FORMAL METHODS AND TOOLS FOR TESTING COMMUNICATION PROTOCOL SYSTEM SECURITY

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the
Graduate School of The Ohio State University

By

Guoqiang Shu, M.S.

* * * * * *

The Ohio State University
2008

Dissertation Committee:

Dr. David Lee, Adviser
Dr. Neelam Soundarajan
Dr. Dong Xuan

Approved by

Adviser
Graduate Program in Computer Science and Engineering
Communication protocol is the cornerstone of today’s network and distributed computing system. The correct functioning of a protocol relies on both a flawless specification and a correct implementation. While prevalent work focuses on validating the soundness of specification, implementations are inherently more complicated and might introduce discrepancy and very likely vulnerabilities. Hence, study on theory and practice of protocol testing is of significant importance. Furthermore, with the increasing consideration for properties such as security, privacy and robustness of protocol systems, the role of protocol testing goes beyond the traditional notion of checking conformance of an implementation. Existing solutions for testing security features rely largely on expert insights of protocols and manual efforts, and could hardly be generalized or automated.

Formal methods have proven to be promising toward developing automated and generic protocol verification and testing methods. This thesis studies a model based formal approach for testing security related properties for communication protocol systems. The first part of the work introduces a new formal protocol model – Symbolic Parameterized Extended Finite State Machine (SP-EFSM). SP-EFSM model augments the traditional communicating EFSM model with (1) a symbolic protocol message language and (2) parameterized input and output symbols in order to cope
with the rich semantics of modern protocol with cryptographic primitives. A protocol is specified by a set of SP-EFSM (each is called a component) interacting with each other through sending and receiving messages. Various properties – both functional and non-functional – and verification problems could be specified using this formal model.

The power of the proposed methodology is demonstrated by its application to several newly emerging and interesting research problems related to network protocol security and reliability. The second part of this thesis shows that the formal modeling techniques can contribute to solutions that are fundamentally more general and automatic than existing ones. We first study the problem of black box testing of message confidentiality – an important security property under Dolev-Yao attacker model. While it is well known that for validation problem could be reduced to a reachability problem in the composed SP-EFSM model, testing this property faces a major challenge that the specification may not be available or comprehensive to cover all implementation details. For this reason it is hard apply classic conformance testing approaches. To overcome this difficulty a supervised learning based approach is used to discover the internal structure of black-box implementations by active testing and to validate the confidentiality property on-the-fly.

The same formal modeling methodology is also applied to solve the problem of black-box protocol fingerprinting. Protocol fingerprinting refers to the process of distinguishing between different protocol implementation by their input and output behaviors, and it has been regarded as both a potential threat to cyberspace security and also as an effective mechanism for network management. SP-EFSM model is used
to record the distinguishing structural aspects of an implementation by its states and transitions. A complete taxonomy of fingerprint matching and discovery problems is identified, based on (1) how much information about the candidate implementations is known and (2) whether the fingerprinting experiment is active or passive. For fingerprint matching algorithm, we propose an online separation algorithm for active experiment and concurrent passive testing for passive experiments. For fingerprint discovery problem, there are two cases: if the protocol specification is available as a nondeterministic SP-EFSM, we apply across verification and back-tracing technique for active and passive discovery, respectively; if no specification is available, we take the learning approach described above and discover the fingerprint by active testing.

In last part of the thesis, the practicality of the proposed formal model is investigated. In the test generation module of a real world security testing system, SP-EFSM model is integrated and used to specify sophisticated features of a security device. Algorithms for manipulating the model are implemented, including particularly reachability analysis algorithm and test sequence generation algorithm. Test sequences are automatically translated into template of executable low level test cases. By providing test cases with high fault coverage and reducing significantly amount of manual effort required, we show that using formal techniques benefits network protocol testbed design in a realistic way.
DEDICATION

Dedicated to my wife, Mrs. Lei Chai

To my parents, Mr. Zhencheng Shu and Mrs. Hua Jin
ACKNOWLEDGMENTS

I would like to extend my gratitude to my adviser, Dr. David Lee, for his inspiration, continuous encouragement, generous tolerance, and financial support during my graduate studies. I have no doubt that I will benefit from what I have learnt from him - both technical knowledge and academic spirits - for the rest of my career.

I would like to thank Dr. Neelam Soundarajan for his guidance and help in the area of software engineering especially during the early stage of my Ph.D. study. Appreciation also goes to Dr. Dong Xuan and Dr. Paul A. G. Sivilotti for joining my Ph.D. candidacy examination and thesis reading committee and giving lots of valuable comments and suggestions.

I am indebted to many of my colleagues, coauthors and friends from the CSE department. I have been working with Dr. Dongluo Chen, Zhijun Liu, Na Li, Lifeng Sang and Yating Hsu on various research projects related to this thesis. Hui Cao and Wenjun Gu have helped me constantly during these years. Having those wonderful people around me has made my life fruitful and enjoyable.

Finally, I would like to express special thanks to my dear wife and parents for their strong, lasting, unconditional love and support.
VITA

Oct 14, 1978
Born in Beijing, P.R.China

1996-2000
B.S. in Computer Science
Peking University, Beijing, P.R. China

2000-2003
M.E. in Computer Science
Institute of Software, Chinese Academy of Sciences, P.R. China

2003-Present
OSU Fellow
Graduate Teaching and Research Associate,
The Ohio State University

PUBLICATIONS


FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

<table>
<thead>
<tr>
<th>Field</th>
<th>Instructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer Networking</td>
<td>Prof. David Lee</td>
</tr>
<tr>
<td>Software Methodology</td>
<td>Prof. Neelam Soundarajan</td>
</tr>
<tr>
<td>Software Systems</td>
<td>Prof. Paul Sivilotti</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>ii</td>
</tr>
<tr>
<td>DEDICATION</td>
<td>v</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>vi</td>
</tr>
<tr>
<td>VITA</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>xiii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xiv</td>
</tr>
<tr>
<td>Chapters:</td>
<td></td>
</tr>
<tr>
<td>1 Background and Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Status and Limitation of Existing Approaches</td>
<td>2</td>
</tr>
<tr>
<td>1.2.1 Problem of Testing Security Properties</td>
<td>3</td>
</tr>
<tr>
<td>1.2.2 Problem of Protocol Fingerprinting</td>
<td>4</td>
</tr>
<tr>
<td>1.2.3 Problem of Protocol Fuzz Testing</td>
<td>6</td>
</tr>
<tr>
<td>1.3 Summary of Related Works</td>
<td>7</td>
</tr>
<tr>
<td>1.4 Significance of the Study</td>
<td>11</td>
</tr>
<tr>
<td>1.5 Overview of the Dissertation</td>
<td>12</td>
</tr>
<tr>
<td>2 A Formal Communication Protocol Model</td>
<td>14</td>
</tr>
<tr>
<td>2.1 Introduction to Basic Models</td>
<td>14</td>
</tr>
<tr>
<td>2.1.1 The Finite State Machine Model</td>
<td>14</td>
</tr>
<tr>
<td>2.1.1 The Extended Finite State Machine Model</td>
<td>16</td>
</tr>
<tr>
<td>2.2 The SP-EFSM Model</td>
<td>17</td>
</tr>
<tr>
<td>2.2.1 Protocol Message</td>
<td>18</td>
</tr>
<tr>
<td>2.2.2 SP-EFSM Model</td>
<td>19</td>
</tr>
<tr>
<td>2.2.3 An Example of TCP Protocol Model</td>
<td>20</td>
</tr>
<tr>
<td>2.3 Reachability Analysis of SP-EFSM Model</td>
<td>22</td>
</tr>
</tbody>
</table>

x
6 Conclusions.............................................................................138

Bibliography..............................................................................140
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 2.1: Example of SP-EFSM transition in NoFR Model</td>
<td>22</td>
</tr>
<tr>
<td>Table 3.1: Detection of errors in two implementations of N-S-L protocol</td>
<td>44</td>
</tr>
<tr>
<td>Table 3.2: Test sequence lengths generated by Algorithm 3.3</td>
<td>53</td>
</tr>
<tr>
<td>Table 3.3: Three protocols used in experiments (a) N-S-L (b) TMN (c) SSL 3.0</td>
<td>64</td>
</tr>
<tr>
<td>Table 4.1: Taxonomy of network fingerprinting problems</td>
<td>77</td>
</tr>
<tr>
<td>Table 4.2: Passive matching using the fingerprint model from Figure 2.2</td>
<td>79</td>
</tr>
<tr>
<td>Table 4.3: Fingerprinting sets from Nmap</td>
<td>86</td>
</tr>
<tr>
<td>Table 4.4: Passive fingerprinting on a TCP trace generated by TBIT</td>
<td>95</td>
</tr>
<tr>
<td>Table 5.1: Three types of PCOs in VCSTC</td>
<td>117</td>
</tr>
<tr>
<td>Table 5.2: Node mapping of the test case with 2 FAT nodes</td>
<td>126</td>
</tr>
<tr>
<td>Table 5.3: Summary of the firewall/IDS test suite of VCSTC</td>
<td>129</td>
</tr>
<tr>
<td>Table 5.4: Performance evaluation of the VCSTC testbed</td>
<td>134</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.1: An SP-EFSM model of TCP protocol with no fast retransmission</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>Figure 2.2: An SP-EFSM model and its Reachability Graph with four states</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>Figure 2.3: SP-EFSM model of Needham-Schroeder-Lowe (N-S-L) Protocol. The protocol language $L = {KU0, KU1, KU2, N01, N02, N10, N11, N2}$</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>Figure 3.1: SP-EFSM model of the intruder</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>Figure 3.2: Example of a mutation operator $\delta_{PA}$</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>Figure 3.3: Boolean expressions in N-S-L protocol specification</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>Figure 3.4: Learning-based active testing procedure</td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>Figure 3.5: Example of using $L^*_fsm$ to learn a three-stated FSM</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>Figure 3.6: Result of testing (a) N-S-L (b) TMN (c) SSL protocol implementations</td>
<td>66</td>
<td></td>
</tr>
<tr>
<td>Figure 3.7: Experiments with the learning based testing approach</td>
<td>71</td>
<td></td>
</tr>
<tr>
<td>Figure 4.1: Nmap candidate machine model of Solaris 9.0</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td>Figure 4.2: Nmap candidate machine model of Cisco IOS 11.0</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td>Figure 4.3: SP-EFSM transition diagram of TCP Tahoe implementation</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>Figure 4.4: SP-EFSM transition diagram of TCP NewReno implementation</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>Figure 4.5: Example of active fingerprint discovery with specification</td>
<td>98</td>
<td></td>
</tr>
<tr>
<td>Figure 4.6: Example of passive fingerprint discovery with specification</td>
<td>98</td>
<td></td>
</tr>
</tbody>
</table>
Figure 4.7: Performance evaluation of fingerprinting algorithms with TCP and SSL Protocol................................................................. 104

Figure 5.1: Architecture of VCSTC platform capsulated within a single server that connects to both internal and external interface of a target device .......... 114

Figure 5.2: Example of a simplified abstract test case ......................................................... 119

Figure 5.3: Internal structure of a hybrid honeynet with two types of nodes ............ 122

Figure 5.4: Example of test case with dynamic node remapping................................. 125

Figure 5.5: A simple SP-EFSM model of port blocking feature (left) and the corresponding reachability graph as a FSM (right)............................ 130

Figure 5.6: Test code and library generated from SP-EFSM test sequences.......... 131

Figure 5.7: Honeyd CPU load percentage for three networks of THIN nodes each sending UDP traffic at 2KB/s ................................................................. 135
CHAPTER 1

BACKGROUND AND INTRODUCTION

1.1 BACKGROUND

Communication protocol is the cornerstone of today’s network and distributed computing system. The correct functioning of a protocol relies on both a flawless specification and a correct implementation [51,52]. While prevalent work focuses on validating the soundness of specifications [60,66,67], implementations are inherently more complicated and might introduce discrepancies and sometimes vulnerabilities [83,102]. Hence, study of theory and practice of protocol testing is of significant importance. Furthermore, with the increasing importance for properties such as security, privacy and robustness of protocol systems [38,49,90,103], the role of protocol testing goes beyond the traditional notion of checking functional conformance [29,33,55,69,70] of an implementation. On one hand, some protocols have explicit security related requirements, such as secrecy, authentication, anonymity, as so forth. A protocol implementation must fulfill such special goals to be considered as correct. On the other hand, analysis of some security characteristics relies on functional testing. The focus of testing in this case is not the main features as documented in the manual, but rather those usually ignored by a normal user.
Examples include testing for reliability, resilience under malicious attacks, and testing for implementation privacy (fingerprinting). In the rest of this thesis, we refer to both types of activities as testing for protocol system security.

Testing for protocol security has recently attracted increasing interest from both solution vendors in network industry and research communities. It is to a large extent unknown whether and how the traditional protocol functional testing approach could be extended to meet these new challenges. In parallel, software security and reliability testing problems have been addressed by researchers using predominantly white-box approaches, in which the source code of the system under test is available for analysis. These works are not directly applicable to the task of testing for protocol security since the source code is usually not available and testing has to be done using black-box techniques. Thus, our focus in this thesis is exclusively on black-box testing.

1.2 STATUS AND LIMITATION OF EXISTING APPROACHES

Existing solutions for testing security related features rely largely on human insights or expert knowledge of the target protocol [24,28,103], and therefore cannot easily be generalized or automated. This is due both to the lack of theoretical foundation and to lack of powerful tool support. We give three examples below to show the current status of research and practice.
1.2.1 Problem of Testing Security Properties

Security protocols have been playing an important role in the critical distributed systems such as E-Commerce and military infrastructure. Most security protocols use cryptography to achieve data transmission, authentication and key distribution [66,67] in a hostile environment [3]. The existence of diverse intruders renders the resilience of those protocol systems more significant, and more challenging. Various formal modeling and analysis techniques, such as BAN logic, model-checking and strand spaces [59,73,86] have been developed in the recent years to ensure the correctness of security protocol system. These works are focused on validating the protocol specification. However, errors can also be introduced to the system in implementation phases, even if the specification is proven to be flawless. Furthermore, interconnected communication system interfaces may result in security problems, such as message content exposure. Systematic testing approaches for security protocols have been largely neglected by the research community, even though numerous reports show programming errors in security-critical systems are very common [97,98].

Testing for system security, often known as penetration testing [98] or red-team testing refers to the activity of executing a predefined test script with the goal of finding a security exploit. Thomson in [97] classified four general penetration testing methods: (1) Testing dependency; (2) Testing unanticipated user input; (3) Expose design vulnerabilities; and (4) Expose implementation vulnerabilities. Under these guidelines practical testing has been conducted in industry and proved to be very
helpful. Nonetheless, most of the current penetration testing activities are ad-hoc and rely on expert knowledge of target systems or existing exploits [38]; the cost of a comprehensive testing is high and the response time is too long. On the other hand, current testing methods are largely at system level on system misconfiguration [89] or unexpected side effect of operations [19]. Protocol level penetration testing has not drawn adequate attention yet is crucial for discovering security protocol implementation errors. Particularly, automated test selection and execution techniques are desirable for complex protocols and for real-time response to security flaws.

1.2.2 Problem of Protocol Fingerprinting

Network protocol fingerprinting has recently emerged as an interesting and also challenging problem for both researchers and system administrators. The activity of fingerprinting refers to the process of identifying a unique protocol implementation by its input and output behaviors. It has been shown by the prevalent fingerprinting tools that most Internet protocols such as ICMP, TCP, TELNET and HTTP are all subjected to this activity [15,36,87,106]. This universal presence is due to an inherent fact that network protocols cannot be completely and deterministically specified; instead numerous opportunities are left for an implementation to distinguish itself [91]. Protocol fingerprinting technique is becoming a cornerstone for various aspects of cyberspace security and privacy. It was originally regarded as a potential threat to network privacy, for the goal of disclosing sensitive system information. The situation is exacerbated when such information leads to platform-specific attacks [106]. For this reason alone, it is crucial to understand the nature of the fingerprinting problem in
order to design effective countermeasures. On the other hand, the same technique could also facilitate identification of the unique activity of network intruders, which in turn benefits the monitoring and defense infrastructure.

Protocol fingerprinting works can be dated back to the simple forms of remote operating system (OS) detection. Its task is to determine which OS is running on a remote host by exploiting the unique behavior of various protocols such as TCP and ICMP. Nmap [106] is one of the most popular OS fingerprinting tools. It provides nine special testing packets to determine more than 1000 different versions of operating system. While OS fingerprinting relies largely on analyzing the behavior of a TCP stack, TCP fingerprinting itself has been a very popular topic [20,75]. Identifying certain parameters of TCP implementations, such as initial congestion window (ICW) and retransmission timeout (RTO) value can contribute to monitoring and measurement of network performances. One interesting problem is to determine the congestion control algorithm implemented on a TCP stack. In [74] the design of TBIT tool is presented, which is effective in identifying TCP Tahoe, Reno, NewReno, and RenoPlus algorithms, as well as the support of SACK and ECN options. Technically, this is a much harder fingerprinting problem because it requires expertise of TCP to design an effective probing test. For instance, the author of [36] suggests that the difference among NewReno and Reno will be discovered only when multiple packets are dropped within the same congestion window. TBIT tool follows this observation and provides a test involving 20 input data segments. Most of the recent work about protocol fingerprinting has been focused on the development of software tools for both retrieving the fingerprint (actively and passively) and defending against malicious
fingerprinting. While many such tools have demonstrated significant practical value, these ad-hoc approaches have certain limitations. First, as the number of network protocols keeps increasing, we need a general method that is suitable for analyzing most, if not all, of them. Second, most of the current fingerprinting methods use fairly short probing sequences; these simple fingerprints could be erased easily. In general, discovering a fingerprint may need an arbitrarily long probing sequence. Third, future protocol specifications may become too complex for human engineers to analyze manually, and will need automated methods and tools. Finally, with the current ad-hoc fingerprinting methods it is difficult to conduct rigorous proof about the validity and effectiveness of the fingerprinting experiments.

1.2.3 Problem of Protocol Fuzz Testing

Network-based fuzz testing is another very effective approach to improve the security and reliability of protocol system implementations [46]. It works by mutating the normal traffic at the ingress interface of a component in order to reveal unwanted behavior such as crashing or confidentiality violation [11]. Identifying such flaws is extremely important since they might be exploited by malicious parties to launch attacks. On the other hand, it has been reported that for today’s complicated system these flaws are ubiquitous due to incorrect assumptions on the input data. However, unlike software fuzz testing where white-box approach [39,40] is widely used, protocol fuzz testing is usually conducted in an ad-hoc manner with input selected either randomly or manually [40]. With such restrictions it is very difficult to measure the comprehensiveness of testing and the level of test automation is low. With
knowledge of the protocol message format, some preliminary systematic approaches such as message type covering become feasible. However they are in general inaccurate for various reasons. For example, messages of same type could serve very different roles in a protocol session and therefore should be distinguished in testing.

Comprehensive fault coverage could be achieved, if any specification of the target protocol is available. In the case with no specification the current practice is well known to be inefficient and tedious.

1.3 SUMMARY OF RELATED WORKS

For over half a century, formal methods have empowered us to model, analyze and test communication protocols rigorously and to a large extent, mechanically. This section surveys the existing research work of using formal methods for network protocol modeling, validation and testing.

The problems of formally modeling and validation of security protocols have been well studied. Given a specification of the protocol and the desired property, various rigid approaches were invented to prove/disprove it under certain assumptions of the untrusted operational environment (including particularly attackers). One of the earliest foundational works of protocol validation is due to Dolev and Yao [27]. They modeled two restricted yet realistic categories of security protocol and proved the complexity of deciding message confidentiality property. Many following works such as [33] further extended their results using the same attacker’s model. These
fundamental results led to a common understanding that the general problem of validating security properties is hard even for simple protocols.

There was enormous effort of generalizing traditional formal systems to deal with practical security primitives like encryption/decryption, hash function and nonce. For example, BAN logic [17] introduced “belief” into an axiomatic system to reason about cryptographic protocols under the assumption of honest principals (unlike the Dolev-Yao model). Its successors like PCL [30] further enhanced its capability and hence were capable of proving many real-life protocols such as TLS, IEEE 802.11 and E-commerce transaction protocols [4]. Lowe et al. [59,60] used CSP to model protocols and found various new security flaws with their prototype tool FDR. A slightly different state based approach Mur-phi [69] was developed by Mitchell et al. and they applied it to formally analyze SSL 3.0 [70]. Model checking technique was also revisited [64,67] to check security properties such as secrecy by considering the knowledge an intruder could obtain. Strand space [105,35] was one of a few novel methods invented for security protocols. A strand space is a collection of strands with a graph structure generated by causal interaction, and protocol correctness claims may be expressed in terms of the connections between strands of different kinds. Other representative types of work include the Prolog based protocol analysis tool Interrogator [71], the extended calculus Spi [1], formal treatments for specific category protocols [16,45,72,105,107]. A common theme of the above methodologies was that the cryptographic primitives of the protocol are abstract and often assumed to be perfect. A separate thread of work took a computational perspective and studied quantitatively the impact of specific cryptographic primitives used in security
protocols. Recently it was shown [2] that these two views could be reconciled. In addition to protocol validation, formal approaches were also proved useful in modeling and analyzing the effect of attacks at network level with multiple nodes and/or protocols involved [82,89]. Formal models for trust and reputations [50] were also important topics related to security protocol design that drew wide attentions.

Compared to the validation problems for security protocols, testing problems are more challenging due to the fact that security flaws are normally caused by the unspecified yet complicated portion of protocol implementations. Under lots of situations protocol testing has to be conducted using black-box approaches, which renders many white-box software testing methods [78] unsuitable. Numerous reports showed that errors in security-critical communication systems are very common, and various protocol-specific testing tools have been developed to facilitate the activity of “penetration testing”. On the other hand, systematic and general security testing approaches have been largely neglected by the research community. Formal methods have great potential of benefit in this area, as already demonstrated by the recent research advances.

Test generation process could naturally benefit from rigid analysis of security protocol specification the same way as for functional testing, given that the security properties under test are also formally described. For example, in [45] fail-safety property was considered. A tool AUTOFOCUS was developed to test electronic purse protocol using state transition specification. Our previous work [92] studied message confidentiality under Dolev-Yao assumption of attacker and proposed active and passive heuristic algorithms for test selection. Due to the well known state explosion
problem and the nature of learning, comprehensive active test generation is very costly. Fuzz testing is an economic and effective alternative [79] where normal protocol message sequences are modified in a way that is likely to manifest security flaws. Formal specification could be used to drive the test generation process by providing insights and heuristics for fuzz operator, such as input mutants [103,8,12]. In case protocol specification is not available, the problem of systematically generating test with high coverage remains open. Also some security property such as authentication is difficult to be specified with regard to a black-box implementation, posing another challenge. Meanwhile, formal methods could be applied to protocol implementations for purpose of security analysis. In [19] FSM model was used to model implementation vulnerabilities, while in [91] extended FSM model was used to describe distinguished input/output features of a specific implementation, which is later used for protocol fingerprinting. Applying formal machine learning techniques to discover an unknown implementation is another emerging direction. Our previous work [93] studied a black-box learning based scheme [10,32] and used it to facilitate security testing. Testing and analysis of policy based security system (e.g. Firewall, IDS) has always been of primary interest for its practical importance. The correctness of such system depends heavily on the user configuration. Gouda et al. studied the access control system [42, 43] using formal models, laying out a foundation for firewall analysis. Others also studied fault detection [6] problems for given policies and conformance testing [25] problem. Practically, specialized network testbeds (e.g. [76]) are often required in order to cater the unique requirements and features for security devices [7].
1.4 SIGNIFICANCE OF THE STUDY

To the best of our knowledge the methodology proposed by this thesis is the first methodology for testing network protocol security. The results of this thesis can serve as a solid theoretical foundation for building critical protocol testing tools for the next generation. The proposed methodology supports modeling of communication protocol systems and provides solutions for a wide spectrum of problems that are related to testing black-box protocol implementations. These solutions powered by formal methods have two particular advantages over the existing solutions. (1) Universality. The formal methodology and hence the corresponding testing tools are applicable to a broad category of communication protocols, which is an obvious improvement over the current protocol specific approaches. (2) Automation. The formal methodology provides algorithms that could, in principle, be fully automated and minimize the required manual effort. This is particularly useful for complicated protocols for which the size of solution space is formidable.

The specific contributions of this thesis are threefold, and may be summarized as follows. First, the thesis proposes a formal model for communication protocols called Symbolic Parameterized Extended Finite State Machine (SP-EFSM). The new model is an extension of traditional Extended Finite State Machine (EFSM) and is more expressive by inclusion of symbolic message and input/output parameterization. A set of SP-EFSM together models a protocol system. It is shown that the model is
capable of specifying industrial size protocols such as TCP congestion control and SSL handshaking sub-protocol.

Second, the SP-EFSM model is applied in investigation of many emerging problems for testing protocol security. This thesis studies generic solutions for black-box security testing problem and fingerprinting problem. For protocol fingerprinting problem the study contributes a complete taxonomy of the problem and algorithms for each problem. For security testing problem, the formal methodology suggests a learning-based approach to overcome the complexity brought by unspecified behavior of a protocol implementation. This effort is among the first toward general and automated approaches to solve these problems. The experimental result achieved so far shows the formal approach is promising.

Third, this thesis contains an evaluation of the practicality of the proposed formal method. A real life security testing platform VCSTC is designed that could support using SP-EFSM as the modeling mechanism. Experience with the VCSTC testing platform further justifies that the proposed methodology is feasible and of good practical value for the networking and security industry.

1.5 OVERVIEW OF THE DISSERTATION

This thesis may be divided into four major topics. Chapter 2 introduces the formal modeling system: SP-EFSM and its relationship with its predecessor models. Chapter 2 also discusses how communication protocols are modeled formally using SP-EFSM. Chapter 3 and 4 are devoted to the application of SP-EFSM model and related algorithms in security testing. Chapter 3 studies black box testing problem of
security properties using message confidentiality property as an example; and Chapter 4 studies protocol fingerprinting system. The study of both problems includes problem statement, presentation and analysis of algorithms, as well as experimental study using real protocols. The next Chapter investigates the integration of SP-EFSM model into the VCSTC testing system. Finally Chapter 6 concludes this thesis.
CHAPTER 2

A FORMAL COMMUNICATION PROTOCOL MODEL

This chapter presents the formal protocol model: SP-EFSM. We first present the classic Finite State Machine (FSM) model based on which SP-EFSM is defined; then the extension is introduced followed by a discussion of the relationship between two models. Finally a section is devoted to how communication protocol is modeled using SP-EFSM model.

2.1 INTRODUCTION TO BASIC MODELS

2.1.1 The Finite State Machine Model

FSM has been widely used in system specification of various areas, including network protocols, high level software design, real-time reactive systems, and logical circuits [55]. An FSM contains a finite number of states and produces output on state transitions after receiving an input symbol. It is often used to model control portion of a protocol. Validation and testing problems are well studied [51,52,53,54] for FSM.
**Definition 2.1:** A deterministic Finite State Machine (FSM) is a six-tuple $M = \langle S, s_{init}, I, O, \delta, \lambda \rangle$, where $S$ is a finite set of states of size $N$; $s_{init}$ is the initial state;

$I$ is the finite input alphabet of size $P$; $O$ is the finite output alphabet of size $Q$;

$\delta : S \times I \rightarrow S$ is the state transition function;

$\lambda : S \times I \rightarrow O$ is the output function.

Note that state transition function is defined to be deterministic (nondeterministic FSM is outside the scope of this thesis). Also, if $\delta$ and $\lambda$ are total functions then the machine is **complete** otherwise the machine is **partial**. In practice the FSM that user supplies is often incomplete (that is, for some state and input symbol, the behavior is unspecified); in such cases usually a preprocess step is carried to make the machine complete by adding self-loop transitions or error-handling states.

Note the state transition function and output transition are usually extended to calculate the destination state and output upon a sequence of input symbols instead of a single input. A typical testing problem is the following: given a specification FSM $M_{spec}$, we want to generate a set of input sequences (test suite) to check whether an implementation $M_{impl}$ (as a black-box) conforms to $M_{spec}$, i.e. we observe the same output $\lambda(M_{spec}, \text{seq})$ if we input seq to $M_{impl}$, for any seq. It is known that without any assumptions on $M_{impl}$ it is impossible to derive a finite suite satisfying this goal. There are some natural assumptions that are made. First, $M_{spec}$ is **reduced** (**minimized**), meaning that no two states are equivalent. Second, $M_{impl}$ does not change during the test and it has no more state than $M_{spec}$. Last, usually we assume $M_{spec}$ is strongly connected. Under these assumptions, conformance is equivalent to isomorphism.
Define a **checking sequence** as a sequence \( x \) such that \( \lambda(M_{\text{spec}}, x) \neq \lambda(M', x) \) for any \( M' \) that has no more than \( N \) states and is not isomorphic to \( M_{\text{spec}} \). Various algorithms of calculating checking sequences are proposed in the literature [55]. Given the assumptions described above, both the time complexity and the length of resulting checking sequence are polynomial of \( N \) and \( P \).

Finally, note that a FSM is often described as a graph. Define the state transition diagram \( G(M) \) of a FSM \( M \) as a digraph whose nodes are the states of \( M \), and there is an edge from \( S_1 \) to \( S_2 \) if and only if \( \delta(S_1, x) = S_2 \) for some \( x \).

### 2.1.1 The Extended Finite State Machine Model

Despite of its proven usefulness, FSM is not sufficient to model in an efficient way the data intensive protocols (such as TCP) which are very common in today’s Internet. Extended FSM (EFSM) enriches FSM with variables as part of the state, and transitions with guard and actions. In general EFSM has the same power as Turing Machine and therefore can handle any complex protocols well, while when in practice we consider only variables with finite domains, EFSM is equivalent as FSM.

Each EFSM has an equivalent FSM representation, called its reachability graph [51], which takes the combination of EFSM state and variable values as state, and inherit all the transitions without guards and actions (I/O only). Because of the well known state explosion problem, reachability graph of any nontrivial EFSM will be large despite the invention of online minimization algorithms [54]. The definition of EFSM is as follows and we defer the discussion of reachability graph into next section together with the proposed new model.
**Definition 2.2:** An Extended Finite State Machine (EFSM) is a quintuple $M = \langle S, s_{init}, I, O, X, T \rangle$, where

- $S$ is a finite set of states; $s_{init}$ is the initial state;
- $I$ is the input alphabet of size $P$; each $O$ is the output alphabet of size $Q$;
- $X$ is a vector denoting a finite set of variables with default initial values;
- $T$ is a finite set of transitions. For each $t \in T$, $t = \langle s, s', i, o, p(x, i), a(x, i, o) \rangle$ is a transition where $s$ and $s'$ are the start and end state, respectively; $i$ and $o$ are the input/output symbols; $p(x, i)$ is a predicate, and $a(x, i, o)$ is an action on the current variable values and input/output symbols.

### 2.2 THE SP-EFSM MODEL

For purpose of modeling security protocols we further extend EFSM in the following ways. (1) Parameterize input and output symbols in order to model the rich formats of modern protocol messages. Consequently the predicates and actions in the EFSM can refer to the input/output parameters; moreover (2) Add symbolic protocol message language with cryptographic primitives such as encryption/decryption/hash function. Both state variables and input/output parameters have value domain defined by this language. This extension is useful in order to cope with the rich semantics of security protocols.
2.2.1 Protocol Message

We define the security protocol message type as follows. First, there are three atom types: \textit{Int}, \textit{Key} and \textit{Nonce}. A value of type \textit{Int} is a non-negative bounded integer; a value of type \textit{Key} ranges over a finite set $K$ of keys and a value of type \textit{Nonce} ranges over a finite set $N$ of nonces. $K$ and $N$ are not necessarily disjoint. Some keys are generated fresh and treated as nonce. \textit{Key} type contains both symmetric keys and asymmetric key pairs. For the latter we use $ku$ to represent a public key and $kr$ for a private key. We treat \textit{Key} and \textit{Nonce} as symbols in the sense that we are not concerned with their exact (numerical) values. In many practical protocols it is the case that some atoms are calculated from others, for example in SSL protocol the encryption key block is calculated using the master secret, which is in turn calculated from the pre-master secret and two nonces. We explicitly model such dependency relationship using a set of Derivation Rules (known as “equivalence” in the Casper language [60]). A derivation rule $dr_i$ is of the form $dr_i = <(N_{s1}, N_{s2}, N_{s3},..., N_{sk}) \rightarrow N_d>$, which means $N_d$ could be calculated using $N_{s1}...N_{sk}$ together. The rules could be graphically represented by an AND-OR graph.

A **protocol message** is recursively defined as: (1) An atom; (2) Encryption of a message with a key; (3) the Secure Hash of a message or (4) Concatenation of messages. A message can be represented by a string. For example $E(k_b, (k_a, H(n_a)))$ is a messages that is formed by encrypting the concatenation of $k_a$ and the hash of $n_a$ with another key $k_b$. Given $A=\langle K, N \rangle$, a set of keys and nonces, denote $L(A)$ as the message type and the set of messages formed using atoms in $A$. $L(A)$ is obviously infinite, and even if we restrict the number of atoms that a message contains, its size is
exponential. There are some basic operations defined on message type. Let $msg$ be a message, function $\text{Elem}(msg,i)$ calculates the $i$-th component in $msg$, and $D(k,msg)$ returns $m'$ iff. $msg = E(k,m')$ when $k$ is a symmetric key or $msg = E(k',m')$ when $(k, k')$ is an asymmetric key pair. Both functions are partial and they are undefined for messages with incompatible formats. Define function $\text{Encl}: L(A) \rightarrow L(A)$ as for $\Omega \subseteq L(A)$, $\text{Encl}(\Omega)$ is the enclosure of $\Omega$ under functions $\text{Elem}$, $D$ and $E$.

2.2.2 SP-EFSM Model

Now we can define a Symbolic Parameterized Extended Finite State Machine (SP-EFSM) model that uses finite (length-restricted) protocol message set $L(A)$ as input and output alphabet.

**Definition 2.3:** A Symbolic Parameterized Extended Finite State Machine (SP-EFSM) is a 7-tuple $M=\langle S, s_{\text{init}}, A, I, O, X, T \rangle$ where

1. $S$ is a finite set of states;
2. $s_{\text{init}}$ is the initial state;
3. $A$ is a finite set of atoms with certain derivation rules, and $L(A)$ is the set of messages formed using atoms in $A$;
4. $I = \{i_0, i_1, \ldots, i_{P-1}\}$ is the input alphabet of size $P$; each input symbol $i_k$ ($0 \leq k < P$) contains a parameter $\pi(i_k)$ of type $L(A)$;
5. $O = \{o_0, o_1, \ldots, o_{Q-1}\}$ is the output alphabet of size $Q$; each output symbol $o_k$ ($0 \leq k < Q$) contains a parameter $\pi(o_k)$ of type $L(A)$;
6. $X$ is a vector denoting a finite set of variables of type $L(A)$ with default initial values;

7. $T$ is a finite set of transitions; for $t \in T$, $t = \langle s, s', i, o, p(x, \pi(i)), a(x, \pi(i), \pi(o)) \rangle$ is a transition where $s$ and $s'$ are the start and end state, respectively; $\pi(i)$ and $\pi(o)$ are the input/output symbol parameters; $p(x, \pi(i))$ is a predicate, and $a(x, \pi(i), \pi(o))$ is an action on the current variable values and parameters.

The meaning of predicate and action follows the tradition of state machine theory [55]. There is an implicit semantic notion of protocol session in the model: A session variable $SessionID$ is initialized with 0 and incremented each time the machine leaves state $S_{init}$. All atoms in set $N$ are associated with a specific session and the derivation rules apply to atoms in the same session. In SP-EFSM we use parameterized input and output symbols to model the critical content of data packet. For instance, TCP data packets carry Sequence Number and Acknowledgement Number as parameters. Same all variants of EFSM, SP-EFSM model has the same computing power as Turing Machine, and it can be used to appropriately model network protocols.

2.2.3 An Example of TCP Protocol Model
Figure 2.1: An SP-EFSM model of TCP protocol with no fast retransmission

The state variables, guards and actions of transition are omitted.

An SP-EFSM can be graphically represented by its state transition diagram.

Figure 2.1 shows outline of the SP-EFSM model of a simplified congestion control protocol of TCP implementation. It contains five states: initial state (SYN), slow start (SS), congestion avoidance (CA), retransmission (REX) and finish (Fin). The most important two input/output symbols are $PKT[x,y]$ and $ACK[x]$, where $PKT$ is a segment of data packets parameterized by starting and ending sequence numbers; and $ACK$ is acknowledgement packet parameterized by acknowledgement number. Table
2.1 shows one transition of this model. Upon input ACK the transition takes the machine from state SS to CA. The guard checks that the acknowledge number is valid and the useful window is larger than zero. The action of this transition reduces the threshold and reset cwnd according to the TCP RFC. The output of this transition is PKT parameterized by the current useful window. Following [91] we refer to this oversimplified implementation as NoFR later in the chapter.

<table>
<thead>
<tr>
<th>Name</th>
<th>SS_CA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start State</td>
<td>SS</td>
</tr>
<tr>
<td>End State</td>
<td>CA</td>
</tr>
<tr>
<td>Input</td>
<td>ACK(ack)</td>
</tr>
<tr>
<td>Output</td>
<td>PKT(to_send,snd_nxt-1)</td>
</tr>
<tr>
<td>Guard</td>
<td>(cwnd&gt;=ssthresh &amp;&amp; (ack &gt; snd_una) &amp;&amp; (ack&lt;=snd_max) &amp;&amp; (ack+cwnd&gt; snd_nxt))</td>
</tr>
<tr>
<td>Action</td>
<td>ssthresh=cwnd/2; if (ssthresh&lt;2) ssthresh=2;cwnd = 1;snd_nxt = snd_una+1;ack_dup = 0;</td>
</tr>
</tbody>
</table>

Table 2.1: Example of SP-EFSM transition in NoFR Model

2.3 REACHABILITY ANALYSIS OF SP-EFSM MODEL

An SP-EFSM is a compact representation of a FSM, called its reachability graph, which can be calculated by exploring its configuration space and eventually removing the predicate and parameters of the transitions.
A configuration $cfg$ of SP-EFSM $M$ is a combination of its state and variable values: $cfg = \langle s, V \rangle$, where $s \in S$ and $V:X \rightarrow L(A)$. Denote the set of all configurations as $CFG$. Upon an input, a transition takes the SP-EFSM from one configuration to another if the predicate is satisfied. The initial configuration $cfg_0$ is determined by $s_0$ and the default initial value of state variables. A configuration $cfg$ is reachable if there is a sequence $seq \in I^*$ (with specific parameter values) that takes the machine from $cfg_0 \in CFG$ to $cfg$. The reachability graph of an SP-EFSM is a FSM $(S^R, S^R_0, I^R, O^R, f_{next}, f_{output})$ defined as follows: $S^R = \{ cfg | cfg \in CFG \land cfg$ is reachable $\}$; $S^R_0 = cfg_0$, $I^R = L(A)$; $O^R = L(A)$; $f_{next} : S^R \times I^R \rightarrow \text{POW}(S^R)$, $f_{next}(cfg, i) = \{ cfg' | \exists t \in T, t$ takes machine from $cfg$ to $cfg'$ upon $i$ $\}$; $f_{output} : S^R \times I^R \rightarrow \text{POW}(O^R)$, $f_{output}(cfg, i) = \{ o \in O^R | \exists t \in T, t$ outputs $O^R$ from $cfg$ upon $i$ $\}$.

Even this definition allows nondeterminism in the reachability graph, in this thesis we only study deterministic protocol implementation. We say a SP-EFSM is deterministic if its reachability graph is deterministic. Note that it is still possible that from a state in a deterministic SP-EFSM there are multiple out-going transitions on the same input, as long as the predicates are mutually exclusive. Also note that although $f_{next}$ and $f_{output}$ are total function, they may have empty set value for many inputs. We call such an FSM partially defined. Finally, $f_{next}$ and $f_{output}$ are lifted in a natural way to be defined for a sequence of input (instead of a single input) therefore we can compute the output of a SP-EFSM given a sequence as input.

Given a SP-EFSM model, its reachability graph could be calculated mechanically using a breadth-first search algorithm. The algorithm is included in the prototype tool and we outline it below. The algorithm could be efficiently
implemented given the specific language of choice in which transition predicates and actions are written.

Algorithm 2.1. (Reachability analysis for SP-EFSM model)

Input: SP-EFSM Model <S, s\text{init}, A, I, O, X, T>

Output: Reachability Graph M_{rch};

1. Initialize M_{rch} as an empty FSM; Q as an empty queue;
2. \( \text{cfg}_0 = <s_{\text{init}}, v_{\text{init}}>; \) // \( v_{\text{init}} \) is the default values of state variables
3. visited = {};
4. \textbf{foreach} \( p \) \textbf{in} I \textbf{do}
5. \hspace{1em} \text{Val}(p) = \text{GET\_PARAM\_VALUES}(p); // Calculate the value domains
6. \text{FSM\_ADD\_STATE}(M_{rch}, \text{cfg}_0);
7. \text{ENQUE}(Q, \text{cfg}_0);
8. \textbf{while} (!\text{EMPTY}(Q))
9. \hspace{1em} \text{cfg}(s_1,v_1) = \text{DEQUE}(Q) ;
10. \textbf{foreach} \( t(s_1,s_2,p_{\text{in}},p_{\text{out}},\text{pred},\text{act}) \) \textbf{in} T \textbf{do}
11. \hspace{2em} \textbf{foreach} \( v_{\text{in}} \) \textbf{in} \text{Val}(p_{\text{in}}) \textbf{do}
12. \hspace{3em} \textbf{if} (!\text{EVALUATE}(\text{pred}, v_1, v_{\text{in}})) \textbf{continue}; // t is not enabled
13. \hspace{3em} v_2 = \text{EVALUATE}(\text{act}, v_1, v_{\text{in}}, v_{\text{out}}); // execute the action
14. \hspace{3em} \textbf{if} (\text{cfg}(s_2,v_2) \notin \text{visited})
15. \hspace{4em} \text{FSM\_ADD\_STATE}(M_{rch}, \text{cfg}(s_2,v_2)) ;
16. \hspace{4em} \text{ENQUE}(Q, \text{cfg}(s_2,v_2));
17. \hspace{3em} \text{visited} += \text{cfg}(s_2,v_2);
This straightforward algorithm works by visiting each new configuration and trying to reach more unknown configuration through a transition. It uses many primitive operations on queue and FSM (capitalized function calls), for which detailed implementation is omitted. This algorithm has time complexity $O(P \cdot N)$ where $P$ is the number of packets values and $N$ is the number of reachability configurations (states in FSM). Figure 2.2 shows the output this algorithm on a single-stated SP-EFSM model. Note that the resulting FSM should be further reduced to a minimized machine in which for each pair of states there exists an input sequence that can separate them. For instance, in Figure 2.2 the states (0, 0) and (1, 0) could be merged and so are (0, 1) and (1, 1). One challenge for implementing this algorithm is the evaluation of predicates and actions, because they are defined to be expressive to model real protocols. In our tool for SP-EFSM model we employ online code generation technique to achieve symbolic execution, and we can support writing predicate and action using modern programming language such as Java.
2.4 COMMUNICATION PROTOCOL SPECIFICATION

A protocol contains at least two principals. Each principal is identified uniquely. We use SP-EFSM to specify a principal in the communication protocol.

Definition 2.4: A communication protocol component is specified by a SP-EFSM with the following constraints: (1) It contains a variable $id$: the unique identifier of this component within the protocol system. (2) The input/output (I/O) alphabet is each divided into two subsets: internal (interaction within the component) and external
(interaction with another component). External input (output) symbols carry an extra parameter to denote the message receiver (sender). (3) No transition contains both an external input and an external output.

According to (2) we denote an external input message received from \( M \) as \( M_a?i \) and an output to \( M_a \) as \( M_a!o \). The semantics of message sending/receiving follow the typical synchronous communication model such as in CSP: the I/O is executed as a rendezvous and simultaneously. The restriction (3) is concerned with the modeling of attackers that is introduced in next quarter. We use two chained transitions (one with external input only and one with external output only) to simulate such scenario and it does not affect the capability of the model in general.

Definition 2.5: A communication protocol is specified as a set of protocol components: \( M_{spec} = \{M_1,M_2,\cdots,M_C\} \), where each component has a unique \( id \), and that they all share the same message type \( L \).

Each component machine \( M_k \) represents a principal in the protocol system. It is possible that two machines are identical except for the identifiers, meaning there are symmetric peers in the protocol. Naturally we require that the sender/receiver parameter in Definition 2.4 is within range \([1..C]\). A trace from the specification is a set of ordered input/output events generated by the component machines. From the traces, an external event corresponds to a message exchange between two component
machines, and they are observable to the network monitor. We define an observable trace (message trace) as a sequence of messages exchanged among components.

Definition 2.6: A message trace from a protocol $M_{\text{spec}}$ with message type $L$ is $tr = \{<s_1,r_1,msg_1>,<s_2,r_2,msg_2>,...,<s_k,r_k,msg_k>\}$, where the messages are sequentially generated by $M_{\text{spec}}$ with no other messages in between, and each triple contains the identifier of the sender machine $s_i$, receiver machine $r_i$ and the message $msg_i$ in $L$. Denote $Msg(tr)$ as the set of messages appeared in $tr$. Denote all message traces $M_{\text{spec}}$ generates as $TR(M_{\text{spec}})$. 
(a) Initiator
Figure 2.3: SP-EFSM model of Needham-Schroeder-Lowe (N-S-L) Protocol. The protocol language $L = \{ KU0, KU1, KU2, N01, N02, N10, N11, N2 \}$. 

(b) Responder
As an example, we show how the classic Needham-Schroeder-Lowe (N-S-L, [73]) two-way authentication protocol is modeled using SP-EFSM. There are two components machines for the initiator and responder, with id 0 and 1, respectively. Each of the components possesses a key pair and two nonces for each session (assuming there could be 3 valid identities). I/O symbols \{Ask, Rpl, Cfm\} represent three types of message exchanged in the protocol. The transitions are shown in Figure 2.3 with the guard and actions omitted due to space limit. Specifically, the message trace that leads to a successful run of the protocol is \{<M_0, M_1, Ask[1,E(KU1,N01,0)>], <M_1, M_0, Rpl[0,E(KU0,N01,N10,1)]>, <M_0, M_1, Cfm[1,E(KU1,N10)]> \}.

We define a **protocol component implementation** as a completely defined deterministic finite state machine that has the same input alphabet as the specification component. In this thesis we study only black-box implementations, meaning that they can only be explored by input/output behaviors. Structural information such as source code or state identification is not available.

This chapter is concluded with a note on applying SP-EFSM and its simplified ancestors (including PEFSM, EFSM and FSM) in practical problems. SP-EFSM is most general (and most powerful) form in the family; however it is not always the appropriate one. For a particular protocol or problem, simpler models might be preferable. Furthermore, certain algorithms (such as the learning algorithm we study in the thesis) are not directly applicable on SP-EFSM, and we need to transform it to an equivalent FSM first.
CHAPTER 3

TESTING SECURITY PROPERTIES

The following two chapters present the result of applying the SP-EFSM formal model to a series of practical security testing problems. This chapter focuses on a representative problem of testing message confidentiality property (a.k.a. secrecy property). First this property is formally defined with regard to the Dolev-Yao assumptions of attackers. For black box testing of security properties like message confidentiality, the most outstanding challenge is that unlike conformance testing, security testing deals with the behaviors beyond what is specified. The limitation is implicitly reflected by assumptions about the implementation (e.g. number of states) in the existing works. Based on SP-EFSM a learning based active testing approach is proposed to overcome this limitation. In this chapter we describe some simple heuristic active and passive testing procedures; then we present the result of our new method.
3.1 MESSAGE CONFIDENTIALITY PROPERTY

3.1.1 Model of Dolev-Yao Intruder

The specification formalism we have introduced in Chapter 2 defines the honest parties in the protocol system, while we shall also define the behavior of the malicious party rigorously. In this thesis we adopt Dolev-Yao’s assumptions for two party message exchange protocols [27] that define a widely accepted powerful intruder model. It has been proved that one intruder poses the same security threat as multiple intruders and we model only one in our study.

An intruder is first a legitimate principal of the communication system; it can not only initiate a session with any other component machine $M_a$ but also be the (passive) peer of any session. Furthermore, the intruder is capable of intercepting messages between any two legitimate principals. The important effect of this behavior is that the semantics of message sending and receiving in the original communicating SP-EFSM model are altered. A transition in $M_a$ with output message $M_b!msg$ now will be jointly executed with a transition in $M_I$ that takes input $M_a\rightarrow M_b?msg$, instead of the transition in the intended receiver $M_b$. This should be clearly distinguished with the first case where the intruder $M_I$ is the intended receiver (e.g. $M_a$ outputs $M_I!msg$). Similarly, the intruder can inject any message, impersonating any other machines. That is, $M_I$ can send output $M_a\rightarrow M_b!msg$ to $M_b$ and this output matches the transition of $M_b$ with input $M_a?msg$.

Besides the capability of catching and injecting normal protocol traffic, the intruder is also assumed to be able to generate any new message based on all and only
the messages it possesses. Formally, we define the knowledge of the intruder as a set
of messages $\Omega = Encl(\Omega_0+MSG)$ where $\Omega_0$ is the initial knowledge known to the
intruder containing the public and intruder’s own information, and $MSG$ represents the
set of messages the intruder has received. Function $Encl(L)$ is defined as the enclosure
of $L$ under the functions $Elem()$, $D()$ and $E()$. Therefore, $\Omega$ can be regarded as all the
messages that the intruder is able to construct, using only the messages it obtains.

Once the intruder gains a message it will not forget it and $\Omega$ never shrinks. As far as a
realistic testing scenario is concerned, we have to assume the intruder has the
capability of recognizing the message format, either by guessing the data field, or by
reading the meta-info such as an XML schema.

![Figure 3.1: SP-EFSM model of the intruder](image)

$M_I$
State Variable
$\Omega$: $L$
Parameters
$a, b \in [1..C]$  
$\text{msg} \in L$

$\text{Intercept } (M_a \rightarrow M_b?\text{msg}) / -$

$[a \neq I] \llbracket \Omega = Encl(\Omega + \{\text{msg}\}) \rrbracket$

$\text{Inject } (M_a \rightarrow M_b!\text{msg})$

$\llbracket \text{msg in } \Omega \rrbracket \& [b \neq I] \llbracket \{ \rrbracket$
Figure 3.1 shows the SP-EFSM model of the intruder. $M_I$ contains only one state and two transitions for message interception and injection respectively. \emph{Intercept} transition takes any message $msg$ sent from $M_a$ to $M_b$ as input. The guard ensures $msg$ is not from $M_I$ itself and the action updates the knowledge set. \emph{Inject} transition outputs a message in the current knowledge set to another machine $M_b$ under the disguise of $M_a$. The model of $M_I$ is obviously independent of the other component machines. Note that the existence of Dolev-Yao attacker alters the message exchange behaviors of the whole system to allow message interception and injection. That is, whenever an attacker is composed with other SP-EFSM representing honest principals, the input (output) transitions of the latter are always fired with \emph{Inject} (\emph{Intercept}) transition of $M_I$. Passive attacker (or eavesdropper) could be regarded as a special case of Dolev-Yao attacker that injects a message to the original recipient immediately after interception.

3.1.2 Message Confidentiality Property

Among the various security properties, we now focus on secrecy, also known as \textbf{Message Confidentiality}. Given the specification of protocol \{\(M_1, M_2, \ldots, M_C\}\} and the attacker \(M_I\), the global behavior of the whole protocol system under investigation is described by the Cartesian product of all the machines: \(M_1 \times M_2 \times \cdots \times M_C \times M_I\). Since all the transitions involve one of the two intruder transitions, a message trace produced by the system can hence be described by an interleaving sequence of \emph{Intercept} and \emph{Inject} transitions; each contains a message as parameters. Intuitively, the protocol is insecure if the attacker could learn a secret from any message trace.
Given a protocol specification $M_{spec}$ and a set of secret messages $S$, it is of primary interest to check whether $M_{spec}$ is secure, w.r.t. $S$. This is the classic protocol validation problem and has been extensively studied with numerous approaches such as model-checking and theorem proving. It is well understood that in general this problem is undecidable, and even with a finite number of messages and sessions it is still NP-hard [84]. In our framework, we restrict the problem to finite session and it could be reduced to a state reachability problem [92] of the integrated model.

We now focus on black-box testing problems. An implementation of protocol component specified by $M_i$ could be any deterministic machine $B_i$ that shares the I/O alphabet and message language $L$ with $M_i$. For security testing, we are not interested in arbitrary implementations; instead we focus on those that include the specified normal behavior. We call an implementation $B$ an acceptable implementation with respect to $M_{spec}$ if $TR(B_i) \supseteq TR(M_i)$ for all $1 \leq i \leq c$. This notion of acceptability is slightly different than weak conformance defined in [55]: in addition to the agreement on the normal behavior, an acceptable implementation may contain extra input symbols, and therefore allow extra traces correspondingly. In general trace inclusion relationship does not imply the structural subsumption of either the SP-EFSM model or its reachability graph. The implementation may contain more states and a significantly different transition diagram. Implementations are treated as pure black-box (i.e. with unobservable internal structure). The definition of insecurity is therefore defined for protocol implementation as follows.
Definition 3.1: Implementation $B = \{B_1, B_2, \ldots, B_C\}$ is insecure with regard to the confidentiality of message $m \in L$ if and only if there exists a message trace $tr$ such that $m \in Encl(\Omega_0 \cup Msg(tr))$.

Obviously, the goal of message confidentiality testing of this chapter is to find security violation of a given black-box acceptable implementation $B$.

3.1.3 Needham-Schroeder-Lowe Protocol Revisited

Now we consider the message confidentiality property for the N-S-L protocol described by Figure 2.3. It is helpful to refer to the following message exchange sequence between two principals: initiator and responder.

$$A \rightarrow B \text{ (Ask)}: A.B.E(KU_{B_1}, (N_A, A)) \quad (1)$$

$$B \rightarrow A \text{ (Rpl)}: B.A.E(KU_{A_1}, (N_A, N_B, B)) \quad (2)$$

$$A \rightarrow B \text{ (Cfm)}: A.B.E(KU_{B_2}, (N_B)) \quad (3)$$

The protocol functions as follows. The initiator $A$ encrypts a nonce with the responder $B$’s public key and sends it (message (1)) to $B$. $B$ then decrypts it and encrypts it together with another nonce using $A$’s public key (message (2)). Finally $A$ gets the second nonce and sends it back (message (3)). The purpose of N-S-L protocol is to allow both principles authenticate each other and exchange some secrets (two nonces), which later on can be used to construct shared keys. We assign index 0, 1 and 2 to $M_A$, $M_B$ and the intruder $M_I$. The intruder can participate legally as both the initiator and responder. The atom messages in this protocol include the public keys


\( (KUA, KUB \text{ and } KUI) \) and the nonces. In order to express the security requirement conveniently, we distinguish the nonces used for different peers. For instance, the nonce \( M_B \) uses to challenge \( M_I \) is \( N_B[I] \). Initially the intruder only knows its own key and nonces, i.e., \( \Omega_0 = \{KUI, N_I[A], N_I[B]\} \). The secret message set is \( M^* = \{N_A[B], N_B[A]\} \); the intruder should not obtain the nonces that are only supposed to be shared only between \( A \) and \( B \).

### 3.2 SIMPLE ONLINE TESTING SCHEMES

#### 3.2.1 A Simple Passive Monitoring Algorithm

A passive tester or monitor of security protocol implementation is easy to devise. The intruder (tester) intercepts all messages among the component machines, updates its knowledge, replies if the message is directed to itself, and otherwise forwards it without any modification. The testing terminates when the intruder derives any secrets. The procedure is shown in Algorithm 3.1. As inherent to all passive testing approaches, this algorithm only utilizes part of the intruder’s capability and it is suitable when the intruder could only conduct eavesdropping [3].
Algorithm 3.1. (Passive monitoring algorithm)

\textbf{Input:} \{\(B_1, B_2, \ldots, B_C\)}, message secrets \(M^*\), Intruder initial knowledge \(\Omega_0\).

\textbf{Output:} security flaw if observed.

\textbf{begin}

1. \(\Omega=\Omega_0\);

2. \textbf{while} (true)

3. execute \textit{Intercept} (\(B_i \rightarrow B_j\?msg\)) transition with any \(B_i\);

4. \textbf{if} (succeed)

5. \textbf{if} (\(M^* \cap \Omega \neq \phi\)) return flaw;

6. \textbf{if} (\(j=I\))

7. generate reply \(msg'\);

8. execute \textit{Inject} (\(M_I \rightarrow B_i\?msg'\)) transition;

9. \textbf{else}

10. execute \textit{Inject} (\(B_i \rightarrow B_j\?msg\)) transition;

\textbf{End}

3.2.2 An Active Guided Random Walk Algorithm

Now we study active testing approaches that utilize the full power of the intruder. One simple-minded method of active testing is random walk. Starting with an initial knowledge set, the intruder (tester) randomly chooses either to intercept a message from a pair of principals or to construct a message using its current knowledge and send it to a principal. Pure random walk has several limitations; the
coverage of the model is not high and, more importantly, it does not use the intruder’s knowledge acquired. We present a guided random walk approach with a high coverage and fully utilizing the intruder’s knowledge acquired.

The approach is adaptive and unstructured in terms of the composite (global) state machine. We keep track of the current state $S_i$ and variable values $X_i$ for each black box $B_i$ in order to guide the selection of next transitions. Note that in general tracking current state is not always possible even under the assumption that $B_i$ contains no transition errors; it is due to the fact that part of the message is encrypted and intruder can not utilize the information to infer the current transition and state if he does not have the key. In this case, the algorithm makes a random guess.

At each step, the intruder always tries to intercept the messages coming from every machine $B_i$. Once a message is intercepted, the state of the sender as well as the intruder’s knowledge is updated. Then the intruder constructs a message and injects it to a machine to fire a carefully selected transition. Our algorithm selects transition and message based on the following criteria. First the transitions of all component SP-EFSMs should be covered fairly. The algorithm keeps track of a counter $cnt[t]$ for each transition $t$, and at each step the one that has been executed least is favored. Moreover, we only select the transitions that could possibly be enabled by some input message and ignore those transitions that will definitely not be triggered (the current state variables themselves disable the predicate). We calculate $T_{true}$ as the set of all possible transitions:

$$T_{true} = \{ t < S_i, S'_i, I, p, a > \mid t \in M_i \text{ and } \exists \text{msg} \in \Omega: p(X_i, \text{msg}) = \text{true} \}$$
Once a transition \( t \) is determined, we construct an enabling input message for \( t \) using a greedy algorithm. Ideally we want an input message that will lead the machine to a state that can generate more new knowledge. That is, for all candidate messages we calculate the destination state \( S' \) of \( t \), and select one that enables at least one output transition \( t' \) with parameter \( \text{msg}' \) not in \( \Omega \). We use subroutine \( \text{lookahead}(\Omega, S, X, t) \) to calculate such messages. If such messages do not exist or there are ties, an enabling message is randomly picked:

\[
\text{lookahead}(\Omega, S, X, t < S_i, S_i', I, p, a>) = \\
\{ \text{msg} \mid p(X, \text{msg}) = \text{true} \text{ and } (\exists t < S_i', S_i'', O(\text{msg}'), p', a' > : p'(X_i) = \text{true} \text{ and } \text{msg}' \not\in \Omega) \}
\]

Algorithm 3.2. (Active testing algorithm: guided random walk)

**Input:** \( \{B_1, B_2, \ldots, B_C\} \), secrets \( M^* \), Intruder initial knowledge \( \Omega_0 \), \( L>0 \).

**Output:** An adaptive test sequence.

**begin**

1. initialize each \( M_i \), for all transition \( t \), \( \text{cnt}[t] = 0 \);
2. \( X=<X_1, \ldots, X_C>, S=<S_i, \ldots, S_c> \);
3. \( \Omega=\Omega_0 \), \( \text{seq}=\phi \);
4. while (seq.length < \( L \))
5. \hspace{1em} foreach component in \( B_i \) do
6. \hspace{2em} execute \( \text{Intercept}(B_i \rightarrow B_j)?\text{msg} \) with \( B_i \);
7. \hspace{2em} if (succeed)
8. \hspace{3em} deduce or guess the transition \( t \);

**end**
To avoid infinite tests, the algorithm terminates when either the secret message content is obtained or the length of test sequence reaches a preset limit. This algorithm is more effective than random walk because the greedy heuristics take into account both coverage and intruder knowledge acquisition. However, it still has many inherent limitations. For example, calculation of $T_{true}$ and $lookahead()$ is rather expensive. Also, the effectiveness of the heuristic relies on the estimation of current state and variable values, and if it fails the algorithm behaves the same as random walk. Advanced passive testing techniques [51] that estimate data portion more accurately could be applied here to improve the performance.

3.2.3 Experiments of Simple Active and Passive Algorithms

We conduct an experiment of Algorithm 3.2 on N-S-L protocol specified as Figure 2.3. Two implementations are created with a common programming error in
each. Then we treat them as black-boxes and run the algorithm to test for confidentiality violations.

**Implementation X:** The responder does not verify the encrypted identifier of the initiator after it receives Ask message, and proceeds as if it were correct.

**Implementation Y:** The initiator does not verify the encrypted identifier of the responder after it receives Rpl message, and proceeds as if it were correct. This error was first uncovered by Lowe [59] as a design flaw in the original Needham-Schroeder protocol.

For both Implementation X and Y errors have been detected. Table 3.1 shows the successful test sequences for them. In the first test sequence, at the beginning the intruder intercepts an Ask message from $M_0$ to $M_1$, and updates the state to $<S_{i1},S_0>$. Now three transitions are feasible and as the result $M_1$?Ask is selected. $\text{Lookahead()}$ returns a random message that enables $M_1$?Ask because no message will further trigger an output transition. In the second round we intercept an Rpl message, and the intruder will obtain a secret ($N_0[1]$) and terminate the test. The sequence for $Y$ is more complex. After injecting an Ask message to $M_1$ and intercepting the response, we have two transitions in $T_{true}$. $M_1!Cfm$ is chosen and executed with a random message. At next step $M_0$ happens to initiate a session with $M_I$. This is a rare event yet critical for detecting errors in this implementation. The only transition that could be enabled is $M_0$?Rpl, and now the intruder happens to have a message to enable it. The last step is the interception of Cfm message from $M_0$ that exposes the nonce – secret $N_{i1}[0]$. 
<table>
<thead>
<tr>
<th>States (X)</th>
<th>Action</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;S₀, S₀&gt;</td>
<td>Intercept $M₀ \rightarrow M₁$? Ask (0.1.E(KU[1], N₀[1], 0))</td>
<td>$Ω^* = {E(KU[1], N₀[1], 0)}$</td>
</tr>
<tr>
<td>&lt;S₁, S₀&gt;</td>
<td>Inject $M₂ \rightarrow M₁$! Ask (2.1.E(KU[1], N₀[1], 0))</td>
<td>$T_{true} = {M₀?Rpl, M₀?Rst, M₁?Ask}$, $t = M₁?Ask$</td>
</tr>
<tr>
<td>&lt;S₁, S₁&gt;</td>
<td>Intercept $M₁ \rightarrow M₂$? Rpl (1.2.E(KU[2], N₀[1], N₁[2], 0))</td>
<td>$Ω^* = {N₀[1], N₁[2]}$ $N₀[1] \in M^*$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>States (Y)</th>
<th>Action</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;S₀, S₀&gt;</td>
<td>Inject $M₀ \rightarrow M₁$! Ask (0.1.E(KU[1], N₂[1], 0))</td>
<td>$T_{true} = {M₁?Ask}$ $t = M₁?Ask$</td>
</tr>
<tr>
<td>&lt;S₀, S₁&gt;</td>
<td>Intercept $M₁ \rightarrow M₀$? Rpl (1.0.E(KU[0], N₂[1], N₁[0], 1))</td>
<td>$Ω^* = {E(KU[0], N₂[1], N₁[0], 1)}$</td>
</tr>
<tr>
<td>&lt;S₀, S₁&gt;</td>
<td>Inject $M₀ \rightarrow M₁$! Cfm (0.1.E(KU[1], N₂[1], 0))</td>
<td>$T_{true} = {M₁?Rst, M₁?Cfm}$ $t = M₁?Cfm$</td>
</tr>
<tr>
<td>&lt;S₀, S₂&gt;</td>
<td>Intercept $M₀ \rightarrow M₂$? Ask (0.2.E(KU[2], N₀[2], 0))</td>
<td>$Ω^* = {N₀[2]}$</td>
</tr>
<tr>
<td>&lt;S₁, S₂&gt;</td>
<td>Inject $M₂ \rightarrow M₀$! Rpl (2.0.E(KU[0], N₂[1], N₁[0], 1))</td>
<td>$T_{true} = {M₀?Rpl}$ $t = M₀?Rpl$</td>
</tr>
<tr>
<td>&lt;S₁, S₂&gt;</td>
<td>Intercept $M₀ \rightarrow M₂$? Cfm (0.2.E(KU[2], N₁[0], 0))</td>
<td>$Ω^* = {N₁[0]}$ $N₁[0] \in M^*$</td>
</tr>
</tbody>
</table>

Table 3.1: Detection of errors in two implementations of N-S-L protocol.

The two sequences show both Implementation X and Implementation Y contain message confidentiality flaws.
3.3 MUTATION TESTING SCHEME

In this section we investigate mutation testing of security protocol, and design structured and preset test sequences. As introduced earlier mutation testing is a powerful technique for detecting specific types of security errors. Given the specification \( M_{\text{spec}} = \{ M_1, M_2, \ldots, M_C \} \), we introduce some faults, resulting in a mutant \( \{ M_1', M_2', \ldots, M_C' \} \). Given a set of mutants \( P \), a test suite is generated such that for each mutant \( p \), there is at least one test sequence that distinguishes (detects) it with the specification (correct implementation). A main challenge of mutation testing, when applied to software in general, is that the number of mutants (therefore the number of tests required) is huge. The situation is not mitigated in our EFSM model given its equivalent computing power of Turing machine. We model a security flaw as a mutation function \( \delta \) on a specification EFSM, and a type of fault \( F \) as a set of similar mutation functions. A mutant under \( F \) is the application of one or more such functions. If the type \( F \) contains \( k \) functions, then the number of mutants is \( O(2^k) \).

One can take two hypotheses to reduce the number of mutants generated [26]. First, competent programmer hypothesis assumes that an implementation only contains a small number \( (C) \) of faults. This reduces the number of mutants to \( O(k^C) \), which is still quite large. Second, coupling effect hypothesis states that the test sequences used to distinguish mutants with simple fault are sensitive enough to also uncover complex fault. Clearly this is not always true. Given an arbitrary mutation function, a test sequence that obtains the secret on \( \delta_i(M_{\text{spec}}) \) may not be effective for
\( \delta_2 \delta_1(M_{spec}) \). In fact, mutant \( \delta_2 \delta_1(M_{spec}) \) could even be secure. On the other hand, if we could select test sequence that satisfies this property, then the number of mutants could be further reduced to \( k \). For message confidentiality testing, we can reduce the number of mutants based on this observation.

3.3.1 A Fault Model: Predicate (Guard) Absence

There are generally two categories of security sensitive fault in the protocol model. The first is message format fault. For example, one might use the private key to encrypt part of the message instead of the public key, or attach an unnecessary part, both giving the intruder more information. This type is easier to observe since it changes the alphabet of some component machines. The second category of fault is related to the predicate or action of the transitions, but has no effect on the message types. Based on the observation of security protocols, a commonly encountered implementation error is neglecting critical condition checking. Usually an action is taken place only if some condition – predicate - is satisfied by the current state and/or the input message. For example in the N-S-L protocol, the responder only replies to the message \( \text{Ask}(x_1, x_2. E(k, n_1, x_3)) \) when the \( x_2 \) is equal to its own index, and similarly the initiator only generates to the Cfm message when it verifies the responder’s reply with the same nonce as the one it sends out. If the programmer neglects to check such condition such as in Implementation \( X \) and \( Y \) in Section 3.2, it is likely that the resulting implementation is insecure. This type of fault is reflected in the SP-EFSM model as the absence of part of the predicate in a transition - or often called a guard.
Assuming the predicate is specified as a conjunctive normal form of Boolean expressions (i.e. \( b_1 \& b_2 \& b_3 \)), we formally define this fault type.

Definition 3.2: For all the transitions \( t_j, j=0,1,\ldots \), from a state \( s \) with a same input/output symbol \( y \) of an SP-EFSM model, a predicate absence (PA) mutation operator \( \delta_{PA}(s,y,t,b) \) with regard to a Boolean expression \( b \) in the predicate \( p_j \) of \( t=t_i \), removes \( b \) from \( p_j \) and adding \((!p_i)\) to \( p_j \) for all \( i \neq j \).

Basically the mutation operator removes one Boolean expression from a transition. In order to keep the resulting machine deterministic, we add its negation to all other transitions with the same start state and input/output symbol. Figure 3.2 shows an example of a mutant obtained using the operator \( \delta_{PA}(S_1,Y, t, [a=1]) \). It is obvious that all operators with same states and I/O symbols are commutative.
Definition 3.3: For a protocol specification $M_{spec}$, a predicate absence (PA) fault type $F_{PA}$ is obtained by applying one or more PA mutation operators on $M_{spec}$. A mutant under $F_{PA}$ is defined as $\delta_S(M_{spec}) = \delta_1 \delta_2 \ldots \delta_n(M)$, where $S = \{\delta_1, \delta_2, \ldots, \delta_n\} \subseteq F_{PA}$, and for any $\delta_i(s, y, t_a, b_a), \delta_j(s, y, t_b, b_b) \in S$, $t_a = t_b$.

A mutant under the PA fault type is the result of application of a set of PA mutation operator, each removing a Boolean expression from a predicate. Note that although this definition does not limit the number of faults in one mutant, it relies on the competent programmer hypothesis to assume that for each combination of component machine, state and I/O symbol, only a predicate from one transition could be removed. Consequently, if each transition contains a constant number of Boolean
expressions, there are totally $O(T)$ mutation operators and $O(2^{C \times N \times P})$ mutants where $T$ is the number of transitions, $C$ is the number component machines, $N$ is the maximum number of states and $P$ is the number of I/O symbols. Intuitively a mutant with more predicate missing should allow more transitions to be executed and therefore the security flaws are “monotonically” increasing with inclusion of more faults in $F_{pa}$. This is formulated in the following proposition.

Definition 3.4: A progressive I/O sequence of a communicating system is an I/O sequence that does not trigger any “else” transition of any component machine.

Proposition 3.1: For any two mutants $\delta_{S1}(M)$ and $\delta_{S2}(M)$ under $F_{pa}$ with $S_1 \subseteq S_2$, if a progressive I/O sequence $seq$ could be generated by $M_I$ and $\delta_{S1}(M)$, then $seq$ could also be generated by $M_I$ and $\delta_{S2}(M)$.

Sketch of proof: The proof of this proposition is quite straightforward using an induction on the length of the sequence. Suppose a prefix of $seq$ has already been executed by $\delta_{S2}(M)$ and the next message in $seq$ will trigger transition $t$ in $M_I$, if $\delta_{S2}(M)$ has the same $t$ as $\delta_{S1}(M)$ then $t$ will be executed. If $\delta_{S2}(M)$ further removes some expressions from $t$, then the current states and input message will satisfy the guard of the new transition, since $t$ is not the “else” transition, and, therefore, $t$ is executed.

An important implication of Proposition 3.1 is that if a progressive test sequence discovers a message secret for $M$, and we apply some other mutation functions to introduce more errors, the same test sequence can still expose the message content on the new mutant. In other words, faults do not cancel the evidence of each other with regard to a progressive test sequence. We remark that singularity about
“else” transition does not decrease the applicability of this model because this special type of transition is usually used to model the behavior in abnormal conditions, and will not be included in an I/O sequence that achieves the functionalities of the protocol.

3.3.2 Mutation Test Generation Algorithm

Now we describe the procedure of generating test sequences for monotonic flaw type of $F_{PA}$. The goal is to generate a set of test sequence that distinguishes all mutants under $F_{PA}$. One valid concern would be that not all mutants are necessarily insecure according to the confidentiality requirement and it is reasonable to only focus on mutants, which lead to message confidentiality violations. This is a well-studied validation problem and we shall not digress here. For simplicity, we treat all mutants as potentially insecure and generate tests to detect each of them.

---

Algorithm 3.3 (Test generation for fault type $F_{PA}$)

*Input:* $M_{spec} = \{M_1, M_2, \ldots, M_C\}$, secrets $M'$.

*Output:* test suite $S$, fault type $F'_{PA}$

**begin**

1. $S = \{\}; F'_{PA} = \{\}$;

2. remove all “else” transitions from $M_{spec}$

3. calculate and minimize $M_1 \times M_{spec}$;

4. **foreach** mutation function $\delta_i$ **do**
5. calculate $\delta_i(M_{\text{spec}})$;

6. calculate and minimize $M_I \times \delta_i(M_{\text{spec}})$;

7. if $(M_I \times M_{\text{spec}} \neq M_I \times \delta_i(M_{\text{spec}}))$

8. $t =$ separating sequence of $M_I \times M_{\text{spec}}$ and $M_I \times \delta_i(M_{\text{spec}})$

9. $S = S + \{t\}$;

10. $F'_{PA} = F'_{PA} + \{\delta_i\}$;

11. return $S$

end

Algorithm 3.3 applies each mutation function alone to the specification and calculates a progressive separating sequence. This is done by removing all “else” transitions, minimizing the Cartesian product of the mutant and intruder machine, and calculate a separating sequence. The comparison in Line 7 refers to an equivalence test of two machines. The algorithm produces a new fault type $F'_{PA}$ which only contains the mutation functions if the corresponding mutants are distinguishable. The number of test sequences generated by Algorithm 3.3 is no more than the number of mutants in $F'_{PA}$. The time needed for minimization is $O(N\log N)$ with online minimization algorithm [13], and the calculation of separating sequence requires $O(N^2)$ where $N$ is the number of states in the reduced machine. We propose an optimization technique for generating separating sequence online in [91], which will reduce the cost of this algorithm for average case but the worst case complexity is the same.

As far as the fault detection capability is concerned, the test suite generated includes a test case to distinguish every mutant that is derived by applying one
mutation function in $F'_{PA}$. Since all test sequences are progressive sequence, from Proposition 3.1, we have:

Proposition 3.2: Tests generated from Algorithm 3.3 detect all mutants under $F'_{PA}$ in time $O(N^2)$ where $N$ is the number of states in the reduced machine.

Algorithm 3.3 also applies to all other fault models that satisfy Proposition 3.1. Note that the test suite does not discover all faulty mutants in $F_{PA}$; if a mutation function itself is not distinguishable, then Algorithm 3 simply discards it.

3.3.3 Experiments with N-S-L Protocol

We again conduct the experiment using N-S-L protocol. In the specification (Figure 2.3) a total of 19 Boolean expressions are identified, as shown in Figure 3.3. These expressions are used to construct the fault type $F_{PA}$ and the mutants. Among them $\delta_{b12}$ and $\delta_{b7}$ correspond to the three implementations $X$ and $Y$ in Section 3.2.3, respectively. Algorithm 3.3 produces $F'_{PA} = F_{PA} - \{\delta_{b18}, \delta_{b19}\}$ and the set of 17 test sequences. The last two Boolean expressions are not associated with any I/O behaviors and are not observable. The lengths of those sequences are shown in Table 3.2 and the details are omitted. All the sequences are short (less than 4). This set of test sequences detect all implementations with one or more Boolean expressions missing.
Figure 3.3: Boolean expressions in N-S-L protocol specification

Table 3.2: Test sequence lengths generated by Algorithm 3.3
3.4 LEARNING BASED TESTING APPROACH

Both the guided random walk algorithm and the mutation testing algorithm are heuristic in the sense that there is no guarantee of discovering the implementation. In this section we present a significantly more complicated but more powerful learning-based testing approach. By using supervised machine learning techniques we derive a systematic algorithm that eventually obtains the model of implementation and therefore could verify the security properties.

3.4.1 Learning Based Testing Methodology

Given protocol \{M_1, M_2, \ldots, M_C\} and an acceptable implementation \{B_1, B_2, \ldots, B_C\}, we want to test violation of message confidentiality of the latter. However, in practice not all the implementations need to be involved in testing. Specifically, if the components are developed by different organizations or have different credibility, we only want to focus on some of them. Assume that the first \(k\) components are under test and we compose a new system using \(B_1\) to \(B_k\) and the specification, i.e. \(\{B_1, B_2, \ldots, B_k, M_{k+1}, \ldots, M_C\}\). Obviously, if this new implementation is insecure, then so is the original implementation due to the trace inclusion property.
The essence of testing a component is to learn the structure of its reachability graph by black-box testing experiments, which again is an equivalent expanded representation of a SP-EFSM model. Following the theoretical insights of [77], we employ the following procedure shown in Figure 3.4. For each black-box component \( B_i \) under test, an estimation of it (\( B^*_i \)) is maintained. \( B^*_i \) is represented as a FSM and initialized as an empty FSM. These estimations are updated every iteration according to classic algorithm proposed by [10] and ideally they will converge to the target implementation \( B_i \). In section 3.1.3 we present modification to the original learning algorithm \( L^* \) (for automata) to fit our need. Once a new estimation is formed, we will run a validation algorithm to verify message confidentiality, as in definition 5. The algorithm calculates the composed reachability graph of \( \{ B^*_1, B^*_2, \ldots, B^*_k, M_{k+1}, \ldots, M_C \} \) and search for a state of security violation.
**Case 1.** If no violation could be found, we submit the estimation to the teacher of learning algorithm. The teacher is simulated by conformance test generator and executor, which is responsible of answering the question of whether the current conjecture (estimation) is identical to the target black-box. In the case of a negative answer, the teacher will provide a counter-example - in our case a test sequence, to contribute to the next estimation. If the answer is positive, the test process terminates with result “PASS”.

**Case 2.** On the other hand, the validation algorithm may also find a message trace leading to violation. Due to the nature of the learning algorithm, the trace found may not exist in the implementation. In other words, it is a false positive introduced by the estimation. In order to filter those bogus traces, we simply experiment the trace on the black-box. If the trace is verified, we terminate the testing procedure and claim “FAIL”, otherwise the trace is used as a counter-example for at least one component just like the one provided by the teacher.

Finally as shown later in this section, if the procedure finishes with a “PASS”, it does not necessarily mean the last estimation is identical to the implementation, instead it means we can no longer find the discrepancy or we choose not to do so. In this architecture two issues are most important and we elaborated in the following subsections. First, we study our variation of $L^*$ learning algorithm; then we focus on how to simulate the teacher using formal testing techniques.
3.4.2 FSM Learning Algorithm $L_{fsm}^*$

$L^*$ is the classic supervised automata learning algorithm proposed by [10]. Here we modify it to accommodate our finite state machine model. The new algorithm $L_{fsm}^*$ assumes the presence of a teacher that can answer three types of queries. (1) **Output Query**: this corresponds to “membership query” in $L^*$. Given an input sequence, the teacher gives the corresponding output sequence produced by the system. (2) **Counter Example**: the teacher generates an input sequence for which the conjecture produces the incorrect output. (3) **Alphabet Augment**: the teacher gives a set of input symbols that are not presently included in the conjecture. This could be seen as a particular type of counter-example. Note that in $L^*$ the learner starts with knowing the complete input alphabet, however this assumption is not practical in our protocol model because the alphabet usually has a formidable size (number of messages in $L$). Our algorithm will start with any subset (usually those included in the specification) and then gradually learn new inputs from the teacher.

The implementation of $L_{fsm}^*$ is very similar to $L^*$. A data structure called observation table is maintained, where each row represents a distinguished state and each column represents a separating sequence of the current conjecture. The header of a row records set of input sequences that can lead the machine to the state while the content of a row stores the output from a state upon each of the separating sequence. The differences between $L_{fsm}^*$ and $L^*$ are summarized as follows. First, the content of an element in the observation table is a Boolean value in $L^*$ (for automata) and a finite output sequence instead here for FSM. Second, $L_{fsm}^*$ starts with only part of the input alphabet and takes new input as a special type of counter-example. Upon receiving a
new input symbol from the teacher, the algorithm adds it as a new column, and furthermore appends it to each row to create a new “next-state” row. Figure 3.5 shows two consecutive guesses made by the algorithm and the corresponding observation table. Figure 3.5 (a) depicts the initial guess with 2 states and only one input. The teacher replies with a new input symbol “B” and a new conjecture is calculated as shown in Figure 3.5 (b). After submitting this guess to the teacher, a counter example \([A,B,B,B]\) is returned. Then the observation table is updated again and a new guess Figure 3.5(c) is produced with a new state learned.

Using the same proof technique as [3] we can prove that for a target minimized FSM with \(P\) input symbols and at most \(N\) states our algorithm makes at most \(P+N-1\) incorrect conjectures before it eventually learns the target. As we shall see in next section, the effectiveness and cost of learning algorithm depends significantly on how the teacher is implemented, particularly how it generates counter examples and alphabet augments.
Figure 3.5: Example of using $L^*_{fsm}$ to learn a three-stated FSM
3.4.3 Simulating the Teacher in $L^*_{sim}$

This section discusses the implementation of teacher in our framework. Given the black-box $B_i$, the SP-EFSM specification $M_i$ with protocol language $L$, the teacher replies to three types of queries listed in Section 3.4.2. Among them output query is the easiest one: for an input sequence, we send it to $B_i$ and reply with the observed output. Dealing with conjectures is much more complicated, where essentially black-box testing technique has to be used [55]. For the current guess $B^*_i$, the teacher tries to tell the difference between it and the target $B_i$. While previous literature has addressed this problem in general, we elaborate and evaluate a specific algorithm (Algorithm 3.4) here.

First we want to test whether $B_i$ conforms to $B^*_i$ by using a conformance testing suites. Let $P_i$ and $N_i$ be the number inputs and states $B^*_i$ has, respectively. Using the classic checking sequence algorithm we calculate (line 1) a set of input sequences of cardinality $O(P_i \cdot N_i^2)$ and total length $O(P_i \cdot N_i^3)$. Each sequence contains two sub-sequences, one transferring the machine to a certain state and the other verifying that state. The time complexity of this algorithm is also $O(P_i \cdot N_i^3)$. We select a subset of the test suites (line 2) and execute it on $B_i$, and if any test case fails it is used as a counter example.

If $B_i$ passes all checking sequences, it is either identical to the guess or contain more states. Unfortunately the cost of probing extra states is exponential in the number of states, which is due to the hardness of traversing the machine to find the new state. To this end, we generate probing sequences of form: $Seq_x I_x I_y \ldots I_y Seq_y$, where the
head and tail component $Seq_x Seq_y$ come from conformance test sequence just calculated, and the middle is a sequence of random input symbols of $B^\ast_i$ of length $\Delta$ (lines 7-8). This type of probing helps us learn up to $\Delta$ extra states, and obviously we only afford to try a small number ($T$, line 5) of all sequences.

Last but not least, the conjecture could be incorrect because it has missed some inputs. In our protocol model not all the messages in $L$ are necessarily used as input in the specification. In fact specification of real-world protocol is usually very partially defined on inputs, and a large number of unspecified (informally referred to as ill-formatted, abnormal) inputs will not make the machine take any transition or output anything. Due to errors an implementation will have more inputs enabled that we would like to explore. Naturally, the first guess is made by using only those input symbols appeared in the original specification, and at teach iteration the teacher chooses a set of unexplored symbols (lines 10-14) as reply to the learning algorithm.
Algorithm 3.4: (Counter-example for learning algorithm $L_{fsm}^*$)

**Input:** Conjecture FSM $B^*_i$, target $B_i$, protocol message language $L, \Delta$

**Parameters:**
- $0 \leq P_1 \leq 1$: Fraction of checking sequences to apply
- $0 \leq P_2 \leq 1$: Fraction of new input symbols learned at each iteration
- $1 \leq T \leq |L|^\Delta$: Number of state probing sequences to use

**Output:** Counter example or new input set or null (correct guess)

```
begin
1. \{ Seq_{1a}Seq_{1b}, Seq_{2a} Seq_{1b},...,Seq_{K_a} Seq_{K_b} \} = checking sequences of $B^*_i$;
2. Select $K\cdot P_1$ sequences randomly;
3. \textbf{foreach} selected sequence $Seq_{xa}Seq_{xb}$ \textbf{do}
4. \textbf{if} $output(B^*_i, \text{Seq}_{xa}Seq_{xb}) \neq output(B_i, \text{Seq}_{xa}Seq_{xb})$ \textbf{return} $Seq_{xa}Seq_{xb}$;
5. \textbf{do} $T$ times
6. generate random sequence $I_1I_2...I_\Delta$;
7. select a random checking sequence $Seq_{xa}Seq_{xb}$;
8. \textbf{if} $output(B^*_i, \text{Seq}_{xa} I_1I_2...I_\Delta Seq_{xb}) \neq output(B_i, \text{Seq}_{xa} I_1I_2...I_\Delta Seq_{xb})$
   \textbf{return} $Seq_{xa}Seq_{xb}$;
9. $L' = L - I(B^*_i)$; $S = \{ \}$;
10. \textbf{foreach} $I_x$ in $L'$ \textbf{do} with probability $P_2 \frac{|L|}{|L'|}$ do $S=S+\{I_x\}$;
11. \textbf{if} ($S \neq \{ \}$) \textbf{return} $S$;
12. \textbf{return} null;
end
```
The complexity of the algorithm is straightforward to analyze. For conformance test sequences each of length $O(N^*)$, a total number of $O(P_1 \cdot K) = O(P_1 \cdot P^* \cdot N^{*2})$ will be calculated and executed, where $N^*$ is the number of the states in $B^*_1$ and $P^*$ is the size of its alphabet. Similarly $O(T)$ probing sequences and $O(P_2 \cdot L)$ new inputs will be processed. Hence the total cost of the teacher in one iteration is $O(P_1 \cdot P^* \cdot N^{*3} + T \cdot N^* + P_2 \cdot P)$. On the other hand, each counterexample it generates has length $O(N^*)$. Note that at line 1 we do not actually produce any test sequence; instead it is postponed until it is decided which one to execute. On the other hand, in every round $L^*_fsm$ algorithm takes time $O(P^* \cdot N^{*2})$ to update the observation table and compute the next conjecture. Since there are at most $O(N+1/P_2) = O(N)$ iterations, the total cost of the learning procedure for one protocol component is at most $O(P_1 \cdot P \cdot N^4 + T \cdot N^2 + P \cdot N^3 + N \cdot f_{ver}(N))$ where $f_{ver}(n)$ is the cost of validating the security for a conjecture with $n$ states.

### 3.4.4 Evaluation of the Proposed Approach

We conduct two types of experiments using a prototype tool that incorporates all algorithms proposed, including the message confidentiality validation algorithm, the learning algorithm $L^*_fsm$, and Algorithm 3.4 for the teacher. We apply the methodology to three well-known security protocols. We also simulate a large number of random protocol component models and show how various choices of the learning strategy affects its outcome.
We select three well known protocols: Needham-Schroeder-Lowe (N-S-L) mutual authentication protocol, TMN key exchange protocol and SSL 3.0 handshake protocol. For each of them one component is chosen to be tested and an implementation is used as the target black-box. In N-S-L protocol implementation we introduce a flaw first discovered by [59] in the initiator component; in TMN we use the specification of client as the implementation, which itself is insecure [60]; in SSL we use one open source implementation of the client, and use the Sun Java Development Kit (JDK) 1.3 server implementation as specification. From manual inspection of the client source code, we observe the discrepancy between it and the specification, however, both our test approach and human reasoning come to the conclusion that such discrepancy does not lead to security flaws. Detail of the three implementations is summarized in Table 3.3.

| Protocol | # of state in spec. FSM | # of msg in spec. $(|L_0|/|L|)$ | # of input in spec. | Spec. is secure | Impl. is secure |
|----------|-------------------------|---------------------------------|---------------------|-----------------|-----------------|
| N-S-L $M_{init}$ | 18 | 4/978 | 221 | Yes | No |
| TMN $M_{client}$ | 45 | 6/6665 | 287 | No | No |
| SSL $M_{client}$ | 137 | 7/1575 | 529 | Yes | Yes |

Table 3.3: Three protocols used in experiments (a) N-S-L (b) TMN (c) SSL 3.0
(a) N-S-L Protocol

(b) TMN Protocol
Figure 3.6: Result of testing (a) N-S-L (b) TMN (c) SSL protocol implementations
For each protocol implementation, we run our testing algorithm three times. The results are depicted as Figure 3.6. We record the information of each conjecture our algorithm makes: number of state (X-axis) and number of input (Y-axis). Each point represents one conjecture and the big one on top-right corner is the target implementation. For the first run, we start the learning with a set of normal inputs ($L_0$ in Table 2). We use parameters $P_1=1$, $T=1024$, $P_2=0.25$, $\Delta=16$. For all three protocols the target is successfully learned, though the message confidentiality validation result is different: for N-S-L at the 5th conjecture violation is identified and confirmed; for TMN all conjectures are insecure (since the specification itself is flawed); and for SSL the implementation is confirmed secure. We also run the algorithm with other two different configurations. Since the total number of input is large, we run the algorithm once with $P_2=0.1$ and only try up to 75% of the total input. Not surprisingly the result shows that missing inputs could be critical, preventing us from learning all the states (hence we may possibly miss the security violation). On the other hand, instead of starting with normal inputs, we try random initial input set (with $P_2=1$, $T=1024$ and $P_2=0.1$). For both N-S-L and TMN the algorithm finishes with the correct conjecture, while for SSL it finishes with a 2-stated wrong conjecture. This particular case shows the importance with initial input set. A reasonable selection (e.g. from protocol specification) helps the learning algorithm identify the basic structure of the black-box and some critical states, which otherwise can only be discovered by extensive and expensive probing for counter examples and may never succeed. Overall the
experiments on those three real protocols justify that proposed learning-based testing methodology is effective.

In order to better demonstrate the performance of our testing algorithm with different parameters, we also conduct experiments on a large pool of randomly created protocol components represented by FSM. Each FSM is generated independently with a specified range of size of input/output alphabet and state. First we want to see the role of input alphabet in the algorithm. We use a pool of 1000 FSM with 32-128 states, 16-64 inputs and 16-32 outputs. First we vary the probability that an input message is included in the initial conjecture – $P_{init}$, and the portion of unknown input added for each new iteration – $P_2$. The result is shown by Figure 3.7(a). It is clear that adding more input decreases the number of total conjectures needed (but the cost of each one is increased), and on the other hand starting with more input always lead to less conjectures. These two parameters mainly affect the speed of learning, and they have almost no impact on the accuracy. The learning accuracy is defined as the percentage of states identified by our algorithm when it terminates. Factors deciding accuracy are the total number of input messages learned and the number of input sequence probed by the teacher in pursuit of a counter-example for each conjecture. For the same pool of random machine the trend is shown in Figure 3.7 (b). We try different upper bound of the percentage of total input learned (from 20% to 100%), and select different parameter of $P_i$ (fraction of checking sequences used) and $T$ (number of extra state probing sequences used). Results show that using more checking sequences improve the success ratio significantly, by warranting more counter-examples to be found. Also, the improvement from probing a small number of
sequences, e.g. 1024, is rather marginal, since the total number of probing sequences is huge (exponential of the number of extra states). Note that we do not always need to achieve 100\% state accuracy in practice, if a security violation could be confirmed from an earlier conjecture.
(a) Effect of initial input and input addition ($P_2$) on number of guesses. $P_I=0.1$, $T=128$, 
$\Delta =16$

(b): Effect of max learned input and probing sequences ($P_I$) on learning success ratio. 

$P_2=0.5,P_{init}=0.2$, $\Delta =16$
(a) Average number of guesses for target of different size. \( P_{ini}=0.2, P_I=0.5, \)
\[ T=128, P_2=0.25, \Delta =16 \]

(b) Total learning time for FSM of different scale. \( P_{ini}=0.2, P_I=1, T=128, \)
\[ P_2=0.25, \Delta =16 \]

Figure 3.7: Experiments with the learning based testing approach
Finally, we change the scale of the target by both number states and inputs, and for each combination we run the algorithm on 100 randomly generated FSM. Figure 3.7 (c) depicts the result and we could see that given fixed parameters, the number of guesses is quite stable when the size of target grows large. Figure 3.7 (d) shows the total cost of our testing algorithm running on a single Pentium 4 3.8GHz computer with 2GB memory. It is clear that total cost grows in linear fashion with number input increasing (note that both axis are of logarithmic scale), and in fact it is also in polynomial of the number of states. This result is consistent with our formal analysis in Section 3.4.3. The running time for the largest simulated FSM is about 10 minutes.

3.5 SUMMARY

This chapter studies the problem of testing security property using the proposed SP-EFSM model. Message confidential property under Dolev-Yao’s attacker assumption is used as an example. A series of algorithms, from simple to complicated, are proposed. First we describe a naïve passive algorithm, followed by a guided random walk algorithm that attempts to cover the transitions of SP-EFSM fairly. Next a mutation testing algorithm is proposed. This algorithm uses a heuristic fault model called predicate absence and is capable of discovering security flaws caused by missing the validity checking of input messages. Finally we proposed a learning-based procedure to explore the implementation structure that is beyond the specification. Inspired by the black-box automata model checking theory we use a supervised learning approach with a conformance test generator serving as the teacher. For a
black-box protocol implementation under test, we maintain an estimation (conjecture) model of its structure and update it successively as more is learned by applying testing sequences. Every conjecture is closer to the implementation than the previous one and after each conjecture is constructed a validation process is applied to check against message confidentiality property. The process terminates when no more new behaviors could be learned or a security violation is detected. This methodology is implemented in a software tool that can execute tests automatically. Both N-S-L protocol and randomized FSM models are used in the experimental study and the results show that the proposed approaches are effective.
CHAPTER 4

NETWORK PROTOCOL FINGERPRINTING

4.1 BACKGROUND

This chapter studies the problem of network protocol system fingerprinting and presents a formal methodology based on SP-EFSM model. Fingerprinting problem is closely related to protocol testing problem, while the major difference is that in the former there is no unique correct implementation. With its importance well appreciated, the current state of understanding and practice are both limited. Existing approaches rely largely on expert knowledge of protocols, thus could hardly be automated and generalized. Technically, it is prevalent to encode a fingerprint using a set of traces collected from the implementation, which is neither accurate nor complete. First, the accuracy is jeopardized by the fact that traces contain parameters (e.g. TCP initial window size, SSH certificate, etc.) that differ from run to run, and a trace literally recorded with a specific parameter value is likely to be unmatched with another trace from the same implementation. Second, although there could be infinitely many traces that distinguish an implementation, only a limited number of them could be stored. Consequence, the quality of fingerprints relies on the selection of traces.
We propose a formal methodology for protocol fingerprinting. The principal is inspired by examining the fingerprinting technology for human being, which has been mature for decades. Instead of storing the image taken from a scanning device, a set of key locations – each called a “minutia” is identified and encoded. These minutiae combined uniquely identify the fingerprint and thus could be used to match against new fingers. Similarly in our methodology, we model the “minutiae” of a protocol component implementation by an SP-EFSM specifying its key state transitions with I/O messages. Formally speaking, a Minutiae model is a partially defined SP-EFSM that records the distinguishing structural aspects of an implementation with the help of the rich transition semantics. This model is strictly more expressive than the existing trace model since all valid traces could be reproduced using the process of reachability analysis. Based on the model a complete taxonomy of fingerprint matching and discovery problems is provided. Then I present efficient algorithms for each problem and analyze theirs complexity. Finally some evaluation results are reported.

4.2 A TAXONOMY OF FINGERPRINTING PROBLEMS

We focus on black-box fingerprinting, meaning an implementation could only be inspected by its I/O trace (i.e. internal information such as source code is not available). Also we distinguish active experiments from passive experiments [51]. In an active experiment we are allowed to input anything to the implementation while in the case of passive experiment a trace is given - from monitoring - for analysis and we can not try more inputs.
There are two categories of basic operations. The first is \textit{fingerprint matching}, formally defined as follows. Given an implementation $M_{impl}$ and a fingerprint model $M_f$, we would like to verify whether they match. Define that $M_{impl}$ and $M_f$ match if all traces from $M_f$ could be produced by $M_{impl}$. Alternatively, sometimes we want to match $M_{impl}$ with a set of fingerprints (e.g. from a database). In this case it is desirable to have a faster solution than matching with each fingerprint one by one for every implementation. The second operation is \textit{fingerprint discovery}. For a new implementation, we systematically discover its fingerprint by retrieving as much information as possible. Discovery could be done with or without the guidance of protocol specification. If a specification model could be obtained as a nondeterministic SP-EFSM, we discover a deterministic sub-machine of the specification and extract the fingerprint from this sub-machine. On the other hand, if no specification is available, the fingerprint will be obtained by a machine learning process on the reachability graph. To summarize, we present a first taxonomy for protocol fingerprinting problem. Table 4.1 lists the categorization of the problems and for each one we show the theoretical results that could be applied.
4.3  FINGERPRINTING ALGORITHMS

4.3.1 Fingerprint Matching

Matching a black-box implementation with a fingerprint model (reachability graph) is a classic partial FSM testing problem and it is well studied in the literature. For active experiment, both deterministic algorithm and probabilistic algorithm have been developed. Specifically, let $N$ be the number of states in the reachability graph, $P$ be the number of input packets; we have a deterministic algorithm that selects an input sequence of worst case length $O(P \cdot N^3)$. Alternatively, we could use a sequence of length $2 \cdot P \cdot D \cdot z \cdot \log(1/\varepsilon)$ to detect discrepancy with success probability at least $(1-\varepsilon)$ [55], where $D$ is the maximal length of the input sequence that separates two states in
the FSM, and \( z \) is the maximal number of sequences needed to separate one state from all others. The worst case result of this probabilistic algorithm is the same as deterministic algorithm, however in many cases the resulting sequence could be as short as \( O(P \cdot N) \). Instead of comprehensive matching, we could also apply less complete test sequences (e.g. a transition tour in the original fingerprint model) in exchange for faster operation.

In case of passive matching, we could apply passive testing to either the SP-EFSM model or the reachability graph: a set of possible states is initialized as all states at beginning, and we consume the given trace by inspecting one packet at a time. After each packet the set of states is updated, and when it becomes empty it could be concluded that the fingerprint does not match the implementation.

Table 4.2 shows an unsuccessful matching process with a trace containing three I/O packet pairs. The complexity of this process is \( O(L \cdot P \cdot N) \) where \( L \) is the length of the I/O trace.
<table>
<thead>
<tr>
<th>Trace (I/O)</th>
<th>Current States (x, y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(1)/B(0,1)</td>
<td>{ (x = 1, y = 1) }</td>
</tr>
<tr>
<td>A(0)/B(1,0)</td>
<td>{ (x = 1, y = 0) }</td>
</tr>
<tr>
<td>A(1)/B(0,1)</td>
<td>{ }, fingerprint mismatch detected</td>
</tr>
</tbody>
</table>

Table 4.2: Passive matching using the fingerprint model from Figure 2.2

4.3.2 Fingerprint Group Matching

Group matching is a slightly more intricate problem. Unlike matching with each single fingerprint, in active setting we pre-calculate a set of sequences that could separate all candidate fingerprints and therefore eliminate some unmatched ones quickly.

Given a candidate group of fingerprints, i.e. $C = \{ M_1, M_2, ..., M_k \}$, a test sequence $seq \in \mathcal{I}^*$ separates $M_i$ and $M_j$ if $f_{\text{output}}(M_i, seq) \neq f_{\text{output}}(M_j, seq)$. A fingerprinting set $F$ for a candidate group $C$ is a set of test sequences, such that for each pair of machines in $C$, $F$ contains a sequence that separates them. The goal of active group matching is to calculate $F$ efficiently. Since each fingerprint is partially defined, a sequence separating two of them may not even be defined for another. This fact complicates the problem and makes the algorithm significantly more costly. For simplicity we could assume each pair of candidates could be separated by an input sequence, in which case an implementation can always be matched with a unique
fingerprint model. A trivial solution for fingerprinting set $F$ is the set of all pair-wise separating sequences. This set has cardinality $O(k^2)$. The heuristic algorithm below tries to find a smaller set by refining the set gradually until all candidates are separated.
Algorithm 4.1: (Fingerprinting Set for Candidate Group)

*Input:* candidate group \( C = \{M_1, M_2, \ldots, M_k\} \);

*Output:* fingerprint set \( F \);

*begin*

1. \( F := \{\} \);

2. \( \text{partition} := \{\{1,2,\ldots,k\}\} \);

3. \textbf{while} (\text{partition}.size < k)

4. find machine \( M_i \) and \( M_j \) in the same set;

5. calculate separating sequence \( SEQ \) for \( M_i \) and \( M_j \);

6. \textbf{foreach} set \( S \) in \( \text{partition} \) \textbf{do}

7. \( \text{undef} = \{\} \); remove \( S \) from \( \text{partition} \);

8. initialize \( \text{map} \) as an empty map from \( L(A)^* \) to \( 2^S \);

9. \textbf{foreach} \( x \) in \( S \) \textbf{do}

10. \textbf{if} \( SEQ \) is defined for \( M_x \)

11. \( \text{map}(f_{\text{output}}(M_x,SEQ)) \cup = \{x\} \);

12. \textbf{else} \( \text{undef} = \text{undef} \cup \{x\} \);

13. \textbf{foreach} \( s^* \) in \( \text{map} \) \textbf{do}

14. \textbf{if} \( s^* \cup \text{undef} \notin \text{partition} \)

15. \( \text{partition} = \text{partition} \cup s^* \cup \text{undef} \);

16. \( F := F \cup \{SEQ\} \);

17. \textbf{return} \( F \);

*end*
The worst case execution time of this algorithm is exponential: although there is at most $k^2$ separating sequences calculated, the partition (which is technically not a partition of the candidate group but rather a subset of its power set) could contain $2^k$ sets due to the incompleteness of each SP-EFSM. An extreme case is that all candidates are completely defined SP-EFSM. In this case the algorithm strictly refines the partition and the fingerprinting set contains at most $k$ sequences.

As for separating two candidate SP-EFSMs, a straightforward way is to generate their minimized reachability graph first and compute a separating sequence using classical algorithms for FSM. A smarter way is to expand the extended machines little by little and find the separating sequence online, therefore avoiding the cost of reachability analysis of all candidates. I studied the algorithm based on online minimization with cost $O(k \cdot P \cdot N^2)$ and $O(k \cdot P \cdot N^4)$ for complete and partial fingerprint models respectively.

This algorithm is applied to a database of Nmap fingerprints for TCP. Nmap [106] is the most popular active OS fingerprinting tool. It identifies a TCP stack implementation by using nine test sequences: $T_{seq}$ is for TCP initial sequence number prediction; $T_1$ to $T_7$ are seven specially constructed TCP packets; and $PU$ is for probing unreachable port. In the fingerprint database Nmap stores the encoded response to those test sequences of more than 1300 implementations. Those implementations are classified into 33 categories, spanning from general purpose Operating Systems to VoIP phones. In this section we extract one candidate group for each category from the fingerprint database and calculate the fingerprint set. Note that the set of all nine tests forms a fingerprint set of most implementations but for some
category not all tests are needed. Technically, the test $T_i$ to $T_7$ and $PU$ are quite different from $T_{seq}$ since the design of $T_{seq}$ and encoding scheme of its output are significantly more involved. For simplicity we still treat $T_{seq}$ as a single “virtual” input symbol. Given this assumption, the modeling of candidate machines is very concise. In fact, they are all FSM. Figure 4.1 and Figure 4.2 show the model of Solaris 9.0 and CISCO IOS 11.0 operating system respectively.
Figure 4.1: Nmap candidate machine model of Solaris 9.0

Figure 4.2: Nmap candidate machine model of Cisco IOS 11.0
Models of Nmap candidate machines have the property that if separating sequence of $M_i$ and $M_j$ exists, then it has length 1. This is due to the large number of different outputs for the test inputs. Devising such inputs certainly requires insight and intelligence, and once those powerful input symbols are known both fingerprinting experiments and potential defense scheme become straightforward. For example, it is obvious to see from Figure 4.1 and Figure 4.2 that all inputs except $T_3$ could be used as separating sequence for the two machines. We use Algorithm 4.1 to calculate fingerprinting sets for the largest 10 categories and show the result below in Table 4.3. In general, larger candidate group requires more tests. Two of the categories do not have an exact fingerprint set (shown by * in the last column) since they include implementations with only minor version differences which will not be distinguished by any Nmap test.
<table>
<thead>
<tr>
<th>Category</th>
<th>Size</th>
<th>Fingerprinting Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Purpose</td>
<td>651</td>
<td>({Tseq,T1-T7,PU})*</td>
</tr>
<tr>
<td>Broadband Router</td>
<td>112</td>
<td>({Tseq,T1,T2,T3,PU})</td>
</tr>
<tr>
<td>Router</td>
<td>105</td>
<td>({Tseq,T1,T2,T3,PU})</td>
</tr>
<tr>
<td>Printer</td>
<td>73</td>
<td>({Tseq,T1-T7,PU})*</td>
</tr>
<tr>
<td>Firewall</td>
<td>61</td>
<td>({Tseq,T1,T2,T3,T4,PU})</td>
</tr>
<tr>
<td>Switch</td>
<td>48</td>
<td>({Tseq,T1,T2, PU})</td>
</tr>
<tr>
<td>Terminal Server</td>
<td>43</td>
<td>({Tseq,T1,PU})</td>
</tr>
<tr>
<td>WAP</td>
<td>34</td>
<td>({Tseq,T1,PU})</td>
</tr>
<tr>
<td>Print Server</td>
<td>27</td>
<td>({Tseq,T1,PU})</td>
</tr>
<tr>
<td>Webcam</td>
<td>14</td>
<td>({Tseq,PU})</td>
</tr>
</tbody>
</table>

Table 4.3: Fingerprinting sets from Nmap

In passive fingerprinting we have candidate group \(C\) and a trace \(T\). The trace contains a vector of input symbol and a vector of output symbol. Since in the SP-EFSM model we assume each transition only generates one output, the vectors of input and output have the same length. The goal of passive fingerprinting experiment is to decide the fingerprint machine that could possibly generate \(T\). If \(T\) could only be the trace from one machine we successfully identify the fingerprint. To achieve this goal, we follow a passive testing paradigm. We scan the trace only once and conduct testing with all the candidate machines concurrently; whenever all except but one machine are eliminated, we can terminate the algorithm. The key structure needed to maintain during this process is the state uncertainty for each machine. State
uncertainty represents the knowledge we have about the current state of a machine $M$. If $M$ is deterministic, then the size of uncertainty will not increase. There are many different ways to maintain state uncertainty. Here we calculate the reachability graph for each machine first and maintain the set of states in the resulting finite state machine (corresponding to set of configurations in the original SP-EFSM). This is feasible if we can obtain the finite reachability graph, but not very efficient when the graph is large. The technique of [51] could be used to maintain uncertainty in EFSM.

Our passive matching Algorithm 4.2 works as follows. First we calculate the minimal reachability graph of each machine $M_i$. Then we inspect each pair of I/O symbol starting from the first one in the trace. Initially assume that all machines could possibly be the implementation. The state uncertainty is maintained for each $M_i$, initialized by all its states (line 3). For each I/O pair passively observed, we update each uncertainty by applying the transition function to the current uncertainty. Particularly, we are interested in the event that the state uncertainty of a machine becomes empty (line 11), which implies that this fingerprint does not match otherwise there is a contradiction. When the whole trace is processed, the remaining candidates are the possible implementation.
Algorithm 4.2 (Passive fingerprint matching)

*Input*: candidate group \( C = \{ M_1, M_2, \ldots, M_k \} \), \( T = \langle <I_0, O_0>, <I_1, O_1>, \ldots, <I_{L-1}, O_{L-1}> \rangle \);

*Output*: possible candidate set \( PC \);

begin

1. \( PC := C \);
2. foreach \( i \) in \([1..k]\) do
3. \hspace{1em} calculate minimized reachability graph \( G_i \); \( uncert[i] := \) all states in \( G_i \);
4. \hspace{1em} \( id = 0 \);
5. while \( (PC.size > 1 \ \text{and} \ \ id < L) \) do
6. \hspace{1em} foreach \( M_i \) in \( PC \) do
7. \hspace{2em} \( new\_uncert := \{ \} \);
8. \hspace{1em} foreach \( t <S_{src}, S_{dst}, I_t, O_t> \) in \( G_i \) do
9. \hspace{2em} \hspace{1em} if \( (S_{src} \in uncert[i] \ \text{and} \ I_{id} == I_t \ \text{and} \ O_{id} == O_t) \) then
10. \hspace{2em} \hspace{2em} \hspace{1em} \( new\_uncert = new\_uncert \cup S_{dst} \);
11. \hspace{2em} \hspace{1em} \hspace{1em} \hspace{1em} else
12. \hspace{2em} \hspace{2em} \hspace{1em} \hspace{1em} \hspace{1em} \( PC := PC - \{ M_i \} \); else
13. \hspace{2em} \hspace{2em} uncert[i] := new_uncert;
14. \hspace{2em} id := id + 1;

return \( PC \);

end
Now we analyze the complexity of this algorithm. Lines 5-12 contain a loop that takes one I/O pair at each step. Lines 6-11 inspect each candidate machine and update the state uncertainty. If the size of a reachability graph is no more than $N$, then Lines 6-11 terminate in time $O(k \cdot N^s \cdot P)$ because the number of transitions that must be considered is less than $N^s \cdot P$. Since the trace contains $L$ packets and in the worst case they all need to be inspected, the cost of loop 5-12 is $O(L \cdot k \cdot N^s \cdot P)$. As mentioned in the previous section, the cost of calculating minimized reachability graph is $O(N^{s2} \cdot P)$, therefore the total cost of the algorithm is $O(L \cdot k \cdot N^s \cdot P + k \cdot N^{s2} \cdot P)$. When the algorithm terminates, the resulting candidate set may contain more than one machine. In this case we can not identify a single implementation; instead we eliminate some impossible ones. On the other hand, if the algorithm terminates with only one candidate, then it is the implementation, and we can also construct a fingerprinting sequence starting from the initial state by back-tracking the uncertainties.

Below we present the experiment on matching fingerprints of TCP congestion control algorithms. This is technically a much harder problem than OS fingerprinting using special TCP and ICMP packets as both the model and the test sequences are much more involved. Congestion control schemes implemented in the production TCP stacks include standard TCP Tahoe, Reno, NewReno and so forth. Moreover, it has been reported that some earlier version of Operating Systems deploy nonconforming congestion control scheme such as Tahoe without fast retransmission (NoFR) [74].
Figure 4.3: SP-EFSM transition diagram of TCP Tahoe implementation
Figure 4.4: SP-EFSM transition diagram of TCP NewReno implementation
We use SP-EFSM to model four different implementations: $M_{NF}$ (NoFR), $M_T$ (Tahoe), $M_R$ (Reno), and $M_{NR}$ (NewReno), and then conduct a passive fingerprinting experiment. In a similar way we model the other three. Two of them, Tahoe and NewReno are shown in Figure 4.3 and Figure 4.4. They both have more states than NoFR model. Tahoe implementation uses fast retransmission scheme (FRX) and NewReno implementation uses a variant of fast recovery (FRC) scheme. Note in these models we only have two parameters of output symbol $PKT$, namely the starting sequence number and ending sequence number. This is a simplification for analysis but this will cause some trouble when we model a transition with output of noncontinuous packets. For the models in this experiment, we split the transition to handle. However, to model more complicated schemes such as TCP Selective Acknowledgement (SACK), we should add more parameters to the output to allow maximum flexibility. Another simplification we made is the omission of connection management part of TCP. Since the focus in this experiment is congestion control schemes, we simply use one state (SYN) to represent the initial state and another one (FIN) to represent the connection tear down. It is straightforward to augment these models to include the connection management features.

Recall that from the TCP specifications the major difference between FRX and FRC is that when a packet is lost, FRX brings the machine back to slow start (SS) while FRC inflates the congestion window and continues responding to duplicate acknowledgements. In case of multiple packet loss, Tahoe is capable of retransmitting more than one packet per round trip time but some of them could be unnecessary;
Reno and NewReno can only retransmit one packet per round trip time. Those features imply the fingerprint of each, however, as studied in [74], it requires great insight of those implementation to see how exactly the difference is manifested by output packets.

We now present our automated process on the models without resorting to any expertise of TCP implementations. Two types of traces are used in the passive fingerprinting experiment. The first type is collected by monitoring regular TCP traffic on internet, and our result shows that those traces normally do not contain distinguishing fingerprint. As already noted in the literature of TCP passive measurement [47], the reason is that most TCP traffic (over 90% of all TCP senders) does not have the interaction pattern that can distinguish those implementations.

In our model, if there is no packet loss, then all four machines will follow the state transition pattern SYN-SS-CA-FIN and the I/O behavior will be the same. Furthermore, if the trace contains one packet loss, then it may distinguish Tahoe and NoFR, while Reno and NewReno still behave the same way. The second type of trace is collected using TCP Behavior Inference Tool (TBIT). TBIT is essentially an active fingerprinting tool that uses preset strategy and parameters for setting up a TCP connection and simulates packet losses. TBIT aims at characterizing the TCP sender behavior for web servers. Here we use the proposed passive fingerprinting algorithm to validate the fingerprints produced by the tool. Note that these are very special traces that normally will not be observed. The purpose of using them is to illustrate how passive fingerprinting algorithm works. Both types of traces are captured by Tcpdump program. However, Tcpdump traces are not be directly applicable for our algorithm. In
order to translate them to an I/O trace of SP-EFSM, we develop a packet decoder. The decoder performs the following tasks: (1) Estimate a roundtrip time from the trace and break the trace into sections based on each round trip time. (2) Decode from the packet content the receiver’s window (rwnd). (3) Construct one input symbol with parameters using the packets sending from the client to the server. Consecutive incremental acknowledgements are combined, but not duplicated acknowledgements. (3) Similarly construct one output symbol with parameters using the packets sending from the server to the client; multiple packets are combined into intervals (PKT \([\text{start,end}]\)). In case they can not be combined, as discussed earlier, we add an extra transition with null input. (4) Detect retransmission timeout and insert a special input symbol to trigger the retransmission.

Table 4.4 shows one run of Algorithm 4.2 on the special trace generated by TBIT. Initially the state uncertainty of each candidate is nonempty. The first four rounds are a typical slow start phase restricted by receiver’s window (5 in this case). There is no difference observed. After the duplicated acknowledgement ACK [12] is sent four times, we see a fast retransmission without timeout which rules out \(M_{NF}\). Next, ACK [15] is a partial acknowledgement, which will make \(M_T\) in state SS, \(M_{NR}\) in FRC and \(M_R\) in CA. In this case \(M_R\) can not output PKT [15] due to the limit on window size; hence its state uncertainty becomes empty. Similarly, PKT [17] will make state uncertainty of \(M_T\) empty because \(M_T\) can only send one packet. After the whole trace is consumed our algorithm reports NewReno as the only possible implementation, therefore we have verified the fingerprint sequence.
<table>
<thead>
<tr>
<th>Decoded Tcpdump Trace</th>
<th>Candidate with Non-empty State uncertainty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Output</td>
</tr>
</tbody>
</table>

| ACKSYN | PKT [0,1] | \{ M_{NF}, M_{T}, M_{R}, M_{NR} \} |
| ACK [2] | PKT [2,5] | \{ M_{NF}, M_{T}, M_{R}, M_{NR} \} |
| ACK [6] | PKT [6,10] | \{ M_{NF}, M_{T}, M_{R}, M_{NR} \} |
| ACK [12] | PKT [16,16] | \{ M_{NF}, M_{T}, M_{R}, M_{NR} \} |
| ACK [12] | - | \{ M_{NF}, M_{T}, M_{R}, M_{NR} \} |
| ACK [12] | - | \{ M_{NF}, M_{T}, M_{R}, M_{NR} \} |
| ACK [12] | PKT [12,12] | \{ M_{T}, M_{R}, M_{NR} \} |
| - | PKT [17] | \{ M_{NR} \} |
| ACK [18] | PKT [18] | \{ M_{NR} \} |
| ACK [19] | PKT [19,20] | \{ M_{NR} \} |

Table 4.4: Passive fingerprinting on a TCP trace generated by TBIT

4.3.3 Fingerprint Discovery with Specification

Now we focus on fingerprint discovery problem and discuss the case that we have protocol specification model first. The specification model is a nondeterministic SP-EFSM. Such a model could be obtained from other format of protocol specification (such as RFC draft). Figure 4.5 shows an example of specification with 2 states and 4 packet types. For simplicity we assume all transitions have empty predicate and action. The non-determinism is reflected by multiple transitions from the same states.
and input packets. Such transitions are observable if their outputs are different (so that we can observe by testing on the implementation), otherwise they are non-observable.

In Figure 4.5, transition $t_1$ and $t_2$ are observable but $t_4$ and $t_5$ are not. The deterministic implementation could be any sub-automata of the specification and we identify it by active experiments. The main obstacle is non-observable transitions. Theoretically it could be proven [68] that the best one can do in this case is to enumerate the sub-automata and eliminate the false ones. We follow this paradigm and present an algorithm that outputs a set of candidate models from the specification, which we can use group matching to get the correct one.

---

**Algorithm 4.3 (Active fingerprint discovery with specification)**

*Input:* Spec. model $M_{spec}=\langle \text{Var}, \text{St}, \text{Tr}, \text{Pkt} \rangle$, Impl. $B$;

*Output:* Fingerprint model $M_f$;

1. $C=\{\}$; Initialize $Q$ as an empty queue;
2. enqueue($Q$, $M_{spec}$);
3. while (!empty($Q$))
4. $M \langle \text{Var}, \text{St}, \text{Tr}, \text{Pkt} \rangle = \text{deque}(Q)$;
5. if observable($M$)
6. foreach $s$ in $\text{St}$ do
7. foreach $p$ in $\text{Pkt}$ do
8. input prefix($s$)$p$ to $B$, get last output $O$;
9. remove all $t(s,s',p,O' \neq O)$ from $\text{Tr}$;
10. $C = C \cup M$;
11. else
12. find $t_1(s,s_1,p,O)$ and $t_2(s,s_2,p,O)$; //nonobservable
13. $M_f = \langle \text{Var}, \text{St}, \text{Tr}-t_2, \text{Pkt} \rangle$;
Algorithm 4.3 works by duplicating the specification model and removing the
non-determinism. We explore a pair of non-observable transitions (line 12); make two
copies of the current model – one keeping each transition (line 13-16). A queue is
used to store the intermediate models. If no non-observable transitions could be found
from an intermediate model (line 5), then we can use active test sequences to eliminate
the transitions that is not in the implementation, and the resulting model is a
deterministic SP-EFSM and we add that to the candidate set $C$ (line 10). Finally after
the candidate set is determined we can apply group matching algorithm to identify the
fingerprint. The specification in Figure 4.5 has four candidates according to Algorithm
5 since it has two pairs of non-observable transitions. Note that transition $t_2$ is removed
from all candidates by lines 7-8, given that a test sequence leads implementation to
state $S_i$ and then we observe output $C$ instead of $D$ on input $A$.

As expected the cost of Algorithm 4.3 grows exponentially with the number of
non-observable “branches” in the specification. If the specification contains $R$ non-
observable branches each with $d_1,d_2,...,d_R$ transitions, the cost of the whole procedure
will be $O(R^{d_1+d_2+...+d_R} \cdot P \cdot N^2)$, where $P \cdot N^2$ is the cost for group matching.
Figure 4.5: Example of active fingerprint discovery with specification

Figure 4.6: Example of passive fingerprint discovery with specification
Now we turn to passive discovery with the same specification model. Recall that the goal of passive matching is to disprove that a given trace might be generated by a fingerprint model. Similarly, the goal for fingerprint discovery is to prove that some transitions must not be in the implementation (sub-automata). Back-tracing based algorithms [68] for FSM could be applied to achieve this. A typical passive testing procedure is adopted where a graph maintains the uncertainty of current states after each input is seen. Given a new input packet, if a state can not generate it with any transition, it is eliminated from the graph so as any ancestors that can only reach the eliminated state. This is done by a back-tracing the map. Once there is a unique current state at some level, we can safely remove from the specification its outgoing transitions that have been eliminated from the map before. Due to space limit we only provide an outline of the algorithm and present an example. Interested readers are referred to [68] for details about back-tracing techniques.

---

**Algorithm 4.4 (Passive fingerprint discovery with specification)**

*Input:* Spec. model $M_{spec} = \langle \text{Var, St, Tr, Pkt} \rangle$, Trace $T$;

*Output:* Fingerprint model;

1. $U_0 =$ all configurations in $M_{spec}$; $level = 0$;
2. **while** ($level < T.length$)
3. $<I_{level}, O_{level}> =$ next pair of packets from $T$;
4. Calculate $U_{level+1}$;
5. Remove all $s$ from $U_k$ s.t. $\exists t<s, s', I_k, O_k | s' \in U_{k+1}$
6. **if** $|U_k| = 1$ for any $k$
7. $Tr = Tr - \{t<s, s', I_k, O_k | s \in U_k \land s' \not\in U_{k+1}\}$

---
8. \( \text{level} = \text{level} + 1; \)

9. Remove all non-deterministic transitions from \( M_{\text{spec}} \);

10. Remove all unreachable states from \( M_{\text{spec}} \)

\text{return} \( M_{\text{spec}}; \)

This key of this algorithm is the condition for deleting a transition from \( M_{\text{spec}} \): it is only valid when (1) the trace leads the implementation deterministically to a state at some level (2) the transition’s input matches with the trace but it can not happen (eliminated by passive testing). Again we use the specification model from Figure 4.5 as example. Figure 4.6 shows this procedure. Let the trace be \{A\?$C!(0), \ B\?$C!(0), \ A\?$C!(1), \ B\?$C!(0)\}. At the beginning, all configurations (represented by the two nodes for \( S_1 \) and \( S_2 \)) are possible. After the first I/O pair A?C!(0) is inspected, there are still two sets of possible configurations, one with \( y \) fixed and the other with \( x \) fixed. Similarly, at next level there are three sets. Now, the next pair will eliminate configurations \((S_1, x=0, y=0)\) from the map since no transitions are possible with output \( C!(1) \). Another set is eliminated at next level and the back-tracing procedure (line 5) will further eliminate its ancestors. In Figure 4.6 all configurations marked with a red cross have been deleted till this step. Now, at all levels the current state set is a singleton and some transitions can be removed from \( M_{\text{spec}} \). For instance, at level 1 we know that transition \( t_4 \) must not happen and similarly for \( t_1 \) and \( t_7 \) at level 2 and level 4 respectively. After the trace is seen the resulting machine is deterministic and in fact it is the candidate \( M_4 \) from Figure 4.5. The cost for this algorithm is \( O(L^2 \cdot N \cdot P) \) due to the cost of potential back-tracing of the map.
4.3.4 Fingerprint Discovery without Specification

The techniques discuss in previous section could be applied when a specification model could be obtained, which is not always practical. Sometimes the only information available is the format of I/O packets. Under such circumstances, we use the learning based algorithm discussed in Section 3.4 to discover the structure of the implementation. However, there are significant restrictions due to the lack of specification: (1) we can only discover the fingerprint in the form of its reachability graph, for the simple reason that there is no unique way of reconstructing the high level model elements; (2) given only one implementation and its model we discover, it is impossible to tell whether the model is indeed a fingerprint or it is common with other implementations. Consequently, this is a best-effort approach; (3) as proved in literature of automata theory there is no efficient passive approach. The best practice for passive experiments is just to record all traces. In this section we show how the black-box checking theory provides solution for active experiments.

The goal for active discovery is to identify the reachability graph of an SP-EFSM. Recall that in Section 3.4 we present a learning based algorithm to systematically discover the structure of an implementation for checking security properties. We use the same procedure here except the validation step. Again An estimation model $B^*$ of the implementation $B$ is maintained and initialized as an empty FSM. $B^*$ is updated as more traces are discovered according to $L_{fsm}^*$. This iterative process terminates with the last estimation as the fingerprint obtained, given that either we cannot find any counter-examples or it is too expensive to do so. This algorithm is sketched below.
Algorithm 4.5 (Active fingerprint discovery without specification)

Input: Impl. $B$, Packet definitions $Pkt$;
Output: Reachability graph $M$;
1. Initialize $M$ as an empty FSM;
2. Initialize a teacher simulator for $L^*_{\text{fsm}}$ algorithm;
3. while (true)
   4. $t = \text{counter\_example}(M, B)$;
   5. if ($t = \text{NULL}$) break;
   6. $M = \text{next\_estimation}(M, t, B)$;
7. return $M$;

The subroutine $\text{counter\_example}$ is provided by the teacher and we present an implementation in Algorithm 4.5. Given a conjecture it returns a trace that either distinguishes the current guess and the implementation or contains new input symbols.

Another subroutine $\text{next\_estimation}$ prepares the new guess based on the given trace. The details and examples have been presented in Section 3.4.
(a) Execution time for active single fingerprint matching algorithm

(b) Test length for active group fingerprint matching
The cost of Algorithm 4.5 depends on both the strategy used by the teacher and the \( L_{fsm} \) learning algorithm itself. To obtain a fingerprint model with \( N \) states and \( P \) inputs, at most \((N+P)\) guesses will be made. Since it takes \( O(P^* \cdot N^{*2}) \) to update a guess with \( P^* \) inputs and \( N^* \) states, the total cost in worst case is \( O(T \cdot P^2 \cdot N^2 + T \cdot P \cdot N^3) \) where \( T \) denotes the cost of calculating the counter-example at each round.

We conclude this section by presenting some evaluation results using the prototype fingerprint matching and learning tool. Figure 4.7 depicts a high level
architecture of the system encompassing all algorithms discussed in this chapter. We model the TCP congestion control and SSH handshake protocol and create several fingerprint model for the implementation variations. Figure 4.7 (a) shows the time needed for active single match operation on models of different sizes. Figure 4.7 (b) shows the result of our active group matching algorithm; we see that the total length of test required increases linearly with the size of candidate group. Finally length of test required to learn a fingerprint is drawn on Figure 4.7 (c) for three implementations. On Figure 4.7 (c) each point represents a conjecture made by our learning algorithm and the rightmost one is the last one from which the fingerprint is obtained. The evaluation justify that our methodology could be scaled to deal with moderate-sized fingerprint model. Meanwhile it is noticed that scalability remains an issue and we are addressing it in my on-going work.

4.4 SUMMARY

This chapter studies a new problem related to network security and privacy – protocol fingerprinting – using our SP-EFSM model. In our methodology, we model the fingerprint of a protocol implementation using SP-EFSM to record its distinguishing structural aspects of an implementation with rich transition semantics. It is shown that this formal approach has advantages over existing ad-hoc fingerprinting methods. This chapter presents a complete taxonomy of fingerprint matching and discovery problems. Then we present efficient algorithms for each problem and
analyze theirs complexity. Through this chapter extensive experimental results on Internet protocols are presented showing the effectiveness of the proposed methodology.
CHAPTER 5

INTEGRATION WITH A REAL WORLD TESTBED

This section contains an experimental study of integrating the proposed formal model into a real world security testing system. We first motivate a new design methodology for security device testbed using network node and protocol stack virtualization techniques. In this testbed test cases from SP-EFSM model supplied by the user could be translated into an intermediate level test description language and eventually executable files. The second part of this chapter reports our extensive evaluation of the methodology during the development of a testing platform for the U.S. Department of Defense (DoD). We present our experience of manual and automatic test generation, as well as a brief remark on the performance.

5.1 BACKGROUND

Security, reliability and interoperability are indispensable in today’s distributed heterogeneous information infrastructures. These properties rely on the correct functioning of the increasingly complicated security devices, such as traditional firewall, security switch, intrusion detection system (IDS) and so on [61,65,76,88]. For
modern security devices, testing is no longer considered to involve only the vendor because software components such as user configuration and plug-in have become significant [6,104]. On the other hand, since critical secure devices are often to be deployed at sensitive environment (e.g. government or military network), it is not appropriate to test them using the real network. Instead, target system and configuration are tested using special testbed before deployment. The last several decades have witnessed a great number of mature IP network testbed solutions [14,62], with the focus of integration testing and performance testing. In this chapter we study several key challenges and solutions particularly in testing network security devices.

First, the nature of security testing demands a high level of fidelity between the testbed and the real environment in order to compose realistic test scenario and obtain meaningful assessment. This cannot be achieved by the content-insensitive traffic generation paradigm often adopted in performance testing. Particularly, it is desirable that the testbed mimics the characteristics of the real network including topology, host machine properties and the protocol stack. Fidelity of protocol stack is especially important for (1) generation of realistic background traffic [101], and (2) designing test cases at transport or higher layer, or executing real applications.

While duplicating the real environment or approaches of this nature guarantees fidelity, it could be extremely expensive to scale. An enterprise network normally contains at least hundreds of physical hosts with heterogeneous configurations. Naturally a question to ask here is how many hosts will be involved in a test? While testing for properties like address blocking may require only a few, other features such
as resilience against Distributed Deny-of-Service (DDoS) attack could involve much more. A promising direction toward loyal and scalable solution is to employ rapidly developing virtualization techniques [85,100]. Virtual machine solutions such as VMware ESX server can increase the testbed size roughly by an order while still preserving all the applications; and lighterweight protocol stack virtualization methods can scale much better (e.g. virtual Honeynet [80,81]) at the cost of sacrificing some real applications. In this chapter we propose the integration of both based on the following assumption: even in a test case involving a large number of hosts, the subset that has to run real application simultaneously is often very small. Our experience of developing a firewall/IDS test suite justifies this assumption.

The last but not least notable issue is automation. Security tests usually employ precisely specified sequence of actions from various principals, which essentially requires coordination of the external network, internal network and the device under test itself. The test system should hide this control complicacy to the end user. In addition, as the security features of sophisticated device span over multiple network layers, the test description mechanism should provide corresponding capability and at the same time facilitate automatic test execution.

Motivated by the above insights we propose Virtual Cyber Security Testing Capability (VCSTC) – a novel methodology of testing security device and the associated application solutions with high fidelity, scalability and usability. VCSTC methodology aims at a broad category of target systems that could be deployed at the boundary (gateway) of a local, usually an organizational or enterprise network. The main security-related functionality of such systems includes multi-directional access
control, intrusion detection, virus/worm detection, vulnerability analysis and mitigation. Examples of such devices available on the market include Cisco ASA family, Top Layer Secure Command and many lower-end consumer security appliances. Two networks are involved in using and hence testing such devices: an internal network to be protected, and an untrusted external network. Typically there are four steps in testing: create a model for both internal and external network; emulate the two models on a testbed; develop test cases; and execute the test cases on the testbed. VCSTC methodology spans over all four steps, although model construction step is not related to testing and therefore is not the focus of this thesis. To the best of our knowledge our approach is the first to integrate network emulation and automated testing, and it is distinguished from the existing security testbed solutions (such as DETER [14]) by the following two key aspects.

**Hybrid network emulation:** to test a security device deployed at network boundary, both internal and external network are emulated using the mixture of network host and protocol stack virtualization techniques. The emulated network contains two types of nodes: a small number of FAT nodes that are fully featured virtualized network hosts, and a large number of THIN nodes each having only a virtual TCP/IP stack. Configuration of emulated networks is automatically done according to the user network model. Hosts involved in a test case will be mapped to emulated nodes (FAT or THIN) depending on what is executed on that host. By leveraging the advantage of these two virtualization methods, our hybrid testbed achieves the best balance between loyalty and scalability. We show that our methodology can support up to one thousand emulated nodes on a commodity computer.
Test description language: in order to cater the diverse features of security device VCSTC uses an intermediate level test case description language to facilitate the specification of Point of Control and Observation (PCO) at various layers: IP, Socket and Application. PCOs could be deployed at any host and the target device, while the actually deployment and control are automatic. This language is tightly based on a programming language to provide virtually unlimited expressivity. A test case is dynamically compiled into a native executable before execution by the test driver. We evaluate this scheme by both manual test case generation and using the language as the target of model-based formal test generation methods.

5.2 TESTING METHODOLOGY AND SYSTEM ARCHITECTURE

In VCSTC methodology there are two essential components in security testing: model and test cases. They are independent and developed separately. The network model should contain sufficient information to emulate a real network, and in the meantime not associated with any special devices and therefore generally reusable. VCSTC supports several methods to build a network model: it could be automatically synthesized using network management protocols (e.g. SNMP) and collected network traces, or using random network topology generation; or manually using prevalent modeling language such as UML with software tool assistance.

Test cases could also be constructed through various ways. A test case mainly specifies two things: (1) a set of PCOs and their deployment (2) a sequence of actions
of the PCOs with the expected outcome. Test cases could be made abstract by defining parameters of numerical type or special type like IP address. Such abstract test cases could be concretized by selecting a set of parameter values. To execute a test on a given network model (assuming they are compatible) we first compile it together with all supporting libraries into an executable. Next the network model is automatically emulated and PCOs are deployed at the designated hosts, each of which is controlled by the test driver through a private communication channel. After the test case finishes a log file with all network activities during its execution is returned to the tester along with the test verdict for evaluation. In this framework high fault coverage can be achieved by the combination of three approaches - selecting different network models; generating test cases from a model of the feature under test to cover it more comprehensively; and selecting many combinations of parameter value for an abstract test case.

Figure 5.1 shows the architecture of a full-featured testing system we developed that realizes the VCSTC methodology. The system is an out-of-box product that could be contained within a single server. The modeling module provides an UML compatible environment for creating and validating network models. Models are stored in a database after validation. The operational environment implements a streamline consisting test preparation, test execution and test result processing, all of which are exposed to the user via a Web-based interface. The test generation module accepts abstract test cases (as textual file) generated by the tester or from a formal model. They are then concretized based on a certain parameter selection policy and eventually compiled into a native binary file. The test executor is responsible of
creating emulated network and executing test cases on it. The emulated network is essentially a virtual honeynet with hybrid nodes (details in Section 5.4) implemented using a pool of virtual machines with a central controller. The whole testing workflow is automated to the extent that testers interact with VCSTC server only by providing network models and test cases from Web interface. The test executor hides the complexity of controlling the network emulator and PCOs from the users.
Figure 5.1: Architecture of VCSTC platform capsulated within a single server that connects to both internal and external interface of a target device.
Now we briefly describe the network configuration on the server. Three types of virtualized network interfaces connect the emulator with other components. There are many virtual Network Interface Cards (NIC) of each type grouped together by virtual networks. The private interface VNIC-Control is used by the test executor to control the honeynet and all PCOs. This type of interface is totally invisible to the test cases and the target device. The other two types of interfaces VNIC-Int and VNIC-Ext are bridged with the physical NICs NIC-Int and NIC-Ext on the server, which are connected to the internal port and external port of the target device, respectively. This setup enables emulation of both the internal and the external network, while all emulated packets generated during testing are transformed into real packets at the corresponding VNIC before delivered to the device. Note that for simplicity Figure 5.1 only shows one internal path to the device whereas additional NIC and VNIC could be deployed similarly according to the requirements for testing. Network traffic - both real and emulated - passing all VNIC will be monitored during test execution and readable test reports such as Message Sequence Chart (MSC) are generated afterwards for further analysis.

5.3 TEST SPECIFICATION LANGUAGE

VCSTC uses its own notation for test case specification. The rationale of introducing the new notation is not to replace the traditional high level test specification language such as TTCN-3 or MSC; instead our main incentive is to provide a flexible way of developing test sequences related to security features at all
layers of network protocol stack. Toward this goal our language is tightly based on a
native programming language such that any valid statement of the host programming
language could be embedded in the test code, providing encoding of an input symbol,
for instance. A test case is a textual file with multiple declarative sections (described
below) and a test code section. We show later that the proposed intermediate level
language could be used to interpret test sequences from more abstract formal models
such as EFSM [54].

The most important element in our language is PCO. A test case defines
multiple PCOs on various places and controls their behavior. A PCO is deployed on
either a host of internal/external network or the target device. Every PCO on network
has one of three types: (1) Packet PCO sends and receives raw network packet of TCP,
UDP or IP protocol by taking over the network interface of the host. (2) Socket PCO
manages one TCP or UDP socket. (3) Application PCO handles one user application.
It reads and writes to the application through its standard input/output channel. Table
5.1 summarizes the three types of PCO. A PCO must be bound on a host (i.e. an IP
address) before it can be function, and the mapping could be done by various ways.
The PCO definition might supply a fixed IP address if the test case is design for some
specific network models, or otherwise the binding could be done in test code by
calling run-time API. Note that multiple PCOs with different types could co-exist on
the same host except for Packet PCO due to the nondeterministic behavior under the
situation of multiple network capturers.
<table>
<thead>
<tr>
<th>Type of PCO</th>
<th>Packet PCO</th>
<th>Socket PCO</th>
<th>App. PCO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method of control</td>
<td>Send/Receive raw TCP/UDP/IP packet</td>
<td>Read/Write TCP/UDP socket</td>
<td>Execute native application</td>
</tr>
<tr>
<td>Number on each node</td>
<td>One</td>
<td>Many</td>
<td>Many</td>
</tr>
<tr>
<td>Blocking I/O</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 5.1: Three types of PCOs in VCSTC

The actions of PCOs are defined in the test code section, which eventually returns a test verdict. A test case could contain parameters and become abstract. Abstract test case cannot be executed before assigning parameter values. Our language supports parameters of bounded Integer type and IP address type, and a test concretization algorithm implements parameter value selection according to certain coverage criteria such as random sampling, boundary coverage and so forth. After the test case is concretized it is automatically transformed into the host language for compilation. In this phase auxiliary code such as test case initialization and cleanup routines is generated and weaved together with the test code. During compilation the VCSTC runtime library proving essential functionalities and all user defined libraries are linked. In practice, a lot of reusable routines (e.g. Malware simulation, special packet generation) are encapsulated in the form of library so that the test code could focus on the logic, that is, the sequence of actions from PCO. The VCSTC runtime library implements all types of PCO and the proxies used to control them remotely from the test executor.
1. #TESTCASE OBL_Length_TCP
2. #PARAM {
3.   int\{[0,512]\} NCLIENTS;
4.   int\{[0,1024], [50000,51000]\} PORT;
5. }
6. #PCO {
7.   SOCKET pco_client[NCLIENTS];
8.   APP pco_server;
9.   DEVICE pco_device;
10. }
11. #PACKET {}
12. #VERDICT {
13.   success, failure, unknown, timeout
14. }
15. #TESTBODY
16. {
17.   bind_PCO(pco_client, INTERNAL, RANDOM|NONDUP);
18.   bind_PCO(pco_server, EXTERNAL, RANDOM);
19.   log("Testing length of black-list using TCP");
20.   pco_server.execute("httpd", PORT);
21.   pco_device.config("clear black list");
22.   Vector black_list = new Vector<InetAddress>();
23.   for (int x= 0;x<NCLIENTS;x++) {
24.     if(!pco_client[x].connect (pco_server.getIP(),PORT)) return unknown;
25.     black_list.add(pco_client[x].getIPAddress());
26.     pco_client[x].close();
27.   }
28. }
Now we discuss more details by an example. Figure 5.2 shows a test case that checks whether a security device provides an outgoing source IP black list of sufficient length. The test code uses Java as host language and implements a rather straightforward logic: a Web server is started on an external host by an Application PCO (line 8). There are many (NCLIENT) internal hosts with a Socket PCO on each (line 7). Both the server and clients are selected randomly from the network (lines 17-18). The device is controlled by a Device PCO (line 9). At the beginning the test case launches the server and clears the black list (line 22), followed by a check (lines 24-28) to see whether all clients can reach the server. Next the black list is configured through Device PCO (line 29), and we retry the connections again, expecting that no client can successfully reach the server. Any client’s success in connecting at this time (line 32) proves the black list useless and therefore the test case returns the verdict failure. The test case contains two Integer parameters (lines 2-5): number of clients.
and server ports, which are to be assigned according to a user policy. Note that the execution of the test case is fully automated except for Device PCO. Configuration of target device might need manual activity, depending on the interface a specific device is providing. Many vendors provide programmable configuration mechanism, which could be utilized by VCSTC runtime to fully automate test execution.

We close this section by a remark on the relationship between test case and network model. Test cases like the one in Figure 5.2 do not depend on any network-specific properties such as background traffic and therefore could be executed on any network models. The only implicit constraint is that the network must contain enough (in this case NCLIENT) distinct hosts – this will be checked statically by the test executor when loading the test case. On the other hand, the test case might also explicitly specify a list of compatible network model names if necessary.

5.4 NETWORK HOST AND PROTOCOL STACK VIRTUALIZATION

A test case is executed on an emulated network that from the view of the device under test is the same as a real network. Emulated network is created from a network model using hybrid network virtualization approach. As we mention in Section 5.1, VCSTC mitigates the key challenge of scalability by using a hybrid virtual honeynet. Honeynet [96] is used recently as a best practice of network emulation for the purpose of attack identification. We adapt a hybrid honeynet design where two virtualization techniques are used together to achieve the balance of scalability and fidelity. First we distinguish two terms used in this section: logical
node and *physical* node. A logical node is a network host, either external or internal, defined in a test case. A logical node is identified by its IP address. A physical node is a network host in the emulated network. The test executor maintains a mapping from logical nodes to physical nodes and deploys the PCOs according to this mapping. In our scheme of hybrid honeynet there are two types of physical nodes in the emulated network:

**FAT node**: A FAT physical node is emulated by a complete virtualized host machine. Thanks to the advanced virtualization techniques such node can accommodate any application running at the real host. A repository of pre-configured (e.g. with different Operating System and/or applications) virtual machine images are stored at the server while the honeynet controller selects the proper ones to load. Unfortunately, host virtualization is still very expensive and we cannot afford to emulate the whole network using FAT nodes alone.

**THIN node**: A THIN physical node is emulated by virtualizing only a TCP/IP protocol stack but not the actual resources of a host. Software solutions such as Honeyd [80] accomplish this by overriding the IP protocol stack on a single (possibly virtual) machine and claiming responsibility for a range of IP addresses. Socket based program could be executed on top of the virtualized protocol stack appearing to the outside as running with its own address. This approach is lightweight and therefore very scalable. The cost however, is that the function of PCO deployed on them is limited. Since all programs launch on THIN nodes share the same physical machine and therefore its resources, there is obviously a potential problem of interference. The exact constraints are determined by the virtualization tool used. In our system with
Honeyd as protocol stack emulator if a logical node is mapped to a THIN node, then application PCOs on it can only execute a special type of socket-based EFSM simulation program synthesized from user network traces.

Figure 5.3: Internal structure of a hybrid honeynet with two types of nodes

We create and configure a mixture of these two types of physical nodes in the emulated network, as shown in Figure 5.3. From the view of the emulator, we have a small number \( M \) of FAT nodes and a protocol stack virtualizer supporting \( N \) THIN nodes. These heterogeneous physical nodes are connected by a virtual network switch and form a honeynet. On the other hand, the heterogeneity is made transparent to the test cases. That is, all logical nodes are the same in terms of PCO capabilities.
mapping between logical and physical node is first created by the test executor before a test case is loaded, and is adjusted dynamically by network reconfiguration under some circumstances. The separation of logical nodes and physical nodes has two obvious advantages. First the honeynet resource provisioning could be changed anytime - for example adding more FAT nodes - without affecting any test case. Second, in most test cases only a small number of hosts run real applications (and therefore require to be mapped to a FAT node) simultaneously, despite that the total number of hosts is large. In such situations when a logical node does not have any activities we can remap it to a THIN node at runtime. When the current physical node provisioning can no longer support the execution of a test case, the test executor will get a runtime error and hence returns failure. Below we describe some heuristic guidelines practiced by the test executor for static and dynamic mapping.

Guideline 1 (Static): If a logical node does not have Application PCO, always map it to a THIN node, because Packet and Socket PCOs could both be supported.

Guideline 2 (Static): If a logical node has both Socket and Application PCO, map it to a FAT node if there is one available, otherwise map to a THIN node. Nodes with Application and Packet PCO have lower priority of mapping to FAT node. This is because Packet PCO is easier to migrate dynamically than Socket PCO.

Guideline 3 (Dynamic): Before an Application PCO executes a real user application, mapping need to be adjusted if the logical node is currently mapped to a THIN node. If there is Socket PCO with established TCP connections at this time, we report failure because we cannot migrate TCP connection across physical nodes. Otherwise, if there is an unmapped FAT node, it is remapped to the logical node. If all FAT nodes are
already mapped, we check whether one of them could be swapped to a THIN node, that is, on the current owning logical node no Application PCO is executing and no Socket PCO is connected. If this condition is satisfied, honeynet controller will reconfigure the network (i.e. IP address) and switch the mapping of two logical nodes, therefore allow the user application to be executed on a FAT node. If no FAT node satisfies this condition, failure is reported.

As an example of test case, in Figure 5.2 we have an array of logical nodes (client) with only Socket PCO and another node (server) with only Application PCO. The mapping for this test case is trivial since only one FAT node is needed for the server node and all clients are mapped to THIN nodes.
1. ……
2. #PCO {  
3. APP pco_client[4];  
4. APP pco_server;  
5. DEVICE pco_device;  
6. }  
7. ……
8. #TESTBODY  
9. {  
10. bind_PCO(pco_client, INTERNAL, RANDOM|NONDUP);  
11. bind_PCO(pco_server, EXTERNAL, RANDOM);  
12. ……  
13. pco_server.execute_service(“IIS6.0”);  
14. ……  
15. pco_client[0].execute(“lynx”, "domain.com/page.cgi”);  
16. ……  
17. pco_client[0].terminate();  
18. ……  
19. pco_client[1].execute(“iexplore”, "domain.com/page.cgi”);  
20. ……  
21. pco_client[1].terminate();  
22. ……  
23. pco_server.terminate();  
24. }

Figure 5.4: Example of test case with dynamic node remapping
Figure 5.4 shows a more illustrative example. In this test case we have two client nodes with Application PCO and a server node with Application PCO. Table 5.2 shows the node mapping at several key timing points when the emulated network contains unlimited THIN nodes but only 2 FAT nodes. Before executing the test case, the first two nodes (client[0] and client[1]) get the FAT nodes and the rest (including server[0]) get THIN nodes. Before the server starts (line 13), it needs to be remapped to a FAT node, and client[0] could be swapped out since it is not active. Similarly when client[0] needs to launch its program reconfiguration happens again, swapping it with client[1]. Finally client[1] launches a program, now since client[0]’s PCO has terminated its application, it could be switched to a THIN node and client[1] gets the FAT node. Note that the jitter of mapping in this example is quite unrealistic since in practice the server contains much more FAT nodes.

Dynamic network reconfiguration also involves a reconnection between the PCO proxy (in the test executor) and the physical node through the network interface VNIC-Control of the honeynet (Figure 5.1). When the new IP address becomes usable
on the physical node, the PCO proxy will disconnects the old PCO and connect the new one. On a separate issue, we are currently investigating suitable process migration schemes supporting dynamic remapping including live TCP connections, which fully take the advantage of the hybrid network design.

5.5 EXPERIMENTS AND EVALUATION

The proposed VCSTC methodology has been fully applied in the development of a real security testing platform for the U.S. DoD (Department of Defense). The purpose of this project is to provide critical network infrastructure owners with an effective and easy-to-use mechanism to assess the suitability of a security device or solution with respect to their own infrastructure before investment. In this section we report our experience and evaluation during the development of this platform. We start from a brief overview of the system configuration and some simple practice; then we summarize our effort of integrating automatic test generation techniques. Our system supports generating test cases (in our test description language) from two popular formal models – SP-EFSM and Simplified Firewall Rule Language. We also present performance evaluation of the system installed on commodity hardware in order to justify its feasibility and scalability.

5.5.1 System Configuration and Basic Operations

As discussed earlier, the whole system could be deployed on a single machine. We choose a typical hardware configuration: a Dell Precision 690 workstation with two Xeon 3.2 GHz Due Core CPUs and 2GB memory. The server has two Gigabit
physical NICs (NIC-Int and NIC-Ext). Both modeling module and test executor are implemented in Java 1.5 and Jpcap (a packet manipulation utility). The hybrid honeynet is composed of 5 VMware virtual machines running Ubuntu Linux as guest Operating System – 4 of them with 256MB virtual memory each are used as FAT nodes and the last one with 512MB virtual memory runs Honeyd 1.5 to emulate up to 1024 THIN nodes. The system is used to test several security devices on the market, and our performance evaluation is conducted using Netgear ProSafe FVS318 VPN Firewall/Switch.

We use both network models synthesized from real network and randomly generated large models. For real network, we derive a model from a testbed of the WAN-in-Lab project [56] developed by Caltech. This testbed has 4 Cisco routers with SNMP capability. The whole model contains 39 subnets and totally about 40 distinguished hosts with services available. We imagine the target device is about to be deployed at the gateway of this network and manually develop a small test suite that covers the classic access control and content filtering features common to typical Firewall and IDS. It takes a Java developer two days after one day’s training to write about 50 test cases (Table 5.3). Using these test cases, the tester is able to verify precisely the details of many features of the device that is stated very informally and vaguely from its user manual. For instance, one of the Anti-virus test cases discovers that the device cannot enforce malicious URL blocking when the URL is encoded in HEX form (e.g. “www.abc.com/x.e%78e” for “www.abc.com/x.exe”), which effectively renders this
UML blocking feature useless. Based on this experience we consider our test description language efficient and of good usability.

<table>
<thead>
<tr>
<th>Firewall Feature</th>
<th>Inbound filtering</th>
<th>24 Test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Outbound filtering</td>
<td>24 Test cases</td>
</tr>
<tr>
<td></td>
<td>Port Forwarding</td>
<td>4 Test cases</td>
</tr>
<tr>
<td></td>
<td>Dynamic filtering</td>
<td>1 Test cases</td>
</tr>
<tr>
<td>Anti-virus features</td>
<td></td>
<td>2 Test cases</td>
</tr>
<tr>
<td>Intrusion detection features</td>
<td></td>
<td>3 Test cases</td>
</tr>
</tbody>
</table>

Table 5.3: Summary of the firewall/IDS test suite of VCSTC

5.5.2 Integration of SP-EFSM Model

The applicability of our methodology could be broadened by leveraging the advanced test generation methods. We make an effort to integrate them into our methodology. We investigate automated translation from test sequences derived from formal model to the VCSTC test description language. The first model we implement is SP-EFSM proposed in this thesis. Our test system provides a GUI to specify a feature of the device using SP-EFSM with parameters in I/O message. Figure 5.5 shows an example of a single port blocking feature with two states. From the SP-EFSM model test sequences could be automatically derived using various approaches, such as checking sequences from reachability analysis (Figure 5.5 shows the reachability graph when the range of port variable has 3 values). We then translate
each test sequence into a test case file and then generate an incomplete user library that defines the I/O symbols of the model. Figure 5.6 shows a section of the test case corresponding to the sequence \{Set\_Block[0]/-, Visit[0]/-, Set\_Unblock/-, Visit[0]/Resp\} and an empty method definition for input symbol Set\_Block, for which the test designer is responsible of providing the code to implement this input symbol using the PCOs on internal and external hosts. Note this only needs to be done once and then shared by all test cases for the same model.

Figure 5.5: A simple SP-EFSM model of port blocking feature (left) and the corresponding reachability graph as a FSM (right)
1. #TESTBODY
2. {
3.     input_Set_Block(pco_ext, pco_int, pco_device,0);
4.     ASSERT(output(pco_ext, pco_int, pco_device) == NULL);
5.     input_Visit(pco_ext, pco_int, pco_device,0);
6.     ASSERT(output(pco_ext, pco_int, pco_device) == NULL);
7.     input_Set_Unblock(pco_ext, pco_int, pco_device);
8.     ASSERT(output(pco_ext, pco_int, pco_device) == NULL);
9.     input_Visit(pco_ext, pco_int, pco_device,0);
10.    ASSERT(output(pco_ext, pco_int, pco_device) == Resp);
11.    return success;
12. }
13. #USES LIB_Simple_Port_Blocking

1. public class LIB_Simple_Port_Blocking
2. {
3.     …
4.     public void input_Set_Block(PCO pco_ext, PCO pco_int, PCO pco_device) {
5.         … //user provides implementation of input symbol;
6.     }
7.     …
8. }

Figure 5.6: Test code and library generated from SP-EFSM test sequences
Similarly our system supports generating test cases from firewall configurations. We use a simple grammar to describe firewall rules following classic semantics [6,65]. A rule contains a predicate based packet filter and an action and a configuration is an ordered list of rules. After a user inputs a firewall configuration, test cases with input packets are automatically generated. The elements in the packet filter can be either a value or a wildcard (“*”), and furthermore the user might ask the test case concretization process to select a value by specifying it as a parameter of the rule. For example, a configuration as specified below is composed of two rules A and B. The test case generated from this configuration will contain three parameters (i.e. Src port in A, Protocol in B and Src port in B) and a pair of Socket PCO binding on internal and external network, respectively.

A: “Allow TCP from [10.0.0.1:Param$_1$] to [*.*] through External”

B: “Deny Param$_2$ from [*:Param$_3$] to [192.168.0.2:80] through External”

The test code first enables this configuration through Device PCO, and then essentially sends a packet enabling a subset of rules to see whether the device under test takes the expected action. Clearly the subset of rules triggered by a particular packet depends on the parameter value of all rules. In our example, A and B could be enabled together if Param$_2$=TCP and Param$_1$ = Param$_3$. In fact when both A and B are enabled they conflict with each other, and it is to the interest of the tester how the device will handle. The test case concretization process produces parameter assignments in such a way that most rule subsets are covered. Due to space limit we
omit the detail of the algorithm and test case generated. From our experiences of integrating the two formal models, we believe that our methodology is promising for a variety of application domains related to network security testing.

5.5.3 Performance Evaluation

Finally we remark on the performance evaluation of our system. First we clarify that VCSTC is not targeted for performance testing or load testing therefore it is not designed to meet hard real-time requirements. The purpose of evaluation is instead to justify the feasibility of our design for network model and tests of practical scale. We use a series of micro-benchmarks to measure various aspects of the system, with the focus on the performance penalty incurred by using hybrid network virtualization. The first performance penalty comes from initializing the emulated network. For FAT nodes the controller reset/reload loads a virtual machine image which takes constant time; then the Honeyd engine virtualizes the pool of THIN nodes and launch the PCO on each node. The Honeyd start up time is proportional to the number of THIN nodes as shown in Table 5.4(a), for instance a network of 1024 nodes could take up to 1 minute to initialize. Note that under certain situations it is unnecessary to reinitialize network for each test case, specifically when all test cases share a network model and they all cleanup properly. In addition, communication cost between the test driver and the PCO is not negligible because the control message sent might carry a data portion (e.g. a packet to send from that PCO). We measure the transmission time with various message sizes shown as Table 5.4(b). There is no difference between FAT and THIN nodes since the same control channel is used.
<table>
<thead>
<tr>
<th># of THIN Node</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Startup Time</td>
<td>6.67s</td>
<td>24.13s</td>
<td>30.59s</td>
<td>55.43s</td>
</tr>
</tbody>
</table>

(a) Startup time of hybrid honeynet with different network size

<table>
<thead>
<tr>
<th>Message Size</th>
<th>4KB</th>
<th>8KB</th>
<th>16KB</th>
<th>32KB</th>
<th>64KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trans. Time</td>
<td>&lt;1ms</td>
<td>1.00ms</td>
<td>2.13ms</td>
<td>3.88ms</td>
<td>7.87ms</td>
</tr>
</tbody>
</table>

(b) Transmission time of PCO control messages

<table>
<thead>
<tr>
<th># Connections</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bandwidth (each con.)</td>
<td>6.80Mb/s</td>
<td>2.95Mb/s</td>
<td>2.08 Mb/s</td>
<td>1.90 Mb/s</td>
</tr>
</tbody>
</table>

(c) TCP transmission bandwidth between external and internal network nodes with 1-4 simultaneous connections measured by iPerf

Table 5.4: Performance evaluation of the VCSTC testbed
The packet dispatching mechanism used by protocol stack virtualization tools (i.e. Honeyd) also causes delay in data transmission involving a THIN node. Basically all socket function calls are delegated to the tool and go through internal tunneling, which forms a global bottleneck. We use a benchmarking tool iPerf to measure the bandwidth of concurrent TCP connections between external and internal nodes. If both are FAT nodes the bandwidth for a single link is 8.89Mb/s; if one side is THIN node, it is downgraded as shown in Table 5.4 (c). We believe that this bandwidth limitation is not critical to validity of most security related tests.

![Honeyd CPU load percentage for three networks of THIN nodes each sending UDP traffic at 2KB/s](image)

Figure 5.7: Honeyd CPU load percentage for three networks of THIN nodes each sending UDP traffic at 2KB/s

Another simple benchmark is designed to evaluate approximately the work load of testing server. The dominating factor here again is large number of THIN
nodes virtualized by Honeyd. We create network of different size, then let each node send UDP packet at a given rate to random destination node. This scenario corresponds to a typical test case where all logical nodes carry symmetric tasks. Figure 5.7 shows the CPU usage of the VMware guest OS running Honeyd during a window of 20 seconds. When the network is small (16 nodes) an average 33.4% CPU time is used while a large network (1024 nodes) is likely to saturate the CPU (85.8%). While admittedly being a coarse measurement, this shows that our system is capable of running fairly large models.

5.6 SUMMARY

This chapter presents a security testing system VCSTC that integrates the proposed SP-EFSM model. The novel design of hybrid network emulation provides both fidelity (by network host virtualization) and scalability (by lightweight protocol stack virtualization). We also develop an intermediate level test description language that is suitable for security tests at various network protocol layers. This chapter discusses how test cases are executed automatically on the emulated network model. Extensive experiments have been conducted on our implementation platform, which justify the benefits of our proposed methodology.

On the other hand, we are still at the initial stage of applying network virtualization techniques to testing. Lots of issues remain to be explored in our current approach before its applicability could be further broadened. Our approach aims security testing at IP layer and above. As a matter of fact, VCSTC does not support
routing protocol emulation despite that it generates real IP packets. Consequently, routing related security features cannot be tested under our framework. For similar reason data link layer security features are not supported. Emulating routing in a scalable fashion is a challenging task and it may change the protocol stack virtualization scheme in a drastic way. A promising approach is to use one virtualizer for each routing domain or subnet, and connect them by FAT nodes where routing protocols are implemented. Also the test language is to be augmented to support routing operations at the PCOs. Protocol synthesis from real network is another challenge where network traffic with high fidelity is desired. This is an issue for both network modeling and testbed design. Since running real user applications on top of all virtualized nodes is clearly not practical, we need to synthesize a model of the protocol from network traces [23] and emulate it on the testbed in order to generate (not simply replay) traffic patterns similar to those seen. Using the formal methodology proposed by this thesis we could use a state machine minimization approach [41] to obtain SP-EFSM models from field-decoded protocols (e.g. by Ethereal), and implement a special program to simulate the resulting SP-EFSM on top of both FAT and THIN nodes. We envision this and the enhancement for routing emulation will render our VCSTC a more powerful and useful tool for testing both hardware and software.
CHAPTER 6

CONCLUSIONS

In this thesis we proposed formal methodology for studying testing problems for communication protocol systems. This thesis first introduced a new formal protocol model – Symbolic Parameterized Extended Finite State Machine (SP-EFSM) that extended a classic state machine model with additional model elements to handle the complicated aspects of modern protocols. A protocol could then be specified by a set of communicating SP-EFSM. The proposed methodology was applied to two key research problems related to network protocol security and reliability. By using rigidly proven algorithms on the formal model of protocol specification and/or implementation, we derived solutions that are fundamentally more general and easy to automate compared to existing approaches. For the problem of testing of message confidentiality, the main contribution was a supervised learning based testing technique that overcame the limitation of conformance testing by systematically discovering the internal structure of black-box implementations. For the problem of protocol fingerprinting, a complete taxonomy of fingerprint matching and discovery problems was identified; and efficient algorithms were studied to solve these problems. Both real world protocols and simulated protocols were used in the
experiments to justify the proposed approaches. This thesis also studied how the SP-EFSM model is integrated to a practical security testing platform. The design of the VCSTC testing platform was presented and we showed how test sequences of SP-EFSM model were automatically translated into template of executable low level test case, and then later executed on the target network emulated by VCSTC. To summarize, the proposed formal methodology provided a solid theoretical foundation for solving the emerging security testing problems of communication protocols.
BIBLIOGRAPHY


142


[100] VMware Inc., www.vmware.com


