Bachelor thesis
Computing Science

Gamut: Sifting through Images to Detect Android Malware

Author:
Jordy Gennissen
S4372999
Jordy.Gennissen@rhul.ac.uk
J.Gennissen@student.ru.nl

Supervisor Royal Holloway:
Dr Lorenzo Cavallaro
Lorenzo.Cavallaro@rhul.ac.uk

Daily supervisor
Radboud University:
Dr Veelasha Moonsamy
Veelasha@cs.ru.nl

First Supervisor
Radboud University:
Dr Lejla Batina
Lejla@cs.ru.nl

June 25, 2017

Abstract

Mobile phones are omnipresent in present-day life. Everything can be controlled using this pocket-computer which typically runs on Android or iOS. However, the risk of phones breaches is greater than ever. In the first half of 2016, on average 9,468 Android malware samples are found every day [12]. Malware detection systems in the wild cannot keep up with this growing number.

In this work, I propose Gamut, a tool to generate different images out of an Android app without loss of information. The user can pick a level of semantic when generating the image. Datasets of images are created to train convolutional neural networks on malware detection. In this way, feature engineering is done by the neural network after adding domain knowledge. Hence, this ensures us an automatic feature space while achieving a high performance. Different levels of semantics have different results, peaking at 92% accuracy by combining domain-expert knowledge with the feature engineering by the neural network.

Acknowledgements

Special thanks to Dr Veelasha Moonsamy, Dr Lorenzo Cavallaro for the guidance in doing research and Royal Holloway, University of London for hosting me. Furthermore, I want to thank Dr Guillermo Suárez-Tangil, Dr Jorge Blasco, Dr Santanu Dash and Roberto Jordaney, MSc from the S^2Lab for all the support and help during my research. Lastly, I want to thank McAfee Labs for the dataset.

Contents

1 Introduction **3**

2 Preliminaries **5**
2.1 Android App Structure 5
2.2 DEX Structure 6
2.2.1 The Data Section 6
2.2.2 Code Section and APIs 7
2.3 From 1D to 2D Data 7

3 Related Work **8**
3.1 Malware Visualisation 8
3.2 Feature Selection for Malware Classification 9
3.3 Android Malware Classification and Detection 10

4 Dataset Representation **11**
4.1 Overview of the Data 11
4.2 Gamut 12
4.3 2D Representation 12
4.4 Semantics 12
4.4.1 Mode 0: Greyscale 13
4.4.2 Mode 1: Section Colouring 13
4.4.3 Mode 2: Class Colouring 14
4.4.4 Mode 3: List Method Colouring 15
4.4.5 Mode 4: Labelled Colouring 16
4.5 Image Size 18
4.5.1 Gradients 18
4.6 Datasets generated 19

5 Evaluation **20**
5.1 The Neural Networks 20
5.2 The Framework 21
5.3 Results 21

6 Discussion and Future Work **23**
6.1 The Images 23
6.2 Neural Networks 24

7 Conclusion **25**

Chapter 1

Introduction

Detecting malicious Android apps is a growing problem in the field of computer security. The amount of new malicious Android apps discovered every day is growing significantly. G Data shows an increase in new detections of more than 29% [12]. It has reached the point where manual research on malware does not suffice anymore. Traditional malware detection techniques used are outdated as they rely on static fingerprints of known malware samples, hence not capable of detecting new malware. There is a lot of research on using machine learning for detection and classification with very good results. However, these rely on manual feature engineering, a problem of which the answer might shift over time. Secondly, the features might be biased towards hopes and thoughts about the feature space.

A clear shift has gone from high accuracy to the field of explainability and trusting the way a method works [7, 27]. If the way a malicious sample is recognised sounds plausible, it is more easily accepted as a malicious sample by us humans. However, it remains the question whether this feature space is reliable and will remain reliable for detecting malware. In other words, we do not know for sure whether the feature space is and will remain both correct and complete. In this thesis, I take a closer look at this feature space. We answer the question if we can detect malicious apps by converting the app into an image and using image classification techniques on these images. Secondly, we investigate to what extent the classifier will improve by adding domain knowledge (semantics) to the image.
We do this in two ways: at first we add semantic to the images to enhance the feature space. In later modes, we add more detailed semantic on parts while removing some other parts that might not be useful (i.e. non-suspicious API calls). We found that removing semantics in any way reduces the accuracy of the neural network, despite adding more detailed information onto the suspicious API calls that are also used in various other papers as features. The image dataset without any semantics has an accuracy of 85%, whereas the accuracy tops at 92% when adding semantics.

This thesis presents Gamut[1], a tool to generate images out of Android apps. Gamut can add a specified level of semantics to facilitate explainability. Afterwards, datasets of Gamut images in different levels of semantics are used to train a convolutional neural network. This combines computer driven feature engineering with the traditional manual feature engineering. Not only provides this with a new way of looking at malware and malware detection, it provides a framework for measuring the importance of semantics in the use of Android malware detection.

[1] Gamut is originally a term used in science to describe a complete subset of colours. This relates to the tool, as this tool uses different subsets of colours to highlight information about the binary.

Chapter two will give some background information to my approach. In chapter three, I will discuss the previous research done. Chapter four will give an overview of the datasets and how these are generated. Chapter five will show the full setup for malware detection and discuss the results. Chapter six discusses some problems with this approach and chapter seven will conclude this thesis.

Chapter 2

Preliminaries

2.1 Android App Structure

Android apps are represented as an APK file, a zip file containing a series of files. The zip can contain a number of resources in different folder, however the following files are mandatory for an app to work properly. A detailed structure is also shown in Figure 1.

- *classes.dex*, a binary Dalvik EXecutable (DEX) containing information about the classes and the code. This file also contains all the string data. Large apps can have multiple DEX files, with the following DEX files being classesN.dex (e.g. classes2.dex, classes3.dex). Typically, Android apps do not have more than 2 DEX files.
- *AndroidManifest.xml*, a file containing metadata of the app, such as permissions, name of the app and libraries used. While this might seem an interesting file at first, bear in mind that for e.g. permissions can be requested without being actually used in the code; thus, adding redundant information to the proposed tool. For the sake of simplicity and the nature of the approach, I decided not to use the information contained in the file.
- *META-INF/* is a directory where certificates and the standard Java manifest (MANIFEST.MF) reside.
- *resources.arsc* contains compiled resources and the references to the resources outside this file, such as graphics. These are stored in the *res/* directory.

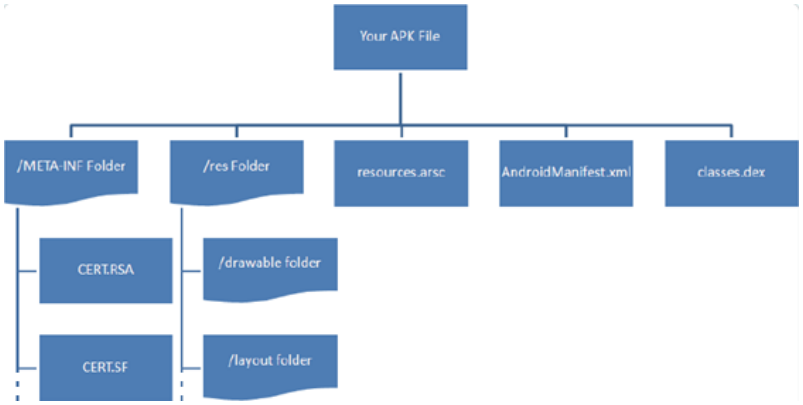


Figure 1: The APK structure [1]

[1] origin: http://2we26u4fam7n16rz3a44uhbe1bq2.wpengine.netdna-cdn.com/wp-content/uploads/013013_1319_AndroidAppl8.png

2.2 DEX Structure

A DEX file has a distinct structure. The first 112 (0x70) bytes consist of a header containing metadata of the DEX file, various IDs with pointers and a data section (see Figure 2 a). The header is shown in more detail in b), which starts with a DEX file magic number, to recognise the file structure as being a DEX file and to read the DEX version. Furthermore, the header contains a pointer to the maplist, I will discuss the maplist further in section 2.2.1.

Next, the header includes offsets (pointers) and sizes of every other section in the DEX file. Next, the DEX file has different IDs (see Figure 2 c), containing pointers to e.g. string data and method prototypes. As this is unimportant for the rest of this thesis, I will not elaborate on this. Finally, the DEX file has a data section.

2.2.1 The Data Section

The data section is shown in more detail in Figure 2 d). This section contains different components which are not referred to in the DEX header. Most of these component are optional (e.g. a debug component).

Yet, some components are mandatory, such as the code and the string data. The code contains actual instructions (the program body). The string data consists out of numbers defining the length of the string, followed by the string itself in `ascii` format.

The data section also has an (optional yet enforced [8]) component which contains the maplist (see Figure 2 f). This list contains sizes and offsets of every section and component in the current DEX file. This is partially redundant, because DEX header contains the section information too. The DEX header does not contain any information about the different components inside the data section. As mentioned in the last section, the pointer to the maplist (which is typically at the end of a DEX file) is found in the DEX header.

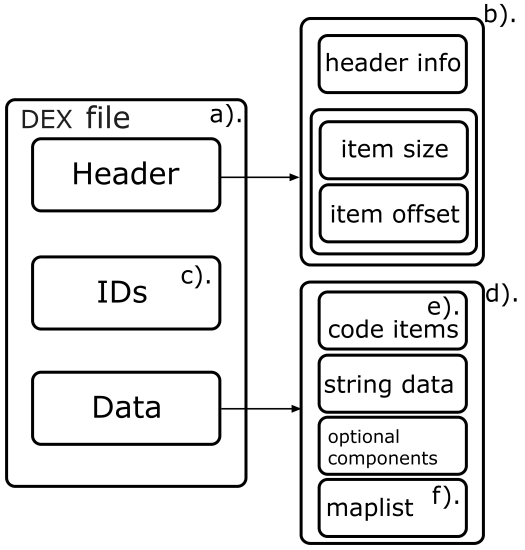


Figure 2: The DEX file structure

2.2.2 Code Section and APIs

The code section (Figure 2 e). exists out of multiple code items. A code item can best be seen as a function when programming. It contains information about the function in terms of amount of input and output arguments. Furthermore, it contains the function body, represented as a set of instructions. This may contain e.g. `goto` instructions or arithmetic operations.

An instruction has a variable size inside this bytecode section. Fortunately, the size of every instruction is fixed per instruction opcode, which is the first byte at every instruction. This opcode defines the instruction type; the rest of this instruction are registers, constants or pointers, depending on the instruction type. This research focuses on one type of opcodes: the virtual invoke opcodes. This invokes an external function. For example, a `send SMS` API call is done by a virtual invoke.

Android API calls are invoked using the `invoke-virtual` (0x6e) opcode or a variation on this: `invoke-virtual/range` (0x74), `invoke-virtual-quick` (0xf8) and `invoke-virtual-quick/range` (0xf9). One of the parameters of this opcode then gives a pointer to a string. Consequently, the string will tell what function to invoke.

2.3 From 1D to 2D Data

In this thesis, I generate images from binary files. When looking at the binary file as a black box, this is a one-dimensional (1D) array of bytes. However, (greyscale) images are two-dimensional, having both a width and a height. Hence, the 1D array needs to be converted into a 2D array in order to generate an image. Individual pixels can be scaled to the size of one byte (with `0xff` being a white pixel and `0x00` being a black pixel), so once the 2D array is created, an image can be generated.

Gamut has two ways of plotting the 1D array as a 2D array. The first and most intuitive way is called *linear plotting*. Setting a width w, this method has a mapping as follows:

$$binary[i] \quad = \quad image[\ i \ \%\ w,\ i\ /\ w\ ] \tag{2.1}$$

with % being the remainder operator and / being an integer floor division. In words, this means the topmost row of pixels in the image correspond to the first w bytes in the binary file. The second row of pixels correspond to the next w bytes, and so on. At the end, black pixel padding (null padding) is done to create the rectangular image.

The second technique used in Gamut is called *Hilbert curve plotting* or short, *Hilbert plotting*. This method uses a space-filling curve by Hilbert [16]. Although the theory behind Hilbert curves is beyond the scope of this thesis, the reason for choosing this curve is the pleasant property of locality. For comparison, in linear plotting, bytes $w-1$ and w are 1 byte away in the 1D vector, in linear plotting the positions are $f(w-1) = (0, w)$ and $f(w) = (1, 0)$. This is taken into account using Hilbert curves. Similar to linear plotting, black pixel padding is added.

Padding however is a downside of Hilbert curves. Hilbert curve images have a size of $2^n \times 2^n$ pixels: square with a power of 2 as width/height. This means Hilbert plotting needs much more padding in total. If we add one byte to a perfect fitting binary on linear plotting, we add w more pixels. When using Hilbert curves, the total image becomes *4 times as big*.

Chapter 3

Related Work

Ever since malicious software entered the field of computer science, people are searching for ways of detecting malware and preventing its propagation. The most used method of doing this is by fingerprinting known malware samples and comparing these to unknown samples. This is still widely used by antivirus programs. It is cheap and works relatively well. However, according to *G Data*, 9,468 new android malware samples are found each day on average [12]. These are clearly too many samples to manually check every time. We need an automated way of recognising new malware. Moreover, signature-based techniques do not work on fully polymorphic malware, where malware can rewrite its entire bytecode. Techniques have been proposed to counter this [6]. However, these techniques still rely on unique signatures created. A more robust technique is clearly preferred.

One thing that helps humans to understand what is going on, is by visualising parts of the program. It is much easier for humans to see patterns in a visual form compared to long strings of characters. Hence, there are various ways to visualise a malware sample. Related work in code visualisation is addressed in section 3.1, feature selection for malware classification is covered in section 3.2, classification and detection of malicious apps is covered in section 3.3.

3.1 Malware Visualisation

Multiple papers have been published in visualisation of binary files. One of the more generic papers is the one by Conti et al. [10], creating a tool to visualise binary files in four different ways. The first way is plotting each byte linearly as discussed in section 2.3 in a greyscale where a `null` byte is depicted as a black pixel and an `0xff` byte as a white pixel. The second approach colours parts of the bytecode to denote whether a specific byte-value exist. This is particularly useful to find compressed sections or `ascii` sections. Thirdly, a traditional hex editor is implemented. At last, they use dot plots to visualise cross entropy of the file. A dot plot is a method to visualise similarity or self-similarity of data.

Dot plots are also used to visualise the memory of malware at runtime, as shown by Wu et al [32]. In this paper, Wu et al. filter out redundant information in the memory and show promising results in manual malware classification. However, this technique suffers from Address Space Layout Randomization (ASLR). ASLR is a technique used to randomise the memory of a program, making it harder to reconstruct the full runtime memory of a program. ASLR has been implemented in Android since Android 4.0 [3] which makes the use of this technique significantly harder on Android. However, this is merely a problem when analysing the memory at runtime.

Jain et al. [17] parsed the sections of the DEX file to plot each section in a slightly different colour. They also plot the file linearly as explained in section 2.3 in this thesis. Furthermore, the first byte of every string object in the string data is coloured bright to

distinguish parts with very long strings from parts with many short strings. This is also a way of recognising the string data section without having to parse the maplist.
An interesting finding in this research was that most repackaging and obfuscation tools move different sections around. By colouring every section in a different colour, one could spot repackaging or obfuscation to the extent where one could manually recognise which tool has been used to do this. Note that repackaging or obfuscation does not directly imply the code is malicious. Especially obfuscation tools are widely used to prevent reverse engineering and repackaging [14].

Another tool that is related to the visualisation of malware or bytecode in general is a tool called Binvis [11]. In three of the four modes in Binvis, different colours are given to bytes in different ranges. For example, in the byteclass mode, `ascii` ranged bytes are coloured blue, whereas `null` bytes (`0x00`) are black and `0xff` byte is white. The last mode looks at the entropy of adjacent bytes and colours a brighter colour purple the further the bytevalues are away from each other. Equal bytevalues get a black pixel in the entropy mode.
Binvis uses a Hilbert curve as mentioned in section 2.3. They use a slight variation where the image size is not necessarily shaped as $2^n \times 2^n$ for an n. I chose not to use this in my research because the images need to be of equal size for the neural network. However, this might be explored in further research.

Nataraj et al. [24] plotted the bytes of Windows malware in a greyscale image in the same way as Conti et al. Afterwards, they apply Gabor filters (GIST) to the images to get features. Finally, K-nearest neighbours (KNN) is used as an automated classification technique. The results have an 98% accuracy on 25 malware families.

3.2 Feature Selection for Malware Classification

. Lately, a wide variety of techniques have emerged for feature extraction and classification/detection. Kang et al. discovered that looking at the app certificate reveals some information about the (malicious) intent of the app. On top of that, a list of permissions and a list of API calls are used as features. A support vector machine (SVM) is used for detection.

There has been a big focus on semantics as a feature, as this makes the classifier more robust [7]. Shen et al. discovered the detection became more accurate at new malware samples when using more semantics as features. Another classifier that shows the importance of semantics is DroidSieve [29]. DroidSieve extracts many features, both syntactic and resource-centric. The extra-trees algorithm is finally used for classification. The focus here is to have a big variety of features, rather than taking into account just a single (novel) feature. The results are high, with up to 99.82% accuracy with zero false positives.

Drebin by Arp et al. [4] and DroidAPIMiner by Aafer et al. [1] both extracted a large list of API calls to check for as features. DroidAPIMiner uses frequency analysis on the API calls of both benign and malicious apps and looks at the biggest difference to take as feature. Moreover, API calls only used by third-party packages are whitelisted.

Drebin has a more straightforward nature. Drebin collects many features such as permissions and network addresses. Here again, explainability is a focus as Drebin returns an explanation on why it labels an app as malware. Whilst Drebin uses a lightweight SVM for detection that can be easily run on a mobile device, DroidAPIMiner tested multiple techniques of which KNN turned out to perform the best in general for their features.

As feature engineering is widely done with promising results and explainability, is has never been enough and robust enough. Feature engineering is manually done, which gives an automatic bias onto the features. This thesis searches for the optimum between manual feature engineering and computer driven feature engineering using neural networks, combining the best of both worlds.

3.3 Android Malware Classification and Detection

The machine learning techniques mentioned in the previous section are well known and widely used by data scientists. However, new and less common techniques have emerged from the field of machine learning, some of which might have advantages in accuracy, robustness towards concept drift, explainability or feature engineering. One example of this is Yerima et al. [34] that uses Bayes for classification of Android malware. The features used here were permissions and API calls, with an accuracy of 92%. Against all expectations, the best results were obtained when using only 15 to 20 features.

Neural networks are proposed in various papers for classification or detection of Android malware. Yuan et al. [35, 36] uses a neural network of fully connected layers with different depth and width to find optimal circumstances. The feature space used consists out of permissions and API calls gathered by using association rule mining on their dataset. Using association rule mining, the biggest difference in permission and API call usage can be found. On top of this, Yuan et al. added behaviour (dynamic analysis) to their feature space. Surprisingly, a network with only two layers turned out to be optimal with an accuracy of 96.8%, compared to other networks up to six layers.
Automated techniques are used to find the features, but not without the assumptions that the difference of API calls and permissions in apps create a good feature space. Besides, using the same dataset for association rule mining and testing can give biased results. This thesis takes a closer look at the feature space and the assumptions when creating such a feature space.

McLaughlin et al. [22] extracts opcode lists to apply convolutional neural networks on these lists in the same sense as has been used in natural language processing (NLP). NLP is an active research field and McLaughlin et al. puts this knowledge to use in Android malware detection. Feature engineering is not done by hand, but is done purely by the neural network instead. The technique in this thesis is similar in the sense that feature engineering is done by the neural network and the techniques used are from a different research field. However, I investigate whether an optimum can be found between hand-picked features and feature engineering by the neural network.

Chapter 4

Dataset Representation

Gamut represents Android apps (APK) as images. It does so by plotting every byte as a pixel according to the bytevalue. In the previous chapter, multiple ways of visualising malware has been shown. Gamut combines multiple approaches together with a novel concept to get an accurate, lossless representation of the app. At first, we discuss the dataset used in the research. Afterwards, we discuss Gamut and the different ways in which Gamut can represent an app as an image. At the end, I mention the datasets generated.

4.1 Overview of the Data

McAfee Labs has provided me with a real-world dataset consisting out of Android apps (APKs) in three categories: malicious apps, benign apps and suspicious apps. This dataset has samples from February 2012 up to February 2016. As this thesis is looking into detection of malware, there is no use for the suspicious samples. Nevertheless, it is important to know these are filtered out instead of being distributed among the benign or malicious samples. The dataset contains 13,805 APKs labeled as malicious and 22,378 APKs labeled as benign. After extracting the DEX file(s) out of the APK, some samples did not have a DEX file. As these apps cannot run on an Android real-world phone, these samples had no use. A few apps had a second DEX file. I decided not to use these, as the DEX files both belong to the same app. This distorts training the network. Moreover, the difference in amount of second DEX files was not proportional. Over a thousand benign apps had a second DEX file, whereas we only found 30 in the malicious dataset. I assumed that the first DEX file is representative for the app.

While parsing, some DEX files did not start with a correct DEX magic. If the file does not start with a proper magic, the file is not recognised as a DEX file and the file is omitted. As a consequence, these files have been omitted in the dataset too.

In total, the total dataset used for training and testing has 13,794 malicious samples and 16,749 benign samples. When using more sophisticated methods of creating the images, some unexpected errors occurred during parsing. This is probably due to corrupt DEX files. Here we have a total of 13,509 malicious samples and 16,700 benign samples.

4.2 Gamut

Gamut is a tool written in C to convert a DEX file into an image without decompiling. Gamut uses a part of the parser written by Makan [21] and parts of the Google Android libdex library [2]. Gamut checks if the file has a correct magic before it continues. After parsing and colouring based on the given parameters, Gamut converts the 1D vector to a 2D vector[1] and outputs this as a PNG image.

4.3 2D Representation

Converting the 1D vector into a 2D vector can be done in multiple ways, as explained in section 2.3. Gamut can plot files both linearly and using a Hilbert curve.

When using linear plotting, the width of the image becomes the biggest power of two, such that the height is larger than the width. Using a power of two, the width of different images remain comparable. Making sure the height is bigger than the width would minimalise padding without losing the 2D nature, as I pad onto the width. If s is the total length of the 1D vector, the maximum amount of padding becomes $\lfloor \log_2(\sqrt{s}) - 1 \rfloor$.

For Hilbert curve plotting, the zero-padding is substantially more. The size remains the smallest square of two for the data to fit. Afterwards, the vector space gets filled using zero-padding. For the image, this means all remaining pixels are black. For s being the total length, the maximum amount of padding becomes $3(s-1)$. However, locality is preserved. Furthermore, locality is preserved upon the 2D space, rather than only horizontally. This is a major advantage when using convolutional layers in a neural network. Convolutional layers have a patch of a fixed size (typically 3×3 or 5×5) that sift over the image. In linear image, pixels that differ one pixel on the y-axis are approximately w bytes away from each other (with w being the width), while Hilbert curves have a 2D locality.

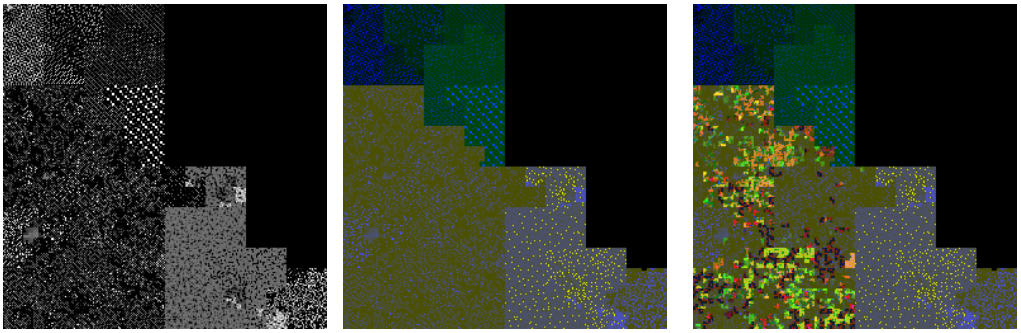
As both plotting techniques have major advantages and disadvantages, it is important to take both into account instead of assuming one is better than the other. Given the plotting technique, Gamut automatically calculates the optimal size of the image and outputs the image in this size.

4.4 Semantics

Semantics play an important role when looking at the bytecode of an app. A random chosen byte might be an `ascii` character as well as an opcode or part of a pointer address. When colouring different bytes in a different way, the image has a layer of semantics upon the raw bytecode.

Because I use a neural network as classification method which is semi-supervised, the method does its own feature extraction. It then becomes debatable whether to add more semantic to the image (hence steering the neural network), or to let the neural network decide what is useful. In Gamut, there are 5 levels of semantics possible (mode 0-4), all of these discussed below.

[1] Actually, after colouring the vector has a depth of 3: red, green and blue. Yet, this still needs to be converted to an $2D$ *RGB* image format



(a) mode 0 (b) mode 1 (c) mode 2

Figure 3: Hilbert images of a malware sample from the McAfee dataset

4.4.1 Mode 0: Greyscale

Mode 0, also referred to as the greyscale mode or the no-semantics mode is the basic way of plotting bytecode into RGB-pixels. The pixels are all greyscale (wich means the red-, green- and blue-values are all equal). The value of the pixel is equal to the bytevalue of the byte in the DEX file. Examples are shown in *Figure 3a and Figure 4a.*

4.4.2 Mode 1: Section Colouring

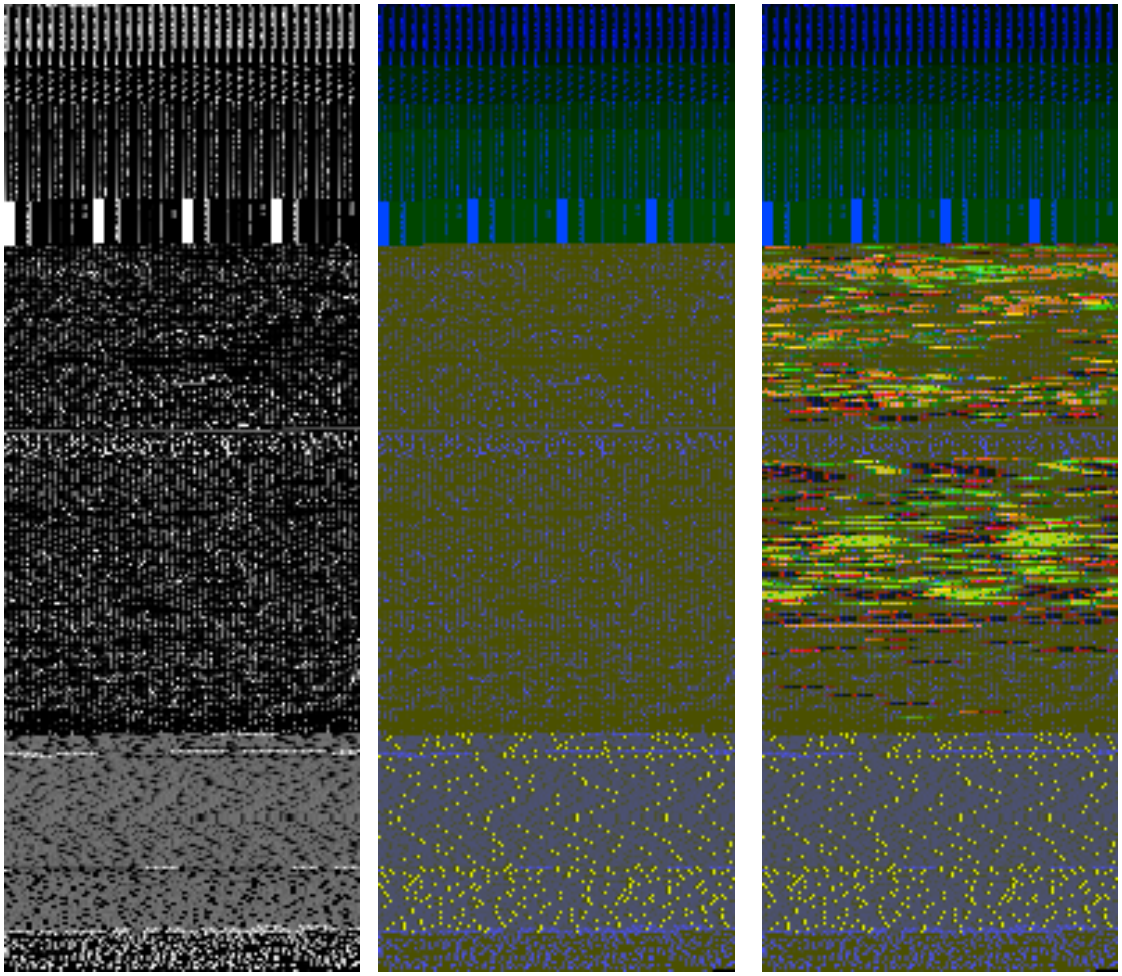
In mode 1, the section colouring mode, we look at the DEX structure as discussed in section 2.2. While colouring the blue-value as the original bytecode value, we use the red- and green-value of the pixel to specify different sections in the bytecode vector. In this mode, we assume that the (order of the) sections in a DEX file are worth distinguishing when detecting malware. More specific, the header has red and green value 0, and every subsequent section has a red and green value of 10 more than the previous one. Although the order of sections is not fixed, colouring is done using the order in section 2.2. A different ordering of sections is not very common and usually points to the use of an obfuscation tool [17].

Some apps have parts in the DEX file which are unaccounted for when parsing the sections with their corresponding sizes. This is unconventional, but it does not break the app. This can be used, i.e. to hide content to be used during runtime. In these parts, we set the red- and green-value to be the maximum: 255. This creates a yellow-white colour, depending on the byte-values (the blue-value) in this part. These parts are easily spotted by looking at the images manually too.

Similar to Jain et al., I colour the beginning of each string object in the string data subsection bright yellow, to distinguish areas with many short strings from long string areas.

The data section is treated here as one section. The different subsections of the data may or may not be present, and the range of a pixel value is limited. Colouring all subsections, of which the majority might not be present, would limit our pixel range even more. As this approach is extended in the following modes, this would use too much pixel range to extend this mode properly.

Examples of this mode are found in *Figure 3b and Figure 4b.*



(a) mode 0 (b) mode 1 (c) mode 2

Figure 4: Linear images of a malware sample from the McAfee dataset

4.4.3 Mode 2: Class Colouring

Mode 2 is called the class colouring mode. This mode is an extension of mode 1, meaning it colours the images in the same way as in mode 1. On top of that, Gamut parses the `invoke-virtual` opcodes.

A red- and green-value is determined by reading two bytes of the `SHA-1` value of the class name. These values are inserted in every byte of the instruction. Concluding, every `invoke-virtual` is coloured and two invokes from the same class get the same colour, apart from the blue-value. However, it can happen that different classes have similar or even equal colours due to collisions.

The next mode colours the `invoke-virtual` according to class plus method. This is explained in the next session. For this mode, I explicitly chose not to do this, as the amount of classes and methods is too big to capture in 2 bytes of information (red- and green value). Using only classes, we minimise the chance of a collision.

Using this mode, we make the assumption that `invoke-virtual` type instructions (i.e. API calls) are particularly interesting for determining whether apps are malicious or not. No harmless API calls are distinguished from potential harmful ones. Examples of this mode can be seen in *Figure 3c and Figure 4c.*

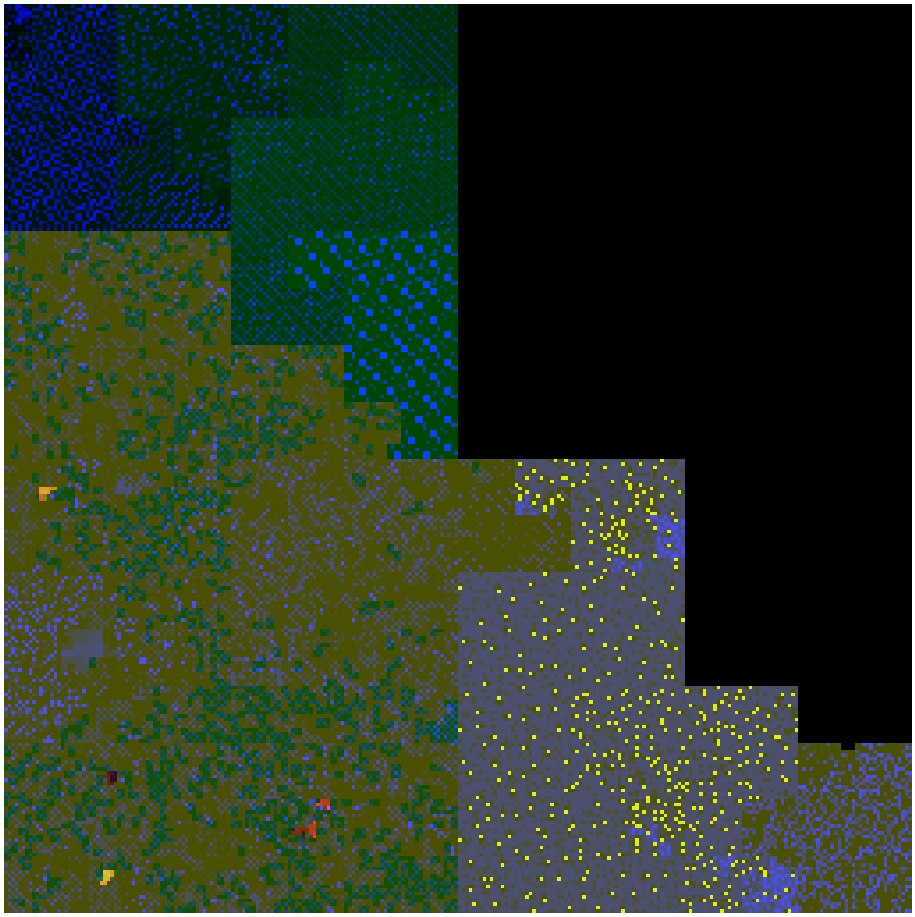


Figure 5: Hilbert image of the malware sample, mode 3

4.4.4 Mode 3: List Method Colouring

Within the list method colouring, I decided to not colour every `invoke-virtual`, but only the API calls of which the class plus method name corresponds to one in a list of suspicious API calls. In other words, API calls are not coloured if not blacklisted, assuming this list is a correct and complete feature space. For the list I used the same list as Drebin by Arp et al. [4]. This list has 1,242 entries of suspicious API calls.

In contrast to the class colouring, this mode extracts colours from the `SHA-1` of the class name with the method name appended. Doing this, different methods from the same class are distinguished. Gamut has to colour 1,242 different API calls using this mode. Collisions or close collisions might still happen, but the chances are significantly smaller. Hence, it is desirable to distinguish two different API calls, even if they originate from the same class.

The amount of opcodes coloured becomes very low in comparison to the class colouring mode. A linear example is shown in *Figure 7a* and a Hilbert plotted example is shown in *Figure 5*. The images may need a thorough inspection to spot every suspicious API call.

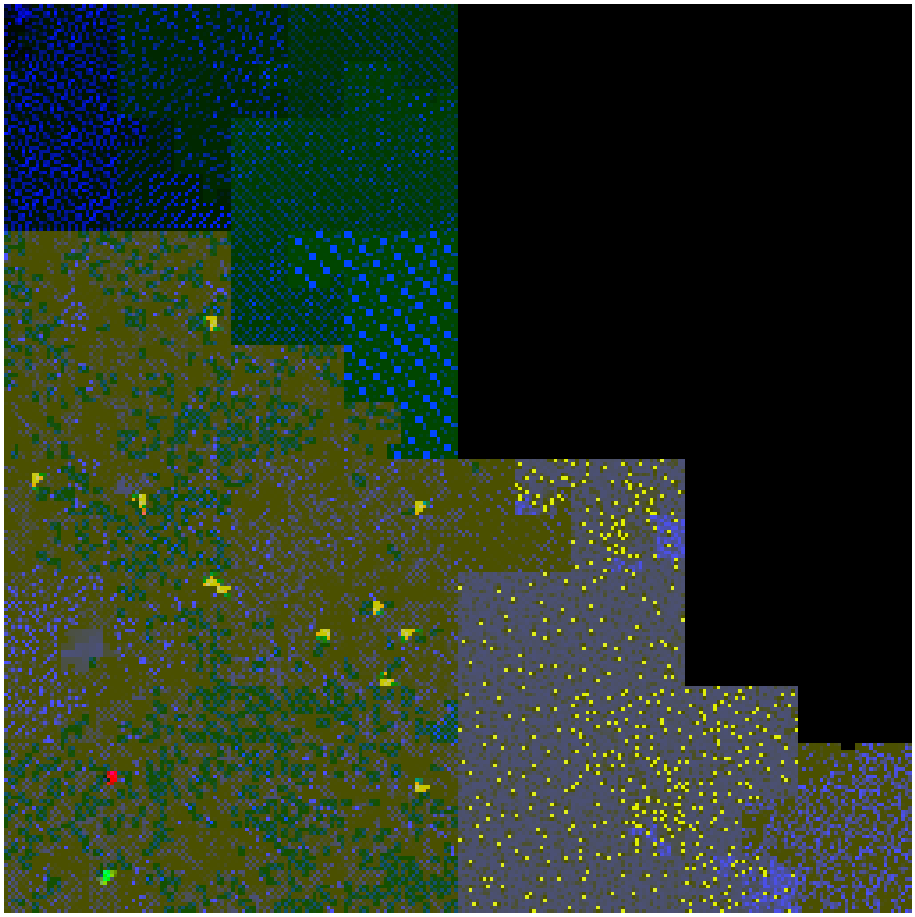


Figure 6: Hilbert image of the malware sample, mode 4

4.4.5 Mode 4: Labelled Colouring

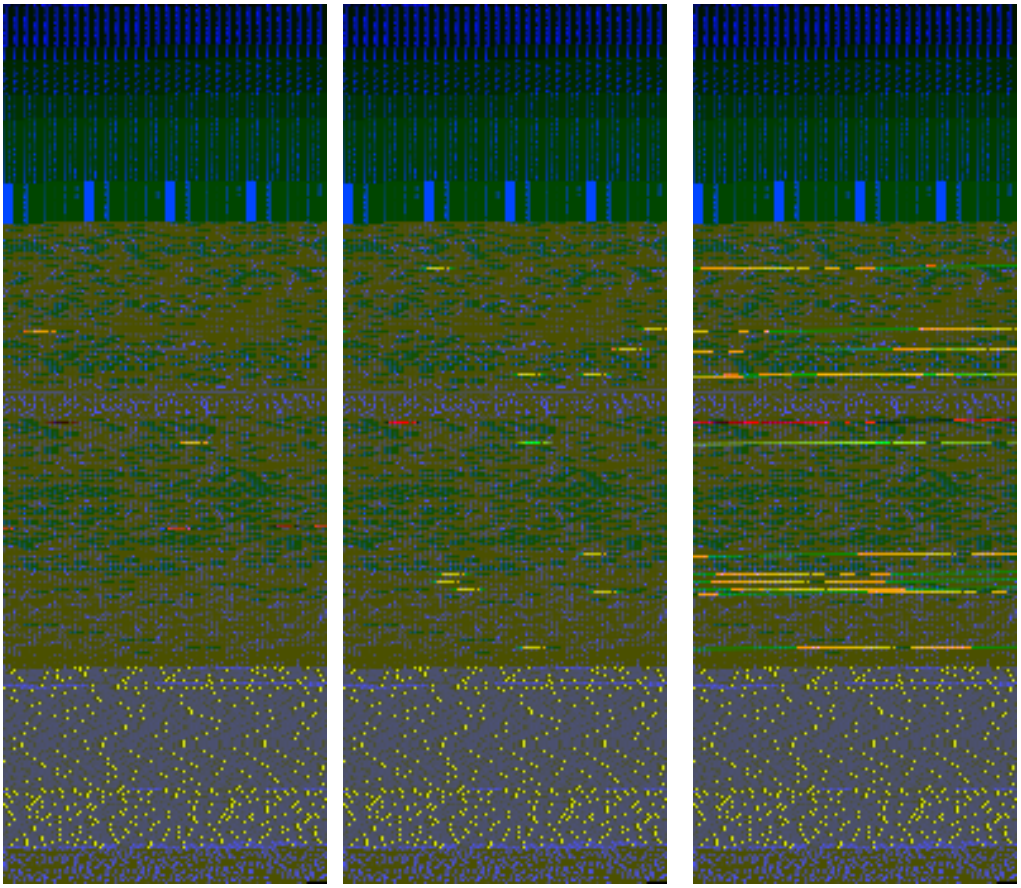
In mode 4 I manually selected a series of suspicious API calls to check for. In other words, this mode uses labels rather than the name of the method as a colour: different API calls from the same label get the same colour.

I labeled these in categories slightly suspicious, very suspicious and obfuscation. Each label has its own red- and green-value. API call examples and their colours are shown in table 4.1. This mode focuses on the explainability of the technique: For every label, there is a clear explanation on why this API call has this label. Assumptions with this mode are that (1) the manually selected feature space is correct and complete, and (2) the labelling done in the given labels are useful for the neural network.

A Hilbert image is shown in *Figure 6*, a linear image is shown in *Figure 7b*. In this image, you can see different types of suspicious API calls clustered together.

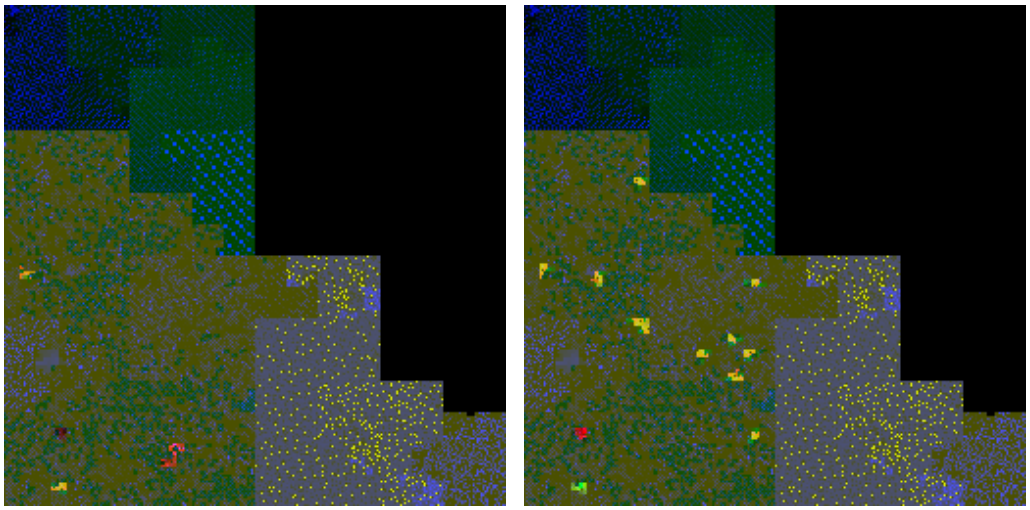
Label	Colour	Examples
Slightly suspicious	Yellow (`0xc8`,`0xc8`,-)	listFiles(), openConnection(), isWifiEnabled()
Very suspicious	Red (`0xfa`, `0x00`, -)	sendTextMessage(), killProcess(), getSimSerialNumber()
Obfuscation	Green (`0x00`, `0xfa`, -)	SecretKeySpec(), getIV(), loadLibrary()

Table 4.1: Labels from mode 4 with example methods



(a) Mode 3 (b) Mode 4 (c) Mode 4 gradient 20

Figure 7: Linear images of the malware sample



(a) Mode 3, gradient 20 (b) Mode 4, gradient 20

Figure 8: Hilbert curve images of the malware sample with a gradient

4.5 Image Size

As further discussed in chapter 5, I use a neural network as classification method onto these images. However, neural networks are only known to be used onto small or downscaled images. This has two main reasons:
(1) neural networks used on images are very resource- and computing-intensive and,
(2) by nature of a fully connected layer, all images used need to have the same size.

The first one is the most important. Alexnet from 2012 used images of size 256×256 [19]. In 2015, Google published GoogLeNet [30], using images of size 224×224. Even if it is possible to process larger images, it would be at the expense of both the depth and the width (i.e. amount of nodes) of the network. On natural images, it is shown that downscaling and resizing has minor influence on the results, whereas bigger neural networks can change results drastically.

The Hilbert images generated from APKs have sizes ranging from 128×128 until 4096×4096. The latter one is a definite maximum, as DEX files have a maximum size smaller than $4096 \times 4096 = 16,777,216$ bytes. In case the app is larger, it would have a second DEX file. Most images however range from 512×512 to 4096×4096. Linear images are slightly smaller due to less padding, but are still in this range compared to the 256×256 or 224×224 pixel images mentioned in the previous paragraph.

As mentioned, fully connected layers need images of exactly the same size. The maximum size our setup could handle were images of 256×256 pixels. Hence, I downscaled every image to 256×256. These are fairly natural compared to the Hilbert curve images and the width of the linear images, because the images get scaled down a power of 2. However, it is to be noted that the images lose a lot of information by downscaling: a 4096×4096 image downscaled to 256×256 would take every 16×16 pixel block (256 pixels in total) and average out this pixel to one. On 128×128, as used with a different experiment, this would be averaging a 32×32 pixel block to one pixel, being 1024 pixels.

4.5.1 Gradients

There is another problem with scaling down images. Not only does it lose a lot of information, in particular it loses the coloured pixels which represent the suspicious API-calls in the list method colouring (3) and the labeled colouring (4). After downscaling, the bright pixels cannot be spotted anymore and the question arises whether a neural network still can.

I address this problem by using image gradients on the bright pixels. Over a fixed length, the API call is coloured around the API call too, fading out the further away the pixel is from the bright-coloured API-call.

d = distance to the API call, s = gradient size

$$c_{new} = c_{old} + c_{\text{API}} \cdot max\{1 - \frac{d}{s}, 0\} \tag{4.1}$$

Once again, this goes for the red and green colours of the pixel, the blue value remains the bytevalue of that byte. Without downscaling, no single bit of information of the DEX file is lost.

The length of the gradient can be given to Gamut as a parameter. However, the actual length (in pixels) will be proportionate to the size of the to-be-generated image. Thus, the gradients of different images are (approximately) of equal size after downscaling. With s as the user-given gradient size, the total gradient length becomes:

$$s_{new} \approx s \cdot \frac{min\{2^{2n} \mid 2^n \geq height\}}{128 \times 128} \tag{4.2}$$

For images of size 128×128, the given gradient size is equal to the given gradient size. For images of a larger size, the gradient size scales linearly on $height^2$. For instance, an image of size 512×512 with a given gradient of 20 will have an actual gradient of $20 \cdot \frac{512^2}{128^2} = 20 \cdot 16$ pixels. Two Hilbert examples are shown in *Figure 8*, a linear example is found in *Figure 7c*.

Gamut also has an option of using a non-scaling gradient. I created one such dataset for comparison. To use the non-scaling gradient in Gamut, one has to give the negative number of the gradient size as a parameter. Explicitly, an image of size 512×512 with a gradient of −20 will have a gradient of exactly 20 pixels.

4.6 Datasets generated

Using different modes in Gamut in linear and Hilbert curve plotting with possible use of the gradient, a total of 18 different image datasets are generated from the original dataset. A detailed description of these are given in table 4.2. After generating the original images, we scaled them down in batches to a size of 256×256 pixels. For tests on 128×128 images, downscaling is done at runtime.

In total, the datasets exist out of more than 1.8TB.

Mode	Plotting	Gradient	Mode	Plotting	Gradient
0	Hilbert	0	3	Hilbert	5
0	Linear	0	3	Hilbert	10
1	Hilbert	0	3	Hilbert	20
1	Linear	0	3	Hilbert	-512 (non-scaling 512)
2	Hilbert	0	3	Linear	20
2	Hilbert	5	4	Hilbert	0
2	Linear	0	4	Linear	0
3	Hilbert	0	4	Hilbert	20
3	Linear	0	4	Linear	20

Table 4.2: Used combinations of modes with plotting methods and gradients

Chapter 5

Evaluation

5.1 The Neural Networks

After generating different image dataset, I used a neural network for the binary classification between labels 'malicious' and 'benign'. I used an Nvidia Tesla K40c for training and testing.

I built two neural networks. The first (CNN2) has 2 convolutional layers and 1 fully connected layer, the second neural network (CNN4) has 4 convolutional layers and 2 fully connected layers. Both use ReLu's as activation function and both have a dropout layer with probability 0.5 [13][28]. Using CNN4 on 256×256 images, the setup ran out of memory. Hence, we only used the first neural network for the 256×256 images. Both neural networks are created using Googles TensorFlow framework [31], based on Python 2.7.

For training, I used a simple holdout, using 80% for training and 20% for testing. Training is done using the stochastic gradient descent (SGD) algorithm, one of the most popular back-propagating algorithms in the field of neural networks. Each image has a label "malicious" or "benign" to perform a binary classification. Training was done for 20 epochs. More epochs turned out not to be useful and would only overfit the model.

Evaluating the model is done using the general notion of accuracy percentage in binary classification. More specifically:

$$Accuracy\ \% = \frac{TP + TN}{TP + TN + FP + FN} \cdot 100$$

where TP is the amount of apps correctly detected as malicious (true positive), TN is the amount of apps correctly detected as benign (true negative), FP is the amount of benign apps erroneously classified as malware (false positive) and FN is the amount of malicious samples incorrectly classified as benign (false negative). Because of its binary nature, 50% would be a bad score and 100% would be a perfect score.

5.2 The Framework

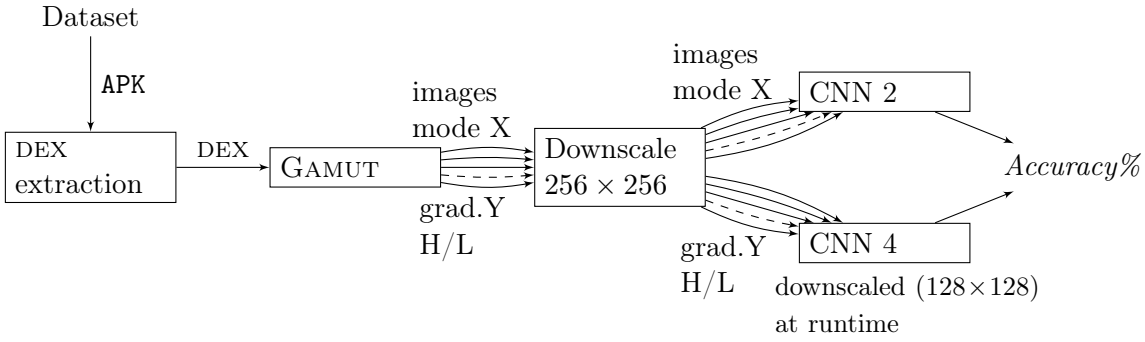


Figure 9: The framework

The first step is to extract the DEX files from the original APK dataset. These DEX files are converted into multiple image datasets using Gamut. The different datasets generated are given in Table 4.2.

Every dataset needs to be downscaled afterwards before it is usable in the neural network. This can be done in runtime, but loading the full-sized images soak up a lot of time and has to be done only once if we do this first. For the smaller images (128×128), this is done during runtime as the overhead here is negligible.

Now, the images are ready to be fed to the neural networks. Every created dataset is used upon both CNN2 and CNN4. The dataset is randomly split, using 80% for training and 20% for testing. As a result, we have a trained neural network with a benchmark of the quality of the network database combination.

5.3 Results

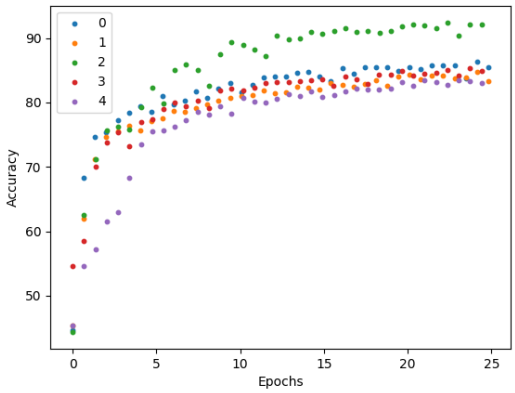
Figure 10 depicts the accuracy of different datasets with respect to the amount of epochs using the neural network of 2 convolutional layers and 1 fully connected layer. The images used have a size of 256×256 pixels. Various results are also shown in table 5.1.

In general, the linear plotted images seem to have slight better results. This is easily explained: more padding is needed to create Hilbert-curve images. Hence, the information loss in the linear images is lower in comparison to the Hilbert-curve images.

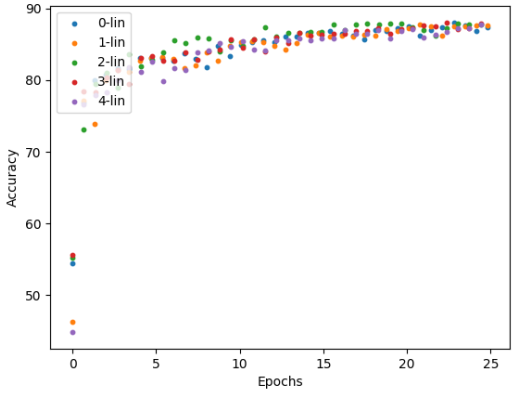
However, when looking at the y-axis of Figure 10a, you can see the scale is slightly different compared to the other images. This is because the mode 2 Hilbert-curve images without gradient have a significant higher accuracy than any other dataset, peaking over *92% accuracy*.

Mode	L/H	Gradient	acc at 20 ep
0	H	0	85%
0	L	0	88%
1	L	0	87%
2	H	0	**92%**
2	L	0	87%
3	H	20	86%
4	H	20	86%
4	L	20	88%

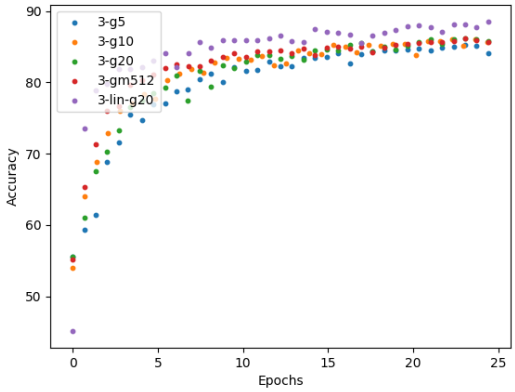
Table 5.1: Various datasets with accuracy on the 2-conv network



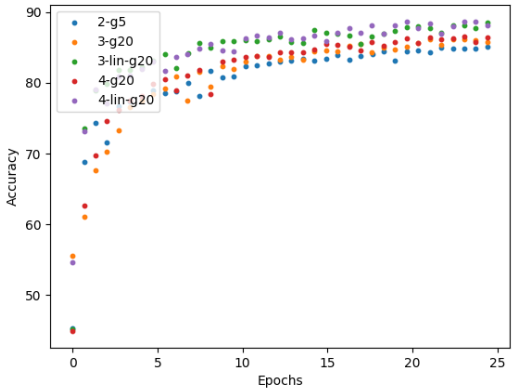
(a) Hilbert curve images without gradient



(b) Linear plotted images without gradient



(c) Various gradients and plotting techniques, mode 3



(d) Miscellaneous plots with a gradient

Figure 10: Accuracy vs. amount of epochs with a network with 2 convolutional layers

Some results from the tests with 4 convolutional layers are shown in table 5.2. On average, these have slightly better results in comparison to the small neural network. However, the best results are still gained by using mode 2 on the smaller network. Recapping, mode 2 is the mode that colours every API call according to the class it originates from. This mode has the highest level of undirected semantics, as higher modes focus on just a part of the API calls.

Mode	L/H	Gradient	acc at 20 ep
0	H	0	86%
0	L	0	89%
1	L	0	88%
2	H	0	89%
2	L	0	89%
3	H	20	85%
4	H	20	85%
4	L	20	**90%**

Table 5.2: Various datasets with accuracy on the 4-conv network

The higher accuracy is most likely due to the detail of the image in 256×256 pixels rather than the 128×128 pixels downscaling needed for the 4 convolutional layer network. Another reason could be that the linear images become distorted because image sizes are variable. Pixels are not averaged out two-to-one, which stretches images out of proportion. Linear images have a flexible length, so different images get stretched in different ways, making the images less comparable. On top of this, Hilbert curves preserve locality both horizontally and vertically, making this better for usage with convolutional layers.

Chapter 6

Discussion and Future Work

The setup has proven to work well for malware detection. Neural networks take a lot of time to train, but once trained, classification is very fast, and thus feasible to use locally offline. However, with a 92% accuracy this is not something to be used directly, as approximately 8% of apps would be incorrectly classified (about one app per 12 or 13 apps).

Improvements on this method are mentioned below. First, I will mention improvements for the images and problems with the images, secondly I give some thought on the possibilities within the neural network.

6.1 The Images

In terms of the images, more semantics could be added for different parts of the DEX file. For example, data section components as the debug section might be given a different colour if present, the string data subsection could be parsed for anomalies or 3rd party code could be left out [5]. Many options are left open.

However, in order to know what to extend, it would be interesting to look for explainability in the neural network [37]. The downscaled images will make this extra hard to trace. After this, the pixel-values can be read to determine the original part of the DEX file. This can be done for two reasons: understanding whether it makes sense to recognise the malware in such a way [27] and the other way around: understanding what might be good recognition points to recognise malware in general. This could enhance the decisions made in adding semantics and gives a better understanding of malware in general.

A problem with the current approach is the downscaling of the images. Images reach up to 4096×4096 pixels, to be scaled down to 256×256 or 128×128 pixels. A lot of detail gets lost here. The amount of information getting lost might degrade images to the extend where other modes outperform them. Nonetheless, modes 3 and 4 with a gradient should be affected less in comparison with the other modes, as the extra semantics added scale along with the downscaling of the image.

The gradient approach did work slightly better than the ones without a gradient, but not to the extent where it outperforms the class colouring mode. Possibly, the CNN looks at combinations of API calls, detecting certain combinations. Without specifically colouring the non-suspicious API calls (as we do in the list method colouring and the labelled colouring), the semantic of these API calls are very hard to identify.

A way to circumvent the downscaling, would be to use recurrent models as proposed by Mnih et al. [23] to look for interesting regions in the image. Afterwards, the model looks at this region at higher resolution. Regardless, it is not likely to be able to process images of original size so far.

6.2 Neural Networks

Using a recent dataset and holding out a random 20%, it gives a clear picture of the quality for a freshly trained neural network. However, it is not known if this technique is sensitive to concept drift. Hence, it would be interesting to see confidence rates of this neural network when confronted with concept drift [18].

Because of the focus on image generation, I chose to run simple neural networks. However, better results may be achieved by using state-of-the-art techniques in neural nets, such as ELUs [9], PReLUs and RRELUs [33]. More may be experimented using spacial pyramid pooling to overcome variable image sizes [15]. This is a layer in between the convolutional layers and the fully connected layers to convert the variable-sized height and width to a fixed size to have a proper input for fully connected layers. This could be combined with an enhanced Hilbert curve that does not need to be of square size (as done in Binvis [11]) to reduce padding. Finally, more sophisticated optimisation algorithms could be used.

The usage of neural networks in security is not not uncontested either. Papernot et al. [26] [25] have shown to output images wrongly classified with minimal distortion. Even though minor distortions might have a huge impact on program code, it is a field to be explored if this neural network can be circumvented using adversarial examples.

Finally, it is interesting to know if this proposed technique works well on multiclass classification.

Chapter 7

Conclusion

In this thesis, I propose Gamut, a novel and off-the-shelf approach to convert DEX files into images with a user-controlled level of semantics. Afterwards, image datasets are used for malware detection.

A simple convolutional neural network achieved *92%* accuracy on the class-coloured images, the mode where every single `invoke-virtual` instruction gets coloured. This shows convolutional neural networks work well on non-natural images, despite the explanation for quality of neural networks by Lin et al. [20]. With the class-coloured images as best mode, our findings suggest that computers are still better at malware feature engineering than us humans are. However, providing computers with more semantics will improve the feature engineering of the computer.

Bibliography

[1] Y. Aafer, W. Du, and H. Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *International Conference on Security and Privacy in Communication Systems*, pages 86–103. Springer, 2013.

[2] Android. nougat libdex library, google git. https://android.googlesource.com/platform/dalvik/+/nougat-release/libdex. Accessed: 2016-10-06.

[3] Android. Security enhancements in android 1.5 through 4.1. https://source.android.com/security/enhancements/enhancements41.html. Accessed: 2017-05-03.

[4] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.

[5] M. Backes, S. Bugiel, and E. Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 356–367. ACM, 2016.

[6] S. Cesare, Y. Xiang, and W. Zhou. Malwise—an effective and efficient classification system for packed and polymorphic malware. *IEEE Transactions on Computers*, 62(6):1193–1206, 2013.

[7] W. Chen, D. Aspinall, A. D. Gordon, C. Sutton, and I. Muttik. More semantics more robust: Improving android malware classifiers. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 147–158. ACM, 2016.

[8] R. Chiossi. A deep dive into dex file format. http://elinux.org/images/d/d9/A_deep_dive_into_dex_file_format--chiossi.pdf. Accessed: 2017-05-02.

[9] D.-A. Clevert, T. Unterthiner, and S. Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[10] G. Conti, E. Dean, M. Sinda, and B. Sangster. Visual reverse engineering of binary and data files. In *Visualization for Computer Security*, pages 1–17. Springer, 2008.

[11] A. Cortesi. Binvis. http://Binvis.io/. Accessed: 2017-05-03.

[12] GDATA. G data mobile malware report. 2016.

[13] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, volume 9, pages 249–256, 2010.

[14] Guardsquare. Mobile app security software for android apps. https://www.guardsquare.com/en/dexguard. Accessed: 2017-05-06.

[15] K. He, X. Zhang, S. Ren, and J. Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. In *European Conference on Computer Vision*, pages 346–361. Springer, 2014.

[16] D. Hilbert. Ueber die stetige abbildung einer linie auf ein flächenstück. In *Mathematische Annalen*, pages 459–460, 1891.

[17] A. Jain, H. Gonzalez, and N. Stakhanova. Enriching reverse engineering through visual exploration of android binaries. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, page 9. ACM, 2015.

[18] R. Jordaney, Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro. Misleading metrics: On evaluating machine learning for malware with confidence. *Tech. Rep.*, 2016.

[19] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[20] H. W. Lin and M. Tegmark. Why does deep and cheap learning work so well? *arXiv preprint arXiv:1608.08225*, 2016.

[21] K. Makan. A very rudimentary android dex file parser. https://github.com/k3170makan/dexinfo. Accessed: 2016-10-06.

[22] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé, and G. Joon Ahn. Deep android malware detection. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 301–308. ACM, 2017.

[23] V. Mnih, N. Heess, A. Graves, and K. Kavukcuoglu. Recurrent models of visual attention. In *Advances in Neural Information Processing Systems 27*, pages 2204–2212. 2014.

[24] L. Nataraj, S. Karthikeyan, G. Jacob, and B. Manjunath. Malware images: visualization and automatic classification. In *Proceedings of the 8th international symposium on visualization for cyber security*, page 4. ACM, 2011.

[25] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. Berkay Celik, and A. Swami. Practical black-box attacks against deep learning systems using adversarial examples. *arXiv preprint arXiv:1602.02697*, 2016.

[26] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The limitations of deep learning in adversarial settings. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 372–387. IEEE, 2016.

[27] M. T. Ribeiro, S. Singh, and C. Guestrin. " why should i trust you?": Explaining the predictions of any classifier. *arXiv preprint arXiv:1602.04938*, 2016.

[28] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[29] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, and L. Cavallaro. Droidsieve: Fast and accurate classification of obfuscated android malware. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 309–320. ACM, 2017.

[30] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[31] TensorFlow. Tensorflow framework. https://www.tensorflow.org/. Accessed: 2017-05-02.

[32] Y. Wu and R. H. Yap. Experiments with malware visualization. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 123–133. Springer, 2012.

[33] B. Xu, N. Wang, T. Chen, and M. Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.

[34] S. Y. Yerima, S. Sezer, G. McWilliams, and I. Muttik. A new android malware detection approach using bayesian classification. In *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, pages 121–128. IEEE, 2013.

[35] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue. Droid-sec: Deep learning in android malware detection. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 371–372. ACM, 2014.

[36] Z. Yuan, Y. Lu, and Y. Xue. Droiddetector: android malware characterization and detection using deep learning. *Tsinghua Science and Technology*, 21(1):114–123, 2016.

[37] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *European Conference on Computer Vision*, pages 818–833. Springer, 2014.