Machine Learning Classification of Obfuscation using Image Visualization

Colby B. Parker, J. Todd McDonald[a] and Dimitrios Damopoulos[b]
Department of Computer Science, University of South Alabama, Mobile, AL, U.S.A.

Keywords: Software Protection, MATE Attacks, Code Visualization, Neural Networks, Obfuscation.

Abstract: As the need for new techniques to analyze obfuscated software has grown, recent work has shown the ability to analyze programs via machine learning in order to perform automated metadata recovery. Often these techniques really on disassembly or other means of direct code analysis. We showcase an approach combining code visualization and image analysis via convolutional neural networks capable of statically classifying obfuscation transformations. By first turning samples into gray scale images, we are able to analyze the structure and side effects of transformations used in the software with no heavy code analysis or feature preparation. With experimental results samples produced with the Tigress and OLLVM obfuscators, our models are capable of labeling transformations with F1-scores between 90% and 100% across all tests. We showcase our approach via models designed as both a binary classification problem as well as a multi label and multi output problem. We retain high performance even in the presence of multiple transformations in a file.

1 INTRODUCTION

Obfuscation involves the process of transforming a given program into one that is syntactically different but semantically equivalent (Collberg and Nagra, 2009). This new obfuscated program now has its inner workings, code and/or data, changed so that they are hidden and difficult to understand to attackers, both human and machine. Obfuscation is an important security tool for intellectural property protection, but is also employed by malware authors. Deobfuscation exists to remove transformations so that further analysis can proceed . From a protection viewpoint, an adversary could employ metadata recovery attacks which reveals the type of obfuscation used (Salem and Banescu, 2016). This information then fuels other techniques, meaning speedy and effective metadata recovery methods are of high importance (Coogan et al., 2011).

When applied to a program, different transformations exhibit varying degrees of complexity (Tofighi-Shirazi et al., 2019). By studying these side effects, one can associate patterns with different transformations (Jones et al., 2018) (Banescu et al., 2016). By profiling these unique side effects, it is possible to create a classifier using machine learning which can identify what transformations were used (Tofighi-Shirazi et al., 2019). This kind of analysis can be used to expedite and enhance software analysis and metadata recovery attacks (Salem and Banescu, 2016).

It has been shown that a piece of software can be transformed into a gray scale image (Nataraj et al., 2011). Prior work using binary visualization has mostly focused on determining if a given binary is malware or not, with extended work on labeling malware families. We extend this further by using binary visualization and Convolutional Neural Networks (CNN) to label transformations implemented in a binary to hinder reverse engineering and analysis. Our work showcases not only a more fine-grained analysis, but also can be used to label protections in non-malware binaries (legitimate software). To the best of our knowledge, no one has performed this kind of analysis using binary visualization.

In this paper, we show how images of obfuscated binary code can be analyzed by machine learning algorithms to derive features based on spatial relationships. Our tests frame this as both a binary problem through the use of many individual models as well as a multi-label problem via a single model. Both approaches are shown to be highly effective at classifying a transformation in a program, even in the presence of layered transformations.

[a] https://orcid.org/0000-0001-5266-7470
[b] https://orcid.org/0000-0002-7193-0710

Parker, C., McDonald, J. and Damopoulos, D.
Machine Learning Classification of Obfuscation using Image Visualization.
DOI: 10.5220/0010607408540859
In *Proceedings of the 18th International Conference on Security and Cryptography (SECRYPT 2021)*, pages 854-859
ISBN: 978-989-758-524-1

Contributions. We make the following contributions:

- We show how visualization of binary code and the use of CNN enable metadata recovery attacks on obfuscated programs, with no reverse engineering required.
- We show effectiveness of supervised learning models trained on images of obfuscated programs at classifying transformations, with F1-scores $>$ 90% for both binary and multi-label classification across a range of transformations.
- We demonstrate binary image analysis provides higher granularity and F1-scores $>$ 90% for samples with layers of obfuscating transformations.
- We show our image analysis technique is effective against well known obfuscators such as Tigress [1] and OLLVM (Junod et al., 2015).

2 BACKGROUND

In this section we briefly discuss the basics of software obfuscation and introduce the obfuscating transformations used in our work. We then explain concepts related to supervised machine learning and the area of code visualization.

2.1 Obfuscating Transformations

Obfuscation transformations are traditionally divided into three categories: 1) *layout*, focuses on making source-code unreadable; 2) *data*, focuses on replacing data structures; and 3) *control*, manipulates branch structures. It is possible for transformations to be classified as dynamic, which implies that code changes at run-time (Schrittwieser et al., 2016). Typical transformations (Collberg and Nagra, 2009) include:

- **Virtualization:** creates a process-level virtual environment by constructing an interpreter in the form of a large switch statement with a unique instruction set.
- **Control-flow Flattening:** combines branch statements into one large seemingly infinite loop controlled by a dispatcher
- **Just-in-Time Compilation:** adds dynamic compilation to the transformed program that replaces target statements in the original program with method calls unique to that statement.
- **Encoding:** obscures static values within a program via parameterized functions instead of clear text to help avoid pattern matching.
- **Opaque Predicates:** An if-then-else statement that always evaluates to the same true or false condition, crafted in order to confuse static analysis.

[1] http://tigress.cs.arizona.edu/

2.2 Supervised Machine Learning

Supervised Learning is the category of machine learning that deals with the analysis of labeled data, with the goal of correctly labeling previously unseen samples (Sutskever and others., 2014). This is accomplished using a mapping function $f(X) = Y$ where X is an unlabeled input of the kind previously analyzed and Y is the correct label for X (Murphy, 2012). The function is found by using a machine learning algorithm to train a model on the data. Proper learning will approximate f to allow the correct labeling of any new sample. When the labels being predicted represent classes this task is known as classification. The number of classes will further specify a task as binary classification (classes = 2) or multi-class classification (classes $>$ 2). It is also possible in certain tasks for inputs to belong to more than one class, resulting in multi-label classification (Bishop, 2006). Machine learning classification of images involves capturing spatial relationships of the pixels (Albawi et al., 2017). While there are many ways to achieve this, Convolutional Neural Networks (CNN) have become a popular choice. CNNs were designed for image analysis in mind and create information from image pieces, called convolutional layers, in order to understand the whole (Sainath et al., 2013). CNN-based image analysis has also been used widely for malware detection (Kabanga and Kim, 2017) (Kalash et al., 2018).

2.3 Code Visualization

Since the pixel of a gray-scale image is a value between 0 and 255, it is possible to translate a file into an image by having the bytes of the file become the pixels of the new image. From this newly formed image, it is possible to visually see certain features, patterns, and structures that are present in the source file (Seok and Kim, 2016). For instance, when used on a program such as a Windows binary, the sections of the program (.text, .data. etc.) are distinct from each other. Previous work has shown that code images can be used for malware classification (Vasan et al., 2020). Fig. 1 is an example of what a typical image made from an executable file will look like.

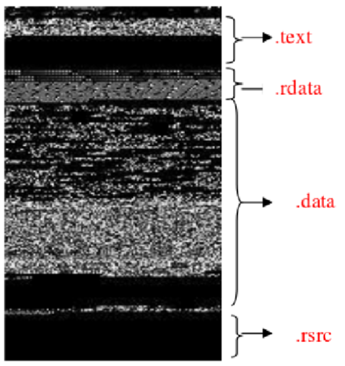


Figure 1: A windows executable converted to gray-scale.

3 METHODOLOGY

In this section we will walk through the steps of our methodology. To train and test our models, we first create a data set of images produced from both clean and obfuscated programs. The obfuscation of our samples is accomplished with the Tigress and OLLVM obfuscators. We apply one or more transformations to each program, grouping the into samples with a single transformation and those with two or more. This is to allow us to perform multi-label classification. We have our own python code to then take in the programs and produce the needed images. With our two data sets created, we move to the design and training of our CNN. Multiple version of our CNN our trained to to be used in different experiments that will be explained in Section 4.

3.1 Obfuscation Images

Using existing data sets of C programs, we use obfuscators to apply various transformations in order to create a large set of obfuscated programs. The application of obfuscation is done in two sets. For the first, we apply only single transformations to the clean files. Every file is obfuscated with each transformation to produce a number of variants equal to the number of transformations. Then, for the second set, we once again obfuscate the clean samples, but this time performing multiple transformation layered on top of each other. This produces a number of variants equal to the number of chosen permutations. Once all the obfuscated samples are produced, we compile the samples and use a Python script in order to create our obfuscation images. As mentioned before, this process is done by reading in a file byte by byte and creating a .png gray-scale image, whose pixels are formed from each consecutive byte.The size of a generated image is based on the size of the binary in question. Any pixels in an image that do not have a corresponding byte in the binary will be left a black pixel (a zero). For our images, the width of the image is based on the data length of the input file, while the height is obtained from the file size and width. Since we tested and trained on small samples, this produced images of a similar size. It would be possible to fix one of these values, resulting in images of a constant height or width. Our width ranges are based on the ones given in (Seok and Kim, 2016). Once all images have been created, we form two data sets. One contains images with either none or only a single layer of obfuscation. The second contains images with either none or a permutation of obfuscations.

3.2 Convolutional Neural Network

For our supervised learning model, we choose to use a CNN, as high accuracy has been observed from CNN when classifying images made from malware samples (Nataraj et al., 2011). For our model architecture, we choose to use a small model consisting of two convolutional layers which feed into two dense layers. This is because models with this architecture and others similar to it have been shown to be proficient at classifying the Malimg (Bensaoud et al., 2020; Mallet, 2020) data set without requiring high degree of resources. The specifics of our model can be seen in Fig. 2.

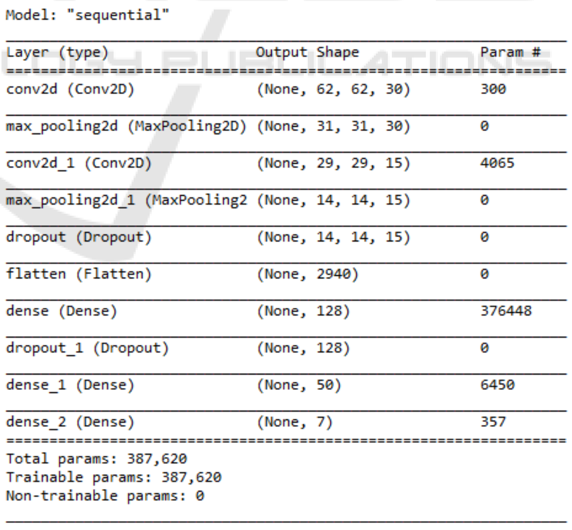

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 62, 62, 30)        300
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 31, 31, 30)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 29, 29, 15)        4065
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 15)        0
_________________________________________________________________
dropout (Dropout)            (None, 14, 14, 15)        0
_________________________________________________________________
flatten (Flatten)            (None, 2940)              0
_________________________________________________________________
dense (Dense)                (None, 128)               376448
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 50)                6450
_________________________________________________________________
dense_2 (Dense)              (None, 7)                 357
=================================================================
Total params: 387,620
Trainable params: 387,620
Non-trainable params: 0
_________________________________________________________________
```

Figure 2: Architecture of our CNN.

Our input layer is shaped to take in the pixel values of our images as opposed to extracting some feature or aspect of the whole image. This is to see how much information can be obtained without a high degree of preproccesing or prior analysis. Since images from various binaries can vary in size, all images are resized to 64*64 before being given into the model.

3.3 Multi-label Classification

We frame our classification problem multiple ways using the types described in Section 2. We do this to show the overall robustness of code image analysis, as well as to see which method achieves the best results

Binary-classification. Classification of obfuscating transforms becomes a binary problem by having a classifier only detect a specific transformation. This requires a unique classifier for each transform, but allows the potential for the model to better learn specific features of the transformation without requiring large amounts of data or a larger network. This enhanced understanding could result in high accuracy even on multi-layered samples. Our CNN can become a binary classifier by reducing the number of nodes in the output layer to 2. Then, by grouping the data sets by type, our binary CNN can be used across both data sets (Tofighi-Shirazi et al., 2019).

Multi-class and Multi-label. Both Multi-Class and Multi-label classification is accomplished via training one model to detect multiple classes of transformations (Tsoumakas and Katakis, 2009). The difference lies in the number of transformations present on the input files. If samples are only obfuscated with one transform, then the model would only have to label the one transform. This is a Multi-Class model. However, when given the Multi-Layered samples which each have more than one transformation, the model must now identify all transformations in use. This is a Multi-Label problem (Murphy, 2012). Multi-Class CNN can be easily used for Multi-Label problems by making use of the sigmoid activation function in the output layer (Sutskever and others., 2014). This allows us to test our model on both the single and multi layer data sets.

4 EXPERIMENTS

To evaluate the effectiveness of image analysis for obfuscation detection, we perform a number of experiments on both our single and multi layer data sets. These experiments are:

1. Binary classification of obfuscating transforms on single and multi layer samples.
2. Multi class classification of obfuscating transforms on single layer samples.
3. Multi label classification of transforms on multi layer samples.

For training and evaluating the models in our experiments, we make use of traditional k-fold cross validation with 10 folds as well as the functionality-based validation approach from in Salem et al. related work. Models were evaluated using the F1-Score to take into account false positives and negatives produced by the models.

4.1 Data Sets and Environment

The C programs used to create our obfuscated variants are the same as the ones used in (Banescu et al., 2017) and consist of:

1. A set of 48 programs with few lines of code constructed by varying code characteristics such as: symbolic inputs, depth of control flow, amount of loops, etc.
2. Programs automatically generated by the RandomFuns transformation of the Tigress C Diversifier/Obfuscator.
3. Non-cryptographic hash functions
4. Algorithms taught in Bachelor level computer science and programming courses.

This brings us to a data set of 5,136 source c files. After producing the obfuscated variants with Tigress and OLLVM, our data set consists of over 100,000 images split between both data sets. A list of the commands for our obfuscators and the permutations used for the multi layered samples will be made available, along with the code used for our experiments. Transformations from both obfuscators have parameters which when altered change how the transformation is applied. These were varied during sample generation to ensure that there is variance for specific transformations within the data set.

All experiments were performed on a desktop computer running Windows 10 with an AMD Ryzen 7 3700X processor and an Nvidia 2070 Super GPU. The Keras python package with a Tensorflow back-end was used for the creation and training of our CNN. An Ubuntu VM was used for running Tigress.

4.2 Transformation Classification

We first focus on transformations from the Tigress obfuscator, followed by OLLVM. We then evaluate the ability of our models to classify all transformations present in our data set. We begin each phase first with samples containing a single transformation, before moving to multiple transforms. All models were evaluated using our two approaches and can be seen in the relevant tables. Kfold results are in black while the functional results are in red. Clean samples are those with no obfuscations present

Table 1: F1-Scores for all Tigress transformations.

Transform	Single		Layered	
	Binary Label	Multi-Class	Binary Label	Multi-Label
Flatten	99.9%/99.8%	99.8%/99.9%	99.3%/99.3%	98.9%/96.5%
Opaque Predicate	99.8%/99.9%	99.9%/99.9%	99.7%/99.6%	99.6%/99.5%
Virt	99.9%/100%	100%/100%	99.9%/99.9%	99.9%/99.7%
Jit	100%/100%	100%/100%	100%/100%	99.9%/100%
EncodeA	99.5%/99.8%	99.7%/99.9%	99.1%/98.7%	98.6%/97.2%
EncodeL	99.8%/100%	99.9%/99.9%	99.8%/99.7%	99.7%/98.6%
Clean	100%/99.8%	99.8%/100%	99.9%/99.9%	99.9%/99.7%
Average F1-Score	99.8%/99.9%	99.9%/99.9%	99.6%/99.6%	99.5%/98.7%

Tigress. A set of binary models and a multi-class model were created and evaluated for this portion. Both types of models performed comparably, with both achieving f1-scores above 99% for all transformations. Both evaluation methods produced similar results. Samples with multiple transforms were evaluated next, with both types of models again achieving similar scores. The added complexity of layered transformations only slightly impacted classification, with only some transformations seeing a 1 - 2% decrease to score. Table 1 contains the results for these tests.

OLLVM. Similar to previous, our models were able to classify the OLLVM transforms accurately across single and multi layer samples and both evaluation methods, with only marginal differences. The exception is 'sub' transform, which achieved a low f1-score in the multi layered samples due to a hugh number of false positives. Table 2 shows the F1-scores for our models.

All Transformations. Table 3 shows the results for the final test. Training the models across all transformations had very limited to no impact on the performance of the models. We chose to combine the Bcf and Fla transformations from OLLVM with the Flattening and Opaque Predicate transforms from Tigress to view the impact on classification. Both the binary and multi model performed extremely well, both maintiaining high scores. This implies that, at least for our data set, transformations can be classified accurately even if the model is expected to identify a broad range.

4.3 Limitations

We acknowledge limitations in the experiments performed. Since we make use of the same data used in other papers, we inherit the flaws of the data, namely that it consists only of small programs and may not

Table 2: F1-Scores for all OLLVM transformations.

Transform	Single		Layered	
	Binary Label	Multi-Class	Binary Label	Multi-Label
Fla	99.3%/99.2%	99.8%/99.7%	99.4%/99.4%	97.1%/94.9%
Bcf	99.8%/99.8%	99.9%/99.9%	99.7%/99.9%	99.8%/99.8%
Sub	99.9%/99.7%	99.6%/98.8%	94.8%/96.4%	75.3%/77.7%
Clean	99.9%/99.8%	99.9%/100%	100%/99.9%	99.9%/100%
Average F1-Score	99.7%/99.6%	99.8%/99.6%	98.5%/98.9%	93.0%/93.1%

be a true reflection of the larger scope of software. However, the features of these programs would be present in larger samples, allowing our image analysis to still be applicable and accurate. Another potential limitation is that as programs become larger, so do the images created from them. Larger images would become more computationally expensive to analyze and train on. However, this can be compensated for by extracting parts of the image for analysis. Dynamic obfuscation methods could potentially evade our method, however it is possible image analysis could be used to detect the portions of the sample responsible for dynamically altering the code. This has been shown to be possible via other methods.

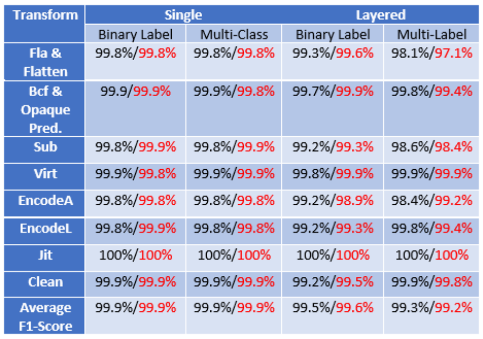
5 RELATED WORK

(Salem and Banescu, 2016) demonstrated that an ML model could be effectively trained to perform metadata recovery on an obfuscated program, instead of using manual reverse engineering to identify transformations used. They proposed that since transformations leave uniquely identifiable side effects in programs, it would be possible to use ML to detect these changes. (Tofighi-Shirazi et al., 2019) furthered metadata recovery to account for the layering of multiple obfuscations in a single program, allowing detection of multiple transformations types versus one. Their fine-grained approach was achieved via semantic analysis of code segments. Both of these approaches have the same end goal as our method of image analysis; however, our method does not rely on any direct analysis of the program.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we demonstrated effectiveness of a supervised learning approach based on code visualization for classifying obfuscating transformations. The experiments performed in this work represent an initial study and proof of concept to determine if visualization is viable for this extended analysis and to serve as the basis for more in depth work. Even on samples

Table 3: F1-Scores for all transformations.



Transform	Single		Layered	
	Binary Label	Multi-Class	Binary Label	Multi-Label
Fla & Flatten	99.8%/99.8%	99.8%/99.8%	99.3%/99.6%	98.1%/97.1%
Bcf & Opaque Pred.	99.9/99.9%	99.9%/99.8%	99.7%/99.9%	99.8%/99.4%
Sub	99.8%/99.9%	99.8%/99.9%	99.2%/99.3%	98.6%/98.4%
Virt	99.9%/99.8%	99.9%/99.9%	99.8%/99.9%	99.9%/99.9%
EncodeA	99.8%/99.8%	99.8%/99.8%	99.2%/98.9%	98.4%/99.2%
EncodeL	99.8%/99.9%	99.8%/99.8%	99.2%/99.3%	99.8%/99.4%
Jit	100%/100%	100%/100%	100%/100%	100%/100%
Clean	99.9%/99.9%	99.9%/99.9%	99.2%/99.5%	99.9%/99.8%
Average F1-Score	99.9%/99.9%	99.9%/99.9%	99.5%/99.6%	99.3%/99.2%

with multiple transformations present, a CNN trained on such data was able to properly label the transformations. Across all testing, in only one instance did any of our models perform below a 90% F1-score and most tests performed at > 95 F1-score. We conclude that code visualization offers a potentially powerful adversarial means for classifying program protection types, without the need for reverse engineering or symbolic execution. Use and expansion of this avenue of analysis could greatly enhance applicability of the technique to other avenues of metadata recovery attacks and software analysis. In showing that image analysis can be used to classify obfuscating transformations, we believe that there many directions that this work can be taken in. One potential avenue is to explore the granularity of our classification by testing if image analysis can detect features of obfuscating transform deeper than simply the type. It is also worth exploring if image analysis be used to label the portions of the image that correspond to transformed code. That capability would assist even further in reverse engineering and analysis. The perceived limitation of file size could also be explored in this way, as if sections of an image could be labeled as obfuscated instead of a whole image, this would allow the use of sub image searching. This would allow large programs to be broken into many smaller images, requiring less intensive analysis.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation under the CyberCorps Scholarship for Service program, grant DGE-1564518.

REFERENCES

Albawi, S., Mohammed, T. A., and Al-Zawi, S. (2017). Understanding of a convolutional neural network. In *ICET'17*.

Banescu, S., Collberg, C., et al. (2016). Code obfuscation against symbolic execution attacks. In *ACSAC '16*.

Banescu, S. et al. (2017). Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning. In *USENIX SEC'17*.

Bensaoud, A., Abudawaood, N., and Kalita, J. (2020). Classifying malware images with convolutional neural network models. *CoRR*, abs/2010.16108.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg.

Collberg, C. and Nagra, J. (2009). *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional.

Coogan, K., Lu, G., and Debray, S. (2011). Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *ACM CCS '11*.

Jones, L., Christman, D., Banescu, S., and Carlisle, M. (2018). Bytewise: A case study in neural network obfuscation identification. In *CCWC '18*.

Junod, P., Rinaldini, J., Wehrli, J., and Michielin, J. (2015). Obfuscator-llvm – software protection for the masses. In *SPRO '15*.

Kabanga, E. K. and Kim, C. H. (2017). Malware images classification using convolutional neural network. *J. of Comp. and Com.*, 6(1).

Kalash, M. et al. (2018). Malware classification with deep convolutional neural networks. In *NTMS '18*.

Mallet, H. (2020). Malware classification using convolutional neural networks.

Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. The MIT Press.

Nataraj, L., Karthikeyan, S., Jacob, G., and Manjunath, B. S. (2011). Malware images: Visualization and automatic classification. In *VizSec '11*.

Sainath, T. N., Mohamed, A.-r., Kingsbury, B., and Ramabhadran, B. (2013). Deep convolutional neural networks for lvcsr. In *ICASSP '13*.

Salem, A. and Banescu, S. (2016). Metadata recovery from obfuscated programs using machine learning. In *SSPREW '16*.

Schrittwieser, S., Katzenbeisser, S., et al. (2016). Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Comput. Surv.*, 49(1):4:1–4:37.

Seok, S. and Kim, H. (2016). Visualized malware classification based-on convolutional neural network. *J. of The Korea Inst. of Info. Sec. & Crypt.*, 26(1).

Sutskever, I. and others. (2014). Sequence to sequence learning with neural networks. In *NIPS '14*.

Tofighi-Shirazi, R., Asavoae, I. M., and Elbaz-Vincent, P. (2019). Fine-grained static detection of obfuscation transforms using ensemble-learning and semantic reasoning. In *SSPREW '19*.

Tsoumakas, G. and Katakis, I. (2009). Multi-label classification: An overview. *Int. J. Data WH & Mining*, 3.

Vasan, D. et al. (2020). Imcfn: Image-based malware classification using fine-tuned convolutional neural network architecture. *Computer Networks*, 171.