Machine Learning for Attack Vector Identification in Malicious Source Code

Victor A. Benjamin, Hsinchun Chen
Department of Management Information Systems
The University of Arizona
Tucson, AZ 85721, USA
vabenji@email.arizona.edu, hchen@eller.arizona.edu

***Abstract*— As computers and information technologies become ubiquitous throughout society, the security of our networks and information technologies is a growing concern. As a result, many researchers have become interested in the security domain. Among them, there is growing interest in observing hacker communities for early detection of developing security threats and trends. Research in this area has often reported hackers openly sharing cybercriminal assets and knowledge with one another. In particular, the sharing of raw malware source code files has been documented in past work. Unfortunately, malware code documentation appears often times to be missing, incomplete, or written in a language foreign to researchers. Thus, analysis of such source files embedded within hacker communities has been limited. Here we utilize a subset of popular machine learning methodologies for the automated analysis of malware source code files. Specifically, we explore genetic algorithms to resolve questions related to feature selection within the context of malware analysis. Next, we utilize two common classification algorithms to test selected features for identification of malware attack vectors. Results suggest promising direction in utilizing such techniques to help with the automated analysis of malware source code.**

Keywords - Cyber security, Cybercrime, Hacker, Malware analysis, Static analysis

I. INTRODUCTION

With computers and Internet-based devices becoming more ubiquitous throughout society, the security of our networks and information technologies is a growing concern. Recent years have witnessed more sophisticated malware spreading itself to unsuspecting individuals, while high-profile hacking incidents have disrupted various government and industry IT resources. As the need for more cyber security research is evident, more researchers have taken interest in exploring related projects.

In particular, one area of growing interest is the exploration of underground hacker web communities. Often times, security researchers can discover new security threats, trends, and hacker behaviors by observing such communities [1,2,3]. Existing studies in this area vary, often having unique approaches and intentions for studying hacker communities. Popular examples include content analyses, social network analyses, and malware analyses [4,5,6]. Such studies have explored hacker communities across multiple geopolitical regions, with many reporting observations of hackers sharing various cybercriminal assets and seeking collaboration with one another [2,3,7].

Instances of collaboration and shared resources have been discussed in many studies aiming to learn more about the social behaviors of hackers [3,6]. In particular, source code for various malicious programs and scripts are commonly distributed freely among hackers [2]. However, source code may not always be accompanied by complete documentation, and in many cases, no documentation provided at all. Researchers interested in studying forums across multiple geopolitical regions may often encounter documentation written in unfamiliar languages. Lastly, source code can often times be too obfuscated for humans to interpret clearly, or the programming language used to write some code could exist outside a researcher's skill set. For these reasons, research on malicious source code has been largely limited; the development of automated tools that could help security researchers overcome such limitations would be of great asset. In this study, we explore a subset of popular machine learning methodologies for the automated analysis of malware source code files.

II. LITERATURE REVIEW

To form the basis for this research, literature is reviewed from the following three areas:

- Past research on hacker communities
- Previous work on malware analysis
- Machine learning methodologies

A. Past research on hacker communities

Many past studies have observed that hackers often congregate within virtual communities, commonly in the form of IRC communities and online forum [3,6]. Additionally, researchers have discovered hacker communities to exist across multiple geopolitical regions, most notably the US, China, and Russia [2]. These communities are the target of many security researchers and social scientists wanting to understand more about cybercriminal behaviors and operations.

One key finding observed in multiple studies is the collaborative nature of hackers, as many appear to openly share cybercriminal assets and knowledge with their peers [2,3]. Further research has concluded that the communal behaviors expressed by many community participants are driven by a desire to increase notoriety and status among

peers; reputation appears to be a major social driver in hacker communities [2].

Essentially, individual hackers may share their personal resources for increased social status and opportunity for collaboration to pursue advanced tasks. Thus, many use hacker communities to pool together resources and tools [7]. Specifically, cybercriminal tools, tutorials, services, and raw malware source code are often shared, which can all be used for deeper analysis of hacker communities [2]. In particular, the different variants of malware shared within hacker communities are often of interest to researchers; one popular security-related research stream involves the collection and analysis of various malware, such as those openly shared between hackers.

B. *Malware analysis*

Analysis of malware has traditionally consisted of two methods of analysis – *dynamic* analysis and *static analysis* [8,9]. Dynamic analysis involves executing malware and studying run-time behaviors Static analysis involves analyzing source code or other forms of program instructions without actually executing malware. Both forms of analysis complement each other well and are used widely among security researchers.

Most malware analysis research tends to involve dynamic analysis of malicious binary files, as source files or program instructions are often difficult to retrieve [10,11]. Dynamic analysis involves the execution of malware binaries in sandboxed environments, such as a virtual machine, so that malware execution behaviors can be logged in real-time. Malware can then be classified based on captured behaviors, often times utilizing machine learning for automated analysis [5,12].

Unfortunately, dynamic analysis has many limitations[9, 13]. Analysis is generally limited to one malware at a time, as execution is necessary. Additionally, malware can contain hidden execution logic; for example, a virus may execute a reserved portion of its code only on some specific date.

Conversely, static analysis of malware can resolve many limitations met in dynamic analysis. The full control flow of a program can be assessed, revealing any hidden execution paths and all program logic. Additionally, static analysis does not require sandboxing or the execution of malware; instead, researchers often rely on reverse engineering malware binaries in attempt to retrieve program instructions and potential behaviors [12]. Then, machine learning is often used to automatically analyze collected data [9,13,14]. However, there appears to be little work exploring the static analysis of original source code files, as this type of data has traditionally been more difficult to retrieve than malware binaries [13].

However, recent work has identified that malicious source code can now be found commonly within hacker communities [2]. It is now more viable than in the past to conduct research in ton malware source code. Additionally, as machine learning has been used in current malware research, it provides a natural start for exploring the analysis of malware source code

C. *Machine learning methodologies*

Machine learning classification has been useful for analysis across various disciplines, including multi-lingual sentiment classification, identification of different groups of individuals in social, and the identification botnet network traffic among normal network [15,16,17]. In the context of malware analysis, literature reveals how machine learning has been useful to security researchers in various studies. However, many limitations have been raised in prior work in regards to using machine learning for malware analysis. One limitation is that the selection of meaningful features for classification is a non-trivial task [14]. The number and types of features to use becomes a difficult decision, especially when wanting to achieve optimal machine learning performance. Another limitation is that much current research appears to only take advantage of one- or two- class machine learning techniques [5,14]. The lack of research using multi-class machine learning seems to have prevented in-depth analysis of malware beyond the simple detection of malicious files from the benign.

By reviewing research utilizing machine learning outside of the malware analysis context, potential solutions can be identified to the aforementioned limitations in current research. A review of past work reveals that genetic algorithms (GAs) are useful when optimizing feature selection for use with machine learning algorithms [15,16,18]. Essentially, GAs are automated stochastic search algorithms that can be used to identify optimal values in a vast search space by mimicking the natural process of evolution. Solution sets are generated in generations and incorporate crossover and mutation mechanisms to discourage premature convergence of GAs. In the context of feature selection, the GA will constantly try new combinations of features for classification, while slowly progressing to choosing combinations of features that yield higher classifier accuracy. The pairing of GAs and classifiers is often referred to as the wrapper model, and has been an effective solution for feature selection in many studies [15,18]. This technique could potentially be used to resolve issues with feature selection in the context of malware analysis.

While a GA can solve feature selection limitations, other techniques must be utilized to advance automated classification for malware analysis. A review of popular classification techniques reveals that support vector machines (SVMs) and decision trees such as C4.5 are commonly used in research utilizing machine [19]. These algorithms can be utilized for more advanced analyses of malware; finer-grained classification may be possible by categorizing between more multiple explicit classes. For example, traditional malware analysis research typically only focuses analysis on one language or target platform at a time [5,14]. More advanced classifiers can be built to analyze malware of all different

types simultaneously. Such techniques can be used to create more comprehensive and scalable tools.

III. Research Gaps and Questions

Researchers exploring hacker communities may be limited by the lack of tools available to analyze malicious source code files. Often times, code can simply be too cryptic and difficult to read (Figure 1). Many researchers also simply lack the technical expertise to interpret source code files across multiple programming languages. Lastly, the majority of traditional malware analysis research seems to have focused analysis of malware binaries; little to no work has been done on the numerous types of malware source code researchers will encounter within hacker communities. In this study, we use machine learning to automatically classify the attack vector present in various malware source code files. We utilize the wrapper model for classification purposes, and perform analysis on malware written in three different programming languages.

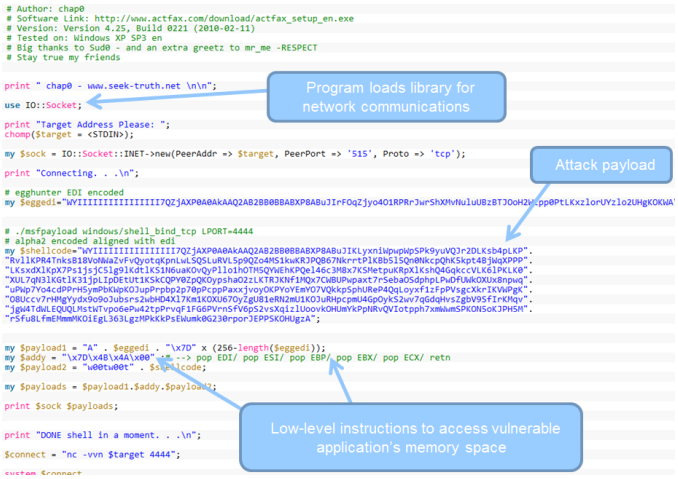

Figure 1 – An example of a Perl exploit that attempts a remote buffer overflow attack on a popular Windows and Unix mailserver software. Malicious code such as this can be difficult for researchers to interpret in their explorations. Automated static analysis tools can help in such scenarios

We pose the following research questions:

1. Can malware source code be categorized using automated means?
2. How can one generate an appropriate feature set for malware source code classification?
3. Can a generalized technique be taken for malware source code files written in various programming languages?

IV. Research Testbed and Design

For this study, we analyze malware source code files collected from an exploit sharing community, the Exploit Database (Figure 2). The Exploit Database claims to have just over 20,000 raw malware source files categorized into one of four attack vectors: local memory attacks, remote code execution attacks, web application exploits, and denial of service attacks. Additionally, malware source files are written in a variety of programming languages and target various platforms.

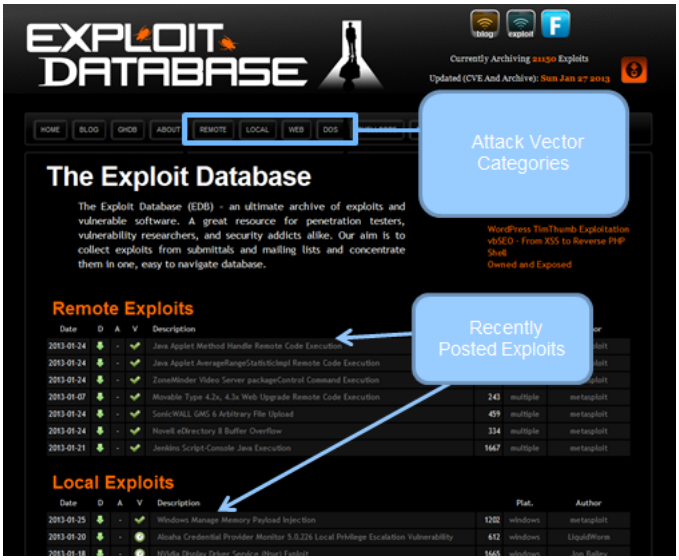

Figure 2 – The Exploit Database contains a comprehensive collection of malicious source code written in various programming languages. Code is categorized within several categories, with the main four types of exploits being local memory and program attacks, remote code execution attacks, web application exploits, and denial of service attacks.

After using a web crawler to collect data, we noticed many of duplicate source files and that contained only text referring viewers to visit other communities for full source code. We remove such instances from our testbed. Additionally, we only focus on code written in three languages for the context of this study. We analyze malware written in Perl, Python, and Ruby; all three are popular languages, while Ruby is especially unique because it hosts a widely adopted a security framework called the Metasploit Framework (MSF). MSF appears to be used by many security professionals and researchers for exploit testing [20]. A summary of our testbed can be found in Table I.

TABLE I. Testbed Summary

Programming Language	Local	Remote	Web Application	Denial of Service	Total
Perl	421	1,172	119	27	1,739
Python	196	260	659	356	1,471
Ruby (MSF)	153	172	116	262	703
Total	770	1,604	894	645	3,933

The contents of each source code file were transformed to a bag of words vector representation of all content per language. However, non-alphanumeric characters are excluded in order to remove language-reserved syntactic characters, such as quotes that commonly encapsulate strings, periods to separate objects from method calls, etc. As these features would occur in every source file, they would not be useful for classification. Additionally, we only retained words that occur more than 10 times within our data set. This is to help avoid

unnecessarily sparse feature sets. We were left with 3,699 Perl features, 2,484 Python features, 5,408 Ruby features, for a total of 11,591 features. The vectors created from these features will be used as inputs for our GA to select.

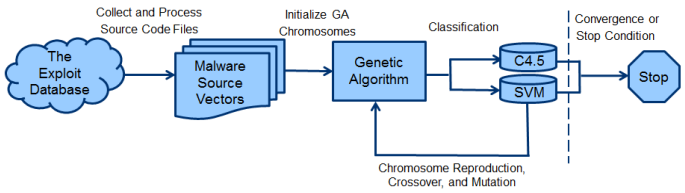

Figure 3 – Our research design. First, we accessed the Exploit Database, and made us of an automated crawler to collect source code files. Then we processed source code files, extracted relevant contents, and produced vectors from which the GA uses to select features. Chromosomes are randomly initialized for the first generation of the GA, and then sent to a classifier for one or two-stage (i.e. hierarchical) classification. The classifier accuracy resulting from chromosome (i.e. feature set) is used to assign probabilities to each chromosome for reproduction into the next generation; chromosomes yielding higher accuracy have greater probability of being selected. After reproduction, chromosomes are randomly chosen for crossover and randomly mutated. The process repeats itself until convergence of chromosomes or a stop condition is met.

Along with a vector from which to select features from, a GA requires several other input parameters for its various stages of computation. We provide pseudo-code summary of our GA in Figure 4, and also summarize the process here. In our study, chromosomes are representations of each language's word vector; if a chromosome bit is activated by the GA, the corresponding feature will be used as a feature for source code classification. The initial generation of chromosomes is randomly generated. Additionally, we use a population size of 50 chromosomes, and run 200 generations of evolution, which is consistent with past research [15]. Each chromosome (feature set) is used to classify exploits with 10-fold validation. We use classification accuracy of each feature set to evaluate the fitness of each chromosome. Thus, we make use of the wrapper model mentioned in reviewed literature. A roulette wheel algorithm is run to randomly choose chromosomes to pass on to a new generation for evaluation. Chromosomes that had a higher fitness core have higher probability to be chosen. After chromosomes are chosen, they are crossed over and then mutated. We use a low mutation rate (2.5%) and high crossover rate (60%) – these values are similar to what has been utilized in past GA research [15,18].

A separate experiment is run for each programming language Prevents any influence one programming language may have on another We first run classification experiments for each language using all available features in our testbed in order to develop a baseline accuracy score After baseline scores are obtained, experiments are conducted using our GA and wrapper model. The baseline scores allow us to measure the performance increase a GA provides For each experiment, we use two different methods for classification, the C.45 algorithm and SVMs. It is common to evaluate multiple classification problems for a given problem context Depending on data characteristics, different techniques may work better Additionally, we run each experiment in two different configurations In one configuration, we just classify each source code file into any of the four attack vector classes.

Algorithm 1. A wrapper model GA

Input: A feature vector for the GA to choose features from; GA parameters such as crossover rate, mutation, rate, chromosome population size, and desired generations to compute are required
Output: Classifier accuracy and an optimized feature set

Let *c* be the crossover rate, *m* be the mutation rate, *p* be the chromosome population size, and *g* to be the amount of generations to compute, *f* to be the amount of features included in the input vector, and let each chromosome bit represent one unique feature. All of these are variables must be specified before the GA can function.

Let *s* of each chromosome be the resulting classifier accuracy when using the feature set represented by a chromosome. This is generated outside of the GA during run-time

```
1: Randomly initialize p chromosomes with f bits
2: for each g do
3:       for each chromosome in g do
4:            Classification returns s
5:       end
6:       Weight each chromosome based on s
7:       for i ← 0 to f do
8:            Randomly select weighted chromosome
9:                 from g to reproduce into g + 1
10:      end
11:      g = g + 1
12:      Crossover chromosomes based on c
13:      Mutate chromosomes based on m
14: end
15: return greatest s value and associated feature set
```

Figure 4 – Pseudo-code for our genetic algorithm. The first generation of chromosomes (i.e. feature sets) is randomly initialized and chromosomes are used for classification. The accuracy yielded from each chromosome is used to create weights for chromosome reproduction. After chromosomes are reproduced, random crossover and mutation modifiers occur to prevent pre-mature convergence of the algorithm, and to ensure the GA searches multiple portions of its total search space. Then, these new chromosomes are used for classification. This process repeats itself until converge or a stop condition is met.

In the second configuration, we test a simple prototype of hierarchical classification of malware source code files. Since our test bed is split between one class of offline exploits and three classes of Network-based attacks, we first classify exploits based on their reliance of networking libraries in their code. Then, if an exploit belongs to the network-based attacks class, we further categorize it into one of the three existing network-based attack vector classes.

V. HYPOTHESES

Our first set of hypotheses is concerned with the effectiveness of classifiers without GA support for automated malware analysis. Specifically, we will run the C4.5 algorithm and SVM using all features we have per language. These baseline results will help us evaluate the effective of the GA for feature selection.

H1: Malware source code files can be accurately classified through automated means that utilize machine learning

> *H1a:* A C4.5 classifier will be an effective automated technique to categorize malware source code files. We will use a stand-alone C4.5 algorithm without GA support for grounding a baseline accuracy
>
> *H1b:* A SVM classifier will be an effective automated technique to categorize malware source code files. We will use a stand-alone SVM algorithm without GA support for grounding baseline accuracy

Our second set of hypotheses seeks to measure the change in accuracy from baselines classification to the wrapper

model. We test the classifiers on all three programming languages while using the GA to assist with feature selection.

H2: Classification results will become more accurate when utilizing the GA and wrapper model to assist in feature selection

H2a: Use of the wrapper model with malware source code files written in Ruby (MSF) will yield better accuracy than baseline classification

H2b: Use of the wrapper model with malware source code files written in Perl will yield significant accuracy will yield better accuracy than baseline classification

H2c: Use of the wrapper model with malware source code files written in Python will yield better accuracy than baseline classification

Our last set of hypotheses focus on the difference in results when attempting to classify malware between our two different experiment configurations. First, we will attempt to classify into each one of the four attack vector classes simultaneously. Then, we will try a hierarchical classification model that first classifies between local-based or network-based attacks, and then further classifies network-based attacks to their specific vector.

H3: Classification results will become more accurate an hierarchical classification

H3a: Classification accuracy between all exploit classes simultaneously (configuration one) will be lower than that of the first stage of hierarchical classification (configuration two).

H3b: Classification accuracy between all exploit classes simultaneously (configuration one) will be lower than that of the second stage of hierarchical classification (configuration two).

V. RESULTS

The results of our wrapper model experiment, as well as the percent gain in accuracy when compared to baseline results, can be viewed in Table II. Overall, the wrapper model provided modest gains. Additionally, it is important to note that the GA constructed feature sets were on average 40% the size of all features available for a given language; this would help researchers conduct analyses of malware much quicker, and needing less powerful hardware (e.g. memory) to perform analysis.

Manual scrutiny of optimized feature sets reveals that programming libraries and method calls appear to be the most useful for classification (Table III). For example, a Python remote code execution attack may import the *socket* library for networking functionality, while a local memory attack would lack such behavior. Bytecode encoding (i.e. \x45, \x32, \x21, etc.) is also represented across a variety of features, and seems to be used frequently in memory-based attacks. Our findings are consistent with dynamic analysis studies focusing on observing malware execution behaviors for classification purposes (Shabtai et al, 2011).

TABLE II. WRAPPER MODEL CLASSIFICATION RESULTS

	Perl	**Python**	**Ruby**
SVM	82.52% (+1.37%)	81.89% (+5.79%)	84.98% (+1.75%)
C4.5	86.35% (+1.59%)	85.41% (+0.91%)	88.84% (+3.09%)
1st Stage SVM	94.27% (+0.80%)	88.03% (+2.11%)	92.21% (+2.26%)
2nd Stage SVM	86.68% (+1.67%)	85.26% (+3.08%)	91.39% (+1.31%)
1st Stage C4.5	95.31% (+0.14%)	94.74% (+3.42%)	94.36% (+0.23%)
2nd Stage C4.5	88.31% (+1.50%)	88.05% (+1.75%)	92.71% (+0.73%)

Parentheses include accuracy increase relative to baseline

Additionally, we scrutinized the misclassified source files to better understand potential reasons for Type I and Type II errors. It appears that the majority of misclassified malware contained elements of multiple attack vectors or utilized custom program libraries instead of mainstream packages. Both of these cases present difficulty for our model, but could potentially be solved with more advanced classification.

TABLE III. EXAMPLES OF TOP FEATURES PER LANGUAGE

Perl	**Python**	**Ruby**
Bytecode Encoding	Bytecode Encoding	Bytecode Encoding
http	Socket	Buffer
Host	Shellcode	Sql
Syswrite	Php	Port
Server	File	url

Overall, the increase in accuracy from the baseline to GA-optimized classification is in similar degree to what was observed in other wrapper model research (Abbasi et al, 2008) The GA increased baseline accuracy on average of 1.86%. The lowest gain was a 0.14% bump when using C4.5 for the first half of two-stage classification; baseline accuracy for this experiment was already very accurate at 95.17% The highest observed boost was 5.79%, and it occurred when pairing a GA with a SVM to classify malware between all four classes simultaneously. The baseline here was only at 76.10%.

VI. DISCUSSION

Both baseline and wrapper model testing resulted in above average classification accuracy supporting *H1* and all its sub-hypotheses. This technique appears to be promising as a method for analyzing malware source codes that could be

found within hacker communities. The C4.5 classifier appeared to be the best performer on our test bed, leading to high classifier accuracy and supporting hypothesis *H1a*. The SVM also led to above average classifier accuracy, supporting *H1b*, though accuracy was lower than the C4.5 decision tree.

The wrapper model proved to be an effective method for developing feature sets that boost classifier performance, supporting *H2*. Additionally, accuracy gains were seen across all tested languages, supporting *H2a*, *H2b*, and *H2c*. The wrapper model is a promising approach to enabling classification of new collections of malware, especially those written in different programming languages.

Lastly, classifier accuracy was always better during hierarchical classification when compared against accuracy in classifying between all four attack vectors simultaneously. This supports *H3* and all sub-hypotheses. This result demonstrates potential for classifying more expansive data sets using a hierarchical classification approach; varying levels of classification granularity can be explored.

This work has much room for possible research extension. One immediate avenue to consider is other GA techniques for improved accuracy. For example, the entropy-weighted genetic algorithm (EWGA) utilizes information gain at the feature level for more efficient mutation of chromosomes. This can result in better feature sets for classification (Abbasi et al, 2008). Additionally, we can test classification of source files based on other elements, such as categorizing malware based on their target platforms or identifying potential authors in some cases. Hierarchical classification seems promising for more deep analyses.

VII. CONCLUSION

Cyber security is a growing field of interest for many researchers. However, many lack the ability to interpret malware source code files that are commonly found in hacker communities. Here we use a wrapper model approach to automatically analyze malware source code files of various types. Experiments are performed across three different programming languages and four attack vector classes, all yielding promising results. This work can be advanced by developing finer-grained classifiers and improving feature selection algorithms for deeper levels of analysis.

ACKNOWLEDGEMENTS

This work was supported in part by Defense Threat Reduction Agency, Award No HDTRA1-09-1-0058, and by the National Science Foundation under Grant No. CBET-0730908.

REFERENCES

[1] Imperva, "Hacker Intelligence Initiative, Monthly Trend Report #5," *Imperva Hacker Intelligence Initiative*, no. 5, October, 2011.

[2] V. Benjamin and H . Chen, "Securing Cyberspace: Identifying Key Actors in Hacker Communities," IEEE International Conference on Intelligence and Security Informatics, 2012, pp. 24-29.

[3] T. J. Holt, D. Strumsky, O. Smirnova, and M. Kilger, "Examining the Social Networks of Malware Writers and Hackers," *International Journal of Cyber Criminology*, vol. 6, no. 1, pp. 891–903, 2012.

[4] J. Mielke. and H. Chen, "Botnets, and the CyberCriminal Underground," *IEEE International Conference on Intelligence and Security Informatics 2008*, pp. 206–211, 2008.

[5] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious JavaScript code," *Proceedings of the 19th international conference on World wide web - WWW '10*, p. 281, 2010.

[6] M. Yip, "An Investigation into Chinese Cybercrime and the Applicability of Social Network Analysis," *ACM Web Science Conference*, 2011.

[7] S. Goel, "Cyberwarfare Connecting the Dots in Cyber Intelligence," *Communications of the ACM*, vol. 54, no. 8, p. 132, Aug. 2011.

[8] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "RiskRanker : Scalable and Accurate Zero-day Android Malware Detection Categories and Subject Descriptors," *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, pp. 281–293, 2011.

[9] Moser, C. Kruegel, and E. Kirda, "Exploring Multiple Execution Paths for Malware Analysis," *2007 IEEE Symposium on Security and Privacy (SP '07)*, pp. 231–245, May 2007.

[10] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether : Malware Analysis via Hardware Virtualization Extensions," *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pp. 51–62, 2008.

[11] M. Ismail and N. Zainal, "A static and dynamic visual debugger for malware analysis," *2012 18th Asia-Pacific Conference on Communications (APCC)*, pp. 765–769, Oct. 2012.

[12] Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, "Andromaly": A behavioral malware detection framework for android devices," *Journal of Intelligent Information Systems*, vol. 38, no. 1, pp. 161–190, 2011.

[13] Willems, T. Holz, and F. Freiling, "Toward Automated Dynamic Malware Analysis Using CWSandbox," *IEEE Security and Privacy Magazine*, vol. 5, no. 2, pp. 32–39, Mar. 2007.

[14] J. Sahs and L. Khan, "A Machine Learning Approach to Android Malware Detection," *2012 European Intelligence and Security Informatics Conference*, pp. 141–147, Aug. 2012.

[15] A. Abbasi, H. Chen, and A. Salem, "Sentiment analysis in multiple languages," *ACM Transactions on Information Systems*, vol. 26, no. 3, pp. 1–34, Jun. 2008.

[16] R. Bapna, S. A. Chang, P. Goes, and A. Gupta, "Overlapping Online Auctions: Empirical Characterization of Bidder Strategies and Auction Prices," *MIS Quarterly*, vol. 33, no. 4, pp. 763–783, 2009.

[17] M.-H. Tsai, K.-C. Chang, C.-C. Lin, C.-H. Mao, and H.-M. Lee, "C&C tracer: Botnet command and control behavior tracing," *2011 IEEE International Conference on Systems, Man, and Cybernetics*, pp. 1859–1864, Oct. 2011.

[18] L. Vignolo, D. Milone, C. Behaine, J. Scharcanski, and S. Member, "An Evolutionary Wrapper for Feature Selection in Face Recognition Applications," *2012 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 1286–1290, 2012.

[19] X. Wu, V. Kumar, J. Ross Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, Z.-H. Zhou, M. Steinbach, D. J. Hand, and D. Steinberg, *Top 10 algorithms in data mining*, vol. 14, no. 1. 2007, pp. 1–37.

[20] Metasploit. "Metasploit Framework Penetration Testing Software," www.metasploit.com, 2013.