Accurate Shellcode Recognition from Network Traffic Data using Artificial Neural Nets

Patrick Onotu[*], David Day and Marcos A Rodrigues
Sheffield Hallam University, Sheffield, UK
Email: b2040977@my.shu.ac.uk, {*D.Day, M.Rodrigues*}@shu.ac.uk

Abstract

This paper presents an approach to shellcode recognition directly from network traffic data using a multi-layer perceptron with back-propagation learning algorithm. Using raw network data composed of a mixture of shellcode, image files, and DLL-Dynamic Link Library files, our proposed design was able to classify the three types of data with high accuracy and high precision with neither false positives nor false negatives. The proposed method comprises simple and fast pre-processing of raw data of a fixed length for each network data package and yields perfect results with 100% accuracy for the three data types considered. The research is significant in the context of network security and intrusion detection systems. Work is under way for real time recognition and fine-tuning the differentiation between various shellcodes.

1 Introduction

Network Intrusion Detection Systems (NIDS) monitor, identify and alert to the presence of network traffic indicative of malicious or poor practices i.e, unethical hacking or system misconfiguration. The increasing popularity of cloud computing combined with the limitations of traditional host based protection systems have added to the popularity of NIDS implementation [Day and Zhao, 2011]. The most common method of NIDS operation is signature based, in which packets are examined for patterns which are associated with, or known to be hazardous. An important example of what constitutes hazardous is that of malicious shellcode. Malicious shellcode is a program written for the purposes of opening a shell on a victims machine and allowing a hacker unauthorised command line access. Due to the ability to further leverage such a breach, their successful execution is one of the principal objectives of a hacker, and they are frequently used as the payload for exploitations using system penetration tools such as Metasploit [Zhao and Ahn, 2013]. They are usually executed as a result of exploiting stack or heap-based buffer overflow vulnerabilities in system services which subvert the legitimate flow of code execution to that of the shellcode. Identifying shellcode as malicious traffic is particularly challenging with signature based NIDS due to false positives as shellcode patterns are often indistinguishable to that of some forms of benign traffic. For example, while working as a network security consultant for the Shop Direct Group (UK) using the network intrusion detection tools Sguil and Snort from the Debian based Linux distribution Security Onion, it was noticed that signatures designed to match shellcode frequently also matched other non shellcode binaries as well as jpg image files. The frequency of these false positives was such that the signatures themselves ultimately had to be disabled, rendering them useless. This experience with the false positive problem with shellcode and signature based systems is very common, Microsoft discuss this at length in their patent of methods to detect malicious shellcode with reduced false positives in memory [Shin *et al.*, 2013].

The research discussed in this paper explores a new mechanism to reduce false positives and negatives during

[*]Patrick Onotu is with Akanu Ibiam Federal Polytechnic, Ebonyi State, Nigeria. He worked on this research for his MSc in Automation and Control Engineering at Sheffield Hallam University.

detection of malicious shellcode using Artificial Neural Networks (ANNs). ANNs, biologically inspired by the human brain, are used as modelling mechanism for solving non-linear problems. As there has been significant established successes in using ANNs for pattern recognition e.g. image and speech, it could be argued that the potential to solve the false positive problem when detecting shellcode would be high. As such this paper will discuss our use of an artificial neural network using a supervised feedforward network with back propagation algorithm, to accurately identify shellcode traffic within custom generated traffic samples.

The remainder of this paper is organized as follows. In Section 2 we discuss, compare and critique both traditional and statistical approaches to network intrusion detection. Section 3 outlines the methodology and how samples were generated for testing. In Section 4 we describe the design and implementation of the ANN solution with simulation results. Finally in Section 5, we critically evaluate the results obtained with conclusions and recommendations for future work.

2 Previous Work

2.1 Comparison of signature and anomaly based NIDS

Mainstream NIDS can be categorised as either signature based or anomaly based. Signature based look for patterns in packets which are indicative of an attack. Whereas anomaly based NIDS look for a deviation away from a perceived normal baseline which has been generated over a period of time. Signature based systems can be seen to have two key issues, firstly they are often ineffective against unknown (zero day) attacks. This is due to these attack patterns being unknown and thus not being used for comparison. Secondly, it relies on a human interpretation of the root cause of the attack, sometimes this is erroneous if the context of the attack is not considered when creating the pattern matching rules. In these instances it takes only a small deviation in attack method to cause a false negative. Conversely anomaly based NIDS are not associated as strongly with these limitations and have been shown to be more effective against zero day attacks. However there are issues concerning the behavioural model generated, in the training phase, system performance and with the need for administrator intervention.

2.2 Traditional statistical methods

Decades of research in intrusion detection for computer system security revealed numerous possible approaches to the problem. This was initiated by the introduction of the concept that certain types of threats to the security of computer systems could be identified through the examination of information contained in the computer systems audit trail as seen in the work of [Anderson, 1980]. Three treats were identified as: external penetrations; internal penetrations; and misfeasors. It was later suggested that possible stealthy users could be detected by monitoring the functions that turn off the audit systems, or through a comparison of defined normal usage patterns of system resources with those levels which are currently observed. This method did not eventually prove effective against more intelligent attackers of the system.

From the work of [Denning, 1987], a statistical intrusion detection model was introduced and became a landmark for research in this area. The model was based on computer system keystroke dynamics and which now forms the core of most intrusion detection methodologies in use today. The concept involves the development of an electronic signature of a user based on their individual typing characteristics but modern-day research has revealed better approaches. The work of [Lunt, 1989] introduced a concept that involved anomaly detection, misuse detection and their combined effects and was regarded more effective than the existing methods until 1994. A more advanced methodology that involved pattern recognition and network monitoring was introduced in the work of [Mukherjee *et al.*, 1994]. The pattern recognition concept involved the recording of different penetration scenarios and coded into the system in the form of knowledge representation. The networking monitoring approach involves the use of various network measuring techniques and is a more advanced method of system intrusion detections currently in use. Its advantage over the other methods lies in its independence of network audit data.

2.3 The application of Artificial Neural Networks

The use of artificial neural networks in intrusion detection emerged from the works of [Fox *et al.*, 1990] and [Denault *et al.*, 1994]. Various networks were designed based on the Self-Organizing Feature Map (SOFM) to learn the characteristics of normal system activity and identify statistical variations from the norm that may be an indication of a virus or malicious activity. These methods were in use until [Ryan *et al.*, 1997] used the back-propagation algorithm to develop a system called Neural Network Intrusion Detector (NNID). The network was designed based also on user recognition and tested on a system of 10 users. The network was about 85 percent accurate in detecting malicious activities with about 15 percent false positive detection rate.

The work of [Cannady, 2000] introduced a method for intrusion detection based on the CMAC-Cerebellar Model Articulation Controller [Albus, 1975] which uses adaptive neural networks and the capability to learn new attacks rapidly through the use of a modified reinforced protected system. This was a modified reinforcement learning approach which resulted in an average detection error of 4% percent, compared to 15% in existing intrusion detection.

A remarkable achievement in intrusion detection was recorded when malicious shellcode detection approach was introduced in the work of [Toth and Kruegel, 2002]. Though this approach recorded good success concerning the identification of shellcode that is an indispensable part of an attack vector, it was nevertheless limited regarding the detection of a wider range of polymorphic shellcode exhibiting self-decrypting behaviour.

In [Aida *et al.*, 2010] a framework using a Multi-Layer Perceptron was proposed consisting of four phases: collection of intrusion detection parameters; processing of the filtered parameters; design of the system response manager; and the learning model. A total of 145,587 normal and attack events were collected, 70% of the data was used for training while the remaining 30% was used for testing. Over 90% performance on intrusion detection was recorded.

2.4 Malicious shellcode detection approach

This is regarded as a viable and potent approach for the general detection of code injection system of attacks. It focuses on the recognition of the shellcodes which are essentially part of an attack vector, this technique was initially referred to as abstract payload execution in the work of [Toth and Kruegel, 2002]. According to [Wang *et al.*, 2006] and [Wang *et al.*, 2008] initially this approach was implemented as an attempt to identify the presence of shellcode in network data traffic using static code analysis. However, methods that are based on static code analysis cannot handle malicious code effectively especially if they employ advanced polymorphic tactics such as self-modications. As a form of improvement to this approach, dynamic code analysis using emulation which cannot be hindered by such obfuscations and can detect even extensively polymorphic shellcodes was introduced in the work of [Polychronakis *et al.*, 2009]. This kind of actual payload execution has proven quite effective in practice and being used in network-level and host-level systems for the zero-day detection of both server-side and client-side code injection attacks [Egele *et al.*, 2009].

The above techniques have common limitation which is that they are limited to the detection of certain class of self-modifying shellcodes which are capable of exhibiting self-decrypting behaviour. According to [Polychronakis *et al.*, 2010], to evade signature based detectors shellcode encryption is very often used, that notwithstanding, attackers can still achieve good level of evasiveness without the utilization of any self-decrypting codes and thus rendering the above systems also ineffective. Besides code encryption, polymorphism can also be achieved by transforming the actual contents of the shellcode before initiating the attack – this technique is commonly referred to as metamorphism. Metamorphism is widely utilized by several virus programmers and thus can easily be used for shellcode mutation. The authors in [Polychronakis *et al.*, 2010] stated surprisingly that even plain or ordinary shellcodes, which do not mutate across different platforms, can also evade detection by existing payload execution methods. They also stated that in principle a plain shellcode is no different from any form of metamorphic shellcode, both neither contain a decryption property nor exhibit any self-modications or dynamic code generation. In effect, an attack that uses unknown plain shellcodes that are resistant to static analysis could evade existing detection systems, this was also previously stated in [Chung *et al.*, 2008].

In [Boxuan et al., 2010] a model was proposed for intrusion detection by detecting malicious shellcodes with virtual memory snapshots. From their proposed model, a malicious shellcode detection methodology was designed and implemented. In the method, snapshots of the target processs virtual memory are taken immediately before input data are consumed and fed into a lightweight Detection Before Consumption (DBC-based) malicious code detector. These snapshots were also used to instantiate a runtime environment that emulates the target processs input data consumption. This environment facilitates monitoring shellcodes behaviour. The snapshots helped to examine system calls invoked by executable input data and the parameters thereof as well as the processs execution ow to detect malicious shellcodes. This model suffered set back in its application as a result of significant rate of false positive detections.

3 Method

3.1 The proposed approach and its significance

In order to improve the above existing intrusion detection techniques, an approach involving the recognition of shellcode programming patterns in the midst of other network data with the help of artificial neural networks is proposed and demonstrated in this paper. The detection of shellcode data within network traffic containing a mixture of dynamic link libraries (DLL), jpg image files and shellcode has eluded research thus far with the standard outcome being large numbers of false positives as described above.

The stated aim of this research is to be able to accurately recognise those three classes of data (image, DLL and shellcode) using appropriately designed neural networks structures with minimum false positives. Thus, the significance of the proposed approach lies in its ability to accurately identify shellcode amidst other common data in computer data traffic with absolutely no case of false positive detections when applied using the offline detection-before-consumption methodology. Once the desired performance of the neural network is achieved, research can progress on further issues such as real-time detection and optimization to a wide variety of shellcodes.

3.2 Network traffic data collection

As previously alluded, it has been observed in both working practice and academic research that image files and benign binaries often cause false positives as shellcode using current technology signature based intrusion detection systems. The data collection involved the selection of 100 images, 160 variety of shellcodes and 140 Dynamic-Link Library (DLL) files which were pseudo-randomised and transmitted over a network between two machines, captured and saved as a *.pcap. The exact order in which data packets were transmitted was noted. At the receiving endpoint, packet payloads were extracted, structured and used to train, validate and test the neural networks.

As part of network design procedure, visual data analysis was performed to get a feel for the structure of the data. Figure 1 shows 2D plots of each file type as image, shellcode and DLL. For clarity, only the first 1000 elements of 3 files randomly selected from each class are plotted, padded with zeros where required. The plots were produced by simply converting each byte of data to its decimal equivalent. It can be noted that the three classes of files are rather distinct and, in the case of the images shown, each class can be uniquely identified by visual inspection. We plotted 65% of all data for each class and verified that the data display similar characteristics. These observed pattern variations are used as the core enhancer of the network performance in the classification task.

It is significant to note that a simple conversion from byte to decimal equivalent defines the required preprocessing of data; no other feature extraction, statistical measures, data transformation or corrections are necessary. The data can be used as is raising the possibility of a resulting neural network suitable for real-time applications with raw computer network payload input. Figure 2 depicts the pattern variations from a 3D view of 9 files where 1,000 points are plotted for each: on the horizontal axis, the first 3 files are images, the next 3 are shellcode and the last 3 are DLLs.

3.3 Neural network input and output data structure

All 400 collected samples (100 images, 160 shellcodes and 140 DLLs) were read into Matlab workspace as a column vector of its corresponding pure decimal values. The first 100 elements of each sample were selected and assigned to a variable `DATA`, forming a 400×100 matrix. 70% of the 400 columns were evenly selected and randomised as training data while the remaining 30 percent were retained for testing the generalising power of the network after training. The selection of data into training set and test set was performed using the Matlab code below:

```
A=DATA(:,1:10:end); B=DATA(:,2:10:end);
C=DATA(:,3:10:end); D=DATA(:,4:10:end);
E=DATA(:,5:10:end); F=DATA(:,6:10:end);
G=DATA(:,7:10:end); H=DATA(:,8:10:end);
I=DATA(:,9:10:end); J=DATA(:,10:10:end);
P = [A B C D E F G]; %280 training patterns
TP = [H I J];        %120 test patterns
```

This means that for all groups `A--J` the following ground truth for output classes apply: Image: vectors 1–10 Shellcode: vectors 11–26 DLL: vectors 27–40. The expected neural network outputs were structured as shown below in Table 1.

Table 1: Network output structure

Output Nodes	Class 1 (Image)	Class 2 (Shellcode)	Class 3 (DLL)
Node 1	1	0	0
Node 2	0	1	0
Node 3	0	0	1
Ground Truth Vectors in `A--J`	1–10	11–26	27–40

The expected outputs (targets) were designed and selected evenly and correspondingly with the patterns using the Matlab code below:

```
%Only one node go high per class:
T1=[1;0;0]; T2=[0;1;0]; T3=[0;0;1];
Target=[repmat(T1,1,100),repmat(T2,1,160)
,       repmat(T3,1,140)];
%Define targets for A--J:
AA=Target(:,1:10:end); BB=Target(:,2:10:end);
CC=Target(:,3:10:end); DD=Target(:,4:10:end);
EE=Target(:,5:10:end); FF=Target(:,6:10:end);
GG=Target(:,7:10:end); HH=Target(:,8:10:end);
II=Target(:,9:10:end); JJ=Target(:,10:10:end);
T=[AA BB CC DD EE FF GG]; %Training target
TT=[HH II JJ]; %Test target for statistical purposes
```

4 NETWORK DESIGN, TRAINING AND SIMULATION RESULTS

4.1 Network design

To design or create a feed-forward back-propagation network in Matlab, the function used is `newff` whose syntax is `net = newff(P,T,S,TFi,BTF)` with input parameters as follows:

- `P` is a matrix `RxQ` where `R` is the number of inputs and `Q` equals 2, representing a 2-element row vector of the minimum and maximum values in the inputs. For example, for `Q = [0 255]`, `P=[0 255;0 255;0 255]` which is a 3×2 matrix describing the number of network inputs. `Q` is calculated by the `minmax` built-in function;
- `T` is a matrix of `KxN` where `K` is a column vector describing the number of output nodes and `N` is the number of patterns. For a successful design, every node must switch values between high 1 and low 0 for classification. Therefore, for input patterns with 3 nodes (representing shellcode, image, and DLL) class 1 is represented with a target `T1=[1;0;0]`, class 2 with `T2=[0;1;0]` and class 3 with `T3=[0;0;1]`;
- `S` is the size of the hidden layers (the output layer size is determined from `T`);
- `TFi` is the transfer function of $\mathtt{i}^{\mathtt{th}}$ hidden layer. The transfer functions `tansig` was used for a single hidden layer network while `tansig` and `logsig` were used for a 2 hidden layer network. For the output layer, `purelin` is used; and
- `BTF` is the back-propagation network training function, both `trainlm` and `trainscg` were used.

The definitions of the functions used in the design are as follows. The log-sigmoid (`logsig`) transfer function with outputs between $\{0, 1\}$ as the network input goes from negative infinity to positive infinity:

$$f(x) = (1 + \exp^{-\beta x})^{-1} \qquad (1)$$

The hyperbolic tangent sigmoid (`tansig`) transfer function with outputs between $\{-1, 1\}$ as the network input goes from negative infinity to positive infinity:

$$f(x) = 2(1 + \exp^{-2x})^{-1} - 1 \qquad (2)$$

And the linear (`purelinear`) transfer function in which output neurons can take any value and are not limited to a small range:

$$f(x) = mx + b \qquad (3)$$

The training algorithms used were the Reduced Memory Levenberg-Marquardt (`trainlm`) which requires the storage of some large matrices for training, but in general the algorithm will have the fastest convergence. Furthermore, `trainlm` also yields the lower mean square errors than most learning algorithms. An alternative algorithm was also tested namely the `trainscg` or scaled conjugate gradient algorithm. The algorithm performs well on a large variety of problems with reduced memory requirements especially on networks with large number of weights.

A large number of possible designs can be realised. For instance, using a network design with the input vector fixed to 100, one hidden layer of 10 nodes and one output layer with 3 nodes, we can permute the transfer functions and learning algorithms as follows:

```
NET1=newff(repmat(minmax,100,1), [10 3],{tansig 'purelin'}, trainlm);
NET2=newff(repmat(minmax,100,1), [10 3],{logsig 'purelin'}, trainlm);
NET3=newff(repmat(minmax,100,1), [10 3],{tansig 'purelin'}, trainscg);
NET4=newff(repmat(minmax,100,1), [10 3],{logsig 'purelin'}, trainscg);
```

By the same token, alternative designs can be obtained by changing the size of input vector, number of hidden layers, and the number of nodes in each layer.

4.2 Network training and testing

The network configurations above were trained using the Matlab function `train` as: `NET = train(NET,P,T);` where `P` and `T` are the structured patterns and targets respectively. A neural network can be trained several times in order to improve its performance. After training, if the network has not converged, it can be trained again carrying on from the last weight values. It is possible also to start training afresh, in that case the (`init`) initialization command must be executed before each new `train` command. The (`init`) command ensures that the network is initialised with new connection weights and biases to be updated accordingly as training progresses. As a result, the network performance consistently varies for every training command executed. These variations do not necessarily translate to better performance and so at the end of every training, it is required to evaluate performance and a decision be made on possible parameter alteration towards performance improvement.

The networks were simulated with unseen data using the function `sim` with the corresponding reserved test data as `Y=sim(NET,PT)` where `Y` is the network outputs and `PT` the unseen patterns. The number unseen patterns clearly classified was used to determine the network performance. Out of the 120 test vectors reserved, some sample results are illustrated in Tables 2, 3, and 4 for network NET4 above where results for image are highlighted in green and shellcode in red:

From Table 2 columns 1, 5 and 8 are clearly classified as images (expected result as $[1\ 0\ 0]^T$) where the quoted results of 0.99 are interpreted as 1 while 0.01 as zero; columns 2, 3, 6 and 9 classified as shellcodes (expected $[0\ 1\ 0]^T$); columns 4, 7 and 10 as DLLs (expected $[0\ 0\ 1]^T$). All results match their respective set targets. From Table 3 columns 1 and 4 are classified as images (expected $[1\ 0\ 0]^T$); columns 2, 5, 8 and 9 are classified as shellcodes (expected $[0\ 1\ 0]^T$); columns 3, 6, 7 and 10 as DLLs (expected $[0\ 0\ 1]^T$). These results also match their set targets. From Table 4 columns 1, 4 and 7 are classified as images (expected $[1\ 0\ 0]^T$); columns 2, 5 and 8 are classified as shellcodes (expected $[0\ 1\ 0]^T$); columns 3, 6, 9 and 10 as DLLs (expected $[0\ 0\ 1]^T$). These also match their set targets for each category. It is important to stress here the actual level of confidence in the results, as error is within 0.01 which means that outputs equal or greater

Table 2: 1st Sample Test Data as `TP(:,1:13:end)` where green: image, red: shellcode, black: DLL

Output Nodes	Sample results for 10 test vectors									
Node 1	0.99	0.00	0.00	0.00	0.99	0.00	0.01	0.99	0.00	0.01
Node 2	0.00	0.99	0.99	0.00	0.01	0.99	0.00	0.01	0.99	0.00
Node 3	0.00	0.00	0.00	1.00	0.00	0.00	0.99	0.00	0.00	0.99

Table 3: 2nd Sample Test Data as `TP(:,2:13:end)` where green: image, red: shellcode, black: DLL

Output Nodes	Sample results for 10 test vectors									
Node 1	0.99	0.01	0.01	0.99	0.01	0.01	0.01	0.01	0.01	0.01
Node 2	0.01	0.99	0.00	0.01	0.99	0.00	0.00	0.99	0.99	0.00
Node 3	0.00	0.00	0.99	0.00	0.00	0.99	1.00	0.00	0.00	0.99

than 0.99 are rounded off to 1 and equal or smaller than 0.01 are rounded off to zero.

Table 5 summarises the obtained classification results with high accuracy and high precision as one of images, shellcodes and DLLs without any false positives or false negatives.

5 Discussion and Conclusions

This paper represents a step improvement in shellcode recognition from raw network traffic. The approach to convert bytes to their decimal equivalent independent of their true data type constitutes a simple pre-processing scheme. Also, by selecting only the first 100 samples from each datapacket from data streams leads to a design that is highly accurate with high precision. A number of designs were tested and the most appropriate design was a back propagation network with Levenberg-Marquardt learning algorithm applied to a network with 100 input nodes, a hidden layer of 10 nodes and an output layer with 3 nodes with respective transfer functions of logsig and purelin.

Although this approach holds high promise in intrusion detection applications, it is important to stress a few limitations of the current design for real-time applications as follows:

- Raw network traffic data need to be structured into predetermined block size to fit the neural network input data size: neural networks are trained with specific input data sizes and they cannot be simulated with data size different from the network input design. This has been addressed through fixed size sampling of data packets padded with zeros where required;
- The developed neural network has not been designed for online intrusion detection tasks: the design has not been optimised for integration with live network data traffic as the purpose is to demonstrate its ability to recognise shellcode that is randomly present in network data;
- It requires the collection of computer network data using data traffic capture for offline analysis: due to the current inability of the trained network to be integrated with a live streaming network traffic, malicious detection in this reported research is achieved through neural network simulation with collected or captured data;
- The approach does not differentiate good shellcodes from the bad (malicious) ones; all shellcodes are flagged in the same way in the reported studies.

Due to the severity of the attack, detecting unauthorized shell access is the one of the principal goals of network intrusion detection, and obtaining shell access the primary objective of an attacker. Industry experience working on network traffic analysis with The Shop Direct Group (UK) Ltd, while processing terabytes of network traffic per minute, has shown that dynamic link libraries (DLLs)

Table 4: 3rd Sample Test Data as `TP(:,3:13:end)` where green: image, red: shellcode, black: DLL

Output Nodes	Sample results for 10 test vectors									
Node 1	0.99	0.01	0.01	0.99	0.01	0.01	0.99	0.01	0.01	0.01
Node 2	0.01	0.99	0.00	0.01	0.99	0.00	0.01	0.99	0.00	0.00
Node 3	0.00	0.00	1.00	0.00	0.00	0.99	0.00	0.00	1.00	1.00

Table 5: Results Summary

Total Data Collected	Training Patterns	Test Patterns	Classified Test Patterns	Percentage Accuracy
400	280	120	120	100

which are often downloaded as part of system upgrades, along with image files are the core causes of false positives with traditional shellcode signatures. Removing the false positive problem with shellcode detection is one of the primary objectives in the research area and, prior to this work, being able to do this both effectively and economically has largely proved elusive. Thus, being able to classify shellcode with the level of accuracy and prevision discussed here offers a significant move forward in resolving the issues with shellcode detection. Current and future work include re-training and testing the network ability to recognise the various types of shellcode and thus, be able to pinpoint malicious code. We plan even more extensive and vigorous false positive testing against very large and sector diverse benign traffic sets. Also, we are investigating a design in which our trained network can be actually used and benchmarked for real time applications using streamed network traffic. Success in these areas would lead to further investigations into improving the hardware to allow for scalability with very high speed networks, for example use of Field Programmable Gateway Arrays and Application Specific Integrated Circuits. We also envisage investigations into using the same methods to detect other elements indicative of an attempted system breach e.g. heap and stack based buffer overflow attacks.

References

[Aida *et al.*, 2010] AIDA O.A., AHMED S. and TAMER R. (2010). ”Multilayer Perceptrons Networks for an Intelligent Adaptive Intrusion Detection System”. *International Journal of Computer Science and Network Security*. Vol.10, No.2.

[Albus, 1975] ALBUS, J.S. (1975). ”A New Approach to Manipulator Control: the Cerebellar Model Articulation Controller (CMAC)”. In: *Trans. ASME, Series G. Journal of Dynamic Systems, Measurement and Control*, Vol. 97, pp. 220-233.

[Anderson, 1980] ANDERSON J.P. (1980). ”Computer Security Threat Monitoring and Surveillance”. *Annual Technical Report*. Fort Washington, Pennsylvania. J.P. Anderson Company.

[Boxuan et al., 2010] BOXUAN G., XIAOLE B., ZHIMIN Y., ADAM C.C. and DONG X. (2010). ”Malicious Shellcode Detection with Virtual Memory Snapshots”. *In: IEEE INFOCOM Conference*. San Diego, C.A., USA.

[Cannady, 2000] CANNADY J. (2000). ”Next Generation Intrusion Detection: Autonomous Reinforcement Learning of Network Attacks”. *In: 23rd National Information Systems Security Conference*. Baltimore, USA.

[Chung *et al.*, 2008] CHUNG S.P. and MOK A.K. (2008). ”Swarm Attacks against Network-Level Emulation and Analysis”. *In: 11th International Symposium on Recent Advances in Intrusion Detection*. Massachusetts, USA.

[Day and Zhao, 2011] DAY, D. and ZHAO, Z. (2011). ”Protecting Against Address Space Layout Randomisation (ASLR) Compromises and Return-to-Libc Attacks Using Network Intrusion Detection Systems", *International Journal of Automation and Computing* Vol. 8 no. 4, 472-83. December 6th 2011.

[Denault *et al.*, 1994] DENAULT M., GRITZALIS D., KARAGIANNIS D. and SPIRAKIS P. (1994). "Intrusion Detection: Approach and Performance Issues in Computers and Security". *The Securenet System*. Vol.13, No. 6, pp. 495-507.

[Denning, 1987] DENNING D. (1987). "An Intrusion-Detection Model". *IEEE Transactions on Software Engineering*. Vol. SE-13, No. 2.

[Egele *et al.*, 2009] EGELE M., WURZINGER P., KRUEGEL C. and KIRDA E. (2009). "Defending Browsers against Drive-By Downloads: Mitigating Heap-Spraying Code Injection Attacks". *In: 6th international conference on Detection of Intrusions and Malware, & Vulnerability Assessment*. Como, Italy.

[Fox *et al.*, 1990] FOX K.L., HENNING R.R. and REED J.H. (1990). "A Neural Network Approach Towards Intrusion Detection". *In: 13th National Computer Security Conference*. Washington D.C., USA.

[Lunt, 1989] LUNT T.F. (1989). "Real-Time Intrusion Detection". *Proceedings of IEEE COMPCON.*

[Mukherjee *et al.*, 1994] MUKHERJEE B., HEBERLEIN L.T. and LEVITT K.N. (1994). "Network Intrusion Detection". *IEEE Network*. Pp. 26–41.

[Polychronakis *et al.*, 2009] POLYCHRONAKIS M., ANAGNOSTAKIS K.G. AND MARKATOS E.P. (2009). "An Empirical Study of Real-World Polymorphic Code Injection Attacks". *In: 2nd USENIX Workshop on Large-scale Exploits and Emergent Threats*. Boston M.A., USA.

[Polychronakis *et al.*, 2010] POLYCHRONAKIS M., ANAGNOSTAKIS K.G. AND MARKATOS E.P. (2010). "Comprehensive Shellcode Detection Using Runtime Hauristics". *In: Proceedings of the Annual Computer Security Applications Conference*. Austin Texas, USA.

[Ryan *et al.*, 1997] RYAN J., LIN M. and MIIKKULAINEN R. (1997). "Intrusion Detection with Neural Networks: AI Approaches to Fraud Detection and Risk Management". *Papers from the 1997 AAAI Workshop* (Providence, Rhode Island). Pp. 72–79. Menlo Park, CA: AAAI.

[Shin *et al.*, 2013] SHIN J., LAMBERT J.J. AND LACKEY J. "Evaluating Shellcode Findings." U.S. Patent 8,413,246 issued date April 2, 2013.

[Toth and Kruegel, 2002] TOTH T. and KRUEGEL C. (2002). "Accurate Buffer Overow Detection via Abstract Payload Execution". *In: 5th Symposium on Recent Advances in Intrusion Detection*. Zurich, Switzerland.

[Wang *et al.*, 2008] WANG X., JHI Y.C., ZHU S. and LIU. P. (2008). "Exploit Code Detection via Static Taint and Initialization Analyses". *In: Annual Computer Security Applications Conference (ACSAC)*. California, USA.

[Wang *et al.*, 2006] WANG X., PAN C.C., LIU P. and ZHU S. (2006). "Sigfree: A Signature-Free Buffer Overow Attack Blocker". *In: USENIX Security Symposium*. Vancouver B.C., Canada.

[Zhao and Ahn, 2013] ZHAO Z. and AHN G. "Using Instruction Sequence Abstraction for Shellcode Detection and Attribution" *In: IEEE conference on Communications and Network Security*. National Harbour, MD.

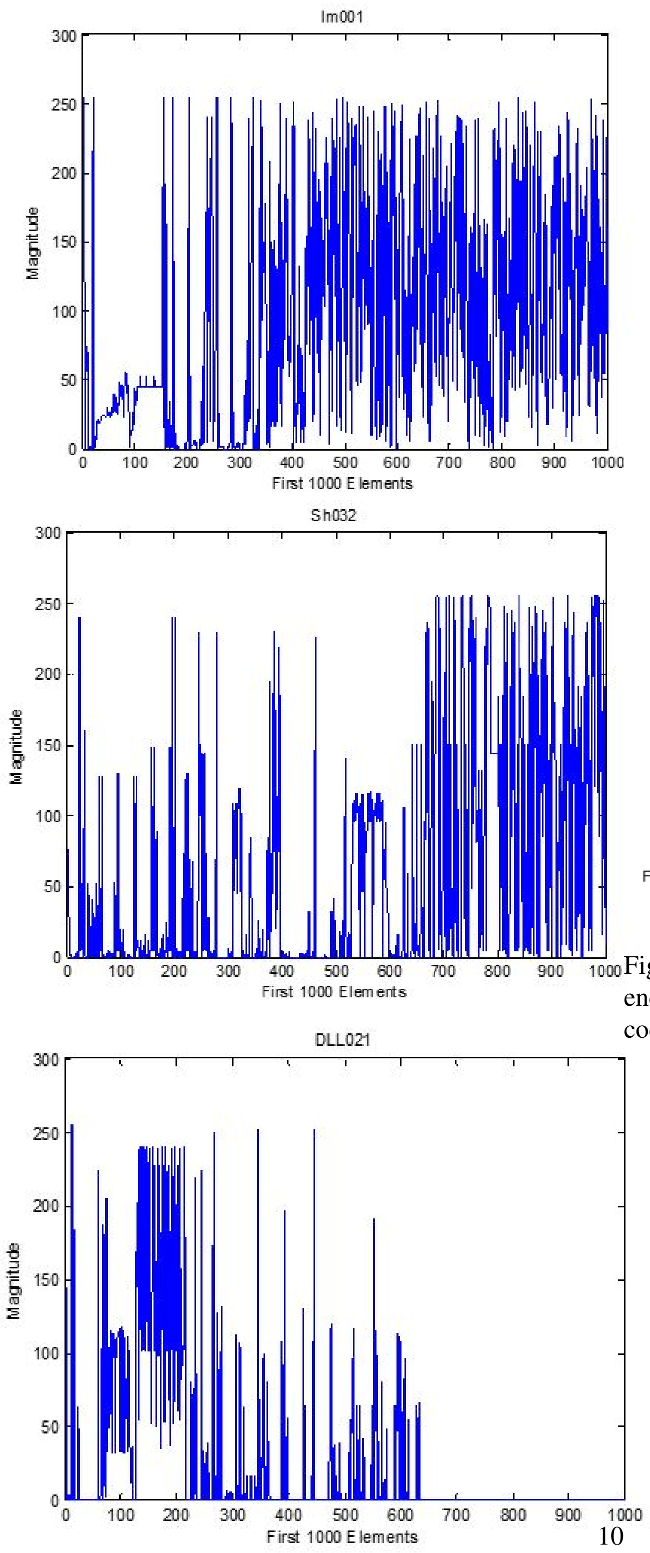

Figure 1: Example plot of 1,000 data points of collected data (padded with zeros where required). Top: a jpeg image, middle: a shellcode, bottom: a DLL.

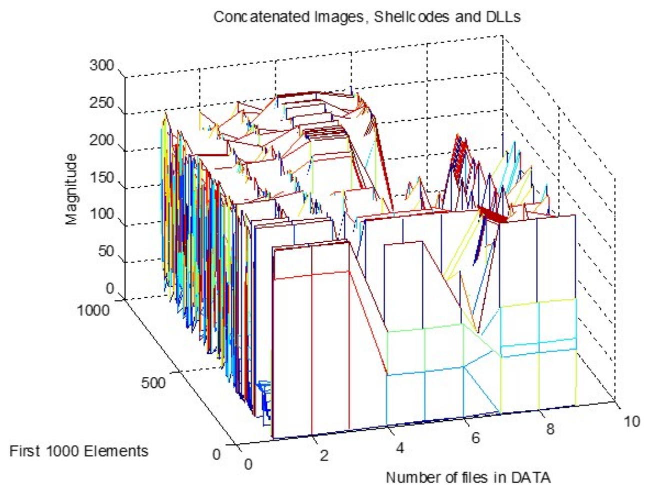

Figure 2: 3D view of 1,000 data points showing differences between classes (files 1–3: image; files 4–6: shellcode; files 7–9: DLL)