A Case Study: Using Architectural Features to Improve Sophisticated Denial-of-Service Attack Detections

Ran Tao[1], Li Yang[2], Lu Peng[1], Bin Li[3], Alma Cemerlic[2]
[1]Department of Electrical and Computer Engineering, Louisiana State University
[2]Department of Computer Science and Engineering, University of Tennessee at Chattanooga
[3]Department of Experimental Statistics, Louisiana State University

***Abstract* — Application features such as port numbers are used by Network-based Intrusion Detection Systems (NIDSs) to detect attacks coming from networks. System calls and the operating system related information are used by Host-based Intrusion Detection Systems (HIDSs) to detect intrusions towards a host. However, the relationship between hardware architecture events and Denial-of-Service (DoS) attacks has not been well revealed. When increasingly sophisticated intrusions emerge, some attacks are able to bypass both the application and the operating system level feature monitors. Therefore, a more effective solution is required to enhance existing HIDSs. In this paper, we identify the following hardware architecture features: *Instruction Count*, *Cache Miss*, *Bus Traffic* and integrate them into a novel HIDS framework based on a modern statistical Gradient Boosting Trees model. Through the integration of application, operating system and architecture level features, our proposed HIDS demonstrates a significant improvement of the detection rate in terms of sophisticated DoS intrusions.**

I. INTRODUCTION

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 <*source IP*, *destination IP*, *source port*, *destination port*>. 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][20]. 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 able to detect this intrusion. In such a scenario, non-privileged access is well 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.

In this paper, 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. According to our experiments, even if DoS attacks could successfully evade captures of NIDS monitors, architectural behaviors will still be triggered: a tremendous jump 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 archi-

tecture features. Our experiments show that the inclusion of architecture features can significantly improve the detection rate of such evasive DoS intrusions.

The rest of this paper is organized as follows: related work is discussed in section 2; our proposed IDS methodology and framework is elaborated in section 3. The experiment results are shown and discussed in section 4. We conclude the paper in Section 5.

II. 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][22].

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 [21] 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.

III. The IDS Framework

A. 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.

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 a format of "src_ip:src_port <-> dst_ip:dst_port". Architectural records are processed as a ratio of event numbers during the current session to a pre-measured normal session without attacks. 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

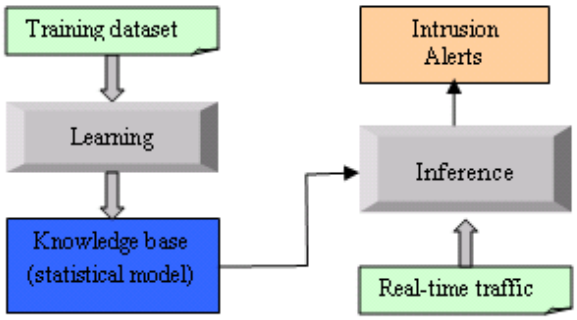

Fig.1. The framework of our Intrusion Detection System

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.

B. The IDS Framework

a. General Structure

The framework of our proposed intrusion detection system consists of a learning module and an inference module as shown in Fig. 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 an attack. 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 the following application and architectural level features.

1. Application (APP) Features: *protocol_type, service, duration, size_from_client, size_from_server, packet_rate and wrong_checksum_rate.*

2. Architectural (ARCH) Features: *instruction_retired, L1_cache_miss, L2_cache_miss, and bus_access.* We select them because a typical network DoS attack can be monitored by observing these events [21].

b. Gradient Boosting Trees (GBT) 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 the following advantages. First, GBT is inherently non-parametric and can handle mixed-type of input variables. Both discrete and continuous data are supported. There is no need of data discretization. GBT doesn't need to make any assumptions regarding the underlying distribution of the values for the input variables. For example, GBT can relieve 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 [23].

Consider the binary classification problem with n observations of the form $\{y_i, \boldsymbol{x}_i\}$, i=1,…,n, where $\boldsymbol{x}_i$ is a multi-dimensional input vector and y_i is the binary response $y_i \in \{-1,+1\}$. In this paper, $\boldsymbol{x}_i$ is the feature in multiple levels and y_i is the prediction result, i.e., attack or not. The negative log-likelihood for the binomial model or *deviance* (also known as cross-entropy) is used as the loss function:

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

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

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

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

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

   a) Set the negative gradient

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

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

   c) $\gamma_{hm} = \arg\min_{\gamma}\sum_{\mathbf{x}_i\in R_{hm}} L\left(y_i,\hat{f}(\mathbf{x}_i)+\gamma\right).$

   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.

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 1. The self-developed DoS 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
Testing	l2 + bsb + fsb + mem + loop + noise

Table 2. Dataset construction

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. At 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.

IV. Experimental Results

We use an Intel Pentium-D PC installed with Redhat Linux 9.0, and connect it to a department LAN. Network traffic information is captured with the *tcpdump* tool. 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.

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.

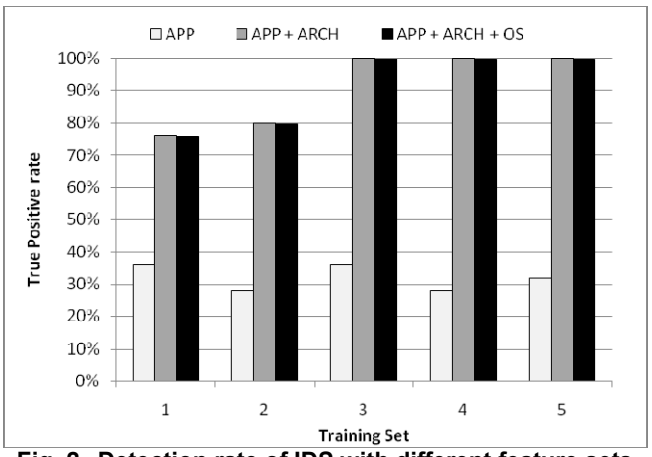

Fig. 2. Detection rate of IDS with different feature sets

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 not have a root- privilege), 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 type target a type of system resources, intentionally exhausting a particular resource, and rendering the system unavailable to legitimate users. Detailed descriptions are shown in Table 1. First three attacks are traversing a certain memory space with the stride of the cache line size (64 bytes in our system).

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 are listed in Table 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.

Firstly, we train an IDS using the GBT model solely with the application level features listed in Section 3. The testing results demonstrate that the APP features are not sufficient to detect the testing DoS intrusions accurately. Results shown in Fig. 2 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 Fig. 2).

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

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	10407	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
...	...													

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

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

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

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. Real-world DoS exploits

features. The results are illustrated in the gray bar group of Fig. 2. 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 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. The IDS built with only APP features 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. 4. 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

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

Table 6. Our IDS performance for mixed datasets

ARCH level, memory or CPU intensive workloads, and malicious DoS attacks have similar characteristics which is difficult to differentiate at the this level.

To solve this problem, 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 then record 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 99.7%. 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 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 Fig. 2. But its average, 90.96%, is still considered as good performance in detecting sophisti-

cated 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.

Apart from our crafted exploits, we also evaluate our proposed scheme using mixed data with real-world exploits. The description of real-world exploits in the experiment is listed in Table 5.

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) 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.

V. CONCLUSION

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 and architecture feature set. 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 provides us valuable information to improve the IDS against such DoS attacks.

ACKNOWLEDGEMENT

This work is supported in part by the Louisiana Board of Regents grant LEQSF (2006-09)-RD-A-10 and the Louisiana State University. This work is supported in part by the Tennessee Higher Education Commission's Center of Excellence in Applied Computational Science and Engineering R04-1302-085 (2006-09) and Odor Wheeler Center R04-1024-032 (2008-09). Anonymous referees provide helpful comments.

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] D. Wagner, P. Soto, Mimicry Attacks on Host-Based Inrusion Detection Systems, *CCS'02,* November 18–22, 2002

[20] H. Wang, D. Zhang and K.Shin. Change-Point Monitoring for Detection for DoS Attacks. *TDSC*, 2004.

[21] D. H. Woo and H.-H. S. Lee, Analyzing Performance Vulnerability due to Resource Denial-of-Service Attack on Chip Multiprocessors, *CMP-MSI,* 2007.

[22] L. Yu and H. Liu, Efficient Feature Selection via Analysis of Relevance and Redundancy. *Journal of Machine Learning Resources*. Vol. 5, 2004, 1205-1224.

[23] 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.