MALWARE DETECTION BASED ON STRUCTURAL AND BEHAVIOURAL FEATURES OF API CALLS

Mamoun Alazab[1], Robert Layton[2], Sitalakshmi Venkataraman[1], Paul Watters[1]

Internet Commerce Security Laboratory ICSL
Graduate School of ITMS
University of Ballarat

m.alazab@ballarat.edu.au[1]
s.venkatraman@ballarat.edu.au[1]
p.watters@ballarat.edu.au[1]
r.layton@icsl.ballarat.edu.au[2]

Abstract

In this paper, we propose a five-step approach to detect obfuscated malware by investigating the structural and behavioural features of API calls. We have developed a fully automated system to disassemble and extract API call features effectively from executables. Using n-gram statistical analysis of binary content, we are able to classify if an executable file is malicious or benign. Our experimental results with a dataset of 242 malwares and 72 benign files have shown a promising accuracy of 96.5% for the unigram model. We also provide a preliminary analysis by our approach using support vector machine (SVM) and by varying n-values from 1 to 5, we have analysed the performance that include accuracy, false positives and false negatives. By applying SVM, we propose to train the classifier and derive an optimum n-gram model for detecting both known and unknown malware efficiently.

Keywords: Code obfuscation, Feature extraction, Malware, n-gram, SVM.

INTRODUCTION

Malicious software (Malware) (McGraw & Morrisett, 2000) affects the secrecy and integrity of data as well as the control flow and functionality of a computer system (Alazab et al., 2009a; Sulaiman et al., 2005). Recent attacks using obfuscated malicious codes (previously unknown malware) have resulted in disruption of services leading towards huge financial and legal implications (Alazab et al., 2009a; Keizer, 2009) (Alazab et al., 2009b; Sharif et al., 2008). Therefore, researchers and anti-malware vendors are faced with the challenge of how to detect such zero day attacks (unknown malwares), which is also a major concern for various computer user groups, including home, business and even government users. Literature studies (Christodorescu and Jha, 2003; 2004; Vinod et al., 2009) on malware detection have shown that there is no single technique that could detect all types of malware. However, there are two techniques commonly used for malware detection, signature-based detection and anomaly-based detection. Anti-malware engines use signatures or 'byte sequence' to detect known malware. These signatures are created by disassembling the file and selecting some pieces of unique code. Hence, signature-based detection is very effective for known malware but the major drawback is the inability to detect new, unknown malicious code that result in zero day attacks (Sharif et al., 2008). On the other hand, anomaly-based detection, which is a technique that uses the knowledge of what constitutes normal behaviour to decide the maliciousness of a program code, has the key advantage and ability to detect zero day attacks. Current anomaly-based techniques use heuristics approach of detection that is inefficient and usually resulting in false positives (Vinod et al., 2009). In this paper, we propose a novel approach to extract the structural and behavioural features from program codes to detect both known and unknown malware.

Windows API calling sequence reflects the behaviour of a particular piece of code (Wang et al., 2009). API enables the programs to exploit the power of the operating system and the malware authors are taking this advantage to make use of the API calls as a vehicle to perform malicious actions. However, Malware writers make use of Application Program Interface (API) calls as a vehicle to inflict systems and to evade from anti-virus (AV) scanners through code obfuscation. Our novel technique of extracting the structural and behavioural features of API calls with the aid of statistical n-gram analysis has resulted in effective malware detection. An n-gram is an n-contiguous sequence the main used is for pattern recognition. It has been used and applied successfully in many areas of computer science applications such as Computer Speech Recognition (Lee and Kawahara, 2009), Language Identification (Choong et al., 2009), Spelling Correction (Bergsma et al., 2009),

Optical Character Recognition (OCR) (Schambach, 2009), Authorship Analysis (Layton et al., 2009) etc. An n-gram method of feature extraction and analysis is quite thorough but time consuming as the size of n increases. To overcome this constraint, we propose an intelligent machine learning technique of feature recognition to train a classifier for identifying malicious code. Literature studies indicate that a predominantly used machine learning technique called, Support Vector Machines (SVMs) for pattern recognition has been applied successfully to classify text, handwritings (Bergsma et al., 2009; Choong et al., 2009) and the like. Since malwares also exhibit the behaviour patterns in the form of a fileprint or feature, this work applies SVM for effectively classifying the program code as either malicious or benign.

CURRENT MALWARE DETECTORS

Malware detectors are used to scan a computer system to identify malware, with the main purpose of preventing it from adversely affecting the system. The current malware detection methods usually rely on existing malware signatures with limited heuristics and are unable to detect those malware that can hide itself during the scanning process (Venkatraman, 2009) and those malware that apply spohisticated obfuscation techniques. An anti-virus (AV) engine must perform 3 main tasks to protect computers: Scanning, Detection, Removal (Microsoft, 2007). A Malware detector D is defined as a function whose determine the exactable program (p) which program is malicious or benign $D: P \rightarrow \{malicious, benign\}$. Modern and traditional anti-malwares scan the programs (p) in a system for a byte sequence or malware signature (s) which it stored in the database engine. If the signature is found in the program (p), it will be identified as a malware, otherwise it is declared as benign, and this is represented in the equation below.

$$D(P) = \begin{cases} Malware \;\; if \;\; s \in p \\ Benign \;\; otherwise \end{cases} \quad (1)$$

The malware signature is a byte sequence that uniquely identifies a specific malware (Vinod et al., 2009). Typically, a malware detector uses the malware signature to identify the malware like a fingerprint. Most AV engines are supplied with a database containing information of existing malware to identify maliciousness, by looking for code signatures or byte sequences while scanning the system. A malware detector scans the system in various locations for characteristic byte sequences or signature (s) that match with the one in the database and declares existence of malware blocking its access to the system. The signature matching process is called signature-based detection and most traditional AV engines use this method. It is a very efficient and effective method to detect known malware. But, the major drawback is the inability to detect new or unknown malicious code and zero day attacks. Therefore, updating the detection engine or AV software daily with latest malware signatures is essential so as to protect the computer system against all known malware. Hence, the more malware signatures (s) are fed into the AV engine, the more effective it is in detecting latest known malwares.

The new threat for computers is that the malware writers are recycling existence malware like w32.Parite (Microsoft, 2010) with different signatures (‘byte sequence') by using obfuscation techniques such as packing, polymorphic transformations and metamorphic obfuscations, instead of creating an entirely new malware. This means that the AV scanners will not be able to detect these new malware due to the non-existence of their fingerprints in the signature database. Hence, there is a need to capture the behaviour of malware based on anomalies or behavioural patterns exhibited by such hidden malware, which is the main focus of this research work.

The results of the following recent studies have been the prime motivation for this research:

1) malware authors are able to easily fool the detection engine by applying obfuscation techniques on known malwares (Sharif et al., 2008),
2) identifying benign files as malware is becoming very high (Keizer, 2009) (Yegneswaran et al., 2005) (false positive),
3) failing to detect obfuscated malware is high (false negative) (Paul, 2008),
4) the current detection rate is decreasing (Symantec, 1997), and
5) current malware detectors are unable to detect zero day attacks (McGraw & Morrisett, 2000).

Code obfuscation is a challenge for digital forensic examiners (Alazab et al., 2009a) with the limitations of signature based detection (Christodorescu and Jha, 2003). As a first step to address these issues, in this paper, we propose a five-step approach of anomaly based detection that captures and analyses the structural and behavioural features of application programming interface (API) calls from program codes or executables using n-gram and SVM to detect and classify even unknown malware. In our previous work (Alazab et al., 2010), we have automated the entire process of effectively extracting the behaviour features of application programming interface (API) calls of the core of Windows operating system. The fully-automated system processes both packed and unpacked portable execution (PE) files and performs reverse obfuscation. This paper is an enhancement to the previous work. We further extract the n-gram distributions of all the API call features from the malicious and benign executables and apply SVM for machine learning. We first extract the most frequently occurring n-grams in each file and collate that list into an overall list for the entire dataset. and then collect the n-gram distribution for each executable for each n-gram in the larger list. We apply principle component analysis (PCA) approach to the result to account for 95% of the variation and for an effective feature reduction process. With the extracted features, our main goal of detecting unknown malware is now possible if our approach caters to all possible code obfuscations techniques that could be adopted by the malware authors. The next section describes the commonly used code obfuscation techniques that are addressed in this research work.

CODE OBFUSCATION TECHNIQUES

In order to evade detection by AV engines, malware authors are applying obfuscating techniques (Alazab, 2009) such as packing (Guo et al., 2008), polymorphism (Stepan, 2005) and metamorphism (Bruschi et al., 2006) in order to transform existing malware. Metamorphic and polymorphic malware transformed malcode into a new code without affecting the original functionality or purpose so that the AV engine's scanning process skips the detection of the signature.

Malware authors are continually developing new techniques for creating and applying obfuscation techniques $T(P)$ on a malware program (p) to produce an obfuscated program (p'), thereby making it very difficult to reserve engineer and decipher the signature successfully, even though the two programs, p and p' are having the same functionality and exhibit the same affect. On the other hand, since p and p' have different byte sequence, AV engines and reverse engineers are applying deobfuscation techniques $D(p')$ on the obfuscated program (p') in order to analyse the malware and to detect the malware as shown in figure 1.

Packers are commonly used today for code obfuscation or compression. Packers are software programs that could be used to compress and encrypt the PE in secondary memory and to restore the original executable image when loaded into main memory (RAM). Malware authors do not need to change several lines of code to change the malware signature mainly because, changing any byte sequence in the PE results in a new different byte sequence in the newly produced packed PE.

Polymorphic malware uses encryption and data appending/ data pre-pending in order to change the body of the malware, and further, it changes decryption routines from infection to infection as long as the encryption keys change. This has led AV experts to develop different scanning techniques, from simple byte sequence matching to more complex techniques such as X-RAYING scanning (Perriot & Ferrie, 2004). In 2006 Symantec Internet Security Threat Report stated the detecting polymorphic Malware such as w32.Polip and w32.Detnat is much more difficult and complex than the other type of Malware (Turner, 2006).

Metamorphic malware changes the code itself without the need of using encryption. In general, there are four techniques commonly used for metamorphic obfuscation. These are, i) *Dead-code Insertion* which is mean do nothing such as a sequence of **NOP**s (No Operation Performed), ii) *Code Transposition* that changing the instructions such as using **JMP**s instructions so that the order of instructions is different than the original one, iii) *Register Reassignment* such as replacing **push ebx** with **push eax** to exchange register names, and iv) *Instruction Substitution* which is replace the instructions in to different instructions with the same result some authors uses a database dictionary of equivalent instruction sequences to make it easier and faster.

In a nutshell, malware authors are continually developing such new techniques for creating malware that cannot be detected by AV engines, and their level of sophistication is continuing to grow. Through our experimental tests, we have found that all above techniques can be used to fool the current detection engines by obfuscating the malware signature. Hence, in order to cater to all the above mentioned obfuscation techniques, our research focuses on unpacking and extracting the behaviour of the malware through API call analysis rather than the typical "pattern matching" detection process that are evaded by obfuscations of the byte sequence through packing, metamorphic and polymorphic techniques. We identify the features of the extracted API calls in the

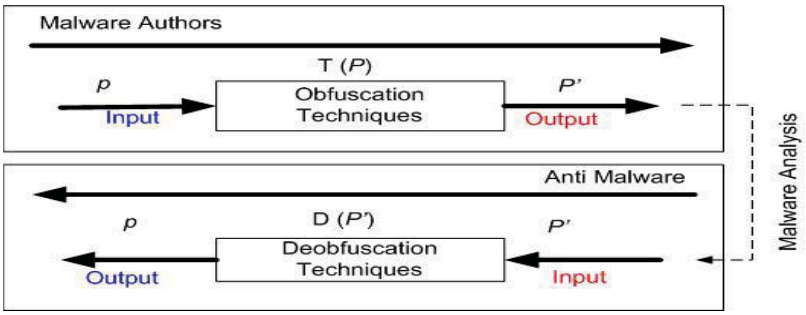

Figure 1 Obfuscation transformation

unpacked executable binary using the n-gram statistical analysis that is described in the next section.

N-GRAMS

An n-gram model, in simple terms, uses the statistical properties of n-grams for predicting the next item in a sequence. It is a subsequence of 'n' items from a given sequence. For this research, the items refer to the list of API calls within an executable file. An n-gram could be of different sizes: 'unigram' referred when the size is n=1; 'bigram' where the size is n= 2; size n=3 referred to "trigram; size (n−1) is termed "Markov model" and size n=4 or more is generally called 'n-gram'. Many disciplines have applied n-gram analysis as the model is efficient and successful in solving classification problems. In this paper, we apply n-grams to the problem of malware detection; by extracting the list of API calls contained within both packed the unpacked malware. A classifier is trained on the differences in n-gram distributions between malicious and benign executable files.

Some studies have analysed 1-gram and 2-gram of ASCII byte values (Stolfo et al.,2006) and computed the frequency and variance of each gram. They observed that applying 'unigram' analysis to Portable Document Format (PDF) files embedded with malcode are pretty effective in malware detection when compared to the COTS AV scanners. However, such studies are limited to document files types and do not have sufficient resolution to include all class of file types. As the number of n-grams is going to be very large, feature selection measures such as ASCII, UNICODE, API and others are being adopted to yield better results. For the obfuscated malware detection problem, this work applies n-gram on API call based features. We propose a five-step approach that results in an effective n-gram feature extraction from API call sequences for classifying executables as malicious or benign with the use of Support Vector Machines (SVM) as the machine learning classifier.

CONTRIBUTIONS

This paper's main contributions are four-fold as given below:

- We have developed a fully-automated system to unpack, de-obfuscate and reverse engineer the program codes and apply feature extraction techniques effectively.
- We have successfully used the system to extract behaviour features of API calls that relate to i) hooking of the system services, ii) creating or modifying files, iii) getting information from the file for changing information about the DLLs loaded by the malware (Alazab et al., 2010), and have extended the previous work done by the authors.
- We have applied the n-gram statistical model to obtain the distribution of the executables for n-values ranging from 1 to 5. We have measured the model based on factors such as accuracy, false positives and false negatives.
- Our proposed approach also includes the use of SVM to train the classifier for machine learning and robust identification of malicious code as against benign code.

PROPOSED APPROACH AND IMPLEMENTATION

Since majority of malware change their byte sequence or 'Signature' by applying obfuscation techniques to evade detection by virus scanners, we first unpack and de-obfuscate the executable using our fully-automated system (programmed in Python programming language). The system uses existing tools such as SQLite (2010) plug-in with IDA pro Dissasembler (2009) to generate eight tables of information, namely Blocks, Functions, Instructions, Names, Maps, Stacks, Segments, TargetBinaries. Each of these tables contains different information about the binary content. Our premise is that Malware files and benign files have different uses and distributions for their of API function calls. Hence, we extract behaviour features from n-gram distributions. For the analysis of the features, we have also considered the machine opcodes that include direct and indirect calls such as Jump and Call operations as well as the function types (e.g. import). Sample SQL command code snippets for feature extraction used the Python program are as shown below.

```
SELECT function_address,src_block_address,name
from Maps, Names"
Where (op='call' or op='jmp') and (type='import' or type='function')
```

As shown in figure 2, we initially extract the n-gram distribution from the dataset, train the SVM classifier with benign files and then test for malicious files for accurate identification of malware. We present the approach in the form of five steps given below:

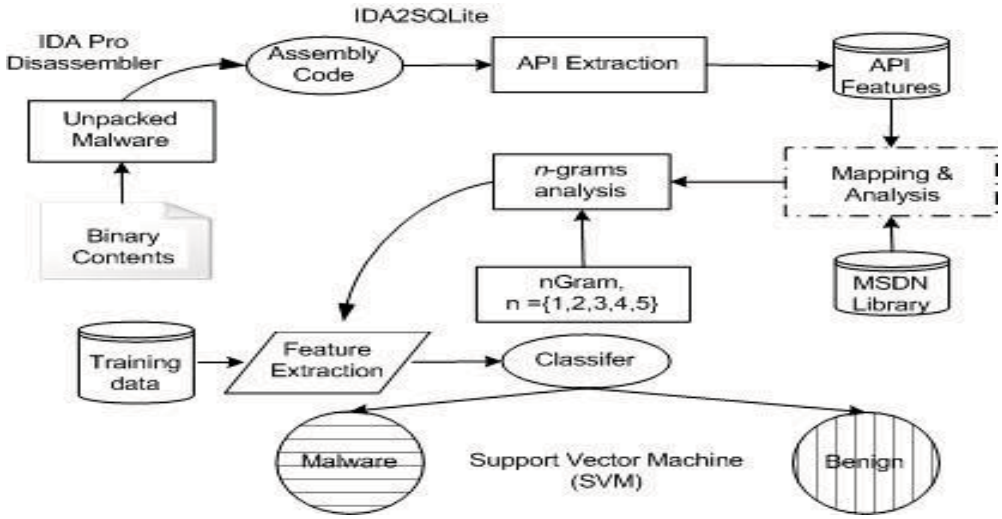

Figure 2 Fully-automated architecture to distribute the API function calls

Step 1: Unpack the malware and disassemble the binary executable to retrieve the assembly program.

For our implementation, we have used PEiD (2010) to unpack the binary executable. PEiD is commonly used for most common packers, cryptors, compilers and even signature-based packer detection in PE files. We then disassembled the code using Interactive Disassembler Pro (IDA Pro) as it automatically recognizes API calls for various compilers.

Step 2: Extract API calls and important machine-code features from the assembly program.

Our fully-automated system processes the outputs from Step 1 with the aid of SQLite and IDA pro to generate the database containing the application programming interface (API) calls (.idb) automatically from the entire dataset of malware and benign programs. API call features are extracted from the assembly code of the executables so that the generated information could be used for effective analysis.

Step 3: Map the API calls with MSDN library and analyse.

Using the downloaded Windows API from Microsoft Developer Network (MSDN) (2010) our automated system compares and matches API calls outputted from Step 2 with the look-up table of API libraries from MSDN.

For the analysis, we consider extracted features such as frequency of call, call sequence pattern and actions immediately preceding or after the call. We considered specific actions that lead to invalid memory references or undefined registers or invalid target jumps for refining the extracted API call features.

Step 4: Extract binary N-gram features.

To extract the n-gram distributions of all of the malicious and benign executables, we first count the frequency of each n-gram within the entire corpus. Once that has been completed, we reduce this list to the top 100 most frequent n-grams. The above procedure is replicated for n values between 1 and 5 inclusive.

Step 5: Train a classifier and build a model using support vector machine (SVM).

SVMs have performed well on traditional text classification tasks, and on executable files. The supervised learning SVM method is a reliable and popular technique for data classification (Cortes and Vapnik, 1995). SVM is considered easier to use than many machine learning approaches such as Neural Networks (NN). Hence, in this step, SVM classification technique is used to construct an *N*-dimensional hyperplane separates the dataset into two groups, namely, 'Malware' and 'Benign'. For addressing the zero day malware detection, we focus on the n-gram feature extraction from API sequences using SVM as the training algorithm. Initially, in this step, we separated the data into two: training and testing data sets. We then applied SVM to the training data with the goal to produce a model, which is then used to predict the target of the test data. In order to achieve a higher accuracy of the predictive model for generalisation, we have used the K-fold cross-validation approach for the test data. With the experimental data set, we have adopted a 10-fold cross-validation approach (Swets and Pickett, 1982), which is commonly used to estimate how well the trained SVM model is going to perform in the future.

For the experiment in this research work, we have used the LIBSVM tool (Chang and Lin, 2001). Both benign files and known malicious files are used to train the SVM classifier so that the model could be used to test for new obfuscated malware that evades detection from AV scanners. We performed the verification and validation of our proposed method for malware detection based on the following standard measures:

1. False Positives (**FP**): Number of wrongly identified benign code, when the model detects benign file as malware.
2. False Negatives (**FN**): Number of wrongly identified malicious code, when the model fails to detect the malware.

Among the four basic types of kernels used by SVM to map the training vectors to the N-dimensional space, we have applied the Radial Basic Function (RBF) kernel. This is because it can handle the nonlinear cases. We

$$K(x, y) = \exp\left(- \frac{|| x - y ||^2}{\sigma^2} \right) \tag{2}$$

have tested the classification performance based on $1/\sigma^2$ and C parameters from the equation given below, where $C > 0$ is the penalty parameter of error term.

EXPERIMENTAL RESULTS

In order to conduct an experimental investigation, we have applied our methodology described in Section VI to the dataset collected from honeypots, honeynet project and other sources between July 2009 and November 2009. We have used 314 executable files in total with 72 benign files and 242 malware infected files that have been uniquely named according to their MD5 value. From our experimental study, we observe that the overall

accuracy of our classifier decreases as n increases, as shown in table 1. The trend observed here could be due to the specific dataset that was used for the experimental testing. However, any generalisation of the observed trend could only be emphatically determined with larger and wider range of datasets. The initial experimental result of 96.5% accuracy for unigrams is still very promising as a benchmark for improvements in our future research work. Further to this, there are clear trends in both the false positive and false negative values with increasing values of n. While unigrams create the better n-gram models for the values shown in table 1, the high false negative rate indicates that there is still work to be done on improving this value.

LIMITATIONS OF THE STUDY AND FUTURE WORK

Our automated system makes use of existing unpacking tools, such as PolyUnpack (Royal et al., 2006), Renovo (Kang et al., 2007), OmniUnpack (Martignoni et al., 2007), Ether (Dinaburg et al., 2008), and Eureka (Sharif et al., 2008) that are still under research and development. If the existing tools are unable to unpack a packed malware that uses an unknown packing algorithm, this would pose a limitation for Step 1 of our automated system. However, our approach from Step 2 onwards would still work in this case by conducting a manual unpacking in Step 1. Another limitation of our system is that Step 3 is based on the latest updates of Microsoft with the MSDN library of API call list. We believe this is done up-to-date, as MSDN library forms the main reference for the mapping of the API calls in both malware and benign files.

Future work in this area includes techniques to increase the accuracy of the system. The FP rate increases while the FN rate decreases as n increases, indicating that it could be possible to use a boosting technique to apply a classifier model derived from a higher n value to first determine if a file appears to be malicious, then using a model derived from a lower n value so as to more accurately determine if this suspected file is in fact malicious.

Table 1
Experimental results from SVM classifier by applying n-grams

n-value	Accuracy	False Positives	False Negatives
1	96.50%	1.91%	1.56%
2	92.99%	6.36%	0.63%
3	88.22%	11.46%	0.03%
4	85.99%	14.01%	0.00%
5	85.03%	14.97%	0.00%

CONCLUSION

In this research work, the behavioural and structural features based on API calls are automatically extracted from the binary of a program. The extracted features are subjected to a statistical n-gram analysis to classify a program as either malicious or benign effectively with the aid of a supervised SVM machine learning technique. In a nutshell, this research provides four main contributions. The first and foremost contribution is, outlining of a methodology to extract behaviour features of API calls that relate to various malware behaviour such as i) hooking of the system services, ii) creating or modifying files, iii) getting information from the file for making changes about the DLLs loaded by the malware. The second contribution is, providing a statistical analysis of the API calls from the programs using n-gram model. In our model, the n-gram analyses the similarities and the distance of unknown malware with known behaviour so that obfuscated malware could be detected efficiently. The third contribution is, developing a fully-automated tool to unpack, de-obfuscate and reverse engineer the program codes without any need for manual inspection of assembly codes. The last contribution is, applying SVM machine learning to train the classifier for a robust identification of known as well as unknown malware.

ACKNOWLEDGMENT

This research was conducted at the Internet Commerce Security Laboratory and was funded by the State Government of Victoria, IBM, Westpac, the Australian Federal Police and the University of Ballarat.

REFERENCES

Alazab, M. (2009). “Investigation techniques for static analysis of NTFS file system images”, Annual Research Conference, Internet Security, University of Ballarat.

Alazab, M.; Venkatraman, S. & Watters, P. (2009a) “Effective digital forensic analysis of the NTFS disk image”, Ubiquitous Computing and Communication Journal, 4, 1.

Alazab, M., Venkatraman, S. & Watters, P. (2009b), "Digital Forensic Techniques for Static Analysis of NTFS Images", Proceedings of International Conference on Information Technology (ICIT2009). IEEE Computer Society, ISBN 9957-8583-0-0.

Alazab, M.; Venkataraman, S. & Watters, P. (2010). "Towards Understanding Malware Behaviour by the Extraction of API calls", Accepted to the IEEE 2nd Cybercrime and Trustworthy Computing Workshop (CTC-2010), 2010.

Bergsma, S.; Edmonton, A.; Lin, D.; View, M. & Goebel, R. (2009,). "Web-scale N-gram models for lexical disambiguation", Proceedings of the 21st international joint conference on Artificial intelligence, 1507-1512.

Bruschi, D.; Martignoni, L. & Monga, M. (2006). “Detecting self-mutating malware using control-flow graph matching”, Lecture Notes in Computer Science, Springer, 4064, 129.

Chang C.C. and Lin C.J. (2001). LIBSVM: a library for support vector machines. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

Christodorescu, M. & Jha, S. (2003). “Static analysis of executables to detect malicious patterns”, Proceedings of the 12th conference on USENIX Security Symposium, 12, 169-186.

Christodorescu M. and Jha S. (2004). “Testing malware detectors”, In Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004), Boston, MA, USA, ACM Press, 34–44.

Choong, C.; Mikami, Y.; Marasinghe, C. & Nandasara, S. (2009). "Optimizing n-gram Order of an n-gram Based Language Identification Algorithm for 68 Written Languages", International Journal on Advances in ICT for Emerging Regions (ICTer), 2, 21.

Cortes C. and Vapnik V. (1995). “Support-vector network”, Machine Learning, 20, 273-297.

Dinaburg, A.; Royal, P.; Sharif, M. & Lee, W. (2008). "Ether: Malware analysis via hardware virtualization extensions", Proceedings of the 15th ACM conference on Computer and communications security, 51-62.

Guo, F.; Ferrie, P. & Chiueh, T. (2008). "A study of the packer problem and its solutions", *Recent Advances in Intrusion Detection,* 98-115.

IDA Pro Dissasember (2009). DataRescue, An Advanced Interactive Multi-processor Disassembler, http://www.datarescue.com, October, 2009.

Kang, M.; Poosankam, P. & Yin, H. (2007). " Renovo: A hidden code extractor for packed executables", Workshop On Rapid Malcode WORM'07 Proceedings of the 2007 ACM workshop on Recurring malcode, 46 – 53.

Keizer, G. (2009). “Symantec false positive cripples thousands of Chinese PCs” http://www.computerworld.com/s/article/9019958/Symantec_false_positive_cripples_thousands_of_Chinese_PCs?intsrc=hm_list, August 2009.

Layton, R.; Brown, S. & Watters, P. (2009), " Using Differencing to Increase Distinctiveness for Phishing Website Clustering", Proceedings of the 2009 Symposia and Workshops on Ubiquitous, Autonomic and Trusted Computing, 488-492

Lee, A. & Kawahara, T. (2009). "Recent Development of Open-Source Speech Recognition Engine Julius", Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC).

Martignoni, L.; Christodorescu, M. & Jha, S. (2007). "OmniUnpack: Fast, Generic, and Safe Unpacking of Malware", Proceedings of the Annual Computer Security Applications Conference (ACSAC).

McGraw, G & Morrisett. G (2000). “Attacking malicious code: A report to the infosec research council”, IEEE Software, 17(5), 33–44.

Microsoft, (2007). “Understanding Anti-Malware Technologies”, http://download.microsoft.com/download/0/c/0/0c040c8f-2109-4760-a750-96443fd14ef2/Understanding Malware Research and Response at Microsoft.pdf, August 2009 .

Microsoft, (2010). Win32/Parite, http://www.microsoft.com/security/p-ortal/Threat/Encyclopedia/Entry.aspx?name=Win32%2fParite, Feb 2010.

Microsoft Developer Network (MSDN). (2010). Windows API Functions, http://msdn.microsoft.com/en-us/library/aa383749%28VS.85%29.aspx. January 2010.

Paul, N. (2008). “Disk-Level Behavioral Malware Detection”, University of Virginia, Citeseer, Doctor of Philosophy Dissertation, Chapter 2.

PEiD. (2010). Snaker, Qwerton, Jibz & xineohP, http://www.peid.info/, Jan 2010.

Perriot, F. & Ferrie, P. (2004). "Principles and practise of X-RAYING", *Virus Bulletin Conference (VB 2004),* , 51-66.

Royal, P.; Halpin, M.; Dagon, D.; Edmonds, R. & Lee, W. (2006). "PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware", IEEE Computer Society, the 22nd Annual Computer Security Applications Conference (ACSAC'06), 289-300.

Schambach, M. (2009). "Recurrent HMMs and Cursive Handwriting Recognition Graphs", 2009 10th International Conference on Document Analysis and Recognition, 1146-1150.

Sharif, M.; Yegneswaran, V.; Saidi, H.; Porras, P. & Lee, W. (2008). "Eureka: A framework for enabling static malware analysis", Computer Security - ESORICS, Lecture Notes in Computer Science LNCS, Springer, 5283/2008, 481-500.

SQLite, www.sqlite.org/, February 2010.

Stepan, A. (2005). "Defeating polymorphism: Beyond emulation", *Proceedings of the Virus Bulletin International Conference*.

Stolfo, S.; Wang, K. & Li, W. (2006). "Towards stealthy malware detection", Malware Detection, Advances in Information Security, Springer, 27, 231-249.

Sulaiman, A.; Ramamoorthy, K.; Mukkamala, S. & Sung, A. (2005). " Disassembled code analyzer for malware (DCAM)", Information Reuse and Integration IRI-2005 IEEE International Conference on., 398-403.

Swets J. and Pickett. R. (1982). “Evaluation of diagnostic system: Methods from signal detection theory”. Academic Press.

Symantec. (1997).“Understanding Heuristics: Symantec's Bloodhound Technology”, Symantec White Paper Series, XXXIV, 1-14.

Turner, D. (2006). “Semantic internet security, threat report", http://eval.symantec.com/mktginfo/enterprise/white_papers/ent-whitepaper_-symantec_internet_security_threat_report_x_09_2006.en-us.pdf, Trends for Jan 2006 - jun 2006, X.

Venkatraman, S. (2009). "Autonomic Context-Dependent Architecture for Malware Detection", Proceedings of International Conference on e-Technology (e-Tech2009), International Business Academics Consortium, ISBN 978-986-83038-3-6, 8-10 January, Singapore, 2927-2947.

Vinod, P.; Jaipur, R.; Laxmi, V. & Gaur, M. (2009). “Survey on Malware Detection Methods”, Hack. 74.

Wang, C.; Pang, J.; Zhao, R. & Liu, X. (2009). "Using API Sequence and Bayes Algorithm to Detect Suspicious Behavior", 2009 International Conference on Communication Software and Networks, 544-548.

Yegneswaran, V.; Giffin, J.; Barford, P. & Jha, S. (2005). " An architecture for generating semantics-aware signatures", Proceedings *of the 14th USENIX Security Symposium,* 97-112.