RansomShield: A Visualization Approach to Defending Mobile Systems Against Ransomware

NADA LACHTAR, University of Michigan, USA
DUHA IBDAH, University of Michigan, USA
HAMZA KHAN, University of Michigan, USA
ANYS BACHA, University of Michigan, USA

The unprecedented growth in mobile systems has transformed the way we approach everyday computing. Unfortunately, the emergence of a sophisticated type of malware known as ransomware poses a great threat to consumers of this technology. Traditional research on mobile malware detection has focused on approaches that rely on analyzing bytecode for uncovering malicious apps. However, 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 variants released every year makes it difficult for solutions to efficiently scale their coverage.

To address these concerns, this work presents RansomShield, an energy efficient solution that leverages CNNs to detect ransomware. We evaluate CNN architectures that have been known to perform well on computer vision tasks and examine their suitability for ransomware detection. We show that systematically converting native instructions from Android apps into images using space-filling curve visualization techniques enable CNNs to reliably detect ransomware with high accuracy. We characterize the robustness of this approach across ARM and x86 architectures and demonstrate the effectiveness of this solution across heterogeneous platforms including smartphones and chromebooks. We evaluate the suitability of different models for mobile systems by comparing their energy demands using different platforms. In addition, we present a CNN introspection framework that determines the important features that are needed for ransomware detection. Finally, we evaluate the robustness of this solution against adversarial machine learning attacks using state-of-the-art Android malware dataset.

CCS Concepts: • **Security and Privacy → Intrusion/anomaly detection and malware mitigation**.

Additional Key Words and Phrases: Ransomware, malware, machine learning, convolutional neural networks, visualization, hilbert, entropy, mobile security, Android, intrusion detection system, instruction set architecture, energy efficiency.

ACM Reference Format:
Nada Lachtar, Duha Ibdah, Hamza Khan, and Anys Bacha. 2021. RansomShield: A Visualization Approach to Defending Mobile Systems Against Ransomware. In *ACM Transactions on Privacy and Security*. ACM, New York, NY, USA, 29 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The unprecedented growth in mobile systems has transformed the way we approach everyday computing. With more than 87% of the market share, the Android ecosystem has become an indispensable technology that enables the majority of today's digital devices. For instance, it was estimated that in 2019 Google Play averaged more than 230 million daily downloads from its app store, underscoring Android's uncontested dominance in the mobile world. Unfortunately, the popularity of this platform has garnered interest from cybercriminals, spurring an onslaught of creative malware that harnesses this technology for profit. To this end, mobile consumers have been fraught with sophisticated ransomware campaigns that aim to maliciously encrypt their data and lock them out of their devices in return for ransom money. Recently, Trend Micro's Mobile App Reputation Service (MARS) observed a 415% surge in new ransomware [65] relative to prior years after it had discovered nearly half of a million ransomware samples that are tailored for mobile devices. This trend underscores the need for researchers to devise solutions that can safeguard mobile systems against the insurgence of such malware.

A significant body of research has explored various techniques to defend against computer and mobile ransomware. Work by Kharraz et al. [31] investigated the use of filesystem patterns for detecting ransomware on a computer system. The solution involves the generation of a temporary artificial user environment that is used to screen running applications as a way of safely detecting ransomware behavior. However, this approach is vulnerable to ransomware that block their encryption activity while running within a virtual environment. Other solutions include [33], which tracks cryptographic keys that are generated by the operating system through the use of a key vault. The saved keys are extracted from the vault in the event that malicious behavior is detected and used to decrypt any affected files. Unfortunately, the aforementioned system can be bypassed by ransomware that directly embed encryption algorithms within the application. Thus, bypassing the use of any crypto services available through the system that would otherwise trigger the detection process. Additional techniques proposed the use of standard hardware performance counters for detecting malicious activity [15, 50]. However, such techniques have been shown to be unreliable in predicting malicious activity [78]. Additional work by Chen et al. [11] explored the use of finger movements on smartphones for inferring the presence of ransomware activity on mobile systems. They treat finger movements as an indicator that the user approves the application's activity as opposed to some unsupervised encrypting process that is taking place without the user's knowledge. However, the proposed solution suffers from the inability to detect locker-based ransomware which does not involve encryption. Finally, other recovery techniques, such as FlashGuard [24] leverage out-of-place writes used to mitigate long erase latencies associated with flash drives as a mechanism for recovering previously encrypted files. However, ransomware can generate large temporary files that can result in data being lost if the recovery process is not activated immediately. In addition, the solution has limitations when the amount of free storage is low which is common in mobile devices. Furthermore, other work [38, 39, 56] examined static analysis techniques for detecting general malware. However, the aforementioned solutions rely on either XML-based permission files or Dalvik bytecode. Unfortunately, cybercriminals are now obfuscating malicious content directly in native form in order to evade such detection mechanisms [69]. Such obfuscation techniques underscore the need to augment mobile systems with the ability to detect malware at the native instruction level.

In this work, we demonstrate the effectiveness of combining CNNs with native instructions for detecting mobile ransomware. We show that our approach can achieve a near ideal accuracy of 99.7% when converting the native instructions of Android apps into images through space-filling curves. We demonstrate that our work scales beyond ARM-based smartphones to offer ransomware detection on other mobile platforms that utilize a completely different instruction set. To this end, we characterize different CNN models and show that our method is effective in detecting mobile ransomware across RISC and CISC architectures using two of the most popular instruction sets: ARM and x86. Thus, having the ability to detect ransomware across heterogeneous devices ranging from smartphones to chromebooks. In addition, we evaluate the suitability of our solution for mobile platforms by characterizing the energy demands of different models and propose a design that introduces minimal energy and application startup costs for the common case. Finally, we evaluate the robustness of our approach against adversarial machine learning attacks.

Overall, this paper makes the following contributions:

- Presents a novel approach that demonstrates the effectiveness of space-filling curve visualization techniques in converting mobile apps into images and its relevance to ransomware detection when used with CNNs.
- Evaluates the efficacy of combining CNNs with native instructions that span the ARM and x86 instruction sets and the relevance of this approach in detecting mobile ransomware. Thus, providing detection coverage across heterogeneous mobile devices including smartphones and chromebooks.
- Discusses the impact of adversarial machine learning attacks through newly generated ransomware derived from Generative Adversarial Networks and evaluates defenses that can be applied to mitigate the effect of such attacks.
- Characterizes the energy efficiency and accuracy of different CNN models and evaluates their suitability across a range of ARM and x86 platforms.
- Proposes a solution that offsets the computational demands of CNNs on mobile systems by optimizing the common use case to incur minimal application startup and energy costs.
- Implements a framework that performs CNN introspection designed to extract important features that are relevant for ransomware detection across both ARM and x86 instruction sets.

The rest of this paper is organized as follows: Section 2 provides background information. Section 3 characterizes the detection of ransomware in native code form using state-of-the-art commercial anti-malware. Sections 5 and 4 present the design and threat model of the proposed system. Section 6 discusses the methodology for evaluating the approach. Section 7 presents the results of our evaluation. Section 8 discusses the overall solution and its limitations. Section 9 details related work; and Section 10 concludes.

2 BACKGROUND

2.1 Native Execution in Android Platforms

Android is prolific software that is designed to promote application portability and execution performance across a broad spectrum of mobile systems. Applications on this platform are made up of a collection of resources that are bundled in the form of an Android Package Manager (APK). Such APKs contain descriptions of the system resources they require, as well as, an executable that contains Java bytecode. This bytecode is referred to as a Dalvik executable (`dex` file).

A core component for running applications on Android is the Android Runtime system (ART). Unlike the original Dalvik Virtual Machine (DVM) design that leveraged just-in-time (JIT) compilation for launching Android apps, ART employs ahead-of-time compilation. Therefore, instead of repeatedly translating Java bytecode every time a user launches a given application, ART generates a one time binary that can be executed natively onto the device. The same binary is used for subsequent launches of the application. ART accomplishes this through a `dex2oat` module that extracts the `dex` file from the app's APK and converts it into an `OAT` binary (ahead-of-time file). This approach significantly speeds up application performance compared to the JIT-based approach. Our solution makes use of such `OAT` files as the basis for generating images that can be consumed by our trained CNNs.

2.2 Space-filling Curves

A space-filling curve is a function that has the ability to map sets of data into a multi-dimensional hypercube[18, 19, 59]. 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. Multiple bodies of work have explored space-filling curves. This includes the Peano [51], Z [68], and Hilbert curves [21]. However, the Hilbert curve has proven to outperform other methods due to its ability to preserve spatial information [16, 45]. This property is important to this study since it preserves the linear ordering of instruction opcodes present in apps after they are transformed into images. Thus, opcodes that occur in a given sequence will correlate to pixels that are also adjacent to each other in the generated image. This study makes use of the Hilbert space-filling curve for mapping instructions associated with a given app into pixel locations on a 2D image [14]. Leveraging this transformation enables our design to harness CNNs, a powerful class of networks that can recognize complex patterns present in the form of 2D images with high accuracy.

More formally, the Hilbert curve entails recursively refining a 2×2 lattice into sub-lattices that undergo systematic rotations to ensure that the entry and exit points of the sub-lattices remain adjacent to one another. This can be interpreted as an n dimensional space consisting of 2^n vertices where each vertex is represented as an n-bit integer $b = [\beta_{n-1}, ...\beta_0]$ such that $b \in Z_{2^n}$ and each bit β_i corresponds to the position of the vertex along a given dimension i. The ordering of the different vertices is enforced through the use of gray codes with the i^{th} gray code integer conforming to equation (1). We define the curve's entry point $e(i)$ into a given i^{th} sub-hypercube through equation (2) to enforce adjacency of the curve's exit point in one sub-hypercube to the curve's entry point in the next sub-hypercube.

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 RGB colors. 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 described in equation (3) over a window of n pixels to define the intensity of the aforementioned components. Although, not as granular, visualizing data through entropy has the benefit of highlighting encrypted sections that is often an indicator of malicious content. An example of this is shown in Figure 1b.

$$gc(i) = i \oplus (i >> 1) \tag{1}$$

$$e(i) = \begin{cases} 0, & i = 0 \\ gc(2\lfloor \frac{i-1}{2} \rfloor), & 0 < i \leq 2^n - 1 \end{cases} \tag{2}$$

$$H(X) = -\sum_{i=0}^{n} p(x_i) logp(x_i) \tag{3}$$

2.3 Convolutional Neural Networks

Convolutional neural networks (CNN) have demonstrated significant promise in tackling a multitude of computer vision problems [12, 23, 29, 30, 52, 53, 72, 79]. A convolutional neural network consists of *input*, *output*, and *hidden* layers. As shown in Figure 2, the hidden layers are usually composed of a series of convolution, normalization, activation, pooling, and fully-connected layers.

Convolution Layer. Represents a central component of the CNN design. It consists of kernels (filters) that it learns during the training phase. The kernels are convolved with the input image in strides over the spatial dimension to

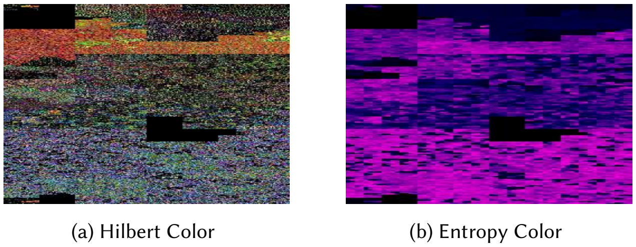


Fig. 1. Example of native code for a ransomware app transformed into images using Hilbert and entropy-based colors.

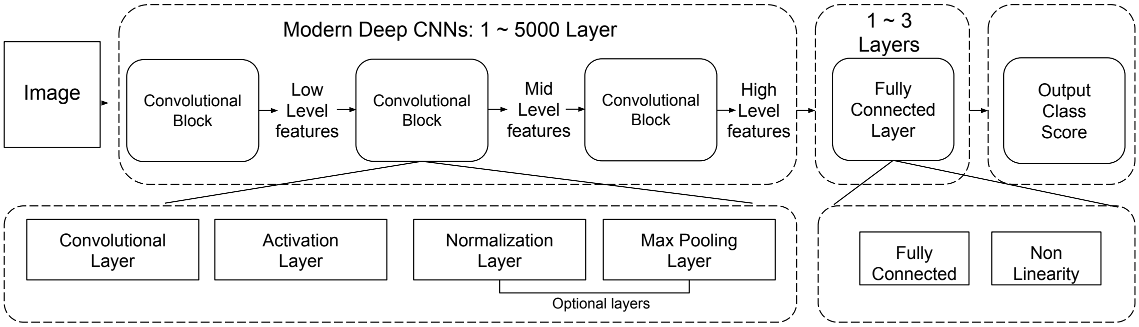


Fig. 2. Overview of the typical components within a convolution neural network.

create 2D feature maps. The feature maps are obtained by performing dot product operations between the weights of the kernel and small regions of the input that is known as the receptive field. This approach enables each layer to generate successively higher levels of abstraction for the input data.

Batch Normalization Layer. This layer minimizes the internal covariate shift by normalizing the different features of the computed feature maps.

Activation Layer. CNNs attempt to approximate mathematical functions that are non-linear in nature. Therefore, to increase non-linearity within the network, an element-wise activation function such as rectified-linear unit (e.g. *ReLu*) is applied to the input data. In the case of the *ReLu*, negative values are removed from feature maps by setting such values to zero.

Pooling Layer (Max Pooling). This layer serves the purpose of reducing the complexity of the remaining layers by decreasing the number of hyperparameters, amount of computation in the network, and required memory footprint. The max pooling layer achieves this by down sampling the data it receives along the spatial dimensions.

Fully-connected Layer. This layer serves the purpose of inferring high level features that are beyond edges, curves, and shapes. The neurons within this layer receive their inputs from the activation units of the preceding layer. The output produced by each neuron is computed as the weighted sum of all its inputs. In order to generate a probability distribution that could be used for predicting different classes, the CNN's final fully-connected layer is typically connected to softmax and argmax layers.

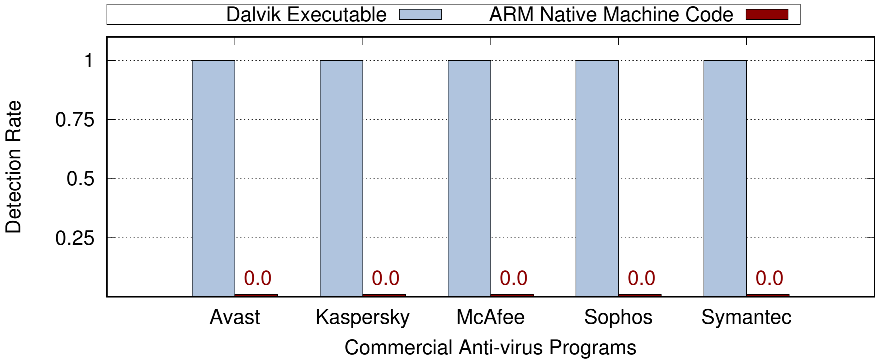


Fig. 3. Detection rates of ransomware from three families when using high-level Dalvik bytecode (dex file) vs. the same dex file converted into native ARM machine code.

3 THE CASE FOR NATIVE CODE SOLUTIONS

Various software vendors provide anti-malware solutions that are meant to address the increasing landscape of threats induced by mobile malware. In order to characterize the effectiveness of the aforementioned solutions against mobile ransomware, we conducted experiments that involved five leading commercial anti-malware solutions available through VirusTotal [67]: Avast, Kasperky, McAfee, Sophos, and Symantec.

We selected 60 ransomware samples from three different families that span device locking and storage encryption capabilities. More specifically, we selected the Aples, Simple Locker, and Jisut ransomware families. The Aples family represents ransomware with device locking capability that pretends to be an anti-virus application as a way of luring users into running the application with the privileges the ransomware needs [40, 70]. After the ransomware is launched, it proceeds by locking the phone screen, then employs scare tactics to extort the user to pay a ransom. Simple Locker is another device locking ransomware family that follows a similar approach. In addition to device locking capability, it performs storage encryption through an AES algorithm that consumes a hardcoded key. The last family of ransomware that we tested is known as Jisut. This ransomware operates by redirecting key press events to a meaningless action in order to keep the device locked.

In order to assess the ability of the selected anti-malware programs to detect ransomware in native machine code, we conducted a comparison test that entailed using files from the aforementioned ransomware families in two forms: Dalvik code (dex file) that uses Java bytecode and the same binary, but in ARM native machine code (OAT file). Figure 3 illustrates the results of this experiment after running 60 ransomware applications in the two executable forms: dex and OAT. In the first case, we can see that the detection rate of ransomware is uniform across all anti-malware programs for all three families of ransomware. We observe that all anti-malware programs are able to detect all the ransomware samples with 100% accuracy when presented with executables that are in high-level Dalvik bytecode. On the other hand, none of the anti-malware programs are able to detect the same exact ransomware dataset when presented in native ARM machine code. This experiment underscores the vulnerability of today's mobile systems to attacks that can subvert commercial anti-malware programs by embedding malicious code directly in native form. Therefore, the need for mobile anti-malware solutions that can analyze malicious code directly in its native form.

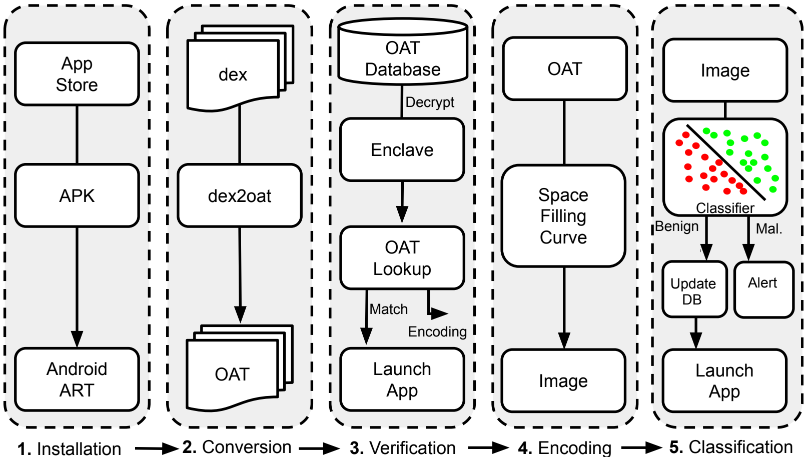


Fig. 4. Overview of the RansomShield system.

4 THREAT MODEL

In this section we detail the threat model and the assumptions we make about the attackers and how it applies to our design. We make the assumption that ransomware is installed on the machine through the proper mechanisms already supported by the operating system. For instance, in the case of Android, this process entails downloading a .apk file from a mobile application provider such as Google Play Store or any other third party store. We assume no privilege attacks occur on the device and that the operating system and underlying firmware of the device are up to date and haven't been compromised. In addition, we assume the presence of obfuscated code that relies on repackaging attacks, payload encryption, and malicious payload embedded directly in native form [10, 42, 43, 69]. Furthermore, we make the assumption that the Ransomware developer is aware of the detection infrastructure, with the possibility of having access to the trained CNN for launching an adversarial attack. In the case of storage encryption ransomware, we make the assumption that such ransomware can use encryption algorithms that are available through crypto services provided through the OS or directly embedded within the application. We consider the fact that malicious applications can pose as benign apps that can perform delayed updates as a way of fetching malicious content. In order to thwart such update attacks, our design verifies OAT files on every launch. Thus updated OATs will be evaluated before they are loaded in memory and executed. Our approach of analyzing OAT files on every load of an application introduces overhead in both startup time and energy. However, we show in section 7 that the impact for the common case is minimal. Gaining physical access to a device presents another vector of attack. This approach assumes that an attacker has the appropriate knowledge about the device and resources similar to Graykey [17] in order to bypass existing security measures. However, we believe such attacks defeat the purpose of using ransomware that is primarily focused on spreading to as many users as possible with the goal of making profit through attacking users in large volumes. As such, we do not envision physical attacks being a practical approach in this case.

5 THE RANSOMSHIELD SYSTEM

RansomShield is designed to ensure seamless interaction between mobile users and their applications. A primary objective of RansomShield is to defend against obfuscation techniques that involve injecting native code into repackaged mobile applications. To this end, our design carries out its detection based on the native opcodes of the application. The design also defends against seemingly benign applications that can fetch and install malicious payloads from a command and control (C&C) server. RansomShield thwarts such attacks by performing the detection directly on the device. This detection occurs immediately before the application's binary is dispatched for execution.

Our ransomware detection system is designed to integrate directly into the Android Runtime environment (ART) of the device. The detection process is shown in Figure 4. This begins with step ❶ that corresponds to a user installing a desired application using an APK bundle from the app store. After the user attempts to install the app onto the device, ART extracts the various resources associated with the bundle including `classes.dex` which represents the executable in Dalvik bytecode form. Upon the extraction of the necessary app resources by ART including any obfuscated code, an `OAT` image is generated through the `dex2oat` module to produce an ELF formatted file that contains all the native instructions that will be executed on the device. This corresponds to the conversion phase shown in step ❷.

Upon obtaining an `OAT` image, our design proceeds to the verification phase (step ❸) in order to determine if the app, in its native form, is safe to launch. To achieve this, our design makes use of a small `OAT` database that consists of all the `OAT` names installed on the system and their corresponding hashes. On average the database contains information for approximately 100 apps, which is the number of apps the average user installs [5]. We maintain the most recent hash for each `OAT` image within the database using the `SHA512` algorithm. For added security, the `OAT` database is encrypted using an `AES 256` key that is stored in an enclave. In order to optimize the verification phase and reduce its overhead, we decrypt the database during the boot process of the system. 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 since this implies that the app hasn't changed after its last validated run. Otherwise, we proceed to the encoding phase.

In the encoding phase (step ❹), the design transforms the `OAT` file into a pixelated image that can be consumed by a CNN. We use space-filling curve visualization techniques to acheive this tranformation. This image is used as an input into the classification phase (step ❺). The design makes use of a pre-trained CNN that performs the final classification of the app. 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 to indicate ransomware detection.

In most cases, a user attempting to launch an app will result in step ❸ being activated. This step represents the common case. Steps ❹ ❺ are only used if the user installs a new app or updates an existing app.

6 EXPERIMENTAL FRAMEWORK

6.1 Training Framework

Training experiments were conducted on a system equipped with two Intel Xeon Gold 6152 processors (Skylake architecture), an Nvidia Tesla V100 GPU (Volta architecture), and 768GB of main memory. We re-purposed four CNN architectures that are popular in the field of computer vision to perform ransomware detection. This included: LeNet [36], Alexnet [34], InceptionV3 [60], and VGG-16 [54]. All of the aforementioned CNNs were trained on the V100 GPU with the exception of VGG which required more memory and had to be trained on the main system due to its larger number of hyper-parameters (138 million).

To generate `OAT` files for the APK packages within our dataset, we created a framework based on the Android Open Source Project (AOSP) 6.0.1 release and built it with the `userdebug` option for ARMv7. The framework was used to

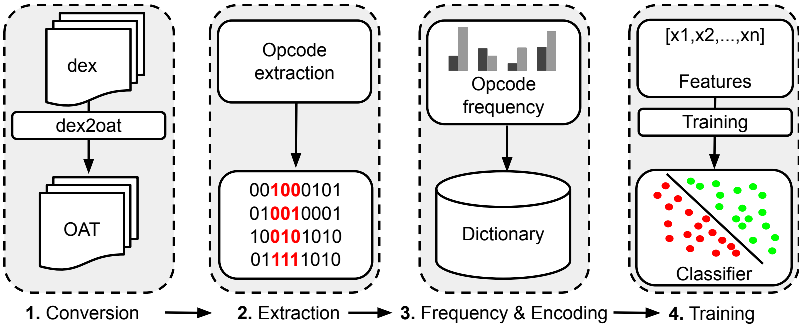


Fig. 5. Methodology for training the reference deep neural network (DNN).

convert the app executables from Dalvik bytecode (`classes.dex`) to Android `OAT` files that conform to a standard Executable and Linkable Format (ELF). We used TensorFlow 1.12 [62] with Python 3.6 for training the CNN architectures. We used [14] to convert Android `OAT` files into 256 × 1024 images. More specifically, we used the Hilbert space-filling curve for mapping instructions into pixel locations within a 2D image. Furthermore, we used two coloring schemes to generate the RGB values for each pixel location: Hilbert and entropy. Hence, we created two sets of data images, one for each coloring scheme. The images were then grouped into TFRecords to facilitate portability across the three CNN architectures within TensorFlow and ensure consistent ordering of the images during the training and testing phases for all three models.

In addition, we evaluated a plain deep neural network (DNN) that we use to compare against our CNN-based design. The DNN architecture consisted of seven hidden layers (same number of layers as LeNet) and 1024 nodes per layer. Since a DNN can't consume images, we devised another approach to enable the evaluation of this architecture. The steps of this approach are summarized in Figure 5. Similar to our CNN-based design, step ❶ converts the `dex` files into the corresponding `OAT`s. Upon completion of the conversion process, native instruction opcodes are extracted from the `text segment` of each `OAT` file (step ❷) with the frequency information of each opcode within the `text segment` computed (step ❸). A dictionary is then used to encode the extracted opcodes and their frequency information into a feature set that the DNN uses for training (step ❹).

All of the models were trained with the Adaptive Moment Estimation optimizer (Adam) after initializing the weights to randomly generated numbers. In addition, we tested different configurations in order to determine the optimal settings. This included using 64 and 128 batch sizes. Furthermore, we varied the number of epochs per batch size configuration across the following settings: 50, and 100 epochs.

6.2 Experimental Platforms

We selected three configurations in order to characterize our design and its suitability across a range of ARM and x86 mobile systems that span low-end, mid-range, and high-end platforms. For the low-end platform, we used a Raspberry PI 3 B+. This configuration is similar to a Nexus 5X smartphone [71]. We attached a Zymbit security module [63] to the Raspberry PI 3 configuration for key storage support. The Zymbit module was used for developing a proof of concept for the verification phase of our design. The second platform included a mid-range configuration that used a Core i3-based x86 system. The mid-range platform was used to represent a low-end x86 chromebook and is similar to the Lenovo C630 chromebook [28]. The last platform included a high-end configuration that used a Core i7-based x86 system to represent the high-end chromebook space. This configuration is similar to the Google Pixelbook chromebook [28]. We used a Trusted Platform Module (TPM) 1.2 for key storage support as part of developing a proof of concept for the verification phase of our design for the x86 platforms. The main hardware components for each platform is summarized in Table 1. We used the aforementioned platforms listed in Table 1 to collect the runtime and energy information. In the case of energy measurements, we used the Running Average Power Limit (RAPL) counters [26] along with the `perf event` interface for measuring energy on the Intel platforms. For the ARM platform, we used the Klein Tools ET920 USB digital meter [64].

6.3 Dataset

We conducted experiments using 2063 ransomware samples available in [69] that included both device locking and storage encryption capabilities. Table 2 summarizes the different ransomware families that we employed in this dataset and their respective capabilities. All of the samples that we tested involved communication with a remote command and control (C&C) server that could be triggered through system events, such as `Boot`, `Call`, and `Sys`. Besides exchanging encryption keys in the case of cryptographic ransomware, this communication is leveraged to fetch and install malicious payloads. In addition to ransomware samples available in [69], we used 13022 samples from AndroZoo [1] as a baseline for benign Android apps. The aforementioned samples were combined to form a dataset that we trained and tested with, which consisted of a total of 15K image samples. We dedicated 80% of the dataset for training, 10% for validation, and 10% for testing.

In terms of images, we used the Hilbert curve for transforming instruction opcodes into pixel locations (x and y coordinates of the image). Each pixel value (RGB value) within an image was generated using two coloring schemes: Hilbert-color and entropy-color. Images with the Hilbert coloring scheme used an order 8 Hilbert curve for mapping instruction opcodes into RGB pixel values. On the other hand, images with the entropy coloring scheme leveraged the Shannon entropy described in equation (3) for defining the RGB values. As result, our experiments consisted of using a dataset for each image type in order to evaluate the effect of different coloring schemes. Therefore, our study used: a 15K dataset for Hilbert images and a 15K dataset for entropy images.

7 EVALUATION

This section examines different quality metrics on how our solution performs across heterogeneous platforms that use the ARM and x86 instruction set architectures (ISA). We analyze different CNN models and their robustness against adversarial machine learning attacks. We also characterize the runtime and energy overheads associated with our system.

Platform	Hardware
Low-end (Cortex-A53)	*CPU: ARMv8 ISA, 1.4GHz, 4 cores* *Memory: 1GB, LPDDR2*
Mid-range (Core i3 8100)	*CPU: x86 ISA, 3.6GHz, 2 cores* *Memory: 8GB, DDR4*
High-end (Core i7 7700K)	*CPU: x86 ISA, 4.2GHz, 4 cores* *Memory: 16GB, DDR4*

Table 1. Summary of platforms tested that include low-end (smartphone), mid-range (low-end chromebook), and high-end (high-end chromebook) configurations.

Family	Samples	File Encryption	Screen Lock	C&C
Aples	21	-	✔	✔
Fusob	1282	✔	✔	✔
Jisut	560	✔	✔	✔
Koler	72	✔	✔	✔
Roop	48	-	✔	✔
Simple Locker	180	✔	✔	✔

Table 2. Summary of the ransomware families used in the evaluation and their capabilities.

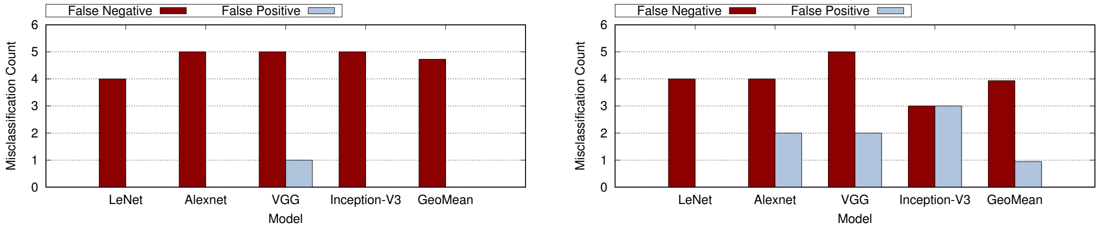
7.1 Model Analysis on Smartphone Platforms

We examined multiple CNN models in the context of smartphones that consume the ARM instruction set. We evaluated each model against commonly used quality metrics that include: accuracy, true positive rate (TPR), false positive rate (FPR), precision, F-score, and area under the curve (AUC). We also evaluated the impact of the Hilbert and entropy coloring schemes on the aforementioned quality metrics.

7.1.1 Hilbert Coloring Scheme on Smartphones. Table 3 summarizes the quality metrics for our smartphone platform. The results are based on the dataset that maps the ARM instructions within apps into pixelated images that use the Hilbert coloring scheme. Overall, we observed that all the CNN models considered in this study performed significantly better than a plain DNN across all metrics. For instance, all the CNN models were able to achieve detection rates and F-scores that are above 97.6%. This is in comparison to the DNN model which scored 80.1% or less for the same metrics. Furthermore, we observed that our system performed well when using the simplest CNN model for classification. The LeNet-based model which only has 7 layers offered the best performance across all the metrics in Table 3, followed by designs with Alexnet (8 layers) and InceptionV3 (48 layers) that had equivalent performances, then VGG (16 layers). The LeNet model had a 0.0% false positive rate with detection rate (TPR) and F-score metrics of 98.1% and 99.0%, respectively. The remaining CNN models which includes Alexnet, InceptionV3 (IV3), and VGG, had similar results that were near the average TPR (97.7%) and F-score (98.8%). However, VGG exhibited a slightly higher false positive rate that in turn translated into relatively lower F-score and precision metrics.

To better understand the performance of the studied models, we examined the amount of false positives and negatives each model generated under the Hilbert coloring scheme. Figure 6a provides a breakdown of the misclassified data into false positives and false negatives. Figure 6a suggests that LeNet is the most conservative with false negatives, yielding four false negatives. These false negatives correspond to one, two, and one misclassifications from the Fusob, Jisut, and Kohler families, respectively. This is slightly better than the remaining models which had five false negatives, and as a result, lower F-scores. This also implies that LeNet is better at detecting ransomware than the remaining models. On the other hand, all of the CNNs performed well in terms of false positives. For instance, LeNet, Alexnet, and InceptionV3 didn’t generate any false positives while VGG yielded a single false positive. Overall, we note that LeNet produced the least number of misclassifictions with the remaining models having close performances. Most importantly, we observe that LeNet demonstrated robustness against obfuscated attacks that embed malicious code directly in native form. LeNet miscalssified only two samples from the Jisut family that has been shown to embed malicious instructions directly in native form alongside standard Dalvik byte code [42].

7.1.2 Entropy Coloring Scheme on Smartphones. Table 4 summarizes the quality metrics for the same platform and ransomware dataset while using the entropy coloring scheme. Overall, we recorded similar results relative to what



(a) Hilbert coloring scheme

(b) Entropy coloring scheme

Fig. 6. False positive and negative misclassifications using the ARM instruction set with (a) Hilbert coloring scheme, and (b) Entropy coloring scheme.

was observed for the Hilbert coloring scheme. For instance, all the CNN models were able to achieve detection rates and F-scores that are above 97.6%. In addition, we observe that all CNNs exhibited similar performances with LeNet having a slight edge across most metrics. Figure 6b shows a breakdown of the misclassification data in terms of false positives and negatives. We observed a geometric mean (GeoMean) of 1 false positive and 4 false negatives across CNNs when using the entropy coloring scheme. Although LeNet had the lowest misclassification count, we observe that InceptionV3 is more conservative than LeNet in terms of false negatives. For instance, InceptionV3 generated a total of 3 false negatives while LeNet had a total of 4 misclassifications of the same type. On the other hand, LeNet had no false positives while InceptionV3 had 3. Furthermore, we observe that unlike other models, LeNet produced similar results irrespective of the image type including the ability to detect obfuscated ransomware represented by the Jisut family.

In general, our results suggest that the Hilbert and entropy coloring schemes are equally capable in detecting ransomware on smartphones that rely on the ARM instruction set. When comparing the two schemes, we observed that, on average, entropy-based images tended to yield lower false negative rates while Hilbert-based images were slightly better at producing lower false positive rates. For instance, we observed a geometric mean (GeoMean) of 0.95 and 3.9 false positives and negatives, respectively, across CNNs for the entropy dataset. This is in comparison to the Hilbert coloring scheme which yielded nearly no false positives and 4.7 false negatives. We attribute the advantage the entropy coloring scheme has over its Hilbert-based counterpart in detecting ransomware, to its ability to recognize the presence of encrypted sections in OAT images. This is because malicious apps tend to include information, such as the C&C server they communicate with and payment details in encrypted form [69]. These encrypted sections result in image segments that have high entropy that the trained models look for as features. As such, the entropy coloring scheme is slightly better at detecting malicious samples compared to the Hilbert-based approach. On the other hand, we find that the Hilbert images are slightly better at classifying benign samples. In general, unlike the entropy-based images which uses a limited number of colors to represent entropy values, Hilbert images utilize the full range of colors from the RGB spectrum. We believe this fine-grained approach, gives Hilbert-based images an advantage in being able to accurately represent a wide range of benign applications that utilize a different collection of instructions. However, we note the difference between the aforementioned schemes is minor. As such, additional factors including resiliency to adversarial machine learning attacks and energy efficiency must be considered as part of determining the optimal solution for mobile platforms. We also note that LeNet outperformed other CNNs across most quality metrics despite being the simplest CNN model in the batch.

Model	Accuracy	TPR	FPR	Precision	F-score	AUC
LeNet	0.997	0.981	0	1	0.990	0.990
Alexnet	0.997	0.976	0	1	0.988	0.988
VGG	0.995	0.976	0.024	0.996	0.985	0.976
IV3	0.997	0.976	0	1	0.988	0.988
DNN	0.934	0.801	0.199	0.737	0.767	0.801

Table 3. Summary of quality metrics for ARM-based CNN models using the Hilbert-color dataset.

7.2 Model Scaling to Other Platforms

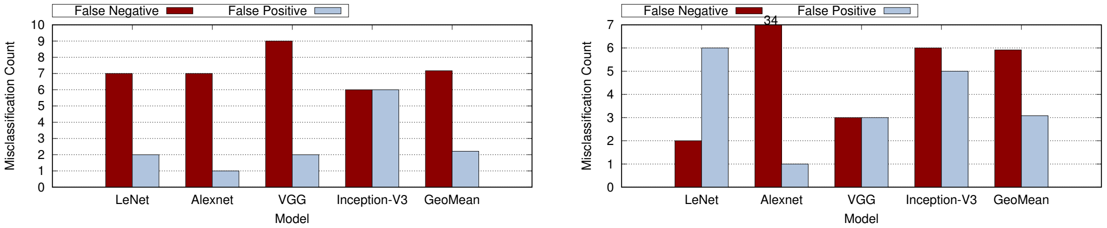
To understand how our solution scales beyond smartphones, we examined the performance of the previously discussed CNNs in the context of chromebook systems that are designed to consume the x86 instruction set.

7.2.1 Hilbert Coloring Scheme on Chromebooks. Table 5 summarizes the quality metrics for the x86 chromebook platform. All the models achieved F-scores that were above 97.1%. In addition, InceptionV3 offered the best detection rate (97.1%) while Alexnet had the best F-score (97.8%). LeNet also produced competitive results by being within 0.5% of all the best metrics under this coloring scheme. In general, we noticed a slight decrease (<1%) across the different metrics when using CNNs with the x86 dataset relative to their ARM counterparts. We believe this is attributed to the number of unique instruction opcodes that apps use in each instruction set. For instance, a total of 5014 unique opcodes were used for apps running on the ARM platform (smartphone) whereas the x86 platform (chromebooks) only used 841 unique opcodes. We believe this finer granularity in the ARM opcodes used within apps gave CNNs a slight advantage during the training process over their x86 counterparts.

Figure 7a illustrates the misclassification results for the x86 Hilbert coloring scheme. We observed that InceptionV3 had the least number of false negatives followed by LeNet and Alexnet. On the other hand, Alexnet, offered the least number of false positives followed by LeNet and VGG. Overall, InceptionV3 was the most effective in terms of false negatives with LeNet being a close second. This is an important metric to consider given that it is directly concerned with the detection of ransomware, as opposed to false positives which reflect how often benign apps are falsely classified as ransomware.

7.2.2 Entropy Coloring Scheme on Chromebooks. Table 6 summarizes the quality metrics for the same x86-based datasets when using the entropy coloring scheme. Overall, we observed a lower average of false negatives in the results relative to what we recorded for the Hilbert coloring scheme. This is evident in Figure 7b which provides a breakdown of the misclassifications across the different x86-based models. We observe that VGG had the least number of misclassifications. It yielded a total of 6 misclassifications compared to LeNet which had 8. However, LeNet was the most conservative in terms of false negatives. It had 2 false negatives while VGG had 3. On the other hand, Alexnet performed the worst by yielding a total of 35 misclassifications with 34 of them being false negatives. This in turn had a direct impact on Alexnet's TPR which was 83.5%. On average, we recorded 3.1 false positives and 5.9 false negatives across CNNs.

Overall, our results suggest that combining LeNet with the entropy coloring scheme is the most secure for x86. It yields the least amount of false negatives, and therefore, the most capable in terms of ransomware detection. On the other hand, combining VGG with the entropy coloring scheme provides the best balanced performance on x86 systems, yielding 3 false positives and 3 false negatives while using the entropy coloring scheme. Therefore, the user would get less notifications of ransomware being falsely detected while using VGG.



(a) Hilbert coloring scheme

(b) Entropy coloring scheme

Fig. 7. False positive and negative misclassifications using the x86 instruction set with (a) Hilbert coloring scheme, and (b) Entropy coloring scheme.

Model	Accuracy	TPR	FPR	Precision	F-Score	AUC
LeNet	0.997	0.980	0	1	0.990	0.990
Alexnet	0.997	0.980	0.002	0.990	0.985	0.989
VGG	0.995	0.976	0.002	0.990	0.985	0.987
IV3	0.996	0.985	0.002	0.985	0.985	0.992
DNN	0.934	0.801	0.199	0.737	0.767	0.801

Table 4. Summary of quality metrics for ARM-based CNN models using the Entropy-color dataset.

Model	Accuracy	TPR	FPR	Precision	F-score	AUC
LeNet	0.994	0.966	0.002	0.990	0.978	0.982
Alexnet	0.994	0.966	0.0007	0.995	0.980	0.983
VGG	0.993	0.956	0.002	0.990	0.973	0.977
IV3	0.992	0.971	0.005	0.971	0.971	0.983
DNN	0.931	0.796	0.204	0.726	0.759	0.796

Table 5. Summary of quality metrics for x86-based CNN models using the Hilbert-color dataset.

Model	Accuracy	TPR	FPR	Precision	F-score	AUC
LeNet	0.995	0.990	0.005	0.971	0.981	0.992
Alexnet	0.978	0.835	0.001	0.971	0.981	0.917
VGG	0.997	0.971	0.004	0.976	0.973	0.991
IV3	0.994	0.971	0.004	0.976	0.973	0.984
DNN	0.931	0.796	0.204	0.726	0.759	0.796

Table 6. Summary of quality metrics for x86-based CNN models using the Entropy-color dataset.

7.3 Adversarial Machine Learning

Research has shown that adding small perturbations to input are sufficient for an attacker to misclassify the input with high confidence. As such, an attacker can fool a detection system by carefully positioning benign instructions into ransomware.

We evaluate the impact of adversarial machine learning (AML) attacks by generating a new ransomware dataset that we use for testing our solution. Each image in this new dataset consists of benign instructions that are carefully embedded into ransomware images in the form of adversarial noise. We rely on two optimization procedures for generating the adversarial noise. We first optimize the original CNN model to classify the original input using the cross entropy loss function shown in (4). We then optimize the adversarial noise to mimic the benign class. We accomplish this through a weighted $l2$ loss function (5) that is combined with the original classifier's loss function to form the total loss function shown in (6). More specifically, y represents the actual class value, p is the predicted value, and n is the number of samples.

$$c(y,p) = -\sum_{n=0}^{n} y_n log(p_n) \tag{4}$$

$$l2(y,p) = \sum_{n=0}^{n} (y_n - p_n)^2 \tag{5}$$

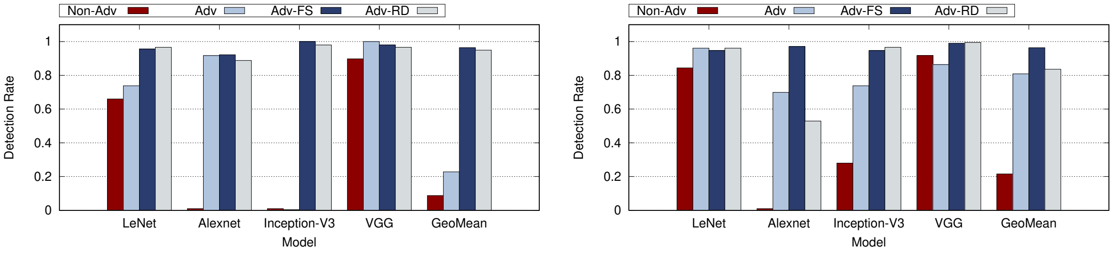
$$t(c,l2) = c + w \times l2 \tag{6}$$

Upon generating the new AML ransomware test set through the weighted loss function, we tested it against our previously trained models discussed in sections 7.1 and 7.2. We refer to these models as non-adversarially aware (*Non-Adv*) due to their training being limited to the original dataset that didn't contain any AML training samples. We then evaluated the ability of each CNN model to correctly classify our AML test set after making them adversarially-aware through different mechanisms. Table 7 summarizes the different approaches we used for hardening our CNN models against AML attacks. The first AML approach consisted of simply re-training each CNN model with a new set of AML examples. We refer to this configuration as adversarially-aware (*Adv*). To support this configuration, we used a Generative Adversarial Network (GAN) to create an AML training set that consisted of 40K adversarial samples. The aforementioned training samples were independently generated through our GAN model and did not rely on the loss function described in equations (4) – (6) that we dedicated for the AML test set. In other words, the AML test set we used to evaluate the robustness of our models against AML attacks relied on equations (4) – (6) for producing the test samples. On the other hand, the AML training set that was used to make the models adversarially aware relied on the GAN to produce the training samples. Using a GAN allowed us to expand our training set by introducing new ransomware variants that were not previously considered in the original dataset.

In addition to the *Adv* configuration, we augmented each adversarially aware CNN model with either Feature Squeezing [74] or Randomized Discretization [77] techniques to further harden them against AML attacks. Feature Squeezing (FS) aims to reduce the color space of each pixel. We accomplished this by reducing the number of bits from 24 to 8 for each pixel in the training set. Randomized Discretization (RD), on the other hand, operates by introducing Gaussian noise to each pixel in the training set. We refer to the models augmented with the aforementioned defenses as *Adv-FS* and *Adv-RD*, respectively. Similar to the *Adv* configuration, both *Adv-FS* and *Adv-RD* included re-training each model with the GAN-based AML training set only.

CNN Configuration	**Description**
Adversarially unaware *(Non-Adv)*	*Original models without any AML defense (baseline).*
Adversarially-aware *(Adv)*	*Plain models only retrained with AML samples.*
Adversarially-aware with Feature Squeezing *(Adv-FS)*	*Models augmented with Feature Squeezing and retrained with AML samples.*
Adversarially-aware with Randomized Discretization *(Adv-RD)*	*Models augmented with Randomized Discretization and retrained with AML samples.*

Table 7. Summary of experiments used to evaluate robustness against AML attacks.



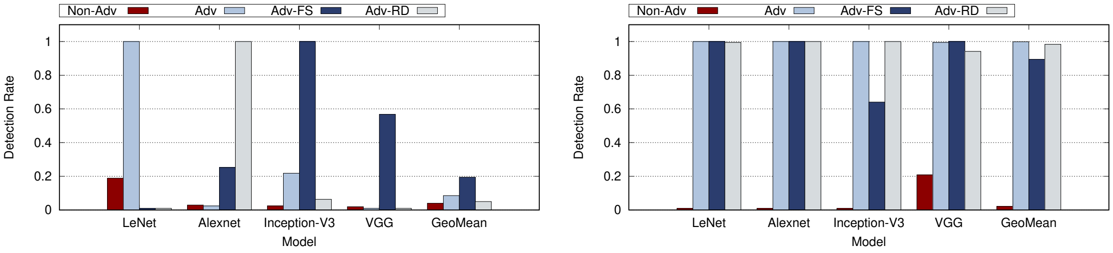
(a) Hilbert coloring scheme

(b) Entropy coloring scheme

Fig. 8. Comparison of adversarially-aware (*Adv**) and non-adversarially-aware (*Non-Adv*) detection rates using the ARM instruction set with (a) Hilbert coloring scheme, and (b) Entropy coloring scheme.

Figure 8 shows a comparison of the detection rates for the adversarially and non-adversarially aware models while using the smartphone configuration. The results are based on the AML test set that maps the ARM instructions into images that use the Hilbert coloring scheme. On average (GeoMean), *Adv-FS* had the best detection rate (96.4%), followed by *Adv-RD* (94.9%), *Adv* (22.8%), and then *Non-Adv* (8.8%) while using the Hilbert coloring system. For instance, we found that combining InceptionV3 with Feature Squeezing (*Adv-FS*) resulted in all of the AML test set samples getting detected. LeNet also offered a competitive detection rate of 96.6% when configured to use Randomization Discretization (*Adv-RD*). We also note that with the exception of VGG, simply re-training CNNs using AML samples without any hardening techniques (*Adv*) resulted in lower detection rates. Similarly, non-adversarially aware models resulted in low rates that ranged between 1.0% – 89.9% with a geometric mean of 8.8%. Such results underscore the importance of integrating adversarial datasets into the training process in order to increase the robustness of CNNs against AML attacks.

A similar trend was observed while using the entropy coloring scheme on the smartphone configuration. On average, *Adv-FS* offered higher rates across CNNs (96.4%), followed by *Adv-RD* (83.6%), *Adv* (80.9%), and then *Non-Adv* (21.6%). However, the highest rate was attained when augmenting VGG with *Adv-RD* (99.5%). On the other hand, LeNet achieved 96.1% when using *Adv-RD*. These results are summarized in Figure 8b. Overall, the average detection rate for the *Adv-RD* configuration across CNNs dropped by 11% when shifting from the Hilbert to the entropy coloring scheme. On



(a) Hilbert coloring scheme

(b) Entropy coloring scheme

Fig. 9. Comparison of adversarially-aware (*Adv**) and non-adversarially-aware (*Non-Adv*) detection rates using the x86 instruction set with (a) Hilbert coloring scheme, and (b) Entropy coloring scheme.

the other hand, the basic *Adv* configuration improved by 58% when using the entropy coloring scheme, while *Adv-FS* averaged similar results irrespective of the coloring scheme.

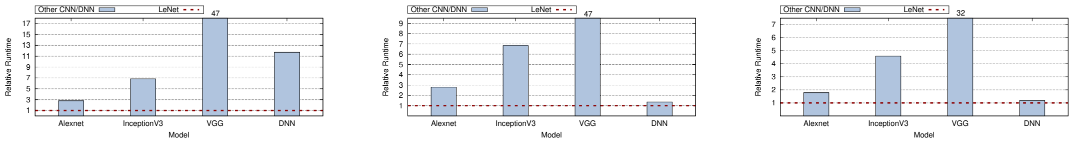
We observe a different trend with our chromebook platform that consumes the x86 instruction set (shown in Figure 9). Overall, we find that the entropy coloring scheme is more suitable for detecting AML attacks. For instance, with the exception of *Adv-FS* on InceptionV3, all of the adversarially-aware configurations were able to detect all of the ransomware samples from the AML test set. For example, combining LeNet with *Adv-FS* resulted in all of the AML test samples being detected. On average, the Hilbert coloring scheme was less effective across CNNs with the AML test set. The average ranged between 4.0% – 16.1%. However, each adversarially-aware model had at least one configuration that offered a perfect detection rate while using the Hilbert coloring scheme. We attribute the overall low average to the large amount of x86 instructions that had to be injected into the AML test set while using the Hilbert coloring scheme. We observed that this scheme required a high number of x86 instructions to be injected before the system could be fooled into misclassifying ransomware as benign. ARM-based ransomware, on the other hand, required significantly less instructions to be injected in order to fool the system. As such, the AML injections in the case of x86 were significantly more aggressive compared to its ARM counterpart. Despite the aggressive nature of the x86 AML samples, our design was able to achieve a detection rate of 100% against adversarial samples across all models while using the Hilbert coloring scheme. However, not all algorithms performed equally across the different models. The entropy-based scheme, on the other hand, is not as susceptible to this issue. This is because in addition to examining instruction clusters, the entropy-based models learn to detect encrypted data sections. Our results show that the overall entropy of encrypted data sections retain high levels of randomness even when noise is injected. This property makes the entropy coloring scheme less susceptible to AML attacks relative to the Hilbert-based approach.

7.4 Runtime and Energy Overhead

We conducted runtime and energy efficiency experiments across multiple platforms in order to understand the suitability of different models for deployment on mobile systems. We selected LeNet as the baseline CNN model for this study due to its simplicity. In addition, this model performed consistently well according to the previously discussed quality metrics across both ARM and x86 architectures.

Figure 10 shows the runtime of different CNN and DNN models across three platforms relative to LeNet. In general, Alexnet (8 layers) and the DNN (7 layers) had the second least impact on performance on ARM and x86 platforms, respectively. On the other hand, the VGG model had the highest performance impact due to its high number of hyper-parameters. We observed that in the case of the ARM architecture, the runtime relative to the LeNet model ranges between 3x to 47x. This is summarized in Figure 10a. The x86 architecture exhibited a similar trend as indicated by Figures 10b and 10c. We observed that the relative runtime for this architecture ranged between 1.2x to 47x. As a reference, we show the results of the DNN model on x86. The x86 DNN model had less overhead than LeNet and also less than the same DNN architecture on ARM. This is attributed to the smaller feature set used in the case of the x86 DNN (841 features) relative to its ARM counterpart (5014 features). We also note that VGG consistently exhibited the worst overhead across both x86 platforms.

Figure 11 shows the energy consumption of different CNN and DNN models across three platforms relative to LeNet. We observed a similar trend to the runtime performance with Alexnet and the DNN being the second most efficient models on ARM and x86 platforms, respectively. For instance, the VGG model proved to be the least efficient model. We observed that in the case of the ARM architecture, the energy relative to the LeNet model ranges between 3x to 47x. This is summarized in Figure 11a. The x86 architecture exhibits a similar trend in terms of relative energy as indicated

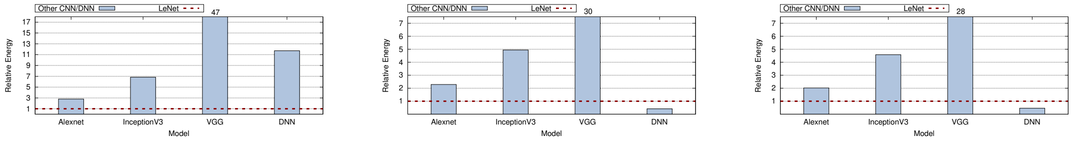


(a) Cortex-A53

(b) Core i3

(c) Core i7

Fig. 10. Runtime relative to LeNet running on (a) low-end (Cortex-A53), (b) mid-range (Core i3), and (c) high-end (Core i7).



(a) Cortex-A53

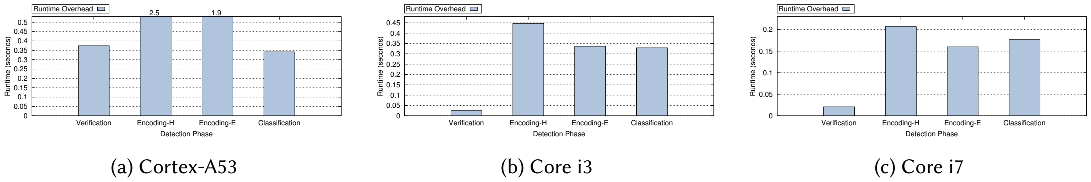
(b) Core i3

(c) Core i7

Fig. 11. Energy relative to LeNet running on (a) low-end (Cortex-A53), (b) mid-range (Core i3), and (c) high-end (Core i7).

by Figures 11b and 11c. We observed that the relative energy consumption for this architecture ranged between 0.5x to 30x. However, the reference DNN on x86 is more efficient than LeNet and the same DNN architecture on ARM due to its smaller feature set. We also highlight that the energy consumption is significantly larger for the VGG model across both x86 platforms. As such, the VGG model is the least suitable for mobile systems despite it's robustness to AML attacks. Overall, our results suggest that LeNet is the most suitable model for mobile systems.

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 12 and 13 show a breakdown of the aforementioned overheads by platform. In our design, the verification phase is executed every time an app is launched and represents the overhead associated with the common case. We measured runtime and energy costs of 374 ms and 1.1 J for the ARM Cortex-A53 platform (smartphone). On the other hand, we observed runtime and energy costs of 25 ms and 32 mJ for the Core i3 platform (low-end chromebook), and 21 ms and 90mJ for the Core i7 platform (high-end chromebook). Although the ARM-based platform has a lower power consumption than its x86 counterparts, the overall energy overhead is lower on x86 due to the minimal execution unit usage for this phase combined with the shorter execution times. In general, 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. 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. We measured runtime and energy costs of 2.5 s and 7.4 J for the ARM Cortex-A53 platform (smartphone configuration), 447 ms and 11.6 J for the Core i3 platform (low-end chromebook), and 206 ms and 22.3 J for the Core i7 platform (high-end chromebook) while using the Hilbert coloring scheme. This is denoted as Encoded-H in Figures 12 and 13. While this cost is relatively higher than the verification phase, this phase is only used for newly installed apps or any apps that undergo updates. Furthermore, we note that the entropy coloring scheme (Encoding-E) is approximately 25% more efficient than its Hilbert counterpart. For instance, entropy-based images on the ARM Cortex-A53 platform required 1.9 s and 5.6 J for runtime and energy, respectively. 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

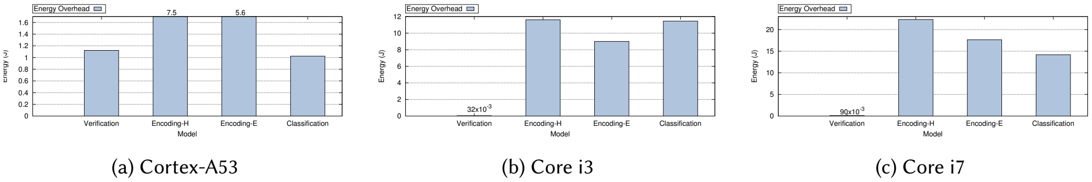


(a) Cortex-A53

(b) Core i3

(c) Core i7

Fig. 12. Breakdown of the runtime overhead for the main design phases (verification, encoding, and classification) on different platforms (a) low-end (Cortex-A53), (b) mid-range (Core i3), and (c) high-end (Core i7).



(a) Cortex-A53

(b) Core i3

(c) Core i7

Fig. 13. Breakdown of the energy consumption for the main design phases (verification, encoding, and classification) on different platforms (a) low-end (Cortex-A53), (b) mid-range (Core i3), and (c) high-end (Core i7).

phase through a CNN. We measured runtime and energy costs of 342 ms and 1 J for the ARM Cortex-A53 platform (smartphone configuration), 118 ms and 5.0 J for the Core i3 platform (low-end chromebook), and 99 ms and 7.0 J for the Core i7 platform (high-end chromebook) while using LeNet. Similar to the encoding phase, we expect this phase to only be used for newly installed apps or apps that undergo updates.

7.5 Final Model Recommendation

We conclude our study with a final recommendation on the CNN model, coloring scheme, and adversarial defense combination to use for smartphone (ARM) and chromebook (x86) platforms. We make this recommendation based on multiple factors that span standard ransomware app detection, robustness against AML attacks, and energy efficiency.

We emphasize energy efficiency in determining the most suitable configuration for smartphones since energy is treated as a first order constraint on such platforms. To this end, we recommend using the LeNet model with the entropy coloring scheme for smartphones. As previously discussed in section 7.1.2, we observe that the entropy coloring scheme has a slight advantage over its Hilbert-based counterpart in detecting ransomware since such malware often packs sensitive data in encrypted form. Unlike Hilbert, the entropy-based approach effectively highlights the presence of such data as features that CNNs can recognize. Our results show that this method offers a 98.0% detection rate against standard ransomware apps. Although InceptionV3 offered a slightly improved detection rate, LeNet lags this model only by 0.5% and has a better overall F-score. LeNet also offers a detection rate of 96.1% against AML-based ransomware under the same entropy coloring scheme after hardening it with Randomized Discretization. Our data shows that ARM-based applications leverage a wide range of instructions (5014 instruction opcodes in our dataset). As such, augmenting ransomware to pass as benign applications involves injecting a diverse set of instructions across an image. Randomized Discretization is suitable against this pattern of attacks since it involves training CNN models to recognize Gaussian noise. We also note that although InceptionV3 and VGG achieved slightly higher detection rates against AML attacks, they are energy inefficient. We find that LeNet is 7x and 47x more energy efficient than InceptionV3 and VGG which makes it suitable for smartphones. Also, configuring our solution with the entropy coloring scheme saves an additional 25% relative to the Hilbert coloring approach while still maintaining a high level of security. Unlike Hilbert which relies on a fine-grained mapping of instruction opcodes into images, the entropy coloring scheme uses a coarse-grained approach that requires significantly less computation.

We also recommend combining the LeNet model with the entropy coloring scheme for chromebook platforms. In addition to LeNet and the entropy coloring approach being the most energy efficient (2x − 30x more efficient than other models), this combination yields the best detection rate against standard ransomware apps (99.0%). Furthermore, augmenting the aforementioned model with Feature Squeezing results in a 100% detection rate against AML attacks. Although Randomized Discretization still offers a high detection rate of 99.5% against AML attacks, we observe that Feature Squeezing is slightly better when used with the x86 instruction set. This is because Feature Squeezing operates by reducing the number of features it depends on for detecting malicious images. Our data shows that on average, x86 applications contain 84% less instruction opcodes compared to applications that are implemented using the ARM instruction set. Since x86 applications generally use significantly less features (instruction opcodes) relative to their ARM counterpart, reducing the number of features CNN models depend on for ransomware detection through Feature Squeezing makes them probabilistically less susceptible to AML attacks.

7.6 Model Introspection and Feature Analysis

Convolutional neural networks (CNN) have shown significant promise in tackling a multitude of computer vision problems. Such CNNs, rely on a cascaded set of convolutional layers that are responsible for learning features that could be used to identify different objects within images. More specifically, convolutional layers use a collection of feature maps as kernel filters that are applied successively to the original input. Such maps consist of weights that are trained to detect a variety of features that can be as simple as edges and curves or as complex as animals and cars. Once the weights of a given map are known, the map's weights are used to activate learnable features by computing the dot product of the input image with the aforementioned weights. Figure 14 illustrates an example of the effect of different feature maps once they are applied to an input image that corresponds to the iconic Ford Model T. From this example, we can see that each feature map activates different areas of the car based on the features it learned. These areas are highlighted through a green shade within the processed images and correspond to the set of features that the CNN model learned to distinguish cars from other objects.

In this study, we examine the different feature maps that we extracted from our trained CNN model and analyze the machine code instructions that correlate to the areas activated by the feature maps. However, before we can discuss how machine code instructions can be derived from images (the inverse process of generating an image – reverse direction), it is important to understand how the pixels of an image are produced from a set of instructions first (forward direction). In other words, how the Hilbert space filling curve algorithm maps instructions to a set of pixels. The Hilbert space curve is designed to represent data through the use of different orders (granularities) of the Hilbert curve. This process requires determining a suitable order for the Hilbert curve, such that all the data points can be transformed into pixels on an image. This grid is then divided into a set of interconnected coordinates. For instance, an m order curve will map data into points that correspond to a path with $2m \times 2m$ coordinates. In other words, an order 1 curve can be thought of as a lattice path for a curve consisting of 4 coordinates that we refer to as a brick. For the purpose of this study, each coordinate is used to represent a byte. Therefore, each Hilbert brick can be used to represent a given instruction. Furthermore, in order to represent a sequence of instructions while preserving their spatial information, a set of Hilbert bricks are connected to form a continuous path, with each subsequent brick undergoing a 90° rotation. This ensures that every last byte of instruction I_n is connected to the first byte of instruction I_{n+1}. This process of

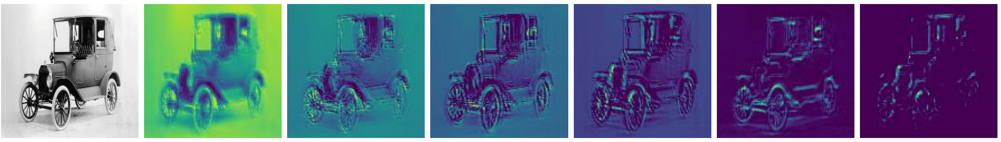


Fig. 14. Visualization of the feature maps generated by one of the convolution layers for an image representing a Ford Model T car.

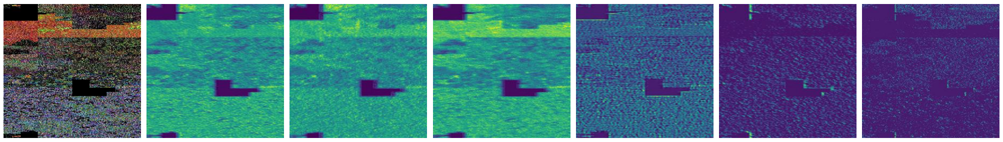


Fig. 15. Visualization of the feature maps generated by one of the convolution layers for the Hilbert coloring-scheme.

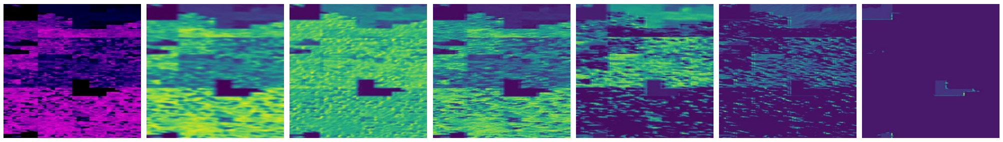


Fig. 16. Visualization of the feature maps generated by one of the convolution layers for the entropy coloring-scheme.

mapping instructions into pixels in the forward direction using bricks is illustrated in Figure 17 (right to left). We can see in Figure 17 that in the case of a branch instruction, starting from the far right, the data is mapped in order with byte 0xD6 mapped to location ’00’, byte 0x1F mapped to location ’01’, etc. The ordered bytes are then positioned onto an order 1 curve to construct a Hilbert brick that is eventually mapped to an order 3 Hilbert curve which is used to construct a segment of the final image. This process is repeated until all of the instructions have been mapped.

Unlike the previously discussed instruction to image mapping, performing introspection requires using the reverse direction. In order to achieve this, we used a our trained LeNet model. As outlined in Figure 17 (left to right), we first extracted the feature maps of the convolutional layers. A visualization of these maps for the Hilbert and entropy coloring schemes are shown in Figures 15 and 16, respectively. Similar to the Ford Model T example in Figure 14, the sections that are highlighted in green represent activated areas that correspond to important features the model has learned. Therefore, in order to derive the instructions that are associated with the aforementioned features, we first extracted the weights of the feature maps the model has learned. We then determined the set of pixels that form the receptive field for each activated region on the image in strides that correspond to the kernel size. We then map each pixel within a given receptive field of the image back to the original instruction that was used to produce the pixels. This requires using the previously discussed Hilbert curve in reverse order. A summary of the process of mapping pixels back to their original instructions is shown Figure 17 (left to right).

Overall, we collected several instructions in the case of the ARM-based images while using the Hilbert coloring scheme. The instructions we collected from the feature maps spanned 13 categories. However, we found that the following instruction categories accounted for more than 80% of the features: load/store, compare/branch, arithmetic, single instruction multiple data (SIMD), and logical. For instance, we observed that instruction from the load/store category accounted for 30% of the features. Given that data encryption involves continuously reading plaintext and writing the corresponding ciphertext information, load and store operations are expected to play an important role in the detection of ransomware. Examples of such instructions include, `ldrsb`, `ldpsw`, and `vldr`. Similarly, we observed several compare-branch instructions. For instance, we observe instructions such as `cbz` and `ccmn`. In addition to arithmetic

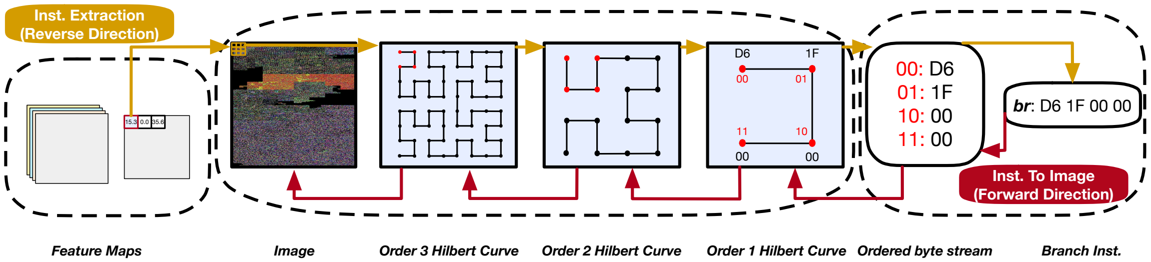


Fig. 17. Methodology for extracting the instructions CNN models learn for classifying ransomware samples.

instructions, we observe SIMD instructions. Such instructions are designed to harness parallelism that is available through SIMD units within ARM processors. We also found other logical instructions that are commonly used in cryptographic operations such as `eon`, `orr`, and `orrn`, in addition to non-SIMD shift operations such as `bic`, and `bics`. Overall, we found that the three main instructions stand out as the predominant features in ransomware detection correspond to these instructions: `cbz` (compare and branch on zero), `add` (addition), and `ldrsb` (load register byte). We find this to be consistent with the features that we collected from the feature maps of the second convolutional layer of the LeNet CNN. Therefore, these instructions play in important role identifying the ransomware. Finally, we observe a similar trend for the entropy coloring scheme. The main difference is that the entropy tracks less features compared to Hilbert. We attribute this to the fact that entropy coloring scheme uses a smaller set of RGB colors in comparison to the Hilbert scheme. Despite this, we still observe the same category of instructions with the `cbz`, `add`, and `ldrsb` instructions representing the dominant features.

A similar analysis over the x86 CNN trained with the Hilbert coloring scheme resulted in several features that spanned 14 instruction categories. However, we observed that three categories encompassed 70% of the instructions that we extracted from the feature maps. These categories correspond to arithmetic, logical, and data move instructions (includes load/store instructions). In the case of the arithmetic type of instructions, we observed addition instructions such as `add`, `adc`, and `imul`, with `adc` being the most frequently used instruction within the feature maps. We also found instructions that are commonly used to perform cryptographic operations, such as `or`, `and`, and `xor`, represented 20% of the features used for ransomware detection. Finally, data move instructions such as `mov` occurred frequently within the feature maps, representing 17% of the total features. Such instructions are used for performing loads and stores on x86 architectures. We observe a similar trend for the x86 CNN trained with the entropy coloring scheme. We find that the arithmetic, logical, and data move instructions contribute to 75% of the instructions. However, unlike with Hilbert, we note that the features for the entropy had the `or` instruction as the most dominant feature followed by `adc` instruction.

Overall, our analysis shows that load, arithmetic, and compare instructions represent the most important features our recommended CNN uses for identifying ransomware tailored for smartphone platforms (ARM instruction set). In the case of x86, we find that arithmetic, logical, and data move instructions stand out as important features in identifying ransomware that is designed x86-based platforms. This observation correlates to the typical behavior of most ransomware applications. In other words, ransomware applications tend to encryption and delete data which requires issuing a large amount of load and store transactions in addition to operations that are needed to perform cryptographic operations, similar to what would be found in algorithms such as the Advanced Encryption Standard (AES).

7.7 Comparison to Other CNN Solutions

Multiple bodies of work explored the use of deep learning for malware detection [3, 20, 22, 41, 47, 66, 73, 75, 76]. Table 8 includes a comparison of our work to other solutions. We outline the differences in terms of metrics associated with the detection methods, datasets, features, robustness against adversarial attacks, and the overall overhead. In general, we find that previous work did not evaluate the robustness of their solutions against adversarial attacks. Our work, on the other hand, provides a thorough discussion on support against such attacks and the importance of making CNN models adversarially aware. For instance, we achieve 99.3%, 99.0%, 0.96%, and 97.5% for the accuracy, precision, recall and F-Score metrics, respectively. More, other studies did not discuss or analyze the performance overhead of their detection systems on the proposed platforms. Since deep learning can be taxing on system resources, proposed solutions may lead to high energy and resource utilization, especially on mobile systems. Our design is solution is designed to offset the computational demands of CNNs on mobile systems by optimizing the common use case to incur minimal application startup and energy costs. For example, work by Vasan et al. [66] harnesses pre-trained models for malware detection across Windows and Android-based IoT platforms. They employed raw binary files with gray scale and RGB colored images for training. Their solution was able to achieve 98.8% and 97.4% accuracy on Windows and Android, respectively. However, the use of raw binaries makes the approach vulnerable adversarial attacks. In addition to our work being adversarially aware, our solution achieves better accuracy. Other work by Zhang et al. [76] proposed the use of opcodes and system calls for training neural networks to detect malware designed for Windows platforms. However, this method was able to attain less than 96.0% across all the metrics. Similarly, [75] proposed the use of N-gram opcodes present in Windows applications as features for a self-attention-based CNN solution. Their metrics, however, were below 90% in all cases.

Other work by Martinelli et al. [41] explored the use of system calls for training an NLP-based model for malware detection on Android devices. The proposed system, however, is less than 80% accurate. Similarly, a study by Nix et al. [47] used a similar approach, but improved the overall accuracy. Work by Hasegawa [20] explored the use of N-bytes located either at the beginning or the end of a raw APKs to train CNN models. The authors report a test accuracy of 96.5% without a discussion on other commonly used metrics. Furthermore, the proposed approach can be easily compromised since the model does not learn the patterns of the full APK. Therefore, the solution is susceptible to triggering many false positives and negatives. Finally, [3, 22, 73] discussed the use of opcodes as features for malware detection. Xiao et al. [73] paired the use of opcodes with a customized CNN model which reported an accuracy of 93.0% and >90% for the remaining metrics. Similarly, R2-D2 [22] used opcodes to train an InceptionV3 model to achieved an accuracy of 98.5%. Our work outperforms these solutions across many metrics. In addition, our work proposes a solution that uses native instructions in order to mitigate delayed attacks that can fetch malicious in native form that can run directly on a victim's platform. Unlike prior work, we also show that our solution is robust against adversarial attacks. Finally, our work includes a framework for performing introspection on CNN models that can better assist researchers in understanding the important features associated with malware detection.

8 DISCUSSION AND LIMITATIONS

8.1 Ransomware Detection and Evasion

Prior research has shown that CNNs possess an excellent ability to recognize complex patterns present in images [12, 23, 29, 30, 52, 53, 72, 79]. Unfortunately, such models cannot directly consume application code without being systematically transformed into 2D images first. Our study relies on space-filling curves to fulfill this transformation.

Work	Platform	Detection Method	Dataset			Features	Spacial Info. Preservation	Detection of Adv. Learning	Accuracy		Precision		Recall		F-Score		Overhead Analysis	Introspection
			Size	Data Source	Ransom Samples				Non-Adv.	Adv.	Non-Adv.	Adv.	Non-Adv.	Adv.	Non-Adv.	Adv.		
[22]	Android	State-of-Art CNN models	2M	Leopard Mobile Inc.	✔	Opcodes	RGB Coloring	✘	0.985	-	0.965	-	0.964	-	0.944	-	✘	✘
[75]	Windows	Self-Attention CNN	1.8K	Virus Total	✔	Opcodes	✘	✘	0.895	-	0.875	-	0.875	-	0.873	-	✘	✘
[66]	Windows	Fine-tuned CNN Models	9.4K	Malimg	✘	Raw Binaries	Gray Scale & RGB Coloring	✘	0.988	-	0.989	-	0.988	-	0.981	-	✘	✘
	IoT Android		17K	IoT - Android Malware	✘	Raw Binaries		✘	0.974	-	0.973	-	0.973	-	0.973	-	✘	✘
[41]	Android	Customized CNN	7K	Google Play & Drebin	✘	System Calls	✘	✘	0.75 - 0.8	-	0.75 - 0.8	-	-	-	-	-	✘	✘
[20]	Android	Customized CNN	7K	AMD, Drebin, Appsapk, & Apkpure	✘	N-Bytes Raw APK	✘	✘	0.965	-	-	-	-	-	-	-	✘	✘
[47]	Android	Customized CNN	1.2K	Contagio repository & App stores	✘	System Calls	✘	✘	0.994	-	1.000	-	0.983	-	-	-	✘	✘
[76]	Windows	Neural Networks	10.4K	VxHeaven	✘	Opcodes & System Calls	✘	✘	0.951	-	0.957	-	0.943	-	0.950	-	✘	✘
[73]	Android	Customized CNN	10.5K	Google Play & AMD	✘	Opcodes	RGB Coloring	✘	0.930	-	>0.90	-	>0.90	-	>0.90	-	✘	✘
Our Work	Android	State-of-Art CNN models	15K	AndroZoo & AMD	✔	Native Instructions	RGB Coloring & Hilbert Space Filling Curve	✔	0.997	0.993	1.000	0.990	0.981	0.961	0.990	0.975	✔	✔

Table 8. Summary of CNN-based Malware Detection Systems.

Unlike other transformations, space-filling curves can impose a linear ordering of points in a multi-dimensional space. This approach ensures that opcodes that occur in a given sequence within an app will correlate to pixels that are also adjacent to each other in the generated image. Once an application is converted into a 2D image, CNNs can be harnessed to recognize visual patterns that correlate to malicious code. Our results underscore the effectiveness of combining CNNs with space-filling curves in reliably detecting both locker and cryptographic ransomware with high accuracy. Unfortunately, cybercriminals are always on the lookout for ways to evade existing detection systems. Reports show that attackers are repacking apps with malicious content directly in native form to evade existing solutions that analyze high level bytecode [69]. Unlike prior work, RansomShield is resilient to such attacks since it implements its defense directly in the Android Runtime system (ART). This allows RansomShield to examine the native instructions of a given app immediately before it is launched on the device. We verified our solution against samples from the Jisut family which employs the aforementioned obfuscation technique [42]. Despite the presence of obfuscated code in these samples, our solution was able to detect 96% of such ransomware.

In addition to obfuscation attacks, a cybercriminal may choose to publish a seemingly legitimate application that defers its attack until it is installed on a victim's device. Once such malware is cleared from the security checks present in the corresponding app store and is installed on a victim's device [7, 8], an update is issued to an associated C&C server to fetch its malicious payload. RansomShield mitigates against such attacks by maintaining a database that tracks all of the OATs installed on the device along with their corresponding hashes. Whenever, an application is updated, a new OAT is compiled. In this case, RansomShield compares the hash of the newly generated OAT file against the most recently saved hash it has in the database for the same OAT. If the hashes don't match, the design runs the newly produced OAT through the full detection system for additional verification.

Finally, an attacker may leverage adversarial machine learning techniques to defeat our detection system. Our design mitigates such attacks by leveraging a Generative Adversarial Network (GAN) to produce a large number of ransomware variants that could be used to defeat our defense. In our study, we used the GAN to generate 40K ransomware variants. Such samples were used to train our CNN models to make them adversarially aware. Although our results demonstrate that our adversarially aware models were able to generalize and detect new adversarial samples that were not seen by the trained models with high accuracy (over 96% for smartphones and over 99% for chromebooks), it is feasible for an attacker to design a ransomware sample that contains specially crafted perturbations that evades our design. Therefore, as an additional safety measure, we harden our models with another layer of protection that uses Feature Squeezing and Random Discretization techniques to make them more robust against AML attacks. Although it is still possible for an attacker to design new samples that could evade our system, we believe our approach raises the bar for malicious actors to penetrate our defense.

8.2 The Opportunity for Efficient Acceleration

The advent of CNNs combined with advancements in hardware has prompted chip vendors to introduce artificial intelligence accelerators [27, 48] including the integration of such accelerators into smartphones [4, 6]. To this end, we discuss how such accelerators could be used to improve the efficiency our design. To illustrate the opportunity for efficient acceleration, we compared the runtime and energy of the most capable processor we used in our evaluation to a standard GPU. This correlated to comparing the performance of the Core i7 processor from our high-end platform in Table 1 to an Nvidia P2000 GPU [49] while running the classification phase of our design. We note that the Core i7 has a Thermal Design Power (TDP) that is 1.2× higher than the Nvidia P2000's TDP, and thus consumes a little more power.

Overall, we observed that the performance of the classification phase under different CNN models improved when using the P2000 accelerator. On average, we observed that the speedup relative to the Core i7 configuration improved by a factor of 2.7×. In general, the classification phase improved when using an accelerator as a function of the model complexity and ranged between 1.3× to 5.2× depending on the CNN model. In particular, the classification phase with LeNet exhibited a 1.3× speedup. This translates to an improvement in the overall time it takes to complete the classification stage in the event that a user installs a new app or performs any updates. We observed a similar trend in energy efficiency when using the accelerator. On average, the efficiency relative to the Core i7 configuration improved by a factor of 14× which is particularly attractive to energy constrained platforms. Similarly, we note that the energy efficiency of the classification phase improved as a function of the model complexity and ranged between 8.6× to 22.8× in energy reduction depending on the CNN model. In particular, the classification phase with the LeNet model exhibited an 8.6× reduction in energy. We note that even when comparing the runtime and energy of the Core i3 processor from our mid-range platform in Table 1 to the P2000 accelerator, we still observed significant improvement in both speedup and energy efficiency across all models. On average, we observed a 5× and 11× in speedup and relative reduction in energy, respectively. We still observed 11× in energy reduction despite the fact that the Core i3 processor has a TDP that is 1.2× less than the TDP of the P2000 accelerator. We believe these results underscore the potential our design has for scaling to future accelerators.

9 RELATED WORK

Ransomware Detection. The rapid growth in ransomware has prompted researchers to explore different solutions through multiple bodies of work [2, 11, 25, 31, 32, 35, 44, 46, 57, 58, 61]. Techniques, such as UNVEIL [31] and CryptoDrop [57] consider the sequence of API calls and the entropy levels of I/O transactions sent to the disk in order to detect cryptographic transactions issued to the filesystem. Work by Kharraz and Kirda [32] proposed mediating access to user files and redirecting privileged requests to a protected area. Other work, such as RWGuard [44] explored the use of decoy files in addition to other I/O features to perform real-time detection of ransomware. Techniques like RansomProber [11] monitor finger movements on mobile devices as a metric for distinguishing between benign and malicious activity. They treat finger movements as an indicator that the user approves the application's activity as opposed to some unsupervised encrypting process that takes place without the user's consent. Other work by Lachtar et al. [35] explored the use of machine learning algorithms for classifying ransomware. However, the algorithms employed in these studies have been shown to not generalize well and suffer from limitations related to overfitting. Recent work by Amin et al. [3] explored the use of recurrent neural networks for malware detection. Although their solution offers a high detection rate against standard malware, it does not consider the impact of adversarial machine learning attacks. A core component of the design proposed in [3] involves the use of long short term memory cells (LSTM) that learn to recognize sequential input. However, this approach is vulnerable to simple perturbations that intersperse NOP instructions across malicious code. Furthermore, the aforementioned work bases its detection on Dalvik byte code that can be evaded by obfuscation attacks that embed malicious instructions directly in native form. Unlike [3], our solution is robust against such obfuscation attacks since it analyzes the final native code to be executed on the platform. We devise an approach that hardens CNNs against adversarial attacks. Finally, LSTMs have been shown to require a significant amount of compute resources relative to CNNs [55]. This makes our solution solution more suitable for deployment on mobile devices that are otherwise, resource constrained.

More recent work by Lachtar et al. [25] proposed an initial idea for harnessing CNNs to perform ransomware detection. Other dynamic solutions include AntiBotics [2], which relies on access controls that periodically re-authenticate applications attempting to modify or delete existing files. This ensures that I/O transactions which affect the filesystem are invoked by authorized users instead of bots. Other work such as EldeRan [58] explore the relevance of desktop related features such as registry key operations in Windows and file extensions as a way of classifying binaries as ransomware. Unlike the aforementioned solutions that mostly focus on ransomware detection in classical desktop environments, this paper explores a lightweight solution that seamlessly integrates into the runtime ecosystem of mobile platforms. We propose a solution that detects both locker and encrypting ransomware while considering the energy and performance constraints of mobile systems.

Ransomware Recovery. Multiple studies have proposed recovery techniques that enable the restoration of encrypted data [9, 13, 24, 33, 37]. PayBreak [33] monitors system-level crypto services and holds generated cryptographic keys in escrow for later recovery. FlashGuard[24] proposes a hardware-based solution that leverages out-of-place writes used to mitigate long erase latencies associated with solid state drives as a mechanism for recovering previously encrypted files. Other work [9, 13] explore backup solutions as a defense mechanism against ransomware. For instance, ShieldFS [13], uses a driver that creates file backups in response to filesystem transactions as a way of safeguarding the integrity of user data. Unfortunately, using such backup approaches are not suitable on mobile systems due to the limited storage and compute resources. Other aforementioned solutions are also not sufficient alone since cryptographic algorithms can be directly embedded in ransomware. Thus, preventing solutions such as PayBreak to get access to cryptographic keys used by ransomware.

10 CONCLUSION

In this study, we propose an energy efficient solution that leverages convolutional neural networks to reliably detect ransomware. We evaluate the effectiveness of using space-filling curves in transforming native instructions into images for use with CNNs to detect mobile ransomware. We show that simple CNNs that are tuned for computer vision tasks can be re-purposed to solve important problems in mobile security. We demonstrate that near ideal accuracy can be achieved for detecting ransomware when combining CNNs with images generated through space-filling curves. We evaluate the robustness of such CNNs to adversarial machine learning attacks and show the importance of hardening CNNs against such attacks. Finally, with many mobile chips introducing artificial intelligence accelerators into their processors such as Apple's A15, Huawei's Kirin 980, and Qualcomm's 845 Snapdragon, we believe our work paves the way for new security applications that can leverage these newly available accelerators in mobile processors beyond the intended computer vision and natural language processing applications.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers and the editor for their feedback and comments on this work. This work was funded in part by the National Science Foundation under grant CNS-1947580.

REFERENCES

[1] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, New York, NY, USA, 468–471.

[2] Or Ami, Yuval Elovici, and Danny Hendler. 2018. Ransomware Prevention Using Application Authentication-based File Access Control. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. ACM, New York, NY, USA, 1610–1619.

[3] Muhammad Amin, Babar Shah, Aizaz Sharif, Tamleek Ali, Ki-Il Kim, and Sajid Anwar. 2019. Android malware detection through generative adversarial networks. *Transactions on Emerging Telecommunications Technologies* (2019), e3675.

[4] AnandTech. 2018. The mate 20 and Mate 20 Pro Review: Kirin 980 Powering Two Contrasting Devices. https://www.anandtech.com/show/13503/the-mate-20-mate-20-pro-review.

[5] App Annie. 2019. The State of Mobile. https://www.appannie.com.

[6] Apple. 2019. A12 Bionic the smartest, most powerful chip in a smartphone. https://www.apple.com/iphone-xs/a12-bionic/.

[7] Avast. 2017. WannaCry WannaBe targeting Android smartphones. https://blog.avast.com/wannacry-wannabe-targeting-android-smartphones.

[8] Avast. 2020. How to Remove Ransomware from Android Devices. https://www.avast.com/c-how-to-remove-ransomware-android.

[9] Muhammet Baykara and Baran Sekin. 2018. A novel approach to ransomware: Designing a safe zone system. In *Digital Forensic and Security (ISDFS), 2018 6th International Symposium on*. IEEE, 1–5.

[10] Victor Chebyshev. 2020. Mobile malware evolution 2020. https://securelist.com/mobile-malware-evolution-2020/101029.

[11] Jing Chen, Chiheng Wang, Ziming Zhao, Kai Chen, Ruiying Du, and Gail-Joon Ahn. 2018. Uncovering the Face of Android Ransomware: Characterization and Real-Time Detection. *IEEE Transactions on Information Forensics and Security* 13, 5 (May 2018), 1286–1300.

[12] L. Chen, P. Bentley, K. Mori, K. Misawa, M. Fujiwara, and D. Rueckert. 2018. DRINet for Medical Image Segmentation. *IEEE Transactions on Medical Imaging* 37, 11 (Nov 2018), 2453–2462.

[13] Andrea Continella, Alessandro Guagnelli, Giovanni Zingaro, Giulio De Pasquale, Alessandro Barenghi, Stefano Zanero, and Federico Maggi. 2016. ShieldFS: a self-healing, ransomware-aware filesystem. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, ACM, New York, NY, USA, 336–347.

[14] Aldo Cortesi. 2015. A library for drawing space-filling curves like the Hilbert Curve. https://github.com/cortesi/scurve.

[15] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. 2013. On the Feasibility of Online Malware Detection with Performance Counters. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*. ACM, New York, NY, USA, 559–570.

[16] C. Gotsman and M. Lindenbaum. 1996. On the metric properties of discrete space-filling curves. *IEEE Transactions on Image Processing* 5, 5 (May 1996), 794–797.

[17] Grayshift. [n.d.]. Introducing GrayKey. https://graykey.grayshift.com.

[18] Chris Hamilton. 2006. Compact hilbert indices. *Dalhousie University, Faculty of Computer Science, Technical Report CS-2006-07* (2006).

[19] C. H. Hamilton and A. Rau-Chaplin. 2007. Compact Hilbert Indices for Multi-Dimensional Data. In *First International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*. 139–146.

[20] Chihiro Hasegawa and Hitoshi Iyatomi. 2018. One-dimensional convolutional neural networks for Android malware detection. In *2018 IEEE 14th International Colloquium on Signal Processing & Its Applications (CSPA)*. IEEE, 99–102.

[21] David Hilbert. 1935. Über die stetige Abbildung einer Linie auf ein Flächenstück. In *Dritter Band: Analysis· Grundlagen der Mathematik· Physik Verschiedenes*. Springer, 1–2.

[22] TonTon Hsien-De Huang and Hung-Yu Kao. 2018. R2-D2: color-inspired convolutional neural network (CNN)-based android malware detections. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2633–2642.

[23] D. Huang, J. J. Lim, L. Fei-Fei, and J. C. Niebles. 2017. Unsupervised Visual-Linguistic Reference Resolution in Instructional Videos. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1032–1041.

[24] Jian Huang, Jun Xu, Xinyu Xing, Peng Liu, and Moinuddin K. Qureshi. 2017. FlashGuard: Leveraging Intrinsic Flash Properties to Defend Against Encryption Ransomware. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)* (Dallas, Texas, USA). ACM, 2231–2244.

[25] N. Lachtar D. Ibdah and A. Bacha. 2020. Towards Mobile Malware Detection Through Convolutional Neural Networks. In *IEEE Embedded Systems Letters (ESL)*, Vol. 19. 126–129.

[26] Intel [n.d.]. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3, Section 14.7 (as of November 2011).

[27] Intel. [n.d.]. Movidius Neural Compute Stick. https://software.intel.com/en-us/movidius-ncs.

[28] Intel. 2019. Chromeboox powered by Intel. https://intel.com.

[29] A. Karpathy and L. Fei-Fei. 2017. Deep Visual-Semantic Alignments for Generating Image Descriptions. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39, 4 (April 2017), 664–676.

[30] Andrej Karpathy, Armand Joulin, and Li Fei-Fei. 2014. Deep Fragment Embeddings for Bidirectional Image Sentence Mapping. In *Proceedings of the International Conference on Neural Information Processing Systems (NIPS)*. 1889–1897.

[31] Amin Kharraz, Sajjad Arshad, Collin Mulliner, William Robertson, and Engin Kirda. 2016. UNVEIL: A Large Scale, Automated Approach to Detecting Ransomware. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association.

[32] Amin Kharraz and Engin Kirda. 2017. Redemption: Real-time protection against ransomware at end-hosts. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 98–119.

[33] Eugene Kolodenker, William Koch, Gianluca Stringhini, and Manuel Egele. 2017. PayBreak: Defense Against Cryptographic Ransomware. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (Asia CCS)*. ACM, New York, NY, USA, 599–611.

[34] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems (NIPS)*. ACM, New York, NY, USA, 1097–1105.

[35] N. Lachtar, D. Ibdah, and A. Bacha. 2019. The Case for Native Instructions in the Detection of Mobile Ransomware. *IEEE Letters of the Computer Society* 2, 2 (June 2019), 16–19. https://doi.org/10.1109/LOCS.2019.2918091

[36] Yann LeCun, LD Jackel, Léon Bottou, Corinna Cortes, John S Denker, Harris Drucker, Isabelle Guyon, Urs A Muller, Eduard Sackinger, Patrice Simard, et al. 1995. Learning algorithms for classification: A comparison on handwritten digit recognition. *Neural networks: the statistical mechanics perspective* 261 (1995), 276.

[37] Jeonghwan Lee, Jinwoo Lee, and Jiman Hong. 2017. How to Make Efficient Decoy Files for Ransomware Detection?. In *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*. ACM, 208–212.

[38] Jin Li, Lichao Sun, Qiben Yan, Zhiqiang Li, Witawas Srisa-an, and Heng Ye. 2018. Significant Permission Identification for Machine-Learning-Based Android Malware Detection. *IEEE Trans. Industrial Informatics* 14, 7 (2018), 3216–3225.

[39] Yuping Li, Jiyong Jang, Xin Hu, and Xinming Ou. 2017. Android Malware Clustering Through Malicious Payload Mining. In *RAID (Lecture Notes in Computer Science, Vol. 10453)*. Springer, 192–214.

[40] Lookout. [n.d.]. U.S. Targeted by Coercive Mobile Ransomware Impersonating the FBI. https://blog.lookout.com/scarepakage.

[41] Fabio Martinelli, Fiammetta Marulli, and Francesco Mercaldo. 2017. Evaluating convolutional neural network for effective mobile malware detection. *Procedia computer science* 112 (2017), 2372–2381.

[42] Alejandro Martín, Julio Hernandez-Castro, and David Camacho. 2018. An in-Depth Study of the Jisut Family of Android Ransomware. *IEEE Access* 6 (2018), 57205–57218. https://doi.org/10.1109/ACCESS.2018.2873583

[43] McAfee. [n.d.]. Android Banking Trojan MoqHao Spreading via SMS Phishing in South Korea. https://www.mcafee.com/blogs/other-blogs/mcafee-labs/android-banking-trojan-moqhao-spreading-via-sms-phishing-south-korea/.

[44] Shagufta Mehnaz, Anand Mudgerikar, and Elisa Bertino. 2018. Rwguard: A real-time detection system against cryptographic ransomware. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 114–136.

[45] Bongki Moon, Hosagrahar V Jagadish, Christos Faloutsos, and Joel H. Saltz. 2001. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on knowledge and data engineering* 13, 1 (2001), 124–141.

[46] Routa Moussaileb, Benjamin Bouget, Aurélien Palisse, Hélène Le Bouder, Nora Cuppens, and Jean-Louis Lanet. 2018. Ransomware's Early Mitigation Mechanisms. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*. ACM, ACM, New York, NY, USA, 2:1–2:10.

[47] Robin Nix and Jian Zhang. 2017. Classification of Android apps and malware using deep neural networks. In *2017 International joint conference on neural networks (IJCNN)*. IEEE, 1871–1878.

[48] Nvidia. [n.d.]. Jetson Nano. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/.

[49] Nvidia. 2019. Unmatched power. Unmatched creative freedom. Nvidia Quadro P2000. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/.

[50] Meltem Ozsoy, Caleb Donovick, Iakov Gorelic, Nael Abu-Ghazaleh, and Dimitry Ponomarev. 2015. Malware-aware processors: A framework for efficient online malware detection. In *International Symposium on High Performance Computer Architecture (HPCA)*. 651–661.

[51] G Peano. 1980. The Principles of arithmetic, presented by a new method. Translated in H. Kennedy (1973). Selected works of Guiseppe Peano.

[52] S. Pereira, A. Pinto, V. Alves, and C. A. Silva. 2016. Brain Tumor Segmentation Using Convolutional Neural Networks in MRI Images. *IEEE Transactions on Medical Imaging* 35, 5 (May 2016), 1240–1251.

[53] Guido Pusiol, Andre Esteva, Scott S. Hall, Michael Frank, Arnold Milstein, and Li Fei-Fei. 2016. Vision-Based Classification of Developmental Disorders Using Eye-Movements. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*. Springer International Publishing, Cham, 317–325.

[54] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision* 115, 3 (2015), 211–252.

[55] Mohammad Hossein Samavatian, Anys Bacha, Li Zhou, and Radu Teodorescu. 2020. RNNFast: An Accelerator for Recurrent Neural Networks Using Domain-Wall Memory. *J. Emerg. Technol. Comput. Syst.* 16, 4, Article 38 (Sept. 2020), 27 pages. https://doi.org/10.1145/3399670

[56] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli. 2018. MADAM: Effective and Efficient Behavior-based Android Malware Detection and Prevention. *IEEE Transactions on Dependable and Secure Computing* 15, 1 (2018), 83–97.

[57] Nolen Scaife, Henry Carter, Patrick Traynor, and Kevin RB Butler. 2016. Cryptolock (and drop it): stopping ransomware attacks on user data. In *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*. IEEE, IEEE Computer Society, Los Alamitos, CA, USA, 303–312.

[58] D Sgandurra, L Muñoz-González, R Mohsen, and EC Lupu. 2016. Automated Dynamic Analysis of Ransomware: Benefits. *Limitations and use for Detection. arXiv preprint* (2016).

[59] R. J. Stevens, A. F. Lehar, and F. H. Preston. 1983. Manipulation and Presentation of Multidimensional Image Data Using the Peano Scan. *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-5, 5 (Sep. 1983), 520–526.

[60] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2818–2826.

[61] Yuki Takeuchi, Kazuya Sakai, and Satoshi Fukumoto. 2018. Detecting Ransomware using Support Vector Machines. In *Proceedings of the 47th International Conference on Parallel Processing Companion*. ACM, ACM, New York, NY, USA, 1–6.

[62] TensorFlow™. 2019. An open source machine learning framework for everyone. https://www.tensorflow.org.

[63] Klein Tools. 2019. Security module for Raspberry PI. https://www.zymbit.com/zymkey.

[64] Klein Tools. 2019. USB Digital Meter, USB-A and USB-C ET920. https://www.kleintools.com.

[65] Trend Micro. [n.d.]. Enterprise Cybersecurity Solutions. https://www.trendmicro.com.

[66] Danish Vasan, Mamoun Alazab, Sobia Wassan, Hamad Naeem, Babak Safaei, and Qin Zheng. 2020. IMCFN: Image-based malware classification using fine-tuned convolutional neural network architecture. *Computer Networks* 171 (2020), 107138.

[67] Virus Total. [n.d.]. VirusTotal. https://www.virustotal.com.

[68] Jihua Wang, Baoquan Wang, Lianshun Zhang, Xianghua Dou, and Liling Zhao. 2004. Theoretical Study on the Z Curves. *Journal of Biomathematics* 19, 2 (2004), 129–135.

[69] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. 2017. Deep Ground Truth Analysis of Current Android Malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer, 252–276.

[70] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. 2017. Deep Ground Truth Analysis of Current Android Malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer, Bonn, Germany, 252–276.

[71] Wikipedia. 2015. Nexus 5X. https://en.wikipedia.org/wiki/Nexus_5X.

[72] Y. Wu, T. Hassner, K. Kim, G. Medioni, and P. Natarajan. 2018. Facial Landmark Detection with Tweaked Convolutional Neural Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 40, 12 (Dec 2018), 3067–3074.

[73] Xusheng Xiao. 2019. An image-inspired and cnn-based android malware detection approach. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1259–1261.

[74] Weilin Xu, David Evans, and Yanjun Qi. 2018. Feature Squeezing: Detecting Adversarial Examples in Deep Neural Networks. *Proceedings 2018 Network and Distributed System Security Symposium* (2018). https://doi.org/10.14722/ndss.2018.23198

[75] Bin Zhang, Wentao Xiao, Xi Xiao, Arun Kumar Sangaiah, Weizhe Zhang, and Jiajia Zhang. 2020. Ransomware classification using patch-based CNN and self-attention network on embedded N-grams of opcodes. *Future Generation Computer Systems* 110 (2020), 708–720.

[76] Jixin Zhang, Zheng Qin, Hui Yin, Lu Ou, and Kehuan Zhang. 2019. A feature-hybrid malware variants detection using CNN based opcode embedding and BPNN based API embedding. *Computers & Security* 84 (2019), 376–392.

[77] Yuchen Zhang and Percy Liang. 2019. Defending against Whitebox Adversarial Attacks via Randomized Discretization. arXiv:1903.10586 [cs.LG]

[78] Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, and Ajay Joshi. 2018. Hardware Performance Counters Can Detect Malware: Myth or Fact?. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security (Asia CCS)*. ACM, New York, NY, USA, 457–468.

[79] Y. Zhu, J. J. Lim, and L. Fei-Fei. 2017. Knowledge Acquisition for Visual Question Answering via Iterative Querying. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 6146–6155.