Forensic Science International: Digital Investigation 45 (2023) 301564

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS 2023 USA - Proceedings of the Twenty Third Annual DFRWS Conference

cRGB_Mem: At the intersection of memory forensics and machine learning

Aisha Ali-Gombe [a, b, *], Sneha Sudhakaran [c], Ramyapandian Vijayakanthan [d], Golden G. Richard III [a, b]

[a] Center for Computation and Technology, Louisiana State University, United States
[b] School of Electrical Engineering & Computer Science, Louisiana State University, United States
[c] Department of Computer Engineering & Sciences, Florida International University, United States
[d] Department of Computer and Information Science, Towson University, United States

ARTICLE INFO

Article history:

Keywords:
Memory forensics
Android
Memory analysis
Malware
CNN
Machine learning

ABSTRACT

Mobile malware's alarming sophistication and pervasiveness have continued to draw the attention of many cybersecurity researchers. Particularly on the Android platform, malware trojans designed to steal user PIIs, crypto miners, ransomware, and on-device fraud continue to infiltrate the primary Google store market and other secondary markets. While much effort has been put in place by the research community and industry to curb this menace since 2012, malware authors have consistently found ways to circumvent the existing detection and prevention mechanisms. Largely this remains so because of the restrictiveness of the feature set used in building the current classification models. Thus, the overarching objective of this paper is to bridge the gap between static and dynamic analysis by exploring the use of in-memory artifacts generated from the concrete execution of Android apps for effective malware classification. Our proposed approach, called *RGB_Mem* trains RGB images generated from in-memory allocation patterns in a Convolutional Neural Network. The result of our classification algorithm achieved an accuracy of 95.98% for samples with known objects and 84.48% for samples with unknown features. These results indicate that artifacts recovered from post-mortem memory forensics can provide a new dimension for training Android malware classification. The post-execution features, which are not impeded by any obfuscation and hooking constraints, provide a more accurate characterization of an app and are, therefore more suitable for classification.

1. Introduction

In the past twelve years, numerous research efforts have been developed to address the challenges posed by Android malware, particularly those leveraging machine learning. Traditional targeted features include opcode values, permissions, API calls, etc., which are often hand-picked attributes chosen by experts. However, given the scale and sophistication of the malware problem, these features are often limited and cannot adequately distinguish between a malicious and benign app and/or execution. Hence in this paper, we propose a methodology that leverages in-memory objects and their allocation sequence as features in a Convolutional Neural Network Model. When applied in our proposed learning algorithm, these distinct feature vectors that individually represent the app's functionality and behavior will result in a more robust and resilient classifier that can detect malware variants with high accuracy. Our proposed approach, titled *RGB_Mem* generates feature vectors using a combination of memory acquisition, object recovery, and reconstruction. We reconstructed objects allocated by each target Android application during execution as an allocation sequence or pattern that maintains its objects' identifications (name) and allocation positions (address). These allocation patterns are then converted to RGB images to form a representation for each app in the dataset. Finally, the images are passed to a variant of the RGB-LED Convolutional Neural Network (Guan et al., 2019) for training and prediction. The evaluation result on our test dataset shows that our optimized model with known features achieved 95.98% test accuracy and 84.48% on the model with unknown

* Corresponding author. Center for Computation and Technology, Louisiana State University.
E-mail addresses: aaligombe@lsu.edu (A. Ali-Gombe), ssudhakaran@fit.edu (S. Sudhakaran), rvijay1@students.towson.edu (R. Vijayakanthan), golden@cct.lsu.edu (G.G. Richard).

https://doi.org/10.1016/j.fsidi.2023.301564

features, i.e., zero-day samples. We compared our accuracy with DexRay (DaoudiJordan et al., 2021) - an existing CNN model that utilizes opcode sequence-converted images as feature vectors for classification. The result indicates *RGB_Mem* outperforms DexRay's 79.20% accuracy in both the known and unknown feature models. This result demonstrates that our novel features selection and engineering, when combined with a CNN model like *RGB_Mem*, fare better for malware classification than traditional approaches.

Relevance to DFIR - Digital Forensics and Incident Response (DFIR) on mobile devices can be challenging. Its difficulty depends on various factors, such as the examiner's background, experience, types and versions of applications, devices, and the specific context of the incident being investigated. This research is particularly relevant in making it easy for examiners to detect if any of the processes running on a target Android device are malicious. A study conducted by AVG Technologies in 2015 found that the average number of active processes on Android devices was around 35 (AVG Technologies, 2015). This number was found to increase significantly when additional applications were launched, with some devices running up to 100 or more processes simultaneously. Another study conducted by researchers at Purdue University in 2017 found that the average number of processes running on iOS devices was around 45 (Chen et al., 2017). This number also varies widely depending on the specific device and usage patterns. Thus, going through each app's runtime activity and memory on a target device to explore their data, data structures, and code one after the other to determine if they are benign or malicious is a very time-consuming process and requires significant expert knowledge. However, with the proposed *RGB_Mem*, the examiner can quickly rule out malware with a high degree of accuracy for both known variants and zero-day samples. In summary, the contributions of our research are as follows.

- The development of new features and feature engineering for Android malware classification that leverage memory allocations to generate RGB images as a representation of an app execution sequence.
- The implementation of an enhanced *model* that takes the RGB images as inputs and classifies them as malware or benign apps.
- The curation of an open-source repository of memory images and RGB images, which can be utilized for Android malware research.

The rest of the paper is organized as follows: Section 2 provides a background on memory analysis; Section 3 presents the main design idea of our proposed technique; Section 4 details the implementation and Section 5 provides the evaluation and experimental validation of the proposed approach; Section 6 discusses the summary of related work; Section 7 summarizes our findings and conclusions.

2. Background and motivation

In recent years, the idea of leveraging memory analysis for cyber threat investigation is consistently gaining ground. Tools like Volatility have extended their capabilities beyond browser history recovery, for instance, to identifying memory segments containing potential malware code or hook tracing structures. In addition, the preliminary work by Hussaini et al. showed that memory artifacts and patterns could potentially indicate application maliciousness (Hussaini et al., 2021). Hence, in this paper, we aim to explore further and extend the capabilities of memory forensics to include generating features for malware classification. By extracting and analyzing these artifacts features, it is possible to classify malware and determine its behavior. One way to do this is by using machine learning techniques to train a model on a dataset of known malware and benign samples and then using that model to classify new samples based on their in-memory artifacts. Compared with static analysis, memory forensics is not typically affected by malware obfuscation and thus can examine and extract functionalities executed by a target app at runtime. Unlike dynamic analysis, on the other hand, postmortem memory analysis for the purpose of feature extraction does not interfere with the target process or modify its execution environment, allowing the malware to exercise its payload fully.

To illustrate the challenges of traditional feature extraction and analysis methods, consider the Plankton malware - one of the first Android malware to use dynamic class loading. Extracting static features such as API calls using existing static analysis tools like Androguard (Anthony and Gueguen, 2013) from this obfuscated sample will result in fewer non-useful features that will negatively affect the learning model. Naturally, we pivot to dynamic analysis when faced with such a scenario. With the dynamic technique, we first have to set up a monitoring/instrumentation environment such as CuckooDroid to hook up all API calls and intermittently hijack execution to perform logging. Here are its two significant drawbacks - environment setup is version dependent, and currently, CuckooDroid, for instance, only supports up to Android v4. Secondly, Plankton has more than 10,000 APIs; hooking them all will result in substantial performance costs and might result in the process termination by the Android system. For instance, to trace all the 10,000 APIs, at minimum, we will inject three new instructions to each call - the trace entry, exit, and logging instructions. This additional instruction will result in about 200% instruction overhead. Given that an instruction overhead of less than 10% is considered ideal for Java method profiling, a 200% increase will tremendously impact CPU and memory usage and likely crash the target app or make it infeasible to execute. Moreover, this change in execution latency can be used by malware for anti-analysis. In contrast, leveraging memory forensics for acquisition and feature extraction of Plankton in-memory artifacts will have very negligible overhead since only the acquisition process will be carried out while the app is running, whereas the extraction process will be wholly offline. More so, this technique is not affected by dynamic class loading or other known traditional obfuscation techniques.

2.1. Android in-memory artifacts

Android applications are primarily written in Java and execute in a runtime engine called the Android Runtime (ART). The primary function of the runtime environment is to provide a tight and unique execution sandbox for each process on the system. Google designed the base runtime as a large structure holding important process-level information such as types and addresses of memory mappings, pointers to different memory segments, process state, thread information, and other relevant process statistics. Thus, as a process executes, it reads and writes data and code in its memory regions within the runtime environment. Furthermore, the operating system also writes to the process memory copies of shared libraries and any IPC objects sent to the process. Runtime object allocations on Android use the Region-space memory allocation algorithm (Ali-Gombe et al., 2019). This algorithm divides the available process virtual address space into smaller regions of 256 KB. Objects are allocated sequentially, starting from the first available region, except for specific threads that request a Thread Local Allocation Buffer (TLAB). Those are allocated in specifically tagged regions called the TLAB Regions. Thus, from a memory analysis perspective, the type of data allocated within a region and its allocation order or pattern can be invaluable in determining the runtime structure of an Android process and, by extension, the

functionality of an app and how that functionality is executed. Hence, this research explores a methodology that leverages this unique process allocation pattern in Android malware classification. To this end, we seek to answer the following research questions.

- R1: Can the proposed approach detect Android malware with known features?
- R2: Can the proposed approach detect Android malware with unknown or zero-day features?
- R3: What is the effectiveness of using in-memory allocation patterns compared to traditional semantic artifacts such as opcode sequences for malware classification?

3. Design overview

This research designs a methodology for Android malware classification leveraging in-memory artifacts. The workflow of our system, as shown in Fig. 1, is made up of three phases: 1) Feature extraction, 2) Image generation, and 3) Classification. The feature generation phase involves sourcing and acquiring runtime features or in-memory artifacts as allocation patterns from a running Android app. The image generation phase then takes the extracted app's allocation pattern and creates an image representation using the RGB color scale. Finally, the classification phase takes the generated image as input vector into a neural network model for binary classification.

3.1. Phase 1 - Feature extraction phase

In this phase, we designed a runtime environment to execute target Android applications for in-memory artifact extraction. Each target app is installed, activated, and exercised before its memory is imaged. Our environment is simulated with real mobile-based data such as contacts, SMS, GPS simulations, files in the SD card, and sample images in the gallery. Furthermore, we configured the execution environment with additional real data, such as an actual IMEI instead of zeros, etc. to thwart common analysis detection

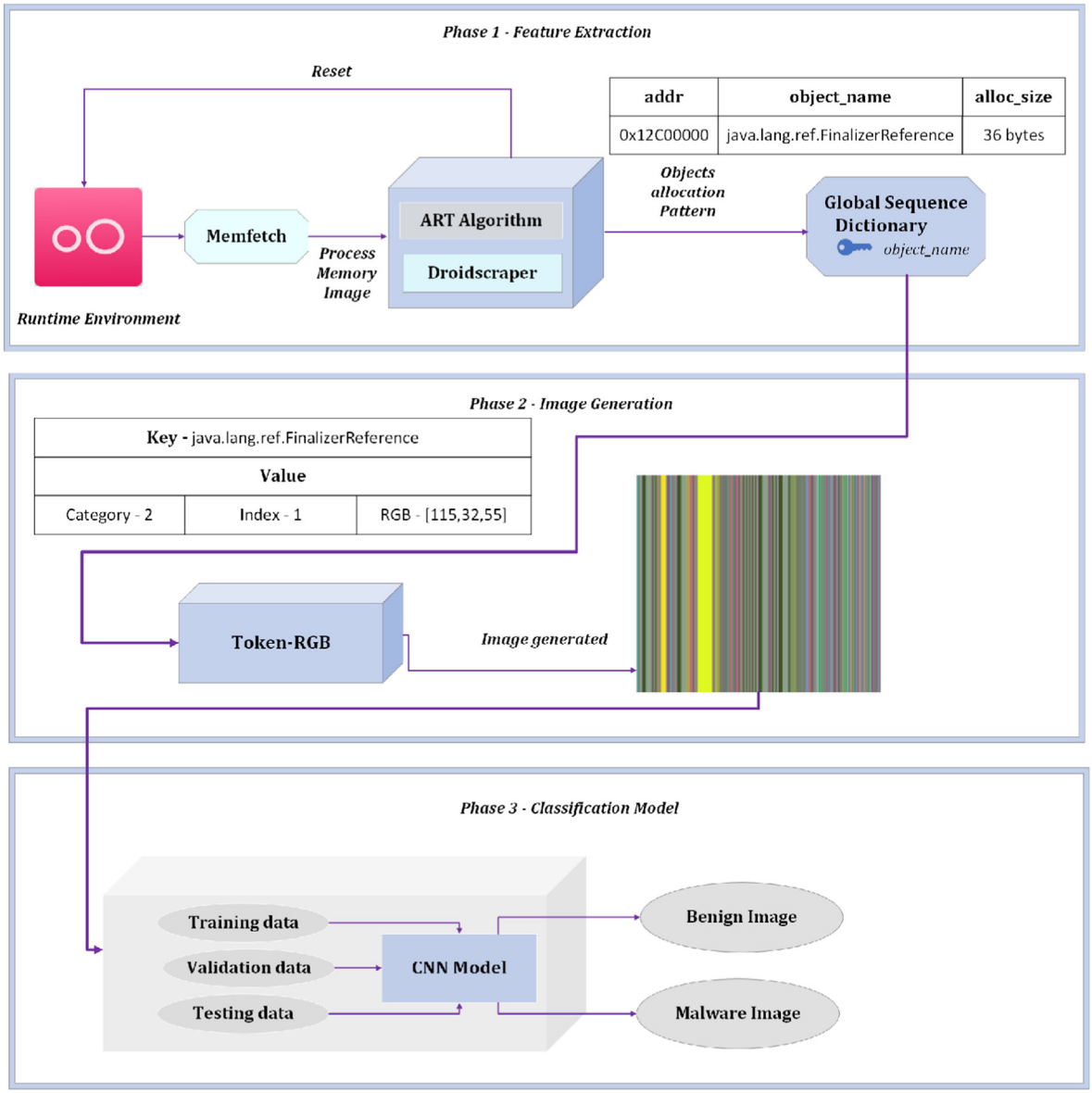

Fig. 1. Proposed design workflow diagram.

schemes. After every app execution, we kept a snapshot of the clean configuration to restore the runtime environment to an uninfected state. While it would have been ideal to collect the data on a real Android device rather than an AVD, however, going through cycles of infection, memory acquisition, snapshot restoration, and reinfection on a real device is infeasible.

For the memory acquisition, a per-process memory dumping tool called Memfetch, which was cross-compiled for Android, is executed in the runtime environment, targeting the app under execution. The output of memfetch (the process memory image) is then copied from the emulator to our analysis machine. The clean state of the emulator is restored for the next app execution. Our choice of process-level memory acquisition is two-fold - 1) given that Android apps execute within their own instance of the runtime, the allocations, and memory mappings for both private and shared processes are carried out within that specific instance; and 2) we can leverage the ART object allocation algorithm to determine allocation regions along with its metadata and the allocated objects. Each acquired memory image thus represents a single execution snapshot for a unique app. This is especially crucial because our goal is to create a resilient model that can detect Android malware *without* knowing all of its features. Hence the model needs to identify distinct generic behaviors in malware and benign Android apps such that when there is zero-day malware, this model can effectively detect it even when all of its features are not known.

Next, each acquired process memory image is analyzed using an updated version of DroidScraper (Droidscraper, 2019) to retrieve the objects allocated during its execution. These objects are later used for generating the feature set for the model training, validation, and testing. Droidscraper is a userland in-memory object recovery and reconstruction tool that recovers the runtime artifacts from Android process memory space. Our choice of Droidscraper as an artifacts recovery tool is motivated by the following reasons.

1. It is the only publicly available tool that parses the region-space allocator and recovers runtime in-memory artifacts from newer Android devices that run ART instead of the older Dalvik runtime.
2. Droidscraper recovers spatiotemporal data from memory allocation regions, meaning the output of its Heapdump plugin dumps objects in the order in which they are allocated, i.e., in regions and sequentially, which translates to the concrete execution flow of the program under analysis. The ability to train a classifier to identify execution patterns is especially crucial since malicious Android apps, based on their characterization (Zhang et al., 2020), often perform similar functions, such as sending SMS or accessing contacts, and are thus likely to have similar execution patterns.
3. Data collection with Droidscraper does not involve building or instrumenting an engine into the emulator or the device and hence it is not associated with any resource overhead (memory or CPU usage) or memory modification during app execution.
4. Unlike traditional methods for dynamic analysis, leveraging Droidscraper does not require expert knowledge of where and what to instrument and log; all that is needed is the snapshot of the target process memory.
5. Related literature has established that Android malware variants are often created mainly by transforming existing samples using obfuscation mechanisms (ZhangFrank and LuechingerStephen, 2021) or the repackaging of malware code into benign applications (Tian et al., 2017). Thus, recovered in-memory execution patterns as features will likely fare better in malware classification than static features.

Thus, in this tool's updated version, we extract objects sequentially by resolving the address and index of every allocation region, which is provided as fields in the *region*'s struct. Starting at offset 0 in each region, we read the first couple of bytes to resolve the object class, name, and size and then add the size to the current offset to get the next object. This step, as shown in Algorithm 1, is repeated for all of the allocated memory regions. The result is an allocation list (also called the allocation pattern) made up of 3-tuple values - the address, object name, and object size.

Algorithm 1. Resolving object allocation patterns for the memory snapshots

Algorithm 1 Resolving object allocation patterns for the memory snapshots

$corpus \leftarrow List()$
for $mem \in mem1, mem2....., memn$ **do**
  $runtime \leftarrow Droidscraper.getRuntime(mem)$
  $heap \leftarrow Droidscraper.getHeap(meme)$
  $regions \leftarrow getRegion(runtime, heap)$
  $sort(regions, regions.index)$
  for $region \in regions$ **do**
    $P\_region \leftarrow List()$
    $endOfRegion \leftarrow region.size$
    $current \leftarrow seekRegion()$
    while $current \neq endOfRegion$ **do**
      $objAddr \leftarrow current + region.addr$
      $classOff = read(4)$
      $name, size \leftarrow resolve(classOffset)$
      $P\_region \leftarrow [objAddr, name, size]$
      $corpus \leftarrow P\_region$
      $current \leftarrow current + size$
    end while
  end for
end for

To illustrate this dataset, we assume an app *a* with *n* allocated objects at the time of memory acquisition. Its allocation pattern P_a is a 3-tuple list of all live objects currently in its runtime memory:

P_a = [[$addr_1$, $object_1$, $alloc\_size_1$], [$addr_2$, $object_2$, $alloc\_size_2$], …, [$addr_n$, $object_n$, $alloc\_size_n$]]where $addr_n$ is the allocation address, e.g., 0x12C00000, $object_n$ is the object name, e.g., java.lang.ref.FinalizerReference, and $alloc\_size_n$ is the allocation size, e.g., 36 bytes.

3.2. Phase 2 - image generation phase

Now that we have collected all allocation pattern lists for our target applications, the next task is to process them into a suitable input for machine learning. Given that this research proposes a CNN-based model, the objective in this phase is to convert the allocation list into an image representation. The options for creating image representations are either grayscale or RGB images. However, grayscale can only represent 256 unique colors from 0 (black) to 255 (white), while RGB can represent up to 16777216 (256 × 256 x 256) unique colors. Considering that object-oriented programming languages such as Java tend to have many diverse object types, RGB image representation is more suitable. Before parsing each input allocation pattern list to create an image, we have to aggregate all the unique objects in the dataset into a repository called the *Global Sequence Dictionary*. Like the Term Dictionary in document classification, this dictionary serves as a global

repository for all unique objects such as *android.os.HandlerThread*, *java.lang.string, com.google.android.gms.analytics.internal.zzag etc.* in our dataset. An entry in the dictionary, which we call a **token**, is mapped to a unique object name as the key and a 3-tuple item that holds the object's category, a unique index key, and a unique RGB value.

Algorithm 2. Creating the sequence dictionary

Algorithm 2 Creating the sequence dictionary

```
index ← 0
for l in list1,list2.....,listn do
    cat ← l.getCat()
    for object in list do
        if ! sequenceDict[object] then
            rgbVal ← randRGB(index, cat)
            sequenceDict[object] ← (cat, index, rgbVal)
            index ← index + 1
        else if sequenceDict[object].getCat() ≠ cat
then
            sequenceDict[object] ← cat
        end if
    end for
end for
```

As shown in Algorithm 2, we traverse all the object allocation patterns in our dataset consecutively to retrieve all unique tokens. We check every object during our traversal to see if it exists in our sequence dictionary; otherwise, we add a new entry and assign a new index and RGB value for the object. Furthermore, a category of enum data type is added to the entry's item based on whether the object is found in malware only, benign apps only, or both. It is important to note that, like in Document classification, no hard and fast rule dictates whether the dictionary of words should only contain words that are present in the training set but not the testing set. However, building the dictionary based on the full corpus, including both the training and testing sets, is generally recommended. According to the textbook "Speech and Language Processing" by Dan Jurafsky and James H. Martin, "the vocabulary used for classification should be derived from the entire collection of documents, not just the training documents, to avoid overfitting the vocabulary to the training data" (JurafskyJames and Martin, 2009). Similarly, the PyTorch documentation for the Torchtext library, which provides tools for text data preprocessing and modeling, recommends building the vocabulary based on the entire corpus: "It is important to build the vocab using only the training set, and not to include any words from the validation and test sets. However, it is still important to ensure that the vocab is built from the entire corpus and not just the training set to avoid encountering out-of-vocabulary (OOV) words at inference time" (PyTorch, 2021). Thus, based on the aforementioned reasons, we built our Global Sequence using all the tokens in our entire corpus.

The final sequence dictionary consists of tokens in three categories: those exclusively in benign apps, those exclusively in malicious apps, and overlapping tokens (present in benign and malicious apps). An example of sequence dictionary token entry is {*java.lang.ref.WeakReference*: *[2, 66, [115,32,55]]*}, where *java.lang.ref.WeakReference* is the token name which also serves as the key; 2 indicates that the token appears in both malware and benign apps; *66* is its index value and *[115,32,55]* is the computed RGB value. After this initial data processing, the sequence dictionary is now an aggregated list of all unique tokens mapped to their unique RGB values.

To create an image representation for an app, we traverse its allocation pattern, decoding each allocated object (token) into a color by looking up its RGB value in the sequence dictionary. Thus, each final image output consists of a band of colors arranged column-wise sequentially following the original in-memory allocation addressing. For example, suppose the sequence dictionary has the following entries: (*java.lang.String*, $Index_1$, RGB_1), (*java.lang.Float*, $Index_2$, RGB_2), …, (*java.lang.Thread*, $Index_{100}$, RGB_{100}), …, ($Token_n$, $Index_n$, RGB_n) and an App **a** has an allocation pattern P_a as [[0x0, *java.lang.String*, 12], [0x*C*, *java.lang.Thread*, 36], [0x30, *java.lang.String*, 24], [0x48, *java.Lang.Float*, 24]],then the image I_a will be represented as (RGB_1, RGB_{100}, RGB_1, RGB_2). For better visualization, we assigned RGB values for benign-only tokens toward the red side of the spectrum (red, yellow, green), and the malicious-only tokens were assigned toward the blue side of the spectrum (blue, purple, and magenta). The RGB values for the overlapping tokens are at the intersection of the spectrum.

3.3. Phase 3 - Classification phase

A Convolutional Neural Network (CNN) is an artificial neural network designed for learning and classifying structured data such as images. CNN is a multi-layer network with convolutional filters trained using the backpropagation. These layers are arranged to detect simpler patterns, such as lines, curves, colors, etc., first, then more complex designs, such as faces and objects. In recent years, CNN has been widely adopted as the base algorithm for general malware classification and Android in particular. To the best of our knowledge, the inputs to the different CNN models in the related literature are either code sequences or other semantic features extracted using static and/or dynamic analysis. In contrast, this paper aims to train a CNN model to take our generated RGB images as inputs and classify them either as representations of malware or benign apps. To this end, we created a network similar to the RGB-LED (Guan et al., 2019). The RGB-LED network is designed as a CNN classifier that learns to uniquely detect and recognize different LEDs in Visible Light Communication (VLC). We chose this CNN model as a classification algorithm primarily due to the uniqueness of our inputs being RGB images and since the allocation patterns for which the input images were created are spatial−temporal, i.e., it exist in both space (the type of data allocated in each address) and time (the order of allocation or feature sequence). Thus, we find the RGB-LED-like CNN model to be a suitable fit.

As shown in Fig. 2, our model is a sequential CNN container comprising 2-stacks of a convolution layer and a max pooling layer, a final convolution layer, a flattening layer, and a linear transformation layer. Our inputs are passed to the model as 200 × 200 x 3 pixels images. A kernel size of 3 × 3 is used in the first two convolution layers for feature extraction, and then a 2 × 2 kernel size in the two max pooling layers for downsampling the feature map. The features are then passed to the last convolution layer, which has a kernel size of 25 × 25, to scale down the feature dimension, after which the down-sized features are flattened and linearized. For prediction, our network uses the Argmax function for the binary classification of the input images.

4. Implementation

We implemented our approach as a system with two modules for 1) Data Engineering and 2) CNN Module. The data engineering is a helper module for feature generation, selection and engineering while the CNN module implements the *RGB_Mem* algorithm.

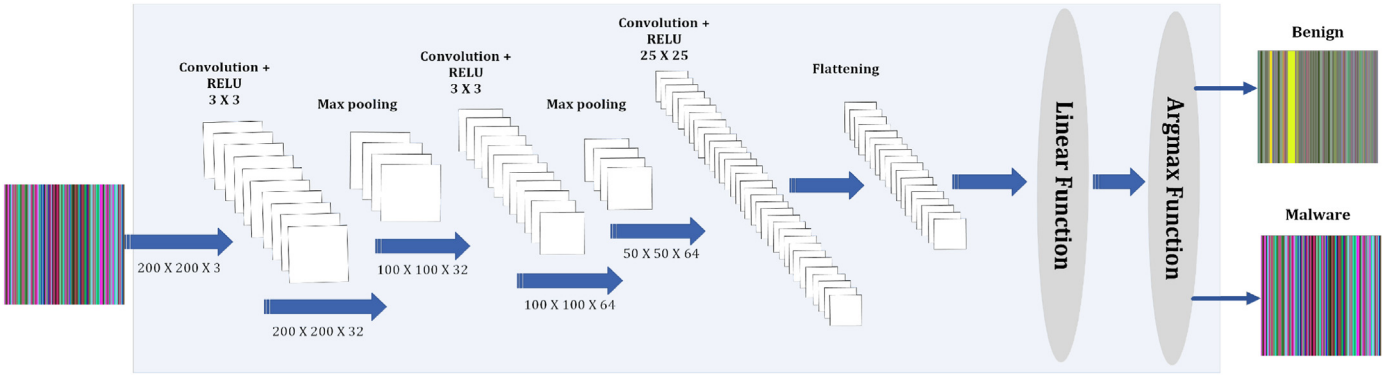

Fig. 2. The architecture of the proposed CNN model.

4.1. Data engineering module

In last decade, there has been significant effort by the research community in developing malware feature datasets such as MIST (Ramilli, 2016), which researchers utilized for developing classification and detection algorithms. Nonetheless, these datasets were generated from pure static or dynamic analysis. To our knowledge, there is no publicly available in-memory object artifacts or memory snapshot repository for Android. More so, dynamic artifacts datasets are generally hard to generate for the reasons itemized in section 3.1. As such, the first module in our system is the data engineering module. This module includes sub-modules for automated app execution and memory acquisition called *ADBAutomate*, a module for sequence dictionary generation, and a module for image creation.

Using ADBAutomate, we executed 850 Android malware and 600 benign apps within two semesters in 2021 with the help of five undergraduate students on different instances of customized Genymotion images. The malware samples were downloaded from VirusShare, and the benign apps were sourced from AndroZoo (Kevin et al., 2016). The memory images of each app execution were acquired one-time using Memfetch. It is important to note that memory is one of the important components of a system that security-wise is unlikely to be affected by traditional obfuscation, thus making its content very valuable for any kind of analysis. However, its acquisition is arduous and precarious; hence, we had to take extra measures during the acquisition process. First, we had to use an emulator instead of real devices, mainly because we had to re-image with a clean state after every app execution, both for malware and benign samples. Secondly, before the acquisition process is started, we check for evidence that the process is active and in the foreground, and after the acquisition, we repeat the same sanity check. This validation is vital because apps moved to the background are more likely to be garbage collected, which may adversely affect the training data. Secondly, because the acquisition process is very intrusive, there are chances of memory corruption in the target process address space, which could result in restarting the process. Thus, a process that is killed and restarted before the acquisition is complete will have to be restarted, re-exercised and reacquired.

With this system setup and the automated utility, we acquired 1411 memory images (823 malware and 588 benign) out of the 1450 apps in our corpus. These apps were correctly executed, exercised, acquired, and reconstructed without error. The acquired memory images are then pre-processed with Droidscraper to extract their allocation patterns. These allocation patterns are fed to the sequence dictionary generation module, which creates a dictionary of tokens, each with a 3-tuple value for the app category, an index, and an RGB value. After the initial data processing, the sequence dictionary contains 18,659 unique tokens, broken down into 9191 benign-only tokens, 6986 malware-only tokens, and 2479 combined tokens, i.e., tokens present in both malware and benign allocations. Using the sequence dictionary as a look-up table, we decode each object in the 1411 allocation patterns into their RGB values. The resulting RGB sequence for each allocation pattern is then plotted as a single image. After this initial conversion, the images in the dataset are of variable sizes. For the training and evaluation, the model requires all input images to be the same size; hence resizing the images before feeding them to the model became necessary. We leverage the resize function of the Torchvision-transforms class to resize all the images in the dataset to a fixed size of 200 by 200. Finally, our dataset is now made up of 1411 RGB images, all of the same size.

4.2. RGB_Mem Classification module

We implemented the 2-class CNN classifier illustrated in the previous section using the Python PyTorch library (Adam et al., 2019). Additional hyper-parameters such as loss function, optimization function, and epoch were fine-tuned during the model implementation. Our network uses a cross-entropy loss function to minimize loss and an Adam function with a learning rate of 0.001 for model optimization. We choose the Adam algorithm over SGD mainly because of the size of our features and the nature of the allocation patterns, which may contain noisy data. We split the generated image dataset into 70% training, 16% validation, and 14% testing to achieve. The model uses the samples in the training set to identify patterns specific to benign and malicious memory allocation lists. Validation data is used to fine-tune the algorithm's parameters and protect the model from over-fitting the training data. In validation and test data, if a particular token is not found in the global sequence dictionary, such tokens are labeled as unknown tokens and are removed from the pattern list during the image generation. The test images are used to evaluate the model's classification accuracy and other associated metrics.

4.2.1. RGB_Mem model optimization -

After generating the initial sequence dictionary, the 1411 image dataset, and the *RGB_Mem* model, we examine if all the tokens currently in the sequence dictionary are useful to our classifier, i.e., the tokens are not redundant. We trained the model with the training set and optimized it with the validation set. Setting the epoch to 20, we compute each epoch's training and validation loss, and the corresponding optimization learning curve is plotted as shown in Fig. 4. This **optimization learning curve**, an essential tool for identifying and selecting the optimal number of features, shows

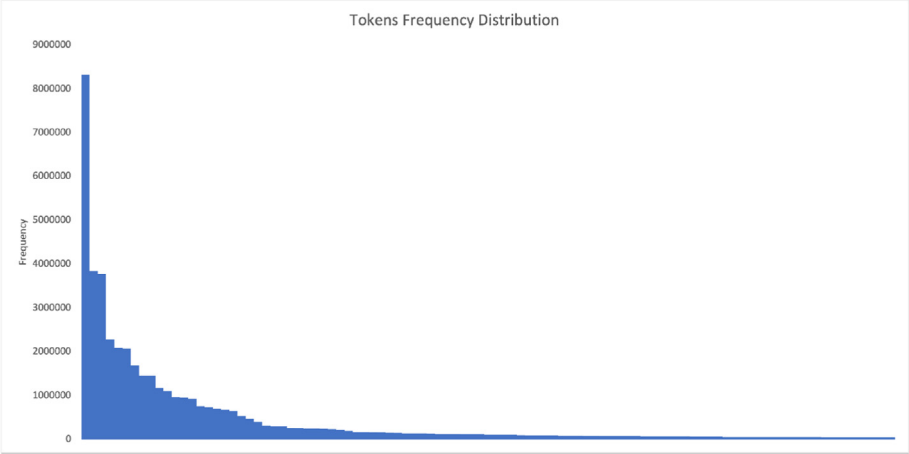

Fig. 3. Token frequency distribution chart.

a noisy validation loss and somewhat flat training loss. This indicates that the model could not learn from the training dataset mainly due to noisy and redundant data.

Furthermore, the images in Fig. 5 visibly illustrated that malware and benign images generated from the noisy tokens are indistinguishable. Hence, we need to employ additional feature selection criteria that enhance the model's performance. To address this, we employ a **Recursive Feature Elimination** method that eliminates features based on their frequency of occurrence in the entire sample set. From the frequency distribution in Fig. 3, it is clear that some tokens have excessively high frequencies, especially in the overlapping category. In contrast, others have moderate to low frequencies, especially in the malware category. More so, the average mean of the token frequency is about 4,159,479, with the highest frequency token being java.lang.String with a frequency of 8,318,957, while more than 17,000 tokens have frequencies less than 100,000. The token frequency distribution is clearly unequal, and tokens such as java.lang.String adds no value to the model.

As such, we begin the elimination method by removing all tokens with a frequency greater than 2000 from the sequence dictionary. This process decreases the size of the sequence

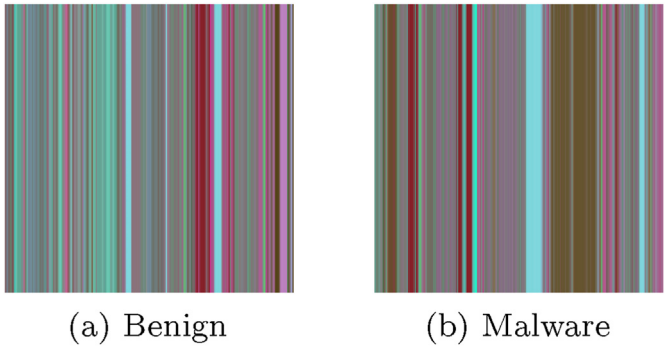

Fig. 5. Image representation for a benign and malware before dimensionality reduction.

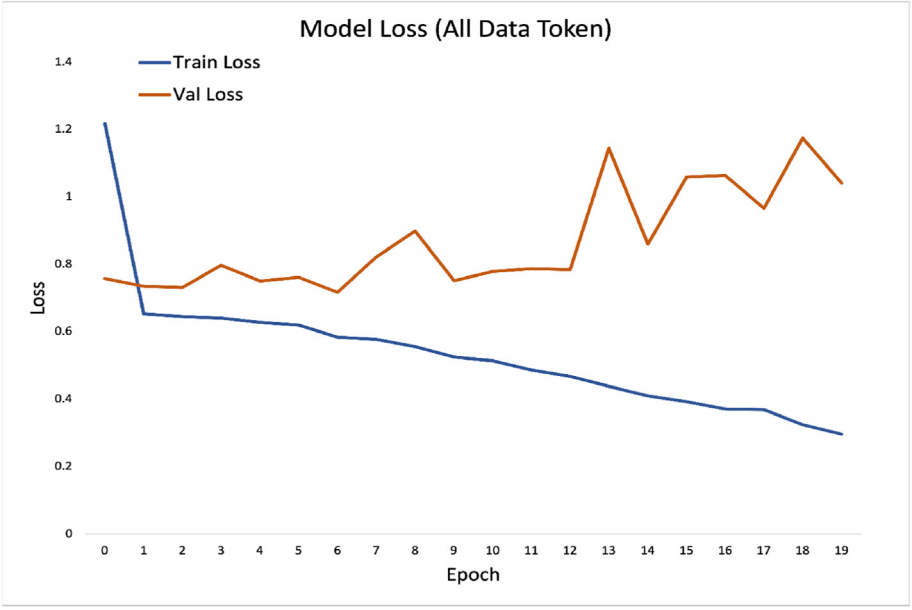

Fig. 4. Before dimensionality reduction.

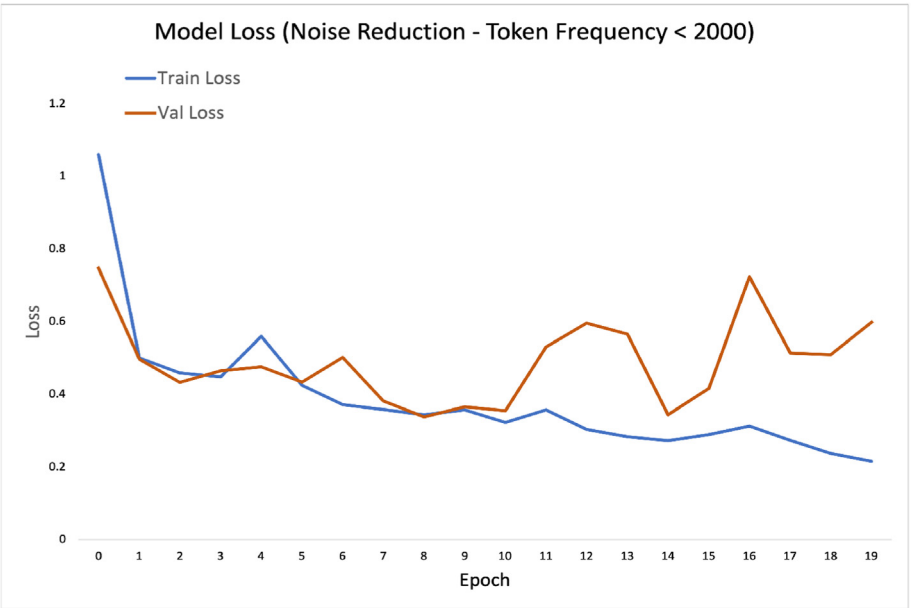

Fig. 6. Dimensionality reduction with object frequency less 2000.

dictionary to 17,917, out of which 1740 are overlapping tokens. We then re-generate the RGB images and re-execute the CNN model with the same initial hyperparameters. As shown in Fig. 6, the model has begun to learn our training dataset, but the noisy validation loss, especially at the end of the epoch, shows the model is still underfitting. Fig. 7 further shows that the malware and benign images generated with the reduced tokens after eliminating some redundancy are somewhat visible and distinguishable (see Fig. 8).

Next, we repeated the same elimination process for tokens with more than 1000 frequencies. This further reduces our sequence dictionary to 17,666 total tokens with 1489 overlapping tokens. The plot, as shown in 8, indicates the best optimization curve compared to the remaining two. Both the training and validation loss curves decrease and stabilize with a slight increase in validation loss compared to training loss towards the end of the epoch. The optimization learning curve for the training and validation loss shows that the model fits both the training and new data with the reduced features (frequency $<$ 1000). This result is further illustrated by visualizing the image generated with these reduced tokens. Fig. 9 shows that the malware and benign RGB images generated with the reduced tokens after eliminating highly frequent and redundant tokens have more distinct colormaps and patterns. Thus, we conclude that the model performs better when redundant features with high frequencies are eliminated.

Now that we have an optimized model, the next task is to test and evaluate our model.

5. Evaluation

To evaluate the effectiveness of our approach, we tested our model on the following objectives.

- RO1 - Android malware detection based on known features - can the model correctly classify an app whose allocated objects were part of the sequence dictionary?
- RO2 - Android malware detection based on unknown features - can the model correctly classify an app in which some of its uniquely allocated objects are not part of the sequence dictionary?
- RO3 - Comparative analysis with state-of-the-earth Android malware detection tools - how effective is the proposed approach compared to existing analysis systems that leverage traditional features such as code sequences, API calls, control flow graphs, etc.?

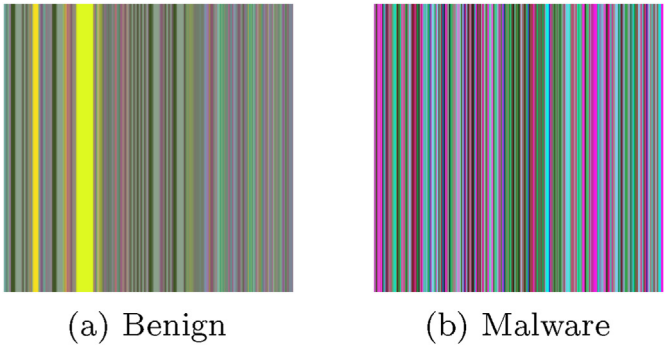

Fig. 7. Image representation after dimensionality reduction with token frequency less than 2000.

5.1. Experiments

We conducted three different tests to evaluate these objectives. All our experiments were conducted offline on a MacBook Pro with 2.9 GHz Quad-Core Intel Core i7 and 16 GB memory capacity.

1. **RO1 - Android malware detection based on known features -** In this experiment, the goal is to examine the performance of our model in terms of F1 score, model accuracy, precision, and recall. The sequence dictionary used by this model has a total of 17,666 tokens (1489 overlapping, 9191 benign-only, and 6986 malware-only tokens). These tokens were used to create 1411 image representations consisting of 823 malware images and 588 benign images. We trained the model with 70% of the data and 16% was used for validation. The performance learning of

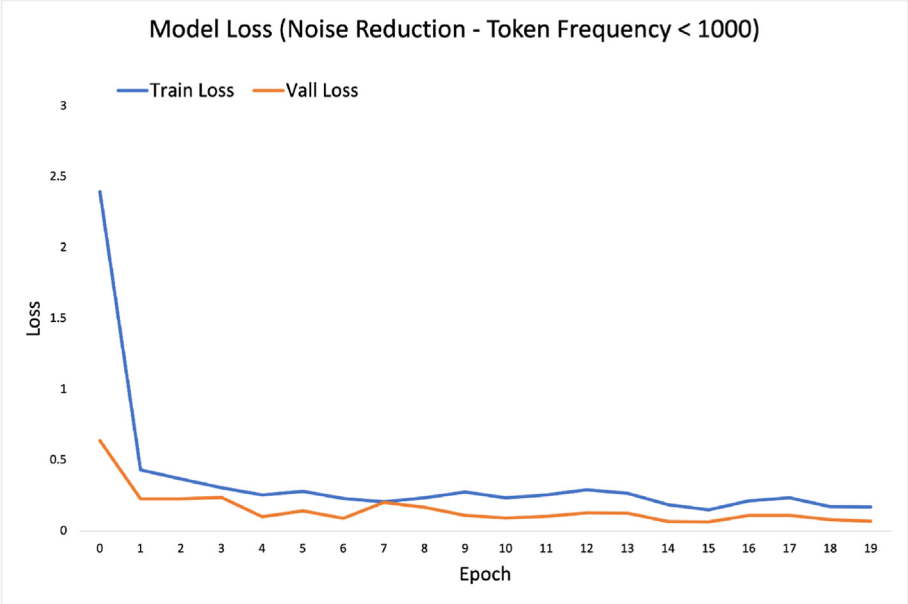

Fig. 8. Dimensionality reduction with object frequency less 1000.

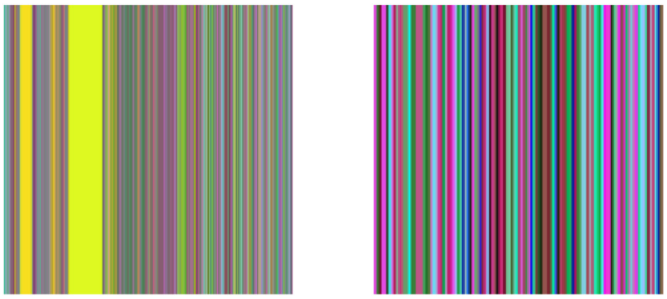

Fig. 9. Image representation after dimensionality reduction with token frequency less than 1000.

this model indicates that the model achieves up to 97% validation accuracy.

We then tested the model on the remaining 14% test RGB images. Note, the test images in this experiment may contain previously seen or unseen features, thus we consider this evaluation as variant detection capability. From the confusion matrix in Table 1, our model achieves an accuracy of 95.98% with an F1-score of 95.24%, 95.89% precision, and a recall rate of 94.59%. This result strongly suggests that with a single memory scan at a random time, our classifier can correctly classify malware and benign apps more than 95% of the time with a malware detection rate of about 94.59%. Furthermore, the feature extraction process is not hindered by obfuscation nor is it affected by resource constraints or human knowledge of what feature to extract and how to extract them. We believe this approach opens a new dynamic to Android malware classification, which, when transferred into an online-realtime detection, can significantly improve the security posture in the Android mobile environment.

Table 1
Confusion matrix for R0.

[[97 3]
[4 70]]

2. **RO2 - Android malware detection based on unknown features -** In the second research question, the goal is to generate the sequence dictionary with only tokens from the training set. This experiment aims to establish if the classifier can detect zero-day malware designed with new data structures and objects unknown to the sequence dictionary. Thus, our validation and test images will only have tokens present in the training set. In this experiment, we still employed the frequency-based dimensionality reduction of < 1000. The total size of our new sequence dictionary is 13,458 (with 1308 overlapping, 6400 benign, and 5750 malware), indicating 4208 fewer tokens. New images are generated from the RGB values of the tokens in the sequence dictionary and then passed to the same model as input. We divided the samples into the same training, and validation split as R0. The performance learning of the model achieves up to 86.9% validation accuracy. When applied to the test dataset, the *RGB_Mem* model achieved an accuracy of 84.48%, with a precision of 81.33%, a recall rate of 82.43%, and an F1-score of 81.88%. The confusion matrix is shown in Table 2. This result indicated that the model could detect unknown, zero-day malware 84% of the time. While the *RGB_Mem* model did not do as well compared to the testing in R0, we believe the training could be improved with an additional dataset. More so, given that Android malware are known to heavily use repackaging, a more comprehensive sequence dictionary can improve the classifier's detection accuracy.

3. **RO3 - Comparative Analysis -** In this last experiment, we compare the performance of our model with the state-of-the-art tool that was published recently and had publicly available code. We aim to select a tool that leverages the same CNN model on different feature sets and performs similar feature engineering compared to our approach. To this end, we leverage the work of (DaoudiJordan et al., 2021) titled *DexRay* which was published in 2021. DexRay converts the Android app's bytecode

Table 2
Confusion matrix for R1.

[86 14]
[13 61]

Table 3
Comparative analysis with DexRay

Tool	Accuracy	F1-Score	Precision	Recall
DexRay	79.30	83.49	85.90	81.21
RGB_Mem Known features	95.98	95.23	95.89	94.49
RGB_Mem Unknown features	84.48	81.88	81.3	82.48

into grey-scale images. These images are then fed into a 1-dimensional Convolutional Neural Network model for training and testing. The published article shows DexRays achieves an F1-score of 96% when evaluated on 158k apps. DexRay's code is available on GitHub (TruX Trustworthy Software at University of Luxembourg). The similarity between our approach and theirs is that both converted features into images which are then used as input to a CNN model. Hence, this is a more accurate and fair comparative analysis. We use the same 1411 apps utilized in our dataset to generate DexRay's images and train its classifier. Their trained model on the same dataset achieved an accuracy of 79%, with 86% precision and 81% recall rate. The model has F1_score 83%. This comparative result, as shown in Table 3 indicates that our classifiers (with known and unknown features) outperform their model's accuracy and recall, even though our model has very minimal training data.

5.2. Discussion

The experiments conducted in R01 and R02 show that *RGB_Mem* can correctly classify Android malware with high accuracy, especially for known variants. We examined the three malware misclassifications and four benign false negatives in R01 and found a common trend - partial execution. Although the app process was not terminated during execution, and the foreground activity continued running, the background services were inactive. For instance, the malware cff7b956f043124a71f6973d8e34770f, which is an SMS trojan by design, has a very minimal footprint. Because that SMS functionality was not activated during execution, the recovered allocation has only a very scanty overlapping pattern and is hence categorized as benign. Likewise, the execution of the other three samples - 9d3bba4d77baef885f8e63fa036b7228, 41f3a952454c4ae5740566aed58d2436 and 25f222a1cf4feaaa7b3-dee43d4de7191 did not perform their background functions and thus no connection were made to their servers. More so, the RGB colors for the three benign apps after the image generation were visually only in the overlapping category and thus were also misclassified by *RGB_Mem*. For R02, even though an 84% accuracy will still be considered acceptable for zero-day malware classification, our conclusion after visualizing the samples is that most of the false positives and negatives fall within the overlapping RGB colors range. This is primarily because their new tokens were unknown to the sequence dictionary and misclassified. Hence, as part of future work, we plan to adopt continual reinforcement learning so that the model can adapt and learn new and unseen tokens during execution.

For R03, our detection accuracy outperforms an existing comparative model with static features. Overall, in terms of feature selection and engineering, the proposed approach explores a novel methodology that may change the community's approach to malware classification. Much like how memory forensics improves incident response with tools like Volatility's Malfind, APIHook, and Hooktracer, *RGB_Mem* can potentially improve malware classification.

5.2.1. Limitation -

Nonetheless, we also acknowledge that the proposed approach has the following limitations - 1) The size of the training and testing dataset- as discussed in section 3, there is currently no known repository for memory images and certainly for recovered allocation patterns. While this is undoubtedly an important limitation, our evaluation result shows that even with a limited dataset, *RGB_Mem* can learn from rich engineered in-memory features to correctly classify known malware variants with excellent accuracy and make a good effort for zero-day malware. 2) The detection accuracies for both R01 and R02 are not perfect. These metrics could be improved with an increased dataset and the introduction of continual reinforcement learning in the model. 3) Currently, the allocation pattern is determined solely by the allocation address, which may sometimes be inaccurate. Thus, as part of future work, we plan to resolve the allocations into an object allocation graph as proposed in the work of (Ali-Gombe et al., 2020). *RGB_Mem* will learn the allocation pattern with better and well-established allocation relation.

5.2.2. Future work -

In addition to utilizing an object allocation graph and adopting continual reinforcement learning, we plan to extend *RGB_Mem* into an online real-time learning system. The goal is to improve the model's accuracy and evaluate its performance on system resources as an on-device memory scanner.

6. Related work

6.1. ML-Based android Classification Based on static Features

Mahindru and Sangal proposed MLDroid, a web-based framework to detect malware in real-world apps. The model is trained using static features as inputs into four different ML algorithms (MahindruAL, 2021). The evaluation of MLDroid on more than 500,000 apps shows that the model achieves up to 98.8% detection accuracy. Nguyen Vu and Jung, on the other hand, proposed AdMat – a framework for characterizing Android applications by extracting the static API calls as nodes in a graph, which are processed into an adjacency matrix that serves as "input images" for the CNN model. The evaluation of AdMat shows that the algorithm adapts to various training ratios and achieves an average detection rate of 98.26% in different malware datasets. It also achieves a 97% accuracy for malware familial classification (Vu and Jung, 2021). Almomani et al. (2022) presented a comprehensive vision-based model for Android malware classification comprising 16 fine-tuned CNN algorithms. The evaluation of (Almomani et al., 2022) on DEBRIN and AMD malware datasets based on static feature analysis showed that the model achieves an accuracy of 99.40% for balanced samples and 98.05%. Other related works are (Hsien-De Huang and Kao, 2018; D'Angelo and PalmieriAntonio Robustelli, 2021; DaoudiJordan et al., 2021; Bissyandé and Klein, 2021; Ali-Gombe et al., 2015; Aisha Ibrahim Ali-Gombe, 2017), all of which leverage the machine and statistical learning-based models with static features for Android malware classification. While almost all the performance metrics of these related researches are commendable,

the use of static features such as opcode sequences and/or other semantic information like API calls, permissions, control flow, etc., have significant limitations. It is an established fact that even benign apps today employ some form of obfuscation to deter reverse engineering, and malware heavily relies on obfuscation to hide their behavior. Thus, the static feature extraction process used by these existing approaches is likely to be affected adversely by different obfuscation such as dynamic class, Java reflection, and encryption (Rastogi et al., 2013). On the contrary, this paper proposes a new dimension to feature engineering using in-memory object allocations and their allocation sequences as feature vectors for a classifier. Our performance metric shows that the proposed technique's accuracy is at par with the related works without the likelihood of our features being adversely affected by simple traditional obfuscation or transformation techniques.

6.2. ML-Based android malware Classification Based on dynamic/ hybrid Features

Rodrigo et al. proposed a hybrid machine-learning model named BrainShield for Android malware classification (Rodrigo et al., 2021). Brainshield is designed as three fully connected neural networks trained on the Omnidroid dataset containing more than 22,000 samples. The first neural network leverages 840 static features, the second network is trained on 3722 dynamic features, and the last is a hybrid network trained on 7081 static and dynamic features. The model evaluation shows that Brainshield achieves an accuracy of 92.9%, 81.1%, and 91.1% for the three networks, respectively. Other related Android machine-learning models that utilize either dynamic or hybrid features as input vectors to machine-learning models are the works of (Sihag et al., 2021; Alzaylaee et al., 2020; Faruki et al., 2019; Rasthofer et al., 2014; Ali-Gombe et al., 2016, 2018). The hybrid/dynamic features range from API call tracing, dynamic permissions, Intents, CPU, Memory, Network, sensor data system, and binder calls or their combinations. While the evaluation results of each of these related works posted more than 90% accuracy rate, compared with the proposed *RGB_Mem*, these traditional hybrid techniques are built purely on manual expert definition of the features, such as what API calls or system calls to trace? What intent to hook and log? Etc. As malware and the number of samples to analyze and classify continuously grow, an expert knowledge-based feature selection may not scale as much compared to our proposed feature engineering process. Moreover, setting and building a robust execution environment for dynamic feature extraction is prone to instrumentation and hooking overhead, thus limiting the amount and type of features to extract. In contrast, *RGB_Mem* takes a more holistic approach to feature engineering and selection by leveraging the in-memory footprint of the app as features without the drawback of resource overhead and/or expert knowledge.

7. Conclusions

This research presents *RGB_Mem* - a CNN model that leverages runtime in-memory artifacts for malware classification. *RGB_Mem* introduces the concept of memory analysis to extract and reconstruct object allocation patterns which are then transformed into RGB images before being processed by the CNN model. The evaluation of the proposed feature engineering on an RGB-CNN shows the model can adequately distinguish malware from benign executions with more than 95% accuracy in known variant detection and 84% accuracy for zero-day malware. Additional comparative analysis shows *RGB_Mem* 's accuracy surpasses the state-of-the-art algorithm that leverages static features on a similar CNN model.

Acknowledgement

This work is supported by the National Science Foundation (NSF) under Grant Number 1850054.

References

Adam, Paszke, Gross, Sam, Massa, Francisco, Adam, Lerer, James, Bradbury, Gregory Chanan, Killeen, Trevor, Lin, Zeming, Gimelshein, Natalia, Antiga, Luca, Desmaison, Alban, Kopf, Andreas, Yang, Edward, DeVito, Zachary, Martin, Raison, Tejani, Alykhan, Chilamkurthy, Sasank, Steiner, Benoit, Lu, Fang, Bai, Junjie, Chintala, Soumith, 2019. Pytorch: an imperative style, high-performance deep learning library. URL. In: Advances in Neural Information Processing Systems, vol. 32. Curran Associates, Inc., pp. 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

Aisha Ibrahim Ali-Gombe, 2017. Malware Analysis and Privacy Policy Enforcement Techniques for Android Applications.

Ali-Gombe, Aisha, Ahmed, Irfan, Richard III, Golden G., Roussev, Vassil, 2015. Opseq: android malware fingerprinting. In: Proceedings of the 5th Program Protection and Reverse Engineering Workshop. ACM, p. 7.

Ali-Gombe, Aisha, Ahmed, Irfan, Richard III, Golden G., Roussev, Vassil, 2016. Aspectdroid: android app analysis system. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy. ACM, pp. 145–147.

Ali-Gombe, Aisha I., Saltaformaggio, Brendan, Xu, Dongyan, Richard III, Golden G., et al., 2018. Toward a more dependable hybrid analysis of android malware using aspect-oriented programming. Comput. Secur. 73, 235–248.

Ali-Gombe, Aisha, Sudhakaran, Sneha, Case, Andrew, Richard III, Golden G., 2019. Droidscraper: a tool for android in-memory object recovery and reconstruction. In: 22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019), pp. 547–559.

Ali-Gombe, Aisha, Tambaoan, Alexandra, Gurfolino, Angela, Richard III, Golden G., 2020. App-agnostic post-execution semantic analysis of android in-memory forensics artifacts. In: Annual Computer Security Applications Conference, pp. 28–41.

Almomani, Iman, Alkhayer, Aala, El-Shafai, Walid, 2022. An Automated Vision-Based Deep Learning Model for Efficient Detection of Android Malware Attacks. IEEE Access.

Alzaylaee, Mohammed K., Yerima, Suleiman Y., Dl-droid, Sakir Sezer, 2020. Deep learning based android malware detection using real devices. Comput. Secur. 89, 101663.

Anthony, Desnos, Gueguen, G., 2013. URL code. google. com/p/androguard. Androguard-reverse Engineering, Malware and Goodware Analysis of Android Applications, vol. 153.

AVG Technologies, 2015. Android app performance trends: avg quarterly report q1 2015. URL. https://www.avg.com/en/signal/android-app-performance-trends-avg-quarterly-report-q1-2015.

Bissyandé, Tegawendé F., Klein, Jacques, 2021. Dexray: a simple, yet effective deep learning approach to android malware detection based on image representation of bytecode. In: Deployable Machine Learning for Security Defense: Second International Workshop, MLHat 2021, Virtual Event, August 15, 2021, Proceedings. Springer Nature, p. 81.

Chen, Shuo, Wang, XiaoFeng, Shao, Yuru, 2017. Understanding and characterizing ios background app refresh. ISBN 978-1-4503-5190-4. In: *Proceedings of the 2017 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2017 ACM International Symposium on Wearable Computers*, UbiComp/ISWC '17. ACM, New York, NY, USA, pp. 769–778. https://doi.org/10.1145/3123024.3123115.

Daoudi, Nadia, Jordan, Samhi, Kabore, Abdoul Kader, Kevin, Allix, Bissyandé, Tegawendé F., Klein, Jacques, 2021. Dexray: a simple, yet effective deep learning approach to android malware detection based on image representation of bytecode. In: International Workshop on Deployable Machine Learning for Security Defense. Springer, pp. 81–106.

Droidscraper, 2019. URL. https://github.com/apphackuno/DroidScraper. Online; accessed 10-January 2018.

D'Angelo, Gianni, Palmieri, Francesco, Antonio Robustelli, 2021. Effectiveness of video-classification in android malware detection through api-streams and cnn-lstm autoencoders. In: International Symposium on Mobile Internet Security. Springer, pp. 171–194.

Faruki, Parvez, Buddhadev, Bharat, Shah, Bhavya, Zemmari, Akka, Laxmi, Vijay, Singh Gaur, Manoj, 2019. Droiddivesdeep: android malware classification via low level monitorable features with deep neural networks. In: International Conference on Security & Privacy. Springer, pp. 125–139.

Guan, Weipeng, Li, Jingyi, Wen, Shangsheng, Zhang, Xinjie, Ye, Yufeng, Zheng, Jieheng, Jiang, Jiajia, 2019. The detection and recognition of rgb-led-id based on visible light communication using convolutional neural network. Appl. Sci. 9 (7), 1400.

Hsien-De Huang, TonTon, Kao, Hung-Yu, 2018. R2-d2: color-inspired convolutional neural network (cnn)-based android malware detections. In: 2018 IEEE International Conference on Big Data (Big Data). IEEE, pp. 2633–2642.

Hussaini, Adamu, Zahran, Bassam, Ali-Gombe, Aisha, 2021. Object allocation pattern as an indicator for maliciousness-an exploratory analysis. In: Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy,

pp. 313–315.
Jurafsky, Dan, James, H., Martin, 2009. URL. In: Speech and Language Processing, second ed. Prentice Hall. https://web.stanford.edu/jurafsky/slp3/.
Kevin, Allix, Bissyandé, Tegawendé F., Klein, Jacques, Androzoo, Yves Le Traon, 2016. Collecting millions of android apps for the research community. ISBN 978-1-4503-4186-8. In: *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16. ACM, New York, NY, USA, pp. 468–471. https://doi.org/10.1145/2901739.2903508.
Mahindru, Arvind, AL, Sangal, 2021. Mldroid—framework for android malware detection using machine learning techniques. Neural Comput. Appl. 33 (10), 5183–5240.
PyTorch, 2021. torchtext: data loaders and abstractions for text and nlp. https://torchtext.readthedocs.io/en/latest/.
Ramilli, Marco, 2016. Malware training sets: a machine learning dataset for everyone. URL. https://marcoramilli.com/2016/12/16/malware-training-sets-a-machine-learning-dataset-for-everyone/ [Online; December 2016].
Rasthofer, Siegfried, Arzt, Steven, Bodden, Eric, 2014. A machine-learning approach for classifying and categorizing android sources and sinks. In: NDSS, vol. 14, p. 1125.
Rastogi, Vaibhav, Chen, Yan, Jiang, Xuxian, 2013. Droidchameleon: evaluating android anti-malware against transformation attacks. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security. ACM, pp. 329–334.
Rodrigo, Corentin, Pierre, Samuel, Beaubrun, Ronald, El Khoury, Franjieh, 2021. Brainshield: a hybrid machine learning-based malware detection model for android devices. Electronics 10 (23), 2948.
Sihag, Vikas, Vardhan, Manu, Singh, Pradeep, Choudhary, Gaurav, Son, Seiil, De-lady, 2021. Deep learning based android malware detection using dynamic features. J. Internet Serv. Inf. Secur. 11 (2), 34–45.
Tian, Ke, Yao, Danfeng, Ryder, Barbara G., Tan, Gang, Peng, Guojun, 2017. Detection of repackaged android malware with code-heterogeneity features. IEEE Trans. Dependable Secure Comput. 17 (1), 64–77.
TruX Trustworthy Software at University of Luxembourg. Dexray. https://github.com/Trustworthy-Software/DexRay.
Vu, Long Nguyen, Jung, Souhwan, 2021. Admat: a cnn-on-matrix approach to android malware detection and classification. IEEE Access 9, 39680–39694.
Zhang, Hua, Qin, Jiawei, Zhang, Boan, Yan, Hanbing, Guo, Jing, Gao, Fei, 2020. A multi-class detection system for android malicious apps based on color image features. In: International Conference on Security and Privacy in New Computing Environments. Springer, pp. 186–206.
Zhang, Xiaolu, Frank, Breitinger, Luechinger, Engelbert, Stephen, O'Shaughnessy, 2021. Android application forensics: a survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations. Forensic Sci. Int.: Digit. Invest. 39, 301285.