A capstone project submitted to the Department of Electrical Engineering and Computing Systems (DEECS) at the University of Cincinnati

MiBeX: Malware-inserted Benign Datasets for Explainable Machine Learning

Wayne Stegner

April 27, 2020

Submitted in partial fulfillment of the degree of
Bachelor of Science in
Computer Engineering

Wayne Stegner: _____ Date: _____

Dr. Rashmi Jha: _____ Date: _____

Acknowledgments

I would like to thank the following individuals for their contributions to this project. Without you, this project would not have been possible.

- The Design Knowledge Company, Dayton, Ohio and Air Force Research Laboratory, Wright Patterson, Ohio for funding this project under Award # AFRL FA8650-18-C-1191.
- Dr. Rashmi Jha for your advisory expertise and guidance on the direction and scope of the project.
- AFRL associates Dr. David Kapp, Dr. Tem Kebede, and Daniel Koraneck for constructive technical discussion and your guidance on the scope of the project.
- Tyler Westland for sharing your knowledge of Linux, Makefile, and helping me troubleshoot my MSFvenom issues.
- Bayley King for giving me advice on training my Malware as Image classifier.

Contents

List of Figures

List of Tables

1 Abstract

In the task of malware detection the method of machine learning has shown high accuracy in distinguishing malicious software from benign software. As malware classification using machine learning shows more promise, it is desirable to extract explanations from the algorithms to validate the reasoning behind the decisions. Santacroce et al. [1, 2] has studied the use of class saliency maps to extract features deemed malicious by malware classification neural networks. However, current malware datasets do not have specific labeled malicious features, so validating the correctness of the extracted malicious features was impractical.

This work explored a method to produce a scalable malware dataset with intelligible malicious features for the purpose of evaluating explanations from machine learning malware classifiers.

First, I collected a pool of benign executable files from the `/bin/` and `/usr/bin/` directories of a default installation of Ubuntu 18.04 64-bit, forming the benign group of the dataset. Next, I used MSFvenom from the Metasploit framework in conjunction with the `make` utility to insert a TCP bind shell Trojan payload into each benign file, forming the malware group of the dataset. Finally, I trained a convolutional neural network to classify the malware and benign files.

I collected 1103 benign files, forming a total dataset of 2206 executable files. Before training the malware classifier, files larger than 100 KB were discarded to reduce zero padding, leaving a total of 1794 files. The classifier achieved a validation accuracy of 99.72% after training for 5000 epochs. While the network was likely overtrained, I demonstrated that the dataset has a distinct set of malicious features that can be classified with machine learning. The method meets my goals for scalability and intelligibility. The method is highly scalable, as it scales directly with the amount of benign files collected. This dataset also has intelligible malicious features due to the nature of having both benign and infected versions of the same program. Due to what I suspect to be a glitch in MSFvenom, the Trojan payload is inserted into the benign file but does not run as expected.

One point of future work for my method is to resolve this issue. Presently, the Trojan is inserted at the end of the file. A second improvement is to investigate different methods of padding the end of the file to make the Trojan appear to be in the middle of the file. I also want to try increasing the diversity of the malicious files by using multiple different payloads from Metasploit. Overall, my dataset generation method has shown promise to produce intelligible, scalable datasets

2 Introduction

As machine learning becomes more prevalent, it is desirable to be able to explain the reasoning behind its decisions. In the task of detecting malicious software, machine learning shows a lot of promise to achieve high accuracy in distinguishing malware from benign software [1, 3–6]. While there are several popular datasets being used for malware detection, they all lack the ability to verify explanations of specific malicious features. Manual feature labeling of an existing dataset is a tremendously time-consuming task and leaves room for error. I have developed a method to produce scalable malware datasets with intelligible malicious features for the purpose of machine learning. I then put this method to the test by producing a dataset and training a machine leaning classifier to differentiate between the malicious and benign files.

2.1 Review of Literature

2.1.1 Explainable Artificial Intelligence

Explaining the behavior of artificial intelligence algorithms is an important task for several reasons [7,8]. Making an AI explain its decisions can help to verify proper performance of the system, as well as leading to system improvements and learning from it. Several case studies have demonstrated the practical need for explainable AI. A medical study examining risk factors of pneumonia discovered a flaw in their model by using an intelligible model [9]. The model learned the pattern that patients with asthma have a lower risk of dying than patients without asthma. While this pattern seems counterintuitive, it was an actual trend in the data because asthmatic patients with pneumonia were given more intense treatment than non-asthmatic patients. By using an intelligible model, underlying flaws were discovered and fixed.

While neural networks are inherently black-box systems, methods of understanding the reason for their decisions [10, 11]. The work in [10] investigates methods for visualizing class models and generating class saliency maps for convolutional neural networks. Class modeling involves optimizing the input image into the network to maximize the output score of a given class. The class model is generated as follows:

$$\arg\max_I S_c(I) - \lambda \|I\|_2^2 \tag{1}$$

In (1), $S_c(I)$ is the class score of class c with L_2-regularized input image I, and λ is the regularization parameter. Essentially, the class model produces the ideal image I of class c.

Also discussed in [10], class saliency maps are techniques to discover the important features in I that contribute the most to $S_c(I)$. The saliency map g_c is defined as:

$$g_c = \frac{\partial S_c(I)}{\partial I} \tag{2}$$

Intuitively, the class saliency map can be interpreted as how sensitive $S_c(I)$ is to variations of each pixel in I. Large values in g_c designate pixels that are more important to $S_c(I)$, while small values in g_c designate pixels that are less important.

2.1.2 Malware as Image

A promising method for detecting malicious programs is Malware as Image [3,5]. In essence, the technique involves representing the executable files as images and then feeding them through a machine learning classifier. The methodology can be very simple, such as the work done in [5]. In this study, the executable file is transformed into a gray-scale image by using each byte from the executable file as an 8-bit pixel brightness. The sequence of bytes is then reshaped into a 64x64 pixel image, then fed into a convolutional neural network. The network has two 3x3 convolutional layers, each followed by max pooling layers. The final layer is a fully connected layer leading into a softmax classifier. Using this methodology, [5] is able to classify a dataset of 365 malware samples with an accuracy of 94%.

In [3], a similar approach is used for transforming the executable file into a gray-scale image. However, instead of putting these images directly into a convolutional neural network, they first extract low-level features, such as intensity-based and texture-based features. These extracted features are then fed into Support Vector Machines to achieve an accuracy of 95%. The dataset in this study contained 25000 malicious files and 12000 benign files.

2.1.3 Malware as Video

Based on the Malware as Image concept, Dr. Jha's lab has previously developed a technique called Malware as Video [1]. The motivation behind this approach is to counteract one of the downfalls of Malware as Image, that is all images must be the same size to feed into the neural network. While Malware as Video is a visual representation classification technique, it addresses the size constraint by breaking the image into video frames and feeding them through a time-distributed convolutional neural network. By using this technique, the amount of padding to an image is drastically reduced, and the network is able to handle files with drastic variations in size. Using this method, Santacroce et al. [1] achieves 99.86% training and 98.74% testing accuracy on the 2015 Microsoft Malware Classification challenge dataset [6].

As a result of this high accuracy, Dr. Jha's lab desired an explanation for what features the network considers malicious. Using the class saliency technique in [10], Santacroce et al. [2] were able to generate image specific saliency maps to show which features were considered malicious and which were benign by the Malware as Video network. To validate that we had indeed extracted the important features in the inputs, we modified input videos by removing non-salient portions. Using the modified video resulted in a 99.84% training and 99.31% testing accuracy, which confirmed that we identified the truly salient portion of the code. For comparison, when removing the salient code from the input, the resulting accuracy is 66.93% for training and 66.75% for testing.

While we know that we have identified the salient code for the network, we still cannot say with certainty that the extracted portions of code are actually malicious. Due to a lack of intelligibility of the dataset, it is difficult to determine which features are malicious and which are benign. It is entirely possible that the dataset has some features which are common among malicious files, but do not actually perform any malicious action. In order to perform a more in-depth feature analysis, we require a dataset with intelligible features.

2.2 Problem Statement

This project addresses the difficulties in applying explainable machine learning to malware classification. More specifically, the problem is that there is no malware dataset with intelligible malicious and benign features. Without such a dataset, it is impossible to properly verify the correctness of feature-based explanations generated by machine learning models.

2.3 Hypothesis

I hypothesize that a custom malware dataset can be created to aid with validating the correctness of malicious features learned by machine learning algorithms.

2.4 Credibility

I am currently a Senior in the undergraduate Computer Engineering degree program, as well as a graduate student through the Accend program pursuing a MS in Computer Engineering. I primarily specialize in software, specifically Python and C/C++. My software experience includes an image processing co-op at L3, several classes at UC, and over one year of hands-on lab experience in Dr. Jha's MIND research lab. I became acclimated with the work by Santacroce et al [1], and even contributed to the saliency map work in [2], where I was a co-author published in NAECON 2019. I spent the summer researching the importance of Explainable Artificial Intelligence, and I let a symposium on the topic for my lab colleagues. To better accommodate the needs of this project, I have taken a graduate course on malware analysis, where I have gotten a better understanding of the behavior of malicious software, as well as different techniques to analyze executable binary file formats. My research has helped to earn a DAGSI grant about contextual malware analysis in avionic embedded systems through the Air Force Research Lab. I drafted the entire proposal, including the research concept and timeline, in just six days with minimal oversight.

3 Discussion

3.1 Design Objectives

The design objectives for this project can be broken into primary and secondary objectives. The primary objectives are:

- Develop a *scalable* method to generate malware datasets.
- The datasets should have *intelligible* malicious and benign features.

The rationale behind these primary objectives is that the overall goal is to use the dataset for explainable machine learning. Use in machine learning, especially deep learning, requires a large dataset to train and validate the network performance. By developing a scalable method to create malware datasets, the performance of such learning algorithms can be validated. In order to validate explanations generated by these machine learning algorithms, the dataset must have known malicious and benign features. That is, specific portions of code must definitively be labeled malicious or benign.

The secondary objectives will help to increase the overall value of this research project. The secondary goals are:

- The method should be repeatable by others.
- The method should involve minimal manual work.
- The method should be applicable to multiple different platforms (i.e. Windows, Linux, Android, etc.).

3.2 Methodology Overview

My methodology consists of two major components. First, the dataset must be generated using the Metasploit framework. Next, the dataset shall be tested using a Malware as Image classification algorithm and evaluate the explanation correctness. An overview of this methodology is shown in Figure 1.

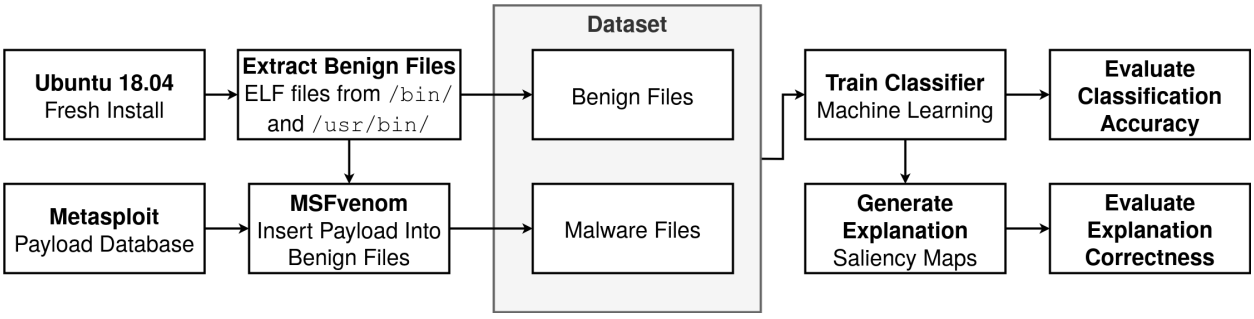


Figure 1: Flow diagram showing methodology overview.

3.3 Timeline

The timeline for this project was shown in a Gantt chart. The Fall schedule is found in Figure 2, and the Spring schedule is found in Figure 3.

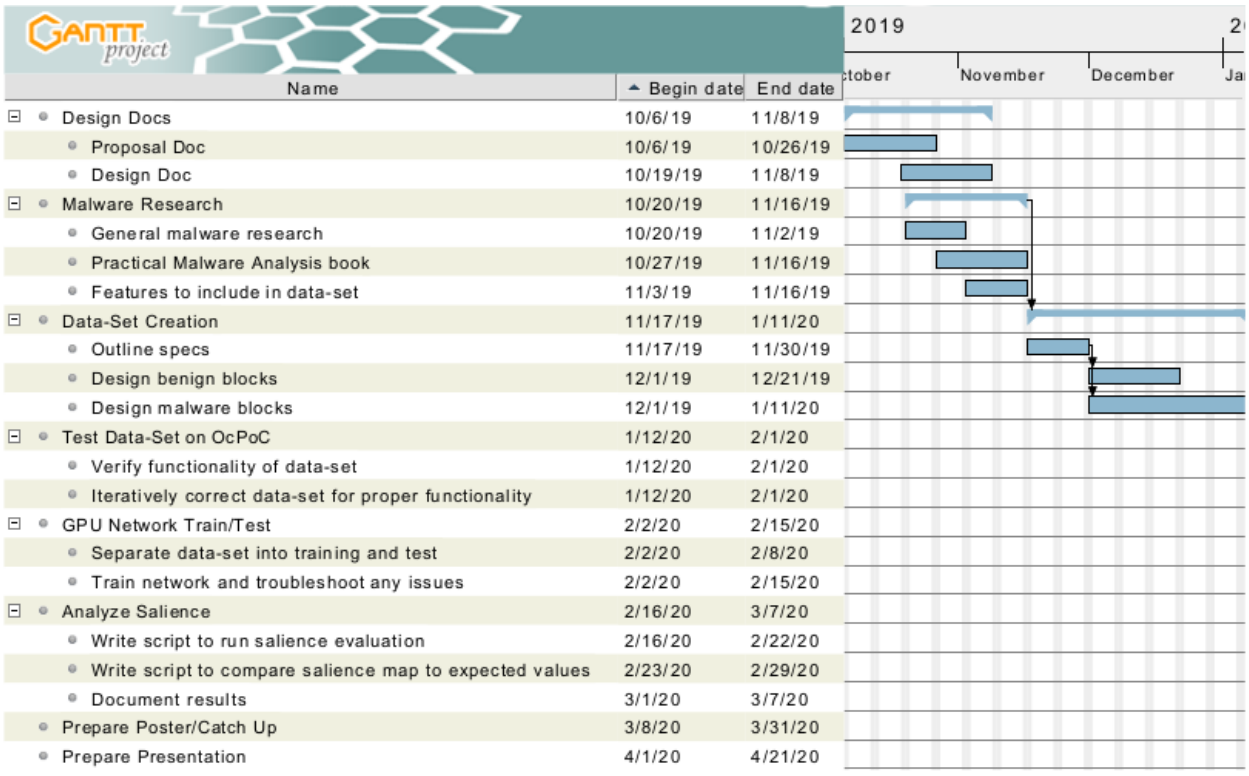
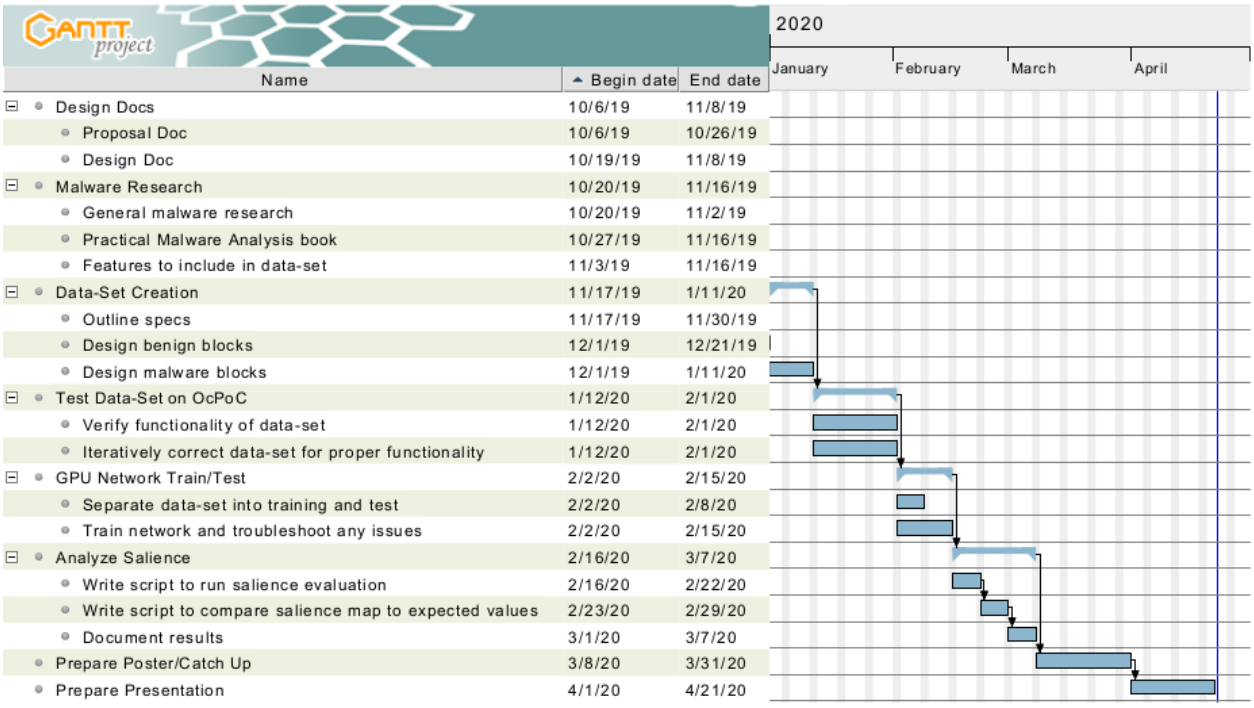


Figure 2: Gantt chart showing Fall semester schedule.



Name	Begin date	End date
Design Docs	10/6/19	11/8/19
Proposal Doc	10/6/19	10/26/19
Design Doc	10/19/19	11/8/19
Malware Research	10/20/19	11/16/19
General malware research	10/20/19	11/2/19
Practical Malware Analysis book	10/27/19	11/16/19
Features to include in data-set	11/3/19	11/16/19
Data-Set Creation	11/17/19	1/11/20
Outline specs	11/17/19	11/30/19
Design benign blocks	12/1/19	12/21/19
Design malware blocks	12/1/19	1/11/20
Test Data-Set on OcPoC	1/12/20	2/1/20
Verify functionality of data-set	1/12/20	2/1/20
Iteratively correct data-set for proper functionality	1/12/20	2/1/20
GPU Network Train/Test	2/2/20	2/15/20
Separate data-set into training and test	2/2/20	2/8/20
Train network and troubleshoot any issues	2/2/20	2/15/20
Analyze Salience	2/16/20	3/7/20
Write script to run salience evaluation	2/16/20	2/22/20
Write script to compare salience map to expected values	2/23/20	2/29/20
Document results	3/1/20	3/7/20
Prepare Poster/Catch Up	3/8/20	3/31/20
Prepare Presentation	4/1/20	4/21/20

Figure 3: Gantt chart showing Spring semester schedule.

4 Methodology

4.1 Dataset Generation

My method for dataset generation has two major goals. The first goal is for the dataset to be easily interpretable for use in explainable machine learning. In this case, interpretability means knowing which portion of the code is malicious and which is benign. The second goal is for the method to be scalable to produce high volume datasets.

To meet these goals, I developed a dataset generation method consisting of two main stages. The first stage involves gathering a pool of executable files, serving as the benign portion of the dataset. In the second stage, a malicious Trojan is inserted into each benign file, forming the malware portion of the dataset.

4.1.1 Gathering Benign Files

I wanted my method of gathering benign files to be easily repeatable. To achieve this goal, I gathered the benign files from a default installation of Ubuntu 18.04 64-bit. For convenience, the files were collected from an installation running on VirtualBox, but the process can be replicated with a physical machine. From this installation, I am interested in extracting executable binary files. In the case of Linux, the most popular file format for executable files is the Executable and Linkable Format (ELF) [12]. In the Linux filesystem, many ELF files can be found in the `/bin/` and `/usr/bin/` directories.

Figure 4 shows a bash script I wrote to scrape these directories for ELF files and copy them to the benign directories. The script calls the `find` command on both directories, which is a Unix program that searches through a given directory and outputs a list of files. The `file` command is called on each output, which displays information about the type of the file (i.e., "ASCII test" or "ELF 64-bit"). From this information, I filter out the ELF files with the `grep` command, which is a command used to print out lines of a string or file matching a particular pattern. In my case, I only want lines to display if the output of `file` contains the string `": ELF"`, designating it as an ELF file. Each file from the filtered group is copied over to the directory designated by `$BEN_DIR`, where they are stored for Trojan injection. After gathering the benign pool, each file has the extension `.benign` to distinguish from their infected counterparts.

```
for dir in /bin /usr/bin; do
  find $dir -type f -exec file {}\
  | grep ": ELF" | grep -v $BEN_DIR\
  | cut -d: -f1 | xargs cp -p -t $BEN_DIR/
done
find $BEN_DIR -type f -exec mv '{}' '{}'.benign \;
```

Figure 4: Bash script to scrape ELF files.

4.1.2 Trojan Insertion

Once the benign files are collected, the malicious dataset is constructed by inserting a Trojan payload into each file. To accomplish this task, I utilized the Metasploit [13] framework. Metasploit is a popular penetration testing framework containing payloads for various platforms, including Linux and Windows. MSFvenom, a tool from the Metasploit framework, allows these payloads to be output in a variety of formats, but I am specifically interested in ELF format. An extensive guide about the capabilities of Metasploit and MSFvenom can be found in [14].

Figure 5 shows an example command used to insert the payload into a sample file. I specify the command to target the 64-bit Linux platform with a TCP bind shell payload. This type of payload opens up a TCP port on the host machine and listens for an incoming connection from the attacker. The template argument designates a file in which to insert the payload. In this case, I insert the payload into the file `ls.benign` and output the infected file to `ls.infected`.

```
msfvenom --arch x64 --platform linux \
  --payload linux/x64/shell_bind_tcp LPORT=6666 \
  --format elf --out ls.infected \
  --template ls.benign
```

Figure 5: Example MSFvenom command call to insert a Trojan into `ls.benign`.

To automate the process of inserting the Trojan payload into every single benign file, I utilized the `make` tool [15]. It is a versatile tool with the capability to automatically determine which components of a project require recompilation. In typical usage, `make` is used in conjunction with a build tool, such as `gcc`. A set of rules in a Makefile define parameters such as input files, output files, and specific commands needed to generate output files. For my application, I treat MSFvenom as my build tool, files ending in `.benign` as my source files, and files ending in `.infected` as my output files. The Makefile rules are configured so that if a `.benign` file does not have a corresponding `.infected` file, the `.infected` file is generated by calling MSFvenom similarly to the command shown in Figure 5.

4.2 Dataset Application

In order to validate my dataset generation method, I used my dataset of 2206 files (1103 of each class) in a malware classification task. I chose to apply Malware as Image classification due to its ease of implementation. I considered using Malware as Video due to Dr. Jha's lab's previous work exploring the technique [1, 2]. However, due to the relative simplicity of implementing and analysing Malware as Image networks, I opted not to use Malware as Video for this example. Figure 6 shows an overview of the Malware as Image method, including a visualization of the network architecture. More specific network parameters will be discussed later.

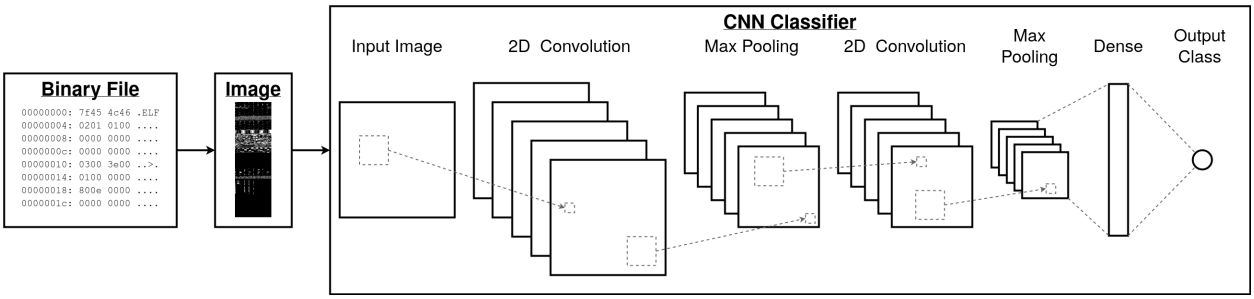


Figure 6: Overview of the Malware as Image classification model.

4.2.1 Preprocessing

To prepare the dataset for the network, the size of the image must be determined. The network requires all inputs to have the same dimensions. For simplicity, I simply pad the end of the files with zeros until they are all the same size. To reduce the issue of having too much padding, I discard any files longer than 100 KB, leaving me with 1794 usable files. Figure 7 shows the total amount of files at a given maximum file-size. I then set the image width to 32 pixels and the height to 3070 pixels.

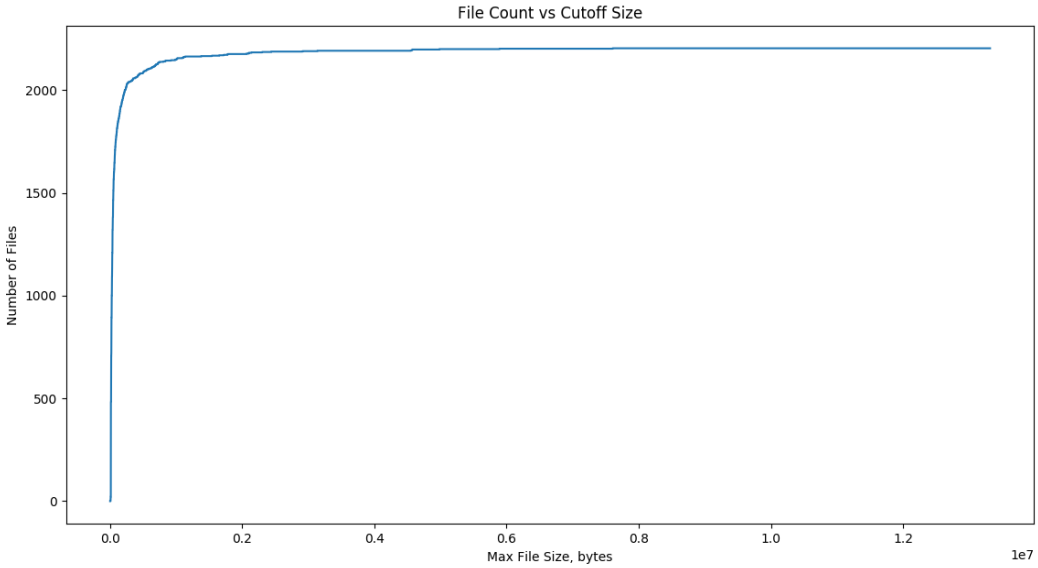
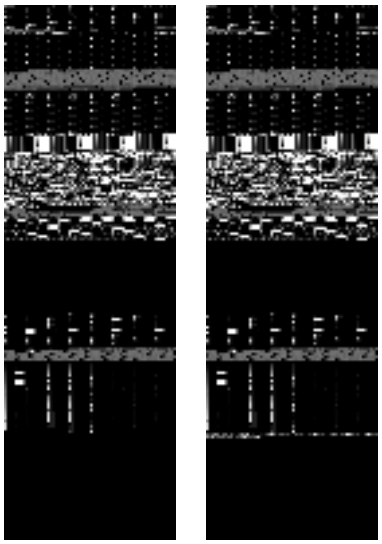


Figure 7: Graph of File Count vs Maximum File-size.

Next, the binary file must be converted into an image file using methodology similar to [5] I achieve this by first reading in the binary file as a byte array. Each byte is then cast into an 8-bit unsigned integer, giving each element of the array an integer in the range [0, 255]. The array is then reshaped to have a width of 32 and a height of 3070, giving me a gray-scale image representation of the original binary file. Note that during the reshaping process, the end of the file is padded with zeros to ensure it will fit the necessary dimensions.

Figure 8 shows examples of image files generated from `clear.benign` and `clear.infected`. Note that for space reasons, these images are 64 pixels wide and 200 pixels tall.



(a) (b)

Figure 8: Example images: (a) `clear.benign` (b) `clear.infected`.

4.2.2 Classification

To classify this dataset, I constructed a two-layer convolutional neural network similar to the one found in [5]. Table 1 shows the specific parameters for constructing my network. The dataset was partitioned 80% training data and 20% testing data. I trained the network for 5000 epochs with a batch size of 32, a learning rate of 0.001, and Binary Cross-entropy as the loss function.

Table 1: Malware Classification Network Architecture

Layer Type	**Activation**	**Notes**
2D Convolutional	tanh	Shape = 4x4
2D Max Pooling	N.A.	Shape = 3x3
Dropout	N.A.	Rate = 0.25
2D Convolutional	tanh	Shape = 3x3
2D Max Pooling	N.A.	Shape = 3x3
Dropout	N.A.	Rate = 0.25
Global Average Pooling	N.A.	N.A.
Dense	tanh	Size = 50
Dense	sigmoid	Size = 1

5 Results and Analysis

5.1 Dataset Generation

My method is able to generate a dataset containing 1103 files of each benign and infected classes, totaling 2206 files. Due to the nature of having both benign and infected versions of the same file, I am easily able to determine which features of the file are malicious and which are benign. A simple diff of the infected file to the binary file will reveal this information, and it can be entirely automated. The use of the `make` command greatly helps to improve the scalability of the method. Because the Makefile rules check which files need to be generated, it is possible to add a few files to the binary partition and have them quickly and automatically infected.

My dataset is greatly advantageous over the 2015 Microsoft Malware Classification Challenge dataset [6] in terms of intelligibility. As previously discussed, I was unable to verify the benign and malicious components of the Microsoft dataset. However, the malicious components of my dataset can easily be identified. This level of intelligibility holds great promise to system verification of malware detection systems of any type.

An interesting behavior I observed from the infected files is that they do not actually run the Trojan payload. This behavior is verified through the use of the `netstat -atn` command. The expected effect of running an infected file is for the port 6666 to open and listen for incoming connections. However, running the infected file did not open any ports. A "Trojan only" ELF file was generated by running the command from Figure 5 without the template argument. Running this file does cause the expected result of opening and listening on port 6666. As seen in Figure 8, there is a section of code in `clear.infected` that is not found in `clear.benign` at the end of the file. This code addition matches exactly with the "Trojan only" generated file, so I have confirmed that MSFvenom is inserting the actual Trojan into the template file. I currently suspect that the lack of Trojan execution from the infected files is due to a glitch in the way MSFvenom modifies the entry-point of the code. However, since the Trojan is actually inserted into the file, it can be found with static analysis.

5.2 Malware Detection

During the training process, I observed the training and validation accuracy over 5000 epochs. These observations are shown in Figure 9. After training, the accuracy is 100.0% on the training set and 99.72% on the validation set. I observe that the training and validation accuracies flatten out after about 300-500 epochs. While this result means that the network is likely overtrained, it does maintain a high validation accuracy throughout the entire duration of training. Therefore, I have demonstrated the ability to use my new dataset for the purpose of training a malware detection machine learning algorithm.

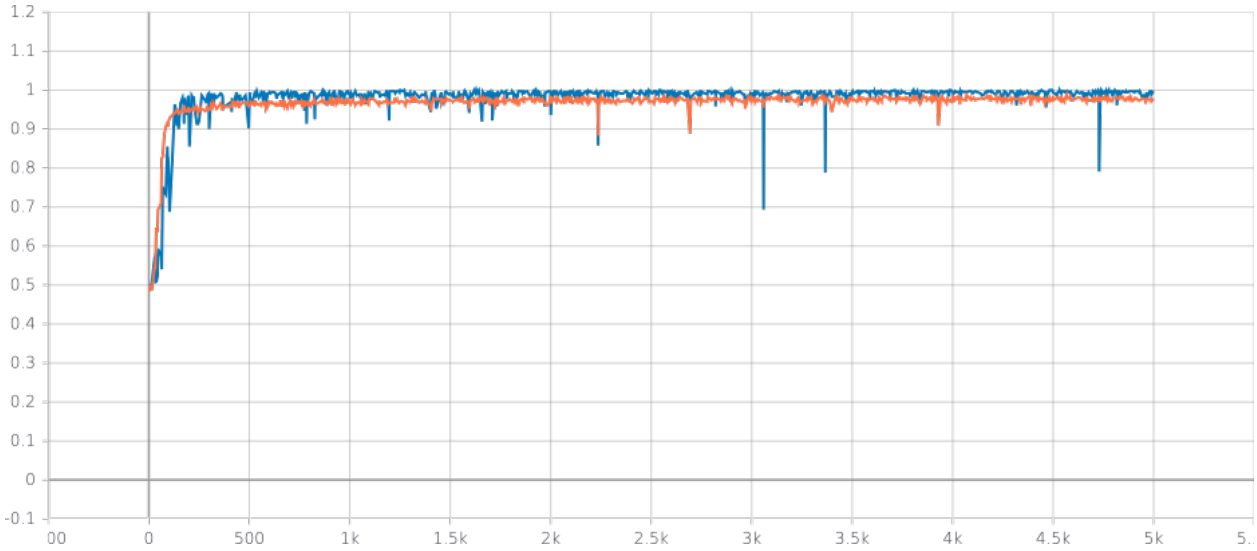


Figure 9: Training (red) and validation (blue) accuracy over 5000 epochs.

I used my network trained at 5000 epochs and measured the performance. These measurements are shown in Table 2. My network successfully classifies the entire training dataset correctly, and it only misses 1 file in the validation dataset.

Table 2: Malware Detection Results

Metric	Train Value	Test Value
True Positive	719	177
True Negative	716	181
False Positive	0	0
False Negative	0	1
Accuracy	100.0%	99.72%

6 Conclusion

My dataset generation method was able to successfully generate a dataset with features distinct enough to train a malware detection network to a testing accuracy of 99.72%. My dataset is also intelligible as to specifically which features are malicious and which are benign. Overall, my dataset generation method has shown promise to produce highly scalable datasets for machine learning. In addition to malware detection, the explainable nature of the method allows the dataset to be used for tasks such as semantic segmentation of executable files. The flexibility of the Metasploit framework allows this method to be extensible to virtually any target platform.

6.1 Problems Encountered

Throughout this project, I encountered many problems that I had to solve. While some of these problems were able to be solved, I did not have time to resolve every issue.

One problem is that the training and validation processes were using a lot of memory. The memory usage would scale linearly with the amount of files in the dataset. The reason for this issue is that initially, I am transforming all of the files into images and storing them in memory before training began. While the dataset generated for this work was a usable size, future considerations include the scalability of the dataset. To resolve this problem, I can use the tensorflow.data.Dataset API. This API allows custom mappings to be applied to datasets during training. Using this technique, I can set the dataset to initially hold a list of file locations and provided a mapping function to transform the binary files into images, meaning only a single batch of data needs to fully load into memory at a time. I have developed code to use this method in my digital image processing class, but I have not yet had time to port the solution to MiBeX.

The biggest ongoing issue is that the Trojans in the dataset do not activate. I have done some troubleshooting on the matter, but have not been able to resolve it entirely. First, I performed a binary file-diff on the benign and infected copies of the same file. I verified that the Trojan was actually being inserted, and there is a small change at the beginning of the file. I assume that the small change at the beginning is where MSFvenom changes the entry-point of the executable, but I have not verified this. Next, I generated a stand-alone payload with MSFvenom and tested its functionality. As expected with a TCP bind shell program, I used the `netstat` command to observe that port 6666 was open and listening for incoming TCP packets while running the stand-alone payload. Additionally, I verified that port 6666 does not open when running the infected file. Further testing must be done to find the exact reason why the Trojan payload does not run in the infected files.

Another unresolved problem is getting TensorFlow 2.1 to generate image-specific class saliency maps. I have found methods that other people have used to generate saliency maps with TensorFlow 2.1, but when I attempt to implement them I receive error messages about the input dimensionality. Unfortunately, I have not had time to resolve these errors.

6.2 Future Recommendations

The first future recommendation is to fix the issue with TensorFlow and calculating the saliency maps. Completing this task will help to round out the story presented by this project and fully realize the proof of concept malware detection application.

The most obvious limitation with this dataset generation method is that the Trojan does not actually run. While this limit still allows the dataset to be used for static analysis, it is completely unusable for dynamic analysis. Fixing this issue is the largest area of improvement for this method. Another interesting area of improvement is adjusting where the Trojan payload is inserted into the file. With MSFvenom, the Trojan is always inserted at the end of the file. However, investigating different padding methods might make the Trojan appear to be in the middle of the file instead of the end.

This method can be expanded upon in several ways. While this work only demonstrates the creation of a two-class dataset, MSFvenom can be configured to insert many different types of Trojan payloads. In fact, MSFvenom has the capability to use custom payloads, as well as payload obfuscation techniques. MSFvenom has the tools available to produce an extremely diverse dataset. The only limiting factor is the amount of payloads that can be found. Increasing the diversity of the types of Trojan payloads will prove beneficial to further testing the performance of various classification techniques. I can also expand upon the size of the dataset by adding more benign files. Ubuntu has a vast collection of files in its software repositories which can be freely downloaded and added to the benign set.

References

[1] M. Santacroce, D. Koranek, and R. Jha, "Exploring the Detection of Malware Code as Video with Compressed, Time-Distributed CNNs," *Submitted but Unpublished*, 2018.

[2] M. Santacroce, W. Stegner, D. Koranek, and R. Jha, "A Foray Into Extracting Malicious Features from Executable Code with Neural Network Salience," *NAECON*, 2018.

[3] K. Kancherla and S. Mukkamala, "Image visualization based malware detection," in *2013 IEEE Symposium on Computational Intelligence in Cyber Security (CICS)*, 2013.

[4] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas, "Malware Detection by Eating a Whole EXE," *arXiv*, oct 2017.

[5] J. Su, D. V. Vargas, S. Prasad, D. Sgandurra, Y. Feng, and K. Sakurai, "Lightweight Classification of IoT Malware based on Image Recognition," *arXiv*, feb 2018.

[6] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi, "Microsoft Malware Classification Challenge," *arXiv*, feb 2018.

[7] L. R. Biggers, C. Bocovich, R. Capshaw, B. P. Eddy, L. H. Etzkorn, and N. A. Kraft, "Explainable AI: understanding, visualizing and interpreting deep learning models," *Empirical Software Engineering*, vol. 19, no. 3, pp. 465–500, 2014.

[8] M. Vagg and M. Leach, "Potentially useful study of transcendental meditation fails to impress because of poor methodology," *Focus on Alternative and Complementary Therapies*, vol. 20, pp. 172–173, dec 2015.

[9] R. Caruana, Y. Lou, J. Gehrke, P. Koch, M. Sturm, and N. Elhadad, "Intelligible Models for HealthCare," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '15*, (New York, New York, USA), pp. 1721–1730, ACM Press, 2015.

[10] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps," *arXiv*, pp. 1–8, dec 2013.

[11] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization," *Proceedings of the IEEE International Conference on Computer Vision*, vol. 2017-Octob, pp. 618–626, 2017.

[12] "Executable and Linkable Format (ELF)," 2001.

[13] "Metasploit."

[14] "Metasploit Unleashed."

[15] R. Stallman, R. McGrath, and P. Smith, "GNU make."

A Code

All of the code for this project can be found at https://github.com/stegnerw/MiBeX.

A.1 Dataset Generation

A.1.1 elf_scraper.sh

```
#!/bin/sh

BEN_DIR=$(realpath $(dirname "$0"))/benign_files
mkdir -p $BEN_DIR
for dir in /bin /usr/bin; do
  find $dir -type f -exec file {}\
  | grep ": ELF" | grep -v $BEN_DIR\
  | cut -d: -f1 | xargs cp -p -t $BEN_DIR/
done
find $BEN_DIR -type f -exec mv '{}' '{}'.benign \;
```

A.1.2 Makefile

```
# Name            : Makefile
# Project         : FIX ME
# Description     : Main Makefile
# Creation Date   : Fri May 16 14:59:49 2014
# Original Author : jharwell
# Editing Authors : Tyler Westland, Wayne Stegner
#
# Note: This file is -j (parallel build) safe, provided you don't mess
   with it
# too much.
#
#  Products:
#  Make Target     Product                 Description
#  ===========     =======                 ===================
#  all             bin/$PROJECT            The main executable
#  clean           N/A                     Removes excutable, all .o

# Project Name
PROJECT = MiBeX
# Directory Definitions
# benign_files/      - Directory to store benign files
# infected_files/    - Directory to store infected files
BENIGNDIR       = ./benign_files
INFDIR          = ./infected_files

# Definitions

# Tell make we want to execute all commands using bash (otherwise it uses
# sh). make generally works best with bash, and as SHELL is inherited from
    the
# invoking shell when make is run, it may have a value like sh, tcsh, etc.
    If
```

```
# you don't do this, then some shell commands will not behave as you
# expect. This is in keeping with the principle of least surprise.
SHELL           = bash

# MSFvenom Compilation Options

# Specify the flags to use when compiling (or injecting Trojan payloads).
define MSFFLAGS
-a x64 --platform linux -p linux/x64/shell_bind_tcp LPORT=6666 -f elf
endef

# Define the "compiler" to use, in this case MSFvenom
MSF         = msfvenom

# Functions
# Recursive wildcard: search a list of directories for all files that
   match a pattern
# usage: $(call rwildcard, $(DIRS1) $(DIRS2) ..., pattern)
#
# All directory lists passed as first arg must be separated by spaces, and
    they
# themselves must be space separated as well. There must NOT be a space
   between
# the last dir list the , and the start of the pattern.
#
# You should never need to modify this.
# usage: $(call rwildcard, $(DIRS1) $(DIRS2) ..., pattern)
rwildcard=$(foreach d,$(wildcard $1*),$(call rwildcard,$d/,$2)  $(filter $
   (subst *,%,$2),$d))

# make-depend: generate dependencies for source files dynamically. Very
   useful
# for including .h files as target dependencies.
# usage: $(call make-depend,benign-file,infected-file,depend-file)
#
# You should never need to modify this.
# usage: $(call make-depend,benign-file,infected-file,depend-file)
make-depend-cxx=$(MSF) -MM -MF $3 -MP -MT $2 $(MSFFLAGS) $1

# Target Definitions
# Define what directories to search for benign files. For us, this will
   just
# be a single source directory, benign_files/.
BENIGNS = $(BENIGNDIR)

# Define the list of files to compile for this project, which is built by
# recursively finding all .cc files in benign_files/.
SRC_MSF = $(call rwildcard,$(BENIGNS),*.benign)

# For each of the .cc files found under benign_files/, determine the name
   of the
# corresponding .o file to create in infected_files/ via pattern
   substitution (patsust).
INFECTED_MSF = $(notdir $(patsubst %.benign,%.infected,$(SRC_MSF)))
```

```
# All targets

# Phony targets: targets of this type will be run everytime by make (i.e.
    make
# does not assume that the target recipe will build the target name)
.PHONY: clean veryclean all documentation

# The default target which will be run if the user just types "make" with
    a
# target name
all: $(addprefix $(INFDIR)/, $(INFECTED_MSF)) | $(INFDIR)

# The Objectifier. This rule says that each file in infected_files/,
    depends on the
# presence of the infected_files/ directory. This is necessary so that
    parallel make
# (make -j) works.
$(addprefix $(INFDIR)/, $(INFECTED_MSF)): | $(INFDIR)


# Bootstrap Bill. This creates all of the order-only prerequisites; that
    is,
# files/directories that have to be present in order for a given target
    build
# to succeed, but that make knows do not need to be remade each time their
# modification time is updated and they are newer than the target being
    built.
$(INFDIR):
  @mkdir -p $@

# The Cleaner. Clean up the project, by removing ALL files generated
    during
# the build process to build the main target.
clean:
  @rm -rf $(INFDIR)

# Pattern Rules
$(INFDIR)/%.infected: $(BENIGNDIR)/%.benign
  $(MSF) $(MSFFLAGS) -o $@ -x $<
```

A.2 Malware as Image Classifier

A.2.1 file_size_analysis.py

```
import os
import matplotlib.pyplot as plt

# Relevant directory and file definitions
proj_dir        = os.path.dirname(os.path.realpath(__file__))
benign_dir      = proj_dir + '/Dataset_Generation/benign_files/'
infected_dir    = proj_dir + '/Dataset_Generation/infected_files/'

# Get list of files
```

```
file_names = []
benign_files = os.listdir(benign_dir)
for f in benign_files:
    file_names.append(os.path.join(benign_dir, f))
infected_files = os.listdir(infected_dir)
for f in infected_files:
    file_names.append(os.path.join(infected_dir, f))

file_sizes = [os.path.getsize(f) for f in file_names]

num_files = []
file_thresh = []
#max_size = max(file_sizes) + 1
max_size = 100000
size_step = 10
for size in range(0, max_size, size_step):
    if size % (size_step*500) == 0:
        print('size: ', size, '/', max_size, ' - ', size/max_size * 100, '
            %')
    num_files.append(len([s for s in file_sizes if s <= size]))
    file_thresh.append(size)

plt.plot(file_thresh, num_files)
plt.xlabel('Max File Size, bytes')
plt.ylabel('Number of Files')
plt.title('File Count vs Cutoff Size')
plt.show()

file_sizes = [s for s in file_sizes if s <= max_size]
print('Remaining files:', len(file_sizes))

n, bins, patches = plt.hist(x=file_sizes, bins=20, color='#0504aa', alpha
    =0.7, rwidth=0.85)
plt.grid(axis='y', alpha=0.75)
plt.xlabel('File Size, bytes')
plt.ylabel('Number of Files')
plt.title('Distribution of File Sizes')
plt.show()
```

A.2.2 FileImage.py

```
import os
import numpy as np
import math
import cv2

class FileImage:
    def __init__(self, file_path, height=100, width=25):
        # Check if file exists
        if not os.path.isfile(file_path):
            raise FileNotFoundError('Could not load binary because file
                does not exist')
        self.file_path = file_path
        self.file_size = os.path.getsize(file_path)
```

```
        self.setImageSize(height, width)

        # Load binary file
        self.file_contents = np.empty(self.file_size, dtype='uint8')
        if not os.path.isfile(self.file_path):
            raise FileNotFoundError('Could not load binary because file
                does not exist')
        with open(self.file_path, 'rb') as f:
            idx = 0
            new_byte = f.read(1)
            while new_byte:
                self.file_contents[idx] = int.from_bytes(new_byte,
                    byteorder='big')
                idx += 1
                new_byte = f.read(1)

    def setImageSize(self, height, width):
        if (width > 0):
            self.img_width = width
        if (height > 0):
            self.img_height = height

    def getImage(self):
        img = np.zeros(self.img_width * self.img_height, dtype='uint8')
        for i in range(min(self.file_size, img.size)):
            img[i] = self.file_contents[i]
        img = img.reshape((self.img_height, self.img_width))
        return img


    def showImage(self):
        img = self.getImage()
        cv2.imshow(self.file_path, img)
        cv2.waitKey()
        cv2.destroyAllWindows()

if __name__ == '__main__':
    f = FileImage('Dataset_Generation/benign_files/ls.benign', 512, 256)
    f.showImage()
```

A.2.3 partition_dataset.py

```
import os
import random
import json

# Constant Definitions
TRAIN_PERCENT    = 80

# Relevant directory and file definitions
proj_dir         = os.path.dirname(os.path.realpath(__file__))
benign_dir       = proj_dir + '/Dataset_Generation/benign_files/'
infected_dir     = proj_dir + '/Dataset_Generation/infected_files/'
train_json       = proj_dir + '/train_set.json'
```

```
test_json       = proj_dir + '/test_set.json'

# Get list of files
max_file_size = 100000
file_names = []
benign_files = os.listdir(benign_dir)
for f in benign_files:
    file_names.append(os.path.join(benign_dir, f))
infected_files = os.listdir(infected_dir)
for f in infected_files:
    file_names.append(os.path.join(infected_dir, f))
file_names = [f for f in file_names if os.path.getsize(f) <= max_file_size
   ]
random.shuffle(file_names)

# Partition files
train_files = []
test_files  = []
first_test  = (len(file_names) * TRAIN_PERCENT) // 100
for fn in file_names[:first_test]:
    if os.path.splitext(fn)[1] == '.benign':
        fclass = 'benign'
    else:
        fclass = 'malicious'
    train_files.append({
        'file_name':    fn,
        'class':        fclass
    })

for fn in file_names[first_test:]:
    if os.path.splitext(fn)[1] == '.benign':
        fclass = 'benign'
    else:
        fclass = 'malicious'
    test_files.append({
        'file_name':    fn,
        'class':        fclass
    })

# Shuffle partitions
random.shuffle(train_files)
random.shuffle(test_files)

# Export json
with open(train_json, 'w') as trainf:
    json.dump(train_files, trainf)

with open(test_json, 'w') as testf:
    json.dump(test_files, testf)

print('Done')
```

A.2.4 pickle_dataset.py

```
import os
import json
import pickle
import numpy as np
import cv2
from FileImage import FileImage

# Constant declarations
IMG_WIDTH   = 32
CLASSES     = ['benign', 'malicious']
NUM_CLASSES = 2

# Relevant directory and file definitions
proj_dir        = os.path.dirname(os.path.realpath(__file__))
train_json      = proj_dir + '/train_set.json'
test_json       = proj_dir + '/test_set.json'
train_pickle    = proj_dir + '/train_set.pickle'
test_pickle     = proj_dir + '/test_set.pickle'

def jsonToPickle(json_path, pickle_path, height, width):
    # Import json
    with open(json_path) as json_fd:
        json_contents = json.load(json_fd)

    # Create dictionaries to store values
    pickle_contents = {
            'file_name':    [],
            'image':        [],
            'class':        []
    }

    for f in json_contents:
        pickle_contents['file_name'].append(f['file_name'])
        img_obj = FileImage(f['file_name'], height, width)
        # cv2.imshow(f['file_name'], img_obj.getImage())
        # cv2.waitKey()
        # cv2.destroyAllWindows()
        pickle_contents['image'].append(img_obj.getImage())
        for i in range(NUM_CLASSES):
            if (f['class'] == CLASSES[i]):
                pickle_contents['class'].append(float(i))
    # Reshape input into np array to make tensorflow happy
    pickle_contents['image'] = np.array(pickle_contents['image']).reshape
       ((-1, height, width, 1))
    # pickle_contents['image'] = np.array(pickle_contents['image']).
       reshape((-1, height, width))
    pickle_contents['class'] = np.array(pickle_contents['class']).reshape
       ((-1, 1))

    # Save pickle file
    pickle.dump(pickle_contents, open(pickle_path, 'wb'))

# Calculate image height
max_size = 0
```

```
with open(train_json) as f:
    data = json.load(f)
    max_size = max(max_size, max([os.path.getsize(elem['file_name']) for
       elem in data]))
with open(test_json) as f:
    data = json.load(f)
    max_size = max(max_size, max([os.path.getsize(elem['file_name']) for
       elem in data]))
print(max_size)
IMG_HEIGHT = (max_size // IMG_WIDTH) + 1

print('Pickling train set...')
jsonToPickle(train_json, train_pickle, IMG_HEIGHT, IMG_WIDTH)
print('Pickling test set...')
jsonToPickle(test_json, test_pickle, IMG_HEIGHT, IMG_WIDTH)

print('Done')
```

A.2.5 train_cnn_v1.py

```
import os
import cv2
import numpy as np
import pickle
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras import models
from tensorflow.keras import losses

# Relevant directory and file definitions
proj_dir        = os.path.dirname(os.path.realpath(__file__))
train_pickle    = os.path.join(proj_dir, 'train_set.pickle')
test_pickle     = os.path.join(proj_dir, 'test_set.pickle')
out_dir         = os.path.join(proj_dir, 'networks/cnn_v1')

# Load train/test data
print('Loading train/test data...')
train       = pickle.load(open(train_pickle, 'rb'))
test        = pickle.load(open(test_pickle, 'rb'))
img_shape   = train['image'][0].shape
print(img_shape)

# Limit TF memory usage
physical_devices = tf.config.experimental.list_physical_devices('GPU')
assert len(physical_devices) > 0, "Not enough GPU hardware devices
   available"
config = tf.config.experimental.set_memory_growth(physical_devices[0],
   True)

# Define network model
print('Constructing network model...')
model = models.Sequential()
model.add(layers.Conv2D(50, (4, 4), padding='same', activation='tanh',
```

```
    input_shape=img_shape))
model.add(layers.MaxPool2D(pool_size=(3, 3)))
model.add(layers.Dropout(0.25))
model.add(layers.Conv2D(25, (3, 3), padding='same', activation='tanh'))
model.add(layers.MaxPool2D(pool_size=(3, 3)))
model.add(layers.Dropout(0.25))
model.add(layers.GlobalAvgPool2D())
model.add(layers.Dense(50, activation='tanh'))
model.add(layers.Dense(1, activation='sigmoid'))
model.compile(loss=losses.BinaryCrossentropy(), optimizer='Adam', metrics
    =['accuracy'])

model.fit(train['image'], train['class'], batch_size=32, verbose=1,
    validation_data=(test['image'], test['class']), epochs=1000, shuffle=
    True)

model.save(out_dir)

print(model.evaluate(test['image'], test['class'], verbose=2))
```