File Checker: Determining Behavioural Signatures of an Executable Binary to Detect Malware

Harshal R. Shinde
SRM Institute of Science and Technology
Vadapalani, Chennai
Tamil Nadu, India

Himanshu Shukla
SRM Institute of Science and Technology
Vadapalani, Chennai,
Tamil Nadu, India

A. Jothimani
Assistant Professor
SRM Institute of Science and Technology
Vadapalani, Chennai
Tamil Nadu, India

Anurag Singh Baghel
SRM Institute of Science and Technology
Vadapalani, Chennai
Tamil Nadu, India

ABSTRACT

The increasing dependency in this technologically advancing world on data is making us vulnerable to frequent cyber-attacks. This study aims at classifying executable binaries(Portable Executable files) based on its run-time behaviour. Traditional approaches to detecting windows-based malware include comparing files hashes, strings, etc., which clearly failed to detect the new world malware kinds - morphed and obfuscated. Although the dynamically based detection distinctly outperformed static based detection techniques, it failed to effectively detect advanced malicious programs. System-call injection attacks usually inject irrelevant calls to alter an execution sequence of malware, thereby making it undetectable to calls based detection systems. The proposed method aims at extracting traces of API calls made to generate possible unique alternative traces in order to detect other malicious API patterns which may be left out due to prevent call injection attacks. A classification model is built by employing the RandomForest algorithm, and its efficiency is compared with other baseline classifiers. This model classifies the data effectively with 91.9% accuracy.

General Terms

Malware Detection, RandomForest Algorithm, Machine Learning Algorithms

Keywords

Executable Binaries, Portable Executable, Malware, Hashes, Morphed, Obfuscated, System-Call Injection Attacks, API Patterns.

1. INTRODUCTION

With the rise in sophistication of computing systems, malicious attacks became more advanced and complex. Maintaining the security of an organization's systems and the network is a day to day struggle, for instance in the first half of 2019, approximately 4.1 billion records of data were breached. Hackers on an average attack 2244 times a day and every 24 seconds. Coming up with new ways like crypto-jacking - hacking third party computers to mine cryptocurrencies, it took on an average of 206 days to identify a data breach. This rate of advancement needs tantamount tackling measures to curb cyber-crime. Almost 92% of malware attacks were carried out by sending out emails and Ransomware attacks - Clop, RaaS, NotPetya, WannaCry and fake windows update, caused millions of dollars to enterprises and individuals. Also, sources say that, by the end of 2020, 83% of enterprise workloads will be on the cloud. Hence protection of information systems and networks from these kinds of attacks is of utmost importance.

Existing antivirus detection methods mainly still include signature-based detection from databases which typically compares the patterns in the databases with that of the file. Also, the databases need to be updated frequently which makes it vulnerable to the new kinds of attacks if a user hasn't updated it. With the development in malware kinds - morphed and obfuscated, it is hard to compare a file as some files can have multiple patterns and some even come with packed programming contents which makes it difficult to identify it's patterns. An increase in creative attacking strategy requires an extra effort with the update of detection strategies.

As of now, two basic types of detection used include static and dynamic. Static methods mainly focus on the functionality of a file by finding strings, checking whether packed or not and scanning PE file headers. Whereas dynamic behaviour focuses on run-time behaviour of a file by monitoring it's API calls made, registries used, the semantics of program and list of other system services used[1]. The static method for malware detection may skip these minute details which would eventually provide faulty outputs. Being able to monitor at its semantic level, it makes difficult for malware to evade the detection mechanisms. Therefore, there is a need to devise new run-time behaviour based strategies for making it anti-escape-proof.

In this paper, they came up with a different and an effective malware detection strategy, which not only makes use of system-level windows API calls sequences but also looks for possible alternative API call patterns thereby making it resilient against irrelevant-calls injection attacks and polymorphic malware. This mechanism observes the behaviour of the program by tracing a pattern of system calls made and using it to create the dataset and train model. File's semantic-patterns were exploited to extract and generate alternative similar patterned paths after crossing a certain threshold value. There exists unnecessary system calls in trace of calls(patterns) which misleads and has no pre-defined objective. After getting a log file containing sequential process calls they generate Sequenced Execution Trace Graph(SETG) to generate possible similar trace call patterns.

Average path probability metric is used to separate more semantically relevant paths which have more chances of depicting a program's behaviour. This generation of alternative paths that also carry almost all information about the behaviour acts as a strong defence for polymorphic malware. They use all these semantically relevant patterns to construct their vector feature space made and train it. Eventually, increasing the level of accuracy of the proposed model by considering both the original trace of calls and it's alternative relevant paths.

2. THE PROBLEM

Malware authors and hackers continuously evolve their anti-detection techniques to evade malware detection. They do so by incorporating false, independent, or irrelevant system calls into malicious code. These techniques are sometimes also called obfuscation, polymorphism and metamorphism. These methods help malware to attack with the same motive by constantly changing it's form thereby making it able to covertly carry out its operations with malicious intent.

3. RELATED WORK

They proposed a solution to the above-described problem of malware detection. Most modern-day malware is aware of existing detection methods and is able to escape the anti-malware systems. One major challenge to anti-malware detection mechanisms is to identify zero-day or novel malware attacks. Their approach observes the run-time behaviour of malware for detection and overcomes the limitations of static detection methods. They discuss the related work of researchers that considers the dynamic behaviour of malware for detection by employing methods like graph construction and n-gram.

Authors in [2] have introduced an approach based on API call sequences, for feature extraction and selection which uses text mining and topic modelling. A same kind of approach was followed by authors [3], to construct their feature vector space by making use of API calls and to remove unnecessary calls they employed two feature selection methods fisher score and CFSSUBSETEVAL.

3.1 N-gram Based

N-gram approach involves a selection of n features from the trace of system-calls over a fixed-sized sliding window. The first attempt ever made to detect malware by employing API calls was done by Hofmeyr et al. (1998) [4]. The authors in [5] have presented an approach that checks for n-feature-fingerprints, present in malware files but absent in benign files. Adversely, then the n-gram approach has some limitations like only contiguous features, the only n features to be considered and lacks when it comes to dimensionality.

3.2 Graph-Based

Graph-based representation characterises the dynamic behaviour of malware into a cluster. This approach involves the construction of a network of features that helps in identifying the control flow or information flow. This generated graph consists of execution traces of system-calls and the ultimate objective of this malware can also be obtained by intercepting other paths carrying almost all similar information. The author [6] presents an approach that extracts semantically-relevant path with help of Asymptotic Equipartition Property(AEP) and uses Average Logarithmic Branching Factor(ALBF) metric to classify paths generated into respective bins for the construction of their feature vector space model. Other similar approaches are followed in [7],[8],[9] papers. Although their proposed robust approach employs graph construction, it doesn't rely solely on exact original call traces but also helps in determining other similar patterned traces as shown in their experiments.

4. PROPOSED SYSTEM

To detect and analyse malware only syntactic comparison is not sufficient as it can mutate to other forms continuously and carry out its functionality covertly. Hence, their proposed mechanism targets dynamic behaviour and is not limited to a static level. In order to understand it's behaviour, it is necessary to observe and understand its semantics and control flow. Therefore, they executed the malware in controlled space to know it. However, advanced malware is virtual environment-sensitive ie., it can sense it's execution in a virtual environment and conceal its original malicious intent to seem benign.

The extraction of information-rich paths with high chances from the original trace of calls required a metric which could be used for quantifying program semantics. And for this, they used the Average Probability Metric(APM) to separate fewer probability paths from information-rich ones. By extracting similar patterned call sequences they build the feature vector space for classification.

4.1 Methodology

Their approach of run-time behaviour analysis was a major reason for not being vulnerable to irrelevant system-calls injection or metamorphic attacks. Two major factors that made the model efficient were multiple path selection and program-specific features. Modern malware mostly depicts mutation ie., exhibits the same behaviour with different program call sequences. For instance, if a hacker wants to keep a log file of a user to know the software used he/she can do so in one of the given ways as -

1. NtCreateFile->NtOpenFile->NtWriteFile->NtReadFile->NtCloseFile,
2. NtWriteFile->NtReadFile->NtCloseFile.

Now they log the information of the paths that more closely depicts the behaviour of the program(malware) and has a higher probability of occurring. Further, to eliminate the possibility of string evasion they convert these paths into numbers. This study helps in identifying malware by transforming it into multiple semantically relevant paths and then utilizing it to build the classification model, thereby checking the efficiency of the model.

4.2 SETG Construction

In order to construct the Sequenced Execution Trace Graph(SETG) graphs, they monitor the execution of binary executable and the system calls invoked are recorded. Now in order to convert these calls into graphs, they considered process calls invoked as nodes and then added an edge between two transitioning nodes. For example, if two process calls are invoked P1 and P2 respectively, then after considering P1 and P2 as nodes or vertices, a directed edge is added to it. So that it represents the transition from node P1 to P2. One important thing to note is that if there is a transition from P2 to P1 in the latter part of the sequence then a different directed edge is also added between them such that sequential call transitions are preserved all the time. At first, they used ProcMon and Cuckoo sandbox to intercept the calls during the execution of the malware on the Windows platform. Process hacker was also used to monitor the process created, modified and deleted. To create a log file they run the ProcMon in the background then run the malware and this ProcMon captures all the system calls made.

However, intercepting API calls is time-consuming which adds extra overhead to the overall process. One can use virtual machines or emulators - VirMon, Cuckoo, Kirda, any.run, and Anubis, to analyse the program's behaviour during execution. Afterwards due to inconvenience caused by using ProcMon they used Cuckoo sandbox.

Definition 1 An SETG G = (N, E) is a directed graph with N set of nodes and an edge E for each transition occurring. Let p_{ij} be the transition probability from node N_i to N_j, such that an edge can be represented as

$$E = \{ Eij \mid Ni \rightarrow Nj;\ Ni, Nj \in \mathrm{N} \}$$

We know that in Markov chains the future state rather than depending on past states is contingent upon the current state. Therefore, for a general transition from node N_i to N_j, the transition probability must satisfy Markov's property[10] and can be represented as

$$p_{ij} = \text{count}(Ni \rightarrow Nj) \,/\, \textstyle\sum^{n}_{h=1} \text{count}(Ni \rightarrow Nh) \qquad (1)$$

$$\forall i \quad \textstyle\sum^{n}_{j=1} p_{ij} = 0 \ , Ni \text{ is isolated node } \forall i \qquad (2)$$

1, or else

Let us consider a sequence of system-calls invoked as $\xi = \{ N_1, N_2, N_3, N_4, N_4, N_2, N_4, N_1, N_3 \}$ and it's execution trace be represented by P.

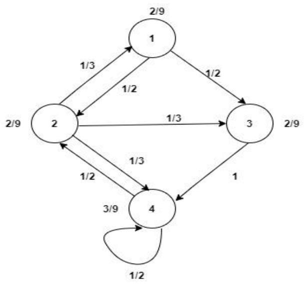
Definition 2 Assuming a path as $P = \{ N_1, N_2, \ldots\ldots.. N_n\}$ where starting node is N_1 and the destination node is N_n with a total of n system calls invoked.

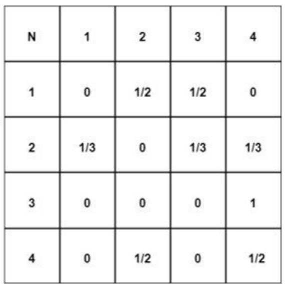
In this case, the execution trace is converted into a SETG with each link representing the transition to subsequent calls invoked. In a SETG, path probability from source to destination (N_S -> N_D) is calculated by computing transition probability of all the individual links present in a selected path. For instance, if path probability is represented by Pr(P) then,

$$\Pr(P) = p_{12} * p_{23} * \ldots\ldots * p_{n(n-1)} \qquad (3)$$

After computing Pr(P) for a given path P then this is multiplied by the initial probability of the source node(N_S). The aim is to derive all the possible paths from source to destination (N_S -> N_D) which is an NP-complete problem[11] and calculate Pr(P) for all the paths. And once the Pr(P) of a path crosses a certain threshold value it is considered along with ξ to construct the feature vector space. However, if the execution trace is very less detailed or overly-detailed than these paths may be rejected as it may act as noise and hamper the model. To elaborate, a file with a short execution trace may not provide essential information required for classification as it covers limited instances while a file with very long execution trace may provide too much detailed data that the model over learns from it and may misclassify the files.

For the considered ξ, it's constructed SETG and transition probability matrix is shown below in Fig 1 along with path probability of all possible paths from source to destination (N_1 ->N_4) in Fig. 3.





N	1	2	3	4
1	0	1/2	1/2	0
2	1/3	0	1/3	1/3
3	0	0	0	1
4	0	1/2	0	1/2

Fig 1: Sequenced Execution Trace Graph (SETG) and Transition Probability Matrix (TPM)

Table 1. Path Probability

Paths	Path Probability
P_1 : 1-2-4	0.0370
P_2 : 1-3-4	0.1111
P_3 : 1-2-3-4	0.0370

After computing path probability for all the paths, Average Probability metric(APM) of the execution trace(ξ) is 0.0617 and out of three possible paths path with trace calls (N_1 - N_3 - N_4) is selected for building up feature space. Similarly, this process is repeated for all the other samples.

4.3 Feature Selection

The main aim of the last step of this experiment was to check the impact of the number of features considered on the proposed model(ie., a trade-off between less numbered features and overly numbered features). Selection of apt data for building the feature space (S) has a great influence over a model's prediction ability. Data is gathered from all the semantically relevant paths and loaded into a binary matrix. A high ranked attribute is the one with high influence, which carries most information and causes less randomisation or deviation in a feature space. According to the author's research in [12], it was observed that out of four categories of features - Registry Edits, DLLs, File Modifications, and System Calls, when 2000 features were considered major contribution was of System Call category over 85% while the rest with below 5% contribution.

After the construction of feature space, the model was trained using an ensemble-based machine learning algorithm - RandomForest algorithm. In this algorithm, randomly, data and features are selected from training data as a subset for a decision tree. It consists of various decision trees with a predefined length of trees and attributes. It generalises the data obtained from various decision trees and predicts with less deviation due to noise. However, they have also compared the results of RandomForest algorithm with that of logistic Regression, Naïve Bayes and decision tree for a comparative study.

5. ANALYSIS AND RESULTS

In this section, they assess their experimental outcomes and also, the efficacy of machine learning algorithms used. Performed by maintaining a balance between the number of malware and benign samples used.

5.1 Dataset

In this experiment to avoid any kind of bias generated by skewed data, they used 124 malware samples and 135 benign samples to create the dataset. Real-world malware and benign samples were used to train this model. The problem with signature-based datasets is that it is restricted to only original forms, malware and defenceless against polymorphic malware. An almost balanced mix of good and bad files was used to train the model. The malware and benign samples were obtained from a trusted online malware analyser-virustotal.com. Since some malware prohibits it's execution the moment it senses an emulator to conceal it's behaviour samples were run in Cuckoo sandbox which runs malware virtually on Windows platform simultaneously to intercept system-process calls, registry and network activities. Once all the log files were obtained, a feature selection criteria was applied to filter data and generate the dataset. From Fig 2 it is clearly evident that malware files generally had way longer path length than benign files. The features generated were over 150 and the same features were used for all the training data.

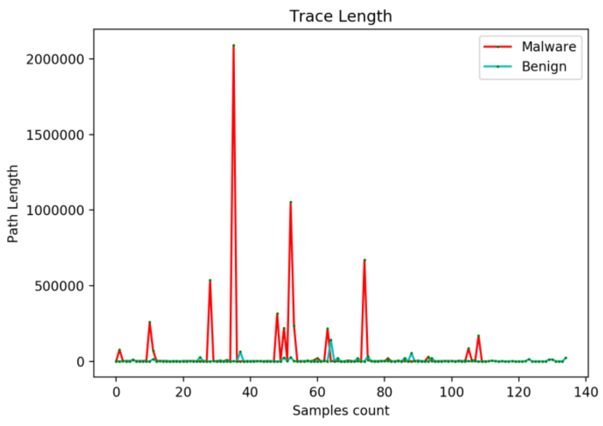


Fig 2: Path length of sample files

5.2 Classification Efficiency

After the features selection measure was employed, training data was fed to the classification algorithms. Several measures were used to compensate for the change in the efficiency measures and finally, the one with high accuracy was selected. Still, to fit the data properly they had to experiment with the number of features selected. The number of selected features ranged from 100 to 200. After selecting a branching factor the model's predictions were closely observed to notice the impact of the number of features selected.

For calibration and checking the efficiency of their model they have incorporated widely used and well-known measures[13]- True Positive Percentage(TPP), True Negative Percentage(TNP), False Positive Percentage(FPP), False Negative Percentage(FNP). On one hand, for an efficient model, TPP and TNP should be as high as possible, on the other hand, FPP and FNP should be as low as possible. TPP tells about the number of malware instances correctly classified while TNP indicates the fraction of benign files correctly classified. To check the efficiency, the only accuracy is not sufficient as it doesn't reveal the entire statistics. The highest accuracy achieved by the model was 91.9%. It can be observed from the data in the above table 2 that TPP fluctuated for different values of N but is highest at N=110. In this case, the model was able to correctly identify 24 malware samples out of 27 randomly selected. Also after N=130, the value of TPP decreased. Interestingly, at N=130, although the TPP is 84.3%, the FPP and TNP values 0 and 1 indicate that the model was able to correctly classify all the benign samples. One benefit of this is that it will allow the execution of trusted applications without interruptions. Sometimes, in terms of process and network activities, kernel and memory accesses, benign files behave similarly to malware files. It was observed that the high values of FNP and FPP were because of this resemblance, which eventually contributed to the misclassification of files. However, the value of FPP and FNP is the lowest at N=130 when compared to other values of N. From Fig 3, it can be deduced that the poor performance of the model for values of N lesser than 130 is because of the exclusion of some high-frequency information-rich links. Additionally, higher values of N provides excessive data, which leads to a generalisation of data and results in decreased efficiency. Hence it can be concluded that although the model efficiently classified malware at N=110, the accuracy was highest at N=130 and provided overall optimal classification results.

Table 2. Classification Efficiency(Percentage)

No. of features (N)	TPP	FPP	TNP	FNP
100	81.8	3.4	96.5	18.1
110	88.8	11.4	88.5	11.1
120	85.7	5.8	94.1	14.4
125	81.8	6.6	93.3	18.1
130	84.3	0	1	15.6
150	82.1	11.7	88.2	17.8
188	77.7	17.1	82.8	22.2
200	80.7	16.6	83.3	19.2

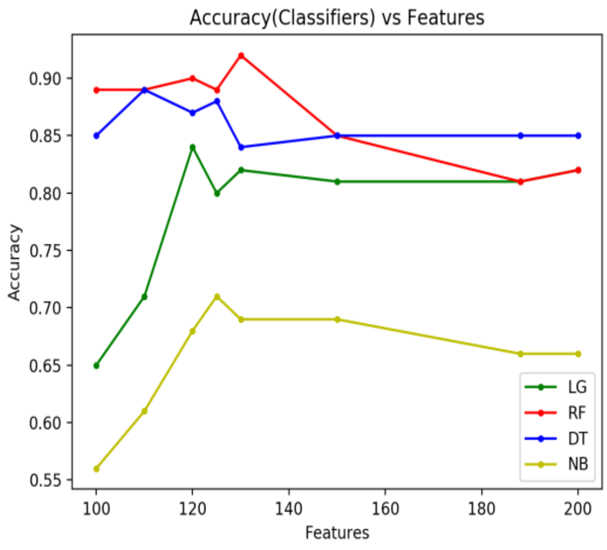


Fig 3: Accuracy(Classifiers) vs Features

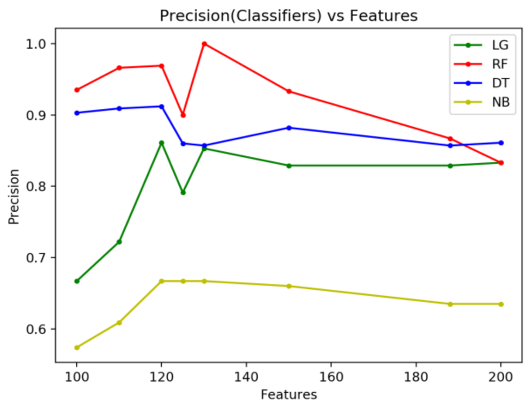


Fig 4: Precision(Classifiers) vs Features

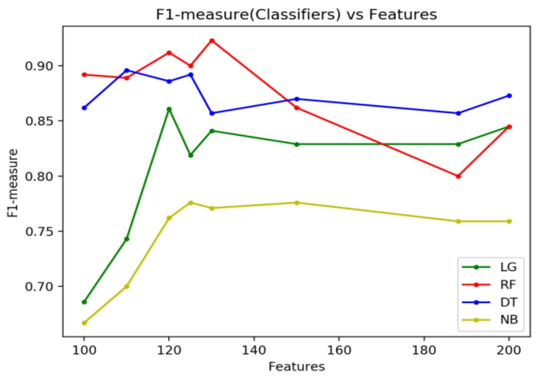


Fig 5: F1-measure(Classifiers) vs Features

For a malware detection model, FPP and FNP denote the number of malware files being predicted as benign and benign files as malware respectively. In this model malware files are considered as positive and benign as negative. Precision represents the fraction of files correctly identified as positive out of all positive files. From Fig 4, it can be inferred that with the increasing value of N, precision also increased and achieved the highest value at N=130. After this point precision decreased substantially and remained almost flat after N=188. A similar trend can also be observed in F1-measure(Fig 5). F1-measure is an important metric for a malware detection model as it considers FPP and FNP for evaluation. And in the case of this model F1-measure metric proves more valuable than accuracy. The proposed model was able to achieve the highest value of F1-measure of 92.3% at N=130, which shows the effectiveness of the model in classifying malware.

Table 3. Classification Accuracy(N=130)

Algorithms	**Accuracy**	**F1-measure**
Logistic Regression	81.8	84.1
RandomForest	91.9	92.3
Decision Tree	83.8	85.7
Naïve Bayes	68.9	77.1

After the branching of features was done, the same dataset was fed to different classification algorithms to check the efficiency. As we were nearing an optimal figure of the number of features selected for some algorithms the accuracy, precision and F1-measure fluctuated. The RandomForest algorithm showed the best overall classification results. However, the efficiency of some algorithms plummeted after features selection number exceeded 130 or were nearing its limit. And became almost flat after 188 for almost all algorithms.

6. CONCLUSION

In this paper, they have focused on the dynamic behaviour of malware and used machine learning algorithms to build a dataset, and train the model. Their proposed method helps them to identify zero-day malicious executable binaries which are most difficult to identify due to polymorphism or metamorphism. To perform this experiment they ran a large set of binaries through emulators and Windows XP(although Microsoft has stopped providing official support to it now) to generate an execution trace. These system-calls are then converted to numerals and finally as nodes or vertices to generate Ordered System-call Graph(SSCG). Average Probability metric(APM) was used then to select semantically relevant paths with high chances of occurring. These paths comprehensively constitute the average behaviour of executable binaries. In addition to the random forest classifier, few others were used to compare the accuracy, F-measure and precision. Random forest yielded the best results with high accuracy and F1-measure after optimal feature selection was applied to the dataset. On average it was observed that their mechanism was more efficient than existing approaches.

Future work will focus on using a large set of executable binaries with a proper balance of malware and benign files to avoid bias, if any, designing of a new path computation algorithm to reduce the computation time, and introduce an additional state change of the OS attribute in the feature selection process to yield higher accuracy.

7. REFERENCES

[1] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in *Proc. of IEEE Symposium on Security and Privacy (SP'10)*, 2005, pp. 32–46.

[2] Sundarkumar, G.G., Ravi, V., Nwogu, I. and Govindaraju, V., 2015, August. Malware detection via API calls, topic models and machine learning. In *2015 IEEE International Conference on Automation Science and Engineering (CASE)* (pp. 1212-1217). IEEE.

[3] A. Sami, H. Rahimi, and B. Yadegari, "Malware detection by behavioural sequential patterns," *Comput. Fraud and Secur.*, vol. 2013, no. 8, pp. 11 – 19, 2013.

[4] Hofmeyr, S. A., Forrest, S., & Somayaji, A. (1998). Intrusion detection using sequences of system calls. *Journal of Computer Security, 6*, 151–180.

[5] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "Accessminer: Using system-centric models for malware protection," in *Proc. of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, 2010, pp. 399–412.

[6] S. Naval, V. Laxmi, M. Rajarajan, M. S. Gaur and M. Conti, "Employing Program Semantics for Malware Detection," in IEEE Transactions on Information Forensics and Security, vol. 10, no. 12, pp. 2591-2604,

Dec. 2015.

[7] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan "Synthesizing near-optimal malware specifications from suspicious behaviours," in *Proc. of IEEE Symposium on Security and Privacy (SP'10)*, 2010, pp. 45–60.

[8] G. Jacob, R. Hund, C. Kruegel, and T. Holz, "Jackstraws: Picking command and control connections from bot traffic," in *Proc. of the 20th USENIX Conference on Security (SEC'11)*, 2011, pp. 29–48.

[9] D. Quist and L. Liebrock, "Visualizing compiled executables for malware analysis," in *Proc. of 6th International Workshop on Visualization for Cyber Security (VizSec'09)*, Oct 2009, pp. 27–32.

[10] J. R. Norris, *Markov Chains*. Cambridge University Press, 1998.

[11] M. J. Quinn and N. Deo, "Parallel graph algorithms," *ACM Comput. Surv.*, vol. 16, no. 3, pp. 319–348, Sep. 1984.

[12] A. Cabrera and R. A. Calix, "On the Anatomy of the Dynamic Behavior of Polymorphic Viruses," 2016 International Conference on Collaboration Technologies and Systems (CTS), Orlando, FL, 2016, pp. 424-429.

[13] T. Fawcett, "An introduction to ROC analysis," *Pattern Recognition Lett.*, vol. 27, no. 8, pp. 861–874, 2006.