THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS & SCIENCES

FORMALLY EVALUATING WIRELESS SECURITY PROTOCOLS

By

ILKAY CUBUKCU

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Spring Semester, 2005

The members of the Committee approve the Thesis of Ilkay Cubukcu defended on April 4, 2005.

Alec Yasinsac
Professor Directing Thesis

Ladislav Kohout
Committee Member

Robert A. van Engelen
Committee Member

The Office of Graduate Studies has verified and approved the above named committee members.

To my husband Nihat and baby daughter Ilkem.

ACKNOWLEDGEMENTS

I wish to express my sincere gratitude to Alec Yasinsac, my advisor, for his encouragement, guidance during the course of this study. Thank you for always being there and helping me whenever I needed. My biggest thanks go to the past and presents members of the security group meets weekly under the supervision of Dr. Yasinsac. Your ideas, suggestions were always helpful for my research. It was also fun to have you around, thank you for the friendship. A special thanks to John Marshall for the valuable discussions especially on evaluation of the wireless protocols with CPAL-ES.

I would like to thank to all of my colloquies and friends for their greatest support and friendship.

Finally, my greatest thanks go to my dear husband, daughter and family for their patience and endless support. This work would not be possible without your help.

TABLE OF CONTENTS

LIST OF TABLES viii
LIST OF FIGURES ix
ABSTRACT x

1 INTRODUCTION 1

2 PROTOCOL ANALYSIS OVERVIEW 5

2.1 Common Procedures used for Analyzing Protocols 5
2.1.1 BAN Logic 5
2.1.2 Strand Spaces 8
2.1.3 NRL Protocol Analyzer 9
2.1.4 The Cryptographic Protocol Language Evaluation System (CPAL-ES) 11

3 PROBLEM DESCRIPTION 19

3.1 The Secure Protocol 19
3.1.1 Protocol Description and Design 19
3.1.1.1 Definition 19
3.1.1.2 Design 19
3.1.1.3 Description 23
3.1.1.4 Message Format 24
3.1.2 Attacks 29
3.1.2.1 Meadows Attack 29
3.1.2.2 Boyd & Mathuria Attack 33
3.2 IEEE 802.1X Standard 36
3.2.1 Protocol Description and Design 36
3.2.1.1 Definition 36
3.2.1.2 Design 36
3.2.1.2.1 IEEE 802.11 Network Design and Security Mechanisms 37
3.2.1.2.2 IEEE 802.1X Standard and the Robust Security Network (RSN) Design and Security issues 38
3.2.1.3 Description 40
3.2.1.4 Message Format 42
3.2.1.4.1 EAP Message Format 42
3.2.1.4.2 RADIUS Message Format 44
3.2.2 Attacks 51

3.2.2.1 Man-In-The-Middle (MIM) Attack 51
3.2.2.2 Session Hijacking Attack 53
3.2.2.3 Denial of Service Attack 54
3.2.2.4 Proposed Solutions for Attacks 54
3.2.2.4.1 Per-Packet authenticity and integrity 54
3.2.2.4.2 Authenticity and integrity of EAPOL messages 55
3.2.2.4.3 Peer-to-peer authentication model 55

4 CPAL EVALUATION 56

4.1 CPAL-ES Evaluation of the Secure Protocol 56
4.1.1 CPAL-ES Evaluation of the Meadows Attack on the Secure Protocol 70
4.1.2 CPAL-ES Evaluation of the Boyd & Mathuria Attack on the Secure Protocol 77
4.1.3 CPAL-ES Evaluation of the Solution to the Boyd&Mathuria Attack on the Secure Protocol 82
4.2 CPAL Evaluation of the IEEE 802.1X Protocol 85
4.2.1 CPAL Evaluation of the Man In the Middle (MIM) Attack on the IEEE 802.1X Protocol 95
4.2.2 CPAL Evaluation of the solution to the Man-in-the-Middle (MIM) Attack on the IEEE 802.1X Protocol 97

5 CONCLUSIONS 99

APPENDIX A: CPAL-ES Encoding of the Secure Protocol 101

APPENDIX B: CPAL-ES Evaluation of the Secure Protocol 103

APPENDIX C: CPAL-ES Encoding of Meadows Attack on the Secure Protocol 105

APPENDIX D: CPAL-ES Evaluation of Meadows Attack on the Secure Protocol 108

APPENDIX E: CPAL-ES Encoding of Boyd & Mathuria Attack on the Secure Protocol 111

APPENDIX F: CPAL-ES Evaluation of Boyd & Mathuria Attack on the Secure Protocol 114

APPENDIX G: CPAL-ES Encoding of Solution to Boyd & Mathuria Attack on the Secure Protocol 117

APPENDIX H: CPAL-ES Evaluation of Solution to Boyd & Mathuria Attack on the Secure Protocol 119

APPENDIX I: CPAL-ES Encoding of IEEE 802.1X Protocol 121

APPENDIX J: CPAL-ES Evaluation of IEEE 802.1X Protocol 125

APPENDIX K: CPAL-ES Encoding of MIM (Man-in-the-Middle) Attack on IEEE 802.1X Protocol 129

APPENDIX L: CPAL-ES Evaluation of MIM (Man-in-the-Middle) Attack on IEEE 802.1X Protocol 133

APPENDIX M: CPAL-ES Encoding of Solution to MIM (Man-in-the-Middle) Attack on IEEE 802.1X Protocol 137

APPENDIX N: CPAL-ES Evaluation of Solution to MIM (Man-in-the-Middle) Attack on IEEE 802.1X Protocol 141

REFERENCES 145

BIOGRAPHICAL SCETCH 149

LIST OF TABLES

Table 2.1.1 Symbols of objects in BAN Logic. 6
Table 2.1.2 An example protocol flow in SN and CPAL notation 14
Table 3.1.1 List of acronyms for the Secure Protocol 20
Table 3.1.2 List of acronyms for the Secure Protocol and attacks 30
Table 3.2.1 List of acronyms for the IEEE 802.1X protocol. 49
Table 4.1.1 List of acronyms for the Secure Protocol in CPAL 58
Table 4.2.1 List of acronyms for the Secure Protocol in CPAL 88

LIST OF FIGURES

Figure 3.1.1 Entire Secure Protocol for wireless networks [AZI94]. 21
Figure 3.1.2 Message-1: Mobile-to-Base (Request-to-join message) 25
Figure 3.1.3 Message-2: Base-to-Mobile 26
Figure 3.1.4 Message-3: Mobile-to-Base 28
Figure 3.1.5 Original Secure Protocol 31
Figure 3.1.6 Meadows Attack 32
Figure 3.1.7 Boyd & Mathuria Attack 34
Figure 3.1.8 Solution to Boyd & Mathuria attack 35
Figure 3.2.1 The Classic 802.11 state machine. 37
Figure 3.2.2 The EEE 802.1X setup. 39
Figure 3.2.3 The EAP stack. 41
Figure 3.2.4 The EAP Packet. 42
Figure 3.2.5 The RADIUS Packet 45
Figure 3.2.6 The complete IEEE 802.1X protocol. 50
Figure 3.2.7 Man-In-The-Middle (MIM) Attack. 52
Figure 3.2.8 Session Hijack Attack 53

ABSTRACT

The Cryptographic Protocol Analysis Language Evaluation System (CPAL-ES) is a tool used to analyze protocols with formal methods. In this thesis, we exercise CPAL-ES against two security protocols, the Secure Protocol of Aziz & Diffie, and IEEE 802.1X Standard protocol.

Analyzing cryptographic protocols with formal methods assist us not only finding the flaws but also in understanding them. CPAL-ES is a nice tool to analyze protocols with formal methods. It has an ability to evaluate not only protocols works in wired environment but also wireless protocols. Our analysis with CPAL-ES makes it possible to explore protocol attacks, prove protocol correctness, and analyze protocols in great detail, as well as test the capabilities of CPAL-ES on the wireless protocols. We discuss and analyze several protocols, including The Secure Protocol and IEEE 802.1X Standard protocol, and show how attacks and solutions are simulated on these protocols with Cryptographic Protocol Analysis Language (CPAL). We also discuss the analysis of the interactions between the sub-protocols (EAP and RADIUS) in IEEE 802.1X Standard protocol. Our analysis of the attacks on the IEEE 802.1X Standard protocol proved that even though it is a useful protocol for wireless LANs, it is not secure. However, the Secure Protocol has strong confidentiality but is computationally expensive due to the public key infrastructure.

CHAPTER 1

INTRODUCTION

The Internet and computers are very important part of our lives today (in schools, banks, battlefields, airports, shopping centers etc.). Life as we know today depend on these and there will be even more use of them in the future. Wireless technology is an essential part of this technological area.

Wireless mobile technology was invented in Bell Laboratories at the second half of the 20th century after the discovery of two way radio communication systems [BZM01]. Wireless technology became even more popular at the beginning of 21st century. The first generation wireless mobile cellular systems (1G) were introduced in early 1980's. They were entirely analog, based on FDM (frequency division multiplex) technology and used in mostly briefcase size large phones placed in vehicles. At the end of 1980s, people needed smaller size phones for communicating. Then, came the second generation cellular industry (2G) such as GSM (Global System for Mobile communications) and PDC (Personal Digital Cellular) were based on digital technology in early '90s. 2G systems improved voice quality, capacity and coverage and reduced the cost. At the beginning of the 21st century, the third generation cellular standard (3G) further increased the system capacity and provides better voice quality, high speed data, low cost and efficient systems [BZM01]. The fourth generation cellular system (4G) is intended for better quality, service and low cost in the 21st century and it is expected to be introduced around 2010. Data transmission rate for 4G systems is 20mbps while it is 200 kbps for 3G systems and 9.6 kbps for 2G systems.

When we think about sharing data and resources between different machines through the network, security becomes a very important issue. Increasing dependence on wireless medium makes it critical to have very secure wireless systems. Wireless networks introduce new possibilities of eavesdropping on communications. The potential attacks are categorized in terms of functioning as *interruption* (system is

destroyed, unavailable and unusable), *interception* (an unauthorized party gains access to the system), *modification* (an unauthorized party gains access and change or corrupt the system), *and fabrication* (an unauthorized party inserts a bogus object into the system) [WIN00]. Another categorization is *passive attacks* and *active attacks* [WIN00]. Passive attacks involve eavesdropping and monitoring the transmission system while active attacks involve modifying the data or creating false data. Since passive attacks don't change the data, it is very difficult to detect these attacks. Therefore, preventing these attacks is much easier than detecting whereas the opposite is true for active attacks. There are two types of passive attack: Release of message content, and traffic analysis. Active attacks are four types: Masquerade, reply, modification of messages, and denial of service. There are many problems inherent in the wireless medium. Any attacker that has a proper wireless receiver can easily eavesdrop on the system and it can be virtually undetectable. We need secure wireless communication protocols in order to prevent attacks [AZI94].

There has been a lot of work done on finding the security flaws and solutions, and new secure techniques in order to maintain reliable and secure communications [ABA99, ARB01, AZI94, BOR01, BOY98, COR02, DON98, FAB98, KEL98, LOV96, MAR03, MEA96, MIS02, SCH96a, SON01, STU01, WIN98, YAS96, YAS00a, YAS00b, and others]. Nonetheless, there is no perfect security with the existing network architecture for wireless networks.

Security protocols are important in providing security. They play an important role for protecting the electronic information shared between parties. Basically, they are the algorithms used for secure communications between different principals. An eavesdropper can get the important information anytime if the system is not secure enough. When principal A wants to communicate securely with principal B, there should be a secure channel between these parties in order to share the data. The channel should be well protected from attackers. Even though the protocols are secured themselves when they execute individually, they may be vulnerable to attacks when interacted with other simultaneously executing protocols [KEL98].

Application of cryptography is essential for secure communications. It is defined in the *Handbook of Applied Cryptography* by A. Menezes Et al. [MOV96] as "The study

of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin authentication". They also mentioned that it is not only providing information security but also set of techniques [MOV96]. The definition of cryptographic protocol stated in this book as "a distributed algorithm defined by a sequence of steps precisely specifying the actions required of two or more entities to achieve a specific security objective" [MOV96].

Security is achieved by using cryptographic tools such as public and private key cryptography, symmetric key cryptography, certificates, signatures, hashing, key agreement and some others. These tools are used to achieve the security goals of *authentication*, *confidentiality* and *integrity*. Addition to these, *nonrepudiation* prevents the sender or the receiver from denying a message that is transmitted between the sender and the receiver, *access control* limits and controls access to the systems, and *availability* prevents system against to attacks due to the loss of availability or reduction [WIN00]. Authentication assures that the communication is authentic means that both entities are authentic and the connection is not imperceptible by a third party. Confidentiality is concerned with protection of the transmitted data by providing the privacy and secrecy on the message content. The goal is preventing the interception type of attacks against the system. Integrity deals with the modification of the transmitted message. It assures that there is no duplication, insertion, modification, reordering or replays. The system is protected against modification or active attacks. The method of integrity prevention can be using message digest (MD) functions or hashing. For more information on MD or Hash functions see [WIN00]. We have some examples of these in the following sections.

Formal methods are used for modeling and verifying security protocols [PAU97]. Formal methods are tools used for protocol design, verification and then specifications use mathematically designed and well formed logical statements. Each formal method uses different modeling and proving techniques (see Chapter 2). Formal methods have been used by many researchers to investigate the known and unknown attacks and provide the solutions to these attacks for secure communications [ABA99, BUR90, BUT99, CHI01, GAA90, KEM89, LOW96, MAR03, MEA95, MEA96, MEA99, MEA00, MEA03, RUB93, SCH96a, SCH96b, WIN98, WOO93, YAS96, YAS99, and others].

Some of them [BUR90, LOW96, MEA96, SCH96b and others] used Needham-Schroeder public-key protocol as test case since it is one of the earliest protocol that has been found, well known and simple. These techniques have been used for the security analysis of both the wired and wireless communication protocols.

In the following section, we have provided some common techniques that have been used to analyze cryptographic protocols. Later, the definition, description, design and message format of both the Secure Protocol [AZI94] and IEEE 802.1X Protocol [MIS02] are discussed in Chapter 3. In Chapter 4, we have presented the CPAL-ES protocol evaluation system and using CPAL-ES analyzed the Secure Protocol and IEEE 802.1X Protocol. Finally, the results are discussed in the last chapter.

CHAPTER 2

PROTOCOL ANALYSIS OVERVIEW

2.1 Common Procedures used for Analyzing Protocols

Different techniques have been developed to verify security protocols such as [BUR90], [DON99], [FAB98], [LOW96], [LOW98], [MEA99], [SON01], [YAS96] and many others. Most of these verification procedures use formal methods and they are automated for fast and easy verification on complex protocols. We have chosen the CPAL-ES to evaluate the selected protocols. CPAL-ES and some other methods are discussed below.

2.1.1 BAN Logic

BAN Logic, logic of beliefs, is developed by Burrows, Abadi and Needham [BUR90] in 1989 and published the revised version in 1990. As stated by Meadows [MEA95], BAN Logic is the well known, most influential of the modal logics developed for cryptographic analysis of protocols such as Rangan's logic of trust [RAN88], Moser's logic of knowledge and belief [MOS89], Bieber's logic of communication (CKT5) [BIE90], Syverson's logic of cryptographic protocols (KPL) [SYV90], and Yahalom, Klein and Beth's logic of trust [YAH93].

With BAN Logic, logical rules are used to analyze protocols by transforming the protocols into a special form using its own formal notation. BAN Logic is designed for analyzing protocols by using the predicates to express the assumptions and the final beliefs for the authentication. Beliefs of the protocols are checked with assumptions to see whether the goals are reached. BAN Logic supports most of the central concepts in the protocol, but not all the authentication methods [BUR90].

BAN Logic is built on many-sorted-model logic. Each message is transformed into a logical formula through idealization, in order to have more useful notation than conventional methods, which describes the protocols and protocol actions symbolically. Each idealized protocol is annotated with an assertion which describes believes of the principals at the point where assertion is inserted in the protocol [BUR90].

Logical formulas, called statements, identify the messages. The encryption keys are the objects of BAN logic. Typically, the objects are denoted and range over as the following symbols shown in Table 2.1.1 [BUR90]:

Table 2.1.1 Symbols of objects in BAN Logic.

A, B, and S	denote	specific principles
Kab, Kas, Kbs	denote	specific shared keys
Ka, Kb, Ks	denote	specific public keys
Ka^(-1), Kb^(-1) and Ks^(-1)	denote	corresponding secret key
Na, Nb, Nc	denote	specific statements
P, Q, and R	range over	principles
K	ranges over	encryption keys
X, Y	range over	statements

Conjunction is denoted by a comma and it is the only propositional connective used in BAN Logic. In addition to conjunction, the constructs such as *P believes X*, *P sees X*, *P once said X* are used to analyze the protocol step by step. These are the predicates of BAN Logic. More of these and their notations are explained in great detail in [BUR90].

BAN Logic uses inference rules to deduce the properties of protocol and make annotations to find the final belief of the protocol. Idealized protocols are annotated with

formulas and these formulas are manipulated with postulates. Annotation for a protocol is the sequence of assertions that are inserted before the first statement and after the each statement. The first assertion is the assumptions, and the last assertion is the conclusion.

The Ban Logic inference rules are jurisdiction rule, message meaning rule and nonce-verification rule. For more rules and details refer to [BUR90].

There are four steps to analyze the protocols with BAN Logic:

- Deriving the idealized protocol from the original protocol,
- Expressing the assumptions about the initial state as statements,
- Transforming the statements into logical formulas by attaching the formulas to the statements as assertions about the state of the system
- Applying the inference rules called postulates to assumptions and the assertions to find beliefs.

These steps repeated when new assumptions are discovered and until there is no idealized protocol derived.

Authentication of the protocols with the BAN Logic divided into two time epochs, the past and the present where the presents begins at the start of the particular run of the protocol consideration. All messages sent before the present are considered to be in the past. Burrows et al. states that “all the beliefs held in the present are stable for the entirety of the protocol run... However, beliefs held in the past are not necessarily carried forward into the present” [BUR90]. This approach is changed by Gaarder et al. as a formula generated earlier in the past could be still fresh. They extend the freshness to the past by using durationstamps that could be verified by using the shared time system. [GAA90]. Another addition by Gaarder Et al. is that BAN Logic is not restricted to Symmetric Key Crypto System and could be extended to use for Public Key Crypto System. Therefore, they have added new constructs such as “*U possesses good public*

key K", "*U possesses some good private key*…", and "*The formula X is signed with the private key*..." [GAA90].

Gong et al. (GNY Logic) is another approach that expands the BAN Logic idea by adding new logical operators such as not-originated-here, reconcilability, the difference between believing and possessing to run the protocol more precisely.

In addition to these, more problems have been discovered with BAN logic that a few flaws had been discovered such as by Boyd Et al. They have suggested some limitations to BAN Logic [BOY93].

2.1.2 Strand Spaces

F. J. T. Fabrega and J. C. Herzog have proposed a protocol analyzing technique called strand spaces to prove correctness of the protocols. It is a simple model and produces reliable proofs of protocol correctness. The machinery of strand spaces model is described in detail in F. J. T. Fabrega Et al. [FAB98]. They have applied this technique to prove the correctness of Needham-Schroeder-Lowe protocol by having detailed view of secrecy protection of exchanged values, achieving authentication and finding the reasons of flaws. It uses partial order technique and induction-like proof method.

In the strand spaces machinery, a strand represents a sequence of events where it is the sequence of actions done by a penetrator (such as send and receive operations of a penetrator) or the execution of legitimate party has a role in a secure protocol (such as send and receive actions of a party in a particular run of a protocol or specific values of data items such as keys and nonces). A strand space is a collection of strands for a legitimate party along with penetrator strands. A bundle is a collection of number of strands where a strand sends the message with another strand that receives the same message. In other words, it is a portion of a strand space where it represents a full exchange of protocol. Each bundle consist of a strand for each legitimate participating party that all must agree on the participants, session keys and nonce. Penetrator strands could participate in a bundle too [FAB98].

In this model, all these actions are represented with a graph structure. The connections between the strands of different kinds represent protocol correctness. The possible messages exchanged among the principals represents the elements of the set *A* are called terms. A positive sign represents sending a term and negative sign represents receiving a term. A strand space is represented as a set with trace mapping where the traces may be originated either from the different principals or the same principals. Therefore the mapping does not need to be injective. A node is represented as a pair of an element of set of strand spaces (a unique strand) and the length of the set. The set of nodes becomes an ordered graph where each edge represents a connection from a node to another node. A bundle is represented as a finite subgraph of the graph where the edges represent the casual dependencies of the nodes. The graph structure of strand spaces discussed in great detail in F. J. T. Fabrega Et al. [FAB98].

Some of the advantages of strand spaces approach as mentioned in F. J. T. Fabrega Et al. [FAB98] are as follows: Strand Spaces offer a few different notions of proof of protocol correctness by using the authentication and secrecy proving methods. Second, it gives the assumptions and the reasons of protocol correctness in great detail. Another advantage is that it has a clear semantics for the assumptions and data items like nonce and session keys are fresh, unique for each protocol run that is very important for the security of the protocol. Finally Strand Spaces works with an explicit model for possible behaviors of a system penetrator. This is important for generating theorems on the abilities of penetrator.

2.1.3 NRL Protocol Analyzer

A special tool of protocol analyzing with formal methods, NRL Protocol Analyzer, is designed for cryptographic protocol verification by C. Meadows [MEA99]. It was used to analyze Internet Key Exchange (IKE) protocol which is developed to provide security support for Internet protocols by IP Security Protocol (IPSEC) Working Group of the Internet Engineering Task Force (IETF).

In the NRL Protocol Analyzer Model, protocols are represented as communicating state machines where each single state machine represents a

participant. An intruder can read, modify or delete the traffic, take a role in cryptographic operations, or communicate with legitimate users. A round represents a local execution where a local execution is a role such as responder or initiator, respect to a party or particular local execution of the protocol. Each round is represented with a round number. In order to make a decision on whether or not the protocol is secure, the user of the Analyzer specifies an insecure state and works backwards from that state until it finds a search space. Therefore each produced path begins with an initial or unreachable state. In order to limit the search space, the State Unifier tool is used to store the data that is proved by an Analyzer as unreachable or reachable under certain conditions. Unreachable states determined and proved by an Analyzer are discarded. If the state is proved as reachable under certain conditions it will try to prove that the conditions can hold [MEA99].

As defined by C. Meadows [MEA99], narrowing is a process that Analyzer determines whether or not a protocol rule could be used to identify an insecure state. The narrowing algorithm is used to find all the substitutions to the variables used in the execution. Terms in the rule output are reducible to the terms in the state description where a term is an expression made from variables, symbols for functions and constants, and used in the protocol specifications. Terms are assumed to obey a set of rewrite rules (for example, the result of decrypting and encrypted term under the same key reduces to the original term). Terms in the state description are assumed to be irreducible which means that there are no further rewrite rules apply. However, terms used as output of rules may not be reducible.

There is no assumption made by analyzer about the limits on the number of executions such that the number of principals participates in different executions, the number of protocol executions, and number of interleaved executions or the number of cryptographic functions used in the executions. Therefore, the search space is originally infinite. On the other hand, Analyzer specifies and proves inductive lemmas by using formal methods on the unreachability of infinite state classes. With this result, the user can narrow the search space. The user provides a seed term to the Analyzer and Analyzer uses it to generate a formal language to prove it. If an intruder determines the

language then it is possible that it can get the data. Therefore, it is important to prove that an intruder never learn a term in the language.

If the protocols are using shared or public key encryption, the seed terms will be master keys, encrypted and decrypted data and terms, concatenation of two terms, and signed data where none of this information is known by an intruder.

C. Meadows has applied the Analyzer Model to a few different cryptographic protocols and found flaws in some of them. More information o these flaws, solutions to the flaws and more details of Analyzer model can be found in [MEA99].

2.1.4 The Cryptographic Protocol Language Evaluation System (CPAL-ES)

CPAL-ES is developed for analyzing cryptographic protocols with formal methods by A. F. Yasinsac in 1996 [YAS96]. It is based on the Weakest Precondition (WP) reasoning which is extension of Hoare logic [DIJ76]. Hoare Logic was knowledge of postconditions to tell us how to find the preconditions where when the program segment is executed with those initial preconditions then that certain postcondition will hold after the execution. This is defined with the following notation:

P {S} Q,

Where *P* is precondition, *S* state and *Q* is the postcondition. We can explain this with the following example:

(y ==5) {y := 2x+1} (x == 2)

In this example, in the case of the execution of the statement "*y=2x+1*", in order the postcondition (*y==5*) to be true after the execution, the precondition *x==2* must hold before the statement is executed.

The extension of Hoare's precondition and postcondition system is described as weakest precondition for a program segment by Dijkstra [DIJ76]. The relation between the weakest precondition Q [DIJ76] for segment *S* and postcondition *R* are defined as:

$$Q = wp(S, R)$$

Dijkstra defines the predicate as a rule which informs the designer about how to derive the corresponding weakest precondition for any postcondition R and an initial state [DIJ76].

CPAL statements and segments are represented with predicates by using the weakest precondition mechanism. The weakest precondition definitions of CPAL statements are given in [YAS96]. For instance, the weakest precondition definition of CPAL catenation statements is:

$$wp(S1; S2, P) = wp(S1, wp(S2, P))$$

Where S1 and S2 are the statements and P is the postcondition. In order for predicate P to be TRUE after "S1; S2" is executed, the weakest precondition of S2 for P must be TRUE after execution of S1 for P [YAS96].

As stated by Yasinsac, if a predicate is described that all the states enables the program segment to the desired result then the preconditions considered to be "weakest" and the weakest precondition of a segment is Verification Condition (VC) for that segment. In order to prove that the segment meets its goals by using WP, CPAL evaluation starts from reverse and the value TRUE is the initial VC. The goal of the segment is the last statement in the protocol [YAS96].

Weakest precondition is used to find the preconditions to guarantee the correctness of protocol being analyzed with CPAL-ES. This precondition is called Verification Condition (VC). It is returned by CPAL-ES evaluation of protocols and plays

an important role during the evaluation process. The three steps of CPAL-ES evaluation process are:

1- Encoding the protocols in CPAL,

2- Translating specifications into VC,

3- Proving the VC.

A TRUE result of VC guarantees protocol success while FALSE result of VC means protocol failure. CPAL is automated except, the first step [YAS99].

CPAL-ES uses a language, called the Cryptographic Protocol Analysis Language (CPAL), to express the protocol actions. Very complex protocols could be specified in CPAL and it allows automated analysis of coded protocols with formal methods. CPAL is based on the de facto standard notation (SN) which is used for cryptographic protocol specification [YAS96], [YAS99]. SN represents all the pseudo codes used to describe cryptographic protocols evaluated in CPAL-ES. The most important actions in CPAL-ES are *send* and *receive* operations. The *send* operation is represented with the symbol -> in SN while there is no symbol for the receiving since the matching *receive* operation is assumed within the *send* operation. On the other hand, the *receive* operation is as important as the *send* operation in CPAL and the symbol <- represents the *receive* operation in CPAL. There are two *send* operations: Secure send is represented with the symbol => and insecure send represented as ->. Secure send is used for the reliable transmissions through a secure channel. In case the transmission goes through an intruder, insecure send is used. In the following example, sending an encrypted message (msg) under key *k* through a secure channel from principle *A* to *B* is shown both in SN and CPAL:

Table 2.1.2 An example protocol flow in SN and CPAL notation

SN
A: -> B {msg}k

CPAL (secure)
A: => B (e[msg]k];
B: <- (msg');
B: msg : = d[msg']k;

CPAL (insecure)
A: -> B (e[msg]k];
I: <- (msg');
I:-> B(msg');
B: msg : = d[msg']k;

Only the send and encrypt operations are explicit while receive, decryption and name binding operations are implicit in SN. All operations (send, receive, encryption, decryption, and name binding) are explicit in CPAL [YAS96].

With SN, a global address space is used to store the data. Therefore it is often not easy to find the origin of data in a protocol. Conversely, specific address spaces are used in CPAL. Address spaces are identified with a dot notation. For example, *A.kab* denotes the key *kab* is in *A*'s address space. CPAL uses a queue structure to store the data during the send and receive operations. As we can see from the table above, when the sender, *A*, sends the message to the receiver, *B*, the message from *A*'s address space is placed in *B*'s queue. This means the message is available for *B* to receive. Then the message is extracted from the front of *B*'s queue to the *B*'s address space with the receive statement. In the case of an intruder, CPAL uses an insecure send which does not assign the message to the receiver *B*'s queue instead assigns it to the intruder's queue. Therefore, there are at least four steps to send the message from principal *A*'s address space to principal *B*'s address space: First the message is sent with a send operation from originator principal *A*'s address space to the intruder *I*'s queue. Then intruder places the message into its address space by executing a receive operation. In the next step, *I* may replace the message from its address space to the receiver *B*'s queue by executing an insure send operation. Lastly, the receiver *B* can receive the message form *B*'s queue to the *B*'s address space by executing a receive operation.

Protocol specification on CPAL is a sequence of actions where an action may contain more than one CPAL statement. The statement separator, semicolon, is used

between two statements. If there is more than one statement, curly braces ({}) is used. Concatenated values are enclosed in angle brackets (<>) and each value separated by a comma. Encrypted values are inserted in square brackets ([]) prefixed with an "*e*" and suffixed by the key. Decryption is presented the same way except that is prefixed by "*d*". Another feature used with CPAL-ES is functions. Functions are mostly used for security reasons with the cryptographic protocols such as HASH, XOR, and MD functions. We can summarize all these with the following example:

```
A: Y := MD(RN1, RN2)
A:  X := <Y, e[a1,a2]kab>;
A: -> B (X);
I: <- (msg);
I: = > B (msg);
B: <-(msg);
B: (Y, CTfor_a1a2) := msg
B: X := d[CTfor_a1a2]kab;
```

In this example, first, the principal *A* composes the MD function on random numbers *RN1* and *RN2*, and assigns it to the value *Y*. Next, *A* composes the identifier *X* by assigning it for the concatenated values, *Y* and encrypted values *a1* and *a2* under the shared key *kab*, and then sends it to principal *B* through an insecure channel. An intruder *I,* who is a passive intruder listening the traffic, intercepts the message (msg) and forwards it to principal *B*. Upon receipt of the message (msg), first *B* separates the concatenated values by a reverse operation and decrypts the encrypted value, *CTfor_a1a2*, under the key *kab*. Then assigns the result of decryption to an identifier X and stores it in its address space.

CPAL-ES uses ASSUME and ASSERT statements as predicates. ASSUME statement is used to specify assumptions to establish the goals for the truth. In other

words, it gives the truths which could be removed from the predicate. It enables the analyzer to modify the predicate being defined through logical analysis and replaces the truths needed to be removed with the boolean value TRUE. ASSERT statement is used to specify protocol goals needs to be proven with the verification techniques. It adds conjunctive condition to the predicate and states the postconditions in terms of weakest precondition. With ASSERT statements, intended goals are inserted in the appropriate places in the protocol. The goal of the protocol stated in an ASSERT statement should be the last statement in a protocol because any statement other than ASSERT does not have any affect on the initial predicate.

Another statement that is identical to the ASSERT statement is GASSERT (global assert) statement. With the usage of GASSERT, each identifier includes the acting ID suffix and the values are compared across the address spaces. Basically, GASSERT allows comparison of data element within the different principal's address spaces while ASSERT is used to compare data elements within a single principal's address space [YAS96].

Assumptions, assertions and all the actions are used by CPAL-ES form the preconditions. If the given protocol executes correctly, then the final VC takes the logical predicate and simplifies to TRUE as seen on the following example [YAS96]:

```
X: assume (A.kab == B.kab);
A: Y := MD(a1);
A: X := <Y, e[a1]kab>;
A: => B (X);
B: <- (msg);
B: (Y,CTfor_a1) := msg;
B: X := d[CTfor_a1]kab;
B: Z := MD(X);
B: assert(Y == Z);
```

B: gassert (A.a1 == B.X);

*** End of Protocol ***

TRUE

****** Simplified predicate follows.

TRUE

In this example, it is assumed that the principal *A* and *B* share the same key, *kab*. The ASSERT and GASSERT statements are used to compare the values sent from the principal *A* to *B* to see whether there is an attacker who changed one of these values. With the ASSUME statement at the beginning of the protocol, the final VC simplifies to TRUE.

The syntax of the ASSUME statement used in a public key encryption is different since the decrypting a value with the same encryption key does not work with the public key encryption. In order to invert the public key encryption, decryption function should use a key that is inverse of encryption key which decrypts the encryption (public and private keys are inverse of each other). The following predicate example shows the syntax of assumption with public key encryption:

X: assume(global.decrypt(k1,k2));

With this assumption, it is assumed that the relation between the keys *k1* and *k2* is "public key decryption under *k2* inverts encryption under *k1*" [YAS99]. The order is important. This assumption reduces the following predicate to TRUE:

A: (dp[ep[A.X]k1]k2) == A.X;

As seen on the predicate above, "*ep*" represents the public key encryption and "*dp*" represents the public key decryption.

ASSUME and ASSERT statements don't have any action in the protocol run and on any address space but allows the principals to encode assumptions and goals for the protocol directly into the specifications increases SN's functionality. Further details of syntax and semantics of CPAL-ES can be found in [YAS96], [YAS99].

CHAPTER 3

PROBLEM DESCRIPTION

3.1 The Secure Protocol

3.1.1 Protocol Description and Design

3.1.1.1 Definition

The secure communication protocol is designed by Aziz & Diffie to prevent unauthorized access to the wireless communication systems. It provides both the privacy for wireless data communication and the authenticity for communicating principals [AZI94].

The method used to determine the Secure Protocol by Aziz & Diffie is BAN Logic developed by Burrows, Abadi and Needham [BUR90]. It is based on the use of modal logics of knowledge and belief that similar to the one that has been developed for the evolution of knowledge and belief analysis in distributed systems. In such a system, there will be a logic which contains many statements about belief and knowledge of messages and inference rules in order to derive believes from believes or knowledge that comes from other believes or knowledge or visa versa [SYV90]. BAN logic consists of statements regarding to the messages sent and received between the stations of the protocol. The detailed information about BAN Logic is discussed in chapter 2.

3.1.1.2 Design

In order to achieve privacy and authentication for a secure wireless link, public and shared key cryptographic techniques are used. To achieve privacy, shared-key cryptography is used and public key is used for authentication and session key setup. A public and private key pair is generated by each participant in the protocol where the private key is kept by the owner participant of the key pair securely and the public key is submitted to a trusted third party called certification authority (CA) over an authenticated channel. Then the CA will issue a certificate to the participant who is acting on behalf of the machine whose public key is being certified. CA creates the certificate for each participant by digitally signing the public key and the machine name pairs of each machine with CA's private key. The base and the mobile stations will be able to communicate in a secure protocol by having a secure backup of the private keys and obtaining a certificate for each machine [AZI94].

Table 3.1.1 shows the list of acronyms used to define the protocol:

Table 3.1.1 List of acronyms for the Secure Protocol

Pub_Mobile	Public key of mobile host
Pub_Base	Public key of base
Priv_Mobile	Private key of mobile
Priv_Base	Private key of base
Pub_CA	Public key of certification authority
Priv_CA	Private key of CA
Cert_Mobile	Certificate of Mobile
Cert_Base	Certificate of Base station
MD(X)	Message Digest function value on contents X
E(X,Y)	Encryption of Y under key X
Sig(X,Y)	*E(X,MD(Y))*
Sig(X,Y)	Signature of Y with key X
Signed(X,Y)	the resulting signed message *{Y, Sig(X,Y)}*

This protocol is a challenge-response protocol engagement where two parties negotiates shared key algorithm by exchanging certificates. It also provides the good forward secrecy which means that in order to be compromised the communication between the mobile station and the base station, both the mobile station and base station's private keys need to be compromised. Having the private components of the public-private key pair of either the mobile station or the base station does not necessarily mean that compromising the wireless link data which has been exchanged by the machine whose private key is compromised [AZI94].

The entire protocol is summarized in Figure 3.1.1 as defined in [AZI94].

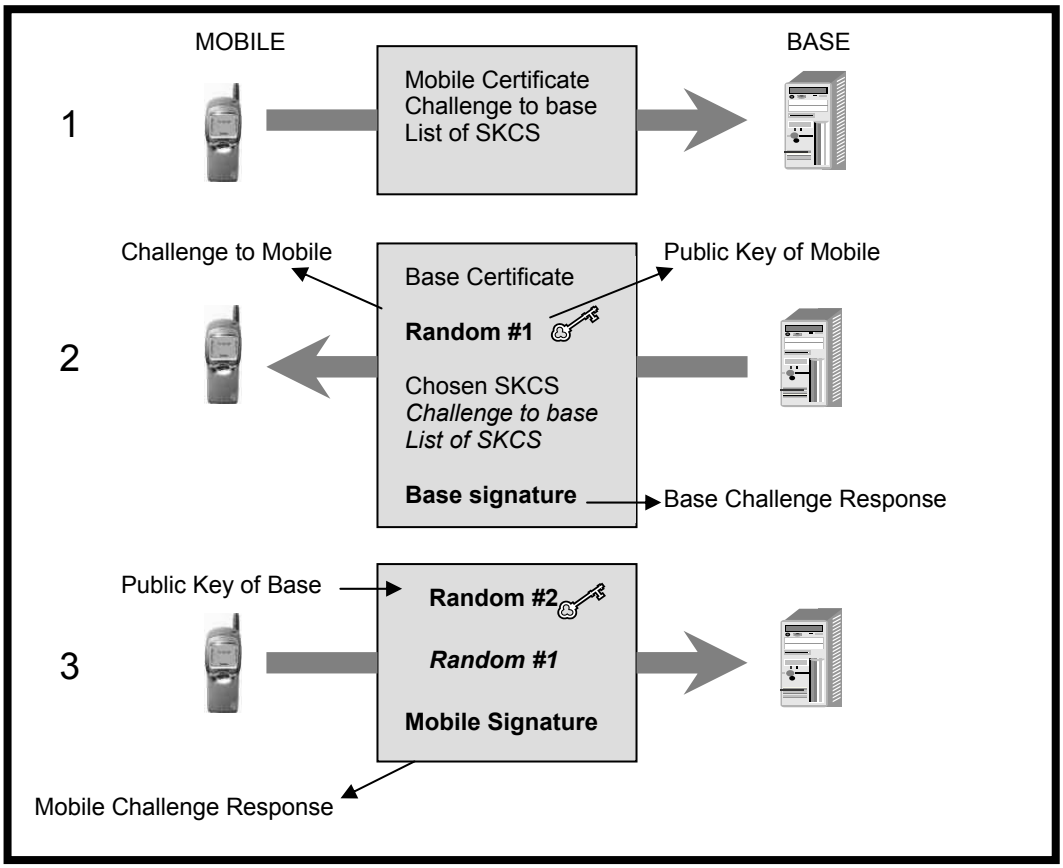

Figure 3.1.1 Entire Secure Protocol for wireless networks [AZI94].

As seen on the Figure 3.1.1, encrypted parts are shown in bold. They are encrypted under either public key for protecting the session key components or private key for digital signature purposes. The parts shown in italic in the figure are not the part of the message itself, but part of the signature block. The signature in message #2 has very important role in the protocol. One of its purposes is to authenticate message #2 in order to protect it against attacks. Another purpose is to authenticate message #1 along with the *List of SKSCs* to protect the list from interposing of other lists of attackers. Finally, it serves as a challenge response to the message #1. In this case the use of public key cryptosystem will be minimized and the protocol will be able to run on a limited computational resource. On the other hand it is a strong security guaranteed system.

Protocol efficiency is an important parameter for protocol analysis. The private key operations are usually the computationally expensive portions of the public key crypto systems such as RSA. Therefore we can count the total number of key operations in the protocol to rank the protocol efficiency. These systems usually pick the keys as to minimize the signature verification and public key encryption processes. The Secure Protocol minimizes the use of public key cryptosystem hence it can be run on the systems that have limited computational resources with strong security [AZI94].

In the Secure Protocol, there are a number of private key operations. Both the base and mobile stations have two different private key operations. Mobile station's first private key operation is decryption of *RN1* and the second is signing message #3. Base station's first operation is signing message #2 and the second is decrypting *RN2* comes with message #3. Hence the total numbers of computationally expensive private key operations of the Secure Protocol are four.

There is no achievement of perfect forward secrecy. Using the combination of Diffie-Hellman key agreement protocol with digital signatures in order to achieve the authenticity, the perfect forward secrecy could be obtained [DIF92]. In this case there would be three computationally expensive operations for each side giving a total of six computationally expensive operations (that is two more operations than what the Secure Protocol does with good forward secrecy). Hence, having good forward secrecy in this protocol is less expensive than having perfect secrecy.

Verifying the authenticity of each side is essential. After this phase, both mobile station and base station may apply a set of access rules before entering the data transfer phase. Access control lists (ACLs), which determine all authorized parties, furnish the access control. Authorization of the parties that are somehow involved in the protocol is important in order to be able to provide authenticity and privacy guarantees which are the two main purpose of the protocol. Another important issue of this protocol is the key exchange. In order to achieve the key exchange, there will be two different keys (one for each direction) used for data transfer. In this case keystream reuse will be prevented.

3.1.1.3 Description

At the beginning of the communication, the mobile station that is trying to connect to the wired network sends the base station it's host certificate (*Cert_Mobile*), *the chosen challenge value (CH1*), and the *list of supported shared-key algorithms* which allow negotiation of the shared-key algorithms with the base station.

CH1 is a randomly chosen 128-bit value. After negotiation of a *shared-key algorithm* chosen from the *list of shared key algorithms* that includes both the algorithm identifier and the key size is sent by the mobile station, the *chosen shared-key algorithm* is used for encrypting the consequent data packets. The host certificate contains a packet of information which consists of the attributes defined in the following order [AZI94]:

{Serial Number, Validity Period, Machine Name, Machine Public Key, Certificate Authority Name}

Here the *serial number* is a unique identifier for each certificate generated by a CA. *Validity period* field is used to indicate the start and end of the time period that the certificate intended to be used. A certificate carries a pair of time and date indications.

The certificate format is the same as described in CCITT X.509 [CCITT88] and PEM (Privacy Enhanced Mail) [KEN93, KAL93]. The certificates for the base station and the mobile station who was trying to connect to the network are generated by CA (the Certification Authority). CA is defined as “an authority trusted by one or more users to create and assign certificates” in CCITT X.509. As stated in PEM, the private component of CA affords a high level of security, otherwise the certificates signed by CA is voided. CA creates the certificate by signing the certificate contents with CA’s private key as follows [AZI94]:

Certificate = Signed (Private-key of CA, Certificate Contents)

= {Certificate Contents, Sig(Private-key of CA, Certificate Contents)}

Where,

Sig(Private-key of CA, Certificate Contents) = E(Private-key of CA, MD(Certificate Contents))

The Message Digest (MD) function is a noninvertible hash function computed on certificate contents. In order to authenticate the certificate contents, a trusted third party *CA* digitally signs *MD* by encrypting it with *CA’s* private key.

3.1.1.4 Message Format

Message 1: Mobile → Base (Request-to-join message)

In the first message there are three parameters. The first parameter is the certificate for mobile station which was created by the third party *CA*. Second parameter is *CH1* which is a randomly generated 128 bit challenge number. The third parameter, the *list of shared-key algorithms*, allows negotiation of the shared-key algorithms with the base station and includes both key size and algorithm identifier.

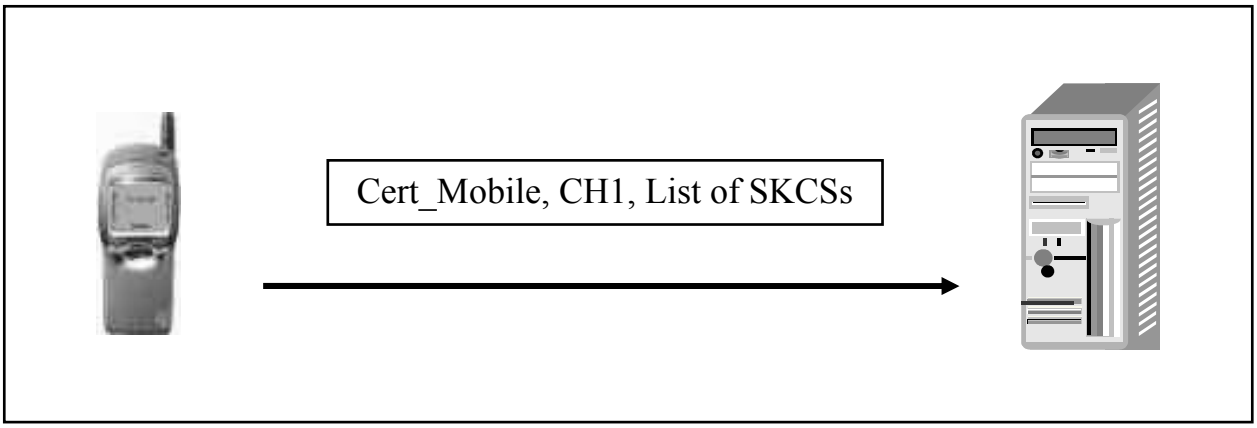

Figure 3.1.2 Message-1: Mobile-to-Base (Request-to-join message)

This first message is not signed in order to be able to enclose the list of shared key algorithms *SKCSs* for the signature verification by the mobile station upon receipt of the second message.

When base station receives this first message, the first step it will do is to verify the signature on *Cert_Mobile*. If it verifies that the public key in the certificate is the certified mobile host's public key, then it will make a decision as it is a valid signature otherwise it is an invalid signature. After the valid signature verification, the base station does not know whether certificate belongs to the mobile station that it submitted. It only knows that the signature belongs to a certified mobile host.

Upon receipt of the first message and signature verification, the base station rejects the connection if the signature is invalid. If it is a valid signature, then base station will create a reply message for the mobile station. In the reply message, the base station will include its certificate which is generated by third party *CA*, *RN1* which is a random number and encrypted with the public key of mobile station and the *chosen SKCS* which is a shared key cryptosystem that is chosen by the base station among the list sent from mobile station. *RN1* is saved internally by the base station for later use. Base station will chose one *SKCS* which seems to be the most secure from the intersection of the two sets of shared-key algorithms that one comes from the mobile station and another one the base station supports. They also agree on the key size by

negotiating downwards to the minimum of the mobile station's suggestion and the value that base station can support for the algorithm it chooses. In order to add the challenge value *CH1* and *the list of SKCSs* to the message, base station computes the message signature by signing encrypted *RN1*, *chosen SKCS*, *CH1* and *list of SKCSs* under the private key of base station.

Message 2: Base → Mobile

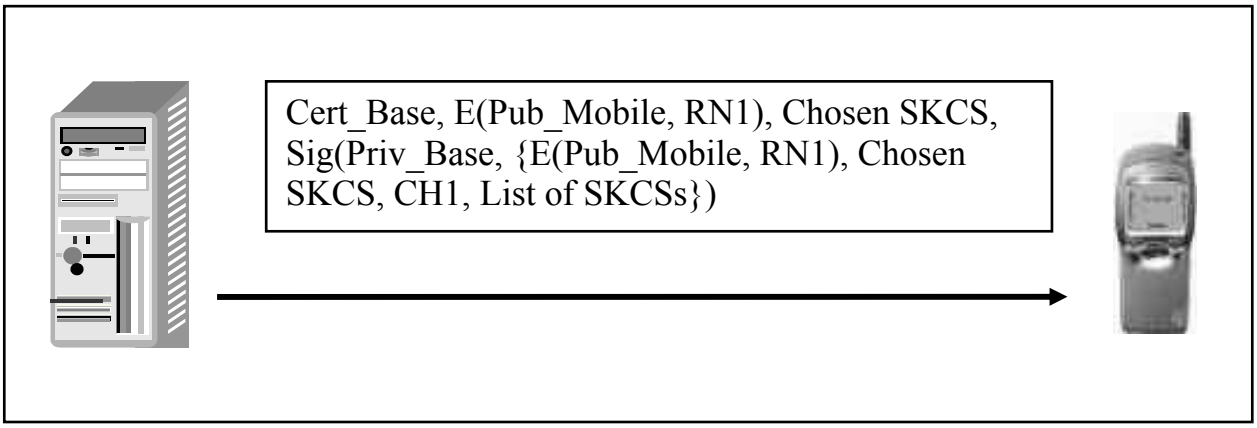

Figure 3.1.3 Message-2: Base-to-Mobile

The second message is sent from base station to mobile station as a reply to mobile station's request. This message consists of four parameters. The first parameter is *Cert_Base*, the certificate for base station generated by a trusted third party CA. The second parameter is the encrypted *RN1* (random number) value which is encrypted under mobile station's public key. The next parameter is the *chosen SKCS* which was chosen by base station among the list of algorithms. It identifies both the associated key size and the chosen algorithm. Finally, the signature signed by base station under

private key of base station is appended to them. The signature contains the encrypted random number *RN1*, the chosen shared key algorithm *SKCS*, challenge value *CH1* and the list of shared key algorithms *SKCSs* sent from mobile station to base station in the first message.

When mobile station receives this message, first it will check for the validation of the *Cert_Base* by verifying the signature on *Cert_Base*. If it is a valid signature, which means that the public key in the certificate belongs to a certified host, then the certificate will be considered valid. In this case mobile station will verify the certificate on the message under *Pub_Base* (public key of the base station).

The signature appended to the message is different than the normal message signing because it includes a component which is not part of the message body but inherent in the protocol. The mobile station verifies the signature under the public key of base station (*Pub_Base*) by taking the message comes from base station and appending to this message the challenge value *CH1* and the list of shared key algorithms *SKCSs* which was originally sent from mobile station to base station in the first message. Because the list of shared key algorithms are not signed, a listening attacker can eavesdrop and weaken the list of shared key algorithms by blocking the original message sent from mobile station to base station and interposing its own list of shared key algorithms. Mobile station can detect this kind of attack after it receives the second message and do the signature verification. If the signature matches, mobile station will consider that base station is a valid host. If it doesn't match, than the mobile station will suspect that the original message is tempered with and will reject the connection because the base station is an intruder and not a valid host.

Upon receipt of the second message, mobile station will decrypt the encrypted part under mobile station's private key (*Priv_Mobile*) and acquire the *RN1* value which is going to be used in order to calculate the session key. Another random number *RN2* will be generated by mobile station to compute the session key as ($RN1 \oplus RN2$). After generating *RN2*, mobile station will encrypt *RN2* under public key of base station (*Pub_Base*). Then it will send this encrypted value to base station in the third message along with the encrypted first random number *RN1* which came in the second message and obtained by mobile station. After the encryption of random numbers, mobile station

will compute the signature by authenticating the encrypted random numbers under private key of mobile (*Priv_Mobile*) and send this in the third message to base station.

Message 3: Mobile → Base

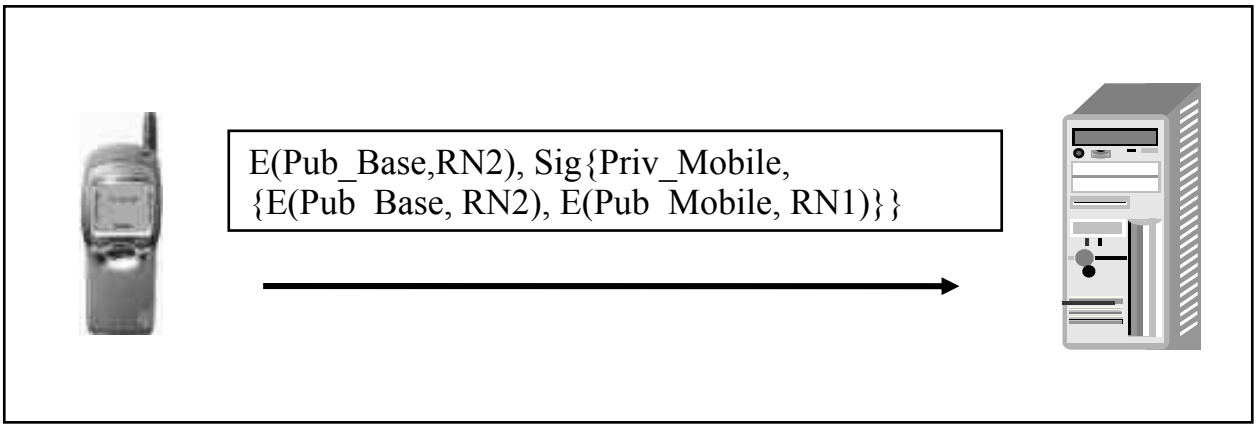

Figure 3.1.4 Message-3: Mobile-to-Base

The third message is sent from mobile station to base station as a response to second message. When base station receives this message, it can obtain second random number *RN2* by decrypting the encryption under private key of base station (*Priv_Base*). Then it can compute the session key as ($RN1 \oplus RN2$).

It is possible that the private keys of one of mobile station can be compromised. In this case an intruder can decrypt the encryption and get the random number. To limit the danger of this kind of attack, two random numbers are used. In this case the attacker has to get both base station's and mobile station's private keys in order to get both of the random numbers *RN1* and *RN2* and session key so that it can precede the communication traffic between mobile station and base station. Therefore getting only

one of the random numbers either *RN1* or *RN2*, attacker gets nothing since both a part of the session key (*RN1* $\oplus$ *RN2*) are absolutely random. On the other hand, if an attacker gets RN1 and the session key (*RN1* $\oplus$ *RN2*) it can generate RN2 or if it can get RN2 and the session key (*RN1* $\oplus$ *RN2*) it can generate RN1.

After obtaining the random number, base station will do the signature verification. Because base station received mobile station's certificate *Cert_Mobile* in the first message and *Cert_Mobile* contains the public key of Mobile host *Pub_Mobile*, it can verify the signature under *Pub_Mobile*. If base station can verify the signature, then mobile station is deemed valid, otherwise it is not a valid host and there may be an attack. In this case, base station will abort the connection.

Valid signature verification and successful acquirement of random number mean the connection occurs successfully. In this case mutual authentication occurs and session key is computed.

3.1.2 Attacks

The Secure Protocol of Aziz & Diffie is vulnerable to attacks. It was critically examined by Meadows and Boyd & Mathuria They identified possible attacks in their papers and suggested the necessary modifications in order to prevent the attacks [MEA95 & BOY98]. Two possible attacks, one proposed by Meadows in 1995 and another one by Boyd and Mathuria in 1998, are discussed in great detail below.

3.1.2.1 Meadows Attack

Meadows attack has been proposed by Catherine Meadows who is one of the important people in the area of cryptographic protocol verification with formal methods. In her publication [MEA95], she gave a survey of formal methods application to analysis of cryptographic protocols. Aziz & Diffie protocol is one of the examples she used in order to explain the different approaches to the applications of formal methods to cryptographic protocol analysis.

The list of acronyms used to define the protocols is listed in the Table 3.1.2:

Table 3.1.2 List of acronyms for the Secure Protocol and attacks

M	Mobile
B	Base
C	Impersonated station
Ix	Intruder impersonating X
KpubX	Public key of host X
KprvX	Private key of host X
CertX	Certificate of X
CHX	Challenge value of host X
RNX	Random Number generated by host X
MD(X)	Message Digest function value on contents X
E(X,Y)	Encryption of Y under key X

In Aziz & Diffie protocol, certificates are created by a *CA* as described earlier in this chapter. More detailed information about how certificates are created will be discussed with the CPAL evaluation of the protocol.

Random numbers *RNB* and *RNM* are used to create the session key. Challenge value, *CHM* is a nonce (randomly generated number) that is generated to guarantee freshness. *SKCSs* are the shared key cryptosystems where base station chooses from the list offered by mobile host. The encryptions in the protocol are the messages signed under either private or public keys and decipherable by the message recipient.

The original Secure Protocol of Aziz & Diffie is as follows:

1. **M → B** (CertM, CHM, List_of_SKCSs)

2. **B → M** (CertB, E(KpubM, RNB), chosen_SKCS, E(KprvB, MD(E(KpubM, RNB), chosen_SKCS, CHM, List_of_SKCSs))

3. **M → B (**E(KpubB, RNM), E(KprvM, MD(E(KpubB, RNM), E(KpubM, RNB))))

Figure 3.1.5 Original Secure Protocol

The first step of the protocol is sending the message-1 from *M* to *B*. When *B* receives the message he will check the validity of certificate. A valid signature indicates that the public key in the certificate belongs to a certified host. He will reject the message if it is invalid or reply with his message which includes his certificate, the encrypted random number *RNB* under mobile host's public key, and *SKCS* that base station chooses out of the list comes from mobile host. This reply message is mapped to an assertion where *B* says that *RNB* is a good key for the communication between *B* and *M*. Upon receipt of the second message, *M* decrypts the encrypted part of the message to verify that it is a meaningful message. It can be verified because it is a meaningful message which means in a recognizable format where *M* can decrypt it with her private key since it is encrypted with *M*'s public key. From this result M concludes as the message was indented for her. As a result, the message means that *RNB* is a good key for the communication between *B* and *M*. Another verification in the second message is the signature on the encrypted message in order to prove that *B* sent the message. In this case *M* concludes as *B* said that *RNB* is a good key for the communication between *M* and *B*. When *B* receives the third message, he gets the result that the message was intended for him because the message encrypted with *B*'s

key. The message says that *RNM* is a good key for the communication between *M* and *B*. After *B* verifies the signed part of the message, he knows that *M* sent the message and he concludes that it was *M* who said that *RNM* was a good key for the communication between *B* and herself.

It is important that the message sent between the participants has to be in such a format that they can understand it and also in case of an attack they can detect it easily. This is why it is an assumption that encrypted message is formatted. According to Meadow's paper [MEA95], the communication fails if the assumption is violated. In this case, there is an attack which is called Meadow's attack. This attack is as follows:

1. **M → B** (CertM, CHM, List_of_SKCSs) (this is intercepted by I)
2. **I_C → B** (CertC, CHM, List_of_SKCSs)
3. **B → C** (CertB, E(KpubC, RNB), chosen_SKCS, E(KprvB, MD(E(KpubC, RNB), chosen_SKCS, CHM, List_of_SKCSs))
4. **I_B → M** (CertB, E(KpubC, RNB), chosen_SKCS, E(KprvB, MD(E(KpubC, RNB), chosen_SKCS, CHM, List_of_SKCSs))

Figure 3.1.6 Meadows Attack

The list of acronyms used in Figure 3.1.6 is described in Table 3.1.2.

As seen above, the first step is sending message from *M* to *B* as a request to join. Instead of *B* receiving this message, it is intercepted by *I* which is an intruder that impersonates *C*. In this case *C* is a rogue principal. Intruder *I* changes only the certificate part of the message by replacing its own certificate and sends the message to *B*. Here the rascal principal *C* replays the legitimate mobile host *M*'s challenge, *CHM*, in this run. When *B* receives the message, he does not know that the message is intercepted by an intruder. Therefore, he sends the package in a format that almost the

same as it was in the second step of the original Aziz & Diffie protocol where *RNB* is encrypted under *C*'s public key instead of *M*'s. The reason for this is *B* thinks that he is communicating with *C* not *M* and *B* trusts *C*, he does not know that *C* is an intruder. Then *C* passes off the same package to *M* as it was *B*'s response. Therefore, once *M* receives the forth message, she thinks that it was coming from *B*. In this package the partial session key is intended for *C* but when *M* receives it, she thinks that it was for *M*. Then she verifies the signature and tries to decrypt the encrypted part of the message, *E(KpubC, RNB),* under private key of *M*. She obtains the result as *D(KprvM, E(KpubC, RNB))* and she thinks that this is the key. Since the intruder persuades *M* that the nonkey, *(KprvM, E(KpubC, RNB)),* is a key, we can say that there is an attack on this communication. This spoof is not detected. On the other hand, it is stated that at worst this attack results in a denial of service and the intruder cannot know whether *M* accepts the nonkey as a key. In this case, the intruder cannot impersonate *M* or *B*, read encrypted traffic. Then the BAN logic result of this protocol that *M* believes that the key she receives from *B* is a good key for the communication between them is not an adequate conclusion anymore [MEA95].

3.1.2.2 Boyd & Mathuria Attack

Boyd and Mathuria [BOY98] have proposed an attack on Aziz&Diffie protocol [AZI94]. This is very similar to Meadows attack. In this attack, *B* is spoofed since he computes a false session key to use with *M*. On the other hand, *M* is not actually in a communication with *B*. The attack is shown in Figure 3.1.7 in detail.

As seen on Figure 3.1.7, the first run starts with a message from a rogue principal *C* to *B* which asks for joining to conference. List of keys *List_of_SKCSs*, *C*'s challenge value *CHC* and Certificate of *M*, *CertM*, are included in the first package. Here *C* sends *CertM* instead of C's certificate *CertC* since *C* has *CertM* from *M* in the earlier runs. In this case *B* thinks that he is communicating with *M*. Then *B* replies with his certificate *CertB* and random number *RNB* encrypted under *M*'s public key along with the chosen key and the hash value which is the encryption of *MD* function under *B*'s private key. *MD* function is the encrypted random number *RNB* under *M*'s public key, and *chosen_SKSC*, *M*'s challenge *CHM* and the list of keys. Then *C* passes off the

same message except sends its certificate *CertC* instead of *B*'s and in the signature *RNM* is used instead of *RNC* and the signature is signed with *C*'s private key not *B*'s.

1. **C → B** (CertM, CHC, List_of_SKCSs)
2. **B → C** (CertB, E(KpubM, RNB), chosen_SKCS, (E(KprvB, MD(E(KpubM, RNB), chosen_SKCS, CHC, List_of_SKCSs))))

1'. **M → C** (CertM, CHM, List_of_SKCSs)

2'. **C → M** (CertC, E(KpubM, RNB), chosen_SKCS, (E(KprvC, MD(E(KpubM, RNB), chosen_SKCS, CHM, List_of_SKCSs))))

3'. **M → C** **(**E(KpubC, RNM), E(KprvM, MD(E(KpubC, RNM), E(KpubM, RNB))))

3. **C → B** **(**E(KpubC, RNM), E(KprvM, MD(E(KpubC, RNM), E(KpubM, RNB))))

Figure 3.1.7 Boyd & Mathuria Attack

Finally, in the last message, *M* sends her random number *RNM* encrypted under *C*'s public key and the signature is signed with *M*'s private key. In the signature *RNM* is encrypted under public key of *C* and *RNB* is encrypted under public key of *M*. Here including *E(KpubM, RNB)* is an assurance to *B* that the partial session key sent by *M* is fresh. After *C* receives this package it passes the same package to *B*. When *B* receives the message he will think that this message is coming from *M*. He will verify the signature and try to *decrypt E(KpubC, RNM).* Since *B* does not know *KprvC*, he will not be able to obtain *RNM* instead he will get the result as *D(KprvM,E(KpubC, RNB))* and he will think that this is the key. In this case *B* will compute a false session key and think that he can use this with *M*. On the other hand *M* is not actually communicating with *B* in a protocol run. This is why we can say that *B* may be spoofed in this attack. The rogue principal *C* does not need to know *RNB* in order to create the message 2'. Since *B*

sends *E(KpubM, RNB)* to *C* and *RNB* is always used as encrypted under *M*'s public key in the protocol run, *C* can send the encrypted value without knowing *RNB*. Therefore in message 2, if *B* could sign *RNB* instead of signing *E(KpubM, RNB),* this attack could be prevented. Furthermore, there could be more conventional challenge/response system to assure freshness of the partial session key sent from *M* to *B* in message 3. Below is the modified version of Aziz&Diffie protocol that has the solutions the Boyd&Mathuria attack:

1. **M → B** (CertM, CHM, List_of_SKCSs)
2. **B → M** (CertB, CHB, E(KpubM, RNB), chosen_SKCS, E(KprvB, MD(E(RNB, M, CHM, Chosen_SKCS))))
3. **M → B (**E(KpubB, RNM), E(KprvM, MD(E(RNM, B, CHB))))

Figure 3.1.8 Solution to Boyd & Mathuria attack

As we can see above, the first request to join message sent from mobile host to base station is the same as original Aziz & Diffie protocol. Second message sent from *B* to *M* is also almost the same except a few changes. The first addition to this package is *B*'s challenge, *CHB*, a 128 bit random number that guarantees the freshness by changing for every new message therefore it is unique and unpredictable. This protects the communication against playback attacks. Another change in this package is *RNB* is not encrypted in order to prevent the run against to an attacker that spoofs the communication without the knowledge of *RNB*. The last addition is *M* in the hashed package. Having the receiver's name in the package ensures the receiver that this package is sent for her.

Third message that sent form *M* to *B* is different than the original third message of Aziz &Diffie Protocol. In the third message, encrypted *RNB* is included as in the original one. Here the changes are in the hash function. Instead of signing the encrypted *RNB* and *RNM*, in the last package *RNB*, *B* and *CHB* values are signed. Again *RNB* is not encrypted to prevent to package against to an attacker spoofing. Finally, *CHB* is added to the signature of third message as a respond to *B*'s challenge in order to assure freshness.

3.2 IEEE 802.1X Standard

3.2.1 Protocol Description and Design

3.2.1.1 Definition

IEEE 802.1X standard protocol is designed to enhance wireless local area networks (WLANs) security which follows IEEE 802.11 Standard. Because of the current problems with the security of WLAN based on 802.11 Standard, new security architecture for 802.11 is needed. Robust Security Network (RSN) which is an approved IEEE 802.1X Standard for Port based Network Access Control is a new design to solve problems with 802.11 Standard. 802.1X Standard provides a strong authentication, access control and key management for WLANs. On the other hand, Mishra [MIS02] showed that current combination of IEEE 802.11 and 802.1X Standards does not provide sufficient level of security. Authentication with 802.1X Standard is done by a central authority in an infrastructure network where each client sends packets to a central authority called access point (*AP*), known as the authenticator and *AP* transmits them to the destination client.

3.2.1.2 Design

Since IEEE 802.1X standard came as a solution to IEEE 802.11 standard, first we will talk about the design and security issues of IEEE 802.11 standard. Then, we will

discuss IEEE 802.1X standard protocol and the attacks found by Mishra [MIS02] in detail.

3.2.1.2.1 IEEE 802.11 Network Design and Security Mechanisms

Classic 802.11 Network design is shown in Figure 3.2.1:

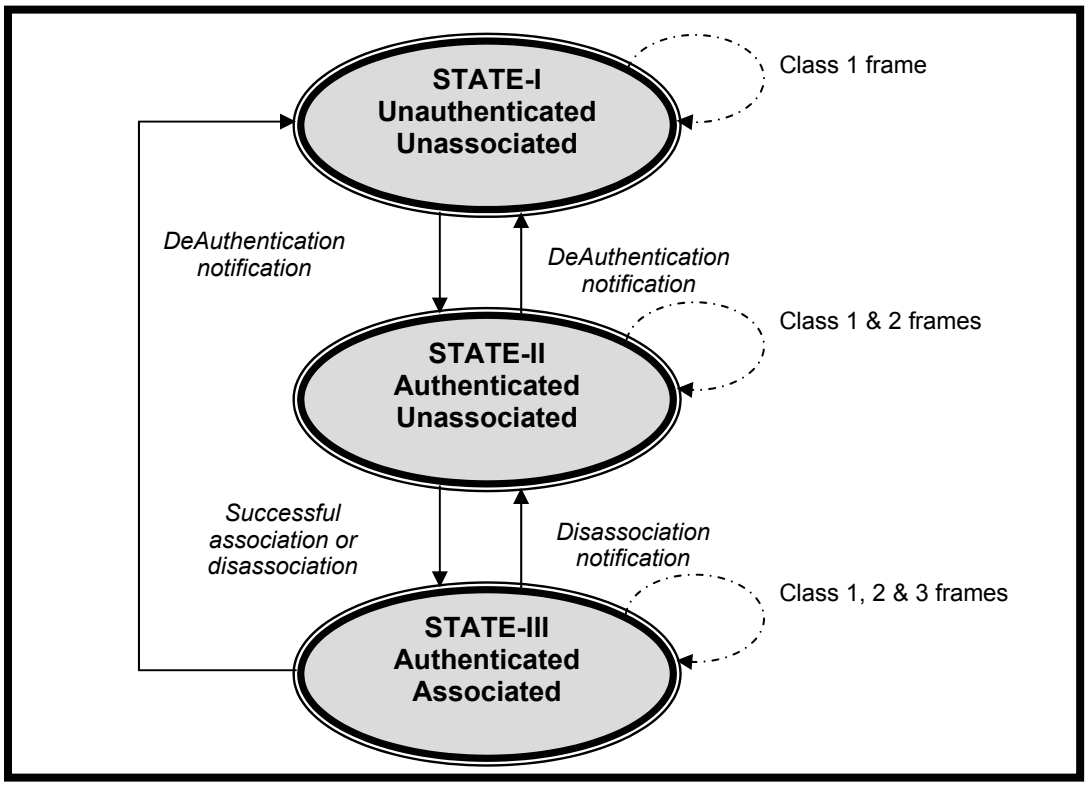

Figure 3.2.1 The Classic 802.11 state machine.

IEEE Standard specifies the Medium Access Control (MAC) mechanism and indicates one of the two modes which are infrastructure (Basic Service Set) mode and ad-hoc, infrastructureless, (Independent Basic Service Set) mode. In the ad-hoc network the clients can directly communicate each other without a trusted third party. On the other hand, there is a central authority, *AP*, which receives the packets from a

client and sends them to the destination clients. In Mishra's work only the security issues of infrastructure mode is discussed [MIS02].

A wireless client starts a relation with a central authority or access point (*AP*) which is called "association". There are three states that show the complete association with an *AP* as stated in [MIS02]. The relationship between these three steps is shown in the Figure 3.2.1.

IEEE 802.11 standard frame can be a management or a data frame. Using Authentication management frames, client or STA (station) and *AP* exchange these frames in order to pass from state 1 to state 2.

Security goals for 802.11 protocol framework are access-control and mutual authentication, flexibility, ubiquitous security, strong confidentiality, and scalability. More information on these can be found in Mishra Et al. [MIS02].

Shared key authentication, open system and MAC-address based access control lists are the primary authentication and address control methods used for IEEE 802.11 standard protocol. In order to provide confidentiality for the network traffic, Wired Equivalent Privacy Protocol (WEP) is used. On the other hand insecurity of this system is shown with the resent work [ARB01, BOR01]. Therefore as solution to these security problems a new technique for WLAN, called Robust Security Network (RSN), which is used with IEEE 802.1X standard is designed by IEEE standard group.

3.2.1.2.2 IEEE 802.1X Standard and the Robust Security Network (RSN) Design and Security issues

Network access authentication is important for the wireless environment since there is no physical boundary to restrict the network access. Such a system for restriction on network connectivity at MAC layer is afforded by RSN. RSN provides this mechanism only via 802.1X standard protocol. A network port which provides the network connectivity is an association between an access point and the station in 802.11 standard protocol. Along with the usage of authentication methods such as certificate based authentication, one-time password, smart cards 802.1X standard

protocol provides an architectural framework and port-based network access control for hybrid network. It is used for wireless 802.11 networks by RSN [MIS02].

RSN security framework is shown below:

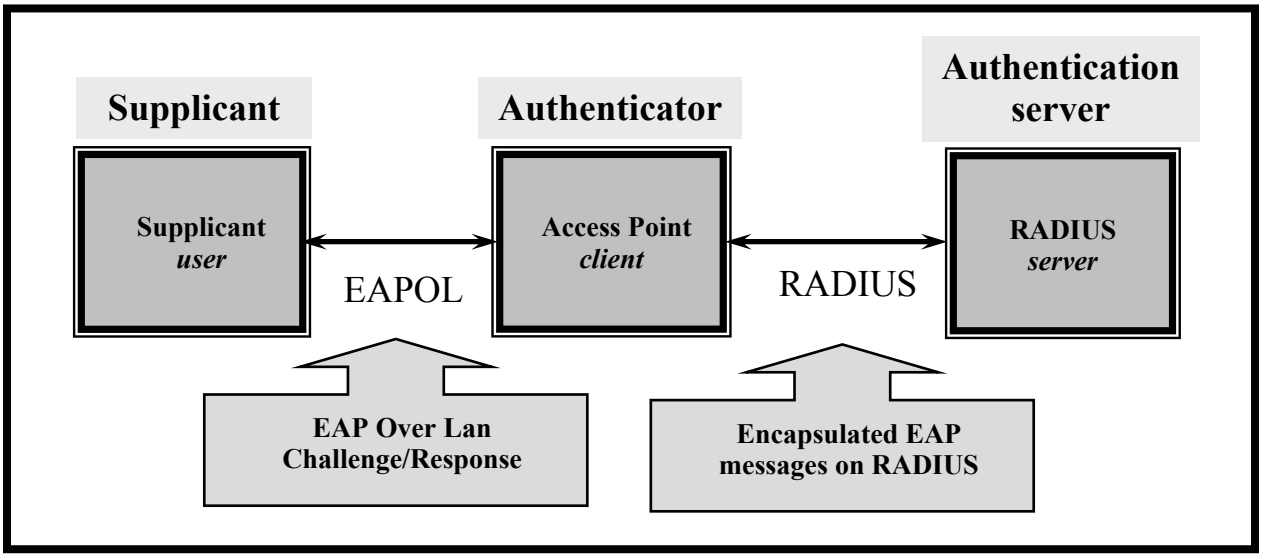

Figure 3.2.2 The EEE 802.1X setup.

As seen in the communication setup above and stated in the IEEE 802.1X standard [IEE01], supplicant, authenticator and authentication server are the three different roles provided by RSN. Supplicant (Ethernet 802.3, wireless PC card, etc) is an entity that uses the service for MAC connectivity offered via a port on the authenticator (Access Point, Ethernet Switch, etc). The network port is an association between the station and the access point as stated earlier. There could be many ports or access points available on a single network for supplicant to authenticate a service via authenticator. Then authenticator connects to a central authentication server (EAP server, RADIUS) which provides a service after successful authentication.

Authentication server, called backend server, provides strong authentication and session keys for supplicants. Therefore, both supplicant and authenticator trust the

integrity of backend server and backend server needs their identity in order to provide the session keys. Because of the inherent broadcast nature of the wireless communications, this authentication process should be protected against to the attacks.

RSN provides the following: Per packet authenticity and integrity framework between AP and RADIUS, scalability and flexibility, access control, and one-way authentication. These properties are very important to protect the system against to attacks and provide strong authentication. For instance, good access control can protect the system against session-hijack attack and lack of mutual authentication can cause Man-In-Middle Attack.

3.2.1.3 Description

Two different mechanisms are used by IEEE standard protocol for the communication between supplicant and authenticator (access point), and authenticator and authentication server (RADIUS server).

The first one is Extensible Authentication Protocol (EAP) [BLU98] used between the supplicant and access point. It is assembled on challenge-response communication system. EAP stack displayed in Figure 3.2.3 shows the communication between EAP layer, authentication layer, and MAC layers. The EAP over LAN (EAPOL) protocol is used to communicate with the higher layer and carry the packages, which have the encapsulated EAP messages, from the authenticator (base station) to the supplicant (mobile host) and vice versa. Therefore, by using the EAPOL messages, EAP messages are exchanged between the supplicant and the authenticator [MIS02].

EAPOL can communicate with the higher layers such as Kerberos and TLS. It has session start and session logoff messages to start and stop the communication. Both EAP and EAPOL messages don't have any sanity check for privacy and integrity. EAP operates on the network layer. Network ports do not need to make their own decisions. In this case, EAP directs messages to the centralized service such as RADIUS (Remote Authentication Dial-In User Service) and this EAP server makes the decisions [MIS02].

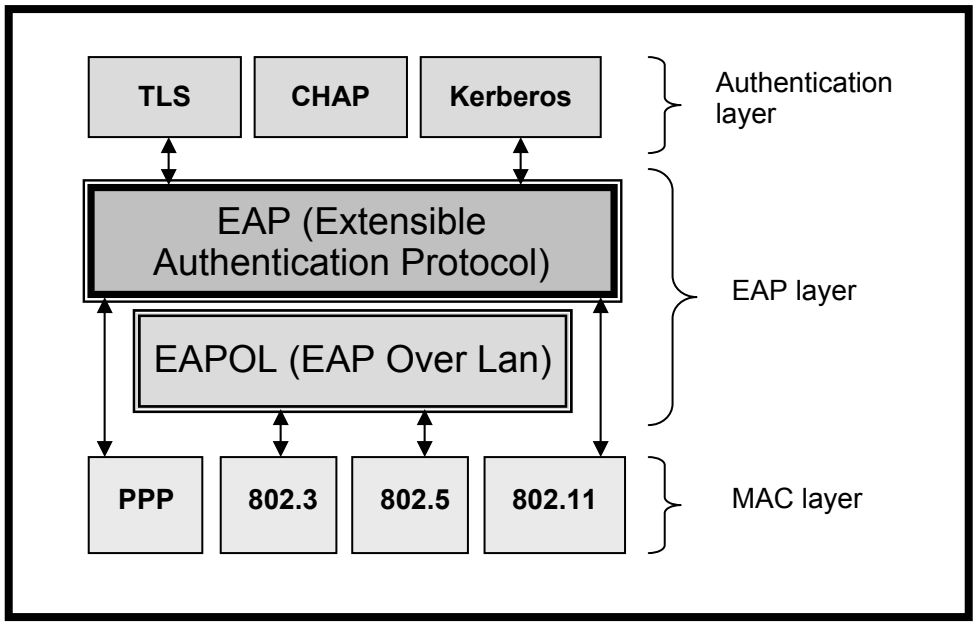

Figure 3.2.3 The EAP stack.

The second message type called Remote Authentication Dial-In User Service (RADIUS) protocol is used for the communication between the access point and the authentication server called RADIUS Server. NAS (Network Access Server) is used as the client of RADIUS server where the client is responsible for carrying the necessary information from user to the RADIUS server and getting the response from server. On the other hand, the RADIUS server is responsible from receiving the user request, authenticating the user and sending all the necessary information to client to pass it to the user [RIG02].

There is integrity check and per packet authenticity between the RADIUS server and access point. Communication between the client and RADIUS server is authenticated under a shared secret. This key is never sent over the network, and user passwords are sent encrypted between these two points in order to stop any unexpected eavesdropping. RADIUS server can use variety of authentication

techniques. If the username and user passwords are provided then it supports PPP, PAP, CHAP, UNIX login and some others [RIG02]. RADIUS uses UDP instead of TCP as a transport protocol for some technical reasons as stated in [RIG02].

EAP message is carried as an attribute in the RADIUS protocol [MIS02].

3.2.1.4 Message Format

3.2.1.4.1 EAP Message Format

A summary of the EAP packet format is shown below where the fields are transmitted from left to right:

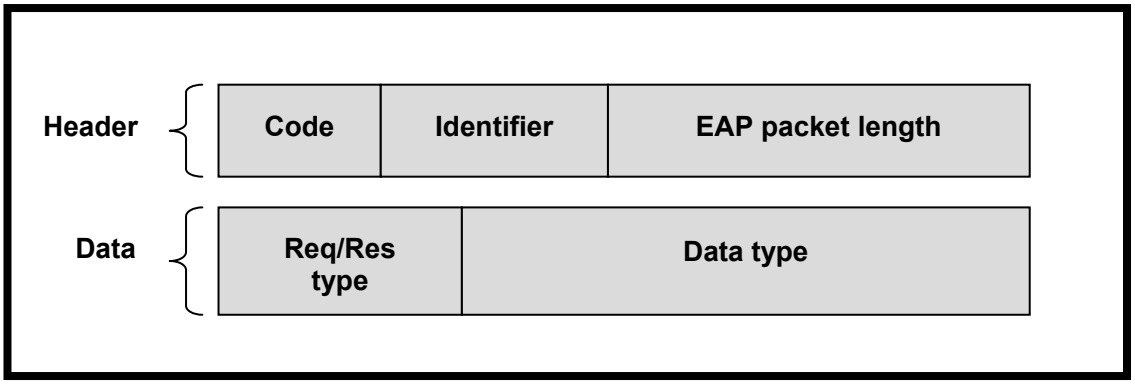

Figure 3.2.4 The EAP Packet.

The first field of the EAP packet is *code* which is one octet and changes from 1 to 4. Therefore there are four types of messages listed as:

1-EAP Request

2-EAP Response

3-EAP Success

4-EAP Failure

Identifier field is also one octet and matches responses with the requests. For a retransmitted request packet due to a timeout, the identifier field will be the same. This field is modified for each new request. For example, for the first request and first response to this request, this field could be 1 and for the second request and response it could be 2. For the situation of a duplicate message, this field again stays the same and duplicate request is discarded.

The last part of the header, which is the third field, is the *length*. This field indicates the length of the EAP package that includes the *Code*, *Identifier*, *Length*, *Req/Res Type* and *Data Type* fields. This is two octet fields and only one type is specified for each request or response.

Request/Response Type field is one octet that indicates the type of the request and response. There should be only one type specified for each EAP request and response. The type field of response and request would be the same except the Nak response Type (only for response) [BLU98]. The Request/response types are assigned as follows:

1-Identity

2-Notification

3-NAK(for response)

4-MD5-Challenge

5- One-Time Password (OTP)

6-Generic token card

More information about this field can be found in [BLU98].

The last field is the *data type*. This field is specified depending on the request and response types we have just discussed. It is either zero or more octets and must not be null terminated. This field varies for each request and response type. For example, for the request/response type 1(identity), response uses this field to return identity and may contain displayable message in the request. More information could be found in L. Blunk Et al. [BLU98].

3.2.1.4.2 RADIUS Message Format

The following figure shows the format of the RADIUS packet used in 801.1X authentication protocol where the fields are transmitted from left to right:

The first field of the RADIUS packet is again the *code* which includes the type of the RADIUS packet. This field is one octet. Packets with the invalid code field are discarded. Following is the list of assigned codes for RADIUS packets that have been used for 802.1X authentication:

- Access-Request
- Access-Accept
- Access-Reject
- Access-Challenge

The second field is the *identifier* that is used for matching the requests to the reply. This field is also one octet. Duplicate requests are detected by the RADIUS server and invalid packets are discarded where the identifier fields of *Access-Request* and *Access-Accept* does not match or *Request Authenticator* does not have the correct response for the *Access Request*.

The *length* is the third field in the RADIUS packet and it is two octets. Octets out of the range of the length (min 20, max 4096) are ignored. The length field shows the

length of the packet which includes the *Code, Identifier, Length, Authenticator, and Attribute* fields.

The next field is *Authenticator* which is sixteen octets. The most important octet is transmitted and the reply from the RADIUS server is authenticated with this value. *Authenticator* field could be either request or respond authenticator.

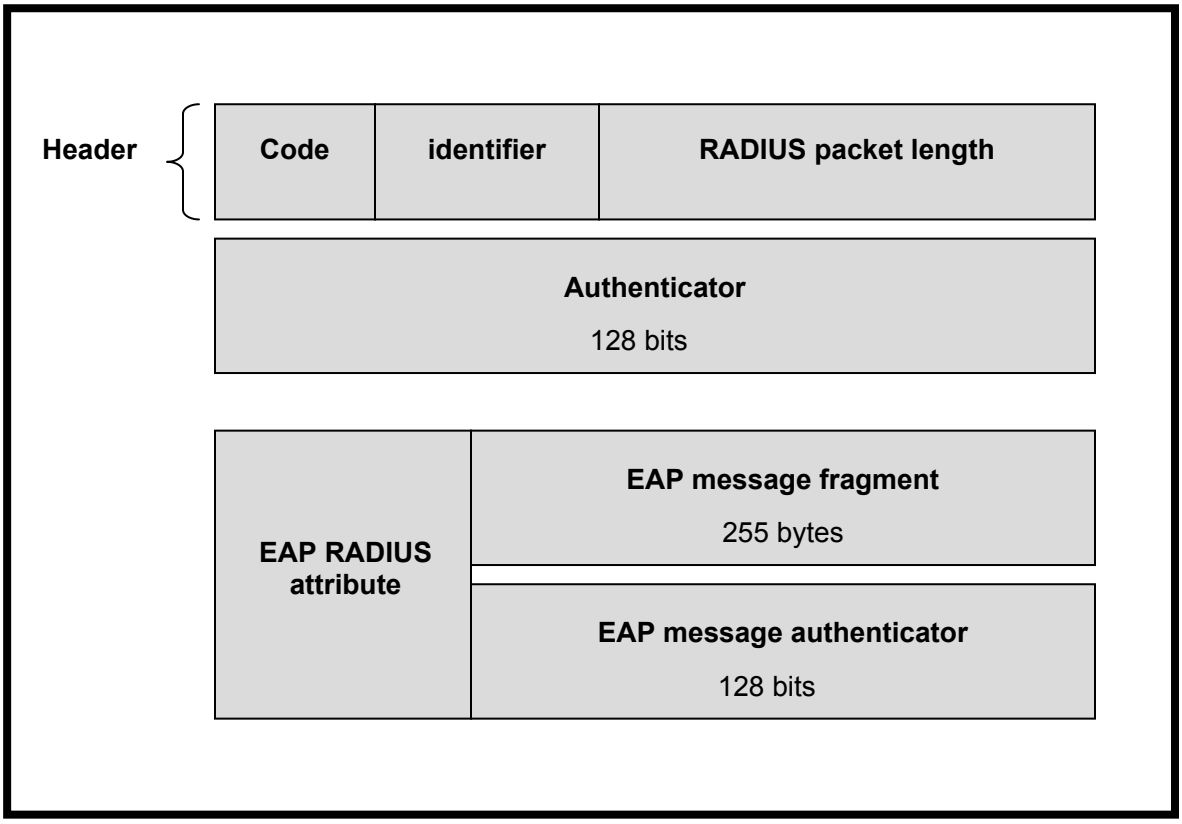

Figure 3.2.5 The RADIUS Packet.

The authenticator in *Access-Request* packets called *Request authenticator* and value of authenticator is 16 octet random numbers. This value should be unique, and unpredictable. It is a large secret password for protecting the mechanism against to attacks and it is shared secret value between the RADIUS server and the client.

The authenticator field in *Access-Challenge, Access-Accept, and Access-Reject* is called *Respond authenticator*. It is a *MD5* hash value that is calculated on radius packets that has the *Code, the Identifier, the Length, the Request Authenticator* fields along with the attributes comes with the response attributes and shared secret.

The last part of the RADIUS packet is *EAP attributes* which comes with the EAP message from the communication between the Access Point and the Supplicant.

So far, we have discussed the general format of the RADIUS package. Now, we will explore the package format in detail for specific packet types which was determined by the Code field at the beginning of the packet.

3.2.1.4.2.1 Message Format for Access-Request Type

Access-Request messages are sent from a user to RADIUS server to provide information about the user whether or not it is allowed to access a specific NAS. The *code* field of a RADIUS packet is set to 1 for *Access-Request*. *Access-Request* contains a username attribute. It contains *NAS-Identifier* and/or *NAS-IP-Address attribute* and *User Password*. It also includes *NAS-port* and/or *NAS-port-type attributes*.

User password is hidden in a MD5 message digest function. The shared secret is shared between NAS and RADIUS server. A one-way 16 octet message digest *MD5* hash function on *Request Authenticator* and the shared secret is xored with the password entered by the user and this value is used as *User-Password* attribute.

The message format for *Access-Accept* type is very similar to a general RADIUS message format as we have seen earlier. The first field is *Code* and it is 1 for *Access Request*. The next field is *Identifier* which changes whenever the attributes changes or a valid reply is received and does not change for retransmission. *Request authenticator* is the next field and changes when a new identifier is used. *Attributes* field has the *username, user-password, client-ID, port_ID, EAP message authenticator* and EAP message which come from the supplicant.

3.2.1.4.2.2 Message Format for Access-Accept Type

RADIUS packets for *Access-Accept* type are sent from RADIUS server to *Access Point* (authenticator). This packet provides the information needed to initiate the delivery to the user. All of the attributes in the message are checked and if they are acceptable then the *Code* filed is set to two for *Message-Accept* and then RADIUS sent the message to the *authenticator*.

The message format is again similar to the format of *Access-accept* type. First field is again *Code* and set to 2 for *Access-Accept*. *Identifier field* is a copy of the identifier field of the *Access-request* packet that is pending for a response from RADIUS. *Response Authenticator* value is a *MD5* hash value that is calculated from *Access-Request fields* of the *Code, the Identifier, the Length, the Request Authenticator* fields along with the *Attributes* that comes with the response attributes and shared secret. The *attributes* field again has the information about the *username, user-password, client and port ID and EAP success message* and *EAP success authentication* message.

3.2.1.4.2.3 Message Format for Access- Reject Type

This is the message sent when the attributes in the packet are not acceptable. In this case, The *Code* field of RADIUS message for *Access-Reject* type is set to 3.

The packet format of the Access-Reject type starts with the *Code* field (set to 3). *Identifier* field is a copy of the *identifier* field of *Access-Request* that reasons the *Access-Reject*. The *Response Authenticator* is calculated from *Access_Request* value. It is a *MD5* hash value which is calculated from *Access-Request* fields of *the Code, the Identifier, the Length, the Request Authenticator* fields along with the *Attributes* comes with the response attributes and shared secret. *Attributes* are again the same as the *attributes* for previous types we have just discussed about.

3.2.1.4.2.4 Message Format for Access- Challenge Type

When The RADIUS server needs to send a challenge message as a response to a request, the *Code* field of challenge message should be set to 11. The *Identifier* field of *Access-Challenge* message should match the *identifier* field of the *Access-Request* message which is pending for a response and also the *Response Authenticator* field is calculated as described earlier and should have the correct response value for the *Access-Request*. If not, then the invalid packets will be discarded. The *Attributes* field may contain a few *Reply* message attributes. Additionally, *Vendor-Specific, Proxy-State, Session-Timeout and Idle Timeout attributes* can be included. Other attributes are not allowed in *Challenge* message [RIG02].

More information on the attributes and the message format for different message types could be found in C.Rigney Et al. [RIG02].

3.2.1.4.2.5 Message Format for IEEE 802.1X Protocol

The complete 802.1X Protocol authentication session has both EAP and RADIUS protocol messages where EAP messages carried as an attribute in RADIUS protocol.

S*upplicant*, *Access Point* and the *RADIUS* are three entities specified in the IEEE 802.1X protocol as we mentioned earlier. The complete 802.1X authentication session which shows the EAP and RADIUS message communication traffic between these three entities is shown on Figure 3.2.6.

The Table 3.2.1 shows the list of acronyms used to define the IEEE 802.1X protocol:

Table 3.2.1 List of acronyms for the IEEE 802.1X protocol.

EAP	Extensible Authentication Protocol
RADIUS	Remote Authentication Dial-In User Service
EAP Req	EAP Request message
EAP Resp	EAP Respond message
EAP Succ	EAP Success message
EAP Failure	EAP Failure message
EAPOL Key	Optional. Communicates higher layers (i.e. TLS)
Id	Supplicant's identity
RAD Acc Req	RAIDUS Access Request message
RAD Acc Chal	RADIUS Access Challenge message
RAD Accept	RADIUS Accept message
RAD Reject	RADIUS Reject message

As seen on Figure 3.2.6, there are two different protocols: The first one is EAP protocol used for the communication between the access point and the supplicant. The other one is the RADIUS protocol used between the access point and the RADIUS. Initial 802.1X communication starts with an unauthenticated client (user, supplicant) requesting to connect to an access point (authenticator). When access point receives this message, it responds with a request that the supplicant change its state into an unauthorized state to enable to send only EAP start message. Then access point returns the EAP Req/Id message requesting the supplicant's identity. When supplicant receives the Id request message, first it changes the code field to 2 for a response and matches the identifier field with the identifier field of the request message. Then supplicant returns the identity with the *EAP Resp/Id* message to access point.

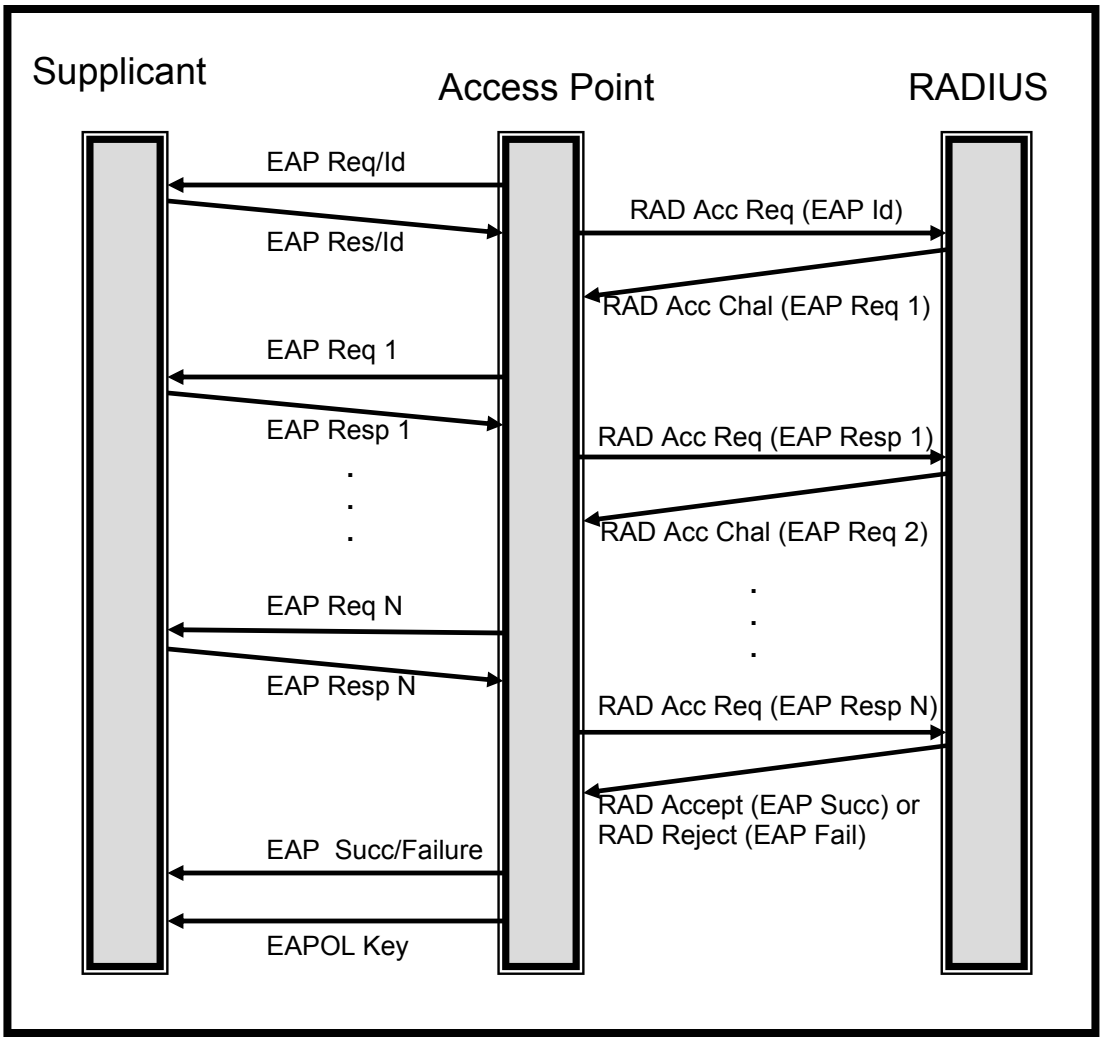

Figure 3.2.6 The complete IEEE 802.1X protocol**.**

Upon receipt of the response message from supplicant, access point forwards the identity for verification by sending the *RAD Acc Req (EAP Id)* message to the RADIUS authentication server which locates on the wired side of the access point. Access point blocks all the traffic such as HTTP, by enabling the port to pass only EAP packets from user to RADIUS authentication server until the identity verification is done by RADIUS. RADIUS responds with *RAD Acc Chal (EAP Req1)* message to the access point. This message is sent when RADIUS, which requires a response, wants to send a

challenge to the user. Upon receipt of this message, first, access point matches the *identifier* field with the *identifier* field of pending *Access-Request* message and then *Response Authenticator* field is checked to see whether it has the correct response for the pending *Access-Request* message. If it is a valid package, then access point sends the *EAP Req-1* message, which is the *attribute* field of message comes from RADIUS, to the supplicant. Then supplicant changes the code to respond as 2, matches the *identifier* and sends the respond to the request of RADIUS with the *EAP Resp 1* message. Access point receives this message and forwards it to the RADIUS as an attribute with the *RAD Acc Req (EAP Resp 1)* message. RADIUS receives the message and sends another challenge message if needed. This is continued until RADIUS is satisfied with the messages it received to make decision whether or not the user is accepted. If all the attribute values received in the *Access Request* messages are acceptable, then RADIUS server accepts the message, provides necessary confirmation information in order to start the delivery service to the client and uses an algorithm to authenticate the user. Then it sends the *Access-Request* message *(RAD Accept (EAP Succ))* to the access point by setting the code field to 2. Otherwise RADIUS server rejects the message, sets the code field to 3 and sends *RAD Reject (EAP Fail)* message to the access point. If Access point receives the *RAD Accept (EAP Succ)* message from RADIUS which means the user is accepted by RADIUS server, it changes the client's state to authorized and opens the client's port to the normal traffic. Then it sends the EAP Succ message to the supplicant. As an option it may send the *EAPOL key*, which is used to communicate to a higher layer (such as TLS), to the supplicant too. On the other hand, if access point receives a reject message, then it sends the *EAP Failure* message to the supplicant and does not authorize the user.

3.2.2 Attacks

3.2.2.1 Man-In-The-Middle (MIM) Attack

The 802.1X protocol provides only one-way authentication. Because 802.1X authenticator sends only *EAP-Request* messages to supplicant, and accepts *EAP-response* messages from supplicant. On the other hand supplicant does not send any

EAP-request message. The supplicant is authenticated to the access point. Because of the one-way authentication of the supplicant to access point where absence of mutual authentication is observed, the supplicant can be exposed to potential Man-In-The-Middle (MIM) attacks.

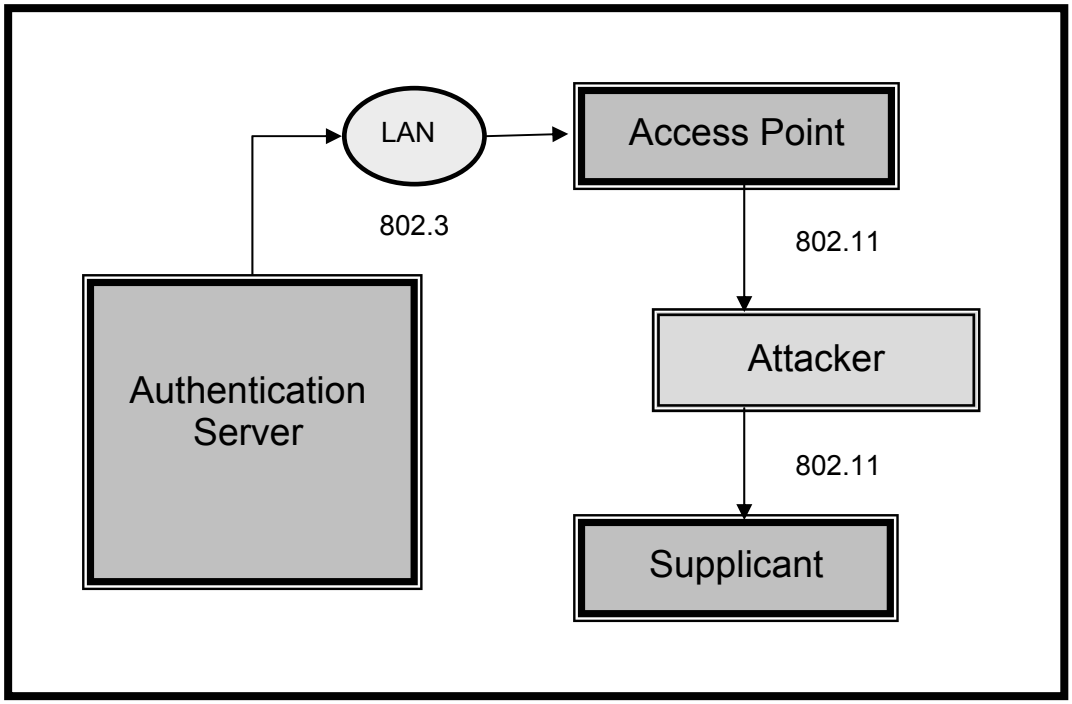

Figure 3.2.7 Man-In-The-Middle (MIM) Attack.

As seen in the figure above, the attacker acts as an access point to the supplicant and client to the access point. After receiving the *RADIUS access-Accept* message from RADIUS authentication server, authenticator sends *EAP-Success* message to the supplicant. This message does not have any integrity even though an authentication technique (like EAP-MD5) used in the higher layer. Supplicant state machine contains an unconditional transfer by setting *EAP success* message to the *eapSuccess* flag that transfers directly to *Authenticated* state. An adversary can forge this package on behalf

of the authenticator and can start MIM attack. In this case, the attacker can get all the network traffic from supplicant [MIS02].

3.2.2.2 Session Hijacking Attack

RSN (Robust Security Network) state machine is similar to 802.11 classic state machines we have talked about earlier except the addition of state-4 called *RSN Associated*. After the *RSN association/reassociation*, higher layer authentication occurs with IEEE 802.1X. Therefore RSN and 802.1X state machines work together. Because of the communication problems between the 802.1X and 802.11 RSN state machines, a session-hijack attack can be performed [MIS02].

Following figure shows the details of session hijack attack:

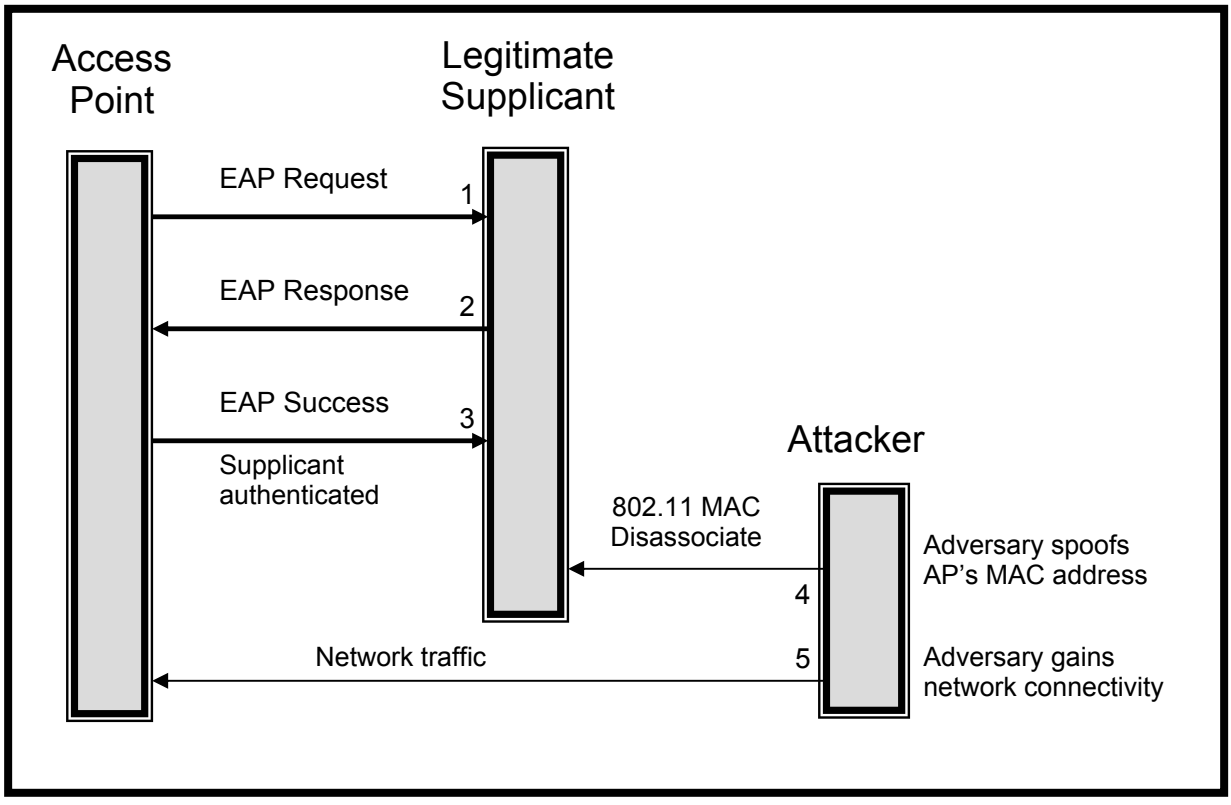

Figure 3.2.8 **Session Hijack Attack**

As we can observe from the figure above, for the message 1, 2 and 3 a legitimate supplicant authenticates itself and EAP authentication has at least three messages (*EAP-Request, EAP-Response,* and *EAP-Success*). Message 4 is sending an 802.11 MAC *disassociate* message from an adversary by using the APs MAC address. Therefore the supplicant gets disassociated and RSN state machine condition changes to *unassociated* state. On the other hand 802.1X state machine of authenticator remains *authenticated* state. In the message 5, the attacker uses the Authenticated supplicant's MAC address and communicates with access point since authenticator is still in *authenticated* state. Then adversary can access the network [MIS02].

3.2.2.3 Denial of Service Attack

The *EAPOL Start/ EAPOL Logoff* messages are sent from the supplicant to authenticator in order to start the authentication process with authenticator/leave the authenticated use of the service. All the fields in packet can be spoofed with MIM attack. In this case an adversary sends an *EAPOL start/EAPOL Logoff* messages to the access point on behalf of the supplicant. This may cause an authenticated client to start a communication with an *attacker/logoff* session. Similarly an *EAP Failure* message is sent from the access point to the supplicant. All these three messages can be spoofed by an adversary. More information on this could be found in [MIS02].

3.2.2.4 Proposed Solutions for Attacks

3.2.2.4.1 Per-Packet authenticity and integrity

Absence of per-packet authenticity and integrity for IEEE 802.11 data and management frames is one the big security problem in protocol history. This can cause the session hijack attack and simple packet forgery attacks. The use of integrity and authenticity of the management and data frames can prevent such attacks. Integrity for

data frames is added when confidentiality is used. On the other hand, for management frames there is integrity protection used [MIS02].

3.2.2.4.2 Authenticity and integrity of EAPOL messages

Lack of authenticity can cause MIM attacks as we explained earlier. Therefore, adding another attribute called *EAP-Authenticator* at the end of message body (only for decision message such as *EAP-Success*) can prevent the packet against MIM attacks. Another solution could be removing explicit *EAP- success* message and instead, using *EAPOL-Key* to show the success at the EAP layer [MIS02].

3.2.2.4.3 Peer-to-peer authentication model

Building the symmetric and scalable authentication techniques on RSN framework, the system becomes a peer-to-peer authenticated model which uses central trusted entity. More details can be found in [MIS02].

CHAPTER 4

CPAL EVALUATION

4.1 CPAL-ES Evaluation of the Secure Protocol

CPAL, Cryptographic Protocol Analysis Language, is a protocol encoding language that helps for analyzing cryptographic protocols with formal methods. The more detailed information about CPAL is given in chapter-2. The CPAL Evaluation System helps us to understand the protocols in detail and proves the correctness of the protocols, tests known attacks, checks the system for possible new attacks and finds solutions to them. In this section we will demonstrate how the interactions during the protocol run could be modeled and represented by CPAL-ES and then, similarly the attacks and attack solutions. One of our goals is to produce the correct implementation of the Secure Protocol with CPAL-ES.

CPAL-ES evaluation of protocols is a three step process:

1. Specifying the protocol actions in CPAL,
2. Generating the verification condition by translating the specification into verification condition,
3. Proving this verification condition with the assumptions [YAS96].

The first step of Aziz&Diffie protocol analysis is implementing the correct version of encoding of the protocol. Then, we can verify and check whether it satisfies its goals. All of the actions and relations among the participating principals in the protocol are checked one by one during the protocol specification. This way we are able to examine the system against possible known and unknown attacks and find the flaws and

solutions to them. Assumptions about the logical beliefs may be used once the specification process is done.

As mentioned in Chapter 3, Aziz&Diffie's Secure Protocol uses both public and shared key cryptographies for session key setup, authentication and privacy. CPAL-ES allows us to analyze both public, asymmetric, key protocols, and private, symmetric, key protocols. A few examples to these can be found in [YAS96]. Not much work has been done with formal methods in the wireless environment especially with CPAL-ES. Our evaluation of the Secure Protocol is the first study on the analysis of wireless public key protocol with CPAL-ES. There is another work which analyses the wireless SRP protocol in ad-hoc network with CPAL [MAR03]. In our work there is no ad-hoc environment (infrastructureless networks); the system is supported by a Central Authority (infrastructure networks). Because CPAL-ES was developed to use in the wired systems, we had some difficulties to use it with wireless protocols, for example, the *send* operator can send to only one principal. In wireless network, every node should be able to send a message to more than one principal at once if needed. To use CPAL-ES with wireless networks, we have to make some assumptions. This problem may not be an issue with the infrastructure network since there is an access point as the central authority. Whereas, it is a big problem with ad-hoc environment. More on this could be found in [MAR03].

CPAL-ES encoding of the Secure Protocol is given in Appendix A and Appendix B shows the CPAL-ES evaluation of the protocol.

Implementing public key protocols with CPAL-ES introduces complexities since we have to deal with both public and private keys and trusted third party called central authority (*CA*). Here *CA*'s role is distributing the certificates, which includes the public keys, of the different principles to each other in order to accomplish a secure session by mostly using the public keys or may be private keys when needed.

The first step of implementation of the Secure Protocol starts with generation of the certificates by *CA*. This part is our addition to the protocol since it was not explained in detail in [AZI94].

The list of acronyms used to define the Secure Protocol in CPAL_ES is shown in Table 4.1.1:

Table 4.1.1 List of acronyms for the Secure Protocol in CPAL

B: Base	**M:** Mobile
CA: Certificate Authority	**I:** Intruder
Kx+: Public key of principal X	**Kx-:** Private key of principal X
Kxy: Shared key between principals X and Y	**sk:** Session Key
MD(Z): The message digest function on contents Z	**f:** Function
e[<X.A,X.B>]X.K: Encryption of A and B in X's address space, under key K in X's address space	**CertX_contents:** Contents of X's certificate
d[X.msg]X.K: Decryption of msg in X's address space under key K in X's address space	**Cert_X:** Certificate of X
ep[ ]: encrypt public (public/private key encryption of public key protocol)	**sn:** Serial Number
dp[ ]: decrypt public (public/private key decryption of public key protocol).	**vp:** Validity Period
CH: Challenge Value (128 bit random number)	**RN:** Random Number

Before the Secure Protocol communication starts, the participants should have their certificates created by a *CA*. First, each principal participating in the protocol run concatenates and stores its public and private key pairs into a single message and sends it to the *CA* by encrypting it in their own address space under a shared key that is shared between each principal and the *CA*. This way public and private key pairs are sent securely by the shared key. This is why, the secure send operator (=>) is used to send the message here. *CA* receives these messages in *CA*'s address space as *msgx* where *x* represents the principal sending message. Then *CA* decrypts these messages and stores into its own address space as *Kx* where *x* is the sending principal (Appendix B, line 10). In the next line, the concatenated message *Kx* splits into its individual

message components as public and private key pairs in *CA*'s address space. In the next step, *CA* concatenates the serial number (*sn*), validity period (*vp*), machine name (*X*), Machine public key (*Kx+*), and *CA* name (*CA*) in its address space and stores them as *CertX_contents* which represents the contents of certificate of principal *X* in C*A*'s address space. In order to generate the certificate of principal *X*, first *CA* computes the *MD* message digest function on *CertX_contents* and assigns the result as *macCA* in its address space. Then it generates *Cert_X*, certificate of *X*, by concatenating *CertX_contents* and encryption of *macCA* under private key of *CA* (*Kca-*) in *CA*'s address space and sends the certificate to principal *X* using secure send (=>).

When *X* receives the message form *CA*, first, it splits up the *Cert_X* into its components as *CertX_components* and *CTforMD* that represents the ciphertext for message digest function *MD* where all operations are in *X*'s address space (Appendix B, line 15-16). Then, it breaks off he *CertX_contents* into its components in *X*'s address space. Finally, *X* decrypts the *CTforMD* under public key of *CA (Kca+)* and stores the result as *macCA* in it's address space and then computes the *MD* function on *CertX_contents* and assigns the result to *macX* in *X*'s address space. Then, *X* compares the *macX* and *macCA* results in the assert statement. If they are not equal then, we may consider of an attack on the protocol run. All these processes explained so far are repeated for each individual principal participating in the protocol run to get the certificate for all parties. In this case, *X* is replaced by the name of the individual party.

After generating certificates and sending them to the individual parties by *CA*, the Secure Protocol starts running with the first message sent from mobile host (*M*) to base station (*B*) with secure send (=>). The first message is also called as *request-to-join* message and a simple one. There is no encryption in the first message. *Cert_M, CH1* and the *list of key algorithms* are concatenated in *M*'s address space and sent in a plain-text. *B* stores the message he receives as *msg1* in his address space (Appendix B, line 37). Next three lines represent the breaking of the *msg1* into its components in *B*'s site. First, he splits up *msg1* as *Cert_M, CH1*, and *List_of_SKCSs* in *B*'s address space. Then he breaks off *Cert_M* as *CertM_contents* and *CTforMD* which is the cipher text of *MD* function in his address space and then *CertM_contents* spited up as *sn2,*

vp2, M, km+, CA in his address space. After getting all the information from *msg1*, *B* creates the next message as a reply to the first message. First, *B* encrypts the *RN1* value under *Km+* and computes the *MD* function on encrypted *RN1*, *chosen_SKCS*, *CH1*, and *List_of_SKSCs* in his address space and stores them as *f1* in his address space. Then *Cert_B, encrypted RN1, chosen_SKCS,* and *encrypted f1* value under private key of *B*, *Kb-*, are concatenated and stored as *msg2* in *B*'s address space. Then *msg2* is sent from *B* to *M* with the secure send (=>). Upon receipt of the second message, first *M* splits up the *msg2* into its components as *Cert_B, CTforRN1* which is cipher text of *RN1, chosen_SKCS, and CTforf1* that is the cipher text of *f1* and stores them in her address space. Then *Cert_B* is broken off to its components as *CertB_contents* and *CTforMDb* which represents the cipher text for *MD* function. Finally, *M* breaks off *CertB_contents as sn1, vp1, B, Kb+* and *CA* in her address space. M decipher s RN1 value from *CTforRN1* by decrypting it under private key of *M*, *Km-*, since it was encrypted under *Km+* and this is a public key protocol and does the same process for *CTforf1* by decrypting it under *B*'s public key, *Kb+*, in *M*'s address space since it was encrypted under private key of, *Kb-*. Now *M* can compute the session key since she has both *RN1* and *RN2* values. *M* computes the *XOR* function on *RN1* and *RN2* values in her address space and stores the result as *sk* that represents the session key, in her address space.

The next step is the generation of third message in *M*'s site. First, *M* computes *MD* function on encrypted *RN1* under *Km+* and *RN2* under *Kb+* in address space of *M* and the result is stored in *M*'s address space as *f2*. Then, *M* concatenates encrypted *RN1* under *Kb+* and encrypted *f1* under *Km-* in her address space and sends it to *B* using secure send operator (=>). When *B* receives the message from *M*, first, he stores them as *msg3* in his address space. Then, he splits up *msg3* as C*TforRN2* and *CTforMD* which represent the cipher text of *RN2* and *MD* function in *B*'s address space. Next, *B* deciphers *CTforRN2* and *CTforMD* by decrypting them under *Kb-* and *Km+* and stores the results as *RN2* and *f2* in address space of *B* (Appendix B, line 55-56). Finally the last step is computing *MD* function on encrypted *RN2* under *Kb+* and encrypted *RN1* under *Km+. B* stores the result as *f3* in his address space. Here *f3* is computed for security issue, to test that he received the right information of *f2*. *f2* and *f3* values

supposed to be the same. If it is so then there is no attack on the system. Then, *XOR* function is computed on *RN1* and *RN2* and stored as *sk* in *B*'s address space.

After the specification which is the first step of CPAL-ES process is done all the syntax errors should be removed. Since CPAL is a parsed language with a syntax checker, we can use it interactively either in WINDOWS/DOS or UNIX environment to quickly and easily check the protocol encoding for syntax errors. Once the syntax free CPAL specification is generated, the specification is translated into verification condition. Translating the procedural protocol description that is given in CPAL into a predicate whose ultimate value is either TRUE or FALSE. The final predicate reflects whether the protocol accomplish its goals within the protocol steps.

Generation of the verification conditions is the second step of evaluation as mentioned in Chapter 4.1. Formal semantics for the protocol is automatically derived by CPAL-ES. This is accomplished in two steps [YAS96]:

1. Beginning from the last protocol statement in the specification, the process is done in reverse to the first statement in the protocol specification and each statement definition is applied to an initial predicate with value TRUE.

2. The definition of encryption, decryption operators, value catenation and extraction and assumptions are applied to the predicate when applicable in the protocol specification. The resulting protocol is scanned for conditions allowing replacement.

During the verification process, each protocol statement in CPAL is translated to weakest precondition predicate. All the weakest precondition transformations of the Secure Protocol as the definition for each statement is applied are shown in Appendix B. Weakest precondition definitions are applied in reverse order. This is why initial predicate state of TRUE is at the end of the predicate. The final predicate is the verification condition for the protocol. If it can be proved from assumptions, then protocol is guaranteed to meet its specified goals. This could be easier for the simple protocols but it may be very difficult with more complex protocols [YAS96].

Because of the public key encryption nature of the Secure Protocol, verification is a very long process. We cannot use the key used for encryption for the decrypting the encrypted value. In order to invert the public key encryption, we should use a decryption key which is the inverse of the encryption key. We can see this in the first few lines of the protocol evaluation (Appendix B) in the assume statements. In order to represent the public key encryption and decryption, *global.decrypt* function is used. We have five of these in our encoding of Secure Protocol. In general, we can show this predicate as:

global.decrypt(k1,k2)

This proposes that public key decryption under *k2* inverts encryption under *k1* that shows the relationship between the keys. More on this can be found in [YAS96]. Creating the predicates for ASSUME and ASSERT statements in the verification conditions plays an important role during the proving process in order to reduce the final predicate to TRUE.

The final step of the CPAL evaluation system process is proving the verification conditions with the result of TRUE with the given assumptions in the protocol. During the verification process, the predicates may have a repetition of logical conditions. Therefore, CPAL-ES has a feature that it can scan the predicate for such condition and may reduce them through the simple logical reductions, which is called the simplification process. After simplification, the predicate is reduced to its simplified form as seen at the end of Appendix-B for the Secure Protocol of Aziz & Diffie. This makes the proving process much easier. The simplification process is repeated and combined with the encryption and catenation operation searches until all of the matches are found and the predicate is reduced to its most simplified form. The analyst will be left with more challenging predicate proving process. He has to make a decision of how many assumptions and assertions necessary in order to prove the final predicate as TRUE and needs to decrease the number of assumptions and assertions used as possible in order to get a predicate easier to prove.

ASSERT statement is used to express beliefs of principals in CPAL in order to specify protocol goals to be proven. A conjunctive condition is added to the predicate defined so far as a result of the application of ASSERT statement definition. The last

statement in the protocol is usually a goal expressed in ASSERT statement. Only the ASSERT statement has an effect on the initial predicate, since CPAL-ES evaluation is based on the values in the predicate. CPAL-ES evaluation does not consider any statement after the last ASSERT statement in the predicate. ASSERT statement is used only for comparing the data elements within the address space of only one principal. When comparison of values in different principal's address space is necessary we need use another statement called "GASSERT" to evaluate matters of security [YAS96].

Assumptions are necessary in order to complete the proof. ASSUME statements enable to modify the predicate based on the logical analysis. They are used to specify assumptions which are stated as predicates for creating the truth of the goals. When truths are replaced with the boolean value TRUE, they may be removed from the predicate. Assumed values are applied continually until no changes to predicate occur [YAS96].

In the Secure Protocol of Aziz & Diffie, we have used seven ASSUME statements [Appendix B] in order to express the equality of keys used between different principals for proving the truth of goals of the protocol.

Consider the following lines from Appendix B where verification condition for decryption and decryption is shown (Appendix B, lines 8-10):

```
B: => CA(e[<B.Kb+,B.Kb->]B.Kbca);
CA: <-(CA.msgb);
CA: CA.Kb := d[CA.msgb]CA.Kbca;
```

In the line 8, B sends *CA* an encrypted value under *Kbca* shared key. This encrypted value is then saved as *msgb* into *CA*'s address space. In the line 10, the verification condition is shown where the decrypted value is stored as *Kb* in *CA*'s address space. We can change the line 10 as:

```
CA: CA.Kb == d[e[<B.Kb+,B.Kb->]B.Kbca)]CA.Kbca;
```

In this verification condition, we observe the predicate as the right side of the equivalence statement fits the form of the decryption of the encryption of Kb+ and Kb-

values under *Kbca* in *CA*'s address space. The entire encryption/decryption values could be replaced by the value *Kb* in *CA*'s address space if the keys of the operations (*Kbca* in *B*'s address space and *Kbca* in *CA*'s address space) were the same. Therefore when we add the following statement (line 3, Appendix B) into the protocol specification that assumes the *Kbca* in *CA*'s address space and *Kbca* in *B*'s address space are equal, then final predicate will be reduced to TRUE automatically:

```
X: assume((CA.Kbca == B.Kbca));
```

As an alternative to this ASSUME statement, we can describe the preexistence of the shared key with a sequence of key exchange as shown below:

```
B: => CA (Kbca);
CA <- Kbca;
```

These two lines replaces the ASSUME statement shown above. Having them in the protocol makes it possible to decrypt the encrypted values with either *B.Kbca* or *CA.Kbca* since evaluation with weakest precondition can detect the values originated from the same source. In this case, decryption will invert the previous encryption since these two shared keys are equal.

Similar situation occurs with the following ASSUME statement in the Secure Protocol specification (line 4, Appendix B):

```
X: assume((CA.Kmca == M.Kmca));
```

This statement represents the assumption which states that principal *M* and *CA* shares the common key, *Kmca*. It is necessary to include this assumption in the predicate in order to get the final predicate as TRUE.

The situation is different with public key encryption. Recall that both shared key cryptography and public key cryptography are used with the Secure Protocol of Aziz & Diffie. The CPAL predicate with shared key cryptography is not much difficult since the same key is used for encryption and decryption processes. On the other hand, with

public key cryptography, inverse of the encryption key is used for the decryption function. This is represented with the following special predicate in general:

global.decrypt(k1,k2)

This predicate represents the relationship between keys *k1* and *k2* where the public key decryption under the key *k2* inverts the encryption under key *k1* as stated in [YAS96]. This situation could be assumed by replacing this relation by an ASSUME statement.

In the Secure Protocol specification, five of these assumptions are used in order to get the final predicate as TRUE. The predicate:

(dp[ep[CA.macCA]CA.Kca-]B.Kca+) == B.macCA

will be reduced to TRUE with the given assumption:

X: assume(global.decrypt(B.Kca+,CA.Kca-));

With this assumption (line 1, Appendix B), the relationship between *CA*'s public key in *B*'s address space *(B.Kca+)* and *CA*'s private key in *CA*'s address space (*CA.Kca-*) is assumed that the decryption under *B.Kca+* inverts the encryption under CA.Kca-. In this case we can assume that the encryption of *macCA*, which is the *MD* function of certificate contents in *CA*'s address space, under *Kca-* could be decrypt as *macCA* in *B'*s address space under *Kca+* in *B*'s address space as seen above.

The similar situation could be observed with the following assumption (line 2, Appendix B):

X: assume(global.decrypt(M.Kca+,CA.Kca-));

This statement proposes that the relationship between the keys *Kca+* in *M*'s address space and *Kca-* in *CA'*s address space is that decryption under *M.Kca+* inverts encryption under *CA.Kca-* is reduces to TRUE for the following predicate:

(dp[ep[CA.macCAm]CA.Kca-]M.Kca+) == M.macCAm

where *macCAm* represents the *MD* function of certificate contents in *CA*'s address space.

Another ASSUME statement is (line 5, Appendix B):

X: assume(global.decrypt(B.Kb-,B.Kb+));

This assumption reduces the following predicate to TRUE:

(dp[ep[M.RN2]M.Kb+] B.Kb-) == B.RN2

In this predicate encryption of random number *RN2* in *M*'s address space under public key of *B* in *M*'s address space, *M.Kb+,* is decrypted under private key of *B* under *B*'s address space, *B.Kb-*. The value came out of decryption is equal to *RN2* in *B*'s address space with the assumption above.

The next ASSUME statement in the predicate is (line 6, Appendix B):

X: assume(global.decrypt(M.Km-,M.Km+));

With this statement it is assumed that there is a relationship between private and public keys of *M (Km+, Km-)* such that decryption under *Km+* inverts the encryption under *Km-.* The predicate below is reduced to TRUE with this ASSUME statement:

M: dp[ep[M.RN1]B.Km+]M.Km- == M.RN1

Where *RN1* is a random number generated, encrypted under *Km+* and sent to *M* by *B*. As seen above, encrypted *RN1* in *M'*s address space under *Km+* in *M*'s address space is decrypted under *Km-* in *M*'s address space and the result is saved as *RN1* in *M*'s address space.

Another assumption is (line 7, Appendix B):

X: assume(global.decrypt(B.Kb+,B.Kb-));

This reduces the following predicate to TRUE:

M: M.f1 := dp[ep[B.f1]B.Kb-]M.Kb+;

Where *f1* is *MD* function on *RNb, M, CHm* and *chosen_SKCS* in *B*'s address space. Here encrypted value of *f1* in *B*'s address space under *B.Kb-* is decrypted under *Kb+* in *M* address space and the result stored as *f1* to *M'*s address space. By looking at this predicate the assumption statement should have the information about *Kb-* in *B*'s address space and *Kb+* in *M*'s address space. On the other hand, as we can see from the last ASSUME statement above, there is only information about *Kb-* and *Kb+* in *B*'s address space. Since *Kb+* was originally generated by principal *B*, sent to *CA* and distributed from *CA* to the other principals needed the information, when *B* has equal *Kb+* and *Kb-, Kb+* in any principal's address space will be the same and equal to *B.Kb-.* Therefore, instead of two different assumptions as seen below that are needed for the Meadows attack specification (Appendix D), we can have only one assumption as mentioned above (line 7, Appendix B):

X: assume(global.decrypt(M.Kb+,B.Kb-));
X: assume(global.decrypt(I.Kb+,B.Kb-));

Finally, the last ASSUME statement is (line 8, Appendix B):

X: assume(global.decrypt(M.Km+,M.Km-));

This is very similar to the previous ASSUME statement except the roles of *Km+* and *Km-* are interchanged. By adding this ASSUME statement the verification condition:

B: dp[ep[M.f2]M.Km-]B.Km+ == B.f2

simplifies to TRUE. Here *f2* is a *MD* function of encrypted *RN1* and *RN2*. In this predicate encrypted *f2* in *M*'s address space under *Km-* in *M*'s address space is decrypted under *Km+* in *B*'s address space and saved as *f2* in *B*'s address space.

In the Secure Protocol specification, there are three ASSERT and three GASSERT statements used to show the goals of the protocol to be proven. Most of them are at the end of the specification as mentioned earlier (Appendix B). The first ASSERT statement is (line 21, Appendix B):

```
B: assert((B.macB == B.macCA));
```

Where *macCA* is the decrypted value, which was encryption of *MD* function on certificate contents created in *CA*'s address space and sent to *B*'s address space and *macB* is *MD* function on certificate contents in *B*'s address space. With this ASSERT statement *B* expresses the goal that we desired to prove of believing that *CA* sent the message by comparing the values one comes from *CA* but now in *B*'s address space and another one derived in *B*'s address space. When this statement is proved we can be sure that certificate contents are the original as created by *CA* and no one in the middle possibly intercept and change the message.

The next ASSERT statement is (line 35, Appendix B):

```
M: assert((M.macM == M.macCAm));
```

This statement is very similar to the previous one, except it compares the values originated in *CA*'s and *M*'s address space and compared when they both are in *M*'s address space. *macCAm* represents the *MD* function on certificate contents in *CA*'s address space and *macM* represents the *MD* function on certificate contents in *M*'s address space. Proving this goal again confirms that the message is not changed.

The last ASSERT statement expresses the goal as whether the third message send form *M* to *B* is changed as seen below (line 58, Appendix B):

```
B: assert((B.f3 == B.f2));
```

where f2 represents the MD function on encrypted *RN1* and *RN2* values originated in *M*'s address space and came to *B*'s queue and the address space when *B* received the third message from *M*. *f3* is *MD* function again on encrypted values of *RN1* and *RN2* where originated in B's address space.

The GASSERT statements shown below create the goal as to whether the random numbers are the same (line 59 & 60, Appendix B):

```
X: gassert((M.RN1 == B.RN1));
X: gassert((M.RN2 == B.RN2));
```

Since this is the assessment of values in different principals, GASSERT is used instead of ASSERT statement. First GASSERT statement compares *RN1* in *M*'s address space and *B*'s address space which was originated by *B*. Second GASSERT represents the similar goal that checks whether *RN2* value that was created by *M* in *M*'s address space and sent to *B*. Therefore proving the equality of *RN2* in *M*'s address space and *B*'s address space results as that *RN1* and *RN2* values sent from/to *B* and *M* are the same and not intercepted by an eavesdropper.

Finally, the last GASSERT statement and also last statement of the predicate is (line 62, Appendix B):

```
X: gassert((B.sk == M.sk));
```

The goal of this statement is to compare the session key values, *sk*, in *B*'s and *M*'s address spaces. Both *B* and *M* calculates *XOR* function on *RN1* and *RN2* values and save the result as s*k* in their address space. Proving this predicate as TRUE means that protocol reaches its goal since both *B* and *M* has the same *RN1*, *RN2* and session key, *sk*, values and they are not changed by an intruder.

CPAL-ES encoding of protocols and attacks is easy to follow and the result of evolution is very simple to understand. CPAL-ES captures the essence of the protocol and shows how the attacks work in great detail. Correctness of each field in a data packet which comes to the receiver by a *send* operation can be easily checked with *assert/gassert* statements. Therefore, *receive* operation feature of CPAL-ES plays very

important role in doing the detailed comparison of data values which comes from mobile-to-base and base-to-mobile stations. This way, problems can be easily found as they occur. In the following sections, some of the attacks on the Secure Protocol are verified by CPAL-ES evaluation.

4.1.1 CPAL-ES Evaluation of the Meadows Attack on the Secure Protocol

CPAL-ES encoding of the Meadows attack on the Secure Protocol is given in Appendix C and Appendix D shows the CPAL-ES analysis of the attack. We have discussed the attack in detail. Here we will only mention the CPAL-ES analysis of the attack. Recall that ASSERT and ASSUME statements play very important role during the analyzing process since ASSERT statements are the goals of the predicate, and ASSUME statements are the predicates that are verified as the truth of the goals. Some of the ASSUME and ASSERT statement are the same as in the Secure Protocol encoding. We will not mention them here again since they play the same role in the protocol. Therefore we will not discuss the assumptions (lines 1&2, Appendix D):

```
X: assume(global.decrypt(B.Kca+,CA.Kca-));
X: assume(global.decrypt(M.Kca+,CA.Kca-));
```

One of the ASSUME statement that is not in the Secure Protocol simplification is (line 3, Appendix D):

```
X: assume(global.decrypt(I.Kca+,CA.Kca-));
```

This statement proposes that the relationship between the keys *Kca+* in intruder *I*'s address space and *Kca-* in CA's address space is that decryption under *CA.Kca-* inverts encryption under *I.Kca+.* This assumption reduces to TRUE for the following predicate:

```
(dp[ep[CA.msgi]CA.Kca-]I.Kca+) == I.macCA
```

where *msgi* represents the encrypted public and private keys of *I*, *Ki*+ and *Ki*-, in *CA*'s address space and *macCA* is the decryption of encrypted *Ki*+ and *Ki*- under *Kca*+ in *I*'s address space.

The next ASSUME statement that is not in the Secure Protocol encoding is (line 4, Appendix D):

X: assume((CA.Kica == I.Kica));

With this statement it is assumed that the Kica, the key shared between *CA* and *I*, in *CA*'s address space and *I*'s address space are equal in order to result the following predicate as TRUE:

CA: CA.Ki := d[CA.e[<I.Ki+,I.Ki->]I.Kica]CA.Kica;

As it seen above, *CA* decrypts the encrypted public and private keys of *I, Ki*+ and *Ki*-, under *Kica* in *CA*'s address space and stores it as *Ki* in *CA*'s address space.

Similarly following two assumptions is used for reducing the similar predicates, in *B*'s and *M*'s address spaces, to TRUE as explained in the Secure Protocol evaluation (line 5&6, Appendix D):

X: assume(CA.Kbca == B. Kbca)
X: assume(CA.Kmca == M. Kmca)

As we discussed in the Secure Protocol evaluation, the following predicate reduces to TRUE with the following assumption (line 7, Appendix D) in Meadows attack:

B: B.RNm:=dp[B.(ep[M.RNm]M.Kb+)]B.Kb-
X: assume(global.decrypt(B.Kb-,B.Kb+));

Another ASSUME statement is (line 8, Appendix D):

X: assume(global.decrypt(B.Kb+,B.Kb-));

where, *Kb+* and *Kb-* are the public and private keys of *B* and decryption under *Kb+* inverts the encryption under *Kb-* in order to result the following predicates to TRUE,

I: I.f1 := dp[I.ep[B.f1]B.Kb-]I.Kb+;
M: M.f1 := dp[M.ep[B.f1]B.Kb-]M.Kb+;

where *f1* is the *MD* function on encrypted *RNb* under *Ki+, chosen_SKCS, CHm* and *List_of_SKCSs* in *B*'s address space. In the first predicate above, encrypted *f1* value under *Kb-* in *B*'s address space is decrypted under *Kb+* in *I*'s address space and stored as *f1* in *I*'s address space. In the second one, same process is done in *M*'s address space instead of *I*'s address space.

The assume statement below is discussed earlier in the Secure Protocol evaluation (line 9, Appendix D):

X: assume(global.decrypt(M.Km+,M.Km-));

This time it reduces the different predicate to TRUE as seen below:

I: I.f2 := dp[I.ep[M.f2]M.Km-]I.Km+

where *f2 Km-* and *Km+* are the private and public keys of *M* and *f2* in *M*'s address space is *MD* function on encrypted *RNm* under *Kbi+* in *M*'s address space and encrypted *RNb* under *Ki+* in *B*'s address space. Here encrypted *f2* value under *Km-* in *M*'s address space is decrypted under *Km+* in *I*'s address space.

Finally the last two assume statements are (lines 10&11, Appendix D):

X: assume(global.decrypt(I.Ki+,I.Ki-));
X: assume(global.decrypt(I.Ki-,I.Ki+));

It is proposed that the relationship between the keys *Ki+* and *Ki-* in intruder *I*'s address space is that decryption under *I.Ki+*, inverts encryption under *I.Ki-* or vise versa. The

following predicates (lines 82&109, Appendix D): are reduced to TRUE with the assumptions above:

I: I.RNb := dp[I.ep[B.RNb]B.Ki+]I.Ki-;
B: B.f3 := dp[B.ep[I.f2']I.Ki]B.Ki+;

where *RNb* is the random number generated by *B*, encrypted under *Ki+* in *B*'s address space and decrypted under *Ki-* and stored as *RNb* in *I*'s address space. In the second predicate *f2'* represents the *MD* function on encrypted *RNm* and encrypted *RNb* values in *I*'s address space.

There are number of ASSERT statements in Meadows attack specification. Again we will only discuss the once are not in The Secure Protocol specifications. The first ASSERT statement is (line 25, Appendix D):

I: assert((I.macI == I.macCA));

Where *macI* is the *MD* function on certificate contents in *I*'s address space and *macCA* is the decryption of *MD* function on certificate contents under *CA*'s public key *Kca+* that comes from *CA*'s address space. Here *I* checks whether he receives the correct version of certificate contents.

In the next ASSERT statement *I* does another similar checking operation on certificate contents by comparing *macMI* and *macCAmi* in *I*'s address space as seen below (line 63, Appendix D):

I: assert((I.macMI == I.macCAmi));

Here *macMI* is the *MD* function on certificate contents in *I*'s address space and *macCAmi* is the decryption of encrypted *MD* function on certificate contents for *M* under public key of *CA, Kca+* in *I*'s address space.

Following is the next ASSERT statement in Meadows attack specification (line 71, Appendix D):

B: assert((B.macIB == B.macCAib));

where *macIB* represents *MD* function on certificate contents of *I* in *B*'s address space and *macCAib* represents *MD* function on certificate contents of *I* in *B*'s address space. Here again the certificate contents in *B*'s address space is checked to see whether it is the correct version. Another check is done on the certificate contents of *B* in *I*'s address space, *macBI*, and decryption of the encrypted certificate contents of *B* in *I*'s address space, *macCAbi* as seen below (line 81, Appendix D):

I: assert((I.macBI == I.macCAbi));

The following statement checks *f1* and *f1'* values in *I*'s address space (line 85, Appendix D):

I: assert((I.f1' == I.f1));

where *f1'* is *MD* function on encrypted *RNb*, *chosen key*, *CHm* and the *list of keys* in *I*'s address space and *f1* is the decryption of encrypted group of values in *B*'s address space as listed above in *I*'s address space.

Following is another ASSERT statement where *macBM* is *MD* function on certificate contents in *M*'s address space and *macCAbm* is decryption of encrypted *MD* function on certificate contents of *B* in *M*'s address space (Line 93, Appendix D):

M: assert((M.macBM == M.macCAbm));

These ASSERT statements reflect the knowledge that an intruder has the information such as certificate contents for the statements above. With this, *I* assert statement, *M* compares the values as seen above. If *macBM* in *M*'s address space is equal to *macCAbm* in *M*'s address space then, certificate content is not changed by an intruder. On the other hand we will not be sure that whether the intruder is only listener and has gotten the information without changing it.

The following three ASSERT statements are the next ones in Meadows attack on

the Secure Protocol specifications (Lines 96,103 & 111, Appendix D):

M: assert((M.f1' == M.f1));
I: assert((I.f2' == I.f2));
B: assert((B.f3' == B.f3));

With the first ASSERT statement, *M* compares the *f1'* and *f1* values in her address space where *f1'* is *MD* function on encrypted *RNb*, *chosen key, CHm*, and *list of keys* and *f1* is decryption of encrypted *f1* value that comes from *I* originally from *B* and includes the *MD* function on list of variables as mentioned above. The next statement shows whether *f2'* and *f2* values in *I*'s address space are equal. Here, *f2'* is *MD* function on encrypted *RNm* and *RNb* values and *f2* is decryption of encrypted *MD* function on encrypted *RNm* and *RNb* originally in *M*'s address space, sent from *M* to *B* but received by *I* therefore in *I*'s address space. Finally the last one checks whether *f3'* and *f3* are equal where *f3'* represents again *MD* function on encrypted random numbers *RNb* and *RNm* in *B*'s address space and *f3* is decryption of encrypted *f2'* that was sent from *I* to *B* and now in *B*'s address space.
The last two assert statements are the last ones in the Secure Protocol Meadows attack specification that are similar to the last two statements in the Secure Protocol specification (Lines 112 & 113, Appendix D):

X: gassert(M.RNm == B.RNm);
X: gassert(I.RNb == B.RNb);

Recall that GASSERT statements are used when the values in two different principals are compared. Here in the first GASSERT statement, *RNm* in *M*'s address space and *RNm* in *B*'s address space are compared to see whether they are the same values. In the second statement *RNb* in *I*'s address space and *RNb* in *B*'s address space are checked to see if both *B* and *I* are the same values.

After starting with the last predicate and checking all the ASSUME and ASSERT statements in the specifications, verification conditions for Meadows attack simplifies to TRUE. With this result, CPAL-ES evaluation of the Secure Protocol captures the core of

the protocol, provides a detailed evaluation and proves the Meadows attack on the Secure Protocol.

When we run the Meadows attack, all the comparisons with *assert/gassert* statements in the protocol encoding prove the correctness of data fields in the packets that comes from base-to-mobile stations or vice versa. This result confirms that an intruder can spoof the message sent from mobile-to-base and base-to-mobile stations. Then, neither base station nor mobile station can prove that the message is spoofed by an intruder because the intruder impersonates C and B. In other words, even the mobile host M sends the first message to the base station B, when B receives the first message that comes from the intruder he thinks that it is coming from C who B thinks that is a certified host since it has a certificate from CA, Cert_C. Then he starts communicating with C by sending his certificate, Cert_B, and the random number *RNb* he creates to construct the session key. This way, the intruder gets some information but not all that is necessary to calculate the session key. On the other hand, when M gets the reply message from the intruder, the followings happen:

- M thinks that the message came from B since the intruder sends B's certificate, Cert_B as his certificate,
- In this case, M thinks that she received the random number, RNb, as it is encrypted under M's public key, Km+,
- Then, she tries to decrypt it under her private key, Km-,
- In fact, RNb is encrypted under C's public key, Kc+, by the intruder impersonating C.
- Then, mobile station M thinks that the random number comes from B is the decrypted value (dp[ep[RNb]Kc+]Km-).
- Therefore M gets the wrong random number in order to calculate the session key.

As a result, even the intruder does not get all the necessary information in order to get the session key, it causes the mobile station M to get the wrong random value and therefore wrong session key. As a result, the Secure Protocol becomes vulnerable to Meadows attack as it is proved by CPAL-ES evaluation.

4.1.2 CPAL-ES Evaluation of the Boyd & Mathuria Attack on the Secure Protocol

CPAL-ES encoding of the Boyd&Mathuria Attack on the Secure Protocol is given in Appendix E and Appendix F shows the CPAL-ES evaluation of the attack.

As mentioned in Chapter 2.1.4, we check the ASSUME and ASSERT statements in order to evaluate the Boyd&Mathuria attack. All of the assumptions in the specification of Boyd&Mathuria attack are discussed either in the Secure Protocol specification or Meadows attack on the Secure Protocol specification since Boyd&Mathuria attack is similar to the Meadows attack. Therefore, we will not discuss these assumptions here again. On the other hand, some of the predicates that are reduced to TRUE by these assumptions are different predicates than they are in the Secure Protocol and Meadows attack Therefore we will talk about them here again.

The following assumptions are already discussed in Meadows attack and some in the Secure Protocol (lines 1-6, Appendix F):

```
X: assume(global.decrypt(B.Kca+,CA.Kca-));
X: assume(global.decrypt(M.Kca+,CA.Kca-));
X: assume(global.decrypt(I.Kca+,CA.Kca-));
X: assume((CA.Kica == I.Kica));
X: assume((CA.Kbca == B.Kbca));
X: assume((CA.Kmca == M.Kmca));
```

The next ASSUME statement is also in the Secure Protocol specification but it reduces the following predicate to TRUE which is almost the same predicate used in the Secure Protocol specification where only *RNb* is used instead of *RN2* (line 7, Appendix F):

```
X: assume(global.decrypt(M.Km-,M.Km+));
M: M.RNb := dp[ep[B.RNb]B.Km+]M.Km-
```

Another assumption is (line 8, Appendix F) the same as in Meadows attack but *f1* in the predicate is different here:

X: assume(global.decrypt(B.Kb+,B.Kb-));

where there is a relation between the public and private keys of *B, Kb*+ and *Kb*-, such that the decryption under *Kb*+ in *B*'s address space inverts the encryption with *Kb*- in *B*'s address space. This assumption reduces the following predicate to TRUE:

I: I.f1 := dp[I.ep[B.f1]B.Kb-]I.Kb+;

where *f1* is the *MD* function on encrypted *RNb* under *Km*+, *chosen_SKCS, CHi* and *List_of_SKCSs* in *B*'s address space. Here encrypted *f1* value under *Kb*- in *B*'s address space is decrypted under *Kb*+ in *I*'s address space and stored as *f1* in *I*'s address space.

Following is the next ASSUME statement in Boyd&Mathuria attack specification where there is a relation between public and private keys of *M*, *Km*+ and *Km*-, that decryption under *Km*+ inverts the encryption under *Km*- (line 9, Appendix F):

X: assume(global.decrypt(M.Km+,M.Km-));

This assumption reduces the predicate below to TRUE:

B: B.f2 := dp[ep[M.f2]M.Km-]B.Km+

where *f2* is the *MD* function on encrypted *RNm* under *Ki*+ in *M*'s address space and encrypted *RNb* under *Kb*+ in *M*'s address space.

The next assumption is (line 10, Appendix F):

X: assume(global.decrypt(I.Ki+,I.Ki-));

where the relation between public and private keys of *I*, *Ki+* and *Ki-*, is that decryption under *Ki+* inverts the encryption under *Ki-*. Following predicate is reduced to TRUE with this assumption:

M: M.f1:= dp[ep[I.f1]I.Ki-]M.Ki+;

Here *f1* is a *MD* function on encrypted *RNb* under *Km+* originally in *B*'s address space, but later stored in *I*'s address space, *chosen_SKCS, CHm and List_of_SKCSs* in *I*'s address space. The result is stored as *f1* in *M*'s address space.

Finally the last assumption is (line 11, Appendix F) that reduces the predicate followed by the assumption is reduced to TRUE:

X: assume(global.decrypt(I.Ki-,I.Ki+));
I: I.RNm := dp[ep[M.RNm] M.Ki+]I.Ki-;

where encrypted *RNm* under *Ki+* in *M*'s address space is decrypted under *Ki-* in *I*'s address space and stored as *RNm* in *I*'s address space. This decryption under *Ki-* inverts the encryption under *Ki+*.

There are few ASSERT statements that are the same as in Meadows attack (lines 25, 39, 55, 73, 84, 112 and 114, Appendix F):

I: assert((I.macI == I.macCA));
B: assert((B.macB == B.macCA));
M: assert((M.macM == M.macCAm));
I: assert((I.macBI == I.macCAbi));
I: assert((I.macMI == I.macCAmi)
X: gassert((M.RNb == B.RNb));
X: assert((B.sk == M.sk));

We will not discuss these again here. The first ASSERT statement that is not in the Secure Protocol and Meadows attack is (line 63, Appendix F):

B: assert((B.macMB == B.macCAmb));

where *macMB* in *B*'s address space is *MD* function on certificate contents of *M* in *B*'s address space and *macCAmb* is the decryption of the encrypted certificate contents of *M* under *Kca+* in *B*'s address space. With this assertion *B* checks the *CertM_contents* in its own address space but came there earlier from *I* and certificate contents of *M* comes from *CA* but actually from intruder. If these are the different values, than *I* changes the certificate contents of *M*.

The next assertion is (line 76, Appendix F):

I: assert((I.f1' == I.f1));

Here *f1'* in *I*'s address space is the *MD* function on encrypted *RNb* under *Km+* in *B*'s address space, *chosen_SKCS, CHi, List_of_SKCSs* in *I*'s address space and *f1* is *MD* function on encrypted *RNb* under *Km+* in *B*'s address space, *chosen_SKCS, CHi*, *List_of_SKCSs* in *B*'s address space. *I* checks these to see if he receives the correct version of *MD* functions.

Another assertion is (line 94, Appendix F):

M: assert((M.macIM == M.macCAim**));**

With this ASSERT statement *M* checks *macIM* and *macCAim* values in *M*'s address space where *macIM* is *MD* function on *CertI_contents* in *M*'s address space and *macCAim* is decryption of encrypted *MD* function on *CertI_contents* in *M*'s address space under *Kca+*.

Lastly, the next assertion is (line 111, Appendix F):

assert((B.f3' == B.f3));

where *f3'* is *MD* function on encrypted *RNm* under *Ki+,* and encrypted *RNb* under *Km+* in *B*'s address space and *f3* is decryption of encrypted *f2* under *Km+* in *M*'s address space that *f2* represents encryption on *RNm* under *Ki+* and encryption of *RNb* under

Km+ in *M*'s address space. With this comparison *B* checks to see whether *MD* function in his address space is the same as *MD* function in *M's* address space. This way an attacker trying to spoof the *MD* functions could be caught.

Finally, we could also add the following assertions to the Boyd & Mathuria specification after the lines 97 and 105 (Appendix F):

```
M:assert (M.f1==M.f1');
M:assert (M.f2==M.f2');
```

Where *f1'* is *MD* function on encrypted *RNb* under *Km+* in *M*'s address space, *chosen_SKCS, CHm*, and *List_of_SKCSs* in *M's* address space and *f1* is decryption of encrypted *f1* under *Ki+* in *M*'s address space. Here encrypted *f1* is encryption of *MD* function on encrypted *RNb, chosen_SKCS, CHm, List_of_SKCSs* in *I*'s address space. *f2* is decryption of encrypted *f2* which is *MD* function on encrypted *RNm* under *Ki+* and encrypted *RNb* under *Km+*. *f2'* is *MD* function on encrypted *RNm* under *Ki+* and encrypted *RNb* under *Km+* in *M*'s address space.

Encoding works with them also and reduces the result of TRUE but in this case computation time increases. Therefore we did not add these in our specification.

In the end, going back from last predicate to the first predicate, all the evaluation of the Boyd&Mathuria Attack on the Secure Protocol reduces to TRUE meaning that the Secure Protocol is vulnerable to the Boyd&Mathuria attack.

CPAL-ES encoding of the Boyd&Mathuria attack shows the attack in great detail. All the values in the data packets are checked with *assert/gassert* statements in detail by the received party. The evaluation clearly shows that TRUE result of all of the assert/gassert predicates result in final predicate as TRUE which confirms the attack.

When we run the Boyd&Mathuria attack encoded with CPAL-ES, we get a similar result as we get from Meadows attack. In Boyd&Mathuria attack, the intruder spoofs B by intercepting the message that comes from mobile station M. M communicates with C whom M thinks that of a certified party. B thinks that he is communicating with M but actually communicating with the intruder C. The intruder that impersonates C receives M's certificate, Cert_M, and the random number cretated by M, RNm, in the message sent from the mobile host M to the intruder. Then, the intruder keeps Cert_M in the

message, changes the challenge value with her own challenge value, CHi, and sends the message to the base station B. Therefore, when B receives the message he thinks that it is coming from M. He encrypts his random value, RNb, under M's public key, Km+, since he thinks that M will get his message. He also puts the same encrypted value along with his challenge value CHi, the list of keys, and chosen key in a hash function and signs it with his private key, Kb-. When C receives the packet, she keeps almost the same information in the packet as it comes from B, except she replaces Cert_B with Cert_C, and CHi with CHm, and signs the values under Ki-. When M receives the message, she thinks that the message comes from certified host C and replies back to him with her encrypted RNm value under Ki+ along with an encrypted RNm under Ki+ and encrypted RNb under Km+. Consequently she signs these two encryptions under Km-. Finally, C forwards the same packet without changing it to the base station B. In this case, when B receives the packet he thinks that the message comes from M and tries to decrypt the encrypted RNm value under Kb-, and considers the result as random number RNm (actually it is d[e[RNm]Ki+]Kmb- not RNm). In reality, this value was encrypted under Ki+ and it should have been decrypted under Ki-. At this time, B has the wrong RNm value. As a result, the base station B has the false RNm value and therefore computes a false session key. This proves that the Secure Protocol is vulnerable to Boyd&Mathuria attack. A solution to this attack is given in the following section.

4.1.3 CPAL-ES Evaluation of the Solution to the Boyd&Mathuria Attack on the Secure Protocol

CPAL-ES encoding of the solution to Boyd&Mathuria Attack on the Secure Protocol is given in Appendix G and Appendix H shows the CPAL-ES evaluation of the solution to the attack.

Again some of the assumptions and assertions in the solution of Boyd&Mathuria Attack are already discussed with evaluation of either the Secure Protocol, or Meadows Attack, or Boyd&Mathuria attack. Therefore the following assumptions and assertions will not be mentioned again (lines 1, 2, 3, 4, 5, 6, 22, 36, 62, 63, 65, Appendix H):

X: assume(global.decrypt(B.Kca+,CA.Kca-));
X: assume(global.decrypt(M.Kca+,CA.Kca-));
X: assume((CA.Kbca == B.Kbca));
X: assume((CA.Kmca == M.Kmca));
X: assume(global.decrypt(B.Kb-,B.Kb+));
X: assume(global.decrypt(M.Km-,M.Km+));

B: assert((B.macB == B.macCA));
M: assert((M.macM == M.macCAm
X: gassert((M.RNb == B.RNb));
X: gassert((M.RNm == B.RNm));
X: gassert((B.sk == M.sk)); --in sec. pro.

The first assumption different than all evaluations discussed earlier is (line 7, Appendix H):

X: assume(global.decrypt(B.Kb+,B.Kb-));

Where there is a relation such that decryption under *Kb*+ inverts the encryption under *Kb*-. This assumption reduces the following predicate to TRUE:

M: M.f1 := dp[ep[B.f1]B.Kb-]M.Kb+];

Here *f1* in *B*'s address space is *MD* function on *RNb, M, CHm and chosen_SKCS* in *B's* address space. As seen above, encrypted *f1* under *Kb*- in *B*'s address space is decrypted under *Kb*+ in *M*'s address space and the result is stored as *f1* in *M*'s address space. As we mentioned earlier we assumed the relation between *Kb*+ and *Kb*- in *B*'s address space. On the other hand, we need a relation between *Kb*+ in *M*'s address space and *Kb*- in *B'*s address space. This is fine since *B* sends both keys to *CA* which is a trusted authority and all the principals get this information in the certificates from *CA*.

Next is the last assumption in specification of solution to Boyd&Mathuria attack that assumes the relation where decryption under *Km+* inverts the encryption under *Km-* (line 8, Appendix H):

X: assume(global.decrypt(M.Km+,M.Km-));

Following predicate is reduced to TRUE with this assumption:

B: B.f2 := dp[ep[M.f2]M.Km-]B.Km+;

where *f2* in *M*'s address space is *MD* function on *RNm, B, and CHb* in *M*'s address space. Encryption of *f2* under *Km-* in *M*'s address space is decrypted under *Km+* in *B*'s address space and the result stored as *f2* in *B*'s address space.

Next two assertions are the only different assertions that are not mentioned earlier (lines 52 and 61, Appendix H):

M: assert((M.f1' == M.f1));
B: assert((B.f2' == B.f2));

With the first ASSERT statement *f1'* and *f1* values in *M*'s address space are compared. Here *f1*' is *MD* function on *RNb, M, CHm,chosen_SKCS* in M's address space and *f1* is decryption of encrypted *B.f1* under *Kb+* in *M*'s address space where *B.f1* is *MD* function on *Rnb, M, CHm, chosen_SKCS* in *B*'s address space.

Second ASSERT statement above checks *f2'* and *f2* values *in B*'s address space where *f2'* is *MD* function on *RNm, B* and *CHb* in *B*'s address space and *f2* is decryption of encrypted *M.f2* under *Km+* in *B*'s address space. Here *M.f2* is MD function on *RNm*, *B*, *and CHb* in *M*'s address space. By comparing *f2* values in *M*'s and *B*'s address spaces *B* is aware of an attack by any intruder.

Finally, after all these evaluations from the last predicate to the first predicate, the evaluation of the solution to Boyd & Mathuria attack on the Secure Protocol of Aziz & Diffie reduces the result as TRUE. This proves that the protocol is not vulnerable to Boyd&Mathuria attack with the solution provided (RNb is signed instead of e[RNb]Km+

in the 2nd and 3rd messages) (see Figure 3.1.8). Because the intruder C has the encrypted RNb value not RNb value, she cannot use RNb in the signature and can easily be determined as an intruder. CPAL-ES evaluation of this solution shows the solution to the Boyd&Mathuria attack and provides a detailed output which clearly shows the communication between the base station and mobile host. This way we can follow each step easily and see how the solution works. Another result of this solution is that it is computationally less expensive since there is less encryption in the 3rd message of the solution than the 3rd message of the original secure protocol.

4.2 CPAL Evaluation of the IEEE 802.1X Protocol

Recall that there are three steps in evaluating the protocols with CPAL: First step is specifying protocol actions, second step is generating the verification conditions and the last step is proving these verification conditions [YAS96].

CPAL evaluation of the IEEE 802.1X protocol starts with the correct implementation of the protocol [Appendix I]. Then, verification conditions are created. All the actions and relations among the participating principals are checked one by one in order to examine the system against possible attacks and find the flaws and solutions to them.

IEEE 802.1X protocol uses shared-key authentication method which is also known as symmetric key cryptography. It is different than the public key cryptography since there is only one key, shared key, used for encryption and decryption while different keys used for encryption and decryption with public key cryptography as we have discussed in the earlier sections. Securing the system with shared key (symmetric key) cryptography is much easier and cheaper than the securing the system with public key cryptography. On the other hand, there is a security problem with it. Since the shared key is needed to be delivered to the recipient for decryption, it could be captured by an adversary. Sharing the shared key between the clients may not have big security problems on a limited scale while it could be a big problem on large scale.

Additionally, MAC-address based access control list and the Wires Equivalent Privacy protocol (WEP) are used to provide confidentiality and security. On the other hand, the system is still not secure. Therefore, Robust Security Network (RSN) is used with IEEE 802.1X protocol. RSN provides such a mechanism that it restricts the network

connectivity at MAC layer to the only authorized users via 802.1X. IEEE 802.1X protocol standard offers an architectural framework on top of authentication methods such as certificate-based authentication, one-time passwords and smartcards. Additionally, it provides port-based network access control. Supplicant, authenticator and access server are the three entities used by RSN to provide security framework as we mentioned earlier. There is a central authority since this is an infrastructure network. Access server authenticates the user via access point. Extensible Authentication Protocol (EAP) which is built on challenge/response communication mechanism is used to permit different variety of authentication mechanisms. On the other hand, they do not have any controlling mechanism for integrity and privacy protection. Remote Authentication Dial-In User Service (RADIUS) protocol provides a mechanism for per packet authenticity and integrity verification [MIS02].

CPAL-ES encoding of the IEEE 802.1X Protocol is given in Appendix I and Appendix J shows the CPAL-ES evaluation of the protocol.

After encoding of IEEE 802.1X is done in CPAL-ES, it is checked for syntax errors easily since CPAL is a parsed language with syntax checker. The final version of our encoding is syntax free. The next step is translating the specification into verification condition. Weakest precondition definitions are applied starting from last statement to the first statement by starting the initial predicate with TRUE value. The definition of encryption, decryption operators, value catenation and extraction and assumptions are applied to the predicate when stated in the protocol specification. During the verification process, CPAL reduces verification for repeated logical conditions with the simple logical reductions called simplification process. Once this process is done, predicate is reduced to its simplified form as seen at the end of Appendix-J. Simplification process is repeated and combined with the encryption. The catenation operation searches until all of the matches are found and the predicate is reduced to its most simplified form.

The final predicate is the verification condition for the protocol. If it can be proved from the assumptions, that the final predicate is TRUE, then, protocol is guaranteed to meet its specified goals. Appendix J has the details of this verification and proving process. As it is clearly seen in Appendix J, the final predicate of CPAL-ES verification of IEEE 802.1X comes TRUE at the end of the verification since all the predicates for ASSUME and ASSERT statements reduced to TRUE. ASSSUME statement is used

when there are ASSUMPTIONS needed to prove the predicates to TRUE. In order to specify protocol goals to be proven, ASSERT statement is used to express beliefs of principals in CPAL. GASSERT is used when comparison of values in different principal's address space is necessary. ASSERT and ASSUME statements are necessary in order to prove the protocol correctly. More information about these could be found in the previous chapter and A.Yasinsac [YAS96].

There are not many assumptions used for the specification of IEEE 802.1X protocol. Whereas, number of ASSERT and GASSERT statements used in order to prove the predicates. They are not all necessary in order to run the protocol without any error. We just wanted to check all the possible predicates to make the final predicate TRUE.

Since the shared key cryptography is used for authenticating the IEEE 802.1X protocol, the protocol specification is not complicated. The same shared key is used for both encryption and decryption. The protocol specifications of IEEE 802.1X protocol starts with the ASSUME statements. There are only two of them used for 802.1X protocol (Appendix J, line 1-2):

```
X: assume((S.kas == A.kas));
X: assume((R.secret == A.secret));
```

With the first assumption, *kas* in supplicants address space and *kas* in access point's address space are compared where *kas* is a shared key between the supplicant and access point. The second assumption compares the value *secret* in access point's address space and RADIUS's address space. The shared *secret* value is shared between the access point and RADIUS server. It is used for the calculation of *User-Password* where the request authenticator and the *secret* value put through a one way *MD5* hash function for creating a 16 octet digest value and the result is xored with the password comes from user and used as the *User-Password* attribute of the *Access-Request* message. With these assumptions we assume that both the supplicant and access point has the same *kas*, shared key, value and both the RADIUS and access point has the same secret value.

The list of acronyms used to define the IEEE 802.1X Protocol in CPAL_ES as follows:

Table 4.2.1 List of acronyms for the Secure Protocol in CPAL

S: Supplicant (user, client)	**passw:** password entered by the user
A: Access Point (authenticator)	**r:** request / respond
R: RADIUS (Remote Authentication Dial-In User Service) server	**s:** success
I: intruder (adversary, attacker)	**rs_id:** Identifier field for EAP-r message
Kxv: Shared key between principals X and Y	**rs_code:** Code field for EAP-r message
secret: Shared secret between A & R. Used to calculate u_passw	**rs_leng:** Length field for EAP-r message
e[<X.A>]X.K: Encryption of A in X's address space, under key K in X's address space	**r_type:** Type field for EAP-r message
d[X.msg]X.K: Decryption of msg in X's address space under key K in X's address space	**r_type_data:** Type-Data filed for EAP-r message
MD5(Z): The message digest function on contents Z	**rcs:** request / challenge / success(accept)
RN: Random Number	**RADrcs_id:** Identifier field for RAD-rc message
CH: Challenge Value (128 bit random number)	**RADrcs_code:** Code field for RAD-rc message
RADAccReq: RADIUS Access-Request Message	**RADrcs_leng:** Length field for RAD-rc message
RADAccChal: decrypt public (public/private key decryption of public key protocol).	**RADrcs_auth:** authenticator field for RAD-rcs message
RADaccept: RADIUS-Accept message	**RADrcs_attr:** attributes filed for RAD-rcs message
EAPreqID: EAP-Request message asking for Id	**u_name:** user_name field for RADrcs-attr field
Id: client's identification	**u_paswd:** user_password field for RADrcs-attr field
EAPrespID: EAP-Response msg responding with Id	**client_id:** user-identifier field for RADrcs-attr field
EAPsucc: EAP-Success message	**port_id:** port number field for RADrcs-attr field
XOR(M,N): xored value of M and N	**msg:** message

First, the protocol starts with message, which has the encrypted passw value under *kas* in supplicant address space, sent from supplicant to access point. Upon receipt of the *msg*, *A* decrypts it. The *passw* value *A* has from decryption and the *passwd* value *S* has in its address space are compared with the following GASSERT statement to see whether or not they are the same values (Appendix J, line 6):

X: gassert((A.passw == S.passw));

Then, *A* creates *EAPreqID* in her own address space and sends it to *S*. When *S* receives the message first she parses the message into individual variables. After this, each variable in *S*'s address space and *A*'s address space are compared with the following statements to find if there is an attacker who changed any of the values sent (Appendix J, lines 11-16):

```
X: gassert((A.EAPreqID == S.EAPreqID));
X: gassert((A.req_code == S.req_code));
X: gassert((A.req_id == S.req_id));
X: gassert((A.req_leng == S.req_leng));
X: gassert((A.req_type == S.req_type));
X: gassert((A.req_type_data == S.req_type_data));
```

Next, *S* sets the identifier and type fields of the respond message that she is planning to send same as those that come from *A* with the request message and she creates the response message and sent it to *A*. Upon receipt of the message, *A* parses the packet into individual variables, stores in her address space and compares each one of them with the ones in *S*'s address space against to attacks. This process can be seen from the following statements (Appendix J, lines 23-30):

```
X: gassert((S.EAPrespID == A.EAPrespID));
X: gassert((S.res_code == A.res_code));
X: gassert((S.res_id == A.res_id));
X: gassert((S.res_leng == A.res_leng));
X: gassert((S.res_type == A.res_type));
X: gassert((S.res_type_data == A.res_type_data));
A: assert((A.res_id == A.req_id));
A: assert((A.res_type == A.req_type));
```

In the next step, *A* calculates the user password. First, she generates the random number *RN1*, creates the *MD5* message digest function on *RN1* and stores the result as

RADreg1_auth which is going to be used as request authenticator field for the next request message. Then another *MD5* function is created on *RADreg1_auth* and *secret* values and it is xored with the *passw* value and the result is stored as *u_passw1* in *A*'s address space to use as user password attribute in the next request message. Then the attributes field of the request message is created by including the necessary attributes along with the EAP message *A* receives from *S* in the previous response message and *MD5* function on EAP message is also attached for security reasons. After all these, *A* sets the *RADAccReq1* message including the *authenticator* and *attributes* fields we have just mentioned and sends the request message to *R*. When *R* receives the message, first he parses the whole packet and then the attributes field. Next *R* calculates a new *MD* function on *EAPrespID* it received with the package in the attributes field and then compares this value with the one he received in the attributes field. Having different *MD5* values mean of an attack. *R* also compares each individual value comes into his address space and the ones in *A*'s address space. Again having unmatched value means of an attack. Following statements are used for this purpose (Appendix J, lines 43-56):

```
X: gassert((A.RADAccReq1 == R.RADAccReq1));
X: gassert((A.RADreq_code == R.RADreq_code));
X: gassert((A.RADreq1_id == R.RADreq1_id));
X: gassert((A.RADreq1_leng == R.RADreq1_leng));
X: gassert((A.RADreq1_auth == R.RADreq1_auth));
X: gassert((A.RADreq1_attr == R.RADreq1_attr));
X: gassert((A.u_name == R.u_name));
X: gassert((A.CT_u_passw == R.CT_u_passw));
X: gassert((A.client_id == R.client_id));
X: gassert((A.port_id == R.port_id));
X: gassert((A.EAPrespID == R.EAPrespID));
X: gassert((S.EAPrespID == R.EAPrespID));
X: gassert((A.CT_EAPrespID == R.CT_EAPrespID));
R: assert((R.CT_EAPrespID' == R.CT_EAPrespID));
```

Additionally *EAPrespID* in *R*'s address space and *S*'s address space are also checked to see whether or not it is the same *EAPrespID* message *S* sent to *R* through *A.*

Next step is creating a new message as response to a request comes from *A*. First *R* calculates an *MD5* value on the request message he sent previously to *A* (*RADAccReq1*) and the secret value in *R*'s address space and stores the result as authenticator field for the response message (*RADResp1_auth*). Then he creates a new EAP request message (*EAPreq1*) and *MD* function on it, and then the attributes field for response message, which is going to be a challenge including the *EAPreq1* and *MD5* on it. After that, he sets the identifier field of the challenge message same as the identifier field of the previous request message comes from *A (RADreq1).* Then he creates the challenge message *(RADAccChal)* by including the *RADResp1_auth* and *RADchall-attr* fields and sent it to *A*.

When *A* creates the challenge message, again she parses the package into individual variables and creates her own authenticator value by calculating a new *MD5* function on *RADAccReq1* and secret values and also calculates another *MD5* on *EAPReq1* since she has these values earlier. Then compares the results with the ones come with the challenge message in order the check whether or not they are changed by an adversary. Additionally, all the other variables came into *A*'s address space are compared with the ones in *R*'s address space with the following GASSERT statements (Appendix J, lines 70-84):

```
X: gassert((R.RADAccChal == A.RADAccChal));
X: gassert((R.RADchal_code == A.RADchal_code));
X: gassert((R.RADchal_id == A.RADchal_id));
X: gassert((R.RADchal_leng == A.RADchal_leng));
X: gassert((R.RADresp1_auth == A.RADresp1_auth));
X: gassert((R.RADchal_attr == A.RADchal_attr));
X: gassert((R.u_name == A.u_name));
X: gassert((R.CT_u_passw == A.CT_u_passw));
X: gassert((R.client_id == A.client_id));
X: gassert((R.port_id == A.port_id));
X: gassert((R.EAPreq1 == A.EAPreq1));
```

X: gassert((R.CT_EAPreq1 == A.CT_EAPreq1));
A: assert((A.RADreq1_id == A.RADchal_id));
A: assert((A.RADresp1_auth == A.RADresp1_auth'));
A: assert((A.CT_EAPreq1 == A.CT_EAPreq1'));

A takes the *EAPreq1* field that comes with the challenge message from *R* and sends it to *S* as next message. Upon receipt of the *EAPReq1* message, *S* parses it into individual pieces again and compares each value in her address space with the ones in *A*'s address space by using the following statements (Appendix J, lines 88-93):

X: gassert((A.EAPreq1 == S.EAPreq1));
X: gassert((A.req_code == S.req_code));
X: gassert((A.req1_id == S.req1_id));
X: gassert((A.req1_leng == S.req1_leng));
X: gassert((A.req1_type == S.req1_type));
X: gassert((A.req1_type_data == S.req1_type_data));

Then S sets the identifier and type fields for the next response message same as the ones that come with the request message, creates the *EAPresp1* message and sends it to *A*. When *A* receives this message again she parses the packet and compares each value in her address space with the ones in *S*'s address space as seen on the following statements (Appendix J, lines 100-107):

X: gassert((S.EAPresp1 == A.EAPresp1));
X: gassert((S.res_code == A.res_code));
X: gassert((S.res1_id == A.res1_id));
X: gassert((S.res1_leng == A.res1_leng));
X: gassert((S.res1_type == A.res1_type));
X: gassert((S.res1_type_data == A.res1_type_data));
A: assert((A.res1_id == A.req1_id));
A: assert((A.res1_type == A.req1_type));

Next, *A* starts setting another request message to sent *R*. First, she generates a new random number (*RN2*), calculates *MD5* function on this value and stores it as *RADreq2_auth* (authenticator field for second request message). *A* calculates another *MD5* message digest function on *RADreq2_auth* and the *secret* value she has. Then it xores this *MD5* function and user's password and stores the result as a new *password,u_passw2,* for *User-Password* attribute in the packet and calculates an *MD5* message digest function on *u_passw2* and another *MD5* on *EAPresp1*. Next, she sets the attributes field by including the necessary fields along with *EAPresp1* and *MD5* function on *EAPresp1*.

The next step is creating the *RADAccReq2* message including the authenticator, attributes field and all the others. Once it is ready, *A* sends the second request message to *R*. Upon receipt of the message, first *R* parses everything in the package and the attributes field. Next, he calculates a new *MD5* function on *EAPresp1* in his own address space and compares this one with *EAPresp1* in *A*'s address space. *R* checks all the values in his address space with the ones in *A*'s address space. *EAPresp1* in *R*'s address space is also compared with the *EAPres1* in *S*'s address space. Once all the following GASSERT statements (Appendix J, lines 120-133) are proved than *R* gets ready for the next response message:

```
X: gassert((A.RADAccReq2 == R.RADAccReq2));
X: gassert((A.RADreq_code == R.RADreq_code));
X: gassert((A.RADreq2_id == R.RADreq2_id));
X: gassert((A.RADreq2_leng == R.RADreq2_leng));
X: gassert((A.RADreq2_auth == R.RADreq2_auth));
X: gassert((A.RADreq2_attr == R.RADreq2_attr));
X: gassert((A.u_name == R.u_name));
X: gassert((A.CT_u_passw2 == R.CT_u_passw2));
X: gassert((A.client_id == R.client_id));
X: gassert((A.port_id == R.port_id));
X: gassert((A.EAPresp1 == R.EAPresp1));
X: gassert((S.EAPresp1 == R.EAPresp1));
X: gassert((A.CT_EAPresp1 == R.CT_EAPresp1));
```

R: assert((R.CT_EAPresp1' == R.CT_EAPresp1));

If *R* needs more information from *S* through *A* in order to authenticate *S*, then above communication between *S*, *A* and *R* repeated. Once *S* is accepted by *R*, then *R* needs to send the accept message. First, *R* calculates *MD5* message digest function on *RADAccReq2* and *secret* value in *R*'s address space. Then creates the *EAPsucc* message and *MD* function on it and sets the attributes field by including *EAPSucc* and *MD* function on *EAPSucc*. Next, he sets the identifier field of *RADsucc* message and identifier filed of *RADreq2* message. Finally sets the *RADaccept* message including all the necessary fields along with the *RADsucc-auth* and *RADsucc_attr* and sends it to *A*. Once *A* receives the message, she parses to individual pieces. Then she calculates another *MD5* digest function on *RADAccReq2* and *secret* values in her address space and *MD* message digest function on it. She calculates another *MD5* on *EAPSucc* in *A'*s address space. At the end of all these, *A* checks each individual variable came into her address book, with the ones are in *R*'s address space as seen o the following statements (Appendix J, lines 147-158):

X: gassert((R.RADaccept == A.RADaccept));
X: gassert((R.RADsucc_code == A.RADsucc_code));
X: gassert((R.RADsucc_id == A.RADsucc_id));
X: gassert((R.RADsucc_leng == A.RADsucc_leng));
X: gassert((R.RADsucc_auth == A.RADsucc_auth));
X: gassert((R.RADsucc_attr == A.RADsucc_attr));
X: gassert((R.u_name == A.u_name));
X: gassert((R.CT_u_passw2 == A.CT_u_passw2));
X: gassert((R.client_id == A.client_id));
X: gassert((R.port_id == A.port_id));
X: gassert((R.EAPsucc == A.EAPsucc));
X: gassert((R.CT_EAPsucc == A.CT_EAPsucc));

Additionally, A checks whether or not *RADreq2_id* and *RADsuccID*, *RADsuccc_auth* come with the message and *RADsuccc_auth'* (calculated in *A*'s

address space), *MD5* on *AEPsucc* comes to *A*'s address space with the message and *EAPsucc'* calculated in *A*'s address space are matches. The ASSERT statements used for this purpose is below (Appendix J, lines 159-161):

```
A: assert((A.RADreq2_id == A.RADsucc_id));
A: assert((A.RADsucc_auth == A.RADsucc_auth'));
A: assert((A.CT_EAPsucc == A.CT_EAPsucc'));
```

Finally, *A* takes the *EAPsucc* message that comes with the attributes field for the *RADAccept* message, and sends it to the *S* as last message. When *S* receives it, first she parses message into individual items as usual, and compares each item that comes into her address from *A* with the ones in *A*'s address space as it can be seen from the following GASSERT statements (Appendix J, lines-165-169):

```
X: gassert((A.EAPsucc == S.EAPsucc));
X: gassert((R.EAPsucc == S.EAPsucc));
X: gassert((A.succ_code == S.succ_code));
X: gassert((A.succ_id == S.succ_id));
X: gassert((A.succ_leng == S.succ_leng));
```

At the end of the evaluation of IEEE 802.1X protocol, the result reduces to TRUE. This means that the CPAL-ES evaluation of IEEE 802.1X protocol works but does not guarantee that it is not vulnerable to possible attacks. CPAL-ES captures the essence of the protocol and helps the analyzer understand the protocol in great detail. It shows how the attacks work in detail in the following sections.

4.2.1 CPAL Evaluation of the Man In the Middle (MIM) Attack on the IEEE 802.1X Protocol

CPAL-ES encoding of the MIM attack on the IEEE 802.1X Protocol is given in Appendix K. Appendix L shows the CPAL-ES analysis of the attack. Since we have discussed the attack in detail earlier, here we will only mention the CPAL-ES analysis of

the attack. CPAL encoding of MIM attack is almost the same as CPAL encoding the IEEE.1X Protocol except the last message sent from *A* to *S*. Hence, we will not discuss the other messages here again.

Recall that ASSERT and ASSUME statements are very important during the analyzing process since ASSERT statements are the goals of the predicate, and ASSUME statements are the predicates verified as the truth of the goals. Therefore, we will talk about the ASSUME and ASSERT statements during the evaluation process. Again there are only two assumptions needed in order to run the program on MIM protocol attack. These are the same assumptions we mentioned earlier for evolution of the IEEE 802.1X protocol.

The last message sent from *A* to *S* could be eavesdropped by an adversary since there is no integrity (Appendix L). Intruder *I* forges the message on behalf of the authenticator (access point) instead of *S* receiving it and gets all the traffic from *S*. He gets information from incepted message into his address space by parsing the message. *I* checks the messages to see whether or not he received them from *A* or *R*'s address spaces with the following GASSERT statements (Appendix L, lines 140-144):

```
X: gassert((A.EAPsucc == I.EAPsucc));
X: gassert((R.EAPsucc == I.EAPsucc));
X: gassert((A.succ_code == I.succ_code));
X: gassert((A.succ_id == I.succ_id));
X: gassert((A.succ_leng == I.succ_leng));
```

Then *I* forward the message to *S*, and when *S* receives it she does not know that it is coming from an attacker since she trusts *I*, and thinks that he is *the authenticator* (access point). Then *S* parses the message into single variables and checks them.

The CPAL-ES evaluation of MIM attack on IEEE 802.1X protocol reduces the result of TRUE which means that MIM attack works. CPAL-ES evaluation of MIM attack shows how the attack works as seen in Appendix L, lines 137-152. Access point A sends the EAP success (EAPsucc) message to the supplicant S after he receives the RADIUS access accept message (RADaccept) from authentication server RADIUS R. EAPsucc message could be forged by an adversary due to lack of two-way

authentication (there is only one-way authentication of supplicant S to the access point A). The attacker acts as an access point to the supplicant S and supplicant to the access point A. This result proves that IEEE 802.1X protocol is not secure and vulnerable to MIM attack. Next section provides a solution to this attack.

4.2.2 CPAL Evaluation of the solution to the Man-in-the-Middle (MIM) Attack on the IEEE 802.1X Protocol

CPAL encoding of the solution to the MIM attack is shown in Appendix M and Appendix N shows the evaluation of the solution on the MIM attack.

As a solution to MIM attack on IEEE 802.1X protocol, the last message sent from A to S could be changed as following (Appendix N, lines 137-142):

```
A: A.succmsg := <A.EAPsucc,f.MD5(A.EAPsucc)>;
A: => S(A.succmsg);
S: <-(S.succmsg);
S: (S.EAPsucc,S.CT_forsucc) := S.succmsg;
S: (S.succ_code,S.succ_id,S.succ_leng) := S.EAPsucc;
S: S.succmsg' := <S.EAPsucc,f.MD5(S.EAPsucc)>;
```

where *MD5* message digest function on *EAPsucc* message is sent along with *EAPsucc* message. If an adversary intercepts the message and changes *EAPsucc*, *S* can find this out easily by getting a new *MD5* function on *EAPsucc* she receives and comparing the result with *MD5* value she received from the message. This checking process is shown on the following GASSERT statement (Appendix N, line 148):

```
S: assert((S.succmsg' == S.succmsg));
```

All the other parts of the encoding and evaluation of solution to MIM attack on the IEEE 802.1X protocol is the same as the encoding and evaluation of the IEEE 802.1X protocol. Therefore, we will not discuss them again here.

Finally, the evaluation of solution to MIM attack on the IEEE 802.1X protocol reduces to TRUE. This verifies that the solution to MIM attack works. CPAL-ES evaluation of this solution is described in Appendix N lines 137-142. Since the attack has occurred due to lack of authentication, the success message (EAPsucc) sent from access point A to supplicant S is authenticated with an MD5 message digest function. This way, no one will be able to forge the message. As a result, even IEEE 802.1X protocol is vulnerable to attacks; there are some solutions that could reduce the vulnerability. On the other hand none of these solutions will make the IEEE 802.1X protocol hundred percent secure because of the nature of the wireless communications.

CHAPTER 5

CONCLUSIONS

Wireless security became a very important issue after the extensive usage of wireless communication tools in these days. Cryptographic protocols, used for the communication between the principals, are analyzed with different techniques to check the system against vulnerabilities, find flaws and propose solutions to flaws, understand the protocols deeply to increase the efficiency and prove the correctness. One of the most efficient cryptographic protocol analysis techniques is formal methods that uses mathematical techniques. In chapter-2, we have an overview of a few techniques designed with formal methods. Each method uses different proving techniques. There are not many wireless protocol tested with formal methods.

In this work, we have analyzed two different wireless communication protocols with CPAL_ES, a tool used to analyze protocols with formal methods: The Secure Protocol of Aziz & Diffie and 802.1X protocol. The Secure Protocol has very complex communication features due to the use of public key cryptography. It has a very long output of WP predicate and specification process that is computationally expensive. 802.1X is the most common, widely used protocol with today's technology ((e.g. 802.11a and 802.11b used for the wireless LANs effective in the short range communications). CPAL-ES evaluation of 802.1X protocol has much shorter output (see Appendix-J) than The Secure Protocol's evaluation. We have also analyzed the known attacks on these protocols and solutions to them. Our evaluation of these attacks confirmed that even though 802.1X protocol is useful especially for wireless LAN communications, it is not secure. The Secure Protocol is also vulnerable to attacks but has strong confidentiality because of the public key cryptography future of the design.

Our analysis of these protocols with CPAL-ES assists us to understand them in great detail and test the capabilities of CPAL-ES especially on wireless protocols. Because of the wireless nature of the communication, a principle could send a message

to more than one principle. In this case, only one “send” operator should send a message to more than one principle at the same time. Unfortunately, this is not possible with the current design of CPAL-ES. As Marshall mentioned, CPAL_ES needs extensions for testing in wireless broadcast environment [MAR03]. There are only a few wireless protocols analyzed with CPAL-ES [MAR03]. More of the wireless protocols could be analyzed with CPAL-ES. The protocols we have analyzed are not too long in communication process. Protocols such as WAP (Wireless Application Protocol) could be analyzed with CPAL-ES in the future after CPAL-ES is extended for the evaluations in wireless environment.

Though CPAL-ES is not designed for wireless communications, we were able to analyze two wireless protocols with CPAL-ES and prove some known attacks. We were not able to find unknown attacks on these protocols. Nevertheless, we showed in great detail the analysis of two wireless protocols that increases the credibility of these protocols and may contribute some other research on this field since there is not much work done on analysis of wireless protocols with formal methods in the literature.

APPENDIX A

CPAL-ES ENCODING OF THE SECURE PROTOCOL

```
--Initial assumptions
X: assume (global.decrypt(B.Kca+, CA.Kca-));
X: assume (global.decrypt(M.Kca+, CA.Kca-));
X: assume (B.Kbca == CA.Kbca);
X: assume (M.Kmca == CA.Kmca);
X: assume (global.decrypt(B.Kb-, B.Kb+));
X: assume (global.decrypt(M.Km-, M.Km+));
X: assume (global.decrypt(M.Km+, M.Km-));

--Message#0B: Base --> Certification Authority
B: => CA (e[<Kb+,Kb->]Kbca);
CA: <- (msgb);
CA: Kb := d[msgb]Kbca;
CA: (Kb+, Kb-) := Kb;

--CA generates the certificate for B
CA: CertB_contents := <sn1, vp1, B, Kb+, CA>;
CA: macCA := MD(CertB_contents);
CA: Cert_B := < CertB_contents,ep[macCA]Kca- >;

--Message#0CA1: Certification Authority --> Base
CA: => B (Cert_B);
B: <- (Cert_B);
B: (CertB_contents,CTforMD) := Cert_B;
B: (sn1, vp1, B, Kb+, CA):=CertB_contents;
B: macCA := dp[CTforMD]Kca+;
B: macB := MD(CertB_contents);
B: assert (macCA == macB);

--Message#0M: Mobile --> Certification Authority
M: => CA (e[<Km+, Km->]Kmca);
CA: <- (msgm);
CA: Km := d[msgm]Kmca;
CA: (Km+, Km-) := Km;

--CA generates the certificate for M.
CA: CertM_contents := <sn2, vp2, M, Km+, CA>;
CA: macCAm := MD(CertM_contents);
CA: Cert_M := < CertM_contents,ep[macCAm]Kca- >;

--Message#0CA2: Certification Authority --> Mobile
CA: => M (Cert_M);
M: <- (Cert_M) ;
M: (CertM_contents,CTforMDm) := Cert_M;
M: (sn2, vp2, M, Km+, CA) := CertM_contents;
M: macCAm := dp[CTforMDm]Kca+;
M: macM := MD(CertM_contents);
M: assert (macCAm == macM);
```

```
--Message#1: Mobile --> Base
M: => B (<Cert_M, CH1, List_of_SKCSs>);
B: <- (msg1);
B: (Cert_M, CH1, List_of_SKCSs) := msg1;
B: (CertM_contents, CTforMDm) := Cert_M;
B: (sn2, vp2, M, Km+, CA) := CertM_contents;

--Message#2: Base --> Mobile
B: f1 := MD(ep[RN1]Km+, chosen_SKCS, CH1, List_of_SKCSs);
B: msg2 := <Cert_B, ep[RN1]Km+, chosen_SKCS, ep[f1]Kb- >;
B: => M (msg2);
M: <- (msg2);
M: (Cert_B, CTforRN1, chosen_SKCS, CTforf1) := msg2;
M: (CertB_contents,CTforMDb) := Cert_B;
M: (sn1, vp1, B, Kb+, CA) := CertB_contents;
M: RN1 := dp[CTforRN1]Km-;
M: f1 := dp[CTforf1]Kb+;

--Message#3: Mobile --> Base
M: sk := XOR(RN1, RN2);
M: f2 := MD(ep[RN2]Kb+, ep[RN1]Km+);
M: => B(<ep[RN2]Kb+, ep[f2]Km->);
B: <- (msg3);
B: (CTforRN2, CTforMD) := msg3;
B: RN2 := dp[CTforRN2]Kb-;
B: f2 := dp[CTforMD]Km+;
B: f3 := MD(ep[RN2]Kb+, ep[RN1]Km+);
B: assert (f2 == f3);
X: gassert (B.RN1 == M.RN1);
X: gassert (B.RN2 == M.RN2);
B: sk := XOR(RN1,RN2);
X: gassert (M.sk == B.sk);
```

APPENDIX B

CPAL-ES EVALUATION OF THE SECURE PROTOCOL

```
1. X: assume(global.decrypt(B.Kca+,CA.Kca-));
2. X: assume(global.decrypt(M.Kca+,CA.Kca-));
3. X: assume((CA.Kbca == B.Kbca));
4. X: assume((CA.Kmca == M.Kmca));
5. X: assume(global.decrypt(B.Kb-,B.Kb+));
6. X: assume(global.decrypt(M.Km-,M.Km+));
7. X: assume(global.decrypt(M.Km+,M.Km-));
8. B: => CA(e[<B.Kb+,B.Kb->]B.Kbca);
9. CA: <-(CA.msgb);
10.CA: CA.Kb := d[CA.msgb]CA.Kbca;
11.CA: (CA.Kb+,CA.Kb-) := CA.Kb;
12.CA: CA.CertB_contents := <CA.sn1,CA.vp1,CA.B,CA.Kb+,CA.CA>;
13.CA: CA.macCA := f.MD(CA.CertB_contents);
14.CA: CA.Cert_B := <CA.CertB_contents,ep[CA.macCA]CA.Kca->;
15.CA: => B(CA.Cert_B);
16.B: <-(B.Cert_B);
17.B: (B.CertB_contents,B.CTforMD) := B.Cert_B;
18.B: (B.sn1,B.vp1,B.B,B.Kb+,B.CA) := B.CertB_contents;
19.B: B.macCA := dp[B.CTforMD]B.Kca+;
20.B: B.macB := f.MD(B.CertB_contents);
21.B: assert((B.macB == B.macCA));
22.M: => CA(e[<M.Km+,M.Km->]M.Kmca);
23.CA: <-(CA.msgm);
24.CA: CA.Km := d[CA.msgm]CA.Kmca;
25.CA: (CA.Km+,CA.Km-) := CA.Km;
26.CA: CA.CertM_contents := <CA.sn2,CA.vp2,CA.M,CA.Km+,CA.CA>;
27.CA: CA.macCAm := f.MD(CA.CertM_contents);
28.CA: CA.Cert_M := <CA.CertM_contents,ep[CA.macCAm]CA.Kca->;
29.CA: => M(CA.Cert_M);
30.M: <-(M.Cert_M);
31.M: (M.CertM_contents,M.CTforMDm) := M.Cert_M;
32.M: (M.sn2,M.vp2,M.M,M.Km+,M.CA) := M.CertM_contents;
33.M: M.macCAm := dp[M.CTforMDm]M.Kca+;
34.M: M.macM := f.MD(M.CertM_contents);
35.M: assert((M.macM == M.macCAm));
36.M: => B(<M.Cert_M,M.CH1,M.List_of_SKCSs>);
37.B: <-(B.msg1);
38.B: (B.Cert_M,B.CH1,B.List_of_SKCSs) := B.msg1;
39.B: (B.CertM_contents,B.CTforMDm) := B.Cert_M;
40.B: (B.sn2,B.vp2,B.M,B.Km+,B.CA) := B.CertM_contents;
41.B: B.f1 := f.MD(ep[B.RN1]B.Km+,B.chosen_SKCS,B.CH1,B.List_of_SKCSs);
42.B: B.msg2 := <B.Cert_B,ep[B.RN1]B.Km+,B.chosen_SKCS,ep[B.f1]B.Kb->;
43.B: => M(B.msg2);
44.M: <-(M.msg2);
45.M: (M.Cert_B,M.CTforRN1,M.chosen_SKCS,M.CTforf1) := M.msg2;
46.M: (M.CertB_contents,M.CTforMDb) := M.Cert_B;
47.M: (M.sn1,M.vp1,M.B,M.Kb+,M.CA) := M.CertB_contents;
48.M: M.RN1 := dp[M.CTforRN1]M.Km-;
```

```
   49.M: M.f1 := dp[M.CTforf1]M.Kb+;
   50.M: M.sk := f.XOR(M.RN1,M.RN2);
   51.M: M.f2 := f.MD(ep[M.RN2]M.Kb+,ep[M.RN1]M.Km+);
   52.M: => B(<ep[M.RN2]M.Kb+,ep[M.f2]M.Km->);
   53.B: <-(B.msg3);
   54.B: (B.CTforRN2,B.CTforMD) := B.msg3;
   55.B: B.RN2 := dp[B.CTforRN2]B.Kb-;
   56.B: B.f2 := dp[B.CTforMD]B.Km+;
   57.B: B.f3 := f.MD(ep[B.RN2]B.Kb+,ep[B.RN1]B.Km+);
   58.B: assert((B.f3 == B.f2));
   59.X: gassert((M.RN1 == B.RN1));
   60.X: gassert((M.RN2 == B.RN2));
   61.B: B.sk := f.XOR(B.RN1,B.RN2);
   62.X: gassert((B.sk == M.sk));

*** End of Protocol ***

TRUE

****** Simplified predicate follows.

TRUE
```

APPENDIX C

CPAL-ES ENCODING OF MEADOWS ATTACK ON THE SECURE PROTOCOL

```
--Initial assumptions
X: assume (global.decrypt(B.Kca+, CA.Kca-));
X: assume (global.decrypt(M.Kca+, CA.Kca-));
X: assume (global.decrypt(I.Kca+, CA.Kca-));
X: assume (I.Kica == CA.Kica);
X: assume (B.Kbca == CA.Kbca);
X: assume (M.Kmca == CA.Kmca);
X: assume (global.decrypt(B.Kb-, B.Kb+));
X: assume (global.decrypt(B.Kb+, B.Kb-));
X: assume (global.decrypt(M.Km+, M.Km-));
X: assume (global.decrypt(I.Ki+, I.Ki-));
X: assume (global.decrypt(I.Ki-, I.Ki+));

--Message#0I: Intruder --> Certification Authority
I: => CA (e[<Ki+, Ki->]Kica);
CA: <- (msgi);
CA: Ki := d[msgi]Kica;
CA: (Ki+, Ki-) := Ki;

--CA generates the certificate for I
CA: CertI_contents := <sn3, vp3, I, Ki+, CA>;
CA: macCA := MD(CertI_contents);
CA: Cert_I := < CertI_contents, ep[macCA]Kca- >;

--Message#0CA1: Certification Authority --> Base
CA: => I (Cert_I);
I: <- (Cert_I);
I: (CertI_contents, CTforMD) := Cert_I;
I: (sn3, vp3, I, Ki+, CA) := CertI_contents;
I: macCA := dp[CTforMD]Kca+;
I: macI := MD(CertI_contents);
I: assert (macCA == macI);

--Message#0B: Base --> Certification Authority
B: => CA (e[<Kb+, Kb->]Kbca);
CA: <- (msgb);
CA: Kb := d[msgb]Kbca;
CA: (Kb+, Kb-) := Kb;

--CA generates the certificate for B
CA: CertB_contents := <sn1, vp1, B, Kb+, CA>;
CA: macCA := MD(CertB_contents);
CA: Cert_B := < CertB_contents, ep[macCA]Kca- >;

--Message#0CA1: Certification Authority --> Base
CA: => B (Cert_B);
B: <- (Cert_B);
B: (CertB_contents, CTforMD) := Cert_B;
```

```
B: (sn1, vp1, B, Kb+, CA) := CertB_contents;
B: macCA := dp[CTforMD]Kca+;
B: macB := MD(CertB_contents);
B: assert (macCA == macB);

--Message#0M: Mobile --> Certification Authority
M: => CA (e[<Km+, Km->]Kmca);
CA: <- (msgm);
CA: Km := d[msgm]Kmca;
CA: (Km+, Km-) := Km;

--CA generates the certificate for M
CA: CertM_contents:= <sn2, vp2, M, Km+, CA>;
CA: macCAm := MD(CertM_contents);
CA: Cert_M := < CertM_contents, ep[macCAm]Kca- >;

--Message#0CA2: Certification Authority --> Mobile
CA: -> M (Cert_M);
I: <- (Cert_M);
I: => M (Cert_M);
M: <- (Cert_M) ;
M: (CertM_contents, CTforMDm) := Cert_M;
M: (sn2, vp2, M, Km+, CA) := CertM_contents;
M: macCAm := dp[CTforMDm]Kca+;
M: macM := MD(CertM_contents);
M: assert (macCAm == macM);

--Meadows Attack
--message#1
M: -> B (<Cert_M, CHm, List_of_SKCSs>);
I:<- (msg1);
I: (Cert_M, CHm, List_of_SKCSs) := msg1;
I: (CertM_contents,CTforMDm) := Cert_M;
I: (sn2, vp2, M, Km+, CA) := CertM_contents;
I: macCAmi := dp[CTforMDm]Kca+;
I: macMI := MD(CertM_contents);
I: assert (macCAmi == macMI);

--message#1'
I: => B (<Cert_I, CHm, List_of_SKCSs>);
B:<- (msg1);
B: (Cert_I, CHm, List_of_SKCSs) := msg1;
B: (CertI_contents,CTforMDi) := Cert_I;
B: (sn3, vp3, I, Ki+, CA) := CertI_contents;
B: macCAib := dp[CTforMDi]Kca+;
B: macIB := MD(CertI_contents);
B: assert (macCAib == macIB);

--message#2
B: f1:= MD(ep[RNb]Ki+, chosen_SKCS, CHm, List_of_SKCSs);
B: msg2 := <Cert_B, ep[RNb]Ki+, chosen_SKCS, ep[f1]Kb- >;
B: => I (msg2);
I: <- (msg2);
I: (Cert_B, CTforRNb, chosen_SKCS, CTforf1) := msg2;
I: (CertB_contents,CTforMDb) := Cert_B;
I: (sn1, vp1, B, Kb+, CA) := CertB_contents;
I: macCAbi := dp[CTforMDb]Kca+;
I: macBI := MD(CertB_contents);
I: assert (macCAbi == macBI);
I: RNb := dp[CTforRNb]Ki-;
```

```
I: f1 := dp[CTforf1]Kb+;
I: f1' := MD(ep[RNb]Ki+, chosen_SKCS, CHm, List_of_SKCSs);
I: assert(f1 == f1');

--message#2'
I: => M (msg2);
M: <- (msg2);
M: (Cert_B, CTforRNb, chosen_SKCS, CTforf1) := msg2;
M: (CertB_contents,CTforMDb) := Cert_B;
M: (sn1, vp1, B, Kb+, CA) := CertB_contents;
M: macCAbm := dp[CTforMDb]Kca+;
M: macBM := MD(CertB_contents);
M: assert (macCAbm == macBM);
M: f1 := dp[CTforf1]Kb+;
M: f1' := MD(CTforRNb, chosen_SKCS, CHm, List_of_SKCSs);
M:assert (f1 == f1');  --worked. 04.21.03

--message#3
M: f2 := MD(ep[RNm]Kb+, CTforRNb);
M: -> B(<ep[RNm]Kb+, ep[f2]Km->);
I: <- (msg3);
I: (CTforRNm, CTforf2) := msg3;
I: f2 := dp[CTforf2]Km+;
I: f2' := MD(CTforRNm, ep[RNb]Ki+);
I: assert(f2 == f2');

--message#3'
I: msg3':= <CTforRNm, ep[f2']Ki->;
I: => B(msg3');
B: <- (msg3');
B: (CTforRNm, CTforf2') := msg3';
B: RNm := dp[CTforRNm]Kb-;
B: f3 := dp[CTforf2']Ki+;
B: f3' := MD(ep[RNm]Kb+, ep[RNb]Ki+);
B: assert (f3 == f3');
X: gassert (B.RNm == M.RNm);
X: gassert (B.RNb == I.RNb);
B: sk := XOR(RNb, RNm);
```

APPENDIX D

CPAL-ES EVALUATION OF MEADOWS ATTACK ON THE SECURE PROTOCOL

```
1.  X: assume(global.decrypt(B.Kca+,CA.Kca-));
2.  X: assume(global.decrypt(M.Kca+,CA.Kca-));
3.  X: assume(global.decrypt(I.Kca+,CA.Kca-));
4.  X: assume((CA.Kica == I.Kica));
5.  X: assume((CA.Kbca == B.Kbca));
6.  X: assume((CA.Kmca == M.Kmca));
7.  X: assume(global.decrypt(B.Kb-,B.Kb+));
8.  X: assume(global.decrypt(B.Kb+,B.Kb-));
9.  X: assume(global.decrypt(M.Km+,M.Km-));
10. X: assume(global.decrypt(I.Ki+,I.Ki-));
11. X: assume(global.decrypt(I.Ki-,I.Ki+));
12. I: => CA(e[<I.Ki+,I.Ki->]I.Kica);
13. CA: <-(CA.msgi);
14. CA: CA.Ki := d[CA.msgi]CA.Kica;
15. CA: (CA.Ki+,CA.Ki-) := CA.Ki;
16. CA: CA.CertI_contents := <CA.sn3,CA.vp3,CA.I,CA.Ki+,CA.CA>;
17. CA: CA.macCA := f.MD(CA.CertI_contents);
18. CA: CA.Cert_I := <CA.CertI_contents,ep[CA.macCA]CA.Kca->;
19. CA: => I(CA.Cert_I);
20. I: <-(I.Cert_I);
21. I: (I.CertI_contents,I.CTforMD) := I.Cert_I;
22. I: (I.sn3,I.vp3,I.I,I.Ki+,I.CA) := I.CertI_contents;
23. I: I.macCA := dp[I.CTforMD]I.Kca+;
24. I: I.macI := f.MD(I.CertI_contents);
25. I: assert((I.macI == I.macCA));
26. B: => CA(e[<B.Kb+,B.Kb->]B.Kbca);
27. CA: <-(CA.msgb);
28. CA: CA.Kb := d[CA.msgb]CA.Kbca;
29. CA: (CA.Kb+,CA.Kb-) := CA.Kb;
30. CA: CA.CertB_contents := <CA.sn1,CA.vp1,CA.B,CA.Kb+,CA.CA>;
31. CA: CA.macCA := f.MD(CA.CertB_contents);
32. CA: CA.Cert_B := <CA.CertB_contents,ep[CA.macCA]CA.Kca->;
33. CA: => B(CA.Cert_B);
34. B: <-(B.Cert_B);
35. B: (B.CertB_contents,B.CTforMD) := B.Cert_B;
36. B: (B.sn1,B.vp1,B.B,B.Kb+,B.CA) := B.CertB_contents;
37. B: B.macCA := dp[B.CTforMD]B.Kca+;
38. B: B.macB := f.MD(B.CertB_contents);
39. B: assert((B.macB == B.macCA));
40. M: => CA(e[<M.Km+,M.Km->]M.Kmca);
41. CA: <-(CA.msgm);
42. CA: CA.Km := d[CA.msgm]CA.Kmca;
43. CA: (CA.Km+,CA.Km-) := CA.Km;
44. CA: CA.CertM_contents := <CA.sn2,CA.vp2,CA.M,CA.Km+,CA.CA>;
45. CA: CA.macCAm := f.MD(CA.CertM_contents);
46. CA: CA.Cert_M := <CA.CertM_contents,ep[CA.macCAm]CA.Kca->;
47. CA: -> M(CA.Cert_M);
48. I: <-(I.Cert_M);
```

```
49.  I: => M(I.Cert_M);
50.  M: <-(M.Cert_M);
51.  M: (M.CertM_contents,M.CTforMDm) := M.Cert_M;
52.  M: (M.sn2,M.vp2,M.M,M.Km+,M.CA) := M.CertM_contents;
53.  M: M.macCAm := dp[M.CTforMDm]M.Kca+;
54.  M: M.macM := f.MD(M.CertM_contents);
55.  M: assert((M.macM == M.macCAm));
56.  M: -> B(<M.Cert_M,M.CHm,M.List_of_SKCSs>);
57.  I: <-(I.msg1);
58.  I: (I.Cert_M,I.CHm,I.List_of_SKCSs) := I.msg1;
59.  I: (I.CertM_contents,I.CTforMDm) := I.Cert_M;
60.  I: (I.sn2,I.vp2,I.M,I.Km+,I.CA) := I.CertM_contents;
61.  I: I.macCAmi := dp[I.CTforMDm]I.Kca+;
62.  I: I.macMI := f.MD(I.CertM_contents);
63.  I: assert((I.macMI == I.macCAmi));
64.  I: => B(<I.Cert_I,I.CHm,I.List_of_SKCSs>);
65.  B: <-(B.msg1);
66.  B: (B.Cert_I,B.CHm,B.List_of_SKCSs) := B.msg1;
67.  B: (B.CertI_contents,B.CTforMDi) := B.Cert_I;
68.  B: (B.sn3,B.vp3,B.I,B.Ki+,B.CA) := B.CertI_contents;
69.  B: B.macCAib := dp[B.CTforMDi]B.Kca+;
70.  B: B.macIB := f.MD(B.CertI_contents);
71.  B: assert((B.macIB == B.macCAib));
72.  B: B.f1 := f.MD(ep[B.RNb]B.Ki+,B.chosen_SKCS,B.CHm,B.List_of_SKCSs);
73.  B: B.msg2 := <B.Cert_B,ep[B.RNb]B.Ki+,B.chosen_SKCS,ep[B.f1]B.Kb->;
74.  B: => I(B.msg2);
75.  I: <-(I.msg2);
76.  I: (I.Cert_B,I.CTforRNb,I.chosen_SKCS,I.CTforf1) := I.msg2;
77.  I: (I.CertB_contents,I.CTforMDb) := I.Cert_B;
78.  I: (I.sn1,I.vp1,I.B,I.Kb+,I.CA) := I.CertB_contents;
79.  I: I.macCAbi := dp[I.CTforMDb]I.Kca+;
80.  I: I.macBI := f.MD(I.CertB_contents);
81.  I: assert((I.macBI == I.macCAbi));
82.  I: I.RNb := dp[I.CTforRNb]I.Ki-;
83.  I: I.f1 := dp[I.CTforf1]I.Kb+;
84.  I: I.f1' := f.MD(ep[I.RNb]I.Ki+,I.chosen_SKCS,I.CHm,I.List_of_SKCSs);
85.  I: assert((I.f1' == I.f1));
86.  I: => M(I.msg2);
87.  M: <-(M.msg2);
88.  M: (M.Cert_B,M.CTforRNb,M.chosen_SKCS,M.CTforf1) := M.msg2;
89.  M: (M.CertB_contents,M.CTforMDb) := M.Cert_B;
90.  M: (M.sn1,M.vp1,M.B,M.Kb+,M.CA) := M.CertB_contents;
91.  M: M.macCAbm := dp[M.CTforMDb]M.Kca+;
92.  M: M.macBM := f.MD(M.CertB_contents);
93.  M: assert((M.macBM == M.macCAbm));
94.  M: M.f1 := dp[M.CTforf1]M.Kb+;
95.  M: M.f1' := f.MD(M.CTforRNb,M.chosen_SKCS,M.CHm,M.List_of_SKCSs);
96.  M: assert((M.f1' == M.f1));
97.  M: M.f2 := f.MD(ep[M.RNm]M.Kb+,M.CTforRNb);
98.  M: -> B(<ep[M.RNm]M.Kb+,ep[M.f2]M.Km->);
99.  I: <-(I.msg3);
100. I: (I.CTforRNm,I.CTforf2) := I.msg3;
101. I: I.f2 := dp[I.CTforf2]I.Km+;
102. I: I.f2' := f.MD(I.CTforRNm,ep[I.RNb]I.Ki+);
103. I: assert((I.f2' == I.f2));
104. I: I.msg3' := <I.CTforRNm,ep[I.f2']I.Ki->;
105. I: => B(I.msg3');
106. B: <-(B.msg3');
107. B: (B.CTforRNm,B.CTforf2') := B.msg3';
108. B: B.RNm := dp[B.CTforRNm]B.Kb-;
```

```
   109. B: B.f3 := dp[B.CTforf2']B.Ki+;
   110. B: B.f3' := f.MD(ep[B.RNm]B.Kb+,ep[B.RNb]B.Ki+);
   111. B: assert((B.f3' == B.f3));
   112. X: gassert((M.RNm == B.RNm));
   113. X: gassert((I.RNb == B.RNb));
   114. B: B.sk := f.XOR(B.RNb,B.RNm);

 *** End of Protocol ***

TRUE

****** Simplified predicate follows.

TRUE
```

APPENDIX E

CPAL-ES ENCODING OF BOYD & MATHURIA ATTACK ON THE SECURE PROTOCOL

```
--Initial assumptions
X: assume (global.decrypt(B.Kca+, CA.Kca-));
X: assume (global.decrypt(M.Kca+, CA.Kca-));
X: assume (global.decrypt(I.Kca+, CA.Kca-));
X: assume (I.Kica == CA.Kica);
X: assume (B.Kbca == CA.Kbca);
X: assume (M.Kmca == CA.Kmca);
X: assume (global.decrypt(M.Km-, M.Km+));
X: assume (global.decrypt(B.Kb+, B.Kb-));
X: assume (global.decrypt(M.Km+, M.Km-));
X: assume (global.decrypt(I.Ki+, I.Ki-));
X: assume (global.decrypt(I.Ki-, I.Ki+));

--Message#0I: Intruder --> Certification Authority
I: => CA (e[<Ki+, Ki->]Kica);
CA: <- (msgi);
CA: Ki := d[msgi]Kica;
CA: (Ki+, Ki-) := Ki;

--CA generates the certificate for I
CA: CertI_contents := <sn3, vp3, I, Ki+, CA>;
CA: macCA := MD(CertI_contents);
CA: Cert_I := < CertI_contents,ep[macCA]Kca- >;

--Message#0CA1: Certification Authority --> Base
CA: => I (Cert_I);
I: <- (Cert_I);
I: (CertI_contents, CTforMD) := Cert_I;
I: (sn3, vp3, I, Ki+, CA):=CertI_contents;
I: macCA := dp[CTforMD]Kca+;
I: macI := MD(CertI_contents);
I: assert (macCA == macI);

--Message#0B: Base --> Certification Authority
B: => CA (e[<Kb+, Kb->]Kbca);
CA: <- (msgb);
CA: Kb := d[msgb]Kbca;
CA: (Kb+, Kb-) := Kb;

--CA generates the certificate for B
CA: CertB_contents := <sn1, vp1, B, Kb+, CA>;
CA: macCA := MD(CertB_contents);
CA: Cert_B := < CertB_contents, ep[macCA]Kca- >;

--Message#0CA1: Certification Authority --> Base
CA: => B (Cert_B);
```

```
B: <- (Cert_B);
B: (CertB_contents, CTforMD) := Cert_B;
B: (sn1, vp1, B, Kb+, CA) := CertB_contents;
B: macCA := dp[CTforMD]Kca+;
B: macB := MD(CertB_contents);
B: assert (macCA == macB);

--Message#0M: Mobile --> Certification Authority
M: => CA (e[<Km+, Km->]Kmca);
CA: <- (msgm);
CA: Km := d[msgm]Kmca;
CA: (Km+, Km-) := Km;


--CA generates the certificate for M
CA: CertM_contents := <sn2, vp2, M, Km+, CA>;
CA: macCAm := MD(CertM_contents);
CA: Cert_M := < CertM_contents, ep[macCAm]Kca- >;

--Message#0CA2: Certification Authority --> Mobile
CA: -> M (Cert_M);
I: <- (Cert_M);
I: => M (Cert_M);
M: <- (Cert_M) ;
M: (CertM_contents, CTforMDm) := Cert_M;
M: (sn2, vp2, M, Km+, CA) := CertM_contents;
M: macCAm := dp[CTforMDm]Kca+;
M: macM := MD(CertM_contents);
M: assert (macCAm == macM);


--Boyd-Mathuria Attack

--message#1
I: => B (<Cert_M, CHi, List_of_SKCSs>);
B: <- (msg1);
B: (Cert_M, CHi, List_of_SKCSs) := msg1;
B: (CertM_contents, CTforMDm) := Cert_M;
B: (sn2, vp2, M, Km+, CA) := CertM_contents;
B: macCAmb := dp[CTforMDm]Kca+;
B: macMB := MD(CertM_contents);
B: assert (macCAmb == macMB);

--message-2
B: f1 := MD(ep[RNb]Km+, chosen_SKCS, CHi, List_of_SKCSs);
B: msg2 := <Cert_B, ep[RNb]Km+, chosen_SKCS, ep[f1]Kb- >;
B: => I (msg2);
I: <- (msg2);
I: (Cert_B, CTforRNb, chosen_SKCS, CTforf1) := msg2;
I: (CertB_contents,CTforMDb) := Cert_B;
I: (sn1, vp1, B, Kb+, CA) := CertB_contents;
I: macCAbi := dp[CTforMDb]Kca+;
I: macBI := MD(CertB_contents);
I: assert (macCAbi == macBI);
I: f1 := dp[CTforf1]Kb+;
I: f1':= MD(CTforRNb, chosen_SKCS, CHi, List_of_SKCSs);
I: assert(f1 == f1');

--message#1'
M: => I (<Cert_M, CHm, List_of_SKCSs>);
```

```
I: <- (msg1);
I: (Cert_M, CHm, List_of_SKCSs) := msg1;
I: (CertM_contents,CTforMDm) := Cert_M;
I: (sn2, vp2, M, Km+, CA) := CertM_contents;
I: macCAmi := dp[CTforMDm]Kca+;
I: macMI := MD(CertM_contents);
I: assert (macCAmi == macMI);

--message#2'
I: f1 := MD(CTforRNb, chosen_SKCS, CHm, List_of_SKCSs);
I: msg2 := <Cert_I, CTforRNb, chosen_SKCS, ep[f1]Ki- >;
I: => M (msg2);
M: <- (msg2);
M: (Cert_I, CTforRNb, chosen_SKCS, CTforf1) := msg2;
M: (CertI_contents,CTforMDi) := Cert_I;
M: (sn3, vp3, I, Ki+, CA) := CertI_contents;
M: macCAim := dp[CTforMDi]Kca+;
M: macIM := MD(CertI_contents);
M: assert (macCAim == macIM);
M: RNb := dp[CTforRNb]Km-;
M: f1 := dp[CTforf1]Ki+;
M: f1' := MD(ep[RNb]Km+, chosen_SKCS, CHm, List_of_SKCSs);

--message#3'
M: sk := XOR(RNb,RNm);
M: f2 := MD(ep[RNm]Ki+, ep[RNb]Km+);
M: => I(<ep[RNm]Ki+, ep[f2]Km->);
I: <- (msg3);
I: (CTforRNm, CTforf2) := msg3;
I: RNm := dp[CTforRNm]Ki-;
I: f2 := dp[CTforf2]Km+;
I: f2' := MD(ep[RNm]Ki+, CTforRNb);

--message#3
I: => B(<ep[RNm]Ki+, CTforf2>);
B: <- (msg3);
B: (CTforRNm, CTforf2) := msg3;
B: f3 := dp[CTforf2]Km+;
B: f3' := MD(CTforRNm, ep[RNb]Km+);
B: assert (f3 == f3');
X: gassert (B.RNb == M.RNb);
B: sk := XOR(RNb, RNm);
X: gassert (M.sk == B.sk);
```

APPENDIX F

CPAL-ES EVALUATION OF BOYD & MATHURIA ATTACK ON THE SECURE PROTOCOL

```
1.  X: assume(global.decrypt(B.Kca+,CA.Kca-));
2.  X: assume(global.decrypt(M.Kca+,CA.Kca-));
3.  X: assume(global.decrypt(I.Kca+,CA.Kca-));
4.  X: assume((CA.Kica == I.Kica));
5.  X: assume((CA.Kbca == B.Kbca));
6.  X: assume((CA.Kmca == M.Kmca));
7.  X: assume(global.decrypt(M.Km-,M.Km+));
8.  X: assume(global.decrypt(B.Kb+,B.Kb-));
9.  X: assume(global.decrypt(M.Km+,M.Km-));
10. X: assume(global.decrypt(I.Ki+,I.Ki-));
11. X: assume(global.decrypt(I.Ki-,I.Ki+));
12. I: => CA(e[<I.Ki+,I.Ki->]I.Kica);
13. CA: <-(CA.msgi);
14. CA: CA.Ki := d[CA.msgi]CA.Kica;
15. CA: (CA.Ki+,CA.Ki-) := CA.Ki;
16. CA: CA.CertI_contents := <CA.sn3,CA.vp3,CA.I,CA.Ki+,CA.CA>;
17. CA: CA.macCA := f.MD(CA.CertI_contents);
18. CA: CA.Cert_I := <CA.CertI_contents,ep[CA.macCA]CA.Kca->;
19. CA: => I(CA.Cert_I);
20. I: <-(I.Cert_I);
21. I: (I.CertI_contents,I.CTforMD) := I.Cert_I;
22. I: (I.sn3,I.vp3,I.I,I.Ki+,I.CA) := I.CertI_contents;
23. I: I.macCA := dp[I.CTforMD]I.Kca+;
24. I: I.macI := f.MD(I.CertI_contents);
25. I: assert((I.macI == I.macCA));
26. B: => CA(e[<B.Kb+,B.Kb->]B.Kbca);
27. CA: <-(CA.msgb);
28. CA: CA.Kb := d[CA.msgb]CA.Kbca;
29. CA: (CA.Kb+,CA.Kb-) := CA.Kb;
30. CA: CA.CertB_contents := <CA.sn1,CA.vp1,CA.B,CA.Kb+,CA.CA>;
31. CA: CA.macCA := f.MD(CA.CertB_contents);
32. CA: CA.Cert_B := <CA.CertB_contents,ep[CA.macCA]CA.Kca->;
33. CA: => B(CA.Cert_B);
34. B: <-(B.Cert_B);
35. B: (B.CertB_contents,B.CTforMD) := B.Cert_B;
36. B: (B.sn1,B.vp1,B.B,B.Kb+,B.CA) := B.CertB_contents;
37. B: B.macCA := dp[B.CTforMD]B.Kca+;
38. B: B.macB := f.MD(B.CertB_contents);
39. B: assert((B.macB == B.macCA));
40. M: => CA(e[<M.Km+,M.Km->]M.Kmca);
41. CA: <-(CA.msgm);
42. CA: CA.Km := d[CA.msgm]CA.Kmca;
43. CA: (CA.Km+,CA.Km-) := CA.Km;
44. CA: CA.CertM_contents := <CA.sn2,CA.vp2,CA.M,CA.Km+,CA.CA>;
45. CA: CA.macCAm := f.MD(CA.CertM_contents);
46. CA: CA.Cert_M := <CA.CertM_contents,ep[CA.macCAm]CA.Kca->;
47. CA: -> M(CA.Cert_M);
```

```
48.  I: <-(I.Cert_M);
49.  I: => M(I.Cert_M);
50.  M: <-(M.Cert_M);
51.  M: (M.CertM_contents,M.CTforMDm) := M.Cert_M;
52.  M: (M.sn2,M.vp2,M.M,M.Km+,M.CA) := M.CertM_contents;
53.  M: M.macCAm := dp[M.CTforMDm]M.Kca+;
54.  M: M.macM := f.MD(M.CertM_contents);
55.  M: assert((M.macM == M.macCAm));
56.  I: => B(<I.Cert_M,I.CHi,I.List_of_SKCSs>);
57.  B: <-(B.msg1);
58.  B: (B.Cert_M,B.CHi,B.List_of_SKCSs) := B.msg1;
59.  B: (B.CertM_contents,B.CTforMDm) := B.Cert_M;
60.  B: (B.sn2,B.vp2,B.M,B.Km+,B.CA) := B.CertM_contents;
61.  B: B.macCAmb := dp[B.CTforMDm]B.Kca+;
62.  B: B.macMB := f.MD(B.CertM_contents);
63.  B: assert((B.macMB == B.macCAmb));
64.  B: B.f1 := f.MD(ep[B.RNb]B.Km+,B.chosen_SKCS,B.CHi,B.List_of_SKCSs);
65.  B: B.msg2 := <B.Cert_B,ep[B.RNb]B.Km+,B.chosen_SKCS,ep[B.f1]B.Kb->;
66.  B: => I(B.msg2);
67.  I: <-(I.msg2);
68.  I: (I.Cert_B,I.CTforRNb,I.chosen_SKCS,I.CTforf1) := I.msg2;
69.  I: (I.CertB_contents,I.CTforMDb) := I.Cert_B;
70.  I: (I.sn1,I.vp1,I.B,I.Kb+,I.CA) := I.CertB_contents;
71.  I: I.macCAbi := dp[I.CTforMDb]I.Kca+;
72.  I: I.macBI := f.MD(I.CertB_contents);
73.  I: assert((I.macBI == I.macCAbi));
74.  I: I.f1 := dp[I.CTforf1]I.Kb+;
75.  I: I.f1' := f.MD(I.CTforRNb,I.chosen_SKCS,I.CHi,I.List_of_SKCSs);
76.  I: assert((I.f1' == I.f1));
77.  M: => I(<M.Cert_M,M.CHm,M.List_of_SKCSs>);
78.  I: <-(I.msg1);
79.  I: (I.Cert_M,I.CHm,I.List_of_SKCSs) := I.msg1;
80.  I: (I.CertM_contents,I.CTforMDm) := I.Cert_M;
81.  I: (I.sn2,I.vp2,I.M,I.Km+,I.CA) := I.CertM_contents;
82.  I: I.macCAmi := dp[I.CTforMDm]I.Kca+;
83.  I: I.macMI := f.MD(I.CertM_contents);
84.  I: assert((I.macMI == I.macCAmi));
85.  I: I.f1 := f.MD(I.CTforRNb,I.chosen_SKCS,I.CHm,I.List_of_SKCSs);
86.  I: I.msg2 := <I.Cert_I,I.CTforRNb,I.chosen_SKCS,ep[I.f1]I.Ki->;
87.  I: => M(I.msg2);
88.  M: <-(M.msg2);
89.  M: (M.Cert_I,M.CTforRNb,M.chosen_SKCS,M.CTforf1) := M.msg2;
90.  M: (M.CertI_contents,M.CTforMDi) := M.Cert_I;
91.  M: (M.sn3,M.vp3,M.I,M.Ki+,M.CA) := M.CertI_contents;
92.  M: M.macCAim := dp[M.CTforMDi]M.Kca+;
93.  M: M.macIM := f.MD(M.CertI_contents);
94.  M: assert((M.macIM == M.macCAim));
95.  M: M.RNb := dp[M.CTforRNb]M.Km-;
96.  M: M.f1 := dp[M.CTforf1]M.Ki+;
97.  M: M.f1' := f.MD(ep[M.RNb]M.Km+,M.chosen_SKCS,M.CHm,M.List_of_SKCSs);
98.  M: M.sk := f.XOR(M.RNb,M.RNm);
99.  M: M.f2 := f.MD(ep[M.RNm]M.Ki+,ep[M.RNb]M.Km+);
100. M: => I(<ep[M.RNm]M.Ki+,ep[M.f2]M.Km->);
101. I: <-(I.msg3);
102. I: (I.CTforRNm,I.CTforf2) := I.msg3;
103. I: I.RNm := dp[I.CTforRNm]I.Ki-;
104. I: I.f2 := dp[I.CTforf2]I.Km+;
105. I: I.f2' := f.MD(ep[I.RNm]I.Ki+,I.CTforRNb);
106. I: => B(<ep[I.RNm]I.Ki+,I.CTforf2>);
107. B: <-(B.msg3);
```

```
   108. B: (B.CTforRNm,B.CTforf2) := B.msg3;
   109. B: B.f3 := dp[B.CTforf2]B.Km+;
   110. B: B.f3' := f.MD(B.CTforRNm,ep[B.RNb]B.Km+);
   111. B: assert((B.f3' == B.f3));
   112. X: gassert((M.RNb == B.RNb));
   113. B: B.sk := f.XOR(B.RNb,B.RNm);
   114. X: gassert((B.sk == M.sk));

 *** End of Protocol ***

TRUE

****** Simplified predicate follows.

TRUE
```

APPENDIX G

CPAL-ES ENCODING OF SOLUTION TO BOYD & MATHURIA ATTACK ON THE SECURE PROTOCOL

```
--Initial assumptions
X: assume (global.decrypt(B.Kca+, CA.Kca-));
X: assume (global.decrypt(M.Kca+, CA.Kca-));
X: assume (B.Kbca == CA.Kbca);
X: assume (M.Kmca == CA.Kmca);
X: assume (global.decrypt(B.Kb-, B.Kb+));
X: assume (global.decrypt(M.Km-, M.Km+));
X: assume (global.decrypt(B.Kb+, B.Kb-));
X: assume (global.decrypt(M.Km+, M.Km-));

--Message#0B: Base --> Certification Authority
B: => CA (e[<Kb+, Kb->]Kbca);
CA: <- (msgb);
CA: Kb := d[msgb]Kbca;
CA: (Kb+, Kb-) := Kb;

--CA generates the certificate for B
CA: CertB_contents := <sn1, vp1, B, Kb+, CA>;
CA: macCA := MD(CertB_contents);
CA: Cert_B := < CertB_contents, ep[macCA]Kca- >;

--***** Message#0CA1 *****: Certification Authority --> Base
CA: => B (Cert_B);
B: <- (Cert_B);
B: (CertB_contents, CTforMD) := Cert_B;
B: (sn1, vp1, B, Kb+, CA) := CertB_contents;
B: macCA := dp[CTforMD]Kca+;
B: macB := MD(CertB_contents);
B: assert (macCA == macB);

--Message#0M: Mobile --> Certification Authority
M: => CA (e[<Km+, Km->]Kmca);
CA:<- (msgm);
CA: Km := d[msgm]Kmca;
CA: (Km+, Km-) := Km;


--CA generates the certificate for M
CA: CertM_contents := <sn2, vp2, M, Km+, CA>;
CA: macCAm := MD(CertM_contents);
CA: Cert_M := < CertM_contents, ep[macCAm]Kca- >;

--Message#0CA2: Certification Authority --> Mobile
CA: => M (Cert_M);
M: <- (Cert_M) ;
M: (CertM_contents, CTforMDm) := Cert_M;
```

```
M: (sn2, vp2, M, Km+, CA) := CertM_contents;
M: macCAm := dp[CTforMDm]Kca+;
M: macM := MD(CertM_contents);
M: assert (macCAm == macM);

--Solution to Boyd-Mathuria Attack
--message#1

M: => B (<Cert_M, CHm, List_of_SKCSs>);
B: <- (msg1);
B: (Cert_M, CHm, List_of_SKCSs) := msg1;
B: (CertM_contents, CTforMDm) := Cert_M;
B: (sn2, vp2, M, Km+, CA) := CertM_contents;

--message#2
B: f1 := MD(RNb, M, CHm, chosen_SKCS);
B: msg2 := <Cert_B, CHb, ep[RNb]Km+, chosen_SKCS, ep[f1]Kb- >;
B: => M (msg2);
M: <- (msg2);
M: (Cert_B, CHb, CTforRNb, chosen_SKCS, CTforf1) := msg2;
M: (CertB_contents, CTforMDb) := Cert_B;
M: (sn1, vp1, B, Kb+, CA) := CertB_contents;
M: RNb := dp[CTforRNb]Km-;
M: f1 := dp[CTforf1]Kb+;
M: f1' := MD(RNb, M, CHm, chosen_SKCS);
M: assert(f1 == f1');

--message#3
M: sk := XOR(RNb, RNm);
M: f2 := MD(RNm, B, CHb);
M: => B(<ep[RNm]Kb+, ep[f2]Km->);
B: <- (msg3);
B: (CTforRNm, CTforf2) := msg3;
B: RNm := dp[CTforRNm]Kb-;
B: f2 := dp[CTforf2]Km+;
B: f2' := MD(RNm, B, CHb);
B: assert(f2 == f2');
X: gassert (B.RNb == M.RNb);
X: gassert (B.RNm == M.RNm);
B: sk := XOR(RNb, RNm);
X: gassert (M.sk == B.sk);
```

APPENDIX H

CPAL-ES EVALUATION OF SOLUTION TO BOYD & MATHURIA ATTACK ON THE SECURE PROTOCOL

```
1.  X: assume(global.decrypt(B.Kca+,CA.Kca-));
2.  X: assume(global.decrypt(M.Kca+,CA.Kca-));
3.  X: assume((CA.Kbca == B.Kbca));
4.  X: assume((CA.Kmca == M.Kmca));
5.  X: assume(global.decrypt(B.Kb-,B.Kb+));
6.  X: assume(global.decrypt(M.Km-,M.Km+));
7.  X: assume(global.decrypt(B.Kb+,B.Kb-));
8.  X: assume(global.decrypt(M.Km+,M.Km-));
9.  B: => CA(e[<B.Kb+,B.Kb->]B.Kbca);
10. CA: <-(CA.msgb);
11. CA: CA.Kb := d[CA.msgb]CA.Kbca;
12. CA: (CA.Kb+,CA.Kb-) := CA.Kb;
13. CA: CA.CertB_contents := <CA.sn1,CA.vp1,CA.B,CA.Kb+,CA.CA>;
14. CA: CA.macCA := f.MD(CA.CertB_contents);
15. CA: CA.Cert_B := <CA.CertB_contents,ep[CA.macCA]CA.Kca->;
16. CA: => B(CA.Cert_B);
17. B: <-(B.Cert_B);
18. B: (B.CertB_contents,B.CTforMD) := B.Cert_B;
19. B: (B.sn1,B.vp1,B.B,B.Kb+,B.CA) := B.CertB_contents;
20. B: B.macCA := dp[B.CTforMD]B.Kca+;
21. B: B.macB := f.MD(B.CertB_contents);
22. B: assert((B.macB == B.macCA));
23. M: => CA(e[<M.Km+,M.Km->]M.Kmca);
24. CA: <-(CA.msgm);
25. CA: CA.Km := d[CA.msgm]CA.Kmca;
26. CA: (CA.Km+,CA.Km-) := CA.Km;
27. CA: CA.CertM_contents := <CA.sn2,CA.vp2,CA.M,CA.Km+,CA.CA>;
28. CA: CA.macCAm := f.MD(CA.CertM_contents);
29. CA: CA.Cert_M := <CA.CertM_contents,ep[CA.macCAm]CA.Kca->;
30. CA: => M(CA.Cert_M);
31. M: <-(M.Cert_M);
32. M: (M.CertM_contents,M.CTforMDm) := M.Cert_M;
33. M: (M.sn2,M.vp2,M.M,M.Km+,M.CA) := M.CertM_contents;
34. M: M.macCAm := dp[M.CTforMDm]M.Kca+;
35. M: M.macM := f.MD(M.CertM_contents);
36. M: assert((M.macM == M.macCAm));
37. M: => B(<M.Cert_M,M.CHm,M.List_of_SKCSs>);
38. B: <-(B.msg1);
39. B: (B.Cert_M,B.CHm,B.List_of_SKCSs) := B.msg1;
40. B: (B.CertM_contents,B.CTforMDm) := B.Cert_M;
41. B: (B.sn2,B.vp2,B.M,B.Km+,B.CA) := B.CertM_contents;
42. B: B.f1 := f.MD(B.RNb,B.M,B.CHm,B.chosen_SKCS);
43. B: B.msg2 :=
    <B.Cert_B,B.CHb,ep[B.RNb]B.Km+,B.chosen_SKCS,ep[B.f1]B.Kb->;
44. B: => M(B.msg2);
45. M: <-(M.msg2);
46. M: (M.Cert_B,M.CHb,M.CTforRNb,M.chosen_SKCS,M.CTforf1) := M.msg2;
```

```
   47.  M: (M.CertB_contents,M.CTforMDb) := M.Cert_B;
   48.  M: (M.sn1,M.vp1,M.B,M.Kb+,M.CA) := M.CertB_contents;
   49.  M: M.RNb := dp[M.CTforRNb]M.Km-;
   50.  M: M.f1 := dp[M.CTforf1]M.Kb+;
   51.  M: M.f1' := f.MD(M.RNb,M.M,M.CHm,M.chosen_SKCS);
   52.  M: assert((M.f1' == M.f1));
   53.  M: M.sk := f.XOR(M.RNb,M.RNm);
   54.  M: M.f2 := f.MD(M.RNm,M.B,M.CHb);
   55.  M: => B(<ep[M.RNm]M.Kb+,ep[M.f2]M.Km->);
   56.  B: <-(B.msg3);
   57.  B: (B.CTforRNm,B.CTforf2) := B.msg3;
   58.  B: B.RNm := dp[B.CTforRNm]B.Kb-;
   59.  B: B.f2 := dp[B.CTforf2]B.Km+;
   60.  B: B.f2' := f.MD(B.RNm,B.B,B.CHb);
   61.  B: assert((B.f2' == B.f2));
   62.  X: gassert((M.RNb == B.RNb));
   63.  X: gassert((M.RNm == B.RNm));
   64.  B: B.sk := f.XOR(B.RNb,B.RNm);
   65.  X: gassert((B.sk == M.sk));

 *** End of Protocol ***

TRUE

****** Simplified predicate follows.

TRUE
```

APPENDIX I

CPAL-ES ENCODING OF IEEE 802.1X PROTOCOL

```
-- Initial assumptions
X: assume(A.kas == S.kas);
X: assume(A.secret == R.secret);

-- password sharing
S: => A (e[passw]kas);
A: <- (msg);
A: passw := d[msg]kas;
X:gassert (S.passw == A.passw);

--MSG-1: Access Point requests from Supplicant
A: EAPreqID := <req_code, req_id, req_leng, req_type, req_type_data>;
A: => S(EAPreqID);
S: <- (EAPreqID);
S: (req_code, req_id, req_leng, req_type, req_type_data) := EAPreqID;
X: gassert(S.EAPreqID == A.EAPreqID);
X: gassert(S.req_code == A.req_code);
X: gassert(S.req_id == A.req_id);
X: gassert(S.req_leng == A.req_leng);
X: gassert(S.req_type == A.req_type);
X: gassert(S.req_type_data == A.req_type_data);

--MSG-2: Supplicant responds to Access Point
S: res_id := req_id;
S: res_type := req_type;
S: EAPrespID := <res_code, res_id, res_leng, res_type, res_type_data>;
S: => A(EAPrespID);
A: <- (EAPrespID);
A: (res_code, res_id, res_leng, res_type, res_type_data) := EAPrespID;
X: gassert(A.EAPrespID == S.EAPrespID);
X: gassert(A.res_code == S.res_code);
X: gassert(A.res_id == S.res_id);
X: gassert(A.res_leng == S.res_leng);
X: gassert(A.res_type == S.res_type);
X: gassert(A.res_type_data == S.res_type_data);
A: assert(req_id == res_id);
A: assert(req_type == res_type);

--MSG-3: Access Point requests from RADIUS
A: RADreq1_auth := MD5(RN1);
A: CT_temp := MD5(RADreq1_auth, secret);
A: u_passw1 := XOR(CT_temp, passw);
A: CT_EAPrespID := MD5(EAPrespID);
A: CT_u_passw := MD5(u_passw1);
A: RADreq1_attr := <u_name, CT_u_passw, client_id, port_id, EAPrespID,
CT_EAPrespID>;
```

```
A: RADAccReq1 := <RADreq_code, RADreq1_id, RADreq1_leng, RADreq1_auth,
RADreq1_attr >;
A: => R (RADAccReq1);
R: <- (RADAccReq1);
R: (RADreq_code, RADreq1_id, RADreq1_leng, RADreq1_auth, RADreq1_attr) :=
RADAccReq1;
R: (u_name, CT_u_passw, client_id, port_id, EAPrespID, CT_EAPrespID) :=
RADreq1_attr;
R: CT_EAPrespID':= MD5(EAPrespID);
X: gassert(R.RADAccReq1 == A.RADAccReq1);
X: gassert(R.RADreq_code == A.RADreq_code);
X: gassert(R.RADreq1_id == A.RADreq1_id);
X: gassert(R.RADreq1_leng == A.RADreq1_leng);
X: gassert(R.RADreq1_auth == A.RADreq1_auth);
X: gassert(R.RADreq1_attr == A.RADreq1_attr);
X: gassert(R.u_name == A.u_name);
X: gassert(R.CT_u_passw == A.CT_u_passw);
X: gassert(R.client_id == A.client_id);
X: gassert(R.port_id == A.port_id);
X: gassert(R.EAPrespID == A.EAPrespID);
X: gassert(R.EAPrespID == S.EAPrespID);
X: gassert(R.CT_EAPrespID == A.CT_EAPrespID);
R: assert(CT_EAPrespID == CT_EAPrespID');

--MSG-4: RADIUS responds to Access Point
R: RADresp1_auth := MD5(RADAccReq1, secret);
R: EAPreq1 := <req_code, req1_id, req1_leng, req1_type, req1_type_data>;
R: CT_EAPreq1 := MD5(EAPreq1);
R: RADchal_attr := <u_name, CT_u_passw, client_id, port_id, EAPreq1,
CT_EAPreq1>;
R: RADchal_id := RADreq1_id;
R: RADAccChal := <RADchal_code, RADchal_id, RADchal_leng, RADresp1_auth,
RADchal_attr>;
R: => A (RADAccChal);
A: <- (RADAccChal);
A:(RADchal_code, RADchal_id, RADchal_leng, RADresp1_auth, RADchal_attr) :=
RADAccChal;
A: (u_name, CT_u_passw, client_id, port_id, EAPreq1, CT_EAPreq1) :=
RADchal_attr;
A: (req_code, req1_id, req1_leng, req1_type, req1_type_data) := EAPreq1;
A: RADresp1_auth':= MD5(RADAccReq1, secret);
A: CT_EAPreq1' := MD5(EAPreq1);
X: gassert(A.RADAccChal == R.RADAccChal);
X: gassert(A.RADchal_code == R.RADchal_code);
X: gassert(A.RADchal_id == R.RADchal_id);
X: gassert(A.RADchal_leng == R.RADchal_leng);
X: gassert(A.RADresp1_auth == R.RADresp1_auth);
X: gassert(A.RADchal_attr == R.RADchal_attr); --03.27.03
X: gassert(A.u_name == R.u_name); --06.06.03
X: gassert(A.CT_u_passw == R.CT_u_passw); --06.07.03
X: gassert(A.client_id == R.client_id); --06.07.03
X: gassert(A.port_id == R.port_id); --06.07.03
X: gassert(A.EAPreq1 == R.EAPreq1); --03.27.03
X: gassert(A.CT_EAPreq1 == R.CT_EAPreq1); --06.07.03
A: assert(RADchal_id == RADreq1_id);
A: assert(RADresp1_auth'== RADresp1_auth);
A: assert(CT_EAPreq1' == CT_EAPreq1);

--MSG-5: Access Point requests from Supplicant
A: => S(EAPreq1);
```

```
S: <- (EAPreq1);
S: (req_code, req1_id, req1_leng, req1_type, req1_type_data) := EAPreq1;
X: gassert(S.EAPreq1 == A.EAPreq1);
X: gassert(S.req_code == A.req_code);
X: gassert(S.req1_id == A.req1_id);
X: gassert(S.req1_leng == A.req1_leng);
X: gassert(S.req1_type == A.req1_type);
X: gassert(S.req1_type_data == A.req1_type_data);

--MSG-6: Supplicant responds to Access Point
S: res1_id := req1_id;
S: res1_type := req1_type;
S: EAPresp1 := <res_code, res1_id, res1_leng, res1_type, res1_type_data>;
S: => A(EAPresp1);
A: <- (EAPresp1);
A: (res_code, res1_id, res1_leng, res1_type, res1_type_data) := EAPresp1;
X: gassert(A.EAPresp1 == S.EAPresp1);
X: gassert(A.res_code == S.res_code);
X: gassert(A.res1_id == S.res1_id);
X: gassert(A.res1_leng == S.res1_leng);
X: gassert(A.res1_type == S.res1_type);
X: gassert(A.res1_type_data == S.res1_type_data);
A: assert(req1_id == res1_id);
A: assert(req1_type == res1_type);

--MSG-7: Access Point requests from RADIUS
A: RADreq2_auth := MD5(RN2);
A: CT_temp2 := MD5(RADreq2_auth, secret);
A: u_passw2 := XOR(CT_temp2, passw);
A: CT_EAPresp1 := MD5(EAPresp1);
A: CT_u_passw2 := MD5(u_passw2);
A: RADreq2_attr := <u_name, CT_u_passw2, client_id, port_id, EAPresp1,
CT_EAPresp1>;
A: RADAccReq2 := <RADreq_code, RADreq2_id, RADreq2_leng, RADreq2_auth,
RADreq2_attr >;
A: => R (RADAccReq2);
R: <- (RADAccReq2);
R: (RADreq_code, RADreq2_id, RADreq2_leng, RADreq2_auth, RADreq2_attr) :=
RADAccReq2;
R: (u_name, CT_u_passw2, client_id, port_id, EAPresp1, CT_EAPresp1 ) :=
RADreq2_attr;
R: CT_EAPresp1'':= MD5(EAPresp1);
X: gassert(R.RADAccReq2 == A.RADAccReq2);
X: gassert(R.RADreq_code == A.RADreq_code);
X: gassert(R.RADreq2_id == A.RADreq2_id);
X: gassert(R.RADreq2_leng == A.RADreq2_leng);
X: gassert(R.RADreq2_auth == A.RADreq2_auth);
X: gassert(R.RADreq2_attr == A.RADreq2_attr);
X: gassert(R.u_name == A.u_name);
X: gassert(R.CT_u_passw2 == A.CT_u_passw2);
X: gassert(R.client_id == A.client_id);
X: gassert(R.port_id == A.port_id);
X: gassert(R.EAPresp1 == A.EAPresp1);
X: gassert(R.EAPresp1 == S.EAPresp1);
X: gassert(R.CT_EAPresp1 == A.CT_EAPresp1);
R: assert(CT_EAPresp1 == CT_EAPresp1'');

--MSG-8: RADIUS responds success to Access Point
R: RADsucc_auth := MD5(RADAccReq2, secret);
R: EAPsucc := <succ_code, succ_id, succ_leng>;
```

```
R: CT_EAPsucc := MD5(EAPsucc);
R: RADsucc_attr := <u_name, CT_u_passw2, client_id, port_id, EAPsucc,
CT_EAPsucc>;
R: RADsucc_id := RADreq2_id;
R: RADaccept := <RADsucc_code, RADsucc_id, RADsucc_leng, RADsucc_auth,
RADsucc_attr>;
R: => A(RADaccept);
A: <- (RADaccept);
A: (RADsucc_code, RADsucc_id, RADsucc_leng, RADsucc_auth, RADsucc_attr) :=
RADaccept;
A: (u_name, CT_u_passw2, client_id, port_id, EAPsucc, CT_EAPsucc) :=
RADsucc_attr;
A: (succ_code, succ_id, succ_leng) := EAPsucc;
A: RADsucc_auth':= MD5(RADAccReq2, secret);
A: CT_EAPsucc' := MD5(EAPsucc);
X: gassert(A.RADaccept== R.RADaccept);
X: gassert(A.RADsucc_code == R.RADsucc_code);
X: gassert(A.RADsucc_id == R.RADsucc_id);
X: gassert(A.RADsucc_leng == R.RADsucc_leng);
X: gassert(A.RADsucc_auth == R.RADsucc_auth);
X: gassert(A.RADsucc_attr == R.RADsucc_attr);
X: gassert(A.u_name == R.u_name);
X: gassert(A.CT_u_passw2 == R.CT_u_passw2);
X: gassert(A.client_id == R.client_id);
X: gassert(A.port_id == R.port_id);
X: gassert(A.EAPsucc == R.EAPsucc);
X: gassert(A.CT_EAPsucc == R.CT_EAPsucc);
A: assert(RADsucc_id == RADreq2_id);
A: assert(RADsucc_auth'== RADsucc_auth);
A: assert(CT_EAPsucc' == CT_EAPsucc);

--MSG-9: Access Point sends success message to Supplicant
A: => S(EAPsucc);
S: <- (EAPsucc);
S: (succ_code, succ_id, succ_leng) := EAPsucc;
X: gassert(S.EAPsucc == A.EAPsucc);
X: gassert(S.EAPsucc == R.EAPsucc);
X: gassert(S.succ_code == A.succ_code);
X: gassert(S.succ_id == A.succ_id);
X: gassert(S.succ_leng == A.succ_leng);
```

APPENDIX J

CPAL-ES EVALUATION OF IEEE 802.1X PROTOCOL

```
1.  X: assume((S.kas == A.kas));
2.  X: assume((R.secret == A.secret));
3.  S: => A(e[S.passw]S.kas);
4.  A: <-(A.msg);
5.  A: A.passw := d[A.msg]A.kas;
6.  X: gassert((A.passw == S.passw));
7.  A: A.EAPreqID :=
   <A.req_code,A.req_id,A.req_leng,A.req_type,A.req_type_data>;
8.  A: => S(A.EAPreqID);
9.  S: <-(S.EAPreqID);
10. S: (S.req_code,S.req_id,S.req_leng,S.req_type,S.req_type_data) :=
   S.EAPreqID;
11. X: gassert((A.EAPreqID == S.EAPreqID));
12. X: gassert((A.req_code == S.req_code));
13. X: gassert((A.req_id == S.req_id));
14. X: gassert((A.req_leng == S.req_leng));
15. X: gassert((A.req_type == S.req_type));
16. X: gassert((A.req_type_data == S.req_type_data));
17. S: S.res_id := S.req_id;
18. S: S.res_type := S.req_type;
19. S: S.EAPrespID :=
   <S.res_code,S.res_id,S.res_leng,S.res_type,S.res_type_data>;
20. S: => A(S.EAPrespID);
21. A: <-(A.EAPrespID);
22. A: (A.res_code,A.res_id,A.res_leng,A.res_type,A.res_type_data) :=
   A.EAPrespID;
23. X: gassert((S.EAPrespID == A.EAPrespID));
24. X: gassert((S.res_code == A.res_code));
25. X: gassert((S.res_id == A.res_id));
26. X: gassert((S.res_leng == A.res_leng));
27. X: gassert((S.res_type == A.res_type));
28. X: gassert((S.res_type_data == A.res_type_data));
29. A: assert((A.res_id == A.req_id));
30. A: assert((A.res_type == A.req_type));
31. A: A.RADreq1_auth := f.MD5(A.RN1);
32. A: A.CT_temp := f.MD5(A.RADreq1_auth,A.secret);
33. A: A.u_passw1 := f.XOR(A.CT_temp,A.passw);
34. A: A.CT_EAPrespID := f.MD5(A.EAPrespID);
35. A: A.CT_u_passw := f.MD5(A.u_passw1);
36. A: A.RADreq1_attr :=
   <A.u_name,A.CT_u_passw,A.client_id,A.port_id,A.EAPrespID,A.CT_EAPrespID>
   ;
37. A: A.RADAccReq1 :=
   <A.RADreq_code,A.RADreq1_id,A.RADreq1_leng,A.RADreq1_auth,A.RADreq1_attr
   >;
38. A: => R(A.RADAccReq1);
39. R: <-(R.RADAccReq1);
```

```
40.  R:
   (R.RADreq_code,R.RADreq1_id,R.RADreq1_leng,R.RADreq1_auth,R.RADreq1_attr
   ) := R.RADAccReq1;
41.  R:
   (R.u_name,R.CT_u_passw,R.client_id,R.port_id,R.EAPrespID,R.CT_EAPrespID)
   := R.RADreq1_attr;
42.  R: R.CT_EAPrespID' := f.MD5(R.EAPrespID);
43.  X: gassert((A.RADAccReq1 == R.RADAccReq1));
44.  X: gassert((A.RADreq_code == R.RADreq_code));
45.  X: gassert((A.RADreq1_id == R.RADreq1_id));
46.  X: gassert((A.RADreq1_leng == R.RADreq1_leng));
47.  X: gassert((A.RADreq1_auth == R.RADreq1_auth));
48.  X: gassert((A.RADreq1_attr == R.RADreq1_attr));
49.  X: gassert((A.u_name == R.u_name));
50.  X: gassert((A.CT_u_passw == R.CT_u_passw));
51.  X: gassert((A.client_id == R.client_id));
52.  X: gassert((A.port_id == R.port_id));
53.  X: gassert((A.EAPrespID == R.EAPrespID));
54.  X: gassert((S.EAPrespID == R.EAPrespID));
55.  X: gassert((A.CT_EAPrespID == R.CT_EAPrespID));
56.  R: assert((R.CT_EAPrespID' == R.CT_EAPrespID));
57.  R: R.RADresp1_auth := f.MD5(R.RADAccReq1,R.secret);
58.  R: R.EAPreq1 :=
   <R.req_code,R.req1_id,R.req1_leng,R.req1_type,R.req1_type_data>;
59.  R: R.CT_EAPreq1 := f.MD5(R.EAPreq1);
60.  R: R.RADchal_attr :=
   <R.u_name,R.CT_u_passw,R.client_id,R.port_id,R.EAPreq1,R.CT_EAPreq1>;
61.  R: R.RADchal_id := R.RADreq1_id;
62.  R: R.RADAccChal :=
   <R.RADchal_code,R.RADchal_id,R.RADchal_leng,R.RADresp1_auth,R.RADchal_at
   tr>;
63.  R: => A(R.RADAccChal);
64.  A: <-(A.RADAccChal);
65.  A:
   (A.RADchal_code,A.RADchal_id,A.RADchal_leng,A.RADresp1_auth,A.RADchal_at
   tr) := A.RADAccChal;
66.  A:
   (A.u_name,A.CT_u_passw,A.client_id,A.port_id,A.EAPreq1,A.CT_EAPreq1) :=
   A.RADchal_attr;
67.  A: (A.req_code,A.req1_id,A.req1_leng,A.req1_type,A.req1_type_data) :=
   A.EAPreq1;
68.  A: A.RADresp1_auth' := f.MD5(A.RADAccReq1,A.secret);
69.  A: A.CT_EAPreq1' := f.MD5(A.EAPreq1);
70.  X: gassert((R.RADAccChal == A.RADAccChal));
71.  X: gassert((R.RADchal_code == A.RADchal_code));
72.  X: gassert((R.RADchal_id == A.RADchal_id));
73.  X: gassert((R.RADchal_leng == A.RADchal_leng));
74.  X: gassert((R.RADresp1_auth == A.RADresp1_auth));
75.  X: gassert((R.RADchal_attr == A.RADchal_attr));
76.  X: gassert((R.u_name == A.u_name));
77.  X: gassert((R.CT_u_passw == A.CT_u_passw));
78.  X: gassert((R.client_id == A.client_id));
79.  X: gassert((R.port_id == A.port_id));
80.  X: gassert((R.EAPreq1 == A.EAPreq1));
81.  X: gassert((R.CT_EAPreq1 == A.CT_EAPreq1));
82.  A: assert((A.RADreq1_id == A.RADchal_id));
83.  A: assert((A.RADresp1_auth == A.RADresp1_auth'));
84.  A: assert((A.CT_EAPreq1 == A.CT_EAPreq1'));
85.  A: => S(A.EAPreq1);
86.  S: <-(S.EAPreq1);
```

```
87.  S: (S.req_code,S.req1_id,S.req1_leng,S.req1_type,S.req1_type_data) :=
   S.EAPreq1;
88.  X: gassert((A.EAPreq1 == S.EAPreq1));
89.  X: gassert((A.req_code == S.req_code));
90.  X: gassert((A.req1_id == S.req1_id));
91.  X: gassert((A.req1_leng == S.req1_leng));
92.  X: gassert((A.req1_type == S.req1_type));
93.  X: gassert((A.req1_type_data == S.req1_type_data));
94.  S: S.res1_id := S.req1_id;
95.  S: S.res1_type := S.req1_type;
96.  S: S.EAPresp1 :=
   <S.res_code,S.res1_id,S.res1_leng,S.res1_type,S.res1_type_data>;
97.  S: => A(S.EAPresp1);
98.  A: <-(A.EAPresp1);
99.  A: (A.res_code,A.res1_id,A.res1_leng,A.res1_type,A.res1_type_data) :=
   A.EAPresp1;
100. X: gassert((S.EAPresp1 == A.EAPresp1));
101. X: gassert((S.res_code == A.res_code));
102. X: gassert((S.res1_id == A.res1_id));
103. X: gassert((S.res1_leng == A.res1_leng));
104. X: gassert((S.res1_type == A.res1_type));

105. X: gassert((S.res1_type_data == A.res1_type_data));
106. A: assert((A.res1_id == A.req1_id));
107. A: assert((A.res1_type == A.req1_type));
108. A: A.RADreq2_auth := f.MD5(A.RN2);
109. A: A.CT_temp2 := f.MD5(A.RADreq2_auth,A.secret);
110. A: A.u_passw2 := f.XOR(A.CT_temp2,A.passw);
111. A: A.CT_EAPresp1 := f.MD5(A.EAPresp1);
112. A: A.CT_u_passw2 := f.MD5(A.u_passw2);
113. A: A.RADreq2_attr :=
   <A.u_name,A.CT_u_passw2,A.client_id,A.port_id,A.EAPresp1,A.CT_EAPresp1>;
114. A: A.RADAccReq2 :=
   <A.RADreq_code,A.RADreq2_id,A.RADreq2_leng,A.RADreq2_auth,A.RADreq2_attr
   >;
115. A: => R(A.RADAccReq2);
116. R: <-(R.RADAccReq2);
117. R:
   (R.RADreq_code,R.RADreq2_id,R.RADreq2_leng,R.RADreq2_auth,R.RADreq2_attr
   ) := R.RADAccReq2;
118. R:
   (R.u_name,R.CT_u_passw2,R.client_id,R.port_id,R.EAPresp1,R.CT_EAPresp1)
   := R.RADreq2_attr;
119. R: R.CT_EAPresp1' := f.MD5(R.EAPresp1);
120. X: gassert((A.RADAccReq2 == R.RADAccReq2));
121. X: gassert((A.RADreq_code == R.RADreq_code));
122. X: gassert((A.RADreq2_id == R.RADreq2_id));
123. X: gassert((A.RADreq2_leng == R.RADreq2_leng));
124. X: gassert((A.RADreq2_auth == R.RADreq2_auth));
125. X: gassert((A.RADreq2_attr == R.RADreq2_attr));
126. X: gassert((A.u_name == R.u_name));
127. X: gassert((A.CT_u_passw2 == R.CT_u_passw2));
128. X: gassert((A.client_id == R.client_id));
129. X: gassert((A.port_id == R.port_id));
130. X: gassert((A.EAPresp1 == R.EAPresp1));
131. X: gassert((S.EAPresp1 == R.EAPresp1));
132. X: gassert((A.CT_EAPresp1 == R.CT_EAPresp1));
133. R: assert((R.CT_EAPresp1' == R.CT_EAPresp1));
134. R: R.RADsucc_auth := f.MD5(R.RADAccReq2,R.secret);
135. R: R.EAPsucc := <R.succ_code,R.succ_id,R.succ_leng>;
```

```
   136. R: R.CT_EAPsucc := f.MD5(R.EAPsucc);
   137. R: R.RADsucc_attr :=
      <R.u_name,R.CT_u_passw2,R.client_id,R.port_id,R.EAPsucc,R.CT_EAPsucc>;
   138. R: R.RADsucc_id := R.RADreq2_id;
   139. R: R.RADaccept :=
      <R.RADsucc_code,R.RADsucc_id,R.RADsucc_leng,R.RADsucc_auth,R.RADsucc_att
      r>;
   140. R: => A(R.RADaccept);
   141. A: <-(A.RADaccept);
   142. A:
      (A.RADsucc_code,A.RADsucc_id,A.RADsucc_leng,A.RADsucc_auth,A.RADsucc_att
      r) := A.RADaccept;
   143. A:
      (A.u_name,A.CT_u_passw2,A.client_id,A.port_id,A.EAPsucc,A.CT_EAPsucc) :=
      A.RADsucc_attr;
   144. A: (A.succ_code,A.succ_id,A.succ_leng) := A.EAPsucc;
   145. A: A.RADsucc_auth' := f.MD5(A.RADAccReq2,A.secret);
   146. A: A.CT_EAPsucc' := f.MD5(A.EAPsucc);
   147. X: gassert((R.RADaccept == A.RADaccept));
   148. X: gassert((R.RADsucc_code == A.RADsucc_code));
   149. X: gassert((R.RADsucc_id == A.RADsucc_id));
   150. X: gassert((R.RADsucc_leng == A.RADsucc_leng));
   151. X: gassert((R.RADsucc_auth == A.RADsucc_auth));
   152. X: gassert((R.RADsucc_attr == A.RADsucc_attr));
   153. X: gassert((R.u_name == A.u_name));
   154. X: gassert((R.CT_u_passw2 == A.CT_u_passw2));
   155. X: gassert((R.client_id == A.client_id));
   156. X: gassert((R.port_id == A.port_id));
   157. X: gassert((R.EAPsucc == A.EAPsucc));
   158. X: gassert((R.CT_EAPsucc == A.CT_EAPsucc));
   159. A: assert((A.RADreq2_id == A.RADsucc_id));
   160. A: assert((A.RADsucc_auth == A.RADsucc_auth'));
   161. A: assert((A.CT_EAPsucc == A.CT_EAPsucc'));
   162. A: => S(A.EAPsucc);
   163. S: <-(S.EAPsucc);
   164. S: (S.succ_code,S.succ_id,S.succ_leng) := S.EAPsucc;
   165. X: gassert((A.EAPsucc == S.EAPsucc));
   166. X: gassert((R.EAPsucc == S.EAPsucc));
   167. X: gassert((A.succ_code == S.succ_code));
   168. X: gassert((A.succ_id == S.succ_id));
   169. X: gassert((A.succ_leng == S.succ_leng));

 *** End of Protocol ***

TRUE

****** Simplified predicate follows.

TRUE
```

APPENDIX K

CPAL-ES ENCODING OF MIM (MAN-IN-THE-MIDDLE ATTACK) ON IEEE 802.1X PROTOCOL

```
-- Initial assumptions
X: assume(A.kas == S.kas);
X: assume(A.secret == R.secret);

--MSG-0:Supplicant sends it's password to Access Point
S: => A (e[passw]kas);
A: <- (msg);
A: passw := d[msg]kas;
X:gassert (S.passw == A.passw);

--MSG-1: Access Point requests from Supplicant
A: EAPreqID := <req_code, req_id, req_leng, req_type, req_type_data>;
A: => S(EAPreqID);
S: <- (EAPreqID);
S: (req_code, req_id, req_leng, req_type, req_type_data) := EAPreqID;
X: gassert(S.EAPreqID == A.EAPreqID);
X: gassert(S.req_code == A.req_code);
X: gassert(S.req_id == A.req_id);

--MSG-2: Supplicant responds to Access Point
S: res_id := req_id;
S: res_type := req_type;
S: EAPrespID := <res_code, res_id, res_leng, res_type, res_type_data>;
S: => A(EAPrespID);
A: <- (EAPrespID);
A: (res_code, res_id, res_leng, res_type, res_type_data) := EAPrespID;
X: gassert(A.EAPrespID == S.EAPrespID);
X: gassert(A.res_code == S.res_code);
X: gassert(A.res_id == S.res_id);
A: assert(req_id == res_id);
A: assert(req_type == res_type);

--MSG-3: Access Point requests from RADIUS
A: RADreq1_auth := MD5(RN1);
A: CT_temp := MD5(RADreq1_auth, secret);
A: u_passw1 := XOR(CT_temp, passw);
A: CT_EAPrespID := MD5(EAPrespID);
A: CT_u_passw := MD5(u_passw1);
A: RADreq1_attr := <u_name, CT_u_passw, client_id, port_id, EAPrespID,
CT_EAPrespID>;
A: RADAccReq1 := <RADreq_code, RADreq1_id, RADreq1_leng, RADreq1_auth,
RADreq1_attr >;
A: => R (RADAccReq1);
R: <- (RADAccReq1);
R: (RADreq_code, RADreq1_id, RADreq1_leng, RADreq1_auth, RADreq1_attr) :=
RADAccReq1;
```

```
R: (u_name, CT_u_passw, client_id, port_id, EAPrespID, CT_EAPrespID) :=
RADreq1_attr;
R: CT_EAPrespID':=MD5(EAPrespID);
X: gassert(R.RADAccReq1 == A.RADAccReq1);
X: gassert(R.RADreq_code == A.RADreq_code);
X: gassert(R.RADreq1_id == A.RADreq1_id);
X: gassert(R.RADreq1_auth == A.RADreq1_auth);
X: gassert(R.RADreq1_attr == A.RADreq1_attr);
X: gassert(R.u_name == A.u_name);
X: gassert(R.CT_u_passw == A.CT_u_passw);
X: gassert(R.EAPrespID == A.EAPrespID);
X: gassert(R.EAPrespID == S.EAPrespID);
X: gassert(R.CT_EAPrespID == A.CT_EAPrespID);
R: assert(CT_EAPrespID == CT_EAPrespID');

--MSG-4: RADIUS responds to Access Point
R: RADresp1_auth := MD5(RADAccReq1, secret);
R: EAPreq1 := <req_code, req1_id, req1_leng, req1_type, req1_type_data>;
R: CT_EAPreq1 := MD5(EAPreq1);
R: RADchal_attr := <u_name, CT_u_passw, client_id, port_id, EAPreq1,
CT_EAPreq1>;
R: RADchal_id := RADreq1_id;
R: RADAccChal := <RADchal_code, RADchal_id, RADchal_leng, RADresp1_auth,
RADchal_attr>;
R: => A (RADAccChal);
A: <- (RADAccChal);
A:(RADchal_code, RADchal_id, RADchal_leng, RADresp1_auth, RADchal_attr) :=
RADAccChal;
A: (u_name, CT_u_passw, client_id, port_id, EAPreq1, CT_EAPreq1) :=
RADchal_attr;
A: (req_code, req1_id, req1_leng, req1_type, req1_type_data) := EAPreq1;
A: RADresp1_auth':= MD5(RADAccReq1, secret);
A: CT_EAPreq1' := MD5(EAPreq1);
X: gassert(A.RADAccChal == R.RADAccChal);
X: gassert(A.RADchal_code == R.RADchal_code);
X: gassert(A.RADchal_id == R.RADchal_id);
X: gassert(A.RADresp1_auth == R.RADresp1_auth);
X: gassert(A.RADchal_attr == R.RADchal_attr);
X: gassert(A.CT_u_passw == R.CT_u_passw);
X: gassert(A.EAPreq1 == R.EAPreq1);
X: gassert(A.CT_EAPreq1 == R.CT_EAPreq1);
A: assert(RADchal_id == RADreq1_id);
A: assert(RADresp1_auth'== RADresp1_auth);
A: assert(CT_EAPreq1' == CT_EAPreq1);

--MSG-5: Access Point requests from Supplicant
A: => S(EAPreq1);
S: <- (EAPreq1);
S: (req_code, req1_id, req1_leng, req1_type, req1_type_data) := EAPreq1;
X: gassert(S.EAPreq1 == A.EAPreq1);
X: gassert(S.req_code == A.req_code);
X: gassert(S.req1_id == A.req1_id);

--MSG-6: Supplicant responds to Access Point
S: res1_id := req1_id;
S: res1_type:= req1_type;
S: EAPresp1 := <res_code, res1_id, res1_leng, res1_type, res1_type_data>;
S: => A(EAPresp1);
A: <- (EAPresp1);
A: (res_code, res1_id, res1_leng, res1_type, res1_type_data) := EAPresp1;
```

```
X: gassert(A.EAPresp1 == S.EAPresp1);
X: gassert(A.res_code == S.res_code);
X: gassert(A.res1_id == S.res1_id);
A: assert(req1_id == res1_id);
A: assert(req1_type == res1_type);

--MSG-7: Access Point requests from RADIUS
A: RADreq2_auth := MD5(RN2);
A: CT_temp2 := MD5(RADreq2_auth, secret);
A: u_passw2 := XOR(CT_temp2, passw);
A: CT_EAPresp1 := MD5(EAPresp1); --06.07.03
A: CT_u_passw2 := MD5(u_passw2); --06.07.03
A: RADreq2_attr := <u_name, CT_u_passw2, client_id, port_id, EAPresp1,
CT_EAPresp1>;
A: RADAccReq2 := <RADreq_code, RADreq2_id, RADreq2_leng, RADreq2_auth,
RADreq2_attr >;
A: => R (RADAccReq2);
R: <- (RADAccReq2);
R: (RADreq_code, RADreq2_id, RADreq2_leng, RADreq2_auth, RADreq2_attr) :=
RADAccReq2;
R: (u_name, CT_u_passw2, client_id, port_id, EAPresp1, CT_EAPresp1 ) :=
RADreq2_attr;
R: CT_EAPresp1':= MD5(EAPresp1);
X: gassert(R.RADAccReq2 == A.RADAccReq2);
X: gassert(R.RADreq_code == A.RADreq_code);
X: gassert(R.RADreq2_id == A.RADreq2_id);
X: gassert(R.RADreq2_auth == A.RADreq2_auth);
X: gassert(R.RADreq2_attr == A.RADreq2_attr);
X: gassert(R.u_name == A.u_name); --06.07.03
X: gassert(R.CT_u_passw2 == A.CT_u_passw2);
X: gassert(R.EAPresp1 == A.EAPresp1);
X: gassert(R.EAPresp1 == S.EAPresp1);
X: gassert(R.CT_EAPresp1 == A.CT_EAPresp1);
R: assert(CT_EAPresp1 == CT_EAPresp1');

--MSG-8: RADIUS responds success to Access Point
R: RADsucc_auth := MD5(RADAccReq2, secret);
R: EAPsucc := <succ_code, succ_id, succ_leng>;
R: CT_EAPsucc := MD5(EAPsucc);
R: RADsucc_attr := <u_name, CT_u_passw2, client_id, port_id, EAPsucc,
CT_EAPsucc>;
R: RADsucc_id := RADreq2_id;
R: RADaccept := <RADsucc_code, RADsucc_id, RADsucc_leng, RADsucc_auth,
RADsucc_attr>;
R: => A(RADaccept);
A: <- (RADaccept);
A: (RADsucc_code, RADsucc_id, RADsucc_leng, RADsucc_auth, RADsucc_attr) :=
RADaccept;
A: (u_name, CT_u_passw2, client_id, port_id, EAPsucc, CT_EAPsucc) :=
RADsucc_attr;
A: (succ_code, succ_id, succ_leng) := EAPsucc;
A: RADsucc_auth':= MD5(RADAccReq2, secret);
A: CT_EAPsucc' := MD5(EAPsucc);
X: gassert(A.RADaccept== R.RADaccept);
X: gassert(A.RADsucc_code == R.RADsucc_code);
X: gassert(A.RADsucc_id == R.RADsucc_id);
X: gassert(A.RADsucc_auth == R.RADsucc_auth);
X: gassert(A.RADsucc_attr == R.RADsucc_attr);
X: gassert(A.u_name == R.u_name);
X: gassert(A.CT_u_passw2 == R.CT_u_passw2);
```

```
X: gassert(A.EAPsucc == R.EAPsucc);
X: gassert(A.CT_EAPsucc == R.CT_EAPsucc);
A: assert(RADsucc_id == RADreq2_id);
A: assert(RADsucc_auth'== RADsucc_auth);
A: assert(CT_EAPsucc' == CT_EAPsucc);

--MSG-9: Access Point sends success message to Supplicant, intruder I
intercepts the message
A: -> S(EAPsucc);
I: <- (EAPsucc);
I: (succ_code, succ_id, succ_leng) := EAPsucc;
X: gassert(I.EAPsucc == A.EAPsucc);
X: gassert(I.EAPsucc == R.EAPsucc);
X: gassert(I.succ_code == A.succ_code);
X: gassert(I.succ_id == A.succ_id);
X: gassert(I.succ_leng == A.succ_leng);

--MSG-10: Intruder sends success message to Supplicant
I: => S(EAPsucc);
S: <- (EAPsucc);
S: (succ_code, succ_id, succ_leng) := EAPsucc;
X: gassert(S.EAPsucc == I.EAPsucc);
X: gassert(S.EAPsucc == R.EAPsucc);
X: gassert(S.succ_code == I.succ_code);
X: gassert(S.succ_id == I.succ_id);
X: gassert(S.succ_leng == I.succ_leng);
```

APPENDIX L

CPAL-ES EVALUATION OF MIM (MAN-IN-THE-MIDDLE ATTACK) ON IEEE 802.1X PROTOCOL

```
1.  X: assume((S.kas == A.kas));
2.  X: assume((R.secret == A.secret));
3.  S: => A(e[S.passw]S.kas);
4.  A: <-(A.msg);
5.  A: A.passw := d[A.msg]A.kas;
6.  X: gassert((A.passw == S.passw));
7.  A: A.EAPreqID :=
   <A.req_code,A.req_id,A.req_leng,A.req_type,A.req_type_data>;
8.  A: => S(A.EAPreqID);
9.  S: <-(S.EAPreqID);
10. S: (S.req_code,S.req_id,S.req_leng,S.req_type,S.req_type_data) :=
   S.EAPreqID;
11. X: gassert((A.EAPreqID == S.EAPreqID));
12. X: gassert((A.req_code == S.req_code));
13. X: gassert((A.req_id == S.req_id));
14. S: S.res_id := S.req_id;
15. S: S.res_type := S.req_type;
16. S: S.EAPrespID :=
   <S.res_code,S.res_id,S.res_leng,S.res_type,S.res_type_data>;
17. S: => A(S.EAPrespID);
18. A: <-(A.EAPrespID);
19. A: (A.res_code,A.res_id,A.res_leng,A.res_type,A.res_type_data) :=
   A.EAPrespID;
20. X: gassert((S.EAPrespID == A.EAPrespID));
21. X: gassert((S.res_code == A.res_code));
22. X: gassert((S.res_id == A.res_id));
23. A: assert((A.res_id == A.req_id));
24. A: assert((A.res_type == A.req_type));
25. A: A.RADreq1_auth := f.MD5(A.RN1);
26. A: A.CT_temp := f.MD5(A.RADreq1_auth,A.secret);
27. A: A.u_passw1 := f.XOR(A.CT_temp,A.passw);
28. A: A.CT_EAPrespID := f.MD5(A.EAPrespID);
29. A: A.CT_u_passw := f.MD5(A.u_passw1);
30. A: A.RADreq1_attr :=
   <A.u_name,A.CT_u_passw,A.client_id,A.port_id,A.EAPrespID,A.CT_EAPrespID>
   ;
31. A: A.RADAccReq1 :=
   <A.RADreq_code,A.RADreq1_id,A.RADreq1_leng,A.RADreq1_auth,A.RADreq1_attr
   >;
32. A: => R(A.RADAccReq1);
33. R: <-(R.RADAccReq1);
34. R:
   (R.RADreq_code,R.RADreq1_id,R.RADreq1_leng,R.RADreq1_auth,R.RADreq1_attr
   ) := R.RADAccReq1;
35. R:
   (R.u_name,R.CT_u_passw,R.client_id,R.port_id,R.EAPrespID,R.CT_EAPrespID)
   := R.RADreq1_attr;
```

```
36.  R: R.CT_EAPrespID' := f.MD5(R.EAPrespID);
37.  X: gassert((A.RADAccReq1 == R.RADAccReq1));
38.  X: gassert((A.RADreq_code == R.RADreq_code));
39.  X: gassert((A.RADreq1_id == R.RADreq1_id));
40.  X: gassert((A.RADreq1_auth == R.RADreq1_auth));
41.  X: gassert((A.RADreq1_attr == R.RADreq1_attr));
42.  X: gassert((A.u_name == R.u_name));
43.  X: gassert((A.CT_u_passw == R.CT_u_passw));
44.  X: gassert((A.EAPrespID == R.EAPrespID));
45.  X: gassert((S.EAPrespID == R.EAPrespID));
46.  X: gassert((A.CT_EAPrespID == R.CT_EAPrespID));
47.  R: assert((R.CT_EAPrespID' == R.CT_EAPrespID));
48.  R: R.RADresp1_auth := f.MD5(R.RADAccReq1,R.secret);
49.  R: R.EAPreq1 :=
   <R.req_code,R.req1_id,R.req1_leng,R.req1_type,R.req1_type_data>;
50.  R: R.CT_EAPreq1 := f.MD5(R.EAPreq1);
51.  R: R.RADchal_attr :=
   <R.u_name,R.CT_u_passw,R.client_id,R.port_id,R.EAPreq1,R.CT_EAPreq1>;
52.  R: R.RADchal_id := R.RADreq1_id;
53.  R: R.RADAccChal :=
   <R.RADchal_code,R.RADchal_id,R.RADchal_leng,R.RADresp1_auth,R.RADchal_at
   tr>;
54.  R: => A(R.RADAccChal);
55.  A: <-(A.RADAccChal);
56.  A:
   (A.RADchal_code,A.RADchal_id,A.RADchal_leng,A.RADresp1_auth,A.RADchal_at
   tr) := A.RADAccChal;
57.  A:
   (A.u_name,A.CT_u_passw,A.client_id,A.port_id,A.EAPreq1,A.CT_EAPreq1) :=
   A.RADchal_attr;
58.  A: (A.req_code,A.req1_id,A.req1_leng,A.req1_type,A.req1_type_data) :=
   A.EAPreq1;
59.  A: A.RADresp1_auth' := f.MD5(A.RADAccReq1,A.secret);
60.  A: A.CT_EAPreq1' := f.MD5(A.EAPreq1);
61.  X: gassert((R.RADAccChal == A.RADAccChal));
62.  X: gassert((R.RADchal_code == A.RADchal_code));
63.  X: gassert((R.RADchal_id == A.RADchal_id));
64.  X: gassert((R.RADresp1_auth == A.RADresp1_auth));
65.  X: gassert((R.RADchal_attr == A.RADchal_attr));
66.  X: gassert((R.CT_u_passw == A.CT_u_passw));
67.  X: gassert((R.EAPreq1 == A.EAPreq1));
68.  X: gassert((R.CT_EAPreq1 == A.CT_EAPreq1));
69.  A: assert((A.RADreq1_id == A.RADchal_id));
70.  A: assert((A.RADresp1_auth == A.RADresp1_auth'));
71.  A: assert((A.CT_EAPreq1 == A.CT_EAPreq1'));
72.  A: => S(A.EAPreq1);
73.  S: <-(S.EAPreq1);
74.  S: (S.req_code,S.req1_id,S.req1_leng,S.req1_type,S.req1_type_data) :=
   S.EAPreq1;
75.  X: gassert((A.EAPreq1 == S.EAPreq1));
76.  X: gassert((A.req_code == S.req_code));
77.  X: gassert((A.req1_id == S.req1_id));
78.  S: S.res1_id := S.req1_id;
79.  S: S.res1_type := S.req1_type;
80.  S: S.EAPresp1 :=
   <S.res_code,S.res1_id,S.res1_leng,S.res1_type,S.res1_type_data>;
81.  S: => A(S.EAPresp1);
82.  A: <-(A.EAPresp1);
83.  A: (A.res_code,A.res1_id,A.res1_leng,A.res1_type,A.res1_type_data) :=
   A.EAPresp1;
```

```
84.  X: gassert((S.EAPresp1 == A.EAPresp1));
85.  X: gassert((S.res_code == A.res_code));
86.  X: gassert((S.res1_id == A.res1_id));
87.  A: assert((A.res1_id == A.req1_id));
88.  A: assert((A.res1_type == A.req1_type));
89.  A: A.RADreq2_auth := f.MD5(A.RN2);
90.  A: A.CT_temp2 := f.MD5(A.RADreq2_auth,A.secret);
91.  A: A.u_passw2 := f.XOR(A.CT_temp2,A.passw);
92.  A: A.CT_EAPresp1 := f.MD5(A.EAPresp1);
93.  A: A.CT_u_passw2 := f.MD5(A.u_passw2);
94.  A: A.RADreq2_attr :=
     <A.u_name,A.CT_u_passw2,A.client_id,A.port_id,A.EAPresp1,A.CT_EAPresp1>;
95.  A: A.RADAccReq2 :=
     <A.RADreq_code,A.RADreq2_id,A.RADreq2_leng,A.RADreq2_auth,A.RADreq2_attr
     >;
96.  A: => R(A.RADAccReq2);
97.  R: <-(R.RADAccReq2);
98.  R:
     (R.RADreq_code,R.RADreq2_id,R.RADreq2_leng,R.RADreq2_auth,R.RADreq2_attr
     ) := R.RADAccReq2;
99.  R:
     (R.u_name,R.CT_u_passw2,R.client_id,R.port_id,R.EAPresp1,R.CT_EAPresp1)
     := R.RADreq2_attr;
100. R: R.CT_EAPresp1' := f.MD5(R.EAPresp1);
101. X: gassert((A.RADAccReq2 == R.RADAccReq2));
102. X: gassert((A.RADreq_code == R.RADreq_code));
103. X: gassert((A.RADreq2_id == R.RADreq2_id));
104. X: gassert((A.RADreq2_auth == R.RADreq2_auth));
105. X: gassert((A.RADreq2_attr == R.RADreq2_attr));
106. X: gassert((A.u_name == R.u_name));
107. X: gassert((A.CT_u_passw2 == R.CT_u_passw2));
108. X: gassert((A.EAPresp1 == R.EAPresp1));
109. X: gassert((S.EAPresp1 == R.EAPresp1));
110. X: gassert((A.CT_EAPresp1 == R.CT_EAPresp1));
111. R: assert((R.CT_EAPresp1' == R.CT_EAPresp1));
112. R: R.RADsucc_auth := f.MD5(R.RADAccReq2,R.secret);
113. R: R.EAPsucc := <R.succ_code,R.succ_id,R.succ_leng>;
114. R: R.CT_EAPsucc := f.MD5(R.EAPsucc);
115. R: R.RADsucc_attr :=
     <R.u_name,R.CT_u_passw2,R.client_id,R.port_id,R.EAPsucc,R.CT_EAPsucc>;
116. R: R.RADsucc_id := R.RADreq2_id;
117. R: R.RADaccept :=
     <R.RADsucc_code,R.RADsucc_id,R.RADsucc_leng,R.RADsucc_auth,R.RADsucc_att
     r>;
118. R: => A(R.RADaccept);
119. A: <-(A.RADaccept);
120. A:
     (A.RADsucc_code,A.RADsucc_id,A.RADsucc_leng,A.RADsucc_auth,A.RADsucc_att
     r) := A.RADaccept;
121. A:
     (A.u_name,A.CT_u_passw2,A.client_id,A.port_id,A.EAPsucc,A.CT_EAPsucc) :=
     A.RADsucc_attr;
122. A: (A.succ_code,A.succ_id,A.succ_leng) := A.EAPsucc;
123. A: A.RADsucc_auth' := f.MD5(A.RADAccReq2,A.secret);
124. A: A.CT_EAPsucc' := f.MD5(A.EAPsucc);
125. X: gassert((R.RADaccept == A.RADaccept));
126. X: gassert((R.RADsucc_code == A.RADsucc_code));
127. X: gassert((R.RADsucc_id == A.RADsucc_id));
128. X: gassert((R.RADsucc_auth == A.RADsucc_auth));
129. X: gassert((R.RADsucc_attr == A.RADsucc_attr));
```

```
   130. X: gassert((R.u_name == A.u_name));
   131. X: gassert((R.CT_u_passw2 == A.CT_u_passw2));
   132. X: gassert((R.EAPsucc == A.EAPsucc));
   133. X: gassert((R.CT_EAPsucc == A.CT_EAPsucc));
   134. A: assert((A.RADreq2_id == A.RADsucc_id));
   135. A: assert((A.RADsucc_auth == A.RADsucc_auth'));
   136. A: assert((A.CT_EAPsucc == A.CT_EAPsucc'));
   137. A: -> S(A.EAPsucc);
   138. I: <-(I.EAPsucc);
   139. I: (I.succ_code,I.succ_id,I.succ_leng) := I.EAPsucc;
   140. X: gassert((A.EAPsucc == I.EAPsucc));
   141. X: gassert((R.EAPsucc == I.EAPsucc));
   142. X: gassert((A.succ_code == I.succ_code));
   143. X: gassert((A.succ_id == I.succ_id));
   144. X: gassert((A.succ_leng == I.succ_leng));
   145. I: => S(I.EAPsucc);
   146. S: <-(S.EAPsucc);
   147. S: (S.succ_code,S.succ_id,S.succ_leng) := S.EAPsucc;
   148. X: gassert((I.EAPsucc == S.EAPsucc));
   149. X: gassert((R.EAPsucc == S.EAPsucc));
   150. X: gassert((I.succ_code == S.succ_code));
   151. X: gassert((I.succ_id == S.succ_id));
   152. X: gassert((I.succ_leng == S.succ_leng));

 *** End of Protocol ***

TRUE

****** Simplified predicate follows.

TRUE
```

APPENDIX M

CPAL-ES ENCODING OF SOLUTION TO MIM (MAN-IN-THE-MIDDLE ATTACK) ON IEEE 802.1X PROTOCOL

```
--Initial assumptions
X: assume(A.kas==S.kas);
X: assume(A.secret==R.secret);

--MSG-0:Supplicant sends it's password to Access Point
S: => A (e[passw]kas);
A: <- (msg);
A: passw := d[msg]kas;
X:gassert (S.passw == A.passw);

--MSG-1: Access Point requests from Supplicant
A: EAPreqID := <req_code, req_id, req_leng, req_type, req_type_data>;
A: => S(EAPreqID);
S: <- (EAPreqID);
S: (req_code, req_id, req_leng, req_type, req_type_data) := EAPreqID;
X: gassert(S.EAPreqID == A.EAPreqID);
X: gassert(S.req_code == A.req_code);
X: gassert(S.req_id == A.req_id);

--MSG-2: Supplicant responds to Access Point
S: res_id := req_id;
S: res_type := req_type;
S: EAPrespID := <res_code, res_id, res_leng, res_type, res_type_data>;
S: => A(EAPrespID);
A: <- (EAPrespID);
A: (res_code, res_id, res_leng, res_type, res_type_data) := EAPrespID;
X: gassert(A.EAPrespID == S.EAPrespID);
X: gassert(A.res_code == S.res_code);
X: gassert(A.res_id == S.res_id);
A: assert(req_id == res_id);
A: assert(req_type == res_type);

--MSG-3: Access Point requests from RADIUS
A: RADreq1_auth := MD5(RN1);
A: CT_temp := MD5(RADreq1_auth, secret);
A: u_passw1 := XOR(CT_temp, passw);
A: CT_EAPrespID := MD5(EAPrespID);
A: CT_u_passw := MD5(u_passw1);
A: RADreq1_attr := <u_name, CT_u_passw, client_id, port_id, EAPrespID,
CT_EAPrespID>;
A: RADAccReq1 := <RADreq_code, RADreq1_id, RADreq1_leng, RADreq1_auth,
RADreq1_attr >;
A: => R (RADAccReq1);
R: <- (RADAccReq1);
R: (RADreq_code, RADreq1_id, RADreq1_leng, RADreq1_auth, RADreq1_attr) :=
RADAccReq1;
```

```
R: (u_name, CT_u_passw, client_id, port_id, EAPrespID, CT_EAPrespID) :=
RADreq1_attr;
R: CT_EAPrespID' := MD5(EAPrespID);
X: gassert(R.RADAccReq1 == A.RADAccReq1);
X: gassert(R.RADreq_code == A.RADreq_code);
X: gassert(R.RADreq1_id == A.RADreq1_id);
X: gassert(R.RADreq1_auth == A.RADreq1_auth);
X: gassert(R.RADreq1_attr == A.RADreq1_attr);
X: gassert(R.u_name == A.u_name);
X: gassert(R.CT_u_passw == A.CT_u_passw);
X: gassert(R.EAPrespID == A.EAPrespID);
X: gassert(R.EAPrespID == S.EAPrespID);
X: gassert(R.CT_EAPrespID == A.CT_EAPrespID);
R: assert(CT_EAPrespID == CT_EAPrespID');

--MSG-4: RADIUS responds to Access Point
R: RADresp1_auth := MD5(RADAccReq1, secret);
R: EAPreq1 := <req_code, req1_id, req1_leng, req1_type, req1_type_data>;
R: CT_EAPreq1 := MD5(EAPreq1);
R: RADchal_attr := <u_name, CT_u_passw, client_id, port_id, EAPreq1,
CT_EAPreq1>;
R: RADchal_id := RADreq1_id;
R: RADAccChal := <RADchal_code, RADchal_id, RADchal_leng, RADresp1_auth,
RADchal_attr>;
R: => A (RADAccChal);
A: <- (RADAccChal);
A:(RADchal_code, RADchal_id, RADchal_leng, RADresp1_auth, RADchal_attr) :=
RADAccChal;
A: (u_name, CT_u_passw, client_id, port_id, EAPreq1, CT_EAPreq1) :=
RADchal_attr;
A: (req_code, req1_id, req1_leng, req1_type, req1_type_data) := EAPreq1;
A: RADresp1_auth':= MD5(RADAccReq1, secret);
A: CT_EAPreq1' := MD5(EAPreq1);
X: gassert(A.RADAccChal == R.RADAccChal);
X: gassert(A.RADchal_code == R.RADchal_code);
X: gassert(A.RADchal_id == R.RADchal_id);
X: gassert(A.RADresp1_auth == R.RADresp1_auth);
X: gassert(A.RADchal_attr == R.RADchal_attr);
X: gassert(A.CT_u_passw == R.CT_u_passw);
X: gassert(A.EAPreq1 == R.EAPreq1);
X: gassert(A.CT_EAPreq1 == R.CT_EAPreq1);
A: assert(RADchal_id == RADreq1_id);
A: assert(RADresp1_auth'== RADresp1_auth);
A: assert(CT_EAPreq1' == CT_EAPreq1);

--MSG-5: Access Point requests from Supplicant
A: => S(EAPreq1);
S: <- (EAPreq1);
S: (req_code, req1_id, req1_leng, req1_type, req1_type_data) := EAPreq1;
X: gassert(S.EAPreq1 == A.EAPreq1);
X: gassert(S.req_code == A.req_code);
X: gassert(S.req1_id == A.req1_id);

--MSG-6: Supplicant responds to Access Point
S: res1_id := req1_id;
S: res1_type := req1_type;
S: EAPresp1 := <res_code, res1_id, res1_leng, res1_type, res1_type_data>;
S: => A(EAPresp1);
A: <- (EAPresp1);
A: (res_code, res1_id, res1_leng, res1_type, res1_type_data) := EAPresp1;
```

```
X: gassert(A.EAPresp1 == S.EAPresp1);
X: gassert(A.res_code == S.res_code);
X: gassert(A.res1_id == S.res1_id);
A: assert(req1_id == res1_id);
A: assert(req1_type == res1_type);

--MSG-7: Access Point requests from RADIUS
A: RADreq2_auth := MD5(RN2);
A: CT_temp2 := MD5(RADreq2_auth, secret);
A: u_passw2 := XOR(CT_temp2, passw);
A: CT_EAPresp1 := MD5(EAPresp1);
A: CT_u_passw2 := MD5(u_passw2);
A: RADreq2_attr := <u_name, CT_u_passw2, client_id, port_id, EAPresp1,
CT_EAPresp1>;
A: RADAccReq2 := <RADreq_code, RADreq2_id, RADreq2_leng, RADreq2_auth,
RADreq2_attr >;
A: => R (RADAccReq2);
R: <- (RADAccReq2);
R: (RADreq_code, RADreq2_id, RADreq2_leng, RADreq2_auth, RADreq2_attr) :=
RADAccReq2;
R: (u_name, CT_u_passw2, client_id, port_id, EAPresp1, CT_EAPresp1 ) :=
RADreq2_attr;
R: CT_EAPresp1' := MD5(EAPresp1);
X: gassert(R.RADAccReq2 == A.RADAccReq2);
X: gassert(R.RADreq_code == A.RADreq_code);
X: gassert(R.RADreq2_id == A.RADreq2_id);
X: gassert(R.RADreq2_auth == A.RADreq2_auth);
X: gassert(R.RADreq2_attr == A.RADreq2_attr);
X: gassert(R.u_name == A.u_name);
X: gassert(R.CT_u_passw2 == A.CT_u_passw2);
X: gassert(R.EAPresp1 == A.EAPresp1);
X: gassert(R.EAPresp1 == S.EAPresp1);
X: gassert(R.CT_EAPresp1 == A.CT_EAPresp1);
R: assert(CT_EAPresp1 == CT_EAPresp1');

--MSG-8: RADIUS responds success to Access Point
R: RADsucc_auth := MD5(RADAccReq2, secret);
R: EAPsucc := <succ_code, succ_id, succ_leng>;
R: CT_EAPsucc := MD5(EAPsucc);
R: RADsucc_attr := <u_name, CT_u_passw2, client_id, port_id, EAPsucc,
CT_EAPsucc>;
R: RADsucc_id := RADreq2_id;
R: RADaccept := <RADsucc_code, RADsucc_id, RADsucc_leng, RADsucc_auth,
RADsucc_attr>;
R: => A(RADaccept);
A: <- (RADaccept);
A: (RADsucc_code, RADsucc_id, RADsucc_leng, RADsucc_auth, RADsucc_attr) :=
RADaccept;
A: (u_name, CT_u_passw2, client_id, port_id, EAPsucc, CT_EAPsucc) :=
RADsucc_attr;
A: (succ_code, succ_id, succ_leng) := EAPsucc;
A: RADsucc_auth' := MD5(RADAccReq2, secret);
A: CT_EAPsucc' := MD5(EAPsucc);
X: gassert(A.RADaccept == R.RADaccept);
X: gassert(A.RADsucc_code == R.RADsucc_code);
X: gassert(A.RADsucc_id == R.RADsucc_id);
X: gassert(A.RADsucc_auth == R.RADsucc_auth);
X: gassert(A.RADsucc_attr == R.RADsucc_attr);
X: gassert(A.u_name == R.u_name);
X: gassert(A.CT_u_passw2 == R.CT_u_passw2);
```

```
X: gassert(A.EAPsucc == R.EAPsucc);
X: gassert(A.CT_EAPsucc == R.CT_EAPsucc);
A: assert(RADsucc_id == RADreq2_id);
A: assert(RADsucc_auth'== RADsucc_auth);
A: assert(CT_EAPsucc' == CT_EAPsucc);

--MSG-9: Access Point sends success message to Supplicant.
A: succmsg := <EAPsucc, MD5(EAPsucc)>;
A: => S(succmsg);
S: <- (succmsg);
S:  (EAPsucc, CT_forsucc) := succmsg;
S: (succ_code, succ_id, succ_leng) := EAPsucc;
S: succmsg' := <EAPsucc, MD5(EAPsucc)>;
X: gassert(S.EAPsucc == A.EAPsucc);
X: gassert(S.EAPsucc == R.EAPsucc);
X: gassert(S.succ_code == A.succ_code);
X: gassert(S.succ_id == A.succ_id);
X: gassert(S.succ_leng == A.succ_leng);
S: assert(succmsg == succmsg');
```

APPENDIX N

CPAL-ES EVALUATION OF SOLUTION TO MIM (MAN-IN-THE-MIDDLE ATTACK) ON IEEE 802.1X PROTOCOL

```
1.   X: assume((S.kas == A.kas));
2.   X: assume((R.secret == A.secret));
3.   S: => A(e[S.passw]S.kas);
4.   A: <-(A.msg);
5.   A: A.passw := d[A.msg]A.kas;
6.   X: gassert((A.passw == S.passw));
7.   A: A.EAPreqID :=
   <A.req_code,A.req_id,A.req_leng,A.req_type,A.req_type_data>;
8.   A: => S(A.EAPreqID);
9.   S: <-(S.EAPreqID);
10.S: (S.req_code,S.req_id,S.req_leng,S.req_type,S.req_type_data) :=
   S.EAPreqID;
11.X: gassert((A.EAPreqID == S.EAPreqID));
12.X: gassert((A.req_code == S.req_code));
13.X: gassert((A.req_id == S.req_id));
14.S: S.res_id := S.req_id;
15.S: S.res_type := S.req_type;
16.S: S.EAPrespID :=
   <S.res_code,S.res_id,S.res_leng,S.res_type,S.res_type_data>;
17.S: => A(S.EAPrespID);
18.A: <-(A.EAPrespID);
19.A: (A.res_code,A.res_id,A.res_leng,A.res_type,A.res_type_data) :=
   A.EAPrespID;
20.X: gassert((S.EAPrespID == A.EAPrespID));
21.X: gassert((S.res_code == A.res_code));
22.X: gassert((S.res_id == A.res_id));
23.A: assert((A.res_id == A.req_id));
24.A: assert((A.res_type == A.req_type));
25.A: A.RADreq1_auth := f.MD5(A.RN1);
26.A: A.CT_temp := f.MD5(A.RADreq1_auth,A.secret);
27.A: A.u_passw1 := f.XOR(A.CT_temp,A.passw);
28.A: A.CT_EAPrespID := f.MD5(A.EAPrespID);
29.A: A.CT_u_passw := f.MD5(A.u_passw1);
30.A: A.RADreq1_attr :=
   <A.u_name,A.CT_u_passw,A.client_id,A.port_id,A.EAPrespID,A.CT_EAPrespID>
   ;
31.A: A.RADAccReq1 :=
   <A.RADreq_code,A.RADreq1_id,A.RADreq1_leng,A.RADreq1_auth,A.RADreq1_attr
   >;
32.A: => R(A.RADAccReq1);
33.R: <-(R.RADAccReq1);
34.R:
   (R.RADreq_code,R.RADreq1_id,R.RADreq1_leng,R.RADreq1_auth,R.RADreq1_attr
   ) := R.RADAccReq1;
```

```
35.R:
   (R.u_name,R.CT_u_passw,R.client_id,R.port_id,R.EAPrespID,R.CT_EAPrespID)
   := R.RADreq1_attr;
36.R: R.CT_EAPrespID' := f.MD5(R.EAPrespID);
37.X: gassert((A.RADAccReq1 == R.RADAccReq1));
38.X: gassert((A.RADreq_code == R.RADreq_code));
39.X: gassert((A.RADreq1_id == R.RADreq1_id));
40.X: gassert((A.RADreq1_auth == R.RADreq1_auth));
41.X: gassert((A.RADreq1_attr == R.RADreq1_attr));
42.X: gassert((A.u_name == R.u_name));
43.X: gassert((A.CT_u_passw == R.CT_u_passw));
44.X: gassert((A.EAPrespID == R.EAPrespID));
45.X: gassert((S.EAPrespID == R.EAPrespID));
46.X: gassert((A.CT_EAPrespID == R.CT_EAPrespID));
47.R: assert((R.CT_EAPrespID' == R.CT_EAPrespID));
48.R: R.RADresp1_auth := f.MD5(R.RADAccReq1,R.secret);
49.R: R.EAPreq1 :=
   <R.req_code,R.req1_id,R.req1_leng,R.req1_type,R.req1_type_data>;
50.R: R.CT_EAPreq1 := f.MD5(R.EAPreq1);
51.R: R.RADchal_attr :=
   <R.u_name,R.CT_u_passw,R.client_id,R.port_id,R.EAPreq1,R.CT_EAPreq1>;
52.R: R.RADchal_id := R.RADreq1_id;
53.R: R.RADAccChal :=
   <R.RADchal_code,R.RADchal_id,R.RADchal_leng,R.RADresp1_auth,R.RADchal_at
   tr>;
54.R: => A(R.RADAccChal);
55.A: <-(A.RADAccChal);
56.A:
   (A.RADchal_code,A.RADchal_id,A.RADchal_leng,A.RADresp1_auth,A.RADchal_at
   tr) := A.RADAccChal;
57.A: (A.u_name,A.CT_u_passw,A.client_id,A.port_id,A.EAPreq1,A.CT_EAPreq1)
   := A.RADchal_attr;
58.A: (A.req_code,A.req1_id,A.req1_leng,A.req1_type,A.req1_type_data) :=
   A.EAPreq1;
59.A: A.RADresp1_auth' := f.MD5(A.RADAccReq1,A.secret);
60.A: A.CT_EAPreq1' := f.MD5(A.EAPreq1);
61.X: gassert((R.RADAccChal == A.RADAccChal));
62.X: gassert((R.RADchal_code == A.RADchal_code));
63.X: gassert((R.RADchal_id == A.RADchal_id));
64.X: gassert((R.RADresp1_auth == A.RADresp1_auth));
65.X: gassert((R.RADchal_attr == A.RADchal_attr));
66.X: gassert((R.CT_u_passw == A.CT_u_passw));
67.X: gassert((R.EAPreq1 == A.EAPreq1));
68.X: gassert((R.CT_EAPreq1 == A.CT_EAPreq1));
69.A: assert((A.RADreq1_id == A.RADchal_id));
70.A: assert((A.RADresp1_auth == A.RADresp1_auth'));
71.A: assert((A.CT_EAPreq1 == A.CT_EAPreq1'));
72.A: => S(A.EAPreq1);
73.S: <-(S.EAPreq1);
74.S: (S.req_code,S.req1_id,S.req1_leng,S.req1_type,S.req1_type_data) :=
   S.EAPreq1;
75.X: gassert((A.EAPreq1 == S.EAPreq1));
76.X: gassert((A.req_code == S.req_code));
77.X: gassert((A.req1_id == S.req1_id));
78.S: S.res1_id := S.req1_id;
79.S: S.res1_type := S.req1_type;
80.S: S.EAPresp1 :=
   <S.res_code,S.res1_id,S.res1_leng,S.res1_type,S.res1_type_data>;
81.S: => A(S.EAPresp1);
82.A: <-(A.EAPresp1);
```

```
83.A: (A.res_code,A.res1_id,A.res1_leng,A.res1_type,A.res1_type_data) :=
   A.EAPresp1;
84.X: gassert((S.EAPresp1 == A.EAPresp1));
85.X: gassert((S.res_code == A.res_code));
86.X: gassert((S.res1_id == A.res1_id));
87.A: assert((A.res1_id == A.req1_id));
88.A: assert((A.res1_type == A.req1_type));
89.A: A.RADreq2_auth := f.MD5(A.RN2);
90.A: A.CT_temp2 := f.MD5(A.RADreq2_auth,A.secret);
91.A: A.u_passw2 := f.XOR(A.CT_temp2,A.passw);
92.A: A.CT_EAPresp1 := f.MD5(A.EAPresp1);
93.A: A.CT_u_passw2 := f.MD5(A.u_passw2);
94.A: A.RADreq2_attr :=
   <A.u_name,A.CT_u_passw2,A.client_id,A.port_id,A.EAPresp1,A.CT_EAPresp1>;
95.A: A.RADAccReq2 :=
   <A.RADreq_code,A.RADreq2_id,A.RADreq2_leng,A.RADreq2_auth,A.RADreq2_attr
   >;
96.A: => R(A.RADAccReq2);
97.R: <-(R.RADAccReq2);
98.R:
   (R.RADreq_code,R.RADreq2_id,R.RADreq2_leng,R.RADreq2_auth,R.RADreq2_attr
   ) := R.RADAccReq2;
99.R:
   (R.u_name,R.CT_u_passw2,R.client_id,R.port_id,R.EAPresp1,R.CT_EAPresp1)
   := R.RADreq2_attr;
100. R: R.CT_EAPresp1' := f.MD5(R.EAPresp1);
101. X: gassert((A.RADAccReq2 == R.RADAccReq2));
102. X: gassert((A.RADreq_code == R.RADreq_code));
103. X: gassert((A.RADreq2_id == R.RADreq2_id));
104. X: gassert((A.RADreq2_auth == R.RADreq2_auth));
105. X: gassert((A.RADreq2_attr == R.RADreq2_attr));
106. X: gassert((A.u_name == R.u_name));
107. X: gassert((A.CT_u_passw2 == R.CT_u_passw2));
108. X: gassert((A.EAPresp1 == R.EAPresp1));
109. X: gassert((S.EAPresp1 == R.EAPresp1));
110. X: gassert((A.CT_EAPresp1 == R.CT_EAPresp1));
111. R: assert((R.CT_EAPresp1' == R.CT_EAPresp1));
112. R: R.RADsucc_auth := f.MD5(R.RADAccReq2,R.secret);
113. R: R.EAPsucc := <R.succ_code,R.succ_id,R.succ_leng>;
114. R: R.CT_EAPsucc := f.MD5(R.EAPsucc);
115. R: R.RADsucc_attr :=
   <R.u_name,R.CT_u_passw2,R.client_id,R.port_id,R.EAPsucc,R.CT_EAPsucc>;
116. R: R.RADsucc_id := R.RADreq2_id;
117. R: R.RADaccept :=
   <R.RADsucc_code,R.RADsucc_id,R.RADsucc_leng,R.RADsucc_auth,R.RADsucc_att
   r>;
118. R: => A(R.RADaccept);
119. A: <-(A.RADaccept);
120. A:
   (A.RADsucc_code,A.RADsucc_id,A.RADsucc_leng,A.RADsucc_auth,A.RADsucc_att
   r) := A.RADaccept;
121. A:
   (A.u_name,A.CT_u_passw2,A.client_id,A.port_id,A.EAPsucc,A.CT_EAPsucc) :=
   A.RADsucc_attr;
122. A: (A.succ_code,A.succ_id,A.succ_leng) := A.EAPsucc;
123. A: A.RADsucc_auth' := f.MD5(A.RADAccReq2,A.secret);
124. A: A.CT_EAPsucc' := f.MD5(A.EAPsucc);
125. X: gassert((R.RADaccept == A.RADaccept));
126. X: gassert((R.RADsucc_code == A.RADsucc_code));
127. X: gassert((R.RADsucc_id == A.RADsucc_id));
```

```
  128. X: gassert((R.RADsucc_auth == A.RADsucc_auth));
  129. X: gassert((R.RADsucc_attr == A.RADsucc_attr));
  130. X: gassert((R.u_name == A.u_name));
  131. X: gassert((R.CT_u_passw2 == A.CT_u_passw2));
  132. X: gassert((R.EAPsucc == A.EAPsucc));
  133. X: gassert((R.CT_EAPsucc == A.CT_EAPsucc));
  134. A: assert((A.RADreq2_id == A.RADsucc_id));
  135. A: assert((A.RADsucc_auth == A.RADsucc_auth'));
  136. A: assert((A.CT_EAPsucc == A.CT_EAPsucc'));
  137. A: A.succmsg := <A.EAPsucc,f.MD5(A.EAPsucc)>;
  138. A: => S(A.succmsg);
  139. S: <-(S.succmsg);
  140. S: (S.EAPsucc,S.CT_forsucc) := S.succmsg;
  141. S: (S.succ_code,S.succ_id,S.succ_leng) := S.EAPsucc;
  142. S: S.succmsg' := <S.EAPsucc,f.MD5(S.EAPsucc)>;
  143. X: gassert((A.EAPsucc == S.EAPsucc));
  144. X: gassert((R.EAPsucc == S.EAPsucc));
  145. X: gassert((A.succ_code == S.succ_code));
  146. X: gassert((A.succ_id == S.succ_id));
  147. X: gassert((A.succ_leng == S.succ_leng));
  148. S: assert((S.succmsg' == S.succmsg));

 *** End of Protocol ***

TRUE

****** Simplified predicate follows.

TRUE
```

REFERENCES

[ARB01] W. A. Arbaugh, N. Shankar, and J. Wang. *Your 802.11 Network has no Clothes.* In Proceeding of the First IEEE International Conference on Wireless LANs and Home Networks, December 2001.

[ABA99] M. Abadi. *Your Security Protocols and Specifications.* In Foundations of Software Science and Computation Structures: Second International Conference, FOSSACS'99, pp.1-13, 1999.

[ARB01] W. A. Arbaugh, N. Shankar, and J. Wang. *Your 802.11 Network has no Clothes.* In Proceeding of the First IEEE International Conference on Wireless LANs and Home Networks, December 2001.

[AZI94] A. Aziz, and W. Diffie. *Privacy and Authentication for Wireless Local Area Networks.* In IEEE Personal Communications, First Quarter, pp. 25-31, 1994.

[BIE90] P. Bieber. *A Logic of Communication in a Hostile Environment.* In Proceeding of the Computer Security Foundations Workshop III, pp. 14-22, IEEE Computer Society Press, June 1990.

[BOR01] N. Borisov, I. Goldberg, and W. Wagner. *Intercepting Mobile Communications: The insecurity of 802.11.* In Proceeding of the Seventh Annual International Conference on Mobile Computing and Networking, pp.180-188, 2001.

[BOY93] C. Boyd, and W.Mao. *On a Limitation of Ban Logic.* Advances in Cryptology-EYROCRYPT '93, pp. 240-247, 1993.

[BOY98] C. Boyd, and A.Mathuria. *Key Establishment Protocols for Secure Mobile Communications.* A selective survey in: Australasian Conference on Information Security and Privacy, pp.344-355, 1998.

[BIRD92] R. Bird, I. Gopal, A. Herzberg, P. Janson, S. Kuteen, R. Molva, and M. Yung. *Systematic Design of a Family of Attack-Resistant Authentication Protocols*, IBM Raleigh, Watson & Zürich Laboratories, 1992.

[BLU98] L. Blunk, and J. Vollbrecht. *PPP Extensible Authentication Protocol (EAP)*, RFC 2284, March 1998.

[BUR90] M. Burrows, M. Abadi, and R. M. Needham. *A Logic of Authentication.* ACM Transactions on Computer Systems, vol.8, no.1, pp.18-36, Feb 1990.

[BUT99] L. Buttyán. *Formal Methods in the design of cryptographic protocols.* Technical Report, no. SSC/1999/38 Feb 1990, Swiss Federal Institute of Technology (EPFL). Lausanne, November 1999.

[BZM01] Q. Bi, G. I. Zysman, and H. Menkes. *Wireless Mobile Communications at the Start of the 21st Century.* IEEE Communications Magazine, no. 1, pp. 110-116, January 2001.

[CHI01] J. Childs. *Evaluating the TLS Family of Protocols with Weakest Precondition Reasoning*. Technical report TR-000703, Florida State University, July 2001.

[CCI88] CCITT Recommendation X.509. *The Directory-Authentication Framework*. 1998.

[COR02] R. Corin, and S. Etalle. *An Improved Constraint-Based System for the Verification of Security Protocols.* In M. Hermenegildo and G. Puebla, editors, 9th International Static Analysis Symposium (SAS), vol. 2477, pp. 326-342, Madrid, Spain, September 2002.

[DIF92] W. Diffie, P. C. V. Oorschot, and M. J. Wiener. *Authentication and Authenticated Key Exchanges.* Designs, Codes and Cryptography, Kluwer Academic Publishers, pp. 107-125, 1992.

[DIJ76] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall Series in Automatic Computation, Prentice Hall Inc. Englewood Cliffs, NJ, 1976.

[DON98] B. Donovan, P. Norris, and G. Lowe. *Analyzing a Library of Security Protocols using Casper and FDR.* In Proceedings of 1999 Workshop on Formal Methods and Security Protocols, July 1999.

[FAB98] F. J. T. Fabrega, J.C. Herzog, and J. D. Guttman. *Strand Spaces: Why is a Security Protocol Correct?.* Proceedings 1998 IEEE Symposium on Security and Privacy, May 1998.

[GAA90] K. Gaarder, and E.Snekkenes. *On the Formal Analysis of PKCS Authentication Protocols*. Advances in Cryptology-AUSCRYPT'90, Springer-Verlag, 1990.

[GON90] L. Gong, R. Needham, and R. Yahalom. *Reasoning about Belief in Cryptographic Protocols.* Proceedings of the 1990 IEEE Computer Society Symposium on Research in Security and Privacy, 1990.

[IEE01] IEEE. *Standards for Local and Metropolitan Area Networks: Standard for Port Based Network Access Control*. IEEE Draft P802.1X/D11, March 2001.

[KAL93] B. Kaliski. *Privacy Enhancement for Internet Electronic Mail: Part-IV: Key Certification and Related Services*. RFC 1424, RSA Labs., February 1993.

[KEM89] R. A. Kemmerer. *Analyzing Encryption Protocols Using Formal Verification Techniques.* IEEE Journal on Selected Areas in Communications, vol. 7, no. 4, pp. 448-457, May 1989.

[KEL98] J. Kelsey, B. Schneier, and D. Wagner. *Protocol Interactions and the Chosen Protocol Attack.* Security Protocols, 5th, International Workshop April 1997, Proceedings, pp. 91-104, Springer-Verlag, 1998.

[KEN93] S. Kent. *Privacy Enhancement for Internet Electronic Mail: Part-II: Certificate-Based Key Management*. RFC 1422, BBN Communications, February 1993.

[LOW96] G. Lowe. *Breaking and Fixing the Needham-Schroeder Public Key Protocol Using FDR.* In Proceedings of TACAS, vol.1055 of LNCS, pp. 147-166, Springer-Verlag, 1996.

[LOW98] G. Lowe. *Casper: A Compiler for the Analysis of the Security Protocols.* Journal of Computer Security, vol.6, pp. 53-84, 1998.

[MAR03] J. D. Marshall. *An Analysis of the Secure Routing Protocol for Mobile Ad-Hoc Network Route Discovery: Using intuitive Reasoning and Formal Verification to Identify Flaws*. Technical Report TR-030502, Florida State University, 2003.

[MEA95] C. Meadows. *Formal Verification of Cryptographic Protocols.* A survey in: Proceedings of Advances in Cryptology-Asiacrypto'94. Lecture Notes in Computer Science, Springer-Verlag, vol. 917, pp.133-150, 1995.

[MEA96] C. Meadows. *The NRL Protocol Analyzer: An Overview.* Journal of Logic Programming, vol.26, no.2, February 1996.

[MEA99] C. Meadows. *Analysis of Internet Key Exchange Protocol Using the NRL Protocol Analyzer.* Proceedings of the 1999 Symposium on Security and Privacy. IEEE Society Press, May 1999.

[MEA00] C. Meadows. *Open Issues in Formal Methods for Cryptographic Protocol Analysis.* Proceedings of DISCEX 2000, pp.237-250, IEEE Computer Society Press, 2000.

[MEA03] C. Meadows. *Formal Methods for Cryptographic Protocol Analysis: Emerging Issues and Trends.* IEEE Journal on Selected Areas in Communication, vol. 21, no.1, pp.44-54, January 2003.

[MIS02]A. Mishra, and W. A. Arbaugh. *An Initial Security Analysis of the IEEE 802.1X Standard.* Technical Report CS-TR-4328, University of Maryland, February 2002.

[MOS89] L. Moser. *A Logic of Knowledge and Belief for Reasoning About Computer Security.* In Proceeding of the Computer Security Foundations Workshop II, pp. 57-63, IEEE Computer Society Press, June 1989.

[MOV96] A. J. Menezes, P.C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996

[PAU97] L. C. Paulson. *Mechanized Proofs of Security Protocols: Needham-Schroeder with Public Keys. Technical Report 413, University of Cambridge, Computer Laboratory, January 1997*.

[RAN88] P. V. Rangan. *An Axiomatic Basis of Trust In Distributed Systems.* In Proceeding of the 1988 Symposium on Security and Privacy, pp. 204-211, IEEE Computer Society Press, April 1988.

[RIG02] C. Rigney's, S.Willens, A. Rubens, and W.Sipmson. *Remote Authentication Dial In User Service (RADIUS).* RFC 2865, June 2000.

[RUB93] A. Rubin, and P. Honeyman. *Formal Methods for the Analysis of Authentication Protocols.* Technical Report, CITI TR 93-7, October 1993.

[SCH96a] M. S. Schneider. *Security Properties and CSP.* In IEEE Computer Security Symposium on Security and Privacy, IEEE Computer Society Press, 1996.

[SCH96b] M. S. Schneider. *Using CSP for Protocol Analysis: The Needham-Schroeder Public key Protocol.* CSD-TR-96-14, Royal Holloway, University of London, 1996.

[SON01] D. Song, S. Brezin, and A. Perrig. *Key Athena: A Novel Approach to Efficient Authomatic Security Protocol Analysis.* Special issue of Journal of Computer Security, vol.9, no.1, 2 pp.47-74, 2001.

[STU01] A. Stubblefield, J. Ioannidis, and A. D. Rubin. *Using The Fluhrer, Mantin, and Shamir Attack to Break WEP.* ATT Labs Technical Report, TD4ZCPZZ, Revision 2, August 21, 2001.

[SYV90] P. Syverson. *Formal Symantics for Logics of Cryptographic Protocols.* In Proceeding of the Computer Security Foundations Workshop III, pp. 32-41, IEEE Computer Society Press, June 1990.

[WIN98] J. M. Wing. *A Symbolic Relationship Between Formal Methods and Security.* Technical Report, CMU-CS-98-188, 1998.

[WOO93] T. Y. C. Woo, and S. S. Lam. *A Semantic Model for Authentication Protocols.* In Proceedings of the 1993 IEEE Symposium on Research in Security and Privacy, pp. 178-194, IEEE Computer Society Press, May 1993.

[YAH93] R. Yahalom, B. Klein, and T. Beth. *Trust Relationship in Secure Systems: A Distributed Authentication Perspective.* In Proceeding of the 1993 IEEE Syposium on Security and Privacy, pp. 150-164, IEEE Computer Society Press, May 1993.

[YAS96] A. Yasinsac. *Evaluating Cryptographic Protocols.* PhD Thesis, University of Virginia, 1996.

[YAS99] A. Yasinsac, and W. A. Wulf. *A Framework for a Cryptographic Protocol Evaluation Workbench.* Proceedings of the Fourth IEEE International High Assurance Systems Engineering Symposium (HASE99), Washington D.C., Nov 1999.

[YAS00a] A. Yasinsac. *Active Protection of Trusted Security Services.* Florida State University, Computer Science Technical Report TR 000101, Jan 2000.

[YAS00b] A. Yasinsac. *Detecting Intrusions in Security Protocols.* 1st ACM Conference on Computer and Communications Security Workshop on Intrusion Detection, November 1-4, 2000.

BIOGRAPHICAL SCETCH

ILKAY CUBUKCU

Ilkay Cubukcu was born on November 3, 1971 in Afyon, Turkey. She graduated from Cay High School, Afyon, Turkey in June 1988. In June 1993, she earned a BS degree from the Technical University of Istanbul in Istanbul, Turkey in Meteorological Engineering. From September 1994 to June 1996, she thought in various schools in Turkey while she was working on her teaching degree from Yildiz Technical University in Istanbul. From February 1997 to August 2000, she worked as a research assistant at NHMFL (National High Magnetic Field Laboratory) in Tallahassee, FL. She started Computer Science master's program at Florida State University and worked as a teaching assistant with a full scholarship from August 2000 to August 2002. In May 2001, she joined security group of CS Department at FSU. From August 2002 to July 2003, she worked as research assistant at COAPS (Center for Ocean-Atmospheric Prediction Studies) at FSU.

Her research interest is on wireless network security. Ilkay co-authored the paper titled "A Family of Protocols for Group Key Generation in Ad Hoc Networks" presented at Proceedings of the IASTED International Conference on Communications and Computer Networks (CCN02) in 2002.