Detecting Internet Worms Using Data Mining Techniques

Muazzam SIDDIQUI
Institute of Simulation & Training
University of Central Florida
siddiqui@mail.ucf.edu

Morgan C. WANG
Department of Statistics and Actuarial Sciences
University of Central Florida
cwang@mail.ucf.edu

Joohan LEE
School of Electrical Engineering & Computer Science
University of Central Florida
jlee@cs.ucf.edu

ABSTRACT

Internet worms pose a serious threat to computer security. Traditional approaches using signatures to detect worms pose little danger to the zero day attacks. The focus of malware research is shifting from using signature patterns to identifying the malicious behavior displayed by the malwares. This paper presents a novel idea of extracting variable length instruction sequences that can identify worms from clean programs using data mining techniques. The analysis is facilitated by the program control flow information contained in the instruction sequences. Based upon general statistics gathered from these instruction sequences we formulated the problem as a binary classification problem and built tree based classifiers including decision tree, bagging and random forest. Our approach showed 95.6% detection rate on novel worms whose data was not used in the model building process.

Keywords
Data Mining, Worm Detection, Binary Classification, Static Analysis, Disassembly, Instruction Sequences

1. INTRODUCTION

Computer virus detection has evolved into malware detection since Cohen first formalized the term computer virus in 1983 [13]. Malicious programs, commonly termed as malwares, can be classified into virus, worms, trojans, spywares, adwares and a variety of other classes and subclasses that sometimes overlap and blur the boundaries among these groups [24]. The most common detection method is the signature based detection that makes the core of every commercial anti-virus program. To avoid detection by the traditional signature based algorithms, a number of stealth techniques have been developed by the malware writers. The inability of traditional signature based detection approaches to catch these new breed of malwares has shifted the focus of malware research to find more generalized and scalable features that can identify malicious behavior as a process instead of a single static signature.
The analysis can roughly be divided into static and dynamic analysis. In the static analysis the code of the program is examined without actually running the program while in dynamic analysis the program is executed in a real or virtual environment. The static analysis, while free from the execution overhead, has its limitation when there is a dynamic decision point in the programs control flow. Dynamic analysis monitors the execution of program to identify behavior that might be deemed malicious. These two approaches are combined also [23] where dynamic analysis is applied only at the decision-making points in the program control flow.
In this paper we present a static analysis method using data mining techniques to automatically extract behavior from worms and clean programs. We introduce the idea of using sequence of instructions extracted from the disassembly of worms and clean programs as the primary classification feature. Unlike fixed length instructions or n-grams, the variable length instructions inherently capture the programs control flow information as each sequence reflects a control flow block.
The difference among our approach and other static analysis approaches mentioned in the related research section are as follows.
First, the proposed approach applied data mining as a complete process from data preparation to model building. Although data preparation is a very important step in a data mining process, almost all existing static analysis techniques mentioned in the related research section did not discuss this step in detail except [25]. Second, all features were sequences of instructions extracted by the disassembly instead of using fixed length of bytes such as n-gram. The advantages are:

1. The instruction sequences include program control flow information, not present in n-grams.
2. The instruction sequences capture information from the program at a semantic level rather than syntactic level.
3. These instruction sequences can be traced back to their original location in the program for further analysis of their associated operations.
4. These features can be grouped together to form additional derived features to increase classification accuracy.
5. A significant number of sequences that appeared in only clean program or worms can be eliminated to speed up the modeling process.
6. The classifier obtained can achieve 95% detection rate for new and unseen worms.

It is worth noting that a dataset prepared for a neural network classifier might not be suitable for other data mining techniques such as decision tree or random forest.

2. RELATED RESEARCH

[18] divided worm detection into three main categories; Traffic monitoring, honeypots and signature detection. Traffic analysis includes monitoring network traffic for anomalies like sudden increase in traffic volume or change in traffic pattern for some hosts etc. Honeypots are dedicated systems installed in the network to collect data that is passively analyzed for potential malicious activities. Signature detection is the most common method of worm detection where network traffic logs, system logs or files are searched for worm signatures.

Data mining has been the focus of many malware researchers in the recent years to detect unknown malwares. A number of classifiers have been built and shown to have very high accuracy rates. Data mining provides the means for analysis and detection of malwares for the categories defined above. Most of these classifiers use n-gram or API calls as their primary feature. An n-gram is a sequence of bytes of a given length extracted from the hexadecimal dump of the file. Besides file dumps, network traffic data and honeypot data is mined for malicious activities.

[17] introduced the idea of using tell-tale signs to use general program patterns instead of specific signatures. The tell-tale signs reflect specific program behaviors and actions that identify a malicious activity. Though a tell-tale sign like a sequence of specific function calls seems a promising identifier, yet they did not provide any experimental results for unknown malicious programs.

The idea of tell-tale signs was furthered by [10] and they included program control and data flow graphs in the analysis. Based upon the tell-tale signs idea, they defined a security policy using a security automata. The flow graphs are subjected to these security automata to verify against any malicious activity. The method is applied to only one malicious program. No other experimental results were reported to describe algorithm efficiency, especially on unseen data.

In another data mining approach, [20] used three different types of features and a variety of classifiers to detect malicious programs. Their primary dataset contained 3265 malicious and 1001 clean programs. They applied RIPPER (a rule based system) to the DLL dataset. Strings data was used to fit a Naive Bayes classifier while n-grams were used to train a Multi-Naive Bayes classifier with a voting strategy. No n-gram reduction algorithm was reported to be used. Instead data set partitioning was used and 6 Naive-Bayes classifiers were trained on each partition of the data. They used different features to built different classifiers that do not pose a fair comparison among the classifiers. Naive-Bayes using strings gave the best accuracy in their model.

A similar approach was used by [15], where they built different classifiers including Instance-based Learner, TFIDF, Naive-Bayes, Support vector machines, Decision tree, boosted Naive-Bayes, SVMs and boosted decision tree. Their primary dataset consisted of 1971 clean and 1651 malicious programs. Information gain was used to choose top 500 n-grams as features. Best efficiency was reported using the boosted decision tree J48 algorithm.

[9] used n-grams to build class profiles using KNN algorithm. Their dataset was small with 25 malicious and 40 benign programs. As the dataset is relatively small, no n-gram reduction was reported. They reported 98% accuracy rate on a three-fold cross validation experiment. It would be interesting to see how the algorithm scale as a bigger dataset is used.

[22] proposed a signature based method called SAVE (Static Analysis of Vicious Executables) that used behavioral signatures indicating malicious activity. The signatures were represented in the form of API calls and Euclidean distance was used to compare these signatures with sequence of API calls from programs under inspection.

Besides data mining, other popular methods includes activity monitoring and file scanning. [19] proposed a system to detect scanning worms using the premises that scanning worms tend to reside on hosts with low successful connections rates. Each unsuccessful or successful connection attempt was assigned a score that signals a host to be infected if past a threshold. [14] proposed behavioral signatures to detect worms in network traffic data. [16] developed Honeycomb, that used honeypots to generate network signatures to detect worms. Honeycomb used anomalies in the traffic data to generate signatures.

All of this work stated above, that does not include data mining as a process, used very few samples to validate their techniques. The security policies needed human experts to devise general characteristics of malicious programs.

Data preparation is a very important step in a data mining process. Except [25], none of the authors presented above have discussed their dataset in detail. Malicious programs used by these researchers are very eclectic in nature exhibiting different program structures and applying the same classifier to every program does not guarantee similar results.

3. DATA PROCESSING

Our collection of worms and clean programs consisted of 2775 Windows PE files, in which 1444 were worms and the 1330 were clean programs. The clean programs were obtained from a PC running Windows XP. These include small Windows applications such as calc, notepad, etc and other application programs running on the machine. The worms were downloaded from [8]. The dataset was thus consisted of a wide range of programs, created using different compilers and resulting in a sample set of uniform representation. Figure 3 displays the data processing steps.

Malware Analysis

We ran PEiD [5] and ExEinfo PE [2] on our data collection to detect compilers, common packers and cryptors, used to compile and/or modify the programs. Table 1 displays

Table 1. Packers/Compilers Analysis of Worms

Packer/Compiler	Number of Worms
ASPack	77
Borland	110
FSG	31
Microsoft	336
Other Not Packed	234
Other Packed	83
PECompact	26
Unidentified	140
UPX	67

Table 2. Packers/Compilers Analysis of Worms and Clean Programs

Type of Program	Not Packed	Packed	Unidentified
Clean	1002	0	49
Worm	624	340	140
Total	1626	340	189

the distribution of different packers and compilers on the worm collection.

The clean programs in our collection were also subjected to PEiD and ExeInfo PE to gather potential packers/crytpors information. No packed programs were detected in the clean collection. Table 2 displays the number of packed, not packed and unidentified worms and clean programs.

Before further processing, packed worms were unpacked using specific unpackers such as UPX (with -d switch) [6], and generic unpackers such as Generic Unpacker Win32 [3] and VMUnpacker [7].

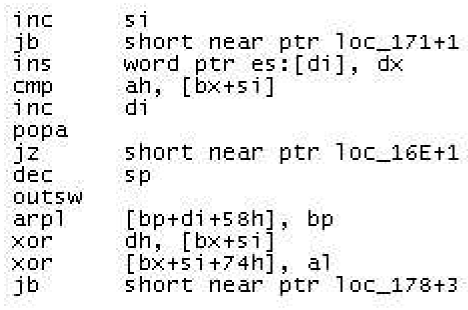
File Size Analysis

Before disassembling the programs to extract instruction sequences, a file size analysis was performed to ensure that the number of instructions extracted from clean programs and worms is approximately equal. Table 3 displays the file size statistics for worms and clean programs.

Table 3 indicates the that the average size of the clean programs is twice as large as average worm size. These large programs were removed from the collection to get an equal file size distribution for worms and clean programs.

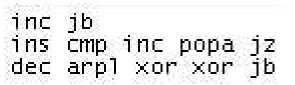
Table 3. File Size Analysis of the Program Collection

Statistic	Worms Size (KB)	Cleans Size (KB)
Average	67	147
Median	33	43
Minimum	1	1
Maximum	762	1968



```
inc     si
jb      short near ptr loc_171+1
ins     word ptr es:[di], dx
cmp     ah, [bx+si]
inc     di
popa
jz      short near ptr loc_16E+1
dec     sp
outsw
arpl    [bp+di+58h], bp
xor     dh, [bx+si]
xor     [bx+si+74h], al
jb      short near ptr loc_178+3
```

Figure 1. Portion of the output of disassembled Netsky.A worm.



```
inc jb
ins cmp inc popa jz
dec arpl xor xor jb
```

Figure 2. Instruction sequences extracted from the disassembled Netsky.A worm.

Disassembly

Binaries were transformed to a disassembly representation that is parsed to extract features. The disassembly was obtained using Datarescues' IDA Pro [4]. From these disassembled files we extracted sequences of instructions that served as the primary source for the features in our dataset. A sequence is defined as instructions in succession until a conditional or unconditional branch instruction and/or a function boundary is reached. Instruction sequences thus obtained are of various lengths. We only considered the opcode and the operands were discarded from the analysis. Figure 1 shows a portion of the disassembly of the Netsky.A worm.

Parsing

A parser written in PHP translates the disassembly in figure 1 to instruction sequences. Figure 2 displays the output of the parser. Each row in the parsed output represented a single instruction sequence. The raw disassembly of the worm and clean programs resulted in 1972920 instruction sequences. 47% of these sequences belonged to worms while 53% belonged to clean programs.

Feature Extraction

The parsed output was processed through our Feature Extraction Mechanism. Among them 1972920 instruction sequences, 213330 unique sequences were identified with different frequencies of occurrence. We removed the sequences that were found in one class only as they will reduce the classifier to a signature detection technique. This removed 94% of the sequences and only 23738 sequences were found common to both worms and clean programs. Each sequence was considered as a potential feature.

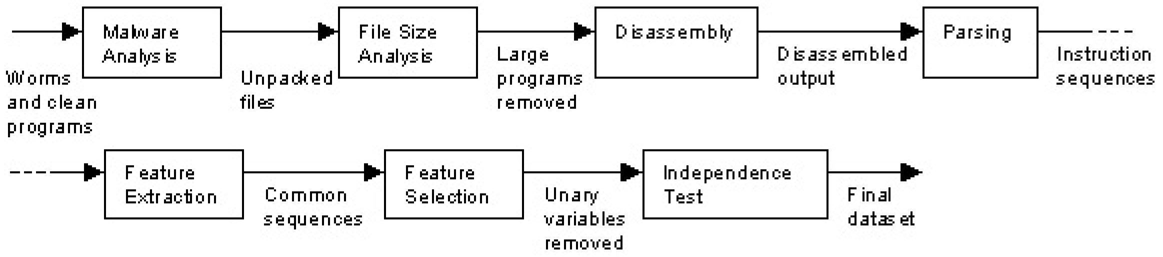


Figure 3. Data preprocessing steps.

Feature Selection

The Feature Selection Mechanism considered frequency of occurrence of each sequence in the entire data to be the primary selection criteria. Sequences with less than 10% frequency of occurrence were identified as rare items are were not included in the dataset. This removed 97% of the sequences and only 679 sequences were selected. The dataset consisted of frequency of occurrence of each of these sequences in each file. A binary target variable identified each file as worm or clean.

Using the occurrence frequency as the primary data item in the dataset enabled us to consider the features as count variables.

Independence Test

A Chi-Square test of independence was performed for each feature to determine if a relationship exists between the feature and the target variable. The variables were transformed to their binary representation on a found/not found basis to get a 2-way contingency table. Using a p-value of 0.01 for the test resulted in the removal of about half of the features that did not showed any statistically significant relationship with the target. The resulting number of variables after this step was 268.

4. EXPERIMENTS

The data was partitioned into 70% training and 30% test data. Similar experiments showed best results with tree based models for the count data [21]. We built decision tree, bagging and Random forest models using R [1].

Decision Tree

A decision tree recursively partitions the predictor space to model the relationship between predictor variables and categorical response variable. Using a set of input-output samples a tree is constructed. The learning system adopts a top-down approach that searches for a solution in a part of the search space. Traversing the resultant tree gives a set of rules that finally classified each observation into the given classes. We used the decision tree model to obtain a set of rules that can classify each sample into either malicious or benign class.

The decision tree model we built in R used Gini as split criterion with a maximum depth of 15.

Bagging

Bagging or Bootstrap Aggregating is a meta-algorithm to improve classification and regression models in terms of accuracy and stability. Bagging generates multiple versions of a classifier and uses plurality vote to decide for the final class outcome among the versions. The multiple versions are created using bootstrap replications of the original dataset. Bagging can give substantial gains in accuracy by improving on the instability of individual classifiers. [11]

We used classification trees with 100 bootstrap replications in the Bagging model.

Random Forest

Random forest provides a degree of improvement over Bagging by minimizing correlation between classifiers in the ensemble. This is achieved by using bootstraping to generate multiple versions of a classifier as in Bagging but employing only a random subset of the variables to split at each node, instead of all the variables as in Bagging. Using a random selection of features to split each node yields error rates that compare favorably to Adaboost, but are more robust with respect to noise.[12]

We grew 100 classification trees in the Random forest model. The number of variables sampled at each split was 22.

5. RESULTS

We tested the models using the test data. Confusion matrices were created for each classifier using the actual and predicted responses. The following four estimates define the members of the matrix.

True Positive (TP): Number of correctly identified malicious programs.

False Positive (FP): Number of wrongly identified benign programs.

True Negative (TN): Number of correctly identified benign programs.

False Negative (FN): Number of wrongly identified malicious programs.

The performance of each classifier was evaluated using the detection rate, false alarm rate and overall accuracy that can be defined as follows:

Detection Rate: Percentage of correctly identified malicious programs.

Table 4. Experimental results

Classifier	Detection Rate	False Alarm Rate	Overall Accuracy
Random Forest	95.6%	3.8%	96%
Bagging	94.3%	6.7%	93.8%
Decision Tree	93.4%	13.4%	90%

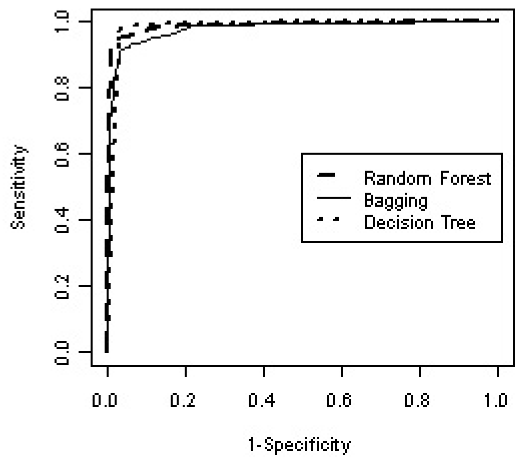


Figure 4. ROC curve comparing decision tree, bagging and random forest test results.

$DetectionRate = \frac{TP}{TP+FN}$

False Alarm Rate: Percentage of wrongly identified benign programs.
$FalseAlarmRate = \frac{FP}{TN+FP}$

Overall Accuracy: Percentage of correctly identified programs.
$OverallAccuracy = \frac{TP+TN}{TP+TN+FP+FN}$

Table 4 displays the experimental results for each classifier. Figure 4 displays the ROC curves for test data for each model. The meta-algorithms performed better than a single decision tree as expected. Random forest performed slightly better than Bagging which is endorsement of its superiority over Bagging as claimed in [12]

6. CONCLUSIONS

In this paper we presented a data mining framework to detect worms. The primary feature used for the process was the frequency of occurrence of variable length instruction sequences. The effect of using such a feature set is two fold as the instruction sequences can be traced back to the original code for further analysis in addition to being used in the classifier. We used the sequences common to both worms and clean programs to remove any biases caused by the features that have all their occurrences in one class only. We showed 95.6% detection rate with a 3.8% false positive rate.

Table 5. Area under the ROC curve for each classifier.

Classifier	AUC
Decision Tree	0.9060
Bagging	0.9775
Random Forest	0.9871

7. FUTURE WORK

The information included for this analysis was extracted from the executable section of the PE file. To achieve a better detection rate this information will be appended from information from other sections of the file. This will include Import Address Table and the PE header. API calls analysis has proven to be an effective tool in malware detection [22]. Moreover header information has been used in heuristic detection [24]. Our next step is to include this information in our feature set.

References

[1] The r project for statistical computing http://www.r-project.org/.

[2] ExEinfo PE. http://www.exeinfo.go.pl/.

[3] Generic Unpacker Win32. http://www.exetools.com/unpackers.htm.

[4] IDA Pro Disassembler. http://www.datarescue.com/idabase/index.htm.

[5] PEiD. http://peid.has.it/.

[6] UPX the Ultimate Packer for eXecutables. http://www.exeinfo.go.pl/.

[7] VMUnpacker. http://dswlab.com/d3.html.

[8] VX Heavens. http://vx.netlux.org.

[9] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan. N-gram-based detection of new malicious code. In *Proceedings of the 28th Annual International Computer Software and Applications Conference - Workshops and Fast Abstracts - (COMPSAC'04) - Volume 02*, pages 41–42, 2004.

[10] J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie, and N. Tawbi. Static detection of malicious code in executable programs. *Symposium on Requirements Engineering for Information Security (SREIS'01)*, 2001.

[11] L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[12] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[13] F. Cohen. *Computer Viruses*. PhD thesis, University of Southern California, 1985.

[14] D. Ellis, J. Aiken, K. Attwood, and S. Tenaglia. A behavioral approach to worm detection. In *Proceedings of the 2004 ACM Workshop on Rapid Malcode*, pages 43–53, 2004.

[15] J. Z. Kolter and M. A. Maloof. Learning to detect malicious executables in the wild. In *Proceedings of the 2004 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2004.

[16] C. Kreibich and J. Crowcroft. Honeycomb creating intrustion detection signatures using honeypots. In *2nd Workshop on Hot Topics in Network*, 2003.

[17] R. W. Lo, K. N. Levitt, and R. A. Olsson. Mcf: A malicious code filter. *Computers and Security*, 14(6):541–566, 1995.

[18] J. Nazario. *Defense and Detection Strategies against Internet Worms*. Van Nostrand Reinhold, 2004.

[19] S. E. Schechter, J. Jung, , and B. A. W. fast detection of scanning worms infections. In *Proceedings of Seventh International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2004.

[20] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 38–49, 2001.

[21] M. Siddiqui, M. C. Wang, and J. Lee. Data mining methods for malware detection using instruction sequences. In *Proceedings of Artificial Intelligence and Applications, AIA 2008*. ACTA Press, 2008.

[22] A. H. Sung, J. Xu, P. Chavez, and S. Mukkamala. Static analyzer of vicious executables. In *20th Annual Computer Security Applications Conference*, pages 326–334, 2004.

[23] Symantec. Understanding heuristics: Symantec's bloodhound technology. Technical report, Symantec Corporation, 1997.

[24] P. Szor. *The Art of Computer Virus Research and Defense*. Addison Wesley for Symantec Press, New Jersey, 2005.

[25] M. Weber, M. Schmid, M. Schatz, and D. Geyer. A toolkit for detecting and analyzing malicious software. In *Proceedings of the 18th Annual Computer Security Applications Conference*, page 423, 2002.