**"Efficient Image Generation for Android Malware Classification: Enhancing Accuracy and Efficiency in Mobile Security"**

**"Efficient Image Generation for Android Malware Classification using Space Filling Curves"**

**Abstract**

Keywords:

Problem Definition

Android application binary files are not two-dimensional or three-dimensional images and by converting them as such will introduce superfluous predictions. Representing a malware binary file as image can distort the original byte sequence structure in application binaries if not carefully done. Converting android files into images introduces spatial correlations amongst pixels in different rows which might not be true. Therefore, the methods used in converting Android binaries into images is largely ineffective. This can lead to the loss of information necessary to effectively characterise an Android app and since ML/DL models rely on this information to carry out predictions, this may affect the detection accuracy of these models.

1. Introduction

Android has an open ecosystem that allows developers to easily create and distribute apps. This openness also makes it easier for malware developers to produce and spread malicious apps. Moreso, Android's permission model relies on users to approve requested permissions. Many users do not fully understand the permission implications, which enables malware to acquire unnecessary permissions. Furthermore, the availability of many third-party app stores with less rigorous security checks increases the distribution channels for malware-infected apps. Also, The large volume of apps on the official app store, Google Play, makes it difficult for Google to thoroughly vet each app for malware. Many malware-infected apps manage to sneak through. Again, the wide variety of Android devices and OS versions leads to fragmentation that makes it difficult to patch all devices quickly. This leaves openings for malware to exploit. According to a recent Karpesrky Security Network report[1], about 4.9 million attacks in Q1 2023 malware samples were detected in the first quarter (Q1) of 2023. The report provides a good snapshot of the current threats faced my mobile phone users in general and Android users in particular.

With the increasing prevalence of Android malware, effective detection and classification techniques are crucial for ensuring the security and privacy of mobile devices. Machine learning (ML) and deep learning (DL) models have shown promising results in identifying Android malware based on various features derived from application binaries.[2][3][4][5][6]. The main steps involved in Android malware classification using machine learning and deep learning include [kaijun Liu 2020] : data collection, data preprocessing, feature selection, model selection and training, model evaluation. Several features have been extracted from android application file, also known as Android application PacKage (APK) and used to characterise the behaviour of malware. Manually programming rules to catch malware is slow and will never keep up with ever-changing malware. Moreover, manually extracted features can be biased towards hopes and thoughts about the feature space[Gamut]. This is why the idea of representing malware binaries as an image was groundbreaking. This approach comes with several benefits. Accurate conversion to images will allow us to use the various deep-learning based CNNs in image classification. Visualizing the malware will let us spot patterns(like in the field of medicine). And all this can be done without having to run the malware. Several works on Android malware classification have been proposed [1][2][3][4][5]][6][7] that make use of image representations of Android binaries. However, the representation of APK binaries using images can introduce distortions that compromise the integrity of the underlying byte sequence structure. This loss of information has the potential to significantly impact the accuracy of ML and DL models, as they heavily rely on this information for making accurate predictions.

Also, the use of fixed-size images is a common approach for representing application binaries in the context of ML and DL models. However, most convolutional neural network (CNN) models, which are widely employed for image-based classification tasks, are designed to work with specific input sizes. Generating images to fit these fixed sizes can lead to distortions and information loss, as the byte sequence structure is not preserved. Consequently, the detection accuracy of Android malware classification models may be negatively affected.

In this research paper, we address the issue of the ineffective conversion of Android binary files into images. while preserving data locality and maintaining the necessary information for effective Android malware classification. Our objective is to develop an approach that can preserve data locality by ensuring that data items that are close to each other in binary files will also be close to each other in the generated image. strategies that mitigate the loss of information caused by image generation techniques, thus enhancing the accuracy of ML and DL models in detecting Android malware. We explore the use of space filling curves in representing Android application binaries as 2 dimensional images. The research makes the following contributions:

1. A novel approach is proposed for representing Android binary files using space filling curves. This approach preserves the locality property of the byte sequences while generating images for Android malware classification.

2. A Convolutional Neural Network (CNN) model is developed and trained using the images generated from both the Hilbert and z-order space filling curves, as well as images generated using the traditional approach.

3. The impact of these images on the accuracy and efficiency of the CNN model is evaluated by comparing the results obtained from space filling images to those obtained from traditional images.

4. The practical implications of the findings for mobile security applications are discussed, emphasizing the importance of carefully considering app-to-image conversion techniques in the context of Android malware classification.

The remainder of this paper is organized as follows: Section 2 provides an overview of related work in Android malware detection and image-based classification techniques. Section 3 describes the methodology and techniques employed in our research. Section 4 presents the experimental setup, including dataset description and evaluation metrics. Section 5 discusses the results and analysis of our experiments. Section 6 provides a detailed discussion of the findings, limitations, and potential future directions. Finally, Section 7 concludes the paper, summarizing the contributions and outlining the significance of our research in the field of Android malware classification.

2. Related work

2.1 Android malware Detection

The papers suggest that various techniques have been used for malware detection using Android DEX file features. Park 2016 proposed a static analysis technique that features code reuse of repackaged malwares using class names and method names extracted from DEX files. Fang 2020 proposed an Android malware familial classification method based on DEX file section features, where the divided the entire DEX file into 8 sections and used this for classification. VarshaM. 2016 proposed a statistical approach for smartphone malware detection that extracts features such as hardware, permission, application components, filtered intents, opcodes, and strings from samples to form a vector space model. Khan 2022 proposed a machine learning-based algorithm to detect Android malware apps through feature extraction and classification of grayscale images, which converts most of the files of APK into a grayscale image and uses different ML models to classify the local features.

2.3 Image-based Malware detection

Recall history of image-based methods [2011\_Nataraj], Why image-based malware detection? Problems with image-based malware detection. Possible illustration.

Image-based malware detection generally involves several steps which comprise: *application components extraction*, *image generation*, *feature extraction* and *classification*. In the *image generation* step, the application component(s) extracted in the previous step is firstly converted to a binary stream. Then, the binary stream is then transformed into a matrix, which can be visualise as an image. Features are then extracted from this image and used for malware detection. Malware image has the advantage that features can be easily extracted without feature engineering and domain expert knowledge[2022\_marwan\_ali, 2020\_singh] and that they can be applied to different platforms, such as Windows and Android [2022\_Mitsuhashi]. Also, conversion to images will allow us to use the various deep-learning based CNNs in image classification[2020\_Bensouad]. Visualising the malware will let us spot patterns(like in the field of medicine). And all this can be done without having to run the malware. This is a big plus for security.

Work on malware visualisation dates back to 2008, where Conti et al. [2008\_Conti], presented design principles for analysing files whose formats are unknown, implemented a visual reverse engineering tool based on the design principles above and validated the efficacy of both the analysis and implementation of the system using some use cases. In 2011, Nataraj et al. [2011\_Nataraj] visualised malware binaries as gray-scale images and proposed a classifacation method using standard image features. They above pioneer works were not based on the Android platform.

Most recently, there have been several works on image-based Android malware detection. For example, in [2023\_Raissa], authors generated Android application images from DEX files using n-grams and simhash and used SVD (Singular Value Decomposition) to extract features for malware detection. They used the entire DEX file for image generation which can be very noisy. In [2022\_Xiaofei Xing] authors converted bytecodes of app methods into gray-scale images, used autoencoder based on CNN to recognise features in gray-scale images and proposed a neural network-based on the autoencoder for classification. Only files with a small data size were chosen for image generation in order to facilitate or avoid the problem of resizing very large files. Given that most malware files have very large sizes, it implies most malware will be discarded in the image generation phase and will not be classified. In [2022\_Zahraddeen], CNN-based architecture that uses transfer learning pre-trained on ResNet to distinguish malware from benign apps was proposed. However, details of how apps were converted into images was not given. In [2022\_Xingyu Li], authors proposed an RGB visualisation technology that directly transforms binary files into unstructured RGB images using DEX file, Certificate file and Resource file. They used pre-trained VGG16 to acquire advanced features by training with RGB images. However, dividing the bit string into three may cause parts of the original files to be split amongst the different parts . Also, transforming code segments into rectangular matrices can lead to distortions in the original byte sequence structure. Generated image file can be very large as it contains noisy data that may not be necessary for malware behaviour characterisation. In [2022\_Mohd Abdul Rahim Khan], a gray-scale image is generated from the DEX files, Certificate files, Manifest files and Resource files by reading bytes from the binary version of these files and converting them into pixels(1 byte = 1 pixel). Then, image features are extracted and used for classification. Using all APK components to generate images can lead to very large and bogus images which may require a lot of computer resources for processing. Also, their method of image generation can lead to distortions in the original byte sequence structure. [2022\_marwan\_ali] generated fifteen different gray-scale images from the combination of various application components (classes.dex, certificate, androidmanifest.xml, resources) . They extracted automatically features from these images which we used for malware family classification using a modified ResNext model. However in this work, authors did not take into consideration the spatial structure of the files while converting them into images. More so, the use of entire file sections add a lot of noisy data to the final image generated.

3. Background

3.1 Android DEX file structure

An Android application is composed of several files and folders all grouped together in an archive file called .apk (Android application Package Kit). This file is a zip-format archive used to distribute and install Android applications. The contents of this file include: manifest file, librairies, resource files, dex files and certificate files. The name of the execuble file is classes.dex which is composed of the code that will be executed by the virtual machine. Figure 1 shows the structure of the DEX file.

classes.dex

header

ids

data

magic

checksum

.

.

data\_size

data\_offset

string\_data\_item

class\_data\_item

code\_Item

debug\_item

…..

maplist

*Header section:* The DEX file starts with a fixed-size header that contains information about the file, such as the file size, the number of items in various sections, and the offset to the data sections. The data\_size and data\_offset provide information that can be used to access the data section of a DEX file.

*ids section*: This section lists all identifiers such as strings, types, fields and methods.

*Data section*: The data section store the actual data used by the classes, methods, and fields. These sections include the code section (containing the bytecode instructions), the debug section (containing debugging information), and other sections for annotations, class references, string data section and more. This information found in this section defines the behaviour of an application[2018\_jaemin jung]. The *string data section* stores the actual string values used in the DEX file. It includes all the string literals, method names, field names, class names, and other constant string values. The *code section* contains the bytecode instructions that make up the methods' executable code. Each method's bytecode is stored in this section, and it is executed by the virtual machine. The c*lass references section* contains references to classes used in the DEX file but not defined within it. It includes external classes referenced by the DEX file, such as classes from the Android framework or other external libraries. These references enable the DEX file to interact with classes and resources outside its own scope. The *debug section* stores debugging information for the classes and methods defined in the DEX file. This information includes line numbers, variable names, and other metadata that aids in debugging and troubleshooting. It allows developers to map the bytecode instructions to the original source code during debugging sessions. The *annotations section* holds metadata about classes, methods, fields, and their associated annotations. Annotations provide additional information and attributes to the elements they annotate. This section includes details such as the type of annotation and the values associated with it.

In general, malicious code often recites in an executable module. Therefore, in this paper, we focus on the data section which contains class data and executable code. Also, using only the data section reduces the amount of noisy and unnecessary data used for malware classification.

**3.2 The Hilbert’s space-filling curve**

A space-filling curve is a continuous function that, roughly speaking, maps a one-dimensional space onto a higher dimensional space [wattenberg]. Space filling curves (SFC’s) are mathematical constructs, also known as continuous fractal curves, the limit of which contain the entire 2-dimensional unit square [psace-filli]. This curve passes through every point of a geometric space. In other words, it fills the entire space without any gaps. Peano introduced the first space filling curve in 1890. Hilbert later simplified Peano's curve in 1891. Other variations have been developed, including Moore's curve, Sierpinski's curve, the Gray-code curve and the Z-order or Morton curve. These curves are important among other reasons because they preserve spatial locality, meaning that points close together on the curve are also close together in the n-dimensional space. They have applications in areas like data compression, image processing, computer graphics, spatial indexing and geometry.

Amongst the space filling curves, the Hilbert’s curve is quite notable because it has been used in a variety of applications.The original construction of the Hilbert curves, with their recursive geometric nature, naturally establishes an order for points in finite square lattices. Specifically, the Hilbert curve has been shown to be the most effective space-filling curve for preserving data locality when used in this manner[2006\_hilbert]. As a result, it has received significant research attention, resulting in the development of various algorithms, each tailored to specific applications[4, 6, 7, 8, 12, 17, 22, 26].

In this paper, we focus on the Hilbert’s curve with 2 dimensions though, theoretically, it generalises to n-dimensions. We consider the Hilbert curve that starts in the upper left corner and finishes in the bottom left. The *order* represents the number of iterations used to generate a curve. Algorithm 1 represents a recursive procedure of generating the Hilbert’s curve of a given order and of dimension 2.

*if order = 1, curve defined on a 2 x 2 lattices as shown in Fig 1.*

*if order = k, where k = n + 1, we construct the curve on a 2n+1 x 2n+1 lattice as follows*

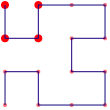
*step 1 : Place a copy of the order=n curve defined on a 2n x 2n, rotated clockwise by 900, in the upper left sub-grid*

*step 2 : Place a copy of the order=n curve defined on a 2n x 2n, rotated counter-clockwise by 900, in the lower left sub-grid*

*step 3 : Place a copy of the order=n curve defined on a 2n x 2n, in each of the right sub-grids*

*step 4: Connect these four disjoint curves in the obvious manner.*

This construction may be visualized in Figure 2 which shows how the second order Hilbert’s curve can be constructed from the first. Figure 3 shows the first nine iterations of the Hilbert’s curve.



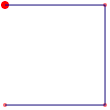
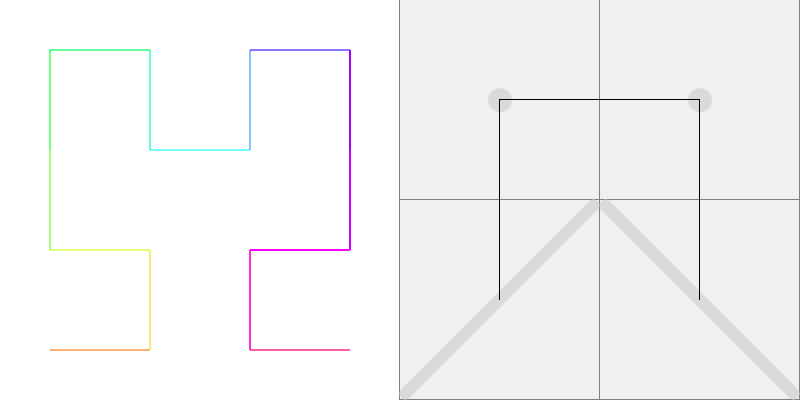
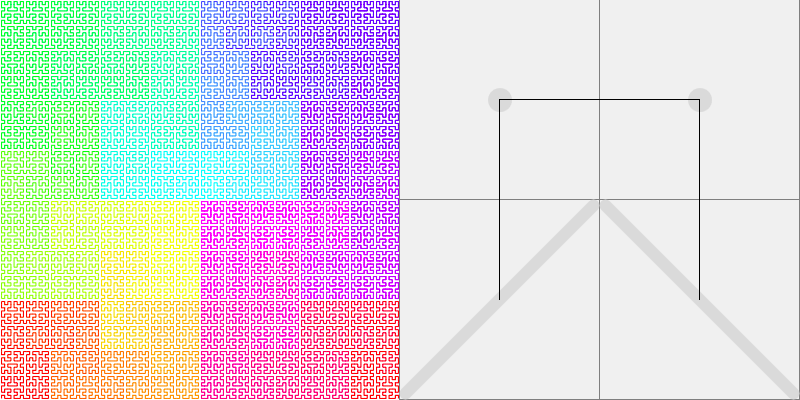


Figure 1: Order 1 Hilbert’s curve Figure 2: Order 2 Hilbert’s curve

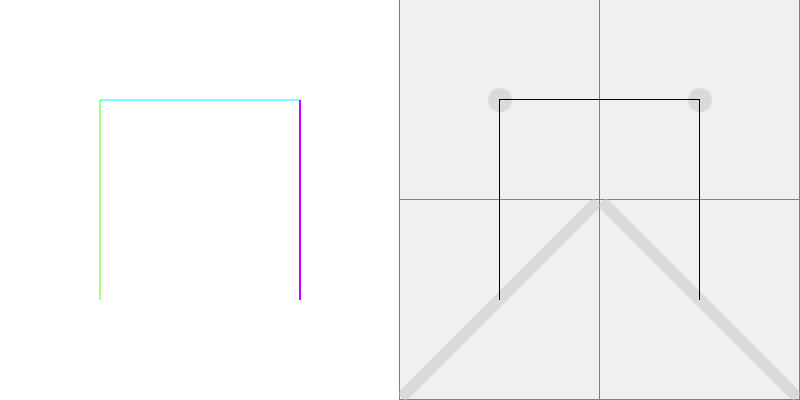
Order 2



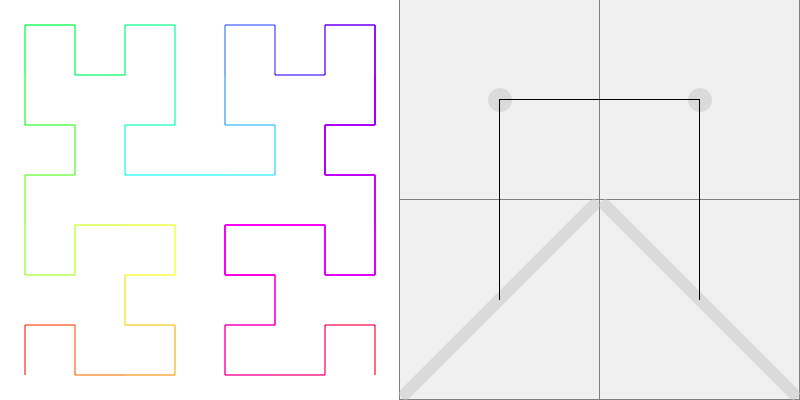
Order 7



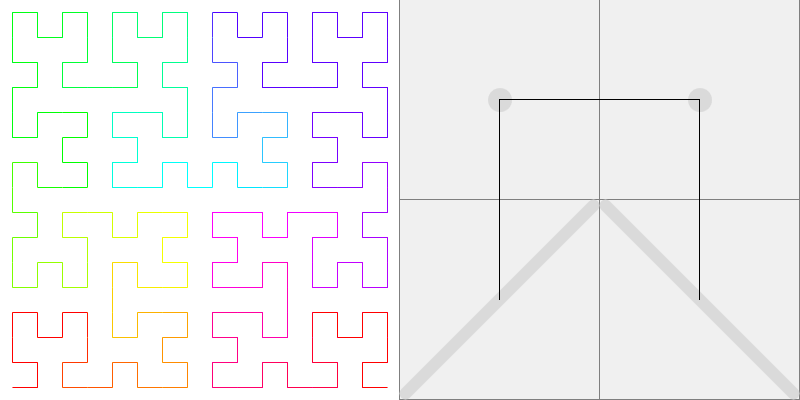
Order 1



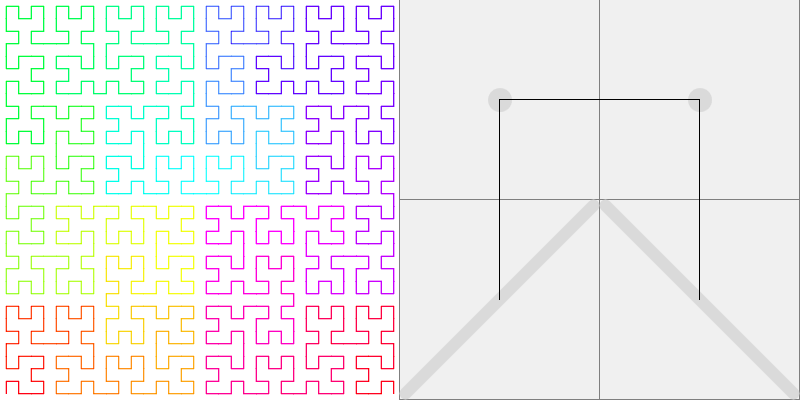
Order 3



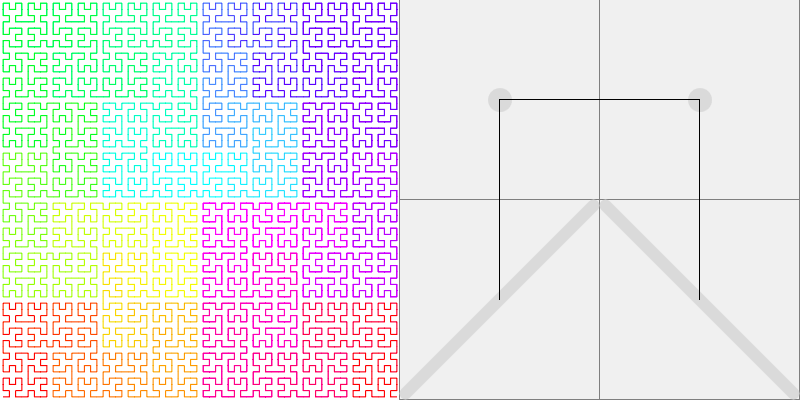
Order 4



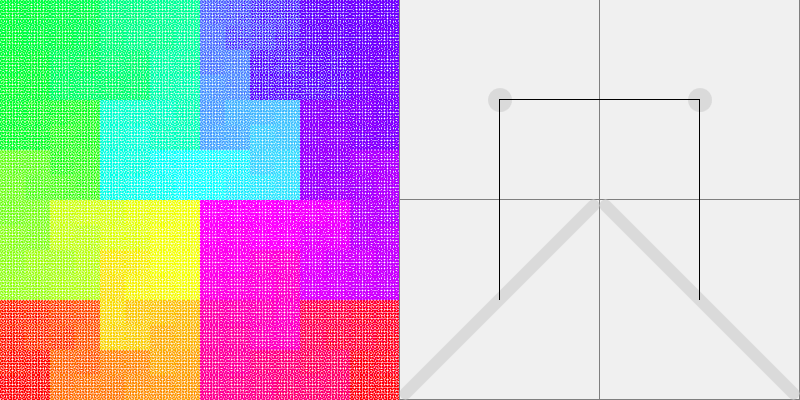
Order 5



Order 6



Order 8



Order 9

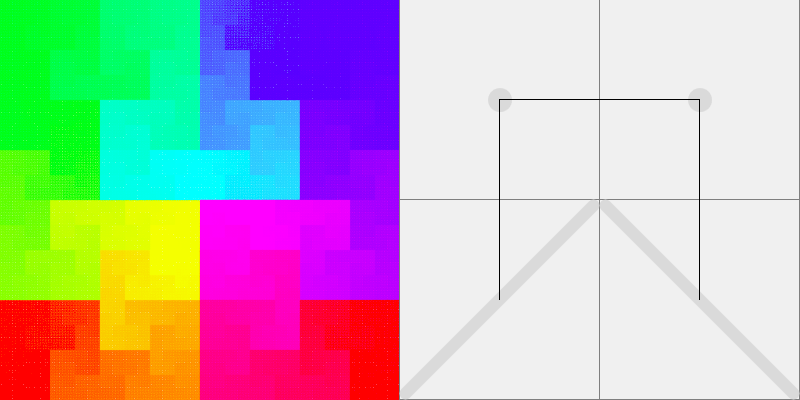


Figure 3: Order 1 to 9 of the Hilbert’s curve

As we proceed to higher orders of the Hilbert’s curve, points in the curve get closer and closer until the plane is completely covered with the curve passing through all the points on the plane. It is worth noting the fact that the Hilbert curves always take steps of unit length: immediate neighbours on the curve are also immediate neighbours in the plane. This translates to a notion of **data locality**: *points close to each other in the plane tend to be close to each other in their associated Hilbert order*.

**3.3 Convolutional Neural Networks (CNNs)**

A Convolutional Neural Network (CNN) is a type of artificial neural network that is particularly effective in analysing visual data, such as images or videos. It is designed to automatically extract features and patterns directly from input data instead of hand engineering [][][][]. The key characteristic of a CNN is its ability to learn and exploit spatial hierarchies of features. This is achieved through the use of convolutional layers, which apply a set of learnable filters to the input data. Each filter performs a convolution operation by sliding across the input, computing dot products at each location. This process effectively captures local patterns and detects features such as edges, corners, or textures. This convolution operation preserves the spatial structure of the input image while also extracting features automatically. The architecture of a Convolutional Neural Network (CNN) typically consists of several types of layers arranged in a sequential manner as shown in Figure 4.

Input image

Convolutional layer

Pooling layer

Fully-Connected layer

Output layer

Figure 4: Structure of a Convolutional Neural Network

The architecture of a Convolutional Neural Network (CNN) typically consists of several types of layers arranged in a sequential manner. The main components of a CNN architecture are:

1. Convolutional Layers: These layers perform the core operation of convolution. Each convolutional layer consists of multiple filters (also known as kernels), which slide over the input data and perform element-wise multiplication and summation. The output of this operation is a feature map that represents the presence of certain features or patterns in the input data.

2. Activation Functions: Activation functions are applied element-wise to the output of each convolutional layer. They introduce non-linearities into the network, allowing it to learn complex relationships between features. Common activation functions used in CNNs include Rectified Linear Unit (ReLU), sigmoid, and hyperbolic tangent (tanh).

3. Pooling Layers: Pooling layers reduce the spatial dimensions (width and height) of each feature map while retaining important features. The most commonly used pooling operation is max pooling, which selects the maximum value within a defined pool size. Pooling helps reduce the computational complexity of the network and provides a form of translation invariance.

4. Fully Connected Layers: After several convolutional and pooling layers, the output is typically flattened into a vector and passed through one or more fully connected layers. These layers connect every neuron to every neuron in the subsequent layer, performing high-level reasoning and decision-making based on the extracted features. The final fully connected layer usually produces the network's output, such as class probabilities in the case of image classification.

5. Softmax Layer: In classification tasks, a softmax layer is often added at the end of the network. It converts the output of the last fully connected layer into a probability distribution over the different classes. This allows the network to provide class probabilities for multi-class classification problems.

In addition to these main components, CNN architectures may include other layers and techniques to improve performance, such as dropout layers to prevent overfitting, batch normalization layers to normalize the inputs, and skip connections to facilitate gradient flow. The specific architecture and the number of layers can vary depending on the task and the complexity of the data being processed.

4. Methodology

The research presented in this paper provides a study of the efficacy

of space-filling curves as a means of representing malware as 2-dimensional images that can be used to classify malware. We explain our approach for image-based Android malware detection. Fig. X illustrates the main steps of our approach which consist of : DEX processing, Image Generation and Classification.

APK



**Step1: DEX Processing**

classes.dex



DEX parser

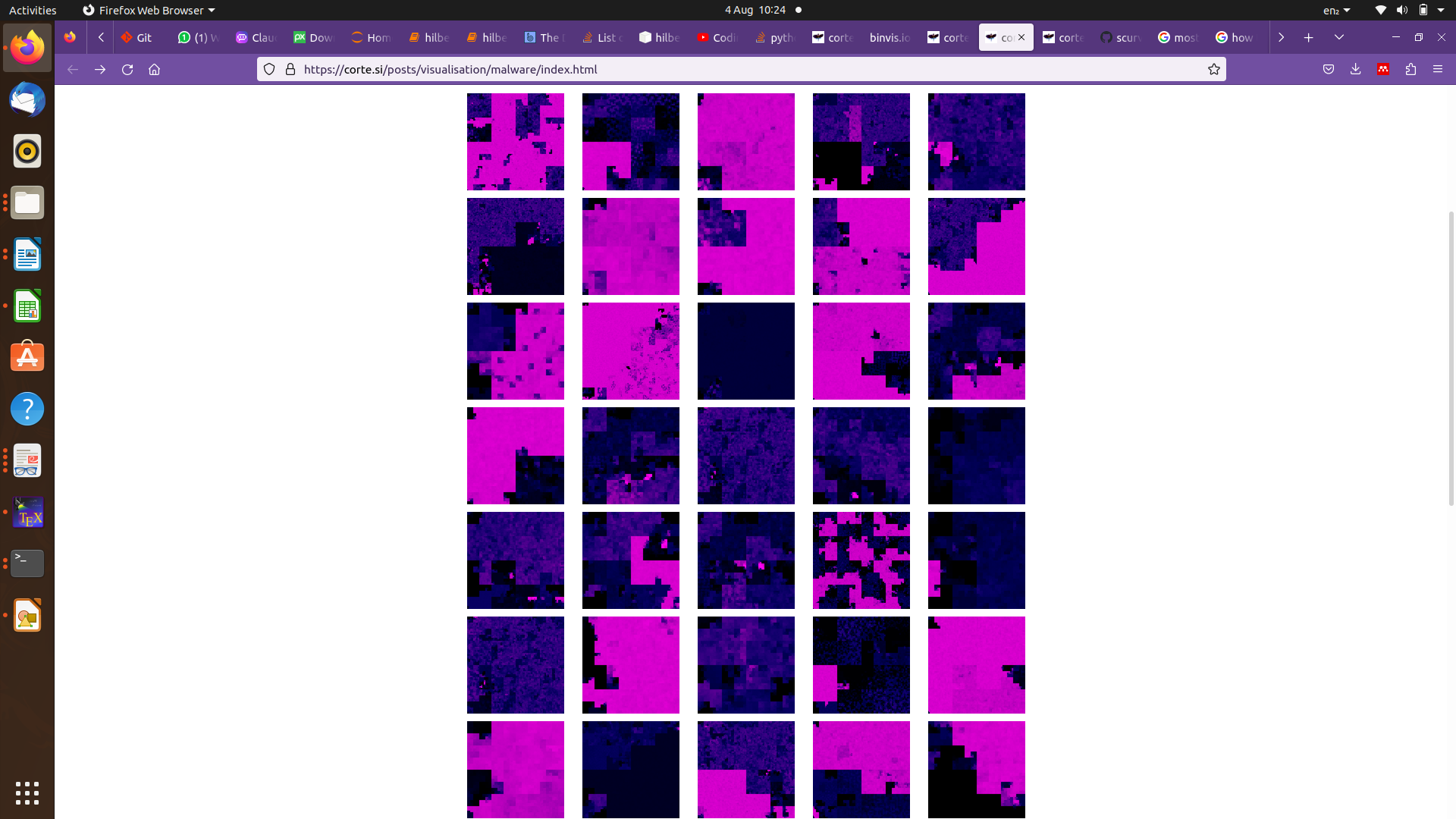
**Step2: Image**

**Generation**

data\_section



image

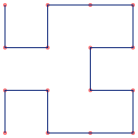


Natural

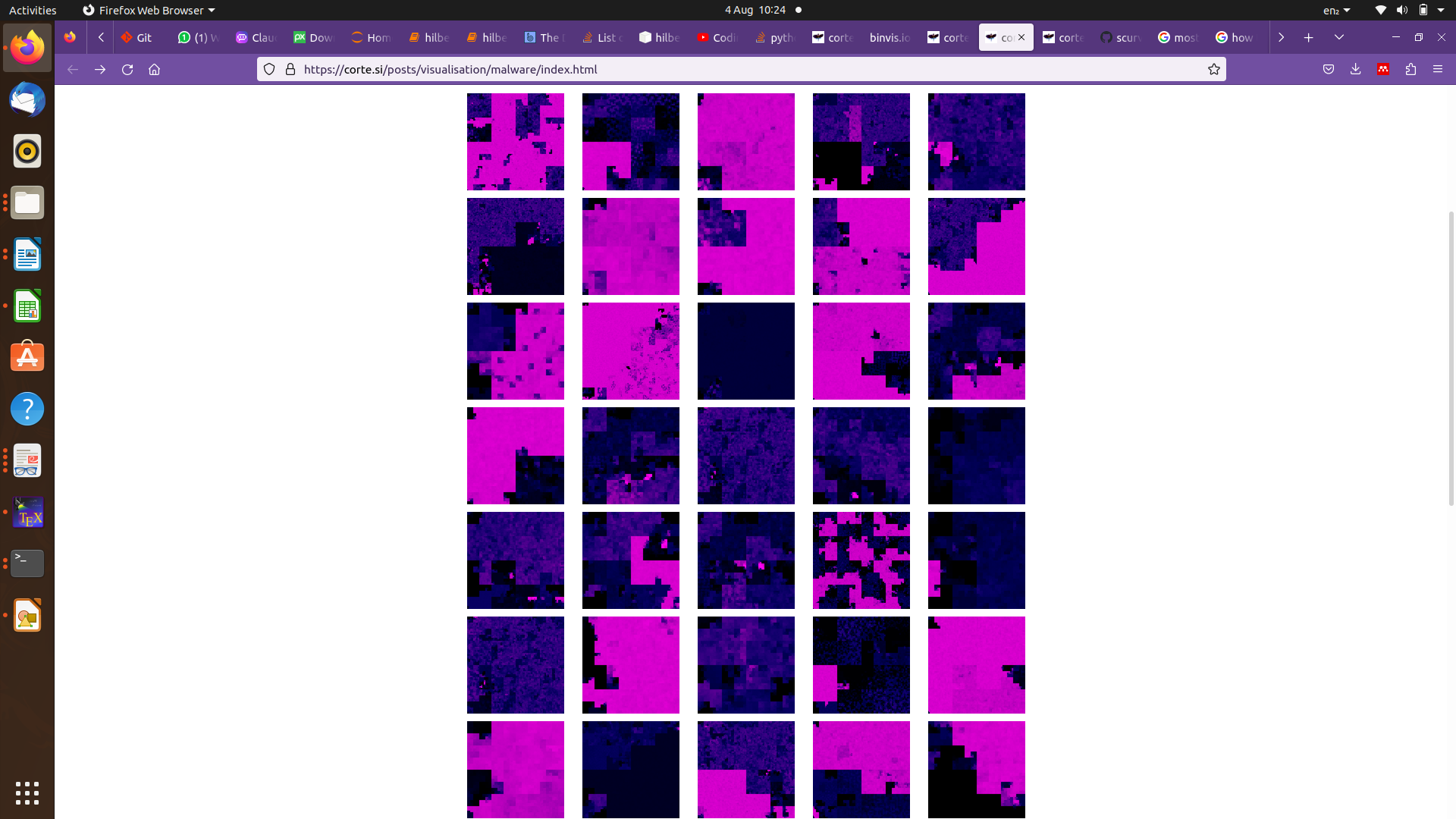
order

traversal

Hilbert traversal



image



**Step3:Classification**

benign

malicious

CNN

benign

malicious

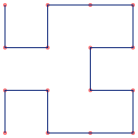
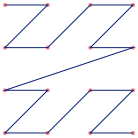
CNN

**4.1 DEX Processing**

The classes.dex file is extracted from the Android apk file (APKs with multiple DEX files are not used). Then, the header section of the DEX file is parsed in order to get the offset to the data section. This offset is then used to extract the data section from the classes.dex file as a data\_section binary file. The reason why we use only the data section of the classes.dex is that the section includes class data and execution code. Thus, the data section defines and represents the behaviour of apps, malicious or benign [2018\_Jaemin Jung]. Also, using only the data section reduces the amount of noisy and unnecessary data used for malware classification, hence reducing the overall size of the image data generated. This can also lead to a reduction in the amount of resources (memory size and computing power) required for model training and classification.

**4.2 Image Generation**

Our goal here was to convert the *data\_section* binary files extracted in the previous step into an image file such that data bytes that are close to each other in the binary file will be close to each other in the image file generated. This property is called "locality preservation" and can be achieved by a family of mathematical constructs called **space filling curves (SFC)**. To envision this for the purpose of malware binary-to-image mapping, the data section binary file was read as a 1 dimensional array of bytes with the code being parsed sequentially, byte by byte. The SFC maps each point (bytes) from the 1-dimensional space to the 2-dimensional space (SFC image map) such that closely located bytes in the binary file space will tend to also be closely located when mapped using the SFC. In order to achieve this, we used the scurve[] Python library to generate three image types: natural image, hilbert’s image, and Z-order image. The last two being SFC images. Scurve uses a 4 class colour scheme when mapping bytes to image: 0x00 for black, 0xFF for white, blue for all printable characters and everything else as red. The 0x00 and 0xFF correspond to the most common padding bytes. The file is sampled at regular intervals, each byte is translated to a colour, and the corresponding pixel is written to the image using each of the following SFC traversals which is illustrated in Fig. X.



Z-order Hilbert

Following the conversion process using the scurve library, three image datasets were generated from the using the following traversals of bytes: natural, z-order and hilbert.

Figure x shows the same variant sample of the xx malware, transformed into 2-dimensional tree-map images using the Natural, Hilbert, and Z-order formats. While similarities can be drawn in terms of the texture regions within the resulting images, it can be seen the layout patterns are distinctly different.

Hilbert\_point algorithm: Convert an index on the Hilbert curve of the specified dimension and

order to a set of point coordinates.

Hilbert\_point

input: dimension, order, i

output: point coordinates

N = Math.pow(2, order)

total = N \* N

points = [createVector(0, 0), createVector(0, 1), createVector(1, 1), createVector(1, 0)]

// Bitmasking

index = i & 3

v = points[index]

// Recursion masking

for (let j = 1; j < order; j++) {

let len = pow(2, j)

// Bit shifting

i = i >>> 2

index = i & 3

// Swaping positions

if (index === 0) {

let temp = v.x

v.x = v.y

v.y = temp

} else if (index === 1) {

v.y += len

} else if (index === 2) {

v.x += len

v.y += len

} else if (index === 3) {

let temp = len - 1 - v.x

v.x = len - 1 - v.y

v.y = temp

v.x += len

}

}

4.3 Classification

5. Experimental Setup

5.1 Dataset

In this experiment, we obtained APK samples from the AndroZoo[15] dataset . AndroZoo is a growing collection of Android Applications collected from several sources, including the official Google Play app market [16]. As of the time of writing this paper, AndroZoo has about 23 million different APKs, each of which has been (or will soon be) analysed by tens of different AntiVirus products to know which applications are detected as malware. We used the *az python script* [18] to download benign and malicious APK samples randomly from the AndroZoo dataset using the following

shell commands:

**Malicious samples command**

*az -n 12000 -d 2015-12-11: -s :1000000 -vt 20: -m VirusShare -md sha256,pkg\_name,apk\_size,dex\_size,dex\_date,vt\_detection,markets -t 40*

This command means: download 12000 apks with the dexdate starting from the 2015-12-11 (inclusive), size up to 1000000 bytes (inclusive), Virus total rating of at least 20, APK present in VirusShare , create a metadata.csv file with the fields: sha256, APK package name, APK size, size of classes.dex file, date attached to the dex file inside the APK zip, the number of AV from VirusTotal (VT) that detected this APK as a malware, list of the markets where the APK was seen and number of threads for concurrent download(40).

It is also important to note that the *az python script* requires two important parameters: an APK key from AndroZoo and an input csv file [19].

Malicious apps were selected by taking apps whose virus total rating is at least 20. The virus total rating gives the number of anti-virus engines in VirusTotal[18] that have rate the app as malicious. For example, a score of 20 shows that 20 anti-virus engines have considered the app as malicious. For the benign set, we selected apps whose virus total rating is 0, meaning none of the anti-virus engines in VirusTotal considers the app as malicious.

**Benign samples command**

*az -n 12000 -d 2015-12-11: -s :1000000 -vt 0:0 -m play.google.com -md sha256,pkg\_name,apk\_size,dex\_size,dex\_date,vt\_detection,markets -t 40*

The initial results from these commands gave 8700 malicious APK samples and 9100 benign samples. We then carried out a further filtering process by considering the size of the classes.dex file. Files whose DEX file size is greater than 100KB were eliminated from the datasets. This was done to avoid the use of large files whose generated images would become large and would require resizing and lots of computing resources for their processing.

Table x. shows the results of the filtering process.

|  |  |  |  |
| --- | --- | --- | --- |
| **Dataset** | **Results from az script** | **DEX size filtering** | **Final dataset used** |
| Benign | 9100 | 5602 | 5000 |
| Malicious | 8700 | 5130 | 5000 |

Table x shows the minimum, maximum and average size of the data section of the APKs. It also shows the ratio of the size of data section to that of the classes.dex file. As shown in the Table x., the size of the data section is about 82.5% of the whole DEX file in average.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Dataset** | **Size** | | | **Ratio** | | |
| **Min** | **Max** | **Avg** | **Min** | **Max** | **Avg** |
| Benign | 0.84KB | 99.3KB | 45.6KB | 74.3% | 98.8% | 82.45% |
| Malicious | 1.53KB | 98.5KB | 48.9KB | 54.1% | 96.4% | 82.53% |

5.2 Hardware and Software Configurations

5.3 Evaluation Metrics

5.4 Model Training and Testing

6. Results Analysis

7. Discussion

8. Conclusion

**References**

[1] https://securelist.com/it-threat-evolution-q1-2023-mobile-statistics/109893/

Open a terminal

conda create -n py27 python=2.7 anaconda

source activate py27

ipython kernel install

source deactivate