

On the effectiveness of memory analysis in revealing maliciousness in android applications

Silvia Lucia Sanna ^{a,*}, Leonardo Regano ^a, Davide Maiorca ^a, Giorgio Giacinto ^{a,b}

^a University of Cagliari, Dept of Electrical and Electronic Engineering, Via Marengo, 2, Cagliari, 09123, Italy

^b Interuniversity National Consortium for Informatics, CINI, Via Ariosto, 25, Rome, 00185, Italy

ARTICLE INFO

Keywords:

Android malware analysis
Android memory forensics
Binary-to-image

ABSTRACT

Nowadays, Android malware has gained attention due to its ability to infect widely used mobile devices. Typically, current detection approaches rely on static and dynamic techniques to extract several kinds of features, to learn and identify common malicious patterns. However, modern Android malware often exhibits anti-analysis techniques (e.g., obfuscation, environmental checks, and runtime loading) hindering its full identification. To address this problem, we propose RAMimgDroid to analyse and detect malware as soon as it is loaded in memory. Furthermore, RAMimgDroid identifies and presents the analyst with relevant artifacts (e.g. variable declaration with their runtime content, encryption keys), useful to understand the runtime behavior. In detail, we extract the volatile memory allocated by the target process at application startup time. Subsequently, we encode the resulting dump as a 2D/1D RGB/grayscale image and feed it to a CNN for fast detection and identification of the most influential patterns (i.e. pixels and, subsequently, bytes of the dumped RAM). We test RAMimgDroid with 2073 Android malware and 2279 benign samples, assessing the effectiveness of various pre-trained and custom CNNs. Our results demonstrate that RAMimgDroid improves the detection of stealthy Android malware evading traditional detection methods (e.g. obfuscated or with anti-decompilation and environmental checks). Additional experiments on memory dumps acquired later in the malware execution show that RAMimgDroid remains effective even when malware behavior evolves over time (e.g. the malicious routine appears after a time of execution, not immediately at application startup). Moreover, we developed RAMstrDroid to extract the readable UTF-8 characters in the meaningful artifacts from the memory dump, interpreted as Indicator of Compromise (IoC), and compared with popular state-of-the-art tools i.e. VirusTotal and MobSF. The current implementation requires a rooted Android device or emulator and is limited to offline analysis because memory acquisition and image conversion remain too resource-intensive for continuous on-device monitoring.

1. Introduction

Nowadays, Android is the most used Operating System (OS) for mobile devices around the world, and Android applications (i.e. Android Package (APK)) are the most downloaded. Among such applications, some malware can be released, for example, to spy on users and steal their sensitive personal data. For this reason, many techniques have been developed considering the code structure or semantics (i.e. static analysis), their execution (i.e. dynamic analysis), or both. Static analysis is insufficient because it cannot observe the behavior of the software; hence, dynamic analysis has been introduced [1]. The output of these analyses is usually used to train Machine Learning (ML) or Deep Learning (DL) algorithms for fast, accurate, and autonomous detection. Most existing approaches rely on manually engineered features, par-

ticularly API calls, either by checking their presence or counting their occurrences [2–5]. However, such methods face several limitations. First, adversarial strategies have been developed to evade API-based detection [6–8]. Second, both static and dynamic approaches depend on predefined features and ground truth assumptions about their relevance [2,9–11]. Obfuscation techniques can conceal or distort these features, reducing detector robustness across different obfuscation strategies [12]. Furthermore, feature-based systems are affected by temporal and spatial drift [13], as applications released in different years may rely on different APIs due to Android version updates, system evolution, and deprecation of Java functions. Modern malware also employs anti-analysis mechanisms, including advanced obfuscation [14–16] and stegomware techniques [17–19], further challenging traditional approaches. Recent studies suggest that memory forensics can provide a

* Corresponding author.

E-mail addresses: silvia.sanna@unica.it (S.L. Sanna), leonardo.regano@unica.it (L. Regano), davide.maiorca@unica.it (D. Maiorca), giorgio.giacinto@unica.it (G. Giacinto).

<https://doi.org/10.1016/j.jisa.2026.104632>

Available online 18 September 2026

2214-2126/© 2026 The Author(s). Published by Elsevier Ltd. This is an open access article under the CC BY-NC-ND license (<http://creativecommons.org/licenses/by-nc-nd/4.0/>).

complementary perspective for malware detection [18,20]. Inspecting process memory allows for direct observation of runtime artifacts, including variable values, decrypted content, and dynamically generated objects, without relying solely on extracted API calls. Since data stored in RAM reflects the actual execution state of the application, memory analysis can reveal behavioral patterns that are difficult to capture through static inspection or high-level monitoring.

In this work, we propose *RAMimgDroid* (pronounced RAMimageDroid), a methodology that analyses the memory allocated to a target process at APK launch time and represents it through image encoding. Image-based representations have been widely adopted in malware detection [4,21–24]. For instance, *DexRay* [25] converts Dalvik Executable (DEX) files into grayscale images, while *RGB_Mem* generates RGB images from memory objects [20]. In our approach, each memory section (e.g., stack, code, libraries, and system data) is dumped separately and encoded as an RGB image. This representation preserves the structural correspondence between memory bytes and pixels, enabling automated pattern extraction while maintaining traceability to the original data. From an adversarial perspective, image-based representations derived from executable memory exhibit structural constraints that distinguish them from natural-image settings. While adversarial perturbations have been well studied in computer vision tasks [26–28], applying similar manipulations to malware-derived images requires consistency with the underlying binary content. Since each pixel corresponds to a memory byte, image-level modifications imply changes in executable memory. Such modifications must preserve correct execution of the program, limiting arbitrary perturbations [29]. Although this does not eliminate the possibility of adversarial manipulation, the tight coupling between visual representation and executable content reduces the feasibility of purely image-level alterations that are independent of program semantics [30–34]. To strengthen the importance and performance efficacy of the memory forensics approach for malware detection, we introduce a complementary tool, *RAMstrDroid* (pronounced RAM-stringDroid) where we extract UTF-8 human-readable characters from the same memory dumps while running the target APK. *RAMstrDroid* is fundamental for automatically comparing the results, such as extracted meaningful artifacts, i.e. IoCs, with human-readable output-based tools like *VirusTotal* and *MobSF*. Additionally, a pre-trained model on *RAMstrDroid* can be used quickly at runtime with less computational cost than converting the RAM dump into images. We investigate the following research questions:

- **RQ1 – Detection:** Can malicious behavior be identified directly by analyzing process memory dumps?
- **RQ2 – RAM Acquisition:** Which memory sections are the most informative for detecting malicious activity?
- **RQ3 – Image Encoding:** Which image encoding strategy is the most effective for representing process memory in malware detection?
- **RQ4 – Anti-analysis Resilience:** How does process memory analysis perform compared to existing tools when analyzing malware employing anti-analysis techniques?

Analyzing process memory provides a direct view of runtime behavior, overcoming several limitations of purely static analysis and high-level monitoring e.g. pre-defined static features like API calls, adversarial attacks, anti-decompilation and obfuscation. However, RAM analysis introduces practical challenges, including determining the appropriate acquisition time, selecting the number of snapshots required to capture relevant behavior, and identifying suitable encoding and analysis techniques for extracting discriminative patterns. Addressing these challenges requires representations that preserve memory structure while enabling effective automated classification. Our main contribution is to demonstrate that representing process memory as 1D or 2D images enables accurate detection of malicious behavior. We analyse the strengths and limitations of RAM-based approaches, highlighting both their ability to capture runtime artifacts and the practical issues related to timing, volatility, and data volume. We show that visual encodings of memory

expose structural patterns that are difficult to identify through conventional byte-level or semantic feature engineering.

We acquire RAM during the early execution phase, immediately after permission granting, when the application has been loaded into memory. To the best of our knowledge, *RAMimgDroid* is one of the first studies to analyse raw process memory at startup and publicly release the source codes, a full methodology, the trained models and a portion of the corresponding dataset (for ethics and safety reasons the full dataset can be requested by email). The public release includes the source code for memory acquisition, image conversion, string extraction, trained models (pth), together with a subset of memory dumps, encoded images, extracted strings. The full dataset, the evaluation results, and split files can be provided to reliable research groups upon motivated request, subject to ethical and safety restrictions. Our dataset includes benign applications from Androzoo [35] and malicious samples identified through *VirusTotal* [36], belonging to high-risk malware families. All applications are executed in a rooted emulator to ensure reproducibility and to restore a clean environment between runs. Since RAM extraction on Android requires super-user privileges, emulation provides the only scalable and automatic solution, more details in Sections 3 and 4. Beyond detection, *RAMimgDroid* identifies the most influential memory sections and pixels in the generated images, allowing correlation with the corresponding memory bytes. This supports forensic interpretation and facilitates understanding of malware behavior beyond standard industrial tools used in malware analysis such as *VirusTotal* and *MobSF* [37], especially when using *RAMstrDroid*. We also introduce a continuous RAM monitoring mechanism capable of detecting malicious activity that arises a time after startup, by inspecting objects allocated dynamically during the process execution. Rather than directly modeling the full semantic content of memory, *RAMimgDroid* captures section-specific structural regularities of runtime artifacts. Since different memory regions (e.g. stack, code, libraries, and system data) correspond to different execution roles, converting each dumped region into a separate image allows the model to learn complementary visual patterns while preserving traceability between pixels and the original memory bytes. The classification is done at APK-level by aggregating all memory regions per-APK.

Our **main contributions** can be summarized as follows: (i) a methodology for acquiring Android process RAM; (ii) early-stage acquisition of process memory immediately after permission granting; (iii) time- and event-based RAM analysis for extracting meaningful artifacts; (iv) a 1D image-creation algorithm mapping memory bytes directly to pixels; (v) public release of a subset of memory dumps, encoded images, strings, and related code; the full dataset will be released upon request to reliable research groups for ethical and safety reasons; (vi) identification of the most influential memory bytes through pixel-level analysis. To the best of our knowledge, *RAMimgDroid* and *RAMstrDroid* are one of the first approaches in the literature to perform Android malware detection based on the entire contents of the analysed process memory.

The remainder of this paper is structured as follows. Section 2 explains some essential concepts of Android application structure, RAM, and image analysis with the most important techniques in the current literature. Section 3 presents the methodology used, Section 3.7 briefly explains the potential threat model, and Section 4 the implementation details. The obtained results are presented in Section 5 and discussed with the presentation of some interesting case studies in Section 6. Limitations with future works are described in Section 7 while Section 8 closes the paper.

2. Background and related works

In this section, we will provide all the technical concepts useful to understand the details of our work and what has been done so far in the literature.

2.1. Android malware analysis

An Android application is a software for the Android OS, packaged as a APK file, which includes Java/Kotlin code, native C/C++ libraries, resources, and a manifest file that defines permissions and app declarations. The Java code is compiled into bytecode (.class files) and then into DEX files, which are executed by the Android Runtime (ART) or, in older versions (up to Android 4.4), by the Dalvik Virtual Machine (DVM). Since Android 5.0 (Lollipop), apps are compiled Ahead-of-Time (AOT) by Android Runtime (ART) during installation, whereas earlier versions used Just-in-Time (JIT) compilation. Variants of DEX files include .odex, .vdex, .bdex, .oat, .art, .quick, and .smali, which provide optimizations, metadata or human-readable representations. At runtime, the app is forked from the Zygote process, which initializes it with preloaded system resources. The app's memory usage is mapped in the `/proc/PID/maps`¹ file, showing different memory sections: memory-mapped files (e.g., dex, native libraries, assets), the ART/Dalvik heap (for dynamically loaded Java objects), anonymous mappings (without file backing), and the stack (function calls and local data). The garbage collector frees unused Java/Kotlin objects to optimize the memory.

Over the years, different studies have been developed to detect vulnerabilities in such applications or identify malicious signatures. Malware analysis is divided into static (i.e. analysis of the code structure) [2,38,39], dynamic (i.e. analysis of the behavior of the application at runtime and capture of the network traffic, performance, hooking functions, extraction of RAM) [5,40] and hybrid (i.e. considering both techniques) [41,42]. The output of the analysis is passed as features to ML algorithms for automatic detection [43,44]. Recently, some works demonstrated that these features must be robust enough so that an application does not bypass the detection mechanism [6]. There are very few works starting to extract statistics of the RAM usage [5] or allocated runtime objects for malware detection [20]. On the other hand, VolMemDroid [45] extracts *Volatility* output from complete Android RAM dumps during malware execution. Hence, to the best of our knowledge, there is no work in the literature that bases Android malware detection on the content of the RAM.

2.2. Android RAM analysis

The main memory (RAM) is the volatile memory used by systems to store temporarily active data, allowing fast read/write access and holding code and information of running applications, the operating system, and processes. Due to this temporary allocation, the RAM content changes over time. Android, built on the Linux kernel, divides RAM into *kernel space* for core functions and hardware interaction, and *user space* for applications running in isolated sandboxes. User space includes the heap, stack, code segments, and memory-mapped files, which are managed by ART (or DVM in older versions). Objects are allocated in the heap when created (e.g. new MyClass() in Java/Kotlin) and de-allocated by the garbage collector when no longer needed.

RAM analysis is valuable because it (i) captures data at a specific runtime moment; (ii) stores data in plaintext, making secrets such as encryption keys recoverable; and (iii) reveals the effects of processes on the system. Two main extraction approaches exist: tools like *Fridump* [46] dump the memory of a single process, while tools like *LiME* [47] capture the full RAM, enabling system-wide analysis.

RAM extraction typically requires root privileges, as the kernel protects direct memory access, and also due to the application's sandbox. Most tools (e.g. *LiME*, *Fridump*) also rely on kernel modules or low-level resources that are inaccessible without root. Bellizzi et al. [48–50] demonstrated that, by repackaging apps with *Frida gadget*,² portions of

process memory can be extracted without root access; however, this approach does not scale to all apps (because of anti-repacking protections) and only provides partial dumps (only a portion of the heap).

In Android, memory can be captured in multiple layers. *Memfetch* [51] operates at the process level by directly reading a target's address space through the `/proc/PID/mem` file, producing a raw dump of memory regions, including the heap, stack, and libraries, but requiring root privileges. In contrast, *Fridump* leverages the *Frida* instrumentation framework to enumerate and selectively dump memory regions at runtime, often without root access. While *Memfetch* offers a complete low-level snapshot suitable for forensic acquisition, *Fridump* provides a more flexible and fine-grained approach ideal for dynamic analysis and runtime object extraction. At the system level, kernel modules like *LiME* or *fnem* [52] acquire full physical memory. Subsequently, the full memory processes are analysed with frameworks like *Volatility* [53] to recover processes, sockets, and kernel structures, offering a system-wide view of a target APK's effects. Intermediate options include structured per-process captures (core dumps produced by gdb or gcore) and full-VM snapshots in *QEMU/Genymotion*; however, sophisticated malware may detect and alter its behavior under emulation. Finally, runtime instrumentation (e.g., *Frida* scripts or *Xposed* modules) avoids wholesale dumping by APKs acting as high-value artifacts (e.g. decrypted payloads, cryptographic keys, live objects, and invoked APIs), making it a powerful complement to both process- and system-level acquisitions.

The analysis of dumps combines string extraction, entropy profiling (to detect packed or encrypted regions), binary carving (DEX, Executable and Linkable Format (ELF), APK recovery), and parsing with frameworks like *Volatility*. Recent research has advanced from kernel-level acquisition techniques (e.g., *fnem*, *LiME*) toward targeted runtime object extraction. Tools such as *DroidScraper* [54] enable Dalvik/ART object retrieval for malware unpacking, while higher-level systems like *Timeliner* [55] reconstruct user-activity timelines from memory snapshots. More recent work has investigated real-time, non-root approaches, e.g. Just-in-Time Memory Forensics (JIT-MF) [48] and its successor *VEDRANO* [56]. These works combine lightweight instrumentation and application virtualization to deliver efficient, stealth-aware extraction.

In summary, Android memory forensics evolved from raw full-RAM dumps toward lightweight, selective, and non-root frameworks. Among these methodologies, RAM analysis uncovers runtime artifacts (e.g. hidden code segments, decrypted payloads, cryptographic material, and transient user data) that cannot be retrieved through static or disk-based inspection, emphasizing its fundamental importance in Android malware analysis and digital forensics.

2.3. Image analysis

Encoding binaries as images is a common technique for visualizing and analyzing binary data, especially in malware detection and steganography. In this approach, raw bytes are mapped directly to pixel values (either grayscale or RGB) and reshaped into a 2D image, typically stored as a lossless PNG to preserve byte-level fidelity. Although JPEG offers smaller file sizes, its compression introduces noise that can degrade learning performance. Before classification, images undergo standard preprocessing and Convolutional Neural Network (CNN) are widely used because they effectively capture spatial patterns through convolution, activation, pooling, and fully connected layers. Depending on the task softmax or sigmoid layers produce the final output. Model training involves iterative optimization using loss functions such as cross-entropy, while performance is evaluated with common metrics including accuracy, precision, recall, F1-score, and confusion matrices. Different works use the images for classification, for example, for pattern identification, such as in medical scenarios [57,58] or object identification, such as autonomous cars, video surveillance [59–61].

Different tools have been developed to convert binary into images [62–64] and subsequently classify the images. As highlighted by

¹ In the `/proc` folder, for each running process, there is a folder named with its Process Identifier (PID), containing a dedicated `maps` file.

² <https://frida.re/docs/gadget/>

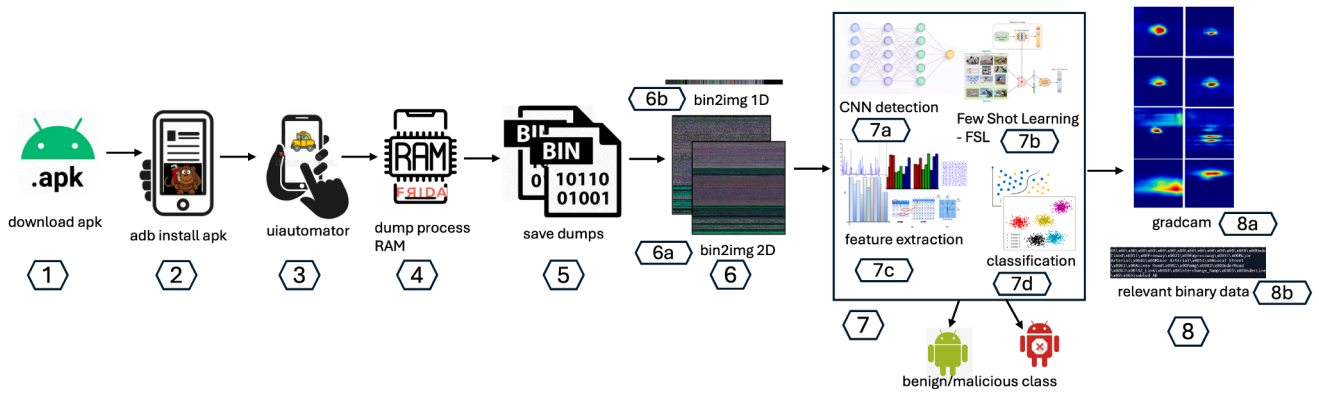


Fig. 1. Pipeline methodology showing how from a given APK (1) it is installed on the device (2), permissions are granted (3) so that it runs in RAM with its default behavior, the RAM of the target application is dumped (4), saved (5) and converted into images (6) in 2D (6a) and 1D (6b). Different AI-based algorithms are applied (7), DL (7a, 7b) and ML (7c, 7d). At the end, most significant Section (8) are highlighted with Grad-CAM (8a) and mapped back to raw memory binary data (8b).

Daoudi et al. in *Dexray* [25], this approach has a limitation due to the spurious correlation between vertical near pixels that do not represent near bytes. In fact, in this work we release an algorithm to map the binary (i.e. the memory dump) into 1D images where each pixel represents a byte and the proximity between pixels perfectly maps the near bytes in memory (see [Algorithm 1](#)).

3. Methodology

In this section, we provide all the technical details used in the methodology pipeline of *RAMimgDroid* ([Fig. 1](#)) to reach the presented results. First of all, given an APK we install it in a rooted emulator in order to: (i) have the super-user privileges to extract the content of the RAM, (ii) restore the clean environment after execution for another APK and (iii) protect a real device from real malware execution with subsequent potential device breaking. Large-scale automated Android malware analysis cannot be effectively performed on real devices due to practical and technical constraints. Memory forensics requires root privileges, but restoring a device to a clean state after each APK execution (necessary to avoid contamination from previous analyses) removes such privileges. Moreover, rooting a physical device cannot be fully automated, as it often involves manual interaction (e.g., pressing hardware buttons) and time-consuming procedures. In contrast, emulators provide a controlled, easily resettable, and rapidly configurable environment, making them far more suitable for scalable and automated malware analysis workflows.

When the APK is started (step 2 on [Fig. 1](#)), we grant all its requested permissions (step 3 on [Fig. 1](#)) to enable the application's full functionality (i.e. microphone, camera, access to files). After granting the permissions the APK is loaded in RAM and we proceed to extract the content of the allocated space in RAM to the target APK (user-level memory extraction, step 4 on [Fig. 1](#)). Until the user accepts or denies the requested permissions, the APK is not loaded in memory, hence the memory cannot be dumped as the process is not found in memory. At this point, in *RAMimgDroid*, the memory can be dumped (step 5 on [Fig. 1](#)) and converted into 2D and 1D images (step 6 on [Fig. 1](#), respectively 6a and 6b), from which we extract specific traditional features or apply CNNs (step 7 on [Fig. 1](#), respectively 7c with 7d for features and 7a with 7b for CNNs). This step provides an automatic classification of the target APK into benign or malicious class, according to the classification of each memory region. By applying Explainable Artificial Intelligence (xAI) on the classification results (step 8 on [Fig. 1](#)), we can find the most influential pixels in the most influential memory regions (step 8a on [Fig. 1](#)), hence determining the most important bytes, useful for the human analysis (step 8b on [Fig. 1](#)). When using *RAMstrDroid* the UTF-8 characters are extracted and Natural Language Processing (NLP) algorithms are applied for classification and xAI algorithms to extract meaningful strings

that are subsequently used as meaningful artifacts, i.e. IoCs especially to compare *RAMstrDroid* and memory forensics for malware analysis with state-of-the-art tools like *VirusTotal* and *MobSF*. As shown in [Table A.10](#), only semantic malware-specific indicators that were manually validated or matched with evidence from *VirusTotal* or *MobSF* are referred to as conventional IoCs. Other highlighted elements, such as structural stack patterns, Android system/path tokens, or low-level byte and string patterns, are instead interpreted as explanatory memory artifacts. Although these artifacts may appear as noise or random bigrams from a human analyst's perspective, they can still be highly discriminative from the model's point of view and may contribute to successful malware detection. For this reason, after manually validating a restricted subset of samples through the case study, we use the term IoC in a broader memory-forensic sense to denote not only human-readable semantic indicators, but also memory-level artifacts that provide meaningful evidence for the classifier's decision.

3.1. Memory dataset collection

The first step in the methodology is the dataset. Since existing public datasets (e.g. *AndroMemDump* [65]) of Android memory captures are limited in completeness, i.e. do not provide the raw RAM content required for our analysis and are also very old (2017), we create a dedicated dataset composed of memory dumps collected during the execution of both benign and malicious Android applications.

To collect these memory dumps (i.e. the first three steps in [Fig. 1](#)), each application is executed in a controlled Android environment that supports repeatable large-scale testing. A fresh device state is required before each execution to guarantee that no residual data, including persistent artifacts left by malware, contaminates subsequent memory captures. This reset step is also important because certain malware families can survive to app removal or power cycles and may affect other applications' memory regions. Although some malware can detect and alter behavior in virtual environments, an emulated device remains the most practical solution for systematic, automated, and scalable experimentation. Thus, in our experimentation, we opted to use an emulated device through the *Genymotion* emulator, as detailed in [Section 4.2](#).

Each application is then launched through its main activity, ensuring that its typical runtime behavior is initiated. Since many Android applications request access to system resources, the methodology includes a phase in which required permissions are granted or denied according to configurable rules. This step ensures consistent behavior across applications and avoids premature termination caused by unresolved permission prompts.

Once the application is running and stable (i.e. it does not crash and remains active in memory), its corresponding RAM region can be extracted. If the application fails to start, crashes, or remains inactive

due to unmet conditions (e.g. requested permissions, target SDK), no meaningful memory dump is produced. Notably, no application in our dataset produced such error, hence we have the dump of each APK we analysed. The collected memory data are later transformed into one- and two-dimensional images directly derived from the raw memory content. Detection models trained on these representations can then identify anomalous or malicious patterns. Finally, significant regions identified by these models are mapped back to the original memory segments to interpret the relevance of specific binary sections.

3.2. Memory dumps

After having prepared the framework and the APK is correctly running in the emulator, we are ready to dump the memory of the target process (i.e. steps 4 and 5 in Fig. 1: *dump process RAM* and *save dumps*). The memory extraction methodology focuses on acquiring only the portion of RAM allocated to the target application, rather than extracting a complete system memory image requiring kernel recompilation and slowing down the pipeline. This extraction yields a focused and manageable dataset that isolates the application's runtime behavior from unrelated system activity and avoids the overhead typically associated with full memory acquisition. To perform this extraction, a dynamic instrumentation framework is attached to the running process, in order to enumerate the process' memory regions. In our workflow, this capability is provided by *Fridump*, which leverages the *Frida* instrumentation engine to inspect, and extract the process's address space in a fine-grained and controlled manner. Each region is accessible at least for reading, representing all meaningful regions of the process address space.

Once the target application is launched and verified to be active, the extraction framework identifies all memory regions currently mapped to its virtual address space. As shown in Table A.3, these include code-related mappings (i.e. mappings associated with APK files, framework JARs, and executable/runtime code artifacts; e.g. *base-apk (BA)*, *base odex + apk*, *base vdex + apk*, *apk-full (AF)*, *system*, and *system-full*), native libraries (i.e. native shared-object mappings and anonymous native-library regions; e.g. *.so* libraries and the library-related portions of *apk-full (AF)*, *apk-full + memory*, and *apk-full + stack*), bytecode areas (i.e. Dalvik/ART bytecode and optimized bytecode artifacts such as *.dex*, *.odex*, and *.vdex* mappings; e.g. *base odex*, *base vdex*, *base odex + vdex*, and the corresponding portions of the extended APK-related configurations), heap and stack regions (i.e. runtime memory areas related to thread stacks, heap-like allocations *anon*, VDSO/VVAR mappings, and broader process-memory selections; e.g. *stack*, *memory*, *stack + vdso*, *complete memory*, *base-apk + memory*, *base-apk + stack*, and *data + stack*), and dynamically loaded or runtime-generated components (i.e. Dalvik anonymous mappings, APK-related anonymous mappings, application-data paths, databases, preferences, cookies, and runtime resource-cache artifacts; e.g. *apk-full (AF)*, *apk-full + memory*, *apk-full + stack*, *data*, and *data + stack*). For each region, the tool determines its access permissions and, when the region is readable, retrieves its raw byte content. This per-region extraction approach ensures that only memory genuinely belonging to the target process is acquired, rather than the full device memory, allowing us to focus the dataset on semantically relevant information. An important characteristic of *Fridump* is its ability to retrieve each region individually while preserving the structural organization of the memory layout. The extracted sections mirror the segmentation reported by the OS (e.g. through the process memory map), enabling a direct correspondence between each dumped file and the specific memory area it represents. This mapping is crucial for two reasons: (i) *traceability*, since every extracted segment can be linked back to its function within the process (e.g. stack, heap, library code), allowing for targeted analysis; (ii) *explainability*, since the regions identified as relevant during ML-based interpretation can be mapped back to specific memory areas and, ultimately, to the program components from which they originate. This means that the stack section mapped in a single memory section is saved in a unique file. Although memory

extraction occasionally triggers warnings when non-readable or inaccessible regions are encountered, the framework is designed to skip such segments gracefully while continuing the acquisition of all readable areas. Since all valid process regions are at least readable, this approach ensures comprehensive coverage of the application's runtime memory.

Compared with kernel-level dumpers, which typically produce large monolithic images and require privileged access with kernel recompilation, the region-oriented extraction used here provides a selective, structured, and non-intrusive alternative. It captures the raw byte content of each memory segment (including non-textual bytes that are often meaningful in malware analysis [66,67]) while avoiding the noise and complexity associated with full system dumps.

3.3. Image-based memory representation

As described in Section 2.3, *RAMingDroid* adopts the binary-to-image method of Ncarkaci et al. [62] to convert raw memory regions into RGB or grayscale PNG images. For this reason, after the memory dump, in the pipeline framework depicted in Fig. 1, we have the *bin2img 1D* and *bin2img 2D* conversion, i.e. the sixth step of binary to image conversion. More in detail, let a dumped memory region be represented as a sequence of bytes

$$B = \{b_1, b_2, \dots, b_n\}, \quad b_i \in \{0, \dots, 255\}.$$

Each byte is mapped to a pixel according to the selected encoding:

- *Grayscale (L)*: $p_i = b_i, \quad i = 1, \dots, n.$
- *RGB*: $p_k = (b_{3k-2}, b_{3k-1}, b_{3k}), \quad k = 1, \dots, \left\lfloor \frac{n}{3} \right\rfloor.$

2D image creation

The byte stream is first mapped to a one-dimensional pixel sequence and subsequently reshaped into a two-dimensional image using a row-major layout, whereby a fixed width (e.g., 256 pixels) is defined and the image is filled sequentially row by row, in accordance with the conventions adopted by standard image processing libraries (e.g., *NumPy*,³ *Pillow*,⁴ *OpenCV*⁵). The 2D image with width W and height H is defined as

$$I_{2D}(r, c) = p_{(rW+c)+1}, \quad 0 \leq r < H, \quad 0 \leq c < W,$$

with height and coordinates r and c , respectively row (r denotes the row index, that is, the vertical position of a pixel in the image) and column values (c denotes the column index, that is, the horizontal position of a pixel in the image)

$$H = \left\lfloor \frac{N}{W} \right\rfloor, \quad r = \left\lfloor \frac{i-1}{W} \right\rfloor, \quad c = (i-1) \bmod W.$$

If W is not provided, it is selected through the heuristic proposed by Nataraj et al. [21], which favors canonical aspect ratios suitable for visual inspection of binary structure. This construction is lossless with respect to others as byte adjacency becomes spatial adjacency along rows, preserving local patterns (headers, code sections, and entropy changes).

1D image creation

To avoid spurious correlations introduced by vertical adjacency in 2D images (i.e. bytes that are far apart in memory may appear close as pixels), *RAMingDroid* also generates 1D images (Algorithm 1) in which all pixels are arranged along a single horizontal line:

$$I_{1D}(0, c) = p_{c+1}, \quad c = 0, \dots, N-1,$$

with width = N (number of bytes), height = 1. This representation is an isomorphic visualization of the raw memory content: the spatial position of each pixel corresponds exactly to the byte's sequential position in the memory region. It preserves locality without introducing artificial neighborhoods, making it suitable for sequence-like analysis.

³ <https://numpy.org/>

⁴ <https://pypi.org/project/pillow/>

⁵ <https://opencv.org/>

Algorithm 1 File-to-image construction (1D and 2D, L/RGB).

Require: Byte sequence $B = \{b_1, \dots, b_n\}$; mode $\in \{L, RGB\}$; layout $\in \{1D, 2D\}$; optional width W for 2D

Ensure: Image array I

```

1: if mode = L then
2:    $N \leftarrow n$ 
3:   for  $i \leftarrow 1$  to  $n$  do
4:      $p[i] \leftarrow b_i$ 
5:   end for
6: else if mode = RGB then
7:    $N \leftarrow \lfloor n/3 \rfloor$ 
8:   for  $k \leftarrow 1$  to  $N$  do
9:      $p[k] \leftarrow (b_{3k-2}, b_{3k-1}, b_{3k})$ 
10:  end for
11: end if
12: if layout = 1D then
13:    $H \leftarrow 1$ ;  $W \leftarrow N$ 
14:   for  $c \leftarrow 0$  to  $W - 1$  do
15:      $I[0, c] \leftarrow p[c + 1]$ 
16:   end for
17: else if layout = 2D then
18:   if  $W$  not provided then
19:      $W \leftarrow \text{HEURISTICWIDTH}(N)$  ▷ e.g., 32, 64, 128, ...
20:   end if
21:    $H \leftarrow \lceil N/W \rceil$ 
22:   for  $i \leftarrow 1$  to  $N$  do
23:      $r \leftarrow \lfloor (i - 1)/W \rfloor$ 
24:      $c \leftarrow (i - 1) \bmod W$ 
25:      $I[r, c] \leftarrow p[i]$ 
26:   end for
27: end if
28: return  $I$ 

```

In summary, 1D and 2D encodings differ only in spatial arrangement. The 1D layout preserves strict byte adjacency, while the 2D layout folds the sequence according to a chosen width W , possibly creating spurious correlations between bytes that were not adjacent in memory. However, 2D representations often highlight structural patterns that are visually and statistically meaningful for downstream learning.

3.4. Image-based memory classification

After converting each memory dump binary corresponding to each memory region into a RGB/grayscale 1D or 2D PNG, each image is processed by a CNN. Classification is ultimately performed at the APK level, not at the image level. This means that regions from the same APK cannot appear across different splits, but only different APK, with all its memory regions and corresponding images, appear in different splits. For each application, the model computes a maliciousness probability for each of its memory-region images; these scores are aggregated using an averaging rule to obtain a single APK-level score. A threshold chosen via Youden's J statistic [68] on the APK-level ROC curve determines the final classification (benign vs. malicious). All performance metrics (accuracy, ROC-AUC, PR-AUC, confusion matrix, and classification report) are computed at this APK level. State-of-the-art works adopt feature extraction (e.g. local binary pattern, histograms, statistics, entropy, FFT) on images to analyse and classify them with traditional ML algorithms (e.g. SVM, KNN, MLP, Random Forest). In the following sections we will highlight both methodology analysis, feature extraction and CNN-based techniques.

3.4.1. Feature extraction

For 2D images, we extract a broad set of classical image descriptors widely used in the Computer Vision literature [69] (see Section 2.3). These include a broad set of complementary descriptors that capture different visual aspects of each memory-region image. *Color histograms* [70] in RGB and HSV spaces describe the distribution of pixel intensities, while *color moments* [71] provide compact global statistics such as *mean*, *standard deviation*, and *skewness* for each channel. Textural information is represented through *GLCM features* [72], which measure spatial co-occurrence relationships, and *Local Binary Patterns (LBP)* [73], which encode micro-texture structures in local neighborhoods. Structural characteristics are extracted using *Histograms of Oriented Gradients (HOG)* [74], which capture edge-orientation patterns, and *Sobel-based edge statistics* [75], which quantify the overall edge magnitude across the image. Global geometric properties are summarized by *Hu moments* [76], a family of rotation- and scale-invariant descriptors. Additional global intensity characteristics are captured by *grayscale statistics*, including *mean*, *variance*, *kurtosis*, and *entropy* [77]. Finally, *FFT-based frequency descriptors* [78] highlight periodic or spectral structures indicative of regular or repetitive memory layouts.

Each descriptor family is stored in a dedicated column as a feature vector, ensuring a fixed schema across all samples regardless of dimensionality differences between regions.

3.4.2. Convolutional neural networks for 1D and 2D representations

CNNs [79–81] are well suited for detecting structural patterns in binary-derived images because they learn hierarchical feature representations that capture both local and global regularities. In our framework, CNN architectures are employed in two complementary ways, depending on whether the memory content is encoded as a 1D or 2D image. Although both representations originate from the same sequence of bytes, their geometries differ significantly, motivating distinct convolutional treatments.

2D CNNs. In this study, we investigate which pretrained convolutional or transformer-based architectures, originally trained on large-scale image datasets, are most suitable for memory-region-based malware detection when reused through transfer learning. The goal is twofold: (i) assess how well general-purpose visual representations transfer to binary-derived memory images, and (ii) compare our approach against existing state-of-the-art CNN-based methods proposed for similar tasks [4,20,25]. To this end, we adopt a transfer learning strategy in which the pretrained feature extraction layers are retained, while the original classification layers are replaced with a task-specific binary classifier distinguishing benign from malicious applications. This approach enables faster convergence, mitigates overfitting, and leverages robust representations learned from large and diverse datasets. We additionally adopt a Few-Shot Learning (FSL) strategy to evaluate the effectiveness of learned visual representations under limited labeled data [82,83]. The final classification layer is removed and the network is used as a fixed feature extractor, subsequently given in input to traditional Machine Learning algorithms like SVM, KNN, Random Forest. This approach allows to assess the generalization capability of the learned representations, reduce dependence on large labeled datasets, and compare different architectures in a low-data regime that is representative of realistic malware analysis scenarios.

1D CNNs. The one-dimensional CNNs are designed to operate on memory representations that preserve the intrinsic sequential ordering of bytes, i.e. 1D images. The methodological goal is to align the model's inductive bias with the linear structure of memory regions, treating each dump as a one-dimensional signal rather than a spatial image. In this framework, convolutional operations are applied exclusively along the horizontal axis, enabling the model to learn discriminative patterns from contiguous memory segments and to progressively aggregate local byte-level structures into higher-level representations. Two complementary strategies are explored: (i) one that adapts an existing pretrained architecture to the one-dimensional setting in order to reuse robust feature

representations learned from large datasets; (ii) and one that relies on a fully custom 1D architecture specifically tailored to sequential memory data. Both approaches aim to capture dependencies within memory regions, providing an effective representation for malware detection based on memory behavior.

CNN Comparison. Although both 1D and 2D CNNs analyse the same underlying bytes, they impose different inductive biases. The 2D models emphasize spatial textures and grid-like patterns that emerge from memory layout regularities, enabling them to detect global structures such as entropy bands or region transitions. By contrast, 1D models emphasize sequential coherence. Overall, CNNs for 1D and 2D representations provide two distinct yet complementary analytical lenses.

3.5. Text-based memory classification

In addition to image-based representations, we introduce a complementary tool *RAMstrDroid* (pronounced as RAMstringDroid), a text-based analysis of memory dumps by extracting their UTF-8-encoded string content, corresponding to the subset of bytes that can be interpreted as human-readable text. It is fundamental for the comparison with current state-of-the-art tools, such as *VirusTotal* and *MobSF*. These strings often include filenames, URLs, command-line arguments, configuration parameters, error messages, and other textual artifacts that may reveal malicious intent or runtime behavior. Although such string representations do not capture the full semantic content of memory, they remain widely used in practical malware analysis and forensic tools due to their interpretability and ease of processing. The extracted strings are treated as textual documents and analysed using Natural Language Processing (NLP) [84,85] techniques to assess the likelihood that a memory dump originates from a malicious application. The extracted strings are treated as textual documents and transformed into numerical features using standard document classification techniques, enabling ML models to identify linguistic patterns indicative of malicious behavior (e.g. URLs, domains, APIs, commands). *RAMstrDroid* has a threefold objective. First, it provides a complementary detection modality that focuses exclusively on human-readable evidence, offering an interpretable baseline against which image-based and binary-level approaches can be compared. Second, it enables a direct and fair comparison with widely adopted analysis platforms such as *VirusTotal* and *MobSF*, which primarily rely on string extraction and text-based reporting to identify indicators of compromise. Third, since image-based analysis is computationally intensive and may not be suitable for real-time deployment, we assess a lightweight methodology that can be applied at runtime. In this context, text-based memory analysis represents the fastest alternative, and we therefore evaluate *RAMstrDroid* detection accuracy by also comparing the extracted strings from the dynamic memory analysis with the extracted static strings from the APK and its decompiled version with *apktool*.

3.6. Data explanation

The last step of our pipeline, as depicted in Fig. 1, is the *Grad-CAM* analysis with the extraction of *relevant binary data*. To interpret model decisions, we apply in *RAMingDroid* xAI techniques such as *Grad-CAM* [86], which highlight the most influential regions of an image in determining the classification outcome. Because each pixel uniquely maps to a byte in the memory dump, *Grad-CAM* highlights can be traced back to specific memory offsets, enabling identification of critical bytes and memory regions (e.g. stack, libraries, code segments). Section 5.2 presents experiments where influential pixels are either isolated or masked to assess their contribution.

We also apply NLP-based explainability techniques (LIME [87], SHAP [39]) in *RAMstrDroid* to human-readable strings extracted from the memory dumps. Strings are vectorized with TF-IDF and classified using logistic regression, enabling extraction of textual meaningful arti-

facts, i.e. IoC manually validated by comparing with standard analysis tools such as *VirusTotal* and *MobSF*.

3.7. Threat model

We consider Android malware that attempts to evade conventional static and dynamic analysis through obfuscation, reflection, runtime loading, anti-decompilation, anti-debugging, anti-emulator checks, delayed payload activation, or adversarial manipulation of static/high-level dynamic features. The attacker controls the APK under analysis but not the analysis infrastructure, the emulator host, the memory acquisition tool, or the trained model. *RAMingDroid* assumes that the target application can be executed and that its user-space process memory can be acquired during the observation window (i.e. with super-user privileges). Kernel-level malware, attacks that compromise the analysis environment, and memory-manipulation strategies explicitly designed to alter the byte-to-image representation while preserving malicious execution are outside the scope of this work and will be addressed in future works.

4. Experimental framework

In this section, we present the obtained results for every experimental case. We used a machine different from the one used for classification to create the dataset.

4.1. Experimental setup

The details of the experimental setup are presented in Appendix A. We adopt an APK-level evaluation strategy. During training, each memory region inherits the label of its originating APK (benign or malicious). During testing, the CNN produces a maliciousness probability for each memory-region image. Since the final prediction target is the APK rather than the individual memory region, all probabilities associated with the same APK are aggregated by averaging them, obtaining a single APK-level maliciousness score. The final APK label is assigned by comparing this aggregated score with a decision threshold selected using Youden's *J* statistic on the ROC curve. Therefore, an APK is classified as malicious when its aggregated score is greater than the selected threshold, and as benign otherwise. All threshold-dependent results reported in the experimental tables use this same APK-level aggregation and thresholding rule. Performance is measured using accuracy, precision, recall, F1-score, and TP/TN/FP/FN rates. Misclassified APKs and their corresponding regions are logged for detailed analysis.

We also consider a region-level evaluation as an ablation study, where regions are independently classified and metrics are computed at region granularity. However, since real-world malware detection operates at the application level, region-level performance alone does not reflect practical effectiveness. Therefore, we primarily report APK-level results, using region-level analysis only for case studies and to identify the most influential memory regions.

In short, the split unit is the APK, not the memory-region image. Specifically for each class, 1,300 APKs, including all their extracted memory regions, are assigned to the training set, 200 APKs, including all their regions, are assigned to the validation set, and the remaining APKs, again with all their regions, are used for testing (i.e. 779 for benign APK and 573 for malware APK). Therefore, all regions extracted from the same APK always belong to the same split: we never use one region of an APK for training and another region of the same APK for validation or testing, since this would introduce region-level leakage and inflate the results. For each experiment, we maintain a manifest that maps every APK to its corresponding memory-region files and split assignment. This mapping is used for APK-level aggregation, misclassification analysis, and explainability, allowing us to trace each APK-level decision back to the most relevant memory regions. Thus, the classification is performed at APK-level by averaging the probabilities of all

regions belonging to the same APK, while region-level outputs are used only as intermediate predictions and for interpretability.

4.2. Implementation details

In this Section we explain all the implementation details related to the pipeline framework described in Section 3 and depicted in Fig. 1, where each paragraph corresponds to a paragraph of the methodology and a step in the pipeline. **Dataset Construction.** To build the dataset, we collected 2074 Android APKs labeled as malicious from *VirusTotal* (spanning 2022–2024) and 2279 benign applications from *AndroZoo* (2020–2022), more details in the Dataset Distribution paragraph Section 4.3. Each sample was executed in a controlled environment based on the *Genymotion* emulator, which we selected due to its scripting flexibility and compatibility with automated device management via *gmtool*. We compared the use of *Genymotion* with the official *Android Studio* emulator and noticed that, in our dataset, evasive samples (i.e. malware with anti-debug implementations) are executed normally in *Genymotion* while in *Android Studio* the installation or execution is blocked. In our experiments, these samples did not trigger emulator-detection behavior in *Genymotion*, even with root enabled. However, this observation is limited to our tested dataset and environment. Further studies are required to compare different sandboxes systematically, and we encourage the community to conduct additional tests on this topic. Before running each APK, the virtual device was restored to a factory-reset state to ensure that no residual malware artifacts, cached services, or persistent components remained in memory. During execution, permissions requested by the application were automatically granted or denied using the *uiautomator2*⁶ framework, which allowed us to interact deterministically with system dialogs that could not be controlled by *monkey*⁷ or *droidbot* [88] because of the random interaction and input. *Monkey* is Android’s official command-line tool that stress-tests an app by sending pseudo-random user/system events, whereas *DroidBot* is a UI-guided Android input generator that explores apps more intelligently through GUI-aware or scripted interactions. Only applications that successfully launched their main activity and remained active in the process list were included in the dataset; crashes or permission failures resulted in discarded samples.

RAM Extraction. Memory acquisition was performed using *Fridump* [46], which relies on *Frida* dynamic instrumentation capabilities. To use it, we installed the *frida-tools* Python package on the host and deployed a matching *frida-server* binary on the emulated device, selecting the correct architecture (armv8, armeabi, x86, or x86_64) and ensuring version compatibility with the NDK and Python bindings. After launching an APK, we verified its presence in the running process list via both Linux commands (*ps -A*) and Frida’s own process enumerator (*frida-ps -U*). *Fridump* was then attached to the target PID to enumerate all mapped memory regions. Since *Fridump* occasionally raises a “memory access violation” when encountering a region lacking the requested permissions, we modified the script to extract all regions marked at least as readable, i.e. dumping all memory areas. Before dumping, we saved the process memory map from */proc/PID/maps* so that each dumped segment, named according to its start address (e.g. 0x7fff3daca000_dump.data), could later be associated with its semantic role (e.g. stack, shared library, dex file, etc.). *Fridump* per-region dumping approach allowed us to avoid full-system memory images while preserving precise structural granularity.

Image Construction. After extraction, in *RAMimgDroid* each memory-region binary file was transformed into (i) 2D image representations using the binary-to-image tool of Ncarkaci et al. [62] and (ii) 1D image representation using our implementation in Algorithm 1. The tool reads raw bytes and converts them into grayscale or RGB pixels,

rasterizing them into 2D images according to a width chosen through Nataraj’s heuristic [62], or alternatively arranging all pixels along a single horizontal row to produce a 1D image. This conversion was implemented using the *Pillow* library,⁸ which generated consistent PNG files from raw byte sequences. Each image file was named using the original memory-region start address so that image-level predictions could later be mapped back to their memory origins.

Training Pipeline. In *RAMimgDroid* for CNN-based experiments, we implemented a training pipeline that loads memory-region images by parsing the previously saved *maps.txt* file associated with each APK, selecting specific region types (stack, heap, native libraries, dex files, etc.). The images were resized, normalized using *ImageNet* statistics [89], and fed into a variety of pretrained CNN and transformer backbones. The training loop employed the *AdamW* optimizer with a *OneCycleLR* policy, class-balanced batch sampling, and region-level inference followed by APK-level score aggregation. Metadata (including misclassified regions, *Grad-CAM* heatmaps, and intermediate logits) was logged for further interpretation and correlation with memory bytes. Model evaluation is conducted at the application level by aggregating predictions from individual memory regions, and performance is quantified using standard metrics such as *ROC-AUC*, *PR-AUC*, confusion matrices, and detailed error analysis, including the identification of misclassified applications and the memory regions most responsible for incorrect predictions. Models were tested on 1D and 2D images, respectively, in grayscale and RGB.

2D CNNs. In 2D CNNs for 2D images in *RAMimgDroid*, we employ pretrained CNN architectures such as *ResNet* [90], *EfficientNet* [91], *MobileNet* [92], *ConvNeXt* [93], *ViT* [94], *InceptionNet* [95], *VGG16* [96], all taken from *PyTorch*,⁹ plus custom models like *DiDroid* [4], *DexRay* [25], *RGB_Mem* [20]. These models retain their convolutional or transformer-based feature extractors, while their final classification layers are replaced with binary-output heads tailored to our task.

1D CNNs. From an implementation perspective, in *RAMimgDroid* one approach adapts a *ResNet-18* backbone to one-dimensional input by retaining its convolutional feature extractor up to the final convolutional block. Instead of applying standard global average pooling over both spatial dimensions, an adaptive pooling operation is used to collapse the vertical dimension while preserving the horizontal extent, effectively transforming two-dimensional feature maps into sequences of one-dimensional embeddings. A lightweight classification head is then applied to these sequential features, combining one-dimensional convolution to model local dependencies along the sequence, pooling operations to standardize feature length, and a multi-layer perceptron with regularization to produce a binary prediction. In parallel, a fully custom one-dimensional CNN is implemented to process horizontally serialized memory dumps directly, without relying on pretrained two-dimensional backbones. This architecture operates on fixed-length one-dimensional input with multiple channels and is composed of successive convolutional blocks that progressively increase representational capacity while reducing sequence length. Pooling layers ensure a consistent feature size for the classifier, and an attention mechanism reweights feature channels according to their discriminative relevance. The final classifier projects the learned representation into a compact latent space and outputs a binary decision. This design enables efficient learning of sequential dependencies in memory data while remaining independent of visual priors derived from natural-image datasets.

Traditional Machine Learning. In addition to end-to-end CNNs, *RAMimgDroid* also employs traditional ML as a complementary analysis strategy. In particular, ML is used in two settings: (i) to classify handcrafted features extracted from the 2D memory-derived images, such as color, texture, gradient, statistical, and frequency descriptors; and (ii) to evaluate Few-Shot Learning (FSL), where features extracted

⁶ <https://github.com/openatx/uiautomator2>

⁷ <https://developer.android.com/studio/test/other-testing-tools/monkey>

⁸ <https://pypi.org/project/pillow/>

⁹ <https://docs.pytorch.org/vision/main/models.html>

Table 1
Malware dataset statistics.

Malware Families		Malware Variants		Package / Process Names		Selected Tags		Year Distribution		
Name	Count	Name	Count	Name	Count	Name	Count	Year	MinSDK	TargetSDK
trojan.coper	288	fakeapp	236	com.metasploit.stage	221	reflection	1678	2008	8	0
trojan.metasploit	206	metasploit	207	com.nameown12	138	obfuscated	1174	2009	468	110
trojan.fakeinst	175	bankbot	96	Chrome	238	runtime-modules	475	2010	311	63
trojan.opfake	117	hiddenad	88	MainActivity	221	contains-elf	263	2011	376	13
trojan.dnotua	93	boxer	80	Google-Chrome	184	crypto	276	2012	170	235
adware.hiddad	66	smssend	69			sudo	147	2013	146	39
Trojan	1876	r002c0pd324	65			detect-debug-environment	318	2014	343	24
SMS	1019	trojansms	49			exploit	192	2015	12	248
Fakeapp	251	spymax	47			CVE TCP*	47	2016	8	14
Adware	122	fakeinst	44					2017	170	137
Agent	61							2018	2	154
Exploit	57							2019	1	32
Pua	35							2020	0	159
Dropper	13							2021	0	5
Miner	2							2022	0	219
Keylogger	2							2023	0	1
Banking	2							2024		0

* CVE entries: CVE-2005-0068, CVE-2015-7759, CVE-2004-0790

from the last layer of ResNet-18 are given to lightweight classifiers instead of retraining the full deep model. This choice is important to assess whether discriminative information is already captured by compact visual descriptors or pretrained embeddings, to reduce dependence on large labeled datasets, and to provide a stronger comparison with deep approaches. To this end, we fine-tune and compare several traditional ML classifiers, including KNN, SVM, Logistic Regression, MLP, Random Forest, LightGBM, CatBoost, XGBoost, and an Ensemble Voting classifier, selecting the best-performing models for each representation. Overall, traditional ML in RAMimgDroid serves both as an interpretable baseline and as an efficient downstream classifier for feature-based and low-data learning scenarios.

Explainability Implementation. In *RAMimgDroid*, Grad-CAM visualizations were computed directly on the convolutional feature maps of each model using *PyTorch* hooks, and the resulting heatmaps were overlaid on the corresponding images. A custom reverse-mapping routine translated highlighted pixels back into their exact byte offsets by inverting the rasterization process. In *RAMstrDroid*, for string-based analysis, we extracted all human-readable strings from each memory dump and APK resource file using strings and custom *regex* filters, then applied *TF-IDF* vectorization and logistic regression. *LIME* and *SHAP* were run on these models to identify influential tokens and compare them with byte-level indicators discovered by *Grad-CAM*.

4.3. Dataset distribution

The dataset has been created with malware samples downloaded from *VirusTotal* and submitted there between 2023 and 2024 but released for different Android versions (i.e. API level) that we used to retrieve the year release as VT only has the submitted year. The malware dataset is illustrated in [Table 1](#) and the benign dataset in [Table 2](#).

Temporal and Source Bias Considerations. The benign and malicious samples used in this study come from different collection sources: benign applications were collected from AndroZoo, whereas malicious applications were obtained from VirusTotal. We acknowledge that different sources may introduce provenance-related confounding factors, such as differences in metadata availability, packaging style, or submission/collection procedures but correspond to real-world different play-stores, authors, styles, etc. However, in this study, we focus on the runtime execution of the app by dumping its RAM and converting into RGB images and such metadata are not objective of this study. Additionally, the collection or submission date should not be interpreted as the actual temporal characteristics of the APK. An application submitted to VirusTotal in a given year may have been developed earlier or

may target older Android API levels; similarly, the AndroZoo collection period does not necessarily correspond to the APK build period. For this reason, we characterize both benign and malicious applications using SDK-related information extracted with *aapt*,¹⁰ namely MinSDK and TargetSDK, and report the corresponding SDK/API-year distributions in [Table 2](#) for benign and [Table 1](#) for malware. MinSDK specifies the minimum Android API level required for an application to run, whereas TargetSDK specifies the Android API level against which the application was designed and tested, influencing the platform behavior and compatibility rules applied to it. The SDK/API-year distribution is obtained by mapping each MinSDK and TargetSDK API level to the release year of the corresponding Android version. Therefore, these years characterize the Android platform generations supported or targeted by the application. These tables show that benign and malicious applications are not separated into disjoint temporal/API spaces. In particular, the benign set is strongly concentrated in TargetSDK/API years 2020–2022, and the malware set also contains samples mapped to the same TargetSDK/API years. Therefore, the dataset does not simply compare recent benign applications against malware belonging to an unrelated APK/API period. Instead, both classes include applications targeting overlapping Android/API generations, although the malware set remains more heterogeneous and also includes samples targeting older Android versions. To further assess whether *RAMimgDroid* relies only on temporally specific artifacts, we added a temporal-drift analysis, later described in [Section 5.3](#). This additional experiment is used to verify whether the learned RAM-image representation can generalize to malware built with SDK/API that were released later with respect to the samples in the training set. The metadata reported in [Table 1](#) are not used as explicit input features, i.e. package names, runtime process names, VirusTotal tags, family labels, collection metadata, and SDK/API-year information are not provided to the CNN. The classifier receives only the image representation derived from memory bytes, while metadata are used for dataset characterization, APK-level aggregation, and post-hoc analysis. [Table 1](#) is therefore descriptive: it reports *VirusTotal* tags and runtime-observed package/process names to characterize the malware set, but *RAMimgDroid* performs only binary benign/malicious classification and does not classify malware families, tags, or process names.

Finally, although the SDK/API-year analysis and the temporal-drift experiment mitigate the concern that the model is simply learning a triv-

¹⁰ Android Asset Packaging Tool (AAPT) is part of the Android build toolchain for compiling and packaging application resources; it can also inspect APKs and extract metadata <https://developer.android.com/tools/aapt2>

Table 2
Benign dataset SDK/API-year distribution at APK level.

SDK/API Year	MinSDK APKs	TargetSDK APKs
2009	1	0
2010	5	0
2011	21	0
2012	322	1
2013	408	1
2014	885	0
2015	399	2
2016	161	0
2017	58	1
2018	9	1
2019	0	4
2020	0	1715
2021	0	410
2022	0	133
Total APKs	2278	2278

The SDK/API year is derived from the MinSDK and TargetSDK values extracted with aapt and does not represent the APK collection or submission year.

ial temporal split, they do not completely eliminate all possible source-related effects. Time-specific, source-specific, or family-specific artifacts may still appear in memory. We therefore explicitly report this aspect as a threat to validity and consider fully source-matched and family-stratified evaluations as future work.

Memory Areas and Image Characteristics. Android RAM is dominated by dynamically linked and optimized code artifacts, while raw APK files are rarely present in memory. The derived image representations of *RAMingDroid* show skewed, mostly elongated dimensions that are similar across benign and malicious samples, making size alone non-discriminative. Instead, distinctive structural patterns emerge when dimensions are analysed per memory region (e.g., uniform stacks, compact libraries, heterogeneous application artifacts). Therefore, discrimination depends on structural and content-level characteristics rather than image size. A detailed breakdown is provided in [Appendix A](#).

4.4. Acquisition methodologies

Analysing current works on Android malware detection based on memory analysis, we observed that such approaches typically rely on external tools for memory acquisition. For example, *RGB.Mem* [20] leverages *Memfetch*¹¹ to acquire the analysed process memory. For this reason, we conducted a comparative analysis¹² between the acquisition methodologies of *Memfetch* and *Fridump* by executing the same application under identical environmental conditions. The extracted memory dump sections were compared using the *difflib*¹³ library, based on the Ratcliff/Obershelp similarity coefficient.¹⁴ Performing this analysis, we observed that the content of the user-space memory regions is identical across the two tools, demonstrating that both acquisition methodologies are valid and capable of extracting the same memory data at the user-space level. The differences observed in the number and structure of output files are related to the acquisition strategies. *Memfetch* operates at a lower level, reading process memory through kernel interfaces such as `/proc/<pid>/mem` and *ptrace*, which allows it to extract larger contiguous memory ranges and, in some cases, kernel-related or kernel-assisted mappings that are not exposed as standard VMAs. In contrast, *Fridump* relies on *Frida* `ReadProcessMemory` API and

strictly adheres to the Virtual Memory Area (VMA) boundaries defined in `/proc/<pid>/maps`, dumping each VMA as an independent file and thereby fragmenting some logical regions into multiple smaller chunks. As a result, *Memfetch* typically produces a higher number of dump files, reflecting a more inclusive and fine-grained capture rather than missing coverage in *Fridump*. Nevertheless, when the analysis is restricted to user-space memory regions and the extracted fragments are normalized at the semantic level, the comparison confirms that *Memfetch* and *Fridump* produce consistent and equivalent acquisition results. As we were interested in the user-space memory acquisition and *Fridump* produces a lightweight coherent result, we opt for this acquisition methodology.

5. Core experimental results

This section reports the main findings of *RAMingDroid*. We first identify the memory-region configurations that are most representative and informative for Android runtime malware detection. We then assess how different region encodings (i.e. 1D vs. 2D, RGB vs. grayscale) affect classification, and proceed with feature-based baselines and CNN-based models. Next, we evaluate robustness under temporal variation, considering both execution-time effects (i.e. the impact of the specific memory dump time during app execution) and longitudinal malware evolution, namely the performance changes that arise when malware samples belong to different collection years and, therefore, reflect different families, behaviors, and Android ecosystem conditions, including changes in OS versions and in the application APIs leveraged by the apps.

After selecting the best-performing configuration, we analyse model behavior through attribution, highlighting the most influential memory regions and pixels and mapping them back to the corresponding byte offsets in the original dumps. This reverse-mapping enables the extraction of concrete meaningful artifacts, i.e. IoC that support human analysis and facilitate comparison with the selected state-of-the-art tools (*VirusTotal* and *MobSF*), in particular when using *RAMstrDroid*. The implications of these results, together with qualitative evidence and tool-based comparisons, are discussed in [Section 6](#) and further examined through case studies in [Section 6.2](#).

5.1. Memory region classification

This experiment investigates how the choice of memory-area selection impacts the detection performance of a fixed model. We adopt *ResNet-18* [90] as a representative backbone due to its relatively small parameter count and lower complexity, which reduces overfitting on limited or specialized datasets (e.g. malware and a few-shot settings) and enables faster training and inference on constrained hardware [97]. Following the memory taxonomy in [Section 2.2](#), we evaluate five configurations, reported in [Table A.3](#): (i) *memory*, grouping memory-management regions (`stack`, `vdso`, `vvar`, `vsyscall`, `heap`); (ii) *stack* only, capturing per-thread execution context (call frames, local variables, arguments, return addresses, and saved registers); (iii) *base*, including code artifacts and optimizations (`base.apk`, `base.vdex`, `base.odex`); (iv) *full apk*, extending *base* with execution-related components such as native libraries, preferences, databases, and cookies; and (v) *data*, comprising application and system runtime files (app data, caches, temporary storage) together with the *full apk* code components. For each configuration, we include the *memory* and *stack* regions to preserve core runtime context. As reported in [Fig. 2](#) and detailed in [Appendix A \(Table A.4\)](#), the performance differences across the considered configurations are overall limited, with most combined regions yielding true-positive and true-negative rates above 96%. In particular, *base-apk* & *stack* achieves 97.4% TP, 2.6% FP, 1.6% FN, and 98.4% TN, while *apk-full* & *memory* reaches the highest TP rate (97.7%) with only 2.3% FP. The selected *data* + *stack* configuration remains slightly below these peak results, with 96.6% TP, 3.4% FP, 3.0% FN, and 97.0% TN, but it is retained as the primary setting because it captures a

¹¹ <https://github.com/citypw/lcamtuf-memfetch>

¹² The full results of the comparative analysis of acquisition methodologies are available at <https://doi.org/10.5281/zenodo.18866707>

¹³ <https://docs.python.org/3/library/difflib.html#module-diffli>

¹⁴ <https://xlinux.nist.gov/dads/HTML/ratcliffObershelp.html>

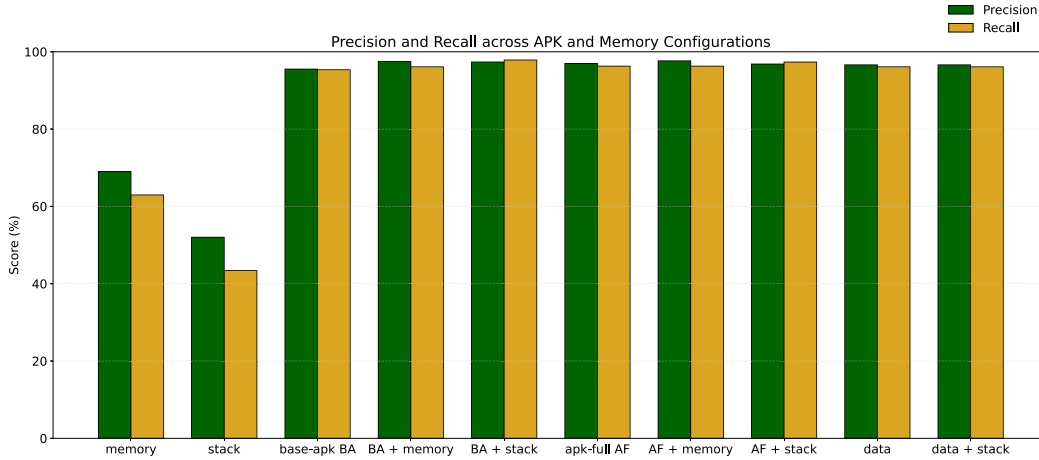


Fig. 2. Precision and Recall for the different selection of combined memory-region configurations with ResNet-18.

broader combination of runtime and application-specific artifacts while preserving a well-balanced error profile. More specifically, this configuration includes initialized and uninitialized data sections together with stack context, thereby covering app resources, intermediate execution data, thread-level runtime traces, installation and optimization stages (e.g. compiled artifacts), cached and local runtime files, app-specific resources. They collectively provide broader behavioral coverage than code-only or memory-only subsets, that are not fully represented by code-only subsets. This suggests that the core APK artifact already contains most of the discriminative signal, while runtime regions provide complementary evidence that further improves robustness when used in targeted combinations rather than in isolation. Overall, these results suggest that the core APK artifact contains most of the discriminative signal, whereas selected runtime regions provide complementary evidence that improves robustness when combined with application data rather than considered in isolation.

Takeaways

- *Data + stack* yields the best performance balance.
- Runtime data improves detection over code-only regions.
- Stack context is critical for robust classification.

5.2. Image representation

The main goal of this section is to identify the most representative image encoding and learning configuration for Android runtime malware detection. To this end, Table 3 provides an overview of the main design dimensions considered in *RAMingDroid*, including 1D versus 2D image construction, RGB versus grayscale encoding, and the different learning settings adopted in the ablation study. The table should therefore be read as a comparative summary of how representation choices affect detection performance, while the following paragraphs discuss each aspect in more detail.

1D vs 2D Encoding. We first analyse the impact of spatial representation by comparing 1D and 2D encodings. Results in Table 3 show that 1D representations consistently outperform their 2D counterparts in malicious detection rate. While 2D images allow the use of standard vision architectures, they introduce artificial vertical adjacency between non-contiguous memory bytes, potentially generating spurious correlations. By contrast, the 1D encoding preserves byte sequentiality and provides more stable and robust performance across models.

RGB vs Grayscale. We then evaluate the impact of color space by comparing RGB and grayscale encodings, shown in Table 3. RGB representations consistently achieve higher or more balanced TPR for

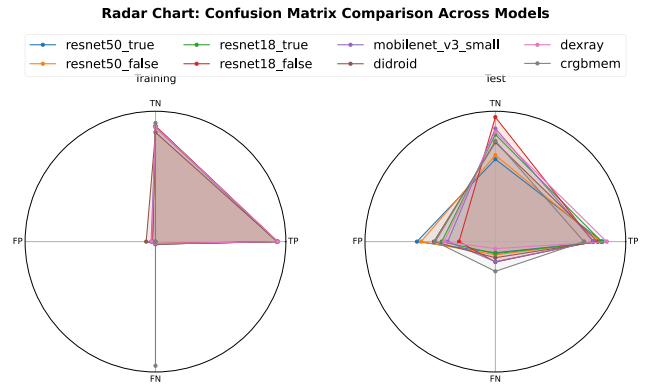


Fig. 3. Training (last epoch) and testing confusion matrix for pre-trained PyTorch CNNs (resnet18, resnet50 and mobilenetv3-small) considering pre-trained weights or not and state-of-the-art CNNs implementations (didroid, dexray, crgbmem).

both benign and malicious classes. This suggests that multi-channel encoding captures richer structural patterns from memory regions, whereas grayscale representations may lose discriminative information. Although grayscale performs competitively in some configurations, RGB combined with 1D encoding yields the best overall trade-off between accuracy and error rates.

CNN Architectures. Building upon the best-performing *data-stack* configuration, we compare multiple CNN architectures, including *ResNet18/50*, *MobileNet* variants, *ConvNeXt*, *EfficientNet*, *VGG16*, and *ViT*, as well as malware-specific networks such as *DiDroid*, *DexRay*, and *RGB_Mem*. As shown in Fig. 3 and detailed in Appendix A (Table A.5), literature CNN implementations underperform on this dataset. Across all experiments, *ResNet-18* achieves the best balance between accuracy and FPR/FNR. Interestingly, loading *ImageNet* pretrained weights does not significantly affect performance, indicating that natural-image priors are not decisive for memory-derived images.

Handcrafted Features and Few-Shot Learning. We further compare deep models with handcrafted image features combined with traditional ML classifiers. First, traditional descriptors (e.g., color moments, LBP, HOG, GLCM, statistical features) provide moderate performance but remain clearly inferior to end-to-end CNN approaches as in Figs. 4 and 5. Secondly, applying Few-Shot Learning in the last layer of ResNet-18 improves robustness compared to handcrafted pipelines; however, the full end-to-end CNN training remains overall superior. These results confirm that deep representation learning is more suitable than man-

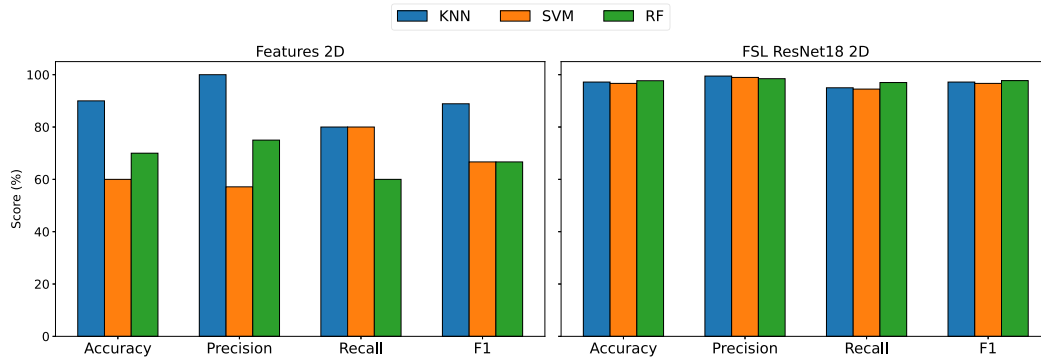


Fig. 4. Details of Feature classification for 2D RGB images. Accuracy, precision, recall and F1 metrics for the classification using all the features (color moments, edge, frequency, GLCM, hog, HSV, Hu, LBP, RGB, statistics) classified with fine tuned algorithms (from left to right: KNN, SVM, Random Forest - RF) compared with Few Shot Learning (FSL) in the last layer of ResNet18.

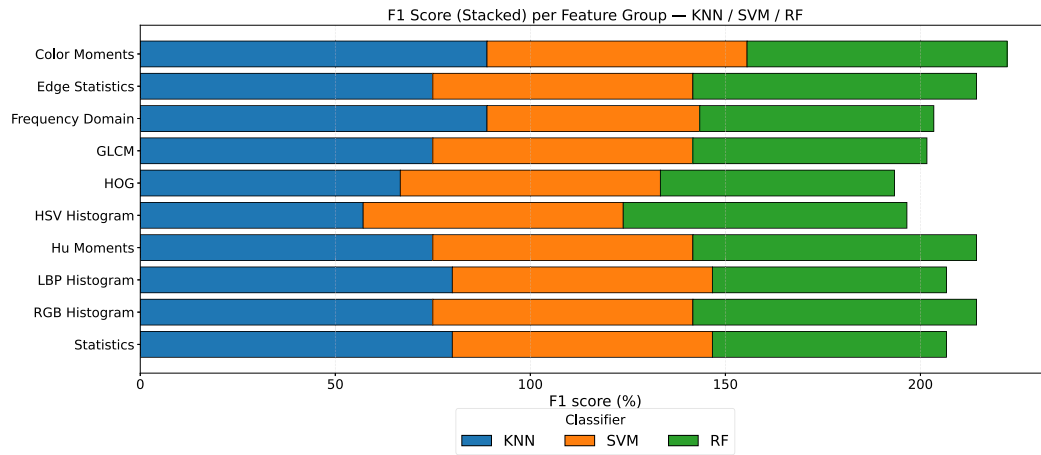


Fig. 5. Details of feature classification for 2D RGB images. F1 score for the classification using all the features (color moments, edge, frequency, GLCM, hog, HSV, Hu, LBP, RGB, statistics) classified with fine tuned algorithms (from left to right: KNN, SVM, Random Forest - RF).

ual feature engineering for runtime memory images as in [Appendix A](#) ([Fig. A.1](#) and [Table A.6](#)).

Attribution-Based Ablation. To better understand model behavior, we conduct attribution-based ablation experiments by removing or isolating the most influential memory regions and pixels identified via *Grad-CAM* ([Table 3](#)). When the most relevant regions are removed from RGB images, malicious TPR collapses dramatically (e.g., down to 2.12% for ResNet18_1D), while benign TPR remains near 100%, demonstrating that these regions are critical for malware discrimination. A similar degradation is observed when removing the most influential pixels. Conversely, classifying only the most relevant regions or pixels does not fully recover performance, indicating that although highly discriminative areas are essential, contextual information from less salient regions also contributes to robust detection. Overall, these results confirm that the learned representations capture meaningful runtime artifacts rather than spurious patterns, and that 1D RGB combined with ResNet-18 provides the most stable and effective configuration, see [Table 3](#). Importantly, these class-wise TPR results should be interpreted as complementary to the shuffle-based validation accuracy reported in the robustness analysis in [Table A.7](#): the former describes how errors are distributed across benign and malicious samples, whereas the latter summarizes the aggregate stability of the model across repeated splits.

Takeaways

- Both 1D RGB and grayscale encodings yield the most stable and effective detection performance.
- ResNet-18 provides the best balance between accuracy and error rates across configurations.
- Attribution analysis confirms that influential regions and pixels are critical for malware discrimination.

5.3. Robustness analysis

We evaluate the robustness of *RAMingDroid* along four complementary dimensions: temporal drift across years, data shuffle, dump-time sensitivity during execution, and resistance to adversarially crafted samples.

Temporal Drift. To assess long-term generalization, we trained ResNet-18 in malware samples from 2017–2018 (291 samples), validated in 2020 samples (159), and tested in 2022 samples (219). Despite differences in Android versions and runtime environments, the model demonstrates strong generalization, achieving 95.56% training accuracy,

Table 3
TPR (%) for benign and malicious samples across CAM modes and models.

Group	Mode	Custom1D		ResNet18_1D		ResNet18_2D	
		Benign	Malicious	Benign	Malicious	Benign	Malicious
Classification							
	RGB	97.81	97.88	98.19	97.52	97.41	96.11
	Gray	97.16	98.41	98.45	97.35	96.87	96.64
Abl. study							
	Remove regions RGB	100.00	16.40	100.00	2.12	99.73	4.60
	Remove pixels RGB	99.61	20.00	100.00	0.00	99.46	42.30
	Classify regions RGB	97.94	81.06	99.87	0.00	99.86	95.22
	Classify pixels RGB	98.19	41.24	100.00	4.25	100.00	1.42

97.75% validation accuracy (98.8% TPR for malware), and 97% test accuracy with a 96.8% malware TPR. This indicates that memory-based representations capture structural runtime behaviors that remain stable across Android evolutions. In contrast, training exclusively on pre-2017 samples leads to unstable predictions, likely due to architectural transitions such as the shift from DVM to ART, suggesting that runtime evolution impacts memory patterns when training data is too temporally distant.

Shuffle. Randomly permuting the sample order at the beginning of each epoch is a key step to prevent the optimizer from exploiting spurious correlations induced by data ordering and to improve generalization. In our setting samples can be naturally clustered by acquisition time, device/app provenance, malware family, or preprocessing pipeline. In this way, presenting data in a fixed order may create long homogeneous stretches (e.g. many malicious or many benign samples in sequence, or visually similar dumps back-to-back), producing biased gradient updates and slower/less stable convergence. Under this configuration, the model exhibits fast convergence and stable learning dynamics across epochs. Validation accuracy consistently stabilizes between 87% and 89%, with peak performance reaching 89%, while ROC-AUC and PR-AUC remain robust at approximately 0.92–0.93, confirming strong class separability and reliable ranking capability. The confusion matrices consistently show high malware recall (≈ 93 – 96%) and slightly lower benign recall (≈ 80 – 84%), indicating a systematic bias toward minimizing false negatives. Although occasional transient drops appear during intermediate epochs, the model rapidly recovers and no progressive degradation is observed in later stages, suggesting limited overfitting. Shuffle and time-drift results are detailed in [Appendix A \(Table A.7\)](#). These aggregate shuffle results should therefore be interpreted together with, rather than as a direct substitute for, the class-wise TPR values reported in [Table 3](#), since accuracy summarizes overall performance whereas TPR highlights class-specific error behavior.

Dump-Time Sensitivity. We further analyse robustness with respect to execution time. For a subset of previously misclassified applications, memory was dumped repeatedly over a two-minute window at 10-s intervals (excluding the first 15 s after startup). In most cases, malware becomes detectable as execution progresses, indicating that early dumps may miss delayed malicious payload activation. A detailed experiment on a Chrome-based malware sample¹⁵ shows that all later acquisitions are correctly classified as malicious. We also observe that the aggregation strategy influences detection: using the average probability across regions provides stable global predictions, whereas the maximum strategy increases sensitivity but also raises false positives. We therefore adopt the average aggregation for standard detection and reserve the maximum strategy for deeper inspection of ambiguous or previously misclassified samples.

SDK Analysis. To further assess whether the classification performance could be driven by TargetSDK-related artifacts ([Tables 1 and 2](#)), we conducted two complementary controlled experiments shown in [Table 4](#). First, we constructed an exactly TargetSDK-matched dataset con-

taining 297 benign and 297 malicious APKs, with identical class distributions across TargetSDK 30, 31, 32, and 33. Across five repeated splits, all three CNN configurations (i.e. ResNet18 2D, ResNet18 1D and Custom 1D) maintained high performance, with mean accuracies ranging from 95.11% to 97.02% and ROC-AUC values from 0.9874 to 0.9931; in particular, ResNet18-2D achieved $97.02 \pm 1.90\%$ accuracy and 0.9922 ± 0.0082 ROC-AUC. These results indicate that the strong classification performance cannot be explained solely by the different TargetSDK distributions of benign and malicious applications. Second, we evaluated a stricter cross-TargetSDK setting using balanced cohorts of 128 benign and 128 malicious APKs for each SDK, training and validating exclusively on one TargetSDK and testing on the other unseen TargetSDK. When training on SDK 32 and testing on SDK 30, performance remained high, reaching $92.11 \pm 1.73\%$ accuracy for Custom1D and $91.25 \pm 7.40\%$ for ResNet18-1D, with ROC-AUC values of approximately 0.986. The reverse SDK 30-to-32 direction was more challenging, with mean accuracies between 66.17% and 69.84%; however, this degradation is consistent with a stronger cross-SDK domain shift, since applications targeting a newer SDK may exhibit different API usage, libraries, compilation and dependency patterns, compatibility behavior, and consequently different runtime-memory structures. Importantly, despite this shift, ROC-AUC remained high, ranging from 0.9377 to 0.9642, showing that the models largely preserved their ability to rank malicious APKs above benign ones even in an unseen SDK domain. The lower threshold-based accuracy therefore appears to reflect, at least in part, a cross-SDK calibration shift rather than a complete loss of the underlying malware-discriminative signal. Overall, the two experiments show that *RAMingDroid* is robust to TargetSDK-related class imbalance and retains substantial discriminative capability across unseen SDK domains, while also indicating that fixed decision thresholds are not fully invariant to changes in the TargetSDK domain.

Adversarial Robustness (ELSA Dataset). To evaluate resistance against adversarial manipulation, we tested our method on a restricted dataset of adversarial APKs from the ELSA competition, part of the ELSA Horizon Europe EU project,¹⁶ specifically crafted to bypass detectors such as *Drebin* and *Mamadroid*. When evaluated using *Entropyzer* for dynamic analysis, these samples evade most traditional classifiers, with only SVM achieving moderate recall (80%). In contrast, our RAM-based approach successfully detects all adversarial samples. This result highlights that memory-level representations capture runtime artifacts that remain observable even when static features and entropy-based signals are manipulated to resemble benign behavior.

Overall, these experiments demonstrate that memory-based detection is robust to temporal drift, execution-time variability, and adversarial evasion techniques, provided that acquisition timing and aggregation strategies are properly handled.

¹⁶ The ELSA competition website is available at the following link: <https://benchmarks.elsa-ai.eu/?ch=6>. The related dataset is available from the competition organizers upon motivated request.

¹⁵ 55f6ddecc88252158d399cf64d22b11c2ecf0a8130bc30c266ccf03d45a59ab3

Table 4

TargetSDK robustness analysis. The TargetSDK-matched experiment evaluates classification after removing the TargetSDK–class correlation, whereas the cross-TargetSDK experiment evaluates generalization to an unseen TargetSDK domain. Results are reported as mean $\pm$ standard deviation over five runs.

Experiment	Model	Accuracy (%)	Benign TPR (%)	Malware TPR (%)	ROC-AUC
<i>TargetSDK-matched: identical TargetSDK distribution for benign and malware</i>					
Matched TargetSDK	ResNet18-2D	97.02 $\pm$ 1.90	99.15 $\pm$ 1.90	94.89 $\pm$ 4.15	0.9922 $\pm$ 0.0082
Matched TargetSDK	ResNet18-1D	95.74 $\pm$ 2.38	99.57 $\pm$ 0.95	91.91 $\pm$ 4.61	0.9874 $\pm$ 0.0174
Matched TargetSDK	Custom1D	95.11 $\pm$ 3.81	98.72 $\pm$ 2.85	91.49 $\pm$ 7.52	0.9931 $\pm$ 0.0063
<i>Cross-TargetSDK: train/validation on TargetSDK 30, test on unseen TargetSDK 32</i>					
SDK 30 $\rightarrow$ 32	ResNet18-2D	68.4 $\pm$ 19.0	97.8	39.1	0.9642
SDK 30 $\rightarrow$ 32	ResNet18-1D	66.2 $\pm$ 15.7	99.2	33.1	0.9377
SDK 30 $\rightarrow$ 32	Custom1D	69.8 $\pm$ 20.3	98.3	41.4	0.9480
<i>Cross-TargetSDK: train/validation on TargetSDK 32, test on unseen TargetSDK 30</i>					
SDK 32 $\rightarrow$ 30	ResNet18-2D	86.3 $\pm$ 20.3	94.5	78.0	0.981
SDK 32 $\rightarrow$ 30	ResNet18-1D	91.3 $\pm$ 7.4	92.7	89.8	0.986
SDK 32 $\rightarrow$ 30	Custom1D	92.1 $\pm$ 1.7	88.8	95.5	0.986

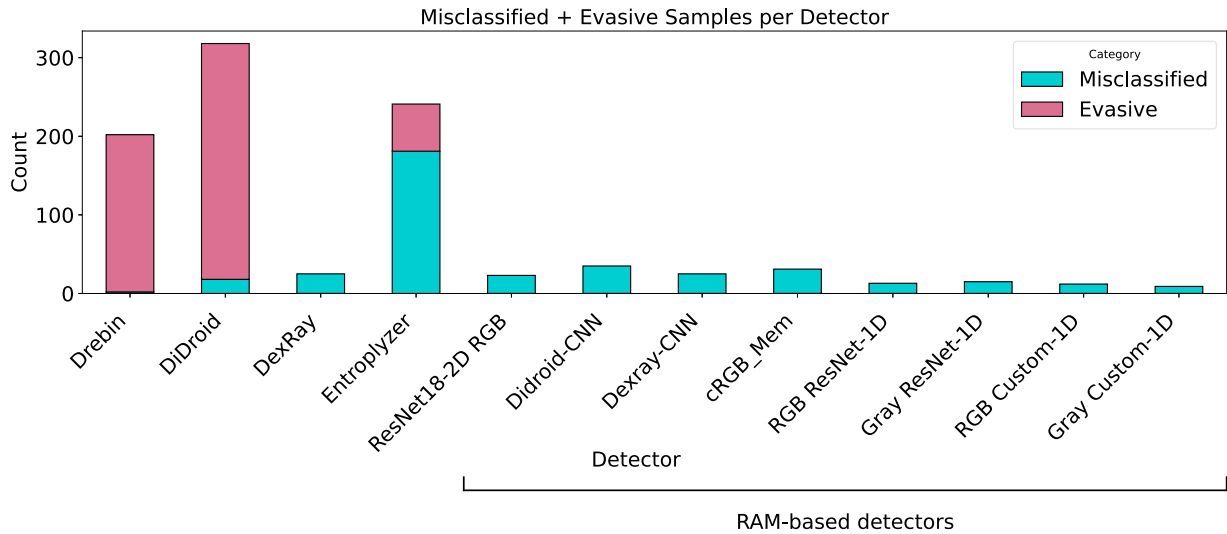


Fig. 6. Number of evasive (Blue, below part in histogram) and misclassified (Pink, upper part in histogram) samples in the different tools, algorithms and models. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Takeaways

- Memory representations remain stable across Android runtime evolutions.
- Proper dump timing and aggregation are critical for reliable detection.
- Runtime memory analysis effectively counters adversarial and evasive malware.

6. Discussion

In the following sections we will highlight specific cases and discuss the most important findings of our experiments.

6.1. Misclassified samples

Analysis Setup and Anti-Analysis Limitations. By using a subset of 573 malware samples for the testing phase, we analysed why and for which reasons some malware are not detected as shown in Fig. 6 and detailed in Appendix A (Table A.8). First of all, we tested our APK dataset with four of the most used approaches in the literature: *Drebin*

[2], *Didroid* [4] and *DexRay* [25] for static feature extraction, and an adapted version of *EntropyLyzr* [5] for dynamic analysis (we used their methodology but modified their code¹⁷). We immediately noticed that about 300 malware apps could not be analysed by those static tools because of anti-reverse countermeasures (i.e. errors during *apktool* and *androguard* analysis) and about 60 apps failed due to anti-dynamic mechanisms such as the impossibility of hooking specific functions.

We modified the *EntropyLyzr* code because it relied on *droidbot* to stimulate the APK. Due to its random input behavior (i.e. sometimes only adjusting device volume instead of interacting with the application), stimulation often failed and the application did not start, further increasing the number of not-analysed samples. As a consequence, no function could be hooked and the memory content remained empty. Therefore, we adopted the *uiautomator* approach used in our dataset generation pipeline, successfully dumping almost every application. It is worth noting that *EntropyLyzr* does not extract RAM content but only memory usage statistics through *dumppsys*.

Feature Engineering vs Memory-Based Analysis. Following traditional methodologies, specific features must be extracted from the APKs, leading to three main issues: (i) the APK may not statically or dynamically use the selected API; (ii) the APK may use the selected API but em-

¹⁷ https://github.com/slsanna/entropylyzer_update_SannaRAM

ploy adversarial techniques to bypass detection; (iii) feature engineering requires the analyst to carefully select discriminative APIs, often involving reverse engineering.

Conversely, in *RAMingDroid*, the analyst does not need to manually identify specific APIs. The only requirement is to execute the application and dump the memory, which contains plain-text artifacts and runtime behaviors, eliminating the dependency on predefined feature sets.

Retraining and Initial Misclassification Results. For anti-analysis APKs, we retrained *RAMingDroid* excluding the 360 samples that were used exclusively in the test set. Using the 2D RGB image representation with data-stack memory selection and a pre-trained ResNet18 backbone, our model correctly identified almost all malware samples, with only 10 misclassifications. These samples were immediately added to the misclassification analysis subset for deeper inspection.

Comparison with State-of-the-Art Approaches. We then compared the misclassified samples of *RAMingDroid* with those produced by *Drebin*, *DexRay*, *Entropyzer*, and *RGB_Mem* (including their CNN architectures). Considering the same test samples, our approach resulted in fewer misclassifications compared to other CNN-based methods. The image representation also plays a role: in 1D encoding, we observed only 9 misclassified samples using grayscale 1D images with a custom CNN architecture. We did not perform a systematic repeated-dump analysis on benign applications; therefore, the impact of repeated dumps on false positives remains part of future work.

Although *Drebin* reported only 2 misclassified malware samples compared to 22 in our approach, it failed to analyse approximately 200 APKs due to anti-analysis protections. In contrast, our memory-based method successfully extracted RAM from all applications. Importantly, whenever static or dynamic analysis failed, RAM analysis remained effective, as at least one CNN trained on the RAM dump detected the malware. Only 2 applications were misclassified by every literature approach and by *RAMingDroid*. These cases likely depend on intrinsic family characteristics or strong anti-analysis behaviors. In a few cases, misclassification was linked to inefficient dump timing, where an early dump at application startup was insufficient to capture malicious activity.

Family-, Tag-, and Year-Based Error Distribution. Misclassification errors are not uniformly distributed across families, tags, and years. Certain families such as Dnotua variants (FakeApp, AACF) and specific PUA subfamilies exhibit extremely high misclassification percentages (above 70%), while larger and more diverse families such as Opfake or Pjapps show only partial misclassification. This suggests that homogeneous subgroups sharing similar obfuscation techniques, runtime behaviors (e.g. reflection, runtime modules, clipboard usage), or structural characteristics are harder for the model to distinguish from benign applications, leading to cluster-level errors.

Tag correlation reinforces this observation: heavily obfuscated or stealthy apps involving send-sms, telephony, or GPS checks tend to be more error-prone, since such features overlap with legitimate behaviors in communication or system-service apps. Year-wise clustering (e.g. 2010–2011, 2017–2018) often corresponds to the introduction of new obfuscation strategies or underrepresented malware families in the training distribution. Notably, most applications in these categories are still correctly identified. Therefore, misclassification is not driven by a single feature but by complex combinations of family traits, obfuscation techniques, and contextual noise.

Homogeneity vs Heterogeneity Effects. In categories where 100% of the samples were misclassified (e.g. PUA.Dnotua/ FakeApp with all 16 samples incorrectly flagged), this typically reflects small and highly homogeneous subgroups where the classifier lacks discriminative power. Conversely, more heterogeneous families such as Trojan.Opfake/TrojanSMS (32 samples, 10 misclassified) or Trojan.Pjapps/Adrd (35 samples, 10 misclassified) exhibit partial misclassification, indicating that diversity in code and behavior improves generalization capability.

Temporal Dumping Strategy. To further investigate these cases, we selected representative applications across families, years, and tags, including the 9 anti-analysis samples misclassified by *RAMingDroid* in

the early-dump setting. For each APK, memory was dumped after an initial 15-s waiting period from application startup and then every 10 s for 30 s, resulting in three additional dumps per APK. Results indicate that detection performance strongly depends on dumping time: all 9 samples missed in the early dump were detected from the first additional snapshot. This confirms that early dumps may miss malicious runtime behavior, whereas later dumps allow the model to capture discriminative artifacts loaded during execution. Notably, network-based detection mechanisms could identify these samples more easily, suggesting that misclassification primarily depends on temporal memory acquisition rather than intrinsic invisibility of the malware.

Takeaways

- Traditional static and dynamic tools failed on over 350 malware.
- Most RAM errors occurred in small or homogeneous families.
- Detection depended on dump timing.
- All adversarial ELSA samples were correctly detected as malware.

6.2. Case study

In this section, we further highlight the importance of RAM analysis with *RAMstrDroid* by comparing the extracted data and meaningful artifacts, i.e. IoC, with *VirusTotal* and *MobSF* reports. According to the dataset distribution, we selected the *Chrome* (package name com.nameown12), *WhatsApp* (package name sigma.ma.le) and *Avast* (package name produces.amber.ultra) malware applications for comparative analysis. All of them belong to the trojan category. We also included the *Klee Prank Adventure* malware app, which was misclassified by every detector except the grayscale encoding using the custom 1D CNN.

Comparison with VirusTotal and MobSF. In these experiments, we considered the *VirusTotal* report, the static and dynamic analysis of *MobSF*, and *RAMstrDroid*, i.e. the strings extracted from RAM through early dumping. Naturally, RAM analysis does not provide malware labels, family attribution, or exploited vulnerability information as *VirusTotal* and *MobSF* do. Moreover, RAM cannot directly infer purely static properties such as permissions, certificates, or manifest declarations, unless the corresponding API calls are executed during the dump window.

However, *RAMstrDroid* reveals richer runtime artifacts compared to *MobSF* and *VirusTotal*. For example, RAM extraction retrieves clipboard content (e.g. captured text or images), file artifacts (e.g. cookies, temporary pictures), API usage logs (e.g. SEND_SMS, RECORD_AUDIO), file paths and hidden libraries (e.g. python2.7/_/ _MozillaCookieJar.pyo in KleePrank), runtime-loaded modules and reflective calls, URLs and emails triggered only at execution time (e.g. tsbogend[at]alpha[dot] // franken), runtime-resolved domains (e.g. ip-api.com), activated services and receivers (notification listeners, accessibility hooks), suspicious shared preferences, and temporary files. Therefore, *RAMstrDroid* captures concrete execution artifacts that static and lightweight dynamic tools may overlook.

Obfuscation, Encryption, and Runtime Decryption Evidence. RAM analysis is particularly effective against obfuscation and encryption techniques. While *MobSF* or *VirusTotal* typically report the presence of reflection or obfuscation mechanisms, *RAMstrDroid*, even with RAM early dumps, exposes the actual decrypted and unpacked artifacts. For instance, RAM clearly reveals control-flow obfuscation (renamed packages), runtime payloads embedded inside the application bundle, native libraries used for obfuscation, runtime dex loading and reflection, string hiding techniques (base64 encoding, encrypted strings decoded at runtime), cleartext network traffic, hardcoded keys, and custom cryptographic routines in native code.

In the case of *KleePrank*, Python .pyo files and temporary unpacked artifacts were visible during execution, indicating encrypted or packed

content in the APK that becomes accessible only at runtime. RAM captures post-decryption strings and network endpoints (e.g. `ip-api` calls) extracted with *RAMstrDroid*, as well as the precise moment when payloads are loaded or files are written. Static analysis, instead, can only observe encrypted blobs without accessing their decoded content.

Emulator Detection and Sandbox Reliability. Due to emulator-detection mechanisms, several malware samples could not be installed on the *Android Studio* emulator (ARM and x86_64). However, they successfully executed and allowed RAM dumping on *Genymotion*, showing that it was suitable for our tested dataset and environment.

String-Based NLP Classification and Explainability. In *RAMstrDroid*, we performed classification based solely on RAM-extracted strings using NLP techniques. For comparison, we also extracted strings statically from the APK, including the manifest, DEX files, and compiled or decompiled code with `apktool`. As shown in Table A.9, the RAM-based approach achieves the highest malware recall, detecting a larger proportion of malicious samples and therefore producing fewer false negatives. However, it exhibits lower malware precision and benign recall, resulting in more benign samples being incorrectly classified as malicious. The `apktool`-based approach obtains the highest accuracy on the successfully processed subset, but it fails to analyse 494 APKs and does not capture dynamically loaded, unpacked, or decrypted runtime artifacts. Therefore, RAM-extracted strings are not superior as a standalone malware detector in this setting, but provide complementary runtime evidence that can increase confidence in malicious predictions and reveal artifacts unavailable to static analysis, as demonstrated in Table A.10. From a forensic perspective, pure string analysis is limited because behavioral correlations are difficult to reconstruct. However, from a malware detection standpoint, deobfuscated strings, encryption keys, or cleartext network artifacts can serve as strong indicators of maliciousness.

The NLP-based classifier of *RAMstrDroid* achieved an F1-score of 87% in distinguishing benign and malicious samples. To interpret the results, we applied *SHAP* and *LIME* explainable AI methods, which consistently identified the strings related to the data memory regions among the most influential features. Although string-based classification is less robust than CNN-based image representations of RAM, *RAMstrDroid* can serve as a lightweight and fast detection layer, particularly useful in real-time scenarios because string extraction can be performed with lower computational overhead than image conversion and CNN inference. Additionally, strings analysis methodology can be used as an auxiliary module to detect obfuscated and anti-analysis malware as shown in Appendix A (Table A.10). We better position *RAMstrDroid* as an auxiliary tool instead of a standalone approach.

Overall Considerations. In summary, the most effective malware detection strategy emerges from the combination of static and dynamic tools. Static analysis enables rapid initial classification and identification of known indicators. Dynamic analysis provides behavioral insights. RAM inspection, both from *RAMimgDroid* and *RAMstrDroid*, plays a crucial complementary role by exposing hidden, decrypted, and runtime-generated artifacts that cannot be observed through static inspection alone. Together, these layers provide a comprehensive understanding of malicious applications.

Comparison between *RAMimgDroid* and *RAMstrDroid*. The comparison between *RAMimgDroid* and *RAMstrDroid* highlights the trade-offs between image-based and string-based memory analysis. The image-based approach preserves the raw byte structure of process memory and enables the model to learn low-level, non-textual patterns that may not be directly visible to human analysts or string extraction tools. This makes *RAMimgDroid* more suitable for robust malware detection, especially in the presence of obfuscation, anti-analysis techniques, runtime-loaded components, or memory artifacts that are not encoded as readable text. However, this approach requires memory-to-image conversion, CNN-based processing, and explainability techniques to map relevant pixels back to the corresponding memory bytes. Conversely, *RAMstrDroid* is faster, lighter, and more directly interpretable, since it ex-

poses human-readable runtime artifacts such as URLs, file paths, API names, commands, cookies, temporary files, and decrypted strings. This is evident in the case studies, where *RAMstrDroid* revealed artifacts such as clipboard data, hidden Python `.pyo` files, runtime-resolved domains such as `ip-api.com`, and suspicious API-related strings such as `SEND_SMS` and `RECORD_AUDIO`, which are useful for forensic interpretation and comparison with *VirusTotal* and *MobSF*. Nevertheless, *RAMstrDroid* may miss non-textual or structural memory evidence, such as repeated byte patterns, padding values, high-entropy fragments, or stack/data-region markers like `00`, `333333`, `ffffff`, and `deadbeef`. Although these patterns are not necessarily meaningful as readable strings, *RAMimgDroid* can still exploit them because every memory byte is preserved in the byte-to-pixel representation. Therefore, the two representations should be considered complementary: *RAMimgDroid* provides stronger low-level detection capability, while *RAMstrDroid* offers lightweight and human-readable evidence for forensic analysis.

Overhead and Latency. In its current implementation, *RAMimgDroid* is intended for sandbox-based malware analysis rather than real-time on-device protection, because memory acquisition, image conversion, and CNN inference introduce non-negligible computational and storage costs. A complete evaluation of acquisition time, conversion time, classification latency, CPU/RAM usage, and storage overhead is left as future work. *RAMstrDroid*, which relies only on string extraction from memory, is expected to be more suitable as a lightweight auxiliary layer, but its real-time deployment also requires dedicated overhead evaluation.

Takeaways

- RAM analysis complements static tools like VirusTotal and MobSF.
- In our tested dataset, Genymotion allowed execution of samples that failed on the Android Studio emulator.
- Combination of static, dynamic, and RAM-based analysis.

6.3. Answers to research questions

RQ1 - Detection. *Can malicious behavior be identified directly from by analyzing the process memory dumps?* Yes, Android malicious behavior can be detected from Android process memory dumps. By considering merely the reported detection performance, *RAMimgDroid* is comparable with state-of-the-art approaches. However, differently from those approaches, *RAMimgDroid* allows for the detection of anti-analysis malware, adversarial samples, and is resistant to time drift. For example, *RAMimgDroid* is able to detect malware that does not immediately exhibit malicious behavior, by continuously monitoring the process memory, i.e. detecting the malware as soon as it starts behaving maliciously.

RQ2 - RAM Acquisition. *Which are the most important memory sections and bytes?* We present to the human analyst the most important memory section representation for the whole dataset, i.e. having the least misclassified samples. These are the sections related to the stack and data of the application. Within this selection, *RAMimgDroid* finds the most influential components, which include the code sections and the stack. Inside each image representing a memory section, we managed to identify the most significant pixels and map them back to the corresponding bytes, i.e. identify the most significant bytes in the memory dump, sometimes even human-readable.

RQ3 - Data representation. *What is the best representation for image encoding?* According to the analysis of *RAMimgDroid* misclassified samples with respect to other state-of-the-art tools and techniques, the best representation for image encoding for malware representation is the 1D image encoding, both in the RGB and grayscale versions. In fact, this encoding produces fewer misclassified samples (only 9 over 500 malware) and can successfully classify a sample that all other tools, CNNs, and image representations classify as benign. Despite this, the performances

are not significantly different between 2D RGB images. However, from a conceptual point of view, 2D RGB images are not ideal because of the spurious correlation between vertical bytes representing non-adjacent memory bytes.

RQ4 - Literature advancement. *Can RAM analysis detect stealthy malware compared to other existing tools and techniques?* Yes, RAM can detect stealthy malware such as adversarial samples, anti-analysis malware, or malware using obfuscation techniques and runtime modules. Sometimes, memory-forensics analysis can misclassify samples when the dump is acquired before the malicious behavior is loaded. In such cases, the early dump may not contain discriminative malicious artifacts, whereas repeated or continuous memory acquisition can successfully identify the malware once the malicious routine is activated, as shown by the *RAMingDroid* results. At the same time, from a manual validation, string analysis from RAM dump (i.e. *RAMstrDroid*) leads to more IoCs detected than other tools such as *VirusTotal* and *MobSF* (e.g. cryptographic keys).

7. Limitations and future works

Despite promising results, our work still faces some limitations. First of all, we had a limited dataset of malware and acquisitions. Moreover, we did not consider the large scale analysis on the malware evolution, i.e. how it behaves in time during different acquisitions or at different events when the user performs specific actions. This analysis should also be extended to benign applications, in order to quantify the impact of repeated or continuous memory dumps on false positives. By releasing the code and methodology, we encourage the community in working on this topic. Specifically, different studies can be made, for example considering different encoding or comparing the full RAM acquisition with the one related to the target process. Additionally, different tests must be made granting different permissions to check the execution correlation according to the granted permissions and the runtime data on RAM. More in-depth analysis can be made on the different malware families and tags, addressing spatial and temporal drift, also through fully source-matched and family-stratified evaluation protocols. We claim that an efficient solution would be the integration of such methodology to other tools and techniques, providing to real-world samples a more complete and efficient malware detection and analysis pipeline i.e. integrating in *VirusTotal* the RAM analysis. Additionally, the image conversion of *RAMingDroid* cannot be applied in real-time Android malware monitoring as it requires high computational resources and time whereas while the device may already be compromised. However, we claim that RAM monitoring by extracting specific string patterns could be an efficient solution as presented in *RAMstrDroid*. Further studies on runtime real-world attack monitoring will be conducted, by also considering overhead and latency. At the same time, we claim to investigate whether privileged vendor-side or kernel-level integration could support less intrusive runtime memory monitoring on real devices, while preserving forensic reliability and limiting performance overhead. In fact, it is also interesting to compare different execution environments and sandboxes, including Genymotion, Android Studio emulators, and real devices, since evasive malware may react differently depending on the analysis environment. Last but not least, Android large-scale malware analysis via memory forensics can be done only in emulators as the memory extraction requires root privileges and malware analysis requires a clear environment (e.g. restore to factory settings) at every APK execution to not have traces of other previous APK behavior for the detection and classification. It derives that restoring a real device to factory settings, involves the deletion of the super-user privileges. Up to now, there is no automatic procedure to root an Android real device, as a combination of physical keys must be pressed. Additionally, on a real device, the rooting procedure takes many minutes of execution while for emulators it is much easier and faster. Last but not least, we also plan to consider specific threat models on memory manipulations to test the

robustness of *RAMingDroid* encoding in Android evasive malware detection.

8. Conclusion

In this paper, we propose a methodology for malware detection based on memory forensics, namely *RAMingDroid* and *RAMstrDroid*. The first one, *RAMingDroid*, is based on 1D and 2D image conversion in grayscale and color from memory dumps acquired during the execution of a target APK. The second approach, *RAMstrDroid*, is based on the extraction of UTF-8 strings from the same memory dumps acquired during the execution of a target APK. Our methodology is promising even against malware employing anti-analysis techniques, adversarial apps, and stealthy samples according to the dataset we adopted. We use *Genymotion* emulator to execute the app and dump the main memory content using *Fridump*. *Fridump* is built on top of *Frida* and extracts a process's memory at runtime. The memory is dumped at the early stage of the execution of the application, i.e. when it is just loaded in RAM after accepting the permissions.

In *RAMingDroid*, we convert the memory dumps into 2D and 1D RGB and grayscale images. From 2D images, we extract specific groups of common features for classification and also evaluate popular CNNs. Each image corresponds to a specific section of the RAM; therefore, each application is represented by a set of memory-region images. The experiments show that pre-trained CNNs are effective for this analysis scenario. We also find that the most significant memory areas are the sections related to data and stack. For 1D images, we propose an image-conversion algorithm and subsequently test two different CNN types. We also infer that RGB encoding slightly outperforms grayscale.

We checked our results with common literature approaches and tools, finding that both *RAMingDroid* and *RAMstrDroid* are comparable to current techniques, sometimes outperforming them. In fact, RAM analysis can detect anti-analysis malware and adversarial samples, but we claim that a combination of these tools is the best approach. We do not propose only a detector, but an analysis system because, given the image conversion of a memory dump, we can identify the most influential memory sections and pixels, thereby providing the human analyst with the most representative bytes and, when available, their readable content.

CRedit authorship contribution statement

Silvia Lucia Sanna: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Resources, Methodology, Investigation, Formal analysis, Data curation, Conceptualization; **Leonardo Regano:** Writing – review & editing, Visualization, Validation, Methodology, Conceptualization; **Davide Maiorca:** Writing – review & editing, Visualization, Validation, Methodology, Conceptualization; **Giorgio Giacinto:** Writing – review & editing, Visualization, Validation, Supervision, Resources, Project administration, Methodology, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

This work was carried out while Silvia Lucia Sanna was enrolled in the Italian National Doctorate on Artificial Intelligence run by Sapienza University of Rome in collaboration with the University of Cagliari. This work was partially supported by Project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU.

Data availability

A subset of data supporting this study are openly available from Zenodo at <https://doi.org/10.5281/zenodo.18866707> and code at <https://github.com/slsanna/RAMimgDroid>. The released material includes the source code for memory acquisition, image conversion, string extraction, model training, and evaluation, together with a subset of memory dumps, encoded images, extracted strings, and trained models. The full dataset, evaluation results, and split files can be provided to reliable research groups upon motivated request, subject to ethical and safety restrictions.

Appendix A. Additional details on the experimental setup and results

This Appendix provides additional details on the experimental setup and on the obtained results. In particular, [Table A.1](#) describes the experimental setup; [Table A.2](#) details the memory area distribution over the dataset; [Table A.3](#) describes the memory regions by names in the paper (graphs and tables), patterns used in the selection mechanism and provides a human-readable description of the area; [Table A.4](#) details the detection performances over the different memory areas selection; [Table A.5](#) reports the results on the detection over data_stack memory regions with different CNNs; [Fig. A.1](#) shows the performances over different ML algorithms comparing the overall features extracted from the 2D RGB images and using FewShotLearning after ResNet18 in 2D RGB images; [Table A.6](#) details the detection performances over different ML algorithms on the features extracted from the 2D RGB images; [Table A.7](#) represents the performances over shuffle and time drift [Table A.8](#) details the misclassified samples over the different state-of-the-art algorithms and tools and the different CNNs on the memory dumps converted in 2D and 1D RGB and grayscale images; finally [Table A.10](#) describes the most relevant strings extracted with SHAP, LIME and NLP algorithms in the strings dataset extracted from the memory dumps.

Table A.1
Experimental setup and evaluation strategy.

Category	Specification
Dataset Creation Environment	Kali-Linux 2024.4 virtual machine with KVM enabled (nested virtualization active).
Purpose of KVM	Hardware acceleration (VT-x/AMD-V), rooted emulator support, Frida-based dynamic analysis, reproducibility via snapshots, safe isolation.
CPU (Acquisition Machine)	AMD Ryzen 9 7945HX, 16 cores / 32 threads, 2501 MHz.
GPU (Acquisition Machine)	VMware SVGA II Adapter (host GPU: NVIDIA GeForce RTX 4090 Laptop GPU).
RAM (Acquisition Machine)	12 GB.
Emulator	Genymotion 3.7.1 (Business), Google Pixel 13 API 33, 4 GB RAM, super-user access enabled.
Analysis Software	Python 3.12.6, uiautomator2 3.2.6, Frida 16.5.2, Fridump 0.1 (commit 3e64ee0), Binary2Image (commit 3e44eaf).
Classification Machine CPU	Intel Xeon W-2235 @ 3.80 GHz.
Classification GPU	NVIDIA RTX A5000 (24 GB VRAM).
RAM (Classification Machine)	32 GB.
OS (Classification Machine)	Ubuntu 20.04.6 LTS.
ML Software	Python 3.8.10, scikit-learn 1.3.2, torch 2.4.1 + cu121, PIL 10.4.0.
Training Strategy	Region-level labeling: each memory region inherits the APK label (benign/malicious).
Testing Strategy (Primary)	APK-level aggregation: average predicted probabilities of regions; threshold = Youden's J statistics.
Evaluation Metrics	Accuracy, Precision, Recall, F1-score, TP, TN, FP, FN.
Misclassification Analysis	Logged misclassified APKs and corresponding memory regions.
Alternative Evaluation	Region-level testing (ablation study); not representative of real-world deployment.
Reported Results	Primary results reported at APK level; region-level used only for case studies and influential region analysis.

Table A.2
Distribution of android RAM memory areas and associated pixel-dimension profiles. Mappings indicate approximate region counts.

Memory Area	Mappings	Widths (px)	Resolutions	Dimension Profile	Structural Characteristics
Shared Libraries (.so)	> 1.1M	128, 256, 512	128 × 256; 256 × 336; 512 × 512	Compact, balanced	Dynamically linked native libraries mapped during execution.
Compiled (VDEX/ODEX)	DEX ~96k	32, 64, 128	32 × 43; 64 × 214	Narrow, elongated	Optimized bytecode segments associated with application logic.
ART/OAT Runtime	~104k	128, 256, 1024	256 × 512; 1024 × 1371 +	Tall, high variance	Precompiled runtime components generated by ART.
Java Archives (.jar)	~151k	64, 128	128 × 256; 256 × 512	Moderately regular	Framework libraries loaded into process memory.
APK Files	~10.7k	32, 64, 128, 1024	32 × 43 to 1024 × 6000 +	Highly heterogeneous	Application package contents mapped at runtime.
Resource Cache / idmap	~8.7k	1024 (fixed)	1024 × 1371 to 1024 × 4000 +	Fixed width	System resource overlay mappings.
Cookies / Web Data	~944	32, 64	Small regions	Sparse, irregular	Persistent web-related storage segments.
Stack	~49.9k	1024 (fixed)	1024 × 2730	Uniform tall	Per-thread execution stack allocations.

Table A.3

Memory-region selection configurations used to group process-memory mappings.

Paper label	Matched memory-map patterns	Description
<i>stack</i>	\[stack\]	Thread stack mappings
<i>memory</i>	\[stack\]; \[vvar\]	Minimal runtime-memory regions
<i>stack + vdso</i>	\[stack\]; \[vdso\]	Stack and VDSO mappings
<i>base odex + apk</i>	base\.odex; base\.apk	APK and ODEX artifacts
<i>base vdex + apk</i>	base\.vdex; base\.apk	APK and VDEX artifacts
<i>base odex + vdex</i>	base\.odex; base\.vdex; base_vdex	Optimized bytecode artifacts
<i>base odex</i>	base\.odex	ODEX artifact only
<i>base vdex</i>	base\.vdex	VDEX artifact only
<i>base-apk (BA)</i>	base\.apk; base\.vdex; base\.odex	Core APK-related mappings
<i>complete memory</i>	\[heap\]; \[anon:libc_malloc\]; \[stack\]; \[vdso\]; \[vvar\]; \[vsyscall\]; \[deleted\]; u:object_r.*; data@resource-cache@	Broad runtime-memory selection
<i>base-apk + memory</i>	\[stack\]; \[vvar\]; base\.apk; base\.vdex; base\.odex	Base APK plus runtime memory
<i>base-apk + stack</i>	\[stack\]; base\.apk; base\.vdex; base\.odex	Base APK plus stack
<i>data</i>	~/data/.*	Android /data mappings
<i>data + stack</i>	~/data/.*; \[stack\]	Data paths plus stack
<i>apk-full (AF)</i>	base\.apk; base\.vdex; base\.odex; /data/.*/lib/.+\.so\$; \[anon:dalvik.*\]; \[anon:.*\.apk.*\]; \.db\$; \.db-shm\$; \.db-wal\$; Cookies; Preferences; ua_preferences; data@resource-cache@; \.dex\$; /data/.+\.odex\$	Extended APK-related mappings
<i>apk-full + memory</i>	\[stack\]; \[vvar\]; base\.apk; base\.vdex; base\.odex; /data/.*/lib/.+\.so\$; \[anon:dalvik.*\]; \[anon:.*\.apk.*\]; \.db\$; \.db-shm\$; \.db-wal\$; Cookies; Preferences; ua_preferences; data@resource-cache@; \.dex\$; /data/.+\.odex\$	APK-full + memory
<i>apk-full + stack</i>	\[stack\]; base\.apk; base\.vdex; base\.odex; /data/.*/lib/.+\.so\$; \[anon:dalvik.*\]; \[anon:.*\.apk.*\]; \.db\$; \.db-shm\$; \.db-wal\$; Cookies; Preferences; ua_preferences; data@resource-cache@; \.dex\$; /data/.+\.odex\$	APK-full + stack
<i>system</i>	framework\.jar; boot\.vdex; boot-framework\.vdex; base\.odex	System/framework artifacts
<i>system-full</i>	dalvik-classes; framework\.jar; boot-framework\.art; boot-core-libart\.art; boot\.oat; boot-framework\.oat; boot\.vdex; boot-framework\.vdex; base\.odex	Extended system/runtime artifacts
<i>libraries</i>	\.so; \[anon:lib.*	Native library mappings

Table A.4

Column-normalized confusion matrices (in %) for different APK and memory region configurations using ResNet-18.

Configuration	TP	FP	FN	TN
all memory regions	69.1	30.9	26.8	73.2
stack	52.0	48.0	38.4	61.6
base-apk	95.5	4.5	3.5	96.5
base-apk & memory	97.5	2.5	3.0	97.0
base-apk & stack	97.4	2.6	1.6	98.4
apk-full	97.0	3.0	2.8	97.2
apk-full & memory	97.7	2.3	2.8	97.2
apk-full & stack	96.8	3.2	2.1	97.9
data	96.6	3.4	3.0	97.0
data & stack	96.6	3.4	3.0	97.0

Legend. *base-apk* refers to features extracted from the main APK binary only; *apk-full* includes all APK-related memory regions; *memory* denotes heap and runtime memory regions; *stack* refers to thread stack memory; *data* corresponds to initialized and uninitialized data sections.

Table A.5

Confusion matrices by CNNs: Train (best epoch), Train (last epoch), and Test. Values are column-normalized percentages: TP/FP over (TP + FP) and TN/FN over (TN + FN).

CNN Model	Pre-trained	Best ep.	Train (Best)				Train (Last)				Test			
			TP	FP	FN	TN	TP	FP	FN	TN	TP	FP	FN	TN
convnext-tiny	False	7	97.5	2.5	2.1	97.9	96.6	3.4	2.1	97.9	67.1	32.9	9.0	91.0
	True	7	98.0	2.0	3.6	96.4	94.7	5.3	2.2	97.8	68.5	31.5	12.9	87.1
crgbmem	False	2	96.4	3.6	5.2	94.8	0.0	0.0	51.2	48.8	59.6	40.4	22.8	77.2
dextray	False	10	98.0	2.0	1.6	98.4	97.0	3.0	1.6	98.4	68.5	31.5	6.0	94.0
didroid	False	8	96.1	3.9	2.7	97.3	92.9	7.1	2.2	97.8	62.4	37.6	14.0	86.0
efficientnet-b0	False	8	98.0	2.0	2.6	97.4	89.5	10.5	1.8	98.2	65.0	35.0	7.0	93.0
	True	10	97.0	3.0	1.6	98.4	97.0	3.0	1.6	98.4	64.3	35.7	8.6	91.4
mobilenet-v2	False	8	97.5	2.5	2.1	97.9	96.6	3.4	1.6	98.4	71.7	28.3	17.6	82.4
	True	9	97.5	2.5	1.6	98.4	97.5	2.5	2.1	97.9	71.6	28.4	18.2	81.8
mobilenet-v3-large	False	10	98.0	2.0	1.6	98.4	98.0	2.0	1.6	98.4	67.5	32.5	12.3	87.7
	True	7	99.0	1.0	2.1	97.9	95.2	4.8	1.6	98.4	65.7	34.3	5.9	94.1
mobilenet-v3-small	False	9	97.0	3.0	2.1	97.9	96.6	3.4	2.1	97.9	67.3	32.7	15.0	85.0
	True	7	97.1	2.9	1.1	98.9	93.8	6.2	1.1	98.9	67.3	32.7	16.8	83.2
resnet18	False	10	97.5	2.5	1.6	98.4	97.5	2.5	1.6	98.4	82.9	17.1	4.1	95.9
	True	10	97.5	2.5	2.1	97.9	85.8	14.2	0.6	99.4	96.6	3.4	3.0	97.0
resnet50	False	9	98.0	2.0	2.1	97.9	97.5	2.5	2.1	97.9	58.6	41.4	13.3	86.7
	True	10	97.5	2.5	2.1	97.9	97.5	2.5	2.1	97.9	57.7	42.3	11.9	88.1
vgg16	False	1	93.5	6.5	13.5	86.5	0.0	0.0	51.2	48.8	55.2	44.8	34.8	65.2
	True	1	97.9	2.1	4.6	95.4	0.0	0.0	51.2	48.8	71.0	29.0	16.3	83.7
vit-b-16	False	6	95.5	4.5	5.2	94.8	87.2	12.8	1.8	98.2	54.6	45.4	14.7	85.3
	True	8	97.9	2.1	4.6	95.4	91.5	8.5	2.8	97.2	58.1	41.9	10.2	89.8

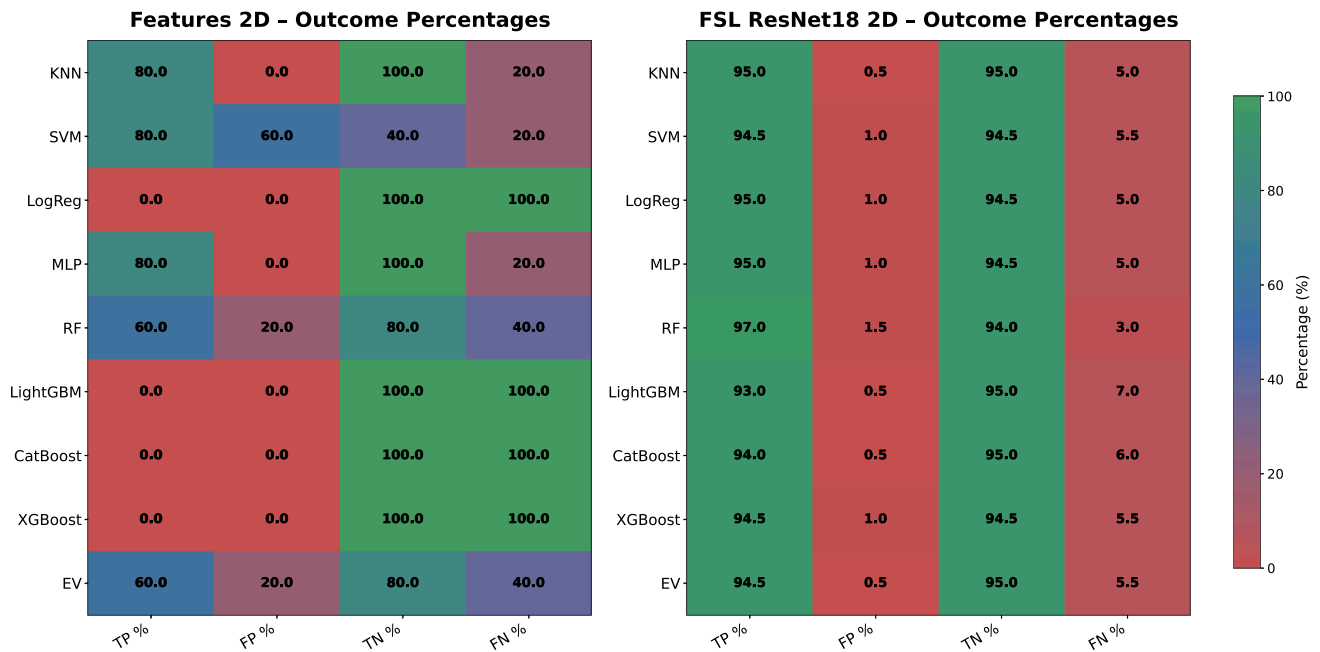


Fig. A.1. Comparison of FSL for ResNet18 (right) and overall extracted features (left) for 2D RGB images. Color map using fine-tuned algorithms (from left to right: KNN, SVM, Logistic Regression – LogReg, MLP, Random Forest – RF, LightGBM, CatBoost, XGBoost, Ensemble Voting – EV).

Table A.6

F1 score (%) across 2D Feature analysis groups (color moments, edge, frequency, GLCM, hog, HSV, Hu, LBP, RGB, statistics) and classifiers (KNN, SVM, Logistic Regression, MLP, Random Forest, LightGBM, CatBoost, XGBoost, Ensemble Voting).

Classifier	Color	Edge	Freq	GLCM	HOG	HSV	Hu	LBP	RGB	Stat
KNN	88.9	75.0	88.9	75.0	66.7	57.1	75.0	80.0	75.0	80.0
SVM	66.7	66.7	54.5	66.7	66.7	66.7	66.7	66.7	66.7	66.7
LogReg	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
MLP	88.9	66.7	66.7	54.5	60.0	66.7	66.7	80.0	54.5	61.5
RF	66.7	72.7	60.0	60.0	60.0	72.7	72.7	60.0	72.7	60.0
LightGBM	0.0	54.5	0.0	54.5	66.7	54.5	0.0	54.5	54.5	0.0
CatBoost	0.0	54.5	0.0	54.5	66.7	44.4	0.0	54.5	54.5	0.0
XGBoost	0.0	0.0	0.0	54.5	66.7	54.5	0.0	54.5	54.5	54.5
EV	66.7	72.7	57.1	60.0	33.3	72.7	72.7	75.0	72.7	60.0

Table A.7

Robustness analysis under random shuffling and temporal drift. Shuffle results are computed on the validation set across five repeated random splits, whereas temporal-drift results are evaluated across different malware years. Table 3 reports the main APK-level classification and CAM-ablation results.

Setting	Split / Protocol	PR-AUC	ROC-AUC	Acc. (%)	Benign TPR (%)	Malware TPR (%)	Note
Shuffle avg.	Random validation split, 5 repeats	0.9194 ± 0.0064	0.9262 ± 0.0037	88.65 ± 0.52	83.00 ± 0.79	94.30 ± 1.15	Stable convergence with low variance
Shuffle best	Best repeat on validation	0.9164	0.9278	89.50	83.00	96.00	Highest validation accuracy
Temporal drift	Train: 2017–2018; Val: 2020; Test: 2022	–	–	97.75 val.; 97.00 test	–	98.80 val.; 96.80 test	Strong cross-year generalization

Table A.8

Failure-aware comparison across detectors. “Failures excluded” reports the performance only on samples successfully analyzed by the detector. “Failures as errors” treats analysis/feature-extraction failures as missed malware detections. We report Precision (PR), Recall (REC), and F1-score (F1), considering the malware class as positive.

Detector	Failures Excluded			Failures as Errors			Failures (%)
	PR	REC	F1	PR	REC	F1	
Drebin	99.23	99.61	99.42	99.23	71.83	83.33	13.38
DiDroid	97.40	96.44	96.92	97.40	60.50	74.64	18.95
DexRay	57.22	81.32	67.17	57.22	81.32	67.17	0.00
EntropLyzr	97.40	98.77	98.08	97.40	89.35	93.20	6.11
ResNet18-2D RGB	96.62	96.11	96.36	96.62	96.11	96.36	0.00
DiDroid-CNN	96.55	93.98	95.25	96.55	93.98	95.25	0.00
DexRay-CNN	98.54	95.75	97.13	98.54	95.75	97.13	0.00
cRGB_Mem	91.14	94.69	92.88	91.14	94.69	92.88	0.00
RGB ResNet-1D	97.52	97.52	97.52	97.52	97.52	97.52	0.00
Gray ResNet-1D	97.86	97.35	97.60	97.86	97.35	97.60	0.00
RGB Custom-1D	97.02	97.88	97.44	97.02	97.88	97.44	0.00
Gray Custom-1D	96.19	98.41	97.29	96.19	98.41	97.29	0.00

Table A.9

Performance and coverage comparison of the NLP-based classification approaches. Acc. evaluated is computed only on the successfully analysed APKs, whereas failure-aware accuracy (FA Acc.) is computed over the complete test set by treating analysis failures as errors. Precision (P), recall (R), and F1-score are reported as percentages.

Approach	Ben.	Mal.	Missed	Coverage	Acc. eval.	FA Acc.	Ben. P	Ben. R	Ben. F1	Mal. P	Mal. R	Mal. F1	Macro F1	Weighted F1	Runtime
APK static strings	778	572	0	100.00	90.44	90.44	88.77	95.50	92.01	93.18	83.57	88.11	90.06	90.36	No
Apktool-extracted strings	454	402	494	63.41	92.87	58.89	89.70	97.80	93.57	97.23	87.31	92.01	92.79	92.84	No
RAM strings (data+stack)	779	571	0	100.00	82.89	82.89	91.14	77.92	84.01	74.85	89.67	81.59	82.80	82.99	Yes

Table A.10

Pattern-level distribution of explanatory features for the stack-based RAM classifier. Percentages represent the proportion of features appearing among the top-5 explanations per sample.

Pattern Class	SHAP (%)	LIME (%)	NLP TF-IDF (%)	Interpretation
Structural Stack Patterns (var=00, 333333, ffffff, deadbeef)	58	76	72	Dominant statistical memory artifacts
Android System / Path Tokens (data, system, odex, vdex, lib64, apk)	29	46	38	Contextual execution environment indicators
Semantic Malware Indicators (stage, metasploit, nameown12)	9	6	12	Explicit malicious artifacts
Noise / Random Bigrams (gi, x8, pz, lt, VN, PK)	4	12	8	Low-level lexical artifacts

References

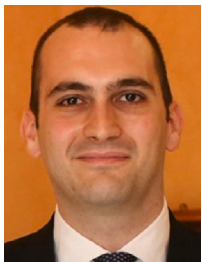
- [1] Moser A, Kirda E, Kruegel C. Limits of static analysis for malware detection. In: Computer security applications conference, annual. Los Alamitos, CA, USA: IEEE Computer Society; 2007, p. 421–30. <https://doi.ieeecomputersociety.org/10.1109/ACSAC.2007.21>. <https://doi.org/10.1109/ACSAC.2007.21>
- [2] Arp D, Spreitzerbarth M, Hubner M, Gascon H, Rieck K, Drebin: effective and explainable detection of android malware in your pocket. In: NDSS: The Internet Society; 2014, <http://dblp.uni-trier.de/db/conf/ndss/ndss2014.html#ArpSHGR14>
- [3] Onwuzurike L, Mariconti E, Andriotis P, Cristofaro ED, Ross G, Stringhini G. Madroid: detecting android malware by building Markov chains of behavioral models (Extended version). ACM Trans Priv Secur 2019;22(2). <https://doi.org/10.1145/3313391>
- [4] Rahali A, Lashkari AH, Kaur G, Taheri L, Gagnon F, Massicotte F. Didroid: android malware classification and characterization using deep image learning. Proceedings of the 2020 10th international conference on communication and network security 2020; <https://api.semanticscholar.org/CorpusID:232220016>
- [5] Keyes DS, Li B, Kaur G, Lashkari AH, Gagnon F, Massicotte F. Entropylyzer: android malware classification and characterization using entropy analysis of dynamic characteristics. In: 2021 Reconciling data analytics, automation, privacy, and security: a big data challenge (RDAAPS). 2021, p. 1–12. <https://doi.org/10.1109/RDAAPS48126.2021.9452002>
- [6] Demontis A, Melis M, Biggio B, Maiorca D, Arp D, Rieck K, et al. Yes, machine learning can be more secure! a case study on android malware detection. IEEE Trans Dependable Secur Comput 2019;16(4):711–24. <https://doi.org/10.1109/TDSC.2017.2700270>
- [7] Bhusal D, Rastogi N. Adversarial patterns: building robust android malware classifiers. ACM Comput Surv 2025;57(8). <https://doi.org/10.1145/3717607>
- [8] Pereira YH, da Costa Y MG, Zarpelão BB. A comprehensive analysis of adversarial attacks against android's malware detection models. In: 2025 32nd International conference on systems, signals and image processing (IWSSIP). 2025, p. 1–5. <https://doi.org/10.1109/IWSSIP66997.2025.11151853>
- [9] Tam K, Feizollah A, Anuar NB, Salleh R, Cavallaro L. The evolution of android malware and android analysis techniques. ACM Comput Surv 2017;49(4). <https://doi.org/10.1145/3017427>
- [10] Lindorfer M, Neugschwandtner M, Weichselbaum L, Fratantonio Y, van der VV, Platzer C. Andrubis – 1,000,000 apps later: a view on current android malware behaviors. In: 2014 Third international workshop on building analysis datasets and gathering experience returns for security (BADGERS). 2014, p. 3–17. <https://doi.org/10.1109/BADGERS.2014.7>
- [11] Islam R, Sayed MI, Saha S, Hossain MJ, Masud MA. Android malware classification using optimum feature selection and ensemble machine learning. Internet Things Cyber-Phys Syst 2023;3:100–11. <https://www.sciencedirect.com/science/article/pii/S2667345223000202>. <https://doi.org/10.1016/j.iotcps.2023.03.001>
- [12] Molina-Coronado B, Ruggia A, Mori U, Merlo A, Mendiburu A, Miguel-Alonso J. Light up that droid! on the effectiveness of static analysis features against app obfuscation for android malware detection. J Netw Comput Appl 2025;235:104094. <https://www.sciencedirect.com/science/article/pii/S1084804524002716>. <https://doi.org/10.1016/j.jnca.2024.104094>
- [13] Minnei L, Eddoubi H, Sotgiu A, Pintor M, Demontis A, Biggio B. Data drift in android malware detection. In: 2024 International conference on machine learning and cybernetics (ICMLC). 2024, p. 157–63. <https://doi.org/10.1109/ICMLC63072.2024.10935015>
- [14] Ruggia A, Nisi D, Dambrà S, Merlo A, Balzarotti D, Aonzo S. Unmasking the veiled: a comprehensive analysis of android evasive malware. In: Proceedings of the 19th ACM Asia conference on computer and communications security. Asia CCS '24; New York, NY, USA: Association for Computing Machinery. ISBN 9798400704826; 2024, p. 383–98. <https://doi.org/10.1145/3634737.3637658>
- [15] Dell'Orco D, Valeriani L, Bianchi G, Pellegrini A, Merlo A. Challenging antivirus against elusive android malware over time. In: 2025 Joint National Conference on Cybersecurity (ITASEC & SERICS 2025). 2025, 3962. <https://ceur-ws.org/Vol-3962/paper34.pdf>
- [16] Valeriani L, Bianchi G, Verderame L, Merlo A. Analyzing the impact of obfuscation on the runtime execution of android apps at kernel level. In: 2024 IEEE European symposium on security and privacy workshops (EuroS&PW). 2024, p. 712–17. <https://doi.org/10.1109/EuroSPW61312.2024.00087>
- [17] Soi D, Sanna SL, Liguori A, Zuppelli M, Regano L, Maiorca D, et al. On the feasibility of android stegomalware: a detection study. Ital Conf CyberSecur 2025;To appear.
- [18] Soi D, Sanna SL, Benedetti G, Liguori A, Regano L, Caviglione L, et al. Analysis and detection of android stegomalware: the impact of the loading stage. In: Proceedings of the 2025 ACM workshop on information hiding and multimedia security. IH&MMSEC '25; New York, NY, USA: Association for Computing Machinery. ISBN 9798400718878; 2025, p. 35–45. <https://doi.org/10.1145/3733102.3733122>
- [19] Dell'Orco D, Bernardinetti G, Bianchi G, Merlo A, Pellegrini A. Would you mind hiding my malware? Building malicious android apps with stegopack. Pervasive Mob Comput 2025;111:102060. <https://www.sciencedirect.com/science/article/pii/S1574119225000495>. <https://doi.org/10.1016/j.pmcj.2025.102060>
- [20] Ali-Gombe A, Sudhakaran S, Vijayakanthan R, Richard GG. cRGB_mem: at the intersection of memory forensics and machine learning. Forensic Sci Int Digit Investig 2023;45:301564. <https://www.sciencedirect.com/science/article/pii/S2666281723000732>. <https://doi.org/10.1016/j.fsidi.2023.301564>
- [21] Nataraj L, Karthikeyan S, Jacob G, Manjunath BS. Malware images: visualization and automatic classification. In: Proceedings of the 8th international symposium on visualization for cyber security. VizSec '11; New York, NY, USA: Association for Computing Machinery. ISBN 9781450306799; 2011, <https://doi.org/10.1145/2016904.2016908>
- [22] Shiva Darshan SL, Jaidhar CD. Windows malware detector using convolutional neural network based on visualization images. IEEE Trans Emerg Top Comput 2021;9(2):1057–69. <https://doi.org/10.1109/TETC.2019.2910086>
- [23] Reilly C, Stephen OS, Thorpe C. Robustness of image-based malware classification models trained with generative adversarial networks. In: Proceedings of the 2023 European interdisciplinary cybersecurity conference. EICC '23; New York, NY, USA: Association for Computing Machinery. ISBN 9781450398299; 2023, p. 92–9. <https://doi.org/10.1145/3590777.3590792>
- [24] Zelinka I, Szczypka M, Plucar J, Kuznetsov N. From malware samples to fractal images: a new paradigm for classification. Math Comput Simul 2024;218:174–203. <https://www.sciencedirect.com/science/article/pii/S0378475423004937>. <https://doi.org/10.1016/j.matcom.2023.11.032>
- [25] Daoudi N, Samhi J, Kabore AK, Allix K, Bissyandé TF, Klein J. Dexray: a simple, yet effective deep learning approach to android malware detection based on image representation of bytecode. In: Wang G, Ciptadi A, Ahmadzadeh A, editors. Deployable machine learning for security defense. Cham: Springer International Publishing. ISBN 978-3-030-87839-9; 2021, p. 81–106.
- [26] Xie C, Wang J, Zhang Z, Zhou Y, Xie L, Yuille A. Adversarial examples for semantic segmentation and object detection. In: 2017 IEEE International conference on computer vision (ICCV). 2017, p. 1378–87. <https://doi.org/10.1109/ICCV.2017.153>
- [27] Thys S, Ranst WV, Goedemé T. Fooling automated surveillance cameras: adversarial patches to attack person detection. In: 2019 IEEE/CVF Conference on computer vision and pattern recognition workshops (CVPRW). 2019, p. 49–55. <https://doi.org/10.1109/CVPRW.2019.00012>
- [28] Huang L, Gao C, Zhou Y, Xie C, Yuille AL, Zou C, et al. Universal physical camouflage attacks on object detectors. In: 2020 IEEE/CVF Conference on computer vision and pattern recognition (CVPR). 2020, p. 717–26. <https://doi.org/10.1109/CVPR42600.2020.00080>
- [29] Demetrio L, Coull SE, Biggio B, Lagorio G, Armando A, Roli F. Adversarial EXamples: a survey and experimental evaluation of practical attacks on machine learning for windows malware detection. ACM Trans Priv Secur 2021;24(4). <https://doi.org/10.1145/3473039>
- [30] Ren H, Huang T. Adversarial example attacks in the physical world. In: Machine learning for cyber security: third international conference, ML4CS 2020, Guangzhou, China, October 8–10, 2020, proceedings, Part II. Berlin, Heidelberg: Springer-Verlag. ISBN 978-3-030-62459-0; 2020, p. 572–82. https://doi.org/10.1007/978-3-030-62460-6_51
- [31] Gu S, Cheng S, Zhang W. From image to code: executable adversarial examples of android applications. In: Proceedings of the 2020 6th international conference on computing and artificial intelligence. ICCAI '20; New York, NY, USA: Association for Computing Machinery. ISBN 9781450377089; 2020, p. 261–8. <https://doi.org/10.1145/3404555.3404574>
- [32] Park D, Yener B. A survey on practical adversarial examples for malware classifiers. In: Reversing and offensive-oriented trends symposium. ROOTS'20; New York, NY, USA: Association for Computing Machinery. ISBN 9781450389747; 2021, p. 23–35. <https://doi.org/10.1145/3433667.3433670>
- [33] Kolosnjaji B, Demontis A, Biggio B, Maiorca D, Giacinto G, Eckert C, et al. Adversarial malware binaries: evading deep learning for malware detection in executables. In: 2018 26th European signal processing conference (EUSIPCO). 2018, p. 533–7. <https://doi.org/10.23919/EUSIPCO.2018.8553214>
- [34] Li D, Li Q, Ye YF, Xu S. Arms race in adversarial malware detection: a survey. ACM Comput Surv 2021;55(1). <https://doi.org/10.1145/3484491>
- [35] Allix K, Bissyandé TF, Klein J, Le Traon Y. Androzo: collecting millions of android apps for the research community. In: Proceedings of the 13th international conference on mining software repositories. MSR '16; New York, NY, USA: Association for Computing Machinery. ISBN 9781450341868; 2016, p. 468–71. <https://doi.org/10.1145/2901739.2903508>
- [36] Virustotal. <https://www.virustotal.com>
- [37] Mobile security framework (mobSF). <https://github.com/MobSF/Mobile-Security-Framework-MobSF>
- [38] Pan Y, Ge X, Fang C, Fan Y. A systematic literature review of android malware detection using static analysis. IEEE Access 2020;8:116363–79. <https://doi.org/10.1109/ACCESS.2020.3002842>
- [39] Soi D, Sanna A, Maiorca D, Giacinto G. Enhancing android malware detection explainability through function call graph APIs. J Inf Secur Appl 2024;80:103691. <https://www.sciencedirect.com/science/article/pii/S2214212623002752>. <https://doi.org/10.1016/j.jisa.2023.103691>
- [40] Dash SK, Suarez-Tangil G, Khan S, Tam K, Ahmadi M, Kinder J, et al. DroidScribe: classifying android malware based on runtime behavior. In: 2016 IEEE Security and privacy workshops (SPW). 2016, p. 252–61. <https://doi.org/10.1109/SPW.2016.25>
- [41] Martín A, Rodríguez-Fernández V, Camacho D. CANDYMAN: classifying android malware families by modelling dynamic traces with Markov chains. Eng Appl Artif Intell 2018;74:121–33. <https://doi.org/10.1016/j.engappai.2018.06.006>
- [42] Yuan Z, Lu Y, Xue Y. Droiddetector: android malware characterization and detection using deep learning. Tsinghua Sci Technol 2016;21(1):114–23.
- [43] Arshad S, Shah MA, Wahid A, Mehmood A, Song H, Yu H. SAMADroid: a novel 3-level hybrid malware detection model for android operating system. IEEE Access 2018;6:4321–39. <https://doi.org/10.1109/ACCESS.2018.2792941>
- [44] Malhotra V, Potika K, Stamp M. A comparison of graph neural networks for malware classification. J Comput Virol Hacking Tech 2024;20(1):53–69. <https://doi.org/10.1007/s11416-023-00493-y>
- [45] Khalid S, Hussain FB. Volmemdroid-investigating android malware insights with volatile memory artifacts. Expert Syst Appl 2024;253:124347. <https://www.sciencedirect.com/science/article/pii/S0957417424012132>. <https://doi.org/10.1016/j.eswa.2024.124347>

- [46] Nightbringer21. Fridump: a universal memory dumper using frida. <https://github.com/Nightbringer21/fridump>; 2019. GitHub repository, version 0.1.
- [47] 504ensicsLabs. LiME: linux memory extractor (Loadable kernel module for volatile memory acquisition). <https://github.com/504ensicsLabs/LiME>; 2012. GitHub repository.
- [48] Bellizzi J, Vella M, Colombo C, Hernandez-Castro J. Responding to living-off-the-land tactics using just-in-time memory forensics (JIT-MF) for android. 2021. arXiv: 2105.05510 ; <https://arxiv.org/abs/2105.05510>.
- [49] Bellizzi J, Vella M, Colombo C, Hernandez-Castro J. Real-time triggering of android memory dumps for stealthy attack investigation. In: Asplund M, Nadjim-Tehrani S, editors. Secure IT systems. Cham: Springer International Publishing. ISBN 978-3-030-70852-8; 2021, p. 20–36.
- [50] Bellizzi J, Vella M, Colombo C, Hernandez-Castro J. Responding to targeted stealthy attacks on android using timely-captured memory dumps. IEEE Access 2022;10:35172–218. <https://doi.org/10.1109/ACCESS.2022.3160531>
- [51] Zalewski ML. memfetch: utility to dump memory of a running process. 2007. Project page (memfetch.tgz) and FreeBSD port; accessed 2025-10-06; <https://lcamtuf.coredump.cx/soft/memfetch.tgz>.
- [52] Brune N. fmem: linux kernel module for volatile memory dumping. <https://github.com/NateBrune/fmem>; 2025. GitHub repository, accessed 2025-10-06.
- [53] Foundation V. Volatility: the memory forensics framework. <https://www.volatilityfoundation.org/>; Open source memory forensics tool.
- [54] Ali-Gombe A, Sudhakaran S, Case A, Richard III GG. DroidScraper: a tool for android in-memory object recovery and reconstruction. In: 22nd international symposium on research in attacks, intrusions and defenses (RAID 2019). Chaoyang District, Beijing: USENIX Association. ISBN 978-1-939133-07-6; 2019, p. 547–59. <https://www.usenix.org/conference/raid2019/presentation/ali-gombe>.
- [55] Bhatia R, Saltaformaggio B, Yang SJ, Ali-Gombe A, Zhang X, Xu D, et al. "Tipped off by your memory allocator": device-wide user activity sequencing from android memory images (Timeliner). In: Proceedings of the NDSS symposium. 2018. https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_04B-4_Bhatia_paper.pdf. <https://doi.org/10.14722/ndss.2018.23324>
- [56] Bellizzi J, Losiouk E, Conti M, Colombo C, Vella M. Vedrando: a novel way to reveal stealthy attack steps on android through memory forensics. J Cybersec Priv 2023;3(3):364–95. <https://www.mdpi.com/2624-800X/3/3/19>. <https://doi.org/10.3390/jcp3030019>
- [57] Litjens G, Kooi T, Beijndorf BE, Setio AA, Ciompi F, Ghafoorian M, et al. A survey on deep learning in medical image analysis. Med Image Anal 2017;42:60–88. <https://doi.org/10.1016/j.media.2017.07.005>
- [58] Esteve A, Robicquet A, Ramsundar B, Kuleshov V, DePristo M, Chou K, et al. A guide to deep learning in healthcare. Nat Med 2019;25:24–9. <https://doi.org/10.1038/s41591-018-0316-z>
- [59] Geiger A, Lenz P, Urtasun R. Are we ready for autonomous driving? The KITTI vision benchmark suite. Conf Comput Vis Pattern Recognit 2013;3354–61. <https://doi.org/10.1109/CVPR.2013.544>
- [60] Zou Z, Shi T, Guo Y, Ye J. Object detection in 20 years: a survey. IEEE Trans Pattern Anal Mach Intell 2019;41(11):2571–95. <https://doi.org/10.1109/TPAMI.2018.2876849>
- [61] Redmon J, Divvala S, Girshick R, Farhadi A. You only look once: unified, real-time object detection. IEEE Conf Comput Vis Pattern Recognit 2016;779–88. <https://doi.org/10.1109/CVPR.2016.91>
- [62] Çarkacı N. Binary-to-image: convert executable binary files into RGB or grayscale images. <https://github.com/ncarkaci/binary-to-image>; 2025. GitHub repository, accessed 2025-10-06.
- [63] Freitas S, Duggal R, et al. Malnet-image: a large-scale image database of malicious software. <https://github.com/safreital/malnet-image>; 2022. GitHub repository for binary-to-image code, accessed 2025-10-06.
- [64] Bhadula T. Malware-image-detection: convert binary malware files to grayscale images and classify with CNN. <https://github.com/TanayBhadula/malware-image-detection>; 2025. GitHub repository, accessed 2025-10-06.
- [65] Homem I, Papapetrou P. Android process memory string dumps dataset 2017; https://su.figshare.com/articles/dataset/Android_Process_Memory_String_Dumps_Dataset/4989773. <https://doi.org/10.17045/sthlmuni.4989773.v1>
- [66] Coull SE, Gardner C. Activation analysis of a byte-based deep neural network for malware classification. In: 2019 IEEE Security and privacy workshops (SPW). 2019, p. 21–7. <https://doi.org/10.1109/SPW.2019.00017>
- [67] Makkawy SJ, De Lucia MJ, Barner KE. Malvis: large-scale bytecode visualization framework for explainable android malware detection. J Cybersec Priv 2025;5(4). <https://www.mdpi.com/2624-800X/5/4/109>. <https://doi.org/10.3390/jcp5040109>
- [68] Youden WJ. Index for rating diagnostic tests. Cancer 1950;3(1):32–5. [https://doi.org/10.1002/1097-0142\(1950\)3:1<32::AID-CNCR2820030106>3.0.CO;2-3](https://doi.org/10.1002/1097-0142(1950)3:1<32::AID-CNCR2820030106>3.0.CO;2-3)
- [69] Chen CH, Pau L-F, Wang P SP, eds. Handbook of pattern recognition and computer vision. Singapore: World Scientific Publishing; 2 ed.; 1998.
- [70] Swain MJ, Ballard DH. Color indexing. Int J Comput Vis 1991;7(1):11–32. <https://doi.org/10.1007/BF00130487>
- [71] Stricker M, Orengo M. Similarity of color images. In: Storage and retrieval for image and video databases III; vol. 2420 of *Proceedings of SPIE*. 1995, p. 381–92. <https://doi.org/10.1117/12.205308>
- [72] Haralick RM, Shanmugam K, Dinstein I. Textural features for image classification. IEEE Trans Syst Man Cybern 1973;SMC-3(6):610–21. <https://doi.org/10.1109/TSMC.1973.4309314>
- [73] Ojala T, Pietikäinen M, Mäenpää T. Multiresolution gray-scale and rotation invariant texture classification with local binary patterns. IEEE Trans Pattern Anal Mach Intell 2002;24(7):971–87. <https://doi.org/10.1109/TPAMI.2002.1017623>
- [74] Dalal N, Triggs B. Histograms of oriented gradients for human detection. In: Proceedings of the IEEE computer society conference on computer vision and pattern recognition (CVPR); vol. 1. 2005, p. 886–93. <https://doi.org/10.1109/CVPR.2005.177>
- [75] Sobel I, Feldman G. A 3x3 isotropic gradient operator for image processing. Presented at the Stanford Artificial Intelligence Project (SAIL); 1968. Commonly cited for the Sobel operator.
- [76] Hu M-K. Visual pattern recognition by moment invariants. IRE Trans Inf Theory 1962;8(2):179–87. <https://doi.org/10.1109/TIT.1962.1057692>
- [77] Gonzalez RC, Woods RE. Digital image processing. Pearson; 4 ed.; 2018.
- [78] Starck J-L, Candès EJ, Donoho DL. The curvelet transform for image denoising. IEEE Trans Image Process 2002;11(6):670–84. Cited by: 2219; <https://www.scopus.com/inward/record.uri?eid=2-s2.0-0036612499&doi=10.1109/2ftIP.2002.1014998&partnerID=40&md5=59f58254aaba67b60424b495328f774>. <https://doi.org/10.1109/TIP.2002.1014998>
- [79] Redmon J, Divvala S, Girshick R, Farhadi A. You only look once: unified, real-time object detection. vol. 2016. 2016, p. 779–88. Cited by: 47592; All Open Access, Green Open Access; <https://www.scopus.com/inward/record.uri?eid=2-s2.0-84986274465&doi=10.1109/2ftCVPR.2016.91&partnerID=40&md5=79a23b9cc271b6f314eea447fb88cc7d>. <https://doi.org/10.1109/CVPR.2016.91>
- [80] Lecun Y, Bengio Y, Hinton G. Deep learning. Nature 2015;521(7553):436–44. Cited by: 75015; All Open Access, Green Open Access; <https://www.scopus.com/inward/record.uri?eid=2-s2.0-84930630277&doi=10.1038/2fnature14539&partnerID=40&md5=e324cb9ec992f892ebc74f3e06078083>. <https://doi.org/10.1038/nature14539>
- [81] He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. vol. 2016. 2016, p. 770–8. Cited by: 211538; All Open Access, Green Open Access; <https://www.scopus.com/inward/record.uri?eid=2-s2.0-84986274465&doi=10.1109/2ftCVPR.2016.90&partnerID=40&md5=f67e8d2a623bac88ad535d2c0a6d374>. <https://doi.org/10.1109/CVPR.2016.90>
- [82] Wang Y, Yao Q, Kwok JT, Ni LM. Generalizing from a few examples: a survey on few-shot learning. ACM Comput Surv 2020;53(3). <https://doi.org/10.1145/3386252>
- [83] Snell J, Swersky K, Zemel R. Prototypical networks for few-shot learning. In: Proceedings of the 31st international conference on neural information processing systems. NIPS'17; Red Hook, NY, USA: Curran Associates Inc. ISBN 9781510860964; 2017, p. 4080–90.
- [84] Kalgutkar V, Stakhonova N, Cook P, Matyukhina A. Android authorship attribution through string analysis. In: Proceedings of the 13th international conference on availability, reliability and security. ARES '18; New York, NY, USA: Association for Computing Machinery. ISBN 9781450364485; 2018. <https://doi.org/10.1145/3230833.3230849>
- [85] Mimura M, Ito R. Applying NLP techniques to malware detection in a practical environment. Int J Inf Secur 2022;21(2):279–91. <https://doi.org/10.1007/s10207-021-00553-8>
- [86] Selvaraju RR, Cogswell M, Das A, Vedantam R, Parikh D, Batra D. Grad-CAM: visual explanations from deep networks via gradient-based localization. In: Proceedings of the IEEE international conference on computer vision (ICCV). 2017, p. 618–26. <https://doi.org/10.1109/ICCV.2017.74>
- [87] Kinkead M, Millar S, McLaughlin N, O'Kane P. Towards explainable CNNs for android malware detection. Procedia Comput Sci 2021;184:959–65. The 12th International Conference on Ambient Systems, Networks and Technologies (ANT) / The 4th International Conference on Emerging Data and Industry 4.0 (EDI40) / Affiliated Workshops; <https://www.sciencedirect.com/science/article/pii/S1877050921007663>. <https://doi.org/10.1016/j.procs.2021.03.118>
- [88] Li Y, Yang Z, Guo Y, Chen X. DroidBot: a lightweight UI-guided test input generator for android. In: Proceedings of the 39th international conference on software engineering companion. ICSE-C '17; IEEE Press. ISBN 9781538615898; 2017, p. 23–6. <https://doi.org/10.1109/ICSE-C.2017.8>
- [89] Deng J, Dong W, Socher R, Li L-J, Li K, Fei-Fei L. ImageNet: a large-scale hierarchical image database. In: 2009 IEEE conference on computer vision and pattern recognition. 2009, p. 248–55. <https://doi.org/10.1109/CVPR.2009.5206848>
- [90] He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR). 2016, p. 770–8.
- [91] Tan M, Le QV. EfficientNet: rethinking model scaling for convolutional neural networks. In: Chaudhuri K, Salakhutdinov R, editors. Proceedings of the 36th international conference on machine learning, ICML 2019, 9–15 june 2019, long beach, california, USA. Proceedings of Machine Learning Research; PMLR; 2019, p. 6105–14. <http://proceedings.mlr.press/v97/tan19a.html>
- [92] Howard AG, Zhu M, Chen B, Kalenichenko D, Wang W, Weyand T, et al. MobileNets: efficient convolutional neural networks for mobile vision applications. CoRR 2017; <https://arxiv.org/abs/1704.04861>.
- [93] Woo S, Debnath S, Hu R, Chen X, Liu Z, Kweon IS, et al. ConvNeXt V2: co-designing and scaling convnets with masked autoencoders. In: 2023 IEEE/CVF Conference on computer vision and pattern recognition (CVPR). 2023, p. 16133–42. <https://doi.org/10.1109/CVPR52729.2023.01548>
- [94] Dosovitskiy A, Beyer L, Kolesnikov A, Weissenborn D, Zhai X, Unterthiner T, et al. An image is worth 16x16 words: transformers for image recognition at scale. In: 9th International conference on learning representations, ICLR 2021, virtual event, Austria, May 3–7, 2021. OpenReview.net; 2021. <https://openreview.net/forum?id=YicbFdNTTy>.
- [95] Szegedy C, Liu W, Jia Y, Sermanet P, Reed SE, Anguelov D, et al. Going deeper with convolutions. In: Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR). 2015, p. 1–9. <https://doi.org/10.1109/CVPR.2015.7298594>

- [96] Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition. In: Bengio Y, LeCun Y, editors. 3rd International conference on learning representations, ICLR 2015, San Diego, CA, USA, May 7–9, 2015, conference track proceedings. 2015, <http://arxiv.org/abs/1409.1556>.
- [97] Mekruksavanich S, Jantawong P, Hnoohom N, Jitpattanakul A. Resnet-based network for recognizing daily and transitional activities based on smartphone sensors. In: 2022 3rd International conference on big data analytics and practices (IBDAP). 2022, p. 27–30. <https://doi.org/10.1109/IBDAP55587.2022.9907111>



Silvia Lucia Sanna is Post-Doc at the University of Cagliari, she received her PhD in Artificial Intelligence for Security and Cybersecurity in January 2026 with a thesis "Artificial Intelligence for Android Stealth-Attack Detection: a Digital Forensics Approach". Her research focuses on AI-driven vulnerability and threat detection on Android using digital forensics features. From her experience as digital forensics consultant, she investigates also new frontiers in Digital Forensics analysis, focusing on mobile, memory, gen-AI data, integrating AI in digital forensics and anti-forensics.



Leonardo Regano received an MSc degree in 2015 and a PhD in Computer Engineering in 2019 from Politecnico di Torino, where he worked as a research assistant for eight years. He is currently an Assistant Professor at the Department of Electrical and Electronic Engineering, University of Cagliari (Italy). His current research interests focus on software security, artificial intelligence and machine learning applications to cybersecurity, security policies analysis, and software protection techniques assessment.



Davide Maiorca received from the University of Cagliari (Italy) the MSc degree (Hons.) in Electronic Engineering in 2012, and the PhD in Computer and Electronic Engineering in 2016. He is currently a Senior Assistant Professor at the Department of Electrical and Electronic Engineering, University of Cagliari. His research fields include analyzing and detecting X86 and Android malware, malicious documents and multimedia applications (e.g., PDF, Microsoft Office), and Adversarial Machine Learning. Dr. Maiorca authored more than 25 research papers and has served as a Program Committee member and reviewer for international conferences and journals.



Giorgio Giacinto is a Professor of Computer Engineering at the University of Cagliari, Italy. His research activities are carried out within the sAlfer Lab research group, where he leads the Cybersecurity research area. His main contributions lie in machine learning approaches to cybersecurity, specifically regarding threat analysis and detection, and are supported by funding from national and international projects. He has published more than 200 papers in international conferences and journals and regularly serves as a member of the Editorial Board and Program Committee for several international journals and conferences. He is a member of the managing committee of the Cybersecurity National Lab and the Artificial Intelligence & Intelligent Systems Lab, both within the CINI consortium in Italy. He is also a member of the expert board for the Italian delegation in the Program Committee of Horizon Europe, Cluster III on "Civil Security for Society." He is a Fellow of the IAPR (International Association for Pattern Recognition) and a Senior Member of the IEEE Computer Society and ACM.