Automated malware detection using artifacts in forensic memory images

Rayan Mosli*† Rui Li* Bo Yuan* Yin Pan*

*College of Computing and Information Science
Rochester Institute of Technology
Rochester, New York
Email:{rhm6501, rxlics, bo.yuan, yin.pan}@rit.edu

†Faculty of Computing and Information Technology
King Abdul-Aziz University
Jeddah, Saudi Arabia

***Abstract*—Malware is one of the greatest and most rapidly growing threats to the digital world. Traditional signature-based detection is no longer adequate to detect new variants and highly targeted malware. Furthermore, dynamic detection is often circumvented with anti-VM and/or anti-debugger techniques. Recently heuristic approaches have been explored to enhance detection accuracy while maintaining the generality of a model to detect unknown malware samples. In this paper, we investigate three feature types extracted from memory images – registry activity, imported libraries, and API function calls. After evaluating the importance of the different features, different machine learning techniques are implemented to compare performances of malware detection using the three feature types, respectively. The highest accuracy achieved was 96%, and was reached using a support vector machine model, fitted on data extracted from registry activity.**

***Index Terms*—Malware, Cuckoo Sandbox, Memory Forensics, Machine Learning**

I. Introduction

The threat of malware to the digital world is continuously growing. According to a report by Mcafee Labs, 40 million new malware samples were introduced in the Mcafee Lab malware zoo in the first quarter of 2015, bringing the total number of malware to a staggering 400 million samples as of that quarter [1]. Traditionally, malware detection depended on signatures found in samples and stored in databases. However, as extracting those samples is labor intensive, traditional signature scanning techniques are struggling to keep up with the rapid growth of malware we are seeing today. Furthermore, signature scanning only works with known malware samples whose signatures were already extracted, and is therefore ineffective against unknown malware. Another approach is detecting malware according to their behavior. This approach involves running the samples and observing their run-time actions. Although it improves the detection of unknown malware, the approach is obstructed by samples that employ virtual-machine evasion techniques. Additionally, running each suspected specimen is resource intensive in both time and computing resources. As a result of the weaknesses observed in the two detection techniques, researches resorted to heuristic approaches that employ machine learning models, trained to learn malware features in order to enhance both malware detection accuracy and speed [2] [3] [4] [5] [6].

A recent venue in the realm of malware detection is memory images. Memory forensics is the branch of computer forensics that analyzes a computers memory dump in order to extract evidence of malicious activity. It is gaining popularity in the forensics community due to the wealth of information found in images, which, when compared to the size of disks, are significantly smaller [7] [8] [9] [10] [11]. It is especially beneficial when a threat incorporates stealth measures against disk forensics. As stated by Ligh, "Evidence is more likely to exist on memory then on disk because all malicious code must be loaded into memory to execute" [12]. There are two main stages in performing memory forensics. The first stage is the acquisition of memory, and it involves acquiring a memory image through the use of acquisition tools such as Memoryze [13] and WinPMEM [14]. A usable memory image is also produced in the form of crash dumps when a system crash occurs. However, crash dumps contain less information in the image when compared to raw dumps. After an image is acquired, it is analyzed in the second stage of memory forensics using tools such as Volatility [15] and Rekall [16].

The current challenge in memory forensics is the lack of automation. Yara rules is the only option for automated malware detection in memory images [17]. However, similar to signature detection, it requires the rules for a sample to be included in order to be able to detect that sample. To automate malware detection in memory images, we propose a heuristic approach based on malware artifacts found in memory images. In this paper, we discuss three artifacts that we analyzed and built models for, achieving an accuracy of up to 96%. The artifacts we explored are registry keys modified by the malware, imported DLLs, and called API functions. Although other malware-detecting artifacts exist, such as network activity and metadata, they either do not comprehensively exist in memory, or are easily manipulated by malware. There are several benefits to detecting malware in a memory image. First, it is more robust against anti-forensic techniques employed by malware. According to the rootkit paradox [18], rootkits must be active in order to fight back when being investigated. As memory images represent a frozen state of a live system, rootkits are inactive, thus unable

978-1-5090-0770-7/16/$31.00 ©2016 IEEE

to tamper with an investigation. Another benefit is the large number of candidate artifacts in a memory image. As images represent a live system, numerous artifacts can be extracted and used to build a model. Finally, with the advancements in memory forensic technologies, in both acquisition and analysis, extracting artifacts is becoming simpler and more reliable, paving the way for machine learning models that enhance both the accuracy and speed of malware detection.

The rest of the paper is organized as follows: Section II presents the related work; our methodology is set forth in section III; analysis and results are presented in section IV; and, finally, our conclusion and future work are discussed in section V and VI, respectively.

II. Related Work

Current research in heuristic malware detection are based on two types of malware features, namely static features, such as binary n-grams and opcode sequences, and dynamic features, such as DNS requests, accessed files, and modified registries. As memory images contain both types of features, as well as residual artifacts from malware operations, it is important to review works concerning both feature types.

Santos et al. used term frequency to extract opcode sequences from malware samples, and, building a Support Vector Machine (SVM) classifier based on a normalized polynomial kernel, they achieved an accuracy of 95.9% [19]. Markel et al., on the other hand, focused on metadata found in PE file headers to build their machine learning model, in which they achieved a 97.9% accuracy with a Decision Tree classifier [20]. Masud et al. combined binary n-grams, opcode sequences, and DLL function calls in a hybrid feature set acquired with hybrid feature retrieval techniques [21]. Saxe et al. used PE imports, PE metadata, and byte entropy to build a deep neural network classifier with dropout method to avoid overfitting [22]. They were able to achieve a 95% accuracy with their model.

Although using static features has its advantages, it is still vulnerable to packed and encrypted samples. As pointed out in [23], packing, encryption, k-ary code, and multistage loaders are challenges to the static feature-based machine learning approaches. However, applying a machine learning model on static features extracted from a memory image is less vulnerable to encryption, as samples must decrypt in memory before running.

Dynamic features were also explored in the literature. Pirscoveanu et al. used DNS requests, accessed files, mutexes, registry keys, and API calls to build a random forest classifier [2]. They acquired the data using Cuckoo Sandbox [24] on virtual machines running INetSim to simulate an internet connection. They achieved 96% accuracy with a weighted average of 89.9% across all malware types. Using virtual machines to analyze malware could be vulnerable to anti-VM technology, and, unless each sample was guaranteed to behave naturally, the data might be skewed towards benign behavior.

AMAL is a behavioral-based malware analysis and classification solution that characterizes samples according to filesystem, registry, network activity, and memory behavioral artifacts [5]. However, the memory portion of the solution depends on Yara rules, so it is not capable of detecting unknown malware in memory images. The system consists of two subsystems, one to run samples and extract features, called AutoMal, and the other to vectorize the extracted features and build the classifier, called MaLabel. According to the authors, AMAL achieves an accuracy of 99.5%.

Using both feature types simultaneously was also explored. Lu et al. used both content-based and behavioral-based features, in addition to multiple classifiers, to improve performance [25]. Association rules were combined with an SVM classifier to reduce false positives. Furthermore, other classifiers, such as decision trees, neural networks, naive bayes, and K-Nearest Neighbor, were also tested, along with ensemble learning, which includes voting, decision tree bagging, decision tree boosting, stacking, and grading.

Memory structures were explored in [26], specifically, Virtual Address Descriptors (VADs), mapped files, and registry. Several machine learning models were tested, with Naive Bayes performing best of all with an accuracy of 98%. In the case of VAD structures, due to their vulnerability to Direct Kernel Object Manipulation (DKOM) attacks, using them as a feature might not be robust in a hostile environment.

Finally, a system for identifying rootkit samples, based on automated analysis by capturing changes to data structures and memory regions commonly targeted by malware, was proposed in [27]. The system requires the sample to be run in a sandbox to decide on its maliciousness. The memory changes are detected through differential analysis between a clean baseline snapshot and an infected image. The malware sandbox tool, Cuckoo, was used to examine kernel structures, such as drivers, modules, SSDT, IDT, and callbacks.

Our approach, on the other hand, extracts artifacts from memory because it offers several advantages. First, encrypted and packed malware must first decrypt or unpack in memory before running, thereby bypassing anti-static analysis techniques employed by malware. Second, memory contains both behavioral and content-based artifacts, thus enabling the extraction of artifacts without using a virtual environment, overcoming anti-dynamic analysis techniques. Finally, memory-based detection can also detect memory-only malware, which does not have any presence on hard-drives.

III. Methodology

Our approach is a reactive one, in that it detects malware that is already on the system. There are several use cases for such an approach. First, anti-virus software will fail with unknown malware; therefore, applying our model on a memory image will either confirm or dispute an AV's claim of a clean system. Second, forensic investigators may find this approach useful when searching for malware in memory images. As stated in [28], "Analysts of the real world are often constrained by budgets and billable hours"; therefore, automating the process of malware detection might save forensic investigators time when analyzing memory images. Once the models are trained, artifacts can be extracted from a memory image using

978-1-5090-0770-7/16/$31.00 ©2016 IEEE

Volatility [15], and used to apply the model on. The following subsections discuss our process, starting from data acquisition to model training:

A. Data Acquisition

Our data acquisition environment consisted of a machine running Ubuntu and a target system running Windows 7. The Ubuntu machine was responsible for running the necessary components, Cuckoo sandbox [24], Fog [29], and InetSim [30], for automating the malware analysis and data acquisition process. The Windows machine was the target system infected by malware samples obtained from VirusShare [31] and VX-Heaven [32]. We analyzed 400 malware samples and 100 benign software. Cuckoo was used to automate the analysis of the samples, which were used to infect a bare-metal machine. Our choice of using a bare-metal system was intended to avoid the anti-VM techniques employed by malware.

To ensure the successful execution of the malware samples, several components were set in place. First, as many malware samples check for internet connectivity before running, we ran InetSim to simulate an internet connection. Furthermore, as malware samples come in different extensions, we installed software to run the most common extensions on the target system, such as Adobe Reader and Google Chrome. We also disabled default Windows security mechanisms, such as Windows Defender, User Account Control (UAC), and the firewall. Moreover, Fog is used to revert the infected machine to a clean state after running a malware sample. Finally, as Cuckoo by default does not handle Blue Screens of Death (BSOD). BSODs are caused by bugs in device drivers, a component common to kernel-level rootkits. Therefore, to accommodate rootkits in our data-set, we had to modify the Cuckoo source code and make it more robust against BSODs. Cuckoo sandbox produces a report in the form of a JSON file of both dynamic activities and static features. Furthermore, Cuckoo scans each sample using VirusTotal [33] and includes the results in the report. We used the Python JSON library to extract the data from the report and write it to a text file. Doing so with the reports of both the malware samples and the benign software, our data-set was ready for the next stage.

B. Feature Extraction

The feature extraction and selection stages, in addition to model training, was performed using Scikit Learn [34], the Python machine learning API. We assigned labels to the documents to differentiate between the benign and the malicious ones. Furthermore, we used cross-validation to determine the model complexity, and used the test data to evaluate and compare the performances. The data we chose to use for our model consist of registry activity, API calls, and DLL imports, all of which are in the form of text. Therefore, before training our model, we first had to transform the data into feature measurements. To do so, we chose to implement the Term Frequency (TF) approach, which assigns each term a unique ID, and counts the occurrence of the term in a each document. Each individual term's occurrence frequency is considered a feature. By creating a feature vector for each document, where terms not in the document are represented by zeros, and combining all the vectors into a matrix, we get a matrix that represents all the features in our data-set. Applying the term frequency approach to our data-set, we obtained 1,642 DLL features, 19,532 registry features, and 38,096 API function call features.

C. Feature Selection

The number of features produced by the feature extraction stage must be reduced before training the model to avoid overfitting. To do so, we performed Term Frequency - Inverse Document Frequency (TF-IDF). TF-IDF reweighs the feature set, giving higher weights to terms that occur less frequently across all the documents. Features that occur in the majority of the documents provide less information about the sample, and are therefore safer to remove from the data set. The IDF of a term *t* is calculated as following [35]:

$$idf_t = log\frac{N}{df_t} \tag{1}$$

Where df_t is the number of samples containing the term *t*, and N is the total number of samples. After the IDF score of a term is calculated, the weight is assigned to the term by multiplying the term frequency by the IDF score. By assigning each term in the corpus a weight, we created a means to select a subset of the best features. However, deciding on the optimal number of features is influenced by the model currently being trained. Therefore, we utilized the pipeline and exhaustive grid search functions in Scikit-Learn [34] to test a different number of features with each machine learning model we trained, thus finding the optimal number of features to select when training a specific model.

D. Model Training

As mentioned in the previous section, we utilized the pipeline and exhaustive grid search functions in Scikit Learn for feature selection and model training. The pipeline function groups several pre-processing steps along with the final step of training a model. The exhaustive grid search, on the other hand, provides a mechanism for searching a parameter space for the optimal parameters for a given pipeline. Parameters, in this context, refers to hyperparameters. Moreover, the exhaustive grid search allows specifying the evaluation metric to be used when testing different parameters. Classification metrics that can be set in the exhaustive grid search function include accuracy, Area Under the Receiver Operating Characteristic Curve (AUROC), and average precision.

We trained and compared the following techniques for classification for all three feature types:

- Support Vector Machine
- Stochastic Gradient Descent on several loss functions
- Decision Tree
- Random Forest
- K-Nearest Neighbor
- Bernoulli Naive Bayesian

978-1-5090-0770-7/16/$31.00 ©2016 IEEE

- Multinomial Naive Bayesian

The following is a brief description of each model, along with their respective hyperparameters:

1) Support Vector Machine: Support Vector Machines (SVMs) are supervised machine learning models that are trained on feature vectors representing data points in space. It separates the data points from different classes with a hyperplane that is made as wide as possible. New data points are classified based on the side of the hyperplane they fall on. The first parameter we need to determine when training the SVM is the kernel type, which comes in the form of a linear function, a polynomial function, a radial basis function, or a sigmoid function. In the case of polynomial kernels, different degrees of polynomials were also tested. Finally, several parameter values for the penalizing term needed to be determined to find the optimal penalty value.

2) Stochastic Gradient Descent: Stochastic Gradient Descent (SGD) is an optimization method used to minimize a target loss function. SGD works by minimizing the error of the underlying classifier through model parameter tuning. In Scikit Learn, SGD can be used to optimize hinge loss functions which produce linear SVMs, log loss functions for logistic regression, modified Huber loss functions, squared hinge loss functions, and the linear loss function used by the perceptron algorithm. In optimizing the aforementioned loss functions, several parameters could be modified to produce a better classification performance. Those parameters include the regularization term and the alpha constant that multiplies the regularization term to indicate its severity.

3) Decision Tree and Random Forest: Decision tree classifiers map features to class labels, where tree interior nodes represent features and tree leaves represent class labels. A tree is learnt by recursively splitting sets of features into smaller subsets until each subset represents a class label. When a new data point is to be classified, it starts at the root node and follows a path down to one of the leaves, according to which features best represents the data point. There are several metrics that can be used in the splitting process of creating the decision tree. Gini impurity is a metric that measures the probability of misclassification, and accordingly splits to minimize the impurity. Splitting is stopped when the impurity reaches zero, or in other words, when nodes map to a single class. Another metric is information gain, which uses the entropy of the parent minus the weighted sum of the children's entropy to decide on the split.

Random forest is an ensemble learning method that utilizes multiple decision trees to improve classification. The classification decision is made by choosing the class produced by the majority of the decision trees. Choosing the number of trees in the forest is a hyperparameter we tested using exhaustive grid search.

4) K-Nearest Neighbor: One of the simplest machine learning models, K-Nearest Neighbor (KNN) classifies new data points according to the majority class of the *k* nearest neighbors to the data point. The value of *k* is used as an input to KNN and can be modified to produce more accurate classifiers. Furthermore, weights can be assigned to the neighbors, where closer neighbors have higher weights. Another approach is to uniformly assign weights to all the neighbors. Methods to find the nearest neighbors include kd-trees, ball trees, and brute force search.

5) Bernoulli Naive Bayesian and Multinomial Naive Bayesian: Bernoulli Naive Bayesian (BernoulliNB) and Multinomial Naive Bayesian (MultinomialNB) are two similar classifiers with subtle differences. BernoulliNB is used to model data that follows the Bernoulli distribution, where features are required to have a binary value. Considering the textual nature of our dataset, the data can be binarized to represent the presence or absence of features, thereby meeting the prerequisite of a BernoulliNB classifier. MultinomialNB, on the other hand, applies Bayes' theorem on multinomially distributed data with frequency of occurrence as features. Both BernoulliNB and MultinomialNB apply Bayes' theorem with the assumption that the features are independent; thus, the term Naive. However, the difference between the two is the penalization of absent features by BernoulliNB, which is otherwise ignored by MultinomialNB. The binarization of input data to BerounlliNB can be done either before training the classifier, or by setting a hyperparameter to specify a threshold that would be used to map features values to binary representation. Both options were tested in training the BernoulliNB model.

IV. Analysis and Results

As previously mentioned, applying the exhaustive grid search on a pipeline can be done using several evaluation metrics. For all three feature types, we searched for the models, and their respective hyperparameters, that produced the best accuracy and AUROC scores separately. The accuracy of the model is calculated by using the following equation:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2}$$

Where TP are true positives, TN are true negatives, FP are false positives, and FN are false negatives. Furthermore, AUROC is calculated by first plotting the true positives over all positives against the true negatives over all negatives, and then calculating the area under the produced curve. Tables 1 and 2 summarize the performance of all the models in terms of accuracy and AUROC scores, respectively. The following is our findings of the search:

A. Registry

When calculating the term frequency of malware registry activity, we found that the most frequent terms are related to tracing, which is used for network troubleshooting and remote access services. This suggests that the malware is preparing the environment for data exfiltration and for remote access by the malware developers. Furthermore, log file sizes of tracing operations were modified through registry values to minimize the footprint of the malware operations. Access to Microsoft Explorer's security zone map settings were also included among the malware's registry activities.

978-1-5090-0770-7/16/$31.00 ©2016 IEEE

In regards to model training, we found that SGD on a hinge loss function performed best in terms of both accuracy and AUROC, although with different parameters. Only 200 features were sufficient to achieve an accuracy of 96%; however, for lowest false positives, 1000 features were required. Furthermore, to obtain the best AUROC score, the data was transformed to binary representation. The highest AUROC score achieved was 0.983.

B. DLLs

There are a limited number of DLLs that can be loaded by software, and that are not custom built libraries. Therefore, the classifiers trained on DLL features depend on the different combinations of loaded libraries in order to discriminate between malicious and benign software. There were 64 different DLLs loaded by the malware portion of our dataset. 22 of those libraries were only loaded by a single sample, which indicates a custom library. The remaining 42 libraries produce 4.3980465×10^{12} different combinations of DLLs to load.
Random forest classifier produced the best results when trained on DLL features. The best accuracy of 90.5% was achieved when Gini Impurity was used as a criterion to split. Furthermore, 15 trees were used to create the forest, using 500 features. For the AUROC, on the other hand, 10 trees were used to create the forest, with entropy as a criterion to split. 200 features were sufficient to obtain the best AURC score of 0.922.

C. APIs

Among the most occurring APIs in the malware dataset is the sleep API. This API is often used by malware to evade automated analysis systems [36]. Luckily, Cuckoo has a feature that overcomes sleep cycles. Another frequent API is the gettickcount API, which is used to detect debuggers. As debuggers often place breakpoints when executing software, they significantly slow down runtime. Therefore, malware utilize the gettickcount API to determine if a debugger is present. Overall, numerous anti-analysis APIs were seen in the malware dataset; these occurrences help enhance the discrimination between malicious and benign software, as anti-analysis is not as common in legitimate software.
SGD on a hinge loss functions provided the best results in terms of accuracy, after reducing the number of features from 38,096 to 11,000. On the other hand, the highest AUROC score was achieved when using SGD on a log loss function without any feature reduction. The highest accuracy and AUROC scores are 93% and 0.958, respectively.

TABLE I
SUMMARY OF ACCURACY SCORES

Classifiers	Registry	DLLs	APIs
SVM	94.4	88.7	92.3
SGD	96	87.8	93
Random Forest	94.9	90.5	91.5
Decision Tree	94.9	88.7	90.7
KNN	93.9	89	90.7
BernoulliNB	93.4	89.6	89.2
MultinomialNB	92.9	85.7	89.7

TABLE II
SUMMARY OF AUROC SCORES

Classifiers	Registry	DLLs	APIs
SVM	0.975	0.919	0.955
SGD	0.983	0.903	0.958
Random Forest	0.969	0.922	0.948
Decision Tree	0.954	0.861	0.877
KNN	0.969	0.903	0.93
BernoulliNB	0.972	0.915	0.906
MultinomialNB	0.968	0.897	0.922

V. CONCLUSION

Malware is one of the greatest and most rapidly growing threats to the digital world. With approximately 40 million new samples each year, traditional detection approaches are struggling to keep up. Researchers have resorted to heuristic approaches to enhance detection rates, and to be able to generalize to unknown malware samples. In this paper, we demonstrated how memory images can be used as a venue for heuristic malware detection. Furthermore, we explored three malware features that can be extracted from a memory image and used to detect the presence of malware on a system.

VI. FUTURE WORK

The road ahead includes exploring more memory artifacts that are commonly targeted by malware. Specifically, we plan to explore kernel structures, such as the System Service Dispatch Table (SSDT), Interrupt Descriptor Table (IDT), Global Descriptor Table (GDT), and kernel drivers. In order to successfully capture malware modification to kernel artifacts, we must employ a trigger-based acquisition system, as proposed by Teller [37]. After we determine the best performing techniques that best fit each feature type, we plan to merge those features and create a multi-layer model that both enhances accuracy and minimizes false positives.

REFERENCES

[1] McAfee Labs, "Threat report," 2015. [Online]. Available: http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q1-2015.pdf

[2] R. Pirscoveanu, S. Hansen, and A. Czech, "Analysis of malware behavior: Type classification using machine learning," in *Proceedings of the 2015 International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA)*, 2015.

[3] M. Z. Mas'ud, S. Sahib, M. F. Abdollah, S. R. Selamat, and R. Yusof, "Analysis of features selection and machine learning classifier in android malware detection," in *Proceedings of the International Conference on Information Science & Applications*, 2014.

[4] K. Berlin, D. Slater, and J. Saxe, "Malicious behavior detection using windows audit logs."

[5] A. Mohaisen, O. Alrawi, and M. Mohaisen, "Amal: High-fidelity, behavior-based automated malware analysis and classification," *Computers & Security*, 2015.

[6] Y.-D. Lin, Y.-C. Lai, C.-N. Lu, P.-K. Hsu, and C.-Y. Lee, "Three-phase behavior-based detection and classification of known and unknown malware," *Security and Communication Networks*, 2015.

978-1-5090-0770-7/16/$31.00 ©2016 IEEE

[7] J. Stuttgen and M. Cohen, "Anti-forensic resilient memory acquisition," *Digital Investigation*, pp. 105–115, 2013.
[8] I. Korkin and I. Nesterov, "Applying memory forensics to rootkit detection," in *Proceedings of the 9th Annual ADFSL 2014 Conference on Digital Forensics*, 2014, pp. 115–141.
[9] S. Vomel and H. Lenz, "Visualizing indicators of rootkit infections in memory forensics," in *Proceedings of 2013 Seventh International Conference on IT Security Incident Management and IT Forensics*, 2013, pp. 122–139.
[10] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin, "Robust signatures for kernel data structures," in *Proceedings of 16th ACM conference on Computer and communications security*, 2009, pp. 566–577.
[11] H. Pomeranz, "Detecting malware with memory forensics," 2013.
[12] M. Ligh, A. Case, J. Levy, and A. Walters, *The Art of Memory Forensics*. Wiley, 2014.
[13] Mandiant, "Memoryze," 2011. [Online]. Available: https://www.fireeye.com/services/freeware/memoryze.html
[14] M. Cohen. (2012) Winpmem. [Online]. Available: https://github.com/google/rekall/tree/master/tools/windows/winpmem
[15] Volatility, "Volatility framework," 2012. [Online]. Available: https://code.google.com/p/volatility/
[16] Google, "Rekall," 2013. [Online]. Available: http://www.rekall-forensic.com/
[17] V. M. Alvarez. The pattern matching swiss knife for malware researchers (and everyone else). [Online]. Available: https://plusvic.github.io/yara/
[18] J. Kornblum, "Exploiting the rootkit paradox with windows memory analysis," *International Journal of Digital Evidence*, 2006.
[19] I. Santos, F. Brezo, X. Ugarte-Pedrero, and P. Bringas, "Opcode sequences as representation of executables for data-mining-based unknown malware detection," *Information Sciences*, pp. 64–82, 2013.
[20] Z. Markel and M. Bilzor, "Building a machine learning classifier for malware detection," *2014 Second Workshop on Anti-malware Testing Research (WATeR)*, 2014.
[21] M. Masud, L. Khan, and B. Thuraisingham, *Data Mining for Detecting Malicious Executables*, 2011, pp. 109–147.
[22] J. Saxe and K. Berlin, "Deep neural network based malware detection using two dimensional binary program features," in *Proceedings of the 10th International Conference on Malicious and Unwanted Software (MALWARE)*, 2015.
[23] H. Nath and B. Mehtre, "Static malware analysis using machine learning methods," *Communications in Computer and Information Science Recent Trends in Computer Networks and Distributed Systems Security*, 2014.
[24] Cuckoo Foundation. (2014) Automated malware analysis - cuckoo sandbox. [Online]. Available: https://www.cuckoosandbox.org/
[25] Y.-B. Lu, S.-C. Din, C.-F. Zheng, and B.-J. Gao, "Using multi-feature and classifier ensembles to improve malware detection," *Journal of Chung Cheng Institute of Technology*, 2010.
[26] M. Aghaeikheirabady, S. Farshchi, M. Iran, and H. Shirazi, "A new approach to malware detection by comparative analysis of data structures in a memory image," in *Proceedings of the 1st International Congress on Technology, Communication and Knowledge*, 2014.
[27] A. Zaki and B. Humphrey, "Unveiling the kernel: Rootkit discovery using selective automated kernel memory differencing," in *Proceedings of the 2014 VIRUS BULLETIN CONFERENCE*, 2014.
[28] B. Blunden, *The rootkit arsenal escape and evasion in the dark corners of the system, second edition*. Jones & Bartlett Learning, 2013.
[29] C. Syperski and J. Zhang. (2015) The fog project. [Online]. Available: https://fogproject.org/
[30] T. Hungenberg and M. Eckert. (2007) Inetsim: Internet services simulation suite. Accessed: 11 10 2015. [Online]. Available: http://www.inetsim.org
[31] Roberts, J-Michael. Virusshare. [Online]. Available: https://virusshare.com/
[32] Vx heaven. [Online]. Available: http://vxheaven.org/
[33] Virustotal. Accessed: 11 15 2015. [Online]. Available: https://www.virustotal.com/
[34] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108–122.
[35] C. Manning, P. Raghavan, and H. Schutze, *An Introduction to Information Retrieval*. Camridge University Press, 2008, ch. 6. Scoring, term weighting and the vector space model, pp. 117 – 120.
[36] A. Lakhani. (2015, 5) Malware sandbox and breach detection evasion techniques. Accessed: 9 23 2015. [Online]. Available: http://www.drchaos.com/malware-sandbox-and-breach-detection-evasion-techniques/
[37] T. Teller and A. Hayon, "Enhancing automated malware analysis machines with memory analysis," 2014.

978-1-5090-0770-7/16/$31.00 ©2016 IEEE