Experiments on Recognition of Malware based on Static Opcode Occurrence Distribution

Jacob Carlson and Anca Ralescu
University of Cincinnati

Table of Contents

Contributions

This report will propose a novel approach to generating executable file parameters for static malware classification using machine learning algorithms. Distributions will be created by focusing on patterns of occurrence of specific opcodes, then comparing distributions to baseline distributions with Kullback-Leibler Divergence. These comparisons will then be used to make classification decisions.

Proposed Plan

Static Malware analysis is an approach in which an executable is inspected for malware without being executed. The most common methods include matching common patterns or deriving a series of parameters to learn from each file, such as the Ember Data set [1]. This paper proposes a method of static malware analysis in which an executable will be translated into a set of distributions that will be compared against ground truth distributions from both benign and malicious code samples. These comparisons will then be used to classify whether a file is malicious or benign.

All compiled code exists as a sequence of op codes, instructing the machine on what to do at the most basic level. While the sequence in which the opcodes are written is not the exact same order in which they will be executed, it is still a very strong representation of the program at hand. To create a set of distributions for an executable, the number of operations between occurrences of the same opcode will be obtained for each common opcode. This data is then placed into bins and divided by the total occurrences to create a distribution of the occurrences of each tracked opcode.

This distribution can then be compared to ground truth distributions using the Kullback-Leibler divergence to determine the difference between distributions. These KL divergence values will then be used as input data into a machine learning model to classify a file as benign or malicious.

Data Acquisition

To train a network, a large dataset was needed. Most online datasets for malware detection do not provide raw files in their datasets, especially at scale; however, Practical Security Analytics, an online security blog, has released a dataset of 200,000+ benign and malicious executables for machine learning purposes [3].

This analysis focuses on the instructions that a machine will execute, so Unix command "`objdump -d`" was used to disassemble the executable into a human readable form that lists the instructions as they would be received by the operating system executing them. This form provides more than the opcode being executed, so extra information such as the line number and the parameters of the instruction must be removed. Each instruction follows a common form, so python was used to filter each entry down to its opcode. To remove the risk of the executable being used and simplify data access, opcode lists will then be stored and used for future operations instead of the executable form.

Distribution Construction

Single Executable Construction

In this method, the final goal is to have a set of distributions that can be compared to ground truth distributions to give insights into the nature of the file. Distributions are created by taking all the jumps from a given op code, *op,* and aggregating it in *b* bins using the bin assignment function ϕ to discretize the data. A jump is the number of instructions invoked between two opcodes of the same type. Given the executable file has been converted to an opcode sequence, the indices of each opcode in the list can be used to determine the jumps between successive op codes.

$$bin\ assignment = \phi(floor\left(\frac{jump}{1000} * bin_{size}\right))$$

Since executables are of variable length and it is possible for a jump to be close to the length of the entire executable if occurrences are sparse, the jump values were capped at 1000 to reduce extreme outliers. These large jumps are infrequent, so keeping them would skew the distributions and subsequent comparisons to such distributions. Any jump value whose bin assignment falls outside the number of bins is omitted.

Distribution Algorithm

```
1. For opcode $op$
    a. Construct a list of the indices at which $op$ occurs in the executable's opcode
       sequence, $freqs_{op}$.
    b. Define $distribution_{op}$ as vector of zeros the size of the number of bins.
    c. For each occurrence of $op$, $op_k$, in $freqs_{op}$,
        i. Calculate the jump value between $op$ occurrences, $jump_{op\_k} = op_{k+1} - op_k$.
           This will result in len($freqs_{op}$) -1 jump values.
        ii. For each jump $jump_{op\_k}$, calculate its bin assignment via $\phi$ and increment
            the count of that bin in $distribution_{op}$.
    d. Divide the value of each bin by the total number of jumps, len($freqs_{op}$) – 1, to
       create a probability distribution for $op$.
```

<u>*Distribution Aggregation*</u>

The process to create aggregated distributions is similar to the algorithm for individual files, with the exception that the distribution is aggregated over a large number of files of the same class. This creates a distribution set representative of what a typical jump distribution of either a benign or malicious file should look like.

Aggregated Distribution Algorithm

```
1. For opcode $op$
    a. Define $distribution_{op}$ as vector of zeros the size of the number of bins.
2. Over $N$ files of class $c$
    a. For opcode $op$
        i. Construct a list of the indices at which op occurs, $freqs_{f\_op}$.
        ii. For each occurrence of $op$, $op_{f\_k}$, in $freqs_{f\_op}$,
            1. Calculate jump value between ops, $jump_{f\_op\_k} = op_{f\_k+1} - op_{f\_k}$. This
               will result in len($freqs_{f\_op}$) -1 jump values.
            2. For each jump $jump_{f\_op\_k}$, calculate its bin assignment via $\phi$ and
               increment the count of that bin in $distribution_{op}$.
3. For opcode $op$
    a. Divide the value in each bin by the sum of all bins in $distribution_{op}$ to create a
       probability distribution of $op$ in $c$.
```

Ten ground truth samples were created for each class, each sample is aggregated over 500 total files and no file was in used in multiple ground truth distributions. To show that there was some form of similarity between distributions of the same class, heatmaps were created to show comparisons between all ground truth samples. Each cell displays the sum of the Kullback-Leibler divergence between two ground truth sets over all the distributions in the sets. This does not completely prove that the distributions are similar, but if the value of the sum is low, then that should show that the set of distributions are more similar than if the value is high. This could be skewed by an exceptionally different set, but this is just a proof of concept. Figure 1 shows heatmap examples of two different opcode sets. Both examples show two distinct sets of low KL Divergence, one grouping of benign ground truth distributions and another grouping of malicious ground truth sets.

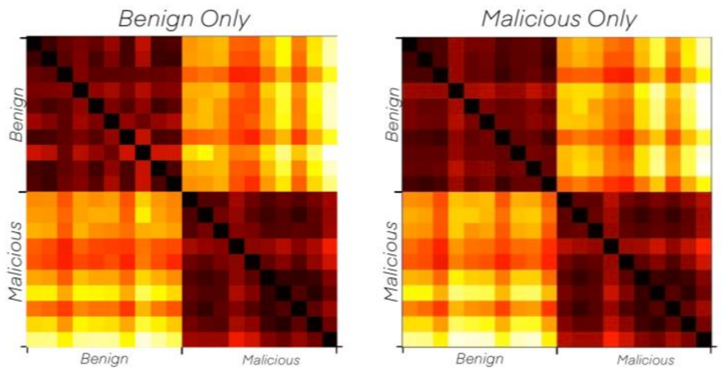

Figure 1. Benign and Malicious Heatmaps
Heatmaps for 20 ground truth samples, 10 of each class. Cells of a darker color indicated lower Kullback-Leibler Divergence values, while cells of a lighter color indicate higher KL Divergence values.

The values obtained from comparing a sample to each executable to both the ground truth distributions over all opcodes will generate a vector of values twice the size of the number of distributions. This will be used as the input to machine learning models for each sample.

Controllable Features

There are a four of controllable parameters that were focused on to show the validity of this process. The number of bins and model used the train the data, do not provide any novel insights so the parameters are not fully optimized. The results of these parameters are also deterministic and are consistent regardless of the selections made to the final parameters. The remaining parameters, opcode set selection and Kullback-Leibler divergence method, were the focus of this work got more thorough testing.

Testing Process

A consistent testing process will be used through this report. A set of 2,000 evenly weighted benign and malicious files will be used to train a model, with 500 samples used for testing. There is overlap in files used between different train and test sets, but the train and test files are never those that were used to create the ground truth sample they are compared to, and no test files are ever in the training set that trained the model they are tested against. From this data we can determine an average accuracy of the model, while the Wilcoxon Signed Rank Test will be used to compare parameter selections, under *a=0.05*.

Granularity of Distributions

One of the controllable factors within the process of creating distributions is the number of bins that the data is placed into. As the number of bins increases, the level of information in the data will increase since there are larger number of comparisons to be made but the chance of misses, or empty bins, also increases. Empty bins will yield higher, and most likely inaccurate, KL Divergences that may skew future predictions.

Two bin sizes were chosen arbitrarily, and all data was tested with distributions made up of 25 and 100 bins. When accuracies are compared over the same set of op codes, with the same KL Divergence method and model, 100 bins was more accurate. This was verified using the Wilcoxon Sign Rank test, where the data is 25-bin accuracy is subtracted from the 100-bin accuracy. The test resulted with Z >5 no matter the parameter set. More variations of bin size could have been tested to find the most optimal number of bins, but the goal was the prove that jump distributions worked so this was not further investigated.

Model Selection

Different models were tested, under the same input, to test the potential of using jump distributions to classify malicious code samples. Multiple models were used throughout the testing off all opcode sets and KL divergence method combinations. The python SciKit Learn framework was used to implement models, since it offers quick implementation and reduces risk of manual error. The Linear SVM model was used as a ground truth due to simplicity. Both Logistic Regression and the Multi-Layer Perception, MLP, Network were more accurate than the Linear SVM model. When Logistic Regression and MLP Network were compared pairwise, under the same stipulations, the MLP network was more accurate (p=1.0).

Model	Z	p
Ridge Regression	-1.14	0.13
Stochastic Gradient Descent	-46.13	0.00
Logistic Regression	13.34	1.00
Multi-Layer Perceptron Network	48.89	1.00

Figure 2. Probability of Model being more accurate than Linear SVM
Z scores and Probabilities of listed model being more accurate than a Scikit Learn's Linear SVM when tested with a Wilcoxon Signed Rank test, using the differences between the accuracies of samples tested on the same data under the same conditions.

Scaling the data before inputting it into the model proved to result in a more accurate model, when each model was compared its unscaled results. The scaled MLP network was also compared to all other scaled models, and no model was determined to be more accurate.

Model	Z	p
Linear SVM	11.73	1.00
Ridge Regression	23.36	1.00
Logistic Regression	8.22	1.00
Multi-Layer Perceptron Network	9.44	1.00

Figure 3. Probability of Scaled Models being more accurate than non-Scaled Models
Z scores and Probabilities of model being more accurate when data is scaled using SciKit Learns Standard Scaler than when data is not scaled, tested with a Wilcoxon Signed Rank test using the differences between the accuracies of samples tested on the same data under the same conditions.

Model	Z	p
Linear SVM	-50.76	0.00
Stochastic Gradient Descent	-55.51	0.00
Ridge Regression	-51.59	0.00
Logistic Regression	-49.21	0.00

Figure 4. Probability of Scaled Models being more accurate than Scaled Multi-Layer Perceptron Network
Z scores and Probabilities of model being more accurate than a scaled Multi-Layer Perceptron Network with a Wilcoxon Signed Rank test, using the differences between the accuracies of samples tested on the same data under the same conditions.

The Scaled Multi-Layer Perceptron Network was the most accurate under all parameter sets. This is not a novel finding, as it had a much higher level of learnable parameters than the model it was compared against. The MLP Network had three hidden layers ([100, 200, 50]) and ran for 300 iterations, the rest of the parameters were unchanged from SciKit Learn's default parameters. These parameter changes were decided purely because they were round numbers and larger than the original input. These attributes are most likely not optimal, and the model could be refined to increase the overall accuracy of the process, but similar to testing the granularity of the distribution, refining accuracy through standard means is not the intention of this work.

Op Code Selection

The executables in the training set are all from the machines running the Windows operating system, but their opcode sets vary and a total of 1,457 different op codes were found in the training set. Only 1.9% of those op codes occur in more than 25% of the sample set, this means that most of the opcodes will not make a difference when being used to make comparisons between files. When an opcode does not occur, the distribution would just be uniform since there were no occurrences to derive jumps from. Consistently comparing uniform distributions to the ground truth distributions would not provide any meaningful information, so an effort was made to find an optimal and commonly occurring set of opcodes.

Benign, Malicious

Simple opcode sets were derived by taking the 50 most occurring opcodes from different sample sets of benign and malicious files only. This ensured that when creating distributions for individual executables, there was a good chance that most of the op codes had enough occurrences to yield a meaningful distribution. 86% of the opcodes overlapped between the two sets, so the distribution sets share most of the same data. This reduces the chances of an opcode from one set being infrequent when compared against either opcode set. The heatmaps for both samples, Figure 2, show a consistent pattern of similarity between ground truth samples of this opcode set. Both heatmaps look alike due to the strong overlap in sets.

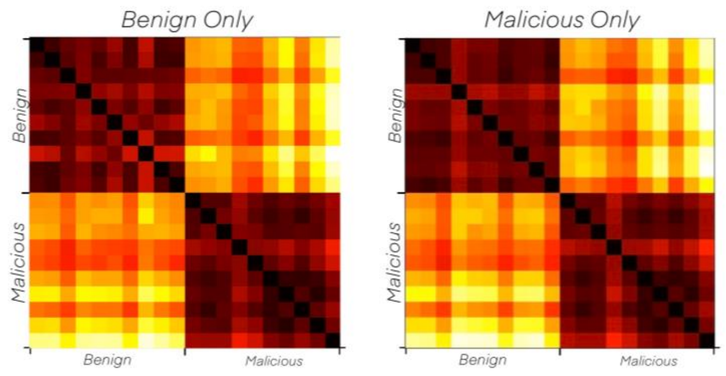

Figure 5. Benign and Malicious Heatmaps
Heatmaps for 20 ground truth samples, 10 of each class. Cells of a darker color indicated lower Kullback-Leibler Divergence values, while cells of a lighter color indicate higher KL Divergence values.

Benign	Malicious
$88.1\% \pm 1.1\%$	$88.7\% \pm 1.7\%$

Figure 6. Accuracy of Benign and Malicious Sets
Average test accuracies and standard deviations are obtained using ten different samples ground truth sample pairs, tested 4 times each using a different random seed each time to differ the data. Resulting in 40 different test cases.
Bins: 100, KL Divergence Method: KL(X||dist), Model: MLP Scaled

Using a Signed Wilcoxon Rank Test, where the input data is the difference between the accuracies under the same conditions (*Bins: 100, KL Divergence Method: KL(X||dist), Model: MLP Scaled*), the malicious set was shown to be more accurate model (Z=3.34, p=1.00).

Union, Intersection, Disjoint

Before using more in-depth approaches, the union, intersection, and disjoint of the benign and malicious opcode sets were used to make comparisons. The union and intersection of the benign and malicious op code sets only differ slightly from the from the original sets, since those sets were so similar. The benign and malicious seats each have seven opcodes that do not belong to the other set, so the benign and malicious union set has 57 opcodes and the intersection set has 43 opcodes. These sets both provide a valuable insight since the union set has more opcodes, which are frequent to at least one set, while the intersection set has fewer opcodes, but they are all frequent to both opcode sets.

Benign – Intersection: `[abcb, cmpw, jns, lock, movzwl, sbbb, sete]`
Malicious – Intersection: `[call, jmpl, leave, popl, retl, sarl, shll]`

When using the previously constructed sets the goal was to find opcodes that occurred frequently enough to create a strong distribution, but the idea behind the disjoint set was to potentially use the lack of distribution to illustrate difference. Opcodes that are common to malicious files but not benign files, and vice versa, were combined. This is illustrated in the disjoint heatmap, which does not follow as clear of a pattern as the previous sets heatmaps. There is a lower more consistent summed KL divergence between the malicious ground truth samples, while the Benign samples seems to be more iatric.

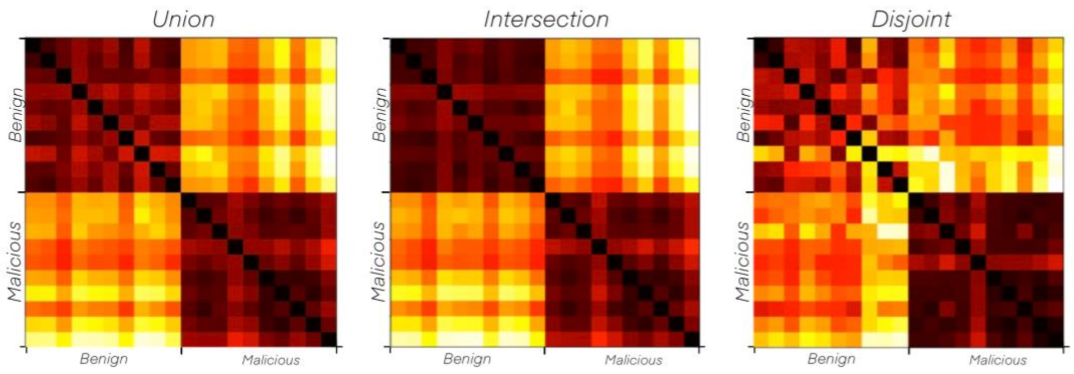

Figure 7. Union, Intersection, Disjoint Heatmaps
Heatmaps for 20 ground truth samples, 10 of each class. Cells of a darker color indicated lower Kullback-Leibler Divergence values, while cells of a lighter color indicate higher KL Divergence values.

Union	Intersection	Disjoint
$88.6\% \pm 1.6\%$	$88.0\% \pm 1.3\%$	$84.9\% \pm 1.6\%$

Figure 8. Accuracy of Union, Intersection, Disjoint Sets
Average test accuracies and standard deviations are obtained using ten different samples ground truth sample pairs, tested 4 times each using a different random seed each time to differ the data. Resulting in 40 different test cases.
Bins: 100, KL Divergence Method: KL(X||dist), Model: MLP Scaled

Opcode Set	Z	p
Union	-0.34	0.366
Intersection	-3.28	0.001
Disjoint	-5.28	0.000

Figure 9. Probability of Opcode set yielding higher accuracy than Malicious Opcode Set
Z scores and Probabilities of opcode sets being more accurate than the malicious opcode set when tested with a Wilcoxon Signed Rank test, using the differences between the accuracies of samples tested on the same data under the same conditions.
Bins: 100, KL Divergence Method: KL(X||dist), Model: MLP Scaled

Opcode Set	Z	p
Intersection	-2.28	0.002
Disjoint	-5.39	0.000

Figure 10. Probability of Opcode set yielding higher accuracy than Union Opcode Set
Z scores and Probabilities of opcode sets being more accurate than the union opcode set when tested with a Wilcoxon Signed Rank test, using the differences between the accuracies of samples tested on the same data under the same conditions.
Bins: 100, KL Divergence Method: KL(X||dist), Model: MLP Scaled

The union opcode set was more accurate than the intersection and disjoint set but failed to be significantly more accurate the malicious opcode set. The malicious opcode set is a subset of the union opcode set, this means that the seven opcodes that are a part of the benign set and not the malicious set seem to not provide any meaningful information and decrease the accuracy of the process. While the intersection opcode set does not have the benign opcodes that seemingly decreased the accuracy of the union opcode set, it has a lower accuracy than both the malicious and union opcode sets. Even though the differences in accuracy are minimal, the opcodes that belong to the malicious opcode set seem to be the reason for increased accuracy.

The disjoint opcode set is significantly smaller than any of the other opcode sets that has been tested thus far. Disjoint had an interesting heatmap, showing strong similarity in the malicious opcode set, but had low accuracy. This could be due to the small sample size, leaving the possibility that opcode sets that are common to only class of opcode could yield meaningful information.

Ratio

The disjoint set had the lowest accuracy of the models so far but still provided some chance that infrequent opcodes could yield valuable information. Finding opcodes that occurred at varied degrees of frequencies between classes showed that it could work, but the disjoint set was very small, so it needed more data to be tested accurately. The goal of the ratio opcode set was to determine the sets of opcodes which are common to one set but not the other.

Since there needs to be enough opcode occurrences in both classes to create distributions, then there needs to be a balance between opcodes that occur frequently in only one class with opcodes that occur frequently enough to create a distribution. A measure was created to determine the viability of an opcode, using the sum of the total occurrences of an opcode and ratio of opcode occurrences between classes.

$$\psi = \left|\left(benign_{freq} + malicious_{freq}\right) + \log\left(benign_{freq} \,/\, malicious_{freq}\right)\right|$$

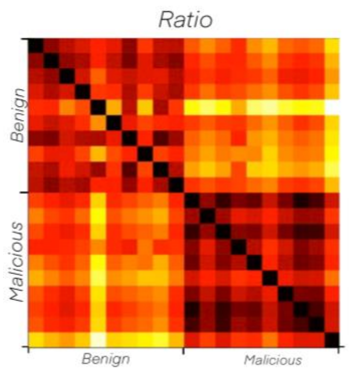

Figure 11. Ratio Heatmap
Heatmaps for 20 ground truth samples, 10 of each class. Cells of a darker color indicated lower Kullback-Leibler Divergence values, while cells of a lighter color indicate higher KL Divergence values.

Ratio
$87.4\% \pm 2.1\%$

Figure 12. Ratio Opcode Set Accuracy
Average test accuracy and standard deviation are obtained using ten different samples ground truth sample pairs, tested 4 times each using a different random seed each time to differ the data. Resulting in 40 different test cases.
Bins: 100, KL Divergence Method: KL(X||dist), Model: MLP Scaled

The ratio opcode sets failed to yield two distinct ground truth sets and did not perform as well as the current leading opcode set, malicious only. This function captured opcodes with high combined frequency and difference between class frequencies, but there was no way to accurately determine whether the two measures are combined optimally. Adding a variable to control the weights of the two values allows the function to obtain different sets.

$$\psi = \left|a\left(benign_{freq} + malicious_{freq}\right) + (1-a)\log\left(benign_{freq} \,/\, malicious_{freq}\right)\right|$$

Setting the parameter, a, to 0.5 will result in the same opcode set obtained before the variable was added. Varying a resulted in two stable sets, opcode sets were constant from a: [0.0, 0.45] and [0.6, 1.0]. When a is 0.0, which results in the same set as when a is any value from 0.0 to 0.45, then ψ only focuses on the ratio of benign to malicious files. This primarily get opcodes that only occur in one class and not the other. A small value is added to both the frequencies of both classes to allow for division when one value is zero.

When a is 1.0, which results in the same set as when a is any value from 0.6 to 1.0, them ψ only focuses on the combined frequency between both classes. This is like union set which combines frequent opcodes from each class, so much so that the a=1.0 ratio opcode set is a subset of the union opcode set.

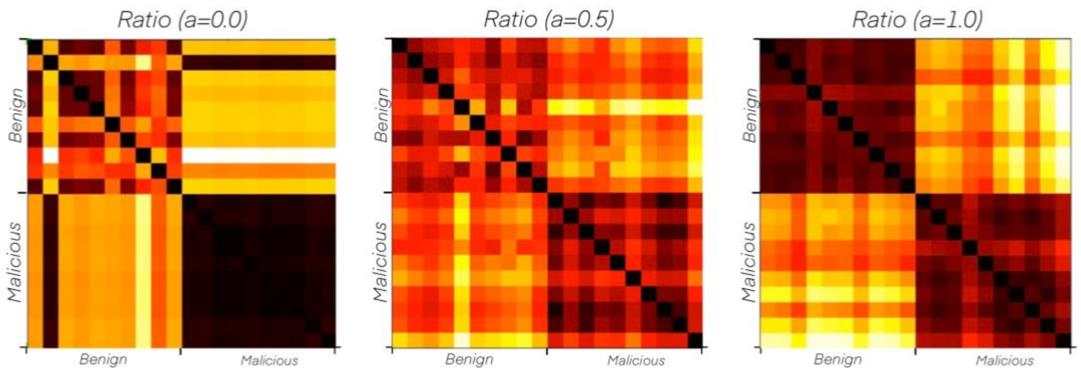

Figure 13. Weighted Ratio Heatmaps
Heatmaps for 20 ground truth samples, 10 of each class. Cells of a darker color indicated lower Kullback-Leibler Divergence values, while cells of a lighter color indicate higher KL Divergence values.

Ratio (*a=0.0*)	Ratio (*a=0.5*)	Ratio (*a=1.0*)
51.5% ± 1.8%	87.4% ± 1.3%	88.3% ± 1.5%

Figure 14. Weighted Ratio Accuracies
Average test accuracies and standard deviations are obtained using ten different samples ground truth sample pairs, tested 4 times each using a different random seed each time to differ the data. Resulting in 40 different test cases.
Bins: 100, KL Divergence Method: KL(X||dist), Model: MLP Scaled

Opcode Set	Z	p
Ratio (*a=0.0*)	-5.49	0.000
Ratio (*a=0.5*)	-4.13	0.000
Ratio (*a=1.0*)	-3.10	0.000

Figure 15. Probability of Ratio Opcode sets yielding higher accuracy than Malicious Opcode set
Z scores and Probabilities of opcode sets being more accurate than the malicious opcode set when tested with a Wilcoxon Signed Rank test, using the differences between the accuracies of samples tested on the same data under the same conditions.
Bins: 100, KL Divergence Method: KL(X||dist), Model: MLP Scaled

None of these methods proved to be significantly more accurate than the malicious opcode set. The set that focused only on the ratio between the opcode sets yields such little information that the model was only slightly more accurate than a guess, proving that opcode misses are detrimental in terms of providing information. This is backed up by the set that focuses only on total occurrences, which has the highest accuracy of all the ratio sets.

Pruning

Even in heatmaps that show a strong pattern, not all ground truth distributions are more similar to other ground truth samples of their own class than those of the opposite class. In examples where aggregated sets are not completely differentiable, meaning that each benign truth sample has a lower KL divergence when compared to other benign samples than when compared to malicious samples and vice versa, then there is little reason to believe that these distributions would include information that represents the true nature of the file, since it is not consistent.

For each distribution in the set of distributions, a distance kernel was created via the Kullback-Leibler divergence between each sample. This distance kernel was then used to determine whether the distributions are linearly separable. If a distribution is not separable amongst classes, then the distribution is removed and will not be used when making comparisons. Pruning the dataset reduces the distributions but increases the cumulative Kullback-Leibler divergence between samples of opposing classes while decreasing the cumulative KL divergence between samples of matching classes. Each opcode set varied slightly, but ~25% of the opcode set was pruned for the Benign, Malicious, Union, Intersection, and Disjoint opcode sets. These distributions consist of a very small superset, so most of the same opcodes are pruned for each opcode set.

Benign Set Pruned: `[lock, sbbb]`
Malicious Set Pruned: `[jmpl, leave]`
Common Pruned: `[addl, cmpb, int3, jg, leal, movw, pushl, rep, testb, xorb]`

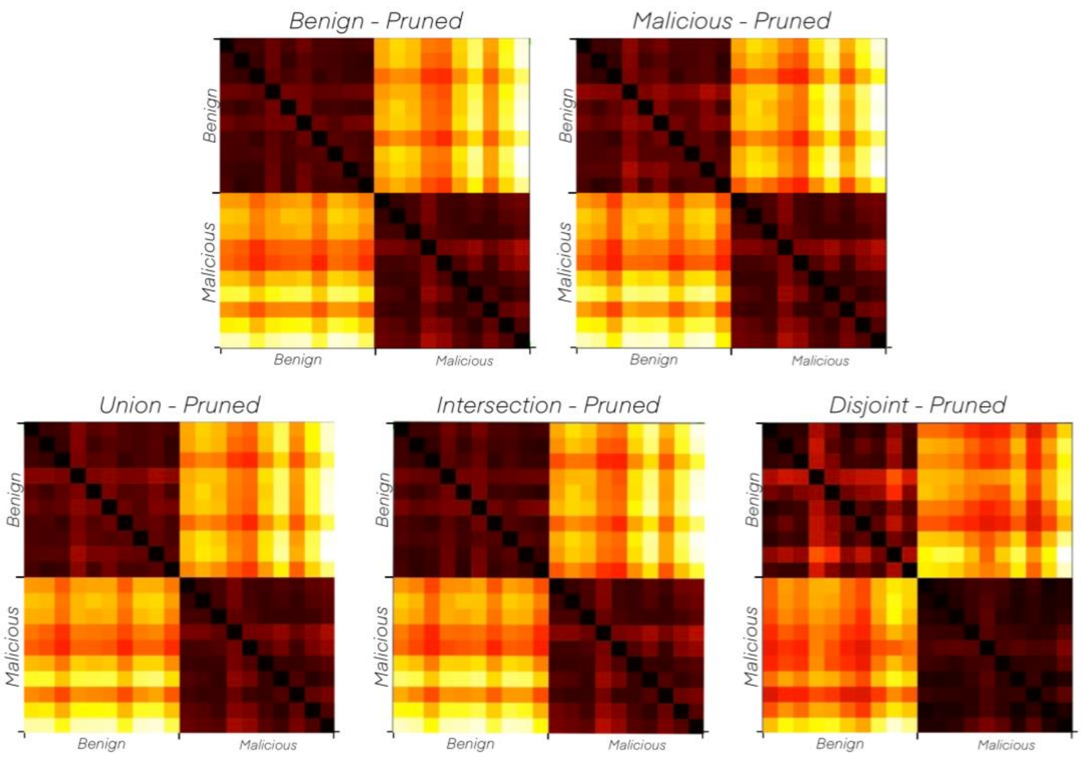

Figure 16. Pruned Heatmaps for Benign, Malicious, Union, Intersection, Disjoint
Heatmaps for 20 ground truth samples, 10 of each class. Cells of a darker color indicated lower Kullback-Leibler Divergence values, while cells of a lighter color indicate higher KL Divergence values.

Benign	Malicious	Union	Intersection	Disjoint
$88.1\% \pm 1.2\%$	$87.7\% \pm 1.4\%$	$88.1\% \pm 1.7\%$	$87.5\% \pm 2.0\%$	$84.3\% \pm 1.7\%$

Figure 17. Pruned Accuracies
Average test accuracies and standard deviations are obtained using ten different samples ground truth sample pairs, tested 4 times each using a different random seed each time to differ the data. Resulting in 40 different test cases. Bins: 100, KL Divergence Method: KL(X||dist), Model: MLP Scaled

Benign		Malicious		Union		Intersection		Disjoint	
Z: 0.11	p: .543	Z: -3.45	p: .000	Z: -1.79	p: .037	Z: -1.30	p: .096	Z: -2.42	p: .008

Figure 18. Probability of Pruned Data being more accurate than Unpruned Data
Z scores and Probabilities of pruned datasets being more accurate than unpruned data sets when tested with a Wilcoxon Signed Rank test, using the differences between the accuracies of samples tested on the same data under the same conditions.
Bins: 100, KL Divergence Method: KL(X||dist), Model: MLP Scaled

Using the Wilcoxon Signed Rank test to determine the if the pairwise different in accuracies between pruned and non-pruned samples was greater, it was determined that pruned data sets did not prove to be substantially (*a=0.05)* better than non-pruned datasets. In most cases, the pruned data set preformed so poorly in comparison to the pruned data that it could be proven, *a=0.05*, that the non-pruned datasets significantly decreased the accuracy.

Removing data that was not consistent across all ground truth samples did not increase accuracy, this was because each model is only exposed to a single ground truth sample. If every sample of a certain class in the training and test sample was alike, but not similar to the ground truth sample, all those samples would still yield similar results when being compared through KL Divergence. This means that the input to the machine learning model would be similar, still helping the accuracy of the model.

There will never be an opcode distribution that is similar across all training and test samples, but reducing the amount of input to the machine learning model did decrease the accuracy of the model. Even with non-consistent opcodes, there was still valuable information to be learned through comparison of the data, and pruning did not conclusively increase the accuracy when training networks.

Op Code Bundle Aggregation

Opcodes as Predictor for Malware [2], a paper focused on static malware analysis paper, used the frequencies of different opcodes as a predictor for benign vs malicious code samples. The paper isolated 14 different opcodes that accounted for roughly 90% of the total opcode occurrences in samples from each class. When comparing this information against the full opcode set used in this experiment, the full opcode set was too infrequent to focus on individual opcodes mentioned in the paper. For example, `mov`, which accounted for 25% of benign (or goodware) instructions and 30% of malicious instructions in the paper, occurred in less than 1% of the samples and accounted for .04% of the total instructions in the combined sets.

This did however lead to a further introspection into the full set of opcodes, which resulted in the realization that most opcodes derive from a one of many common roots. These opcodes that share a common root often accomplish the same goal while differing in implementation or inputs. For example, `add`, `addl`, and `addq` all add two inputs, but they consume different input types. When an assembly programmer or compiler writes a file, their intention is always to add two numbers, but the implementation can change based on the scenario.

Combining any opcodes that are a part of the same family would still be an accurate construction of a distribution, while also increasing the robustness of each distribution since there is a much greater chance high opcode frequency. Having a higher frequency of opcodes decreases the chance of a file generating an empty, uniform distribution, which skews results.

Using the opcodes from *Opcodes as Predictor for Malware* as the root for the opcode families resulted in 12 opcode families containing 90 total opcodes, this will be referred to as the Bilar bundle. Three opcodes, `[jz, retn, jnz]`, were dropped since they did not occur in any of the sample files. While the opcodes originally accounted for __% of the total instructions, the opcodes that are a part of the opcode bundle now account for 63.4% of the total instructions. The pattern of opcode utilization percentage in the Bilar Bundle, Figure 19, somewhat matches the pattern of utilization show for opcodes in *Bilar*, showing that the combining these groups is representative of how these opcodes would be used in stricter opcode set.

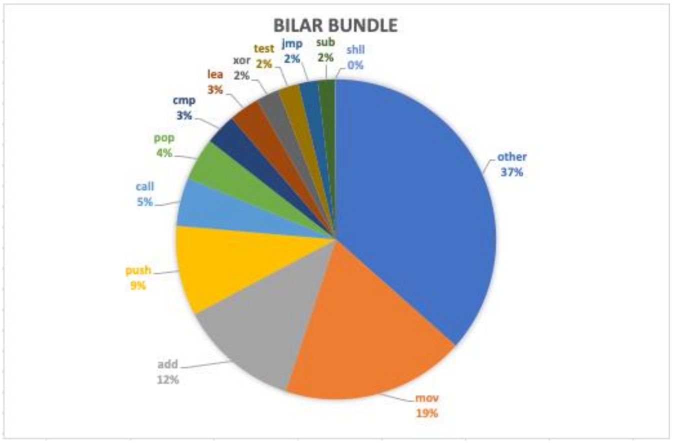

Figure 19. Bilar Bundle Opcode Share Chart
Percentage of the total instruction calls that contained specified root.

While looking through the entire opcode set, it was clear that were other opcode families that could be added that were not a part of the *Opcodes as Predictor for Malware* opcode set. This resulted in full set of root opcode sets, which added in 12 new opcode families, 24 total, and consisted of 714 different opcodes. Accounting for 70.8% of the total instructions. While `retn` did not occur in the opcode set, opcodes with the root '`ret`' were found, so it was returned to the data set. Backing up the findings of *Bilar*, none of the new opcodes accounted for a larger utilization percentage than the opcodes in the Bilar bundle, except for `shll` which has a utilization <1%.

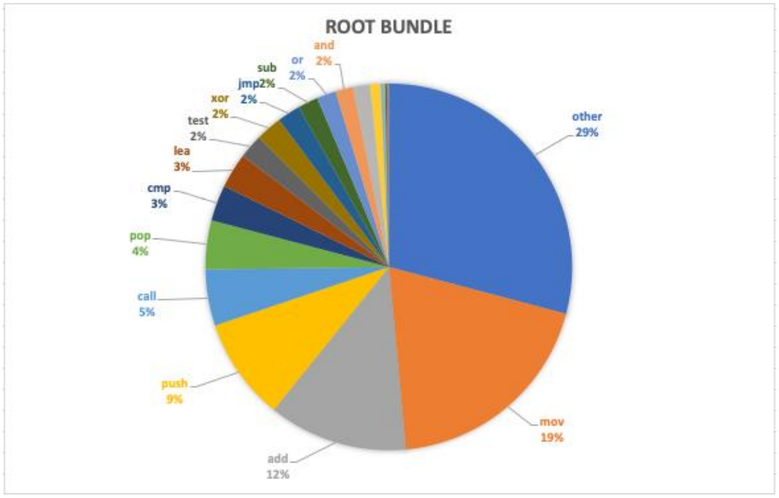

Figure 20. Root Bundle Opcode Share Chart
Percentage of the total instruction calls that contained specified root.
Bundles that were less than 2% of the total number of instructions are omitted from the chart.

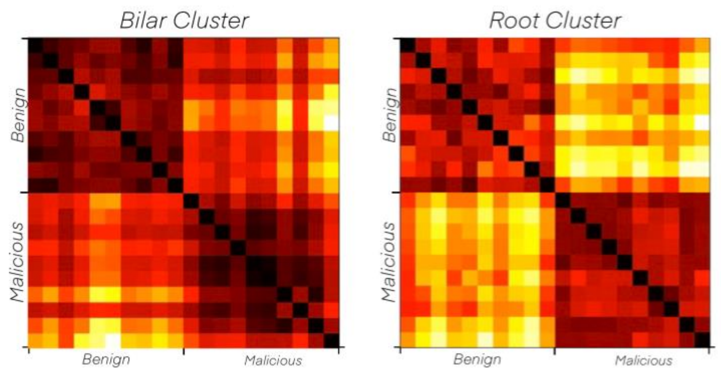

Figure 21. Opcode Bundle Aggregation Heatmaps
Heatmaps for 20 ground truth samples, 10 of each class. Cells of a darker color indicated lower Kullback-Leibler Divergence values, while cells of a lighter color indicate higher KL Divergence values.

Paper Results	Prefix Ops
$82.5\% \pm 2.0\%$	$85.4\% \pm 1.7\%$

Figure 22. Opcode Family Aggregation Accuracies
Average test accuracies and standard deviations are obtained using ten different samples ground truth sample pairs, tested 4 times each using a different random seed each time to differ the data. Resulting in 40 different test cases.
Bins: 100, KL Divergence Method: KL(X||dist), Model: MLP Scaled

Opcode Set	Z	p
Paper Ops	-5.50	0.000
Prefix Ops	-5.15	0.000

Figure 23. Probability of Opcode Bundle sets yielding higher accuracy than Malicious Opcode Set
Z scores and Probabilities of opcode sets being more accurate than the malicious opcode set when tested with a Wilcoxon Signed Rank test, using the differences between the accuracies of samples tested on the same data under the same conditions.
Bins: 100, KL Divergence Method: KL(X||dist), Model: MLP Scaled

Aggregating opcodes did not increase the accuracy of the model; however, for the decrease in sample size it and relatively high accuracy, it did prove to provide new information. This approach was hindered by the lack of common roots, there are an abundance of different operations but a majority of the opcodes that occurred frequently were involved in these bundles. Perhaps these bundles of opcodes could be further subdivided in a meaningful way, while still maintaining the robustness of each distribution.

Combining Opcode Groups

While the Ratio and Opcode Bundle tests failed to provide an opcode set that yielded a higher accuracy than the malicious ground truth set, they did introduce a distinct set of opcodes with minimal or no overlapping opcodes. Having a primarily different set of opcodes provides a different set of KL Divergences, which would not be the case when using any set derived from the Malicious opcode set. Since the opcodes differ heavily, the two sets of distributions can easily be concatenated to create a larger set of comparisons between each file and the ground truth distributions. The Root Bundle and Ratio (*a=1.0*) were chosen since they have much higher accuracies than the other opcodes sets in their test.

Malicious + Ratio (*a=1.0*)	Malicious + Root Bundle	All Three
$88.4\% \pm 1.0\%$	$88.7\% \pm 1.1\%$	$88.6\% \pm 1.2\%$

Figure 24. Combined Opcode Accuracies
Average test accuracies and standard deviations are obtained using ten different samples ground truth sample pairs, tested 4 times each using a different random seed each time to differ the data. Resulting in 40 different test cases.
Bins: 100, KL Divergence Method: KL(X||dist), Model: MLP Scaled

Opcode Set	Z	p
Malicious + Ratio (*a=1.0*)	-2.20	0.014
Malicious + Root Bundle	-0.69	0.247
All Three	-1.01	0.155

Figure 25. Probability of Opcode Family sets yielding higher accuracy than Malicious Opcode set
Z scores and Probabilities of opcode sets being more accurate than the malicious opcode set when tested with a Wilcoxon Signed Rank test, using the differences between the accuracies of samples tested on the same data under the same conditions.
Bins: 100, KL Divergence Method: KL(X||dist), Model: MLP Scaled

Even though the sets featured different distributions and increased the training size, the accuracy was unable to be increased over that of the . This led to the conclusion that there seems to be a limit to how accurate a model trained on Jump distributions can become.

Kullback-Leibler Divergence Method Selection

The Kullback-Leibler Divergence, $D_{KL}(p||q)$, measures the difference of distribution p against distribution q and has been used as the basis of comparison between distributions in this paper. In the context of differentiating a set of distributions against a base standard, it made sense to consider the distribution, or set of distributions, from the given file that we want to classify as p and the ground truth distribution sets as q.

Symmetric KL Divergence | Exploring Flipped Divergence

D_{KL} is asymmetric, and the flipped D_{KL} can be used to provide different insights than the standard method. Flipping the KL Divergence, $D_{KL}(q||p)$, and taking the average of both options, $(D_{KL}(p||q) + D_{KL}(q||p))\ /\ 2$, also provides alternative values that could provide different, and possible more valuable, information.

KL Method	Z	p
KL(dist||X)	0.98	0.837
Symmetric	1.06	0.856

Figure 26. Probability that KL Divergence method yields higher accuracy than KL(X||dist)
Z scores and Probabilities of a more accurate model when using a different KL Divergence method compared to KL(x||dist), using the differences between the accuracies of samples tested on the same data under the same conditions.
Bins: 100, Opcode Set: Malicious, Model: MLP Scaled

Both new options have a high probability of being more accurate than $D_{KL}(x||dist)$, but neither are conclusively better than $D_{KL}(x||dist)$ since they don't meet the level of significance laid out in this test (*a=0.05*).

Isolating Ground Truth Comparison Classes

There are certain distributions sets in which there is only a strong similarity within a specific class of ground truth distributions, seen in Figure 13. In the scenarios in which only one class is similar under an opcode set and an executable set is compared to the class lacking similarity, the data would most likely just be noise. It may be beneficial to not compare the executable to both sets of ground truth distributions and only focus on one ground truth set. This is a similar approach to the pruned datasets.

KL Method	Z	p
KL(X||dist)	-0.73	0.232
KL(dist||X)	-1.55	0.061

Figure 27. Probability of comparing Benign Ground Truth Samples being more accurate than comparing to both Samples

Z scores and Probabilities of accuracies being higher when only comparing to the benign set of ground truth than when comparing to both sets of ground truth samples, using the differences between the accuracies of samples tested on the same data under the same conditions.

Bins: 100, Opcode Set: Malicious Only, Model: MLP Scaled

KL Method	Z	p
KL(X||dist)	-2.49	0.006
KL(dist||X)	-3.12	0.001

Figure 28. Probability of comparing Malicious Ground Truth Samples being more accurate than comparing to both Samples

Z scores and Probabilities of accuracies being higher when only comparing to the malicious set of ground truth than when comparing to both sets of ground truth samples, using the differences between the accuracies of samples tested on the same data under the same conditions.

Bins: 100, Opcode Set: Malicious Only, Model: MLP Scaled

Comparing to only one ground truth samples did not provide any new insights, while it may have removed noise it cut the number of training parameters in half. As seen in in the Pruned Opcode Sets experiment, inconsistent data can still provide meaningful information.

Logarithm Mapping KL Divergence Values

KL Divergence values range from zero, when two distributions are exactly alike, to infinity, when distributions share no similarity. While the KL divergence values derived from these distributions fall in a much smaller range upper bound near ~10. Machine learning algorithms typically perform better when normalized to a standard range [4]. To normalize and more evenly distribute the data, the KL Divergence value was mapped using $\log_{10}$ to put almost all the data in between negative and positive one

On the lower bound, no samples were similar enough to a ground truth distribution to result in a KL Divergence of less than .1, so no samples passed the lower bound of -1. Any KL Divergence value that was greater than 10 will pass the soft boundary of +1, but the number of upper bound outliers is minimal, and these values do not range very far above 10. Any outliers will be very close to +1 so none of these outliers are significant. Most data ends up within a range of negative one to positive one so this was not a very large concern.

KL Method	Z	p
KL(X‖dist)	-0.75	0.226
KL(dist‖X)	0.79	0.786
Symmetric	0.61	0.730

Figure 29. Probability that taking the $\log_{10}$ of the KL Divergence result in higher Accuracy

Z scores and Probabilities of a higher accuracy when taking the $\log_{10}$ of the KL Divergence before being used as input for the model in comparison to no transformation at all, using the differences between the accuracies of samples tested on the same data under the same conditions.

Bins: 100, Opcode Set: Malicious, Model: MLP Scaled

KL Method	Z	p
log(KL(dist‖X))	2.18	0.985
log(Symmetric)	0.78	0.784

Figure 30. Probability of KL transformation against K(x‖dist)

Z scores and Probabilities of a higher accuracy when taking the $\log_{10}$ of the KL Divergence before being used as input for the model in comparison to K(x‖dist), using the differences between the accuracies of samples tested on the same data under the same conditions.

Bins: 100, Opcode Set: Malicious, Model: MLP Scaled

Taking the $\log_{10}$ of KL Divergence values did not provide a significant enough increase in accuracy when compared to the same KL Divergence with no transformation. However, when comparing the current leading KL Divergence method, *KL(x||dist)*, to *$\log_{10}$(KL(dist||x))*, the latter proved to be more accurate with a high enough level of significance. *$\log_{10}$(KL(dist||x))* will now be the primary KL Divergence method.

This does not change any findings, made in the previous sections.

Different Distribution Methods

Most work was focused on jump distributions; however, other distributions methods were also tested. *Opcodes as Predictor for Malware* [2] used the percentage shares of specific opcodes to classify code samples, a similar approach can be done while still following the pattern of creating distributions for each executable. New distributions can be created off the percentage of the file that the opcode consumes at regular intervals, rather than the entire. If there are any major changes in how the degree of occurrence of a certain opcode that could be a reason to suspect on class or another.

Opcode Utilization Distributions

Each distribution compares the proportion of the code that the opcode consumes within that bin. Distributions are no longer created along each op code but rather by bin, which represents a constant percentage of the file. Inverting the direction in which distributions are analyzed. Since comparisons are made to both ground truth distributions for every bin, the number of trainable inputs would increase based off the number of bins rather than the number of total opcodes. This could lead to potential accuracy increase without the need to increase opcode set size, which increases the risks of infrequent opcodes.

$$bin\ assignment = \phi(floor(occurence\ /\ file\ length)\ )$$

Distribution Algorithm

```
1. For opcode op
    a. Define $distribution_{op}$ as vector of zeros the size of the number of bins.
    b. Construct a list of the indices at which op occurs, $freqs_{op}$.
    c. For each op occurrence $op_k$ in $freqs_{op}$,
        i. For each occurrence $op_k$, calculate its bin assignment via $\phi$ and increment
           the count of that bin in $distribution_{op}$.
2. For each bin b
    a. For opcode op
        i. Divide the value of each op bin b by the sum of all ops in the bin,
           $\sum distribution_{op}[b]$ over all ops, to create a distribution.
```

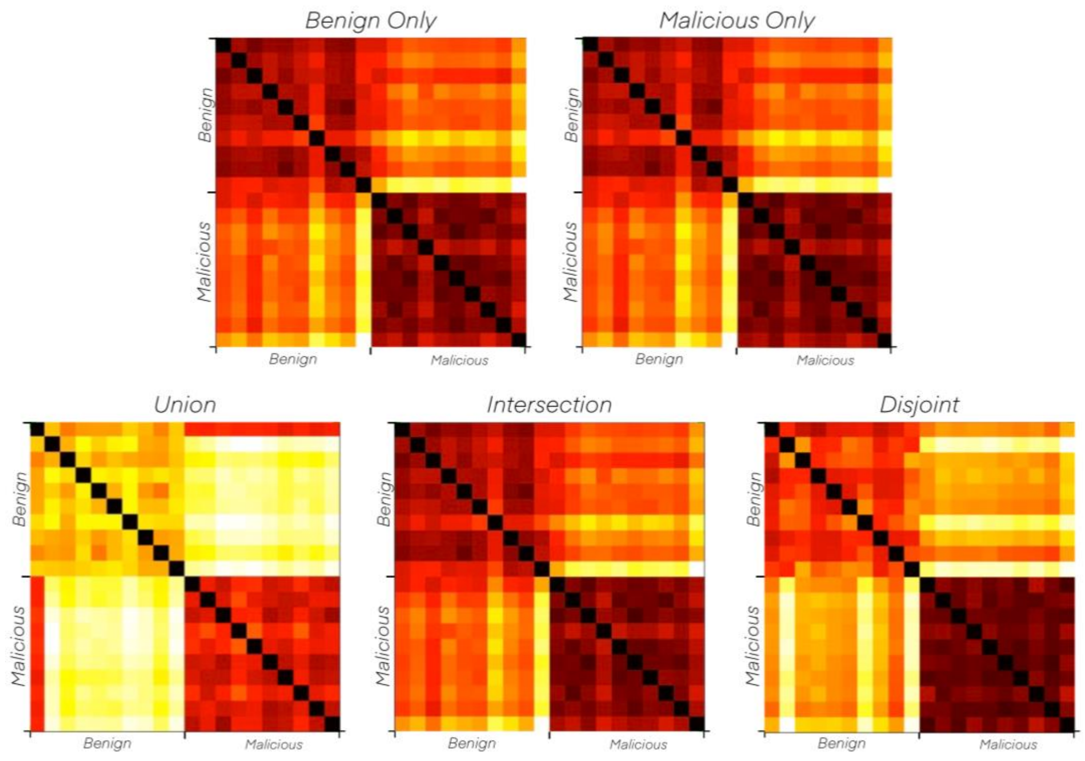

Figure 31. Utilization Heatmaps for Benign, Malicious, Union, Intersection, Disjoint
Heatmaps for 20 ground truth samples, 10 of each class. Cells of a darker color indicated lower Kullback-Leibler Divergence values, while cells of a lighter color indicate higher KL Divergence values.

The heatmaps for the benign and malicious opcode sets are both nearly identical to the intersection opcode set heatmap. While the pattern of the heatmaps matches the union heatmap, the values of the KL Divergence of the heatmaps differ very heavily even though all the opcodes sets differ very minimally. This leads to the conclusion that the disjoint opcode set (*Union – Intersection)* introduces a higher level of inconsistency, that was not included in benign or malicious opcode sets. Infrequent opcode are an even larger hindrance using the share method since the distributions are based off consistent proportions of opcodes, this is backed up by the accuracies in Figure 32.

Benign	Malicious	Union	Intersection	Disjoint
$83.0\% \pm 1.8\%$	$83.0\% \pm 2.0\%$	$78.4\% \pm 3.0\%$	$83.0\% \pm 2.1\%$	$77.5\% \pm 2.2\%$

Figure 32. Opcode Set Accuracies with Utilization Distribution
Average test accuracies and standard deviations are obtained using ten different samples ground truth sample pairs, tested 4 times each using a different random seed each time to differ the data. Resulting in 40 different test cases.
Bins: 100, KL Divergence Method: log(KL(dist||x), Model: MLP Scaled

Opcode Set	Z	p
Benign	-5.49	0.000
Malicious	-5.49	0.000
Union	-5.49	0.000
Intersection	-5.49	0.000
Disjoint	-5.49	0.000

Figure 33. Probability of Utilization Distribution being more accurate than Jump Distribution
Z scores and Probabilities of an opcode set being more accurate when using a share distribution in comparison to a jump distribution using a Wilcoxon Signed Rank test, using the differences between the accuracies of samples tested on the same data under the same conditions.
Bins: 100, KL Divergence Method: log(KL(dist||x), Opcode Set: Malicious, Model: MLP Scaled

The Utilization distribution did not fare as well as the jump distribution. Every single test from the Utilization distribution was less accurate that the Jump distribution test over the same data, this is why the Z score is the same for all opcode sets. Outside of the disjoint set, which still tested the least accurate, the utilization heatmaps failed show distinct regions of difference like they did with jump distribution. Heatmaps are not a perfect representation, but if they do not show much difference between classes it is likely that Utilization does not capture as much information as the jump distributions do.

<u>Opcode Cumulative Utilization Distributions</u>

The approach was taken in a cumulative manor, instead of each bin representing the proportion of opcodes that the opcode takes up in the bin, it is a representation of how much of the code that opcode has consumed so far.

Cumulative Distribution Algorithm

```
1. For op code op
    a. Construct a list of the indices at which op occurs, $freqs_{op}$.
    b. For each op occurrence $op_k$ in $freqs_{op}$,
        i. For each occurrence $op_k$, calculate its bin assignment via $\phi$ and increment
           the count of that bin in $distribution_{op}$ and of each subsequent bin in
           $distribution_{op}$.
2. For each bin b
    a. Divide the value of each op code in bin b by the sum of all ops in the bin, $\sum op_b$,
       to create a distribution.
```

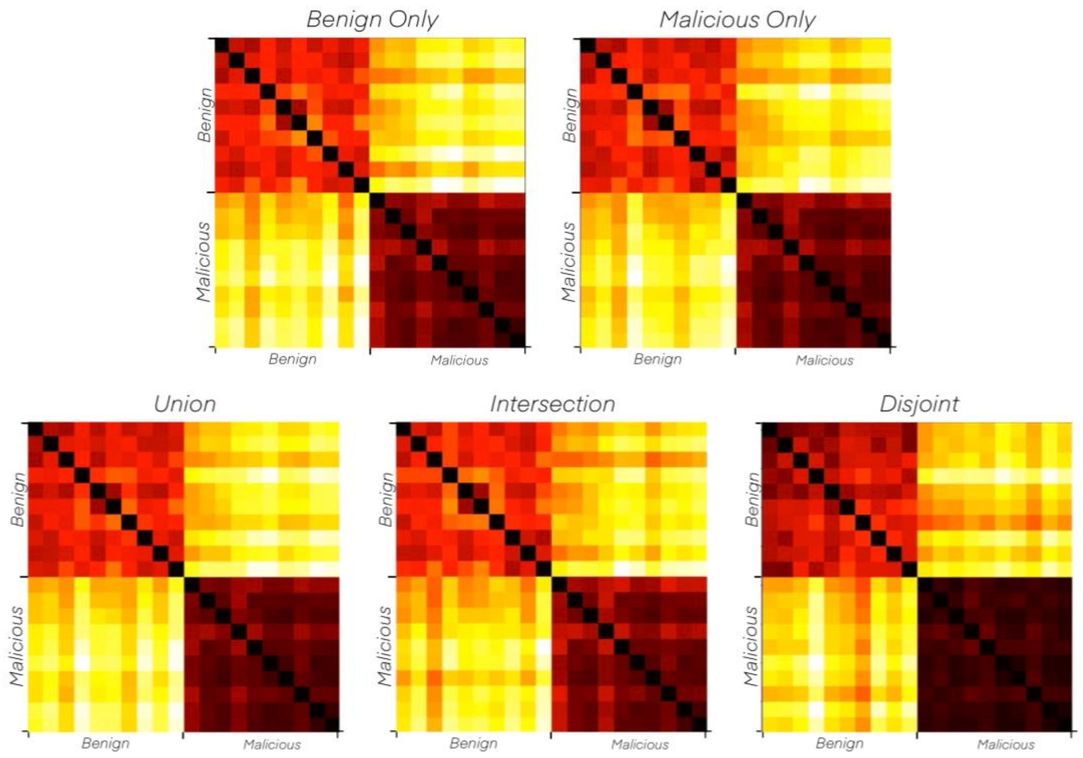

Figure 34. Cumulative Utilization Heatmaps for Benign, Malicious, Union, Intersection, Disjoint
Heatmaps for 20 ground truth samples, 10 of each class. Cells of a darker color indicated lower Kullback-Leibler Divergence values, while cells of a lighter color indicate higher KL Divergence values.

The heatmaps for the benign and malicious opcode sets are both nearly identical to the intersection opcode set heatmap, but they do not differ very strongly from the union opcode set. When compared to the inconsistencies of the share method, there is a low degree of inconsistency when comparing opcodes that are infrequent to one opcode set. This is due to the accumulation of opcode percentages, minimizing the chance of misses in bins that track information from the end of the file.

Benign	Malicious	Union	Intersection	Disjoint
$81.8\% \pm 2.0\%$	$83.1\% \pm 1.6\%$	$83.1\% \pm 1.8\%$	$82.1\% \pm 2.1\%$	$79.9\% \pm 2.0\%$

Figure 35. Opcode Set Accuracies with Cumulative Utilization Distribution
Average test accuracies and standard deviations are obtained using ten different samples ground truth sample pairs, tested 4 times each using a different random seed each time to differ the data. Resulting in 40 different test cases.

Opcode Set	Z	p
Benign	-5.49	0.000
Malicious	-5.49	0.000
Union	-5.49	0.000
Intersection	-5.49	0.000
Disjoint	-5.49	0.000

Figure 36. Probability of Cumulative Utilization Distribution being more accurate than Jump Distribution

Z scores and Probabilities of the Cumulative Utilization distribution method being more accurate than Jump distribution method when tested with a Wilcoxon Signed Rank test, using the differences between the accuracies of samples tested on the same data under the same conditions.

Bins: 100, KL Divergence Method: log(KL(dist||x), Opcode Set: Malicious, Model: MLP Scaled

Opcode Set	Z	p
Benign	-3.10	0.001
Malicious	0.29	0.614
Union	5.18	1.000
Intersection	-2.57	0.005
Disjoint	4.51	1.000

Figure 37. Probability of Cumulative Utilization Distribution being more accurate than Utilization Distribution

Z scores and Probabilities of the Cumulative Utilization distribution method being more accurate than Utilization distribution method when tested with a Wilcoxon Signed Rank test, using the differences between the accuracies of samples tested on the same data under the same conditions.

Bins: 100, KL Divergence Method: log(KL(dist||x), Opcode Set: Malicious, Model: MLP Scaled

The cumulative share distribution did not have any drastic drops in accuracy like the standard share method did since there was more consistently data in each bin. When comparing the accuracies of the two methods, it is clear the standard share distribution loses information from opcodes within the disjoint set, while the cumulative share method does not suffer from the same fate. In Figure 37, cumulative share is verifiably a more accurate when the disjoint data is tested, while the share was more accurate when the data is not included.

Preventing the drops in accuracy through the use of cumulative distributions methods did not increase the overall accuracy of the models when compared the standard share, rather it raised the lower bound of the accuracies. This lower bound was from suboptimal opcode sets anyway, so this is does not affect the which shared distribution was better. Most importantly, using either version of the share distribution method fails to yield higher accuracies than the jump distribution, even with the benefit of larger input sizes.

Combined Distribution Methods

Both share attempts failed to result in increased accuracy; however, they did consume the same data in different ways. Similar to combining opcode sets, the jump and share distributions comparison data was combined. This resulted in two sets of inputs for the same data, meaning that they could be combined and be used in tandem to train a model. Tested under the same conditions, combing the two distribution sets did not increase the test set accuracy. As previously theorized, the Jump distribution seems to be at its highest possible accuracy, and it is now shown that adding in data from a different distribution fails to increase it any higher as well.

Jump + Utilization
88.4% $\pm$ 1.0%

Figure 38. Average Combined Method Accuracy
Average test accuracies and standard deviations are obtained using ten different samples ground truth sample pairs, tested 4 times each using a different random seed each time to differ the data. Resulting in 40 different test cases.
Bins: 100, KL Divergence Method: KL(X||dist), Model: MLP Scaled

Opcode Set	Z	p
Jump + Utilization	-4.45	0.000

Figure 39. Probability of Jump + Utilization Distribution being more accurate than Jump Distribution
Z scores and Probabilities of combined methods being more accurate in comparison to a jump distribution using a Wilcoxon Signed Rank test, using the differences between the accuracies of samples tested on the same data under the same conditions.
Bins: 100, KL Divergence Method: log(KL(dist||x), Opcode Set: Malicious, Model: MLP Scaled

Test Results and Conclusion

The final model was tested against multiple datasets which were not used to train any models. A set of benign files was obtained from analyzing all files with a '.exe' extension on a clean install of a Windows 10 virtual machine. Two malicious file sets were downloaded from VirusShare [5, 6], an online malware repository for security researchers. Each file set had over 13.5GB of malware samples, but only 7500 of those were used from each set due to processing time constraints. A final benign set was obtained from my personal computer, running macOS Big Sir, by analyzing all executable files that had '.bundle', '.dylib', or '.so' extensions. All the models from the testing portion were saved, leaving 40 total models for the data to be tested against. 250 random selected samples from each test set were tested on every model.

Final Model Parameters
Distribution Method – *Jump*
Opcode Set – *Malicious Only*
KL Divergence – *log_{10}(KL (dist || x))*
Model – *Multi-Layer Perceptron*; hidden layers – *[100, 200, 50]*
Bins – *100*

Windows Samples	Benign Samples Only	79.3% $\pm$ 5.8%
VirusShare_00000	Malicious Samples Only	98.4% $\pm$ 0.9%
VirusShare_00005	Malicious Samples Only	97.5% $\pm$ 1.3%
Mac OS Samples	Benign Samples Only	35.8% $\pm$ 4.4%

Figure 40. Average Accuracy of models on purely Test Datasets
Test accuracies and standard deviations are obtained using ten different samples ground truth sample pairs, tested 4 times each using a different random seed each time to differ the data. Resulting in 40 different test cases.
Bins: 100, KL Divergence Method: log(KL(dist||x), Opcode Set: Malicious, Model: MLP Scaled

The malicious data was exceptionally accurate, while the benign samples faltered below the test accuracy. The weighted average of the accuracies is very similar to that of the tests conducted on these models during training, ignoring the MacOS files. There was no reason to believe that the MacOS files would be accurately classified when using a model trained on Window's executables, but the fact that it had an accuracy less than a random decision maker would yield suggests that there is data to be learned from the MacOS samples.

The increase in accuracy over the Malicious data sets could be due to the difference in samples under the training set and test sets. VirusShare may have stricter guidelines for what is considered malware. The training set classifies malicious files based of whether multiple antivirus software identifies it as malicious [3], while VirusShare is cultivated by a digital forensics team [5, 6]. This more hands on approach could mean that the malicious samples in the test set are more malicious and differ more from benign files than the malicious samples in training did. A similar approach could be used to attribute the low accuracy of the Windows samples, but it is also important to note that the training set samples were primarily from Windows 7 [3], so the files could be interpreted differently.

Overall, the test set accuracies still back up that using translating an executable to a set of jump distributions does yield information that can be used to classify the nature of a file as benign or malicious.

Sources

1. Anderson, Hyrum S, and Phil Roth. "EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models."
2. Bilar, D. (2007) 'Opcodes as predictor for malware', *Int. J. Electronic Security and Digital Forensics*, Vol. 1, No. 2, pp.156–168.
3. Lester, Micheal. "PE Malware Machine Learning Dataset." *Practical Security Analysis*, https://www.practicalsecurityanalytics.com.
4. Singh, Dalwinder, and Birmohan Singh. "Investigating the Impact of Data Normalization on Classification Performance." *Applied Soft Computing Journal*, 23 Apr. 2019.
5. Corvus Forensics. "VirusShare_00001." *Virus Share*, https://www.virsusshare.com.
6. Corvus Forensics. "VirusShare_00005." *Virus Share*, https://www.virsusshare.com.

Appendix

A. Op Code Sets

   a. Benign Only - `['adcb', 'adcl', 'addb', 'addl', 'andb', 'andl', 'cltd', 'cmpb', 'cmpl', 'cmpw', 'decl', 'imull', 'incl', 'int3', 'ja', 'jae', 'jb', 'jbe', 'je', 'jg', 'jge', 'jl', 'jle', 'jmp', 'jne', 'jns', 'js', 'leal', 'lock', 'movb', 'movl', 'movw', 'movzwl', 'negl', 'nop', 'orb', 'orl', 'pushl', 'rep', 'sbbb', 'sbbl', 'sete', 'shrl', 'subb', 'subl', 'testb', 'testl', 'xchgl', 'xorb', 'xorl']`

   b. Malicious Only - `['adcl', 'addb', 'addl', 'andb', 'andl', 'calll', 'cltd', 'cmpb', 'cmpl', 'decl', 'imull', 'incl', 'int3', 'ja', 'jae', 'jb', 'jbe', 'je', 'jg', 'jge', 'jl', 'jle', 'jmp', 'jmpl', 'jne', 'js', 'leal', 'leave', 'movb', 'movl', 'movw', 'negl', 'nop', 'orb', 'orl', 'popl', 'pushl', 'rep', 'retl', 'sarl', 'sbbl', 'shll', 'shrl', 'subb', 'subl', 'testb', 'testl', 'xchgl', 'xorb', 'xorl']`

c. Union - $Benign\ Only\ \cup Malcious\ Only$
d. Intersection - $Benign\ Only\ \cap Malcious\ Only$
e. Disjoint - $Union - Intersection$
f. Ratio (Ratio, *a=0.5)* - ['addb', 'addl', 'andb', 'andl', 'bsrw ', 'bswapw', 'btcw', 'btrw', 'btsw', 'cmovnsq', 'cmpb', 'cmpl', 'cvtsi2sdl', 'data16', 'decl', 'ds', 'es', 'fs', 'gs', 'imull', 'incl', 'iretq', 'ja', 'jae', 'jb', 'jbe', 'je', 'jl', 'jle', 'jmp', 'jne', 'js', 'leal', 'lretq', 'movb', 'movl', 'movw', 'nop', 'orl', 'rex64', 'sbbl', 'scasq', 'sgdtq', 'shldw', 'ss', 'subl', 'testb', 'testl', 'xorb', 'xorl']
g. Ratio (Ratio, *a=0.0)* - ['adcs', 'add', 'adds', 'adr', 'and', 'ands', 'asrs', 'b', 'beq', 'bgt', 'bhs', 'bl', 'ble', 'bls', 'blx', 'bne', 'bx', 'cbnz', 'cbz', 'cmp', 'dmb', 'ldm', 'ldr', 'ldrb', 'ldrh', 'ldrsb', 'ldrsh', 'lgdtq', 'lidtq', 'lsls', 'lsrs', 'mov', 'movs', 'movzww', 'mvn', 'mvns', 'orrs', 'pop', 'shldw', 'shrdw', 'sidtq', 'stm', 'str', 'strb', 'strd', 'strh', 'sub', 'subs', 'tst', 'udf'],
h. Ratio (Ratio, *a=1.0)* - ['adcl', 'addb', 'addl', 'andb', 'andl', 'calll', 'cltd', 'cmpb', 'cmpl', 'cmpw', 'decl', 'imull', 'incl', 'int3', 'ja', 'jae', 'jb', 'jbe', 'je', 'jg', 'jge', 'jl', 'jle', 'jmp', 'jne', 'jns', 'js', 'leal', 'movb', 'movl', 'movw', 'negl', 'nop', 'orb', 'orl', 'popl', 'pushl', 'rep', 'retl', 'sbbb', 'sbbl', 'shll', 'shrl', 'subb', 'subl', 'testb', 'testl', 'xchgl', 'xorb', 'xorl']
i. Paper Family - ['add', 'addb', 'addl', 'addpd', 'addps', 'addq', 'adds', 'addsd', 'addss', 'addw', 'call', 'calll', 'callq', 'callw', 'cmp', 'cmpb', 'cmpl', 'cmppd', 'cmpps', 'cmpq', 'cmpsb', 'cmpsl', 'cmpsq', 'cmpss', 'cmpsw', 'cmpw', 'jmp', 'jmpl', 'jmpq', 'jmpw', 'lea', 'leal', 'leaq', 'leave', 'leaw', 'mov', 'movb', 'movd', 'movi', 'movk', 'movl', 'movq', 'movsb', 'movsd', 'movsl', 'movsq', 'movss', 'movsw', 'movw', 'pop', 'popal', 'popaw', 'popfl', 'popfq', 'popfw', 'popl', 'popq', 'popw', 'push', 'pushal', 'pushaw', 'pushfl', 'pushfq', 'pushfw', 'pushl', 'pushq', 'pushw', 'shll', 'sub', 'subb', 'subl', 'subpd', 'subps', 'subq', 'subs', 'subsd', 'subss', 'subw', 'test', 'testb', 'testl', 'testq', 'testw', 'xor', 'xorb', 'xorl', 'xorpd', 'xorps', 'xorq', 'xorw']
j. Prefix Family - ['add', 'addb', 'addl', 'addpd', 'addps', 'addq', 'adds', 'addsd', 'addss', 'addsubps', 'addw', 'and', 'andb', 'andl', 'andnl', 'andnpd', 'andnps', 'andnq', 'andpd', 'andps', 'andq', 'ands', 'andw', 'call', 'calll', 'callq', 'callw', 'ccmp', 'cmovael', 'cmovaeq', 'cmovaew', 'cmoval', 'cmovaq', 'cmovaw', 'cmovbel', 'cmovbeq', 'cmovbew', 'cmovbl', 'cmovbq', 'cmovbw', 'cmovel', 'cmoveq', 'cmovew', 'cmovgel', 'cmovgeq', 'cmovgew', 'cmovgl', 'cmovgq', 'cmovgw', 'cmovlel', 'cmovleq', 'cmovlew', 'cmovll', 'cmovlq', 'cmovlw', 'cmovnel', 'cmovneq', 'cmovnew', 'cmovnol', 'cmovnoq', 'cmovnpl', 'cmovnpq', 'cmovnpw', 'cmovnsl', 'cmovnsq', 'cmovnsw', 'cmovol', 'cmovoq', 'cmovpl', 'cmovpq', 'cmovsl', 'cmovsq', 'cmovsw', 'cmp', 'cmpb', 'cmpeqpd', 'cmpeqps', 'cmpeqsd', 'cmpeqss', 'cmpl', 'cmplepd', 'cmpleps', 'cmplesd', 'cmpless', 'cmpltpd', 'cmpltps', 'cmpltsd', 'cmpltss', 'cmpneqpd', 'cmpneqps', 'cmpneqsd', 'cmpneqss', 'cmpnlepd', 'cmpnleps', 'cmpnlesd', 'cmpnless', 'cmpnltpd', 'cmpnltps', 'cmpnltsd', 'cmpordpd', 'cmpordps', 'cmpordsd', 'cmpordss', 'cmppd', 'cmpps', 'cmpq', 'cmpsb', 'cmpsl', 'cmpsq', 'cmpss', 'cmpsw', 'cmpunordpd', 'cmpunordps', 'cmpunordsd', 'cmpunordss', 'cmpw', 'cmpxchg16b', 'cmpxchg8b', 'cmpxchgb', 'cmpxchgl', 'cmpxchgq', 'cmpxchgw', 'cset', 'csetm', 'div', 'divb', 'divl', 'divpd', 'divps', 'divq', 'divsd', 'divss', 'divw', 'eor', 'fadd', 'faddl', 'faddp', 'fadds', 'fcmovb', 'fcmovbe',

```
'fcmove', 'fcmovnb', 'fcmovnbe', 'fcmovne', 'fcmovnu', 'fcmovu', 'fdiv',
'fdivl', 'fdivp', 'fdivr', 'fdivrl', 'fdivrp', 'fdivrs', 'fdivs', 'fiaddl',
'fiadds', 'fidivl', 'fidivrl', 'fidivrs', 'fidivs', 'fimull', 'fimuls',
'fisubl', 'fisubrl', 'fisubrs', 'fisubs', 'fmadd', 'fmov', 'fmul', 'fmull',
'fmulp', 'fmuls', 'fnmsub', 'fnsave', 'frstor', 'fsub', 'fsubl', 'fsubp',
'fsubr', 'fsubrl', 'fsubrp', 'fsubrs', 'fsubs', 'fxrstor', 'fxsave', 'hsubps',
'idivb', 'idivl', 'idivq', 'idivw', 'imulb', 'imull', 'imulq', 'imulw', 'iretl',
'iretq', 'iretw', 'jmp', 'jmpl', 'jmpq', 'jmpw', 'kandb', 'kmovb', 'kmovd',
'kmovw', 'kxnorb', 'kxnorw', 'lcalll', 'lcallq', 'lcallw', 'lea', 'leal',
'leaq', 'leave', 'leaw', 'ljmpl', 'ljmpq', 'ljmpw', 'loop', 'loope', 'loopne',
'lretl', 'lretq', 'lretw', 'madd', 'maskmovdqu', 'maskmovq', 'max', 'maxpd',
'maxps', 'maxsd', 'maxss', 'min', 'minpd', 'minps', 'minsd', 'minss', 'monitor',
'monitorx', 'mov', 'movabsb', 'movabsl', 'movabsq', 'movabsw', 'movapd',
'movaps', 'movb', 'movbel', 'movbeq', 'movd', 'movddup', 'movdiri', 'movdq2q',
'movdqa', 'movdqu', 'movhlps', 'movhpd', 'movhps', 'movi', 'movk', 'movl',
'movlhps', 'movlpd', 'movlps', 'movmskpd', 'movmskps', 'movntdq', 'movntdqa',
'movntil', 'movntiq', 'movntps', 'movntq', 'movq', 'movq2dq', 'movsb', 'movsbl',
'movsbq', 'movsbw', 'movsd', 'movshdup', 'movsl', 'movsldup', 'movslq', 'movsq',
'movss', 'movsw', 'movswl', 'movswq', 'movupd', 'movups', 'movw', 'movzbl',
'movzbq', 'movzbw', 'movzwl', 'movzwq', 'msub', 'mul', 'mulb', 'mull', 'mulpd',
'mulps', 'mulq', 'mulsd', 'mulss', 'mulw', 'mulxq', 'or', 'orb', 'orl', 'orn',
'orpd', 'orps', 'orq', 'orr', 'orw', 'paddb', 'paddd', 'paddq', 'paddsb',
'paddsw', 'paddusb', 'paddusw', 'paddw', 'pand', 'pandn', 'pclmulqdq',
'pcmpeqb', 'pcmpeqd', 'pcmpeqq', 'pcmpeqw', 'pcmpgtb', 'pcmpgtd', 'pcmpgtw',
'pcmpistri', 'pfadd', 'pfcmpge', 'pfcmpgt', 'pfmax', 'pfmin', 'pfmul', 'pfsub',
'pfsubr', 'phaddd', 'phaddsw', 'phaddw', 'phminposuw', 'phsubd', 'phsubw',
'pmaddubsw', 'pmaddwd', 'pmaxsb', 'pmaxsd', 'pmaxsw', 'pmaxub', 'pmaxud',
'pmaxuw', 'pminsb', 'pminsd', 'pminsw', 'pminub', 'pminud', 'pminuw',
'pmovmskb', 'pmovsxbd', 'pmovsxbq', 'pmovsxbw', 'pmovsxdq', 'pmovsxwd',
'pmovsxwq', 'pmovzxbd', 'pmovzxbq', 'pmovzxbw', 'pmovzxdq', 'pmovzxwd',
'pmovzxwq', 'pmuldq', 'pmulhrsw', 'pmulhrw', 'pmulhuw', 'pmulhw', 'pmulld',
'pmullw', 'pmuludq', 'pop', 'popal', 'popaw', 'popcntl', 'popfl', 'popfq',
'popfw', 'popl', 'popq', 'popw', 'por', 'psubb', 'psubd', 'psubq', 'psubsb',
'psubsw', 'psubusb', 'psubusw', 'psubw', 'ptest', 'push', 'pushal', 'pushaw',
'pushfl', 'pushfq', 'pushfw', 'pushl', 'pushq', 'pushw', 'pxor', 'rdrandl',
'rdrandq', 'ret', 'retl', 'retq', 'retw', 'ror', 'rorb', 'rorl', 'rorq', 'rorw',
'rorxl', 'rorxq', 'save', 'sdiv', 'set', 'seta', 'setae', 'setb', 'setbe',
'sete', 'setg', 'setge', 'setl', 'setle', 'setne', 'setno', 'setnp', 'setns',
'seto', 'setp', 'sets', 'sha', 'sha1msg1', 'sha1msg2', 'sha1nexte', 'sha1rnds4',
'sha256msg1', 'sha256msg2', 'sha256rnds2', 'shll', 'smull', 'sub', 'subb',
'subl', 'subpd', 'subps', 'subq', 'subs', 'subsd', 'subss', 'subw', 'sys',
'syscall', 'sysenter', 'sysexitl', 'sysexitq', 'sysretl', 'sysretq', 'test',
'testb', 'testl', 'testq', 'testw', 'uaddlv', 'udiv', 'umaxv', 'uminv', 'umull',
'vaddpd', 'vaddps', 'vaddsd', 'vaddss', 'vaddsubpd', 'vaddsubps', 'vandnpd',
'vandnps', 'vandpd', 'vandps', 'vcmpeq_uqsd', 'vcmpeq_uqss', 'vcmpeqpd',
'vcmpeqps', 'vcmpeqsd', 'vcmpeqss', 'vcmpfalse_ossd', 'vcmpge_oqsd',
'vcmpgt_oqsd', 'vcmpgtps', 'vcmpgtss', 'vcmplepd', 'vcmpleps', 'vcmplesd',
'vcmplt_oqsd', 'vcmpneq_ussd', 'vcmpnge_uqpd', 'vcmpnge_uqss', 'vcmpngt_uqps',
'vcmpngtps', 'vcmpnltpd', 'vcmpnltsd', 'vcmpordps', 'vcmppd', 'vcmpps',
'vcmpsd', 'vcmpss', 'vdivpd', 'vdivps', 'vdivsd', 'vdivss', 'vfmadd132ps',
'vfmadd132sd', 'vfmadd132ss', 'vfmadd213ps', 'vfmadd213sd', 'vfmadd213ss',
'vfmadd231sd', 'vfmadd231ss', 'vfmaddsub132ps', 'vfmaddsubpd', 'vfmaddsubps',
'vfmsub132sd', 'vfmsub213sd', 'vfmsub213ss', 'vfmsub231ps', 'vfmsubadd132ps',
'vfmsubadd231pd', 'vfmsubadd231ps', 'vfmsubaddpd', 'vfmsubpd', 'vfmsubps',
'vfmsubss', 'vfnmadd132ps', 'vfnmadd132sd', 'vfnmadd213ps', 'vfnmadd213sd',
'vfnmadd231sd', 'vfnmaddps', 'vfnmaddss', 'vfnmsub132ps', 'vfnmsub231sd',
'vfnmsubpd', 'vfnmsubps', 'vfnmsubsd', 'vhaddpd', 'vhaddps', 'vhsubpd',
'vhsubps', 'vmaskmovpd', 'vmaskmovps', 'vmaxpd', 'vmaxps', 'vmaxsd', 'vmaxss',
'vmcall', 'vminpd', 'vminps', 'vminsd', 'vminss', 'vmmcall', 'vmovapd',
'vmovaps', 'vmovd', 'vmovddup', 'vmovdqa', 'vmovdqa32', 'vmovdqa64', 'vmovdqu',
'vmovdqu32', 'vmovdqu64', 'vmovdqu8', 'vmovhpd', 'vmovhps', 'vmovlpd',
'vmovlps', 'vmovmskpd', 'vmovntdq', 'vmovntps', 'vmovq', 'vmovsd', 'vmovshdup',
'vmovsldup', 'vmovss', 'vmovupd', 'vmovups', 'vmsave', 'vmulpd', 'vmulps',
'vmulsd', 'vmulss', 'vorpd', 'vorps', 'vpaddb', 'vpaddd', 'vpaddq', 'vpaddsb',
'vpaddsw', 'vpaddusb', 'vpaddusw', 'vpaddw', 'vpand', 'vpandd', 'vpandn',
'vpandq', 'vpclmulqdq', 'vpcmpeqb', 'vpcmpeqd', 'vpcmpeqq', 'vpcmpeqw',
'vpcmpgtb', 'vpcmpgtd', 'vpcmpgtq', 'vpcmpgtw', 'vpcmpltw', 'vpcmpnleub',
'vphaddd', 'vphaddsw', 'vphaddw', 'vphminposuw', 'vphsubw', 'vpmadd52huq',
'vpmadd52luq', 'vpmaddubsw', 'vpmaddwd', 'vpmaskmovd', 'vpmaskmovq', 'vpmaxsb',
'vpmaxsd', 'vpmaxsw', 'vpmaxub', 'vpmaxud', 'vpmaxuw', 'vpminsb', 'vpminsd',
'vpminsq', 'vpminsw', 'vpminub', 'vpminuq', 'vpminuw', 'vpmovdw', 'vpmovmskb',
'vpmovsxbw', 'vpmovsxdq', 'vpmovsxwd', 'vpmovzxbd', 'vpmovzxbq', 'vpmovzxbw',
'vpmovzxwd', 'vpmovzxwq', 'vpmuldq', 'vpmulhrsw', 'vpmulhuw', 'vpmulhw',
'vpmulld', 'vpmullw', 'vpmuludq', 'vpor', 'vporq', 'vpshaw', 'vpsubb', 'vpsubd',
'vpsubq', 'vpsubsb', 'vpsubsw', 'vpsubusb', 'vpsubusw', 'vpsubw', 'vptest',
'vpxor', 'vpxord', 'vpxorq', 'vsubpd', 'vsubps', 'vsubsd', 'vsubss', 'vxorpd',
'vxorps', 'xabort', 'xaddb', 'xaddl', 'xaddq', 'xor', 'xorb', 'xorl', 'xorpd',
'xorps', 'xorq', 'xorw', 'xrelease', 'xrstor', 'xrstors', 'xsave', 'xsave64',
'xsavec', 'xsaveopt', 'xsaveopt64', 'xsaves', 'xsha1', 'xsha256', 'xstorerng',
'xtest'],
```

B. Random Test Seeds

a. [1, 9, 83, 85]