they communicate in a session. A finite number of sessions is created and, in each session, agents participate in the communication, where each agent plays one of the session roles. Each role is implemented as a class that has a specification frame and a real frame as described in Chapter 5, Section 5.1.2. The specification frame is built from protocol declarations and communication steps, and differs from the real frame only in the vector of received messages. Furthermore, for each message that is sent in a certain role, the specification frame contains the procedure that should be followed by the agent playing the role in order to construct this message. Details about agents procedures are explained in Section 7.2.2.

From protocol specifications, we create instances of the class `RoleSpecification`, which are the roles. Each role contains a specification frame that is created with the role and to which we add role messages as initial knowledge, sent messages, or received messages. Whenever we add a sent message, we create a list of message construction actions and also specify the number of messages that the role receives before sending this message. This number is important since we can use it to know which messages, in the vector of received messages, the role receives before being ready to send a certain message. A role contains also a real frame, which is used only when the role is being played by an agent at a certain session as we will explain later.

A session relates roles together; it contains roles necessary to start protocol execution. Execution on the other hand starts when all session roles are filled by agents. In our implementation, the agents, as instances of the class `Agent`, are the software entities carrying out the actual execution of the protocol. In other words, roles are behavioral descriptions and agents are the communicating entities that behave according to one or more roles in one or more sessions. The class `Session` contains references to the roles of the session and the agents playing these roles, whereas the class `Agent` contains references to the roles played by the agent and the sessions in which the agent contributes. When an agent joins a session as a certain role, all references are updated accordingly, taking into account that the real frame of a role is unique to a certain session and a certain agent as it collects all the messages received by an agent at a specific session.

7.2.2 Intruder Actions and Agents Responses

Agent Responses, are specified by protocol steps, which implicitly describe procedures taken by agents upon receiving messages. We model these procedures as abstract computations in Chapter 4, Section 4.1.3. In our implementation, a procedure is a vector of actions and a temporary storage, where each action represents a computation step, message retrieval or message storage. Computation steps manipulate messages symbolically as in message encryption, for instance. Message retrieval and message storage actions retrieve or store messages in the temporary storage, the list of initially known messages, and the list of received messages. This way, in order to construct a message, we go through its corresponding vector of actions performing one action at a time until the vector ends and we obtain the final result. For instance, if the specification frame contains messages `{Message_m}Key_k1` as a received message and `Key_k1` as initial knowledge, in order to compute the message `Message_m` we have to retrieve `{Message_m}Key_k1`, retrieve `Key_k1`, and then decrypt the former by the latter. Of course, we have to take into account that we may need to store intermediate results of the computation in case we need them later, for this reason a message construction list of actions is equipped with temporary storage. As a follow up on the last example, if we intend to construct `{Message_m}Key_k2`, we first need to compute `Message_m`, store it, compute `Key_k2`, and then perform the encryption.

7.2.3 Game Tree

The main unit of building the game tree is the tree block, which represents possible interactions between the intruder and an agent at a certain protocol step in a certain session. In other words, a tree block is like a tree of the Csg game, in which the intruder sends one of possible messages and the agent replies. The non-determinism of intruder actions is modeled as branching of the tree, whereas the concurrent execution of two or more protocol sessions is modeled as interleaving of tree blocks. Unlike the Csg game tree, the tree block does not contain edges labeled by questions which serve to relate messages to message requests. This does not affect the analysis since we always keep track of the origin of messages and their intended destination. Each path in the tree block contains two edges; the first of which represents the message sent by the intruder to an agent and the second represents the agent's response. The first tree block in the game tree contains only one path; the intruder sends the `start` message of the first protocol session and the agent that plays the initiator role in this session responds with its first message. At the leaf node of the root tree block we attach other tree blocks each of which represents the execution of a protocol step at a certain session. Moreover, several tree blocks having the same root can be considered as a larger tree block as long as we keep track of the protocol step and session number for each message. Then, at leaf nodes of each tree block, we try to extend the path that ends at the leaf by adding more tree blocks. We continue the process recursively until no more tree blocks can be added, which is the case when no more protocol steps at any session can be executed along the path. In our tool, the game tree is built using this method in a depth-first manner.

7.3 Model Checking Algorithm

The model checking algorithm is an implementation of the tableau-based proof system of Chapter 6, Section 6.3. The tableau is implemented as a tree whose root node contains the formula to be proved. The tree grows until we reach leaf nodes, which determine the success or failure of the tableau. The formula is declared to be true if it has at least one successful tableau. In the following we discuss some methods that we implemented in order to increase the efficiency of the algorithm.

We build the tableau tree in a depth-first manner; starting at a certain node, we construct a path in the tree till we reach a leaf before considering other paths.

If we reach an unsuccessful leaf, the property is declared false and we don't need to continue building the tableau. The property is declared to be true if all the leafs of the tableau are successful. However, there are more considerations, since in our formulation a formula may have more than one tableau. This is the case of an "or"-branching for instance, and only one tableau needs to be successful for the formula to be true. Instead of constructing a new tableau for each "or"-branch, we try out each possible path out of the "or"-node, and if one path ends by a successful node, we continue to build the rest of the tableau. If, on the other hand, we cannot find a successful path, we declare the property as false and abort building the tableau. The same rule applies for properties that are existentially quantified. Of course, for "and"-branching and universally quantified properties, all of the paths out of the branching node should be successful.

The model checking algorithm is shown in pseudocode below as a function named "check". The function has as inputs a game tree, a hypothesis set and a logic formula and it returns true or false. It is first called by the game tree, the empty hypothesis set and the formula φ to be checked. The formula must pass the syntactic restrictions described above before being given to the function.

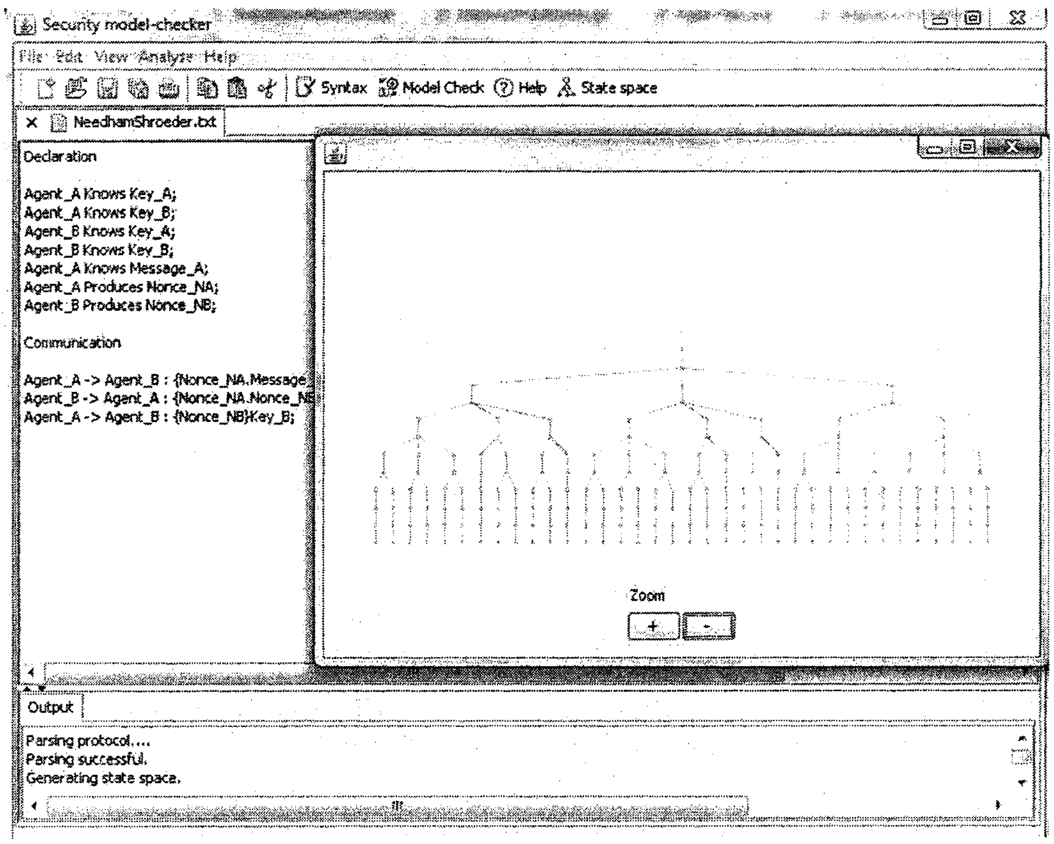
```
fun check:  game tree, hypothesis set, formula $\rightarrow$ {true, false}
check($G, H, \varphi$)
case $\neg\varphi'$ of $\varphi$:
   return $\neg$check($G, H, \varphi'$)
case $\varphi_1 \wedge \varphi_2$ of $\varphi$:
   case true of check($G, H, \varphi_1$):
      return check($G, H, \varphi_2$)
   case false of check($G, H, \varphi_1$):
      return false
   end case
case $[r_1 \mathrel{\circ\!\!\!\mapsto} r_2]\varphi'$ of $\varphi$:
  for each $\sigma$ in $\{\sigma \mid r_1\sigma \in G\}$
    case false of check($G[r_2\sigma/r_1\sigma], H, \varphi'$):
      return false
    case true of check($G[r_2\sigma/r_1\sigma], H, \varphi'$):
      skip
    end case
  end for
  return true
case $\nu Z.\varphi'$ of $\varphi$:
  case true of($G, \nu Z.\varphi') \in H$:
    return true
  case false of($G, \nu Z.\varphi') \in H$:
    return check($G, H \cup (G, \nu Z.\varphi'), \varphi'[\nu Z.\varphi'/Z]$)
end case
```

7.4 Case Studies

In this section, we analyze several well-known protocols and present the result of the analysis. We report on the construction of the game tree and present the properties we checked. A screenshot is shown in Figure 7.1, it depicts the main window of the tool where we can see a protocol specification and the tree visualization window where we can see the game tree generated from the protocol. The game tree window has different levels of magnification so more details of the tree can be revealed.

7.4.1 Needham-Schroeder Protocol

The Needham-Schroeder protocol public key protocol is specified as follows [87]:



```
Declaration

Agent_A Knows Key_A;
Agent_A Knows Key_B;
Agent_B Knows Key_A;
Agent_B Knows Key_B;
Agent_A Knows Message_A;
Agent_A Produces Nonce_NA;
Agent_B Produces Nonce_NB;

Communication

Agent_A -> Agent_B : {Nonce_NA.Message_
Agent_B -> Agent_A : {Nonce_NA.Nonce_NB
Agent_A -> Agent_B : {Nonce_NB}Key_B;
```

```
Parsing protocol....
Parsing successful.
Generating state space.
```

Figure 7.1: Screenshot of the software environment.

$$\begin{aligned}
&\text{Step 1.}\quad A \rightarrow B: \quad \{N_A, A\}_{K_B}\\
&\text{Step 2.}\quad B \rightarrow A: \quad \{N_A, N_B\}_{K_A}\\
&\text{Step 3.}\quad B \rightarrow A: \quad \{N_B\}_{K_B}
\end{aligned}$$

In the protocol specification above, N_A, K_A, N_B, and K_B are nonces and public keys of agents A and B respectively. The specification written in the syntax of our tool is:

```
Declaration
Agent_A Knows Key_A;
Agent_A Knows Key_B;
Agent_B Knows Key_A;
Agent_B Knows Key_B;
Agent_A Knows Message_A;
Agent_A Produces Nonce_NA;
Agent_B Produces Nonce_NB;
Communication
Agent_A -> Agent_B : {Nonce_NA.Message_A}Key_B;
Agent_B -> Agent_A : {Nonce_NA.Nonce_NB}Key_A;
Agent_A -> Agent_B : {Nonce_NB}Key_B;
```

7.4.2 Woo and Lam Protocol

The Woo and Lam authentication protocol is specified as follows [113]:

$$\begin{aligned}
&\text{Step 1.}\quad P \rightarrow Q: \quad P, N_1\\
&\text{Step 2.}\quad Q \rightarrow P: \quad Q, N_2\\
&\text{Step 3.}\quad P \rightarrow Q: \quad \{P, Q, N_1, N_2\}_{K_{PS}}\\
&\text{Step 4.}\quad Q \rightarrow S: \quad \{P, Q, N_1, N_2\}_{K_{PS}}, \{P, Q, N_1, N_2\}_{K_{QS}}\\
&\text{Step 5.}\quad S \rightarrow Q: \quad \{Q, N_1, N_2, K_{PQ}\}_{K_{PS}}, \{P, N_1, N_2, K_{pq}\}_{K_{QS}}\\
&\text{Step 6.}\quad Q \rightarrow P: \quad \{Q, N_1, N_2, K_{PQ}\}_{K_{PS}}, \{N_1, N_2\}_{K_{PQ}}\\
&\text{Step 7.}\quad P \rightarrow Q: \quad \{N_2\}_{K_{PQ}}
\end{aligned}$$

In the protocol specification above, Key_{XY} is the shared symmetric key between agents X and Y. Also, N_1 and N_2 are the nonces produced by agents P and Q, respectively. The specification in the syntax of our tool is:

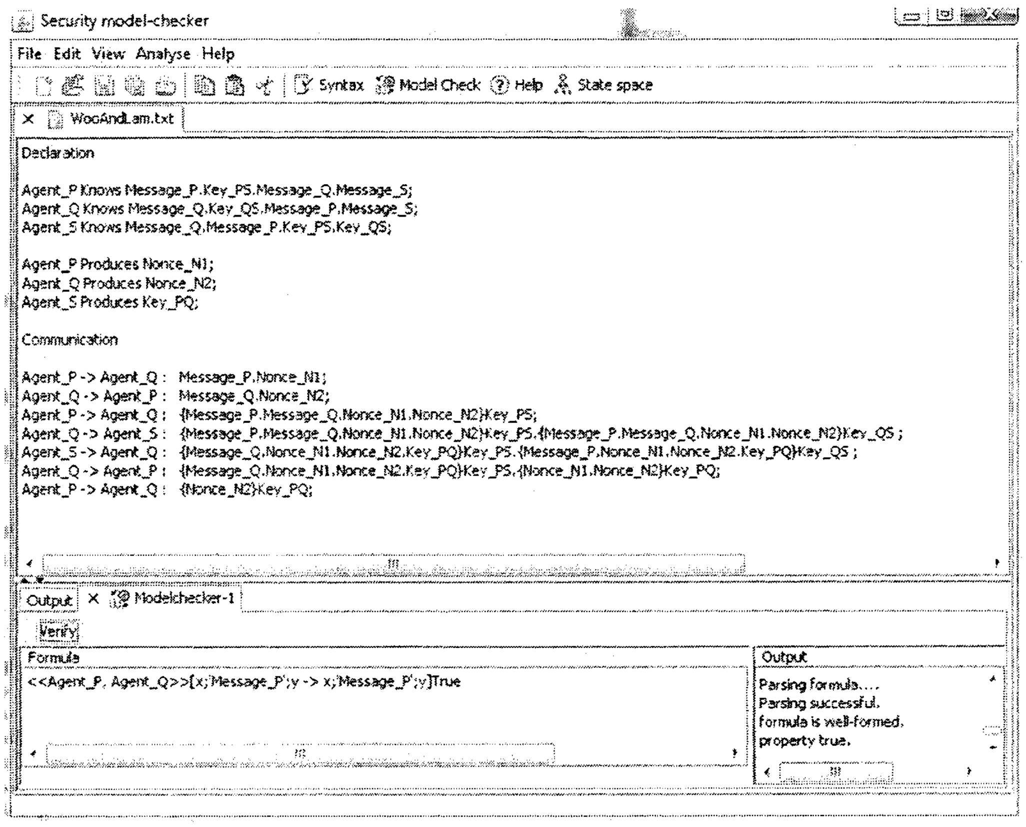
```
 Declaration Agent_P Knows
Message_P.Key_PS.Message_Q.Message_S; Agent_Q Knows
Message_Q.Key_QS.Message_P.Message_S; Agent_S Knows
Message_Q.Message_P.Key_PS.Key_QS; Agent_P Produces Nonce_N1;
Agent_Q Produces Nonce_N2; Agent_S Produces Key_PQ; Communication
Agent_P -> Agent_Q : Message_P.Nonce_N1; Agent_Q -> Agent_P :
Message_Q.Nonce_N2; Agent_P -> Agent_Q :
{Message_P.Message_Q.Nonce_N1.Nonce_N2}Key_PS; Agent_Q -> Agent_S :
{Message_P.Message_Q.Nonce_N1.Nonce_N2}Key_PS.
                     {Message_P.Message_Q.Nonce_N1.Nonce_N2}Key_QS ;
Agent_S -> Agent_Q : {Message_Q.Nonce_N1.Nonce_N2.Key_PQ}Key_PS.
                     {Message_P.Nonce_N1.Nonce_N2.Key_PQ}Key_QS ;
Agent_Q -> Agent_P : {Message_Q.Nonce_N1.Nonce_N2.Key_PQ}Key_PS.
                     {Nonce_N1.Nonce_N2}Key_PQ;
Agent_P -> Agent_Q : {Nonce_N2}Key_PQ;
```

Figure 7.2 shows a screenshot of the tool with the lower half of the window containing one of the verified properties.

7.4.3 ASW Protocol

The ASW protocol [21] is a fair exchange protocol. We give below its specification in our tool. In the specification, in order to save space here we have the following denotations:

```
 me1 = {PbKey_O. PbKey_R.
IDofAgent_T. Message_Text. Nonce_NO}PrKey_O me2 = {{PbKey_O.
PbKey_R. IDofAgent_T. Message_Text.
      Nonce_NO}PrKey_O. Nonce_NR}PrKey_R
ma1 = {Message_Abort. {PbKey_O. PbKey_R. IDofAgent_T. Message_Text.
      Nonce_NO}PrKey_O }PrKey_O
```



```
Declaration

Agent_P Knows Message_P.Key_PS.Message_Q.Message_S;
Agent_Q Knows Message_Q.Key_QS.Message_P.Message_S;
Agent_S Knows Message_Q.Message_P.Key_PS.Key_QS;

Agent_P Produces Nonce_N1;
Agent_Q Produces Nonce_N2;
Agent_S Produces Key_PQ;

Communication

Agent_P -> Agent_Q :   Message_P.Nonce_N1;
Agent_Q -> Agent_P :   Message_Q.Nonce_N2;
Agent_P -> Agent_Q :   {Message_P.Message_Q.Nonce_N1.Nonce_N2}Key_PS;
Agent_Q -> Agent_S :   {Message_P.Message_Q.Nonce_N1.Nonce_N2}Key_PS.{Message_P.Message_Q.Nonce_N1.Nonce_N2}Key_QS ;
Agent_S -> Agent_Q :   {Message_Q.Nonce_N1.Nonce_N2.Key_PQ}Key_PS.{Message_P.Nonce_N1.Nonce_N2.Key_PQ}Key_QS ;
Agent_Q -> Agent_P :   {Message_Q.Nonce_N1.Nonce_N2.Key_PQ}Key_PS.{Nonce_N1.Nonce_N2}Key_PQ;
Agent_P -> Agent_Q :   {Nonce_N2}Key_PQ;
```

```
<<Agent_P, Agent_Q>>[x;'Message_P';y -> x;'Message_P';y]True
```

```
Parsing formula....
Parsing successful.
formula is well-formed.
property true.
```

Figure 7.2: Screenshot of the Woo and Lam protocol.

```
Declaration
Agent_O Knows PrKey_O. PbKey_O. PbKey_R. PbKey_T. IDofAgent_O. IDofAgent_R.
IDofAgent_T. Message_Text. Message_Abort. Message_Resolve. Message_Timeout;
Agent_R Knows PbKey_O. PrKey_R. PbKey_R. PbKey_T. IDofAgent_O. IDofAgent_R.
IDofAgent_T. Message_Resolve. Message_Timeout;
Agent_T Knows PbKey_O. PbKey_R. PbKey_T. PrKey_T. IDofAgent_O. IDofAgent_R.
IDofAgent_T. Message_Abort. Message_Resolve. Message_Timeout;
Agent_O Produces Nonce_NO;
Agent_R Produces Nonce_NR;
Communication
Agent_O -> Agent_R : ;
if(Agent_O Receives Message_Timeout)
    Agent_O -> Agent_T : ma1;
    Agent_T -> Agent_O :{Message_Abort. ma1}PrKey_T;
    if(Agent_T Receives me1.me2)
        Agent_T -> Agent_O: {Message_Abort. me1}PrKey_T;
    endif
endif
Agent_R -> Agent_O : me2;
if(Agent_R Receives Message_Timeout)
    Agent_R -> Agent_T : me1.me2;
    Agent_T -> Agent_R : {me1.me2}PrKey_T;
    if(Agent_T Receives ma1)
        Agent_T -> Agent_O: {me1.me2}PrKey_T;
    endif
endif
Agent_O -> Agent_R : Nonce_NO;
if(Agent_O Receives Message_Timeout)
    Agent_O -> Agent_T : me1.me2;
    Agent_T -> Agent_O : {me1.me2}PrKey_T;
    if(Agent_T Receives ma1)
        Agent_T -> Agent_O: {me1.me2}PrKey_T;
    endif
endif
```

```
Agent_R -> Agent_O : Nonce_NR;
```

Chapter 8

Conclusion and Future Work

In this thesis, we presented a dedicated framework for the specification and verification of security protocols. The problem that we tackled was the design of a new security protocol verification methodology that would cover traditional security properties such as secrecy and authentication in addition to properties such as fairness and money atomicity that have not been fully formalized. We began the thesis by discussing issues regarding security protocols, cryptography and game semantics. Then, the topics of specification and verification of security protocols using formal methods were surveyed. We also discussed game semantics as a novel technique that gives an operational aspect to denotational semantics. We presented a model of security protocols that expresses both communication and computation steps of the protocol using a game semantics framework. This model was then used to ascribe functional and security semantics to protocols. The syntax we developed in order to specify the model is an enhanced version of the currently used standard notation for security protocols. We made the choice of keeping the syntax as close as possible to the standard notation so that it can be easily used by protocol designers and practitioners. Having defined the model, we introduced our logic to specify security properties of the model. Finally, we considered test cases for a number of protocols.

To sum up, we list below the contributions of this thesis:

- A game-theoretic model for security protocols based on game semantics.
- Semantic interpretation of security protocols as strategies over certain defined games.
- Syntactic and semantic definition of a logic to express a wide variety of security properties.
- Development of a software environment to validate research ideas and consider case studies.

As a future work, we propose the following:

- Adding the concept of time into our model to be able to express timestamps and timeouts.
- Enhancing the syntax and the semantics by adding more constructs such as conditionals for the actions of agents.
- Adding probabilistic concepts to the model such as the probability of attacks or the strength of encryption keys.

To summarize, our contribution to the state-of-the-art by this thesis is a formal framework that supports the verification of a wide range of security properties along with a software tool that serves as a testbed for the theoretical ideas that we presented.

Bibliography

[1] M. Abadi. Security protocols and specifications. In *Foundations of Software Science and Computation Structures: Second International Conference, FOSSACS '99*, volume 1578, pages 1–13. Springer-Verlag. Berlin Germany, 1999.

[2] M. Abadi. Security protocols and their properties. In F.L. Bauer and R. Steinbrueggen, editors, *Foundations of Secure Computation, 20th Int. Summer School, Marktoberdorf, Germany*, pages 39–60. IOS Press, 2000.

[3] M. Abadi and V. Cortier. Deciding knowledge in security protocols under (many more) equational theories. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*, 2005.

[4] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *POPL*, pages 104–115, 2001.

[5] M. Abadi and A. Gordon. A calculus for cryptographic protocols: The SPI calculus. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, 1997.

[6] M. Abadi and P. Rogaway. Reconciling two views of cryptography (The computational soundeness of formal encryption). *Journal of cryptology*, 15(2):103–127, 2002.

[7] M. Abadi and M. R. Tuttle. A semantics for a logic of authentication. In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing*, pages 201–216, 1991.

[8] S. Abramsky. Algorithmic game semantics. citeseer.ist.psu.edu/505714.html.

[9] S. Abramsky. Semantics of interaction: An introduction to game semantics. In *Proceedings of the 1996 CLiCS Summer School, Isaac Newton Institute, P. Dybjer and A. Pitts, eds. (Cambridge University Press)*, 1997.

[10] S. Abramsky, P. Malacaria, and R. Jagadeesan. Full abstraction for PCF. In *Theoretical Aspects of Computer Software*, pages 1–15, 1994.

[11] S. Abramsky and G. McCusker. Game semantics. citeseer.ist.psu.edu/abramsky99game.html, 1997. Lecture notes accompanying Samson Abramsky's lectures at the 1997 Marktoberdorf summer school.

[12] S. Abramsky and G. McCusker. Linearity, sharing and state: a fully abstract game semantics for idealized Algol with active expressions. *Theoretical Computer Science*, 197(1–2), 1998.

[13] P. Adao, G. Bana, and A. Scedrov. Computational and information-theoretic soundness and completeness of formal encryption. In *Proceedings of 18th IEEE Computer Security Foundations Workshop*, 2005.

[14] K. Adi. *Formal Specification and Analysis of Security Protocols.* PhD thesis, Universite Laval, 2002.

[15] K. Adi, M. Debbabi, and M. Mejri. A new logic for electronic commerce protocols. *Theor. Comput. Sci*, 291(3):223–283, 2003.

[16] S. Almuhammadi and C. Neuman. Security and privacy using one-round zero-knowledge proofs. In *IEEE International Conference on E-Commerce Technology*, pages 435–438. IEEE Computer Society, 2005.

[17] R. Alur, T. Henzinger, and O. Kupferman. Alternating-time temporal logic. *JACM: Journal of the ACM*, 49, 2002.

[18] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *Proc. 10th International Computer Aided Verification Conference*, pages 521–525, 1998.

[19] R. Anderson. Why cryptosystems fail. *CACM: Communications of the ACM*, 37, 1994.

[20] R. Anderson and R. Needham. Programming Satan's computer. *Lecture Notes in Computer Science*, 1000, 1995.

[21] N. Asokan, V. Shoup, and M. Waidner. Asynchronous protocols for optimistic fair exchange. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 86–99, 1998.

[22] F. Baader and T. Nipkow. *Term Rewriting and all that.* Cambridge University Press, Cambridge, 1998.

[23] E. Barker and J. Kelsey. Recommendation for random number generation using deterministic random bit generators. Technical Report 800-900, NIST, March 2007.

[24] D. Basin, S. Mödersheim, and L. Viganò. OFMC: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208, 2005.

[25] M. Baudet, V. Cortier, and S. Kremer. Computationally sound implementations of equational theories against passive adversaries. In *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP), volume 3580 of Lecture Notes in Computer Science*, 2005.

[26] A. Blass. A game semantics for linear logic. *Annals of Pure and Applied Logic*, 56:183–220, 1992.

[27] C. Boyd and W. Mao. On a limitation of BAN logic. In *EUROCRYPT: Advances in Cryptology*, 1993.

[28] S. Brackin. An interface specification language for automatically analyzing cryptographic protocols. In *Internet Society Symposium on Network and Distributed System Security*, 1997.

[29] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. Technical report, Digital Systems Research Center, 1989.

[30] I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *CSFW: Proceedings of The 12th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1999.

[31] R. Chadha, M. Kanovich, and A. Scedrov. Inductive methods and contract-signing protocols. In *SIGSAC: 8th ACM Conference on Computer and Communications Security*. ACM SIGSAC, 2001.

[32] R. Chadha, S. kremer, and A. Scedrov. Formal analysis of multiparty contract signing. *Journal of Automated Reasoning*, 36:39 – 83, 2006.

[33] J. Chroboczek. *Game Semantics and Subtyping*. PhD thesis, University of Edinburgh, 2003.

[34] E. Clarke, S. Jha, and W. Marrero. Using state space exploration and a natural deduction style message derivation engine to verify security protocols. In *International Conference on Programming Concepts and Methods*, pages 87–106, 1998.

[35] R. Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27(8):725–748, 1990.

[36] H. Comon and V. Shmatikov. Is it possible to decide whether a cryptographic protocol is secure or not? *Journal of Telecommunications and Information Technology*, 4:5–15, 2002.

[37] V. Cortier, S. Delaune, and P. Lafourcade. A survey of algebraic properties used in cryptographic protocols. *Journal of Computer Security*, 14(1):1–43, 2006.

[38] V. Cortier and B. Warinschi. Computationally sound automated proofs for security protocols. In *Proceedings of 14th European Symposium on Programming (ESOP'05), Lecture Notes in Computer Science*, 2005.

[39] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.

[40] C. Cremers. Feasibility of multi-protocol attacks. In *Availability Reliability and Security*, pages 287–294. IEEE Computer Society, 2006.

[41] M. Debbabi, N. A. Durgin, M. Mejri, and J. C. Mitchell. Security by typing. *STTT*, 4(4):472–495, 2003.

[42] N. Dershowitz and J. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science*. Elsevier Science Publishers, 1990.

[43] E. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[44] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[45] N. Durgin, P. Lincoln, and J. Mitchell. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12(2):247–311, 2004.

[46] J. Fabrega, J. Herzog, and J. Guttman. Honest ideals on strand spaces. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 1998.

[47] J. Fabrega, J. Herzog, and J. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, pages 191–230, 1999.

[48] U. Feige, A. Fiat, and A. Shamir. Zero-knowledge proofs of identity. *JCRYPTOL: Journal of Cryptology*, 1, 1988.

[49] T. Furon. A survey of watermarking security. In *International Workshop on Digital Watermarking (IWDW), LNCS*, volume 3710, 2005.

[50] J. Goguen and R. Diaconescu. An Oxford survey of order sorted algebra. *Mathematical Structures in Computer Science*, 4:363–392, 1994.

[51] J. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs.* Foundations of Computing Series. The MIT Press, Cambridge, MA, 1996.

[52] B. Goldburg, S. Sridharan, and E. Dawson. Design and cryptanalysis of transform-based analog speechscramblers. *IEEE Journal on Selected Areas in Communications*, 11(5):735–744, 1993.

[53] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and Systems Sciences*, 28(2):270–299, 1984.

[54] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive systems. *SIAM Journal of Computing*, 18(1):186–208, 1989.

[55] L. Gong, R. Needham, and R. Yahalom. Reasoning about belief in cryptographic protocols. In *Proceedings of the 1990 IEEE Symposium on Research in Security and Privacy*, pages 234–248, 1990.

[56] J. Goubault-Larrecq. A method for automatic cryptographic protocol verification (extended abstract). In *Proceedings of the Workshops of the 15th International Parallel and Distributed Processing Symposium*, volume 1800 of *Lecture Notes in Computer Science*, Cancun, Mexico, 2000.

[57] J. Halpern and R. Pucella. On the relationship between strand spaces and multi-agent systems. *ACM Trans. Inf. Syst. Secur.*, 6(1):43–70, 2003.

[58] A. Huima. Efficient infinite-state analysis of security protocols. In *Workshop on Formal Methods and Security Protocols (FLOC '99)*, August 1999.

[59] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II, III. *Info. and Comp.*, 163:285–408, 2000.

[60] J. Jürjens. Games in the semantics of programming languages. *Synthese (Elsevier)*, 133(1–2), October/November 2002.

[61] R. Kemmerer. Analyzing encryption protocols using formal verification techniques. *IEEE Journal on Selected Areas in Communications*, 7, 1989.

[62] R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *JCRYPTOL: Journal of Cryptology*, 7, 1994.

[63] A. Kerckhoffs. La cryptographie militaire. *Journal des Sciences Militaires*, IX, 1883.

[64] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, 1975.

[65] N. Koblitz. *A Course in Number Theory and Cryptography*. Springer-Verlag, second edition, 1994.

[66] S. Kremer and J. Raskin. A game approach to the verification of exchange protocols - application to non-repudiation protocols. In *Proceedings of the Workshop on Issues in the Theory of Security (WITS '00)*, 2000.

[67] S. Kremer and J. Raskin. Game analysis of abuse-free contract signing. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW'02)*. IEEE Computer Society Press, 2002.

[68] J. Laird. Full abstraction for functional languages with control. In *Logic in Computer Science*, pages 58–67, 1997.

[69] K. Lorenz. Basic objectives of dialogue logic in historical perspective. *Synthese (Elsevier)*, 127(1–2), April/May 2001.

[70] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. *Software - Concepts and Tools*, 17(3):93–102, 1996.

[71] G. Lowe. A hierarchy of authentication specification. In *CSFW*, pages 31–44. IEEE Computer Society, 1997.

[72] G. Lowe. Casper: A compiler for the analysis of security protocols. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, 1998.

[73] S. Lu and S. Smolka. Model checking the secure electronic transaction (SET) protocol. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE Computer Society, 1999.

[74] O. Markowitch, D. Gollmann, and S. Kremer. On fairness in exchange protocols. In *ICISC: International Conference on Information Security and Cryptology*. LNCS, 2002.

[75] W. Marrero, E. Clarke, and S. Jha. Model checking for security protocols. Technical report, Carnegie Mellon University, 1997.

[76] C. Meadows. Language generation and verification in the NRL protocol analyzer. In *Proceedings of the 9th Computer Security Foundations Workshop*, 1996.

[77] C. Meadows. Languages for formal specification of security protocols. In *CSFW '97: Proceedings of the 10th Computer Security Foundations Workshop (CSFW '97)*, page 96. IEEE Computer Society, 1997.

[78] C. Meadows. Formal methods for cryptographic protocol analysis: Emerging issues and trends. *IEEE Journal on Selected Areas in Communication*, 21(1):44–54, 2003.

[79] A. Menezes, P. van Oorschot, and S. Vanston, editors. *Handbook of Applied Cryptography.* CRC Press, 1996.

[80] J. Millen. CAPSL: Common authentication protocol specification language. In *Proceedings of the Workshop on New Security Paradigms*, 1997.

[81] J. Millen, S. Clark, and S. Freedman. The interrogator: Protocol security analysis. *IEEE transactions on software engineering*, SE-13(2), 1987.

[82] Robin Milner. *Communicating and Mobile Systems: The π-calculus.* Cambridge University Press, 1999.

[83] J. Mitchell. Finite-state analysis of security protocols. In *International Conference on Computer Aided Verification*, 1998.

[84] J. Mitchell. Multiset rewriting and security protocol analysis. In *Rewriting Techniques and Applications, LNCS 2378*, pages 101–120, 2002.

[85] J. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using murphi. In *IEEE Symposium on Security and Privacy*, 1997.

[86] D. Monniaux. Abstracting cryptographic protocols with tree automata. In *Sixth International Static Analysis Symposium (SAS'99)*, number 1694 in Lecture Notes in Computer Science, pages 149–163. Springer Verlag, 1999.

[87] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12), 1978.

[88] B. Neuman and S. Stubblebine. A note on the use of timestamps as nonces. *ACM Operating Systems Reviews*, 27(2), 1993.

[89] Peter Csaba Ölveczky and Martin Grimeland. Formal analysis of time-dependent cryptographic protocols in real-time Maude. In *IEEE International Parallel and Distributed Processing Symposium*, pages 1–8. IEEE, 2007.

[90] M. J. Osborne and A. Rubenstein. *A Course in Game Theory.* The MIT Press, 1994.

[91] C. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[92] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, pages 85–128, 1998.

[93] F. Petitcolas, R. Anderson, and M. Kuhn. Information hiding – a survey. *Proceedings of the IEEE (USA)*, 87(7):1062–1078, July 1999.

[94] B. Pierce. *Basic category theory for computer scientists.* The MIT Press, Cambridge, US, 1991.

[95] G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–256, December 1977.

[96] AVISPA Project. Automated validation of internet security protocols and applications. `http://www.avispa-project.org/`.

[97] R. Smullyan. *First Order Logic.* Springer-Verlag, 1968.

[98] M. Schmidt-Schauß. *Computational Aspects of an Order-Sorted Logic with Term Declarations*, volume 395 of *LNCS.* Springer-Verlag, 1989.

[99] B. Schneier. *Applied Cryptography.* John Wiley, 2 edition, 2001.

[100] C. Shannon. Communication theory of secrecy systems. *Bell Systems Technical Journal*, 28:657–715, 1949.

[101] V. Shmatikov and J. Mitchell. Finite-state analysis of two contract signing protocols. *Theoretical Computer Science*, 283, 2002.

[102] V. Shmatikov and U. Stern. Efficient finite-state analysis for large security protocols. In *CSFW: Proceedings of The 11th Computer Security Foundations Workshop.* IEEE Computer Society Press, 1998.

[103] G. Shu and D. Lee. Testing security properties of protocol implementations - a machine learning based approach. In *Proceedings of the 27th International Conference on Distributed Computing Systems ICDCS 07*, 2007.

[104] W. Stallings. *Cryptography And Network Security Principles and Practice.* Prentice Hall, Inc., third edition, 2003.

[105] P. Syverson. Adding time to a logic of authentication. In *CCS'93: Proceedings of the First ACM Conference on Computer and Communications Security*, pages 97–101, 1993.

[106] P. Syverson. A taxonomy of replay attacks. In *Computer Security Foundations Workshop*, pages 187–191, 1994.

[107] A. Tarigan. A survey in formal analysis of security properties of cryptographic protocol. Technical report, AG Rechnernetze und Verteilte Systeme, Universitaet Bielefeld, 2002.

[108] The AVISPA Team. HLPSL tutorial, a beginner's guide to modelling and analysing security protcols. `http://www.avispa-project.org/`, 2006.

[109] J.V. Tucker and J.I. Zucker. Computable functions and semicomputable sets on many-sorted algebras. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 5, pages 317–523. Oxford University Press, 2000.

[110] P. van Oorschot. An alternate explanation of two BAN-logic "failures". In *EUROCRYPT: Advances in Cryptology*, 1993.

[111] B. Warinschi. A computational analysis of the Needham-Schroeder-(Lowe) protocol. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW'03)*, 2003.

[112] W. Wechler. *Universal Algebra for Computer Scientists.* Springer, Berlin, 1992.

[113] T.Y.C. Woo and S.S. Lam. A lesson on authentication protocol design. *Operating Systems Review*, pages 24–37, 1994.

[114] J. Zhou, R. Deng, , and F. Bao. Some remarks on a fair exchange protocol. In *Lecture Notes in Computer Science 1751, Proceedings of 2000 International Workshop on Practice and Theory in Public Key Cryptography*, pages 46–57, 2000.