Discovering Programmer Intention Behind Written Source Code

Gadiel Sznaier Camps [1]
Dept. of Aerospace & Astronautics
Stanford Univ., Palo Alto, CA
gsznaier@stanford.edu

Nicolas Bohm Agostini
Dept. of Electrical and Computer Eng.
Northeastern Univ., Boston, MA
agostini@ece.neu.edu

David Kaeli
Dept. of Electrical and Computer Eng.
Northeastern Univ., Boston, MA
kaeli@ece.neu.edu

***Abstract*—The goal of this work is to leverage natural language processing techniques to assist in the classification and understanding of a programmer's intention from inspecting source code. Our model utilizes well-known machine learning techniques. We find that we can accurately classify C source code into different classes, distinguishing between benign and malicious source code with a high degree of accuracy.**

***Index Terms*—source code comprehension, bag of words, malware detection, machine learning**

I. Introduction

As programming projects grow more complex, it is becoming harder to uncover vulnerabilities and malicious code by reviewing the syntax [1]. Thus, it is extremely important to design automated tools for source code comprehension. However, this goal is easier said than done. Typically, software programs struggle when trying to interpret syntax and comprehend the meaning behind words. There has been a push to design natural language processing (*NLP*) algorithms that can overcome these problems, using a variety of techniques, such the methods proposed in prior studies [2]–[7] to better understand programmer intention. However, while these techniques have greatly increased our ability to transform a corpus of text samples into a form that machines can better understand, they are difficult to apply directly, and so are best utilized when implemented in conjunction with machine learning algorithms. Furthermore, most of these techniques are often targeted for a single language. On the other hand, while a model based on bag of words *NLP* can handle multiple languages, it does not exploit syntactical relationships.

The focus of this paper is to design a model that is able to classify source code as malicious/benign by examining its human-readable instructions. To this effect, we propose several different pre-processing techniques and utilize a simple vectorization technique to process the syntaxs and comments used to create the program. While our results are currently limited to C source code samples, we believe that our methodology is sufficiently general and can be applied to other programming languages.

II. Related Work

Our approach is motivated by a number of prior studies. Shalaby et al. [8] used SVM classifiers to perform source code comprehension. However, they used variable and structure counts as their features. Sachdev et al. [9] studied programmer intent using TTF-IDF to pre-process the data, which we found to produce poorer results when compared to simply sanitizing code of comments. Kuhn et al. [10] used Semantic Clustering to group semantic artifacts, called topics, to identify the programmer's intent. Although the approach looks promising, it is sensitive to the programmer's naming conventions and relies on structured naming practices to make meaningful connections. This can negatively affect accuracy when examining languages such as C, which permits integration with machine code. Finally, the approach used by Ugure et al. [11] is the most similar to our own. They use natural language processing combined with a SVM classifier to interpret their samples. However, as shown in this paper, a Random Forest classifier performs better than an SVM classifier for the same number of samples.

Although most work in the field of malware detection has focused on examining and interpreting binaries, a number of prior studies [12]–[15] have already shown the benefits of leveraging NLP to distinguish between benign and malicious source code [16], [17]. Notably, Gandotra et al. [18] used NLP in tandem with a genetic algorithm and an SVM classifier to distinguish between run-time compiled languages such as Python and Perl. Furthermore, Cen et al. [17] used NLP to assist in static analysis of Java source code on Android.

III. Model Design

We begin by describing the creation of our dataset, the pre-processing methods used, and the models selected to interpret source code samples. As part of this work, we evaluated seven different pre-processing techniques and five different models in an effort to determine the best pre-processing method and model combination that produces the best classification results. Furthermore, we are informed by the work of Sachdev et al. [9], where their goal was to interpret the intention of the programmer using of bag of words vectorization [19].

A. Dataset

To construct our dataset, a database of 4,461 code samples and 59 classes was used. Each class represented a different programming challenge. Since the objectives for the proposed problems were broad, we hoped that the proposed models

[1] This work was done while at Northeastern University.

978-1-7281-4550-1/19/$31.00 ©2019 IEEE
DOI 10.1109/ICMLA.2019.00081

presented in Section III-C could be applied to solve more realistic challenges, such as malware detection. The samples used in this dataset were all written in GNU C. On average, the dataset has 76 samples per labeled class, with a minimum of 20 and a maximum of 150 samples per class. The complete distribution is shown in Table I. The main dataset was created by collecting source code submissions from Codeforces.com [20]. Codeforces is a website that hosts programming challenges, where the competitors have to solve a given task as quickly and efficiently as possible. On average, the submission lengths are short, with 99% of the samples having 100 lines of code or fewer.

Later, for the case study described in Section V, the dataset was modified by adding benign samples taken from GitHub and malicious samples taken from the MalSource Dataset, created by Calleja et al. [21], samples found on GitHub and samples taken from the malware theZoo [22] database. In Sections IV and V we will discuss how we vectorized the dataset samples using one of the pre-processing methods in conjunction with bag of words.

B. Pre-Processing Methods

Before the samples are processed, all symbols are first converted into human readable equivalent words. For example, the symbol, + is replaced with the word _plus_. The modified dataset of C source code samples is then represented using the following seven pre-processing methods:

Naive bag of words: Using this method, no information was removed before applying the models.

Naive bag of words without comments: This method is similar to the previous approach. However, in this method, it is assumed that comments are potentially misleading and should be treated as noise. Therefore, all comments were first removed before being vectorized with bag of words.

Term frequency-inverse document frequency(TF-IDF): In this technique, the samples were processed using TF-IDF, reducing the weight associated with common and rare symbols.

C operators only For this method, only C operators were kept. This includes C keywords such as *int* and *while*, as well as logic, arithmetic and pointer operators such as == and *.

C operators and ANSII standard functions only: For this method, only C operators and function keywords, denoted by the ANSII standard, were kept. The assumption is that, regardless of what the programmer names her/his method(s), these would still explain the programmer's underlying goal.

C operators and POSIX standard functions only: In this method only C operators and function keywords, denoted by the POSIX standard, were kept. It was hypothesized that either POSIX functions or ANSII keywords were more commonly used, as one was more useful than the other in looking for distinguishing features.

C operators, ANSII, and POSIX functions only: In this technique, only C operators and function keywords, denoted by the ANSII and POSIX standards, were kept.

C. Models

Once the data is pre-processed, it is then split into training (60%), testing (20%) and validation (20%) sets to train the 5 different models, resulting in 35 possible combinations. For each pre-processing method, before being trained, we performed 5-fold cross validation to ensure that the pre-processing technique avoids over-fitting the data. Once trained, the models are fed the validation data. Finally, classification results are compared to see which models and pre-processing methods performed best. The models described in this section were created using the Scikit-learn python package [23], consequently, they use some of the package's default parameters.

During evaluation (Section IV), baseline metrics were mostly used for each of the 35 combinations. However, to prevent bias due to class size, each class was weighted based on the number of samples contained in that class. The baseline metrics were as follows:

Linear Regression classifier [24]: The model used L2 penalties, primal formulation, a stopping tolerance of $1e^{-4}$ and an inverse regularization strength of 1. This allowed a constant bias to be added to the decision function, and had a intercept scaling of 1. The model did not shuffle the data prior to being examined and was trained for 100 iterations.

Support Vector Machine (SVM) classifier [25]: For this model, the penalty value was left at 1. The model used a linear function kernel and a shrinking heuristic to classify results. The model did not shuffle the data prior to being examined and did not use probability estimates to aid classification. Furthermore, the stopping tolerance was set to $1e^{-3}$ and we did not bound the number of iterations.

Decision Tree classifier [26]: The model used a Gini impurity criterion, choosing the best split for each node, did not enforce a maximum depth and allowed nodes with more than 1 sample to be split further. The model did not limit the number of leaf nodes and did not enforce a minimum weight fraction on the leaf nodes. We also allowed all features to be considered for a given split. The model did not shuffle the data prior to examination, nor did it pre-sort the data.

Random Forest classifier [27]: The model used 10 decision tree estimators, where each estimator used the Gini impurity criterion. No maximum depth was specified for the estimators, and similar to the decision tree, nodes with more than one sample were allowed to split. No weighted fraction was enforced on the leaf nodes and all features were used to make a decision. In addition, no maximum number of leaf nodes was enforced. No increase in impurities were allowed and a node was allowed to split if its impurity value was higher than $1e^{-7}$. To aid in the generation of different estimators, bootstrap samples were allowed when building the trees. Furthermore, no parallelization was applied to increase the speed of the model. Finally, the data was not shuffled before being examined.

Neural Network [28]: The model used 100 nodes for the first layer and 70 nodes for the second layer. It was trained for 500 iterations before returning a result. The model was set to use a rectified linear unit activation function, a stochastic

TABLE I: C source code sample distribution over the 59 classes of problems collected from the codeforces website. Each class represents a problem proposed by the codeforces staff.

Class Name	926A	940A	946C	965A	926G	985A	946B	980B	952A	934A	931B	978B
# of samples	22	34	41	117	28	56	52	20	150	26	51	117
Class Name	977A	938B	937A	932B	954B	978C	975B	955A	935C	950B	939A	976B
# of Samples	150	75	138	24	23	40	26	72	31	66	150	29
Class Name	979A	935B	962A	984B	982A	961A	960A	940B	980A	946A	940C	926C
# of Samples	121	120	150	33	64	77	78	35	118	150	41	35
Class Name	985B	977B	931A	978A	957A	934B	932A	977C	954A	937B	952C	938A
# of Samples	26	86	125	133	69	68	59	23	128	26	24	113
Class Name	950A	939B	935A	976A	959A	962B	984A	948A	967A	961B	964A	X
# of Samples	150	53	150	105	150	25	134	53	41	23	137	X

gradient-based optimizer solver and a constant learning rate with a step rate of 0.001. The model used an L2 penalty with an alpha value of 0.0001. The data samples were randomly shuffled for each iteration. In addition, the model used a stopping tolerance of $1e^{-4}$, was not allowed to use solutions from previous iterations and was not allowed to terminate early if the model's accuracy did not improve after 10 iterations. The model had a $beta1$ of 0.9 and $beta2$ of 0.999. Numerical stability was left at $1e^{-8}$.

IV. Evaluation

The evaluation consists of two experiments, first we classified the C source code samples into their respective Codeforces problem groups to determine if it was possible to distinguish between samples based on the code produced by the programmer. Then, we explored the robustness of these models by replacing variable names with non-descriptive names (Ex. int *wordCount* $\rightarrow$ int *var0*).

A. Source Code Comprehension

As shown in Table II, the average accuracy across the models and pre-processing methods was around 87%. Since the models performed fairly well, even though their parameters were mostly left unchanged, this suggests that it is unnecessary to use complex processing techniques to interpret the programmer's intention for a given source code. Furthermore, this also suggests that with more time spent in tweaking each of the model's parameters, an even higher accuracy could have been achieved. Out of these implementations, on average, the Random Forest Classifier performed the best and achieved the highest accuracy (91%) when only comments were removed. In comparison, the model that was least successful was the Decision Tree classifier, which had a maximum accuracy of 89%. Similarly, as shown in Tables III and IV, on average, the Random Forest Classifier also had the highest precision and recall when compared to the other models.

Interestingly, across all the models, the pre-processing method that almost consistently performed the best was when the data was first sanitized of all comments. This suggests that the comments, while useful for human comprehension, can instead also introduce misleading and noisy information. In comparison, the pre-processing method that performed consistently the worst was TF-IDF. This is not particularly surprising as TF-IDF assigns lower weight to terms that appear very often or rarely in a dataset, and by reducing the weight of the rare terms, TF-IDF consequently disregarded features that were highly descriptive for the given problem.

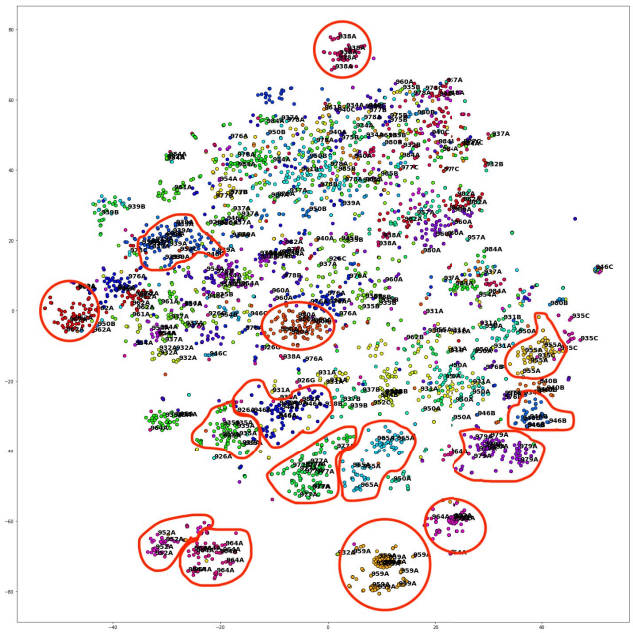

Fig. 1: TNSE plot of the samples for the 59 classes based on features (Perplexity=35). Clusters are shown using red circles.

To better understand the reason behind the high precision and recall displayed by the models when comments were removed, we created a TSNE visualization, shown in Figure 1, for the samples based on features selected by the training set. As shown in the plot, although there was a large degree of overlap between classes, there were also several distinct clusters, suggesting that while difficult, there were enough features to make classification possible. This observation matches the observed precision and recall results.

Examining the accuracy of each of the models more closely, we observed that in addition to having the best accuracy, the Random Forest Classifier was relatively robust as it had a

TABLE II: Model Accuracy for the 5 models using different pre-processing methods. The best accuracy is shown in bold.

	naive approach: use everything	naive approach: ignore comments	TF-IDF	operators and keywords only	operators, keywords, and ANSII funct. only	operators, keywords, and Posix funct. only	operators, keywords, Posix, and ANSII funct only
Logistic Regression Classifier	0.876	0.892	0.810	0.880	0.905	0.881	0.905
Support Vector Machine	0.881	0.896	0.823	0.882	0.895	0.878	0.892
Decision Tree Classifier	0.852	0.875	0.778	0.832	0.841	0.835	0.892
Random Forest Classifier	0.889	**0.910**	0.828	0.882	0.889	0.882	0.901
Neural Network	0.865	0.878	0.807	0.892	0.905	0.885	0.903

deviation of only 8% across the seven different pre-processing techniques. This robustness most likely stemmed from the fact that Random Forest Classifiers make their decisions by aggregating individual results of several decision trees with different feature splits, preventing the model from falling into a local minimum. Which also explains why it performed better than its counterpart, the Decision Tree model.

Based on the model precision and recall presented in Tables III and IV, although the models were able to correctly classify a given problem, they were also less able to recognize if a sample belonged to a given class. This implies that this approach will be good at classifying source code, but will struggle when applied to more sensitive tasks, such as detecting malicious applications, as it can potentially allow malicious code to pass undetected.

B. Variable Agnostic Code Comprehension

To determine if the proposed models are dependent on descriptive variables, the variables names for each sample were replaced with non-descriptive names. For example, if a sample has three variables, *wordcount*, *maxcount*, and *bookname*, then the non-descriptive names are *var0*, *var1*, *var2*. We assumed that programmers tend to follow similar naming conventions when solving a given problem, which would allow the models to more easily distinguish between different problems. Thus, it was proposed that the removal of this information would significantly affect the accuracy of the models. Two tests were then performed to determine the validity of this assumption.

First, the models were trained on samples with non-descriptive variables. Once trained, a validation set of samples with non-descriptive variables was then passed to them and the results were compared to the values in Table II. The objective was to determine if the models would perform worse on more generalized data. The results of this experiment are shown in Table V. However, comparing the results shown in Tables II and V, there was no appreciable difference in accuracy between the two experiments. This suggests that the models are robust to variable names.

In the second test, the models were instead trained using samples with descriptive variable names and then once again given a validation set with samples that had non-descriptive variables names. It was postulated that since the samples were significantly different from the training set, that this would in turn cause the models to perform worse. The results of this experiment are shown in bold in Table V. Similar to the previous experiment, the model accuracy did not decrease significantly when compared to the results in Table II for the naive and without comments pre-processing methods.

V. Case Study: Malware Classification

Malicious software, also known as Malware, is a growing problem that affects both individuals and businesses alike. Often these pieces of software are only discovered in their binary format after the damage has been done. In the past, the main solution to combating new malware has been to check new binaries against an ever-growing signature log of known malicious code [29]. However, this suffers from an inability to protect devices from zero-day attacks [30].

Thus, with promising results [12]–[15], researchers have employed machine learning techniques to identify malicious code that may not have had its signature logged and would go undetected by tools that rely on signature matching. The downside with this approach is that it requires the victim device to download potentially compromised binaries onto the machine and relies on the model to detect anything harmful before the user runs the program. Hence, since open source applications are becoming more frequently used in personal and commercial applications, it would greatly benefit the community if malware could be detected before it has been compiled and distributed as a binary.

Thus, to demonstrate that the techniques proposed in this paper have realistic application, we focused on the three top preforming models from our previous experiments (Logistic Regression Classifier, SVM, and Random Forest Classifier) and the pre-processing approach that led to the highest accuracy (naive bag of Words with comments removed) and trained them to distinguish between benign and malicious source code. The hope was that this would detect attacks at the origin, before the source code has been compiled into a binary.

A. Experimental Setup

For this experiment, the dataset was modified by adding samples taken from the MalSource Dataset created by Calleja et al [21], samples found on GitHub and samples taken from the malware theZoo database [22]. Because malicious programs are often made up of multiple source codes files, some of which are innocuous, each file was first manually examined to see if they would intentionally cause harm if run before being added into the database. The goal of this step was to minimize the model from falsely classifying benign code as malware, while stopping malware from being misclassified, hence negating any possible harm. Furthermore, to more

TABLE III: Model precision for the 5 models using different pre-processing methods. The best precision is shown in bold.

	naive approach: use everything	naive approach: ignore comments	TF-IDF	operators and keywords only	operators, keywords, and ANSII funct. only	operators, keywords, and Posix funct. only	operators, keywords, Posix, and ANSII funct only
Logistic Regression Classifier	0.868	0.880	0.784	0.845	0.871	0.848	0.872
Support Vector Machine	0.859	0.868	0.830	0.846	0.854	0.847	0.849
Decision Tree Classifier	0.800	0.848	0.720	0.794	0.813	0.805	0.827
Random Forest Classifier	0.846	**0.901**	0.793	0.871	0.874	0.857	0.882
Neural Network	0.810	0.872	0.772	0.877	0.876	0.874	0.879

TABLE IV: Model recall for the 5 models using different pre-processing methods. The best result is shown in bold.

	naive approach: use everything	naive approach: ignore comments	TF-IDF	operators and keywords only	operators, keywords, and ANSII funct. only	operators, keywords, and Posix funct. only	operators, keywords, Posix, and ANSII funct only
Logistic Regression Classifier	0.848	0.866	0.801	0.865	0.887	0.866	0.887
Support Vector Machine	0.847	0.860	0.798	0.841	0.852	0.837	0.851
Decision Tree Classifier	0.824	0.858	0.750	0.794	0.807	0.809	0.822
Random Forest Classifier	0.865	**0.892**	0.785	0.860	0.861	0.850	0.872
Neural Network	0.810	0.813	0.741	0.863	0.874	0.853	0.860

TABLE V: Model Accuracy for the 5 models using different pre-processing methods when variable names are replaced with non-descriptive names. Normal text shows results when trained on samples with non-descriptive variables. Boldface text shows the results when trained on samples with descriptive variables.

	naive approach: use everything	naive approach: ignore comments	TF-IDF
Logistic Regression Classifier	0.879 / **0.873**	0.892 / **0.891**	0.830 / **0.686**
Support Vector Machine	0.856 / **0.866**	0.869 / **0.878**	0.862 / **0.652**
Decision Tree Classifier	0.848 / **0.840**	0.862 / **0.873**	0.782 / **0.628**
Random Forest Classifier	0.895 / **0.885**	0.913 / **0.903**	0.844 / **0.709**
Neural Network	0.892 / **0.830**	0.903 / **0.857**	0.850 / **0.625**

TABLE VI: Malware Dataset Distribution

Source code type	Amount
Benign Source	4461
Malware Source	153
Total	4614

TABLE VII: Malware detection average model accuracy, precision, and recall.

Models	Accuracy	Precision	Recall
Random Forest Classifier	0.999	0.999	0.983
Logistic Regression Classifier	1	1	1
Support Vector Machine	1	1	1

closely follow the structure of the source code used for the benign samples, each malicious source code file was treated as a separate sample rather than grouping them together by application. Finally, to reduce misclassification, all malicious code samples were assigned to a single class and while all benign samples were assigned to another class, resulting in a binary classification experiment. The resulting sample distribution is shown in Table VI.

B. Results

As clearly shown in Table VII, the models do much better in distinguishing between benign and malicious code. This sudden increase in accuracy is most likely due to the fact that both classes have larger sample sizes. Similar to the results shown in Tables III and IV, each model's precision is once again higher than their recall. However, the difference between the precision and recall is so small that it can be dismissed as being statistically insignificant.

Surprisingly, both the Logistic Regression and the SVM classifiers have better recall than the RF classifier. Yet, the difference between its precision and the precision of the other classifiers is small enough that the variation can be ignored. In an effort to determine what caused the RF model accuracy to decrease, we present its confusion matrix in Table VIII. Table VIII shows that about 3% of the malware were incorrectly classified as benign. This suggests that, although the detection rate is extremely good, this method still needs to be used in tandem with other detection methods, as even if a single malicious sample is misclassified and allowed to pass, it can still have dramatic consequences.

TABLE VIII: Confusion matrix for malware detection.

	Predicted Benign	**Predicted Malware**
Actually Benign	1	0
Actually Malware	0.03	0.97

Comparing the lengths of the benign source code (10-100 lines) and the malicious source code (10-500+ lines), we noticed that there was, on average, a significant difference in length between the two types. This led us to worry that the high accuracy of our models was in part due to this disparity. Thus, to remove this potential source of bias, we decided to

add an additional 364 new samples of benign code, where their average length was closer to the average length of the malware samples used, and rerun the experiment. The results of this new experiment are shown in Table IX.

TABLE IX: Malware detection average model accuracy, precision, and recall.

Models	Accuracy	Precision	Recall
Random Forest Classifier	0.998	0.999	0.983
Logistic Regression Classifier	0.991	0.941	0.997
Support Vector Machine	0.991	0.956	0.882

From the results shown in Table IX, we see that while the accuracy of the RF classifier remained unchanged, the accuracy of the LR classifier and the SVM decreased significantly. This indicates that the RF classifier is less sensitive to the different features introduced by the lengthy samples, while the other two models were more sensitive. Furthermore, as stated earlier, it again implies that the RF classifier is more robust to false patterns than other models, as it works by aggregating the results obtained by its decision trees estimators.

VI. Conclusion and Future Work

In this paper we explored using a Bag of Words vectorization to identify subtle differences in C source code examples. Equipped with models that assume default parameters, we first were able to interpret the intention behind a programmer's source code and identify which problem a given code sample belongs. Furthermore, we observed that comments, while helpful for human understanding, act as noise, impeding each model's ability to correctly classify samples. In addition, we observed that all the models were robust to changes in pre-processing. However, out of the seven models tested, the Random Forest Classifier performed the best. Moving to a more applied problem, we were also able to distinguish between malicious and benign source code with a high degree of accuracy. A possible future avenue to pursue would be to explore more complex pre-processing methods to help improve detection results between different classes of benign samples. Given that these models assume that the samples are not obfuscated, another avenue of exploration would be to see if the models can detect malicious source code when the attacks are camouflaged.

References

[1] M. M. Carey and G. C. Gannod, "Recovering concepts from source code with automated concept identification," in *ICPC'07*, 2007, pp. 27–36.

[2] A. L. Berger, V. J. D. Pietra, and S. A. D. Pietra, "A maximum entropy approach to natural language processing," *Computational linguistics*, vol. 22, no. 1, pp. 39–71, 1996.

[3] S. Muggleton, H. Lodhi, A. Amini, and M. J. Sternberg, "Support vector inductive logic programming," in *Int. Conf. on Discovery Science*, 2005, pp. 163–175.

[4] J. R. Manning and M. J. Kahana, "Interpreting semantic clustering effects in free recall," *Memory*, vol. 20, no. 5, pp. 511–517, 2012.

[6] L. Wu, S. C. Hoi, and N. Yu, "Semantics-preserving bag-of-words models and applications," *IEEE Trans. on Image Processing*, vol. 19, no. 7, pp. 1908–1920, 2010.

[5] C. Biemann, "Chinese whispers: an efficient graph clustering algorithm and its application to natural language processing problems," in *Proc. of the first workshop on graph based methods for NLP.*, 2006, pp. 73–80.

[7] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *NIPS*, 2013, pp. 3111–3119.

[8] M. Shalaby, T. Mehrez, A. El Mougy, K. Abdulnasser, and A. Al-Safty, "Automatic algorithm recognition of source-code using machine learning," in *2017 16th ICMLA*, 2017, pp. 170–177.

[9] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra, "Retrieval on source code: a neural code search," in *MAPL*, 2018, pp. 31–41.

[10] A. Kuhn, S. Ducasse, and T. Gírba, "Semantic clustering: Identifying topics in source code," *Information and Software Technology*, vol. 49, no. 3, pp. 230–243, 2007.

[11] S. Ugurel, R. Krovetz, and C. L. Giles, "What's the code?: automatic classification of source code archives," in *SIGKDD*, 2002, pp. 632–638.

[12] I. Firdausi, A. Erwin, A. S. Nugroho *et al.*, "Analysis of machine learning techniques used in behavior-based malware detection," in *Int. conf. on adv. in computing, control, and telecommunication technologies*, 2010, pp. 201–203.

[13] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, "Learning and classification of malware behavior," in *Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008, pp. 108–125.

[14] I. Santos, J. Devesa, F. Brezo, J. Nieves, and P. G. Bringas, "Opem: A static-dynamic approach for machine-learning-based malware detection," in *Int. Joint Conf. CISIS¢¬Ä¬ô12-ICEUTE 12-SOCO 12 Special Sessions*, 2013, pp. 271–280.

[15] N. Milosevic, A. Dehghantanha, and K.-K. R. Choo, "Machine learning aided android malware classification," *Computers & Electrical Engineering*, vol. 61, pp. 266–274, 2017.

[16] V. A. Benjamin and H. Chen, "Machine learning for attack vector identification in malicious source code," in *2013 IEEE Int. Conf. on Intelligence and Security Informatics*, 2013, pp. 21–23.

[17] L. Cen, C. S. Gates, L. Si, and N. Li, "A probabilistic discriminative model for android malware detection with decompiled source code," *IEEE Trans. on Dependable and Secure Computing*, vol. 12, no. 4, pp. 400–412, 2015.

[18] E. Gandotra, D. Bansal, and S. Sofat, "Malware analysis and classification: A survey," *J. of Inf. Security*, vol. 5, no. 02, p. 56, 2014.

[19] G. Salton and M. J. McGill, "Introduction to modern information retrieval," 1983.

[20] M. Mirzayanov, "Codeforces," 2017, last accessed 7 July 2018. [Online]. Available: https://codeforces.com

[21] A. Calleja, J. Tapiador, and J. Caballero, "The malsource dataset: Quantifying complexity and code reuse in malware development," *IEEE Trans. on Inf. Forensics and Security*, 2018.

[22] Y. Nativ, "thezoo - a live malware repository," 2019, last accessed 9 April 2019. [Online]. Available: https://thezoo.morirt.com/

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *J. of Machine Learning Res.*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[24] G. A. Seber and A. J. Lee, *Linear regression analysis*. John Wiley & Sons, 2012, vol. 329.

[25] T. Joachims, "Text categorization with support vector machines: Learning with many relevant features," in *ECML*, 1998, pp. 137–142.

[26] S. R. Safavian and D. Landgrebe, "A survey of decision tree classifier methodology," *IEEE trans. on sys., man, and cybernetics*, vol. 21, no. 3, pp. 660–674, 1991.

[27] A. Liaw, M. Wiener *et al.*, "Classification and regression by randomforest," *R news*, vol. 2, no. 3, pp. 18–22, 2002.

[28] M. T. Hagan, H. B. Demuth, and M. H. Beale, *Neural network design*, vol. 20.

[29] Y. Ye, D. Wang, T. Li, and D. Ye, "Imds: Intelligent malware detection system," in *SIGKDD*, 2007, pp. 1043–1047.

[30] P. M. Comar, L. Liu, S. Saha, P.-N. Tan, and A. Nucci, "Combining supervised and unsupervised learning for zero-day malware detection," in *INFOCOM*, 2013, pp. 2022–2030.