Towards Mobile Malware Detection Through Convolutional Neural Networks

Nada Lachtar, *Member, IEEE*, Duha Ibdah, *Member, IEEE*, Anys Bacha *Member, IEEE*,

***Abstract*—Traditional research on mobile malware detection has focused on approaches that rely on analyzing bytecode for uncovering malicious apps. Unfortunately, cybercriminals can bypass such methods by embedding malware directly in native machine code, making traditional methods inadequate. Another challenge that detection solutions face is scalability. The sheer number of malware released every year makes it difficult for solutions to efficiently scale their coverage.**

This paper presents an energy efficient solution that uses convolutional neural networks (CNN) to defend against malware. We show that systematically converting native instructions from Android apps into images using Hilbert space-filling curves and entropy visualization techniques enable CNNs to reliably detect malicious apps with near ideal accuracy. We characterize popular CNN architectures that have been known to perform well on different computer vision tasks and evaluate their effectiveness against malware using an Android malware dataset.

I. Introduction

THE advent of deep learning combined with convolutional neural networks (CNN) has demonstrated significant promise in tackling a multitude of computer vision problems including brain tumor detection, developmental disorder classification, and facial expression recognition.

Prior work has examined malware detection for mobile systems [1], [2]. However, such work either focused on detecting malware using high-level byte code which make them vulnerable to techniques that pack malicious code directly in native form [1], or rely on machine learning algorithms that don't generalize well, can't perform feature extraction, and are susceptible to overfitting [2]. In this paper, we explore the use of CNNs for detecting mobile malware. We evaluate this approach by extensively testing different CNN architectures. We show that our proposed solution is to achieve a detection rate of 99.7% when converting the native instructions of Android apps into images through Hilbert space-filling curve techniques. We evaluate our solution against six ransomware families available in an Android malware dataset [3].

Overall, this paper makes the following contributions:

- Presents a novel approach for using native instructions from mobile apps with convolutional neural networks.
- Characterizes popular CNN architectures and demonstrates their relevance to detecting malicious Android apps while showing that a detection accuracy of 99.7% can be achieved.
- Evaluates the effectiveness of Hilbert space-filling curves and entropy in converting native ARM instructions from Android apps into images and their relevance to mobile malware detection when used with CNNs.

The rest of this paper is organized as follows: Section II provides background information. Section III presents the design of the proposed system. Section V presents the methodology and results of our evaluation; and Section VI concludes.

The authors are with the University of Michigan, Dearborn, MI, 48128.

This work was supported in part by the National Science Foundation under grant CNS-1947580.

II. Background

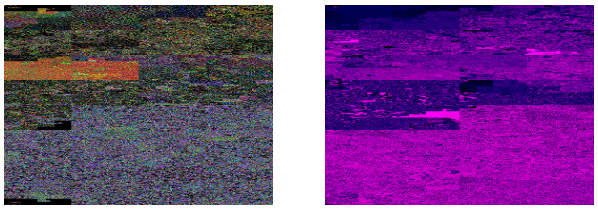
A. Native Execution in Android Platforms

Android applications are consumed in the form of a package that is known as the Android Package Manager (APK). This bundle consists of multiple files including a Dalvik executable (`dex`) file that contains Java bytecode. Once an APK is installed, the app can request from the OS the various services it needs. However, before the app can execute, it must undergo a series of compilation steps that are administered by the Android Runtime system (ART). ART relies on ahead-of-time compilation techniques for executing Android apps. ART archives any given APK by compiling the `dex` file into an `OAT` file (ahead-of-time file) through one of its modules. This approach is designed to significantly speed up application performance compared to the just-in-time compilation approach. Our solution makes use of such `OAT` files as the basis for generating images that can be consumed by our trained CNNs.

B. Space-filling Curves

A space-filling curve is a function that has the ability to map sets of data into a multi-dimensional hypercube. It has the property of passing through all the points in a given space while visiting each point only once. Therefore, it can impose a linear ordering of points in a multi-dimensional space and preserve spatial information. This property is important to this study since it preserves the ordering of instructions present in apps after they are transformed into images.

This study makes use of the Hilbert space-filling curve for mapping instructions into pixel locations within a 2D image based on [4]. We generate two sets of data images that use different coloring schemes. The first scheme makes use of the Hilbert curve for generating a fine-grained RGB palette. This ensures that similar instructions will be assigned similar pixel values based on the RGB. An example of this is shown in Figure 1a. The second scheme is less granular and makes use of entropy in conjunction with the red and blue components of the RGB space. It relies on the Shannon entropy over a window of n pixels to define the intensity of the aforementioned components. Although, not as granular,



(a) Hilbert Color (b) Entropy Color

Fig. 1: Example of native code for a ransomware app transformed into images using Hilbert and entropy-based colors.

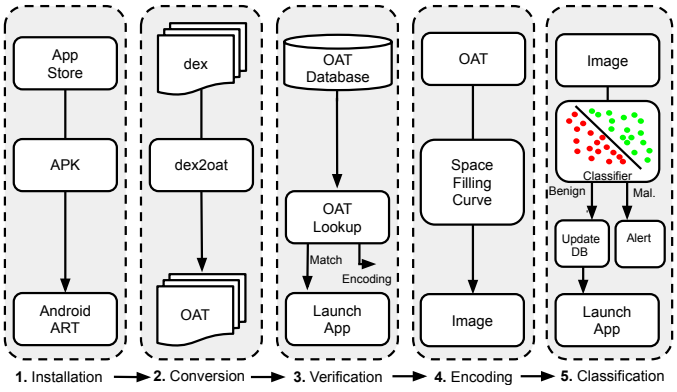


Fig. 2: Detection Solution Overview.

visualizing data through entropy has the benefit of highlighting encrypted content that is often an indicator of malicious content. An example of this is shown in Figure 1b.

III. Design

A primary objective of our design is to defend against obfuscation techniques that involve injecting native code into repackaged mobile apps. To this end, our design relies on native instructions as part of its detection process. An overview of our design is shown in Figure 2. The process begins with a user installing a desired application using an APK bundle from the app store (step 1). This prompts the ART to extract the `classes.dex` which represents the executable in Dalvik bytecode form. The ART then generates an `OAT` image through a `dex2oat` module to produce an ELF formatted file that contains the native instructions that will be executed on the device (step 2). Once an `OAT` image is available, our design proceeds to the verification phase (step 3) in order to determine if the app, in its native form, is safe to launch. To achieve this, we use a small `OAT` database that consists of all the `OAT` names installed on the system and their corresponding hashes. We maintain the most recent hash for each `OAT` image within the database using the `SHA512` algorithm. An important step in this phase consists of the `OAT` lookup. This entails computing the hash of the converted `OAT` file the user is attempting to launch and matching it to the database. If a valid entry for this `OAT` image is found within the database, we compare its computed hash against the most recently saved hash for that same `OAT`. If the hashes match, we launch the app. Otherwise, we proceed to the encoding phase. The encoding phase (step 4) transforms the `OAT` file into a pixelated image that can be consumed by a CNN. We use the Hilbert space-filling curve visualization technique to achieve this transformation. This image is then input into a CNN-based classifier (step 5). In the event that the image is classified as benign, the design updates the `OAT` database to include the most recent hash and launches the app. Otherwise, an alert is issued to the user. In most cases, a user attempting to launch an app will only result in step 3 being activated. This step represents the common case. Steps 4 and 5 are only used if the user installs or updates an app.

IV. Threat Model

We assume malware is installed on a device through mechanisms supported by the OS which entails downloading an `APK` from an app provider. We assume no privilege attacks occur on the device and that the OS hasn't been compromised. We do not consider traditional database attacks in our model since our `OAT` database is internal to the system. Finally, we assume malware can masquerade as a benign app and perform delayed updates as a way of fetching malicious content. To thwart such attacks, our design verifies `OAT` files on every launch.

V. Evaluation

A. Experimental Setup

We re-purposed three CNN architectures for this study: LeNet, Alexnet, and InceptionV3. To generate `OAT` files that correspond to the APKs within our dataset, we created a framework similar to [2] that is based on the Android 6.0.1 release. The framework was used to convert the app executables from Dalvik bytecode (`classes.dex`) to Android `OAT` files that use a standard Executable and Linkable Format. We used TensorFlow 1.12 for testing the CNN architectures and transformed `OAT` files into 256×1024 images through the Hilbert space-filling curve. We used a Raspberry PI 3 B+ (RPI) to characterize the suitability of our design for mobile systems. We used the Klein Tools ET920 USB digital meter for measuring the average power of the RPI platform while we ran the different phases of our design. In addition to power, we collected the runtime information for each phase in order to compute the overall energy consumption.

We conducted experiments using 2063 malware samples from six ransomware families in [3] and 13022 samples from AndroZoo [5] as a baseline for benign Android apps. The aforementioned samples were combined to form two datasets that we trained and tested with, which consisted of the following sizes: 3K and 15K image datasets. The breakdown for the aforementioned datasets consisted of 1500 ransomware and 1500 benign, and 2063 ransomware and 13022 benign images. We dedicated 80% of each dataset for training, 10% for validation, and 10% for testing. We generated two types of images for each dataset with each image type using a different coloring scheme. The first type employed an order 8 Hilbert curve for mapping instruction opcodes into RGB pixel values. The second type used an entropy-based approach for setting the red and blue components in the RGB space. Each pixel value was computed based on entropy level over 256 opcodes resulting in colors that scaled between black and pink depending on the amount of entropy.

Model	Accuracy		True Positive Rate (TPR)		False Positive Rate (FPR)		Precision		F-score	
	Small (3K)	Large (15K)	Small (3K)	Large (15K)	Small (3K)	Large (15K)	Small (3K)	Large (15K)	Small (3K)	Large (15K)
LeNet	0.993	0.997	0.987	0.981	0	0	1	1	0.993	0.990
Alexnet	0.993	0.997	0.993	0.976	0.007	0	0.993	1	0.993	0.988
InceptionV3	0.987	0.997	0.993	0.976	0.020	0	0.980	1	0.987	0.988

TABLE I: Summary of quality metrics under different CNN models for the small (3K) and large (15K) Hilbert-color dataset.

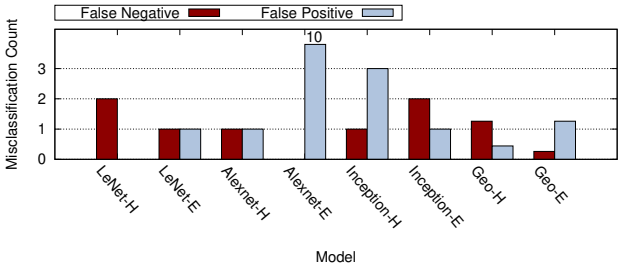
Model	Accuracy		True Positive Rate (TPR)		False Positive Rate (FPR)		Precision		F-score	
	Small (3K)	Large (15K)	Small (3K)	Large (15K)	Small (3K)	Large (15K)	Small (3K)	Large (15K)	Small (3K)	Large (15K)
LeNet	0.993	0.997	0.993	0.985	0.007	0.001	0.993	0.995	0.993	0.990
Alexnet	0.980	0.995	1	0.981	0.067	0.001	0.938	0.995	0.968	0.988
InceptionV3	0.990	0.995	0.987	0.976	0.007	0.002	0.993	0.985	0.990	0.980

TABLE II: Summary of quality metrics under different CNN models for the small (3K) and large (15K) entropy-color datasets.

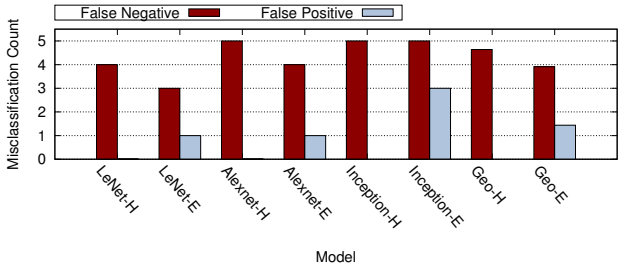
B. Model Analysis

Our evaluation focused on two dataset sizes for each coloring scheme. The first dataset size was small and balanced with an equal amount of malicious and benign images (3K images). The second dataset was larger (15K images), but imbalanced due to more benign samples being present in the dataset. Tables I and II summarize the quality metrics for the different architectures using the aforementioned dataset sizes for each of the coloring schemes. Overall, we observed that LeNet had the best performance when using the Hilbert-color dataset, followed by Alexnet, then InceptionV3. LeNet consistently had the best accuracy and F-score for both dataset sizes. In terms of malware classification, Alexnet and InceptionV3 showed better true positive rates (TPR) of 99.3% when using the smaller dataset (3K). However, their TPRs dropped to 97.6% when using the larger 15K dataset while LeNet's TPR improved to 98.1%. LeNet also exhibited 0% false positive rates for both data sizes. A similar trend was observed with the entropy-color dataset.

To better understand the performance of these models, we examine the amount of false negatives and false positives each model generates. We also compare the performance of the CNN models as a function of coloring scheme: Hilbert-color and entropy-color. Figure 3 shows the count of misclassified apps in terms of false positives and false negatives. We prepend -H and -E to the model names to denote runs that used images generated with the Hilbert and entropy coloring schemes, respectively. We also include the geometric mean of the false negatives and positives across CNNs in order to compare the overall impact of the different coloring schemes. In this study, a false positive is when we misclassify a benign application as malware which is perceived as a nuisance to the user, but doesn't cause any harm to the device. A false negative is when we incorrectly classify a malicious application as benign. A closer look at Figure 3a suggests that Alexnet-H is the most conservative when used with a small dataset, yielding one false negative which is a little better than both LeNet-H and LeNet-E. We also note, that in general entropy-based images result in lower false negatives as shown by the geometric means Geo-H and Geo-E. However, the Hilbert-based images yield lower false positives. This trend persists even with the larger dataset as shown in Figure 3b. We observed that both LeNet-H and LeNet-E performed the best yielding the lowest misclassification counts and least number of false negatives as



(a) Small dataset (3K)



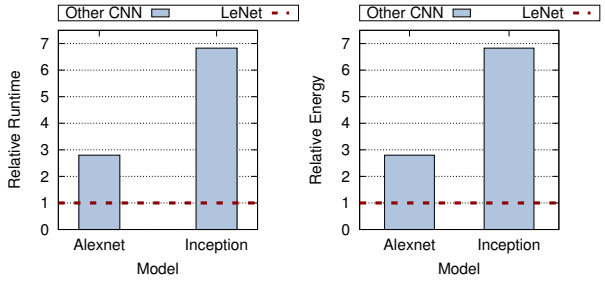
(b) Large dataset (15K)

Fig. 3: Summary of false positive and negative misclassifications for small (3K images) and large (15K images) datasets.

the datasize is scaled. Similarly, we can see from the geometric means that on average, the Hilbert-based images generate the least number of false positives across CNNs ranging from 0 – 3 false positives with a geometric mean under 0.5. The entropy-based images yield the least number of false negatives with a range of 0 – 2 across CNNs with a geometric mean of less than 0.3. Since false negatives represent the number of undetected malware samples in our dataset, we conclude that entropy-based images are better at capturing malicious features compared to Hilbert-based images. As such, we consider using the LeNet model with entropy-based images (LeNet-E) to be the most secure given its low false negative count.

C. Runtime and Energy Overhead

Figure 4a shows the runtime of different CNN models relative to LeNet. In general, Alexnet (8 layers) had the second least impact on performance. We observed that the runtime relative to the LeNet model ranges between 3x to 7x. Figure 4b shows the energy consumption of different CNN models. We observed a similar trend to the runtime performance with Alexnet being the second most efficient model. We observed that the energy relative to the LeNet model ranges between 3x



(a) Runtime (b) Energy

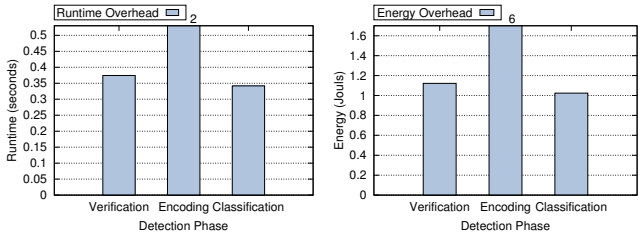
Fig. 4: Runtime and energy of different models relative LeNet.

to 7x. This is summarized in Figure 4b. Our results suggest that LeNet is the most suitable model for mobile systems.

Overall, our system has three main sources of runtime and energy overhead. These sources are the cost of performing the verification, encoding, and classification within our design. Figures 5a and 5b show a breakdown of the aforementioned overheads. In our design, the verification phase is executed every time an app is launched and represents the common case. We measured runtime and energy costs of 374 ms and 1.1 J. It is important to note that this overhead is confined to the startup cost of launching a given app. Our design, doesn't incur any overhead after the app has been launched. Also, having the verification phase allows us to keep the overhead to a minimum by obviating the execution of the remaining phases unless an app is installed or updated. The second source of overhead relates to encoding which entails transforming the app's `OAT` file into an image. This component marks the highest overhead in our design. In the case of Hilbert-based images, we measured runtime and energy costs of 2.5 s and 7.4 J. On the other hand, entropy-based images required 1.9 s and 5.6 J for runtime and energy, respectively. This correlates to a 25% improvement in both runtime and energy relative to the Hilbert-based approach. Although this cost is still higher than the verification phase (5x – 7x), we anticipate this phase to only be used for newly installed apps or any apps that undergo updates. The final source of overhead relates to the classification phase which involves classifying the image obtained from the encoding phase through a CNN. We used the LeNet model to measure the overhead since it performed the best. We measured runtime and energy costs of 342 ms and 1 J. Similar to the encoding phase, we anticipate this phase to only be used for newly installed apps or any apps that undergo updates. Therefore, in addition to LeNet-E being the most secure, we find this model to also be the most efficient in terms of runtime and energy.

D. Comparison to Other Machine Learning Designs

Algorithms such as Support Vector Machines (SVM) and Logistic Regression (LR) are less complex than CNNs, and therefore, more energy efficient. While using a design that employed opcodes within `OAT` files as features for SVM and LR [2], we observed classification runtimes of 23 ms and 19 ms, respectively. Similarly, we obtained 69 mJ (SVM) and 59 mJ (LR) for energy. However, despite the efficiency of these models, the encoding phase alone required 6 s and 18 J



(a) Runtime (b) Energy

Fig. 5: Breakdown by phase in terms of runtime (a) and energy (b) overheads for the LeNet model with entropy-based images.

for runtime and energy which translates to 3x more overhead relative to our end-to-end CNN-based solution. Unlike CNNs, encoding `OATs` for these algorithms are not amenable to harnessing the SIMD engine, and therefore, require more time to complete. Most importantly, such algorithms lack the ability to perform feature extraction, a critical task for effective malware detection. To illustrate this, we trained the SVM and LR models against 15K `OAT` datasets compiled for different platforms: an ARM `OAT` dataset (smartphone platform) and an x86 `OAT` dataset (chromebook platform). We observed a noticeable degradation in the detection rate when going from ARM to x86. With the ARM dataset, both models had detection rates of 98.5%. However, these rates dropped to 88.0% (SVM) and 96.1% (LR) when trained using the x86 dataset, suggesting that new features that are representative of x86 platforms had to be extracted. On the other hand, the LeNet-E model had detection rates of 98.5% and 99.0% for ARM and x86, respectively. This result shows the versatility of CNNs in autonomously extracting complex features and seamlessly adapting to different platforms.

VI. Conclusion

We propose a solution that harnesses visualization techniques for converting application instructions into images and explore their relevance to malware detection when combined with CNNs. We demonstrate that high accuracy can be achieved when combining a simple LeNet model with entropy-based images generated through space-filling curves.

References

[1] T. Hsien-De Huang and H.-Y. Kao, "R2-d2: Color-inspired convolutional neural network (cnn)-based android malware detections," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 2633–2642.

[2] N. Lachtar, D. Ibdah, and A. Bacha, "The case for native instructions in the detection of mobile ransomware," *IEEE Letters of the Computer Society*, vol. 2, no. 2, pp. 16–19, June 2019.

[3] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current android malware," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer, 2017, pp. 252–276.

[4] A. Cortesi, "A library for drawing space-filling curves like the hilbert curve." 2015, https://github.com/cortesi/scurve.

[5] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 2016, pp. 468–471.