LOGIC BASED ANALYSIS OF SECURITY PROTOCOLS IN A UNIFIED VERIFICATION FRAMEWORK

CARLA LUCIA MIRONA MUNTEAN
UL ID: 0839132
MASTER OF ENGINEERING
UNIVERSITY OF LIMERICK

SUPERVISORS:
Professor TOM COFFEY
Dr. REINER DOJEN

Submitted to the University of Limerick
November 2012
Acknowledgment

I would like to take this opportunity to thank sincerely my supervisors Professor Tom Coffey and Dr. Reiner Dojen for their valuable support and guidance for the duration of this project.

My gratitude goes towards Robert and Cornelia Gyorodi from the University of Oradea: thank you for believing in me and supporting me through all my work.

The whole thesis is dedicated to the beloved memory of my grandparents: Popa Nica and Popa Ioan. Thank you for making my childhood as wonderful as it was.

Special thanks to all my family (Lucia, Miron, Fredy, Boby) and my friend, Maria Satmar. I’ll always remember your love, friendship, guidance and support. No matter where life takes me to, you will always be in my heart and memory
DECLARATION

I hereby declare that this thesis is entirely my work and has not been submitted to any other university.

Signed:                          Date: 27/11/2012

Carla Lucia Mirona Muntean

Student ID: 0839132
Abstract

Security protocols are one of the imperative steps in creating and ensuring the secure communication and information processing. Also known as cryptographic protocols or encryption protocols, they are used for secure application-level data transport achieved by using a security function and applying cryptographic methods. A wide variety of different forms of security go beyond the traditional goals of secure authentication, integrity and data confidentiality. Modern applications require more subtle properties like blind signatures used for digital cash, non-repudiation, secure digital time-stamping, etc. However, these protocols are vulnerable to different kinds of attacks and their design in achieving data safety and confidentiality has proven to be challenging and error prone. Informal and intuitive techniques have been proposed to verify such protocols. Several methods such as modal logics and state space exploration were developed to ensure formal verification and validation of security protocols.

This thesis presents the integration of a logic based verification technique into an existing verification framework. The existing framework unifies the verification of protocols through both state based techniques and modal logics. The integration is done through a translator that takes the input file specified in the language of the verification framework and translates it into the language of the logic-based verification technique.

The validity of the proposed translation/integration is demonstrated in an empirical study in which a set of protocols are specified and their verification results are compared using the old logic based verification engine and the new verification framework. Comparison of both verification results establishes their equivalence. This proves validity of the translation process between the two verification engines.
# Table of Contents

CHAPTER 1. Introduction

1.1 Verification of security protocols ................................................. 14

1.2 OSPSL – The Unified Verification Framework ................................. 15

1.3 Research Objectives ...................................................................... 16

1.4 Structure of thesis ......................................................................... 17

CHAPTER 2. Security protocols

2.1 Requirements for security protocols ................................................ 19

2.2 Cryptographic protocols ................................................................. 20

2.2.1 Symmetric cryptographic protocols ............................................ 21

2.2.2 Asymmetric cryptographic protocols ........................................... 22

2.3 Security in protocol design .............................................................. 23

2.3.1 Nonces .................................................................................. 23

2.3.2 Session keys ........................................................................... 24

2.3.3 Timestamps ............................................................................ 24

2.4 Attacks against security protocols .................................................. 24

2.4.1 Freshness attacks .................................................................... 25

2.4.2 Parallel session attacks ............................................................... 25

2.5 Summary ....................................................................................... 26

CHAPTER 3. Formal Logic Based Verifications

3.1 Formal verification of security protocols ............................................ 27

3.1.1 State Space Based Techniques .................................................... 28
3.1.2 Algebraic Term Rewriting Techniques .......................................................... 29
3.1.3 Theorem Proving Techniques ........................................................................... 29
3.1.4 Modal Logics .................................................................................................. 30
3.2 CDVT – A logic based verification language ....................................................... 31
  3.2.1 The Coffey Saidha logic .................................................................................. 32
  3.2.2 CDVT – A logic based verification language .................................................. 33
3.3 Summary ............................................................................................................. 36

CHAPTER 4. OSPSL - The Unified Verification Framework ........................................... 37
  4.1 Specification languages for security protocols ..................................................... 37
  4.2 The Unified Verification Framework ................................................................... 38
  4.3 Protocol specifications in OSPSL ........................................................................ 41
    4.3.1 Declaration Section ....................................................................................... 41
    4.3.2 Denotation Section ....................................................................................... 42
    4.3.3 Role Section ................................................................................................ 43
  4.4 Conclusions ....................................................................................................... 46

CHAPTER 5. Introduction to LEX and YACC ............................................................... 47
  5.1 LEX - A Lexical Analyzer .................................................................................... 47
    5.1.1 The Definition Section ................................................................................ 47
    5.1.2 The Rules Section ....................................................................................... 48
    5.1.3 The Subroutine Section ............................................................................... 48
    5.1.4 Overview on Lex ......................................................................................... 48
  5.2 YACC – Yet Another Compiler Compiler ........................................................... 49
CHAPTER 8. Conclusions................................................................................................................. 94

8.1 Future perspectives of the unified verification framework ................................................. 95

CHAPTER 9. References.................................................................................................................. 96

Appendix A – Semantics in OSPSL .............................................................................................. 103

Figure table of contents

Chapter 1:

• Figure 1.1: The unified framework for security protocol analysis

Chapter 2:

• Figure 2.1: Symmetric encryption
• Figure 2.2: Public key encryption

Chapter 4:

• Figure 4.1: Built-in classes in OSPSL
• Figure 4.2: Andrew’s Secure RPC Declaration Section in OSPSL
• Figure 4.3: Denotation Section in OSPSL for ZG protocol
• Figure 4.4: Role definition for principal A in Andrew Secure RPC protocol
• Figure 4.5: Use of axiom sub-section in the Message section for the ASK protocol

Chapter 5:

• Figure 5.1: Structure of a Lex file
• Figure 5.2: Structure of a YACC file
• Figure 5.3: Simple rule in YACC
• Figure 5.4: Recursive rule in YACC
Chapter 6:

- Figure 6.1: CDVT’s Project Architecture
- Figure 6.2: OSPSL to CDVT translator process

Chapter 7:

- Figure 7.1: Kao Chow Protocol
- Figure 7.2: Verification results for the manually written specifications for the Kao-Chow protocol
- Figure 7.3: Results for the manual written specifications of the amended Kao Chow Protocol
- Figure 7.4: OSPSL description of principal A’s role in Kao-Chow protocol
- Figure 7.5: Results of the generated specs for the Kao-Chow protocol specifications in OSPSL
- Figure 7.6: Verification results for manually written specification of the amended Kao Chow protocol
- Figure 7.7: BCY protocol
- Figure 7.8: CDVT verification results for the manually written specifications of the BCY protocol
- Figure 7.9: Verification results for manually written specifications of the CDFBCY protocol
- Figure 7.10: OSPSL verification results for the BCY protocol
- Figure 7.11: OSPSL verification results for the CDFBCY protocol
- Figure 7.12: Zhou-Gollmann protocol
- Figure 7.13: Assumption section for the CDVT manual specifications of the ZG protocol
- Figure 7.14: Goals section for the CDVT manual specifications of ZG protocol
- Figure 7.15: Verification results of the manual written specifications for the ZG protocol
- Figure 7.16: Verification results of the manual written specifications for the amended ZG protocol
- Figure 7.17: Declaration section for the original ZG protocol in OSPSL
Figure 7.18: DENOTE section in OSPSL for the original ZG protocol
Figure 7.19: Principal A’s role in OSPSL for ZG non-repudiation protocol
Figure 7.20: CDVT translated specification for ZG protocol
Figure 7.21: Verification results for the OSPSL translation of ZG protocol
Figure 7.22: Amended version of the ZG protocol
Figure 7.23: Verification of the ZG amended protocol generated specs in OSPSL

**Table of contents for the tables in the thesis**

**Chapter 3:**
- Table 3.1: Expressions in CDVT
- Table 3.2: Composite Data Construction in CDVT
- Table 3.3: Statement Construction in CDVT

**Chapter 6:**
- Table 6.1: Equivalence of types between OSPSL and CDVT
- Table 6.2: Equivalence of logical operators in OSPSL and CDVT
- Table 6.3: Equivalence of authenticate operators in OSPSL and CDVT
- Table 6.4: Content of the symbolic table of the translator

**Chapter 7:**
- Table 7.1: Attacks found on a given set of protocols using OSPSL and CDVT
**DVD Table of contents**

- OSPSL protocol specifications
- Translator source files
- Electronic copy of present thesis
The internet has revolutionized the world in the latest years through computing and easier world communication for the purpose of development. The availability of the internet, along with home computers and laptops accelerated the process and the adoption of browsers and World Wide Web technology have made the world nowadays to allow user easier access to information, linking the whole globe. However, the global network used is an open and completely insecure medium. With the increased awareness of the Internet, security problems have been brought to the fore. Data security is not only extremely important, but more technically complex than in the past. Authentication, encryption, integrity and safety are important in cultivating and promoting data security.

Security protocols define the process of exchanging secure data among the participants. Also called cryptographic protocols, they are abstract protocols that describe how an algorithm should be defined, using cryptographic methods and apply security-related functions to maintain data authenticity. Without such authentication and security, a malicious attacker could impersonate any of the participants in a protocol and access the network. Modern websites and new mobile systems use more and more complex cryptographic techniques to ensure data confidentiality and safety that go beyond then traditional goals of integrity, confidentiality and safe authentication. Non-repudiation, blind signatures or key encryptions are a few of the complex methods use to ensure a secure communication among the participants.

The biggest challenge for the design of a security protocol is to correctly achieve all goals and maintain data safety under all possible scenarios which imply the existence of an intruder that could maliciously attack the protocol and gain access to data exchanged.
1.1 Verification of security protocols

The verification of cryptographic protocols has become more and more complex over the years. The increasing number of publications that describe various common protocols having numerous flaws, found sometimes several years after their original publication, highlights the complexity of the protocols design process and proves its susceptibility to errors.

Formal and informal techniques have been proposed for cryptographic protocols verification. Traditionally the protocols were analyzed using informal and intuitive techniques. Informal techniques, such as safe design practices proposed for protocol design [AN96] can be used for lower complexity protocols, but have been proven to be less effective in complex message exchanging, where flaws can pass undetected.

Formal methods have been used to analyze security protocols with various degrees of complexity. In the world of formal verification, there are different directions in analyzing the cryptographic protocols. Common approaches for the formal verification techniques are based on state-machines [SM95] [Hui99] [GR02] [HCTD04] and modal logics [CDF03, KWA94, Bra00]. The technique of logic based verification has been accredited to Burrows, Abadi and Needham [BAN89], developers of the BAN logic. Several logics have been developed after on the basis on BAN: GNY [GNY90], CS [CS97] and ZV [ZV01]. All these logics are used to generate concise proof and they all have identified flaws in protocols that were previously considered secure.

As defined by Meadows [Mea03] a formal method combines a language defined to model cryptographic protocols and their security properties, together with an efficient procedure which determines if the model truly satisfies the expected properties. Gritzalis, Spinellis and Georgiadis [GSG99] classify 3 types of protocol analysis: inference-construction, attack-construction and proof-construction.

Inference-construction techniques are based upon modal logics by using the evolution of knowledge and beliefs within a system to show that certain conditions are satisfied. Attack-
construction methodologies try to discover vulnerabilities using algebraic properties of a protocol’s algorithms.

Proof-construction techniques model the computations performed in a protocol and introduces security properties as theorems verified after using automated theorem checkers. Gritzalis, Spinellis & Georgiadis [GSG99] suggested that proof-construction techniques replace state space exploration with theorems about these searches.

Meadows considers that inference-construction techniques are generally weaker than attack-construction methods, because they operate at a higher level of abstraction [Mea00, Mea01]. Paulson [Pau98] sustains that attack-construction and proof construction techniques are complementary: attack-construction techniques are typically easy to use and can provide an assurance that a model satisfies the security criteria, while proof-construction methodologies are more complicated, but allow a more complex usability thorough analysis.

A primary objective of this thesis is to introduce the modal logic techniques into a new tool for the verification of security properties during protocol development. Verification using formal methods of simple protocols such as the authentication protocol of Needham Schroeder [BAN89], which had been considered secure for over a decade, exposed flaws that could be vulnerable to replay attacks. This proves the fact that protocols, simple or complex, are difficult to verify using intuitive techniques.

### 1.2 OSPSL – The Unified Verification Framework

OSPSL is a high-level unified framework dedicated to describe abstract features of varied kinds of security protocols. [Tia07]. The aim of this language is to facilitate the development of security protocols using formal techniques. It is a novel approach regarding the analysis of cryptographic protocols. The framework treats security protocols as a set of collaborating entities and tries to use techniques specific to the object-oriented approach. A special type in the new framework can make use of both logic-based techniques and state-based techniques.
A diagram of this framework is presented below in Figure 1.1.

![Diagram of the unified framework for security protocol analysis](image)

Figure 1.1: The unified framework for security protocol analysis

### 1.3 Research Objectives

The objective of this thesis is to integrate logic based verification techniques into a verification framework by providing an automated translator from the framework to a logic-based verification tool.

An additional part of the main objective is to correctly map the syntax of the high level proposed unified framework to the requirements of building a reasonable modal logic approach that could be analyzed through the proposed logic based verification tool.

As a conclusion, the aim of this thesis is to:

- Study logic-based verification of security protocols
Integrate a logic-based tools in already existent unified verification framework OSPSL by

- Provide a mapping between the high level way of expressing protocols in OSPSL and the way protocols are expressed in CDVT
- Implementing a translator from OSPSL to the language of the logic.

Perform an empirical study on the implemented translator

- Verify several security protocols using the implemented translator
- Compare results against previous CDVT verifications

1.4 Structure of thesis

This chapter presents the structure of this thesis.

- **Chapter 2** introduces different kinds of cryptographic protocols and briefly states the attacks that can be found on these protocols

- In **Chapter 3** the formal logic based verification of security protocols is outlined. The logic based verification tool CDVT is introduced and its syntax is explained.

- In **Chapter 4** a review of the OSPSL verification framework is detailed

- **Chapter 5** introduces the Lex and Yacc parser and compiler, as being the methods used to create the translation between the old logic based verification engine and the OSPSL unified verification framework

- **Chapter 6** presents the process of building the translator from the unified verification framework to CDVT. The architecture of the project is described and the translation process is expressed in detail.
Chapter 7 Validates the translation from the unified verification framework OSPSL to CDVT.

Chapter 8 reviews the present thesis and summarizes its objectives, giving some conclusion and possible improvements that could be done on the unified framework in the future.
CHAPTER 2. Security protocols

Cryptographic protocols are small algorithms that aim to provide a secure communication over a public communication network. The possibility of an intruder to interfere within exchanged messages has made the design of security protocols notoriously difficult and error-prone. An important example would be the Needham-Schroeder public key protocol [NS78] which was intended to provide mutual authentication between the two parties involved. The protocol was considered secure by experts for over a decade before Lowe [Low96] discovered a weakness. This highlights the necessity of protocol verification and the need for automated support to overcome the old manual verifications.

2.1 Requirements for security protocols

In order to be considered secure, cryptographic protocols are required to satisfy classical requirements (such as secrecy and authentication) and more complex properties such as non-repudiation.

The security requirements were classified generally as reachability properties.

- **Confidentiality (Secrecy)** ensures that the information exchanged between the participants should be restricted only to honest parties of the communication. No one else is allowed to know secret information exchanged in the protocol run.

- **Authentication** guarantees that an initiator of a session communicates indeed with the expected responder in the same session. A third party that intends to impersonate either side is strictly forbidden. Authentication is provided by means that have to ensure the source and destination of the message exchanged and is usually achieved through certificates or digital signatures. Secure authentication is a major priority when
establishing secure communication. Authentication refers to *entity authentication* and *data origin authentication*. The first one is achieved through a protocol that proves to one party involved about the second’s identity. It is very important to be known that the second party was active at the time the exchanged data was sent. *Data origin authentication* techniques provide to the receiver the evidence about the identity of the sender.

- **Data freshness** is an important requirement as the content of the message exchanged should belong to the current run of the protocol.

- **Data integrity** is the property that ensures should not be modifiable.

- **Non-repudiation** is the property that doesn’t allow any of the participants to deny having been involved in the protocol run.

- **Non interference** requires that the secret information shared during the protocol run will not be used by intruders.

- **Fairness** property requires that no protocol participant should gain an advantage over the other participants involved.

With the growth of e-commerce, additional properties are required for almost all protocols:

- **Atomicity** considers that all operations in an electronic transaction are either complete or fully abort. An intermediate state is not allowed.

### 2.2 Cryptographic protocols

Protocols using cryptographic mechanism are called usually cryptographic protocols. Their goal is to enable parties implied in the protocol run to communicate securely over an insecure network. They require that the parties agree on keys related to their communication. These are
also called *Key Exchange Protocols*. Depending on the key used in exchanging the messages, there are two kinds of cryptography: symmetric cryptography and asymmetric cryptography.

### 2.2.1 Symmetric cryptographic protocols

Traditional cryptography first used symmetric algorithms where for both encryption and decryption is used a common key, called *shared-key*. The *key distribution center* sends keys over the secure channel to the encrypted side, while the information exchanged is encrypted and becomes *ciphertext*.

A Trusted Third Party (TTP) also known as a Key Distribution Center is often necessary. It sends secret keys through secure channels. Cleartext is encrypted with these keys and becomes ciphertext. In the decrypt part, the ciphertext is decrypted using the same secret key send by TTS and becomes again cleartext.

The symmetric encryption is presented in *Figure 2.1*.

![Figure 2.1: Symmetric encryption](image-url)
2.2.2 Asymmetric cryptographic protocols

An alternative to the symmetric cryptography is public-key cryptography. It refers to a cryptographic system which uses two mathematically linked keys. One of the two keys is used to encrypt the information wanted to be secretly exchanged, while the other key is used to decrypt the message. Neither key can be used to do both actions. One of the two keys is kept secret by a certain agent and is referred to as the "private" key of this agent. This private key represents the identity of its owner.

The second key, called the "public" key, is made available to the participants of the protocol. The cryptographic approach usually uses asymmetric key algorithms, hence the name of “asymmetric key cryptography”. By publishing the public key, the producer gives the power to anyone who gets its copy to produce messages only he can read. Unlike symmetric key algorithms, the public key ones do not require a secure initial exchange of one or more secret keys between the participants. The use of these algorithms also allows authenticity of a message to be checked by creating a digital signature of a message using the private key, which can be verified and read using the public key.

Figure 2.2 presents the public key encryption.

Figure 2.2: Public key encryption
There are several protocols in this category, such as IKE (Internet Key Exchange protocol), SET (Secure Electronic Transactions Protocol), TMN protocol, and the famous Needham-Schroeder public-key authentication protocol. The last one has been taken as a paradigmatic example for analysis by many verification techniques. The protocol aims to provide mutual authentication.

### 2.3 Security in protocol design

To achieve the security property that they should fulfill, protocols use different mechanisms.

#### 2.3.1 Nonces

In cryptographic message exchange, a nonce is an arbitrary number used only once to sign a cryptographic message belonging to one protocol session. Usually it is a random or pseudo-random number issued in an authentication protocol to ensure that old communications cannot be reused in a replay attack. A nonce is known only by its generator and it is used to verify the authentication and freshness of a message.

When an agent A wants to initialize a communication with an agent B, A may generate a nonce and encrypts it together with the message to be exchanged through a set of encryption methods. The methods used should ensure that only B can decrypt the message. After sending all this information, A waits for a response. If someone returns the same nonce in a reply, A can retrieve it and understand that the reply really belongs to the current protocol run and the originator of the message is B.

Nonces are one of the most common methods used by protocol designers. They are used in HTTP digest access authentication to calculate an MD5 digest of the passwords. A nonce may be used also to ensure security in a stream cipher.
2.3.2 Session keys

A session key is asymmetric key used once for encrypting messages in a communication session. Some protocols only use session keys, without nonces, such as TMN protocol [LR97]. This is a protocol for digital mobile communication, where each user communicates with another one via a network center.

2.3.3 Timestamps

Timestamps are used to ensure freshness of messages exchanged during the protocol run. A timestamp is a sequence of characters which expose the date or time when a certain message was sent. A trusted timestamp is usually generated by a Trusted Third Party (TTP) acting as a Time Stamp Authority (TSA). It is used to express the validity of a data before a certain time. Its creation is based on digital signatures and hash function. First, a hash is calculated from the data to be sent and this hash is sent to the TTP. The TSA concatenates a timestamp to the hash and builds a new hash out of this concatenation. The latest hash is digitally signed with the private key of the TTP and the signed hash and the timestamp are sent back to the timestamp requester.

2.4 Attacks against security protocols

Cryptographic protocols are designed to ensure security services over insecure networks. It is important that a security protocol correctly achieves all its goals.

An attack is an action that exploits the weaknesses existent in the design of a security protocol and allows an entity to compromise the security of the protocol. The attacks describe all the steps the malicious intruder must take in order to compromise the reliability and safety of the exchanged information during the protocol run.

The external entity who tries to gain these kinds of advantages over the weakness of the protocol is called an intruder. Dolev and Yao [DY83] introduce the first formal model of an intruder as being constrained by the methods imposed by the cryptographic system. The communication is
made over an insecure channel, but the attack exploits only the weaknesses of the security protocol and not of the cryptographic system. The intruder has a complete control over the insecure networks and can eavesdrop, insert, delete, modify, intercept and synthesize messages. The intruder can be a *passive intruder* when he eavesdrops on the communication channel or an *active intruder* when he performs a certain action inside the protocol (insert, delete or delay messages exchanged).

An attack over a protocol can be performed also by an entity or a group of entities participant in a protocol run. These parties are called *insiders* or *dishonest principals* and their goal is to gain access over the other participants in the protocol.

Several classes of attacks were built over the years [BM03] [BAN90] [Low95] [Low 96] [DS81] [Aur97] [NKW07] [HLLKC95] [HLS00].

2.4.1 Freshness attacks

A *freshness attack* is the attack where an intruder maliciously uses the messages exchanged in previous protocol sessions. It is the most common attack on authentication and key-establishment protocols. If the messages exchanged in an authentication protocol do not carry enough freshness identifiers, an intruder can authenticate himself and copy the messages exchanged in the previous runs of the protocol. Examples of replay attacks include [BM03][BAN90][DS81][Aur97].

2.4.2 Parallel session attacks

A *parallel session attack* requires the existence of parallel execution of multiple protocol runs. The intruder uses messages from one session to synthesize messages in another session. Examples of parallel session attacks on security protocols include [BM03] [Low95] [Low 96] [NKW07]. There are many forms of parallel session attacks:

- **Man-in-the-middle-attack** is a form of eavesdropping when the attacker makes random connections between the victims relaying messages between them and
making them believe they are talking with each other over a secure connection. The attacker is able to intercept the messages exchanged between the honest participants and is able to inject new ones. A man-in-the-middle attack can succeed only if the attacker can impersonate the participants and is a lack of mutual authentication between the participants.

- **Multiplicity attack** is a parallel session attack where the principals disagree on the number of runs they have successfully established with each other.

- **A type flaw attack** involves the replacement of a component included in one message with another message, of a different type by an intruder. A type flaw attack occurs when a recipient accepts a message as valid, but interprets the bit sequence differently from the creator of the message.

- In an **oracle attack** the intruder starts a new run of the protocol and uses one of the participants as an oracle for appropriate answers to challenges in the first protocol run.

### 2.5 Summary

Security protocols are the critical necessary elements to build a secure communication, assuring its authenticity, confidentiality and integrity. Apart from their exchanged messages and differences of behavior depending on the network they run on, protocols must fulfill their goals under all possible scenarios.

Existing flaws in the protocol design can be maliciously used by an intruder to attack the protocol and compromise the security property all protocols must achieve. The attack describes the steps an intruder makes in order to secretly defeat the reliability and goals of the protocol. Depending on the steps performed, more attacks are presented and analyzed.
CHAPTER 3. Formal Logic Based Verifications

Security protocols are built on lower level cryptographic algorithms. Designing security protocols to be susceptible to attacks has proven to be a challenging and error prone activity. The three main phases in the design of security protocols are similar with the necessary steps required for designing other systems: specification, analysis and implementation. In the specification phase, the requirements and parameters of the security protocol are defined. Once the objectives and goals of the protocol are defined, the set of exchanged messages and communication steps are developed.

3.1 Formal verification of security protocols

In the analysis phase, the security protocol is analyzed using different techniques (self practice methods or formal methods). The last phase imposes the build of the security protocol to be used for a real system. The implementation phase can take place if and only if the protocol to be implemented has successfully been through the analysis phase where all the flaws and weaknesses that make it vulnerable to attacks have been corrected.

The analysis of the protocol can be done using formal or informal methods. From the informal methods used, the most common are safe practice design methods [AN96]. However, the absence of formal verification has been proven to be risky and leading treacherously to weaknesses remaining undetected [Ros95, NC03, CDF03]. As an example is the BCY protocol developed by Beller, Chang and Yacobi [BCY93] demonstrated the feasibility of public-key cryptography in mobile communications. Later, weaknesses were discovered by Carlson [Car94]. Mu and Varadharajan [MV96] corrected Carlsons version and another derivate of the BCY protocol was published. Horn, Martin and Mitchell [HMM02] identified weaknesses also in Mu and Varadharajan’s version. In 2003, Coffey, Dojen and Flanagan [CDF03] published formal
verifications on the original BCY protocol and its versions and found a weakness present in all versions of the BCY protocol.

Another example would be the result obtained by Lowe [Low97] who, using Hoare’s CSP algebra [Hoa85] and CSP model checker [Ros95] detected an attack on the Needham Schroder public key protocol [NS78], which had been thought to be secure for almost 20 years.

Both these case studies underline the fact that the design process of security protocols is an activity that has been proven to be hard, complex and error prone. It is also showing that the formal verifications is “an imperative step in the design of security protocols” [DC05].

Formal verification methods can be used with various techniques [CDF03], including: state-machines, algebraic term rewriting, theorem proving and modal logics.

This thesis focuses on the translation method from the unified verification framework to a logics based verification tool and thus the modal logics is presented in more detail. A short description of all the other methods is presented, along with a short comparison between the techniques, but their detailed description is outside the scope of the thesis.

### 3.1.1 State Space Based Techniques

State machines can be used to analyze protocols using a method well known as the reach ability analysis technique [WES78] which requires that the global state of the system is expressed in each transaction. Each global state is analyzed and when discovered a state where an attacker gains access to secret information, the protocol is considered to be insecure.

The logic based analysis and model checking ([CJM00], [BMV03], [AC04]) are complementary methods. The logic based approach has the advantage of better scaling with the system size and if the result is correct, then the proof holds under all circumstances (independent of the intruder model). On the other hand, model checkers search for flaws (invalid variants) through every possible state of the system. If the complexity of the system increases, the model checking
becomes impractical (the state space explosion problem) and several strategies are needed to make the analysis feasible.

### 3.1.2 Algebraic Term Rewriting Techniques

The algebraic term rewriting techniques \[BHK06\], \[BHK09\], \[BN02\] used for the formal verification of security protocols are a similar approach of the state machine technique. The state machine techniques begin with an insecure state and try to show that there is no existent path from the original state to the insecure state. On the opposite side, the algebraic term rewriting technique starts with the specification of the initial state and an insecure state is shown as not able to be reached. Although they are related, the algebraic term rewriting technique is considered to be an improvement of the other as it can lead to discover more weaknesses.

The weakness of the algebraic rewriting technique is that only actions represented by the protocol are considered. Eavesdropping and replay attacks can be easily found, but the results fail when an intruder tries to introduce new messages in the conversation. A reason why this method is not so used is the fact that it requires a state of complexity which reduces the interest of the protocol designers.

### 3.1.3 Theorem Proving Techniques

The manually generated proofs for verifying security protocols are difficult to follow and use. Several tools for theorem proving techniques have been developed to generate the formal proof of mathematical prepositions. These tools expect as an input mathematical statements which are proved step by step with the user being able to direct the proof at any stage. The method relies on the fact that a security protocol can be specified precisely and special formulas of correctness can be defined. As a result, the statement saying that a protocol is secure can be expressed in mathematical terms. These statements are redirected to a theorem prover and the search for the proof can be followed. The theorem proving technique is better at demonstrating the correctness of a protocol and it requires the user to follow the right path throughout the whole process. The
usability of the technique relies on the user’s intelligence, intuition, practice and good mathematical skills. Despite the complexity required, theorem provers can be considered a huge advantage for protocol designers when combined also with modal logic techniques. [Coh00], [Paul98], [Schn98].

3.1.4 Modal Logics

The technique of logic-based formal verification was born due to Burrow, Abadi and Needham, developers of the BAN logic [BAN90]. Several other logics followed and have been developed on the basis of BAN: Li Gang, Roger Needham and Raphael Yahalom [GNY90], Syverson and van Oorschot [SO94]; Coffey Saidha [CS97]; Zhang and Varadharajan [ZV01]. All these logics have identified a number of flaws in protocols previously considered secure and brought several improvements over BAN, by being applicable to a wider range of protocols. Throughout this thesis, the logic used for protocol verification will be the Coffey Saidha logic.

The process of logic based formal verification involves the following steps:

1. **Formalization of the protocol messages**: is the transformation of the protocol specifications into the language of the logic. Each protocol message is expressed as a logical formula.

2. **Specification of the initial assumptions**: involves the formally specification of the initial beliefs and possessions held at the beginning of the protocol run by all the principals involved in the communication process.

3. **Specification of the protocol goals**: refers to expressing the desired protocol goals to be achieved in the language of the logic, in terms of possessions and beliefs help by the all involved parties, after the protocol run.

4. **Application of the logical postulates**: involves checking if the desired protocol goals can be derived from the initial assumptions and protocol steps, using the logical postulates from the proposed logic.
If at least one of the proposed goals fails, the protocol will be verified as “false”. The failure will point to either an existing weakness in the protocol design or an error in the protocol formalization process. In case of the first, the protocol has to be redesigned and verified again.

A successfully verified protocol can be considered secure within the scope of the logic.

Security protocols are represented by a set of predicates which represent the protocol’s initial
- assumptions \( A = A_1, A_2, ..., A_A \)
- steps \( S = S_1, S_2, ..., S_s \)
- goals \( G = G_1, G_2, ..., G_G \)

All these predicates need to be expressed in the language of the logic. Apart from all predicates, the language must contain a set of inference rules \( R = R_1, R_2, ..., R_R \). Each rule has a left hand side (LHS) and a right hand side (RHS). The LHS is the conjunction of valid predicates and the RHS has to be derived from the LHS, meaning \( LHS \Rightarrow RHS \).

Proving that a protocol is verified in the scope of the logic becomes equivalent with proving that the set of goal \( G \) using the set of inference rules \( R \) and the set of initial assumptions \( A \) and steps \( S \). Proving a goal \( G_N \) using a finite set of valid predicates \( \Sigma = A \cup S \), and a finite set of inference rules \( R \). Not being able to prove a goal means that:

\[
\forall R' \in R \text{ such that } (\text{applying } P \text{ to } \Sigma \Rightarrow G)
\]

and leads to the conclusion that there exists a flaw or a weakness in the protocol that could make it susceptible to an attack. The exact flaw that makes possible the existence of the attack is not directly identified.

### 3.2 CDVT – A logic based verification language

Logic is considered to be the philosophical study of valid reasoning. Over the years, different kinds of logic were developed: the term “informal logic” refers to the study of natural language
arguments, while “formal logic” denotes the logic that relies on inference, meaning the act or process of deriving logical conclusions from premises known or assumed to be true.

At the beginning, logical methods proposed for formal verification of security protocols were based on logic of belief [BAN89], [GNY90], [GS90] [KG91] and logic of knowledge [Bie90],[Syv90].

The logic of beliefs is considered useful in evaluating the reliability and trust of a protocol, while the logic of knowledge is useful for proving the security of a protocol. Most proposed logics of beliefs offer a limited scope for their application by being able to analyze only authentication protocols. The logics of knowledge offer a good verification on a high number of security protocols, but they require high level skills to be understood and thus they lack flexibility in expressing complex protocols.

3.2.1 The Coffey Saidha logic

The Coffey Saidha logic [CS97] was initially proposed to be used on public-key cryptographic protocols. The technique combines both logics: the modal logic of knowledge and the logic of beliefs.

It provides a belief operator and two knowledge operators:

- propositional knowledge operator which deals with the knowledge of statements and facts
- predicate which deals with the knowledge of objects (cyphertext, cryptographic keys, messages exchanged, etc)

Axioms are used to describe the low level properties of cryptographic communication such as encryption and decryption, transmission and reception of a message, the ability of an entity to encrypt a message, etc.

The logic considers the following axioms:
1. The protocol takes place in a communication environment which is considered to be hostile. The data communication system is reliable and messages loss and transmission errors are happening only with interference from a malicious attacker.

2. The public-key cryptosystem is ideal: the encryption and decryption functions are non-invertible, without the proper knowledge of the pair key and easily invertible with proper knowledge of the appropriate pair key.

3. A public key used in the protocol is considered to be valid if its validity period didn’t expire and its corresponding secret key is known only by its owner.

4. If an entity is able to encrypt/decrypt a data exchanged during the protocol run, then the entity must know the data.

### 3.2.2 CDVT – A logic based verification language

The CDVT verification engine [DLC08] is an automated system that implements a modal logic of knowledge and belief using Layered Proving Trees [DC05] and a parser to read in the protocol specification from a text file, which contains the formal specification of the protocol to be verified as follows: elements follow regular expressions as expressed in Table 3.1, data components are build according to Table 3.2 and statements construction rules are defined in Table 3.3.
Composite data components are constructed according to the second table, where elements follow the regular expressions as given in Table 3.1 and Data represents an arbitrary data element (either atomic unit or composite data)

<table>
<thead>
<tr>
<th>Component</th>
<th>Regular Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Principal</td>
<td>[AB-EIJLMQRSTUVWXYZ][A-Za-z0-9_]*</td>
</tr>
<tr>
<td>Trusted Principal</td>
<td>TTP[A-Za-z0-9_]*</td>
</tr>
<tr>
<td>Symmetric Key</td>
<td>K[a-z][a-zA-Z0-9_]*</td>
</tr>
<tr>
<td>Public Key</td>
<td>K[a-z][A-Za-z0-9_]*Pub</td>
</tr>
<tr>
<td>Private Key</td>
<td>K[a-z][A-Za-z0-9_]*Priv</td>
</tr>
<tr>
<td>Nonce</td>
<td>N[a-z][A-Za-z0-9_]*</td>
</tr>
<tr>
<td>Timestamp</td>
<td>TS[a-z][A-Za-z0-9_]*</td>
</tr>
<tr>
<td>Function</td>
<td>F[A-Za-z0-9_]*</td>
</tr>
<tr>
<td>Hash</td>
<td>H[A-Za-z0-9_]*</td>
</tr>
<tr>
<td>Binary Data</td>
<td>[a-z][A-Za-z0-9_]*</td>
</tr>
</tbody>
</table>

Table 3.1: Expressions in CDVT [DLC08]

<table>
<thead>
<tr>
<th>Composite Data</th>
<th>Textual Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concatenation</td>
<td>Data,Data</td>
</tr>
<tr>
<td>Group Element</td>
<td>(Data)</td>
</tr>
<tr>
<td>Symmetric Encryption</td>
<td>(Data)Data</td>
</tr>
<tr>
<td>Public Key Encryption</td>
<td>(Data)Kpub</td>
</tr>
<tr>
<td>Private Key Encryption</td>
<td>(Data)KPriv</td>
</tr>
<tr>
<td>Function of Data</td>
<td>F(Data)</td>
</tr>
<tr>
<td>Hash of Data</td>
<td>H(Data)</td>
</tr>
</tbody>
</table>

Table 3.2: Composite Data Construction in CDVT [DLC08]
Statements are defined as by the rules presented in Table 3.3, where elements follow the regular expressions as given in Table 3.1. *Data* is either an atomic data unit or a composite data, *operator* is any of “send”, “receive” or “possess”, “i” indicates the indexed discrete time and *Statement* represents an arbitrary statement.

<table>
<thead>
<tr>
<th>Principal operator at[i] Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Principal know at[i] Statement</td>
</tr>
<tr>
<td>Principal believe at[i] Statement</td>
</tr>
<tr>
<td>Principal know at[i] NOT ( Statement )</td>
</tr>
<tr>
<td>Principal believe at[i] NOT ( Statement )</td>
</tr>
<tr>
<td>( Statement )</td>
</tr>
<tr>
<td>( Statement AND Statement )</td>
</tr>
<tr>
<td>( Statement IMPLY Statement )</td>
</tr>
</tbody>
</table>

Table 3.3: Statement Construction in CDVT [DLC08]

Each line of the textual specification file is preceded by a label. Assumptions are labeled “An”, protocol steps are labeled “Sn” and protocol goals are labeled “Gn”, where n numbers each group sequentially. Every line must be closed with a semicolon (‘;’) and comments are introduced by a double forward slash (‘//’ – C++ style comments).

The inference rules provided are the standard rules of natural deduction. The axioms of the logic express the fundamental properties of public-key cryptographic protocols such as the ability of a principal to encrypt/decrypt based on knowledge of a cryptographic key. The axioms also reflect the underlying assumptions of the logic.
3.3 Summary

This chapter shortly introduces the formal methods for analysis of security protocols: state machines, algebraic term rewriting techniques, theorem proving techniques and modal logics.

The logic-based approach is described in more detail as being the most common methods used in formal analysis. The chapter introduces also the CDVT logic based verification language and reviews in a few words the Coffey Saidha logic [CS97].
CHAPTER 4. OSPSL - The Unified Verification Framework

Formal specification languages, such as Ina Jo [SA85], Z [Spi89], RAISE [XJ11], LOTOS [BD89] [Eij89] and Estelle [Dia89], have been developed to model the functionality of computer-based systems. With the development of formal methods for security protocols, some of these languages, such as Ina Jo [SA85] and LOTOS [BD89], have also been applied to specify security protocols. The use of these general-purpose specification languages in specifying security protocols was limited and ineffective. Specification languages dedicated to security protocols have thus been developed to facilitate the application of formal methods on security protocols.

4.1 Specification languages for security protocols

Formal specification languages for security protocols provide a rigorous mathematical basis so that formal methods can be used to design and analyze the protocols.

A security protocol specification language encourages an abstract view of the protocol, which separates specification from implementation, focusing on what the protocol should achieve rather than how to accomplish it.

A formal specification language for specifying security protocol generally requires more specific features than a common specification language. It should easily be able to express features such as protocol participants (trusted third party, server, normal entities), encryption-keys, timestamps, nonces, hash functions, etc. It should have an interface with formal techniques so that the specified protocol can be utilized by formal methods in design and verification of security protocols. For example, a protocol specification language suited for logic-based formal methods should have the facility to formally specify the initial assumptions, the message exchanges and the protocol goals.
For a protocol designer, to be able formally specify a security protocol requires an extensive knowledge of the formal technique being used and familiar knowledge about the formal methods. For easier specifications, high-level languages were proposed. These provide an intermediate representation of security protocols and can be translated through automated translators into the formal language of the formal method: ISL [Bra97] which was developed for the Automatic Authentication Protocol Analyzer; Casper [Low98] which was used as a front end for the FDR model checker; and Carlsen’s “Standard Notation” [Car94b] which was designed to be translated to per-process CKT5 specifications. All these languages were dedicated to a specific technique or tool, and thus security protocols had to be re-specified for each technique or tool.

Fixing this issue came with the idea of Jonathan Millen who developed a common language, CAPSL, intended to serve as an interface to other tools and languages [Mil97].

Its successor MuCAPSL [MD02] was published later, which is an extension of the first with language features to handle protocols for secure group management. Both CAPSL and MuCAPSL have the remarkable advantages in combing the powers of a variety of techniques for security protocol analysis. Recently, HLPSL [vo05] was proposed, which shares the virtues of CAPSL and MuCAPSL for its applicability for multiple formal techniques.

CAPSL, MuCAPSL and HLPSL [vo05] had been considered as a new direction of the research in the area.

### 4.2 The Unified Verification Framework

The Unified Verification Framework (OSPSL) is an already-existent framework developed firstly by Lian Tian [Tia07].

In the Unified Verification Framework the protocols are seen as systems that consist of collaborating entities (objects). The behavior and flow of the protocol is thus achieved through a
collaboration between these objects, while the state of the protocol is a combined state of all objects inside.

The objects describe the principals and their way of interact. Every object belongs to a class and all classes are derived from the base class, called OBJECT.

*Figure 4.1* describes the class hierarchy in the unified verification framework.

From the OBJECT class there are five derived classes: MESSAGE, STATEMENT, ARRAY, BOOLEAN and PROPERTY. Each of these classes inherits attributes and methods from the OBJECT class and adds new ones. The classes DATA and CONJUNCTION have MESSAGE as the parent class. DATA is used to declare message units with a functional declared boundary, while CONJUNCTION defines concatenation of DATA. The subclass of DATA is ATOM, which contains more subclasses that declare indivisible units, used in constructing messages. The already-built-in subclasses of ATOM are SYMKEY (used to define symmetric keys), PUBKEY (used to introduce public keys), PRIVKEY (used to declare private keys), NONCE, TIMESTAMP, etc.

ARRAY class enables modeling of properties that are shared among a set of objects. This is a type that is currently developed in OSPSL. The same behavior is set for the subclasses of ARRAY. The already-built-in subtypes of ARRAY are PRINPAIR (used to declare a pair of principals), KEYPAIR (used to introduce a pair of keys) and PRINGROUP (developed for group protocols).

BOOLEAN class is used to express the Boolean result of a logical operation and can be either true or false. The PROPERTY class is used to introduce subclasses that are useful in describing properties characteristic to different protocol participants. The built-in subclasses of PROPERTY class are:
• **FRESH**: A formula X is fresh if X is generated specifically for the current session and has not been sent in a message before the current protocol run.

• **RECOGNIZABLE**: A formula X is recognizable to a principal P if P has certain expectations about the contents of X before actually receiving X. P may recognize a particular value, such as its own identifier, a particular structure, or a particular form of redundancy.

• **EXPOSED**: A formula X is exposed if X is a public value and any possible (legal or illegal) principal can learn this value.

• **RANDOM**: A formula X is random if the value of X is generated as “lack of bias or correlations”, and thus is unpredictable.

• **TRUSTY**: A principal P is trusty if P is an honest authority and thereby P can be trusted about messages or statements that are generated or transmitted by P.

The STATEMENT objects correspond to the logic operators of knowledge and belief used to express different statements, communication primitives used to express the flow of messages during a protocol run, authentication requirements for the protocol, possession, etc. The STATEMENT object correspond to the logic operators of knowledge and belief, communication flow (send() and receive()) on MESSAGE objects, authentication requirements for MESSAGE objects (authenticate()) or other statements that express possession (hold()), trust (trustOn()).

The participant entities in the message exchange during the protocol run are objects of type PRINCIPAL. The PRINCIPAL class inherits all the methods and attributes of class ATOM and adds new ones on top of these. There are two attributes characteristic to the PRINCIPAL class: pubKey and privKey added for expressing the public/private keys, along with two methods to use them: return_pubK(), return_privKey(). This class is considered to be one of the most important classes in OSPSL as it inherits the STATEMENT objects of the ATOM class. Objects of type
PRINCIPAL interact with objects of other classes through the STATEMENT attributes of the class. Thus, a security protocol is described as being a collection of PRINCIPAL objects and instantiations of the attributes of these objects.

Classes in OSPSL are presented in Figure 4.1.

![Figure 4.1: Built-in classes in OSPSL](image)

### 4.3 Protocol specifications in OSPSL

In OSPSL, a protocol specification contains two main parts: the first part includes the declaration section, where all the attributes are declared and the other section describes the protocol participants, their initial beliefs and possessions and the whole role they play in the message exchanged during one protocol run.

#### 4.3.1 Declaration Section

The section introduces all the identifiers used to present varied components in a protocol specification language. Each of these identifiers must be of a certain type and must be declared only once. In the declaration section each participant in the protocol must be declared.
Figure 4.2 presents the declaration section for the NSPK protocol in OSPSL.

```
DECLARATION
  PRINCIPAL A, B;
  SYMKEY Kab, Kab1;
  NONCE Na, Nb;
```

Figure 4.2: Andrew’s Secure RPC Declaration Section in OSPSL

The declaration section can be optional followed by a denotation section.

### 4.3.2 Denotation Section

This section allows every previously declared identifier to carry the “value” of an expression of the same type. This enables key denotation and expression convenience. Thus, every time two participants exchange a long message, instead of using the whole message when expressing a protocol step, the user can used its denotation, which will be easier to write and to follow in debugging.

Figure 4.3 states the denotation section for the ZG protocol in OSPSL.
4.3.3 Role Section

In the role section, the role of each participant in the protocol is presented separately. A role definition must start with the keyword ROLE followed by the name of the principal that holds the current role declaration. A role definition contains subsections which describe the initial beliefs of the participant and the steps where the current principal is involved.

4.3.3.1 Assumptions Sub-Section

Includes a list of statements that describes the initial states of the current principal (initial possessions, beliefs and knowledge).
4.3.3.2 Message Sub-Section

Includes all the messages exchanged in the current protocol run, where the current principal participates. Each message is labeled by a sequence number indicating the protocol step when the message was exchanged. **Preconditions**, which state pre-requisites for message emission can be specified together with the sending statement.

**Expectations**, which describe the goals to be achieved on completion of a message reception, can be specified together with a receiving statement. Expectations are tied with each step, meaning that specified goals are to be satisfied immediately after the current step took place. **Figure 4.4** defined the role definition for principal A in the Andrew Secure RPC.

```
ROLE A
{
  ASSUMPTION
    A.hold: A, B;
    A.know: Na.is(FRESH);
  MESSAGE
    1 A.send: A, Kab(Na);
    2 A.receive: B, Kab(Na,Nb);
      EXPECTATION:
        A.authenticate(B):Nb;
    3 A.send: A, Kab(Nb);
    4 A.receive: Kab(Kab1, Nb);
}
```

Figure 4.4: Role definition for principal A in Andrew Secure RPC protocol

4.3.3.3 Axiom Sub-Section

The axiom section introduces statements which are considered to be axioms, meaning they are always considered to be true. This section can be part of the ASSUMPTION section, in which case, all axioms defined are considered to happen at the beginning of the protocol run and are considered to be true all along the protocol run. The axiom section can be places also in the
MESSAGE section, before the EXPECTATION section. In this case, the time when the message exchange occurs is considered the time when the axiom is accepted as true.

The Axiom section is formed by two sub-sections: the HYPOTHESIS sub-section and the CONCLUSION sub-section.

Figure 4.5 presents the axiom section for the ASK protocol.

```
3 U.receive: DHU(KtttPriv{V, KvPub, Tsv, TTP}, Nv);

AXIOM

HYPOTHESIS:
   U.hold: KuPriv;
   U.believe: KvPriv.isValidKeyTo(V);

CONCLUSION:
   U.believe: DHU.isValidKeyTo(V);

AXIOM

HYPOTHESIS:
   U.hold: KuPriv;
   U.believe: KvPriv.isValidKeyTo(V);

CONCLUSION:
   U.believe: F(DHU, Nv).isValidKeyTo(V);

EXPECTATION:
   U.hold: F(DHU, Nv);
   U.know: F(DHU, Nv).is(FRESH);
   U.believe:F(DHU, Nv).isValidKeyTo(V);
```

Figure 4.5: Use of axiom sub-section in the Message section for the ASK protocol
4.4 Conclusions

This chapter presents the unified verification framework with its high level specification language OSPSL.

The unified verification framework is the ideal candidate for logic-based analysis in protocol specifications. The language is easily adaptable to be used also with state-space based techniques. In the next chapters, a translator from OSPSL to a logic based verification engine is proposed and the benefits of formal logics will be shown.
CHAPTER 5. Introduction to LEX and YACC

5.1 LEX - A Lexical Analyzer

Lex is a computer program that generates lexical analyzers. It is a program designed for lexical processing of character input stream which accepts a high-level definition for character string matching and generates a program which recognizes regular expressions. The given regular expressions are specified by the user in the source specifications given as an input.

The Lex source file then associates the regular expressions and the program fragments.

The Lex utility generates a C through an yylex() function which can be used as an interface to YACC. The general format of a Lex File consists of three sections:

The structure of a Lex file is divided into three sections:

5.1.1 The Definition Section
Consist of external C definitions used in the lex actions or subroutines. The definitions defined could be

- preprocessor directives (#include, #define macros). These are simply copied to the lex.yy.c file
- Lex definitions which are essentially the lex substitution strings, lex start states and lex table size declarations

C/C++ code could be contained in this part, in which case it will be copied verbatim in the generated source file
5.1.2 The Rules Section

Represents the basic part which defines the regular expressions and the corresponding actions to be taken when these expressions are matched. This section associates regular expressions patterns to C/C++ statements. When the lexer discovers that the input text matches a pattern, the associated C/C++ code is executed.

5.1.3 The Subroutine Section

Contains statements and methods copied verbatim in the generated source file.

Thus the form of a Lex file is as presented in Figure 5.1.

```
{definitions}
%%
{rules}
%%
{user routines}
```

Figure 5.1: Structure of a Lex file

5.1.4 Overview on Lex

Lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. Lex programs recognize only regular expressions; Yacc writes parsers that accept a large class of context free grammars, but require a lower level analyzer to recognize input tokens. When used as a preprocessor for a later parser generator, Lex is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. The name yylex is what Yacc expects its lexical analyzer to be named, so that the use of this name by Lex simplifies interfacing.
5.2 **YACC – Yet Another Compiler Compiler**

The computer program **Yacc** is a parser generator developed by AT&T for the Unix operating system in 1970. It generates a parser based on an analytical grammar expressed via BNF.

Yacc provides a general tool for describing the input to a computer program. The Yacc file specifies the structures of its input, together with code to be invoked as each such structure is recognized. Then, it generates a function (parser) which calls the routine of the lexical analyzer to pick up the tokens from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions.

A Yacc file has the structure very similar to a Lex file composed by the definition section, the rule section and the code section.

### 5.2.1 The Definition Section

Contains C/C++ code used for the definition of global variables and prototypes of methods defined in the code segment. It may contain associativity rules (which describe priority of operations) and definitions of tokens which are written to a header file when Yacc compiles the file to be parsed.

### 5.2.2 The Rule Section

The rules section contains the grammar of the language to parse. It is a set of actions described in C/C++ code, associated with each matching right-hand side.
5.2.3 The Code Section

It is necessary that this section has the method main() which calls the yyparse() method. The method calls a routine yylex() every time it wants to obtain a token from the input. Returns 0 if the input is valid according to the grammar rules.

Thus, a specification file looks like the one described in Figure 5.2:

```
declarations

rules

Programs
```

Figure 5.2: Structure of a YACC file

5.2.4 YACC Grammar Rules

Yacc rules define a legal sequence of tokens in the language to be built. A rule has the syntax defined in Figure 5.3.

```
rules :TOKEN1 TOKEN2
       ;
```

Figure 5.3: Simple rule in YACC

The above rule defines a non-terminal symbol named rules, defined in terms of TOKEN1 and TOKEN2.

YACC handles also recursive rules, defined as in Figure 5.4.

```
rules :rule
       |rules rule
       ;
```

Figure 5.4: Recursive rule in YACC

Describing rules in terms of itself creates a new rules that is formed by one or more rules.
5.2.5 Overview on YACC

The parser produced consists of a finite state machine with a stack. It is also capable of reading and remembering the next input token (called the lookahead token). The current state is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called shift, reduce, accept, and error. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs to search a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls yylex to obtain the next token.

2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

5.3 Summary

This section introduces a short overview over Lex and Yacc, their file structure and their cooperation facilities when used together in writing translators or compilers.

During the first phase the compiler reads the input and converts strings in the source to tokens. Regular expressions are used to create patterns to lex so it can generate code that will allow it to scan and match strings in the input. Each pattern specified in the input to lex has an associated action. Typically an action returns a token that represents the matched string for subsequent use by the parser. Lex reads the patterns and tokens and generates the C code for a lexical analyzer. When the lexical analyzer finds identifiers in the input stream it enters them in a symbol table. The grammar is a file describing rules that apply on tokens or patterns.
Yacc reads the grammar and generates C code for a syntactic analyzer or parser. The syntax analyzer uses the grammar rules defined in the .yacc file, fact which allows it to analyze tokens from the lexical analyzer and create a syntax tree, which has a hierarchical structure of tokens. Yacc’s code generation does a depth-first walk of the syntax tree to generate.
CHAPTER 6. Designing a translator from the Unified Verification Framework to CDVT

The new Unified Verification Framework is meant to integrate both state-space based verification techniques and logic-based techniques. To achieve this, a translator from the unified verification framework (OSPSL) to the logic-based verification tool (CDVT) is proposed.

6.1 Architecture of the project

Security protocol specifications defined in the unified verification framework OSPSL are translated in an equivalent specification described in the low-level logic based verification language CDVT. The translator is integrated in the Unified Verification Framework along with other translators to the low-level state-space languages and it is transparent for the user: the programmer writes only the specifications in OSPSL while the translator handles all the work.

All OSPSL files are integrated in the CDVT project which is structured in the following main folders: Source Files, Header Files, LEX Source Files and YACC Source Files.

The Source Files folder contains the all .cpp files in the project, belonging to both translator and CDVT. The generated parser and lexical analyzer .cpp files are contained here. The LEX Source file contains both lexical analyzers for CDVT and OSPSL and the YACC Source file contains the lexical analyzers for both.

The architecture of the project is presented in Figure 6.1.
6.1.1 Use of Lex and Yacc in OSPSL’s translator

The method writeToFile of the class CMRDparser included in the Lex Source File takes as an input the text file which contains the specs for a security protocol.

The file is parsed and lexically analyzed building a symbol table which contains all tokens taken from the specifications. The symbol table contains specific information about the principals involved in the communication, roles, messages, secret requirements, etc. When parsing, the tokens are extracted using the lexical analyzer and the principals, logical connectors and statements for OSPSL are created. Thus, the translator takes the spec files in OSPSL and creates the spec files in CDVT, sending them directly to the tool.

When creating each object, the get methods in OSPSL call the get methods from the CDVT project and thus, in the same time, the logical connectors, statements and identifiers are created in CDVT.

The translation process from OSPSL to CDVT is described in Figure 6.2.
6.1.2 Equivalence of Types between OSPSL and CDVT

Table 6.1 describes the equivalence of types between OSPSL and CDVT. The main idea of OSPSL was to introduce a high level language for protocol verification capable of verifying security protocols using logic based techniques. As such, the equivalence of types between the two is almost straightforward.
<table>
<thead>
<tr>
<th>OSPSL</th>
<th>CDVT</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRINCIPAL</td>
<td>PRINCIPAL</td>
<td>Principals involved in the protocol</td>
</tr>
<tr>
<td>SYMKEY</td>
<td>SYMKEY</td>
<td>Symmetric keys</td>
</tr>
<tr>
<td>PUBKEY</td>
<td>PUBKEY</td>
<td>Public keys</td>
</tr>
<tr>
<td>PRIVKEY</td>
<td>PRIVKEY</td>
<td>Private keys</td>
</tr>
<tr>
<td>NONCE</td>
<td>NONCE</td>
<td>Nonces</td>
</tr>
<tr>
<td>HASH</td>
<td>HASH</td>
<td>Hash functions</td>
</tr>
<tr>
<td>FUNCTION</td>
<td>F</td>
<td>Functions</td>
</tr>
<tr>
<td>PRINPAIR</td>
<td>NOT AVAILABLE</td>
<td>Pair of principals sharing the same data</td>
</tr>
<tr>
<td>KEYPAIR</td>
<td>NOT AVAILABLE</td>
<td>Pair of keys belonging to a principal</td>
</tr>
<tr>
<td>DATA</td>
<td>DATA</td>
<td>Data to be exchanged through the protocol</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>TIMESTAMP</td>
<td>Timestamps</td>
</tr>
</tbody>
</table>

Table 6.1: Equivalence of types between OSPSL and CDVT

Variables of these types are kept in the symbol table according to their type and treated as such. There are types in OSPSL that have no equivalent in CDVT. As an example, when specifying public key protocols, OSPSL has variables of type PUBKEY and PRIVKEY. The KEYPAIR type links the public and private key of a principal, making them related. In CDVT this is achieved through the names of the PUBLIC KEY and PRIVATE KEY. The variable of type KEYPAIR will contain two references to the PUBLIC and PRIVATE KEY it links. When these two keys are reunited under the KEYPAIR type, the translator calls the method `setInverseKeys` from CDVT tool to link the two keys together.

### 6.1.3 Equivalence of logical operators

Table 6.2 describes the equivalence of communication and logical operators in OSPSL and CDVT. The most important logical operators from CDVT are reunited in OSPSL.
In CDVT the authentication of one principal towards the other is a combination of logical operators. As a corresponding operator for this combination, in OSPSL there is the `authenticate` method of the class `PRINCIPAL`. The equivalence between the two is described in Table 6.3.

### Table 6.3: Equivalence of authenticate operators in OSPSL and CDVT

<table>
<thead>
<tr>
<th>OSPSL</th>
<th>CDVT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Principal1.authenticate(Principal2):data;</td>
<td>Principal1 know at[time] (Principal2 send at[time] data)</td>
</tr>
</tbody>
</table>

#### 6.1.4 Statement translation

The logical operator of possession is defined in OSPSL by the method `hold`. The `hold` method is equivalent to the CDVT `possess` operator. As such, the following expression in OSPSL `principal.hold: data;` will be translated in CDVT as `principal possess data.`
The logical belief operators defined in OSPSL by the method believe. It is the equivalent of the know operator in CDVT. The table contains a few of the most used expression in OSPSL and their equivalent in CDVT.

6.2 Roles equivalent

6.2.1 Assumption translation

In OSPSL, the initial assumptions appear in each role of each principal, under the ASSUMPTION section. In this case these assumptions are parsed, lexically analyzed and translated to CDVT. They are all kept in the assumption vector from where they are read when building the output CDVT file. The time considered for these assumptions is considered to be $t_0$.

The equivalent for the CDVT IMPLY statement is the OSPSL AXIOM section that can appear either in the MESSAGE section or in the ASSUMPTION. In OSPSL the AXIOM section is formed by the HYPOTHESIS section and the CONCLUSION section. These sections are parsed, the statements are formed and they are all reunited in one single CDVT IMPLY statement. Note that the OSPSL AXIOM statement is meant to form axioms that are considered to be true from the start of the protocol.

6.2.2 Translation of protocol steps

The protocol steps appear in OSPSL under the MESSAGE section. A protocol step in OSPSL has the following form:

\[
\text{time Principal. send: data; or time Principal. receive: data;}
\]

All these are parsed, translated to CDVT statements and kept in the Goals vector.
6.2.3 Translation of security requirements

In OSPSL, each step can be followed by an EXPECTATION section. All statements situated in the EXPECTATION section are translated straight to CDVT security requirements (goals that the protocol is expected to achieve). The EXPECTATION section is not mandatory. In CDVT the security requirements are enumerated in the goal section.

Authentication requirements are treated the same as authentication assumptions, with the difference that they are helpful in the Goals vector.

6.3 The symbolic table of the translator

There is one main symbol table built for the OSPSL translator. This is a class which reunites get methods for all existent built-in types in OSPSL. It contains also the get and set methods for the logical operators used for the translation.

Each object of this class has the following members, described in Table 6.4.
<table>
<thead>
<tr>
<th>Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmrdGetIdentifier()</td>
<td>The get methods that returns the identifier of an object</td>
</tr>
<tr>
<td>cmrdGetType()</td>
<td>Get method that returns the type of the object</td>
</tr>
<tr>
<td>cmrdGetObjectCount()</td>
<td>Returns the number of objects that the current object points to</td>
</tr>
<tr>
<td>cmrdGetObject(int index=0)</td>
<td>Returns the current object</td>
</tr>
<tr>
<td>cmrdGetContext()</td>
<td>Returns the context of an expression</td>
</tr>
<tr>
<td>string cmrdName</td>
<td>Name of identifier</td>
</tr>
<tr>
<td>string cmrdCDVTName</td>
<td>associated CDVT name</td>
</tr>
<tr>
<td>vector&lt;void *&gt; cmrdObject</td>
<td>object representing identifier</td>
</tr>
</tbody>
</table>

Table 6.4: Content of the symbolic table of the translator

For each object of this class, there are methods that contain information about the object. The most important information about each object would be its identifier (its name), type, the number of objects that the current object points to, the corresponding CDVT name, etc. The different types of the objects are held in a separate file.

In public key cryptography, there are two different key: public and private. In both OSPSL and CDVT these keys can be defined separately using the types PUBKEY and PRIVKEY. In CDVT these keys are bounded by their names: both names (the public and private key) have to start with the capital “K” followed by the principal’s name and the token “Pub” or “Priv”, according to its type. In OSPSL the binding of the keys is done through the KEYPAIR type. In the KEYPAIR type the first element given as an argument is the name of the public key, while the second element given as an argument is the private key. The translation and key binding is not straightforward. In the translation process there is a method setInverseKeys which binds the two keys in CDVT.
Each symbol table object of the class may point to one or more other objects. For example a variable of type KEYPAIR will point to both public and private keys of a principal. Each time a user enters the KEYPAIR’s name, in the translation process, the objects that the KEYPAIR points to will be used to connect the keys in CDVT.

In the DENOTE section of the protocol, each denote variable points to the expression it denotes. The same as for the KEYPAIR, when using the DENOTE variable in the specification, the value that this points to will be the one used in the translation process.

### 6.3.1 Structures and vectors of the translator

The translator uses three vectors to hold the assumptions, steps and goals for each protocol described.

For each step of the protocol, the structure holds a pointer to the step name, an integer variable stating the time when the step occurs and the operator type the step belongs to (send or receive).

Similarly, for the goals, the structure contains elements that retain the time when the goal should be achieved and the goal name.

### 6.4 Implementation Issues

The OSPSL to CDVT translator has different parts. First, the input file that retains the protocol specifications in OSPSL is parsed and the tokens are analyzed. Then, the symbol table is created and the objects are added to the symbol table class. After these two phases are finished, the specifications for the generated CDVT file are created. The assumptions, steps and goals structures are filled and output in a new file.
The translator is implemented in C++ and it uses the lexical parser *Lex* and the syntactic analyzer *Yacc*. The *Syntax Analyzer* parses the input file containing the specifications in OSPSL using grammar rules similar to BNF. (Appendix A). The *Lexical Parser* provides the tokens sed by the *Syntax Analyzer*. All tokens are defined in the parser and taken, one by one, from the protocol specifications in the unified verification framework. If one token cannot be found in the parser, a syntax error occurs. In the same time with the syntactical analyzes, the symbol table is created and its elements are added, as soon as a sequence of tokens matches one grammar rule. Once the text is parsed correctly to the end, it means that all grammar rules have been found and the output file is generated, containing the specifications in CDVT. During the syntactic analysis, if a grammar rule is found to be matching a sequence of tokens, an element is added to the symbol table (when defining an assumption, it is immediately added to the symbol table and then to the assumption vector).

All elements and types are defined as mentioned in the previous subchapter and they are entirely defined in C++. Once the text is all parsed, the symbol table contains all the variables defined in the protocol and the assumption, steps and goals vectors are filled.

### 6.5 Summary

This chapter presents the process of translation from the unified verification framework to the logic based verification tool, CDVT. The translation is done through three phases, where the first two (syntax analyzing of the input file and creation of the symbol table) can be done simultaneously. After these two phases are completed, the symbol table and the protocol parts associated vectors are filled and able to generate output code.

The CDVT is a logic based verification language, therefore it has the main operators used for modal logics (operators of possession, knowledge and belief). The translator handles all the work for the user, all translation process being hidden from the programmer.
CHAPTER 7. Empirical validation of the translation from OSPSL to CDVT

The OSPSL verification framework aims to integrate both modal logics and state-space based tools. In Chapter 6, a translator from the unified verification framework to CDVT (a logic based verification engine) was presented, based on the modal logic characteristics defined in Chapter 4, considering the Coffey-Saidha logic [CS97].

In this chapter an empirical study will be made on a set of security protocols. The collection contains a set of protocols that were considered secure, but also protocols that were known to be easily attacked. The goal of this study is to validate the translation process from OSPSL to CDVT presented in Chapter 6. All protocols will be analyzed in OSPSL, verified through their generated specs in CDVT and then compared with the verifications made on the manually generated specs in the same tool.

7.1 Kao-Chow Protocol

The Kao-Chow protocol [KC95] is a mutual authentication and key distribution protocol that uses a third trusted party for key generation and distribution. Its aim is to ensure strong authentication. The two parties that want to communicate securely trust S to issue a fresh secret session key Kab. The protocol aims to authenticate the principals A and B to each other using their nonces Na and Nb.

At the first step, A sends to the server its identity, a fresh nonce Na and B’s identity. In the next step server S will issue to B two tickets that contain the session key. Both tickets consist of the identities of A and B, the nonce Na and the session key Kab. The first ticket is encrypted with Kas and the second is encrypted with the Kbs. As B can decrypt the second ticket it now is in
possession of the session key. In step three B will forward the first ticket from the second message along with the nonce Na encrypted under Kab and its own nonce Nb to A. On receipt of this message A retrieves the session key Kab and verifies the correctness of the encrypted nonce. In the final step A will encrypt the nonce Nb with Kab and sends it to B.

The Kao-Chow protocol is presented in Figure 7.1.

![Figure 7.1: Kao Chow Protocol](image)

The authors discuss limitations of Kao-Chow initial version and proposed another two versions (Kao-Chow v2 & Kao-Chow v3) for this protocol. Kao-Chow v2 aims to overcome the limitations in the first version, by adding an extra fresh key Kt generated by the server that is discarded after each protocol run. Kt is included in message two when the distribution of Kab is initiated.

1. A \rightarrow S: A, B, Na
2. S \rightarrow B: \{A, B, Na, Kab, Kt\}Kas, \{A, B, Na, Kab, Kt\}Kbs
3. B \rightarrow A: \{A, B, Na, Kab, Kt\}Kas, \{Na, Kab\}Kt, Nb
4. A \rightarrow B: \{Nb, Kab\}Kt
Chapter 7. Empirical validation of the translator

The third version of the protocol is an extension of v2 to encompass tickets. In step three B generates a new ticket containing Kab and a timestamp Ta:

1. A → S: A, B, Na
2. S → B: {A, B, Na, Kab, Kt}Kas, {A, B, Na, Kab, Kt}Kbs
3. B → A: {A, B, Na, Kab, Kt}Kas, {Na, Kab}Kt, Nb, {A, B, Ta, Kab}Kbs
4. A → B: {Nb, Kab}Kt, {A, B, Ta, Kab}Kbs

7.1.1 Results of the verification for the Kao-Chow protocol in CDVT

After manually specifying the specifications for the Kao-Chow protocol in CDVT, the results are shown in Figure 7.2.

<table>
<thead>
<tr>
<th>Verification Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Assumptions</td>
</tr>
<tr>
<td>2. Protocol Steps</td>
</tr>
<tr>
<td>3. Protocol Verification</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>✗ Protocol is Verified is False</td>
</tr>
<tr>
<td>✗ (3): B know at[2] NOT(Zero send at[0] {(((A,B),Na),Kab)}Kas) is assumed False</td>
</tr>
<tr>
<td>✗ (4): A possess at[3] Kab is True</td>
</tr>
<tr>
<td>✗ (6): A know at[3] NOT(Zero send at[0] {(((A,B),Na),Kab)}Kas) is True</td>
</tr>
<tr>
<td>✗ (8): A know at[3] NOT(Zero send at[0] {Na}Kab) is True</td>
</tr>
<tr>
<td>✗ (10): B know at[4] NOT(Zero send at[0] {Nb}Kab) is True</td>
</tr>
</tbody>
</table>

Figure 7.2: Verification results for the manually written specifications for the Kao-Chow protocol

Investigation of the failed protocol goals reveals that the protocol suffers from a freshness weakness: B’s inability to establish freshness of the message component containing the session
key in protocol step 2 prevents B to accept the session key. Thus neither key establishment for B nor authentication of A to B is achieved by the protocol.

The discovered weakness of the protocol can be exploited by an intruder.

Version 2 of the Kao-Chow protocol uses a fresh key Kt to avoid freshness weaknesses. Investigations of the properties of the Kt reveal that it does not add to the security of the protocol. If Kt is a confirmed shared fresh secret between A and B, then there is no need to establish another session key Kab, as A and B could simply use Kt for communication.

On the other hand, if the key Kt is not a confirmed shared fresh secret but would be stored after each protocol run and the Kt produced in futures runs would be compared to the old Kts the replay would be impossible. However, the authors introduced this key with the condition that it is discarded after each session. Therefore it has exactly the same properties as Kab, which makes it redundant.

7.1.2 Amended version of the Kao-Chow protocol

[RLC98] propose that rather than sending the request for communication directly to the server, the initiating principal A will send the request along with the nonce Na to the principal B. B then will forward the communication request to the server along with Na and its own nonce Nb. Thus, the tickets that contain the session key can be identified by both principals as fresh, i.e. as belonging to the current protocol run.

Consequently, any attempt by an intruder to replay message 3 will fail, as B can identify the replay through the wrong value of Nb. Another change has been made for optimization in step three. The ticket issued by the server for A includes only B’s identity and A’s nonce rather than having both principals’ identities and nonces. Similarly, the ticket sent to B contains only A’s identity and B’s nonce.

Results for the proposed amended version are found in Figure 7.3.
1. Assumptions

2. Protocol Steps

3. Protocol Verification

Figure 7.3: Results for the manual written specifications of the amended Kao Chow Protocol

Formal verification of the amended protocol provides confidence in the correctness and effectiveness of the proposed modifications.

7.1.3 Specifications for the Kao-Chow protocol in OSPSL

Figure 7.4 presents the role of principal A in the Kao-Chow protocol.
In the same way, the role descriptions for the principals B and S are described. The verification results for the generated specs of the Kao-Chow protocol in OSPSL are outlined in Figure 7.5.
1. Assumptions

2. Protocol Steps

3. Protocol Verification

- Protocol is Verified is False
  - (3): A know at[4] NOT(Zero possess at[0] {{B, (Na, Kab)}|Kas) is True
  - (4): A know at[4] NOT(Zero send at[0] {{B, (Na, Kab)}|Kas) is True
  - (6): A know at[4] NOT(Zero possess at[0] {{Na, Nb}}Kab) is True
  - (7): A know at[4] NOT(Zero send at[0] {{Na, Nb}}Kab) is True
  - (8): B possess at[3] Kab is True
  - (10): B know at[3] NOT(Zero possess at[0] {{A, (Nb, Kab)}}Kbs) is True
  - (11): B know at[3] NOT(Zero send at[0] {{A, (Nb, Kab)}}Kbs) is True
  - (13): B know at[5] NOT(Zero possess at[0] {Nb}Kab) is True
  - (14): B know at[5] NOT(Zero send at[0] {Nb}Kab) is True

Figure 7.5: Results of the generated specs for the Kao-Chow protocol specifications in OSPSL

7.1.4 Amended Kao-Chow protocol in OSPSL

Verification of the amended version for the Kao-Chow generated protocol specifications using OSPSL lead to the same results as verifying the amended version in CDVT.

The verification results for the CDVT manually specifications are outlined in Figure 7.6.
7.1.5 Conclusions after the verification of the Kao-Chow protocol

As seen above, the attacks results obtained are the same for the manual written specifications, as well as for the generated specifications in OSPSL. Investigations show the presence of a freshness flaw in both cases, where the protocol is specified in CDVT and then in OSPSL. On the other hand, when analyzing both amended version written first in CDVT and then OSPSL, verifications state that no attack has been found. As a conclusion, in both cases, using CDVT or OSPSL, the results after verifying the original Kao-Chow protocol, as well as the amended version proposed by Dojen, Lasc and Coffey, are the same. Thus, formal verifications for the Kao-Chow protocol, original and amended version, provide confidence in the correctness and effectiveness of the translator process, as they are all leading to the same verification results.
7.2 **BCY protocol**

The BCY protocol [BCY93] demonstrated the feasibility of public-key cryptography in mobiles. In 1994 Carlsen [Car94] discovered a weakness and proposed an amended version. This version was also discovered to be faulty by Mu and Varadharajan in 1996 [MV96] and another fix was proposed. Horn, Martin and Mitchell [HMM02] identified a weakness in Mu and Varadharajan’s version, but did not provide a solution. Coffey, Dojen and Flanagan [CDF03] published formal verification for the BCY protocol, proposing an amended version was proposed and also formally verified.

The BCY protocol is presented in *Figure 7.7*.

![Figure 7.7: BCY protocol](image)

In this thesis, the BCY initial form and the Coffey, Dojen and Flanagan proposed BCY (called CDFBCY) were considered to be analyzed using CDVT and OSPSL.

### 7.2.1 Results of the verification of the BCY protocol in CDVT

Results after the CDVT verification are presented in *Figure 7.8* for the BCY protocol.
The results shown in Figure 7.8 identify some failures of the protocol. First two goals fail as U cannot establish the validity of V’s public keys. Goal G8 fail as U cannot establish the key agreement with V, while G14 fail as U cannot recognize V through the message sent at step 3 of the protocol run. In the same way, V cannot identify U through the message sent at step 4. All these failures have been proven to be the cause of two weaknesses existent in the protocol: principals cannot validate the certificates and V cannot establish that the session key received is fresh. In such case, the original BCY protocol provides neither authentication nor key agreement.

7.2.2 Amended version of the BCY protocol

The redesign of the BCY protocol was proposed by Coffey, Dojen and Flanagan [CDF03] focusing on validating freshness of the certificates exchanged in the key agreement and freshness of the session key used during the protocol run.
The amended version proposed to introduce $T_s$, a timestamp included in both $U$ and $V$'s certificates. A nonce $N_v$ is added also in the construction of the session key and both parties contribute now towards the session key. The resulted protocol contains the following steps:

\[\text{BCY'1: } V \rightarrow U: \{V, K_v^d+, K_v^m+, T_{sv}\}K_s^{-} \]
\[U \text{ computes } Y = \{R_u\}K_v^m+, K_{K} = (K_v^d+K_v^d)K_u^{-}, SK = \{R_u,R_v\}K_K\]
\[\text{BCY'2: } U \rightarrow V: Y, \{(U,K_u^+, T_{su}\}K_s^{-})R_u\]
\[V \text{ computes } R_u = \{Y\}K_v^m-, K_{K} = (K_u^+K_v^d)K_d^{-}, SK = \{R_u,R_v\}K_K\]

\[\text{BCY'3: } V \rightarrow U: \{\text{data}_V\}SK\]
\[\text{BCY'4: } U \rightarrow V: \{\text{data}_U\}SK\]

The goals for the CDFBCY protocol are identical with the ones from the original version.

Figure 7.9 presents the results of the amended CDFBCY protocol in CDVT.

<table>
<thead>
<tr>
<th>1. Assumptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. Protocol Steps</td>
</tr>
<tr>
<td>3. Protocol Verification</td>
</tr>
</tbody>
</table>

- Protocol is Verified is True
- (3) : $U$ possess at[1] $(K_v^1Pub,K_v^2Pub)$ is True
- (4) : $U$ possess at[1] $(K_v^1Pub)K_u$Priv is True
- (5) : $U$ possess at[1] $\{N_u,N_v\}(K_v^1Pub)K_u$Priv is True
- (6) : $U$ know at[1] NOT(Zero possess at[0] $\{N_u,N_v\}(K_v^1Pub)K_u$Priv) is True
- (8) : $V$ possess at[2] $K_u$Pub is True
- (9) : $V$ possess at[2] $K_v^1$Pub is True
- (10) : $V$ possess at[2] $\{N_u,N_v\}(K_u$Pub)$K_v^1$Priv is True
- (11) : $V$ know at[2] NOT(Zero possess at[0] $\{N_u,N_v\}(K_u$Pub)$K_v^1$Priv) is True
- (12) : $U$ know at[3] NOT(Zero send at[0] $\{\text{data}_V\}(\{N_u,N_v\}(K_v^1Pub)K_u$Priv) is True
- (14) : $V$ know at[4] NOT(Zero send at[0] $\{\text{data}_U\}(\{N_u,N_v\}(K_u$Pub)$K_v^1$Priv) is True

Figure 7.9: Verification results for manually written specifications of the CDFBCY protocol

The results shown for the amended protocol using CDVT logic based verification tool state that all goals are satisfied, proving correctness in the design of the proposed security protocol.
7.2.3 Results of the verification of the BCY protocol in OSPSL

Similarly, results after OSPSL translation are outlined in Figure 7.10.

1. Assumptions
2. Protocol Steps
3. Protocol Verification

Figure 7.10: OSPSL verification results for the BCY protocol

Verifications of the original BCY protocol in OSPSL have been proven to lead to the same conclusions as the ones that appeared after verifying the protocol with the logic based tool. Goals that are not satisfied prove the same weakness existent in the BCY protocol.

The results shown in Figure 7.10 identify the same failures of the protocol: U cannot establish the validity of V’s public keys. Goal G10 fail as U cannot establish the key agreement with V, while the other goals are failing because U cannot recognize V through the message sent at step 3 and, similarly, V cannot identify U through the message sent at step 4.

The results shown identify one more time that the original BCY protocol provides neither authentication nor key agreement.
7.2.4 CDFBCY protocol in OSPSL

Analyzing the amended CDFBCY protocol using OSPSL leads to the same results as verifying the CDFBCY protocol using CDVT.

All goals are proven to be true as shown in Figure 7.11, meaning that both freshness of the session key and key agreement were established.

![Figure 7.11: OSPSL verification results for the CDFBCY protocol](image)

### 1. Assumptions

### 2. Protocol Steps

### 3. Protocol Verification

- Protocol is Verified is True

- (3) : U possess at[1] (Kv1Pub, Kv2Pub) is True
- (4) : U possess at[1] (Kv1Pub) KuPriv is True
- (5) : U possess at[1] (Nu, Nv) (Kv1Pub) KuPriv is True
- (6) : U know at[1] NOT (Zero possess at[0] (Nu, Nv) (Kv1Pub) KuPriv) is True
- (7) : U know at[1] NOT (Zero send at[0] (Nu, Nv) (Kv1Pub) KuPriv) is True
- (8) : U know at[3] NOT (Zero possess at[0] (data V) (Nu, Nv) (Kv1Pub) KuPriv) is True
- (9) : U know at[3] NOT (Zero send at[0] (data V) (Nu, Nv) (Kv1Pub) KuPriv) is True
- (12) : V possess at[2] (KuPub) Kv1Priv is True
- (13) : V possess at[2] (KuPub) Kv1Priv is True
- (14) : V possess at[2] (Nu, Nv) (KuPub) Kv1Priv is True
- (15) : V know at[2] NOT (Zero possess at[0] (Nu, Nv) (KuPub) Kv1Priv) is True
- (16) : V know at[2] NOT (Zero send at[0] (Nu, Nv) (KuPub) Kv1Priv) is True
- (17) : V know at[4] NOT (Zero possess at[0] (data U) (Nu, Nv) (KuPub) Kv1Priv) is True
- (18) : V know at[4] NOT (Zero send at[0] (data U) (Nu, Nv) (KuPub) Kv1Priv) is True

### 7.2.5 Conclusions after the verification of the BCY protocol

Just as for the Kao-Chow protocol, as well as for the Zhou-Gollmann protocol, the results obtained after verifying the BCY protocol using OSPSL and CDVT are the same. At the beginning, when verifying the original protocol, both tools discover the same weakness:
freshness and key agreement. Moreover, when analyzing the amended BCY in both CDVT and OSPSL, no attack is found. As a conclusion, in both cases, the results are the same. Thus, formal verifications for the BCY provide more confidence in the correctness and effectiveness of the translator process.

### 7.3 ZG non-repudiation protocol

The Zhou Gollmann protocol was published by Zhou and Gollmann in 1996 [ZG96]. It is a non-repudiation protocol which used an online trusted third party.

The Zhou-Gollmann protocol is presented in Figure 7.12.

![Zhou-Gollmann protocol diagram](image)

**Figure 7.12: Zhou-Gollmann protocol**

Principal A initiates the protocol by sending to B the flag `dataNRO`, B’s identity, the nonce `Na` and the message `dataA` encrypted with session key `Kab`, as well as all these signed with A’s private key.
Principal B can now decrypt the message sent by A using A’s public key, but cannot decrypt the message \( \text{dataA} \) at this time, as he does not possess the session key \( \text{Kab} \). B replies by sending to A the flag \( \text{dataNRR} \), A’s identity, the nonce \( \text{Na} \) and the encrypted message \( \text{\{data\}Kab} \) as well as all these signed with B’s private key.

On reception of this message, A confirms that the correct nonce \( \text{Na} \) is used and then sends to the TTP a message containing the flag \( \text{dataSUB} \), B’s identity, the nonce \( \text{Na} \) and the session key \( \text{Kab} \) as well as all these signed by its private key. The TTP decrypts A’s message using A’s public key and now possesses the session key \( \text{Kab} \). Then the TTP compares the nonce/key pair sent by A against a list of previously used pairs. If a unique pair has been used, the TTP accepts the message as a genuine request by A. Otherwise the message is considered a replay and will be discarded. In the next two steps the TTP sends a confirmation message to both principals. This confirmation message is intended to complete nonrepudiation of origin and receipt. These messages contain the confirmation flag \( \text{dataCON} \), the identities of A and B, the nonce \( \text{Na} \) and the session key \( \text{Kab} \) as well as all these signed by the TTP. In case of a dispute, if principal A claims to have sent the message to B, the judge will ask A to provide the message that has been sent and the nonrepudiation of receipt for this message. The evidence that A has to give is composed of the nonrepudiation of receipt provided by B and the confirmation message provided by the TTP. If all the evidence is proven to be valid, the judge closes the case and declares A’s assertion as correct. On the other hand, if A falsely claims to have sent the message, A will not be in possession of complete and valid evidence and A’s claim will be refuted.

Similar for principal B: If B claims to have received a message from A the judge will request B to provide the message and the non-repudiation of origin. The evidence that B has to present is composed of the non-repudiation of origin for the message being sent (provided by A) and the confirmation message provided by the TTP. If B has all this evidence, the judge will assert B’s claim to be correct. Again, if B falsely claims to have received a message from A, B will no be able to provide the complete and valid evidence.
7.3.1 Specifications of the ZG non-repudiation protocol in CDVT

*Figure 7.13* show the manually written assumption declaration for the Zhou-Gollmann non-repudiation protocol in CDVT. Assumptions A1 and A2 express the fact that before the start of the protocol, A possesses both TTP’s and B’s public keys. A3 and A4 express the fact that A is aware both B and TTP possess their own private keys before the protocol starts. This binds the public keys to the identity of the corresponding principal. A5 specifies that A possesses the nonce Na and assumption A6 states the fact that A knows that no other principal possess this nonce.
A1: A possess at[0] KttpPub;
A2: A possess at[0] KbPub;
A3: A know at[0] B possess at[0] KbPriv;
A4: A know at[0] TTP possess at[0] KttpPriv;
A5: A possess at[0] Na;
A6: A know at[0] (not(Zero possess at[0] Na));
A7: B possess at[0] KttpPub;
A8: B possess at[0] KaPub;
A9: B know at[0] TTP possess at[0] KttpPriv;
A10: B know at[0] A possess at[0] KaPriv;
A11: TTP possess at[0] KaPub;
A12: TTP possess at[0] KbPub;
A13: TTP know at[0] A possess at[0] KaPriv;
A14: TTP know at[0] B possess at[0] KbPriv;
A15: TTP know at[0] (not(Zero possess at[0] (dataSUB,B,Na,Kab)));

Figure 7.13: Assumption section for the CDVT manual specifications of the ZG protocol

[MDCG09]

A7 and A8 express the fact that B is in possession of both A’s and TTP’s public key and assumptions A9 and A10 specify that B knows before the protocol starts that both A and TTP are in possession of their own private keys. Assumptions A11 and A12 state that TTP is in possession of both A and B’s public keys. A13 and A14 declare that TTP knows both A and B are in possession of their private keys before the protocol run. A15 expresses the fact that the
TTP is able to detect previously used nonce/session key pairs (which will not be accepted by the TTP).

Similarly, Figure 7.14 presents the goal section for the manually written CDVT specs in Zhou-Gollmann protocol.

Goals G1 states that B in possession of the evidence of origin (Note that this evidence is incomplete without the confirmation message). Similar, goals G2, G3 determine that A is in possession of the evidence of receipt from B (evidence is incomplete without confirmation
message) and that B is indeed the source of the message. Goals G4 to G6 are related with the fact that TTP has obtained confirmation of submission of key \( Kab \) by A and that the nonce/key pair hasn’t been used in a previous protocol run. G7 to G9 establish that B receives the message that TTP sent in step 3 and now B is in possession of the session key \( Kab \) and the clear text message dataA. Both G10 and G11 are related to TTP’s authentication towards B: they express that B knows that the message received at step 4 of the protocol run was sent by TTP and that this message is fresh. G12 to G14 state that A has now the complete evidence of receipt for the message sent to B. G12 expresses A’s possession of the confirmation of the session key sent by TTP. G13, G14 describe the fact that A knows the message received at step 5 actually came from TTP and is fresh.

### 7.3.2 Results of the verification for the ZG protocol in CDVT

Results of the verification show that the protocol is considered to be false. The problem is caused by the failure of principal B to ensure validity of the confirmation evidence received in the protocol step S4. This confirmation evidence is required to complete the evidence of origin received in step S2. Goals are failing due to B’s inability to establish freshness of the message received from TTP (protocol step S4), which includes the confirmation evidences created by the TTP. Further, as this message doesn’t contain anything that B can recognize as being fresh, B is unable to bind the received message to the current protocol run. Therefore, B cannot distinguish a new valid message from an old and potentially compromised message replayed by an attacker. As a consequence, the protocol is susceptible to a replay attack that allows an attacker to trick principal B into accepting messages from an old protocol run. *Figure 7.15* shows the results of the verification through CDVT tool of the manual specification for the ZG protocol.
An amended version for the ZG protocol was proposed by Muntean, Dojen and Coffey in 2009 [MDC09]. The steps of the amended protocol are analogous to the original version, but included the nonce $Nb$ in steps 2 to 5. Inclusion of both nonces $Na$ and $Nb$ into the confirmation evidence $A$ and $B$, are able to establish freshness of this evidence.

The results of the proposed fix are stated in Figure 7.16.
Figure 7.16: Verification results of the manual written specifications for the amended ZG protocol

7.3.4 Specifications of the ZG non-repudiation protocol in OSPSL

In this part, the original version of the ZG protocol is analyzed in the unified verification framework. First, the object meant to define the protocol are declared, as seen in Figure 7.17.
The PRINCIPAL objects define the three principals involved in the protocol run. The public and private keys of each principal are defined as PUBLIC and PRIVATE KEY respectively, while the new session key $K_{ab}$ is declared as being a SYMKEY object.

The DENOTE section of the protocol specification declares the most important notations used in the protocol description. It defined the evidence of origin (EOO), evidence of receipt (EOR), evidence of submission (EOS) and evidence of confirmation (EOC) as being a concatenation of data. These notations are to be used when describing the protocol, reducing the user’s effort of writing data concatenation each time it refers to the non-repudiation necessary evidence.

*Figure 7.18* describes the DENOTE declaration in OSPSL for the ZG protocol.
DENOTE

KaPub = A.pubK;
KaPriv = A.privK;
KbPub = B.pubK;
KbPriv = B.privK;
KttpPub = TTP.pubK;
KttpPriv = TTP.privK;
EOO = dataNRO, B, Na, Kab{dataA};
EOR = dataNRR, A, Na, Kab{dataA};
EOS = dataSUB, B, Na, Kab;
EOC = dataCON, A, B, Na, Kab;

Figure 7.18: DENOTE section in OSPSL for the original ZG protocol

For each principal involved in the protocol run there is a role declaration. Each role contains two sections: ASSUMPTION section where initial role of the principal is described and the MESSAGE section where the set of exchanged messages is declared.

Figure 7.19 presents the role declaration for principal A in the ZG protocol.
ROLE A
{
ASSUMPTION
    A.hold: KaPriv, KaPub, KttpPub, KbPub, Na, Kab,
            dataA, dataNRO, dataSUB;
    A.know: Na.is(FRESH);
    A.believe: KbPriv.isValidKeyTo(B);
    A.believe: KttpPriv.isValidKeyTo(TTP);
MESSAGE
    1 A.send: EOO, KaPriv{EOO};
    2 A.receive: EOR, KbPriv{EOR};
        EXPECTATION:
            A.hold: KbPriv{EOR};
            A.authenticate(B): KbPriv{EOR};
    3 A.send: EOS, KaPriv{EOS};
    5 A.receive: EOC, KttpPriv{EOC};
        EXPECTATION:
            A.hold: EOC, KttpPriv{EOC};
            A.authenticate(TTP): KttpPriv{EOC};
            A.know: EOC.is(FRESH);
}

Figure 7.19: Principal A’s role in OSPSL for ZG non-repudiation protocol

Assumption in principal A’s case, state that before the protocol run A possess it’s private and public key, the nonce Na, the session key Kab, along with the message to be exchanged. They also express the fact that A is aware that TTP and B, each hold, their own private key.

Similarly, the roles of B and TTP are described.
After being parsed and then lexically analyzed, the specifications are translated into the corresponding CDVT specifications.

Figure 7.20 presents the result of the translation:

<table>
<thead>
<tr>
<th>Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0: A possess at[0] (KaPriv, (KaPub, (KttpPub, (KbPub, (Na, (Kab, (dataA, (dataNRO, dataSUB)))))))));</td>
</tr>
<tr>
<td>A1: A know at[0] NOT(Zero possess at[0] Na);</td>
</tr>
<tr>
<td>A2: A know at[0] B possess at[0] KbPriv;</td>
</tr>
<tr>
<td>A3: A know at[0] TTP possess at[0] KttpPriv;</td>
</tr>
<tr>
<td>A4: B possess at[0] (KbPriv, (KbPub, (KttpPub, (KaPub, dataNRR))));</td>
</tr>
<tr>
<td>A5: B know at[0] A possess at[0] KaPriv;</td>
</tr>
<tr>
<td>A6: B know at[0] TTP possess at[0] KttpPriv;</td>
</tr>
<tr>
<td>A7: TTP possess at[0] (KttpPriv, (KttpPub, (KaPub, (KbPub, dataCON))));</td>
</tr>
<tr>
<td>A8: TTP know at[0] A possess at[0] KaPriv;</td>
</tr>
<tr>
<td>A9: TTP know at[0] B possess at[0] KbPriv;</td>
</tr>
<tr>
<td>A10: TTP know at[0] NOT(Zero possess at[0] (dataSUB, (B, (Na, Kab))));</td>
</tr>
<tr>
<td>S0: B receive at[1] ((dataNRO, (B, (Na, {dataA}Kab))), {(dataNRO, (B, (Na, {dataA}Kab)))}KaPriv);</td>
</tr>
<tr>
<td>S1: A receive at[2] ((dataNRR, (A, (Na, {dataA}Kab))),(dataNRR, (A, (Na, {dataA}Kab))))KbPriv);</td>
</tr>
<tr>
<td>S2: TTP receive at[3] ((dataSUB, (B, (Na, Kab))),(dataSUB, (B, (Na, Kab))))KbPriv);</td>
</tr>
<tr>
<td>S3: B receive at[4] ((dataCON, (A, (B, (Na, Kab))),(dataCON, (A, (B, (Na, Kab))))}KttpPriv);</td>
</tr>
<tr>
<td>G0: A possess at[2] {(dataNRR, (A, (Na, {dataA}Kab)))}KbPriv;</td>
</tr>
</tbody>
</table>
G2: A possess at[5]
  \{\text{dataCON}, (A, (B, (Na, Kab)))\}, \{\text{dataCON}, (A, (B, (Na, Kab)))\}\KttpPriv;
G4: A know at[5] NOT(Zero possess at[0] (dataCON, (A, (B, (Na, Kab)))))
G5: A know at[5] NOT(Zero send at[0] (dataCON, (A, (B, (Na, Kab)))))
G6: B possess at[1]
  \{\text{dataNRO}, (B, (Na, \{dataA\}Kab))\}, \{\text{dataNRO}, (B, (Na, \{dataA\}Kab))\}\KaPriv;
G11: B know at[4] NOT(Zero possess at[0]
  \{\text{dataCON}, (A, (B, (Na, Kab)))\}, \{\text{dataCON}, (A, (B, (Na, Kab)))\}\KttpPriv);
G12: B know at[4] NOT(Zero send at[0]
  \{\text{dataCON}, (A, (B, (Na, Kab)))\}, \{\text{dataCON}, (A, (B, (Na, Kab)))\}\KttpPriv);
G13: TTP possess at[3]
  \{\text{dataSUB}, (B, (Na, Kab))\}, \{\text{dataSUB}, (B, (Na, Kab))\}\KaPriv;

Figure 7.20: CDVT translated specification for ZG protocol

7.3.5 Results of the verification for the ZG protocol in OSPSL

The analysis shows that there is a weakness in the protocol. The results of the verification are presented in Figure 7.21.
1. Assumptions
2. Protocol Steps
3. Protocol Verification

<table>
<thead>
<tr>
<th>Step</th>
<th>Assumption/Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A possess af2[0] [dataNRR, (A, (N1, {dataA}, (Kab)))] KtpPriv is True</td>
</tr>
<tr>
<td>2</td>
<td>A know af1[1] B send af2[1] [dataNRR, (A, (N1, {dataA}, (Kab)))] KtpPriv is True</td>
</tr>
<tr>
<td>3</td>
<td>A possess af5[0] [dataCON, (A, (B, (N1, Kab)))] KtpPriv is True</td>
</tr>
<tr>
<td>4</td>
<td>A know af2[1] TTP send af5[1] [dataCON, (A, (B, (N1, Kab)))] KtpPriv is True</td>
</tr>
<tr>
<td>5</td>
<td>A know af3[1] NOT[Zero possess af1[0]] [dataCON, (A, (B, (N1, Kab)))] KtpPriv is True</td>
</tr>
<tr>
<td>6</td>
<td>A know af3[1] NOT[Zero send af1[0]] [dataCON, (A, (B, (N1, Kab)))] KtpPriv is True</td>
</tr>
<tr>
<td>7</td>
<td>B possess af1[1] [dataNRR, (B, (N1, {dataA}, (Kab)))] KtpPriv is True</td>
</tr>
<tr>
<td>8</td>
<td>B possess af4[1] [dataCON, (A, (B, (N1, Kab)))] KtpPriv is True</td>
</tr>
<tr>
<td>9</td>
<td>B possess af4[2] Kab is True</td>
</tr>
<tr>
<td>10</td>
<td>B possess af4[3] dataA is True</td>
</tr>
<tr>
<td>11</td>
<td>B know af3[1] TTP send af4[1] [dataCON, (A, (B, (N1, Kab)))] KtpPriv is assumed False</td>
</tr>
<tr>
<td>12</td>
<td>B know af4[2] NOT[Zero possess af1[0]] [dataCON, (A, (B, (N1, Kab)))] KtpPriv is assumed False</td>
</tr>
<tr>
<td>13</td>
<td>B know af4[2] NOT[Zero send af1[0]] [dataCON, (A, (B, (N1, Kab)))] KtpPriv is assumed False</td>
</tr>
<tr>
<td>14</td>
<td>TTP possess af3[1] [dataSUB, (B, (N1, Kab))], [dataSUB, (B, (N1, Kab))], KtpPriv is True</td>
</tr>
<tr>
<td>15</td>
<td>TTP know af3[1] A send af3[1] [dataSUB, (B, (N1, Kab))], KtpPriv is True</td>
</tr>
<tr>
<td>16</td>
<td>TTP possess af3[2] Kab is True</td>
</tr>
</tbody>
</table>

Figure 7.21: Verification results for the OSPSL translation of ZG protocol

As seen in Figure 7.21, the result shows the original Zhou-Gollmann protocol as being false. The problem detected is the same, being caused by the failure of principal B to ensure validity of the confirmation evidence received in the protocol step S4.

7.3.6 Specification for the amended ZG protocol in OSPSL

The attack assumes that the protocol has been previously executes successfully by principals A and B, and that the intruder has been maliciously recorded the messages exchanged in the previous protocol run. During the attack, the intruder takes on the roles of both principal A and TTP.

The attack begins with the intruder impersonating principal A by sending to B the initial message from the previous recorded exchange. B, who assumes that the message is sent by principal A, replies with the second step as per the protocol specification. The intruder again takes on the role of A and intercepts the message. The third step is omitted as the intruder impersonates both, principal A and the TTP. In the fourth step the intruder masquerades as the TTP and sends to B the fourth message from the recorded session. Analogous to step 3, step 5 is also omitted.
At the end of the protocol run, B is convinced he had exchanged a new message with principal A, when in fact he did not.

### 7.3.7 Amended version of the ZG non-repudiation protocol

In order to fix the flaw, B needs to have a proof in the message received from the TTP to confirm that this message belongs to the current protocol run. It is proposed to include a second nonce Nb, which is generated by principal B, in the exchanged messages: In the second step B submits this nonce as part of its message to A, who will forward it as part of step 3 to the TTP. Inclusion of both nonces Na and Nb into the confirmation evidence distributed by the TTP ensures that both, principals A and B, are able to establish freshness of this evidence. The protocol steps of the proposed amended protocol are presented in Figure 7.22. Operation of the amended protocol is analogous to the original protocol as detailed in Section III, but includes the nonce Nb in steps 2 to 5.

1. \(A \rightarrow B: \text{dataNRO,B,Na,\{dataA\}Kab, \{dataNRO,B,Na,\{dataA\}Kab\}KaPriv}\)
2. \(B \rightarrow A: \text{dataNRR,A,Na,Nb, \{dataA\}Kab,\{dataNRR,A,Na,\{dataA\}Kab\}KbPriv}\)
3. \(A \rightarrow \text{TTP}: \text{dataSUB,B,Na,Nb,Kab, \{dataSUB,B,Na,Nb,Kab\}KaPriv}\)
4. \(\text{TTP} \rightarrow B: \text{dataCON,A,B,Na,Nb,Kab, \{dataCON,A,B,Na,Nb,Kab\}KttpPriv}\)
5. \(\text{TTP} \rightarrow A: \text{dataCON,A,B,Na,Nb,Kab, \{dataCON,A,B,Na,Nb,Kab\}KttpPriv}\)

Figure 7.22: Amended version of the ZG protocol [MDC09]

### 7.3.8 Validation of the ZG amended version

The proposed fix is specified in analyzed in the unified verification framework. The amended protocol is specified in OSPLS and analyzed with CDVT. The results of the analysis are presented in Figure 7.23.

The analysis confirms that no attacks are possible on the amended version of ZG protocol.
Chapter 7. Empirical validation of the translator

1. Assumptions
2. Protocol Steps
3. Protocol Verification

- Protocol is Verified is True
  - (3): A possess at[5] {dataCON,(A,B,(Na,((Na,(Nb,Ka)))))}{(dataCON,(A,B,(Na,((Na,(Nb,Ka))))))}KtpPriv is True
  - (5): A know at[5] NOT(Zero possess at[0] (dataCON,(A,B,(Na,((Na,(Nb,((Na,(Nb,Ka)))))))))) is True
  - (6): A know at[5] NOT(Zero send at[0] (dataCON,(A,B,(Na,((Na,(Nb,Ka)))))) is True
  - (7): B possess at[1] {dataNRO,(B,((Na,((dataA,Ka))))))KapPriv is True
  - (8): B possess at[4] {dataCON,(A,B,(Na,((Na,((Nb,Ka))))))}KtpPriv is True
  - (9): B possess at[4] Kap is True
  - (10): B possess at[4] dataA is True
  - (12): TTP possess at[3] {dataSUB,(B,((Na,((Na,(Nb,Ka))))))}KapPriv is True

Figure 7.23: Verification of the ZG amended protocol generated specs in OSPSL

7.3.9 Conclusions after the verification of the ZG protocol

The results obtained are the same for the manual written specifications, as well as for the generated specifications in OSPSL. For the Zhou-Gollmann protocol the freshness weakness is discovered in both cases. For the original version, both verifications

On the opposite side, for the amended version, no attacks were found when analyzing the manually written specs, nor the generated ones. The fact that the verification results are all leading to the same results proves the correctness of the translation process.

7.4 Empirical validation of security protocols

This section presents an empirical study conducted over a set of benchmark protocols for secrecy and authentication. The purpose of this empirical study is to validate the translation process from OSPSL to CDVT, by showing the equivalence of the analysis results of the manually written specifications in CDVT and the ones generated in the unified verification framework.
The comparative study is carried over a set of protocols for secrecy and authentication and it contains both secure and insecure protocols. In the validation study the result of the CDVT analysis is compared with the analysis results after considering the generated specifications in OSPSL. Table 7.1 synthesizes a comparison on the results obtained when analyzing manually written specifications in CDVT and generated OSPSL CDVT specifications, for the same protocols.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>CDVT</th>
<th>OSPSL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Attacks found</td>
<td></td>
</tr>
<tr>
<td>ASK</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>CDFASK</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>BCY</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>CDFBCY</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>ASPeCT</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>KAO-CHOW</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>KAO-CHOW AMENDED</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>TMN</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>WMF</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>WMF LOWE</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>YAHALOM</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>ZHOU GOLLMANN</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>ZHOU GOLLMANN AMENDED</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

Table 7.1: Comparison table with the attacks found on a given set of protocol when analyzing using OSPSL generated specs or CDVT manually written specs

In the second and third column, the results of the analysis are represented. If there is an attack, then Y confirms that the attack is detected and an N if the attack is not detected.
The results of the study confirm that analyzing the generated files obtained in the translation and the manually written specifications in CDVT, lead to the same results and therefore validates the translation process. All attacks detected when considering manually written specifications are detected also when analyzing the generated specifications in OSPSL.

7.5 Conclusions

In this chapter, an empirical study was performed on a series of protocols. The purpose of this chapter is to validate the logic analysis of security protocols in the unified verification framework and the translation process from the unified verification framework to CDVT tool. This validation is based on an empirical study in which a set of protocols are manually specified in CDVT and analyzed. The same protocols are then validated through OSPSL and their generated specification files in CDVT are analyzed against the same tool.

The first section analyses three protocols: Kao-Chow, BCY and Zhou-Gollmann. The same flow is considered for all protocols, when their validation is compared against manually written specifications.

The Zhou Gollmann non-repudiation protocol is presented in detail by describing its manually written specifications in CDVT and the results revealed after analyzing this specifications. The existence of a freshness flaw in the protocol is outlined and the [MDC09] amended version is validated, by analyzing it in CDVT.

The same original protocol is then analyzed using OSPSL. The specifications are described and the generated file is presented and validated by analyzing it against the same tool, leading to the same results obtained for the manually written specifications.
A typical security schema consists of a number of principals such as people, computers, companies, magnetic card readers, etc. which communicate using a variety of channels like phones, radio, email, while carrying data through physical devices such as tickets, cards, etc. The security protocols are the rules which guide and govern all these communications. They should be designed in such a way that they provide security over insecure networks.

Protocols may be extremely simple (swiping a card through a card reader to enter the building, a conversation between two persons over the phone, etc.) or very complex.

Security protocols are part of everyday life, but their design becomes more and more difficult when interacting with different environments. In recent years, formal methods have shown their applicability in formal verification of security protocols and they became more and more popular with increasing complexity of modern protocols (non-repudiation, Diffie-Hellmann key exchange, etc.).

In this thesis, the unified verification framework is described. The framework integrates both state space based techniques and logic based techniques for verifying security protocols. The specification language of the framework (OSPSL) was specially designed to allow logic based techniques to be integrated in the framework. The language develops a set of methods that map directly over the operators of logic and belief used in verifying security protocols with logic based techniques, making it easier to specify complex protocols such as non-repudiation protocols, Diffie-Hellmann protocols, etc.

The analysis of security protocols in the unified verification framework using modal logics is validated through an empirical study. For this purpose, simple and complex protocols were chosen to analyze the behavior and results of the verification. These protocols are specified in OSPSL and analyzed through the logic based verification tool CDVT. A comparison is made between the manually generated specifications in CDVT and their results. The study reveals the
equivalence of results between OSPSL and CDVT, validating the translation between these two frameworks.

The objectives of the thesis have been reached, now being possible to specify security protocols using a high level specification language and analyze them using logic based verification techniques. The translator designed and proposed takes the high level protocol specification in OSPSL, translates them to the low-level logic based tool CDVT and outputs the results in the logic-based tool format.

### 8.1 Future perspectives of the unified verification framework

The unified verification framework is still in developing process, being under continuous improvement. This section analyzes the framework from the modal logics point of view.

In Chapter 5 it is mentioned that OSPSL has some limitations when describing group protocols. This approach is being developed through the PRINGROUP type, where more participants can be defined and treated as a group. However, considering the logic based approach of the framework, there are no limitations for the number of participants involved in a communication. The protocols that have security claims such as non-repudiation and non-inference are treated using this approach.
CHAPTER 9. References


97


98


Carla Lucia Mirona Muntean

Chapter 9. References


Protocol specification in OSPSL is composed of keywords, delimiters and statements. Semantics of this language is based on the interpretations of the statements. The composition of statements is defined within the type definitions (presented in Appendix A), where a syntactic expression returning STATEMENT type is considered as a valid statement. Therefore, the semantics of this language is relied on the interpretation of the type definitions.

**Symbols.**

\( \land \) conjunction \hspace{1cm} \( \lor \) disjunction

\( \rightarrow \) mathematical implication \hspace{1cm} \( \neg \) complementation

\( \forall \) universal quantification \hspace{1cm} \( \exists \) existential quantification

\( \in \) membership of a set \hspace{1cm} \( \subseteq \) subset of a set \hspace{1cm} \( \supseteq \) superset of a set

\( \cap \) join of two sets \hspace{1cm} \( \cup \) meet of two sets \hspace{1cm} / \hspace{1cm} set exclusion.

**Notation of Sets.**

\( E: \) the set of all possible entities.

\( \Omega: \) the set of legal entities in a protocol.

\( \Xi: \) the set of trusted entities in a protocol.

\( \Delta: \) the set of statements in a protocol.

\( T: \) the set of discrete time in a protocol.

\( \Psi: \) the set of message in a protocol.
\( K \): the set of crypto – keys in a protocol.

**Notation of Predicate Operators.**

\( \Pi \): emission operator. \( \Sigma \): reception operator.

\( \Theta \): possession operator. \( \Phi \): knowledge operator.

\( \Gamma \): belief operator.

**Other Notations.**

\((m_1, m_2)\): the concatenation of \(m_1\) and \(m_2\). \( k^{-1}\): the inverse key of \(k\).

\( k\{m_1\}\): the encryption of \(m_1\) using \(k\). \( O(k)\): the owner of \(k\).

\(*m_1\): \(m_1\) is fresh.