Security Counters

A Project Report

submitted by

AJAY SAJU JACOB

in partial fulfilment of the requirements
for the award of the degree of

DUAL DEGREE (B.Tech and M.Tech)

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

June 2021

THESIS CERTIFICATE

This is to certify that the thesis titled **Security Counters**, submitted by **Ajay Saju Jacob**, to the Indian Institute of Technology, Madras, for the award of the degree of **Dual Degree (B.Tech + M.Tech)**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Chester Rebeiro
Research Guide
Professor
Dept. of Computer Science
IIT Madras, 600 036

Prof. Nitin Chandrachoodan
Research Guide
Associate Professor
Dept. of Electrical Engineerin
IIT Madras, 600 036

Place: Chennai

Date: 20/06/2021

ACKNOWLEDGEMENTS

I thank my guide Prof. Chester Rebeiro, co-guide Prof. Nitin Chandrachoodan and Nikhilesh Kumar for all the guidance that they gave throughout the course of the project. I would further like to thank The National Mission on Interdisciplinary Cyber Physical Systems and SAMSUNG India Electronics Limited for supporting my project through the "SAMSUNG IITM PRAVARTAK TECHNOLGIES FOUNDATION Fellowship".

ABSTRACT

The use of Hardware Performance Counters (HPCs) to profile malicious behavior of programs has grown popular in literature in the last decade (Demme *et al.*, 2013). However, the deployment of HPCs in malware detection is an unintended consequence from the design point of view of these counters. In this work we try to explore the possibility of designing counters that are tuned for malware detection at design time. To this end we plan to come up with, (a) a metric to represent to quantify the malicious behavior of a program as a function of its execution parameters, eg. instruction sequence, (b) a model that takes the sequence of this metric observed during the course of the program execution and classifies malicious and benign behavior.

TABLE OF CONTENTS

CHAPTER 1

Background

1.1 Introduction

The proliferation of computers in any domain is followed by the proliferation of malware in that domain. Viruses, rootkits, spyware, adware and other classes of malware exploit the ever increasing number of vulnerabilities in systems to execute malign code. Conventional software based anti-virus systems have been used to detect and quarantine such malwares. However they have a few key flaws:

1. The anti-virus systems are itself vulnerable to attack due to bugs in the anti-virus or at the OS/kernel level.
2. Production anti-virus software typically use static characteristics of malware such as suspicious strings of instructions in the binary to detect threats. However these can be overcome by data obfuscation techniques that malware writers use to generate semantically similar code which look drastically different.

This is where we see the advantages of a hardware based malware detection system

1. It cannot be shut down even if kernel is compromised.
2. It might be able to protect against kernel exploits and other attacks against hypervisors.
3. As we have access to process information at the instruction level, it is easier to add arbitrary static and dynamic monitoring capabilities.

The seminal work in this direction was Demme *et al.* (2013), which used the existing Intel performance counters to build a HW based malware detector. The work showed that the data from performance counters can be used to identify malware. The dynamic detection techniques they employ were found to be robust to minor variations in malware programs. They propose a HW modification which allows the detector to run securely beneath the system software. However, from a design point of view, the deployment of performance counters in malware detection is an unintended consequence.

1.2 Problem Statement

We try to explore the possibility of designing counters that are tuned for malware detection at design time. To this avail, we plan to come up with,

1. A metric to quantify the malicious behavior of a program as a function of its execution parameters, eg. instruction sequence
2. A model that takes the sequence of this metric observed during the course of the program execution and classifies malicious and benign behavior.

1.3 Program representation

Developing a program representation is a problem that has several long standing implications in the realms of understanding the inner workings of malware, finding vulnerabilities in existing systems, and detecting patent infringements in released software. General program representation tools depend on generating a robust vector representation of the assembly code in order to get a glimpse at the semantic relevance of portions of the code. The conventional manual feature engineering techniques of developing signatures for specific portions of code has been shown to be ineffective against different compiler optimizations and data obfuscation techniques which have been used by malware developers to generate semantically similar codes that look distinct from each other.

Several works have surfaced recently which attempt to curb this flaw by learning the lexical semantic relationship along with feature vectors for an assembly function, so that the semantics of any piece of code has a part to play in the sections representation. Platforms like asm2vec (Ding *et al.*, 2019) which is primarily a binary clone search tool are examples of the same and along with features to check similarities between 2 assembly functions, it also learns a semantic vector representation for every token in the assembly code. Another work relating to the same is code2vec (Alon *et al.*, 2018), which tries to generate a fix-length vector for any code snippet, that can be used to predict the semantic propoerties of the snippet. This is performed by decomposing code to a collection of paths in its abstract syntax tree, and learning the atomic representation of each path simultaneously with learning how to aggregate a set of them.

However these are both static models. They do not explicitly run the code snippet provided but simply generate representations of the same. In this work, we are interested in hardware based malware detection, which would be mean a dynamic tool that detects presence of a malware during runtime. Hence in this work, we attempt to develop a vector representation for the x86 instruction set, by looking at the instruction sequences during execution of a program and by using NLP techniques we generate semantic embedding on the instruction set. We test the robustness of the same by observing how changing the base code corpus affects the vocabulary learnt along with compiler optimizations. We then evaluate the usefulness of such a vector representation by using it for applications like program type classification and trying to establish various types of code blocks encountered in a typical program.

1.4 Experimental Plan

1. Build a corpus $\mathcal{P}$ (at least 40K programs) of random programs collected by different means like via web crawlers, public databases and program generation tools.
2. Divide $\mathcal{P}$ into two random halves $\mathbf{P}$ and $\mathbf{Q}$.
3. Use Intel Pin tool (Luk *et al.*, 2005) to generate the instruction sequence I_i^P and memory access sequences M_i^P for each program $P_i \in \mathbf{P}$. Similarly, generate the instruction sequence I_i^Q and memory access sequences M_i^Q for each program $Q_i \in \mathbf{Q}$.
4. For both halves $\mathbf{P}$ and $\mathbf{Q}$, use language models like word2vec or Glove to come up with a set of mathematical embeddings, E_P and E_Q for each instruction. Figure out a similar way to represent the memory access sequences. Essentially we want to have $E_P \equiv E_Q \equiv E_{\text{inst}}$. This would mean that E is a stable and universal embedding on the instruction set. A similar approach can be taken for the memory access sequences to get E_{mem}.
5. Design $c = f(E_{\text{inst}}, E_{\text{men}})$.
6. Build a model $\mathbf{M}$ that takes in c values for a program during its execution and classifies it as malicious or benign.
7. Iteratively refine c and $\mathbf{M}$.
8. If the number of parameters to calculate c becomes huge we can also split them into multiple metrics, hence multiple counters and in turn multiple features to $\mathbf{M}$.

CHAPTER 2

Experiments

2.1 Data Collection

In order to obtain a sufficient number of programs to build the vocabulary, the first approach used was to build a Github crawler to sample repositories which are primarily C-based. Initial attempts were using standard scraping libraries like Scrapy, but for URLs of github repositories, the hit-rate was simply too low.

To get a better success-rate, the Git API was used (the python library of it). The query feature can be used to randomly sample repositories. The response for each repo also contains a dictionary for language-wise split-up of # of lines.

Every response can only give atmost 1000 repos per query. And so the query was made date wise. Further, size of the repos were filtered in query to be between 1kb-1Mb, in order to remove empty and massive repos.

There is a 60 req/minute and 1500 req/hour rate limit per user('s access token), which although we were never exceeding, seemed to be triggered at around 20 minutes. Hence access tokens from multiple (5) accounts were used and switching was done when one tokens rate is exceeded. In finality, a rate of about 90 successful repos were cloned every 10 minutes(running queries for 3 separate years simultaneously).

The current number of predominantly C based repos (in which atleast 55% of the lines of code are in C) on a year wise split is:

Year	# Repos
Before 2012	2167
2012	2030
2013	3209
2014	4089
2015	5316
2016	6518
2017	7167
2018	8259
2019	7669
2020	4595
Total	51019

Another source of C programs was the code corpus used for the work Karampatsis *et al.* (2020). It boasts 4601 projects, with around 750k individual C programs. However as the work is about using the code corpus to generate a uniform vocabulary using NLP, a majority of the programs were lacking necessary scripts and other dependencies to be run. In total only about 7k programs were compilable and hence usable.
Further the csmith tool (Yang *et al.*, 2011) was used to generate random C programs to generate large purely C based code copora.

2.2 Extracting Instruction Sequences

As a hardware based solution would essentially have information of a process at the instruction level, a way simulate this would be to study the dynamic execution of a C program. The Intel Pin tool (Luk *et al.*, 2005) can be used to insert code into a program to collect run-time information. These instrumentation functions can be used to monitor the program during dynamic execution, including printing out the instructions that are being executed. We will be using this to generate the instruction mnemonics of various programs and use this to build a general vocabulary for each instruction.

2.3 Using asm2vec to generate vocabulary

The asm2vec tool is a platform that can be used as binary clone search tool, to check similarities of various functions between 2 separate program. For each token (which is any part of the executable, including function name, instruction, operands etc), the tool learns 2 multidimensional vectors, out of which one is used for checking similarity and the other is used to hold the semantics learnt about the token.

What is to be noted is that asm2vec is a **static** analysis tool unlike pin. Only the executable is required for the tool to learn a vocabulary for the various instructions of the program and the dynamic running of the executable does not come into play in this case.

However the tool learns a vocabulary for each executable. Hence we learn a vocabulary from a batch of files by simply initializing the vocabulary at each point as the vocabulary learnt from previous executables. As discussed before, in order to test the robustness of the vocabulary learnt, we perform the algorithm on 2 different data sets and observe the level of similarity between the 2 learnt vocabularies. We will be using the cosine-similarity metric for the sake of these experiments.

2.3.1 Preliminary Tests

We use 2 datasets of 1k programs each and examined the average cosine similarity between the learned vocabularies. Hence at each file we have a vocabulary that is learnt from all the programs till then. The initial values we received were:

Repo 1 File No.	Repo 2 File No.	Cosine Sim	# Tokens
1000	1000	0.0031	16187
10	10	-0.004	322
100	100	0.00016	1108

Within the same repository the similarity remained pretty consistent from one file no. to the next (similarity between file no. 900 and 1000 was 0.985). The observations from the same are,

- The number of tokens increases to around 10k when many of these tokens are just

operands which rarely occur again. Hence their semantics play no real part in the similarity of the vocabularies.

- Between different permutations the cosine similarity is quite low. However in the same permutation on comparison of different file numbers, the similarity was still quite high.

Inferences:

- Perhaps due to the blowup in number of tokens, a token is being rarely used and so nothing much is learned about it. Hence checking cosine similarity of just the instruction set is a viable option.
- Performing the same tests on larger datasets might see improvement.

2.3.2 Final Test

We now use data-sets with 10k programs each in order to record behaviour. For calculating the results in this case, we only look at the cosine similarities of the x86 instruction set, as non-operation based tokens may not have enough test cases in the programs, to learn anything relevant about. Further we are looking at only instructions that have a sizeable fractions of occurrences in the corpus. Let this threshold be denoted by t. For different values of the t, the following results were obtained.

Repo 1 File No.	Repo 2 File No.	t	# Tokens	Cosine Sim
10000	10000	0.001	48	0.019
10000	10000	0.01	13	-0.002
10000	10000	0	226	0.023
1000	1000	0.001	48	0.022
100	100	0.001	48	0.028

Another concern was that the order of input of the programs are having an effect on the learned vocabulary. In order to remove this bias a random subset of 8k program each out of each dataset was permuted and the results computed.

Repo 1 File No.	Repo 2 File No.	t	# Tokens	Cosine Sim
8000	8000	0.001	48	-0.11
1000	1000	0.001	48	-0.107
100	100	0.001	48	0.099

Although the value of the cosine similarity is considerably larger than in the first case, its similar value throughtout the file nos. seems to suggest that it is simply because of initialization that these elevated values are observed.Hence we infer

- The asm2vec tool seems to be unable to pick up on similar embeddings for the instruction set, when it is learnt on 2 seperate datasets. This may be due to a variety of reasons besides the possibility of the original assumption being false.
- The asm2vec is a static tool, whose original purpose is as a binary clone detection software. Hence the semantics that the tool learns might just be particular to that program and not hold any information about the general dynamic beviour of the tokens.
- It could also be that we are still using too few programs to observe any meaningful results.

2.4 Using word2vec to generate vocabulary

A piece of code can be viewed as a series of instructions which could be modeled effectively as sentences. Hence we train word2vec (Mikolov *et al.*, 2013) with the series of instructions that are executed in a program and observe the vocabulary that is generated. What is to be noted here, is that as we are looking for the exact sequence of instructions that are executed when the code is run, and hence this would be a dynamic analysis. Pin tool will be used here to print out the instruction mneumonics during run time.

In order to learn a vocabulary from a dataset of multiple programs, the instructions mneumonics of the programs are appended into a single file which is passed as input to word2vec. This method would have the disadvantage that at the intersection of 2 programs, a few instructions would be misidentified as being in the same neighbourhood when they belong to different programs. However considering that the number of instruction is very large, the number of such neighbourhoods would be a very small fraction.

In order to populate the datasets, csmith was used which generates random programs (in addition to programs from the previous corpus). Two datasets of 20k programs were generated and their corresponding pin tool outputs were obtained.

2.4.1 Preliminary Tests

The 2 datasets of 20k programs are now passed on to word2vec and vocabularies are learnt for the same for different parameters:

- size: The dimensionality of the vectors learnt for each instruction in the vocabulary.
- window: Size of the window considered in the neighbourhood of a token.
- sample: Threshold for frequency of a word. A word with higher frequency will be randomly downsampled.
- negative: number of negative samples used in the training process.
- iter: Number of iterations used in the learning process.
- min-count: discard words that are less frequent
- cbow: 1 for using bad of words model. 0 for skip gram

Note: When running pin tool for the programs generated by csmith, as the pin tool is a dynamic tool, a timeout was set on the execution of the executable. The timeout was set as 2 mins for repo1, but for repo2 it was relaxed to 1 min as the hitrate for a program to go beyond this limit remained about the same ($\sim 10\%$) and so it saved a lot of time to use a smaller timeout than 2 mins.

In terms of metrics to compare similarity between 2 vocabularies, 5 metrics have been tested:

- Universality: Reciprocal of the standard deviation of the euclidean distance between vectors of the same word.
- Reciprocal of the coefficient of variation of the Euclidean distances.
- Cosine Sim: Average of the cosine similarity between vectors
- Reciprocal of the standard deviation of the angle between vectors
- Reciprocal of the ratio of variation of angle between the vectors.

Out of the several parameter combinations below are a few sampled results. (parameter values are in order as listed and default cbow was used)

Params	Univ	Ufrac	Csim	Cstd	Cfrac
5,1,1e-5,3,5,1	11.017	0.636	0.681	1.241	0.630
5,1,1e-5,3,5,100	10.990	0.639	0.679	1.238	0.634
5,1,1e-5,3,5,1000	10.963	0.643	0.676	1.235	0.638
10,1,1e-5,3,20,1000	16.772	0.697	0.692	1.386	0.692
10,2,1e-5,5,5,1	16.845	0.689	0.697	1.393	0.684
10,2,1e-5,10,5,100	16.808	0.693	0.695	1.390	0.688
10,3,1e-5,5,20,100	16.808	0.693	0.695	1.390	0.688
25,1,1e-5,5,5,100	25.722	0.699	0.673	1.352	0.699

Some takeaways from the tests,

1. Although with an increase in vector sizes we saw a marked improvement in metrics, the total number of instructions (tokens) observed was 120 and so at higher sizes we would simply be overfitting.
2. In smaller values of the window, for popular instructions like add, the 2 vectors that were generated from the 2 datatsets were pretty much identical. As even a single line of code comprises of several instructions, it is expectable that at such low window values, something of this sort is observable. The influence of larger window sizes is to be explored.
3. Similarity between different instructions in the same vocabulary would also be an interesting avenue.
4. Variations between bag-of words and skip-gram models of computation is to be explored.

2.4.2 Testing variation of window and model types

We focused mainly on the variation of the similarity metrics with the window and the computation model parameters. For size 10, the following table depicts the same:

Window	cbow	Univ	Ufrac	Csim	Cstd	Cfrac
1	0	16.808	0.693	0.695	1.390	0.688
1	1	16.808	0.693	0.695	1.390	0.688

3	0	16.808	0.693	0.695	1.390	0.688
3	1	16.808	0.693	0.695	1.390	0.688
5	0	16.808	0.693	0.695	1.390	0.688
5	1	16.808	0.693	0.695	1.390	0.688

What we observe was that for a particular value of size, the obtained vocabulary is independent of the window and computation model parameters. And comparing the vocabulary learnt between 2 different datasets, we see that for popular instruction (with very high frequency), the learnt vectors are identical.
Hence we infer from this experiment that,

- The window parameter doesnt seem to have an effect on the learnt vocabulary. This suggests that intrinsic semantic relationship between instructions is quite simple, and the immediate neighbourhood of the instruction is enough to describe this relationship.
- For very frequent instructions, the vector representation learnt from both datasets are identical and on average there does seem to be a high degree of similarity between the 2 learnt vocabularies. Hence we conclude that the algorithm is in fact able to generate a robust embedding for x86 instruction set.

2.4.3 Testing effect of compiler optimizations

We now test how the vocabulary changes when we compile programs with compiler optimizations. This is relevant as robustness to compiler optimization is a very desirable feature in a code embedding as if it remains relatively invariant to compiler optimizations, then it implies that the semantics of the instruction set learnt is invariant to the same, which means any hardware based malware detection system built on this embedding would also be resistant to compiler optimization based morphing of malicious code.

To test the same, we compile the programs of repo1 with different gcc optimization flags and use this to build a code embedding using word2vec and the intel pin tool.We then study its similarity to repo1 (which is essentially the same set of programs but without

the compiler optimizations) and repo2 (which is an entirely different set of programs). The results of the same are tabulated below:

Flag Used	Comparison with	Window	Univ	Ufrac	Csim	Cstd	Cfrac
-03	repo1	3	9.843	2.049	0.089	1.283	1.851
-03	repo1	5	15.240	2.414	0.126	1.502	2.106
-03	repo1	10	23.989	2.919	0.030	1.895	2.847
-03	repo2	3	9.973	2.109	0.063	1.306	1.922
-03	repo2	5	15.293	2.496	0.076	1.506	2.197
-03	repo2	10	24.597	2.976	0.041	1.931	2.877
-01	repo1	5	14.164	2.231	0.124	1.542	2.140
-01	repo2	5	14.622	2.354	0.087	1.568	2.244
-0s	repo1	5	15.527	2.587	0.067	1.649	2.410
-0s	repo2	5	16.038	2.695	0.055	1.705	2.522
-0fast	repo1	5	15.227	2.410	0.127	1.501	2.105
-0fast	repo2	5	15.325	2.486	0.085	1.500	2.174
-0g	repo1	5	15.749	2.606	0.083	1.660	2.406
-0g	repo2	5	16.187	2.734	0.057	1.705	2.535

We observe that although the cosine similarity metric has deteriorated in value, the other metrics seem to have not undergone such a massive change. The rigidity in the value of Cstd seems to suggest that all the vectors have simply been rotated by a certain constant angle. On visible inspection of the learnt vocabularies, we see that for the instruction 'mov', the embedding learnt remains intact while there are significant changes for every other instruction. This suggests that although the current vocabulary is not robust to compiler optimizations, it might be the case that usage of a larger dataset of programs would be able to generate more robust embeddings on the instruction set.

CHAPTER 3

Evaluation and Results

For the experiments attempted in this section, The TreeBasedCNN Database has been used

3.1 Labelling Code Blocks

There has been work done in identifying several distinct phases during program execution. Hence we can consider a program as a series of blocks and we investigate whether we can find similarities between blocks across programs.

By the principle of proximity, we assume that a block will contain instructions that are close to each other, and so any instruction that causes a jump in the instruction pointer (like jumps, branches, calls etc), signify the end of the block. This now provides us with a set of varying length sequences, with each sequence being the instruction mnemonics of the block. The problem now becomes a unsupervised learning problem, of finding structure amongst the various block mnemonics.

For each block, we translate the mnemonics to 2-dimensions by transforming each instruction to its learned vocabulary vector. Hence now we obtain a set of variable size 2-D vectors each representing a block from the programs.

3.1.1 Preliminary Test

For the first set of tests, we flatten the 2-D vector obtained for each block in order to obtain a single variable length vector associated with each block. We do this procedure on the feasible programs from the TreeBasedCNN dataset from the class 78 and 41. In terms of the frequencies of the sizes of the vector, we see that a vast majority are within the size of 20 (20 instructions).

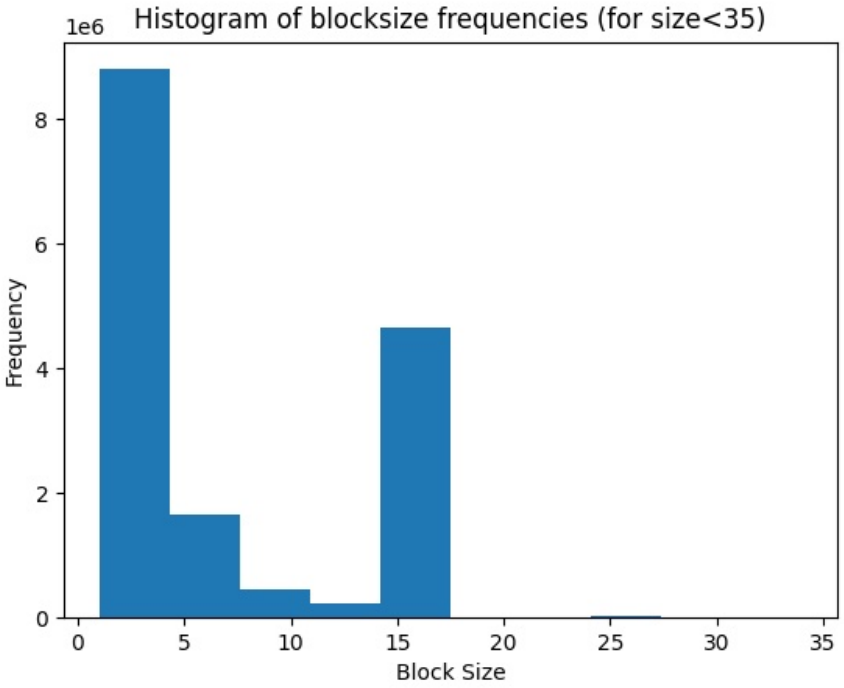

Figure 3.1: Histogram of block size (for size<35)

However it was observed in the histogram of blocks above this size, we notice an interesting group of blocks at around size 300. Although the frequency of these are very few, their relative frequency in comparison to other large block sizes is indeed interesting.

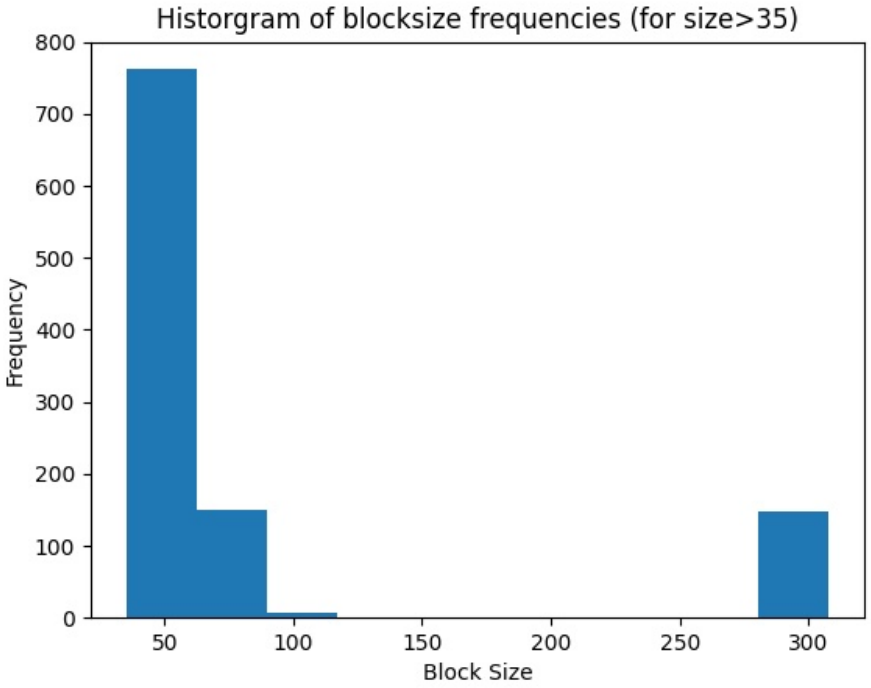

Figure 3.2: Histogram of block size (for size$\geq$35)

In order to tackle the problem, we tried the most straight-forward approach, which is to use k-means clustering on the given vectors and obtain labels for each vector. However k-means can only be applied to data which is of the same dimensionality, for which we pad all the vectors upto size 30 (we remove blocks of size > 30 as these are rare). We had a total of 15816569 such vectors for the clustering process. (Note: Sklearns k-means clustering package gave seg fault when trying to run with the vocab of size =5 , and hence we resorted to use the vocab of size =3). $k = 2, 3, 5, 10$ were the values of k for which the labels were generated.

In order to qualitatively assess the clustering, some method of being able to represent the data in lower dimensions was needed. We used t-SNE (t-distributed stochastic neighbor embedding), to develop an embedding for each vector in 2 dimensions. We then plotted the same along with the labels on a scatter, in order to see if clusters are qualitatively sound. The results are shown below for the various k-values. (Note: For the t-SNE fitting step, only a random sample of 100k vectors from the original input pool was used, as it was taking too much time to process the complete input).

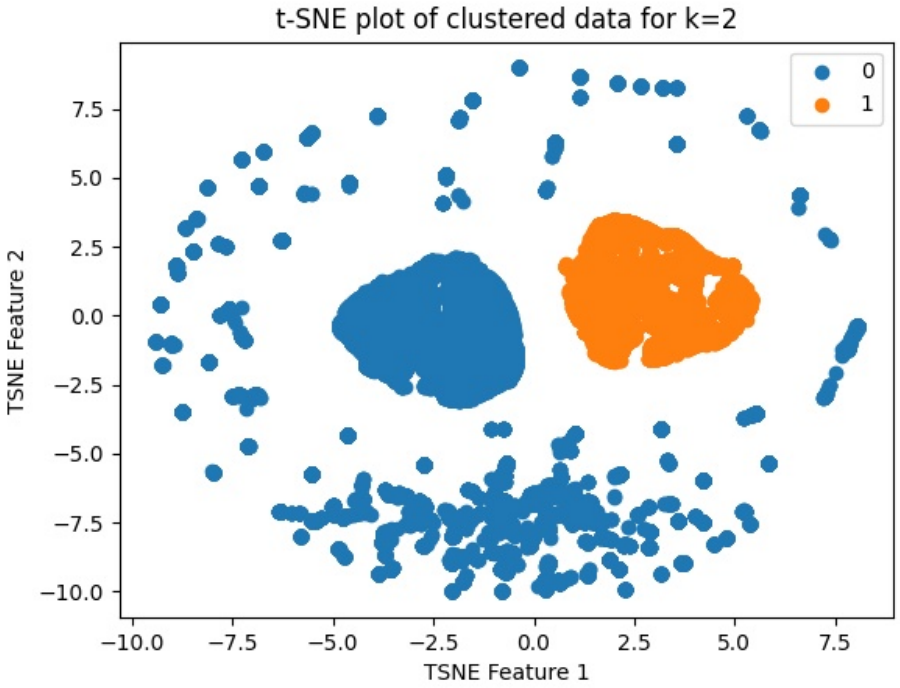

Figure 3.3: t-SNE scatter plot for k=2 (Preliminary Test)

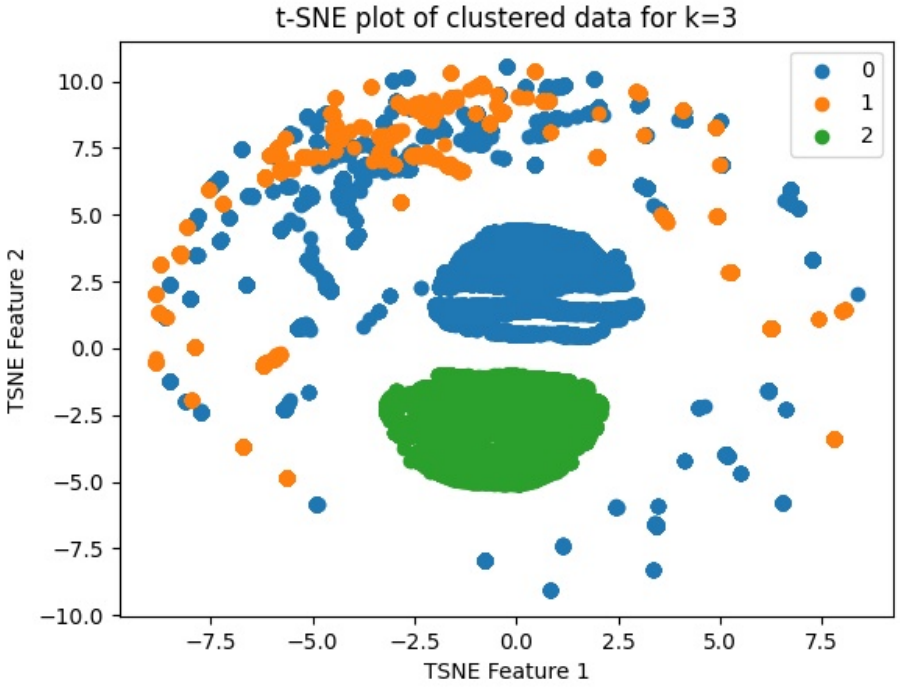

Figure 3.4: t-SNE scatter plot for k=3 (Preliminary Test)

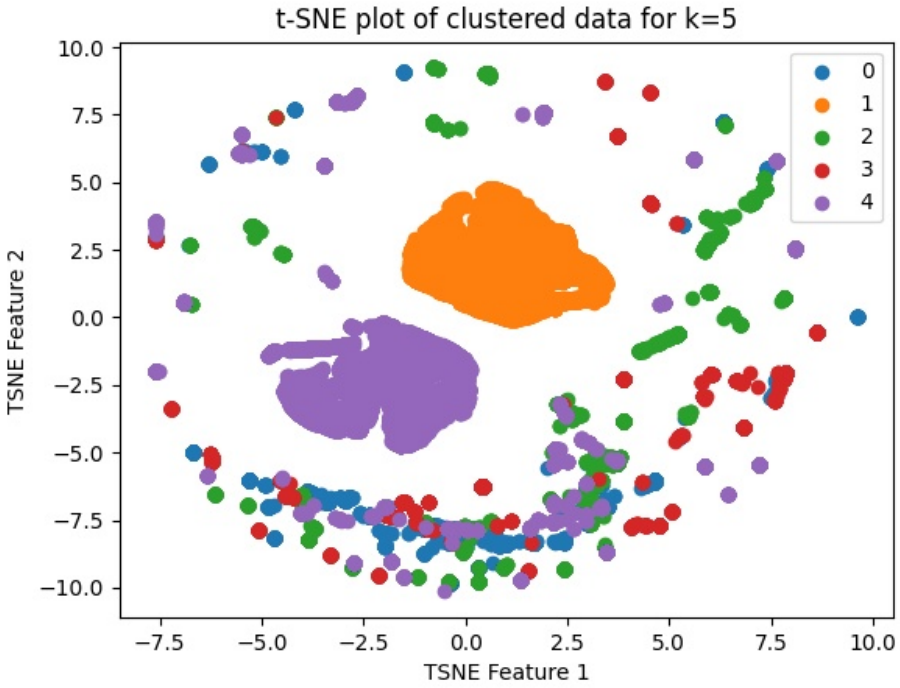

Figure 3.5: t-SNE scatter plot for k=5 (Preliminary Test)

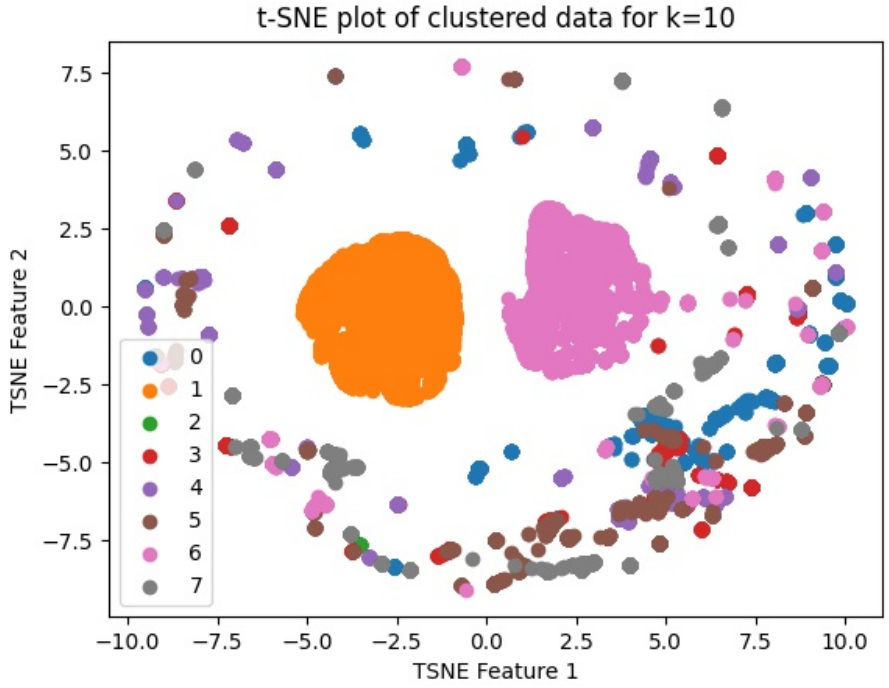

Figure 3.6: t-SNE scatter plot for k=10 (Preliminary Test)

From the t-SNE plots, it clear that there are 2 clusters of similar blocks in the data. However there does seem to be a lot of blocks that cant be classified accurately. There could a variety of reasons for the issues in the clustering,

- k-means clustering might be an inadequate approach for the data provided. The vectors are all of different sizes, while for k-means generally we expect vectors of the same dimensions.
- While padding to make all the vectors of the same size, the ordering of the dimensions is important as k-means is sensitive to the ordering of the features in the vector. Hence the padding is influencing the data.
- There might be several repetitions of the same block which is skewing the data.

3.1.2 k-means clustering on large dataset

We looked at now clustering the original dataset of 20 thousand programs, as the current working data set is quite small. It was noticed in the original preliminary test that a lot of blocks were duplicates. For eg, the label 0 blocks from the previous experiment (for k=10), were all exclusively the same sequence. Hence we removed duplicate sequences for this iteration and worked on clustering the unique blocks encountered. It was noticed that there were 250k unique blocks identified in our code (of size < 30 instructions) and k means clustering was attempted on this data for $k = 2, 3, 5, 10$. The t-SNE plots for same are as follows:

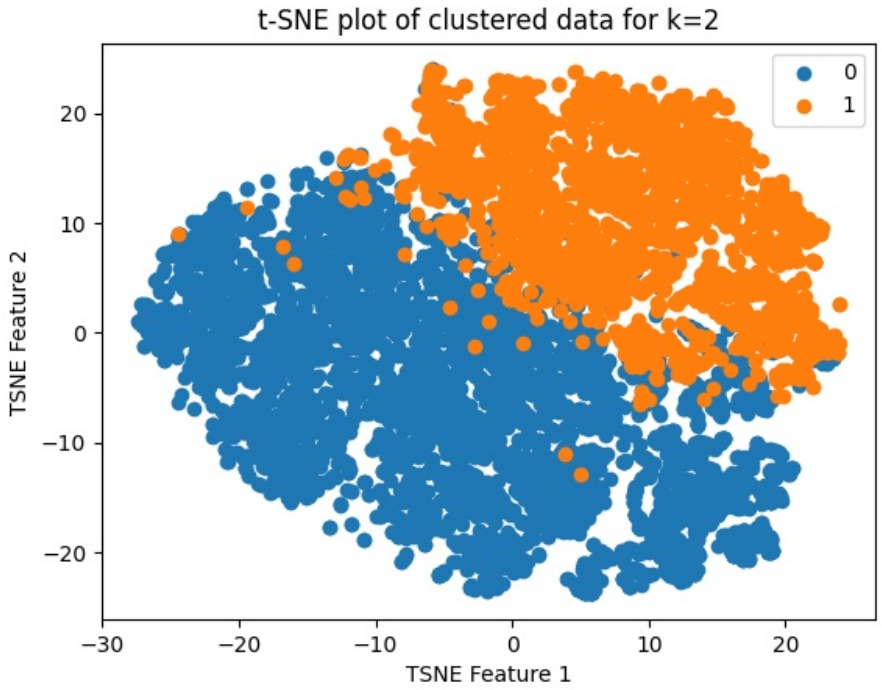

Figure 3.7: t-SNE scatter plot for k=2 (Large Dataset)

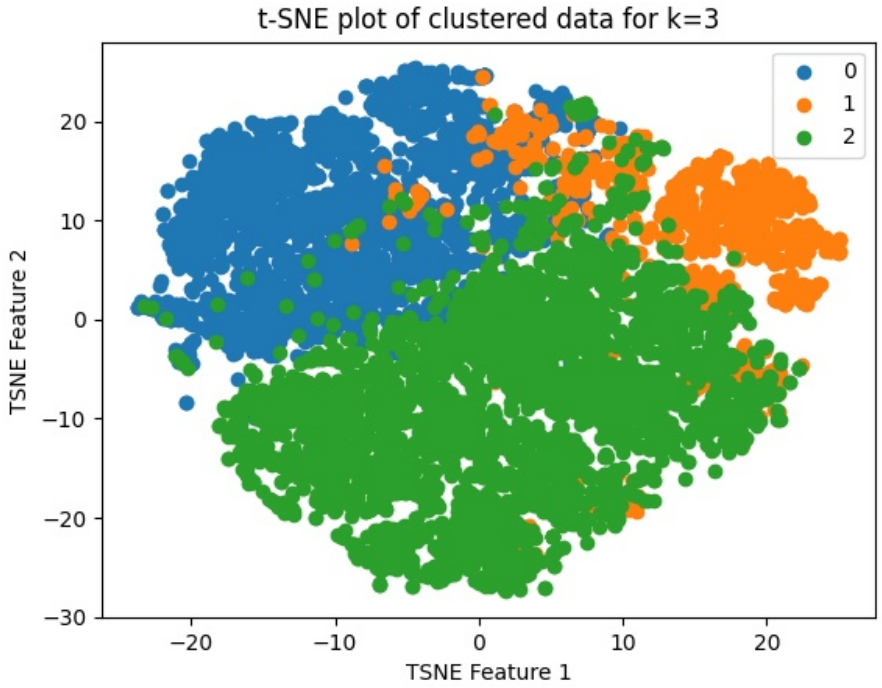

Figure 3.8: t-SNE scatter plot for k=3 (Large Dataset)

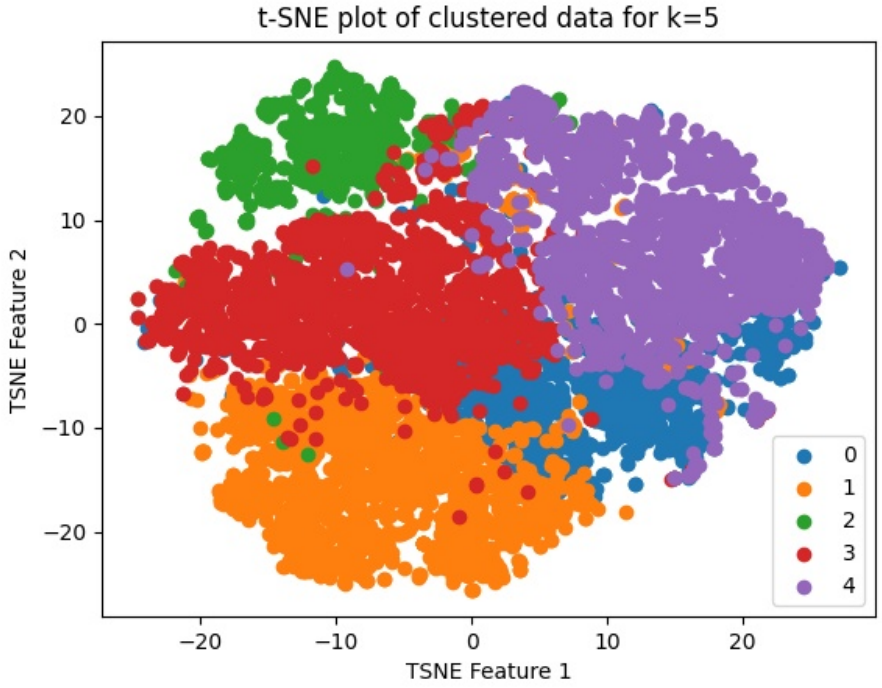

Figure 3.9: t-SNE scatter plot for k=5 (Large Dataset)

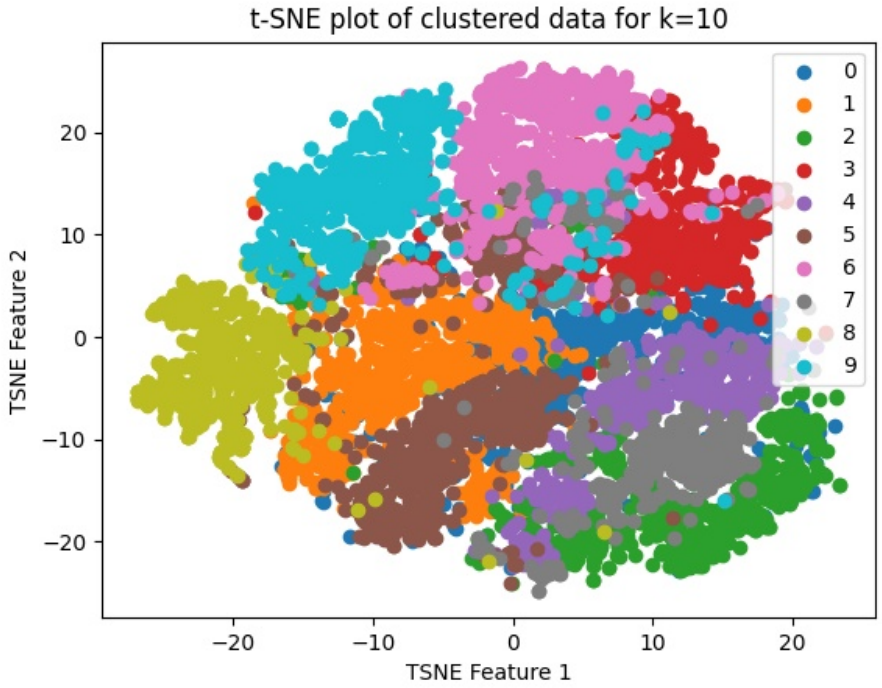

Figure 3.10: t-SNE scatter plot for k=10 (Large Dataset)

3.1.3 Using DTW for time series clustering

As the instruction sequences are in order of execution and if we assume that our vocabulary is able to retrieve some sort of semantics about each instruction, we can think of the sequences of vectors in a block as a time series of sort. A primary concern we had when clustering using k-means was the relevance of order. It could be 2 blocks are similar but one is an offset version of another, but both perform the same semantics. After padding, this could mean that these 2 sequences are labelled differently when they are semantically similar. Hence to curb this flaw we use the DTW (dynamic time warping) algorithm as the distance metric between sequences. The plots we obtain using t-SNE for the same are below:

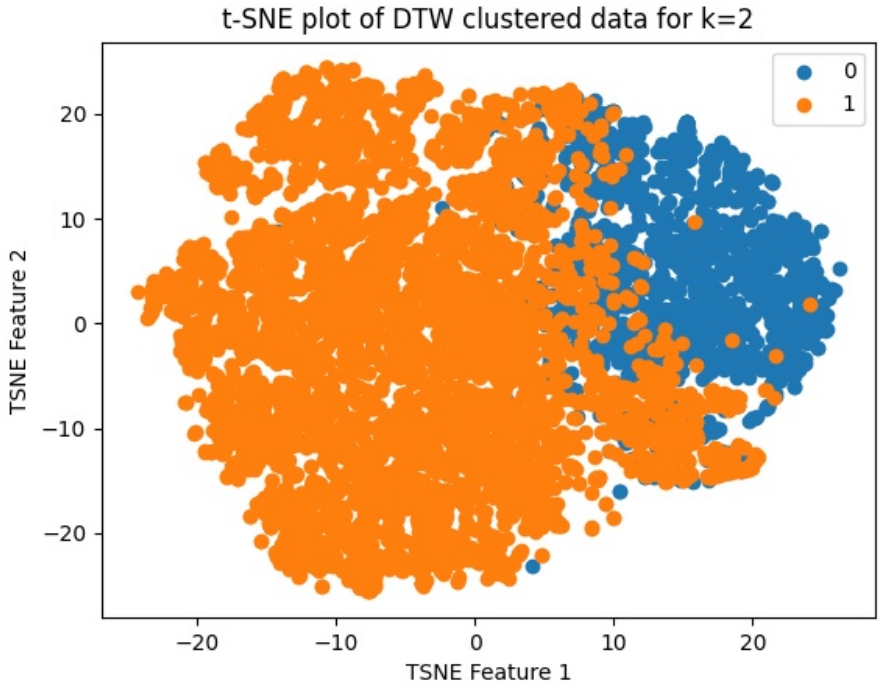

Figure 3.11: t-SNE scatter plot for k=2 (DTW Clustering)

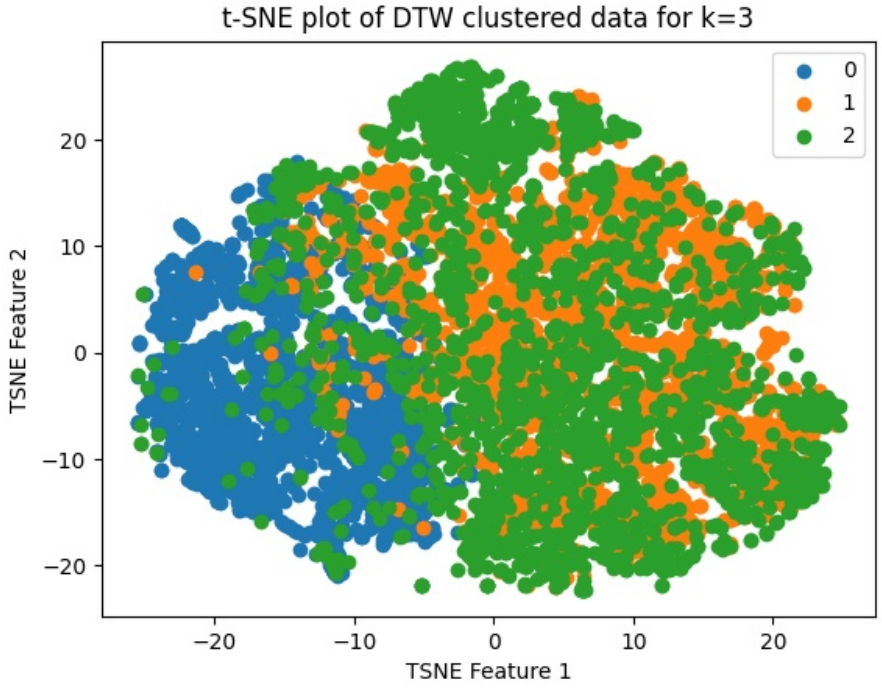

Figure 3.12: t-SNE scatter plot for k=3 (DTW Clustering)

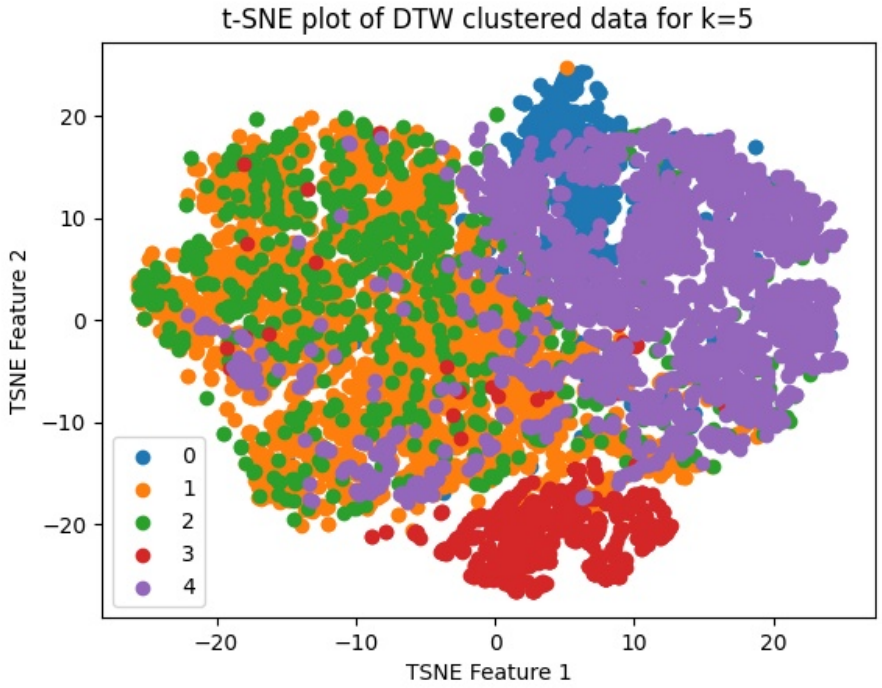

Figure 3.13: t-SNE scatter plot for k=5 (DTW Clustering)

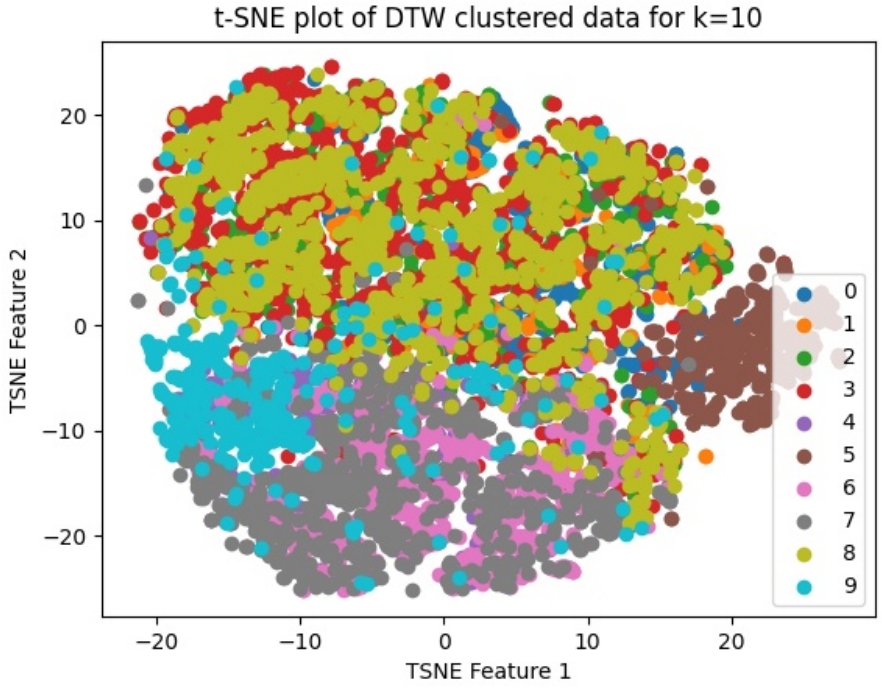

Figure 3.14: t-SNE scatter plot for k=10 (DTW Clustering)

Visually this clustering does seem worse than the previous experiment. But this is to be expected as t-SNE tries to maintain the relationship of distances between points. However the time series k-means clustering algorithm takes DTW as its distance metric,

and so even if the euclidean distance between a point and its centroid is large, it might have a small distance wrt. DTW.

3.2 Predicting Program Types

In order to evaluate how effective our learnt vocabulary is at representing the semantics of each instruction, we attempt to use it to solve a classification problem. We use the TreeBasedCNN database for the same. The dataset has a large number of programs labelled according to type. We tried to find program types that have large number of programs that are compilable. From the same we isolate 2 labels, 41 and 78 which have 59 example programs each which are feasible. We attempt to perform binary classification on these 2 sets with the input we have as the instruction mnemonic of the programs. As defined for clustering, we define a block in the program as a sequences of instruction in the mnemonics of the program that are terminated by any instruction that can cause a jump in the instruction pointer (due to observations made earlier, we also enforce that a block has maximum length 30).

3.2.1 Preliminary Tests

Every instruction is represented by a vector in the vocabulary. Hence we represent a program as a sequence of vectors and we flatten this 2-D object to obtain a single variable size vector for each program. This is taken as the feature set of input. As this is a variable length vector, padding is performed to maintain same size. As the ordering of instructions is relevant. we would want to use a model that is able to take into account the ordering of the vectors. Hence the hidden layer of our model is an LSTM with 32 nodes followed by an output FCNN (Fully connected Neural Network) layer to predict the label utilizing the weights learnt in the LSTM layer. An important model parameter in this case, is the number of instruction that are considered as a single unit of time. If we assume this parameter is some value 'k', then the input at each timestamp would be the flattened vector of the vocabulary representations of k instructions. The table below records our result of the model's prediction on the test data (0.2:0.8 test-train split)

# Instructions	Accuracy	f1-score	precision	recall
100	0.5	0.66	0.5	1.0
500	0.545	0.6875	0.524	1.0
1000	0.64	0.6875	0.524	1.0
10000	0.682	0.625	0.476	0.91

From the results tabulated we can see that as the parameter is increased, the model is able to improve over random guess and is actually able to differentiate between program type with some (limited) confidence. An issue with this approach is the fact that the vectors are quite large and so overfitting is inevitable when compounded with the small dataset.

3.2.2 Proof of Concept

We make the following assumption: **The program type is a complex function of the blocks we encounter during dynamic execution**. Hence we first one-hot encode every distinct block that we observe in the dataset. Then we represent each program as a histogram on this one-hot encoding (by inspecting the frequency of each block in the programs' instruction mnemonic). We use this transformation as our models input, with output the model tries to predict being the program type. This is in essence a feature vector binary-classification problem. We used the xgBoost classifier to model the problem and found that the model had perfect accuracy in predicting the program type on the test set. Hence we conclude that our underlying assumption is true. However, the experiment in this fashion cannot simply be extended to larger datasets. In the large dataset consisting of 20,000 programs, we had approximately 250,000 unique blocks which would mean the histograms generated would be have these many bins. This would make the model computationally expensive and prone to overfitting.

3.2.3 Using clustering labels for classification

As discussed in the previous section, simply one-hot encoding all the unique blocks discovered in the data would lead to computational difficulties. To circumvent this, we label each block according to the clustering processes described in section **3.1**. We then look at the labels of the blocks present in a program and prepare a histogram from the same. In this fashion we are able to augment the input to a constant size and if the clustering is able to derive any semantic meaning, we hypothesise that the program type is a complex function on the histogram of labels in the instruction mnemonic. Tabulated below are the results on the test data when using different types of classification models and clustering algorithms.

Clustering type	Model	Accuracy	f1-score	precision	recall
k-means (k=3)	Dense(4)+Dense(2)	0.53	0.7	0.55	1.0
k-means (k=3)	xgBoost	0.96	0.96	1.0	0.92
k-means (DTW, k=3)	Dense(4)+Dense(2)	0.61	0.66	0.5	1.0
k-means (DTW, k=3)	xgBoost	1.0	1.0	1.0	1.0

We observe that the xgBoost classification algorithm is able to perform very well on the data, and for the DTW based time series clustering, it is completely accurate on the test data, which seems to suggest that there is a strong possibility that our assumption is right. However due to the small size of the training set, it is possible that there is significant bias in the dataset and the classification algorithm is detecting features that are present due to these inherent biases.

CHAPTER 4

Conclusions and Future Work

In conclusion, in this work we explored the possibility of developing a vocabulary for the x86 instruction set by observing the instruction mnemonics of typical C programs. We test the robustness of the same by comparing the learnt vocabularies between 2 different code corpuses and found that the learnt vocabularies are quite similar by various metrics and very frequently instructions actually have identical representations in both vocabularies. We further evaluated the vocabulary learnt by using it for identifying various blocks within the program instruction mnemonics and using this we attempted to classify program types.

With the aid of the vocabulary, we can now attempt to find structural differences between a specific class of malware and benign code. The ultimate goal of the same would be to generate a metric/function that can be used to differentiate between the malicious and benign code. We then can implement the same as a dedicated hardware counter which is able to dynamically detect malicious code running in the system.

APPENDIX A

Terminologies and definitions

A.1 Sequence Learning Terminologies

1. Crawler : A crawler is a script which automatically scowers the World Wide Web in a methodical fashion.

2. Github: A distributed version-control system.

3. Code Corpus: A large set of structured code.

4. API (Application Program Interface): It is a set of programming codes and rules that are used for data transmission between a web server and a client.

5. Intel Pin Tool (Luk *et al.*, 2005) : Pin is a dynamic binary instrumentation framework for the IA-32, x86-64 and MIC instruction-set architectures that enables the creation of dynamic program analysis tools. As a hardware based solution would be looking at the behaviour of any process dynamically, the tool was used to monitor program behaviour during execution and particularly for extracting useful features like instruction mnemonic during execution.

6. asm2vec: As described in Ding *et al.* (2019), asm2vec is a clone search tool which circumvents code obfuscation and compiler optimization based augementations to code. The asm2vec tool takes as input an executable and learns a vocabulary for each token (any function name, instruction or operand in the executable), and can be used to check similarity between 2 executables. It also learns a semantic vocabulary for each token. We will be using this fact to try and achieve a generic vocabulary for instructions. asm2vec is based on the popular NLP tool word2vec.

7. word2vec: It is an algorithmic technique used in NLP introduced by the seminal work Mikolov *et al.* (2013). The system can utilize one of 2 model architectures, either the skip-gram or the continuous bag-of-words model (CBOW), to generate a semantic vocabulary for each word encountered in the training text. Both models predict the current word from a window of surrounding context words, but the key difference is that the CBOW model is indifferent to the orders of the context window words, while the skip-gram models weighs nearby context words more heavily than more distant context words.

8. Csmith (Yang *et al.*, 2011) : Csmith is a tool that can generate random C programs that statically and dynamically conform to the C99 standard.

9. TreeBasedCNN Dataset (Mou *et al.*, 2015) : The following dataset houses programs divided according to different types. Hence it is useful for attempting classification.

A.2 Clustering based Technologies

1. k-means clustering: An unsupervised machine learning algorithm which given a set of data points in n-dimensional points finds k-centroids and k-clusters such that similar data points are grouped together into the same cluster.

2. t-SNE: t-distributed stochastic neighbor embedding (t-SNE) is a statistical method for visualizing high-dimensional data by giving each datapoint a location in a two or three-dimensional map while trying to maintain the distance relationships between points.

3. DTW: In time series analysis, dynamic time warping (DTW) is one of the algorithms used for measuring similarity between two temporal sequences, which may vary in speed.

A.3 Machine Learning Technologies

1. Artificial Neural Network: Is a weighted network of nodes or neurons which aims to find underlying structure in data we are trying to fit and eventually predict.

2. Recurrent Neural Network (RNN): A class of ANNs that feeds the input of previous node as well as conventional input to the current node. This helps the network in being better at understanding time based relationships in data.

3. LSTM: A special class of RNN's, LSTM has feedback connections. It can not only process single data points (such as images), but also entire sequences of data, and solves the dissapearing gradient problem that is sometimes observed in RNNs, and hence can retain more 'memory' across a time-based sequence. It is popularly used in handwriting recognition speech recognition and anomaly detection in network traffic.

APPENDIX B

Miscellaneous Experiments

B.1 Visualing DTW cluster centres

As described in subsection **3.1.3**, we used DTW to cluster the blocks and obtain labels for each block. As DTW is able to look past temporal shifts in time series, the cluster centroids obtained by DTW when represented as a sequences should have different forms if the number of clusters used is adequate. However as each point of the sequences would be 3-dimensional, the plot would have been in 4-dimensions. Hence to visualize this data, we used t-SNE to reduce the dimensionality of the data. The following are the results obtained for different values of k.

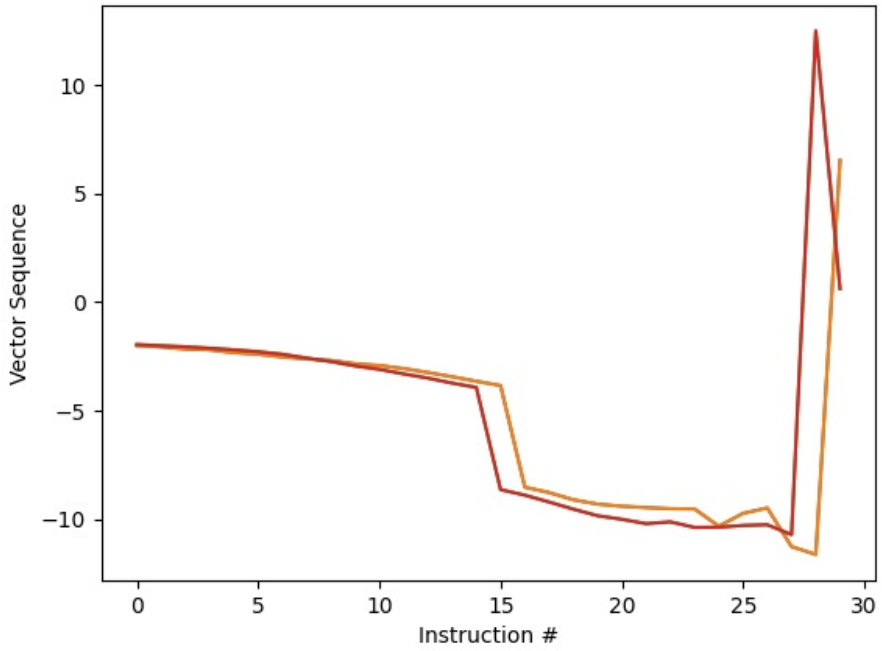

Figure B.1: Cluster Centroids for k=2

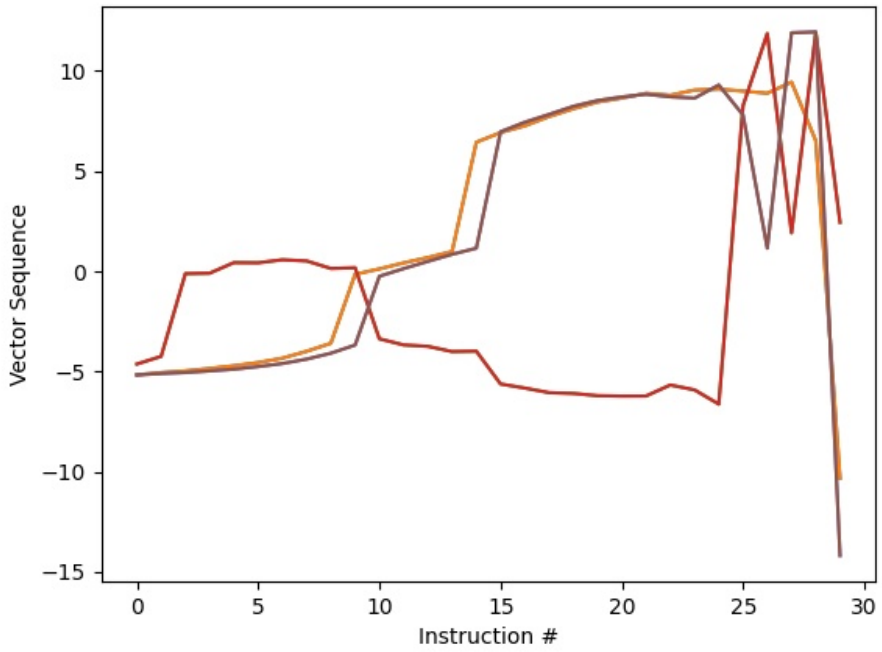

Figure B.2: Cluster Centroids for k=3

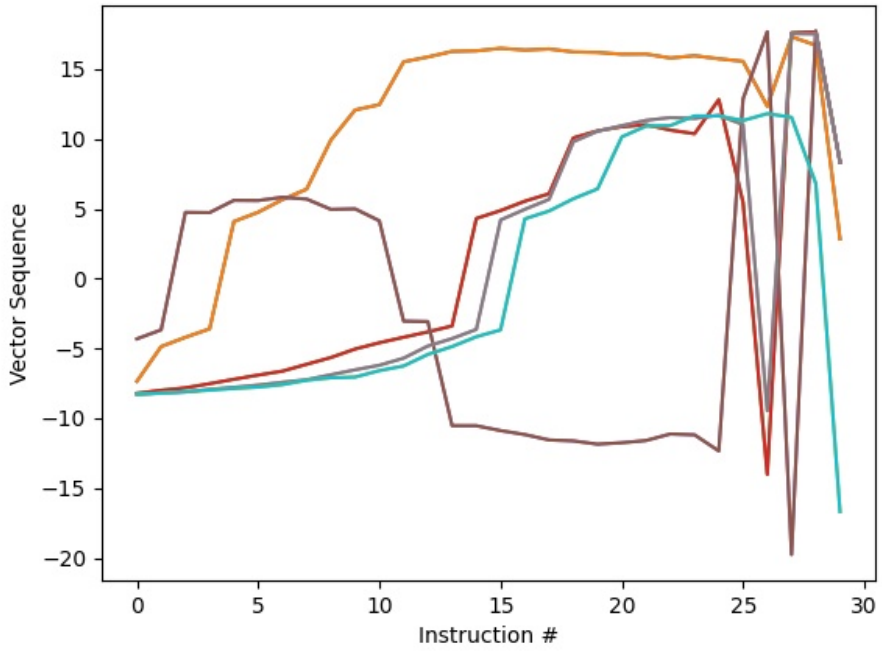

Figure B.3: Cluster Centroids for k=5

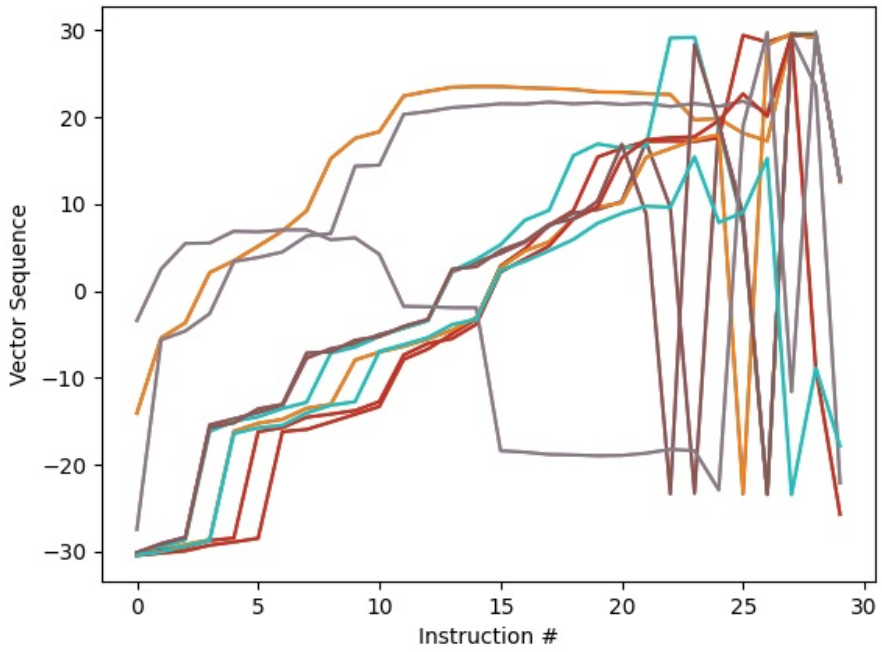

Figure B.4: Cluster Centroids for k=10

There does seem to be a number of centroids that are shifted versions of others until variations in the tail section, especially for higher values of k. Visually from the same, it does seem that beyond k=5, we are perhaps constructing several clusters for semantically similar blocks.

B.2 Block Histogram of benign vs malicious code

To see if we can visually observe any difference between benign and malicious code, we decided to take 2 programs, one a sample program from label 78 of the TreeBasedCNN Database, and the receiver program of a side channel attack (P+P_L1_Covert_Channel_slow), extract all the distinct blocks that we see from their instruction mnemonics and one hot encode the same. Then we represent each program as a histogram on this encoding and the results are given below.

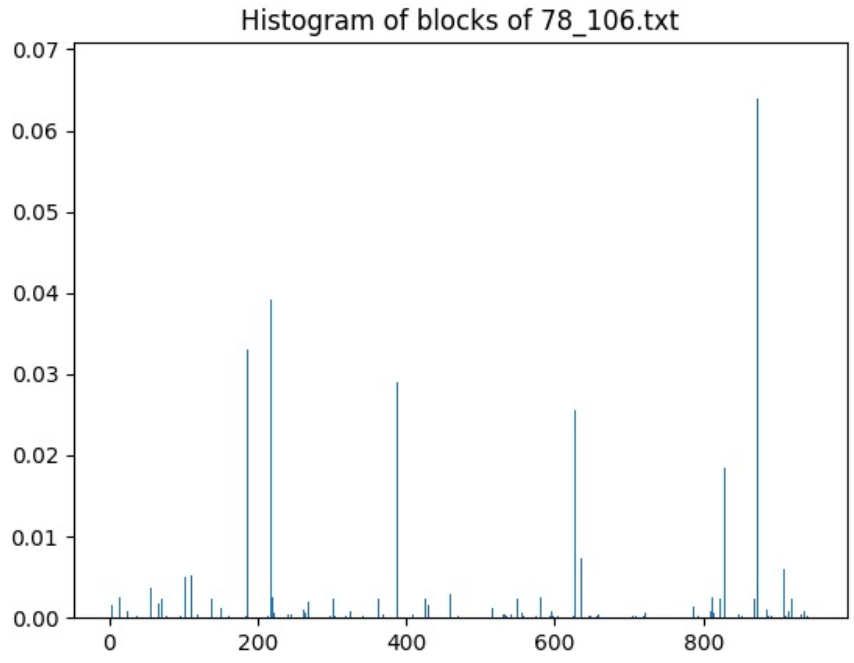

Figure B.5: Histogram of blocks (Benign Code)

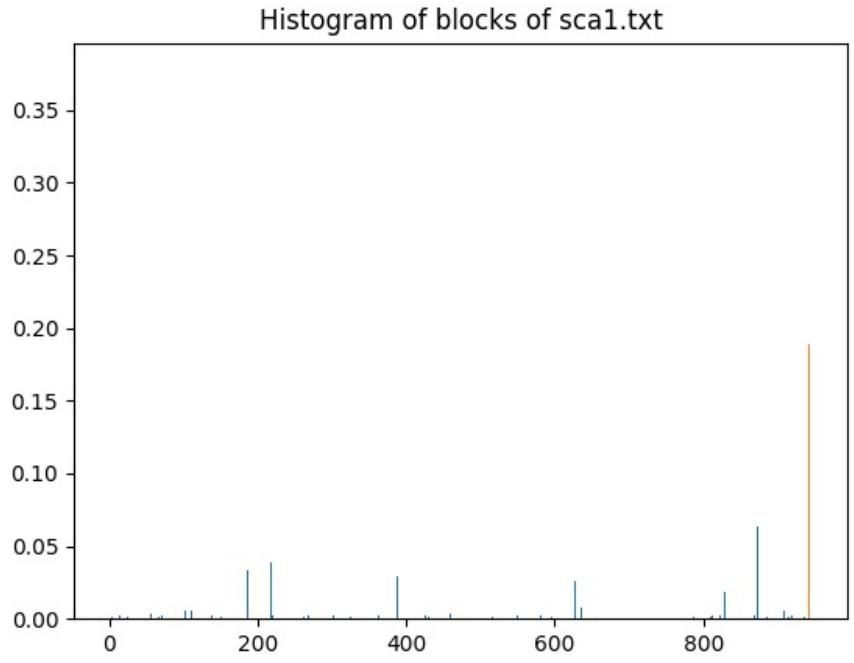

Figure B.6: Histogram of blocks (Malicious Code)

A significant fraction of the blocks in the side channel attack have size > 30 instructions (denoted by the final orange bar in the plot), while it is seen than in benign programs, the frequency of such blocks are usually minuscule. This is a key difference

we can visually observe apart from the difference in structure of the 2 histograms.

REFERENCES

1. **Alon, U.**, **M. Zilberstein**, **O. Levy**, and **E. Yahav** (2018). code2vec: Learning distributed representations of code.

2. **Demme, J.**, **M. Maycock**, **J. Schmitz**, **A. Tang**, **A. Waksman**, **S. Sethumadhavan**, and **S. Stolfo**, On the feasibility of online malware detection with performance counters. volume 41. 2013.

3. **Ding, S. H. H.**, **B. C. M. Fung**, and **P. Charland**, Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. *In 2019 IEEE Symposium on Security and Privacy (SP)*. 2019.

4. **Karampatsis, R.-M.**, **H. Babii**, **R. Robbes**, **C. Sutton**, and **A. Janes** (2020). Big code != big vocabulary: Open-vocabulary models for source code.

5. **Luk, C.-K.**, **R. Cohn**, **R. Muth**, **H. Patil**, **A. Klauser**, **G. Lowney**, **S. Wallace**, **V. J. Reddi**, and **K. Hazelwood** (2005). Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, **40**(6), 190–200. ISSN 0362-1340. URL `https://doi.org/10.1145/1064978.1065034`.

6. **Mikolov, T.**, **K. Chen**, **G. Corrado**, and **J. Dean** (2013). Efficient estimation of word representations in vector space.

7. **Mou, L.**, **G. Li**, **L. Zhang**, **T. Wang**, and **Z. Jin** (2015). Convolutional neural networks over tree structures for programming language processing.

8. **Yang, X.**, **Y. Chen**, **E. Eide**, and **J. Regehr**, Finding and understanding bugs in c compilers. *In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11. Association for Computing Machinery, New York, NY, USA, 2011. ISBN 9781450306638. URL `https://doi.org/10.1145/1993498.1993532`.