Chapter 15

DETECTING REMOTE EXPLOITS USING DATA MINING

Mohammad Masud, Latifur Khan, Bhavani Thuraisingham,
Xinran Wang, Peng Liu and Sencun Zhu

Abstract This paper describes the design and implementation of DExtor, a data-mining-based exploit code detector that protects network services. DExtor operates under the assumption that normal traffic to network services contains only data whereas exploits contain code. The system is first trained with real data containing exploit code and normal traffic. Once it is trained, DExtor is deployed between a web service and its gateway or firewall, where it operates at the application layer to detect and block exploit code in real time. Tests using large volumes of normal and attack traffic demonstrate that DExtor can detect almost all the exploit code with negligible false alarm rates.

Keywords: Server attacks, exploit code, data mining, attack detection

1. Introduction

Remote exploits are often used by attackers to gain control of hosts that run vulnerable services or software. Typically, an exploit is sent as an input to a remote vulnerable service to hijack the control flow of machine instruction execution. Attackers sometimes inject executable code in the exploit that is run after a successful hijacking attempt. We refer to such remote code-carrying exploits as "exploit code."

Several approaches have been proposed for analyzing network flows to detect exploit code [1, 4, 8–11]. An attack can be prevented if an exploit is detected and intercepted while it is in transit to a server. This approach is compatible with legacy code and does not require changes to the underlying computing infrastructure. Our solution, DExtor, follows this strategy. In particular, it uses data mining to address the general problem of exploit code detection.

Exploit code usually consists of three parts: (i) a NOP sled at the beginning of the exploit, (ii) a payload in the middle, and (iii) return addresses at the end. The NOP sled is a sequence of NOP instructions; the payload contains the attack code; the return addresses point to the code to be executed. Thus, exploit code always carries some valid executables in the NOP sled and payload. It is considered to be an "attack input" to the corresponding vulnerable service; inputs that do not exploit a vulnerability are referred to as "normal inputs." For example, in the case of a vulnerable HTTP server, benign HTTP requests are normal inputs while requests that exploit a vulnerability are attack inputs. If we assume that normal inputs only contain data, then exploit code detection reduces to a code detection problem.

Chinchani and Berg [1] justify this assumption by maintaining that "the nature of communication to and from network services is predominantly or exclusively data and not executable code." However, certain exploits do not contain code (e.g., integer overflow exploits and return-to-`libc` exploits); we do not consider such exploits in this work. It is also worth mentioning that exploit code detection is fundamentally different from malware detection, which attempts to identify the presence of malicious content in an executable.

Our data mining approach uses three types of features to differentiate between attack inputs and normal inputs. They are: (i) useful instruction count, (ii) instruction usage frequency, and (iii) code vs. data length. The process has several steps. First, training data consisting of attack inputs and normal inputs is collected. Next, the training examples are disassembled. Following this, the three types of features are extracted from the disassembled data. Several classifiers (Support Vector Machine (SVM), Bayes Net, Decision Tree (J48) and Boosted J48) are then trained and the best classifier is selected as the classification model. When DExtor is deployed in a networking environment, it intercepts inputs destined to the network service and tests them against the classification model; attack inputs are blocked in real time.

DExtor has several advantages over existing exploit code detection techniques. DExtor is compatible with legacy code and transparent to the services it protects. The current version operates on Windows platforms with the Intel 32-bit architecture, but can be adapted to any operating system and hardware simply by modifying the disassembler. Also, DExtor does not require any signature generation and matching. Finally, DExtor is robust against most attack-side obfuscation techniques.

DExtor also has forensic applications. For example, it can be used to analyze network traffic sent to a server before a crash or compromise.

This helps determine whether the incident was caused by a code-carrying exploit and also assists in identifying the source of the attack.

2. Related Work

Several techniques have been proposed for detecting exploits in network traffic and protecting network services. The three main categories of techniques are signature matching, anomaly detection and machine-code analysis.

Signature matching is used in intrusion detection systems such as Snort [8] and Bro [4]. These systems maintain a signature database of known exploits; an alert is raised when traffic matches a signature in the database. Signature-based systems are easy to implement, but they are defeated by new exploits as well as by polymorphism and metamorphism. DExtor does not use signature matching to detect exploit code.

Anomaly detection techniques identify deviations in traffic patterns and raise alerts. Wang and co-workers [11] have designed PAYL, a payload-based system that detects exploit code by computing several byte-level statistical measures. Other anomaly-based detection systems are an enhanced version of PAYL [10] and FLIPS [5]. DExtor differs from anomaly-based systems in two respects. First, anomaly-based systems are trained using normal traffic characteristics and detect deviations from these characteristics; DExtor considers both normal and attack traffic in building its classification model. Second, DExtor uses instruction patterns instead of raw byte patterns to construct its classification model.

Machine code analysis techniques apply binary disassembly and static analysis of network traffic to detect the presence of executables. DExtor falls in this category. Toth and Kruegel [9] have used binary disassembly to find long sequences of executable instructions and identify the presence of a NOP sled. DExtor also applies binary disassembly, but it does not need to identify a NOP sled. Like DExtor, Chinchani and Berg [1] detect exploit code based on the assumption that normal traffic should contain no code. They apply disassembly and static analysis, and identify several structural patterns and characteristics of code-carrying traffic. However, unlike DExtor, their detection approach is rule based. SigFree [12] also disassembles inputs to server processes and applies static analysis to detect the presence of code. It applies a code abstraction technique to locate useful instructions in the disassembled byte stream and raises an alert when the useful instruction count exceeds a predetermined threshold. DExtor applies the same disassembly technique as SigFree, but does not use a fixed threshold. Instead,

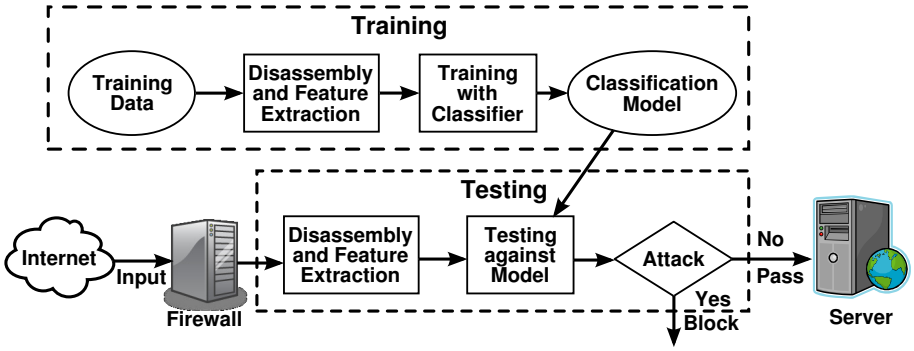

Figure 1. DExtor architecture.

it applies data mining to extract features and uses them to distinguish between normal traffic and exploits.

3. DExtor

This section describes the DExtor architecture and its main components.

3.1 DExtor Architecture

DExtor is deployed in a network between a network service and its gateway or firewall (Figure 1). It is first trained offline with real instances of attacks (e.g., exploits) and normal inputs (e.g., HTTP requests), and a classification model is constructed. Training consists of three steps: disassembly, feature extraction and classification. When it is deployed in a network, DExtor intercepts and analyzes all inputs to the service in real time; inputs that are identified as attacks are blocked.

3.2 Data Disassembly

The disassembly algorithm is similar to that used by SigFree [12]. Each input to the server is considered to be a byte sequence. There may be more than one valid assembly instruction sequences corresponding to a given byte sequence. The disassembler uses an "instruction sequence distiller" to filter redundant and illegal instruction sequences. The main steps of this process are:

- Step 1: Generate instruction sequences
- Step 2: Prune subsequences
- Step 3: Discard smaller sequences

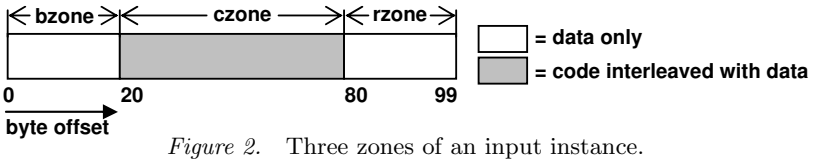

Figure 2. Three zones of an input instance.

- Step 4: Remove illegal sequences
- Step 5: Identify useful instructions

3.3 Feature Extraction

Feature extraction is the heart of DExtor's data mining approach. Three important features are used: (i) useful instruction count, (ii) instruction usage frequency, and (iii) code vs. data length.

- **Useful Instruction Count:** The useful instruction count (UIC) is the number of useful instructions found in Step 5 of the disassembly process. This feature is important because a real executable should have a large number of useful instructions; on the other hand, pure data should have no useful instructions.

- **Instruction Usage Frequency:** The instruction usage frequency (IUF) is the frequency of an instruction in a normal or attack sample. Intuitively, normal data should not have any bias toward any specific instruction or set of instructions. Thus, normal data should have a random IUF distribution. On the other hand, since exploit code performs specific (malicious) activities, it must have a bias toward a set of instructions and its IUF distribution should have some pattern.

- **Code vs. Data Length:** Exploit code has a NOP sled, payload and return addresses. Consequently, each input instance is divided into three zones: beginning zone (bzone), code zone (czone) and remainder zone (rzone) (Figure 2). Typically, the bzone corresponds to the first few bytes of an input that could not be disassembled and contains only data. The czone follows the bzone and contains the bytes that were successfully disassembled; it probably contains some code. The rzone contains the remaining bytes in the input that cannot be disassembled; it generally contains only data.

  The normalized lengths (in bytes) of the three zones should have different distributions for normal inputs and attack inputs. Intuitively, normal inputs should have the czone at any location with equal probability, implying that the bzone and rzone distributions

should be random. Also, since normal inputs have little to no code, the length of the czone should be near zero. On the other hand, exploit code begins with a NOP sled, which implies that the length of bzone is zero. Also, the length of the czone for exploit code should be greater than that for normal inputs. Thus, the differences in the distributions of zone lengths for normal and attack inputs can be used to identify the type of input.

3.4 Feature Combination

The features computed for each input sample are: (i) UIC – a single integer, (ii) IUF – k integers denoting the instruction frequencies (k is the number of different instructions in the training data), and (iii) CDL – three real numbers corresponding to the lengths of bzone, czone and rzone. Thus, $k+4$ features are considered: the first $k+1$ feature values are integers and the last three are real numbers. These $k+4$ features constitute the combined feature vector for an input instance.

3.5 Classification

The Support Vector Machine (SVM), Bayes Net, Decision Tree (J48) and Boosted J48 are used for classification. The SVM classifier is robust to noise and high dimensionality; also, it can be fine-tuned to perform efficiently in a problem domain. The Bayes Net classifier is capable of finding the interdependencies existing between different attributes. The Decision Tree (J48) classifier has an excellent feature selection capability, and requires much less training and testing time than other classifiers. The Boosted J48 classifier is useful because of its ensemble methods.

4. Experimental Setup and Results

This section describes the experimental setup and results.

4.1 Data Set

The data set contained real exploit code as well as normal traffic to web servers. Strong efforts were undertaken to ensure that the data set was as diverse, unbiased and realistic as possible.

The exploit code was obtained by generating twenty unencrypted exploits using the Metasploit framework [7]. Next, nine polymorphic engines (ADMmutate [6], clet [2], Alpha2, CountDown, JumpCallAdditive, Jumpiscodes, Pex, PexFnstenvMov and PexFnstenvSub) were applied to the unencrypted exploits. Each polymorphic engine was used to generate 1,000 exploits, yielding a collection of 9,000 exploits.

The normal inputs were traces of HTTP requests/responses to/from a web server. The traces were collected by installing a client-side proxy that monitored and captured all incoming and outgoing messages. More than 12,000 messages containing HTTP requests and responses were collected. The responses comprised text (`.javascript`, `.html`, `.xml`), applications (`x-javascript`, `.pdf`, `.xml`), images (`.gif`, `.jpeg`, `.png`), audio (`.wav`), and flash content.

Two types of evaluation were performed on the data. First, five-fold cross validation was conducted to measure the accuracy and the false positive and false negative rates. Second, the performance of the classifiers was tested on new exploits. This was done by training a classifier using the exploits generated by eight of the nine polymorphic engines and testing it using the exploits generated by the ninth engine. The test was performed nine times by rotating the polymorphic engine that was tested. Normal examples were distributed in the training set and test set in equal proportions.

4.2 Experiments

The experiments were run on a 2 GHz Windows XP machine with 1 GB RAM. The algorithms were written in Java and compiled with JDK version 1.5.0_06. The Weka ML Toolbox [13] was used for the classification tasks. SVM classification used the C-Support Vector Classifier (C-SVC) with a polynomial kernel and $\gamma = 0.01$. Bayes Net used a simple estimator with $\alpha = 0.5$ and a hill-climbing search for the network learning. J48 used tree pruning with $C = 0.25$. Ten iterations of the AdaBoost algorithm were performed to generate ten models. Each of the three features was tested alone on a classifier (with the classifier being trained and tested with the same feature).

4.3 Results

Three metrics were used to evaluate the performance of DExtor: accuracy (ACC) and the false positive (FP) and false negative (FN) rates. ACC is the percentage of correctly classified instances, FP is the percentage of negative instances incorrectly classified as positive instances, and FN is the percentage of positive instances incorrectly classified as negative instances.

Table 1 presents the performance of the classifiers for various features. The highest accuracy (99.96%) was obtained for DExtor's combined feature (Comb) with the Boosted J48 classifier. The other features have lower accuracies than the combined feature for all the classification techniques. Also, the combined feature has the lowest false positive

Table 1. Performance of classifiers for different features.

Feature	**IUC**	**IUF**	**CDL**	**Comb**
Metric	ACC/FP/FN	ACC/FP/FN	ACC/FP/FN	ACC/FP/FN
SVM	75.0/3.3/53.9	99.7/0.2/0.1	92.7/12.4/0.6	99.8/0.1/0.2
Bayes Net	89.8/7.9/13.4	99.6/0.4/0.4	99.6/0.2/0.6	99.6/0.1/0.9
J48	89.8/7.9/13.4	99.5/0.3/0.2	99.7/0.3/0.3	99.9/0.2/0.1
Boosted J48	89.7/7.8/13.7	99.8/0.1/0.1	99.7/0.3/0.5	99.96/0.0/0.1
SigFree		38.5/0.2/88.5		

rate (0.0%) obtained with Boosted J48. The lowest false negative rate was also obtained for the combined feature (0.1%). In summary, the combined feature with Boosted J48 classifier produced near perfect detection.

The last row of Table 1 shows the accuracy and false alarm rates of SigFree with the same data set. SigFree used UIC with a fixed threshold of 15. It has a low false positive rate (0.2%), a high false negative rate (88.5%) and an overall accuracy of only 38.5%.

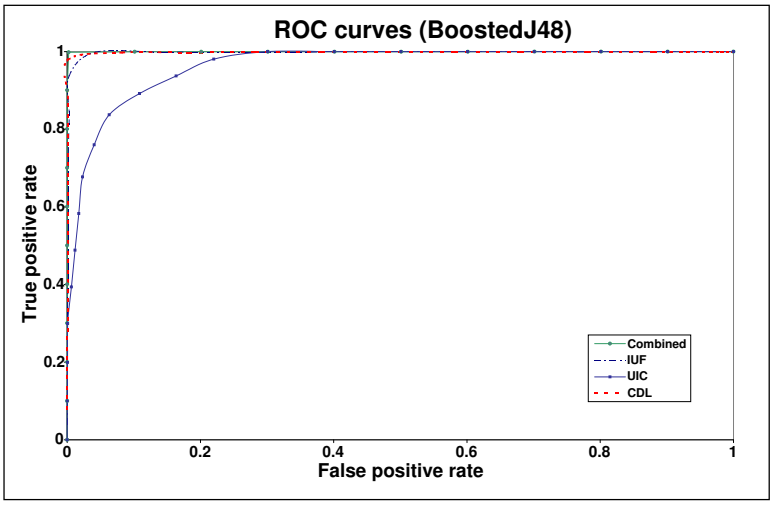

Figure 3. ROC curves for different features with BoostedJ48.

Figure 3 shows the receiver operating characteristic (ROC) curves for different features with the BoostedJ48 classifier. The area under the curve (AUC) is the highest for the combined feature (which is 0.999). The ROC curves for the other classifiers have similar characteristics; they are not presented due to space limitations.

Table 2. Effectiveness at detecting new exploits.

Classifier	**SVM**	**BNet**	**J48**	**BJ48**
Metric	ACC/FP/FN	ACC/FP/FN	ACC/FP/FN	ACC/FP/FN
Admutate	86.4/0.2/31.7	57.4/0.0/100	98.2/0.0/4.3	99.7/0.0/0.6
Alpha2	99.9/0.07/ 0.0	56.4/0.0/100	56.4/0.0/100	56.4/0.0/100
Clet	100/0.0/0.0	99.6/0.07/0.8	99.9/0.1/0.0	99.9/0.07/0.0
CountDown	99.8/0.4/0.0	100/0.0/0.0	100/0.0/0.0	99.8/0.3/0.0
JmpCallAdditive	100/0.0/0.0	98.1/0.0/4.6	99.9/0.1/0.0	100/0.0/0.0
JumpisCode	99.4/0.08/1.4	96.2/0.08/8.8	99.9/0.07/0.0	99.9/0.07/0.1
Pex	99.7/0.2/0.4	99.4/0.0/1.4	99.8/0.2/0.2	99.8/0.1/0.3
PexFnStenvMov	99.9/0.0/0.0	99.1/0.0/2.1	99.9/0.07/0.1	99.9/0.0/0.2
PexFnStenvSub	99.7/0.2/0.3	99.3/0.0/1.7	99.8/0.08/0.1	99.9/0.08/0.0

Table 2 shows DExtor's ability to detect new kinds of exploits. Each row reports the detection accuracies and false alarm rates for one particular engine-generated exploit. As described earlier, each classifier was trained using the exploits generated by the eight other engines and tested using exploits from the ninth engine. For each engine, the training set contained 8,000 exploits and about 10,500 randomly selected normal samples, and the test set contained 1,000 exploits and about 1,500 randomly chosen normal samples. The results in Table 2 show that all the classifiers successfully detected the new exploits with an accuracy of 99% or higher.

The total training time was less than 30 minutes, including disassembly time, feature extraction time and classifier training time. This amounts to about 37 ms per kilobyte of input. The average testing time for the combined feature set was 23 ms per kilobyte of input, including disassembly time, feature value computation time and classifier prediction time. SigFree, on the other hand, required 18.5 ms for testing each kilobyte of input. Since training is performed offline, DExtor requires only 24% more running time than SigFree. Thus, the price-performance trade-off is in favor of DExtor.

4.4 Analysis of Results

Figure 4 (left-hand side) shows the IUF distributions of the 30 most frequently used instructions in normal inputs and attack inputs. Clear differences are seen in the two distributions. The first five instructions have high frequencies (> 11) for attack inputs, but have zero frequencies for normal inputs. The next sixteen instructions in attack inputs have frequencies close to two while the corresponding frequencies for normal

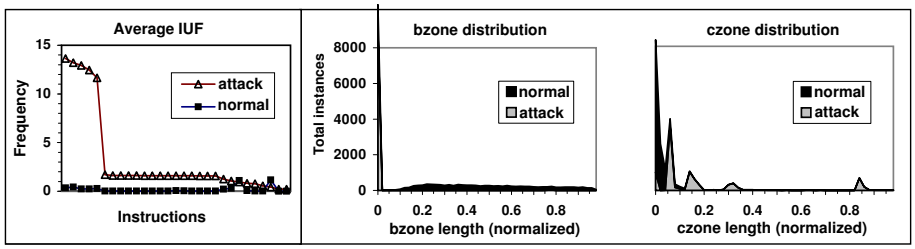

Figure 4. Instruction usage frequencies and zone length distributions.

inputs are again near zero. An attacker who intends to write exploit code that mimics normal inputs should avoid using these 21 instructions, but it is difficult to create exploits without using these instructions.

Figure 4 also presents the distributions of the CDL feature values. The histograms show the numbers of input samples having specific lengths (as a fraction of total input size) for bzone (center) and czone (right-hand side). The histograms are generated by dividing the entire range of normalized bzone and czone lengths ([0, 1]) into 50 equal-sized bins, and counting the number of input instances that fall in each bin. Note that most of the attack samples in the bzone histogram have values in the first bin (i.e., [0, 0.02)); on the other hand, the bzone values for normal samples are spread over all the bins. Therefore, an attacker wishing to mimic normal traffic should craft exploits that do not have any code in the first 10% of the exploit; but this is difficult to accomplish because exploits begin with a NOP sled. Similarly, the czone histogram shows that most of the normal samples have czone values in the range [0, 0.05] whereas attack samples mostly have czone values greater than 0.05. Therefore, in order to mimic normal traffic, an attacker should keep his code length within 5% of the exploit's length. For a 200-byte exploit, this leaves only 10 bytes for the attack code – including the NOP sled, making it extremely difficult to write the exploit.

5. DExtor Characteristics

This section discusses the robustness of DExtor's exploit code detection methodology along with its limitations.

5.1 Robustness

DExtor is immune to instruction re-ordering because instruction order is not considered in exploit code detection. Also, detection is unaffected by the insertion of junk instructions as this only serves to increase the frequencies of the junk instructions. Likewise, DExtor is immune to

instruction replacement as long as all the most frequently used instructions are not replaced. DExtor is also robust against register renaming and memory re-ordering because registers and memory locations are not considered in exploit detection. Obfuscation by inserting junk bytes can affect the disassembler, especially when junk bytes are inserted at locations that are not reachable at run-time. However, this problem is addressed by the recursive traversal strategy employed by the disassembly algorithm [3].

5.2 Limitations

DExtor is partially affected by branch function obfuscation, which obscures the control flow of an executable so that disassembly cannot proceed. Currently, there is no general solution to this problem. When branch function obfuscation is present, DExtor is likely to produce fragmented code blocks, missing some of the original code. This does not impact detection unless the missed blocks contain large numbers of instructions.

DExtor is certainly limited by its processing speed. Currently, DExtor has a throughput of 42 KB/sec in a real environment. Such a low throughput is unacceptable for an intrusion detection system that must handle several gigabits per second. Fortunately, DExtor is intended to protect just one network service, which requires much less throughput.

Nevertheless, the throughput issue can be addressed using faster hardware and optimizing all the software components (disassembler, feature extractor and classifier). Also, certain incoming traffic can be excluded from analysis. For example, because exploit code is typically a few kilobytes in length, bulk inputs to the server with size greater than a few hundred kilobytes are unlikely to be exploit code. Both these solutions should increase DExtor's throughput sufficiently to enable it to operate effectively in real-time environments.

6. Conclusions

DExtor uses data mining very effectively to detect and block exploit code. Designed to operate at the application layer, DExtor is positioned between the server and its gateway or firewall. It is completely transparent to the service it protects, and can be deployed as a stand-alone component or coupled with a proxy server. DExtor is robust against most attack-side obfuscation techniques. Tests using large volumes of normal and attack traffic demonstrate that DExtor can detect exploit code with very high accuracy and negligible false alarm rates. Furthermore, DExtor is able to detect new types of exploits with high accuracy.

DExtor is also useful in forensic investigations, especially in determining whether a crash or compromise was caused by a code-carrying exploit. In addition, it can assist in identifying the source of the exploit.

Acknowledgements

This research was supported by the Air Force Office of Scientific Research under Contract No. FA9550-06-1-0045 and by the National Science Foundation under CAREER Grant No. CNS-0643906.

References

[1] R. Chinchani and E. Berg, A fast static analysis approach to detect exploit code inside network flows, *Proceedings of the Eighth International Symposium on Recent Advances in Intrusion Detection*, pp. 284–308, 2005.

[2] T. Detristan, T. Ulenspiegel, Y. Malcom and M. von Underduk, Polymorphic shellcode engine using spectrum analysis, *Phrack*, vol. 11(61), 2003.

[3] C. Kruegel, W. Robertson, F. Valeur and G. Vigna, Static disassembly of obfuscated binaries, *Proceedings of the Thirteenth USENIX Security Symposium*, pp. 255–270, 2004.

[4] Lawrence Berkeley National Laboratory, Bro intrusion detection system, Berkeley, California (bro-ids.org), 2007.

[5] M. Locasto, K. Wang, A. Keromytis and S. Stolfo, FLIPS: Hybrid adaptive intrusion prevention, *Proceedings of the Eighth International Symposium on Recent Advances in Intrusion Detection*, pp. 82–101, 2005.

[6] S. Macaulay, ADMmutate: Polymorphic shellcode engine (www.ktwo.ca/security.html), 2007.

[7] Metasploit, The Metasploit Project (www.metasploit.com).

[8] Snort.org, Snort (www.snort.org).

[9] T. Toth and C. Kruegel, Accurate buffer overflow detection via abstract payload execution, *Proceedings of the Fifth International Symposium on Recent Advances in Intrusion Detection*, pp. 274–291, 2002.

[10] K. Wang, G. Cretu and S. Stolfo, Anomalous payload-based network intrusion detection and signature generation, *Proceedings of the Eighth International Symposium on Recent Advances in Intrusion Detection*, pp. 227–246, 2005.

[11] K. Wang and S. Stolfo, Anomalous payload-based network intrusion detection, *Proceedings of the Seventh International Symposium on Recent Advances in Intrusion Detection*, pp. 203–222, 2004.

[12] X. Wang, C. Pan, P. Liu and S. Zhu. SigFree: A signature-free buffer overflow attack blocker, *Proceedings of the Fifteenth USENIX Security Symposium*, pp. 225-240, 2006.

[13] Weka, Weka 3: Data mining software in Java, University of Waikato, Hamilton, New Zealand (www.cs.waikato.ac.nz/ml/weka).