Université de Pau et des Pays de l’Adour

Doctoral Thesis

Security Verification of Protocol Implementation

Author: Yulong Fu

Supervisor: Prof. Ousmane KONÉ

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Computer Science

Devant le jury suivant:

Richard CASTANET                 Professeur Emérite             INSTITUT POLYTECHNIQUE DE BORDEAUX
Jean COUVREUR                    Professeur des Universités     UNIVERSITE D’ORLEANS
Ousmane KONE                     Professeur des Universités     UNIVERSITE DE PAU ET DES PAYS DE L’ADOUR
Vladik KREINOVICH                Professeur                     UNIVERSITE DU TEXAS
Jan-Georg SMAUS                  Professeur des Universités     UNIVERSITE PAUL SABATIER TOULOUSE3
Regarding the development of computer technologies, computer systems (Protocol implementations) have been deeply used in our daily life. Those systems have become the foundation of our modern information society. Some of them even take responsibilities for many essential and sensitive tasks (e.g., Medical Treatment System, E-Commerce, Airplane System, Spaceship System, etc.). Once those systems are executed with problems, the loss on the economy may reach an unacceptable number. In order to avoid these disappointing situations, the security of the current systems needs to be verified before their installations. But, with the complexity of the system growing, added to the dynamic features of the system architecture, the manually ad-hoc test cases are hard to design. A suitable automatic security verification method is always needed.

In this thesis, we present the results of our research on the problem of automatic security verification. In our work, we believe the network systems have and are implemented from protocol specifications and the system components are the implementations of protocol. Security verification to a system can be translated to the problem of the verification of the corresponding concurrent protocol implementations. According to this, we study the classic protocol testing methods and extend it to verify the security of protocol implementation automatically. We first focus on the verification to Threatening Request attacks and DoS attacks. We find that those attacks are came from the outside of the system and aim to exhaust the system resource or abilities to block the system. If the system is robust, this kind of attack can be avoided. Base on this point, we believe the Threatening Request and DoS attacks can be guaranteed by verifying the robustness test cases. We use an extended transition model: Glued-IOLTS, to describe and combine multiple components together and present them within one reachable graph. Then by adding the refusal graphs and a simple security policy to make the model become robust.
Nonetheless, a possible attack may also happen from the inside components of the system, which may relate to man-in-the-middle attacks. In this case, an intruder is considered hidden inside of the system, and has the ability to eavesdrop the transmitting messages and attack the system. To verify this situation, we propose another enhanced model of Protocol Testing: SG-IOLTS, which defines the security properties into system transitions and combines protocol components together with the insecure medium of intruder machines. With this proposed model, the malicious actions of intruders are considered to connect the legal components. With this model and the proposed algorithms, the generated test cases will contain the actions of the intruder and should be taken as the probable authentication attack scenarios. Then by verifying these possible attacks on the protocol implantations, the security of authentication can be verified.

But in the process of test generation, because of the complexity of multiple protocols, added to the robustness actions or intruder actions, the combination of scenarios to be computed may increase at an explosive speed. To address this, we define the security experience of engineers as Security Objectives, and use them to control the test generations. In this case, the states and transitions considered during each test generation are controllable. The test cases are generated on-the-fly and have its specific security purpose.

In this thesis, we also studied the problem of intrusion detection system and propose a transition based model to enforce the detecting ability of the anomaly IDSes. Because the detected traffics from IDS monitors can identify the protocol type used, the messages are possible to be concatenated as protocol message sequences. Meanwhile, because the used protocol type is detected, the corresponding protocol standards can be used to propose a non-anomaly profile. In our method, we utilize the proposed Glued-IOLTS to model and present a non-anomaly profile of the system. Then, these message sequences are rewritten as the format of the Glued-IOLTS, and are compared with the transitions of non-anomaly profile to find the intrusion. We propose an algorithm to compare the sequence and result the attack types if it exists.

Finally, all methods we proposed are implemented with the tools and experimented through the case study of RADIUS protocol.
# Contents

## Abstract

## Contents

1 Background and Introduction

1.1 Overview of Information Security

1.2 Background of Protocol Security Verification

1.2.1 Protocol Security Verification Methods

1.2.1.1 Protocol Security Analysis

1.2.1.2 Protocol Testing

1.2.1.3 Conformance Testing

1.2.1.4 Robustness Testing

1.2.1.5 Security Testing

1.2.2 Intrusion Detection System

1.3 Significance of the Study

1.4 Overview of the dissertation

2 Protocol Security Testing with Secure Medium

2.1 Problems and Assumptions

2.1.1 Threatening Requests Attacks

2.1.2 Concurrent Components System

2.1.3 Test Architecture

2.2 A Robustness Model: Extended Glued-IOLTS

2.2.1 Refusal Graph

2.2.2 Protocol Security Policy

2.2.3 The Extended Glued-IOLTS method

2.2.4 Algorithm to calculate test cases

2.3 Case study of RADIUS authentication protocol

2.3.1 Introduction of RADIUS protocol

2.3.2 Construction of Glued_IOLTS

2.3.3 Calculate the $S^+_glu$ and the possible traces

2.3.4 Assess the robustness of the networked implementations

2.4 Conclusion

3 Security Testing with Insecure Medium

3.1 Problems and Assumptions

3.2 Secure Transition Models
3.3 Intruder Model and SG-IOLTS ........................................ 51
   3.3.1 Modeling of the Medium ................................. 52
3.4 Verification of Security Protocol Implementation ............... 56
   3.4.1 Test Architecture ........................................ 57
   3.4.2 Transition Traces Generation ............................ 57
   3.4.3 Test Verdict ................................................ 62
3.5 Case Study Results ............................................... 63
   3.5.1 An Example of NSL protocol ......................... 63
   3.5.2 Computed Transition Sequences to Verification Scenarios .. 64
   3.5.3 Example of RADIUS Protocol ...................... 66
3.6 Conclusions ....................................................... 70

4 Security Objectives ................................................. 71
   4.1 Problems and Assumptions ................................. 71
   4.2 Description of Security Objective ....................... 72
   4.3 Synchronous Product ........................................ 75
   4.4 Security Objectives over Test Selection .................. 76
      4.4.1 Security Objectives over Test Generation with Secure Medium . 77
      4.4.2 Security Objectives over Test Generation with Insecure Medium . 80
4.5 Case study of RADIUS authentication protocol ................... 82
   4.5.1 Examples of SO over Test Generation with Secure Medium ... 82
   4.5.2 Examples of SO over Test Generation with Insecure Medium ... 84
   4.5.3 Assess the security of the networked implementations ........ 86
   4.5.4 Review of this technique ............................... 86
4.6 Conclusion ........................................................ 87

5 Analysis on Intrusion Detection System ............................. 88
   5.1 Problems and Assumption .................................... 88
   5.2 An Automata-based A-NIDS Solution ......................... 90
      5.2.1 The target system and IDS architecture ............ 90
      5.2.2 Intrusion Detection Approach ........................ 91
      5.2.3 Traffic Detection ....................................... 92
      5.2.4 Message classification and Profile selection .......... 92
         5.2.4.1 Message classification .......................... 92
         5.2.4.2 Profile selection .................................. 93
      5.2.5 Verification ........................................... 95
   5.3 An Experiment over RADIUS system ........................ 96
   5.4 Conclusion .................................................... 99

6 Conclusions and Future Works ....................................... 102

A Test Generation and Analysing Tools ................................ 106
   A.1 Requirements .................................................. 106
   A.2 Application Design .......................................... 108
   A.3 Guides of how to use ........................................ 110
      A.3.1 Java Environment ...................................... 110
Chapter 1

Background and Introduction

1.1 Overview of Information Security

Since the first computer has been developed in 1945, the computer systems have served and helped us human beings for more than half a century. Those systems deeply affect our life and make it become more colorful, convenient and efficient. Nowadays, the computer aided systems are almost used everywhere and became the foundation of our modern society. But, along with those brightness, some problems such as information leakages, unauthenticated visits or even cybercrimes [44] are also happened and become more and more serious.

According to the report of McAfee, an average of 60000 new pieces of malware have been detected each day from the network in the year of 2010 [58], and surges 26% by April 2011 [69]. One significant feature of network attacks is that the attackers usually don’t need to leave their home, but can pull off crimes with billions of dollars through the network. Attacks through network are safe and easy to the hackers. In the example of Albert Gonzalez [80], he and his team broke into the databases of the well-known retail giants including TJ Maxx, Barnes Noble and BJ’s Wholesale Club, and gain access to more than 180 million payment card accounts between 2005 and 2007. He and his crew were estimated to cost the companies they compromised more than $400 million in reimbursements, forensics and legal fees [58]. All those statistic data and examples tell us how dangerous of the modern Internet, and remind us of the importance of network security.
In order to protect the network and achieve the transition information security, three key concepts are proposed as the core principle of information security: Confidentiality, Integrity and Availability (CIA).

- **Confidentiality** means the capacity of preventing the unauthorized individuals or systems, ensuring that information is accessible only for those who are authorized. For example, before we login our operation system, we will be asked to input our personal user-name and password to protect our private data from unauthorized access and destructions; In CDMA system, the signal ratios, which are sent by the base station, will be encrypted by using PIN code to make the unauthorized users away from the core meaning of the signal. Encryption and its corresponding Security Protocol are the main methods to solve the confidentiality.

- **Integrity** means the data cannot be modified without undetectable. Any changes to the transit data should be detected and informed to the user. The problem of integrity is violated when a message is actively modified in transit. For instance, people always detect the integrity by verifying some checking codes, which are generated from the protected data through some algorithm, and impossible to be decompiled. For example, many encryption methods such as CRC, Hash code, MD5 etc. are used popularly to verify the data integrity.

- **Availability** restricts the stability of the system, which means the information data (or services) must be available when it is needed. This feature signs that the system resource: the hardware (the computing systems), the software (the security polices) and the mediums (communication channels) should be functional correct. For example, the UDP flooding attack may send unacceptable numbers of packets to the network, aiming to block the communication channels to make the services or data unavailability to users. The corresponding attack method is like a DoS attack, and by improving the system’s robustness, the availability of the system can be improved to a certain degree.

In 2002, Donn Parker proposed an alternative model according to the classic CIA. During his work, instead of three, Donn recalls six atomic elements of information: confidentiality, possession, integrity, authenticity, availability, and utility. Although this model is more precise, the classic CIA is still popular and well accepted. To achieve confidentiality, the principal approach is cryptography. The usages of cryptography can be dated back to about 600 BC: the so called “Atbash cipher of Hebrew”, which encodes each
Chapter 1. Background and Introduction

Figure 1.1: CIA [81]

character with different orders in alphabet table. The first encryption device appeared around 400 BC, when the Spartans allegedly used a “Scytale” for encryption. The “Scytale” is a rod with random diameter. The sender needs to wrap a long and thin paper (or belt) around the rod, and writes the message on it. When the paper delivered to the receiver, the messages on the paper are the wrong sequence order of characters, and need to wrap it on another rod with the same diameter to decrypt it. The diameter of the rod is a kind of security key of the device. After the appearance of the computer system, use and challenge of confidentiality both come to a new age [23]. With the ability of computer, the encryption and decryption become easy and are utilized as a common method to protect information systems.

The earliest attempt to crack the encrypted code started as early as 1932 by Polish researchers. Based on their researches, by the end of the Second World War, a team of researchers in Bletchley Park was able to break coded messages on a daily basis. It involved the development of dedicated code breaking machines that tried to recover the key from the encrypted text. In 19th century, Auguste Kerckhoff states that an encryption scheme should be secure even if everything except the key is known to the enemy. This is often referred to as Kerckhoff’s law. Later in the 20th century Claude Shannon formulated a similar notion, starting that “the enemy knows the system”, which is referred to as Shannon’s Maxim. From 1949 onwards, when Claude Shannon published his seminal paper on information theory, numerous scientific publications appeared on the
topic of cryptography, and for a large part of them focus on discovering new encryption schemes.

In 1976 Diffie and Hellman published their key paper, in which they introduced a mechanism that later known as asymmetrical encryption (public key). This scheme circumvents the problem with traditional symmetrical cryptography, where both the sender and receiver must have the same key. This breakthrough has resulted in a number of variations on these schemes, leading to international standards for encryption mechanisms. Both symmetric and asymmetric encryption schemes are used extensively today for the encryption of internet traffic, wireless communications [78], smart-card applications, cell phone communications, etc. But only cryptography method is not enough to guarantee the security in communications, since cryptography used in a wrong way could still disclose some vital information.

For example, in the Needham-Shroeder Public Key (NSPK) protocol [55]:

\[
\begin{align*}
\text{Msg 1. } & i \rightarrow r \ (\text{Ask}): \{n_i, i\}_{pk_r} \\
\text{Msg 2. } & r \rightarrow i \ (\text{Rpl}): \{n_i, n_r\}_{pk_i} \\
\text{Msg 3. } & i \rightarrow r \ (\text{Cfm}): \{n_r\}_{pk_r}
\end{align*}
\]

This protocol uses public key encryption to obtain a mutual authentication, and we assumed the encryption can never be broken, which means that only the private key can decrypt the encryption. The nonce \(n_i\) and \(n_r\) are supposed to be kept secret during the protocol execution. Although there is no problem of cryptography, the NSPK protocol still has flaws. Just as Gavin Lowe pointed in 1995, an intruder could attack the protocol by manipulating the messages of the protocol. We show an example of this attack scheme below.

\[
\begin{align*}
\text{Msg 1. } & \text{Alice} \rightarrow \text{Job}: \{n_{\text{Alice}}, \text{Alice}\}_{pk_{\text{job}}} \\
\text{Msg 1'} \ (\text{Alice}) & \text{Job} \rightarrow \text{Bob}: \{n_{\text{Alice}}, \text{Alice}\}_{pk_{\text{Bob}}} \\
\text{Msg 2'} \ & \text{Bob} \rightarrow (\text{Alice})\text{Job}: \{n_{\text{Alice}}, n_{\text{Bob}}\}_{pk_{\text{Alice}}} \\
\text{Msg 2. } & \text{Job} \rightarrow \text{Alice}: \{n_{\text{Alice}}, n_{\text{Bob}}\}_{pk_{\text{Alice}}} \\
\text{Msg 3. } & \text{Alice} \rightarrow \text{Job}: \{n_{\text{Bob}}\}_{pk_{\text{job}}} \\
\text{Msg 3'} \ (\text{Alice}) & \text{Job} \rightarrow \text{Bob}: \{n_{\text{Bob}}\}_{pk_{\text{Bob}}}
\end{align*}
\]

Attacker Job first receives Msg 1 from Alice, then he masquerade as Alice to cheat with Bob, which makes Bob believe that he is communicating with Alice. Because the
intruder places himself between the two honest agents, this attack becomes known as a man-in-the-middle (MITM) attack.

There is a useful metaphor about cryptography: the Encryption is like a bicycle lock. Although it is the most important parts to protect your bicycle, if you don’t care how to use it, your bicycle can be stolen, maybe like the situation shown in Figure 1.2 (Photo taken around the library of University of Bordeaux 1).

Security protocols mean to ensure some forms of secure communication, and they usually try to establish this by using some form of encryption. It defines how to organize communications and how to use the encryptions. Security protocols are under most of our current communication systems, such as secure Internet communications, cell phone networks, as well as the communication between credit cards, ATM machines, and banks. For these applications, it is crucial that no malicious party can disturb the intended working of the protocol, or eavesdrop on something it is not supposed to hear.

In 1983, Dolev and Yao published their paper to reason the security protocol with three assumptions: First, cryptography is assumed to be perfect: a message can only be decrypted by somebody who has the right key; Second, messages are considered to be abstract terms; Third, the network is assumed to be under full control of the adversary [26]. Then people found that given a security protocol, it is possible to develop mathematical techniques to derive security properties of a protocol under Dolev-Yao assumptions. As a result, the work of Dolev and Yao has been possible for a branch of research which can be roughly summarized as the so called black-box security protocol analysis in [23]. The analysis models based on these abstractions are black box, in the
sense that they consider encryptions as abstract functions with some particular properties. Instead of modeling all cryptographic details and properties, this kind of method assumes that somebody has already invented a perfect cryptographic scheme, which we can use in building security protocols. However, building the three properties sketched above into a precise mathematical model, with the apparent assumption, and clearly defined security properties, has proven to be a hazardous task. In 1989 Burrows, Abadi and Needham published their ground breaking paper [64] on logic for authentication (BAN logic), which also depends on the same black-box assumption as the Dolev-Yao model. Using this logic, they were able to prove that several protocols satisfy a form of authentication. Later, a large number of security protocol formalisms and tools are developed. Although those techniques are complex and not compatible, the security issues of the considered protocols can be verified. But just as the authors mentioned in [40] [30], some security vulnerabilities may also come from the processes of protocol implementing. The security of the protocol implementations should be verified, which drives us to the methods of “Protocol Testing”. “Protocol Testing” is a general way to verify the protocol implementations, which has been proposed and widely accepted in the industries since Gonenc proposed the Distinguishing Sequences in 1970. After that, because the uncertain states of composite systems (protocol implementations) can be inferred through the specific input sequences and the corresponding execution results, the protocol implementations are possible to be verified through testing. Other test methods such as T, D, U, W methods are proposed later to calculate the sequence of transitions, and some industry Formal Description Techniques such as SDL (Specification and Description Language), Estelle and LOTOS (Language Of Temporal Ordering Specification) are published also. Nonetheless, those testing methods usually concern the functional conformance of protocol implementation, the security issues of protocol implementations are rarely discussed.

On next section, we will focus on the crucial sector of information security: the Protocol Security Verification. We will introduce the general models, popular methods and approaches, and the problems concerned in this thesis.

1.2 Background of Protocol Security Verification

A successful attack on system or application is usually based on some system vulnerabilities. For a protocol based system or application, those vulnerabilities are usually...
happened within the designing of the protocol or during the processes of implementing. Vulnerabilities may be also caused by some limitations of materials or unexpected misuses. In order to detect those vulnerabilities and to prevent those possible attacks, the methods of Protocol Security Verification, which include Security protocol analysis, Protocol security testing, etc., are proposed in the literature. Meanwhile, intelligent Intrusion Detection Systems (IDSs) is also proposed and used to reduce the probabilities of malicious executing on the system. In the following parts of this section, we are going to discuss some concepts of them, which are related to our works of this thesis.

1.2.1 Protocol Security Verification Methods

1.2.1.1 Protocol Security Analysis

Security protocol is a kind of communication protocol, which defines communication rules, sequences, even cryptographic methods to keep the exchanging information secure. Security protocols are the underlying of most current communication systems, such as secure Internet communication between credit cards, ATM machines and banks. For these applications, it is important to grantee the security of the communication, such as no malicious party can disturb the intended working or eavesdrop on something he was not supposed to hear, etc. A security protocol analysis method can be used to prove the security of the using protocol.

Methods of Security Protocol Analysis are widely used to analyze and prove the design of a security protocol. In Security Protocol Analysis, the protocols are usually described with mathematical (logic) models. But these models usually become very complex even for some simple encryption schemes. It is only feasible to reason about the security of detailed protocols by abstracting away from some (cryptographic) details. Consequently, Black-box Security Protocol Analysis is proposed by abstracting the details of cryptographic as black-box and dealing them with assumptions, which hugely reduce the complexities of the security analysis.

The black-box security protocol analysis method was introduced in 1983 by Dolev and Yao [26] with two main properties. First, the cryptography is assumed to be perfect: a message can only be decrypted by someone who has the proper key (there is no way to crack the scheme). And second, the messages are considered to be abstract terms: either the intruder learns all messages inside of the encryption (because he has the key), or he learns the encrypted message. Instead of modeling all cryptographic details and
properties, people are assumed to have already invented a perfect cryptography scheme, which can be utilized in building security protocols.

After the work of Dolev and Yao, many researchers have put enthusiastic involvements on this topic, and many successful frameworks have been developed.

**BAN Logic**  The BAN logic is named after its inventors: Mike Burrows, Martin Abadi and Roger Needham. It is a logic of belief and derivation [21]. The aim of using BAN logic is to analyze authentication protocols by deriving the beliefs that honest principals correctly executing a protocol can come to a result as the one of the protocol execution [15]. The approach is to “idealize” the messages in the protocol specification into a logical formula asserts. Then by BAN rules to derive the correctness of those assertions. But in BAN logic, it doesn’t contain the logical inversions, therefore, it can’t be used to prove a protocol flawed [60]. In the notation of BAN, it usually use believe (|≡), received (<), said (|∼), control (⇒), flash (#), etc. to denote the relations between the principles and atoms. For example \( P |≡ A |∼ X \) means that the principle \( P \) believes the principle \( A \) has sent \( X \) before. \( X \) is either a message or a formula expression of BAN logic. The BAN logic uses these expressions to describe the protocol specifications and the state of the protocol execution.

In an analysis, the security protocol needs to be written as BAN assertions and assumptions first, then is derived by some general BAN rules (Message meaning rule, Nonce Verification, etc. For details please read [64]). The purpose of the derivation analysis is to prove the communicated principles believe the secret keys and nonces, and to make sure those keys and nonces are fresh.

**Spi-calculus**  The spi-calculus is an extension of the pi-calculus, which is designed to represent and analyze cryptographic protocols [6]. The Spi-calculus is a process based calculus, which models the concurrent computation with terms, process, and communicate channel. The Spi-calculus extends pi-calculus with cryptographic operations. It is designed for the description and analysis of security protocols. The Spi-calculus shares many advantages with the pi-calculus, and has a well defined semantic. It has been successfully applied for the analysis of security protocols, e.g. the use of it in [29]. It has been also successfully used to analyze the web service security [14].

In Spi-calculus, it defines the attacker executing in parallel with the protocol executing process. The adversary can be any process and has an initial knowledge \( S \), which is a
finite set of terms of protocol. If $M$ is a secret message, and it has been sent through the public channel, the adversary can obtain the secret, and the protocol is not secure. The Spi-calculus can write the protocols into coding, then utilize its reduction rules and equivalence rules to rewrite the code to analyze the security protocols.

**PCL** The PCL (Protocol Composition Logic) is a formal logic for stating and proving security properties of network protocols that use public and symmetric key cryptography [24]. It is designed around a process calculus, but with actions such as $Send(X, t)$, $Receive(X, t)$, $Verify(X, t)$ etc., to identify different steps of the protocol execution. For example, $Send(X, t)$ holds in a run if thread $X$ sent the term $t$ as a message. The PCL is designed to support compositional reasoning, including parallel composition of different protocols, and sequential composition of protocol steps. With PCL, the security of some complex security protocols (such as PKMv2), which are composed by multiple security protocols, can be proved.

PCL uses a set of formal syntax and semantics to describe and derive the security protocols. PCL also define a “Cord Spaces”, which represent a multiset of cords. A cord space represents a group of processes which communicate and distribute computation. PCL utilizes many cords reactions to model synchronous communication. Since actual communication networks are asynchronous, it defines a buffer cord: $[receive x; send x]$ to store the sent messages until someone is ready to receive them. It defines various rules, such as $GenericRules$, $PossessionAxioms$, $SequencingRules$, $HonestyRules$ etc. to prove and deduce the security.

**Scyther** Scyther provides language and model checking tools for security protocol analysis. Alike as PCL, Scyther also concerns the problems of multiple protocols concurrent system (composition components system). But it doesn’t express the protocols on cord space. In Scyther, the executions of the protocols are based on runs. Each role of protocols has a role specification, which can identify the status of the role and defines run and instantiate for the protocol executions. The communication between different runs is considered as asynchronous, and has an output buffer (denoted as $BS$) from the sending run and an input buffer (denoted as $BR$) from the receiving run.

As we have presented, those security analysis methods can model the security protocol with formal definitions, syntax, semantics, formulas and rules. They can prove the security and correctness of the protocol by logic deduction. Those methods are concentrated on the formal expressions and the preciseness of the results. Nonetheless, these methods
are usually difficult to be understood and to use, knowledge of using those methods and clear deduction logics are prerequisites. Meanwhile, those analytical methods are usually restricted to the corresponding protocol, which makes they are hard to be adopted to the automatic security verification.

1.2.1.2 Protocol Testing

Test is a general logic of verification, which uses predefined test sequences to result the system under test, and deduces the unobservable information to verdict the system. Test is widely used in the industry (both software and hardware), and helped many companies to reduce billions of losses every year. The idea of using the test on verifying protocol implementation is Protocol Testing [70]. Protocol testing is mainly based on the black-box approach. Therefore the nature of the protocol specification has a strong influence on protocol testing. In fact, the methods for the development of test cases are largely dependent on the specification formalism. Most communication protocols have a reactive nature; therefore specification languages for reactive systems are favored which precisely define the temporal ordering of interactions. It is therefore understandable that the finite state machine model is often used for defining protocol specification. For this reason, most work on protocol testing has been based on FSM models.

Figure 1.3a shows a communication system from the point of view of two users. The users interact with the communication service through interactions, called service primitives, exchanged at so-called service access points (SAP). The definition of the behavior of the box which extends between the two users is called the service specification. It defines the local rules of interactions at a given service access point, as well as the so-called end-to-end properties which relate the interactions at the different access points and represent the communication properties, such as end-to-end connection establishment or reliable data transfer.

Figure 1.3b shows a more detailed view of the service box. Two protocol entities (PE) communicate through an underlying, more simple communication service. The definition of the required behavior of a protocol entity is called the protocol specification, and involves the interactions at the upper and lower service access points. In addition, the protocol specification usually identifies different types of so-called protocol data units (PDU, or messages) which are coded and exchanged between protocol entities through the underlying medium.
Natural languages are not suitable to describe the protocol specifications. Although the use of a specification written in native language gives the illusion of being easily understood, but leads to lengthy and informal specifications which often contain ambiguities and are difficult to check for completeness and correctness. Many different formal description techniques have been proposed for the protocol engineering cycle, including finite state machines (FSM), Petri nets, formal grammars, high-level programming languages, process algebras, abstract data types, and temporal logic. The simpler models, such as FSM, Petri nets and formal grammars, were often extended by the addition of data parameters and attributes in order to naturally deal with certain properties of the protocols, such as sequence numbering and addressing. In this work, we utilize FSM methods to describe the protocol.

**Finite State Machine** The method of Finite State Machine was first used to model and design the sequential logic circuits. Then was introduced to analyze computer system and communication protocol recently. It is conceived as an abstract machine, which can be modeled by a finite number of states. The machine is in only one state at a time, and the state at any given time is called the *current state*. The machine can change from one state to another when initiated by a triggering event or condition.

There are two types of finite state machines: Mealy machines and Moore machines. The theories are very similar for the two types, but Mealy machine model finite systems more properly and are more widespread than Moore machines. When we talk of a finite state machine in the thesis, we mean the finite mealy machine.
• **Mealy Machine.** Mealy Machine is a determined finite state machine, whose output values are determined both by its current state and the current inputs.

• **Moore Machine.** Moore Machine is also a finite state machine, but the output values are determined solely by its current state.

**Definition 1.1 (Finite State Machine).** A finite state machine (FSM) $M$ is a 6-tuples Mealy Machine: $M = (s_0, S, I, O, \delta, \lambda)$ where:

- $s_0$ is the initial state;
- $S$ is a finite and non empty set of states;
- $I$ is a finite and non empty set of input actions;
- $O$ is a finite and non empty set of output actions;
- $\delta : S \times I \rightarrow S$ is the state transition function;
- $\lambda : S \times I \rightarrow O$ is the output function.

In some kinds of problems, the initial state $s_0$ is not given, and the Finite Mealy Machine changes to be a 5-tuples $(S, I, O, \delta, \lambda)$. A FSM can be represented by a state transition diagram, which is a directed graph $G = (V, E)$ [53, 54]. Where, the vertices of $V$ reflect to the states of $S$, and the edges of $E$ contain the elements of input actions $I$ and output actions $O$. The outgoing edges from a state $s \in S$ lead to $\delta(s, a)$ for all $a \in I$, and they are labeled as “a/b”, where “a” is the input symbol and “b” is the output symbol of $\lambda(s,a)$.

**Example 1.1:**

An example of FSM is shown in Figure 1.4. Where we know: $S = \{s_0, s_1, s_2, s_3\}$, input alphabet $I = \{a, b\}$, and output alphabet $O = \{0, 1\}$. For instance, applying starting at $s_0$ produces output $\lambda(s_0, a) = 0$ and moves to the state $\delta(s_0, a) = s_1$.

A walk in $G$ is a finite non-null sequence of consecutive edges: $W = \{(v_1, v_2; L_1) (v_2, v_3; L_2)\ldots\}$. $G = (V, E)$ is reachable, if any pair of state $(s, t)$ in $V$, there exists a sequence of adjacent vertices which starts with $s$ and end with $t$ [4].
Extended Finite State Machine  The Extended FSM (EFSM) enriches FSM with variables as part of the state, and makes transitions with guards and actions. It is widely acknowledged that EFSM is a very powerful model for test derivation [73]. It is used in a number of industrially significant specification techniques, such as SDL [2], Estelle [3], Statecharts [39], UML [5] etc. An EFSM can be viewed as a compressed notation of FSM. Generally speaking, it is possible to unfold it into a pure FSM by expanding the values of the parameters and variables, assuming that all the domains are finite [65]. These variables are from additional enabling conditions in the transitions of the underlying FSM. As a result, different transitions may occur in response to the same combination of input event and starting (major) state in an EFSM. A transition in an EFSM may be triggered by three types of enabling conditions: the input event, the current (major) state and a boolean expression involving minor state variables. Each transition of EFSM consists of three operations: the output operation, the state transition (changing major state) and operations that alter values of the minor state variables.

Definition 1.2 (Extended Finite State Machine). An Extended Finite State Machine (EFSM) is a 6-tuples: $M = (s_0, S, I, O, T, V)$ where:

- $s_0$ is the initial state;
- $S$ is a finite and nonempty set of states, $S = \{s_j \mid 0 \leq j \leq n\}$;
Chapter 1. Background and Introduction

- $I$ is a finite and nonempty set of input actions;
- $O$ is a finite and nonempty set of output actions;
- $T$ is a finite and nonempty set of transitions;
- and $V$ is a finite and nonempty set of variables.

The transition element $t \in T$ is described as a 5-tuple:
$$t = (\text{source}, \text{dest}, \text{input}, \text{pred}, \text{compute block}).$$
Here, $\text{source}$ and $\text{dest}$ are the states in $S$ represent the starting state and the tail state of transition $t$. $\text{input}$ is either an input action from $I$ or empty. $\text{pred}$ is a predicate expressed in terms of the variables in $V$, the parameters of the input and some constants: $\text{pred}=\{p(x, i) \mid x \in V \cup \text{Consts}, i \in I\}$. The $\text{compute block}$ is a computation block which consists of the assignment statements of the current state and the output statements.

A component of a transition can also be represented by prefixing the transition with a followed name of the component. For example $t.p\text{red}$ represents the predicate components of the transition $t$. A context of $M$ is the set of \{(\text{var}, \text{val}) \mid \text{var} \in V \text{ and } \text{val} \text{ is a value of } \text{var} \text{ from its domain}\}. A valid context of a state in $M$ is a context which is established when $M$’s execution proceeds along a walk from the initial state to the given state.

**Example 1.2:**

A simple coffee machine can serve with two kinds of coffee if the clients insert enough money. This simple process can be modeled with EFSM as Fig.1.5. After the client inserts coins, the machine changes to state $s_1$, and uses $\text{Coin_count}$ to record the amount of coins inserted. The client then can choose the preferred coffee. If the coins inserted are enough, the machine changes to the corresponding state and serve the corresponding coffee; otherwise, it returns to the state $s_0$ and wait for the next insertion from the client. In this example, by using EFSM, we have $s_0=\{s_0\}$, $S=\{s_0, s_1, s_2, s_3\}$, $I=\{\text{?Insert}(\text{Coins}), \text{?Choose}_1, \text{?Choose}_2\}$, $O=\{!\text{Serve}(\text{coffee}_1), !\text{Serve}(\text{coffee}_2)\}$, $V=\{\text{Coin_count}\}$, and the transitions of $T$ are listed in the Table 1.1 below:

**Labeled Transition System** The labeled transition system is another popular specification formalism to model the protocol system. It models the behaviors as transitions, and serves as a semantic model for various formal specification languages [76] [49].
Chapter 1. Background and Introduction

Figure 1.5: Example of Extended Finite State Machine

Table 1.1: Transitions of EFSM Example

<table>
<thead>
<tr>
<th>Transition actions</th>
<th>source</th>
<th>dest</th>
<th>input</th>
<th>pred</th>
<th>compute_lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>?Insert(Coins)</td>
<td>s0</td>
<td>s1</td>
<td>?Insert(Coins)</td>
<td>Coin_count=Coins</td>
<td></td>
</tr>
<tr>
<td>?Chose_coffee1;</td>
<td>s1</td>
<td>s0</td>
<td>?Chose_coffee1</td>
<td>Coin_count&lt;10</td>
<td></td>
</tr>
<tr>
<td>Coin_count&lt;10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?Chose_coffee1;</td>
<td>s1</td>
<td>s2</td>
<td>?Chose_coffee1</td>
<td>Coin_count &gt;=10</td>
<td></td>
</tr>
<tr>
<td>Coin_count&gt;=10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?Chose_coffee2;</td>
<td>s1</td>
<td>s0</td>
<td>?Chose_coffee2</td>
<td>Coin_count&lt;20</td>
<td></td>
</tr>
<tr>
<td>Coin_count&lt;20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!Serve(coffee1)</td>
<td>s2</td>
<td>s0</td>
<td>!Serve(coffee1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>!Serve(coffee2)</td>
<td>s3</td>
<td>s0</td>
<td>!Serve(coffee2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Definition 1.3 (Labeled Transition System).

A labeled transition system is a 4-tuple array \( \langle S, L, T, s_0 \rangle \) where

- \( S \) is a countable, non-empty set of states;
- \( L \) is a countable set of labels;
- \( T \) is the transition relation, where \( T \subseteq S \times (L \cup \{\tau\}) \times S \)
- \( s_0 \) is the initial state.

The labels in \( L \) represent the observable actions which occur inside of the system. The state \( s_i \) will change to another state \( s_j \) after receiving the actions in \( L \). The definition of \( T \) reveals the relations between states in \( S \), and the sign of \( \tau \) denotes the internal and unobservable actions. For example, the equation \( (s_0, a, s_1) \) represents an observable
transition ∈ T and \((s_0, \tau, s_1)\) represents an unobservable transition of the system. A trace is a finite sequence of observable actions. The set of all traces over \(L\) is denoted by \(L^*\), and \(\varepsilon\) denotes the empty sequence. If \(\sigma_1, \sigma_2 \in L^*\), then \(\sigma_1 \ast \sigma_2\) is the concatenation of \(\sigma_1\) and \(\sigma_2\). \(|\sigma|\) denotes the length of trace of \(\sigma\).

**Definition 1.4**

Let \(P = \langle S, L, T, s_0 \rangle\) be a labeled transition system, \(s\) and \(s' \in S\), and let \(\mu_i \in L \cup \{\tau\}, a_i \in L, \) and \(\sigma \in L^*\), then we have the following notations:

\[
\begin{align*}
    s \xrightarrow{\mu} s' & \quad =_{df} \quad (s, \mu, s') \in T \\
    s \xrightarrow{\mu_1 \ldots \mu_n} s' & \quad =_{df} \quad \exists s_0, \ldots, s_n : s = s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} s_n = s' \\
    s \xrightarrow{\epsilon} s' & \quad =_{df} \quad s = s' \text{ or } s \xrightarrow{\tau \ldots \tau} s' \\
    s \xrightarrow{a} s' & \quad =_{df} \quad \exists s_1, s_2 : s \xrightarrow{a} s_1 \xrightarrow{a} s_2 \xrightarrow{a} \ldots \xrightarrow{a} s_n = s' \\
    s \xrightarrow{\sigma} s' & \quad =_{df} \quad \exists s' : s \xrightarrow{\sigma} s' \\
    \text{trace}(p) & \quad =_{df} \quad \{\sigma \in L^* \mid p \xrightarrow{\sigma}\} \\
    \text{init}(p) & \quad =_{df} \quad \{a \in L \mid p \xrightarrow{a}\}
\end{align*}
\]

**Definition 1.5:**

An input-output transition system \(P\) is a labeled transition system in which the set of actions \(L\) is partitioned into input actions \(L_I\) and output action \(L_U (L_I \cap L_U = \emptyset, L_I \cup L_U = L)\), and for which all input actions are always enabled in any state.

If \(q \in S\), then \(\text{Out}(q)\) denotes all the output labels from \(q\), \(\text{In}(q)\) denotes all the input labels to \(q\), and \(\text{Out}(S, \sigma)\) denotes the output of \(S\) after \(\sigma\). If \(\exists a \in L_I\), then we write \(?a\) to mark this label is an input label; If \(\exists b \in L_U\), then we write \(!b\) to express this label \(b\) is an output label. \(\text{ref}(q)\) represents the input actions which are not accepted by state \(q\).
1.2.1.3 Conformance Testing

Conformance testing aim at verifying the implementation under test (IUT) is conformed with the protocol specification. In this kind of problem, the IUTs are usually taken as black-box and can be only observed through their input/output behaviors [27].

In this kind of problem, only the strongly connected specification A should be considered (reachable graph). Once the implementations are given, they will never change during the test execution. Starting point for conformance testing is a specification in some formal notation, and an implementation is a device or program interacting with its environment, which is considered as a black box. Test cases are derived from the specification, and applied to the implementation, such that from the results of applying them it can be concluded whether the implementation conforms to the specification.

Paper [50] summarized three kinds of conformance testing methods:

- **Conformance Testing with structure equivalence.** In this kind of test method, the protocol specifications are modeled with FSM or its extensions, and are used to guide the test generation to check whether the IUTs are structural equivalent with the specifications. The states are “equivalence” if it can run from these states with the same input-output sequences.

- **Black-box Conformance Testing.** In this kind of test, the protocol specifications and IUTs are also modeled with FSM or its extensions. The system transitions are categorized into “internal actions” (actions happened between system states) and “external actions” (actions happened between system states and environments), and only the “external actions” are concerned in this test method. The test cases are still guided by the modeled protocol specifications, and those generated test cases should be executed through the IUTs. Then by comparing executing consequences and the outputs of the test cases, the test cases can be judged (Fault or Pass).

- **Test Oriented Conformance Testing.** In this kind of test, the test cases are not generated exhaustively any more. The test cases are generated on the fly by the proposed test objectives. Each time of test generation, only the test cases match with the test objective is selected and generated.
1.2.1.4 Robustness Testing

Robustness is the degree to which a system or component can execute correctly in the presence of invalid inputs or stressful environmental conditions [42]. Robustness testing concerns the appearance of a behavior which possibly jeopardizes the rest of the system, especially under a wrong input. A system with robustness means that the system can be executed without crashing, even when it is used inappropriately [48].

The robustness testing has real and important means to some critical system, such as aircraft system, airspace system, medical control system etc. While considering some kinds of system insecure just come from the inconsiderable requests (threatening requests), the robustness testing is considered to enforce the system security.

Some researchers use Fuzz sets to construct the robustness test cases [62], while others use random variants to the conformance test sequences [37] or just generate them from a robustness model [48]. We develop a robustness test method for a concurrent and networked system, and present it in Chapter 2.

1.2.1.5 Security Testing

“Security Testing” implies testing methods and approaches to verify the security. It is mostly related to software testing in the literature, such as the work of [59] and [10], but not only limit on it. “Software testing consists of the dynamic verification of the behavior of a program, on a finite set of test cases, suitably selected from the usually infinite executions domains, against the specified expected behavior” [13]. Software testing usually covers all the stages of the various existing development processes, and the testing techniques should be adapted to the way the system is developed. For instance, the V-model implies that a specific software testing technique has to be applied at each step (see Figure 1.6). The concerns of security testing should be also performed during the developing life cycle (see Figure 1.7, security testing is presented as the risk related analysis or test and penetration testing here).

As mentioned in [66], the security testing necessarily involves the following two diverse approaches:

- testing security mechanisms to ensure that their functionality is properly implemented, and
performing risk-based security testing is motivated by understanding and simulating the attacker’s approach.

Most of the white box security testing methods belong to the first approach, where the testers are accessible to the code, and can design test cases directly according to the security mechanisms. The most used method of the second approach is Penetration Testing [10]. Penetration Testing is currently the most used in the industry, involves security experts who try to perpetrate attacks by playing the role of a hacker and trying
to attack the system and exploit its vulnerabilities. It needs to create a threat model first, which is a detailed written description of the key risks to the application [75]. This approach is applicable to uncover the weakness, flaws and vulnerabilities in the code, but most of the current penetration testing activities are ad-hoc and rely on expert knowledge of target systems or existing exploits.

Extra security test approaches, such as fault-injection based, access control based and some specific security testing methods also perform actively in their specific domain. The fault-injection based techniques apply fault injection to the application environment. The application environment is perturbed by modifying environment variables, files or processes utilized by the application under test. Then the application has to resist to this perturbation and must not have an insecure behavior that may lead to a security flaw. Testing access control is based on the security policies [20], which are analyzed through mechanisms such as RBAC [57], OrBAC [19], XACML, etc.. The exact security testing such as Bypass Testing, specifically focuses on the process of testing web applications by bypassing client-side input invalidation and triggering the server-side input validation.

1.2.2 Intrusion Detection System

An intrusion detection system (IDS) is a device system or software applications, which can detect malicious activities or policy violations by monitoring the network traffics or system actives. Once the usual actives are detected, the IDS produces reports to a management station. An IDS is the high-tech equipment of a fault alarm, which is configured to monitor information gateways, hostile activities, and knew intruders. IDS knows how to parse and interpret network traffic and/or host activities [11]. IDS focus on detecting and preventing the intrusive activities, which were not detected by conventional security mechanisms. These activities were usually distinguished as the activities from legitimate users, which may result of outsider’s attacks that have passed through a firewall, the use of stolen passwords, and any other activities that were not prevented by authentication, authorization, or other security subsystem [71].

The IDS can be divided into four major independent components: Event Generator, Event Analyzer, Event Database, and the Response Unit. The event generator is the component that sample’s activity from the network environment and converts the information into objects that can be used by other components. After converting the information into objects, the generator stores the objects in the event database. The
event analyzer retrieves the objects from the event database and analyses them in order to detect intrusions [61].

In [11], base on the way to detect the network traffic, the authors classify IDSes from their functionalities and are loosely grouped the following three categories:

- **Network-based intrusion detection system (NIDS).** NIDS monitors the entire network from the perspective of the location where it is deployed. It can eavesdrop all the communications through the network. Figure 3.2 depicts a network using NIDS. The units have been placed on strategic network segments and can monitor network traffic for all devices on the segment.

- **Host-based intrusion detection system (HIDS).** HIDS reflects only the system it resides. Figure 3.4 depicts a network using HIDS on specific servers and host computers. The two HIDSes locate on the “Mail Server” and “Web Server” separately, which are going to load each corresponding rule set.

- **Distributed intrusion detection system (DIDS).** The standard DIDS functions in a Manager/Probe architecture. NIDS detection sensors are remotely located and report to a centralized management station. Figure 3.5 shows a DIDS composed of four sensors and a centralized management station. Sensor NIDS 1 and NIDS 2 are protecting the public servers. Sensor NIDS 3 and NIDS 4 are protecting the host systems in the trusted computing base.
There are various approaches to intrusion detections, such as Threshold Detection, Anomaly Detection, Rule-Based Penetration Identification, Model-Based Intrusion Detection and Intrusion Prevention, which are mentioned in [43]. But generally, two categories of IDS are usually distinguished: misuse detection and the anomaly detection. The former uses the traces or templates of known attacks, while the latter builds profiles of non-anomalous behaviors of computer system’s active subjects. For example, IDIOT [52] and STAT [43], use patterns of well-known attacks or weak spots in the system to match...
and identify known intrusions. The main advantage of misuse detection is that it can be accurately and efficiently detect instances of known attacks. The principal disadvantage is that it lacks the ability to detect the truly innovative (i.e., newly invented) attacks. On the other hand, Anomaly detection (sub) systems, for example, the anomaly detector in IDES \[56\], flag observed activities that deviate significantly from the established normal usage profiles as anomalies, i.e., possible intrusions. For example, the normal profile of a user may contain the averaged frequencies of some system commands used in his or her login sessions. If for a session that is being monitored, the frequencies are significantly lower or higher, then an anomaly alarm will be raised. The principal advantage of anomaly detection is that it does not require prior knowledge of intrusion and can thus detect new intrusions. The principal disadvantage is that it may not be able to describe what the attack is and may have a high false positive rate.

\section*{1.3 Significance of the Study}

The work of this thesis focusses on the security verification and attack detection of protocol implementations. In this thesis, we mainly contribute on the following topics:

- Protocol Security Verification with Secure Medium, which uses testing to verify the security problems such as Threatening Request and DoS attacks. In this method, we define the relation between robustness and security, and guarantee the security of the protocol implementations by verifying their robustness. The model (Glued-IOLTS) we used in the approach connects the multiple system components as one transition system, and present it in one reachable graph. Which demonstrates the transition movements of one component and the transition synchronization between different components with a way of clear and controllable (comparing with Parallel Composition \[74\]). Then by adding the refusal graphs and a simple security policy, which can model the inopportune and invalid inputs respectively, the Extended Glued-IOLTS becomes robust and can be used to generate the robustness test cases to verify the security. This approach considers the whole system as one black-box and trusts the connected components. It supposes the attacks only come from the outside and the medium between those trusted components is secure.
Protocol Security Verification with Insecure Medium, which also use test approach to verify the authentication problems of protocol implementations. This test approach faces on more precise authentication attacks, and believes an intruder may hide inside of the system and has the ability to eavesdrop the transmitting messages and attack the system. In this case, the system components and the common medium should not be trusted anymore and some detail features of the authentication should be identified. To adapt this, we propose an enhanced method of IOLTS: SG-IOLTS, and use it to model the multiple components and possible intruders. In this case, the generated test cases may contain some actions of the intruder and should be taken as promising attack scenarios.

Test Generation with Security Objectives, which define the security experience of engineers as security objective, and use it to control the test generations on the fly to avoid the probable state explosion problem.

Transition based IDSes Analysis, which proposes a transition based method to enforce the detecting ability of the anomaly IDSes. This method solves problems of normal anomaly IDSes, and can be considered to improve the anomaly based IDS products.

1.4 Overview of the dissertation

This thesis can be divided into six major topics. Chapter 1 presents the background of the work and discusses some existed methods of the related topics. From Chapter 2, our concerns about security testing for protocol implementations are presented. In Chapter 2, an approach based on a secure medium is proposed. In this approach, the protocol implementations are modeled as concurrent components and are supposed to be trusted. Possible attacks such as DoS attacks are considered to come from the outside of the system. Base on this, a robustness transition model: The Extended Glued-IOLTS is defined and algorithms are proposed to generate test cases, which are going to be used to verify the security of the protocol implementations automatically. In Chapter 3, the hypothesis of secure medium is broken and the possible intruders are considered during the communication. An enhanced model of the transition system: SG-IOLTS, which defines the security properties into the transitions and intruder machines into the medium, is proposed. With this model and the proposed algorithms, the generated test cases will contain the actions from the intruders and are thought to be the possible
attack scenarios. But as the complexity of the system growing, the test cases considered may grow explosively. To avoid this possible limitation, in Chapter 4, we introduce the security experiences to be Security Objectives and make the test generation on-the-fly. In Chapter 5, we discussed the problem of anomaly IDSes and propose FSM based analytical approach to solve this problem. Finally, in Chapter 6, we conclude this thesis.
Chapter 2

Protocol Security Testing with Secure Medium

A central idea in systems engineering is that the complex systems are built by assembling components [16] [51] under protocol. These components need to conform their specifications by exchanging messages which conform the network communication protocols. These components are potentially to be attacked if the communication protocols have any defects and the attackers know the protocols as a prerequisite. The Network System must implement mechanisms to avoid those possible attacks (threatening requests which are aimed at disrupting or crashing the system, and then provoking some denial of service attacks). Therefore, before deploying these components into a system, a phase of preliminary test is necessary to detect the capacity of resisting the threatening inputs [18]. These threats generally occur through a mix of normal and malicious requests which must be checked to evaluate the robustness of the system. But as the systems and the related security issues become complex and difficult to be analyzed, new methods are required for both modeling and analysis. In this chapter, not like [41,77], which use fuzz set to generate errors, we extend the protocol testing methods: IOLTS to our Extended Glued-IOLTS to define the possible malicious inputs. This is a robustness model for the specification of networked system, which should contain all possible outside attacks (In this model, we assume all internal components are trusted, and the communication medium insider of the system is closed, which means that the attacks can only come from outside and the attacker can’t get and modify the internal transitions). In this way, the ancient methods of protocol testing can be used to analyze the robustness of network implementations, and detect the potential security defects cause of it. We
propose an approach to use this method and algorithms and tools to generate the test automatically.

2.1 Problems and Assumptions

2.1.1 Threatening Requests Attacks

One kind of popular network attack is DoS (Denial of Service) attack, which aims to block the system services by exhausting the resources. Threatening Request Attacks is one typical DoS attacks. In this type of attack, the attackers usually have the knowledge and experiences of the running communication protocols, and know well the defects of the system protocols. During a attack, the attackers send some specified messages (Threatening Requests) which conform the structures and syntax of communication protocols but with some dangerous message parts. After the devices receive those request messages, because those messages conform the syntax of network protocol, the devices will execute them as legal. Once the dangerous messages parts are executed, if the device (implementation) is not designed robust, the corresponding system may be blocked or even crash down. Figure 2.1 shows an example of RADIUS protocol. As we can see: the second message contains a dangerous message part “99”. This part of message represents the attribute code of the RADIUS protocol, but this number “99” is not defined in the protocol specification. Because the other parts are still same as the normal message and the whole message conforms the RADIUS protocol format, the difference may not be detected. Once the server received this message, it will treat the message as a legal one. But when the server executes the message until the undefined part, it will not understand and perhaps will block the system services. To avoid this kind of network attacks, the network devices are required to be robust, and a predefined Robustness Testing can help to verify this kind of problems.

Those malicious messages are abstracted as observable labels in state machine model, and may not be defined in the protocol specification. But the robustness of the system considers the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions [22]. The robustness model of the system must contain those possible malicious messages. Then we can use robustness testing which will be introduced later to verify the possible threatening attacks. But before that, we need a model of state machine to contain those possible malicious labels.
2.1.2 Concurrent Components System

The system, exactly the modern network systems usually contains multiple devices. Those devices may distribute in many different places, and are connected by conforming to the same network protocols. Here we take those distributed network devices as components, and we use Concurrent and Networked Components to describe this kind of network system. The concurrent and networked components refer to the networked system which has many local or remote components to work together to finish some network functions. These components are connected through some materials or mediums, and messages or data are exchanged between them. Figure 2.2 presents a simple example of concurrent components which has n components which work together and communicate through a common medium. We assume the whole system (includes the multiple components and the Medium between the components) as a block box, the malicious users only exist outside of the system and they can’t operate on the inside medium of the system.

The problem of multiple components has been considered by many researchers, and already propose many modeling and testing methods. In [51], Koné and Castanet research on the problem of testing generation for interworking system. They generate the conformance test cases in a way of “on-the-fly” [1], which used test propose (TP) to generate the test cases step by step to reduce the testing complexity. In [38] and [9],

![Figure 2.1: An Example of Threatening Request](image-url)
the authors considered the interoperability testing of two concurrent components, and proposed their so-called C-Methods to describe the concurrent system, then derive the paths between two components to generate the interoperability test cases. Those methods are significant useful under their assumptions and specific models. In our work, we take some ideas from those previous works, and conclude them to be a new model, which is presented in next sections.

2.1.3 Test Architecture

The implementations of communication protocol are special softwares \[28\], the methods of protocol implementation verification must have some connections to software testing. In a specification based protocol testing, an executable test case is a sequence of inputs and outputs, which are derived from the protocol specifications. Then these input sequences of the test cases are executed through the implementations (IUT) by the tester (or test engine) to obtain the corresponding output results. These test cases are the traces of the graph of the protocol specification. The conformance testing is to see whether the traces of the protocol implementations also exist in the same way as protocol specifications \[53\]. When running the test cases through an implementation, the input sequences of the test case will cause the corresponding output sequences. And if the outputs are similar as the pre-defined outputs from the specification, then we say this test case running is successful, and we note “Pass” to this test case. Otherwise, the test running fails, and the test case is marked as “Fail” \[25\].

A general test architecture is presented as Figure 2.3. For concurrent components, each of the IUTs (implementation under test) has two kinds of interfaces. The lower interfaces \(LI_i\) are the interfaces used for the interaction of the two IUTs. These interfaces
are only observable but not controllable, which means a lower tester \((LT_i)\) connected to such interfaces can only observe the events but not send stimuli to these interfaces. The upper interfaces \((UI_i)\) are the interfaces through which the IUT communicates with its environment. They are observable and also controllable by the upper tester \((UT_i)\).

### 2.2 A Robustness Model: Extended Glued-IOLTS

In this section, we introduce our model to the networked components system which can be against the problems presented above. We use the formal finite model IOLTS and extend it with robustness concerns. To do this, we need to use Refusal Graph to model the situations when accepted labels(messages) appear on the wrong machine state, and define a simple Protocol Security Policy to model the situation when the unaccepted labels (messages) received by the system.

#### 2.2.1 Refusal Graph

Refusal Graph presents the refused inputs for each state in the state machine. It is a normal concept used by conformance testing [48]. A Refusal Graph is a deterministic and minimal graph. It can identify the refusal set of labels for each state, which may become the reason of possible errors during the execution. Refusal Graph has the similar definition as IOLTS [46,49], which is presented as below:
Definition 2.1 (Refusal Graph): A RGraph is a labeled graph represented by a 5-tuple \( (S, L, T, Ref, s_0) \) where:

- \( S \) is a finite set of states.
- \( L \) is a finite set of events.
- \( T \) is a set of transitions.
- \( Ref \) is an application which defines for each state the set of events that may be refused.
- \( s_0 \) is an element of \( S \) called initial state.

Figure 2.4 gives an example of Refusal Graph. The dotted lines represent the refusal inputs of each state. The corresponding refusal actions set of each states are: \( Ref(q0) = \{?b\} \), \( Ref(q1) = \{?a\} \), \( Ref(q2) = Ref(q3) = \{?a, ?b\} \).

2.2.2 Protocol Security Policy

A security policy is the primary way in which management’s expectations for security are transited into specific, measurable, and testable goals and objectives. It is crucial to
take a top down approach based on a well-stated policy in order to develop an effective security architecture [18, 79]. Protocol Security Policy is a Security Policy for protocol, which defines some security rules and procedures of the protocols [79]. Normally, those policies are defined by the standard organizations and written inside of the documents of the protocol.

Different protocols may take different security strategies against the insecure inputs. Even in the same protocol, for different insecure inputs at different states, the corresponding security procedures may also be different [12]. But whatever the strategy is, if we consider the whole protocol and abstract the security procedures as one state (called as Graph Invalid Block) of the system, for the situation of no quiescence, only two possible situations should be considered:

- Protocol ignores the insecure requests, which means after the security procedures state (GIB), the finite state machine goes back to the state which receives the insecure requests.

- Protocol deals the insecure requests with some security considerations, which means after the security procedures state (GIB), the finite state machine goes to some state different with the state received the insecure requests.

The first situation can be presented in Figure 2.5. We add one state named GIB (Graph Invalid Block) to abstract the procedures of dealing the insecure inputs. The label (?Invalid) represents the insecure messages, and label (\!μ) presents the outputs from the security procedures. We notice that this kind of protocol security concerns add a loop to each state which receiving the insecure messages.

The second situation is presented in Figure 2.6. In this situation, the labels have the same means as Figure 2.5, the state $s_n \neq s_i$, the dotted label between $s_n$ and $s_i$ represents transitions \{ $s_n \Rightarrow s_i$ \}. The state $s_i$ can be before or after of state $s_n$, both states can be the initial states.

With this simple security policy, the unacceptable inputs can be dealt. Notice these policies are modeled by states and transitions; it is possible to add them into the system state machine.
2.2.3 The Extended Glued-IOLTS method

Figure 2.7 presents an overview of our method. The components of the system are modeled by IOLTS first, and then are connected to be a Glued System [16], which is modeled by the Glued-IOLTS. This Glued-IOLTS is extended later by adding some security models to make it contain all possible inputs to the system.
In a system of concurrent and networked components, the components need to communicate and work together. These concurrent components have their own specifications, and normally, they can be tested separately. But we need a method to analyze those components together to generate the robustness test cases.

As we described, the concurrent components communicate with each other through a common medium using their lower interfaces, and receive the messages from the environments through their upper interfaces (see Figure 2.3). We separate the states of each component which are directly connected to the common medium into higher_level and low_level states. The higher_level states connects with the states of the components, and the low_level states are the projection of the system states in the common medium. We use the low_level states to define the common medium.

**Definition 2.1 (Higher/lower level state and secure medium):**

The states of the concurrent and networked components system have two levels:

- higher_level state $s_{i,u}$ connects to the environment or other states of the same component.
- lower_level state $s_{i,l}$ connects to the states of other components

A secure medium is a subset of the lower_level interfaces of the states, which stimulate the messages to other components, and transitions between them. $S_i$, $L_i$ denote the states and labels in $IOLTS_i$, $S_j$ and $L_j$ denote the state and labels in $IOLTS_j$, then if $\exists l \in L_i$, $s_{i} \in S_i$, $l \in Out(s_i)$, and $s_{j} \in S_j$, $l \in L_j$, $l \in In(s_j)$; the transition of the common medium between $IOLTS_i$ and $IOLTS_j$.
and $IOLTS_j$ is presented as $s_i L \xrightarrow{\nu} s_0 L$. We make $S_M$ to denote all the states, and $T_M$ to denote all the transitions in the medium.

In other articles (for example in [38]) the common medium is called “glue code”. And inside the medium, it follows FIFO rules. We assume the medium has an infinite memory and untouchable. We assume this medium is secure and do not consider the faults happened in the common medium here.

With the help of common medium, we can glue the components together [30]. We connect the medium states and the stimulated component initial states with the same label as the medium state received (denoted as $L_M$). Then the different components are glued.

**Definition 2.3 (Glued IOLTS):**

A Glued IOLTS represents a set of IOLTS $\langle S_i, L_i, T_i, s_i L \rangle$ (i=1,n) and a medium M, which is a 4-tuple:

$IOLTS_{glu} = \langle S_{glu}, L_{glu}, T_{glu}, s_{glu} L \rangle$, which

- $S_{glu} = \langle S_1 \cup S_2 \cup ... \cup S_n \cup S_M \rangle$,
- $L_{glu} = \langle L_1 \cup L_2 \cup ... \cup L_n \rangle$,
- $s_{glu} L = \langle s_1 L, s_2 L, ..., s_n L \rangle$ is the initial state,
- $T_{glu} \subseteq S_{glu} \times L_{glu} \times S_{glu}$

$T_{glu} = \{ (s_1, s_2, ..., s_i, ..., s_m) \ xrightarrow{\alpha} 
(s_1, s_2, ..., s_i', ..., s_m) \ | \ (s_i, \alpha, s_i') \in T_i \cup T_M \}$,

$T_M = \{ (s_i L, \mu, s_j L) \ | \ i \neq j, \mu \in Out(s_i L) \cap In(s_j L) \}$

**Example 2-1:**

Figure 2.8 below shows a concurrent system with two components. The component specifications can be expressed as $S1$ and $S2$ represent the machine state of each system respectively. In this example, we know the Medium states are $S_M = \{ S1 L, S15 L, S22 L \}$. The Glued IOLTS are presented in Figure 2.9. The states of the medium (the lower_level interface of $S1$ or $S2$) are presented with a dual ring.
Figure 2.8: Example of two simple components - Specification

Figure 2.9: Example of two simple components - Glued IOLTS

Modeling of Threatening Requests

The networked system communicates with its environment through different messages inputs. We use label to abstract the messages input, and denote it as $m_{in}$. The inputs
to a system contains the following three categories:

- **Acceptable Inputs** (denoted as $L_{acc}$) which are defined by the specifications, and sent or received inside the system, or between the components and the environment.

  \[ \{ m_{in} \in L_{acc} | m_{in} \in L, \exists s_i, s_j \in S, s_i \xrightarrow{m_{in}} s_j, \text{ and } m_{in} \in In(s_j) \} \]

- **Inopportune Inputs** to state $s_i$ (denoted as $L_{inos_i}$), which can be taken as a subset of “Acceptable Inputs”, but is sent in the wrong state of the system. As such, they must be considered as threats to states because they risk to degrade the system operation.

  \[ \{ m_{in} \in L_{inos_i} | m_{in} \in L_{acc}, s_i \in S, m_{in} \notin In(s_i) \} \]

- **Invalid Inputs** (denoted as $L_{inv}$), which are out of the band of “Acceptable Inputs”.

Those invalid inputs are threats to the system.

\[ \{ m_{in} \in L_{inv} | \forall s_i \in S, m_{in} \notin In(s_i) \} \]

The invalid inputs ($L_{inv}$) are a set of labels (actions) to the state, which are not defined in the specification. We make $\Theta_{s_i}$ to represent all the possible input actions to the state $s_i$, $In(s_i)$ represents all the defined input actions by the specification, then the invalid inputs of $s_i$ are noted as: $L_{inv}(s_i) = \Theta_{s_i} \setminus In(s_i)$. For example, in the RADIUS protocol, which is shown in Figure 2.13: at the state $s_3$ of NAS server, in the specification, the NAS can accept the actions “?new_id”, or a internal actions “r”, so $In(s_{3_NAS}) = \{ ?new_id \}$, and according to the specification, the “packet identifier” has 8 bit, which means $0 \sim 255$ are possible, and the actions of (new_id) must be one number between ($0 \sim 255$). Then the $L_{inv}(s_{3_NAS}) = (0 \sim 255) \setminus (new_id)$, there are 254 elements in $L_{inv}(s_{3_NAS})$.

The Acceptable Inputs of the components are already modeled and presented as labels in the Glued-IOLTS. The problem is how to model the Inopportune Inputs and the Invalid Inputs into the Glued-IOLTS. With the help of Refusal Graph and Protocol Security Policy which are introduced before, we can solve this problem. We use the concepts of Refusal Graph to model the Inopportune Inputs and the concepts of Protocol Security Policy to model the Invalid Inputs respectively. We use a simple state of GIB (Graph Invalid inputs Block, see Figure 2.12) to describe those security processes with Protocol Security Policy. Then we join the two models into the Glued-IOLTS to be an extended-Glued-IOLTS: $S^+_{glu}$ to describe all the possible paths.
Example 2.2:
Figure 2.11 is the extended specification of the example 2.1. By adding the elements of invalid and inopportune input actions, the $S_{glu}^+$ includes all possible actions.

2.2.4 Algorithm to calculate test cases

We say that the implementation of concurrent and networked components is robust if it suits the following conditions:

Definition 2.4 (Robustness).
The implementations of a concurrent and networked components system are denoted as $IUTs$, $S_{uni}$ represents the specification of the those implementations, then:

$IUTs$ Robust $S_{uni} \equiv_{def} \forall \sigma \in traces(S_{glu}^+) \Rightarrow Out(IUTs, \sigma) \subseteq Out(S_{glu}^+, \sigma)$
According to the Definition 2.4, we know that in order to check the robustness of this system, we need to check whether the traces in $S^+_{glu}$ can also be found in its implementation.

The robustness test case can be generated through the following approach:

- Analyze the specifications to figure out the concurrent system described using Glued IOLTS $S_{glu}$.
- Calculate the $S^+_{glu}$
- Calculate all the possible paths of the $S^+_{glu}$ to generate the test cases.
- Test Cases run on the implementation. If the implementation can pass all the test cases, the implementation is robust. If not, the implementation fails the robustness test.

We give an algorithm in Listing 2.1 to calculate the possible traces of $S^+_{glu}$. We assume the “initial” states are reachable, and we define the “end” states as the states which the system return to the “initial” state or suspended after those states. The inputs of
this algorithm are the Extended Glued Specification. The pair \( (stimulate, response) \)
denotes the actions between different systems, and the function \( \text{opt}() \) in the algorithm is
to calculate the corresponding actions in this pair. The algorithm uses two recursions to
trace back the specifications from the “end” states to the “initial” states. The algorithm
uses an Arraylist “Trace” to record all the passed labels. When the algorithm reaches
the “initial” state, it uses the function \( \text{Check.glue}() \) to detect the action inputs from the
common medium. If it finds that the passed traces need the inputs from the medium,
then it adds the corresponding medium label, and continues to trace back to another
system. If it cannot find the requirements from the past traces, the algorithm stops this
traceback, and continues to the next trace.

<table>
<thead>
<tr>
<th>Inputs: the states of Extended Glued Specification ( S ), the labels of Extended Glued Specification ( L );</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outputs: possible trace arraylists ( \text{trace}[m] );</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>( \text{int} \ k, m, n = 0; )</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>( \text{ArrayList} \ \text{trace}[m], \ L_{sti}[k]; )</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>( \text{public main}() { )</td>
</tr>
<tr>
<td>- ( \text{ArrayList}&quot;state&quot; s_end; )</td>
</tr>
<tr>
<td>- ( \text{For} (\text{int} \ i = 0; i &lt; S.\text{size}(); i++) { )</td>
</tr>
<tr>
<td>- ( \text{if}(S.\text{get}(i).\text{getStatus}().\text{equals}(&quot;end&quot;)); )</td>
</tr>
<tr>
<td>- ( s_{\text{end}}.\text{add}(S.\text{get}(i)); )</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
The program is written in Java language. The inputs of this algorithm are the state and label set of $S^+_\text{glue}$, and the outputs will be a ArrayList which contains all possible travail traces of $S^+_\text{glue}$. From line 6 to 21 are the part of main() function, within this function, the program first calculate the set of end state and store them in a ArrayList of "s_end", then invoke the function of "Traceback(State)" for each element of this arraylist. From line 22 to 39 is the code of Traceback function. It track backward the graph from the given state to the initial state of the machine. In line 23, an Arraylist $L$ is create to store all input labels to the state $s$. If $s$ is the initial state, the traceback finish. Otherwise, the program add the $i$th labels of $L$ to the end of Arraylist of trace $[m+i]$ to store the trace. Then the program calculate the pre_state of the stored label, and continue the "Traceback()" function. This process will continue until it reach the initial state of the graph. After the function "Traceback(State)", the program go back to its main function, and invoke the function "Check_glue(ArrayList)" to verify whether in the checked trace,
the initial state is glued to another component. If the initial state is glued to another components, the program add the glued label to the end of the trace, and continue use the function "Traceback(State)" to trace backward until the initial state of another component. If not, the checked trace is recorded and return this ArrayList. From line 39 to line 57 are the code for the function of "Check_glue(ArrayList)". Noticed that we store all the labels of stimulate of the pair < stimulate, response > in the an Arraylist "L_stimulate", and take it as a known condition when the state machines are input. The ArrayList L_sti[k] records the actions in one trace which will stimulate another system.

Example 2.3 (Description of algorithm)
Consider the case in Example 2.2. Two system components are connected through messages transitions, and the extended glued IOLTS of the two system component S1 and S2 is considered as the input to the program. The program do not need to store the graph but use "State", "Label" and "Transition" classes to store the information of the graph. After the input of the system, the program can calculate the possible traces automatically. One possible output trace of this algorithm is {?b !a ?y ?x !b ?x ?a !a ?y ?x}. For this trace, the algorithm begins from the end state 4 of system 1, and traces back to the initial state 0 of S1. Then it finds the stimulate actions ?b in the trace, and adds the medium action !b to this trace and continues to trace back in S2. When it reaches the initial state 0 of S2, it finds another stimulate action ?a exist in the trace, then it needs to add the medium action !a to this trace, and continues to trace back in S1 until the initial state of S1. The resulting traces are the tests sekeletons.

2.3 Case study of RADIUS authentication protocol

2.3.1 Introduction of RADIUS protocol

RADIUS (Remote Authentication Dial In User Services) protocol is a network protocol between three basic components: Client, NAS (Network Access Server), and RADIUS server. The three components are connected and work together, to finish the handshaking and AAA (authentication, authorization, and accounting) security processes. RADIUS protocol is often used by ISPs and enterprises to manage the accesses to the internet or internal networks, wireless networks, and integrated e-mail services.

The authentication and authorization characteristics in RADIUS are described in RFC 2865 [68], and accounting is described by RFC 2866 [67]. In the RFC 2865, RADIUS
Table 2.1: RADIUS Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
<th>Explains</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Access-Request</td>
<td>Identify a client want to connect to the server through this message</td>
</tr>
<tr>
<td>2</td>
<td>Access-Accept</td>
<td>Identify the previous request is accepted</td>
</tr>
<tr>
<td>3</td>
<td>Access-Reject</td>
<td>Identify the previous request is rejected</td>
</tr>
<tr>
<td>4</td>
<td>Accounting-Request</td>
<td>Identify the message with account information</td>
</tr>
<tr>
<td>5</td>
<td>Accounting-Response</td>
<td>Identify the response from the server to the previous accounting request</td>
</tr>
<tr>
<td>11</td>
<td>Access-Challenge</td>
<td>Another request need to be sent by following new requirement in this message</td>
</tr>
<tr>
<td>12</td>
<td>Status-Server</td>
<td>Identify Server status</td>
</tr>
<tr>
<td>13</td>
<td>Status-Client</td>
<td>Identify Client status</td>
</tr>
<tr>
<td>255</td>
<td>Reserved</td>
<td>Not defined</td>
</tr>
</tbody>
</table>

defines the message patterns, authentication methods, encryption methods, etc. It is a standard accepted by most of the manufactories.

RADIUS protocol defines and distinguish messages with well defined message formats. As it is defined in [68]: the first octet of a RADIUS is called as “code”, which is used to identifier the usage of the message. In RFC 2865, the standard defines 9 types of RADIUS messages which are presented in Table 2.1. The next octet of message is called “Identifier”, which can identify the message itself. Then it is the part of “Length”, which is used to identify the length of this RADIUS message. It covers two octets, and has a value between 20 octets and 4096 octets. The following 16 octets of “Length” are defined as ”Authenticator”. This value is used to authenticate the reply from the RADIUS server, and is used in encryption. The left octets of the message are “Attributes”, it carries the specific information of the message.

In this chapter, we ignore some detail security properties and abstract the exchanging messages between NAS, RADIUS and Clients with labels. The IOLTS model of each system components are presented in Figure 2.12.

2.3.2 Construction of Glued_IOLTS

Because the client components only stimulate the requests or receive the finial results, it can be taken as one part of the environment. So in the RADIUS protocol, the communications between two components: NAS and RADIUS server should be well considered. These two components should be glued together to verify the security.
Figure 2.12: IOLTS of RADIUS Protocol

Figure 2.13 is the Glued-IOLTS after we glue the NAS and the RADIUS components together. The interactions \( \tau \) of the RADIUS server represent the processes of security checking. All the significations of labels can be found in Table 2.

Figure 2.13: RADIUS-NAS-Glued-Auth
2.3.3 Calculate the $S_{gl}^+$ and the possible traces

After we got the Glued-IOLTS model, it still need to be added the GIB and the refusal graphs to represent the invalid inputs and inopportune inputs. The $S_{gl}^+$ can be obtained and presented as Figure 2.14. The Robustness test cases are calculated through the algorithm proposed before, which is a process of tracing back. The computed traces represent the test skeletons. For example, one possible trace of Figure 2.14 after calculation is \{!Ac_rej, ?Ac_rej,n, !Ac_req,n, ?Known_id, ?Ac_req, !Ac_rej,n, Tau, Tau, ?Ac_req,n, !Ac_req,n, ?Known_id, ?Ac_req\}. This is a simple trace which does not include the invalid inputs and inopportune inputs. When calculate this trace, the algorithm trace back through \{!Ac_rej, S16, ?Ac_rej,n, S14, !Ac_req,n, S12, ?Known_id, S11, ?Ac_req, S10\} to the initial state of S1, then the algorithm check the past trace and find there is an element of the stimulate pair \{?Ac_rej,n, !Ac_rej,n\}. So the algorithm adds the corresponding action (!Ac_rej,n) into the trace, and move to S2 to continue the traceback. When it reaches the initial state of S2, the passed trace is \{!Ac_rej, S16, ?Ac_rej,n, S14, !Ac_req,n, S12, ?Known_id, S11, ?Ac_req, S10, !Ac_rej,n, S24, Tau, S22, Tau, S21, ?Ac_req,n, S20\}. The algorithm check the past trace again, and find another unfinished stimulate pair \{?Ac_req,n, !Ac_req,n\}. Then as it has done before, it add the corresponding action (!Ac_req,n) into the trace, and move to S1 to continue the traceback process. This recursion will end until it cannot find the unfinished stimulate pair in the past trace. The first 10 calculated traces, which contain at most one possible invalid input are presented in Listing 2.2. Totally there are 16397 possible traces by considering the invalid inputs. For more details of the calculated traces by considering the invalid inputs, please check AppendixB.
Figure 2.14: RADIUS-NAS-Glued-Auth-Plus

Trace1: {?Ac_req, ?new_id, !inq_Auth;}
Trace2: {?Ac_req, ?new_id, A1_3, !e, !inq_Auth;}
Trace3: {?Ac_req, A1_1, !e, ?new_id, !inq_Auth;}
Trace4: {A1_0, !e, ?Ac_req, , ?new_id, !inq_Auth;}
Trace10: { ?Ac_req, A1_1, !e, ?Known_id, !Ac_req_n, ?Ac_req_n, tau, tau, !Ac_accept_n, ?Ac_req, ?Known_id, !Ac_req_n, ?Ac_accept_n, !Ac_accept;}

Listing 2.2: Results of Trace back
2.3.4 Assess the robustness of the networked implementations

At the end of this approach, the calculated traces should be translated into test cases. Considering it is a black-box test and only the observable actions could be considered as the test case. Although the internal actions such as \( \text{tau}, (\text{Ac}_\text{req}_\text{n}) \) etc. are important, they should be eliminated from the test case. For example, the Trace10 of Listing 2.2 express a complex process of RADIUS protocol: The Client (or Adversary) sends an Access Request (Ac\_req) to the NAS Server. After the NAS Server receives this message, it modifies its state for receiving the authentication messages. But the following message arrived is an insecure message (A11), which is not executable by the NAS Server. The NAS Server is a robustness implementation of RADIUS protocol, so it can accept the insecure inputs without any insecure response (in this example, the NAS server ignores the insecure message) and generate an output (e) to record or identify this error. Then the certificated information arrived at NAS server (?Know\_id), then the NAS Server checks this user id, and finds it is an already known id (Known\_id), and sends a message \{Ac\_req\_n\} with the client’s security information to the RADIUS Server. Then the RADIUS Server checks the client’s encryption method, and finds it is supported by the server (the first \( \tau \)), then it checks the authentication of the client (the second \( \tau \)), and sends back a \{Ac\_accept\_n\} message to the NAS Server. The NAS Server then sends a \{Ac\_accept\} to the client (Adversary) to accept the connection. The test case transformed from Trace10 of Listing 2.2 are \{Ac\_req, A11, e, ?Know\_id, Ac\_accept\}, which identifies if the input sequence of the implementation is \{Ac\_req, A11, ?Know\_id\}, the output of the implementation could be \{e, !Ac\_accept\}.

When the tester runs these test cases on the implemented system, and find out the system is blocked, re-start, or even crash down; then we know that this system is not robust, and the problem happens in the corresponding state. After executing all the test case of the selected traces, if the tested system can work well, then we say this system is robust, which means the system can be secure when receiving the Malicious Requests Attacks.

2.4 Conclusion

In this chapter, we propose our method to verify the security of protocol implementation through robustness. We believe by modeling the network system, and checking its robustness, the potential security defects can be detected and then be fixed. We use
the concurrent components to model and simulate the network system, and we extend the definition of Labeled Transition System to express this multiple components system. Then we give an approach and tools to the robustness tests generation. The security RADIUS protocol is introduced in this chapter also, and is taken as a case study to explain our method. This protocol will be also used later in other chapters.
Chapter 3

Security Testing with Insecure Medium

3.1 Problems and Assumptions

In Chapter 2, we use Input/Output Labeled Transition to model the components of the networked and concurrent systems, then glue those components together through a secure medium (within the model of Extended Glued-IOLTS). By using this approach, the robustness transition sequences, which may contain several DoS or Threatening Request attacks, are calculated and verified. But this approach assumes the connected medium is secure and untouchable, which is not satisfied in many actual cases. For example, authentication protocols (such as NSL protocol) usually consider the system components are untrusted and the transmitted messages can be modified by the intruder (Man-in-the-middle attacks and replay-attacks). To adapt this kind of security requirements, a more comprehensive model, which should concern the important protocol security properties (atoms like nonce, encrypted messages, configures, etc.) and the possible intruders, is needed.

In this chapter, we present our transition model to adapt this requirement. We define the critical security properties into the system transitions, by extending the classic IOLTS to be a Secure IOLTS (S-IOLTS). We follow the Dolev-Yao assumptions and model the intruders with this S-IOLTS. Then those modeled intruder machines are used to construct an insecure medium, which is going to be used to glue the multiple system components together [34].
3.2 Secure Transition Models

Although the classic IOLTS can describe the system components with simple and clear semantics and reachable graph, it still can’t satisfy some modeling requirements for some security authentication protocols. In those protocols, the security features (such as nonce, encrypted methods, etc.) participate in the process of state transition. Those essential features should be considered, but the class IOLTS model does not define them and make it is hard to describe some security protocol. In order to analyze the authentication problems of protocol components, we extend the definition of transition of IOLTS by defining guards and actions (similar as the idea of EFSM), and model atoms and variables into the machine to form a secure IOLTS model (S-IOLTS) [33].

Definition 4.1 (S-IOLTS):

A S-IOLTS represents 6-tuple set: $M = \langle S, L, s_0, A, X, T \rangle$, where

- $S$ is the finite set of states;
- $L$ is the finite of labels, which contains input and output labels;
- $s_0$ is the initial state;
- $A$ is a finite set of atoms with certain derivation rules, and $L(a)$, $a \in A$, is the set of labels using atoms $a$;
- $X$ is a finite set of variables;
- $T$ is a finite set of transitions. for $t \in T$, $t = \langle s, s', l, \text{pred}(x, \pi(l)), \text{act}(x, \pi(l)) \rangle$, where $s$ and $s'$ represent origin and target state of the transition; $l \in \{L \cup \tau\}$, $\tau$ represent the internal actions. $x \in X$; $\pi(l)$ represents the parameter of $l$ which receive the variable $x$, $\text{pred}(x, \pi(l))$ is a predicate, called as guard; and $\text{act}(x, \pi(l))$ represents the actions on the current values and parameters.

Comparing with the existed EFSM model, the proposed security extension of IOLTS focusses on the requirements of security protocols. It defines the atoms of the label $l$ as a new tuple of the transition model. This atom $a$ is one element of the contents of the message $l$. It may be one variable or constant or may be consisted by several variables and constants. In some practical protocols, some atoms are calculated from other atoms, for example, in RADIUS protocol, the nonce sent from NAS server is an atom, which is
a result calculated from the nonce received (an atom of the client) and the its NAS id (an atom of NAS). We call such dependency relationship as Derivation Rules. The \( \pi(l) \) represents a parameter of \( l \), which has the same type as \( x \), and can receive the value of variable \( x \). For example, if a message label \( l \) is \( \{ \text{Hash}(\text{nonce}_i, \text{sid}) \} \), then the \( \text{nonce}_i \) and \( \text{sid} \) are the possible \( \pi(l) \) of \( l \).

**Example 4.1**: NSL protocol is a security authentication protocol, which is usually taken as an example to explain the analyzing methods of security protocols. In NSL, two components are included: initiator and receiver, which need to use unique nonce and asymmetric cryptography to keep the communication security. The protocol narration simply contains three message transitions, which are: \( \{ i \rightarrow r: \{ n_i, i \} \}_{pk_r} \), \( \{ r \rightarrow i: \{ n_i, n_r, r \} \}_{pk_i} \) and \( \{ i \rightarrow r: \{ n_r \} \}_{pk_r} \). Figure 5.3 models the initiator of NSL by using \( S-IOLTS \). The state marked as “0” in the figure is the state \( s_0 \). Other states also follow the same rule. The initiator of NSL sends its nonce, which is encrypted with the public key of the responder, to the responder; and waits for the response from the responder. Once the corresponding response arrived, it needs to decrypt it with its private key and verify the nonce received. If the nonce received is the same as the nonce it sent to the responder, it sends the nonce of the responder back. Otherwise, it sends an error to identify the communication is not secure.

In this example of S-IOLTS, the \( S=\{ s_0, s_1, s_2, s_3, s_4 \} \) is the finite set of states; \( L = \{ ?\text{init}, !m_r1, ?m_i1, ?m_r2, ?e, ?\text{Ack} \} \) is the finite set of labels; \( s_0 \) is the initial state, and each time the machine left this state, a new session is created (\( \text{sid} \) increases); \( A=\{ \text{sid}, \text{nonce}_i, pk_i, pv_i, pk_r \} \); \( X=\{ x_1, x_2, x_3, x_4 \} \), where \( x_1 \) is used to record the \( \text{nonce}_i \) in \( !m_r1 \) through the action \( \text{Get}_n1(m) \), and \( x_2 \) is used to record the \( \text{nonce}_r \) in \( !m_r1 \) through the action \( \text{Get}_n2(m) \); \( x_3 \) records the session id of the received message, this is through the action \( \text{Get}_s(m) \); and \( x_4 \) records the result of description through action \( \text{Dec}(m, \text{key}) \). There are 7 transitions in this example, which are listed in Table 3.1. In S-IOLTS, we still use \( \tau \) to denote the internal actions, but we add sequence numbers behind \( \tau \) to distinguish different internal actions.

By using this model, the important security properties are emphasized and concerned in the transitions of the system.
3.3 Intruder Model and SG-IOLTS

Modern systems usually contain multiple concurrent components, which may be deployed distributed but cooperate with each other through the exchanged messages. The

<table>
<thead>
<tr>
<th>s</th>
<th>s'</th>
<th>label</th>
<th>Guards</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₀</td>
<td>s₁</td>
<td>ᵪinit</td>
<td></td>
<td>sid++</td>
</tr>
<tr>
<td>s₁</td>
<td>s₂</td>
<td>m_r₁</td>
<td>(x₃ = sid)</td>
<td></td>
</tr>
<tr>
<td>s₂</td>
<td>s₀</td>
<td>ᵪe</td>
<td></td>
<td>x₃ = Get_id(m_r₁), x₄ = Dec(m_r₁, pv₁)</td>
</tr>
<tr>
<td>s₂</td>
<td>s₃</td>
<td>m_r₁</td>
<td>(x₃ = sid) &amp;&amp; (x₄ &gt; 0) &amp; x₁ = nonceᵢ</td>
<td>x₁ = Get_n1(m_r₁), x₂ = Get_n2(m_r₁)</td>
</tr>
<tr>
<td>s₃</td>
<td>s₄</td>
<td>m_i₂</td>
<td></td>
<td>x₁ = Get_n1(m_r₁), x₂ = Get_n2(m_r₁)</td>
</tr>
<tr>
<td>s₄</td>
<td>s₀</td>
<td>ᵪAck</td>
<td></td>
<td>x₃ = Get_id(m_r₁), x₄ = Dec(m_r₁, pv₁)</td>
</tr>
</tbody>
</table>
security verifications over such kind of system usually need to consider those multiple components together as a whole system to guarantee the security [30]. While the implementations of security protocols naturally contain multiple components (at least one initiator and one responder), we should join those components first, then verify the security of the entire reactive system.

With the proposed S-IOLTS, the distributed components can be modeled as discrete transition systems. We need a way to combine them together and a method to analyze their security. Some researchers solve the combination problem by defining relations (such as Simulation, Bisimulation, Ready-Simulation, etc.) [51, 76], which can join and present the multiple components in a formal way. Others intended to model the connection medium, and use it to glue the distributed components together [32, 38]. Considering the purpose of our security verification, the computable transitions traces should be the core consideration and we should follow the second approach to combine the components.

### 3.3.1 Modeling of the Medium

The common medium exists between components and transmits messages to the connected components to finish the communication. It is usually abstracted as an infinite memory, which can record and forward the passing messages with an order of first in first out (FIFO). But in modern security theories, this universal medium becomes not simple: a potential intruder may exist in the medium and control the messages through it. The transmitted messages may be learnt, deflected, and modified when it passes through the medium.

**Intruder Model** An intruder may serve as a system components and attempt to attack the system. It is a powerful component and usually has the abilities to eavesdropping, insertion or interception over the transmitting or transmitted messages. One important question of security protocol analysis is how to model the intruder.

The intruder usually has a set of pre-defined atoms, which is called as *Initial Knowledge*. This knowledge set may be updated by eavesdropping the exchanging messages. After the intruder eavesdropping the transmitted message, it may forward the message to the next component or modify the current message by using its initial knowledge. The intruder can also initially generate an attack message and send it to any component of
Chapter 3. Security Testing with Insecure Medium

54

Figure 3.2: General Intruder of S-IOLTS

the network. Figure 3.2 use the S-IOLTS to give two general models of intruder, where the functions of the intruder are described with states and transitions.

- Model 1 describes the intruder of “no-block”. In this model, the intruder learns knowledge from the received message ?m, and attack the system by modifying the contents of m or create new messages. The messages received by this model will not be blocked. The state set \( S = \{ s_0, s_1, s_2, s_3 \} \). The transition \( s_0 \rightarrow s_1 \rightarrow s_0 \) represents the process of eavesdropping a transmitting message. The transition \( s_0 \rightarrow s_1 \rightarrow s_3 \rightarrow s_0 \) represents the process of receiving message then generate attack messages. The transition \( s_0 \rightarrow s_2 \rightarrow s_0 \) represent the intruder initiates an attack directly to any components known in its initial knowledge.

- Model 2 presents the situation where the intruder blocks the received messages. This model usually has a close relation with the time parameters, and is not considered in our method.

Intruder transitions begin from the initial state \( s_0 \). When the intruder receives the input label (\(?m\) ), it learns this message and extends its knowledge (with action \( Update(K, m) \), where \( K \) represents the initial knowledge). If it does not receive the attack command (\(?att\) ), it forwards the received message \( m \) as an output (\(!m\) ) to the original receiver; if it receives the attack command, it modifies the received message \( m \) with its current
knowledge (through the action $\text{fake}(K,m)$), and sends the modified message $!m.o \in M.a$ to attack the next component; it may also create new attack messages according its extended knowledge (action presented by $\text{Create}(K)$), and send the result $m.o \in A.a$ to any component known by the intruder. The intruder can also trigger an attack without receiving any input message. The label set $L = \{?m,!m,?\text{att},!m.o,!a.a\}$ represents different kind of message inputs and outputs. The attack command ($?\text{att}$) is an internal action of intruder machine, which represents the attack decision of the intruder. The intruder has atoms set $A=\{\text{sid}, \text{Insid}, K\}$, where $K$ represent the knowledge of the intruder, and it may contain other atoms of $A$ ($K \subseteq A$). Normally, $\text{sid}$ and $\text{Insid}$ also belongs to the intruder knowledge $K$. The intruder uses intruder identification ($\text{Insid}$) to record the sequence of the interactions between intruder and different components. $X=\{x_1,x_2,\ldots,x_m\}$ is a variable set, where $m$ equals to the number of updated atoms of knowledge $K$.

The transition actions of intruder:

- $\text{Update}(K,m)$ presents the learning process of the intruder. If $A_m$ represents the set of atoms of message $m$, $K$ represents the knowledge of the intruder, then: $K \xrightarrow{\text{Update}(K,m)} K \cup A_m$;

- $\text{fake}(K,m)$ presents the action of modifying the received message to be attack message. The modified message must match the format of $m$. If $m'=\text{fake}(K,m)$, $A'$ represents the atom sets of $m'$, then: $A' \cup A \subseteq K$, $A' \cap A \neq \emptyset$ and $m'$ match $m$;

- $\text{Create}(K)$ presents the action of creating a new message from the knowledge of $K$. The generated message $m'$ may be a robustness message to the system;

- $\text{Insid}++$ presents the increasing of the intruder identification. Because multiple intruders may exist in the system, the $\text{Insid}$ is defended to identify them.

**Insecure Medium and SG-IOLTS Model** In our previous work of [30], a definition of secure medium, which assumed the components of system are trusted and the common medium is untouchable, was proposed and was used to glue the components together. This secure medium is defined by the transitions, which starts from the lower level state of one component and end at the lower level states of another component. But this secure medium is not suitable for security protocol, which basically concerns the
components are not trusted. To adapt this requirement, we put the intruder model into the common medium and propose an insecure medium for security protocol.

**Definition 2 (Insecure Medium):** An insecure medium is a common medium which contains a set of intruder transition machines \(\langle S_{int,i}, s_{int,i,0}, L_{int,i}, A_{int,i}, X_{int,i}, T_{int,i}\rangle\), and connect to the system components with transitions between lower level states.

We use \(M_{medium}\) to denote the insecure medium and use \(M_{int,i}\) to denote the \(i\)th intruder machine in the medium. If two components \(M_i\) and \(M_j\) \((i \neq j)\) are connected through the medium (see Figure 3.3), and \(M_{int,i}\) is the intruder between \(M_i\) and \(M_j\), then the transition \(t_{ij}\) (sent from component \(M_i\) to \(M_j\)) should be presented with three parts: the transition between \(M_i\) and \(M_{int,i}\) \((t_{int,i})\), the transitions in \(M_{int,i}(t_{int,i})\), and the transition between \(M_{int,i}\) and \(M_j\) \((t_{int,j})\). \(t_{int,i}\) can be presented as: \(s_{int,i,lower}^{m} \rightarrow s_{int,i,0,lower}\), where \(s_{int,i,lower}\) represents the lower interface of the \(m\)th state of \(M_i\) and \(s_{int,i,0,lower}\) represents the lower interface of the initial state of \(s_{int,i}\). \(t_{int,i} \in T_{int,i}\). And \(t_{int,j}\) can be presented as: \(s_{int,i,0,lower}^{n} \rightarrow s_{j,0,lower}\), where \(s_{int,i,0,lower}\) represents the lower interface of the \(n\)th state of \(M_{int,i}\) and \(s_{j,0,lower}\) represents the initial state of \(M_j\).

With this insecure medium, the multiple system components can be glued together, the Glued Security IOLTS model is considered as follow:

**Definition 3 (SG-IOLTS):**
A SG-IOLTS represents a set of S-IOLTS \( \langle S_i, s_i, 0, L_i, A_i, X_i, T_i \rangle \) \((i=1, \ldots, n)\) glued by an insecure medium \(M_{\text{medium}}=\{M_{\text{int}_0}, \ldots, M_{\text{int}_n}\}\), which is a 6-tuple set:
\( \langle S_{sg}, L_{sg}, s_{0sg}, A_{sg}, X_{sg}, T_{sg} \rangle \). It is an extension of Glued-IOLTS, where transitions are defined with predicates and actions.

**Example 2:** the SG-IOLTS of NSL protocol is presented in Figure 3.4. The two S-IOLTS components are connected with the medium, which contains 4 intruder machines. For space reason, some predicates of the transitions are omitted.

### 3.4 Verification of Security Protocol Implementation

Testing is the principal protocol verification method, which is a verification activity consisting of extracting knowledge from an Implementation Under Test (IUT) [51]. The main difficulty of protocol testing is that: the IUTs are black-boxes. They need to use test cases or test sequence (observable sequences of actions) to execute during the test experiment to deduce the unobservable information. By using the proposed SG-IOLTS model, the security properties and actions are defined inside of the system transitions, and the multiple components are connected through the insecure medium. A transition sequence for one secure session is going to demonstrate the possible changes of security
properties, possible intruder actions, and the corresponding reactions. This kind of transition sequence is usually an attack scenario to the protocol system. If we obtain those possible scenarios, and execute them on the protocol implementations, we can verify the security of the protocol implementations.

3.4.1 Test Architecture

A typical test architecture for multiple components testing can be simply presented in Figure 3.5a, where the test system obtains the messages through the upper interface. In interoperability context, each of the IUTs has two kinds of interfaces. The lower interfaces are the interfaces used for the interaction of the two IUTs. These interfaces are only observable but not controllable. It means that a tester connected to such interfaces can only observe the events but not send stimuli to these interfaces. The upper interfaces are the interfaces through which the IUT communicates with its environment. They are observable and also controllable by the tester.

The test architecture of SG-IOLTS can be presented as Figure 3.5b, where the mediums are modeled and under control of the test system. In this case, although the internal messages of system components are still sent from the lower interfaces, the test system can obtain and control them through the medium.
3.4.2 Transition Traces Generation

One benefit of the reachable graph is on clearly presenting the transitions of the protocol communication. Through the reachable graph, the possible transition traces can be calculated, which are test cases to verify the protocol implementation. By using the SG-IOLTS, a security protocol and its intruder are modeled within a reachable graph, if we can find the suitable algorithms, the test cases to verify the security of the protocol implementations can be calculated.

**Traceback Algorithm**  The traceback algorithm is a kind of deep first search algorithm, which is used in Chapter 2 to calculate the traces of the determined finite state machine. The algorithm search for the connected transitions and record the passed traces. We recall this algorithm and present in Table 3.2, where we use \( t_{pre\_state} \) to present the source state of the transition \( t \); and the expression \( t_i \) means the \( i^{th} \) transition of the transition set \( T_{record} \). The trackback algorithm stops when the initial state is obtained. But our test problem contains multiple system components, and multiple initial states are existing. So we need to use the another algorithm, which is call “check_glue”, to decide whether to stop the traceback or to direct the traceback to another component. This check_glue algorithm is presented in Table 3.3. The expression \( t_j.stemuli \) is used to calculate the corresponding stimuli transition of \( t_i \), which is a transition of the medium. The stimuli transition represents an output transition of one component, which can stimulate another component/components to change the states. If this stimuli transition is found, then the traces are guided to the medium or another component, the traceback algorithm can be used again to calculate the transition traces.

These algorithms can be used to calculate the possible transition traces in the concurrent components system with secure medium. But now, we need to consider the transition traces of the intruder machines which existed between system components. In this case, when the “traceback” arrives at the initial state of the component, it will find the connected transitions are the outputs of the intruder machine, which are not written under the message format of protocol. For example, in NSL protocol (see Figure 3.4), one possible traces back from \( s_0 \) of initiator \( i \) may direct the traceback as: \{(?e, [], []), \{m_i,1, [], []\}, (?init, [], [sid++])\}. Then a “initial” state \( S_{i_0} \) is reached, and the “check_glue” algorithm should be used. So the algorithm checks in the past transitions and find (?e, [], []) need a stimuli transition and try to search it (message match !e) from the connected transitions of the lower interface of \( S_{i_0} \), which is the outputs of intruder.
Table 3.2: Traceback Algorithm

<table>
<thead>
<tr>
<th>Input:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Glued Transition System M</td>
</tr>
<tr>
<td>Transition Array $trace_i$, $T_{record}$</td>
</tr>
<tr>
<td>Transition $t$</td>
</tr>
<tr>
<td>Output:</td>
</tr>
<tr>
<td>Transition sequences $trace_i$</td>
</tr>
<tr>
<td>Begin</td>
</tr>
<tr>
<td>record $t$ in $trace_i$;</td>
</tr>
<tr>
<td>If $t$.pre_state is initial state{</td>
</tr>
<tr>
<td>check_glue($trace_i$, M);</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>else{</td>
</tr>
<tr>
<td>record all transitions point to $t$.pre_state.higher_interface in $T_{record}$;</td>
</tr>
<tr>
<td>For each $t_i$ $\in$ $T_{record}$ {</td>
</tr>
<tr>
<td>record $t_i$ in $trace_i$;</td>
</tr>
<tr>
<td>traceback($t_i$, $trace_i$, M);</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>End</td>
</tr>
</tbody>
</table>

Table 3.3: check_glue Algorithm

<table>
<thead>
<tr>
<th>Input:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Glued Transition System M</td>
</tr>
<tr>
<td>Transition Array $trace_i$</td>
</tr>
<tr>
<td>Output:</td>
</tr>
<tr>
<td>Transition Array $trace_i$</td>
</tr>
<tr>
<td>Begin</td>
</tr>
<tr>
<td>For each transition $t_j$ in $trace_i$ {</td>
</tr>
<tr>
<td>If $t_j$.stimuli is not empty and is not in $trace_i$ {</td>
</tr>
</tbody>
</table>
|     record $t_j$.stimuli in $trace_i$;  
|     set $t_j$.stimuli as empty;        |
|     traceback($t_j$.stimuli, $trace_i$, M); |
|   }                                  |
| }                                    |
| End                                  |
machine: !m, !a, or !m, or !m, or !m. But clearly, the search will be stopped because those outputs of the intruder are not written as the format of protocol components. However, the message !m is only a forward of the received message from the previous component. Its type should be the same as the received message. If the intruder received !e from the previous component, then the output !m will match !e, and this !m should be added to the trace of the “check glue” algorithm, the “traceback” should continue. In order to solve this problem, We propose the modified check glue algorithm, which is presented in Table 3.4, to lead the trace to intruder machines or system components.

As before, we use t to represent the transition, which is going to be checked; M to represent the transitions set of the glued system (SG-IOLTS); tracei represents a transition array, which records the visited transitions; We define T_lower as a transition array, and use it to temporarily record the transitions connected to the lower_interface of the same state; and trace_hassti is an ArrayList of transition. It is used to identify the transitions of tracei, which has the corresponding stimuli transitions.

In this method, the initial states of the intruder machines are considered. When the algorithm arrives at the initial state, according to the labels of the current transition (we assume that the labels used by the intruder machines are reserved), the possibilities of the resulting components can be deduced. We also define intruder_id to identify the intruder machines when multiple intruder machines may be connected to the same state. In this situation, the algorithm will obtain the next intruder id int_id from the transitions connected to the lower_interface of the end state of the current transition, and continues to traceback to the one whose int_id equals to intruder_id. The initial value of intruder_id is set up when the first initial state is found (the first time to use check glue). The value of trace_hassti.length identifies how many transitions in the trace have stimuli transition, and this value also identifies how many intruder machines are connected to this initial state. According to this, the initial value of intruder_id can be calculated. When the trace leaves the intruder machine, the intruder_id = intruder_id -1.

In the real protocol cases, after each protocol session, the transition traces will return back to the initial states and wait for the next protocol session. We define the transitions ended with initial states as “end transition” of each protocol session. When using those algorithms to generate the test cases, the transitions t of the “end transitions”, should be considered as the primary input transitions of the traceback algorithm. Then the algorithm will automatically find all those connected transitions and store them as a transition sequence, which is the possible test cases.
Table 3.4: Modified check glue Algorithm

Input:
Determined Transitions System M;
Transition Array trace_i;
checked transition t;
Output:
Transition Array trace_i,
Begin
String message_type = null;
Int intruder_id;
ArrayList<transition> trace_hassti;
Transition Array T_lower,
For each transition t_j in trace_i{
    If t_j.stimuli is not empty {
        record t_j in trace_hassti;
    }
}
If (trace_hassti.length > 0){
    record all transitions point to t_j.pre_state.lower interface in T_lower;
Get intruder id of T_lower[(trace_hassti.length-1)*3] and record it in intruder_id
For each t_l in T_lower{
    If t_l.label equals to ?att{
        record t_l in trace_i;
        For each t_m in trace_hassti {
            trace_hassti.get(m).stimuli is empty;
            intruder_id = intruder_id - 1;
            output trace_i;
        }
    }
    If t_l.label equal to ?m{
        If message_type is not empty{
            If t_l.label equals to message_type{
                For each t_m in trace_hassti {
                    trace_hassti.get(m).stimuli is empty;
                    intruder_id = intruder_id - 1;
                    traceback(t_l, trace_i, M);
                }
            }
            else{
                For each t_m in trace_hassti {
                    trace_hassti.get(m).stimuli is empty;
                    intruder_id = intruder_id - 1;
                    traceback(t_l, trace_i, M);
                }
            }
        }
        else{
            For each t_m in trace_hassti {
                trace_hassti.get(m).stimuli is empty;
                Get intruder id of t_l and records as int_id;
                If (int_id == (intruder_id)){
                    If t_l.label equals to lm{
                        message_type = trace_hassti.get(0).stimuli.label;
                        record t_l in trace_i;
                        traceback(t_l, trace_i, M);
                    }
                }
            }
        }
    }
}
End
3.4.3 Test Verdict

The SG-IOLTS model considers the intruder as part of the system. By using the proposed algorithm over SG-IOLTS, the generated test cases may contain the actions of intruders, which make the transition sequences become the possible attack scenarios. When a secure implemented IUT executes these kind of test cases, it should be able to distinguish those attacks, report or ignore this attack, then execute some positive actions to protect the IUT from the attacks (for example, stop the execution of the test case). If the IUT cannot distinguish the attack scenarios, this IUT must have some security vulnerabilities. According to this idea, we give the following Test Verdict to judge the test results:

Definition 4.4 (Test Verdict):

If the considered test sequence contains label $?\text{att}$, and if after executing such case on the implementation, the implementation results a same output as the test sequence, then we say this implementation is implemented insecure. The Test Verdict is Fail. Otherwise, the Test Verdict is Pass.

The label $?\text{att}$ identifies the attack attempts of the intruder, test sequences contain label $?\text{att}$ must contain the malicious attack messages from the intruder also. The kind of sequences is possible attack scenarios in the system. If a security implemented IUT executes the kind of test cases, it should be able to distinguish those attack messages and results errors or quiescence, then stop the test execution. But the test cases we used are generated from the SG-IOLTS model, which assumes the intruders have the abilities to forge the attack messages as regular messages. Test cases with $?\text{att}$ demonstrates the successfully attacks to the protocol implementation. So if the test results are the same as the test cases, a successful attack happened to the verified implementation. So this protocol implementation is insecure and the test verdict should be “Fail”.
3.5 Case Study Results

3.5.1 An Example of NSL protocol

We use the algorithms presented to calculate possible test cases for NSL protocol. The SG-IOLTS reachable graph of NSL is presented in Figure 3.4. Computed possible transition traces are presented in Listing 4.1. For space limitation, we omitted the guard and actions of the transitions, and use ?att, [K > 0], [Insid + +, A.a = Create(K)]; ?att.1 represents the transition ?att, [], [M.a = fake(K, m), A.a = Create(K), M.o = (M.a) ∪ (A.a)], ?m.i1.l to represent the transition (?m.i1, [x4 < 0], [x4 = Dec(?m.i1, pv.r)]) from s.r0 to s.r1; ?m.i1.r represents the transition from s.r0 to s.r2; ?m.i2.l represents the transition from s.r3 to s.r4; ?m.i2.r represents the transition from s.r3 to s.r0; ?m.r1.l represents the transition from s.i2 to s.i3; m.r1.r represents the transition from s.i2 to s.i4.

By using the proposed algorithm, 148 traces are found automatically for NSL protocol. We only list the first 20 tracks in the Listing 3.1. The traceback will begin from the end transition of the initial components, which are the one ended with the initial states. In this example, the end transitions are \{(i.s2, ?m.r1.l, i.s0), (i.s2, ?e, i.s0), (i.s4, ?Ack, i.s0)\}. The program needs to traceback from each transition of this set. For example, the 15th trace is calculated by the following processes: The Algorithm begin from one end transition of the initiator i: \{i.s4, ?Ack, i.s0\}. The traceback algorithm check the state i.s4 and find it is not an initial state, so it calculates all the transitions connected to its higher interface and record them in T.record. Here there is only one transition: \{i.s3, !m.i2, i.s4\}. We then add this transition into the trace, and continue to trace back until the transition \{i.s0, ?init, i.s1\} is found, where i.s0 is the initial state, and the modified check_glue algorithm should be used. In this state, the past recorded trace \{trace\} = \{?Ack, !m.i2, ?m.r1.r, !m.i1, ?init\}, where ?Ack and ?m.r1.r have stimuli transitions. We then need to calculate all the transitions connected to the lower interface of state i.s0. And the value of Intruder.id is going to be set as the id of the second connected intruder. The message_type is set to be “Ack”. Then the traceback is recalled to calculate all connected transitions with id equals to the intruder.id. In the trace 15, we take \{int4.s1, i.s0, !m, [], []\} as the transition, and continue to trace back until the initial state of Intruder 4. Then the recorded trace is \{?Ack, !m.i2, ?m.r1.r, !m.i1, ?init, !m, ?m\}, and we call check_glue. Because the label of the transition is like ?m, and the message_type is not null, so the algorithm goes to the transition, which connects to the lower interface of state int4.s0 and has a label of message_type. The transition is \{r.s5,
The computed sequences contain the actions of protocol implementations and the intruder, which then are used to direct the test cases.

3.5.2 Computed Transition Sequences to Verification Scenarios

The computed sequences contain abstract actions of the intruder, which is not included in the protocol specification and cannot be executed by the protocol implementations. We need to transfer those actions to the understandable actions defined in the protocol.
As we said before, because the intruder model is given, the insecure medium is a white-box to the tester. The tester can construct messages to match the transition sequences required by the protocol implementation. So the transition ?m should be replaced by the next transition in the trace. !m is an output transition of the replaced ?m, and !m_o and a_o must match the first transition required by the previous component (marked as !m_o(match(mi)), where !m_o to match the message type of mi). The first 20 sequences after replacing the intruder actions with implementations actions are listed in Listing 4.2. For example, 15th traces of Listing 4.1 is: {?Ack, !m_i2, ?m_r1_r, !m_i1, ?init, !m, ?m, !Ack, ?m_i2_r, !m_r1, ?m_i1_r, !m_o, ?att, ?m, !m_i2, ?m_r1_r, !m_i1, ?init, !m, ?m, !m_r1, ?m_i1, !m, ?m, !m_i1, ?init}. By replacing the intruder actions with the protocol actions, the trace changes to be: {?Ack, !m_i2, ?m_r1_r, !m_i1, ?init, !Ack, ?Ack, !Ack, ?m_i2_r, !m_r1, ?m_i1_r, !m_o(match(m_i2)), ?att, ?m_i2, !m_i2, ?m_r1_r, !m_i1, ?init, !m_r1, ?m_r1, ?m_i1, !m_i1, !m_i1, !m_i1, ?init}. In order to show the meaning of these traces, we complete the transitions with guards and actions, and rewrite and present them step by step in a front to back order in Table 3.5.

<table>
<thead>
<tr>
<th>Transition</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>{i=0, i=1, ?init, [], []}</td>
<td>Initiator i begin</td>
</tr>
<tr>
<td>{i=0, int1=0, !m_i1, [], []}</td>
<td>Initiator i send !m_i1 to medium</td>
</tr>
<tr>
<td>{int1=0, int1=1, ?m_i1, [Insid++, Update(K, m_i1)]}</td>
<td>Intruder int1 receive ?m_i1, and learn it to its Knowledge</td>
</tr>
<tr>
<td>{int1=1, r=0, !m_i1, [], []}</td>
<td>intr1 forwards message to receiver r</td>
</tr>
<tr>
<td>{r=0, r=2, ?m_i1, [y1&gt;0], [y1=Dec(?m_i1)]}</td>
<td>r receive ?m_i1, and successfully decrypted it</td>
</tr>
<tr>
<td>{r=0, int2=0, !m_r1, [], []}</td>
<td>r send !m_r1 to the medium</td>
</tr>
<tr>
<td>{int2=0, int2=1, ?m_r1, [Insid++, Update(K, m_r1)]}</td>
<td>Intruder int2 receive ?m_r1, and learn it to its Knowledge</td>
</tr>
<tr>
<td>{int2=1, r=0, !m_r1, [], []}</td>
<td>int2 forwards message to i</td>
</tr>
<tr>
<td>{i=0, int1=1, ?init, [], []}</td>
<td>review transition 1</td>
</tr>
<tr>
<td>{i=0, i=2, !m_i1, [], []}</td>
<td>Initiator receive ?m_i1,</td>
</tr>
<tr>
<td>{i=0, i=2, ?m_i1, [sid++, [M_a=Fake(K, m)], A_o=Create(K), M_o=M_o cap A_o]}</td>
<td>successfully decrypt the message, and find sid is correct, nonce_i is correct.</td>
</tr>
<tr>
<td>{int3=0, int3=1, !m_i2, [], []}</td>
<td>Initiator send !m_i2 to the medium</td>
</tr>
<tr>
<td>{int3=1, i=2, i=3, !att, [], [M_a=fake(K,m)], A_o=Create(K), M_o=M_o cap A_o]}</td>
<td>m3 receives !m_i2 and learn it to its Knowledge</td>
</tr>
<tr>
<td>{int3=2, r=3, !m_r2, [], []}</td>
<td>int3 uses the Knowledge K to modify !m_i2,</td>
</tr>
<tr>
<td>{i=0, r=2, ?m_i1, [y1&gt;0], [y1=Dec(?m_i1)]}</td>
<td>or create a new message.</td>
</tr>
<tr>
<td>{r=2, int4=0, !m_r1, [], []}</td>
<td>int3 sends the attack message, which match !m_i2 to r</td>
</tr>
<tr>
<td>{r=2, int4=3, r=3, !m_i2, [], []}</td>
<td>transition in r when send !m_r1</td>
</tr>
<tr>
<td>{r=2, i=3, i=4, !m_r1, [y1&gt;0], [y1=Dec(?m_i1)]}</td>
<td>r receive !m_i2</td>
</tr>
<tr>
<td>{r=2, i=3, i=4, !m_r1, [], []}</td>
<td>successfully decrypt this message, and find sid is correct, nonce_r is correct.</td>
</tr>
<tr>
<td>{r=2, int4=0, !Ack, [], []}</td>
<td>r send !Ack to the medium</td>
</tr>
<tr>
<td>{int4=1, r=4, !Att, [], [Insid++, Update(K, m_i1)]}</td>
<td>int4 receives !m_i2 and learn it to its Knowledge</td>
</tr>
<tr>
<td>{int4=1, r=4, !Att, [], []}</td>
<td>int4 forwards this message to i</td>
</tr>
<tr>
<td>{i=0, i=1, ?init, [], []}</td>
<td>review transition 1</td>
</tr>
<tr>
<td>{i=0, i=1, i=2, ?m_i1, [], []}</td>
<td>review transition 10</td>
</tr>
<tr>
<td>{i=0, i=2, ?m_i1, [sid++, [M_a=Fake(K,m)], A_o=Create(K), M_o=M_o cap A_o]}</td>
<td>review transition 11</td>
</tr>
<tr>
<td>{i=0, i=2, ?m_i1, [sid++, [M_a=Fake(K,m)], A_o=Create(K), M_o=M_o cap A_o]}</td>
<td>transition in i after send !m_i2</td>
</tr>
</tbody>
</table>

Trace 3: ?att, !a_o(match(Ack)), ?init, !m_i1, ?m_r1_r, !m_i2, !Ack, \( \backslash \)}
3.5.3 Example of RADIUS Protocol

An example of RADIUS protocol, which is modeled by SG-IOLTS, is presented in Figure 3.6. In this example, three components are considered to communicate and four intruder machines are implemented in the medium between components.
The client component has 5 states, \( S = \{ s_0, s_1, s_2, s_3, s_4 \} \) is the finite set of states. \( L = \{ ?A_c.req, ?A_c.rej, ?A_c.accept, ?A_c.challenge, t_1, t_2, t_3 \} \) is the finite set of labels, where \( t_1, t_2, t_3 \) represents the internal actions. \( s_0 \) is the initial state. \( A = \{ m \} \), where \( m \) represents the required encryption methods by RADIUS protocol. \( X = \{ x_1 \} \), where \( x_1 \) is the variable to record the encryption method received from the NAS server. Action \( Get_m() \) get the encryption method from the message ?A_c.challenge.

The NAS component has 8 states, \( S = \{ s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7 \} \) is the finite set of the states. \( L = \{ ?A_c.req, ?A_c.rej, ?A_c.accept, ?A_c.challenge, A_c.accept, !A_c.req, !A_c.rej, !A_c.accept, !Inq.Auth \} \) is the finite set of labels. \( A = \{ m.n, id \} \), where \( m.n \) represents the support encryption methods by NAS and \( id \) records all active identifications to NAS. \( X = \{ y_1, y_2, y_3, y_4 \} \), where \( y_1 \) is the variable to record the identification of client and \( y_2 \) is the flag to say whether \( y_1 \) is known by NAS; \( y_3 \) uses to record the required encryption methods of RADIUS server and \( y_4 \) is the flag to say whether the NAS server support the required encryption method. Action \( Get_id() \) obtains the identification from the message ?A_c.req, then with the action \( check(id, y_1) \) to check the identification in \( id \). Action \( check.m() \) verify the encryption methods in the support method list.

The RADIUS component has 5 states, \( S = \{ s_0, s_1, s_2, s_3, s_4 \} \) is the finite set of the states. \( L = \{ ?A_c.req, !A_c.rej, !A_c.accept, !A_c.challenge, t_4, t_5 \} \) is the finite set of labels, where \( t_4 \) and \( t_5 \) are internal actions. \( A = \{ m.s, pwd \} \), where \( m.s \) represents the
encryption methods of the RADIUS server and pwd records the corresponding authentication passwords. \(X = \{z_1, z_2, z_3\}\), where \(z_1\) record the encryption method of message, \(z_2\) is the flag to verify whether the used encryption method of the received messages is right or wrong; \(z_3\) is the flag to verify the authentication. Action \(\text{check}_m()\), \(\text{Get}_m()\) are same as the actions of NAS component. Action \(\text{verify}_\text{pwd}()\) verify the authentication of the client.

By using the algorithm we proposed in the last section, 394 traces are calculated and we present the first 25 traces in the Listing 4.3. For space reason, we use the contents of labels to represent the corresponding transition. To distinguish the transitions with similar labels, we use \(?\text{att}_1\) to represent the transition \(?\text{att}, [K > 0], [\text{Insid} + +, A_a = \text{Create}(K)]\); \(?\text{att}, 2\) represents the transition \(?\text{att}, [], [M_a = \text{fake}(K, m), A_a = \text{Create}(K), M_o = (M_o) \cup (A_o)]\); \(?\text{Ac}_\text{req}, \text{I}\) represents the transition \(\{?\text{Ac}_\text{req}, [y_2 > 0], [y_1 = \text{Get}_\text{id}()], y_2 = \text{check}(id, x_1)\}\); \(?\text{Ac}_\text{req}, \text{R}\) represents the transition \(\{?\text{Ac}_\text{req}, [y_2 < 0], [y_1 = \text{Get}_\text{id}()], y_2 = \text{check}(id, x_1)\}\); \(?\text{Ac}_\text{req}, \text{N}\) represents the transition \(\{?\text{Ac}_\text{req}, [z_2 < 0], [z_1 = \text{Get}_m()], z_2 = \text{check}_m(z_1, m_s)\}\); \(?\text{Ac}_\text{req}, \text{N}, \text{R}\) represents the transition \(\{?\text{Ac}_\text{req}, [z_2 > 0], [z_1 = \text{Get}_m()], z_2 = \text{check}_m(z_1, m_s)\}\); \(?\text{Ac}_\text{challenge}, \text{N}, \text{I}\) represents the transition \(\{?\text{Ac}_\text{challenge}, [y_4 > 0], [y_3 = \text{Get}_m()], y_4 = \text{check}_m(y_3, m_n)\}\); \(?\text{Ac}_\text{challenge}, \text{N}, \text{R}\) represents the transition \(\{?\text{Ac}_\text{challenge}, [y_4 < 0], [y_3 = \text{Get}_m()], y_4 = \text{check}_m(y_3, m_n)\}\).

Similar as the example of NSL, the test cases are calculated from the end transitions of initial component, which is a set: \(\{(\text{client}_s, 2, \text{tau}_s, \text{client}_s, 0), (\text{client}_s, 3, \text{tau}_3, \text{client}_s, 0), (\text{client}_s, 4, \text{tau}_4, \text{client}_s, 0)\}\). For example, the Trace 5 of Listing 4.1 is calculated as the following process: the algorithm begins from the initial state \(\text{client}_s, 0\) of the client, and take the transition \(\{\text{client}_s, 2, \text{client}_s, 0, \text{tau}_2, [], []\}\) which is marked as \(\text{tau}_2\) in Listing 4.1, as the initial transition. The end state \(\text{client}_s, 2\) is not an initial state, so the algorithm go to calculate all transitions connected to its \(\text{heigher interface}\) of \(\text{client}_s, 2\). There is only one transition \(\{\text{client}_s, 1, \text{client}_s, 2, ?\text{Ac}_\text{req}, [], []\}\) connected to this \(\text{client}_s, 2\), so we add it into the trace, and traceback to this transition. This process continued until the transition \(\{\text{client}_s, 0, \text{client}_s, 1, ?\text{Ac}_\text{req}, [], []\}\) is added to the trace, and an initial state \(\text{client}_s, 0\) will be found. The check glue algorithm then is used to connect the trace to the intruder machine. Now the trace\(=\{\text{tau}_2, ?\text{Ac}_\text{req}, !\text{Ac}_\text{req}\}\), where \(?\text{Ac}_\text{req}\) has two stimuli transitions: the transition \(\{\text{nas}_s, 5, \text{nas}_s, 0, ?\text{Ac}_\text{req}, [], []\}\) and \(\{\text{nas}_s, 6, \text{nas}_s, 0, !\text{Ac}_\text{req}, [], []\}\) of NAS machine. Because they are not included in the trace, the algorithm go to calculate all transitions connected to its \(\text{lower interface}\) of this \(\text{client}_s, 0\). They are \(\{\text{int}_s, 4, \text{client}_s, 0, !\text{m}, [], []\}\), \(\{\text{int}_s, 4, \text{client}_s, 0, !\text{a}_a, [], []\}\).
and \{int4_s, client_s_0, lm_o, [], []\}. In trace 5, we add \{int4_s_1, client_s_0, lm, [], []\} to the trace, and because the label equals to !m, the message_type is set to be !Ac_rej. Then the traceback is reused to trace the transitions back to the initial state of the intruder (int4_s_0). At this moment, the trace are \{t2, ?Ac_rej, !Ac_req, !m, ?m\}. Notice that after the final check_glu, the stimuli transition of ?Ac_rej in the trace has been marked as empty. But ?m has stimuli transitions, which are the output transitions of NAS components. Then the lower connected transitions are calculated. Because in Trace 5, the message_type is not empty, so the transition, whose label equals to message_type will be added to the trace. Then the traceback algorithm is recalled to calculate the trace until the initial state of NAS server (nas_s_0), where the trace are \{t2, ?Ac_rej, !Ac_req, !m, ?m, !Ac_rej, ?Ac_rej_n, !Ac_rej_n, ?Ac_rej_n, !Ac_rej_n, !Ac_rej_n, !Ac_rej_n, t4, !Ac_rej, t5, !Ac_rej, t2\}. At this state, we found two transitions: \{?Ac_rej_n\} has one stimuli transition \{RAD_s_4, RAD_s_0, !Ac_rej_n\}, and \{?Ac_req\} has one stimulus transition \{client_s_0, client_s_1, !Ac_req\}. Because of the sequence order, the lower connected transitions of nas_s_0 are needed to be calculated. But at this state (nas_s_0), we found two intruder machines (intruder 3 and Intruder 1) are connected. Because the predefined intruder_id = 4 (it is an input of the program, which equals to the amount of the intruder used in the system), the transitions with int_id = 3, which identifies the transitions of intruder 3 are going to be considered. After passing the intruder machine, the intruder_id reduce 1 (intruder_id–). Then the traceback and check_glu algorithms is recalled alternately to construct the trace 5. The trace ends when all stimuli transitions are included in the trace. Other traces are calculated using the same processes.
3.6 Conclusions

In this chapter, we propose an extension of transition system to model the security protocols, where the security properties are considered as guards of the transitions, and a network intruder is considered by the insecure medium; then this insecure medium to glue the multiple components together. The model can help us to generate possible transition sequences, which are the abstract message scenarios to verify the implementations of security protocol. In the future work, we plan to classify those possible transition traces and make it possible to form an intrusion database for security protocol implementation, which can help the intrusion detection system to detect the possible threats.
Chapter 4

Security Objectives

As we discussed in last chapter, the network systems are composed of implementations of network protocols, and can be modeled through concurrent and networked components [16], [51], [31]. Some potential defects may happen in the protocol implementations which make the system risk to be attacked and those defects can be verified through the method we proposed. But this method we proposed is based on the usage of finite state machine, and with the complexity of the protocol increasing, the states and transitions which should be considered are growing explosively. Which will cause the computational time and memories become the problems during the test generation. In this chapter, we focus on this problem and propose an on-the-fly test method to solve it.

4.1 Problems and Assumptions

Generally the potential defects considered by the verification problem of network protocols can be divided into two categories [7, 8]: the defects existed in the design of the protocol, and the defects occurred during the implementing process. For the first category, the protocols should be well analyzed, and many methods and works, such as formal security proof methodologies, black-box security protocol analysis etc. have been proposed on this problem. For the second category, the method of protocol testing [53] is used before. Unlike other analysis approaches, testing requires concrete experiment against a real, physical implementation. The method we presented in chapter 3 can produce the appropriate and sound scenario to feed the experiment. We focus on the second category of protocol defects and proposed a protocol security testing method, which was referred to as protocol experiment. But because the model considered all
possible actions, the general problem of state space explosion occurred. This meant that the tools for automatic tests generation may fail when trying to calculate all the tests at the same time, in particular when malicious or undesired inputs occurred.

In this chapter, we face this problem, and define the security knowledge of the test engineer as security objective. A Security Objective captures one very specific attack scenario [35]. The objective is to test the protocol implementations in order to demonstrate their robustness against the specific attack addressed. Thus, in the test generation processes, the so-called security objective becomes the target. We propose a method that calculates the corresponding test (for the specific attack scenario) on-the-fly, i.e. without exploring the entire model, allowing us to avoid the possible explosion. The considered systems are under the assumptions of secure or insecure medium.

4.2 Description of Security Objective

The Security Objective describes the security purposes of the tester. It reveals the most important security concerns from the experiences and knowledge of the test engineers. It can be used to generate the security test cases on-the-fly. The Security Objective can also be expressed formally as an IOLTS.

Definition 4.1: Security Objective:
A Security Objective of a system \( \langle S; L; T; s_0 \rangle \) is a deterministic and complete IOLTS, it is a 4-tuple \( SO = \langle S^{SO}, L^{SO}, T^{SO}, s_0^{SO} \rangle \), where:

- \( S^{SO} \) is a finite set of states,
- \( L^{SO} \) is a finite set of events, \( L^{SO} \subseteq (L \cup Inv) \). While \( Inv \) represents the set of invalid inputs; \( Inv \cap L = \emptyset \),
- \( T^{SO} \) is a set of transitions.
- \( s_0^{SO} \) is an element of \( S^{SO} \) called initial state.

\( S^{SO} \) contains some states labeled with Accept or Reject, which are noted as \( Accept^{SO} \) and \( Reject^{SO} \). The trace before \( Accept^{SO} \) is accepted by the security objective; the trace before \( Reject^{SO} \) is rejected by it (i.e. it is not to be considered by the current test aim). \( L^{SO} \) contains all possible labels and includes the insecure inputs to the system, the word “complete” in the definition means that each state allows all actions,
Chapter 4. Security Objectives

i.e. \( \forall q \in S_{SO}, \forall a \in L_{SO}, q \xrightarrow{a}, \) and a trap state \( q \) has a loop on each action i.e. \( \forall a \in L_{SO}, q \xrightarrow{a} q. \) We use the label marked with * to express all the possible actions between states, the expression \( \{ \ast \setminus (a,b) \} \) means the possible actions except \( a \) and \( b. \) This is how we make sure that the Security Objective contains all possible transition of the system. In a real system, some security objectives does not necessary need to contain the \( \text{Reject}^{SO} \) state condition, but a condition with the state \( \text{Accept}^{SO} \) is necessary. Those sequences may contain loops, but we only consider one occurrence of loop in our algorithm for the minimum length of the test sequence.

Example 4.1:

Figure 4.1 shows a simple specification of an IOLTS system. In case of considering the uncertain inputs, we add two error dealing states (state e) as the simple security policies to form the Figure 4.2. If we have a Security Objective such as below:

**Security Objective statement.** The system can deal with the unconsidered or malicious inputs (noted as \( ?xxx,n \)) which follow the input action \( ?a \) and never block the system; otherwise, the unconsidered inputs should be refused. According to this security desire, we give the Security Objective as Figure 4.3.

With the help of the Security Objective, the possible test sequences can be divided into different subsets. We use \( M_{SO_n} \) to denote the sequences resulted from \( SO_n, \) and
Chapter 4. Security Objectives

Figure 4.2: Example 4.1 - With Simple Security Policy

Figure 4.3: Example 4.1 - Security Objective
\(M_{SO_1} \cup \ldots M_{SO_n}\) can be infinitely approaching to the test space of security. The test cases of \(M_{SO_n}\) are generated and executed one \(SO\) at a time. This solves the problem of explosion and make the achievement of the implementations quickly. During this process, only the sequences related to security objective are considered, it is smaller than considering the whole test space. In this article, we assume the tester has the required expertise to specify a good coverage set of Security Objectives.

### 4.3 Synchronous Product

A Synchronous product can compare different systems in a parallel way [45]. The security objective emphasizes the security concerns of the tester and should be considered parallel during the system execution. We need to use Synchronous Product to compare the security objectives and the specification of the protocol (SSO).

**Definition 4.2: Synchronous Product**

Let \(M = (S^M, L^M, T^M, s^M_0)\) be an IOLTS, \(SO = (S^{SO}, L^{SO}, T^{SO}, s^{SO}_0)\) represents the corresponding Security Objective. The synchronous product \(S \times SO\) is an IOLTS:

\[
SP = (S^{SP}, L^{SP}, T^{SP}, s^{SP}_0)
\]

equipped with two disjoint sets of states \(Accept^{SO}\) and \(Reject^{SO}\), and defined as follows:

- **its alphabet** is \(L^{SP} \subseteq (L^M \cup L^{SO})\)
- **its state set** \(S^{SP}\) is the subset of \((S^M \times S^{SP})\) reachable from the initial state \(s^{SP}_0 = (s^M_0, s^{SO}_0)\) by the transition relation \(T^{SP}\).
- **the transition relation** \(T^{SP}\) is defined by:
  \[
  \exists q, q' \in S^M, \text{ and } \exists p, p' \in S^{SO}, (q, p) \leadsto^{SP} (q', p') \iff q \leadsto^M q' \text{ and } p \leadsto^{SO} p'
  \]
- **Accept^{SP}** and **Reject^{SP},** are defined as follows:
  - \(Accept^{SP} = S^M \times Accept^{SO}\)
  - \(Reject^{SP} = S^M \times Reject^{SO}\)

**Example 4.2:**

The Synchronous product of example 4.1 is presented in Figure 4.4 below:

\[
L^{SP} = \{?a, ?b, !d, !z, !c, !m, !n, ?xxx_n, !xxx_1, !xxx_2\} \subseteq (L^M \cup L^{SO})
\]

The synchronous product makes the specification selected together with the Security Objective, and identifies which trace should be accepted and which should be ignored.
(rejected) by the selection. Finally, we need to consider the Accept (and Reject) states and generate the test cases with these traces. In this example, the considered traces should be the ones passed through \(\{s_0, ?a, s_1, ?xxx_n, e, !xxx_1, s_1, !d\}\), and \(\{s_0, ?a, s_1, ?z, s_0, ?b, s_2, ?xxx_n, e, !xxx_2, !b\}\).

![Diagram](image)

**Figure 4.4: Example 4.2 - Synchronous Product**

### 4.4 Security Objectives over Test Selection

After using the Synchronous Product on Security Objective (SO) and the analyzed Protocol System \(S^M\), the resulted model will identify the transition traces of \(S^M\), which satisfies the SO, with the end state of \(Accept(SO)\) or \(Refuse(SO)\). In this case, the transition traces of \(S^M\) are categorized by the proposed SO. If we can use this specific SO to guide the security test generations processes, the considered states and transitions during the calculating will be limited to the corresponding SO. Then by carefully choosing the SO, the state exposition problem of protocol testing can be
avoided. We adopt this idea to the test approaches proposed in Chapter 2 and Chapter 3.

4.4.1 Security Objectives over Test Generation with Secure Medium

Before using SOes over the test generation, we need to make sure the model of protocol system (the $S^M$) contains the transitions related to $SO$. In the method of “Test Generation with Secure Medium”, the considered attacks are “Threatening Requests” or “DoS attacks”, which are inopportune inputs or invalid inputs to the system. So the considered $SO$ may contain some inopportune inputs and invalid inputs in the security statements. As we described in the Chapter 2, the proposed Extended Glued-IOLTS uses “Refusal Graph” to model the inopportune inputs and uses “protocol security policy” to deal the invalid inputs. It contains the required security transitions, and should be taken as the $S^M$ of the test generation. The procedures of the “SO guided test generation over secure medium protocol testing” are presented in Figure 4.5. After giving the protocol specification, the components of the protocol are modeled by IOLTS firstly, then connected to become a Glued System which is modeled by the Glued-IOLTS. This Glued-IOLTS is extended later by adding Refusal Graph and GIB to make it contain all possible message inputs. Then the Security Objectives are used during the test selection, and the test cases are generated.

![Diagram](image)

**Figure 4.5:** $SO$ guided: test generation with secure medium

After we use the extended Glued-IOLTS to model the concurrent and networked components system, we can calculate all the possible traces of the system. Then by using
those traces, the robustness test cases can be generated. But the process of calculating
the possible traces could be very complex, and may need a lot of time and memory,
depending on the size of the system and the size of the state of each component. We
can also know that not all of the calculated possible traces are possible inputs of the real
system, some of the calculated traces perhaps never occurred to the real system. The
concept of security objective can be used to select traces, and make the test cases have
their purpose. For each Security Objective, the corresponding test cases are generated
then tested. With the Security Objective, the test cases are generated on-the-fly.

We develop a JAVA program to generate the protocol robustness test cases. The data
structure is presented in Figure 4.6. The basic elements of state machine, state, label
and transition are taken as classes in our data structure. Another class node is needed to
manage the corresponding states in system and medium. Each class has its properties
and functions, which identify the system information and make the execution of the
program become possible. Each state of the machine has corresponding one instance
of class “state” and two instance of class “node” (lower interface and higher interface).
The function “getState_father()” can return the state name of the current node. The
class “transition” is designed with two states and one label: pre_state identify the initial
state of the transition, nex_state identify the end, and label is the one between the two
state. Given a “transition”, we can easy to calculate its beginning state, end state and
label by call the functions of “getPre_state()”, “getNex_state()” and “getTra_label()”.
The properties “sti_transition” identifies the pair < stimulate; response >.

Base on these data structure, we give our algorithm in Listing 4.1 to calculate the
test cases. The algorithm is a depth first search, but it traverses from the end to the
beginning of each trace. The inputs to this algorithm are the Extended-Glued-IOLTS
and its security objectives. The algorithm uses recursions to trace back in the extended
Glued-IOLTS according to the conditions of the security objectives. The cycle of the
recursion will stop when the trace reaches the “initial” state and the passed trace does not
contain any unfinished stimulate sets (a pair (stimulate, response) defined in transition,
and t.getStiLabel() can return the corresponding transition in the pair). During the
trace back, the program make the pruning and select the traces accepted by the security
objective.

```
1 Inputs: Transitions of Glued IOLTS of Protocol: ArrayList<transition>
    tra_sp;
2   Security Objective: ArrayList<transition> so_lab;
3 Outputs: ArrayList<ArrayList> trace_list;
```
Figure 4.6: Data Structure

<table>
<thead>
<tr>
<th>state</th>
<th>transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>name : string</td>
<td>pre_state : node</td>
</tr>
<tr>
<td>higher_interface : node</td>
<td>nex_state : node</td>
</tr>
<tr>
<td>lower_interface : node</td>
<td>tra_label : label</td>
</tr>
<tr>
<td>status : string</td>
<td>ati_transition : transition</td>
</tr>
<tr>
<td>+set_state_name()</td>
<td>+setTra_label()</td>
</tr>
<tr>
<td>+get_state_name()</td>
<td>+getTra_label():label</td>
</tr>
<tr>
<td>+set_state_status()</td>
<td>+setPre_state()</td>
</tr>
<tr>
<td>+get_state_status()</td>
<td>+setNex_state()</td>
</tr>
<tr>
<td></td>
<td>+getNex_state():node</td>
</tr>
<tr>
<td></td>
<td>+setSti_transition()</td>
</tr>
<tr>
<td></td>
<td>+getSti_transition():transition</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>label</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>label_name : string</td>
<td></td>
</tr>
<tr>
<td>+set_label_name()</td>
<td></td>
</tr>
<tr>
<td>+get_label_name():string</td>
<td></td>
</tr>
</tbody>
</table>

```java
5  ArrayList<transition> accept_tran_sp;
6  ArrayList<transition> trace_list;
7  ArrayList<transition> temp_tra;
8  ArrayList<transition> L_sti;
9
10 public static void main(String args[]){
11     Search_label(tp_lab.get(tp_lab.size()));
12     For(transition tra:accept_tran_sp){
13         traceback(tra);
14     }
15     For(ArrayList trace:trace_list){
16         For(transition t:trace){
17             System.out.println(t.getTra_label.getLabel_name());
18         }
19     }
20 public void ArrayList<transition> search_Label(label l1){
21     for(int i=0;i<transt.size();i++){
22         if(transt.get(i).getTra_label().getLabel_name().
23            equals(l1.getLabel_name())){
24             accept_tran_sp.add(transt.get(i));
25         }
26     }
27 public void Traceback(transition tra){
28     temp_tra.add(tra);
29     int flag=0;
30     ArrayList<transition> temp_same_state=new ArrayList<transition>();
31     if(tra.getPre_state().getState_father().getState_status().equals("initial")){
32         if(!Check_glue(temp_tra).isEmpty()){
33             traceback(Check_glue(tra));
34         }else{
35             temp_same_state.add(tra);
36             if(temp_same_state.size()>0){
37                 if(Check_glue(temp_same_state).isEmpty()){
38                     traceback(Check_glue(temp_same_state));
39                 }else{
40                     traceback(temp_same_state);
41                 }
42             }
43         }
44     }
```
4.4.2 Security Objectives over Test Generation with Insecure Medium

As we mentioned in the last section, the model of $S^M$ should be specifically before using the SO to guide the test generation. In the method of “Test Generation with Insecure Medium”, the considered attacks may be the transitions sent from the inside intruders. So the considered SO may contains the transitions of the possible intruders.
As we described in Chapter 3, the proposed SG-IOLTS models the intruders as transition systems and uses them as the medium of the system. It contains the required transitions of SO, and can be used as $S^M$. The procedures of the “SO guided test generation over insecure medium protocol testing” are presented in Figure 4.7.

Figure 4.7: SO guided: test generation with insecure medium

We use the same data structure presented in Figure 4.6, and the algorithm used is almost similar as the algorithm presented in Listing 4.1. The differences are the input transitions should be changed to SG-IOLTS. And in the function of $\text{Check\_glue}()$, because the \langle stimulus, response \rangle pair may contain multiple transitions when considering the intruder machine (for example, the $?m$ of intruder machine can correspond to all output labels of the system), a loop should be added in $\text{Check\_glue}()$ to record all possible transitions. Listing 4.2 presents the changes of the algorithm.

```
Inputs: Transitions of SG-IOLTS: ArrayList<transition> tra_sp;
Security Objective: ArrayList<transition> so_lab;
Int intruder_id;\n
Outputs: ArrayList<ArrayList> trace_list;

public transition Check_glue(ArrayList<transition> trace){
    For(transition t:trace){
        If (t.getStiLabel() in L_stiulate){
            for(each t:t.getStiLabel()){  
                L_sti.add(t);
            }  
        }  
    }
    return L_sti;
}
```

Listing 4.2: Pseudo Code of Algorithm for Insecure Medium Test Generation
4.5 Case study of RADIUS authentication protocol

4.5.1 Examples of SO over Test Generation with Secure Medium

In the “Test Generation with Secure Medium”, we assume the system ignores the threatening requests, and by the work we list in Chapter 2, the $S^+_glu$ of RADIUS can be obtained and presented as Fig. 2.14. Then by considering the security requirements (Security Objectives), the robustness test cases are calculated through our algorithm. The test cases should be generated with the Security Objectives which are normally defined by the system managers. And according to the required security objectives, the corresponding test cases can be generated.

Example 4.3:
We give a Security Objective where an already authenticated user attempts to send an undesired request (malicious or not) which is a threat that could disrupt the service in the RADIUS System:

Security Objective statement. The server can deal with an undesired request (?XXX_n) which is sent by an already known client; and then the sever send back a message of (!Ac_accept_n) to the NAS server.

The already known client means this request message is received after the NAS server received the (?Known_id) message. Which means only the client knowledge by the NAS server can be accepted by the RADIUS server, and all the threatening requests occurred during this process can be dealt with by the RADIUS server. This Security Objective can be presented as in Figure 4.8. With this Security Objective, the corresponding traces can be calculated by our tools, and the results are listed in the Listing 4.2. The results of this example contain 5 traces: trace 1 represents a normal conformance test trace; and from trace 2 to trace 5, the situations of the invalid inputs are considered in different states of the system.

| Trace 1: S1_0, ?Ac_req, S1_1, ?Known_id, S1_2, !Ac_req_n, S2_0, ?Ac_req_n, S2_1, !TAU, S2_2, !TAU, S2_3, !Ac_accept_n. |
| Trace 2: S1_0, ?Ac_req, S1_1, ?XXX_n, GIB_S1_1, !mu, S1_2, !Ac_req_n, S2_0, ?Ac_req_n, S2_1, !TAU, S2_2, !TAU, S2_3, !Ac_accept_n. |
| Trace 3: S1_0, ?Ac_req, S1_1, ?Known_id, S1_2, ?XXX_n, GIB_S1_2, !mu, S2_0, !Ac_req_n, S2_1, !TAU, S2_2, !TAU, S2_3, !Ac_accept_n. |
| Trace 4: S1_0, ?Ac_req, S1_1, ?Known_id, S1_2, !Ac_req_n, S2_0, ?Ac_req_n, S2_1, !TAU, S2_2, !TAU, S2_3, !Ac_accept_n. |
| Trace 5: S1_0, ?Ac_req, S1_1, ?Known_id, S1_2, !Ac_req_n, S2_0, ?Ac_req_n, S2_1, !TAU, S2_2, !TAU, S2_3, !Ac_accept_n, GIB_S2_2, !mu, S2_2, !TAU, S2_3, !Ac_accept_n. |
Example 4.4.

We give another Security Objective for the RADIUS System, with a new client which was not authenticated before, and tries to send an undesired request to the system:

**Security Objective statement.** The server can deal with unconsidered request (?XXX_n) which is sent by an new client (with ?new_id); and then the sever send back a message of (!Inq_Auth_n) to the NAS server to ask for the authorization.

This security objective wants to make sure the implementation is anti-threatening requests when a new client connecting to the server. It can be presented in Figure 18, and the selected traces after our tools are listed in Listing 4.3. In this example, according to the security objective, the results contain 3 different test traces: trace 1 is also a conformance test case, aiming to check the function conformance of the implementations by considering the security objective. The trace 2 and trace 3 are the traces of the invalid inputs at different states.
4.5.2 Examples of SO over Test Generation with Insecure Medium

In the “Test Generation with Insecure Medium” method, the intruders are considered and modeled as medium to connect the multiple components of the system. Through the work of Chapter 3, the $SG-IOLTS$ of RADIUS can be presented as Fig. 4.10. Then similar as the process we done before: by considering the Security Objectives, the test cases are selected through the algorithm.

**Example 4.5:**

**SO statement:** the NAS receive request accept message ($?Ac$_$_accept_n$) should be after it receive the access request message ($?Ac$_$_req$).

This SO can be presented as the transition machine in Figure 4.11. After using this SO, 87 test cases are selected. We present the first 10 in the Listing 4.3. The reader can find the whole results in Appendix.

```
Trace 103: t2, ?Ac$_req$, !Ac$_req$, !m$_o$, ?att$_2$, ?m, !Ac$_accept$, ?Ac$_accept_n$, !Ac$_req_n$, ?Ac$_req_1$, ?a$_a$, ?att$_1$.
Trace 104: t2, ?Ac$_req$, !Ac$_req$, !m$_o$, ?att$_2$, ?m, !Ac$_accept$, ?Ac$_accept_n$, !Ac$_req_n$, ?Ac$_req_1$, !m, ?m, !Ac$_challenge_n$, ?Ac$_req_n$.$l$, !a$_a$, ?att$_1$.
Trace 105: t2, ?Ac$_req$, !Ac$_req$, !m$_o$, ?att$_2$, ?m, !Ac$_accept$, ?Ac$_accept_n$, !Ac$_req_n$, ?Ac$_req_1$, !m, ?m, !Ac$_challenge_n$, ?Ac$_req_n$.$l$, !m, ?m, !Ac$_req_n$, ?Ac$_req_1$, ?a$_a$, ?att$_1$.
Trace 106: t2, ?Ac$_req$, !Ac$_req$, !m$_o$, ?att$_2$, ?m, !Ac$_accept$, ?Ac$_accept_n$, !Ac$_req_n$, ?Ac$_req_1$, !m, ?m, !Ac$_challenge_n$, ?Ac$_req_n$.$l$, !m, ?m, !Ac$_req_n$, ?Ac$_req_1$, ?a$_a$.
Figure 4.10: RADIUS components and intruders

Figure 4.11: Example 4.5

4.5.3 Assess the security of the networked implementations

After the generation of the test cases, we need to use these test cases to verify the implementations. The implementations are tested by checking the outputs with the outputs of the test suites. If the outputs of the implementations are the same as the outputs of the test suites, the implementations are robust. The results in Listing 4.2 are the selected traces by the Security Objective presented in Example 5. Those traces are calculated by considering the invalid inputs which have occurred in different states. In this way, when executing the test cases, the tester can observe which state the system is not robust. For example, the second trace of example 5: \{S1_0, ?Ac_{req}, S1_1, ?Invalid\_Inputs, GIB, S1_1, !mu, S1_1, ?Known\_id, S1_2, !Ac_{req,n}, S2_0, ?Ac_{req,n}, S2_1, TAU, S2_2, TAU, S2_3, !Ac_{accept,n}\}, can be used to generate a test case to identify the robustness at state S1_1 (the second state in component 1). When the tester runs this test on the implemented system, and observe that the system is blocked, re-started, or even crashed down, then we know that this system is not robust, and the problem will happen in the state of S1_1. After executing all the test cases of the selected traces, if the tested system can work well, then we say that this system is robust according to our security objective, which means the system can be secure when receiving the Threatening Requests Attacks.

4.5.4 Review of this technique

The usage of Security Objective reduces the considered states and transitions during the test generation. With the consideration of security objective, people only need to calculate the transitions which are related to the SO for each process of the on-the-fly testing. Comparing with the results of RADIUS protocol which we calculated in the last chapter, the computation time by using SO is clearly reduced (See Table 4.1). The average computation time by using SO is 1 ms, but without SO, it is 360 ms. The usage of SO also make each test execution process with a clearly security meanings: each set of test cases corresponds to one solid security purpose.
4.6 Conclusion

In this chapter, by proposing the Security Objective, we generate the test cases on-the-fly to verify the security of protocol implementations. This method can evade the problem of state explosion of the method proposed in Chapter 3. The proposed Security Objective is used to support and assist the security engineer to automatically calculate the test cases under an acceptable complexity. The SOs are defined more dependent on the experience and knowledge of the tester. Those experiences and knowledge may come from the happened security attacks, company’s security policies or even the technique reports of the security analysis organizations.

We can use the SOs to evade the problem of testing explosion, but the limitation of the approach is that we have not a firmed method to generate them. This limitation of the approach is also linked (intrinsic) to automatic test generation. In general to compute the exhaustive possible tests cases would encounter the combinatorial explosion scalability problem, which is a known open question. To address this, one needs to investigate some kinds of heuristics to find a tradeoff to an acceptable level of tests.
Chapter 5

Analysis on Intrusion Detection System

5.1 Problems and Assumption

An intrusion detection system (IDS) is a device system or software applications, which can detect the malicious activities or policy violations by monitoring the network traffics or system actives [11]. As mentioned in [72], an IDS can be generally divided into four major components: Event Monitor, Event Database, Event Analyzer, and the Response Unit. The Event Monitor is responsible for detecting the system or environment actives and converts them as some specific formats and store them in the Event Database. The Event Analyzer retrieves the modeled actives from the Event Database and analyses them in order to detect the intrusions. Once the unusual actives are detected, the Response Unit produces reports to a management station to warn a risk. A general IDS architecture based on the considered four types of functions is presented in Fig. 5.1.

IDS focus on detecting and preventing the intrusive activities, which were not detected by conventional system security mechanisms. For some inherited systems, because of some history or economic reasons, some powerful security mechanisms are hard to be deployed. However, the IDS can be used to solve this problem, because it need nothing to change to the target system.

An IDS may be either host or network-based [36]. A host based IDS analyzes events mainly related to OS information. While, a network-based IDS analyzes network related events, such as traffic volume, IP addresses, service ports, etc. Meanwhile, according
to the way of detecting the intrusion, two main categories of IDS are usually discussed: Misuse IDS and Anomaly IDS. The former uses the traces or templates of the known attacks, while the latter builds profiles of non-anomalous behaviors of computer system’s active subjects. For example, IDIOT [52] and STAT [43], use patterns of well-known attacks or weak spots in the system to match and identify known intrusions. The main advantage of misuse IDS is that it can be accurately and efficiently detect instances of known attacks. The principal disadvantage is that it lacks the ability to detect the truly innovative attacks. On the other hand, Anomaly IDS [56] does not require prior knowledge of intrusion and can thus detect new intrusions. But it may not be able to describe what the attack is and may have a high false positive rate.

In a NIDS, the functions of an IDS are comparable as an antivirus software to system files: it inspects the contents of exchanging messages to observe the possible attacks and decide whether and when to fire the fault-alarms [36]. NIDS uses several monitors to detect the network traffics, and uses some predefined “signatures” or “patterns” on those detected messages to identify the possible malicious attacks messages. For some misuse-based NIDSes, the “signatures” are usually defined by some known attacks. They may be a name (in characters) within the body of the attack code, targeted resources during the attack or the way these resources are targeted (attack pattern). For some the anomaly-based NIDS (A-NIDS), the non-anomalous profile (normally “patterns”), which identifies the normal behaviors of the system is modeled. The A-NIDS will raise an alarm once the behavior of the network does not match with its normal behavior model [47]. But, in this case, some unusual behaviors which are not caused by attacks (i.e., behaviors from the user’s bad operations or bad network conditions) will also trigger false alarms, and make the high False Positive (FP) alarms rate become the main problem of anomaly IDS.
Meanwhile, in some cases of attack (for example, replay-attack), the attacker can obtain the legitimate signatures from the past messages and use them to construct new attack messages. Those attacks are usually difficult to be detected by most of the anomaly IDses [71].

In our work, we focus on the problems of Anomaly Network Intrusion Detection System, we noticed those mentioned problems and propose a transition system based analysis method to increase the abilities of A-NIDS.

In our work, we assume the messages detected from different IDS monitors happened with sequence, and by using the transition model to describe those detected messages and bt comparing them with the non-anomalous profile, the abnormal behaviors can be detected. We believe this approach can help the A-NIDS to detect those missed intrusions such as replay-attacks and because the detected intrusion messages are going to be identified with suitable attack types, the FP alarm rate will be reduced significantly.

### 5.2 An Automata-based A-NIDS Solution

Intrusion Detection System usually uses two sources of data: network traffic data and audit trail data (audit data) to capture the system activities and to analyze the possible attacks [82]. In this work, we focus on the security of network protocol systems, where the data source we concerned are network traffics. The finite automata model is one of the major method of anomaly detection techniques [63,71], which considers the network system with states and transitions: a state represents the current status of the target system and the transitions represent the active actions/messages to different states. The current state changes only when it receives the corresponding transitions. For example, in a network system, the network traffics are usually modeled as transitions.

#### 5.2.1 The target system and IDS architecture

The considered target system, which is supervised under an IDS is a networked system. In such system, the different network devices may connect to each other through lines (wired) or radios (wireless). In order to detected the required messages traffics, we assume the IDS monitors are deployed with an ability to detected the network traffics of each devices in this network. For a general situation, we assume the target system use
IPv4 protocol to organize the network layer data transition. Then the detected traffic messages can be parsed and the required information can be obtained.

The IDS architecture over the target network system may be presented as Figure 5.2. As we presented in the figure, the monitors collect the sent and received IP traffics of each components and send them to IDS analyse engine to detect the possible malicious attacks. Notice the possible intruders should be also under monitoring by the IDS. The actions of the intruder are also recorded and analyzed by the IDS.

5.2.2 Intrusion Detection Approach

Our intrusion detection approach can be described as the following three steps which are also depicted as Figure 5.3:

- Traffic detection. The powerful IDS monitors detect the network traffics; then by checking the session and the timestamp information, those messages can be classified into different message sequences. Meanwhile, the type of protocol used in these messages can also be identified.

- Message classification and Profile selection. According to the type of used protocol, those detected messages are rewritten as the format of the transitions model. Meanwhile, the corresponding profile, which is modeled from the protocol specification are selected.
Verification. Those modeled transition sequences are verified with the proposed profiles to detect the abnormal.

### 5.2.3 Traffic Detection

According to the IDS architecture presented in the last section, the IDS monitors are in charge of the traffic detections. The monitors, which can be softwares or hardwares, obtain the received and sent IP traffics of the corresponding devices, and record them into digital files. Those files then are sent to the IDS Event Analyser. According to the time sequence of those messages, a combination IDS detection data file can be obtained, where the detected messages are listed with orders. Figure 5.4 demonstrates an example of the combination IDS detection data.

### 5.2.4 Message classification and Profile selection

#### 5.2.4.1 Message classification

Through the detected message, many information related to the target system can be deduced. For example, the information such as session id, time stamps, protocol type, cryptograph methods etc. can be detected. And through the information of session id
(sid) and time stamps (ts), the network traffics can be classified into message sequences. A normal message sequence usually contains messages exchanging during one session, but if the protocol contains multiple components, different session id may be found in one sequence. We assume that the network connections from different components happen sequently. Then within one selected window size N, the messages belong to one sequence can be detected.

Continuing the Example 1: the IDS detects the network traffic and get some data as Figure 5.4. Where, C1, C2 ... represent different initiator; S1, S2 represent different responder. !M_{ask,c1} represent the “ask message” sent from C1, !M_{rpl,s1} represents the “rpl messages” sent from the responder S1. !M_{cfm,c1} represent the “cfm message” sent from C1. The “xxxx” represents no message happen. As the use of the transition system, “?” represents the input label and “!” represents the output label. The square represents the window, which is used to select the message sequences. In this example, during the selected window, message outputs of components are {xxxx, !M_{ask,c1}, ?M_{ask,c1}, !M_{ask,c2}, ?M_{ask,c2}, !M_{rpl,s1}, ?M_{rpl,s1}, !M_{rpl,c2}, ?M_{rpl,c2}, !M_{cfm,c1}, ?M_{cfm,c1}, !M_{cfm,c2}, ?M_{cfm,c2}, xxxx, xxxx, !M_{ask,c3}, ?M_{ask,c3}, !M_{rpl,s2}, ?M_{rpl,s2}, !M_{cfm,c3}, ?M_{cfm,c3}, xxxx}. Where, {!M_{ask,c1}, ?M_{ask,c1}, !M_{rpl,c2}, ?M_{rpl,c2}, !M_{cfm,c1}} and {!M_{ask,c3}, ?M_{ask,c3}, !M_{rpl,s2}, ?M_{rpl,s2}, !M_{cfm,c3}, ?M_{cfm,c3}} have different session id (sid). But the messages of the second session happened between the message of the first session, so {!M_{ask,c1}, ?M_{ask,c1}, !M_{ask,c2}, ?M_{ask,c2}, !M_{rpl,s1}, ?M_{rpl,s1}, !M_{rpl,c2}, ?M_{rpl,c2}, !M_{cfm,c1}, ?M_{cfm,c1}, !M_{cfm,c2}, ?M_{cfm,c2}} belongs to one sequence, and {!M_{ask,c3}, ?M_{ask,c3}, !M_{rpl,s2}, ?M_{rpl,s2}, !M_{cfm,c3}, ?M_{cfm,c3}} belongs to another sequence.

5.2.4.2 Profile selection

Through the message obtained, the type of used network protocol can be identified. Those network protocols are usually broadcast standards, and make the transition system machine possibly to be built. The transition system can usually present the protocol with reachable graph, which makes the transition sequence verification becomes how to find the walk in the graph.

Continue the example of NSPK, by using the Glued-IOLTS, the protocol specification can be presented as Figure 5.5.
Chapter 5. Analysis on Intrusion Detection System

**Figure 5.4**: Message Concatenation

**Figure 5.5**: Glued IOLTS of NSPK
5.2.5 Verification

Verification of the detected transition sequences is to find the *walk* in the reachable graph of the Glued-IOLTS. During the verification, we may need to adapt some past transitions into the detected sequence to complete the walk in Glued-IOLTS. After doing this, if the transition sequence can find the corresponding walk, it means the detected messages traffics are a normal messages. Otherwise, message traffic contains some possible attacks to the system.

During the process of verification, according to the transition where the walk cannot continue, and the past transition in the walk, we can deduce some attack types in this sequence.

- Jam-attack, which means the attacker blocks the transmitting transitions. An intruder can simply take the transmitting message away from the medium to attack the system. In this case, the walk of the detected sequence can be found in the Glued-IOLTS, but the end state of this walk will not be the end state of the transition machine. It is a partial sequence of Glued-IOLTS.

- Fake-attack, which means the attacker modifies the transmitting transitions and sends it to the receiver. This kind of attack may contain many strategies of modification, but here, we only consider the modifications which occur in the changes of the modeled transition (the model transition label will change). If a sequence contains the fake attack, the verification cannot find the corresponding walk in the Glued-IOLTS. But the fake actions may happen at the transition which makes the walk stopped, or may happen before.

- Replay-attack, which means the attacker uses the happened transition to attack the system. This kind of transition to the Glued-IOLTS is inopportune inputs. The walk will stop at the inopportune transition, and also this transition can be found in the past transitions.

In order to detect those attacks automatically, we propose an algorithm in the Table 5.1. The inputs to the algorithm are one of the modeled label sequences detected by IDS($l_{ids}$) and the glued transition system ($T_{sys}$). First of all, the algorithm searches for the transitions in $T_{sys}$, which have the same label as the first label of $l_{ids}$, and record the results in a transition list of $t_{temp}$. Then for each transition $t_i$ in $t_{temp}$, the algorithm compares the label of the next transition of $t_i$ and the next label of $l_{ids}$. Removes $t_i$ from...
If the transition with the same label can be found, record it in \( t_{\text{temp}} \). Backup this \( t_{\text{temp}} \) as \( t_{\text{temp bac}} \). Then, repeats the process until the end of \( l_{\text{ids}} \) or the \( t_{\text{temp}} \) is empty. During the repeating, the algorithm records the past labels of \( l_{\text{ids}} \) in \( l_{\text{pass}} \). When the algorithm stop, If the algorithm found all labels of \( l_{\text{ids}} \) in \( T_{\text{sys}} \), we go to check the final state of the walk in \( T_{\text{sys}} \). If the final state is an “end” state, the \( l_{\text{ids}} \) is secure. Otherwise, this \( l_{\text{ids}} \) contains Jam-attack. If the algorithm stops when comparing \( l_{n} \) of \( l_{\text{ids}} \) and result the \( t_{\text{temp}} \) is empty, then for each transition \( t_{j} \) in \( t_{\text{temp bac}} \), compares the label of the next transition of \( t_{j} \) and the passed label \( l_{i} \) in \( l_{\text{pass}} \). If \( l_{i} \) is same as the label of the next transition of \( t_{j} \), record the next transition of \( t_{j} \) in \( t_{\text{temp}} \), backup \( t_{\text{temp}} \) to \( t_{\text{temp bac}} \), record \( l_{i} \) in \( l_{\text{pass}} \). Then, compare the \( l_{n} \) with the next transitions of \( t_{\text{temp}} \). If \( l_{n} \) can be found in the next transition, record \( l_{n} \) in the \( l_{\text{pass}} \) and move to the next label of \( l_{\text{ids}} \). Otherwise, reconsider the passed labels until the end of \( l_{\text{pass}} \). If after considering the labels of \( l_{\text{pass}} \), the \( l_{n} \) still cannot be found in the transition sequence. Then the \( l_{\text{ids}} \) must contain some modifications. The algorithm returns “fake-attack”. Meanwhile, if \( l_{\text{pass}} \) contains \( l_{n} \), then the \( l_{\text{ids}} \) contains a replay, the algorithm returns “replay-attack”.

Continue the example of NSL, the first label sequence will return a ”replay-attack” result. When executing this sequence in the algorithm, the algorithm will stop when it reaches label \{!M\_ask,c2\}, and because this label (!ask). The second sequence will result a “secure” as the result.

5.3 An Experiment over RADIUS system

According to our intrusion detection approach and the algorithm, we developed a prototype of the A-NIDS and deploy it over a RADIUS system. We use the Wireshark as the IDS monitor and install it on every devices of this experiment. Wireshark is a free network protocol analyzer and can detect and record the network traffics. The target RADIUS system contains one server and several clients (see Figure 5.6). In this system, one router (R1) and one server (S1) are used to serve multiple clients. There are clients connected through wireless connection (wc1, ..., wc4), and also some clients connected through a wired connection (lc1, ..., lc3).

The RADIUS protocol is an application layer protocol, which transmits data through UDP traffics. It uses the port number 1812 or 1645 to communicate. So when the monitor (Wireshark) obtains the IP traffics, by checking the port number of the UDP
Table 5.1: Algorithm to verify transition sequences in Transition System

<table>
<thead>
<tr>
<th>Input:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label Array <code>l_{ids}</code>: //one transition sequence detected by IDS.</td>
</tr>
<tr>
<td>Transition Array <code>T_{sys}</code>: //the transition system of the protocol.</td>
</tr>
<tr>
<td>Output:</td>
</tr>
<tr>
<td>secure, fake-attack, jam-attack, replay-attack</td>
</tr>
<tr>
<td>Begin</td>
</tr>
<tr>
<td>Transition Array <code>t_{temp}</code>;</td>
</tr>
<tr>
<td>Transition Array <code>t_{next}</code>;</td>
</tr>
<tr>
<td>Label Array <code>l_{pass}</code>;</td>
</tr>
<tr>
<td>String result;</td>
</tr>
<tr>
<td>int flag=0; Search <code>l_{ids}[0]</code> in <code>T_{sys}</code> and record the results in <code>t_{temp}</code>;</td>
</tr>
<tr>
<td>For each transition <code>t_{i}</code> in <code>t_{temp}</code>{</td>
</tr>
<tr>
<td>record the next transition of <code>t_{i}</code> in <code>t_{next}</code>;</td>
</tr>
<tr>
<td>record <code>l_{ids}[0]</code> in <code>l_{pass}</code>;</td>
</tr>
<tr>
<td>} For (int <code>i</code>=1; <code>i</code>&lt; <code>l_{ids}</code>.length; <code>i</code>++){</td>
</tr>
<tr>
<td>flag++;</td>
</tr>
<tr>
<td>If (<code>t_{temp}</code>isnotempty){</td>
</tr>
<tr>
<td>record the next transition of <code>t_{i}</code> in <code>t_{next}</code>;</td>
</tr>
<tr>
<td><code>t_{temp}</code>bac=<code>t_{temp}</code>;</td>
</tr>
<tr>
<td>remove <code>t_{i}</code> from <code>t_{temp}</code>;</td>
</tr>
<tr>
<td>Search <code>l_{ids}[i]</code> in <code>t_{next}</code> and record the results in <code>t_{temp}</code>;</td>
</tr>
<tr>
<td>record <code>l_{ids}[i]</code> in <code>l_{pass}</code>;</td>
</tr>
<tr>
<td>For each <code>l_{k}</code> in <code>l_{pass}</code> {</td>
</tr>
<tr>
<td>Search <code>l_{k}</code> in <code>t_{next}</code> and record the results in <code>t_{temp}</code>;</td>
</tr>
<tr>
<td>If (<code>t_{temp}</code>isnotempty){</td>
</tr>
<tr>
<td>continue;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>If (<code>l_{ids}[i]</code> in <code>l_{pass}</code>){</td>
</tr>
<tr>
<td>result=&quot;replay-attack&quot;;</td>
</tr>
<tr>
<td>return result;</td>
</tr>
<tr>
<td>} else{</td>
</tr>
<tr>
<td>result=&quot;fake-attack&quot;;</td>
</tr>
<tr>
<td>return result;</td>
</tr>
<tr>
<td>} }</td>
</tr>
<tr>
<td>If(flag==<code>l_{ids}</code>.length){</td>
</tr>
<tr>
<td>If(<code>t_{i}.nextState().getStatus.equals(&quot;end&quot;)</code>){</td>
</tr>
<tr>
<td>result=&quot;secure&quot;;</td>
</tr>
<tr>
<td>return result;</td>
</tr>
<tr>
<td>} else{</td>
</tr>
<tr>
<td>result=&quot;jam-attack&quot;;</td>
</tr>
<tr>
<td>return result;</td>
</tr>
<tr>
<td>result=&quot;secure&quot;;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>End</td>
</tr>
</tbody>
</table>

messages, the RADIUS messages can be distinguished. For the simplicity of the experiment, we make the target system only execute the RADIUS applications: we install the FreeRADIUS server on the server side (S1), and the RADIUS client (for example, NTRadPing) on the client side (wc1, lc1 ... ) to construct an experiment environment. The clients send RADIUS requests messages to the RADIUS server through the router, then RADIUS server verifies the authentication and responses the requests [68].

The IDS monitors record the sent/received data of each components, analyze them and restore them into the files like Table 5.2. Such files of different monitors then are sent
to the IDS Event Analyser to construct the message sequences.

The IDS Event Analyser in this experiment is an application we developed with JAVA. It can concatenate the IDS detected messages as sequences, models those message sequences and implements our algorithm to detect the possible intrusion (see Figure 5.9). As the network traffics happen sequently, the detected traffic data from different monitors may happened as Figure 5.7. For example, we randomly choose a window size like
the square in the figure and found three modeled message sequences: \{xxxx, !Ac_req_w1, ?Ac_req_w1, !Ac_req_w1_n, ?Ac_req_n_w1, !Ac_accept_n_w1, ?Ac_accept_n_w1, !Ac_accept_w1, ?Ac_accept_w1, xxxx\}, \{xxxx, !Ac_req_w2, ?Ac_req_w2, !Ac_req_w2, ?Ac_req_w2, !Ac_req_w2, ?Ac_req_n_w2, !Ac_accept_n_w2, !Ac_accept_w2, ?Ac_accept_w2, xxxx\}, and \{xxxx, !Ac_req_l1\}. In this example, the first transition sequence is a normal connection sent from the client wc1 to the server. The second sequence is a connection from wc2 to wc3 (this is may be cause the wc3 declares himself as a NAS server), then wc3 forwards the request of wc2 to the real server. This sequence contains a replay-attack. And the third sequence is not a complete sequence. If the IDS only verifies the signature of the message, it will not find the problem of the second transition sequence. In our IDS approach, we only need to search this transition trace in the corresponding reachable graph, which is a non-anomalous profile of the target system.

According to the RADIUS protocol specification, the non-anomalous profile can be presented as Figure 5.8. With the algorithm we proposed, the program can verify the detected traffics automatically, see Figure A.13. For detail information of the application, please go to AppendixA.

5.4 Conclusion

In this article, we investigated the problems of anomaly network IDS methods and proposed an automata-based approach to improve the detection abilities of A-NIDS.
The proposed approach assumes the IDS can distinguish the protocol type used in the network traffic, then use the standard protocol specifications of the corresponding protocol to generate the non-anomalous profile. Meanwhile, the detected messages from IDS monitors are modeled as sequence of transitions using the same format as the proposed model. Finally, by verifying the detected sequences with the trace of specification model, we can judge whether the detected messages sequence contains intrusion or not.

We proposed the algorithm to distinguish the possible attack type automatically, which can significantly reduce the FP alarm rate. We also develop a prototype of this A-NIDS and experiment it over a RADIUS system.

Figure 5.8: Glued IOLTS for RADIUS

Figure 5.9: GUI of IDS
The principle idea of the proposed method is on comparing the detected sequences with the transition model. It is similar as the “language recognition” problem, which needs to find the corresponding walk in the graph of the transition model. But our method is specifically designed for the Glued-IOLTS model, which attaches the detected message sequences with some partial sequences to fit the format of Glued-IOLTS model.
Chapter 6

Conclusions and Future Works

The work of this thesis is on verifying the security of protocol implementations. In this thesis, we use protocol security testing methods, which are based on the classic Protocol Testing, to guarantee the security of protocol implementations. The protocol implementations are usually presented as the components of the networked and current system, which is connected to be a system through a common medium. In our work, we first assume this universal medium to be secure and use it to glue multiple components to be one transition system. Then we use the proposed algorithms automatically generate the required test sequences to verify the security. Later, the assumption of secure medium was removed and the potential intruders are considered into the medium. Meanwhile, the important security features of the message contents, such as nonce and encrypted key, etc. are also recalled and are modeled. In this thesis, we also discussed the Security Objectives, which was proposed to reduce the complexities of security test generation, and the use of a transition system on an Intrusion Detection System.

Summary

In Chapter 2, we discussed the security testing based on a secure medium. In this chapter, the security defects caused by misuse or type-flaw errors, such as DoS attack and Threatening request attack are studied. Those kinds of attacks aim at exhausting the resources or blocking the communication medium, and can be avoided by improving the robustness of the protocol implementation. We extended the classic IOLTS model to our “Extended Glued-IOLTS” to generate robustness test cases for protocol implementations. This extended model, considered the network and concurrent components
as one black-box, and trusted all components inside of the box. This model first described the protocol components with IOLTS, and separated the states into two levels: the higher level and lower level. The higher level states performed as normal state of IOLTS, but the lower states were in charge of the connections between components. In this way, the different IOLTS components can be glued together. Then we added the “Refusal graph” to generate the misuse cases (called inopportune inputs) and “Security Policy” states to model the type-flaw errors (called as invalid inputs) to the glued model to finally produce an extended model. We proposed algorithms and program tool to use this model and generated test cases automatically.

In Chapter 3, we proposed a method to reduce the considered transitions during each test generation. Because of the complexity of protocol, added to the multiplicity of possible malicious inputs, the combination of scenarios to be computed may increase at an explosive speed, which makes the system memory or computation time becomes the main problem of test generation. To address this, we gave the concept of Security Objectives and used it through Protocol Security Testing to generate the test cases on-the-fly.

In Chapter 4, by considering the power potential intruders and the limitations of finite transition models, we proposed the SG-IOLTS model to model the protocol implementations. This new model, which is an extension of IOLTS, defines the security features into system transitions and proposes a general intruder machine inside of the shared medium. The test cases generated from this model contain the actions of the probable intruders, which are possible attack scenarios the protocol implementations.

In Chapter 5, we noticed the anomaly IDS have difficulties on detecting the replay-attacks and high FP alarm rate. We propose a transition model based intrusion detection approach to solve this problem. The proposed approach concatenated detected messages as message sequences, and uses the Glued-IOLTS model, which is presented in Chapter 2, to define the non-anomaly profiles. Then those concatenated sequences were verified in the transition system to find the abnormal. If an intrusion is detected, the method can also identify the attack types. We proposed an algorithm to make the verification and distinguish automatically.

Although the presented methods can verify the security of protocol implementations and increase the abilities of the anomaly IDS, they still have some limitations. Just as we mentioned in each chapter, these methods are defined based on some specific assumptions, which restrict the use of the verification methods. First of all, in the work
of this thesis, we believe the implementations have the general protocol specifications and can be presented by transition models. However, in some real cases, this assumption may be hard to achieve. Secondly, the proposed verification methods ignore the use of time parameters of security protocol, which makes the time based protocols cannot be verified by our methods. Thirdly, the proposed insecure medium simply modeled three basic actions of the intruder as attack messages, but in the real case, the attacks from the intruder may concatenate with those basic actions or partial actions. The actual attack cases may be more complex. And the fourth, the proposed method to the anomaly IDS assumes the detected messages can be concatenated as sequences, which are hard to be achieved in most of the parallel processing network environment.

**Future Research Directions**

Along with the development and the use of computer technologys, the topic of security verification is always growing and interesting. In the future, we are going to focus on the following security problems:

**Optimize Intruder Model**

The intruder model we used in this thesis is based on the Dolev-Yao assumption, which has been proposed more than 10 years. Despite the fact that it is the foundation of modern protocol security analysis, the security concerns seem not enough for some kinds of security protocol. Recent security protocols usually become more complex and do not simply use the long term key (the utilized Public and Private key pair may change regularly) to finish the cryptograph, which may increase the processes of intruder model. To enhance the intruder model will be one direction of our future work.

**Consideration of Time Parameter**

In our method, we neglect the time parameter of protocol. Nevertheless, the time parameters such as time-stamp are widely used and important security concerns of protocol and its implementations. We will introduce time into our method in the forthcoming work.
Security Testing for Embedded System

The embedded devices, such as mobile phones and tablets, usually work under a very complex environment. Comparing with the customary network terminals, those embedded devices are more easily be attacked. Security verification to those embedded devices is vital and important. Regardless of the fact that the embedded devices also need to implement the security protocols to keep them away from attacks, but because of some specific features of those embedded system, such as the limitation of battery and memories, the security testing of those systems is not straightforward. We may also take this topic as our future work.
Appendix A

Test Generation and Analysing Tools

Regarding the requirement of automatic test generation and some user experiences, we propose a GUI tools for the automata based protocol design, analysis and verification. The application is developed with JAVA languages.

A.1 Requirements

This application should contain the following functions:

- **Inputs:**
  - A GUI (graphic user interface) which should be easily used by the tester to input/design the protocols/SOs. The concepts used on this GUI should be closer to transition based models and protocol design. Which are state, node, transition, components, labels etc. The designed protocol should be presented as a reachable graph.
  - The results of the graph input should be possible to be saved, opened and modified. The save documents should be in the format of XML.

- **Test Generation:**
  - Test Generation of secure medium and G-IOLTS model. An implementation of the algorithm of Chapter 2.
Appendix A. Applications

Figure A.1: Use Case Diagram

- Test Generation of insecure medium and SG-IOLTS model. An implementation of the algorithm of Chapter 3.
- Test Generation over Security Objectives. An implementation of the algorithms of Chapter 4.

• Intrusion Detection System Analyser:
  - Input function of the documents detected by IDS monitors.
  - Analyze and model detected messages to decide the protocol type.
  - According to the detected protocol type to concatenate the messages as sequences.
  - Verify each of the sequences over the normal pattern.

• Output:
  - The generated test cases should be presented by GUI.
  - Test cases results can be reviewed step by step.
  - IDS verification results should be presented by GUI.
  - If IDS verification results are successful, the trace should be possible verified step by step.

The use case diagram is presented in Fig. A.1.
A.2 Application Design

According the requirement, we define the application into the following three levels:

- Automata transition description level, which means the objects in the models, such as “state”, “label”, “transition”, “higher-level state”, “lower-level state”, “initial state” etc. should be easily presented. Those automata objects are used in our test generation algorithms, so our Java application should respect those objects, and translate them as “class”.

- Graph presentation level, which is the graphic user interface (GUI). With the automata level classes, the transitions of the state machines can be described and our test generation algorithms can be achieved. But, this automata based application is not easy to use. The user need to design the protocol specification and input them to the application line by line. To simplify the operations, the GUI level is proposed.

- Storage level, which describes the functions of operating system files, such as the format of “save” document, how to read and presents those features, etc.

In order to describe the Automata transitions, define 4 classes: state, node, label and transition (see Fig. A.2) are defined in our application. The class “state” simply describes the automata transition state, which defines “name”, “status”, “higher_node” and “lower_node” as the properties. The class “label” describes the transition labels, which contains “label_name” as its properties. The class “node” defines the lower or higher interface of the state. The class “transition” defines the transitions of the automata. This class will use two instances of the class “node” and one instance of “label” to define an transition.

For the Graph presentation level, we have more classes and functions. The class “Draggablelabel” is an extension of JLabel, which is used to draw the state. The class “ConnectorContainter” is an extension of JLabel, which is used to draw the system components. The class “JConnector” connects two different states (instances of Draggablelabel), and use the class “label_text” to draw the labels of the transition. The class “grid_panel” is an extension of class “ConnectorContainter”, but propose a grid-drawn panel for the user to draw components and lowers-level transitions, insecure transitions between components. The class “label_text” is an extension of JLabel, which presents the name of the transition labels and with many functions to control the corresponding transition.
The class “componentMover” defines the objects can be moved by the mouse and the class “componentResizer” defines the resizable of the objects.

For the requirement of organizing and sharing the designed protocol specifications and security objectives, a class “outfunctions” are used to achieve those file operation functions. See Fig. A.3 to get the relations between those classes.
Appendix A. Applications

The “SGUI.java” contains the main function of the program, and it has about 3000 lines of java codes. Totally, in our application, we have about 6000 lines of Java codes.

A.3 Guides of how to use

A.3.1 Java Environment

In order to use this application, an Java runtime environment (JRE) or Java Development Kits (JDK) should be installed in you computer. It is very easy to instal Java, you can find many tutorials from the powerful internet. Generally speaking, you need to download Java SDK from the website of Oracle, then install it just by clicking next step.

A.3.2 Usages of the application

After you install Java on you computer, you can run the application by double click the application icon. Then a window like Fig. A.4 will appear. Generally speaking, this application contains 3 main functions. Each function corresponds to the contents of one chapter of this thesis (chapter 3, 5 and 6).

The users need to begin from creating a new project by click the “New Project” button or opening an existed project by click the “Open Project” button. Fig. A.5 shows the
Appendix A. Applications

Figure A.5: New Project

window of generating the new project. After the user defines the project name, file locations and the using model, a new project will be generated.

In the application, we applied each of our purposes on one tabbed panel. We defined those functions as follow:

- “Design Protocol”. Within this frame, the users can design the protocol specifications by simply drag and draw. After designing the protocol, the program can calculate the possible robustness test results. See Figure A.6. In this panel, the user can generate the component by click the button of “component”, and put it on the panel. A component may means one finite state machine of the concurrent system. Users can resize the range of the component by using mouse. The states and labels of the transition can be added inside of the components by clicking the corresponding buttons. Right click on the added state, users can modify the name of the state, delete this state or set the status of this state. Right click on the text of the label, users can modify the name of the label or delete this transition (see Fig A.7). After the transitions of the components are designed, people need to use the “Lower State” and “Lower Label” to connect different components (see Fig A.9). After the Glued-IOLTS model is designed, the user can calculate the corresponding robustness based security test cases by click the button “Run”, and the correspond results will appear in the under the graph. See Fig. A.8 as an example. Users can verify the transition exchanges step by step by clicking the button “simulation”. See Fig. A.10 as an example.

- “Design SO”. In this frame, users can design their security objectives. The operations are similar as design a protocol. After a SO is defined, users need to choose the corresponding protocol specification by click the button of “select protocol”. Then by click the button of “Run”, the program calculates the test results under this SO. See Figure A.11 as an example. Notice the end state of the SO should
The application saves the designed protocol and the security objectives as XML files (see Figure A.16 as an example), and organize the files into three different folds: Protocol, Security Objectives and IDS. The Protocol fold are used to store the files related

![GUI of Design Protocol](image-url)

**Figure A.6: GUI of Design Protocol**

named as “Acp” or “Ref”, which represents the traces are accepted or refused by the SO.

- “IDS”. In this frame, after reading the IDS.txt document, the program can model and concatenate the messages which are detected by IDS monitors to sequences, then by verifying the detected messages sequences with the proposed normal standards to identify the intrusions. See Figure A.12. The users need to click the button “select IDS” to load the IDS detected messages. According to the contents of “IDS.txt”, the program detects the type of protocols used and translate the IDS message in a format of “Msg_id:IDS_id !/? Protocol_type Ip_source Ip_dest Time Data”. Then these messages can be modeled with the our Guled-IOLTS patterns by click the button “model”. After this, the user need to choose the normal pattern by click the button “Normal Pattern”, and a new window like Fig. A.13 will appear. Then in this new window, by click the button “Verify”, the checking results of the detected messages are presented in the right-below of this window (see Fig. A.14). And the user can also see the simulation of the transitions exchanges by click the button “simulation” if the selected traces are successful(see Fig. A.15 ).
to “Protocol Design” functions, it will contain a “design.xml”, “xxx.pro” and “Testresult.txt”. The ‘design.xml’ and “Testresult.txt” are temporary file to calculates test cases. “xxx.pro” is the xml file to describe the designed protocol. The fold of Security Objectives is used to store the different security objectives. According to different protocol, the numbers of SO files may change. In the fold of IDS, one or multiple “ids.txt” may be found. And the normal patterns of the protocols are recommended stores in this fold.
A.4 Characteristics of this application

This application is developed to satisfy the requirements of our Glued-IOLTS model based test generation. The significant characteristic of this application is: it can describe the medium (secure or insecure) between concurrent components and use the transitions of medium to connect different components. Some characteristics of this application are shown in the Table A.1:
Appendix A. Applications

Figure A.11: GUI of Design Security Objective

Figure A.12: GUI of IDS

Table A.1: Characteristics of each class

<table>
<thead>
<tr>
<th>Class Name</th>
<th>No. public Methods</th>
<th>No. Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>14</td>
<td>80</td>
<td>Class of object state</td>
</tr>
<tr>
<td>node</td>
<td>4</td>
<td>20</td>
<td>Interface of lower(higher) interface of state</td>
</tr>
<tr>
<td>transition</td>
<td>14</td>
<td>50</td>
<td>Class of object transition</td>
</tr>
<tr>
<td>label</td>
<td>2</td>
<td>10</td>
<td>Class of object label</td>
</tr>
<tr>
<td>DraggableLabel</td>
<td>10</td>
<td>450</td>
<td>Class to present state as graph</td>
</tr>
<tr>
<td>ConnectorContainer</td>
<td>28</td>
<td>230</td>
<td>Class to present components as graph</td>
</tr>
<tr>
<td>grid_panel</td>
<td>3</td>
<td>66</td>
<td>A container to store “ConnectorContainer” objects and other objects</td>
</tr>
<tr>
<td>JConnector</td>
<td>17</td>
<td>300</td>
<td>Class to construct transitions</td>
</tr>
<tr>
<td>ConnectLine</td>
<td>13</td>
<td>350</td>
<td>Class to set up the type of connecting line</td>
</tr>
<tr>
<td>label_text</td>
<td>4</td>
<td>275</td>
<td>Class to present the labels context as graph</td>
</tr>
<tr>
<td>intruder</td>
<td>1</td>
<td>75</td>
<td>Class to construct the possible network intruder</td>
</tr>
<tr>
<td>ComponentMover</td>
<td>4</td>
<td>350</td>
<td>Class to make the object become movable</td>
</tr>
<tr>
<td>ComponentResizer</td>
<td>4</td>
<td>450</td>
<td>Class to make the object become resizable</td>
</tr>
</tbody>
</table>
Figure A.13: IDS Verification Panel

Figure A.14: IDS Verification Results

Figure A.15: IDS Simulation Results
Table A.2: Results of comparing

<table>
<thead>
<tr>
<th>Tool name</th>
<th>Web Support</th>
<th>Compare Model</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visual Paradigm</td>
<td><a href="http://www.visual-paradigm.com/product/vpuml/">http://www.visual-paradigm.com/product/vpuml/</a></td>
<td>UML state machine Model</td>
<td>1. A commercial UML design tool, but can’t describe multiple components within one reachable graph.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2. The generated state diagrams are hard to be exported as a XML file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3. No open source.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4. Don’t support extended parameters, such as guards and actions.</td>
</tr>
<tr>
<td>Calife</td>
<td>No support</td>
<td>Hybrid Automata Extended Timed Automata</td>
<td>1. Calife defines most of the elements of transition systems, and can present state diagrams as XML file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>But, Calife don’t take component as an element of the graph. Different components are represented as different xml files in Calife, which means, the concurrent system cannot be presented as one reachable graph.</td>
</tr>
<tr>
<td>IBM rational rose</td>
<td><a href="http://www-01.ibm.com/software">http://www-01.ibm.com/software</a> /products/en/ratiosoefami/</td>
<td>UML state Machine diagram</td>
<td>1. Rational rose is a UML design and model based code generator developed by IBM. Its state machine diagram can satisfy most parts of our requirement. The problem is also on the solution of concurrent components.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2. It is not open source software and very expensive.</td>
</tr>
<tr>
<td>Topcased</td>
<td><a href="http://www.topcased.org/">http://www.topcased.org/</a></td>
<td>UML State Machine Diagram</td>
<td>1. Topcased is an open source UML designer. Although it uses different concepts as our model (there are no concept of component, lower/higher state, etc.), but some concepts of this model are compatible with our requirements.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2. It is open source and xml based.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3. Don’t support guards and actions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2. Visio is not open source.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3. Don’t support extended state machine</td>
</tr>
<tr>
<td>Ours</td>
<td><a href="mailto:Yulongf@gmail.com">Yulongf@gmail.com</a></td>
<td>State-Machine Diagram</td>
<td>1. Our tool is developed based on our requirement. It is designed as the requirement of our glued model, and also compatible with other models.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Glued-IOLTS</td>
<td>2. We may develop an extension of Topcased for our tool.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Secure Glued-IOLTS</td>
<td></td>
</tr>
</tbody>
</table>
## Appendix B

### RADIUS Robustness Test Cases

<table>
<thead>
<tr>
<th>nas0</th>
<th>Req, nas1</th>
<th>? Known id, nas2</th>
<th>? Ac_req, R0, ? Ac_req, R1, tau, R2</th>
<th>! Ac_challenge_n, nas0</th>
<th>? Ac_req, nas1</th>
<th>? Known id, nas2</th>
<th>? Ac_req, nas4</th>
<th>? Ac_challenge_n, nas5, tau, nas3</th>
<th>! Inq_Auth, nas0</th>
<th>? xx, GIB, ! e, nas3</th>
</tr>
</thead>
<tbody>
<tr>
<td>nas0</td>
<td>Req, nas1</td>
<td>? Known id, nas2</td>
<td>? Ac_req, R0, ? Ac_req, R1, tau, R2</td>
<td>! Ac_challenge_n, nas0</td>
<td>? Ac_req, nas1</td>
<td>? Known id, nas2</td>
<td>? Ac_req, nas4</td>
<td>? Ac_challenge_n, nas5, tau, nas3</td>
<td>! Inq_Auth, nas0</td>
<td>? xx, GIB, ! e, nas3</td>
</tr>
<tr>
<td>nas0</td>
<td>Req, nas1</td>
<td>? Known id, nas2</td>
<td>? Ac_req, R0, ? Ac_req, R1, tau, R2</td>
<td>! Ac_challenge_n, nas0</td>
<td>? Ac_req, nas1</td>
<td>? Known id, nas2</td>
<td>? Ac_req, nas4</td>
<td>? Ac_challenge_n, nas5, tau, nas3</td>
<td>! Inq_Auth, nas0</td>
<td>? xx, GIB, ! e, nas3</td>
</tr>
<tr>
<td>nas0</td>
<td>Req, nas1</td>
<td>? Known id, nas2</td>
<td>? Ac_req, R0, ? Ac_req, R1, tau, R2</td>
<td>! Ac_challenge_n, nas0</td>
<td>? Ac_req, nas1</td>
<td>? Known id, nas2</td>
<td>? Ac_req, nas4</td>
<td>? Ac_challenge_n, nas5, tau, nas3</td>
<td>! Inq_Auth, nas0</td>
<td>? xx, GIB, ! e, nas3</td>
</tr>
<tr>
<td>nas0</td>
<td>Req, nas1</td>
<td>? Known id, nas2</td>
<td>? Ac_req, R0, ? Ac_req, R1, tau, R2</td>
<td>! Ac_challenge_n, nas0</td>
<td>? Ac_req, nas1</td>
<td>? Known id, nas2</td>
<td>? Ac_req, nas4</td>
<td>? Ac_challenge_n, nas5, tau, nas3</td>
<td>! Inq_Auth, nas0</td>
<td>? xx, GIB, ! e, nas3</td>
</tr>
<tr>
<td>nas0</td>
<td>Req, nas1</td>
<td>? Known id, nas2</td>
<td>? Ac_req, R0, ? Ac_req, R1, tau, R2</td>
<td>! Ac_challenge_n, nas0</td>
<td>? Ac_req, nas1</td>
<td>? Known id, nas2</td>
<td>? Ac_req, nas4</td>
<td>? Ac_challenge_n, nas5, tau, nas3</td>
<td>! Inq_Auth, nas0</td>
<td>? xx, GIB, ! e, nas3</td>
</tr>
<tr>
<td>nas0</td>
<td>Req, nas1</td>
<td>? Known id, nas2</td>
<td>? Ac_req, R0, ? Ac_req, R1, tau, R2</td>
<td>! Ac_challenge_n, nas0</td>
<td>? Ac_req, nas1</td>
<td>? Known id, nas2</td>
<td>? Ac_req, nas4</td>
<td>? Ac_challenge_n, nas5, tau, nas3</td>
<td>! Inq_Auth, nas0</td>
<td>? xx, GIB, ! e, nas3</td>
</tr>
<tr>
<td>nas0</td>
<td>Req, nas1</td>
<td>? Known id, nas2</td>
<td>? Ac_req, R0, ? Ac_req, R1, tau, R2</td>
<td>! Ac_challenge_n, nas0</td>
<td>? Ac_req, nas1</td>
<td>? Known id, nas2</td>
<td>? Ac_req, nas4</td>
<td>? Ac_challenge_n, nas5, tau, nas3</td>
<td>! Inq_Auth, nas0</td>
<td>? xx, GIB, ! e, nas3</td>
</tr>
<tr>
<td>nas0</td>
<td>Req, nas1</td>
<td>? Known id, nas2</td>
<td>? Ac_req, R0, ? Ac_req, R1, tau, R2</td>
<td>! Ac_challenge_n, nas0</td>
<td>? Ac_req, nas1</td>
<td>? Known id, nas2</td>
<td>? Ac_req, nas4</td>
<td>? Ac_challenge_n, nas5, tau, nas3</td>
<td>! Inq_Auth, nas0</td>
<td>? xx, GIB, ! e, nas3</td>
</tr>
<tr>
<td>nas0</td>
<td>Req, nas1</td>
<td>? Known id, nas2</td>
<td>? Ac_req, R0, ? Ac_req, R1, tau, R2</td>
<td>! Ac_challenge_n, nas0</td>
<td>? Ac_req, nas1</td>
<td>? Known id, nas2</td>
<td>? Ac_req, nas4</td>
<td>? Ac_challenge_n, nas5, tau, nas3</td>
<td>! Inq_Auth, nas0</td>
<td>? xx, GIB, ! e, nas3</td>
</tr>
<tr>
<td>nas0</td>
<td>Req, nas1</td>
<td>? Known id, nas2</td>
<td>? Ac_req, R0, ? Ac_req, R1, tau, R2</td>
<td>! Ac_challenge_n, nas0</td>
<td>? Ac_req, nas1</td>
<td>? Known id, nas2</td>
<td>? Ac_req, nas4</td>
<td>? Ac_challenge_n, nas5, tau, nas3</td>
<td>! Inq_Auth, nas0</td>
<td>? xx, GIB, ! e, nas3</td>
</tr>
<tr>
<td>nas0</td>
<td>Req, nas1</td>
<td>? Known id, nas2</td>
<td>? Ac_req, R0, ? Ac_req, R1, tau, R2</td>
<td>! Ac_challenge_n, nas0</td>
<td>? Ac_req, nas1</td>
<td>? Known id, nas2</td>
<td>? Ac_req, nas4</td>
<td>? Ac_challenge_n, nas5, tau, nas3</td>
<td>! Inq_Auth, nas0</td>
<td>? xx, GIB, ! e, nas3</td>
</tr>
</tbody>
</table>

119
Appendix B. Robustness Test Cases
Appendix B. Robustness Test Cases
Bibliography


[12] BERR. Information security: How to write an information security policy. technical report, Departement for Business Enterprise and Regulatory Reform, BERR.


