Ministry of Higher Education and

Scientific Research
University of Babylon
College of Science for Women
Department of Computer Science

ANALYSING SECURITY PROTOCOLS USING SCENARIO BASED SIMULATION

A Project

Submitted to the Council of College of Science for women University of Babylon in Partial Fulfillment of the Requirements for the Degree of Higher Diploma in Science / Computer Science

By

Enas Hassan Abade Rasol

Supervised By

Dr. Farah Al-Shareefi

2021 A.D. **1443 A.H.**

بسم الله الرحمن الرحيم

{وَمَن يَتَّقِ اللَّهَ يَجْعَل لَّهُ مَخْرَجًا وَيَرْزُقْهُ مِنْ حَيْثُ لَا يَحْتَسِبُ وَمَن يَتَوَكَّلْ عَلَى اللَّهِ فَهُوَ حَسْبُهُ إِنَّ اللَّهَ بَالِغُ أَمْرِهِ قَدْ جَعَلَ اللَّهُ لِكُلِّ شَيْءٍ قَدْرًا}

صدق الله العلي العظيم

سورة الطلاق / أية {2 . 3 }

Supervisor Certification

I certify that this research titled **"Analysing Security Protocols Using Scenario Based Simulation "**

was prepared at the Department of Computer Sciences/ College of Science for the Women/ University of Babylon, by (**Enas Hassan Abade Rasol**) as partial fulfillment of the requirements for the degree of Higher Diploma of Science in Computer Science .

Signature:

Name: Dr. Farah Al - Shareefi

Date: / / 2021

Address: University of Babylon/College of Science for Women

The Head of the Department Certification

In view of the available recommendations, I forward the research entitled **" Analysing Security Protocols Using Scenario Based Simulation "** for debate by the examination Committee .

Signature:

Name : Dr. Farah Al - Shareefi

Date: / / 2021

Address: University of Babylon/College of Science for Women

Certification of the examing committee

We, the member of the examing committee, certify that we have read this project entitled (ANALYSING SECURITY PROTOCOLS USING SCENARIO BASED SIMULATION) **and after examining the higher diploma student** (Enas Hassan Abade Rasol) **in its content in 28\12\2021, and that in our option and it is accepted as a project for the degree of higher diploma in science\computer science with a degree** *(verygood).*

Committee chairman:

Signature:

Name: ***SAIF MAHMOUD KHALAF***

Scientific order:

Date: \ \2021

Committee member:

Signature:

Name: ***ISRA MOHAMED ABED***

Scientific order:

Date: \ \2021

Committee member (supervisor):

Signature:

Name: ***FARAH MOHAMED AL-Shareefi***

Scientific order:

Date: \ \2021

Date of examination: \12\2021

Deanship authentication of collegeof science for women.

Approved for the college committee of grade studies.

Signature:

Name: ***Faez Ali Rashid***

Scientific order: Prof. Dr.

Address: Dean of college science for women **Date: \ \2021**

Dedication

To: the believing messenger and the saving human (Mohamed (P.B.U.G)).

To: my Parents .

To : all who stood by me in my research .

To: my professors at the College of Science for Women .

Enas Hassan Abade Rasol

Acknowledgments

I am grateful to God Almighty and his Prophet Mohammed (PBUH),to begin with .

I also extend my deepest thanks and appreciation to

Dr. Farah Al- Shareefi which had the main role in Supervising

my research and guiding me in every step .

I extend my sincere thanks and appreciation to all my esteemed

professors in the Department of Computer Science / College

of Science for Women .

Enas Hassan Abade Rasol

List of Contents

List of Figures

List of Tables

List of code

List of Abbreviations

Abbreviation	Meaning
ASM	**Abstract State Machines**
pk	**public key**
prk	**Private key**
PUBK	**A public key**
SECK	**A private key**
SHRK	**A shared key**
NC	**A nonce**
ENCR	**An encrypted component**
ID	**An identity**
MITM	**Man-in-the-Middle**
NS	**Needham Schroder**
AS-RPC	**Andrew Secure Remote Procedure Call**
DASM	**Distributed Abstract State Machines**

Abstract

This project is an approach for utilizing Analysing Security Protocols Using Scenario Based Simulation.

A possible attack scenario defines the flow of communications but not their messages' content. Scenarios can help decrease the number of protocol's runs that must be investigated during attack simulation. It is possible to reduce the number of runs even more by lessening the number of messages generated by intruders. The capability of the intruder to form messages is limited by the different factors including : type matching and anticipated message content.

The method presented in this study utilizes two tools that underpins the Abstract State Machines method : AsmetaL for building models and AsmetaS for achieving the simulation. To demonstrate the efficacy of the approach, several protocols are inspected.

CHAPTER ONE

GENERAL INTRODUCTION

1.1 Introduction

A great number of formal analysis methods have been proposed for building accurate and secure models of protocols. Majority of these methods, which rely on the Dolev-Yao model of an intruder [1], and they postulate the following: (1) based on the intruder's background information, the intruder can create whatever message it wants; and (2) an unrestricted number of sessions can be launched by the intruder.

The first assumption may cause some undue messages with erroneous type or format, which will most are likely to be dismissed by the truthful participants. The second assumption, when combined together with the first, typically gives rise to the undecidable verification issue because of the unlimited state space to be inspected. Several tools, such as ProVerif and Tamarin [2,3], have managed the state space explosion problem, though at the price of possible non cessation of the analysis process, especially when the non-trivial algebraic properties, like commutative encryption, are used.

The authors of [4] cope with the unanswerable verification problem through simulating security protocols with scenarios tailored for attacks of the destination and message origin type [5]. A scenario is a protocol's run made up of many sessions wherein the pattern of arranging the protocol steps for the associated sessions is defined. This is an active way for lessening the number of protocol runs, when searching for an attack through simulation.

This research focuses on the implementing the above method by applying an efficient formal method which is called the Abstract State Machine (ASM) method [6].

ASMs are a universal state machine formalism that can be used to model any algorithm at an adequate level of abstraction. The ASM method has been chosen for the following reasons: 1) the method's generality, which means that the ASM method can be used to define any system, at a needed level of abstraction. 2) the ASM method has a rich and simple syntax, as well as precise semantics, which together will aid in the specification of models that balance between abstraction and accuracy. 3) this method possesses an executable language called ASMETA Language (AsmetaL) [7] to describe the protocol, intruder, and attack scenarios, as well as a simulator tool called ASMETA Simulator (AsmetaS) [8] to simulate the protocol runs.

This work analyzes the Andrew Secure Remote Procedure Call Protocol [9] and the Needham-Schroder (NS) public key protocol [10] as examples.

1.2 Problem Statement

Despite the fact that the simulated scenarios based on attack schemes method have been developed to address the undecidable verification issue, the message space with this method is still increased with undesirable messages resulting in the undecidabilty issue again. The main reason is that, the intruder in this method creates messages depending only on the assumed message's type format without considering the anticipated message content. This accordingly will explode the message space with superfluous messages that consequently leads to increase the protocol runs. Furthermore, this method is executed by the Estelle description language which is incomprehensible by different classes of people.

1.3 Research Objectives

This research project aims at tackling the above mentioned problems by:

1. Formulating a principle for the intruder to construct messages based on the receiver's expectations about the message type, format, and content.

2. Employing the idea of attack pattern scenarios in a de facto verification process depending on simulation wherein the ASM methodology is applied.

1.4 Related Work

For the analysis of security protocols, a number of formal methods were utilized. Tools, like Spin and NuSMV model checking, have been used to automatically verify security protocols, for example papers in [11,12]. In a successful way, these tools demonstrate that insecure states are unattainable, but only with a bounded assumptions for creating protocol sessions or generating fresh components or protocol sessions.

Regarding the unlimited analysis issue, ProVerif [2], is a common tool that has evolved to handle this. Typically to analyze an unlimited sessions, ProVerif utilizes rules of Prolog style and implements an abstract representation approach. It uses applied pi calculus as its specification language. ProVerif tries to prove two properties: secrecy and correspondence. However, ProVerif may enter an endless loop, resulting in a stack overflow, for particular protocols, such as the TP protocol, and some algebraic properties.

Recently, the Tamarin tool is developed as a tool for symbolically model checking protocols to manage the unlimited analysis [3]. Tamarin's specification language is built on multiset rewriting rules. In terms of describing security properties, Tamarin's language is more expressive than ProVerif, since it enables

direct modeling for temporal aspects. Despite this, it has the same non-termination restriction as ProVerif.

The ASM approach has been used in theory to design and analyze security protocols while modeling intruder capabilities [13,14]. This work, however, does not address anticipated content when constructing messages.

This study is most similar to the works [15] and [4]. The first one limits the intruder's capability to construct just a specific types of messages that can be approved by certain agents. It is possible that relying just on type matching will still result in undesirable messages. The authors in [4] describe an approach for simulating security protocols employing scenarios tailored for several attack schemes. This is executed by using the Estelle language. The findings revealed that it is a fruitful approach for diminishing the number of runs related to a protocol under analysis. In this work, however, the intruder uses the approach of [15] to generate messages.

1.5 Project Outline

This project is organized as follows: Chapter Two covers the ASM technique and security protocols. Chapter Three explains our methodology in details. Chapter Four debates on the project's findings, while Chapter Five wraps up the project.

CHAPTER TWO

THEORETICAL BACKGROUND

2.1 Introduction

This chapter presents some underlying information of the security protocols, and the applied formal method, that is known as Abstract State Machines (ASMs).

2.2 Cryptographic Primitives

Since constructing and developing security protocols necessitates the use of cryptographic primitives in order to fulfill their security-related objectives (authentication, integrity, confidentiality, etc.), this section briefly defines these primitives.

Cryptographic primitives can be defined as algorithms for safeguarding messages sent over the internet and accomplishing protocol services [16]. A cryptography algorithm applies mathematical functions to encrypt, decrypt, or hash sensitive messages. In general, an encryption operation is the process of turning plain-text for a certain message into cipher-text text. Decryption, on the other hand, is the process of converting cipher-text back to its original plain-text. Hashing is an irreversible process for creating a reduced fixed-length output from an input text of unknown length.

The following cryptographic primitives are differentiated based on the number of keys required for encryption and decryption operations:

- **Symmetric Key Cryptography** is a traditional cryptography in which a single key is used for both encryption and decryption [17]. The key is usually kept hidden and shared between two communication parties. The symmetric encryption key must be exchanged between the participating parties before it can be utilized for decryption.

- **Public Key Cryptography (Asymmetric Key Cryptography)** is a kind of cryptography at which no secret key is disclosed between participants [18]. Usually, public key cryptography employs two keys: a public key that is known to all communication parties, and a private key that is only known to its actual owner. The encryption process in public key cryptography is open to everyone who has a known public key, while the decryption process is only available to the owner of the related private key. Some public key paradigms enable you to encrypt using the private key and decrypt with the related public key.

1.3 Notation of Security Protocols

In view of readability, we utilize the fundamental notation known as Common Syntax(CS) [19], which is widely used in the literature to define security protocols [20]. A protocol is a set of steps for exchanging messages in a specific order. Each step has the following structure:

$$i.P_i \rightarrow Q_i: M_i$$

where i is the i^{th} protocol's step, such that $1 \leq i \leq n$, n is a total number of protocol steps, P_i is the presumed sender of step i, Q_i is the presumed receiver of step i, M_i is the message of that step. P_i and Q_i might be legal participants with identities, such as $A, B, C, D,$ and so on, or a trusted third-party server that has the identities, such as S, S_0, S_1, and so on.

A message may have one or more encrypted or unencrypted components (also called fields) that are encrypted using either a public key or symmetric key encryption technique. The unencrypted component might be an identity, a key, a fresh component like timestamps or nonces, or just plain-text. The encrypted

component, commonly abbreviated as $\{m\}_k$, to represent a protocol message or sub message that has been encrypted using the key k. The following notations are used to express the key: $pk(P)$ and $prk(P)$ indicate the public and private key, respectively, of a participant, whereas $ssk(P,Q)$ signifying a symmetric key owned P and Q participants.

Commonly, in the protocol specification, any message component is represented as variables. A **single session** of a protocol is an instantiation of these variables to generate a sequence of concrete identities and message content. A protocol session is the generation of a sequence of concrete identities and message content by instantiating these variables. **Multiple sessions** are a number of (potentially interleaved) instantiated sequences with their unique identifier numbers for differentiating them. From a constant number of potentially simultaneous sessions, a **protocol run** is resulted. A protocol run represents a concrete instantiation of a series of protocol steps.

2.4. Protocol Attacks

A protocol run that satisfies a harmful property, such as disclosing a secret, is referred to as an **attack** on a protocol. A deceitful participant known as intruder with I identity launches an attack on a protocol. The $I(X)$ notation denotes that the honest participant is impersonated by the intruder.

An abstracted protocol run is called an **attack scenario** that is specified based on known attacks and without considering the message content. Indeed, excluding the repetition situation of already delivered messages that can be specified, a scenario does not define message content. As shown in Figure 2.1, a scenario has predetermined information about assigning participant identities and intruder impersonations to each role of participants. A Man In The Middle (MITM) atatck

scenario is depicted in Figure 2.1. In the left column of this figure, a pattern of arranging protocol steps for the involved sessions has been specified, including the assignment of identities and intruder impersonation to each role of participants. The right column, on the other hand, has a meta-variable M_i, which has a value supplied by the intruder or the honest participants.

$$
\begin{aligned}
&1.1\ A \rightarrow I &&: M_1\\
&2.1\ I(A) \rightarrow B &&: M_1\\
&2.2\ B \rightarrow I(A) &&: M_2\\
&1.2\ I \rightarrow A &&: M_2\\
&1.3\ A \rightarrow I &&: M_3\\
&2.3\ I(A) \rightarrow B &&: M_3
\end{aligned}
$$

Figure 2.1 An example of scenario for MITM attack

The intruder who has total control over the communication between the honest players can launch several skilled attacks against protocols. The following are the major categories that these attacks fall under.

- **Man in the Middle (MITM) Attack**: A Man in the Middle (MITM) attack is one in which an intruder surreptitiously intercepts, replays, and modifies communications between two participants, making them believe they are only conversing with each other. An intruder is present between the two participants in this attack.
- Simple Replay (Simple REPL) Attack. With this type, any eavesdropped message or sub-message from previous sessions is fraudulently replayed by the intruder. In fact, the first session is conducted without impersonation, but the second session involves impersonation when the non-fresh message is detected and delivered.

2.5 Examples of Security Protocols

This section provides two protocol examples which have different goals.

2.5.1 Needham-Schroder Public-Key Protocol

The Needham-Schroder Public key (NSPK) protocol [21], or simply (NS), is developed with the aim of accomplishing mutual authentication between A and B participants, as shown in Figure 2.2 (a). Because these participants believe that they only know about the freshly produced nonces N_A and N_B, A and B are sure that they are interacting with the intended correspondent in this protocol. This is erroneous, because the protocol is vulnerable to the MITM attack [22] depicted in Figure 2.2 (b).

$$1.\ A \rightarrow B : \{A, N_A\}_{pk(B)}$$
$$2.\ B \rightarrow A : \{N_A, N_B\}_{pk(A)}$$
$$3.\ A \rightarrow B : \{N_B\}_{pk(B)}$$

(a) The NS protocol's messages

$$1.1\ A \rightarrow I : \{A, N_A\}_{pk(I)}$$
$$2.1\ I(A) \rightarrow B : \{A, N_A\}_{pk(B)}$$
$$2.2\ B \rightarrow I(A) : \{N_A, N_B\}_{pk(A)}$$
$$1.2\ I \rightarrow A : \{N_A, N_B\}_{pk(A)}$$
$$1.3\ A \rightarrow I : \{N_B\}_{pk(I)}$$
$$2.3\ I(A) \rightarrow B : \{N_B\}_{pk(B)}$$

(b) The Lowe attack on NS protocol

Figure 2.2 The NS protocol example

2.5.2 Andrew Secure Remote Procedure Call Protocol Example

The Andrew Secure Remote Procedure Call (or simply AS-RPC) protocol [9], depicted in Figure 2.3 (a), is a protocol which employs the symmetric key encryption method [17]. Its purpose is to deliver a new session key which is $ssk(A, B)$' to two participants A and B who already have a $ssk(A, B)$ key.

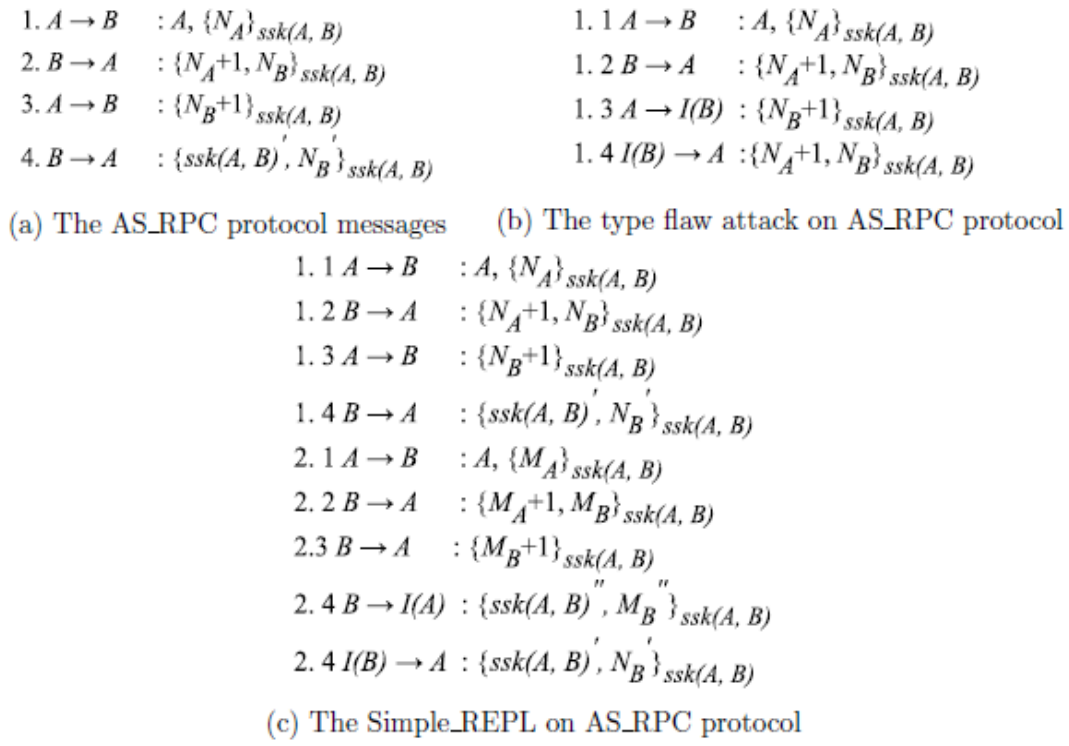

Figure 2.3 The AS-RPC protocol example

Despite the fact that this protocol appears to be well-designed, it contains two flaws. The first is an implementation flaw [19, 23], whereas the second is a freshness flaw [24]. In particular, according to [19], in case that a same length bit sequence is used for both the $N_A + 1$ nonce and the $ssk(A,B)'$ key, then the protocol is vulnerable to the type flaw attack, in which the NA +1 may simply be inserted as the new session key, as shown in Figure 2.3 (b). According to [24], there is no assurance that the new shared key $ssk(A,B)'$ is realy fresh and was produced in the same session. As a result, the intruder can easily replay the fourth message in a different session, as shown in Figure 2.3 (c).

2.6 The Applied Formal Method: Abstract State Machines

Abstract State Machines (ASMs) is a rather new formal method that is originally introduced by Yuri Gurevich [25, 26] as a universal formalism for building a model to any algorithm at an acceptable level of abstraction. He developed the basic hypothesis of ASMs formalism as: each algorithm, regardless of the way for abstracting it, is emulated step-by-step by a suitable ASM. The concept of abstract states gives ASMs their generality. In classical state machines, like Turing machines and finite state machines, states have symbolical representation via a collection of symbols. Abstract states, on the other hand, have syntactical and semantical representation that is achieved by mathematical structures of items from domains with defined functions and predicates.

ASMs also have transition relations that are defined by rules for describing the updating of the function interpretations from one state to the next. ASM specifications specifies how the system's state changes over time as a result of transition rules. ASMs are being farther upgraded into a practically and arithmetically well-founded method for designing and analyzing a high-level system [27, 28]. For real-world problems, this method links the gap between human comprehension, formalization, and executable state machines. ASMs enhance the development process of a system by establishing a precise high-level mode that is destined for producing executable code. ASMs have a wide range of useful applications, including specifying different systems, such as sequential, parallel, and distributed [27], specifying dynamic databases [29], determining the specification for programming languages like UML, Java, and Prolog [30, 31, 32, 33, 34], demonstrating compiler correctness [35], modeling and simulating ecurity protocols [36, 37, 38], and so on.

The ASM method is based on three fundamentals concepts:

- **Basic abstract states** are mathematical abstractions which are relied on abstract states, to define the systems structure, and on transition rules, to specify the dynamic behavior of systems;
- **A ground model** is a mechanism for capturing the determined requirements of a systems through a succinct and rigorous conceptual model;
- **Stepwise refinement** is a general approach to building a hierarchical refined models from an abstract ground model depending on the design decision. The final produced model is more comprehensive and implementation-linked one.

2.6.1 The First Concept: Basic Abstract State Machines

ASMs, or basic ASMs, were created to describe the situation in which a single agent can perform multiple actions at the same time. Later, as discussed in Section 2.6.4, this concept is extended to distributed multi-agents that taking action and reaction in a concurrent or no concurrent pattern.

Basic ASMs are formally defined as finite sets of move rules with the following form:

$$\textbf{If } Condition \textbf{ Then } Updates$$

This is the key form for transiting from one abstract state into another. The Condition (or guard) is a predicate formula that has a first-order relation with true or false interpretation. While, the term "Updates" refers to a limited number of update functions of the following:

$$f(t_1, \dots, t_n) \coloneqq t$$

where f is the selected name of an n-ary function, $(t_1, \dots, t_n)$ are first-order arguments (terms) of of a function, t is the value of updating a function. In other

words, when a finite set of functions changes their values, the transition occurs from one state to another. A basic ASM is made up of four parts:

- **A Signature (Σ):** denotes the desired definitions functions and domains.
- **Initial states** (I): are a set of states specified by imposed conditions on the signature.
- **Transition rules (TR):** are a set of relations that defines sets of update dedicated for abstract states.
- **Main rule (R):** represents a main machine's transition rule without arity.

Illustrated Example

This section shows an illustrated example, called Automated Train Door (ATD) [39], in order to familiarize the reader with ASMs formalism.

With ATD system, a train's physical door is controlled by a computerized controller, which receives sensitive data and issues commands. The ATD system is typically made up of the following parts: a) computerized controller; b) door_sensor; c) train_sensor; d) emergency_sensor; e) actuator; and f) door. The operating principle of the ATD system is best comprehended in terms of its parts. The computerized controller handles the sensor inputs to give an open or close door command. The door sensor transmits data about the position of the door and the presence of obstacles. The train sensor sends out signals that show the train's motion and position in relation to the platform. When an emergency situation arises, such as a fire or toxic gas, the emergency sensor sends out an emergency sign. The actuator operates the door in accordance with the issued control command. The requirements state that the door should not close on a person who is standing in the doorway, that the door should not be opened while the train is in movement situation or when it is not completely aligned or positioned with a

platform, and that the passengers should be able to escape in the event of an emergency.

Vocabulary Employed in ASMs

This section explains some vocabulary used in ASMs to help at understanding the way of executing this machine.

- **Domain**

A domain (also known as a universe) is a "finite or infinite" collection of items used by a machine. An ASM state $\mathfrak{S}$ has a super domain or super universe $\mathfrak{D}$ that is partitioned into smaller domains belonging to certain categories.

- **Function**

In ASMs, a function is defined via its name and parameters using the below form:

$$f(t_1, \dots, t_n)$$

where f is a name selected to a function, and $t_1, \dots, t_n$ are the n parameters of a function; n is the number of function parameters (also called the function's arity). Functions without arity (i.e., zero arity or nullary functions) are called constants. A selected name of a function is established in the signature part. When the parameters t_i of f are evaluated to their new values, for example v_i, and an $f(v_1, \dots, v_n)$ is evaluated to v in the current state, then the function f is updated to v as well to be the new value of this function in the next state. A function name together with its associated parameter's values account for a location. Typically, a value of a location is deduced from a pair of function's value and its indicated parameter values.

Functions are divided into two categories: **Basic functions** and **Derived functions**. Derived functions are supplementary functions that have a specification or computational design to return values for presented read-only parameters. The basic functions differ from the derived functions in that they do not have specific specification for updating their values; rather, they are updated by the machine (its rules), the environment (the machine's user), or both. Basic functions are more divided into static and dynamic functions, depending on how their values are updated.

Static functions have values that never change during machine execution, i.e. they have constant values.

Dynamic functions are functions that have values changing from one state to the next, i.e., these functions are similar to variables in usual programming language. Dynamic functions are divided into three categories: controlled functions, monitored functions, and shared functions. The dynamic functions that can only be updated and read through the machine itself is called **controlled functions**. While when the values of dynamic functions can be read from the external environment and written by the machine's user, then the dynamic functions are known as **monitored functions**. The dynamic **shared functions**, on the other hand, are updated and read by the rules of the machine together with its external environment.

- **Signature**

A signature is a fixed set of names related to functions and domains that must be declared. There is a finite number of arity for each function name. The Boolean function's values: *True* and *false*, and under-defined function: *undef* represent 0-ary (zero arity) functions that are presumed in the signature.

When creating an ASM, the signature is defined. For example, the signature of the ATD example is shown in Figure 2.4. In this Figure, the listed domains: *Availability*, *MotionStatus*, *PositionStatus*, *DoorStatus*, and *Status*, reflect the availability situation of both an emergency and obstacle: *existing* or *not*, the state of train's motion: the train is in *moving* or *stopped* situation, the possible set of train's position: the train is *aligned* with the platform or *not*, the potential set of the door's states: the door is *opened*, the door is *closed*, the door is *opening*, and the door is *closing*, and the feasible operational situation of the controller: *sensing* information or *implementing* a command.

The functions: *obstacleSensor*, *emergencySensor trainMotionSensor*, and *trainPositionSensor* represent monitored predicates for denoting the presence of an obstacle and an emergency in the external environment, as well as the input of the environment that relates to the train's motion and position, respectively. The controlled functions: *state, emergency, doorStatus, obstacleStatus, trainPosition, and trainMotion*, indicate the mode of the operation, the emergency, and the up to date state of the door, the obstacle's state at the doorway, the position of the train and its motion, respectively. The 0-ary functions which are *sensing* and *executing* are employed to refer to an atomic object that is a member of the *Status* Domain. The derived function: *safeSituation* returns *true* value; in case that the door is opened or it is opening in an emergency situation.

EXAMPLE (Signature):
Domains
Status
DoorStatus={OPENED, CLOSED, OPENING, CLOSING}
PositionStatus={NOT_ALIGNED, ALIGNED}
MotionStatus={STOPPED, MOVING}
Availability={EXIST, NOT_EXIST}
Monitored Functions
trainMotionSensor: *MotionStatus*
trainPositionSensor: *PositionStatus*
emergencySensor: *Availability*
obstacleSensor: *Availability*
Controlled Functions
doorStatus: *DoorStatus*
trainMotion: *MotionStatus*
trainPosition: *PositionStatus*
emergency: *Availability*
obstacleStatus: *Availability*
state: *Status*
Static Functions
sensing: *Status*
executing: *Status*
Derived Functions
safeSituation: *Boolean*
where
safeSituation $\Longleftrightarrow$
$\exists s \in$ {OPENED, OPENING}: *doorStatus*= s

Figure 2.4: The signature of the automated train door example

- **State and Set of Update**

A *state* $\mathfrak{S}$ for a signature $\sum$ is an arithmetical structure incorporating: the superdomain $\mathfrak{D}$, together with the evaluation of terms and formulas defined for every function name assumed in $\sum$. An update of $\mathfrak{S}$ is manifested as the blow form:

(*loc*, *v*)

Where *loc* is a state's location, and *v* is an item from the D. A collection of updates is called an ***update set***.

- **Rules**

In ASMs, a rule (also called transition rule) constitutes an update set on transition states to characterize the behavior of the system. A rule can be either one of the basic rules listed below, or a complex rule made up of several basic rules. The following are the basic TR transition rules:

- **Skip**: It results in an update set that is empty.
- **Updating ($f(t_1, \dots, t_n) := t$)**: the task of this rule is updating the value the function $f(t_1, \dots, t_n)$ to the value t, such that $t_1, \dots, t_n$, and t are terms of first-order calculus.
- **Block (*M par N*)**: It simultaneously evaluates *M* and *N* rules and causes unified update sets calculated by *M* and *N*.
- **Sequential (*M seq N*):** It implements the rules *M* and *N* in a successive style, begining with *M*.
- **Conditional (*if Q then M otherwise N*)**: It examines whether a Boolean expression, which is *Q*, has a true value, then it implements the *M* rule, otherwise it implements *N* rule.
- **Let (*let x=t in M*):** It assigns the *t* value to a logical variable *x*, and implements *M*. The produced update set is the set calculated by *M*.
- **Forall *(forall x with Q do M)***: It simultaneously implements the *M* rule for each *x* satisfying *Q*.
- **Choose (*choose x with Q do M ifnone N*)**: It selects *x* that meets a given condition *Q* and then implements the *M* rule. However, it just implements *N* when there is no x satisfying *Q*.
- **Call ($r(t_1, \dots, t_n)$)**: It implements the rule r with its presented parameters $t_1, \dots, t_n$, such that this rule is specified earlier.

The machine has also a rule with zero arity, called main rule, that illustrates only one step of ASM. This rule is executed iteratively. Perhaps, a main rule fails or soundly terminates when there is no new update set is yielded or when no rule is applicable.

To illustrate the above rules, the ATD example is used for this purpose. This example has one main rule and seven descriptive complex rules. From the complex rules, four rules are dedicated to update the state of the door, i.e., *doorState*, that is *CLOSED* at the start. These rules are: *ClosingToClosedOrOpened, OpenedToClosing, ClosedToOpening , and OpeningToOpened*. In addition three out of the seven complex rules are allocated to receive the sensors' data. These rules include: ObstacleSensing, TrainMotionSensing, and EmergencySensing. While he main rule, which has the name: TrainDoorController, is specified to capture the ATD's behavior.

Figure 2.5 lists the main rule for ATD example, see Figure 2.5. In this rule, in case that the current state of the ATD is *SENSING*, then the three dedicated rules for sensing information are called; Otherwise, the ATD system is in EXECUTING state. At this state the emergency must first be examined to handle it through opening the door. When there is no emergency, the four rules that are modeled to change the door's state are called in parallel.

mechanism of ASMs by default [28].

From the four rules, only the ClosedToOpening rule is displayed as it similar to the rest rules, see Figure 2.6. In this rule, the controller first examines whether the door's state is CLOSED and the train motion's state is STOPPED and its position is ALIGNED with the platform, then the door state is updated to OPENING.

the controller updates the state of the door into OPENED, if it is at OPENING state.

```
TrainDoorController =
    if state = SENSING then
      EmergencySensing
      TrainMotionSensing
      ObstacleSensing
      state := EXECUTING
      emergency := NOT_EXIST
     else if emergency=EXIST then
              HandleEmergency
          else
              ClosedToOpening
              OpeningToOpened
              OpenedToClosing
              ClosingToClosedOrOpened
              state:= SENSING
    where
      EmeregencySensing =
          emergency:= emeregencySensor
      HandleEmergency =
          if not safeSituation then
             doorStatus = OPENED
```

Fig 2.5 Train Door Controller's main rule

```
ClosedToOpening =
    if doorStatus = CLOSED and trainMotion = STOPPED and
      trainPosition = ALIGNED then
         doorStatus := OPENING
```

Fig 2.6: The Closed_To_Opening rule for program Train Door Controller

- **Machine Step and Machine Run**

The process of simultaneously firing or executing, firing all transition rules in a given state to produce the next state is referred to as a computation step for a machine. While a run is a finite or infinite series of consecutive states that result from the execution of steps.

2.6.2 The Second Concept: Ground model

A ground model concept represents a conceptual model for capturing informal system requirements in a precise, succinct, pliable, and understandable manner. The ground model can be created graphically by carrying out *control state ASMs*. This state is an ASM that has rules are diagrammatically and textually shown in Figure 2.7, such that the machine stays at a given control state *i*, if the $condition_j$ is not met.

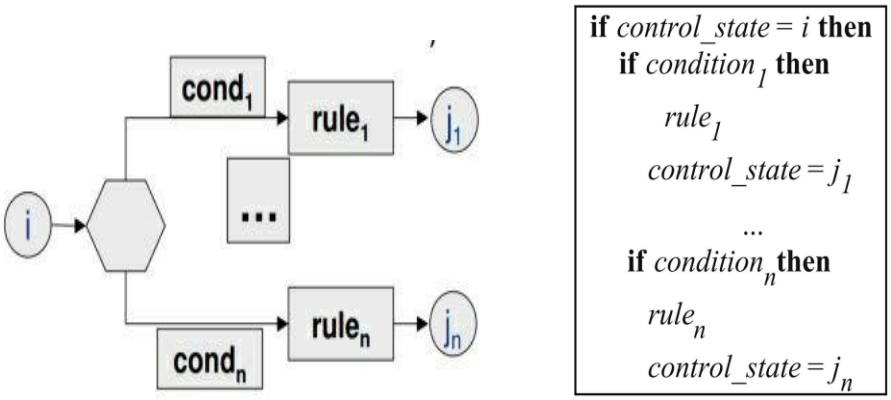

Figure 2.7:Control state ASMs

2.6.3 The Third Concept: Stepwise Refinement

Stepwise Refinement is a method of consecutive refinement that allows a designer to gain a more elaborated model from a brief one. This can be accomplished by fine-tuning (refining) the signature, the flow of operation, or both. At each refining step, the gained model must be proven right in relation to the previous upper one, i.e., the more refined model executes the functionality of the abstract model and its own as well, while maintaining the main system's attributes, such as safety. The ground mode, i.e., the first abstract model, is similar to the system's description perspective (usually informal description), whereas the final refined model is similar to the programmer's perspective (executable code).This

incremental change in system design allows for the detection of silent assumptions and ambiguities in the system's requirements during passing abstraction levels.

2.6.4 Distributed ASMs

The basic ASMs are extended into a Distributed ASMs (DASM) in order to encompass the formalization of multiple agents that use a synchronous and asynchronous way for acting and reacting [27].

With synchronous DASM, each agent runs their own machine in parallel and synchronously, using a global clock for the system, and every agent participates to the global states via the union of each signature related to each machine. The synchronous machine's runs are a sequence of steps that are completely ordered.

Asynchronous DASM presents a united global system perspective for coincident successive computations of each agent per se, such that each agent has its own speed to run its basic ASM in its state without the need for a global clock. In DASM, a computation step performed by a single agent is called move (rather than step. A run for DASM is a partial order set of agent's moves that are partially ordered.

2.6.5 An Executed ASM Tool: ASMETA Framework

Various tools have been developed around ASMs over the last two decades with the aim of providing executable models and underpining the analysis tasks, including: simulation and verification. From all the accessible tools, the ASMETA framework was selected for the context of this project. ASMETA, or the ASM mETAmodelling Framework, is one of the most inclusive modeling and analysis tools designed for ASMs [40, 41]. Recognizing the shortcomings of existing ASM tools, such as restricted coverage of aspects during the complete development

process, expressing ASMs into a language accompanied by syntax that is strictly based on the execution environment, and the inapplicability of unifying ASM tools, the ASMETA framework was created to provide ASMS with an unchanging notation and interoperable tools based environment.

The Java programming language is used to implement ASMETA. On the website [42], the ASMETA framework is available for educational use.

Various tools are included in the ASMETA framework, such as the ASMETA Language (AsmetaL) that has a concrete syntax for ASMs. In addition, AsmetaL is notated in a way which is very similar to both basic ASM and multi-agents ASM [27]. AsmetaL includes a Standard Library, a set of predeclared ASM domains (Integer, Boolean, String, etc.) and different functions declared on those domains, as well as an AsmetaLc compiler for compiling and translating AsmetaL into specifications that can then be implemented by the simulator.

AsmetaL has four sections: the first section is called header that is dedicated for importing the Standard Library and for specifying domains and functions; the second section is known as a body that is devoted for inserting the textual notation of static concrete domain, derived and static functions, and the rules; the third notation is the main rule; and the final section is the initialization for determining the initial values of the controlled functions. For instance, see the AsmetaL specification for the ATD example in Code 2.1.

```
asm TrainDoorController
import StandardLibrary
signature:
  // DOMAINS
  enum domain DoorStatus={OPENED | OPENING
                        | CLOSING | CLOSED}
  enum domain Availability={EXIST | NOTEXIST}
  enum domain PositionStatus = {NOTALIGNED
                              | ALIGNED}
  enum domain MotionStatus = {STOPPED | MOVING}
  enum domain Status = {SENSING | EXECUTING}
  // FUNCTIONS
  controlled state: Status
  controlled doorStatus: DoorStatus
  controlled trainMotion: MotionStatus
  controlled trainPosition: PositionStatus
  controlled emergency: Availability
  controlled obstacleStatus: Availability
  monitored trainMotionSensor: MotionStatus
  monitored trainPositionSensor: PositionStatus
  monitored obstacleSensor: Availability
  monitored emergencySensor: Availability
  derived allowToOpen: Boolean
  derived allowToClose: Boolean
definitions:
  function allowToOpen=
    if doorStatus=CLOSED and
       trainMotion=STOPPED and
       trainPosition=ALIGNED then
            true
    else
            false
    endif
  function allowToClose=
    if doorStatus=OPENED and
       obstacleStatus=NOTEXIST then
            true
    else
            false
    endif
  rule r_closed_to_opening=
   if allowToOpen then
      doorStatus:=OPENING
   endif
  rule r_opening_to_opened=
    if doorStatus=OPENING then
       doorStatus:=OPENED
    endif
  rule r_opened_to_closing=
    if allowToClose then
       doorStatus:=CLOSING
    endif
  rule r_closing_to_closed_or_opened=
  if doorStatus=CLOSING then
    if obstacleStatus=NOTEXIST then
      par
        doorStatus:=CLOSED
        trainMotion:=MOVING
        trainPosition:=NOTALIGNED
      endpar
    else
       doorStatus:=OPENED
    endif
  endif
 rule r_EmergencySensing=
   emergency:=emergencySensor
 rule r_TrainMotionSensing=
   if doorStatus=CLOSED then
    if trainMotionSensor=STOPPED then
      par
        trainMotion:=STOPPED
        trainPosition:=trainPositionSensor
      endpar
    endif
   endif
 rule  r_ObstacleSensing=
   if doorStatus=CLOSING or
      doorStatus=OPENED then
     obstacleStatus:=obstacleSensor
   endif
// MAIN RULE
  main rule r_Main =
    if state=SENSING then
     par
       r_EmergencySensing[]
       r_TrainMotionSensing[]
       r_ObstacleSensing[]
       state:=EXECUTING
     endpar
    else
      if emergency=EXIST then
        doorStatus:=OPENED
      else
        par
          r_closed_to_opening[]
          r_opening_to_opened[]
          r_opened_to_closing[]
          r_closing_to_closed_or_opened[]
          state:=SENSING
        endpar
      endif
    endif
// INITIAL STATE
default init s0:
  function doorStatus = CLOSED
  function trainMotion = MOVING
  function trainPosition = NOTALIGNED
  function obstacleStatus = NOTEXIST
  function emergency= NOTEXIST
  function state = SENSING
```

Code 2.1: AsmetaL specification of the ATD example

Furthermore the ASMETA b framework has another tool called ASMETA Simulator (AsmetaS) for building a run of the specification that need to be

simulated [43]. The AsmetaS underpins different useful tasks across the simulation, such as consistency inspecting in order to find any inconsistent updates, arbitrary simulation for randomly simulating the specification, where the simulator by itself selects the input values in a random way.

CHAPTER THREE

PROPOSED METHODOLOGY

3.1 Introduction

This chapter displays a methodology for analyzing security protocols and designing secure models for them.

3.2 A Proposed Methodology: Analyzing Security Protocols

Our proposed methodology, which is shown in Figure 3.1, comprises two phases: the Formal Specification and *simulation* phases.

- The ***Formal Specification Stage*** serves as a foundation for the subsequent stages. There are four aspects to it:
  (1) The ***Protocol Aspect*** is a modular model designed for any protocol that allows various participants to be added as needed.
  (2) The ***Intruder Aspect*** offers a generic model for intruder behavior, including the capabilities related to update the knowledge and generate the messages.
  (3) The ***Attack Scenarios Aspect*** represents the built-in specifications of attack scenarios into the intruder model to outline its behavior, as inspired by [4].
  (4) The ***Invariant Security Properties Aspect*** specifies the invariant requirements that must be satisfied by the designed protocol model.

  Note that the second and third aspects of any protocol may be utilized straight away without any changes, while the remainder requires just minor adjustments based on the protocol's needs. Sections 3.2.1-3.2.4 go through each aspect in great depth.

- On the other hand, the ***Simulation Stage*** is devoted for analyzing the acquired specifications from the previous stage by using the AsmetaS tool.

This simulation generates protocol runs based on an attack scenario and determines if the invariant properties have been violated. In the event of a violation, we may rectify the problem by returning to our protocol specification.

The aspects related to the formal specification stage are discussed in depth in the below subsections.

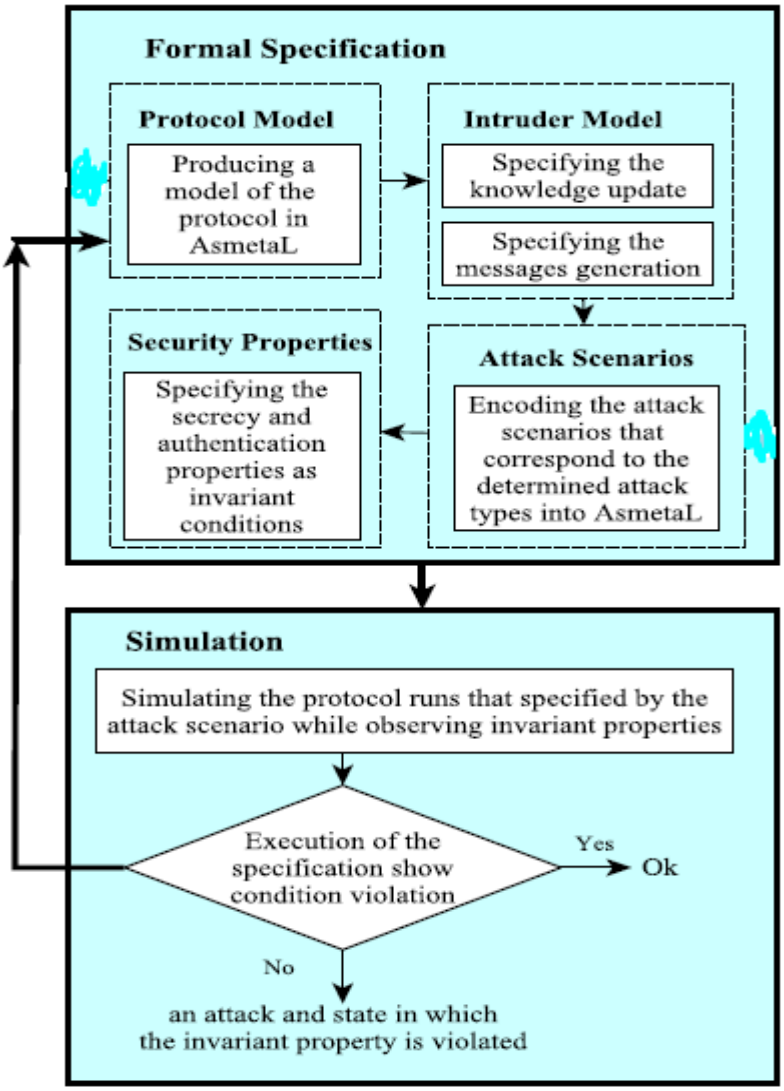

Figure 3.1 The proposed Methodology

3.2.1 Protocol Aspect

The protocol aspect is dedicated for modelling the protocol parties and their key operations for message creation as AsmetaL rules. In this part, we use AmetaL to simulate the NS protocol. To list the protocol names, we establish the domain Protocol=NS, TP. Because each participant has a distinct role, we create a rule for each one, such as the "r_Initiator rule" for the initiator (see Code 3.1). The initial step in creating these rules is to define the domain "Id={AA, BB, II}" which represents the set of IDs shared by all NS protocol participants, including the intruder.

```
rule r_Initiator($InId in Id, $ResId in Id)=
 if state=SEND then
  par
    if wantToInit($InId) and msgNo=1 then
      seq
        start(NS, sNo, 1):=true
        r_GenerateNc[]
        ncInt($InId, $ResId):=n
        r_Encrypt[[toString($InId), n], pk($ResId)]
        r_Send[[toString(cypher)], $InId, $ResId]
        wantToInit($InId):=false
      endseq
    endif
    if msgNo=3 and start(NS, sNo, 3)=false then
      seq
         start(NS, sNo, 3):=true
         r_Encrypt[[at(plainText, 1n)], pk($ResId)]
         r_Send[[toString(cypher)], $InId, $ResId]
      endseq
    endif
  endpar
 else
   if inBox($InId)!=undef and msgNo=2 and finish(NS, sNo, 2)=false then
    seq
      r_Decrypt[first(inBox($InId)), prk(pk($InId))]
      inBox($InId):=undef
      if plainText!=undef and
         ncInt($InId, $ResId)=first(plainText) then
            finish(NS, sNo, 2):=true
      endif
    endseq
   endif
 endif
```

Code 3.1 The “r_Initiator’’ of the NS protocol

In the following step, the control flow of a protocol under specification is determined, for example, determining when a message must be sent. Each participant can be in one of two states: "SEND" or "RECEIVE", depending on

whether they are sending or receiving a message. When the initiator wishes to commence, that is, when the Boolean function" wantToInit(Id)" returns true, the protocol's session begins. The "msgNo" controlled function (in conventional programming languages, a controlled function issimilar to a variable, while a static function is equivalent to a constant) stores a number related to each message. In the same way, the "sNo" function saves the current session number. The "start" function is set to denote that a message pertaining to a certain session is going to be sent.

The final step explains how to define and handle each message. A message is specified as a suqence of Strings . When the participant notices that there is a message in his or her inbox, he or she knows that a message has been received. A created message, on the other hand, is sent by adding this message to the the outBox function. To construct a message, the participant can use the "r_GenerateNc" rule to produce a nonce, see Code 3.2, the "r_Encrypt" rule to encrypt the message, and the "r_Decrypt" rule to decrypt it, see Code 3.3.

```
rule r_GenerateNc=
  extend Nonce with $n do
    n:=$n
```

Code 3.2 Generating a nonce rule

To abstractly model the encryption and nonce generation processes, we use the "extend construct" to augment an abstract domain for the operation outputs with a new element.

```
dynamic abstract domain Cipher
dynamic abstract domain Key
enum domain Id={AA | BB}
controlled plainText: Seq(String)
controlled cipher: Cipher
controlled key: Cipher-> Key
controlled plain: Cipher-> Seq(String)
static pk: Id-> Key
static sk: Key-> Key
static ssk: Prod(Id, Id)-> Key
static pKA: Key
static pKB: Key
static sSKAB: Key
function pk($id in Id)=
  switch $id
    case AA: pKA
    case BB: pKB
  endswitch
function ssk($id1 in Id, $id2 in Id)=
 if (($id1=AA) and ($id2=BB)) or (($id1=BB) and ($id2=AA)) then
    sSKAB
 endif
function sk($k in Key)=
  switch $k
    case pKA: sKA
    case pKB: sKB
    case sSKAB: sSKAB
  endswitch

rule r_Encrypt($m in Seq(String), $k in Key)=
 choose $c1 in Cipher with (plain($c1)=$m and key($c1)=$k) do
     cipher:=$c1
 ifnone
   extend Cipher with $c2 do
     par
        cipher:=$c2
        plain($c2):=$m
        key($c2):=$k
     endpar
rule r_Decrypt($c in String, $k in Key)=
  choose $c1 in Cipher with (toString($c1)=$c and $k=sk(key($c1))) do
     plainText:=plain($c1)
  ifnone
     plainText:=undef
```

Code 3.3 The encryption and decryption operations

The specification signature is firstly introduced in Code 3.3. The "Cipher" is a cyphertext's infinite domain. The "key" for a particular "Cipher" element is represented by the unary function "key". Plain text is represented by the nullary function "plainText", which is a sequence of Strings. The "cipher" belongs to the "CipherText" domain. The "method" is a cipher method that was used to encrypt the plain text message. We define two rules after the signature: the "r_encrypt"

rule, which converts the supplied plain message into an encrypted message using the specified key and method, and the "r_decrypt" rule, which converts the encrypted message into plain text using a specific key and technique. The "r_encrypt" rule initially selects an element in the CipherText domain whose plain text is identical to the supplied message. The cyphertext for the message is represented by this element. When selecting an element returns nothing (the supplied message has not been encrypted previously), then this rule will produce a new cyphertext given its plain text, key, and method, by expanding the CipherText domain. While the "r_decrypt" rule selects a cyphertext item from the CipherText domain that matches the provided cyphertext and returns the plain text of that item. If no such item exists, this rule will return an empty string.

3.2.2 Intruder Aspect

The intruder in the traditional Dolev-Yao intruder model [1] can construct and transmit any message depending on its knowledge. This can cause a problem with the message space filling up with redundent messages that are more likely to be rejected by the recipient. We overcome this problem by restricting the intruder's ability to construct and deliver only messages that meet the receiver's expectations in terms of content and type format. For instance, sending "$\{NA\}_{pk(B)}$" as a third message in the NS protocol, will be refused by the responder, since it expects to receive an encrypted nonce with the same value as the one sent previously. To do so, we firstly construct an enumerative domain named "Type" that contains all of the potential types of message components:

$$Type = \{ANY , ENCR , SHRK, SECK, PUBK,NC,ID\}$$

where ANY is any component, ENCR is an encrypted component, SHRK is a shared key, SECK is a private key, PUBK is a public key, NC is a nonce, and ID is an identity.

The intruder's analysis is then captured by representing each message in a protocol as a syntax tree. By analysis, we mean either breaking down the message into its constituent components or generating a new message from these components. The next two definitions support the tree representation, which is shown in Figure 3.2.

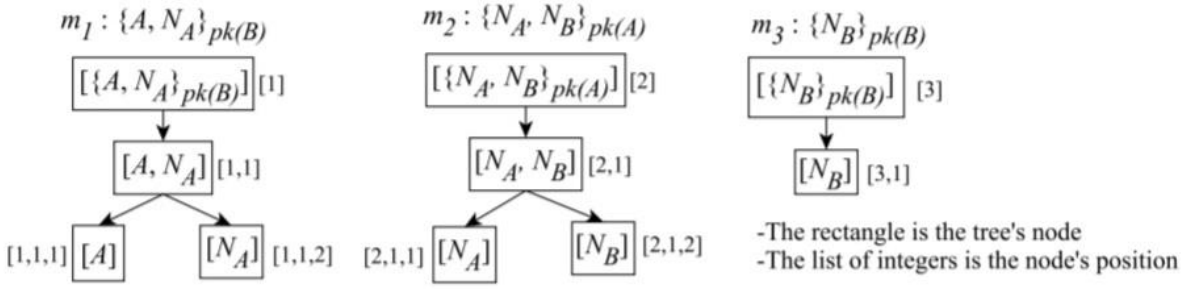

Figure 3.2 Message tree representation examples for the NS protocol

Definition 3.1: *let m be a message defined as a sequence of components, then its syntactic tree has the following properties:*

(1) The root is labeled with the whole message, whereas the other nodes are labeled with the sub components of this message;

(2) a node in this tree that is labeled with a sequence of more than one component has a number of children;

(3) a node that is labeled with a sequence of one encrypted component, then it will have one child, which is labeled with a sequence of its unencrypted components;

(4) the leaves are single sequences consisting of unencrypted components;

(5) a number equal to the message number is assigned to the root;

(6) the children for each node are allocated a left-to-right ordering number, starting at 1.

Definition 3.2: *Let T_i be a tree for a message with the number i, where $i = 1,\ldots,n$. Then the position of the root node for T_i is a single sequence consisting only i. Whereas the position of other nodes is determined by concatenating the parent node's position with the child order number for this node.*

The final step depends on the messages' tree representation to update the intruder's knowledge with the eavesdropped messages, and construct the acceptable messages. This step is illustrated in the following subsections. Note that, the AsmetaL specifications for this step are available in [44].

- **Updating the Knowledge of the Intruder**:

In order to store all the spied messages and their components, which are sequences of String, in the intruder's knowledge, we introduce the function:

''*k*: *Seq*(*Seq*(*String*))''

In fact, the "r_UpdateTheKnowledge" rule usually begins by storing the whole stolen message in the "*k*" function. After that it saves every message component in "*k*", in the meantime it checks if the component's type is encrypted, in order to decrypt it (if feasible), and to add the decrypted parts to "*k*". The static function "*type*" specifies the type of each component as a constant for our model: *"Type: Prod(Protocol, Seq(Integer))*". This function returns the type of the node that has a given location in a given protocol's name. For instance, we have type(NS, [3])=[ENCR] and type(NS, [3, 1])=[NC] for the third message in the NS protocol

that contains two nodes in its tree representation. While k is updated, the location of each node will be saved in the following function: position: Prod(Seq(String), Integer, Integer)) −> Seq(Integer). By just providing the node's value, the session number, and the message number, this function allows quick access to any message's chosen node.

- **Generating Messages.**

Here, we limit the intruder's ability to generate messages by taking into account the type format and the expected content of a message. The type format refers to the component types that are assumed by the receiver. For example, the proper type format for the second message in the NS protocol is an encrypted message using a public key. While the expected content refer to the values of the components that the recipient would anticipate, for example considering the previous message, the value of its first component, which is a nonce, within encryption must be equivalent to the nonce provided in the first message for the NS protocol. As a result, the following static functions for message generation are defined: (The first two functions are associated with each node in the tree representation of the message, while the rest are associated with the encryption key):

(1) *type: Prod(Protocol, Seq(Integer))->Type* : It has been stated earlier;

(2) *posExpVal: Prod(Protocol, Seq(Integer))->Seq(Integer)* : It returns the anticipated value's position (in case it exists) for a given position for a node. e.g., the nonce node of the third message related to the NS protocol, which is at [3, 1], has the same expected value of the node at [2, 1, 2], i.e. *posExpVal (NS*, [3, 1])= [2, 1, 2] . As a result, the "*k"* sequence will be examined to see if it contains a node with a position returned by the " *posExpVal "* function before generating this message.

(3) *posKeyId : Prod(Protocol, Seq(Integer)->Seq(Integer)* : Returns a unique identifier for the key of the provided encrypted component/ node's location;
(4) *keyType : Prod(Protocol, Seq(Integer)->Seq(Type)* : It specifies the key type for a specific encrypted component/node's location.
(5) *posKeyVal : Prod(Protocol, Seq(Integer)>Seq(Integer)* : It returns the predicted key position for the encrypted component's supplied location.

We define the "*r_GenerateMsg*" rule for producing the potential acceptable messages based on all of the preceding functions.

3.2.3 The Attack Scenarios Aspect

The attack scenarios idea is introduced in [4]. This idea defines different schemes related to the origin and destination attacks described in [5], such as Man-in-the-Middle (MITM), and Simple Replay (Simple REPL). For each of the above schemes, a generic algorithm is developed that, given protocol parameters such as the number of its sessions and steps, generates an attack scenario for that scheme. In addition, the algorithms require to specify the assignment of the party's identity and the intruder impersonation to each role.

To return the concrete simulated scenario, the generated scenario is first expressed in Common Syntax [45], then translated into an Estelle specification [46]. In a scenario, the honest participants and the intruder generate messages, but the intruder's messages are generated based on component types.

In our work, we explicitly encode the attack scenarios into AsmetaL, and we limit the intruder's message creation activity to the predicted message content and types. The following static functions that help with assignment identification are used in our AsmetaL specification of a scenario for a certain attack type: $p: Prod(Integer, Integer, Task) -> Id,$ with $Task = \{SENDER, RECEIVER\}$,

and $imp: Prod(Id, Integer) -> Boolean.$ The first function retrieves the provided task's identity: SENDER, RECEIVER for the party's role at the given session and message numbers, while the second function determines if the given identity for a party participated in a particular session number is impersonated. Below we show the specification for the attack scenarios.

- **Man in the Middle Attack Scenario:**

  The Man In The Middle (MITM) attack tries to intercept, replay, and change communications between two participants while making them feel they are only talking with each other. The attack scenario is arranged as follows: an odd message number is placed in a rising sequence of session numbers, while an even message number is placed in a decreasing sequence of session numbers. For example, if there are two sessions and three messages, the steps will be performed in the following order:(1.1) (2.1) (2.2) (1.2) (1.3) (2.3). One of the three potential assignments shown in Table 3.1 is used to create this scenario.

Table 3.1 The possible assignments for MITM attack scenario

No	**Odd sessions**		**even sessions**	
	Initiator	**Responder**	**Initiator**	**Responder**
1	A	I(B)	I(A)	B
2	A	I	I(A)	B
3	A	I(B)	I	B

```
rule r_ExchangeMsgs($p in Protocol)=
let ($s=p(sNo, msgNo, SENDER)) in
 let ($r=p(sNo, msgNo, RECEIVER))in
   par
     if state=SEND then
       par
        if $s=II or
           imp($s, sNo, msgNo)=true then
             r_Intruder[NS]
        else
         par
           r_Initiator[$p, $s, $r]
           r_Responder[$p, $s]
         endpar
        endif
        state:=RECEIVE
       endpar
     endif
     if state=RECEIVE then
      par
        if $r=II or
           imp($r, sNo, msgNo)=true then
           par
             r_UpdateTheKnowledge[$p]
             sNo:=sNo+dir
           endpar
        else
         par
           r_Initiator[$p, $r, $s]
           r_Responder[$p, $r]
         endpar
        endif
        state:=SEND
      endpar
     endif
   endpar
  endlet
endlet
rule r_MITM_Scenario($p in Protocol)=
seq
  stop:=totalSno
  begin:=1
  dir:=1
  while msgNo<=totalMsgNo and existAttack
      =true do
   seq
     while ((stop=totalSno and sNo<=stop)
           or (stop=1 and sNo>=stop)) and
     (existAttack=true) do
       r_ExchangeMsgs[$p]
     dir:=(-1)*dir
     par
       stop:=begin
       begin:=stop
     endpar
     msgNo:=msgNo+1
     sNo:=begin
   endseq
endseq
```

Code 3.4 The r_ExchangeMsgs and the r MITM scenario rules

Code 3.4 represents the AsmetaL model of the MITM scenario. The “r_MITM Scenario” rule, which models the scenario arrangement, and the “r_ExchangeMsgs” rule, which is invoked by the “r_MITM” Scenario rule to send and receive a message by the participants, are both included in Code 3.4. The “*totalSno*” function represents a number of sessions, the “*totalMsgNo*” function represents a number of messages, and the “*stop*”, “*begin*”, and “*dir*” functions aid in accomplishing the growing and decreasing order of the session numbers in the MITM Scenario rule.

There are twosituations are executed in parallel in the "r_ExchangeMsgs" rule. Firstly, when "state=SEND", the SENDER's identity is inspected; if it is "II" or it is impersonated, then a message will be sent by the intruder. Otherwise, both the "r_Initiator" and the "r_Responder" rules are concurrently activated to send a message, relying on the sender's identity, by either the initiator or the responder.

Secondly, when "state=RECEIVE", the identity of the RECEIVER is verified, either to update the intruder's information or to receive the current message from the initiator/responder. The intruder can transmit a message by creating all possible messages and sending them one by one until one of them is is approved. If not, then no such attack scenario exists.

- **Simple Replay Attack Scenario**

The following is a description of the Simple Replay Attack (Simple_REPL) scenario. First, only honest individuals participate in the initial session in order to store the message, which may then be replayed in subsequent sessions. Second, the subsequent sessions are performed as they are without any change until the following parameter: an input parameter named "repl". It identifies the beginning step, at which the intruder replays a message. Before this step, messages are exchanged exclusively between honest participants, then the sent message is intercepted at the “repl” stage, and a message from the first session is resent to the recipient by impersonating its original sender. This indicates that the impersonation takes place just at this step. Table 3.2 shows the assignments for the Simple_REPL scenario.

Table 3.2 The possible assignments for Simple REPL attack scenario

No	First Session			Next sessions		
	Initiator	**Responder**	**Server**	**Initiator**	**Responder**	**Server**
1	A	B	S	A	B	S
2	A	B	S	I	B	S

In Code 3.5, this scenario is depicted. The “r_SimpleReplay” rule defines the scenario arrangment in Code 3.5, and the “r_InterceptSend” rule is invoked by

the “r_SimpleReplay” rule to intercept a message from the intruder and resend a message from a prior session to the recipient.

```
rule r_InterceptSend($p in Protocol)=
 let ($s=p(sNo, msgNo, SENDER)) in
   let ($r=p(sNo, msgNo, RECEIVER))in
     seq
       par
          r_Initiator[$p, $s,$r]
          r_Responder[$p, $s]
       endpar
       choose $id1 in Id, $id2 in Id
       with ($id1!=II) and ($id1!=$id2)
       and (outBox($id1, $id2)!=undef) do
             outBox($id1, $id2):=undef
       r_Intruder[$p]
       par
          r_Initiator[$p, $r, $s]
          r_Responder[$p, $r]
       endpar
       state:=SEND
     endseq
   endlet
 endlet
```

```
rule r_SimpleReplay($p in Protocol)=
 let ($s=p(sNo, msgNo, SENDER)) in
   seq
     msgNo:=1
     sNo:=1
     state:=SEND
     reply:=4
     existAttack:=true
     while msgNo<=totalMsgNo do
       r_ExchangeMsgs[$p]
     sNo:=sNo+1
     msgNo:=1
     wantToInit($s):=true
     while sNo<=totalSno do
       seq
         while msgNo<reply do
           r_ExchangeMsgs[$p]
         while msgNo<=totalMsgNo do
           seq
             r_InterceptSend[$p]
             msgNo:=msgNo+1
           endseq
         sNo:=sNo+1
         msgNo:=1
       endseq
   endseq
 endlet
```

Code 3.5: The r_InterceptSend and the r_SimpleReply rules

3.2.4 The Invariant Properties Aspect

To determine whether a protocol is vulnerable to an attack in response to a certain attack scenario, we utilize invariant testing for the security properties at simulated states of the protocol run. We are looking to check two security conditions: *secrecy* and *authentication*.

The secrecy condition refers to ensuring that no confidential information is acquired by outsiders. In the NS protocol, for example, we have the following condition:

$$\boldsymbol{invariant}\ inv\_sec\ \boldsymbol{over}\ ncInt, k: \boldsymbol{not}(\boldsymbol{contains}(k, ncInt(AA, BB)))$$

It informally indicates that the nonce of AA that is sent to BB is not contained in the intruder knowledge. The authors in [47] define the authentication property as follows: “when an authenticating principal finishes its part of the protocol, The authenticated principal must have been present and participated in its part of the protocol”. This can be expressed as an invariant condition over the start and end functions, as in “the NS protocol":

$$\boldsymbol{invariant}\ inv\_auth\ \boldsymbol{over}\ start, finish: \boldsymbol{implies}(finish(NS, 2, 1), start(NS, 2, 1))$$

It indicates, informally, that if a participant successfully completes receiving the first message in the second session, then sending this message has already begun in this session. Remember that only the honest participants may change the start and end functions, not the intruder.

CHAPTER FOUR

RESULTS AND DISCUSSION

4.1 Introduction

This chapter discusses our findings related to simulating the runs of protocols, such that these runs are specified by the scenario of the anticipated attack type while monitoring the invariant properties.

4.2 Results and Discussion

Our performed experiments focus on simulating protocol runs that are described by the attack scenario while keeping track of invariant properties. If no stated invariant property is violated, then there will be no successful attack linked to these properties and the type of modeled attacks when the associated scenario is simulated.

We gained three scenarios by simulating the NS protocol with the MITM scenario using two sessions. Only one of the obtained scenarios has messages that comply with the attack depicted in Figure 2.2 (b), and its two invariant properties described in Section 3.2.4 have been broken. The intruder delivers messages to honest participants three times during this simulation. The number of produced messages is reduced by the addition of the expected content check, compared to the type matching approach [1]. In our method, for example, the intruder creates six messages in the first step of the second session: $\{A, N_A\}_{pk(B)}$, $\{B, N_A\}_{pk(B)}$, $\{I,N_A\}_{pk(B)}$, $\{A,N_I\}_{pk(B)}$, $\{B, N_I\}_{pk(B)}$, $\{I, N_I\}_{pk(B)}$. While, according to the type matching approach, 24 messages are produced. Because the intruder is only allowed to use the expected responder's public key rather than all of the known keys, such as *pk(A), pk(B), pk(I),* and *prk(I),* fewer messages are produced.

It is worth noting that the attack scenarios provide guidance to seek for attacks by examining all runs with the proper message formats and kinds, and this can minimize the number of evaluated runs. Because the number of performed sessions and the number of steps in each session have a positive effect on the number of protocol runs, lowering the intruder's produced messages will reduce the number of protocol runs. When the attack scenarios are combined with the goal of minimizing the intruder's messages, protocol runs can be drastically reduced.

Concerning the AS-RPC protocol, we acquired the attack in Figure 2.3 (c) by executing the Simpl_REPL scenario with two sessions.

Although we cannot claim that our method is complete, we can say that it discovers attacks with specific defined scenarios, such as MITM and Simple REPL on protocols vulnerable to these attacks, though it does not discover attacks with unspecified scenarios, for example type flaw attacks.

CHAPTER FIVE

CONCLUSION AND FUTURE DIRECTION

5.1 Conclusion

In a simulation-based verification procedure, this project pursues the approach described in [1] to analyze security protocols by employing predefined attack scenarios. In two respects, our method varies from [1]. The scenarios are first defined in AsmetaL and then simulated with the AsmetaS tool. Second, we apply two requirements to limit the intruder's produced messages to those that fulfill the following: the expected message content and type matching.

5.2 Future Directions

To extend our research, we are intending to perform the following:

- Other types of attacks, such as a type-flaw attack, should be specified in future study.
- An automated tool for generating the ASM specification of a protocol under investigation directly from its abstract specification in Common Syntax. The modeling and analysis procedures will be considerably simplified as a result.

References

[1] Dolev, D., Yao, A.: On the security of public key protocols. IEEE Trans. Inf. Theory 29(2), 198–208 (1983)

[2] Blanchet, B.: An efficient cryptographic protocol verifier based on Prolog rules. In: 14th IEEE Computer Security Foundations Workshop, pp. 82–96. IEEE (2001)

[3] Meier, S., Schmidt, B., Cremers, C., Basin, D.: The TAMARIN prover for the symbolic analysis of security protocols. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 696–701. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8 48

[4] Jakubowska, G., Dembi´nski, P., Penczek, W., Szreter, M.: Simulation of security protocols based on scenarios of attacks. Fund. Inform. 93(1–3), 185–203 (2009)

[5] Syverson, P.: A taxonomy of replay attacks. In: Computer Security Foundations Workshop VII (CSFW 1994), pp. 187–191. IEEE (1994)

[6] B¨orger, E., St¨ark, R.: Abstract State Machines: A Method Forhigh-level System Design and Analysis. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-642-18216-7

[7] Gargantini, A., Riccobene, E., Scandurra, P.: Model-driven language engineering: The ASMETA case study. In: The Third International Conference on Software Engineering Advances. ICSEA, pp. 373–378. IEEE (2008)

[8] Gargantini, A., Riccobene, E., Scandurra, P.: A metamodel-based language and a simulation engine for abstract state machines. J. UCS 14(12), 1949–1983 (2008)

[9] Mahadev Satyanarayanan. Integrating security in a large distributed system. ACM Transactions on Computer Systems (TOCS), 7(3):247-280, 1989.

[10] Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. Commun. ACM 21(12), 993–999 (1978)

[11] Ben Henda, N.: Generic and efficient attacker models in SPIN. In: SPIN Symposium on Model Checking of Software, pp. 77–86. ACM (2014)

[12] Lomuscio, A., Pecheur, C., Raimondi, F.: Automatic verification of knowledge and

time with NuSMV. In: Proceedings of the Twentieth International Joint Conference on Artificial Intelligence, pp. 1384–1389. IJCAI/AAAI Press (2007)

[13] Bella, G., Riccobene, E.: Formal analysis of the Kerberos authentication system. J. Univers. Comput. Sci. 3(12), 1337–1381 (1997)

[14] Bella, G., Riccobene, E.: A realistic environment for crypto-protocol analyses by ASMs. In: Workshop on Abstract State Machines, pp. 127–138 (1998)

[15] Jakubowska, G., Penczek, W., Srebrny, M.: Verifying security protocols with timestamps via translation to timed automata. In: Proceedings of the International Workshop on Concurrency, Specification and Programming (CS&P 2005), pp. 100–115 (2005)

[16] Jonathan Katz, Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. Handbook of applied cryptography. CRC press, 1996.

[17] Michael Willett. Cryptography old and new. Computers & Security, 1(2):177-186, 1982.

[18] Whitfield Diffie and Martin Hellman. New directions in cryptography. IEEE transactions on Information Theory, 22(6):644-654, 1976.

[19] Clark John and Jacob Jeremy. A survey of authentication protocol literature. Technical Report 1.0, University of York, 1997.

[20] Florent Jacquemard. Security protocols open repository, 2003.

[21] Roger M Needham and Michael D Schroeder. Using encryption for authentication in large networks of computers. Communications of the ACM, 21(12):993-999, 1978.

[22] Gavin Lowe. An attack on the needham-schroeder public-key authentication protocol. Information processing letters, 56(3):131-132, 1995.

[23] John Clark and Jeremy Jacob. Attacking authentication protocols. High Integrity Systems, 1:465-474, 1996.

[24] Michael Burrows, Martin Abadi, and Roger Michael Needham. A logic of authentication. Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences, 426(1871):233-271, 1989.

[25] Yuri Gurevich. Evolving algebras 1993: Lipari guide. Oxford University Press, 324:9-36, 1995.

[26] Yuri Gurevich. May 1997 draft of the ASM guide. 1997.

[27] **Egon B¨orger and Robert Stark . Abstract State Machines: A method for high-level system design and analysis. Springer, Heidelberg, 2003.**

[28] **Egon B”orger and Alexander Raschke. Modeling Companion for Software Practition-ers. Springer, Heidelberg, 2018.**

[29] **Bradley Fordham, Serge Abiteboul, and Yelena Yesha. Evolving databases: An application to electronic commerce. In International Database Engineering and Ap-plications Symposium (IDEAS), pages 191-200. IEEE, 1997.**

[30] **Egon B”orger and Dean Rosenzweig. A mathematical definition of full Prolog. Science of Computer Programming, 24(3):249-286, 1995.**

[31] **Varsha A whad and Charles Wallace. A unified formal specification and analysis of the new Java memory models. In International Workshop on Abstract State Machines, pages 166-185. Springer, 2003.**

[32] **Egon B’orger and Wolfram Schulte. A programmer friendly modular definition of the semantics of Java. In Formal Syntax and Semantics of Java, pages 353-404. Springer,1999.**

[33] **Steven C Cater and James K Huggins. An ASM dynamic semantics for standard ML. In International Workshop on Abstract State Machines, pages 203-222. Springer,2000.**

[34] **Kevin Compton, Yuri Gurevich, James Huggins, and Wuwei Shen. An automatic verification tool for UML. Univ. of Michigan, EECS Dept. Tech. Report CSE-TR-423, 2000.**

[35] **Wolf Zimmermann and Thilo Gaul. On the construction of correct compiler backends: An ASM-approach. Journal of Universal Computer Science, 3(5):504-567,1997.**

[36] **Giampaolo Bella and Elvinia Riccobene. Formal analysis of the Kerberos authentication system. Journal of Universal Computer Science, 3(12):1337-1381, 1997.**

[37] **Giampaolo Bella and Elvinia Riccobene. A realistic environment for crypto-protocol analyses by ASMs. In Workshop on Abstract State Machines, pages 127-138, 1998.**

[38] **Rui Xue and Deng-Guo Feng. New semantic model for authentication protocols in ASMs. Journal of Computer Science and Technology, 19(4):555-563, 2004.**

[39] **John Thomas and Nancy Leveson. Performing hazard analysis on complex, softwareand human-intensive systems. In 29th International System Safety Conference. International System Safety Society Unionville, VA, 2011.**

[40] **Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. Model-driven language**

engineering: The ASMETA case study. In International Conference on Soft-ware Engineering Advances, pages 373-378. IEEE, 2008.

[41]Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A model-driven process for engineering a toolset for a formal method. Software: Practice and Experience, 41(2):155-166, 2011.

[42] The ASMETA framework. http://asmeta.sourceforge.net/. Accessed: 2019-02-21.

[43] Angelo Michele Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A Metamodel-based language and a simulation engine for Abstract State Machines. Journal of Universal Computer Science, 14(12):1949-1983, 2008.

[44] Farah, A.S., Lisitsa, A., Clare, D.: The AsmetaL specifications for five security protocols and five attack scenarios. https://doi.org/10.5281/zenodo.2628743

[45] Clark, J.A., Jacob, J.L.: A survey of authentication protocol literature. Technical report 1.0 (1997)

[46] Budkowski, S., Dembinski, P.: An introduction to Estelle: a specification language for distributed systems. Comput. Netw. ISDN Syst. 14(1), 3–23 (1987)

[47] Woo, T.Y., Lam, S.S.: A semantic model for authentication protocols. In: Proceedings of the 1993 IEEE Computer Society Symposium on Research in Security and Privacy, 1993, pp. 178–194. IEEE (1993)

الخلاصة

في هذا المشروع ، تم تقديم طريقة تستخدم المحاكاة القائمة على السيناريو لغرض تحليل بروتوكولات الأمان. يحدد سيناريو الهجوم المحتمل انسياب الاتصالات ولكن ليس محتوى رسائلهم. تساعد السيناريوهات في تقليل عدد عمليات تشغيل البروتوكول التي يجب فحصها أثناء محاكاة الهجوم. بالإضافة الى ذلك, من الممكن تقليل عدد مرات التشغيل عن طريق تقليل عدد الرسائل التي تم إنشاؤها بواسطة المهاجمين. قدرة المهاجم على تكوين الرسائل تحدد بالعوامل التالية: مطابقة النوع ومحتوى الرسالة المتوقع. تستخدم طريقتنا لإنجاز المحاكاة. AsmetaS لبناء النماذج و AsmetaL أداتين تدعمان طريقة آلات الحالة المجردة: لغة

لإثبات فعالية نهجنا ، تم فحص العديد من البروتوكولات.

وزارة التعليم العالي والبحث العلمي
جامعة بابل : كلية العلوم للبنات
قسم علوم الحاسوب

عنوان المشروع

تحليل بروتوكولات الامان باستخدام السيناريو المستندة الى المحاكاة

مشروع مقدم الى مجلس كلية العلوم للبنات في جامعة بابل
كجزء من متطلبات الحصول على درجة الدبلوم العالي في العلوم\ علوم الحاسبات

من قبل
أيناس حسن عبادي رسول المظفر
باشراف
د. فرح محمد الشريفي

1443ه 2021 م