2008

Auto Red Team: a network attack automation framework based on Decision Tree

Song Lu
Iowa State University

Auto Red Team: a network attack automation framework based on Decision Tree

by

Song Lu

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Iowa State University

Ames, Iowa

2008

DEDICATION

To the memory of my father

To my mother

And

To my wife, Xueji Chang

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

In this thesis we discuss our research in incorporating Machine Learning into network attack automation. The key idea is to audit the traffic between the attacker and the target machine, then apply Decision Tree Learning methods on the audit data to generate a set of rules, and create a smart attacker that is guided by those rules and is capable of launching attack sequence according to the response from the target machine. By conducting experiments on Linux platform, we constructed a framework named Auto Red Team (*ART*) that audits traffic, compose training data, and generate an smart attacker by feeding those training data into a Decision Learning Tree model. Experiments shows that the *ART* can realize an effective and accurate attack automation. Beside basic data analysis on the experiment data, we also apply a statistical method, *Principle Component Analysis* on the experiment data to verify the generated rules. Although the *Principle Component Analysis* can not completely explain the rules by the Decision Tree module, some convincing explanations on the relationship between those rules and certain Principal Components were given.

CHAPTER 1. Introduction

In this chapter, we first describe the current state of attack automation and existing problem. We then provide our approach with emphasis of its contributions. Finally we give the roadmap of this thesis.

1.1 Problem Statement

The number of potential targets for attacks against computer network is soaring because of the increasing network scale, more and more diverse network user population, the evolution of the information technologies, and other factors. These factors adversely impact the efficiency of current network security systems. On the other hand, the threat of attack increases with sophisticated technologies that can obfuscate the nature of attack, and make the attack more dynamic and polymorphic. Thus, researches in new protection model against the increasing attack threat is becoming extremely important.

Among other protection technologies, proactive and offensive approaches can provide network security experts and system administrators with better help on identification of potential security holes in current network infrastructure. One of the proactive methods is *Red Teaming*, an approach that deploys network security experts to play attackers, showing how a real attacker would exploit the system's vulnerability. With *Red Teaming*'s help, the system can be more effectively protected against the exploits from both script kiddies and sophisticated hackers.

Government agencies and institutions have already benefited from the application of *Red Teaming*s. For example, the US Homeland Security Department launched simulation of computer terrorist attacks on computer, banking, and utility systems in October 2003 [Bridis,

T. (2003)]. The simulation exposed un-seen security holes, providing opportunities to the security experts to fix these holes before outside hackers could find them. Another example is the *Cyber Defense Competition* hosted by Information Assurance Center at Iowa State University. During the competition, the Blue teams build network systems including database and websites, and the Red team that consists of skilled hackers performs penetration on those network systems. Through the competition, the attendants will get deeper understanding on network security.

To achieve *Red Teaming*'s goal, the Red Team member must identify a real hacker's inner working, investigate all points of interest, and perform thorough penetration tests on them. Some of those investigation and penetration tests are replicable. Security experts would be emancipated from such redundant tasks if there were a robot taking over those replicable tasks. There are certain softwares to perform simple automatic tasks. For example, the Hacker-Safe from McAfee provides port scan to find vulnerability of a system. However, an automatic attacker that can really alleviate the human being's work loads should have profound knowledge about the attacks, should be able to play single attack or a set of attacks consisting of multiple types of exploits, and should be able to dynamically change its attack strategy according to the response from the target system, which requires the automatic attacker to be capable of learning from attacks. To our best knowledge, there is no such a robot acting like a smart attacker to play exploits against the network systems.

1.2 Our Approach

To address the above problem, we propose a design idea and implementation detail of a software, Automatic Red Team (*ART*) that is an attack automation framework based on decision tree learning. By launching attacks selected from an attack database, sniffing traffic between attacker and target, composing training data, and generating new attacking strategy through a decision tree module, the *ART* platform achieves attack automation with less human supervision.

However, the intention of the *ART* framework is not to totally replace the network security experts. On the contrary, we do not believe that the security assurance task would become a fully automatic, plug-n-play job, at least in the near future. The main purpose of the *ART* framework is to assist network security professionals in assessing the vulnerability of network systems by taking over some simple and replicable penetration tasks.

The key contributions of our research works are listed as below:

Artificial Intelligence The *ART* incorporates a Machine Learning technology, C4.5 Decision Tree, making the *ART* a smarter attack platform than others without Artificial Intelligence technologies.

Polymorphic Attacks Based on an attack database and guided by attack strategies, the *ART* can launch multiple types attacks against the target in an attack scenario. The attack database can be updated to keep up with the development of attack technologies.

Portability among Platforms The inter-dependencies among attacker, attack database and the Decision Tree module are minimized. By playing plain-vanilla attack, the attacker does not need to know the details of the individual exploit program, so that the switch or upgrade the attack database would not have impact on the attacker, and vice versa. The Decision Tree module receives the standardized training data from a Training Data Generator, and the generated rules by the module will be incorporated into a new attacker by an Attack Strategy Composer. Thus, there is no direct connection between the attack database and the attacker, minimizing the dependency between them. Thus, migration or upgrade of the *ART* framework becomes less stressful.

By building such a software, we made a meaningful attempt towards the attack automation that would benefit the improvement of cyber space security. We believe that our approach is a useful tool to be able to alleviate the network security professionals' work loads during *Red Teaming*.

1.3 Thesis Outline

The rest of the thesis is organized as the following: Chapter 2 reviews the related works: 1) attack simulation and modeling; 2) the application of machine learning in network security, especially for intrusion detection; 3) the application of statistical methods in network security. Chapter 3 provides the design and implementation detail of the *ART* platform, and Chapter 4 presentes a case study on the *ART* framework: 1) describing the experiment environment and procedure; 2) analyzing the experiment result; 3) verifying the generated attack strategy with a statistical method, *Principle Component Analysis*. Finally, Chapter 5 concludes our work and give a discuss of the future work.

CHAPTER 2. Review of Related Works

In this chapter, we reviewed the related works on Attack Automation, the application of Decision Learning Tree in network security such as Intrusion Detection, and the application of statistic models such as *Principle Component Analysis*.

2.1 Attack Modeling and Simulation

Howard, J.D., T.A. (1998) developed a Common Language, a high level taxonomy, to classify and understand computer security incident information including the attack category and related terms.

Researchers took a lot of efforts to make the simulation environment more and more closed to the real cyber attack / defense scenario. Lau, F., R.H. (2000) simulated a Distributed Denial of Service (DDOS) attack using ns-2 network simulator. They also found that the class-based queuing algorithm can guarantee the legitimate user's bandwidth even under the DDOS attack. However, their focus is on the impact of DDOS on the queuing algorithm and the attack method in their simulation is simple and fixed. Chi, S.D., P.J. (2001) developed a hierarchical and modular modeling environment to simulate the cyber attacks. Compared with Lau, F., R.H. (2000), Chi, S.D., P.J. (2001)'s simulation environment is more complicated. The environment consists of attacker model, analyzer model and network models including node model, router model and topology model.

Some studies were devoted to model the behavior of the attackers. Kotenko, I., E.M. (2003) implemented an attack simulator that provides a hacker-agent and network-agent system based on malefactors intention modeling, ontology-based attack structuring and state machines specification of attack scenarios. However, their attack action was limited by a pre-defined parameters such as Number of attack Steps, Percentage of Intention Realization, Percentage of Firewall Blocking, and Percentage of Attack Realization. Based on a discrete-event simulation model, Kuhl, M.E., J.K., K.C. (2007) incorporated IDS module in their attack simulator so that the simulator can use the IDS alerts to test and evaluation the attack and the system securities.

2.2 Application of Machine Learning and Statistical Method in Network Security

Machine Learning is able to understand the difference between normal and anomalous patterns by being trained with training data, and generate classifiers. These classfiers can be used to detect network attacks. Therefore, there are numerous efforts on applying Machine Learning on security, especially anomaly detection.

Debar H., B.M., S.D. (1992) utilized the Artificial Neural Networks to learn normal traffic pattern and classify the new traffic into attacks or normal traffic. Lane, T., B.C. (1997) used similarity measurement to compare the current user input with the profiled user's sequence of action (Unix commands) so that the anomaly can be detected. Labib, K., R.V. (2002) used a Self-Organized-Map to cluster the real-time network traffic data so that the abnormal traffic can be easily spotted in a GUI interface. Konrad, R., P.L. (2008) proposed a framework using kernel-based learning for sequence similarity measurement. Li, X., Y.N. (2003) presented an algorithm classifier for intrusion detection, using different features of raw activity data in computer network systems and different sizes of observation windows. Stein, G., C.B., W.A., H.K. (2005) added generic algorithm to prune the output of their Decision Learning Tree algorithm, improving the efficiency.

Based on the assumption that the attack traffic is statistical different from the normal traffic, many research works are also done on intrusion detection by applying multivariate statistical methods such as *Principle Component Analysis* or *Cluster Analysis*. Compared with other IDS algorithm such as signature-based methods or data mining-based methods, the statistical methods does not have to rely on labelled training data. Consequently the statistical methods can do better on detection of new types of attacks.

Shyu, M.L., C.S., S.K. C.L. (2003) proposed an Intrusion Detection System by using *Principle Component Analysis*, in which the difference of a normal instance from an anomaly one is transferred into the distance in the Principle Component Space. Stefano, Z., S.M. (2004) proposed a two-tier architecture for intrusion detection. The first tier uses unsupervised clustering algorithm to reduce the network traffic down to a tractable size. The clustered data then is to passed to the second tier, a traditional anomaly detection algorithm. In his book, Eskin, E., A.P., P.L., S.S. (2002), provided improvements in accuracy by making the clusters be adaptive to changing traffic patterns. Leung, K., L.C. (2005) developed a system based on density-based and grid-based high dimensional clustering algorithm.

CHAPTER 3. Design and Implementation of the *ART* Framework

3.1 Introduction

In this chapter, we describe the design and implementation of the Auto Red Team (*ART*), an attack automation framework based on Decision Learning Tree. We start from the enumeration of the design goal, subsequently the basic concept of the traffic sniffing, training data composing, and attacking strategy generating through Decision Tree Learning. Then we detail on the Design and implementation of the sub-modules of the framework. Finally, we discuss the efficiency and accuracy of this framework, and issues such as porting this software to other platforms.

3.2 Design Goal

The *ART* framework should should meet the following criteria.

Launch multiple round of attacks The *ART* framework should have a attack program library so that multiple attacks can be selected and launched against the targets. The framework should be able to maintain and update the current attack program library to keep up with the the quick evolving attack in real world.

Self training ability The framework should be capable of training itself by incorporating certain Artificial Intelligence approach so that it can generate corresponding *Attack Strategy* for different type of targets.

Differentiate between success and failure of the attack result The *ART* framework should be able to tell whether the attack is successfully complete or failed by analyzing the data generated from the traffic between the attack program and the target. Since the attacker plays the vanilla attack, the attacker has no knowledge about the attack program. So the attacker must rely on the traffic between the attack program and the target to make decisions.

Choose next step based on the response from target The *ART* framework must be capable of making choice of next steps from one of the followings: stop attacking if the last attack succeeds, or change to other type attack because the target seems to be immune on the current types of attack, or stay in the same type of attack but change a different attack program.

According to the goals, the *ART* framework should be able to sniffer traffic between attacker and targets, compose training data based on the captured traffic data, generate / update the attack strategies by feeding those training data into a Decision Learning Tree model, and apply these strategies onto the attacker.

3.3 Infrastructure of the *ART* framework

The *ART* framework consists of three major parts, attack program library, attacker trainer, and attacker. There are two modes of the *ART* – Training Mode and Attack Mode. In Training Mode, the framework launches multiple attacks to the target, and generate training data from the captured packets between the attacker and the target. These training data will be fed into a Decision Learning Tree module to generate a set of attack rules named *Attack Strategy*. In Attack Mode, the attacker incorporates the generated *Attack Strategy* and launches "smart" exploits according to the target response and the *Attack Strategy* until the exploit succeeds or all exploits fail. Figure 3.1 shows the infrastructure of the *ART*. The detailed description of the sub-models is listed below.

Attack Database The Attack Database stores all the attack programs that are used against the targets. Instead of re-inventing the wheel, and also for concept-prove purpose, we choose the exploit library of *Metasploit* [Metasploit (2008)] as the attack database of the

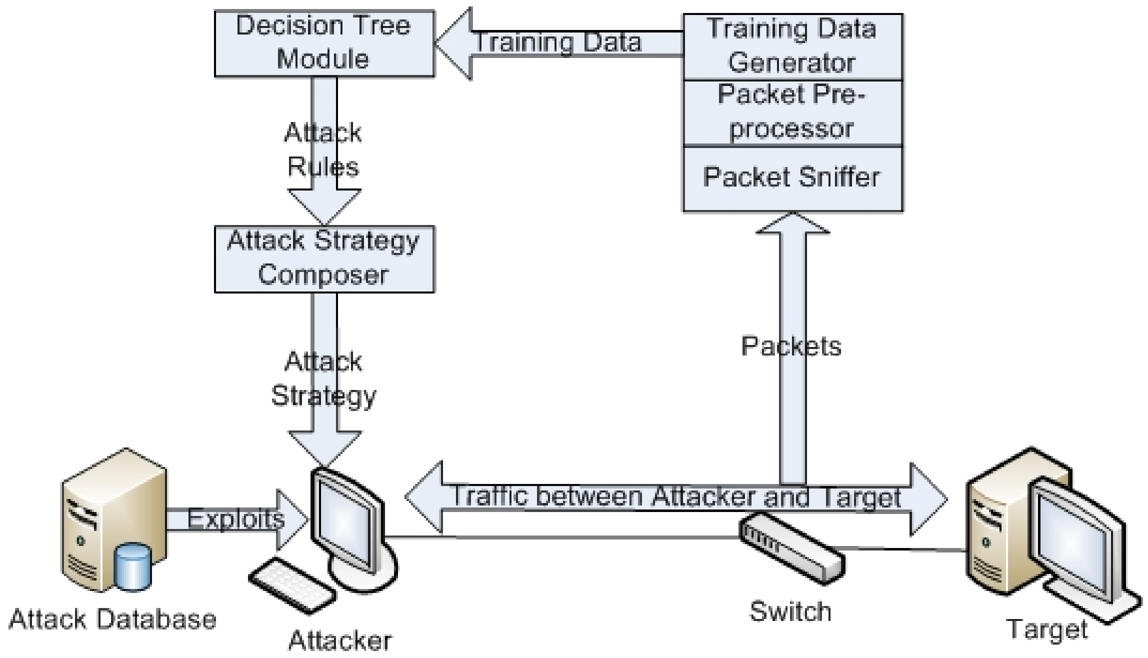

Figure 3.1 Infrastructure of the *ART* framework

ART. *Metasploit* is a development platform for creating security tools and exploits. It provides various exploit programs, for example, *Microsoft IIS 5.0 IDQ Path Overflow* for Windows and *Samba nttrans Overflow* for Linux. *Metasploit* also provides the payloads that perform tasks upon a successfully penetration. Typical payloads are `adduser` to add an Administrator, and `reverse_http` to open an reverse HTTP tunneling stager.

Attacker The Attacker selects an exploit from the Attack Database, and launches the exploit to the target. The attacker should only play Plain-Vanilla exploit, which means that the Attacker has little knowledge of the inside of the exploit, making the Attacker rely on the response and the *Attack Strategy* to tell the exploit result. According to the response from the target and the *Attack Strategy*, the Attacker decides whether the exploit succeeds or not, and takes the next step: launching an exploit that belongs the same category with last one, or launch an exploit in different category, or stop exploiting after trying all possible means, or stop for a successful exploit.

Target The Target is a computer that is subject to be attacked by Attacker. Currently we use a PC with Windows XP Professional SP2 as the Target.

Packet Sniffer The Packet Sniffer audits the traffic between the Attacker and the Target, captures the all the packets, and pass the packets to the Packet-Preprocessor.

Packet Pre-processor The Packet-Preprocessor analyzes the packets and generates the statistic data of the exploit such as total packet of an exploit, average packet length, exploit duration, TCP flags and other statistic data. During the attack, these statistic data are updated by newly captured packets.

Training Data Generator The Training Data Generator only works under the Training Mode. The Training Data Generator processes the Packet Pre-processor's data collection at the end of every exploit, and combined all the data into a single line of training data. It also asks for the human being's conclusion on the result of the exploit: *Succeed*, or *Failed and Launch Next Exploit in Same Category*, or *Failed and Launch Next Exploit in Different Category*. After consolidating the human being's conclusion and the statistical data into a line of Decision Tree training data, the Training Data Generator will notify the Attacker to launch another exploit to generate new training data or stop according to human being's input.

Decision Tree Module The Decision Tree Module is a modified C4.5 Decision Learning Tree software [J.R. Quinlan (1992)]. The Training Data Generator feeds multiple training date items into the Decision Tree Module. The Decision Tree Module then generates a set of attacking rules.

Attack Strategy Composer The Attack Strategy Composer extracts the attacking rules from the output of Decision Learning Tree Module, transforms those rules into a set of if-else statements named *Attack Strategy*, and injects the *Attack Strategy* into the Attacker. Thus, the Attacker will become a smarter attacker to perform more effective exploit on targets.

There is a loop in the *ART* framework: the Attacker launches attacks according to the current *Attack Strategy*; the packets between the Attacker and target will be used to generate new *Attack Strategy* by the Decision Learning Tree Module; The newly generated *Attack Strategy* will be incorporated into the Attacker to make a smarter Attacker, and so on.

3.4 implementation of the *ART* framework

In this section, we will discuss the implementation detail of the *ART* framework. We start with the Attacker and Attack Database, we then describe the core part, Packet Processor including Packet Sniffer, Packet Pre-processor, and Training Data Generator. Finally, we will introduce how the Attack Strategy Composer transforms the output from Decision Learning Tree to Perl version if-else statements.

3.4.1 Implementation of Attack Database and Attacker

We chose the exploit library of *Metasploit* version 3.1 [Metasploit (2008)] as the attack database of the *ART*. The *Metasploit*, a popular development platform for development of security tools, is used by many network security professionals to perform penetration tests, to verify security patches, and to perform regression tests. The framework is developed in Ruby programming language. Some of its components are also written in C or assembler.

The *Metasploit* provides large exploit database for various Operating Systems including in others, Microsoft Windows series, Linux, Mac OSX and various Unix systems. Currently, we only use its exploits for Windows as our concept-prove implementation. All the Windows exploit categories are listed in Table 3.1.

The *Metasploit* also provides both GUI and CLI. We chose a CLI command as the interface between the *Metasploit* and the Attacker. The command is `msfcli` with syntax as `./msfcli <exploit_name> <option=value> [mode]`. Table 3.2 shows the available modes of this CLI command.

The Attacker is an exploit launcher called by the Training Data Generator during Training Mode and called by users during the Attack Mode. Because *Metasploit* does not provide a

Table 3.1 Exploit Category of *Metasploit* Library for Windows

Category	**Attacking Targets**
Antivirus	Antivirus softwares (ex. Symantec Remote Management)
Backup	System backup softwares (ex. Veritas Backup Service)
Browser	Windows browsers (ex. Internet Explore)
RPC/DCOM	Windows RPC or DCOM (ex. Message Queueing Service)
Driver	Various drivers (ex. Broadcom Wireless Driver)
Firewall	Windows firewalls (ex. ISS PAM.dll ICQ Parser)
FTP	Various FTP servers (ex. WS-FTP Server 5.03)
HTTP	Various HTTP servers (ex. Apache mod_jk 1.2.20)
IIS	Windows IIS (ex. IIS 5.0 Printer Host)
IMAP	IMAP Servers (ex. Novell NetMail IMAP)
ISAPI	IIS ISAPI (ex. IIS's nsiislog.dll)
LDAP	LDAP Services (ex. IMail LDAP Service)
SMB	SMB Services (ex. Workstation Service)
MSSQL	SQL Server (ex. Microsoft SQL Server Resolution)
Misc	Miscellaneous Services (ex. DirectX DirectShow)

command line query interface, we hard-coded the name of the exploits into Attacker so that it can launch these exploits by calling the `msfcli` with the correspondent exploit name. The procedure of launching an exploit is listed below:

1. Select an exploit according to its *Attack Strategy*;

2. Create a pipe;

3. Fork a process to execute the selected exploit by calling `msfcli`;

Table 3.2 The Modes of `msfcli`

Mode	**Description**
Summary	Show information about this module
Options	Show available options for this module
Advanced	Show available advanced options for this module
IDS Evasion	Show available ids evasion options for this module
Payloads	Show available payloads for this module
Targets	Show available targets for this exploit module
Actions	Show available actions for this auxiliary module
Check	Run the check routine of the selected module
Execute	Execute the selected module

4. Profile the output of the exploit through the created pipe;

5. Notify the the caller (Training Data Generator or user) when the exploit ends.

3.4.2 Implementation of Packet Processor

The Packet Processor consists of three sub models – Packet Sniffer, Packet Pre-processor, and Training Data Generator, which work together to accomplish tasks of traffic audit and training data composition. We detail on the implementation in the following.

Packet Sniffer The Packet Sniffer captures the packets between the Attacker and the target by using PCAP library [PCAP Library (2008)], an open source Application Programming Interface (API) for network packet capture. Before the Attacker launches the exploit, the IP addresses of Attacker and target are sent to the Packet Sniffer. The Packet Sniffer then calls PCAP API to set up the packet capture device. The main API sequence that it calls is: `pcap_open_live` to initiate the capture device, `pcap_compile` and `pcap_setfilter` to compile and set capturing filter, and `pcap_dispatch` to designate to call-back function upon the event of capturing packet. Whenever there is a packet captured, the Packet Sniffer forwards it to Packet Pre-processor.

Packet Pre-processor Layer by layer, the Packet Pre-processor breaks down every packet that is forwarded from Packet Sniffer, starting from Link layer, IP layer, to TCP/UDP layer. During the break down, the Packet Pre-processor profiles the packet length in IP layer, the TCP flags, Source and Destination Port, and updates the statistic data starting from the begin of exploit – packet counts, total packet size, average packet size, maximum packet size, minimum packet size, standard deviation of packet size, current interval, total duration, average interval, maximum interval, minimum interval, and standard deviation of the packet interval.

Training Data Generator The Training Data Generator is blocked by function `wait(the process id of the Attacker)`. Upon the completion of the exploit, the Attacker process exits, unblocking the Training Data Generator that will extract the statistic data from Packet Pre-processor and the exploit conclusion from user. After consolidating all the data and conclusion into a single line item, the Training Data Generator re-launches an Attacker to profile a new training data item by forking a new process. However, if the Training Data Generator receives a "start-to-train" command from user, it will feed all the saved training data into the Decision Learning Tree by forking a new process that will run the Decision Learning Tree Module.

3.4.3 Implementation of the Decision Learning Tree and Attack Strategy Composer

The modified C4.5 Decision Learning Tree software is originally written by J.R. Quinlan (1992). The original release is written for BSD 4.3. To port this release onto Linux platform, we made some changes. For example, we changed function pair `calloc` and `cfree` to `malloc` and `free`.

Since the Decision Learning Tree module can not generate output that can be directly incorporated into a program, we need to to change the module's output from human readable rules to certain if - then statements that can be incorporated into some executable script such as Perl or source codes such as C source code. So we constructed a converter written in Perl to make such conversion. For example, The Attack Strategy Composer converts the following rule shown in Figure 3.2 to the statement shown in Figure 3.3. The perl code to convert the above outputs is listed in Appendix 5.4.

```
Rule 5:
          flag = APSF
          total_packets <= 11501
          max_interval_size > 2.2539
          ->  class SUCCEED  [63.0%]
```

Figure 3.2 Example Outputs from `c4.5rules`

```
if{ flag == "ASPF" && total_packets <= 11501 && max_interval_size >= 2.2539 }{
    succeed();
}
```

Figure 3.3 Example Outputs from Attack Strategy Composer

The final output of Attack Strategy Composer is an executable Perl script that becomes a newer, smarter Attacker.

3.5 Discussion

In this chapter, we described the design goals, software infrastructure of *ART* framework, and the implementation details. The framework is an attack automation platform incorporating existing softwares such as *Metasploit* and C4.5 Decision Learning Tree module. The *ART* framework has the following hightlights:

Minimum Dependency between Exploits and the Platform By playing "Plain-Valinlla" exploit against the various targets, the *ART* can easily be expanded and upgraded because the dependency between the exploit and the platform is reduced to the minimum. Therefore, the *ART* can switch to another Attack Database from the *Metasploit*, the currently used Attack Database, if needed.

Artificial Intelligence The *ART* incorporates one of the popular Machine Learning methods, C4.5 Decision Learning Tree module, making the *ART* a smarter attack platform than others without Artificial Intelligence technologies. According to the experiment in next chapter, the accuracy of the decision tree based attacker can be as high as 90.5%.

CHAPTER 4. Experiment: Using Exploits on Windows to Train an Attacker

To verify the *ART* framework, we conducted an experiment that generated 42 training data by launching exploits against a Windows XP machine. We then fed these data into a C4.5 Decision Tree Module, generating a set of rules. These rules, named *Attack Strategy* was fed into a Perl script named Attack Strategy Composer that transforms those rules into a set of *if-else* statements. The Attack Strategy Composer also incorporated those if-else statement into a Perl script, making a new attacker. To exam these rules, we also ran a statistic method, *Principle Component Analysis* on those training data by using R [R Project (2008)]. The result shows that the generated rules partially match the Principle Components generated by the *Principle Component Analysis*.

The rest of this chapter is organized as the following: First of all, we describe the experiment environment including software and hardware settings in Section 4.1. Secondly we present the experiment result – the output tree from the Decision Learning Tree module and the generated rules in Section 4.2. We then apply *Principle Component Analysis* onto the training data and compare the result with the output of Decision Learning Tree in Section 4.3. Finally we give further discussion in Section 4.4.

4.1 Experimental Environment

The experiment settings and the experiment procedure are listed below:

Software and Hardware Settings All the modules of the *ART* framework are implemented on one Intel Celeron 2.4 GHZ Personal Computer with Linux 2.6 Kernel, GCC 4.0 and Perl 5.8. In this experiment, we chose another Intel Celeron 2.4 GHZ with Windows XP

Professional as the target machine for the concept-prove purpose. The Attacker and the target were connected by a 100 MHZ ethernet hub.

Experiment Procedure We chose 42 exploits from the Attack Database and had the Attacker launch them one at a time against the target. Upon every completion of the exploit, we decided whether the exploit succeeded or failed by checking the output of the exploit and the status of the target machine. We then instructed the Packet Processor to conclude the exploit result. After all the 42 exploits finished, we fed the generated training data into a C4.5 Decision Learning Tree module. The module generated a set of rules to guide the Attacker. These rules, named *Attack Strategy*, were incorporated by the Attack Strategy Composer into a new Attacker. Thus, a cycle of launching attack, generating training data, feeding Decision Learning Tree to obtain new *Attack Strategy*, and finally generating a new Attacker is finished.

Training Data There are 13 metrics in a single training data. These metrics and their description are shown Table 4.1

Table 4.1 Metrics of Training Data

Metrics	**Description**
`flag`	TCP flags appear during the exploit
`packet_count`	Number of captured packets
`total_packets`	Accumulated packet size
`ave_packet_size`	Average packet size
`max_packet_size`	Maximum captured packet size
`min_packet_size`	Minimum captured packet size
`std_dev_packet_size`	Standard deviation of packet size
`interval_count`	Number of interval between packets
`total_duration`	Total time of the exploit
`ave_interval`	Average interval
`max_interval`	Maximum interval
`min_interval`	Minimum interval
`std_dev_interval`	Standard deviation of interval

4.2 Experiment Results

Figure 4.1 presents the Decision Tree Output, Figure 4.2 shows the rules generated by Decision Learning Tree Module, and Figure 4.3 demonstrates how the Attack Strategy Composer transferred the rules, shown in Figure 4.2, into a set of if-then statements.

As seen in Figure 4.1, the C4.5 Decision Learning Tree module produced a five-level decision tree. The root of the generated tree is total packet size followed by TCP flags, total duration and etc. The error rate of the decision tree is 9.5%. The rules generated of the C4.5 Decision Learning Tree module also generated four rules regarding to how to conclude the exploit result. The error rate of the examination on these rules are 23.8%. Besides, we also have the following observations:

- As seen in Figure 4.4, the exploits that have total packet size greater than 11,000 are all failed exploits. So it is not surprised to see that the C4.5 Decision Learning Tree module treats the exploit with larger total packet size as first criteria of fail or succeed.
- The "R" in TCP flags stands for a reset signal from target machine, which means that the exploited port is closed. An exploit attacking an closed port will not succeed in most cases. So it is not surprised that all decisions on the flags with "R" are classified as failed attempt.

4.3 Statisitical Analysis

4.3.1 Introduction of *Principle Component Analysis*

The training data can be treated as a matrix with 42 row and 13 columns. Each row represents an exploit and each column represents a metric of the exploit, such as `total_packets` or `ave_interval` shown in Table 4.1. The 42×13 matrix makes it difficult to understand the relationship between the training data and the generated tree. To tackle such a problem, we apply a multivariate statistical method, *Principle Component Analysis* on the training data to verify the generated rules.

```
C4.5 [release 8] decision tree generator        Tue Aug 19 18:24:25 2008
----------------------------------------
Read 42 cases (13 attributes) from target_response.data
Decision Tree:
total_packets > 11501 : FAILED_TRY_SAME_TYPE (5.0)
total_packets <= 11501 :
|   flag = APRS: FAILED_TRY_OTHER_TYPE (0.0)
|   flag = AP: FAILED_TRY_OTHER_TYPE (8.0/2.0)
|   flag = APF: SUCCEED (2.0)
|   flag = AS: FAILED_TRY_OTHER_TYPE (3.0/1.0)
|   flag = APRSF: FAILED_TRY_OTHER_TYPE (1.0)
|   flag = A: SUCCEED (1.0)
|   flag = ARS: FAILED_TRY_OTHER_TYPE (5.0)
|   flag = APSF:
|   |   max_interval_size > 2.2539 : SUCCEED (3.0)
|   |   max_interval_size <= 2.2539 :
|   |   |   packet_count > 24 : FAILED_TRY_SAME_TYPE (2.0)
|   |   |   packet_count <= 24 :
|   |   |   |   total_duration <= 0.045358 : FAILED_TRY_SAME_TYPE (2.0)
|   |   |   |   total_duration > 0.045358 : FAILED_TRY_OTHER_TYPE (6.0)
|   flag = APS:
|   |   packet_count <= 6 : FAILED_TRY_OTHER_TYPE (2.0)
|   |   packet_count > 6 : SUCCEED (2.0/1.0)
Tree saved
Evaluation on training data (42 items):
       Before Pruning           After Pruning
      ----------------   ---------------------------
      Size      Errors   Size      Errors   Estimate
        20    4( 9.5%)     20    4( 9.5%)    (41.9%)   <<
```

Figure 4.1 Generated Tree from C4.5 Decision Learning Tree Module

```
Read 42 cases (13 attributes) from target_response

Rule 4:

        flag = APSF

        total_packets <= 11501

        max_interval_size > 2.2539

        ->  class SUCCEED  [63.0%]

Rule 6:

        flag = APF

        ->  class SUCCEED  [50.0%]

Rule 11:

        total_packets > 11501

        ->  class FAILED_TRY_SAME_TYPE  [75.8%]

Rule 8:

        packet_count <= 6

        ->  class FAILED_TRY_OTHER_TYPE  [67.3%]

Default class: FAILED_TRY_OTHER_TYPE

Evaluation on training data (42 items):

Rule  Size  Error  Used  Wrong       Advantage

----  ----  -----  ----  -----       ---------

   4     3  37.0%     3  0 (0.0%)     3 (3|0)    SUCCEED

   6     1  50.0%     2  0 (0.0%)     2 (2|0)    SUCCEED

  11     1  24.2%     5  0 (0.0%)     5 (5|0)    FAILED_TRY_SAME_TYPE

   8     1  32.7%    17  3 (17.6%)    0 (0|0)    FAILED_TRY_OTHER_TYPE

Tested 42, errors 10 (23.8%)   <<

          (a)  (b)  (c) <-classified as

         ---- ---- ----

            5    3      (a): class SUCCEED

                22      (b): class FAILED_TRY_OTHER_TYPE

                 7    5 (c): class FAILED_TRY_SAME_TYPE
```

Figure 4.2 Generated Rules from C4.5 Decision Learning Tree Module

```
The output of the Attack Strategy Composer is .....
if((flag=="APSF")&&(total_packets <= 11501)&&(max_interval_size > 2.2539)){
        succeed();
}
if((flag=="APF")){
        succeed();
}
if((total_packets > 11501)){
        failed_try_same_type();
}
if((packet_count <= 6)){
        failed_try_other_type();
}
```

Figure 4.3 Rules Transferred by Attack Strategy Composer

Principle Component Analysis is a common approach of multivariate statistical data analysis. Through it, the covariance of data are presented in terms of a few fundamental but un-observable, random quantities. These quantities are called Principle Components. *Principle Component Analysis* is suitable to extract and simplify the relationship between the generated rules and the training data. Generally speaking, *Principle Component Analysis* is a repetitive process that uses linear regression to find a new set of dimension, the number of which usually is less than the dimension number of the original data space. The new set of dimension is better aligned with the data. By *Principle Component Analysis*, the 42×13 training data matrix can be transferred into a less dimension data space.

The original $n \times p$ data matrix can be seen as a set of observable random vectors $X'_{p\times 1}$ with mean μ and covariance matrix Σ. Each vector, X can be written in linear combinations shown in Equation 4.1.

$$X_1 = u_1 + l_{11}PC_1 + l_{12}PC_2 + ... + l_{1m}PC_m + \epsilon_1$$

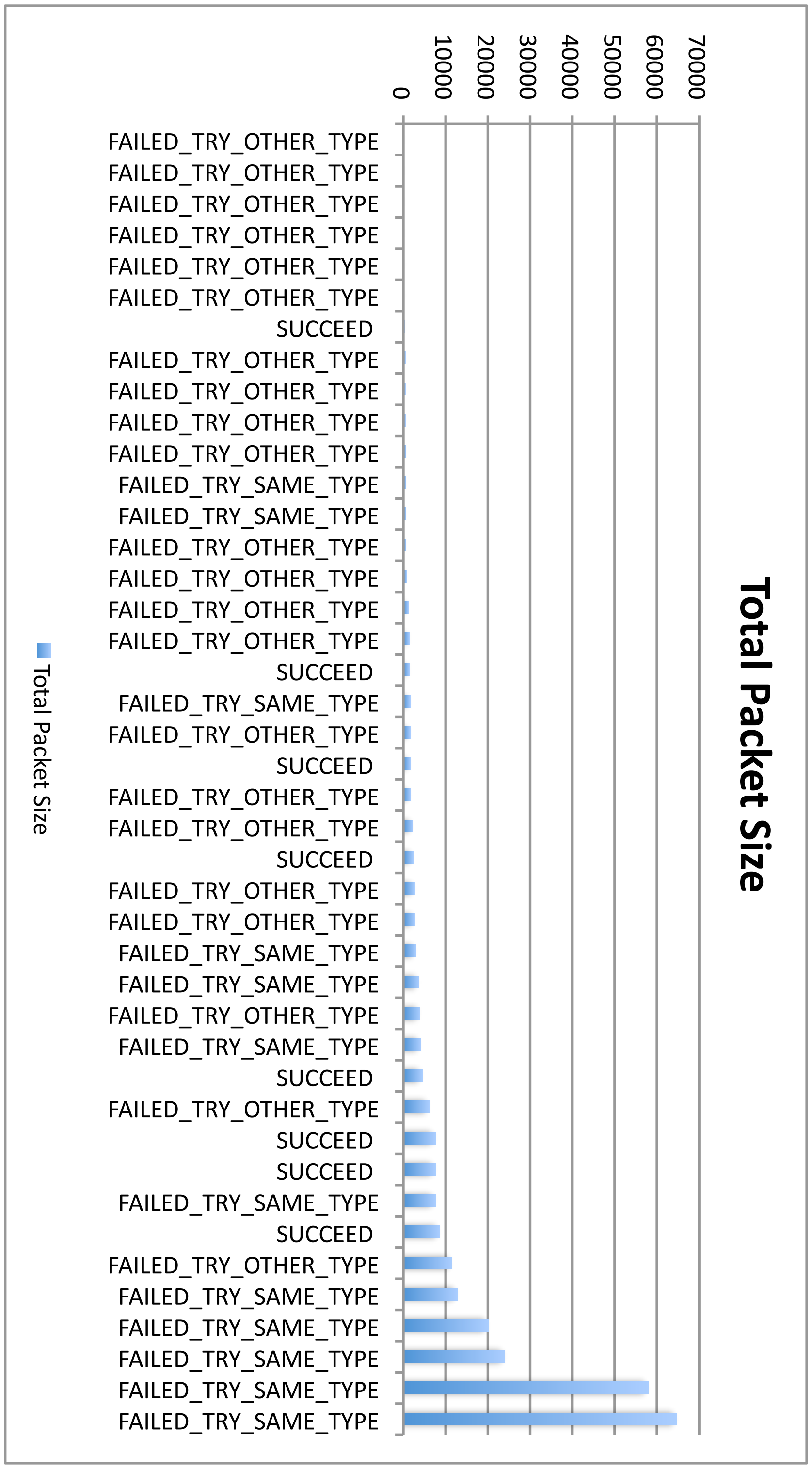

Figure 4.4 Total Packet Size v.s. Exploit Result

$$\begin{aligned} X_2 &= u_2 + l_{21}PC_1 + l_{22}PC_2 + ... + l_{2m}PC_m + \epsilon_2 \\ ... \\ X_p &= u_p + l_{p1}PC_1 + l_{p2}PC_2 + ... + l_{pm}PC_m + \epsilon_p \end{aligned} \qquad (4.1)$$

In Equation 4.1, symbol μ_i is the mean of variable i, while ϵ_i represents ith error, PC_j for jth Principle Component, and l_{ij} is the loading of the ith variable on the jth component. This equation is the model of *Principle Component Analysis*. The power of *Principle Component Analysis* is that – the significant portion of variation in the original data can be explained by the first few Principle Components with the rest of components discarded. For our training data, we can expect to use less number of Principle Components to represent the 42×13 data.

4.3.2 Applying *Principle Component Analysis* on the Training Dta

We wrote a script of R language [R Project (2008)] to apply the *Principle Component Analysis* on the training data. Before applying *Principle Component Analysis*, we deleted the 12th metric, `min_interval` because the the data of this metric in the matrix are all zero. Figure 4.6 shows the first four Principle Components. The cumulative proportion of the variance represented by these four components is 0.835%, making it is enough to use these four components to explain the significant variation in the original data. However, the complete variance of Principle Components 1 to 10 can be found in Figure 4.5.

To understand the relationship between the metrics of training data and these four components, we need to investigate the loadings of Component 1 to 4, which are shown in Table 4.2. The projection of these metrics on Component 1 and 2, which is shown in Figure 4.7, demonstrates the variation distribution, too. Based on these figures, we have the following observations:

- According to Table 4.2, only `flag` and `ave_interval` have positive value in the loading of Component 1, suggesting their variation direction is consistence with that of Component 1, while other metrics take opposite variation direction. Seen from Figure 4.7, the direction difference of `flag` partially explains the fact that the C4.5 Decision Learning

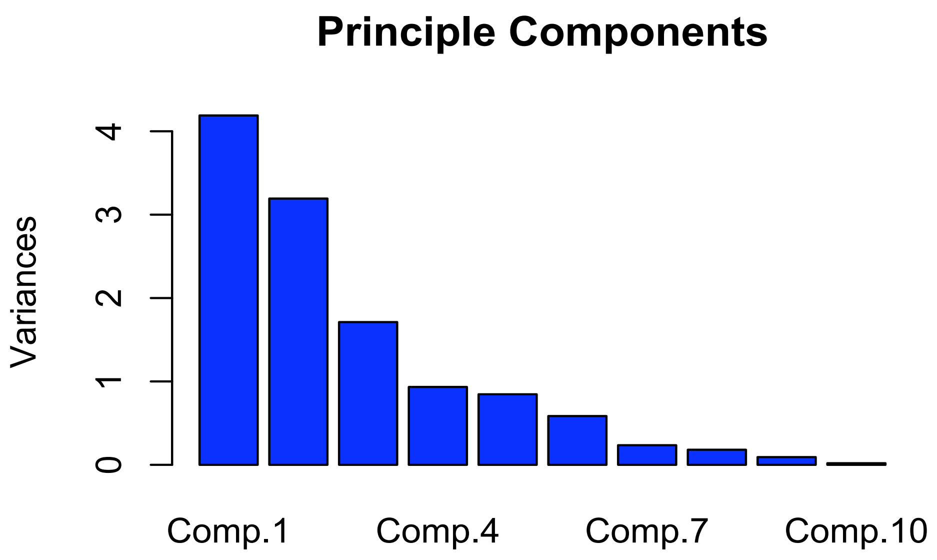

Figure 4.5 Variance of Principle Components

Tree also incorporated the `flag` in its every generated rule. However, why there is no rule to incorporate the `ave_interval` that also has positive value needs more study.

- According to Figure 4.7, V11: `max_interval` and V13: `std_dev_interval` take opposite variation direction against the V1: `flag`. On the other hand, the C4.5 Decision Learning Tree also put the `max_interval` into the third level of the generated tree (See Figure 4.1) and incorporated it into its *Rule 4* (See Figure 4.2). Similarly, why there is no rule to incorporate the V13: `std_dev_interval` needs further investigation.

```
                        Comp.1    Comp.2    Comp.3     Comp.4
Standard deviation     2.0468856 1.7870018 1.3083739 0.96638509
Proportion of Variance 0.3491451 0.2661146 0.1426535 0.07782501
Cumulative Proportion  0.3491451 0.6152597 0.7579132 0.83573823
```

Figure 4.6 Result of *Principle Component Analysis* on Training Data

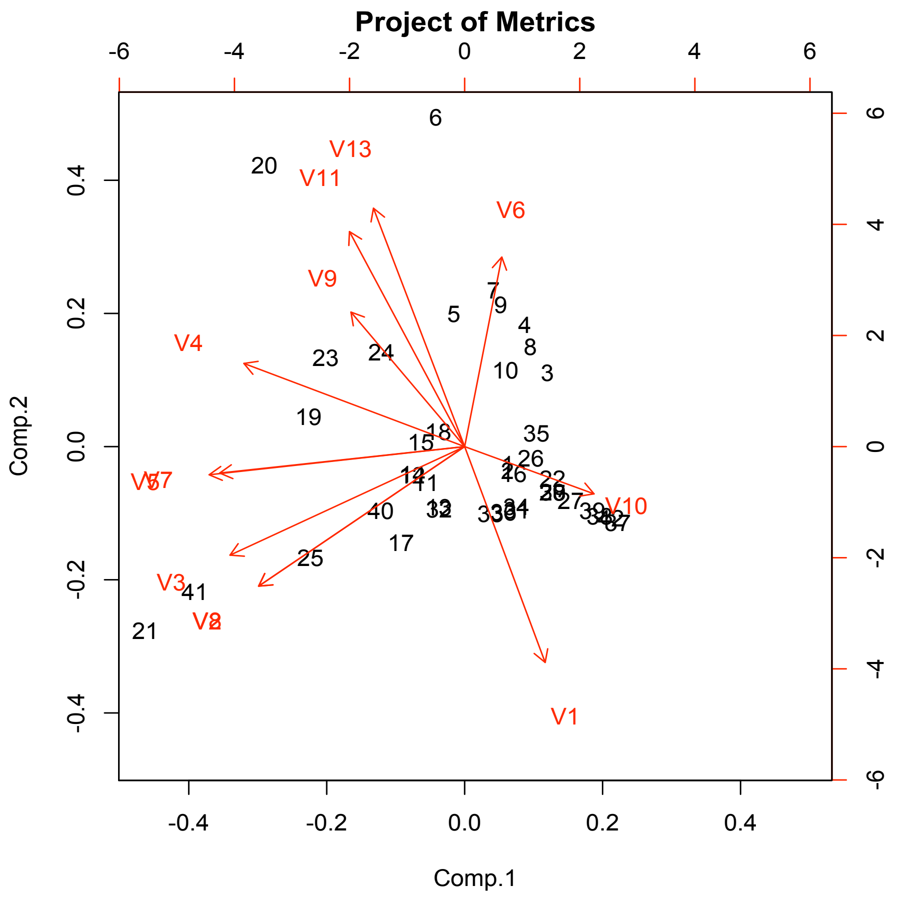

Figure 4.7 Projection of Metrics on Principle Component 1 and 2

Our observation suggests that: somehow, the generated tree and rules reflects the major variation of the Original training data. However, the Decision Learning Tree algorithm is a non-linear system. So the linear *Principle Component Analysis* should not be able to completely explain the output from the C4.5 Decision Tree.

4.4 Discussion

In this chapter, we presented the experiment result including the generated decision tree and the rules that will be incorporated into the Attacker as the new *Attack Strategy*. To further investigate the relationship between the generated decision tree and the metrics, we applied *Principle Component Analysis* on the training data. The loadings of the Principle Components

Table 4.2 Loadings of the First Four Principle Components

Metrics	**Comp. 1**	**Comp. 2**	**Comp. 3**	**Comp. 4**
V1: `flag`	0.132	-0.419	-0.264	0.241
V2: `packet_count`	-0.337	-0.271	0.392	0.000
V3: `total_packets`	-0.384	-0.211	0.262	0.000
V4: `ave_packet_size`	-0.361	0.162	-0.355	-0.111
V5: `max_packet_size`	-0.418	0.000	-0.313	-0.117
V6: `min_packet_size`	0.000	0.368	0.384	-0.371
V7: `std_dev_packet_size`	-0.401	0.000	-0.394	0.000
V8: `interval_count`	-0.337	-0.271	0.392	0.000
V9: `total_duration`	-0.186	0.261	0.000	-0.527
V10 : `ave_interval`	0.212	0.000	-0.156	0.000
V11: `max_interval`	-0.189	0.418	0.000	0.531
V13: `std_dev_interval`	-0.149	0.463	0.000	0.448

and the projection of the metrics on Principle Component 1 and 2 suggest that the Decision Learning Tree algorithm somehow is consistent with the linear space of the training data.

However, this experiment only use a Windows XP machine as target machine, not reflecting the diversity of the targets in real world. Our next step is to use the Attacker against other Operating Systems to get as diverse as possible training data.

CHAPTER 5. Summary and Discussion

In this chapter, we summarize the thesis, review the thesis contributions, and discuss the future work.

5.1 Summary

This thesis describes a framework named *ART*, standing for Auto Red Team, which is a platform utilizing Decision Tree Learning technology to realize attack automation. The key idea is to audit the traffic between the attacker and the target machine, then apply Decision Tree Learning methods on the audit data to generate a set of rules, then incorporate these rules into the attacker, making it be capable of launching attack sequence according to the response from the target machine.

We were motivated by stating the importance of attack automation on Red Teaming that can alleviate security experts and system administrators' burden by taking over certain redundant tasks in penetration test. We provide the background of Red Teaming and attack automation and point out that current Red Teaming softwares can only perform simple attack automation and lack of adaptability, and extensibility. We then proposed our idea to incorporating Decision Tree Learning techniques in order to build a smarter framework. The goal of this thesis is therefore to develop such a software that is adaptive and extensive in penetration test automation.

We then reviewed the current research works in Attack Automation, the application of Decision Learning Tree in network security such as Intrusion Detection, and the application of statistic models such as *Principle Component Analysis* of *Cluster Analysis*. After that, we present the design and implementation details of our platform, *ART* framework. We also presented our case study of the *ART* by conducting experiments on it, showing that the *ART* is able to perform traffic auditing, to extract new attack rules, named *Attack Strategy* that are generated from a Decision Tree Learning module, and to make new attacker that is guided by those *Attack Strategy* and is capable of launching attack sequence according to the response from the target machine. Beside basic data analysis on the experiment data, we also apply a statistical method, *Principle Component Analysis* on the experiment data to verify the generated rules. Although the *Principle Component Analysis* can not completely explain the rules generated by the Decision Tree module, some convincing explanations on the relationship between those rules and Principal Components were given.

5.2 Thesis Contributions

The intention of the *ART* framework is not to totally replace the network security experts. On the contrary, we do not believe that the security assurance task would become a fully automatic, plug-n-play job, at least in the near future. The main purpose of the *ART* framework is to assist network security professionals in assessing the vulnerability of network systems by taking over some simple and replicable penetration tasks.

The key contributions of our research works are listed as below:

Artificial Intelligence The *ART* incorporates a Machine Learning technology, C4.5 Decision Tree, making the *ART* a smarter attack platform than others without Artificial Intelligence technologies.

Polymorphic Attacks Based on an attack database and guided by attack strategies, the *ART* can launch multiple types attacks against the target in an attack scenario. The attack database can be updated to keep up with the development of attack technologies.

Portability among Platforms The inter-dependencies among attacker, attack database and the Decision Tree module are minimized. By playing plain-vanilla attack, the attacker does not need to know the details of the individual exploit program, so that the switch or upgrade the attack database would not have impact on the attacker, and vice versa. The

Decision Tree module receives the standardized training data from a Training Data Generator, and the generated rules by the module will be incorporated into a new attacker by a Attack Strategy Composer. Thus, there is no direct connection between the attack database and the attacker, minimizing the dependency between them. Consequently, migration or upgrade of the *ART* framework becomes less stressful.

5.3 Future Work

There are two interesting and important future directions:

Generating Rules based on Hybrid Data : Currently the *Attack Strategy* is generated from the training data on single type of Operating System. We believe that it would make the *Attack Strategy* more robust if we could conduct experiments on various Operating System and collect training data from those experiments.

Incorporating Complier Techniques : Currently the *Attack Strategy*s are transformed from human readable form to Perl's if-else statement by the Attack Strategy Composer. If we could make those *Attack Strategy*s be incorporated into C source code by modifying a compiler instead of using Perl's script, the new attacker would be still built from C source code and can reflect the new *Attack Strategy* in real-time, just like the *Just in Time* techniques in Java Virtual Machine design. Thus we can expect an improvement on efficiency of the new attacker.

5.4 Closing Remarks

This thesis documented our research in developing and applying Decision Tree Learning techniques to the challenging problem of attack automation applications. A framework named *ART* that is for such an application was designed, implemented, and evaluated for this thesis research. While *ART* has shown great promises, there are still open issues for future research.

APPENDIX:

The Source Code of Attack Strategy Composer

Program 1 Source Code of Attack Strategy Composer (part-1)

```
#!/usr/bin/perl
my $file_name = $ARGV[0];
print "The data file is $file_name.\n";
open(DATA_FILE, "<" . $file_name) or die "can’t open $store_path $!";
# state: 0: initial state, 1: found if state ments, 2: found action
my $state = 0;
# constant of header and signs
my $IFHEADER = "if(";
my $TAIL = "){";
my $END = "}\n\n";
my @outputlines;
my $inputline = "";
my @filelines = <DATA_FILE>;
foreach my $line (@filelines){
        $line =~ s/[\n,\r]//g;
        $line =~ s/^\s+//g;
        #print "Current line is $line\n";
        if ($state == 2){ #save the inputline
                push (@outputlines, $inputline);
                $state = 0;
                $inputline = "";
        }
```

Program 2 Source Code of Attack Strategy Composer (part-1)

```
        if ($line =~ /\w+ [<,=,>]+ [\w+,\d+]/){#if statement
                if ($line =~ / = \w+/){
                        $line =~ s/ = /=="/;
                        $line .= "\"";
                }
                else{
                        $line =~ s/ = /==/;
                }

                if ($state == 0){ #first if statement
                        $state = 1;
                        $inputline .= "$IFHEADER($line)";
                }
                elsif ($state == 1){
                        $inputline .= "||($line)";
                }

        }
        elsif ($line =~ /->/) {#decision
                $line =~ s/->\s+class\s+/\t/g;
                $line =~ s/\[\d+\.\d+%\]$//g;
                $line =~ s/\s+;$/;/g;
                $line = lc($line) . "()";
                if ($state == 0){ #first if statement
                        print "data error.\n";
                }
                elsif ($state == 1){
                        $inputline .= "$TAIL \n\t$line;\n$END";
                        $state = 2;
                }

                #print $inputline;

        }
}

print "The output of the Attack Strategy Composer is ..... \n";
foreach my $line (@outputlines){
        print $line;
}
```

BIBLIOGRAPHY

Bridis, T. (2003). Government simulates national cyber attack. *CRN*, www.crn.com/sections/BreakingNews/dailyarchives.asp?ArticleID=463.

Chi, S.D., P.J. (2001). Network security modeling and cyber attack simulation methodology. *Lecture Notes in Computer Science*, *2119*-2001.

Debar H., B.M., S.D. (1992). A neural network component for an intrusion detection system. *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy*, *1*, 240–250.

Eskin, E., A.P., P.L., S.S. (2002). A geometric framework for unsupervised anomaly detection: Detecting intrusions in unlabeled data. *Applications of Data Mining in Computer Security.*

Howard, J.D., T.A. (1998). A common language for computer security incidents. *SANDIA REPORT*, *SAND98*, 8667.

J.R. Quinlan (1992). C4.5 decision learning tree, http://www.rulequest.com/Personal/c4.5r8.tar.gz.

Konrad, R., P.L. (2008). Linear-time computation of similarity measures for sequential data. *Journal of Machine Learning Research*, *9*, 23–48.

Kotenko, I., E.M. (2003). Experiments with simulation of attacks against computer networks. *Lecture Notes in Computer Science*, *2776*, 183–194.

Kuhl, M.E., J.K., K.C. (2007). Cyber attack modeling and simulation for network security analysis. *Proceedings of the 2007 Winter Simulation Conference*, *1*, 1180–1188.

Labib, K., R.V. (2002). NSOM: A real-time network-based intrusion detection system using self-organizing maps. *Technical Report*.

Lane, T., B.C. (1997). An application of machine learning to anomaly detection. *Proceedings of the 20th National Conference on National Information Systems Security*, *1*, 366–380.

Lau, F., R.H. (2000). Distributed denial of service attacks. *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, *3*, 2275–2280.

Leung, K., L.C. (2005). Unsupervised anomaly detection in network intrusion detection using clusters. *Proceedings of the Twenty-Eighth Australasian Conference on Computer Science*, *102*, 333–342.

Li, X., Y.N. (2003). Decision tree classifiers for computer intrusion detection. *Real-Time System Security*, 77–93.

Metasploit (2008) Metasploit project. http://www.metasploit.com.

PCAP Library (2008). http://www.tcpdump.org.

R Project (2008). http://www.r-project.org.

Richard A. Johnson, D.W. (2002). Applied multivariate statistical analysis. *Pearson Education International*.

Shyu, M.L., C.S., S.K. C.L. (2003). A novel anomaly detection scheme based on principal component classier. *Proceedings of the IEEE Foundations and New Directions of Data Mining Workshop*.

Stefano, Z., S.M. (2004). Unsupervised learning techniques for an intrusion detection system. *Proceedings of the 2004 ACM symposium on Applied computing*, *1*, 412–419.

Stein, G., C.B., W.A., H.K. (2005). Decision tree classifier for network intrusion detection with GA-based feature selection. *Proceedings of the 43rd Annual Southeast Regional Conference*, *2*, 135–141.

ACKNOWLEDGEMENTS

I would like to take this opportunity acknowledge and extend my heartfelt gratitude to those who helped my research and made this thesis possible.

First and foremost, Dr. Doug Jacobson for his insightful guidance and vital support throughout this research and the writing of this thesis. His encouragement and advice inspired me and renewed my hopes for completing my graduate education.

I would also like to thank my committee members for their efforts and contributions to this work: Dr. Tom Daniels and Dr. Ying Cai.

I would additionally like to thank Dr. Morris Chang for his guidance and financial support throughout the initial stages of my graduate study.