

**SOPHISTICATED DENIAL-OF-SERVICE ATTACK DETECTIONS
THROUGH INTEGRATED ARCHITECTURAL, OS, AND
APPLICATION LEVEL EVENTS MONITORING**

A Thesis
Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering

In
The Department of Electrical and Computer Engineering

By
Ran Tao
Bachelor of Engineering in Electrical Engineering,
University of Electronic Science and Technology of China, 2005
August 2009

Acknowledgements

This work could not have been completed without the help and support from a lot of people I am grateful to.

I would like to thank Dr. Lu Peng, my advisor, for the guidance and suggestions throughout the whole research study to help me work towards the completion of my Master program. I would like to thank Dr. Li Yang from Department of Computer Science and Engineering, University of Tennessee at Chattanooga, for her explanations of network security background knowledge and assistance with the experimental procedure. I also wish to thank Dr. Bin Li from Department of Experimental Statistics, Louisiana State University, for giving the instructions on the R software statistical tool, and the help with statistical analysis.

Additionally, I really appreciate Dr. R. Vaidy and Dr. A. Skavantzios for agreeing to be my committee member and taking the time to attend my defense.

I am extremely grateful to my parents for always being there for me, and continued encouragement and support for me to finish my study in LSU. I also want to express my sincere gratitude to all my friends for making my life delightful.

Table of Contents

Acknowledgements.....	ii
List of Tables	v
List of Figures	vi
Abstract.....	vii
Chapter	
1. Introduction.....	1
1.1 Overview.....	1
1.2 Major Contributions.....	3
1.3 Thesis Outline	4
2. Background and Basic Concepts	5
2.1 Types of Intrusion Detection Systems	5
2.1.1 Host-based Intrusion Detection System (HIDS).....	5
2.1.2 Network Intrusion Detection System (NIDS).....	6
2.2 IDS Methodologies	7
2.2.1 Signature-based IDS	7
2.2.2 Anomaly-based IDS.....	7
3. Multi-Level Intrusion Detection System	9
3.1 Methodology	9
3.2 IDS Framework.....	10
3.3 Statistical Model	12
4. Implementation of Multi-Level IDS	15
4.1 Overview.....	15
4.2 Application Level Parser.....	15
4.3 OS Level System Call Tracking.....	17
4.4 Architectural Level Kernel Module	17
4.4.1 Performance Monitoring Counters	18
4.4.2 Implementation of Kernel Module.....	20
5. Datasets Generation	22
5.1 System Environment.....	22
5.2 Developed Denial-of-Service Exploits	22
5.3 Real-world Exploits	24
6. Results and Analysis	26
6.1 Performance Metrics.....	26
6.2 IDS Performance Comparison	26
6.2.1 Evaluation with Developed Exploits	27

6.2.1.1	Two Level Feature Sets.....	27
6.2.1.2	Three Level Feature Set.....	29
6.2.2	Evaluation with Mixed Dataset.....	31
6.2.3	Evaluation with Real-World Local Exploits	32
7.	Related Work	38
8.	Conclusions.....	41
	References.....	42
	Appendix A. Buffer Overflow IDS.....	44
1.	Background.....	44
2.	Behavior of BoF Attacks.....	45
3.	System Call Tracking.....	47
4.	IDS Implementation.....	47
5.	IDS Performance.....	48
	Appendix B. Control Register Layout for Pentium D	50
	Vita.....	52

List of Tables

Table 3-1.	Application Level Features.....	11
Table 3-2.	Architectural Level Features	11
Table 3-3.	OS Level Features	12
Table 4-1.	List of registers used to monitor CPU performance.....	18
Table 4-2.	PC MSR and control registers used.....	19
Table 5-1.	The self-developed DoS exploits.....	23
Table 5-2.	Dataset construction	23
Table 5-3.	Real-world remote DoS exploits	25
Table 5-4.	Real-world local DoS exploits.....	25
Table 6-1.	Measurements of IDS System	26
Table 6-2.	False alarm rate of IDS of one level and two level feature sets	27
Table 6-3.	Sample records	28
Table 6-4.	False alarm rate of IDS for different feature sets	31
Table 6-5.	IDS performance with mixed dataset	32
Table 6-6.	Dataset constructed using developed exploits.....	33
Table 6-7.	Dataset constructed using mixed exploits	33
Table 6-8.	IDS performance comparison.....	35

List of Figures

Figure 1-1.	Framework of multi-level IDS	3
Figure 2-1.	HIDS Topology	5
Figure 2-2.	NIDS Topology	6
Figure 3-1.	The framework of our IDS	10
Figure 4-1.	Network packet header.....	16
Figure 4-2.	Workflow of application level parser.....	16
Figure 5-1.	Pseudo code snippet for BSB DoS exploits	23
Figure 6-1.	Detection rate of IDS with one level and two level feature sets	27
Figure 6-2.	Detection rate of IDS with different feature sets	31
Figure 6-3.	IDS performance comparison	34

Abstract

As the first step to defend against DoS attacks, Network-based Intrusion Detection System is well explored and widely used in both commercial tools and research works. Such IDS framework is built upon features extracted from the network traffic, which are application-level features, and is effective in detecting flooding-based DoS attacks. However, in a sophisticated DoS attack, where an attacker manages to bypass the network-based monitors and launch a DoS attack locally, sniffer-based methods have difficulty in differentiating attacks with normal behaviors, since the malicious connection itself behaves in the same manner of normal connections. In this work, we study a Host-based IDS framework which integrates features from architectural and operating system (OS) levels to improve performance of sophisticated DoS intrusion detection. Network traffic collected from a campus network, and real-world exploits are used to provide a realistic evaluation.

Chapter 1

Introduction

1.1 Overview

Denials of Service (DoS) attacks impose serious threat on the availability and quality of Internet services [15]. They exhaust limited resources such as network bandwidth, DRAM space, CPU cycles, or specific protocol data structures, inducing service degradation or outage in computing infrastructures for the clients. System downtime resulting from DoS attacks could lead to million dollars' loss.

Generally, DoS attacks can be either flooding-based or software exploit-based. In a flooding-based DoS attack, a malicious user sends out a tremendously large number of packets aiming at overwhelming a victim host. For example, in a SYN-flooding attack, a significant number of TCP SYN packets are sent towards a victim machine, saturating resources in the victim machine. We can observe a surge of TCP connections in a short time, which are modeled by a tuple of application features $\langle \text{source IP, destination IP, source port, destination port} \rangle$. In exploit-based DoS attacks, specially crafted packets are sent to the victim system targeting at specific software vulnerabilities in the operating system, service or application. The success of exploitation will either overwhelm or crash the target system. An existing solution to the exploit-based attacks is to patch and update software frequently.

Currently, research work on DoS intrusion detections mainly rely on Network-based Intrusion Detection Systems (NIDSs) [3][5][6][7][8][10][21]. The NIDSs monitor features extracted from network packet headers at the application layer such as packet rate and traffic volume. Ramp-up behaviors and frequency domain characteristics are also studied to aid in

improving the accuracy and performance of IDS [3][6]. On the other hand, Host-based Intrusion Detection Systems (HIDSs) which widely employ audit trails and system call tracking can effectively identify buffer overflow (BoF) attacks [1][2][19]. However, the DoS attacks are not easily observed by such an HIDS and not widely researched in the HIDS literature. Some researchers have proposed to limit the bound of certain system calls [1] such as *fork()*. However, with the advent of large-scale application software, such bounds may seriously impair the performance of normal applications. Moreover, DoS attacks may not involve huge number of system calls at all. Therefore, a more generic solution is needed to detect DoS attacks.

When increasingly sophisticated techniques are adopted by attackers, multi-tier attacks and IP spoofing are emerging to amplify destructive effects and evade detections. The attack patterns or behaviors will be difficult to identify by using only header-based network traffic analysis. For example, in a complicated scenario that an attacker gets around the network monitoring sensors and launches DoS attacks locally, a NIDS may not be able to detect this intrusion. In such a scenario, non-privileged access is good enough to successfully initiate a DoS attack against the host machine: once the attacker obtains the access to the victim machine, even if it is not root-privileged and difficult to further elevate to carry out other destructive or stealthy behaviors, he/she can still easily upload a DoS daemon to massively consume the machine's limited resources. Instead of network information only, information originated and resided on the victim machine should be used to track and monitor such undergoing attacks in this case.

1.2 Major Contributions

In this work, we propose an HIDS with multi-level integrated information from application, operating system (OS), and architecture levels to improve the detection rate of sophisticated DoS attacks. Figure 1-1 illustrates the framework of our proposed IDS framework. At different levels, we use different tools or schemes to collect and extract typical features of possible intrusions. According to our experiments, even if DoS attacks could successfully evade captures of NIDS monitors, architectural behaviors will still be triggered: tremendous jumps of *Instruction Count*, *Cache Miss*, *Bus Traffic* can be found. Based on this observation, a novel HIDS employing a modern statistical Gradient Boosting Trees (GBT) model is proposed to detect sophisticated DoS intrusions through the integration of application, OS, and architectural features. Our experiments show that the inclusion of architectural features can significantly improve the detection rate of such evasive DoS intrusions

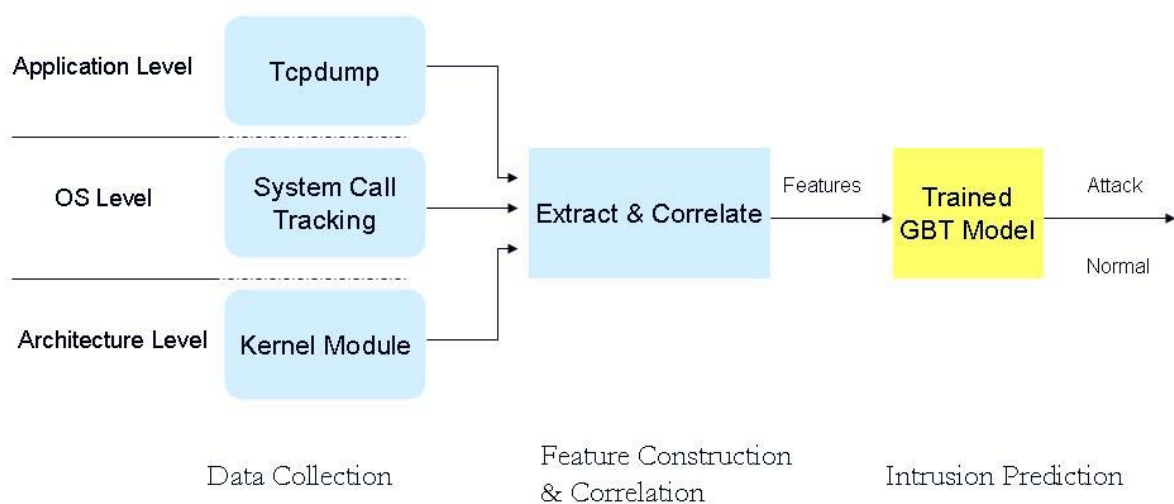


Figure 1-1. Framework of multi-level IDS

1.3 Thesis Outline

The rest of this thesis is organized as follows: background knowledge is introduced in chapter 2. Our proposed IDS methodology and framework is elaborated in chapter 3. Chapter 4 provides the technical details of the multi-level IDS implementation. Dataset generation is described in chapter 5. The experiment results are shown and discussed in chapter 6. Related work is discussed in chapter 7. We conclude the work in chapter 8.

Chapter 2

Background and Basic Concepts

2.1 Types of Intrusion Detection Systems

Intrusion Detection Systems can be broadly divided into two categories: Host-based and Network-based Intrusion Detection System. Other types of IDSs could be considered variants or hybrid of these two basic types.

2.1.1 Host-based Intrusion Detection System (HIDS)

HIDS monitors and analyses the internal behavior of a computing system, including all or part of the dynamic behavior and the state of a computer system. Event logs, audit trails, system call tracking are widely utilized to identify and defend against attacks.

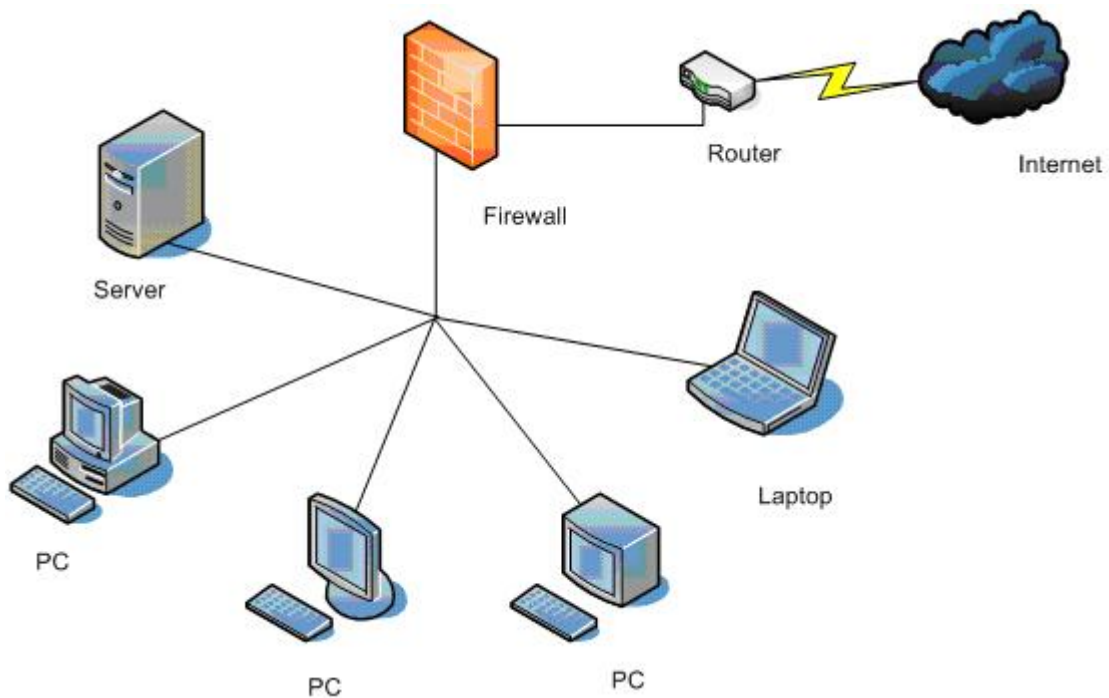


Figure 2-1. HIDS Topology

The topology of HIDS is shown in Figure 2-1. Machines labeled in blue are installed with the HIDS. Since the IDS is host-based, actual installation on the system under monitor is required. Otherwise, the IDS can not gain full access to the internal system information.

2.1.2 Network Intrusion Detection System (NIDS)

As opposed to monitoring the activities that originates on a particular system, NIDS focus on external information outside monitored target. It sniffs network traffic and analyze all in-coming packets, looking for suspicious patterns of malicious connections.

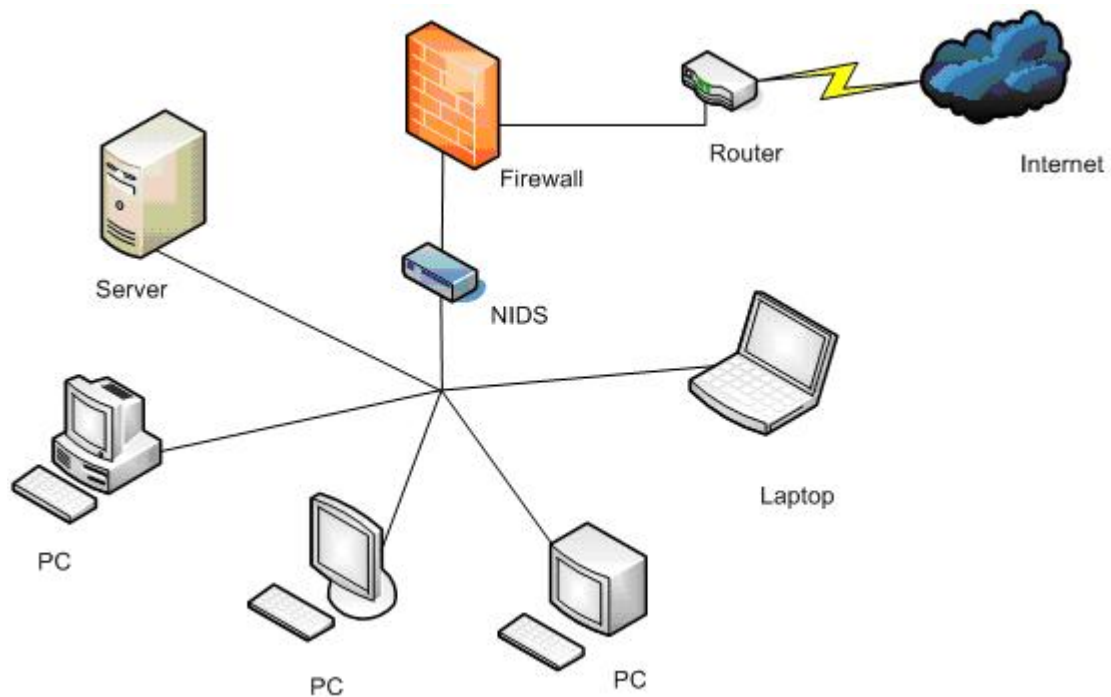


Figure 2-2. NIDS Topology

As shown in Figure 2-2, the NIDS highlighted in blue is usually placed behind the LAN firewall. It can be implemented as software, or appliance hardware. It keeps monitoring the traffic coming from outside network and within the LAN, and it also analyzes the content of

individual packets in search for malicious traffic. To monitor a LAN, only one such device is required to install. Comparing to HIDS, NIDS adopts a centralized infrastructure in the LAN.

2.2 IDS Methodologies

Signature and anomaly detection are two primary IDS approaches. They have their own advantages and disadvantages, and actually, they complement each other in the intrusion detection field.

2.2.1 Signature-based IDS

Signature-based IDS employs specifically known patterns of misuse behavior to predict potential malicious activities. These patterns, i.e. signatures, could be the number of failed log-ins during a certain time frame, or specific patterns matching a portion of network packets.

PROS: Signatures are easy to develop and understand if you know what malicious behavior you're trying to identify.

CONS: Virus database must be constantly updated. A signature must be created for every attack, and novel attacks cannot be detected.

2.2.2 Anomaly-based IDS

Anomaly-based IDS is designed to uncover misuse behavioral patterns by examining network traffic and system activities. They establish a baseline of normal behavior, observe when current behavior deviates statistically from the norm, and flag those activities as possible intrusions.

PROS: Anomaly-based IDS has the ability to promptly detect novel attacks that are

unknown or for which signatures are not developed yet.

CONS: As normal behavior can change easily and readily, there is no standard normal behavior profile. Anomaly-based IDS systems are prone to substantial false positives where attacks may be reported based on deviations from the norm patterns.

Chapter 3

Multi-Level Intrusion Detection System

3.1 Methodology

In our design, we integrate the information which only resides on the host machine under attacks, and then construct a multi-layer IDS to detect sophisticated DoS attacks. The correlation of system architectural behaviors and DoS attacks is analyzed by a modern statistical model employing Gradient Boosting Trees techniques. Architectural features are explored to improve the IDS performance. Our proposed scheme involves multiple steps listed as follows.

- **Step 1: Data Collection**

We use the *tcpdump* utility to record header information of network packets transmitting towards/from the host computer. Architectural behaviors are recorded using a device driver which periodically samples the CPU performance counters and dumps out the performance variation trace. System call tracking function embedded in the Linux kernel is utilized to record OS level events.

- **Step 2: Feature Extraction and Correlation**

Our desired application level features are extracted using a custom network traffic parser which models records by network sessions identified by `src_ip:src_port <-> dst_ip:dst_port`. OS level features are extracted from a system call tracking function embedded in the kernel. Architectural records are processed as a ratio of event numbers during the current session to a pre-measured normal session without attacks.

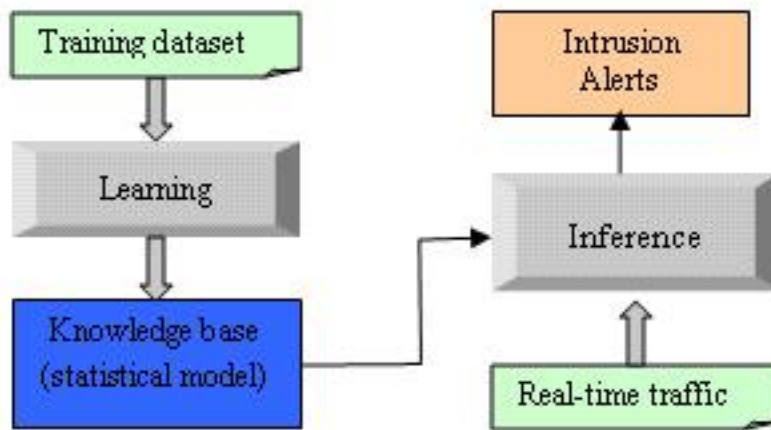


Figure 3-1. The framework of our IDS

Since features of different levels are obtained by different collecting processes, we append a timestamp to each record for the correlation between architectural events and application events during the same session.

- **Step 3: Intrusion Prediction**

As a standard workflow, in this step, each correlated record is fed to the statistical model which has learned the patterns of normal and attack behaviors from the training dataset. It will raise an alert if the given record deviates from normal behaviors.

3.2 IDS Framework

The framework of our proposed intrusion detection system consists of a learning module and an inference module as shown in Figure 3-1. The learning module is used to build up the knowledge from an offline training dataset. The knowledge base contains a statistical model which is learned from observed traffic, and has the ability to predict whether a network connection is malicious or benign. The inference module is the analysis engine of our IDS. Its task is to process the data collected from the sensors in order to identify intrusive activities.

The training set and the real-time traffic include features from different levels as shown in Table 3-1, Table 3-2 and Table 3-3.

1. Application (APP) Feature Set

Table 3-1. Application Level Features

Feature Name	Description	Type
protocol_type	Type of the protocol, e.g., tcp, udp, etc.	discrete
service	Network service on the destination system, e.g., ssh, http, telnet, etc.	discrete
duration	Length of the connection	continuous
size_from_client	Number of data bytes from client to server	continuous
size_from_server	Number of data bytes from server to client	continuous
packet_rate	Number of two-way packets per second	continuous
wrong_checksum_rate	Percent of packets have wrong checksum	continuous

2. Architectural (ARCH) Feature Set

Table 3-2. Architectural Level Features

Feature Name	Description	Type
instruction_retired	Ratio of average instructions committed during a session to a pre-measured normal session	continuous
L1_cache_miss	Ratio of average L1 cache miss during a session to a pre-measured normal session	continuous
L2_cache_miss	Ratio of average L2 cache miss during a session to a pre-measured normal session	continuous
bus_access	Ratio of average bus transactions during a session to a pre-measured normal session	continuous

We select these features because a typical network DoS attack can be monitored by observing these events [22]. These events exhibit very obvious variations during DoS attacks.

3. Operating System (OS) Feature Set

Table 3-3. OS Level Features

Feature Name	Description	Type
forked_socket_session	Forked another network connection	discrete
forked_shell	Forked shell sessions	discrete
forked_from_shell	Forked from another network connection	discrete
coincided_pid	Shares a same pid as another different network connection	discrete

3.3 Statistical Model

The statistical model that we employed for intrusion detection is based on Gradient Boosting Trees (GBT), originally proposed in [4]. GBT is one of several techniques that aim to improve the performance of a single model by fitting many models and combining them for prediction. GBT uses two algorithms: “trees” from the Classification and Regression Tree and “boosting” which builds and combines a collection of models, i.e. trees.

From a user’s point of view, GBT has three major advantages. First, GBT is inherently non-parametric and able to handle mixed-type of input variables. Both discrete and continuous data are supported. There is no need of data discretization. GBT does not need to make any assumptions regarding the underlying distribution of the values for the input variables. Thus, it relieves researchers from determining whether variables are normally distributed, and making transformations if they are not. Second, the tree is adept at capturing complex-structured behavior, i.e. complex interactions among predictors are routinely and automatically handled with relatively few inputs required from the analyst. This is in marked contrast to some other multivariate nonlinear modeling methods, in which extensive input from the analyst, analysis of interim results, and subsequent modification of the method are

required. Third, the tree is insensitive to outliers, and unaffected by monotone transformations and differing scales of measurement among inputs. Despite clear evidence of strong predictive performance, boosting-based learning methods have been rarely used in computer intrusion detection [24].

Consider the binary classification problem with n observations of the form $\{y_i, \mathbf{x}_i\}$, $i=1, \dots, n$, where \mathbf{x}_i is a multi-dimensional input vector and y_i is the binary response $y_i \in \{-1, +1\}$. In this paper, \mathbf{x}_i is the feature in multiple levels and y_i is the prediction result, i.e., attack or benign connection. The negative log-likelihood for the binomial model or deviance (also known as cross-entropy) is used as the loss function:

$$L(y, \hat{f}) = \log(1 + \exp(-2y\hat{f}))$$

The population minimizer of the loss function is at the true probabilities:

$$\operatorname{argmin}_{f(\mathbf{x})} E_{Y|\mathbf{X}}[L(y, f(\mathbf{x}))] = f^*(\mathbf{x}) = \frac{1}{2} \log \left[\frac{\Pr(y=1|\mathbf{x})}{\Pr(y=-1|\mathbf{x})} \right]$$

Or equivalently:

$$\Pr(y=1|\mathbf{x}) = \frac{1}{1 + e^{-2f^*(\mathbf{x})}}$$

where $E_{Y|\mathbf{X}}[L(y, f(\mathbf{x}))]$ is the expectation value of the loss function over Y given the input \mathbf{X} .

The detailed algorithm for GBT in binary classification is the following.

- 1) Initialize $\hat{f}_0(\mathbf{x}_i) = \frac{1}{2} \log \left(\frac{1 + \bar{y}}{1 - \bar{y}} \right)$, where \bar{y} is the average for $\{y_i\}$.
- 2) Repeat for $m = 1, 2, \dots, M$:
 - a) Set the negative gradient:

$$\tilde{y}_{im} = - \left[\frac{\partial L(y_i, \hat{f}_{m-1}(\mathbf{x}_i))}{\partial \hat{f}_{m-1}(\mathbf{x}_i)} \right], \quad i = 1, \dots, n.$$

b) $\{R_{hm}\}_{h=1}^H = H\text{-terminal node tree based on } \{\tilde{y}_{im}, \mathbf{x}_i\}_{i=1}^n$

c) $\gamma_{hm} = \underset{\gamma}{\operatorname{argmin}} \sum_{\mathbf{x}_i \in R_{hm}} L(y_i, \hat{f}(\mathbf{x}_i) + \gamma)$

d) $\hat{f}_m(\mathbf{x}) = \hat{f}_{m-1}(\mathbf{x}) + \nu \times \gamma_{hm} I(\mathbf{x} \in R_{hm})$

3) End algorithm.

Note that ν is the “shrinkage” parameter between 0 and 1 and controls the learning rate of the procedure. Empirical results have shown that small values of ν always lead to better generalization error rates [4]. In this study, we fix ν at 0.01. During each iteration, an H-terminal node tree, which partitions the \mathbf{x} space into H-disjoint regions, $\{R_{hm}\}_{h=1}^H$, is fitted based on the current negative gradient for the loss function.

Chapter 4

Implementation of Multi-Level IDS

4.1 Overview

We summarize network traffic captured in a tcpdump file into high-level connections. Specifically, a connection is a sequence of network packets starting and ending at some well defined points in time, between which data flows under a well defined protocol from a source IP address to a target IP address. Before processing the data with the IDS training algorithm, raw network traffic has to be pre-processed and summarized into connections or high-level events. Each connection is described with a set of features which IDS models can utilize to detect possible intrusions.

4.2 Application Level Parser

Our modified parser based on an open source utility Chaosreader extracts desired information in the application level out from the recorded tcpdump files, and groups packets into sessions by `src_ip:src_port <-> dst_ip:dst_port`, thus we will obtain a set of preprocessed data in the format that each entry represents a network connection, together with application features flagged accordingly.

Our desired information is stored in different header levels, thus the parser strips the headers level by level, and construct the features we need.

The workflow of the parser is shown in Figure 4-2.

magic number		
major version	minor version	
time zone offset		
timestamp accuracy		
snapshot length		
link layer type		### Global Header
ts_sec		
ts_usec		
length of packet captured		
orig_len		### Packet Header
vers hdr_len serv	packet legnth	
identification	flags fragment offset	
ttl	protocol ip hdr checksum	
ip source addr		
ip dest addr		### IP Header
tcp src port	tcp dest port	
sequence number		
acknowledgement number		
hdr_len na tcp flags	window size	
tcp checksum	urgent pointer	### TCP Header
data		
udp src port	udp dest port	
length	checksum	### UDP Header

Figure 4-1. Network packet header

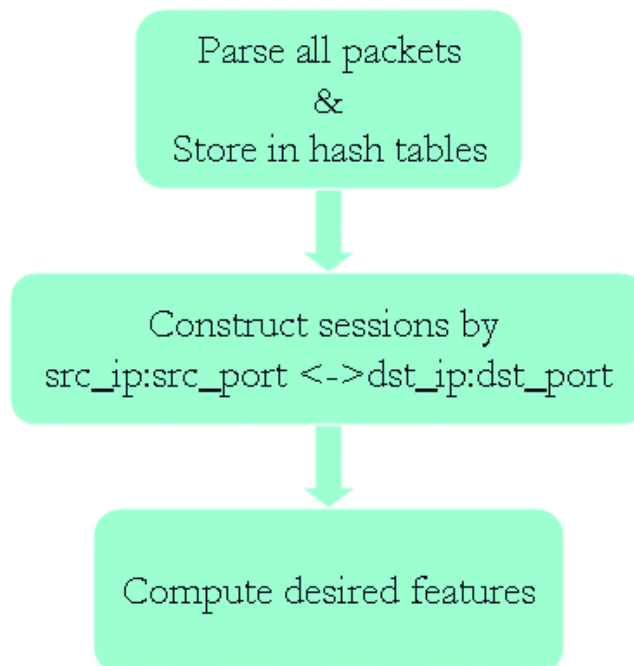


Figure 4-2. Workflow of application level parser

4.3 OS Level System Call Tracking

At operation system level, we employ a system call tracking function based on BackTracker [9] to record all system call driven events, and related network socket sessions. We append a timestamp to every record; add full IP address and port number information to all socket sessions, in order to correlate OS events with other level events. BackTracker discovers sequences of steps that occurred during an intrusion. Starting at a single detection point (e.g., a suspicious file), it identifies files and processes that could have affected that detection point and displays chains of events in a dependency graph. BackTracker is able to record every dependency-causing event among OS objects, and span up a dependency chain starting from one object. Complete information such as process forking, file operations, and program execution (which is important for security analysis), is recorded in a system-call oriented manner.

4.4 Architectural Level Kernel Module

The Intel Pentium-D processor provides us with adequate performance counters to illustrate the CPU's dynamic performance profile. A kernel module is implemented to sample the performance counters in regular intervals. We set the sampling interval to 0.5s, balancing the tradeoff between system performance overhead and accuracy of monitored performance variation. Thus, at regular interval, the values of these four architectural counters which have most representative architectural variation under a DoS attack are recorded and dumped to a trace file. The timestamp recorded together with other performance counters is used to correlate architectural events with network connections parsed from tcpdump files.

4.4.1 Performance Monitoring Counters

Performance monitoring is introduced in Pentium processor. A set of performance monitoring registers are provided to monitor or measure processor performance profile. The performance monitoring mechanism varies with processor families. The target system used in the experiment is installed with Pentium-D processor, which features Intel NetBurst Microarchitecture. 18 Performance Counter MSR (PC MSR) are provided to count events. Each PC MSR is associated with one Counter Configuration Control Register (CCCR) to set up a specific counting method. Every PC MSR and CCCR pair is controlled by a subset of 45 Event Selection Control Registers (ESCR). Table 4-1 lists the registers we need to use to sample CPU performance events. All registers are read/written using RDMSR, WRMSR, or RDPMSR instruction.

Table 4-1. List of registers used to monitor CPU performance

Register	Name	Function
MSR	Model Specific Register	Monitor performance, debug system, enable/disable model specific functions, etc.
PC MSR	Performance Counter MSR	Store actual event number
ESCR	Event Selection Control Register	Select events to be monitored by specific PC MSR
CCCR	Counter Configuration Control Register	Set up the associated performance counter to function in a specific style

Table 4-2. PC MSR and control registers used to sample desired architectural events

Event to Monitor	Register Address	Register Name
Inst_Retired	0x3b8	MSR_CRU_ESCR0
	0x36c	MSR_IQ_CCCR0
	0x30c	MSR_IQ_COUNTER0
L1_Cache_Miss	0x3cc	MSR_CRU_ESCR2
	0x370	MSR_IQ_CCCR4
	0x310	MSR_IQ_COUNTER4
L2_Cache_Miss	0x3a1	MSR_BSU_ESCR1
	0x362	MSR_BPU_CCCR2
	0x302	MSR_BPU_COUNTER2
Bus_Access	0x3a2	MSR_FSB_ESCR0
	0x360	MSR_BPU_CCCR0
	0x300	MSR_BPU_COUNTER0

A specific procedure needs to be followed to correctly configure the control registers and enable performance counters to count on the correct events:

Step 1: Identify events to monitor.

Step 2: For each selected event, choose an ESCR which support the event.

Step 3: Identify the CCCRs and PC MSRs associated with the ESCRs.

Step 4: Set up ESCR with correct event mask and privilege level.

Step 5: Set up CCCR with correct ESCR mask. (Cascading and event filtering are optional)

Step 6: Set CCCR enable flag to start event counting.

Step 7: Sample PC MSRs to poll out CPU performance profile.

For those four architectural events we need to monitor, PC MSR and control registers listed in Table 4-2 need to be configured or sampled.

4.4.2 Implementation of Kernel Module

Although RDPMC instruction is allowed to execute in all privilege levels, control registers have to be configured with WRMSR instruction, which can only be executed at privilege level 0. Thus, a kernel module, i.e. device driver, is implemented to manipulate on hardware registers, and obtain the information of system runtime performance variations. Linux kernel allows kernel modules to be loaded and unloaded at run time, without rebooting the system. It runs at privilege level 0 and has full access to all system devices.

A virtual file under /proc directory is also created to dump out information from kernel space to user space. /proc is a virtual file system that contains a hierarchy of dynamic virtual files. Those files represent current status of the kernel. They allow users and applications to gain insight of system's kernel status, and communicate configuration changes to the kernel.

The implemented module creates two directories under /proc; they serve as an interface between kernel space and user space:

/proc/perf_mon/setting: Used to configure counter sampling

/proc/perf_mon/event: Used for user-mode application to poll out counter values.

The /proc/perf_mon/setting entry supports four types of command:

Start: Initiate all ESCRs and CCCRs, place sampling routine on wait queue to periodically sample PC MSRs at predefined intervals.

Clear: Set all control registers to zero, clear PC MSRs.

Stop: Set all ESCRs and CCCRs to zero, stop event counting, but values in PC MSRs are still preserved.

Read: Increment internal read number counter.

Once sampling routine is initiated, it will keep writing event numbers to the `/proc/perf_mon/event` entry buffer. Since memory space for a `/proc` entry is limited, we implemented a user mode application to flush all event numbers from `/proc/perf_mon/event` entry at pre-calculated intervals. And the event entry will be refreshed after all contents are dumped out.

Chapter 5

Datasets Generation

5.1 System Environment

We use an Intel Pentium-D PC installed with Redhat Linux 9.0 as target system, and another machine installed with Suse 10 as attacker/client system. Both machines are connected to a department LAN. Exploits are launched from attacker to target system. Network traffic information is captured with the *tcpdump* tool.

5.2 Developed Denial-of-Service Exploits

Nowadays more sophisticated techniques are emerging to escape IDS detections; in this work, we assume crackers have gained unauthorized access to the victim machine (they may only have non-privileged access), and then intend to launch local DoS daemons. To emulate this scenario, we design five local DoS exploits which are used to model local DoS exploits exhausting different system resources. Each exploit target a specific type of system resources, intentionally exhausting a particular resource, and rendering the system unavailable to legitimate users. Table 5-1 outlines the detailed descriptions of these exploits. First three attacks are traversing a certain memory space with the stride of the cache line size (64 bytes in our system). In this case, regardless of set associativity of data cache, constant cache miss will be generated upon each memory access.

```

allocate an array of 32KB;

__asm__ __volatile__(
    "movl (pointer to array), %edx\n\t"
);
while ( 1 ){
    __asm__ __volatile__(
        " movl 0(%edx), %ebx\n\t "
        " movl 64(%edx), %ebx\n\t "
        " movl 128(%edx), %ebx\n\t "
        " movl 192(%edx), %ebx\n\t "
        " movl 256(%edx), %ebx\n\t "
        .
        .
        .
        " movl 32640(%edx), %ebx\n\t "
        " movl 32704(%edx), %ebx "
    )
}

```

Figure 5-1. Pseudo code snippet for BSB DoS exploits

Table 5-1. The self-developed DoS exploits

Attack Type	Description
L2 Cache DoS	Target L2 cache, sweep through L2 cache space
BSB DoS	Target backside bus bandwidth, sweep through twice the L1 D\$ size, saturate backside bus
FSB DoS	Target front-side bus bandwidth, sweep through twice L2 cache size, saturate front-side bus.
Memory DoS	Target memory space; keep allocating memory space, max out memory usage.
Loop DoS	Target CPU usage, infinite dummy instruction.

Table 5-2. Dataset construction

Dataset	Combination
Training 1	l2 + bsb + fsb + mem
Training 2	l2 + bsb + fsb + loop
Training 3	l2 + bsb + mem + loop
Training 4	l2 + fsb + mem + loop
Training 5	bsb + fsb + mem + loop
Testing	l2 + bsb + fsb + mem + loop + noise

We launch these exploits multiple times over a LAN to obtain five different training datasets, each containing only four exploit types. Details of the training and testing sets construction are listed in Table 5-2.

The testing dataset includes a full set of the above five exploit types, 25 attack instances in total, and is injected with noise traffic data of CPU or memory intensive operations such as tar, compile, scp etc. Those noises are included in order to evaluate the ability of the IDS to differentiate normal operation and attack traffic. In addition, we also include 3630 normal connections.

5.3 Real-world Exploits

Apart from our crafted exploits, we also evaluate our proposed scheme using mixed data with real-world remote DoS exploits, and dataset of real-world local DoS exploits. The descriptions of real-world remote DoS exploits in the experiment are outlined in Table 5-3, and description of those local DoS exploits are listed in Table 5-4.

We divide the mixed data of hand-crafted exploits and real-world remote DoS exploits into five training datasets and one testing dataset. Training data contains partial set of all exploits, while the testing data contains a full set of all exploit types. Testing data also include significant amount of noise traffic, which is injected intending to evaluate the ability of the IDS to detect exploits never seen before and avoid the false alarms.

The real-world local DoS exploits will be used separately to evaluate the ability of our proposed IDS to identify novel local DoS attacks it has not seen before.

Table 5-3. Real-world remote DoS exploits

Attack	Description
CVE-2003-0132	Apache memory leak, drains memory via large chunks of linefeed characters.
CVE-2003-0543	OpenSSL integer overflow, causes Apache server to enter CPU intensive loop.
CVE-2004-0493	Apache memory exhaustion.
CVE-2004-0942	Apache multiple space header DoS, drains CPU resource.

Table 5-4. Real-world local DoS exploits

Attack	Description
Memory leak local DoS - 1	Kernel vulnerability causing exhaustion of system memory resource, inducing system crash.
Memory leak local DoS - 2	Kernel vulnerability allows non-privileged users to read kernel memory and system performance degradation.

Chapter 6

Results and Analysis

6.1 Performance Metrics

We use the typical four measurements to evaluate performance of our IDS. True positive (TP) rate is the ratio of the number of correctly detected attacks and the total number of attacks. The true negative (TN) rate is the ratio of the number of normal connections and the total number of normal connections. The false positive (FP) rate is the ratio of the number of normal connections that are incorrectly classified as attacks and the total number of normal connections. False negative (FN) is the rate of missed attacks.

Table 6-1. Measurements of IDS System

	Predicted Normal	Predicted Intrusions
Actual Normal Connection	True Negative (TN)	False Positive (FP)
Actual Intrusions (Attacks)	False Negative (FN)	True Positive (TP) (correctly detected intrusions)

6.2 IDS Performance Comparison

We conduct three sets of experiments, to evaluate the ability of different IDS schemes to identify simulated and real-world DoS exploits.

6.2.1 Evaluation with Developed Exploits

Two sets of experimental results are shown to demonstrate how architectural events help to increase the true positive rate, and how OS events assist in decreasing false positives.

6.2.1.1 Two Level Feature Sets

Firstly, we train an IDS using the GBT model with only the application level features listed in Chapter 3. Results shown in Figure. 6-1 reveal that IDS built on the application features alone can only recognized around 30% of such DoS attacks (refer to the light bar group in Figure 6-1). These testing results demonstrate that the APP features are not

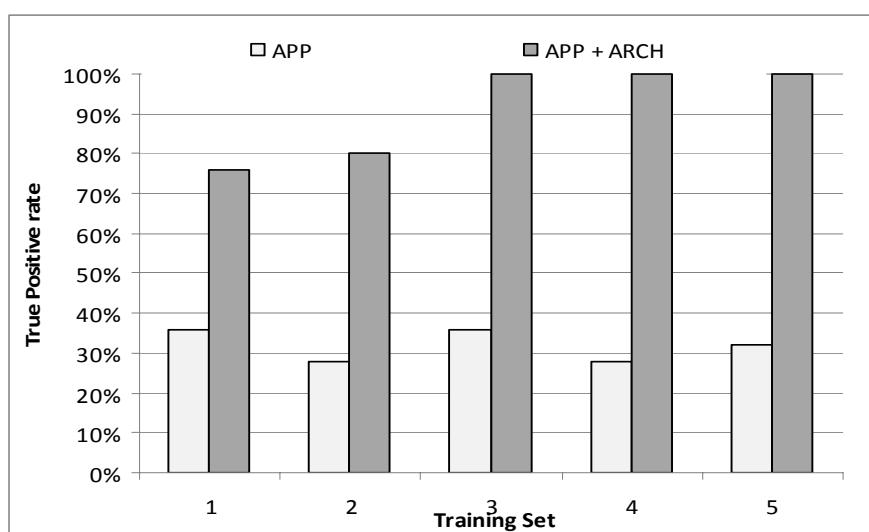


Figure 6-1. Detection rate of IDS with one level and two level feature sets

Table 6-2. False alarm rate of IDS of one level and two level feature sets

Training Set	False Positive Rate (%)	
	APP	APP + ARCH
1	0	0.19
2	0	0.08
3	0	0.19
4	0	0.19
5	0	0.19

Table 6-3. Sample records. The label is the actual attribute of the connection, pred_1 is prediction from APP framework, pred_2 is the prediction result from APP+ARCH framework

Num	Service	type	duration	size_server	size_client	pkt_r	wrong_cks_r	ic	l1_m	l2_m	bus_acc	label	pred_1	pred_2
1	ssh	tcp	77.26	3993	2004	2.01	0.05	1.14	1.48	2.67	1.73	normal	normal	normal
2	ssh	tcp	355.16	0407	2148	0.77	0.05	55.75	3602.53	1.18	0.96	attack	normal	attack
3	ssh	tcp	219.15	11393	3540	1.62	0.03	264.11	0.81	0.84	0.87	attack	normal	attack
4	ssh	tcp	228.68	17121	3300	1.97	0.03	11.73	2.56	3.27	25.56	attack	normal	attack
...	...													

sufficient to detect the DoS intrusions accurately. Over half of attack instances successfully escape from application level monitor.

This result is expectable since we assume that our multi-step attacks can bypass the application level feature monitors and launch DoS exploits locally. The network connection behaves exactly the same as other normal connections. No typical properties such as traffic bursts of the DoS attacks could be observed at the application level. Therefore, the IDS can not differentiate them from other normal operations.

To demonstrate the effectiveness of architectural monitors, we conduct another experiment with added architectural features. The results are illustrated in the gray bar group of Figure 6-1. From the figure, we can see that the capability to detect novel multi-step DoS attacks is greatly improved to an average of 91.2% by integrating ARCH features. For training set 3, 4 and 5, we achieve a detection rate almost to 100%.

A few example records are shown in Table 6-3 to illustrate the different behaviors of malicious and benign operations monitored from multi-level features. For the ARCH event columns, we list the ratio of the numbers of the event during a session to a pre-measured normal session. The first entry is a normal *ssh* connection that is commonly seen in a local

network. The next three entries are BSB, loop, memory DoS attacks. Each of them has manifest architectural variations (see the bolded italic numbers), but the application (APP) level features stay in the same pattern as a normal connection. This explains why in a sophisticated DoS attack scenario, intrusions can escape detection from APP level. The IDS built with APP features only can not distinguish such attacks from other normal sessions. Therefore, it lacks sufficient information to make a correct judgment.

However, ARCH features also bring in false positives compared to pure APP feature framework as shown in Table 6-2. Even though the false positive rate is as low as an average of 0.17%, considering the amount of normal connections is large, over 3000 records, the actual number of false alerts is not negligible. The most challenging issue to integrate ARCH features into IDS is how to reduce false positives, since at ARCH level, memory or CPU intensive workloads, and malicious DoS attacks have similar characteristics which is difficult to differentiate at the this level.

6.2.1.2 Three Level Feature Set

To reduce false positives brought in by ARCH events, we first analyze the way by which crackers may log-in to the victim system. In practice, remote Buffer-Overflow (BoF) and guessing password are mainly used to gain unauthorized access to the target machine. After crackers gain illegal access to the victim system, a DoS attack may be launched. In this paper, we assume that an illegal user will conduct a BoF attack first to obtain access to the target system then start a DoS attack. In this scenario, we enforce the IDS with BoF detection capability with OS level monitors and then write prediction results into the system event log. We can distinguish between a normal heavy duty program and an illegal DoS attack in this

way: we search the event log and check if a BoF exploit was found in this connection before. If it was found and architectural events also show an abnormal pattern, we think that the system is under DoS attack; otherwise, we believe that there is a legal heavy duty program running on the target machine, i.e., the system is in a normal state.

We conducted experiments integrating OS level features into the IDS to detect remote BoF attacks. The OS features we employed include: *forked_socket_session*, *forked_shell*, *forked_from_shell*, *coincided_pid*. Those features are obtained using BackTracker's [9] system call tracking function embedded in the Linux kernel. Through an experiment, we achieve an average True Positive rate of 90.3%, True Negative rate of 99.6%. Detailed experiment procedure is elaborated in Appedix A.

With the highly accurate BoF detection rate, we apply the results into DoS detections in the way described in the last paragraph to reduce false alarm rate induced by ARCH monitors. As shown in the last column of Table 6-4, the false positive rate is almost reduced to zero in all of the cases. The true positive rate is slightly affected as shown by the dark bar in Figure 6-2. But its average, 90.81%, is still considered as good performance in detecting sophisticated DoS attacks.

Note that we only take BoF for example here, just to demonstrate that additional information could be utilized to reduce the false positives. Guessing password can also be accurately identified by extracting other information from the application payload data.

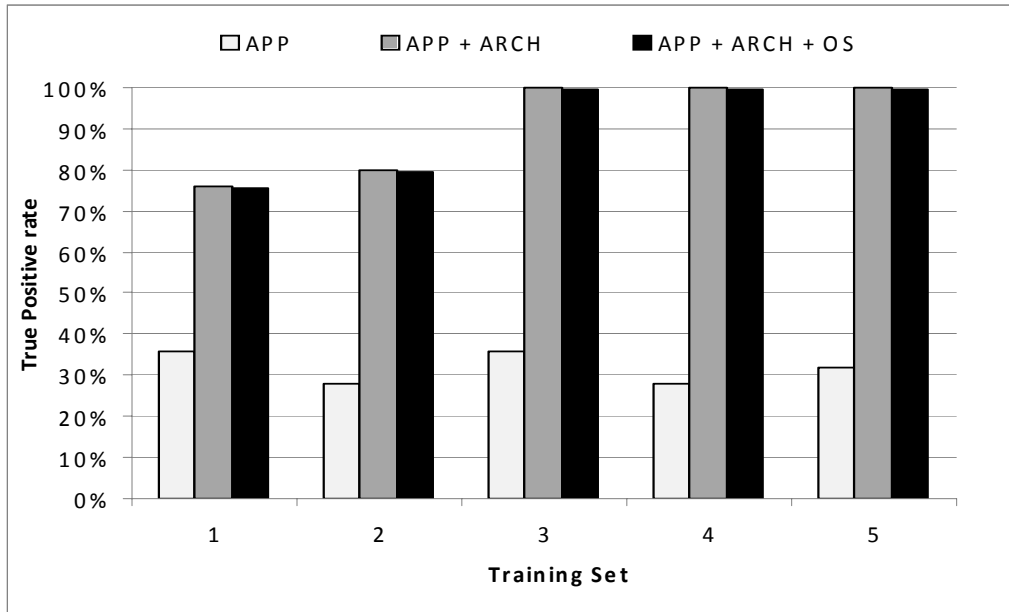


Figure 6-2. Detection rate of IDS with different feature sets

Table 6-4. False alarm rate of IDS for different feature sets

Training Set	False Positive Rate (%)		
	APP	APP + ARCH	APP + ARCH + OS
1	0	0.19	0.00051
2	0	0.08	0.00022
3	0	0.19	0.00051
4	0	0.19	0.00051
5	0	0.19	0.00051

6.2.2 Evaluation with Mixed Dataset

Apart from our crafted exploits, we also evaluate our proposed scheme using mixed data with real-world remote DoS exploits. Remote DoS exploits involve a simpler attack scenario. Attackers only need to initiate a one-step procedure: launch the attack against a target system remotely. Using this set of datasets, we intend to simulate a realistic situation that both remote DoS and sophisticated DoS exploits are mixed together. Real network traffic tend to

be sophisticated, it will rarely contain only one type of attacks. The descriptions of real-world remote exploits in the experiment are given in Table 5-3.

We also divide the data into five training datasets and one testing dataset. It is guaranteed that two exploit types are absent from the training data, while the testing data contains a full set of all exploits. A huge number of noise traffic is injected into the testing data. The strategy is intended to evaluate the ability of the IDS to detect exploits never seen before and avoid the false alarms. Results using mixed dataset (shown in Table 6-5) also prove the effectiveness of integrating architectural level features. In this experiment, the total number of normal connections is 9412 and the total number of attack instances of 472.

Table 6-5. Our IDS performance for mixed datasets

Training Set	# of False Alarms		# of Missed Attacks	
	APP	APP + ARCH + OS	APP	APP + ARCH + OS
1	28	12	2	0
2	33	11	2	0
3	50	11	3	0
4	36	17	2	0
5	47	9	2	0

6.2.3 Evaluation with Real-World Local Exploits

Having shown how ARCH features benefits the IDS using our developed exploits, we test the system with two real-world local DoS exploits (Table 5-4) separately to further demonstrate the soundness of our work. These two exploits have been used by real hackers in the wild, to impair production servers.

We use two sets of training data: one constructed with only the hand-crafted exploits, the

other one mixed with real-world remote DoS exploits (Table 6-6 and Table 6-7). Note that for both sets, first five training data only contains a subset of all exploit types, and the training 6 contains a full set of all exploit types.

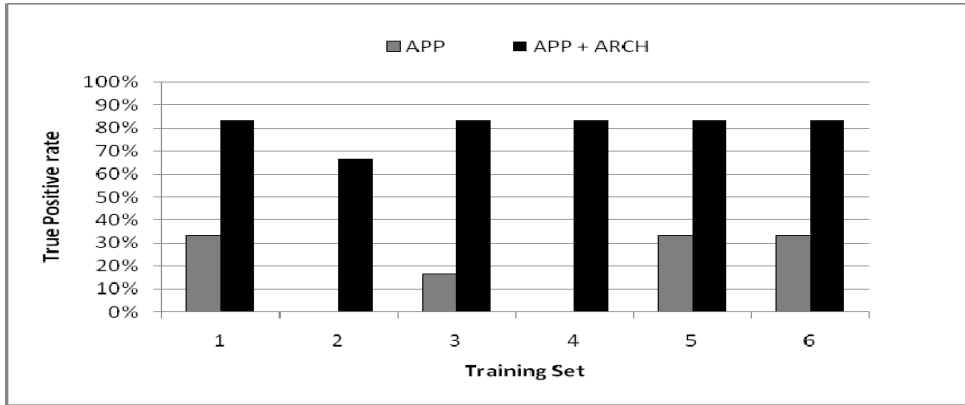
Table 6-6. Dataset constructed using developed exploits

Dataset	Combination
Training 1	l2 + bsb + fsb + mem
Training 2	l2 + bsb + fsb + loop
Training 3	l2 + bsb + mem + loop
Training 4	l2 + fsb + mem + loop
Training 5	bsb + fsb + mem + loop
Training 6	l2 + bsb + fsb + mem + loop

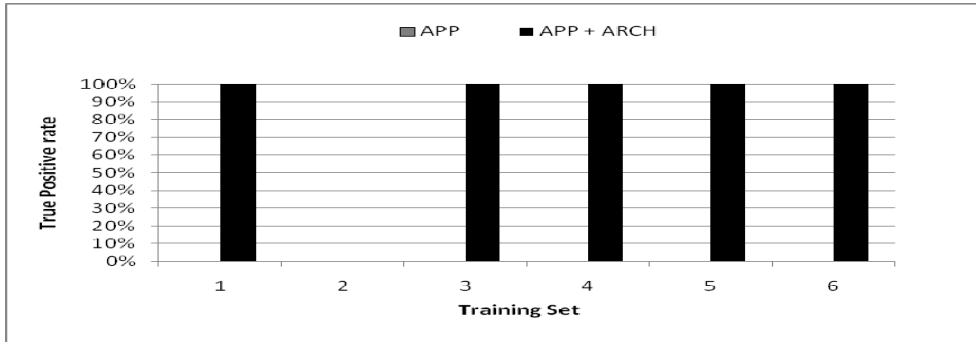
Table 6-7. Dataset constructed using mixed exploits

Dataset	Combination
Training 1	l2 + bsb + fsb + mem + real_world_exploit
Training 2	l2 + bsb + fsb + loop + real_world_exploit
Training 3	l2 + bsb + mem + loop + real_world_exploit
Training 4	l2 + fsb + mem + loop + real_world_exploit
Training 5	bsb + fsb + mem + loop + real_world_exploit
Training 6	l2 + bsb + fsb + mem + loop + real_world_exploit

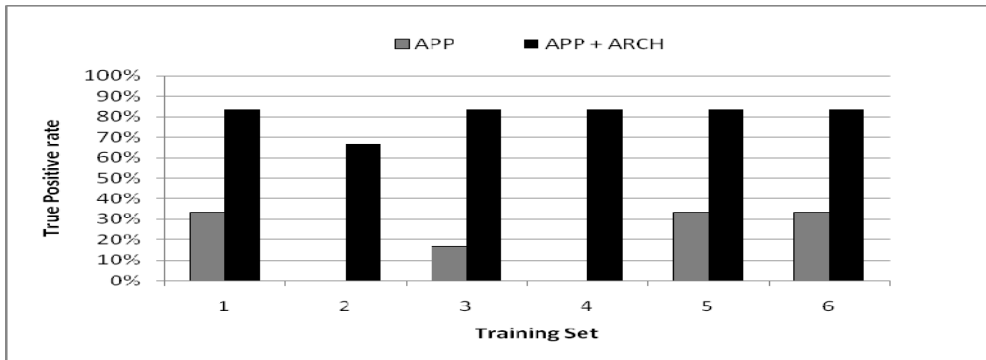
Figure 6-3 shows the comparison of True Positive rates using different training and testing datasets. Table 6-8 outlines the number of false alarms of different experimental sets. In Figure 6-3, group a's results are based on training sets listed in Table 6-6, which are combinations of self-developed exploits. The APP + ARCH IDS achieves an average detection rate of 80.6% for Mem-leak-dos-1 attack, and 80.3% for Mem-leak-dos-2 attack (five out of six datasets have 100% detection rate). Meanwhile, the APP IDS's average detection rates in these two cases are 19.4% and 0 separately.



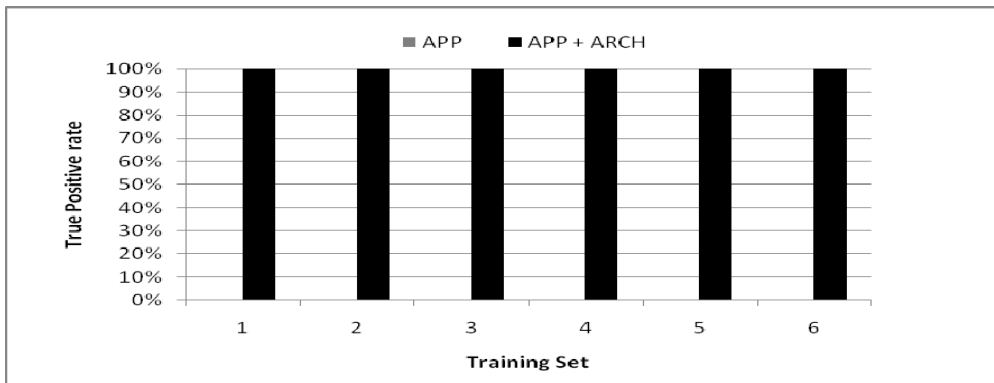
a(1) Detection rate of IDS with different feature sets tested using Mem-leak-dos-1 exploit



a(2) Detection rate of IDS with different feature sets tested using Mem-leak-dos-2 exploit



b(1) Detection rate of IDS with different feature sets tested using Mem-leak-dos-1 exploit



b(2) Detection rate of IDS with different feature sets tested using Mem-leak-dos-2 exploit

Figure 6-3. IDS performance comparison (group a's training data consists of hand-crafted exploits, group b's training data consists of mixed-data)

Table 6-8. IDS performance comparison (in each table, left group of columns indicates the IDS is trained with self-developed exploits, right group’s training data is based on mixed datasets)

a. Number of false alarms for Mem-leak-dos-1

Training Set	# of False Alarms			
	Crafted Exploits		Crafted + Real-world Exploits	
	APP	APP + ARCH	APP	APP + ARCH
1	2	0	1	2
2	1	0	1	0
3	1	0	1	0
4	1	0	1	0
5	1	0	1	0
6	1	0	1	0

b. Number of false alarms for Mem-leak-dos-2

Training Set	# of False Alarms			
	Crafted Exploits		Crafted + Real-world Exploits	
	APP	APP + ARCH	APP	APP + ARCH
1	1	0	1	2
2	1	0	4	0
3	2	0	2	0
4	1	0	0	0
5	1	0	0	0
6	1	0	0	0

The reason why the APP + ARCH IDS detects none of the mem-leak-dos-2 exploit when trained using training set 2 is that the exploit type missing from the training set, which is mem-dos, has the exact same architectural features as the attack. Therefore, even though the IDS is well trained with other exploit types, it fails to detect this particular exploit efficiently. The result of training set 6 tells that when trained with full set of all exploit types, the IDS can accurately identify all intrusion instances.

For group b, which is trained with mixed datasets of developed and real-world remote exploits as listed in Table 6-7, the average TP rates for APP + ARCH IDS are 88.9% and 100% for Mem-leak-dos-1 and Mem-leak-dos-2 separately; APP IDS can only detect 30.6% or none of those two types of attack instances. The injected real-world remote exploits in the training data improve the detection rate of APP + ARCH IDS as compared to group a. They remedy the degradation induced by absence of the mem-dos from the training data 2, since they bring in similar exploit types that have the same architectural behavior as the testing exploit. The 100% accuracy is obtained in attack detection using this set of training data. This indicates that with more comprehensive training data, our proposed IDS can achieve more accurate detection results.

Number of false alarms is shown in Table 6-8 by grouping the results by the testing data. Table 6-8(a) lists the results for two sets of training data detecting mem-leak-dos-1 attack. The APP IDS raises an average of 1.17 or 1 false alarm for two training sets, and the APP + ARCH IDS raises 0.33 or no false alarm for those two training sets. When the volume of network traffic grows, the difference of number of false alarms raised by the two IDSs will increase significantly. Table 6-8(b) shows the average number of false positives for APP IDS is 1.17 and 1 tested using mem-leak-dos-2 exploit, while the average number is 0 and 0.33 for APP + ARCH IDS.

In conclusion, the testing results also demonstrate that ARCH features are of significant use in identifying sophisticated DoS attacks. APP features alone can not reveal the intrusive behaviors by monitoring at the application level. By using our crafted exploits or real-world exploits, attackers can manage avoid detection by APP monitors, and directly induce drastic system performance degradation, with the APP monitors still showing everything is normal.

With addition of ARCH features, alarms will be triggered in this case because DoS attack can not be achieved without inducing numerous ARCH level activities. Even though attackers could escape from being caught at other levels, ARCH features will show all suspicious activities.

Chapter 7

Related Work

Modern DoS attacks employ many advanced and sophisticated techniques to amplify the damage and elude detections or mitigations of countermeasures. IP spoofing is widely adopted by hackers to mask the real source of attacks, or launch reflective DoS attacks; Distributed DoS is used to initiate attacks from multi-source; low-rate pulsing method is utilized to reduce average packet rate and evade network monitors. Based on a header analysis, frequency domain characteristics are studied to improve the IDS performance [3][6], a ramp-up behavior is also considered as a way to distinguish between single- or multi-source attacks. In [8][10], authors propose to take a spectral analysis to detect shrew attacks which consist of short time bursts repeating at a maliciously chosen low frequency. This kind of low-rate attack sends out packets at certain fixed intervals, to intentionally reduce the average packet rate, rendering the IDS unable to discover undergoing attacks. To defend against IP spoofings, various off-line IP trace-back techniques are proposed to pinpoint the real origin of DoS attack [17][18], some on-line countermeasures are also developed to filter out those spoofed packets, help sustain service availability during attacks: [7] presents a Hop-Count Filtering scheme to utilize the Time-to-Live(TTL) value in the IP header to filter out spoofed IP packets.

Recent work on intrusion countermeasures include machine learning IDS techniques, alert correlation, alert fusion and feature analysis. Machine learning techniques, such as decision tree, neural network, Bayesian network, are applied to detect network intrusions. Alert correlation attempts to correlate IDS alerts based on the similarity between alert

attributes, previously known attack scenarios, or prerequisites and consequences of known attacks [16]. Alert fusion combines detection outputs of the same attack from different independent detectors. Feature Analysis tries to optimize the information gained from multiple dimensional features through feature bagging, relevance and redundancy analysis, and feature weight classification [11][13][14][23].

In the HIDS literature, various techniques utilizing system call tracking and auditing trails are proposed. System call arguments are integrated to capture data-flow behaviors of programs, and improve attack detections in HIDS [2]. A policy-driven solution is presented in [1] to define and enforce process behavior rules controlling processes' access to system resources. All system behaviors are monitored in real-time by a modified kernel.

Basically, research works investigating DoS attack utilize sniffer-based methodologies. They only rely on analyzing network traffic information at the application level. These network-based schemes suffer from fast traffic, switched network, information encryption, and most importantly, they have little knowledge of what is really going on in the victim machine. Significant useful information on the victim host is neglected. HIDS against DoS attacks are not widely researched since it is difficult to find a generic and low-cost way to defend against such attacks. We propose to utilize the strong correlation of architectural behaviors with DoS attacks, and employ multi-layer features to construct an IDS model. Close to our work, Woo and Lee [22] have observed performance degradation of multi-threaded workload under architectural DoS attacks. However, they do not further study the correlation of architectural behavior and DoS attacks and apply into an IDS in identifying and preventing such attacks. In our work, we are exploring architecture features to enrich the existing feature set used for intrusion detection research and demonstrate its effectiveness in a

systematic approach. OS level system events are also employed in our integrated IDS to reduce the false alarms. R. Tao et al. [19] has proposed to use architectural features to improve sophisticated DoS attack detections, which is the basis of our work. We have extended the work and evaluated our IDS using more comprehensive datasets to further prove the soundness of the multi-level IDS.

Chapter 8

Conclusions

We have conducted experiments to demonstrate that an IDS using only application features failed to detect sophisticated DoS attacks because these attacks appear normal if their behaviors are only monitored by the application feature set. In order to detect the missed DoS attacks, we use a combination of application, OS, and architecture feature set. Both hand-crafted exploits and real-world exploits are used to evaluate the soundness of the multi-level IDS. Our experimental results showed improved IDS performance. In summary, we propose the idea that if crackers use sophisticated schemes to evade defense, the architectural level behavior monitored in conjunction with application and OS level features provides us valuable information to improve the IDS against such DoS attacks.

References

- [1] S. N. Chari and P. C. Cheng. BlueBoX: A Policy-Driven, Host-Based Intrusion Detection System. *TISSEC*, 2003
- [2] A. Chaturvedi, E. Bhatkar, R. Sekar. Improving Attack Detection in Host-Based IDS by Learning Properties of System Call Arguments. *In Proceedings of the IEEE Symposium on Security and Privacy*, 2006
- [3] Y. Chen, K. Hwang, and Y.-K. Kwok. *Collaborative Defense against Periodic. Shrew DDoS Attacks in Frequency Domain. TISSEC*, 2005
- [4] J. H. Friedman, Greedy function approximation: a gradient boosting machine. *The Annals of Statistics*, 29, 2001, 1189–1232.
- [5] M. Handley, C. Kreibich and V. Paxson. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. *USENIX Security Symposium*, 2001.
- [6] A. Hussain, J. Heidemann, C Papadopoulos. A Framework for Classifying Denial of Service Attack. *In Proceedings of ACM SIGCOMM*, 2003.
- [7] C. Jin, H. Wang and K. G. Shin. Hop-Count Filtering: An Effective Defense against Spoofed Traffic. *CCS* 2003.
- [8] C. Jin, H. Wang and K. G. Shin. On a New Class of Pulsing Denial-of-Service Attacks and the Defense. *NDSS*, 2005.
- [9] S. T. King, and P. M. Chen, 2003. Backtracking intrusions. *SIGOPS Oper. Syst. Rev.* 37, 5, 223-236, 2003.
- [10] A. Kuzmanovic and E.W. Knightly. Low-Rate TCP-Targeted Denial of Service Attacks. *In Proceedings of ACM SIGCOMM*, 2003.
- [11] A. Lazarevic and V. Kumar, Feature bagging for outlier detection. *In Proceedings of the Eleventh ACM SIGKDD international Conference on Knowledge Discovery in Data Mining*, 2005
- [12] W. Lee and S. Stolfo, Data mining approaches for intrusion detection, *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, 1998.
- [13] Y. Li and L. Guo, TCM-KNN scheme for network anomaly detection using feature-based optimizations. *In Proceedings of the ACM Symposium on Applied Computing*, 2008.

- [14]H. Liu and L. Yu. Towards integrating feature selection algorithms for classification and clustering. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 17(3), 2005, 1-12.
- [15]David Moore, Geoffrey M. Voelker and Stefan Savage. Inferring Internet Denial-of-Service Activity. *USENIX*, August 2001.
- [16]P. Ning and D. Xu. Hypothesizing and Reasoning About Attacks Missed by Intrusion Detection Systems. *ACM Transactions on Information and System Security*, Vol. 7(4), 2004, 591-627.
- [17]S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical network support for IP traceback. *In Proceedings of ACM SIGCOMM*, Stockholm, Sweden, August 2000.
- [18]A. C. Snoren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, S. T. Kent, and W. T. Strayer. Hash-based IP Traceback. *In Proceedings of ACM SIGCOMM '2001*, San Diego, CA, August 2001.
- [19]R. Tao, L. Yang, L. Peng, B. Li and A. Cemerlic. A Case Study: Using Architectural Features to Improve Sophisticated Denial-of-Service Attack Detections. *In Proceeding of the 2009 IEEE Symposium on Computational Intelligence in Cyber Security*, Nashville, TN, Mar. 2009.
- [20]D. Wagner, P. Soto, Mimicry Attacks on Host-Based Intrusion Detection Systems, *CCS'02*, November 18–22, 2002.
- [21]H. Wang, D. Zhang and K.Shin. Change-Point Monitoring for Detection for DoS Attacks. *TDSC*, 2004.
- [22]D. H. Woo and H.-H. S. Lee, Analyzing Performance Vulnerability due to Resource Denial-of-Service Attack on Chip Multiprocessors, *CMP-MSI*, 2007.
- [23]L. Yu and H. Liu, Efficient Feature Selection via Analysis of Relevance and Redundancy. *Journal of Machine Learning Resources*. Vol. 5, 2004, 1205-1224.
- [24]Z. Yu and J. Tsai, An efficient intrusion detection system using a boosting-based learning algorithm. *International Journal of Computer Applications in Technology*, Vol. 27 (4), 2007, 223-231.

Appendix A

Buffer Overflow IDS

1. Background

Buffer overflow is one of the most prevalent methods hackers utilize to obtain security breach. The basis for buffer overflow is that no built-in boundary check is imposed by certain programming languages, such as C, C++. The code shown in Figure 1 is a valid program that compiles with no error. But the adjacent buffer which is beyond the allocated space will be overwritten.

```
int main () {  
    int buffer[10];  
    buffer[100] = 10;  
}
```

Figure 1. Sample code of buffer overflow

Generally the hacker would utilize the programming languages' insufficient bound checking, store data beyond the boundaries of a fixed-size buffer. Adjacent memory locations would be overwritten with other buffers, variables, or program flow data, resulting in abnormal program behavior, such as memory exception, system crash, or security breach if deliberately exploited by malicious hackers.

2. Behavior of BoF Attacks

Previous techniques detecting BoF attacks focus on parsing and analyzing the payload information during a connection, extract application level features, and feed the feature events to the statistical model to build the IDS. Such schemes highly rely on characteristics certain exploits exhibit and require comprehensive domain knowledge in feature construction. In our work, we propose an adaptive anomaly network IDS utilizing extracted features from the OS level together with application level features to gain higher accuracy.

Unlike DoS attack, which involves huge number of certain activities, e.g. large amount of network requests, BoF attack generally exploits the vulnerability in a single connection by being embedded in the payload data. Network connection information from application level is not enough to identify and correlate attack sessions. We group network connections by `src_ip:src_port <-> dst_ip:dst_port`, and the remote buffer overflow will obtain a interactive shell session connected to a different port from which it originally launches the attack, resulting in multiple distinct network sessions which are difficult to identify the attack and correlate those distinct malicious connections. Analyzing the payload does not provide us useful information to solve the problem. Each exploit will exhibit different payload characteristic; they do not share a common pattern. Constructing features solely from the application level is possible, but extremely low efficient, because different exploits will need different features constructed accordingly, extensibility to cover novel attacks is not high, and requires extra time and work. However, from OS level, some typical abnormal behaviors of BoF attacks could be identified using BackTracker, which records every OS event.

Figure 2 illustrates an example observed using BackTracker's system call

```

proc_28088_0 [label="icecast" tv="1211191967.623887_28088",shape=box];
proc_28088_0 -> proc_28089_0; // EVENT: fork_event
proc_28088_0 -> shmem_51_0; // EVENT: shmem_write_interval
proc_28089_0 [label="icecast" tv="1211191977.773887, sh"
tv="1211191985.183887_28089",shape=box];
proc_28089_0 -> proc_28095_0; // EVENT: fork_event
proc_28095_0 [label="sh" tv="1211192006.413887, 12"
tv="1211192006.453887_28095",shape=box,style=filled];
proc_114_0 [label="rc.sysinit" tv="1210778795.993805, mount"
tv="1210778795.993805_114",shape=box];
proc_114_0 -> file_229940_0; // EVENT: write_event
proc_121_0 [label="rc.sysinit" tv="1210778796.043805, mount"
tv="1210778796.043805_121",shape=box];
proc_121_0 -> file_229940_0; // EVENT: write event
socket_542490_0 [label="130.39.149.110:15604 <-> 130.39.149.107:8000,
3957079625",shape=diamond];
socket_542490_0 -> proc_28088_0; // EVENT: socket recv
socket_542505_0 [label="130.39.149.110:7058 <-> 130.39.149.107:30464,
3963829249",shape=diamond];
socket_542505_0 -> proc_28089_0; // EVENT: socket_recv

```

Figure 2. Observation of BoF exploit using BackTracker

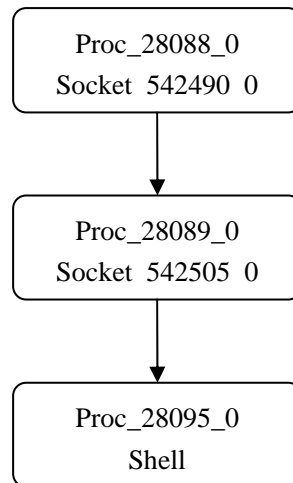


Figure 3. Execution flow of a sample BoF exploit

tracking function. The highlighted system events exhibit abnormal behaviors that could assist us to identify BoF exploit. As shown in Figure 3, the execution flow of this BoF exploit consists of three steps:

Step 1: Attacker connects to target system from network

Step 2: Malicious network connection spawn another different network session

Step 3: Obtain interactive shell session through the forked network session

3. System Call Tracking

In our work, BackTracker's OS level event logging and dependency tracking is utilized to extract the OS level features of each network connections. BackTracker discovers sequences of steps that occurred during an intrusion. Starting at a single detection point (e.g., a suspicious file), it identifies files and processes that could have affected that detection point and displays chains of events in a dependency graph. BackTracker is able to record every dependency-causing event among OS objects, and span up a dependency chain starting from one object. Complete information such as process forking, file operations, and program execution (which is important for security analysis), is recorded in a system-call oriented manner. That information could be utilized to provide a novel view of exploit behavior analysis from the OS level.

4. IDS Implementation

We integrate both the application level and OS level features. The application features we find informative in the case of remote BoF attacks are: *type*, *service*, *duration*, *size_from_server*, *size_from_client*, *packet_rate*, *wrong_checksum_rate*. And we also add four OS features (Table 1) according to our observation from the OS level to improve the prediction accuracy of our IDS: *forked_socket_session*, *forked_shell*, *forked_from_socket*, *coincided_pid*.

Table 1. OS level features

Features	Description
forked_socket_session	forked another network connection
forked_shell	forked shell sessions
forked_from_shell	forked from another network connection
coincided pid	share a same pid as another different network connection

These features in Table 1 are observable using a modified BackTracker. Modifications to BackTracker include attaching a timestamp to every system event; exporting detailed IP address and port number for socket sessions. With those changes, we can correlate high level network session records with system events recorded by BackTracker’s system call tracking function embedded in Linux kernel.

5. IDS Performance

Exploits we used to train and test our BoF IDS are all from real-world. We successfully utilized those tools to exploit targeted software vulnerabilities, and obtained interactive shell sessions. Table 2 gives detailed information of BoF exploits we employed.

The 60 exploit instances are divided into five training datasets, with one type of exploits absent from each training set. And the testing data include all types of BoF exploits. Testing result shown in Figure 4 demonstrates that OS level events help improve the accuracy of True Positive rate. An average True Negative rate of 99.6% is also considered as a good performance.

Table 2. Description of BoF exploits

CVE Number	Name	Description
CVE-2002-0177	Icecast AVLLib Buffer Overflow Vulnerability	Remote user may send arbitrary long string to the server, leading to a stack overflow and execution of user supplied code. Execution privilege of the Icecast server will be obtained.
CVE-2003-0201	Samba 'call_trans2open' Remote Buffer Overflow Vulnerability	Anonymous user may corrupt sensitive locations in memory and execute arbitrary commands by passing excessive user-supplied data into a static buffer.
CVE-2004-0396	CVS Malformed Entry Modified and Unchanged Flag Insertion Heap Overflow Vulnerability	A remote heap overflow could occur when handling user-supplied input for entry lines with 'modified' and 'unchanged' flags, possibly leading to arbitrary code execution.
NA	Dr.Cat Drcatd Multiple Buffer Overflow Vulnerabilities	Unauthorized access and/or elevated privilege on the vulnerable system may be achieved by exploiting the vulnerability of insufficient boundary checks of some functions of this application.

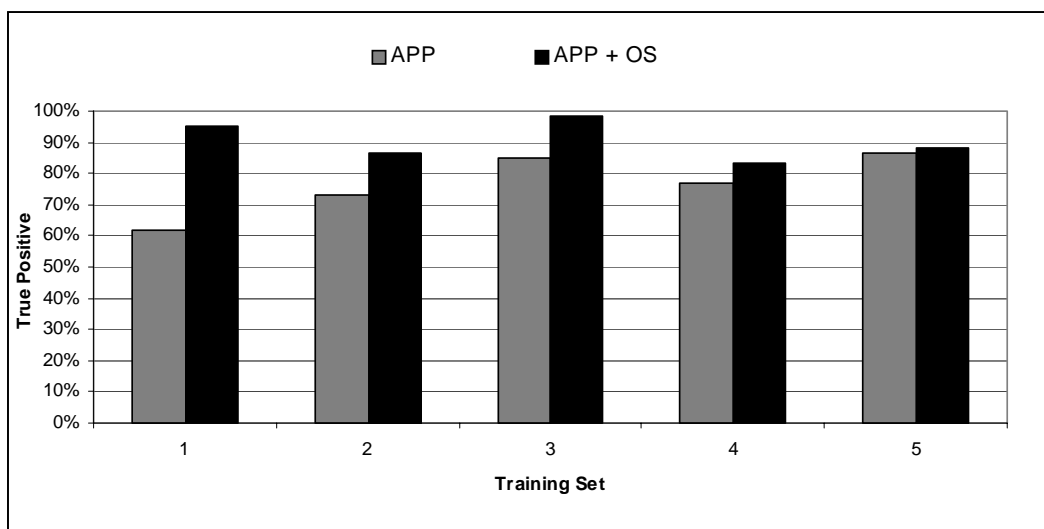


Figure 4. True Positive rate comparison

Appendix B

Control Register Layout for Pentium D

ESCR MSR:

The layout of ESCR MSR is illustrated in Figure 1.

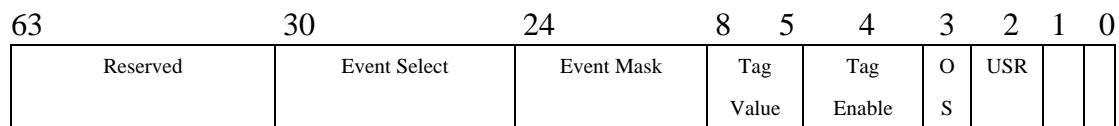


Figure 1. ESCR layout

Bit 0-1: Reserved

Bit 2: USR flag – set counter to count events when processor operates in user mode

Bit 3: OS flag – set counter to count events when processor operates in privileged level.

Bit 4: Tag enable – enable tagging of μ ops.

Bit 5-8: Tag value field – select a tag value to associate with a μ op.

Bit 9-24: Event mask field – select events to count from event class selected

Bit 25-30: Event select field – select a class of events to count.

CCCR MSR:

CCCR MSR controls the filtering and counting of events, together with interrupt generation. The layout of CCCR MSR is illustrated in Figure 2.

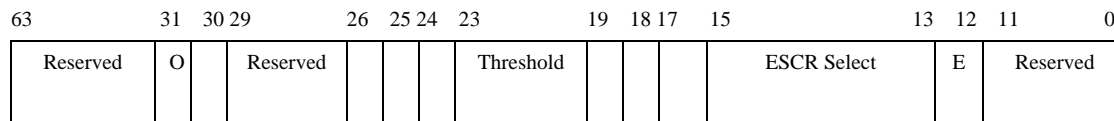


Figure 2. CCCR layout

Bit 0-11: Reserved.

Bit 12: Enable flag – enable events counting

Bit 13-15: ESCR select – select the ESCR to be used to select the events to be counted.

Bit 16-17: Reserved – must be set to 11B.

Bit 18: Compare flag – enable filtering of the event counting.

Bit 19: Complement flag – configure how the event count is compared with the threshold value.

Bit 20-23: Threshold value – set the threshold value to be compared with.

Bit 24: Edge flag – enable rising edge detection of filtering event counts.

Bit 25: FORCE_OVF flag – force a counter overflow on every counter increment.

Bit 26: OVF_PMI flag – enable a performance monitor interrupt (PMI) to be generated when the counter overflow occurs.

Bit 30: Cascade flag – enable one counter in a pair to start counting when the other counter overflows.

Bit 31: OVF flag – indicate that the counter has overflowed when set

Vita

Ran Tao was born in Chengdu, Sichuan, China. She obtained the bachelor's degree in electrical engineering, from the University of Electronic Science and Technology of China, in June 2005. In August 2006, she joined the department of Electrical and Computer Engineering, Louisiana State University to pursue graduate study in computer architecture. She is currently a candidate for the degree of Master of Science in Electrical Engineering, which will be awarded in August 2009.