MalCrawl

Detecting Malicious Source Code on GitHub

RJ Joyce and Shantanu Hirlekar

Background and Motivation

There is abundant research on the problem of accurately detecting and blocking a malicious executable before it is run on a victim's computer. However, there is surprisingly little research on the similar problem of determining whether or not a set of source code compiles into a malicious executable. The likely reason that there is not much focus on this topic in the malware research community is because source code does not have the capability to directly harm a computer. However, the public release of a malware's source code can indirectly cause widespread harm. People without the technical skill to write their own malware can easily compile a malware's public source code and use it for their own criminal purposes. Sophisticated malware authors can use a malware's public source code as a foundation and add additional functionality to it rather than developing a malware family entirely from scratch. In addition, these authors can add desirable features, such as antivirus evasion techniques, from a malware's public source code into their own malware.

The most significant release of malware source code to date was undoubtedly the leak of the Zeus malware's source code in 2011. Originally discovered in 2007, Zeus is one of the most notorious malware families ever created. It uses keylogging and credential theft to steal money from the bank accounts of its victims and advanced stealth techniques to evade detection by antivirus software. By 2010, it was estimated to have caused over $100,000 in damage and to have infected 3.6 million computers in the U.S. alone ("Zeus" 2). Its source code was sold and subsequently leaked onto the internet in May of 2011, which allowed anyone to compile the source code and use the Zeus malware for their own criminal purposes. Even six years later, authors of other banking trojans are reusing the advanced and powerful capabilities implemented

in its source code. Elements from the leaked Zeus source code have been incorporated into nearly every major banking trojan developed since, including Gameover, Ice IX, Spyeye, Citadel, Shylock, Kins, Bugat, Torpig, Panda, and Sphinx.

The topic of open-source malware was again brought into the limelight in 2015 after the source code of Hidden Tear was published to the project sharing site GitHub. Hidden Tear was branded as a proof-of-concept educational ransomware by its developer, Utku Sen. Sen's primary motivation for publishing Hidden Tear was to "provide a source code for newbies, students who are trying to understand the process" (Sen 1). However, with only a few minor changes, the source code could be weaponized to hold a victim's files for ransom. The publication of the Hidden Tear source code allowed many people without the skills to develop a fully-functional ransomware to enter the ransomware 'industry.' In February 2016, five months after the Hidden Tear source code was published on GitHub, the antivirus company Kaspersky stated that they had identified 24 other ransomware families that contained some part of Hidden Tear's source code (Cimpanu 1).

MalCrawl Project Vision

Similarly to how antivirus software can distinguish between malicious and benign executables, our goal was to write a proof-of-concept program that could determine whether a set of source code will be compiled into a malicious or benign executable. When we began the project, we envisioned that our research could be used to prevent malware authors from hosting their source code on GitHub and other project sharing sites. To demonstrate that our proof-of-concept program had real-world applications, we decided to create a crawler that could

identify malware source code hosted on GitHub.

Data

We downloaded the source codes of 68 malware families from various GitHub repositories, including the source codes of the Zeus banking trojan and the Hidden Tear ransomware. Although Utku Sen removed both Hidden Tear and his other open-source ransomware, Eda2, from GitHub after he received backlash from the security community, we were able to acquire the source codes of both of these families by contacting him via email. We used the website gitrandom.com to download a random sample of 100 clean GitHub repositories. Tar files containing the malicious and clean source code used in the project can be provided upon request.

YARA Rules

Our original plan for detecting malicious GitHub repositories was to scan them with open-source YARA rules. YARA is a pattern-matching tool used by malware researchers. A YARA rule contains any number of strings, byte sequences, and regular expressions followed a condition. The following is an example of a YARA rule that is used to detect version 1.1.3.4 of the Zeus banking trojan:

```
rule Zeus_1134 {
   strings:
      $mz = {4D 5A}
      $protocol1 = "X_ID: "
      $protocol2 = "X_OS: "
      $protocol3 = "X_BV: "
      $stringR1 = "InitializeSecurityDescriptor"
      $stringR2 = "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; SV1)"
   condition:
```

($mz at 0 and all of ($protocol*) and ($stringR1 or $stringR2))
}
(Xylitol 1).

The YARA command line program can be used to scan files with a set of rules. For example, if the scanner uses the YARA rule above, any file starting with the bytes '0x4D5A,'containing the strings "X_ID: ," "X_OS: ," and "X_BV: ," and either the string "InitializeSecurityDescriptor" or the user agent "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; SV1)" will be detected as Zeus version 1.1.3.4. YARA rules can be written for any type of file, although they normally written for executable files. Most YARA rules are written to detect a single malware family, but some are written to detect generic malicious traits.

We were interested in learning whether a YARA rule that was written to detect a malware family could also detect the source code of that family. We tried to find YARA rules that were written to detect each of the 68 malware families in our dataset. However, we were only able to find publicly available YARA rules for 16 of them. Links to the YARA rules are provided in the appendix. We then scanned each of the 16 sets of source code with its corresponding YARA rule. The experiment resulted in 8 of the YARA rules correctly identifying the source code of the malware family it was written for. Next, we investigated each YARA rule that did not detect its malware family's source code. Many of these rules contained conditions based on byte patterns and file sizes, so we removed them and repeated the experiment. However, no additional sets of source code were detected.

Next, out of curiosity, we scanned all 60 sets of malware source code with all of the YARA rules in the open-source Yara Rules project on Github. This project contains hundreds of YARA rules written to detect both specific malware families and generic malicious attributes.

The source codes of the NoobKit and Alina malware families were detected by a YARA rule that detects rootkits. In addition, the source codes of the Blackhole and Rig exploit kits were detected by multiple rules that detect webshells. As a result of these experiments, we learned that YARA rules designed to detect a single family cannot be used to reliably discover malicious source code. However, we discovered that YARA rules that detect generic malicious traits, such as the rootkit and webshell rules described above, could be potentially useful for detecting malware source code. We kept these findings in mind as we proceeded to the next step of the project.

Support Vector Machines

Our next approach to the problem of determining whether source code compiles into a malicious or benign program was to use machine learning. We chose to use a support vector machine as our machine learning algorithm. A support vector machine uses supervised learning to separate two distinct classes of data. In the support vector machine, data is represented by high-dimensional vectors,, where each element of a vector corresponds with a single feature of the data. Support vector machines then determine the parameters of a hyperplane that optimally separates the two classes of data. A support vector machine can classify a new feature vector by determining which side of the hyperplane it is on (Ray 1).

The main challenge of implementing a support vector machine was identifying the features that distinguish malware source code from benign source code. After exploring various source codes in our dataset, we observed that comments, variable names, and function names best convey the purpose of the code. For example, if the source code file contains a function named “stealPassword(),” it is likely malicious. We compiled a list of strings and regular

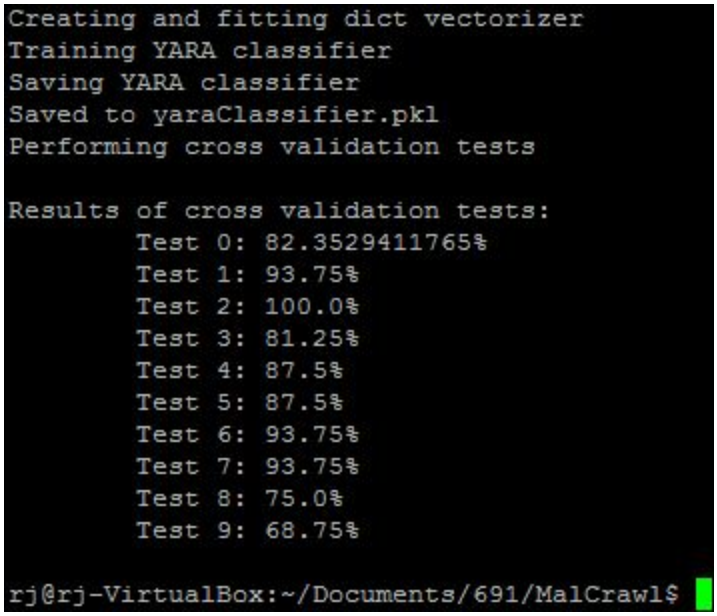
expressions that represent features in a set of source code. The list includes the names of many types of malware, words and phrases associated with malicious behaviour, and regular expressions to detect functionality such as network communication, registry keys, mutexes, cryptography, and bitcoin wallets.

We chose to use YARA rules to extract these features from the source code. We wrote 179 YARA rules based on our list of strings and regular expressions. These rules can be found in the YARA/heuristics.yar file in the MalCrawl repository. Although we could have used only regular expressions to extract the features in our list, we chose to use YARA for this task because YARA rules can contain regular expressions and because it can perform efficient file scanning. We also chose to use YARA so that it would be possible to use other open-source rules in conjunction with our own in the future.

We implemented the YARA support vector machine in Python 2.7 using the Scikit-Learn library. Its source code is located in the yaraClassifier.py file in the MalCrawl repository. The program first creates a vector with a dimension of 179, where each element of the vector corresponds to an individual YARA rule. Next, the program selects all of the files in a program's source code and scans each of them with the 179 rules in YARA/heuristics.yar. Every time a YARA rule matches a file, the corresponding element of the vector is incremented by one. Finally, after all of the files are scanned, each element in the vector is divided by the total number of files in the program's source code. This process is repeated for each set of source code in the training set.

The vectors are implemented as Python dictionaries, where each element of a vector is represented by a key-value pair. The key corresponds to the name of the YARA rule and the

value corresponds to the percent of files that the YARA rule matched. Our program converts each dictionary into a Scikit-Learn DictVectorizer object and then passes each DictVectorizer into a Scikit-Learn SVC object. The support vector machine then trains itself on 60 of the malicious repositories and 100 of the clean repositories. The remaining 8 malicious repositories were left out for later testing. To measure its accuracy, the support vector machine uses tenfold cross-validation. The output of the YARA classifier's training and cross-validation are shown in the image below.



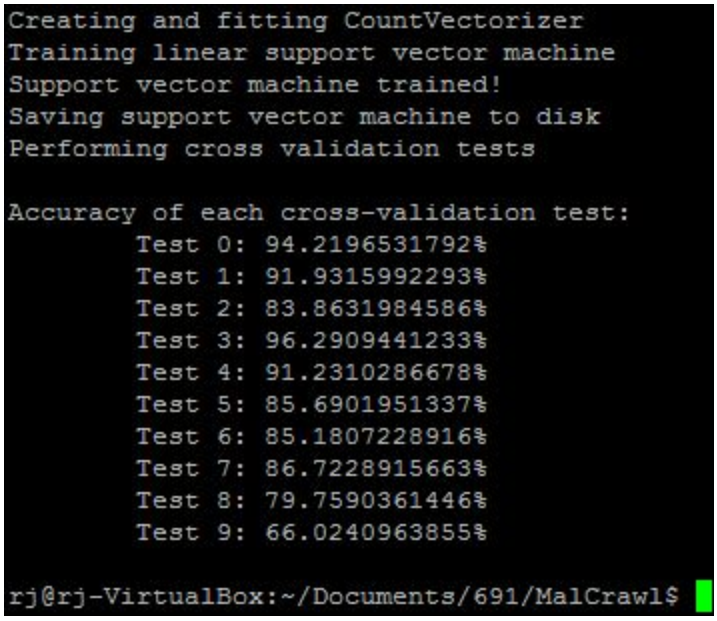
The cross-validation results showed that the YARA classifier was able to distinguish between the malicious and benign source code in the training set with an average accuracy of 86.36 percent. This accuracy was too low to be acceptable. Although we had many ideas to improve the YARA classifier, we decided to stop developing it because we believed that another approach was more promising.

Natural Language Processing

We believed that using natural language processing in combination with machine learning would be successful because our problem involves the parsing and interpretation of large amounts of textual data. We decided to a use a bag-of-words support vector machine as a proof-of-concept because of the model's simplicity. Like the YARA support vector machine, this classifier uses a hyperplane to separate high-dimensional vectors, where each element of a vector corresponds to a feature. However, the bag-of-words classifier has two major differences from the YARA classifier. The first difference is that each vector represents a single source code file rather than the entire corpus of source code of a program. The second difference is how the vectors are featurized. In the bag-of-words model, each element of a vector is equal to the number of times a specific word occurs in the vector's corresponding file. The main advantage of the bag-of-words classifier over the YARA classifier is that the dimension of each vector is equal to the number of unique words in the training corpus. This is because support vector machines are better at separating higher-dimension vectors.

Like the YARA classifier, the bag-of-words support vector machine was implemented in Python 2.7 using the Scikit-Learn library. Its source code is in the file bagOfWordsClassifier.py in the MalCrawl repository. We faced a unique natural language processing challenge because our training data was composed of many programming languages. We decided to normalize each source code file by converting them into collections of tokens. We defined a token as any string of 4 to 25 letters, numbers, hyphens, and underscores. Each time the program identifies a token, it removes all of the hyphens and underscores from it and converts all of the capital letters to lowercase. Each normalized file is converted to a vector using a Scikit-Learn CountVectorizer

object. The support vector machine then trains itself on each vector using the same 60 malicious repositories and 100 clean repositories. The results of the cross-validation tests are shown below.



The average accuracy of all ten cross-validation tests was 86.09 percent. However, these results cannot be compared to the results of the YARA classifier, because the bag-of-words support vector machine classifies individual files instead of an entire set of source code. Classifying individual files allows us to assign a score to the source code of a program which indicates how malicious it is. The score of each set of source code is calculated as follows:

$$\frac{|Malicious\ Files| - |Clean\ Files|}{|All\ Files|} \times 100$$

A set of source code can have a score between negative 100 and positive 100. A score close to negative 100 indicates that the source code is benign, and a score close to positive 100 indicates that the source code is malicious. We calculated the scores of all of the source codes in our training set to test the fitness of the bag-of-words classifier, which are shown the graph

below. Benign source code is represented in blue and malware source code is represented in red.

All of the clean source code in the training set received a score of negative 87 or less and all of the malware source code in the training set received a score of 94 or greater. Clean source code had an average score of negative 99.60 while malware source code had an average score of positive 99.59. We were impressed by how well the bag-of-words classifier had fit to the training set. We used these findings it as a baseline for our next natural language processing machine learning algorithm.

Topic Modeling

Topic modeling is a natural language processing concept that is used to summarize the central ideas of a document. We intended to use the bag-of-words support vector machine as a template for a topic modeling classifier. The only major difference between the bag-of-words and topic modeling classifiers is the method by which the source code is featurized. Rather than setting each vector element equal to the number of occurrences of a word, the topic modeling

classifier sets each vector element equal to the term frequency-inverse document frequency value of a word. The tf-idf value is equal to the frequency of a word in a document divided by the frequency of that word in the entire corpus of documents ("Tf-idf" 1). Using the tf-idf value rather than the word count lowers the weights of any words that occur commonly in most documents. The words with the highest tf-idf values in a document can be considered the topics of that document.

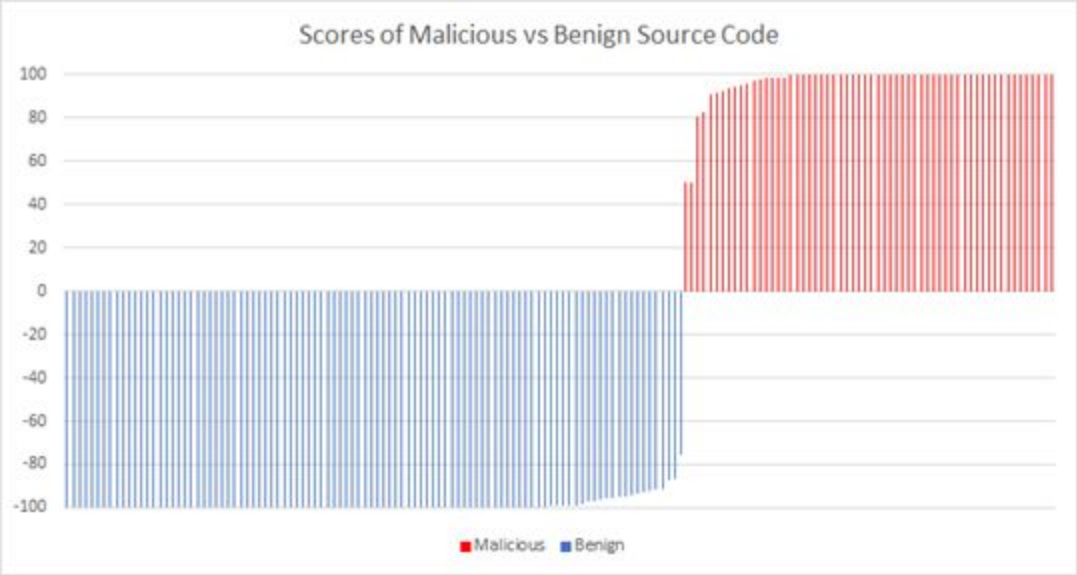
Topic modeling avoids the negative traits of the two previous machine learning classifiers; it represents the features of the source code in a higher number of dimensions than the YARA classifier, and its features have more meaning than the bag-of-words classifier. The topic modeling classifier was implemented in almost the exact same way as the bag-of-words classifier. However, it uses Scikit-Learn 'TfidfVectorizer' objects rather than 'CountVectorizer' objects. The source code of the topic modeling classifier can be found in the file topicModelingClassifier.py in the MalCrawl GitHub repository. The results of the topic modeling modeling are shown below.

```
Creating and fitting tfidf vectorizer
Training linear support vector machine
Support vector machine trained!
Saving support vector machine to disk
Performing cross validation tests

Accuracy of each cross-validation test:
        Test 0: 97.8082851638%
        Test 1: 97.2061657033%
        Test 2: 94.7976878613%
        Test 3: 97.2061657033%
        Test 4: 90.4842206697%
        Test 5: 67.1163575042%
        Test 6: 89.0602409639%
        Test 7: 92.8192771084%
        Test 8: 86.4096385542%
        Test 9: 65.8554216867%

rj@rj-VirtualBox:~/Documents/691/MalCrawl$
```

The cross-validation results showed that the the topic modeling classifier could determine whether a file belonged to the source code of a clean or malicious program with 87.88 percent accuracy, a nearly 2 percent increase over the bag-of-words classifier. All of the source codes in the training set were assigned a score calculated in the same way as the bag-of-words classifier. The graph below shows the scores of all of the source code in the training set. Benign source code is represented in blue and malware source code is represented in red.



All of the clean source code received a score of -70 or less and all of the malware source code received a score of 50 or greater. Clean source code had an average score of negative 98.69 while malware source code had an average score of positive 96.78. Unexpectedly, even though the topic modeling classifier performed better in cross-validation testing, the topic modeling classifier was more closely fitted to the training set.

Bayesian Networks

We decided to implement a Bayesian network machine learning model. The Bayesian

formula states that given a class variable 'y' with dependent feature vector 'X_1' through 'X_n', we have the possible relationship as follows:

$$P(y \mid x_1, \ldots, x_n) = \frac{P(y)P(x_1, \ldots x_n \mid y)}{P(x_1, \ldots, x_n)}$$

We decided to implement the Bayesian network model because it can be used to determine the probability that a GitHub repository is malicious or benign. The Bayesian network model classifies and learns relatively fast as compared to the other sophisticated methods. Each distribution can be independently estimated as a one-dimensional distribution and the model does not require a large amount of training data to learn compared to some of the other machine learning algorithms (Lee 1).

We implemented the Bayesian network classifier using the "reverend.thomas" Python library. The source code of the Bayesian classifier can be found in the 'bayesianClassifier.py' file in the MalCrawl repository. The "reverend.thomas" library allows the Bayesian network to be trained on the inputs of malware and benign source code using its "train()" method. Then, for each file in the testing data, Reverend's "guess()" method is used to predict the maliciousness of the file. The Bayesian network then outputs the probability of the file being malicious and the probability of the file being benign. The output looks as follows:

```
[('malware', 0.7447419231924424), ('clean', 0.5968544173072239)]
[('clean', 0.8412368844646132), ('malware', 0.6071242660812972)]
[('malware', 0.9592055967307577), ('clean', 0.08292373506845407)]
[('clean', 0.8444670686146354), ('malware', 0.502792242594907)]
[('malware', 0.6297623661985474), ('clean', 0.3925433499279436)]
[('malware', 0.550658418285641), ('clean', 0.5449318472802764)]
[('clean', 0.9155309403366647), ('malware', 0.8940102205505925)]
[('clean', 0.608477084009599), ('malware', 0.5418697706571998)]
[('clean', 0.6146197521344771), ('malware', 0.5940140133166332)]
[('clean', 0.7952955795110475), ('malware', 0.20470442048895254)]
[('clean', 0.7887973170738516), ('malware', 0.21120268292614836)]
[('clean', 0.7952955795110475), ('malware', 0.20470442048895254)]
[('clean', 0.7887973170738516), ('malware', 0.21120268292614836)]
[('malware', 0.9999)]
```

Crawler

One of the goals of the project was to make a proof-of-concept program that could identify malware source code hosted on GitHub. To accomplish this goal, we implemented a crawler that can extract all of the code in a public GitHub repository and use the machine learning classifiers described above to determine if the repository is malicious or not. The crawler loads the vectorizer and classifier of a specified machine learning model into memory when it starts up. Then, the crawler enters an infinite loop until it is stopped by the user. At each iteration of the loop, the crawler generates a random number between 0 and 57 million, which is the estimated number of GitHub repositories as of April 2017. The generated number corresponds with the ID of a GitHub repository. Next, the crawler scrapes all of the source code from that repository, normalizes it, and transforms it to fit the vectorizer. Finally, the classifier classifies each file as malicious or benign and calculates an overall score for the source code.

Finding a way to scrape all of the source code from a GitHub repository was the biggest challenge of the project other than creating the machine learning classifiers. In the initial stages of the project, we tried to implement the crawler using Scrapy. Scrapy is an open-source web crawling framework written in Python that extracts HTML data using XPath selectors. Scrapy can request individual HTML elements using its 'start_request()' function. The elements are returned from the webpage in a Scrapy response object and parsed with Scrapy's 'parse()' function. These functions allow Scrapy to to obtain the entire HTML content from the desired web page. After getting the HTML content, data can be extracted from the CSS by providing Scrapy with the desired HTML elements such as 'title,' 'text,' 'div,' and 'span.'

Although the crawler was was able to obtain HTML content from GitHub with Scrapy,

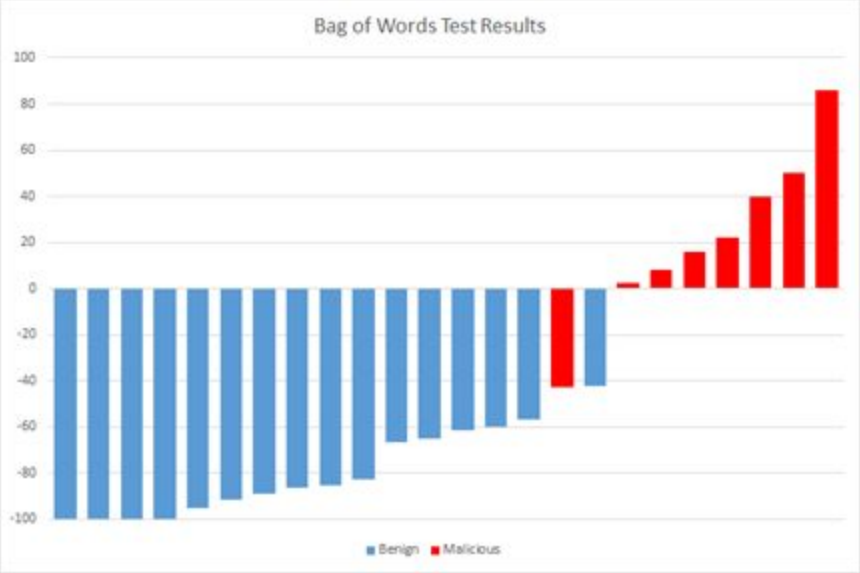
we decided not to use it. Unfortunately, Scrapy cannot properly parse a webpage if certain HTML attributes such as ‘div’ or ‘span’ are absent or have different values than expected. This is because Scrapy has to statically map all of the required HTML attributes. We found that it was infeasible to crawl GitHub with Scrapy because the HTML structure is not the same in each repository. Instead of Scrapy, we tried to use Beautiful Soup. Beautiful Soup is a Python library used for extracting data from HTML and XML files. It is supported by parsers such as lxml and html5lib. Beautiful Soup automatically converts incoming data to Unicode and outgoing data to UTF-8, which ensures that the crawler is able to parse all characters from GitHub, even if they are not in the standard ASCII character set. Beautiful Soup can find all of the links on a webpage with its ‘BeautifulSoup.findall()’ function. This allows the crawler to recursively scrape the source code from all of the subdirectories in a repository.

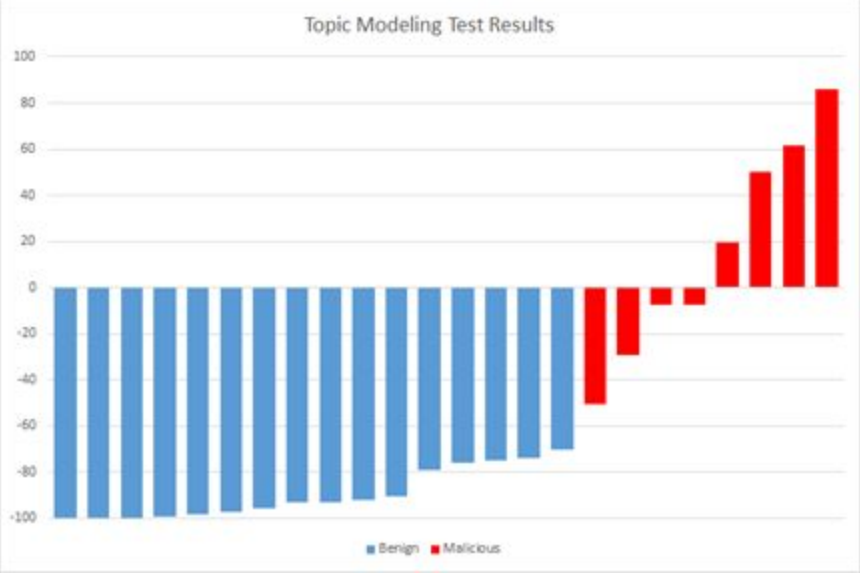
Although it was initially successful, the crawler quickly hit a roadblock due to GitHub’s rate limit. Unauthorized programs are only able to make 60 requests per hour. To increase this limit, it was necessary to obtain a GitHub OAuth token and use the GitHub API. Doing this increased our rate limit to 5,000 requests per hour. We also discovered that GitHub’s API has the functionality to list the individual URLs of all of the files in a repository and to view each file’s raw source code without having to parse any HTML. Therefore, even though we had success with Beautiful Soup, we decided to use these API calls instead.

Crawler Results

We crawled 16 random benign GitHub repositories and the 8 malware repositories that were not in our training set. The crawler used the bag-of-words, topic modeling, and Bayesian

classifiers to classify each repository. The scores from the bag-of-words and topic modeling tests are shown in the graphs below. Benign source code is represented in blue and malware source code is represented in red.

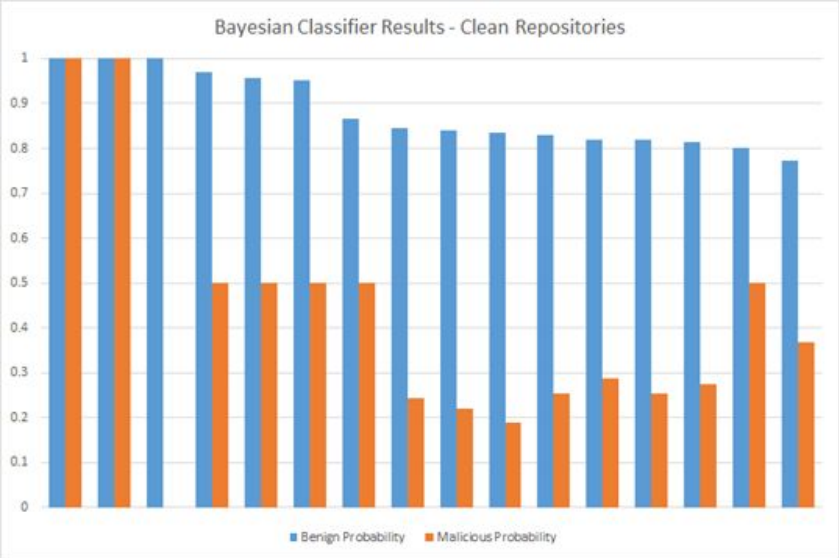


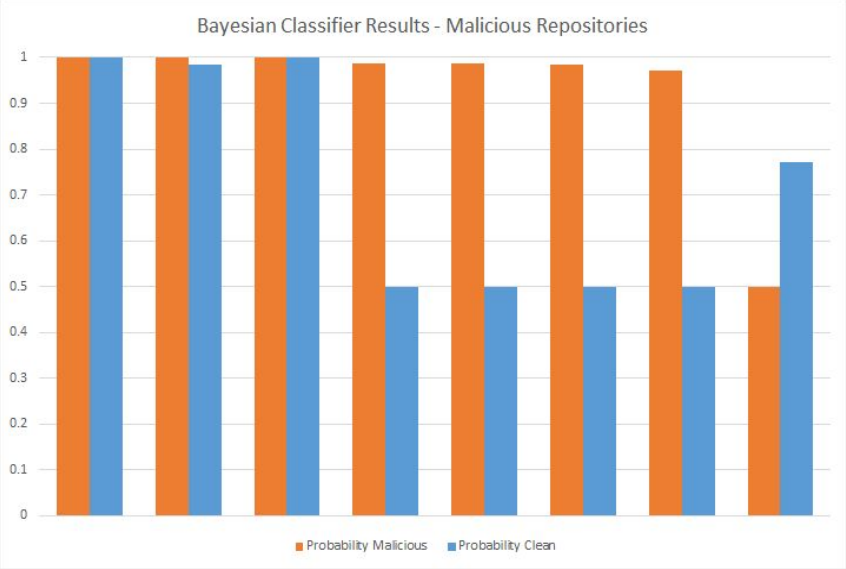


The results showed that both natural language processing classifiers seemed to have overfit themselves to the training set. However, both classifiers were still able to somewhat distinguish between malicious and benign source code. The bag-of-words classifier correctly

gave all of the benign source code a negative score and all but one of the malicious source codes a positive score. The source code of the Android malware GmBot received a score of negative 42.79, which was slightly lower than the highest-scoring clean repository. The topic modeling classifier gave more consistent scores to the benign source code than the bag-of-words classifier but it did not show significant improvement overall. Although it assigned half of the malicious GitHub repositories a negative score, it correctly separated the malicious repositories from the benign ones.

The next two charts show the results of the Bayesian classifier. The Bayesian classifier assigned each GitHub repository a probability of being malicious and a probability of being benign. The first chart shows the malicious and benign probabilities of 16 random benign GitHub repositories, where the malicious probabilities are represented in red and the benign probabilities are represented in blue. Ideally, the Bayesian classifier would assign the 16 benign GitHub repositories a benign probability near 1 and and a malicious probability near 0. The second chart shows the malicious and benign probabilities of the 8 malware source codes it was not trained on. Again, in an ideal situation, the Bayesian classifier would assign the 8 malicious GitHub repositories a malicious probability near 1 and a benign probability near 0.





The Bayes classifier showed moderate ability to distinguish between malicious and clean source code. However, many of the results were inconclusive. For example, 5 of the 24 testing repositories were given nearly the same probability of being clean and malicious. The source code of the Hidden Tear ransomware was the only GitHub repository in the testing set to be classified wrong; it was given a probability of .77 of being benign and a probability of .5 of being malicious. The Bayesian classifier classified the remaining 18 repositories correctly.

Conclusion

Unfortunately, none of the classifiers were able to reach an acceptable level of accuracy for this research to be put into production. However, all three classifiers demonstrated that they could recognize many of the features that distinguish malicious source code from clean source code. We believe that our results show promise and that this topic merits future research. Our results could possibly be improved by increasing the size of the training set, by modifying the way that files are tokenized, or by using different machine learning algorithms.

Appendix

MalCrawl source code:

- https://github.com/joyce8/MalCrawl
- Tar files containing the malware and benign training and testing sets can be provided upon request.

Scrapy Crawler:

- https://docs.scrapy.org/en/latest/

Beautiful Soup:

- https://www.crummy.com/software/BeautifulSoup/bs4/doc/

YARA rules:

- https://github.com/Yara-Rules/
- https://github.com/jipegit/yara-rules-public/blob/master/RAT/BlackShades.yar
- https://github.com/bwall/bamfdetect/blob/master/BAMF_Detect/modules/yara/blackshades.yara
- https://github.com/bwall/bamfdetect/blob/master/BAMF_Detect/modules/yara/andromeda.yara
- https://github.com/bwall/bamfdetect/blob/master/BAMF_Detect/modules/yara/dendroid.yara
- https://github.com/Yara-Rules/rules/blob/master/Mobile_Malware/Android_Dendroid_RAT.yar
- https://github.com/bwall/bamfdetect/blob/master/BAMF_Detect/modules/yara/dexter.yara
- https://github.com/Yara-Rules/rules/blob/master/Exploit-Kits/EK_Zerox88.yar
- https://github.com/Yara-Rules/rules/blob/master/Exploit-Kits/EK_Eleonore.yar
- https://github.com/Yara-Rules/rules/blob/master/Exploit-Kits/EK_Fragus.yar
- https://github.com/Yara-Rules/rules/blob/master/Exploit-Kits/EK_Phoenix.yar
- https://github.com/Yara-Rules/rules/blob/master/Exploit-Kits/EK_Sakura.yar
- https://github.com/Neo23x0/signature-base/blob/master/yara/crime_mirai.yar
- https://github.com/nheijmans/malzoo/blob/master/data/yara_rules/kins.yara
- https://github.com/Yara-Rules/rules/blob/master/malware/MALW_KINS.yar
- https://github.com/Yara-Rules/rules/blob/master/malware/RAT_Ratdecoders.yar
- https://github.com/Yara-Rules/rules/blob/master/malware/MALW_Rovnix.yar
- https://github.com/mattulm/sfiles_yara/blob/master/malware/tinba2.yar
- https://github.com/Yara-Rules/rules/blob/master/malware/RAT_Xtreme.yar
- https://github.com/keithjjones/Yara-Rules/blob/master/malware/Zeus.yar

Works Cited

Cimpanu, Catalin. "Hidden Tear Open-Source Ransomware Spawns 24 Other Ransomware Variants." Softpedia. N.p., 04 Feb. 2016. Web. 26 Apr. 2017. <http://news.softpedia.com/news/open-source-hidden-tear-ransomware-spawns-24-other-ransomware-variants-499937.shtml>.

Lee, Jihwan. "Classification Using Bayes Rule in 1-dimensional and N-dimensional Feature Spaces." Project Rhea. N.p., 2014. Web. 13 May 2017. <https://www.projectrhea.org/rhea/index.php/Bayes_Rule_for_1-dimensional_and_N-dimensional_feature_spaces>.

Ray, Sunil, Kunal Jain, Dishashree Gupta, and Ankit Gupta. "Understanding Support Vector Machine Algorithm from Examples (along with Code)." Analytics Vidhya. N.p., 13 Sept. 2016. Web. 13 May 2017. <https://www.analyticsvidhya.com/blog/2015/10/understaing-support-vector-machine-example-code/>.

Sen, Utku. "Destroying The Encryption of Hidden Tear Ransomware." N.p., 19 Nov. 2015. Web. 25 Apr. 2017. <https://www.utkusen.com/blog/destroying-the-encryption-of-hidden-tear-ransomware.html>.

Tf-idf :: A Single-Page Tutorial - Information Retrieval and Text Mining. N.p., n.d. Web. 13 May 2017. <http://www.tfidf.com/>.

Xylitol. GitHub. N.p., 3 Mar. 2014. Web. 30 Apr. 2017. <https://github.com/Yara-Rules/rules/blob/master/malware/MALW_Zeus.yar>.

"Zeus Malware: Threat Banking Industry." Unisys, May 2010. Web. 26 Apr. 2017. <http://botnetlegalnotice.com/citadel/files/Guerrino_Decl_Ex1.pdf>.