Android Malware Detection Using Data Mining Techniques on Process Control Block Information

by

Heba Ziad Alawneh

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
August 8, 2020

Keywords: Dynamic Malware Detection, Android smartphones, Recurrent Neural Networks, Convolutional Neural Networks, Long short-term memory, Deep Learning

Committee Members

David Umphress, Chair, Professor of Computer Science and Software Engineering
Anh Nguyen, Assistant Professor of Computer Science and Software Engineering
Daniel Tauritz, Associate Professor of Computer Science and Software Engineering
Anthony Skjellum, Professor of Computer Science and Engineering, University of Tennessee at Chattanooga

Abstract

Because smartphones are increasingly becoming the mobile computing device of choice, we are experiencing an increase in the number and sophistication of mobile-computing-based malware attacks. A lot of these attacks target users' sensitive information, such as banking usernames, and passwords. A widespread type of malicious app encrypts user data locking their devices with passwords and asking money to decrypt it. Moreover, they can illegitimately collect browsing-related information or install other apps.

Available malware detection techniques can be categorized as dynamic or static based on the type of features used in the analysis. Using process behavior (as in dynamic analysis) to detect malware is generally more reliable than examining application files only (as in static analysis). Nonetheless, dynamic analysis is more time and computationally intensive. Hence, real-time malware detection is considered a challenging task. The limitations of mobile devices, such as storage, computing capacity, and battery life, make the task even more challenging.

In this research, we propose a dynamic malware detection approach that identifies malicious behavior using deep learning techniques on Process Control Block (PCB) information mined over the process execution time. Our mining approach is performed at the kernel level and synchronized with the process CPU utilization. It precisely tracks changes in PCB parameters over the execution time. It does not only represent the process behavior efficiently but also all threads created by that process.

We then use the PCB sequence information to train a deep learning model to identify malicious behavior. We validated our approach using 2600 benign and 2500 malware-infested recent Android applications. Our mining approach successfully captured more than 99% of context switches for the vast majority of tested applications. Furthermore, our detection model was able to identify malicious behavior at various points of the process execution time using 12 PCBs only with an F1-score of 95.8%. To the best of our knowledge, no available dynamic malware detection technique has achieved such minimal detection time. We also introduce a

closed dynamic malware analysis framework for application testing running on multiple Android phones concurrently.

Acknowledgments

First and foremost, I would like to thank God Almighty for giving me the strength, dedication, and ability to finish this work. I am also deeply thankful for those who supported me professionally and personally throughout my journey here at Auburn University. I am grateful, especially to my mentor, and advisor, Dr. David Umphress. Thank you for your effort and support. Mostly, thank you for encouraging me to freely pursue my research interests while guiding me to stay on course.

I wish to express my sincere thanks to my dissertation committee members. Dr. Anthony Skjellum, for his support to start my Ph.D. studies, Dr. Anh Nguyen, and Dr. Daniel Tauritz. Thank you all for your guidance and valuable input.

Finally, I would like to thank my companion Hamza Alkofahi, for the endless support and always pushing me to achieve my best, my kids Qais and Omar for keeping me happy, my parents Helwa and Ziad Alawneh for believing in me, my brothers and sisters, and to all my friends and family. Thank you for your prayers, love, and support. Without you, my journey would not have been possible.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today, smartphone users number more than 5.1 billion [28], with smartphone penetration rates increasing as well. Many smartphone users are unaware of the fact that most of the malicious smartphone software targets users' private information without their permission. The majority of commercial anti-virus tools follow a signature-based malware detection approach (i.e., they look for specific fingerprints of already discovered malware), and are thus unable to detect new malware (also known as zero-day malware attacks)[104].

According to StatCounter [57], Android is the most popular operating system worldwide. It's also the second most targeted platform after Microsoft Windows. However, only a few Android mobile devices provide effective virus protection, while the majority remaining are poorly protected. Moreover, the number and sophistication of new malware attacks grew significantly during the past two years [1], making it harder with time to detect such attacks.

Over 90% of malware targeting Android platforms in 2018 were Trojan attacks [18]. A Trojan attack is a harmful app that appears as legitimate; however, it performs malicious activity unbeknownst to the user [43]. Trojans can be used for stealing confidential information, creating backdoors, and activating viruses or other malware. Banking and password Trojan attacks tripled within the first quarter of 2018 alone [18]. The latest state-of-the-art banking Trojan named Gustuff can steal the login information of over 100 banking apps and 30 cryptocurrency apps. Moreover, it launches other malware to transfer money from the stolen bank accounts to preconfigured accounts.

Traditional detection techniques such as static and signature-based detection suffer from concealment schemes (obfuscation and encryption), which limit or eliminate their effectiveness. Therefore, using dynamic detection techniques can be the best approach for real-time malware detection. Available malware detection techniques analyze malware either using system activities and hardware utilization or by tracing the information flow through the system. Recent research efforts proposed mining the Process Control Block (PCB) information to detect malicious behavior in Android applications [70, 88, 8]. However, available dynamic techniques are generally time intensive and resource-consuming which makes them unsuitable for detecting malware at run time.

The process control block in an operating system is the data structure used to store all the information that the kernel needs about a specific running process at any given time. Benign processes follow the system's access control policies that are used to specify who can access information, where and when. In contrast, malicious processes violate them by exploiting a system vulnerability (i.e., unintended flaw) to steal information, spy, or attack the infected device. Therefore, some pieces of the information stored in the PCB are likely to reflect the malicious behavior at occurrence.

In this work, we consider the challenges of detecting malware dynamically on smartphone systems, as well as the limitation of available techniques and make the following contributions. In Chapter 3, we propose a novel approach to mining PCB information at the kernel level synchronized with the process CPU utilization. The proposed approach precisely tracks changes in the PCB parameters over the process execution time. It does not only represent the process behavior efficiently but also all threads created by that process, which enables detecting dropper malware. We also introduce a closed dynamic malware analysis framework for application testing running on multiple Android phones concurrently. We validated the approach by mining the PCB information collected from 2615 benign and 2502 malware-infested recent Android applications, spanning a wide range of application categories and malware families.

In Chapter 4, we propose a novel approach for detecting Android malware using deep learning techniques on PCB information. Our detection model combines Recurrent Neural Networks (RNNs) with Convolutional Neural Networks (CNNs) and Deep Neural Networks

(DNNs) into one model to identify the malicious behavior of a process, given a sequence of 12 PCB records collected during its runtime. To the best of our knowledge, no available malware detection technique has achieved such minimal detection time. Using the PCB sequence information of 2615 benign and 2502 malware-infested applications, we show that our approach was able to identify malicious behavior at various points of the process execution time with a high F1-score.

Chapter 2

Literature Review

2.1 Malware Targeting Android Smartphones

Google's Android has dominated the global smartphone market for several years now [57]. As a consequence, it has become the main target of cybercriminals after Microsoft Windows [18]. Android platforms can be considered insecure for many reasons. The first reason is the widespread popularity of Android devices, which makes it more beneficial to attack than other platforms. Another reason is that Android is an open-source platform [13], which means anyone can see the source code, modify it, develop and distribute compatible apps for it. Moreover, although Google monthly releases security updates for the Android platforms, it takes some time before the manufacturers of the Android devices update them [18].

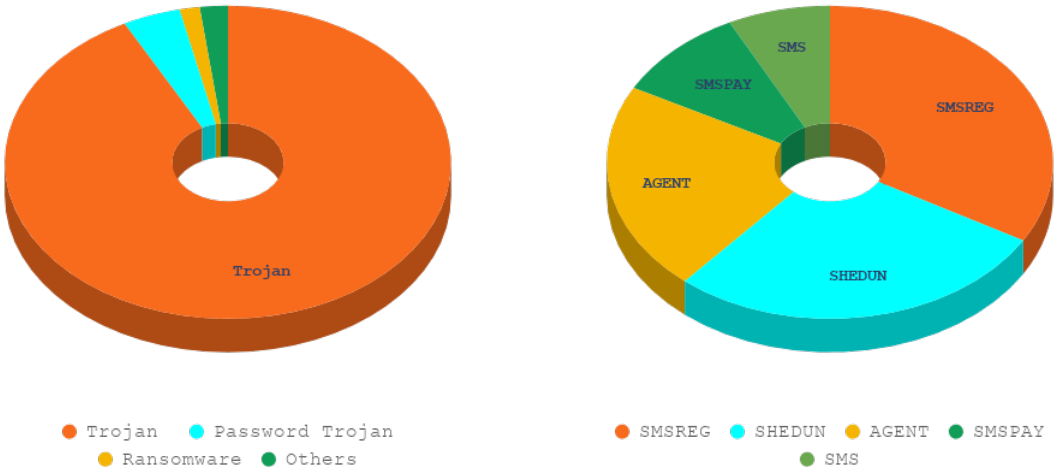
There are a variety of malware attacks targeting Android, including *backdoors, ransomware, spyware, worms, botnets, and trojans* [108, 24, 15, 55, 14]. A *backdoor* [61] is a malicious program used by the attacker to gain unauthorized remote access to the infected device by exploiting specific software vulnerabilities. Examples of Android backdoors are the *Gingerbreak*, the *DroidKungFu*, and the *BaseBridge* [32, 82, 108].

Spyware [17] is malware that appears to be legitimate but monitors the user activities and steals sensitive information such as bank details, passwords, and much more. In 2009, *Mobile Spy* was announced as the first professional spyware for Android phones [24]. This monitoring app runs in the background without a visible icon to the user tracking the users' SMS, GPS locations, photos, and more. In 2017, new spyware apps were detected that are used to analyze user habits, history, and locations for the customization of pop-up advertising [18].

Ransomware [79] is a category of malware that blocks the user from accessing their data by locking the device until a ransom payment is made. In 2014, a significant ransomware attack known as *ScarePackage* infected nearly 900 thousand Android phones within 30 days only [15]. In 2017, the *Lockscreen* [59] ransomware represented nearly 8% of the overall Android malware [18]. This ransomware locks out Android phones with a pop-over browser window that quickly reappears when closing it. To unlock the device, the user has to make a payment in vouchers or cryptocurrency (e.g., Bitcoin) only, as such transactions cannot be traced back to the malware developer. Although the lockscreen malware doesn't encrypt any data in the phone, removing it requires a factory reset that also removes all installed apps, files, and settings in the device.

Worms and *Botnets* [2, 33] are large-scale network attacks targeting Android phones. Worm [2] apps can replicate themselves and spread through shared website links, email attachments, and peer-to-peer file-sharing networks (i.e., a network of connected devices without a central server). Sending SMS messages is another way to spread worms, such as the Chinese worm that infected more than 500K Android devices in 2014 [55]. A *botnet* [33] is a network of compromised Android devices controlled by a remote server called *Bot-master*. *Botnets* can be used to perform *Distributed Denial of Service attacks (DDoS)* [29], such as the *WireX* botnet [71], which was hidden in 300 Google Play Store apps and infected more than 100k Android devices. The vast majority of the Android malware incidents in 2018 were *trojan* attacks [18], as shown in Figure 2.1a. A *trojan* is a harmful app that looks benign, but once activated, can achieve any number of attacks on the host, such as stealing confidential information, creating backdoors, and activating viruses or other malware.

Figure 2.1b shows the top five Android malware in 2018 [18]. Shedun, Agent, SMSPay, and SMS are all trojan malware. SMSReg [65] is a Riskware app that pretends to be a battery improvement app but collects other information unbeknownst to the user. Agent [14] is a malicious app, that once downloaded, runs in the background silently waiting for commands from a Command and Control (C&C) server. Typically, it is given a filename of a legitimate app, which makes it hard to identify. This trojan can use the infected device as part of a DDoS



(a) (b)

Figure 2.1: (a) Android Malware distribution in 2018 (b) Top 10 Android malware in 2018.

attack against some target. It can also steal the users' private information and send them to the remote server.

Shedun [84] is a family of trojanized adware used to install secondary apps for advertisement purposes. This trojan poses as a legitimate app to trick the user into giving it root privileges. Such privileges allow it to install additional apps without the users' permission, further increasing the author's ad revenue.

The Android SMS and SMSPay variants [74, 63, 64] are typically repacked or trojanized apps that are legitimate but have been recompiled and distributed with additional malicious modules. SMSPay adds the functionality of silently sending premium-rate or spam SMS messages, which may add extra charges on the phone bill.

2.2 Malware Detection Techniques

Methods proposed to detect Android malware [42, 93, 98, 31, 72, 109, 103, 70] can be categorized either as signature or non-signature based. A malware signature is created by extracting patterns from a sample (e.g., binary patterns, and MD5/SHA1 hashes) [78]. Signature-based methods detect malware by examining the application files while looking for signatures of

known malware [42, 30, 105]. Such methods can only identify existing malware and fail to identify zero-day malware attacks.

A zero-day exploit [104] uses a heretofore undiscovered software vulnerability (a security weakness) to perform various types of attacks before a fix is released. The extent of the effects of the attack depends on the nature of the vulnerability, but can be as significant as a complete takeover of the device resulting in exfiltration of confidential information such as banking details, usernames, and passwords, or to install other malware.

To overcome the problem of zero-day malware attacks against Android platforms, a number of non-signature based methods have been proposed [93, 98, 68, 31]. These methods can be categorized as either static or dynamic. Static techniques treat applications as artifacts that can be examined for inconsistencies, such as analyzing the accompanying manifest for unusual permissions [93, 45, 89] or analyzing the application's bytecode for instructions, that when executed, could result in an undesired result [5]. Because these techniques examine application files without actually running them, they are vulnerable to obfuscation techniques and fail to detect malicious runtime loaded code [33].

Dynamic detection methods work by observing program execution carried out in a controlled environment [98], by monitoring system activities and hardware utilization [68], and by tracking information flow through the system [31]. Dynamic detection methods have some significant limitations, such as evasive malware [78], that detect being run on emulated environments and modify its behavior accordingly, and limited code coverage (i.e., the ratio of the application code executed during the analysis) [34]. Dynamic analysis can only reveal malicious behavior once the corresponding code has executed. Apps that trigger malicious behavior under certain conditions only (e.g., a specific date or command)[40] can appear benign if the condition is not met. Other challenges facing dynamic mobile malware detection systems are the additional computation overhead required to monitor execution and the possibility of delays in detecting issues [17].

A number of machine learning methods have been proposed to detect Android malware [94, 72, 109, 103, 101, 70, 96, 88]. Wu et al., Narudin et al., and Sheen et al. [94, 72, 56] have used standard classification algorithms with static-based malware detection methods [94,

72] and dynamic methods [56]. Wu et al. [94] used dataflow application program interfaces (APIs) to train the classifiers, while Sheen et al. [72] combined API call-based features with permission-based features. Narudin et al. [56], on the other hand, has used anomaly-based detection utilizing dynamic network traffic. Although these models have high detection accuracy, their complexity and resource consumption is high as well [75, 99].

Yousefi-Azar at al. [102] hashed the Dalvik bytecode to extract the static features and feed them to a neural network model to detect malware. Other research efforts used deep learning to detect malware [109, 103, 41]. [109] employs static dataflow analysis to detect Android malware. Dali Zhu et al. [41] statistically extract the API method calls from the Android .dex file, map them using embedding models, and feed the output to a CNN network. Although the approach in [103] combines static and dynamic analysis, it is based on binary features only. Moreover, the majority of these methods were constructed based on outdated malware samples. Additionally, collected malware datasets may contain a considerable number of duplicate malware samples under unique file hashes as a minimal modification to the Android Package (APK) file can produce a unique file hash. Including a large number of duplicate samples results in biased-datasets and therefore overfitted models.

A novel dynamic detection technique using *genetic footprint* has been proposed [69, 70] to detect malware by mining the information in the process control block (PCB), a data structure maintained by the kernel to represent running processes. Their hypothesis is that the state diagram of malicious processes must be different from those of benign processes because of the difference in activities. Benign processes follow system access control policies that specify the conditions under which information can be accessed, whereas malicious processes violate policies through stealing information, spying, or attacking the infected device by exploiting unintended flaws in the system.

TstructDroid in [70] trains a decision-tree-based classifier on a dataset of 110 malware instances and 110 benign Android apps using a subset (*genetic footprint*) of 32 out of 99 PCB features. The method requires a process execution time of 3s, in addition to the overhead of encoding the PCB features using *Discrete Cosine Transforms* while extracting their statistical features. Similar research [96, 88] presents a Spark-based malware detection framework, which

requires 15 seconds of process execution time prior to the detection process, resulting in a high detection delay and large computation overhead. Our previous work [8] presented a detection framework that uses a similar PCB mining approach with an improved detection speed of 100 ms using Back Propagation Neural Network (BPNN). These PCB-based methods trigger the PCB collection in the user-space, which means they can be delayed or interrupted more frequently leading to a higher chance of missing PCB records. Our proposed mining approach can overcome these limitations by mining the PCB information at the kernel level triggered by process context switching.

In short, static malware detection can provide low detection delay, but focuses on analyzing application code which can be obfuscated. Dynamic detection methods can overcome this limitation at the cost of more time, resource consumption, and limited code coverage. In this research, we present a dynamic malware detection method for Android platforms, that is capable of detecting a variety of malware attacks at various points of process run-time with a low detection delay and high detection performance.

2.3 Android PCB Features

The Linux kernel stores information of running processes in a circular, doubly linked list called the task list [73], where each element is a process descriptor of type `struct task_struct` (the PCB in Linux-based operating systems). The `task_struct` contains all the information that the kernel needs about a specific process, such as the process's priority, state, address space, pending signals, and much more.

In this research, a set of 120 numerical fields were extracted from the kernel PCB. The PCB contains many pieces of information associated with a specific process and its group. These parameters represent CPU scheduling, memory management, and signals information. Table 2.1 lists the *sched_info* struct cumulative counters of the process CPU scheduling. Table 2.2 lists the *mm_struct* parameters accessed by its pointer within the *task_struct*. These memory management pieces of information are used to trace memory usage, page faults, base and limit registers, memory resources demand, and their utilization.

Signals related information including the process and group resource utilization and time counters and the number of threads can be found in the *signal_struct* struct. The extracted parameters are listed in Table 2.3. The *task_struct* includes different types of information describing the exiting state of the process, its parents, its children, and the dead threads in their group. Table 2.4 lists the numerical features of this struct. Some of these features are used in conjunction with other parameters within other structs to manage process execution.

Table 2.1: CPU scheduling *sched_info* Features within *task_struct*.

sched_info Features			
pcount	run_delay	last_arrival	last_queued

Table 2.2: Memory management *mm_struct* Features within *task_struct*.

mm_struct Features			
tlb_flush_pending	hiwater_rss	stack_vm	start_stack
map_count	hiwater_vm	start_code	arg_start
vmacache_seqnum	total_vm	end_code	arg_end
mmap_base	locked_vm	start_data	env_start
mmap_legacy_base	pinned_vm	end_data	env_end
task_size	shared_vm	start_brk	saved_auxv[AT_VECTOR_SIZE]
highest_vm_end	exec_vm	brk	flags

Table 2.3: Signal information *signal_struct* Features within *task_struct*.

signal_struct Features			
utime	cnvcsw	cinblock	posix_timer_id
stime	cnivcsw	coublock	leader
cutime	min_flt	maxrss	oom_score_adj
cstime	maj_flt	cmaxrss	oom_score_adj_min
gtime	cmin_flt	nr_threads	flags
cgtime	cmaj_flt	group_exit_code	is_child_subreaper
nvcsw	inblock	notify_count	has_child_subreaper
nivcsw	oublock	group_stop_count	sum_sched_runtime

Table 2.4: Process information within *task_struct*.

task_struct Features			
nvcsw	utime	init_load_pct	exit_signal
nivcsw	stime	vmacache_seqnum	pdeath_signal
wakee_flip_decay_ts	utimescaled	parent_exec_id	nr_dirtied_pause
jobctl	stimescaled	self_exec_id	pagefault_disabled
atomic_flags	gtime	on_cpu	nr_dirtied
stack_canary	acct_timexpd	wake_cpu	cpuset_slab_spread_rotor
min_flt	flags	on_rq	cpuset_mem_spread_rotor
maj_flt	ptrace	prio	start_time
last_switch_count	wakee_flips	static_prio	real_start_time
sas_ss_sp	rt_priority	normal_prio	acct_rss_mem1
ptrace_message	btrace_seq	nr_cpus_allowed	acct_vm_mem1
dirty_paused_when	policy	rcu_read_lock_nesting	timer_slack_ns
trace	personality	exit_state	default_timer_slack_ns
trace_recursion	tgid	exit_code	state

Chapter 3

Mining PCB information & Building the Application Dataset for Android Malware Detection

3.1 Introduction

Previous research efforts have shown that mining the PCB information can be used to detect malicious behavior in Android applications [70, 96, 88, 8]. In this chapter, we propose a novel approach to mining PCB information at the kernel level triggered by process context switches. The proposed approach guarantees synchronizing the PCB mining with process CPU utilization, thus minimizing duplicate and missing PCB records. We also introduce a closed dynamic malware analysis framework for application testing running on multiple Android phones concurrently. We validated the approach by mining PCB information collected from 2615 benign and 2502 malware-infested Android applications, spanning a wide range of application categories and malware families. We did not rely on available malware datasets [16, 11, 90, 41], as they were static-based, outdated, unavailable by the time we started the research, or did not meet our sample selection criteria determined to produce an efficient and reliable sample set.

The rest of this chapter is organized as follows: Section 3.2 describes the criteria for building the Android (malware-infested and benign) application dataset and discusses its features. In Section 3.3, we describe the system general architecture including test devices, the input generation tool, and the testing environment setup. Section 3.4 describes in detail stages of the PCB mining process and components of the PCB data collector. Section 3.5 evaluates the collected PCB dataset, followed by discussion in Section 3.6.

3.2 Application Data Collection

To train a machine learning model to detect malicious behavior, we collected a dataset of 2615 benign and 2502 malware-infested applications. The benign applications were collected from top-ranked Google Play Store apps, while the malware-infested applications were collected from the VirusTotal malware dataset. In this section, we explain the process and selection criteria for collecting both benign and malware samples.

3.2.1 Benign Application Collection

Benign samples were downloaded from the Google Play Store using an open-source tool called the Raccoon [81], which enabled us to download the apks directly onto the PC. The apps were selected from the list of the most popular Google Play apps according to the open Android market data AndroidRank (the oldest service to provide information and statistics on Google Play store) [49]. The list contains more than 24.5k applications, from which 4020 apk apps were chosen based on the following criteria: (1) The application was not split into functionally duplicate apks, each configured to a specific device characteristic; (2) the application had at least a 4.0 out 5.0 rating; (3) it had been updated since January 2019; (4) it had reached at least a million installs; (5) it was compatible with Android versions 4.0 and later; (6) it did not rely on complex real-time interactive input, such as that used in a game. Such apps were also excluded due to a limitation of the used automated input generation tool [85], which lacks support to non-deterministic user interface events(such as random swipes or customized gestures).

3.2.2 Malware-infested Application Collection

Malware samples were collected and downloaded from the VirusTotal Private API [86] under an academic research access license. VirusTotal [37] is an online scanning service that analyzes files or URLs to detect viruses, trojans, and other types of malware. It aggregates the scanning results of more than 150 antivirus and malware scan engines [27], such as AhnLab Mobile Security [44], Avast Mobile Security Software [36], and Kaspersky Lab [3].

A total of 12060 Android malware samples were downloaded from the VirusTotal malware database. The collected samples were first seen within the time period of 2010 through 2019. Included in the malware samples were information on the malware category; MD5 and SHA256 hash values; the first time the malware was seen and scanned; the permissions requested by the malware-infested app; and the anti-malware engines that scanned the sample and their scanning results.

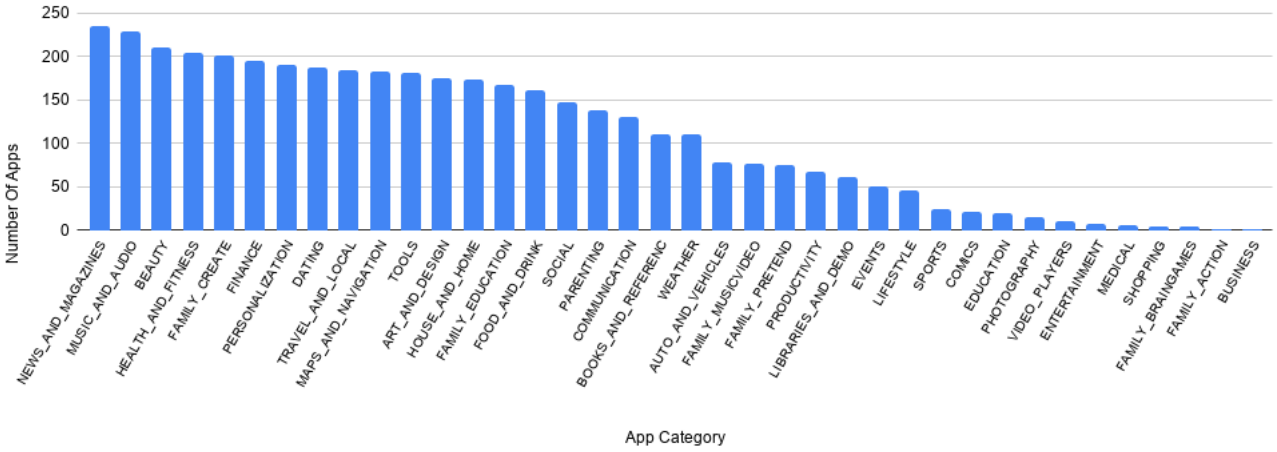
For the Android dataset, VirusTotal aggregates the results of 75 different malware detection engines. Each sample in the database has been scanned by an average of 61 engines. Because these engines vary in performance and reliability, which impacts the reliability of the collected dataset, we ranked them based on the number of samples they recognized as malware. Our top list of malware detection engines consists of 11 sources that scanned and successfully detected more than 80% of the collected malware samples. Six of these engines were listed as the best Android Anti-Malware engines according to the AV-Test mobile security products evaluation of 2019 [80].

To ensure the reliability of the dataset, we collected the malware samples based on the following criteria: (1) more than 40% of the malware detection engines that have scanned the sample classified it as malware, and (2) at least 20% of which are from the top engines.

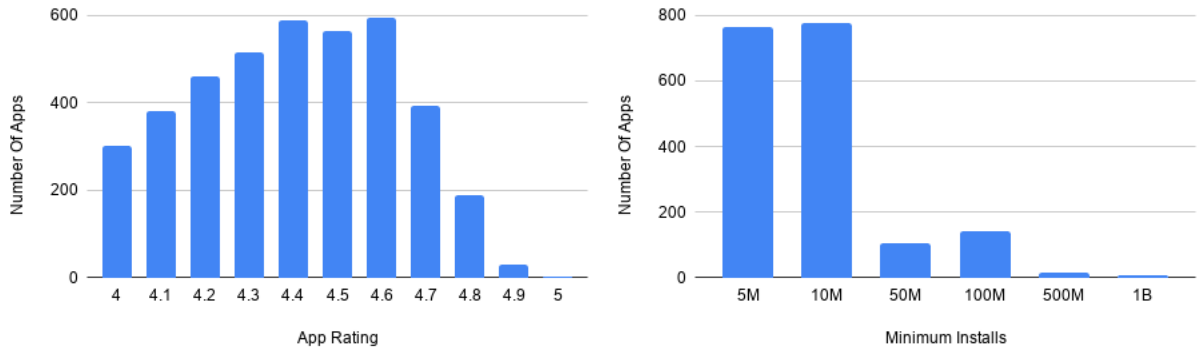
3.2.3 The Android Application Dataset

4020 benign applications spanning 38 categories were selected from the list of 24.5k most popular Android apps, as shown in Figure 3.1a. The dataset has an average rating of 4.4/5. The application distribution based on ranking and minimum number of installs is illustrated in Figures 3.1b and 3.1c.

6724 malware samples were selected from the 12060 samples collected. Upon further examination of the selected malware samples, we realized that a considerable number of these samples were different versions of the same set of apps. We only included one version per app to prevent creating bias toward a certain application behavior. Moreover, the selected samples included damaged apk files, as well as non-apk format apps (Dalvik format) which we also excluded from the dataset. The final malware dataset consists of 3562 unique malware-infested



(a)



(b) (c)

Figure 3.1: Summary statistics of the benign dataset: applications are distributed by (a) category, (b) application rating, and (c) minimum number of installs.

apps. Figure 3.2 illustrates how the majority of the collected dataset belongs to the set of newly developed malware.

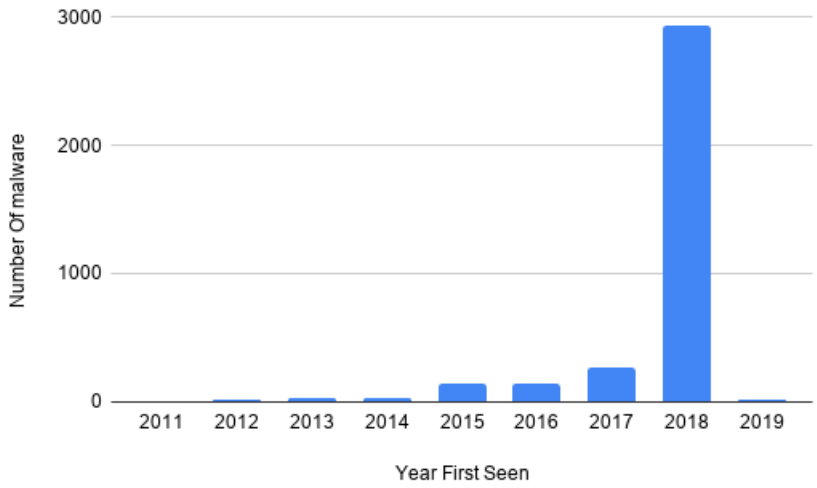


Figure 3.2: The distribution of the malware samples based on the year the malware sample was first seen

Selected malware samples are categorized as Trojan, Riskware, Spyware, Backdoor, Adware, Ransomware, and Exploit [18, 62, 17, 61, 7, 79] and were distributed among 195 malware families. Figure 3.3 presents the distribution of the malware families within the dataset, where the trojanized-Agent, the Adware-Shedun, and the Spyware-SMSSpy represent more than 50% of the collected samples [14, 84, 83]. According to the AV-Test security report, these malware families represented more than 42% of Android malware in 2018 alone.

3.3 Data Collection Architecture

3.3.1 Android Test Devices: Phone, Android, and Kernel Versions

All tests were performed using OnePlus 5t phones running Android version 8.1. A custom modified kernel (4.4.78) was utilized to enable module insertion and root the phone. The kernel was also modified to store PCB information triggered by context switching. When we modified the official Android kernel for OnePlus 5t phones, the Wifi stopped functioning. After further investigation, we found out that the wifi driver is not part of its kernel tree. Although we successfully built the wifi module with the available kernel configuration, we failed to load the

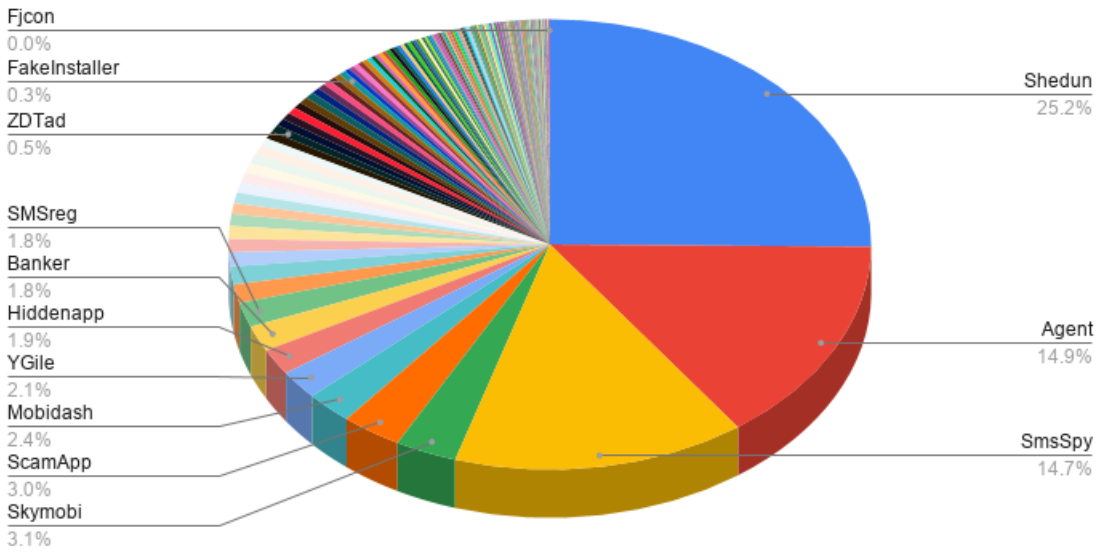


Figure 3.3: Distribution of malware families within the malware dataset

module into the kernel. Alternatively, we have used the LineageOS [48], which is an open-source operating system based on the Android platform. The LineageOS officially supports the OnePlus 5t phones, and their wifi module is part of its kernel tree.

3.3.2 Automated Input Generation for Android Application Testing

Goals of using mobile application testing include detecting bugs, vulnerabilities, and security problems. Manual testing requires significant human resources. Therefore, a considerable amount of effort has and still is being devoted to automating input generation for application testing. Test inputs for mobile applications can be represented by possible interactions with their graphical user interface (GUI), such as clicks, touches, or gestures. An input generation tool produces a sequence of such actions to mimic human interaction. The efficiency of these tools can be measured by code coverage (i.e., the ratio of application code executed during the analysis).

Available techniques can be categorized based on their exploration strategy into random, model-based, and systematic. *monkeyrunner* [85] is the most common tool for testing Android devices by generating a sequence of random interactions. Additionally, several research studies

have been proposed to optimize the random approach [51, 21, 107]. Model-based techniques extract a GUI interaction model for each application [9, 106, 25] or a more general model to exercise different Android applications [39, 23, 47, 22, 46]. Systematic testing uses various algorithms to trigger application behaviors that can only be revealed with specific test inputs, such as symbolic execution, data flow analysis, and evolutionary algorithms [10, 19, 52, 54, 53].

A number of empirical studies have been conducted to compare input generation tools. The authors in [26] used 68 benchmark apps to evaluate the main existing tools by the amount of code coverage achieved given the same time period. Their results show that *monkeyrunner* [85] achieved the best code coverage on average compared to the other tools [51, 10, 9, 39, 19]. It is also easy to use and has wide platform compatibility.

Another study [87] has compared the following tools [85, 107, 54, 47, 19, 77], using real-world industrial apps. They have collected 68 apps of different categories from the top-recommended Google Play Store apps (at that time). Based on their study, *monkeyrunner* accomplished the highest number of covered activities and methods on average. Other main tools were excluded from consideration due to unavailability, limited testing space, version incompatibility, or failure to function on real Android devices or industrial apps [25, 100, 52, 23].

More recent efforts have used machine learning techniques to simulate human behavior while interacting with Android applications (Humanoid)[46]. They used 68 open-source apps and 200 popular Google play store apps to evaluate a number of state-of-the-art test generation tools [85, 39, 77, 21, 47, 54]. Their result confirms that *monkeyrunner* outperforms other test generation tools except for Humanoid. However, the interactive speed of *monkeyrunner* is 10 times faster than Humanoid, and in our research, we try to expose the most application behavior in the least time frame possible. Hence, we have chosen *monkeyrunner* [85] as an input generation tool to automate our application testing as it is easy to use, compatible with all Android versions, outperformed most of the available generation tools, and has the highest interactive speed.

3.3.3 Network Configuration for Application Testing

Orchestrating the sample apps to obtain PCB information required that we also execute them in an environment that appeared as real as possible. The vast majority of the collected applications (both malware-infested and benign) required an internet connection to function properly. Hence, providing internet connection improves the efficiency and correctness of the dynamic analysis. However, providing real internet connection required previous knowledge of the malware behavior to avoid participating in online attacks, such as DDoS attacks [12]. The malware could also reproduce and spread to other devices on the same network, or even through SMS [92].

Monitoring and controlling network traffic for a large number of applications requires a significant amount of time and effort. Therefore, we have used the internet services simulation tool INetSim [91] as in [97, 60, 20] to trick the target apps into believing they are connected online. The INetSim tool simulates several internet services, such as HTTP/HTTPS, DNS, FTP, and IRC. It fabricates responses to the network requests made by the connected devices, spurring applications to expose more functionality.

To simulate both WiFi and an internet connection, we set up a closed network, as shown in Figure 3.4. The network consisted of an ASUS RT-AC66U router running DD-WRT, test Android devices, and a Raspberry Pi 3 running the internet simulation tool INetSim. The router was used as an internet gateway and had the AP-isolation setting enabled [6]. The AP-isolation prevented devices connected through the wireless network from communicating directly to each other, preventing malware from spreading through the network. The Android devices were connected to the network through the router. The router was connected to the Raspberry Pi through an Ethernet cable and was configured to use it as a DNS server.

3.4 Mining PCB Information

The first stage of the PCB data collection process was task management and preparation, where the main management unit assigned the tasks to the connected devices and managed them concurrently. The second stage was application testing and PCB data collection, in which the

Figure 3.4: The network setup for malware dynamic analysis

target application was exercised using *monkeyrunner*, and its PCB information was collected over the application interaction time. The last stage was storing, analysis, and feedback, where the resulting PCB sequence was stored and analyzed, and the termination state of the process was reported back to the main manager.

3.4.1 Tasks Management & Preparation

The main manager running on a Linux machine was responsible for assigning tasks to the connected Android devices. It used a separate thread (task manager) to concurrently manage all connected devices, as shown in Figure 3.5. The task manager was responsible for ensuring the device was on and ready for use prior to starting a task. It monitored the battery level of the testing device, assigning new tasks if the battery level was above 40%, otherwise it postponed the new task until the battery level was above 90%. The task manager also checked if the user issued a request to terminate or suspend the execution prior to starting a new task or while the phone was charging.

Figure 3.6 illustrates the sequence diagram of managing the tasks performed on each device. The task manager starts by extracting the package name (which uniquely identifies the app on the device) from the target apk file. It then installs the target apk and grants requested

permissions by the installed application. Upon installation completion, the manager inserts the data collection modules into the device to prepare for the application testing and PCB mining process. Then, it passes the package name to the application testing unit (ATU) to start the application testing and PCB mining process. The task manager keeps track of the termination state of each task-stage, reports task success or failure, and finally assigns a new task to the device.

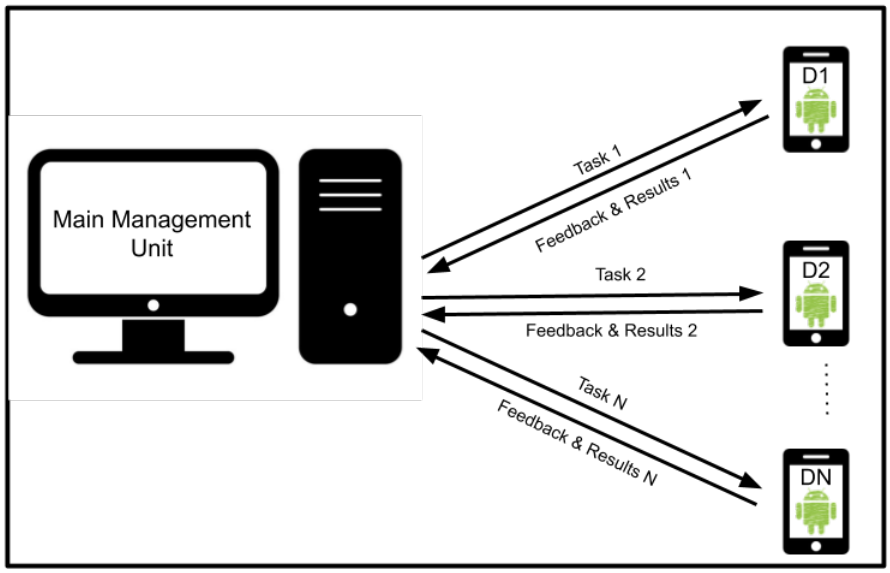


Figure 3.5: Main manager running tasks concurrently on all connected devices

3.4.2 Application Testing & PCB Data Collection

Application Testing & PCB Data Collection Components

The operation of application testing and PCB data collection was performed on the Android device. The system components responsible for the operation were the application testing unit (ATU), the PCB buffer reader, the data collector, and the modified CPU scheduler. The first two components operated in userspace, whereas the second two operated in kernel space.

The ATU was responsible for initiating the PCB buffer reader, sending the target package name (pname), and starting the application exerciser (*monkeyrunner*). After *monkeyrunner* finished execution, the ATU sent a null package name as a signal to the data collector to terminate

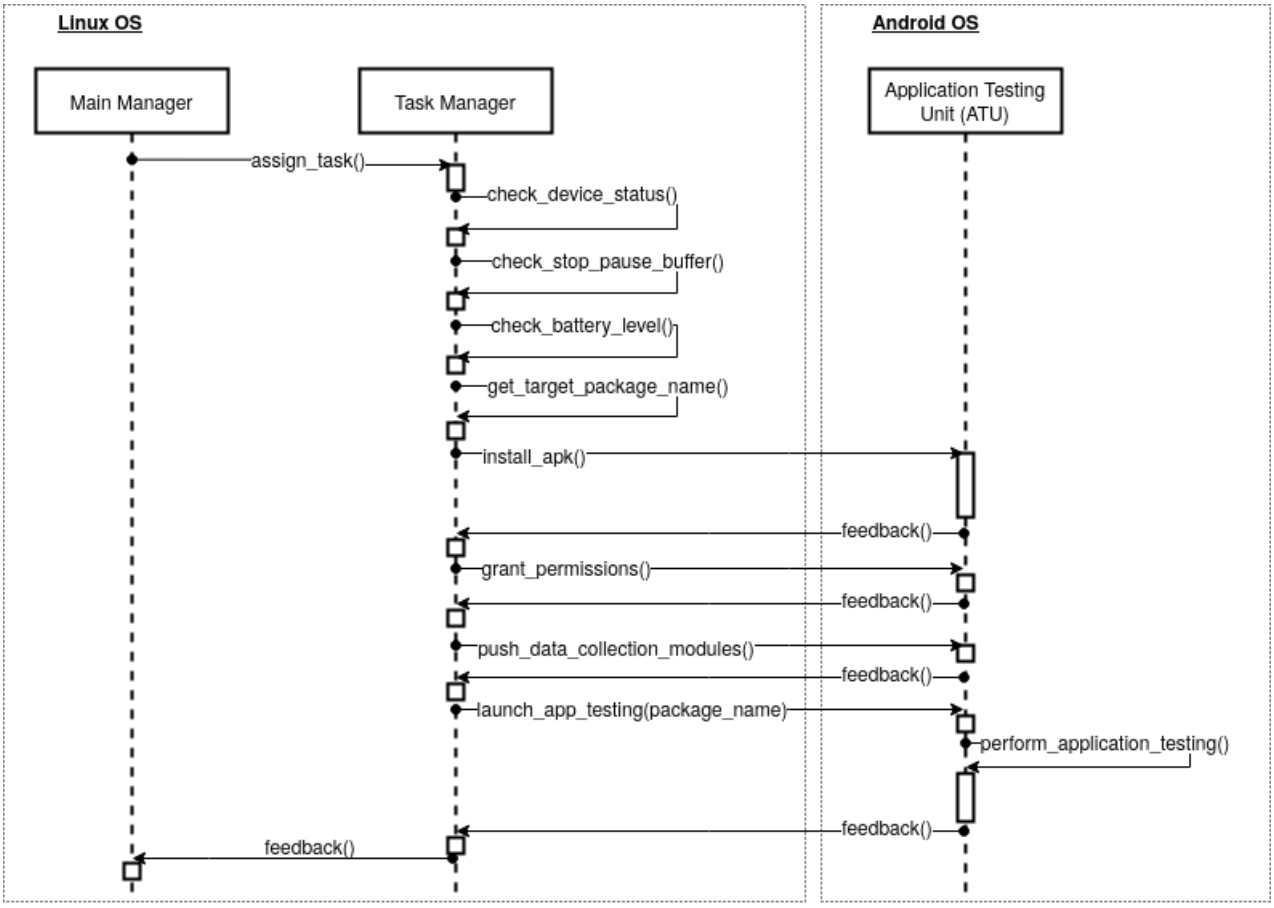


Figure 3.6: Sequence diagram of task management and preparation

the data collection process. It also terminated the PCB buffer reader. The PCB buffer reader was responsible for transferring the collected PCB records from kernel to userspace.

The data collector component was implemented as a kernel module [4] that was loaded at run-time to extend the functionality of the kernel. It created communication channels between the kernel and userspace to exchange required information for the data collection process. The data collector had another major role, that is guiding the modified CPU scheduler to collect the PCB records of all processes running the target application only.

The CPU scheduler [50] is a major part of any complex operating system. It is responsible for optimizing the utilization of the CPU by managing when and for how long each thread can occupy a CPU core. When a new task is selected, the scheduler performs the context switching operation, that is, switching from one task to another.

In order to collect PCB information for a running application, we modified the scheduler (including the context switch function) to identify any process running the target application. At each context switch, the scheduler compared the previously running thread with the target set of threads and if a match is found, it appended its PCB information to a predefined buffer. Ideally, the PCB sequence that represented the exact behavior of the application at run-time would capture PCB information after each instruction was executed. However, it was infeasible to rapidly transfer a large amount of data from the kernel to user-space without highly degrading the system performance. To overcome this problem, we considered the PCB record of a process that was being switched out to estimate process behavior during its CPU-time slice, and the PCB sequence to represent the overall behavior of the process.

Application Testing & PCB Mining Workflow

The application testing and PCB data collection process was triggered by the main management unit sending the target package name to the ATU through an Android Debug Bridge (ADB) connection. Figure 3.7 illustrates the sequence of events between the on-device system components to perform the operation of testing the target application and collecting its PCB sequence.

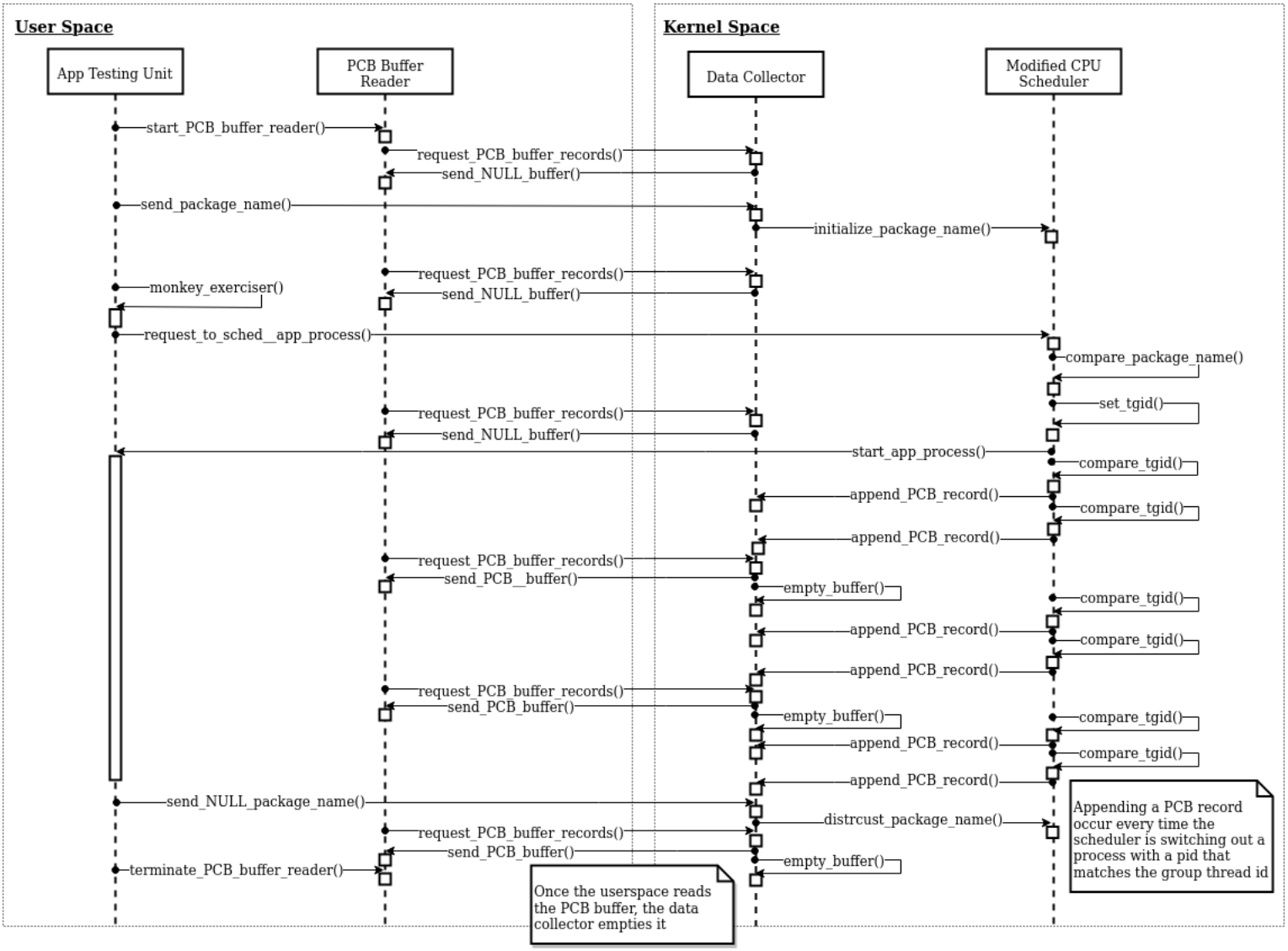


Figure 3.7: The sequence diagram of the application testing and PCB data collection workflow

The first step of application testing and PCB mining was activating the PCB buffer reader, which attempted to read the PCB buffer every 1 second until its termination. Secondly, the ATU passed the target package name to the data collector (using the write /proc file), which forwarded it to the CPU scheduler. The ATU activated the *monkeyrunner*, which launched the target app and simulated user interactions for n seconds.

When the first thread to run the application was created, the scheduler identified it by comparing its package name to the target pname. It then used its pid (thread group id $tgid$) to identify any child threads. Whenever the scheduler was switching out one of the app threads, it appended its PCB record to the PCB's buffer before activating another task. On the other hand, the data collector resets the PCB counter after the buffer reader reads its content (using the read /proc file). Appending a PCB record and resetting the PCB counter were both non-preemptive operations (i.e., they could not be interrupted by other threads), which ensured synchronization between the app threads and prevented overwriting the PCB buffer.

After *monkeyrunner* finished execution, the ATU sent an empty package name to the data collector to terminate the data collection process. The latter signaled the scheduler to destruct its pname and switch off its PCB data collection functionality. Finally, the ATU terminated the PCB buffer reader, the target application, and *monkeyrunner*. By the end of this process, a sequence of PCB records was stored as a text file on the sdcard of the device.

3.4.3 Storing & Analyzing Results

Upon successful completion of the application testing and data collection process, the manager pulled out the resulting PCB file, stored, and ran a preliminary analysis on it. The analysis provided information about the target application threads and the PCB sequence collected.

For each thread, it reported the number of voluntary and involuntary context switches performed. Voluntary context switches occurred when the running thread blocked as it waited for unavailable resources, while involuntary ones occurred when the thread used its CPU-time slice or a higher priority thread requested the CPU. The analysis also reported the occurrence of missed PCB records (if any) and its ratio to the actual number of context switches performed.

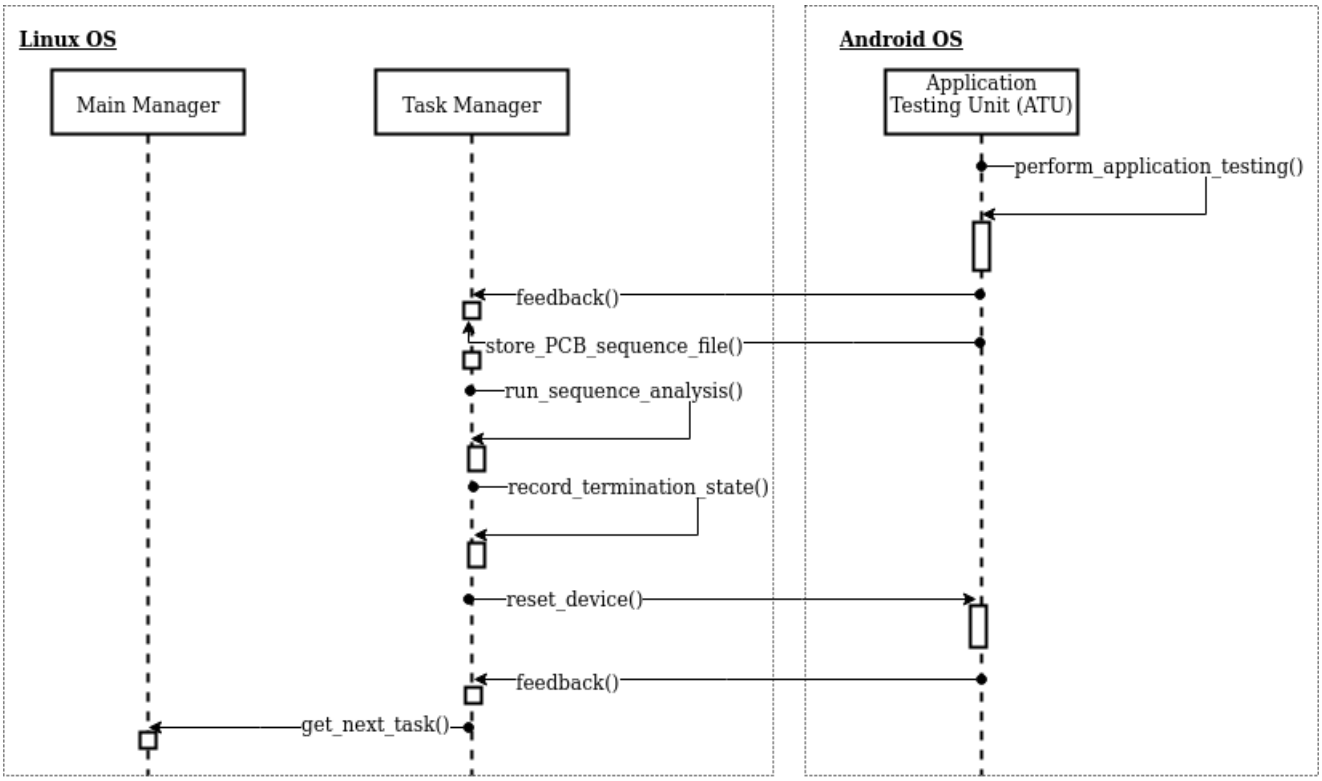


Figure 3.8: Sequence diagram of post application testing and data collection process

We found that PCB misses could occur during the execution of the $compare\_package\_name$ function and before the scheduler identified the target $tgid$.

The task manager kept track of the termination state of each task in case of success, as well as its cause in case of failure. If the task failed due to a *monkeyrunner* crash or an error in the analysis process, the manager re-ran the same task for r maximum times, otherwise, it continued to the next task. Regardless of the success or failure of the task, the manager reset the device back to a clean state (repeated if needed) before starting (or re-starting) a new task.

3.5 The PCB Dataset

We selected 3562 malware-infested and 4020 benign apps to perform the analysis and PCB mining. We successfully ran the analysis and stored the results of 2615 benign and 2502 malware-infested apps. A large number of malware-infested apps either failed to install or to complete the analysis correctly, compared to benign apps. The PCB dataset comprised N PCB sequences collected for each application, where N is the number of threads used by the application during the analysis. Figure 3.9 shows the distribution of benign and malware-infested

applications based on the number of threads. The figure shows only a minority of applications used more than 200 threads during the application runtime. More than 63% of benign apps used 50-150 threads, while 46% of malware-infested apps tended to use less than 50 threads only.

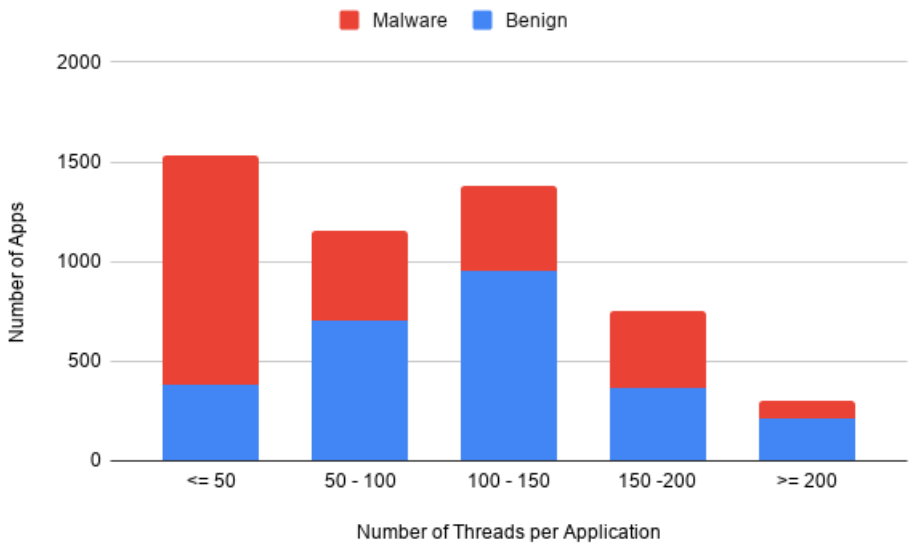
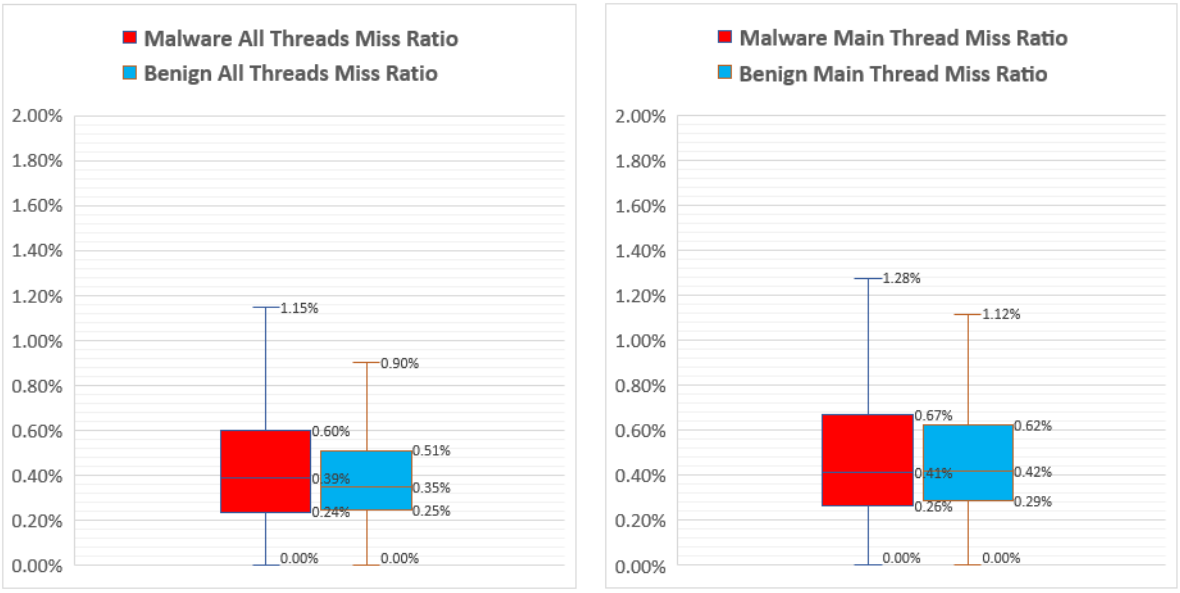


Figure 3.9: The distribution of app tests based on the number of threads running each app

Each application was executed with 15k interactions for an average time of 25 seconds. An average of 18430 and 19068 context switches occurred during the analysis for benign and malware-infested apps, respectively. Figure 3.10a illustrates how application tests were distributed based on the ratio of PCB misses for each application. The PCB miss ratio represents the number of PCB records that were not captured to the total number of context switches that occurred during the application test for all running threads. The figure shows that our approach successfully captured more than 99.5% of context switches for 70% of the tested applications, and more than 99% of context switches for 90% of the tested applications.

Figure 3.10b shows the distribution of app tests based on PCB miss ratio for the main thread running the application (first thread). We can see that the PCB miss rate for the main thread is slightly higher than its value for all threads combined. As discusses earlier (in Subsection 3.4.3) misses can likely happen before the scheduler identifies the $tgid$ of the main thread.



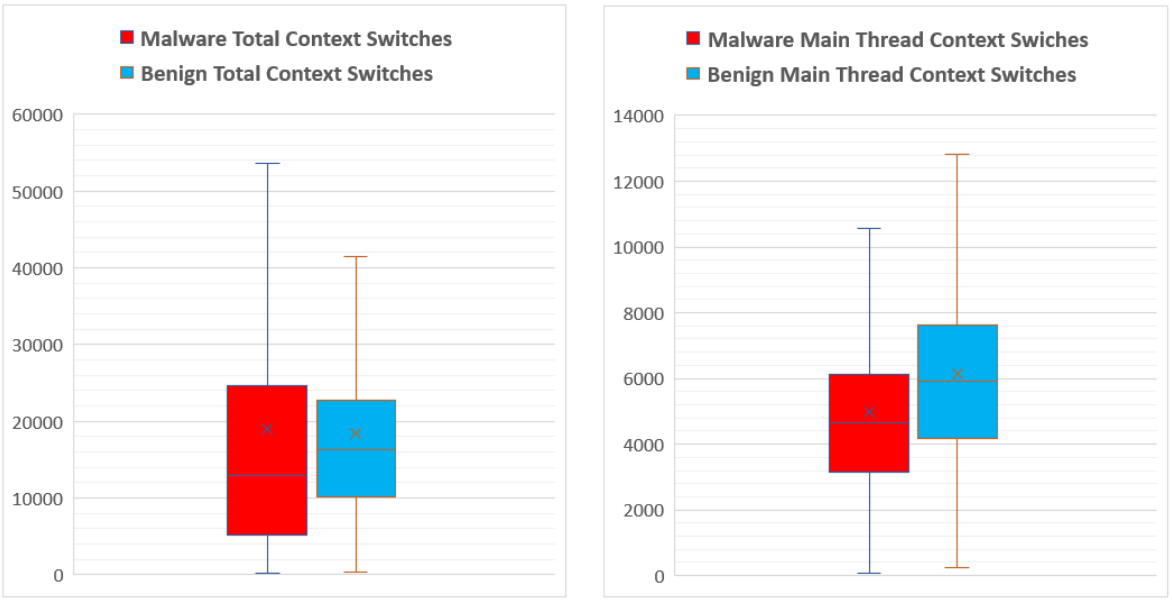
(a) (b)

Figure 3.10: Distribution of app tests for malware-infested and benign based on (a) the PCB miss ratio for all threads during the application test (b) the PCB miss ratio of the main thread during the application test

Figure 3.11 shows the distribution of malware-infested and benign apps based on the total number of context switches that occurred during the application test. Figure 3.11a shows that malware-infested applications were distributed over a wider range of total context switches compared to benign applications that tend to have similar total number of context switches over all. Figure 3.11b shows that the main thread in benign applications was more active (performed more context switches) than in malware-infested applications.

3.6 Discussion

Several research efforts have been directed toward mining PCB information to detect Android malware [70, 96, 88, 8]. In [70], the author used customized system calls to extract PCB information from the target process at a rate of 1ms. The target process is determined by matching the process name of the target application with the list of currently running processes. The research in [96, 88] proposed a similar approach of using kernel modules to transfer PCB information from kernel to user-space. They run the application and find the target process by matching its pid with the list of currently running processing. Mining PCB information



(a) (b)

Figure 3.11: Distribution of app tests for malware-infested and benign based on (a) the number of context switches for all threads combined (b) the number of context switches of the main thread that occurred during the application test

in [70, 96, 88] is requested by the user-space every 1 ms, after finding the target process. However, userspace processes have a lower priority than kernel processes, which means they can be delayed or interrupted more frequently, leading to a higher chance of missing PCB records. Additionally, the timely manner of collecting PCB information can lead to reading duplicate records. Moreover, their approach tracks the behavior of a single thread (the main process) only, whereas Android applications can use hundreds of threads during application runtime. Our proposed approach can overcome these limitations by mining PCB information at the kernel level triggered by process context switching. The high priority of the *CPU scheduler* guarantees that the PCB data mining is synchronized with process CPU time. It also allows us to collect the PCB information of all threads running the application.Table 3.2 compares our mining approach to currently available methods [70, 96, 88].

We obtained access to the dataset collected by [96] and used it to statically compare the performance of their mining approach (also followed in [88]) with our proposed mining approach, as shown in Table 3.3. Our approach successfully collected the PCB information for an average of 113 threads per application, while their approach focused on the main thread only.

Table 3.1: Comparing statistics of our proposed approach with a previous method.

	Shahzad et al. [70]	**Xinning et al. [88, 96]**	**Our Approach**
Android Version	Android 2.3.6	-	Android 8.1
Kernel Version	2.6.35.7	3.4.0	4.4.78
Data Collection Scope	Main process	Main process	Main process & all children
Requires Kernel Modification	Yes	No	Yes
Data Collection is Requested by	Userspace Reading Loop	Userspace Reading Loop	Kernel CPU-Scheduler Context Switching
Data Collection is Triggered by	Target process name found	Target pid found	Launching the matching target process name/ tgid
Data Collection & Transfer Rate	Every 1ms	Every 0.75ms	Every context switching \variable transfer rate
Data Collection & Transfer Size	Single PCB record	Single PCB record	PCB patches
User & Kernel-space Communication Channel	System calls	Kernel module	Kernel module

Table 3.2: Comparing the proposed mining approach with the available methods [70, 88, 96]

Table 3.3: Comparing statistics of the datasets collected using our proposed mining approach and another available method.

	Xinning et al. [96]	Our Dataset
Number of Samples	2550	5117
AVG Number of Threads	1	113
AVG Performed Context Switches (Main)	5647	5578
AVG Performed Context Switches (All)	5647	18431
AVG Miss Ratio	53%	0.57%
Standard Deviation of Miss Ratio	7.8	0.01
AVG Duplication Ratio	82%	0%
Standard Deviation of Duplication Ratio	8.1	0

The average number of context switches performed by the main thread in both datasets is similar. Given our PCB dataset, the main thread context switches on average represented only 40% of the total context switches performed by an application at runtime. The table shows the huge improvement achieved in capturing unique PCB records with minimal miss and duplication rates.

3.7 Conclusion

Building a reliable and efficient machine learning model highly depends on the efficiency of its dataset. Therefore, we have put great efforts into building an Android application dataset to be recent, realistic, diverse, and non-biased. We have successfully tested and collected the PCB information of 2615 and 2502 unique benign and malware-infested apps, respectively. The benign apps were collected from the top-ranked Google Play Store apps, spanning 38 categories. Selected apps were updated since January 2019, have a minimum of 1M installs, and have

an average ranking of 4.4/5. Malware-infested apps were collected from the VirusTotal Private API [37] and classified by a total of 75 Android malware detection engines. The selected samples are categorized as Trojan, Riskware, Spyware, Backdoor, Adware, Ransomware, and Exploit, and were distributed among 195 malware families, where the majority were recently discovered (in 2018, and later).

We introduce a closed dynamic malware analysis framework for testing different applications concurrently on multiple physical Android phones. The framework employs *monkeyrunner* to exercise applications. It also provides a simulated internet connection over WiFi to ensure the security of local and online resources. We used the framework to test and collect the PCB information of the selected apps.

We proposed a novel approach to mine PCB information over process execution time triggered by process context switches. Our results show that the proposed approach was capable of precisely tracking the application PCB information at runtime, and can potentially represent its behavior efficiently. For the collected samples, we were able to track an average of 112 threads per application performing an average of 18430 context switches with an average miss ratio of 0.57%.The proposed approach mined PCB information at the kernel level, synchronized with CPU utilization to minimize data collection overhead, as well as duplicate or missed PCB records. Lastly, it is worth mentioning that synchronizing PCB mining with CPU utilization requires modifying the Android kernel.

Chapter 4

Android Malware Detection using Deep Learning on PCB Information

4.1 Introduction

In this chapter, we propose a novel approach for detecting Android malware using deep learning techniques on PCB information. The proposed detection model uses Recurrent Neural Networks (RNNs) to identify the malicious behavior of a process, given a sequence of n PCB records collected during its runtime. The model was trained and tested using the PCB dataset collected in Chapter 3 of 2502 malware and 2615 benign samples. Our results show that the proposed detection model was able to correctly identify a large percentage of malware and benign samples at various points of their execution time using 12 PCBs only with an average F1-score of 95.8%. We used an RNN architecture for modeling sequence data similar to the one proposed in [95]. The architecture combines RNNs with Convolutional Neural Network (CNNs) and Deep Neural Networks (DNNs) into one model.

CNN, Long short-term memory (LSTM), and DNN models have been widely used in combination to solve machine learning problems with complex data, like sequences, time series, videos, and 2D-maps [66, 58, 67]. There are a number of advantages to combining these models. CNNs have the ability to learn and extract important features without the need for hand-crafted feature selection. RNNs are capable of learning the temporal correlations of such complex data. Finally, DNNs abstract data at each layer, allowing the model to learn complex data hierarchically and produce accurate predictions.

Identifying the malicious behavior of applications at execution time can be challenging. The variable starting point and nature of malicious behavior among different malware, as well

as a large number of features to consider, increase the complexity of the problem. To deal with this complexity, we introduce a combination of CNNs, LSTM, and DNNs into a classifier model to detect Android malware using PCB information collected over the process execution time.

4.2 Using Deep Learning & PCB information to Identify Android Malware Attacks

4.2.1 Data Preprocessing

Data preprocessing prepares the dataset to train the classifier. We have collected PCB information of 2502 malware and 2615 benign samples. Each sample activated between 10 and 250 threads, resulting in a large number of PCB sequences over the entire sample set. For training the classifier, only the main thread PCB sequence per sample was considered. The number of collected PCB records per thread was variable and depends on the lifetime of the thread.

The dataset used to train the classifier was formatted such that each sample has a sequence of n records, and each record consists of m features. Features of zero variance (i.e., observations that have no value to the learning process) were eliminated, reducing the number of features from X to Y. The final step was data normalization, where all features were scaled from their original range to have values between 0 and 1. For each feature x, we determined its minimum and maximum values and normalized each value as follows.

$$\acute{x} = \frac{x - min(x)}{max(x) - min(x)} \tag{4.1}$$

4.2.2 Model Architecture

Figure 4.1 depicts the architecture of the proposed Android malware detection model. As illustrated, the main sequence of n PCBs is divided into s consecutive blocks $[X_1, X_2, ..X_s]$ of length n/s and m PCB features. Each of these blocks is fed into one-dimensional convolutional layers for feature extraction, providing better representation of their features. Specifically, we use two CNN layers with a kernel size of two sliding along the temporal dimension of the data

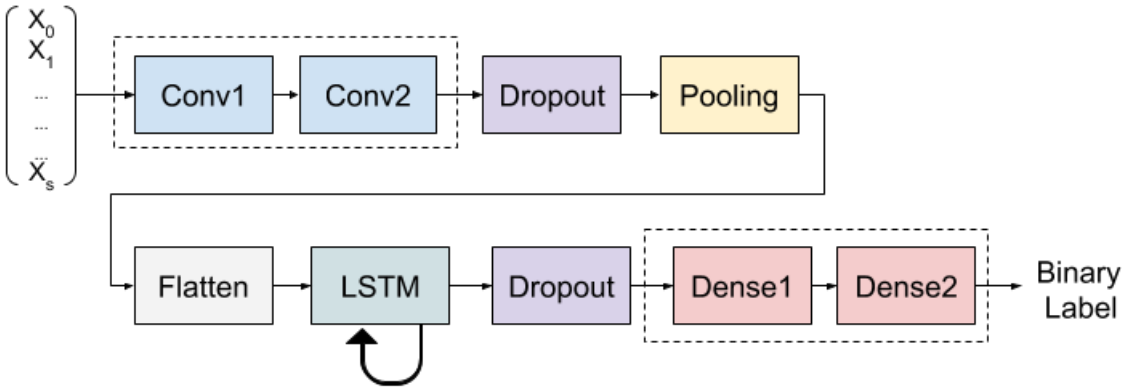


Figure 4.1: The network architecture for detecting Android malware

with padding. The activation unit used in the CNN layers is ReLU activation[38]. For each block, the CNN layers produce 64 and 32 feature maps, respectively.

It is common to use two CNN layers, followed by a dropout and a pooling layer. The dropout layer is used to prevent outfitting by randomly eliminating output links from one layer to the next during training [76]. The dropout rate used is 50% of CNN output is ignored. A one-dimensional max-pooling layer is used to downsample the sequence while maintaining structural information to prevent learning the exact position of feature values within the sequence. This is important as the starting point of the malicious behavior differs from one sample to another. A window size of two and stride of zero are used, halving the size of the CNN feature maps. These feature maps are then flattened into a single feature vector for each block.

The LSTM layer processes the sequence of s blocks and produces a single output of 100 cells (many to one). The ReLU activation was also used in the LSTM layer. Another dropout layer is used after the LSTM layer to reduce overfitting to the training data. Finally, a fully connected layer is used to interpret features extracted from the LSTM, followed by the output layer that is used to make the prediction.

4.3 Experiments & Results

This section discusses the experiments conducted to evaluate the proposed solution and produce the best detection model possible. First, we experiment with different dataset designs and compare the performance of their models. Then we study the PCB features, their distribution among malware and benign processes, and their importance.

4.3.1 Dataset Design Impact on Classifier Performance

To maximize detection performance, we followed different techniques to construct the dataset and compare corresponding performance experimentally. We used the PCB dataset collected in Chapter 3 to create three different versions, as follows. In the first dataset, the PCB sequences were truncated at n PCBs (zero-starting point dataset). The second dataset was built by selecting a sub-sequence of n PCBs at a random point of the process execution (random-starting point dataset). The third dataset was constructed by slicing each of the PCB sequences collected during the analysis into sub-sequences of n PCBs and making them into separate samples (expanded dataset).

When constructing the datasets, zero paddings were used to extend sub-sequences of less than n PCB records. For simplicity, the PCB sub-sequences were labeled by their original sample label. The datasets were shuffled then randomly split into 60% training and 40% test sets and finally normalized before use. The third dataset was shuffled and split into training and test sets before expanding it to evaluate the model on data samples it has not seen before. The training set was then shuffled again before training.

As noted previously, each data sample (sequence of n PCBs) was divided into s blocks for feature extraction using the CNN layers before being fed into the LSTM layer. The sequence length n and number of blocks s were determined experimentally. Five classifier models were trained for each n, such that $n \in \{12, 100, 200, 300, 400, 500, 1000\}$ PCBs for the smaller datasets, while $n \in \{12, 100, 300\}$ PCBs for the larger dataset. The number of blocks $s = 4$ for $n < 500$ PCBs, and $s = 10$ otherwise. The F1-score of the test set (i.e., the weighted average of precision and recall) was used to evaluate all models.

Figure 4.2 shows the F1-score average and variation of the five classifier models trained using the zero-starting point and random-starting point datasets. The x- and y-axes correspond to PCB sequence size and F1-score, respectively. The bar color represents the starting point of the PCB sub-sequence.

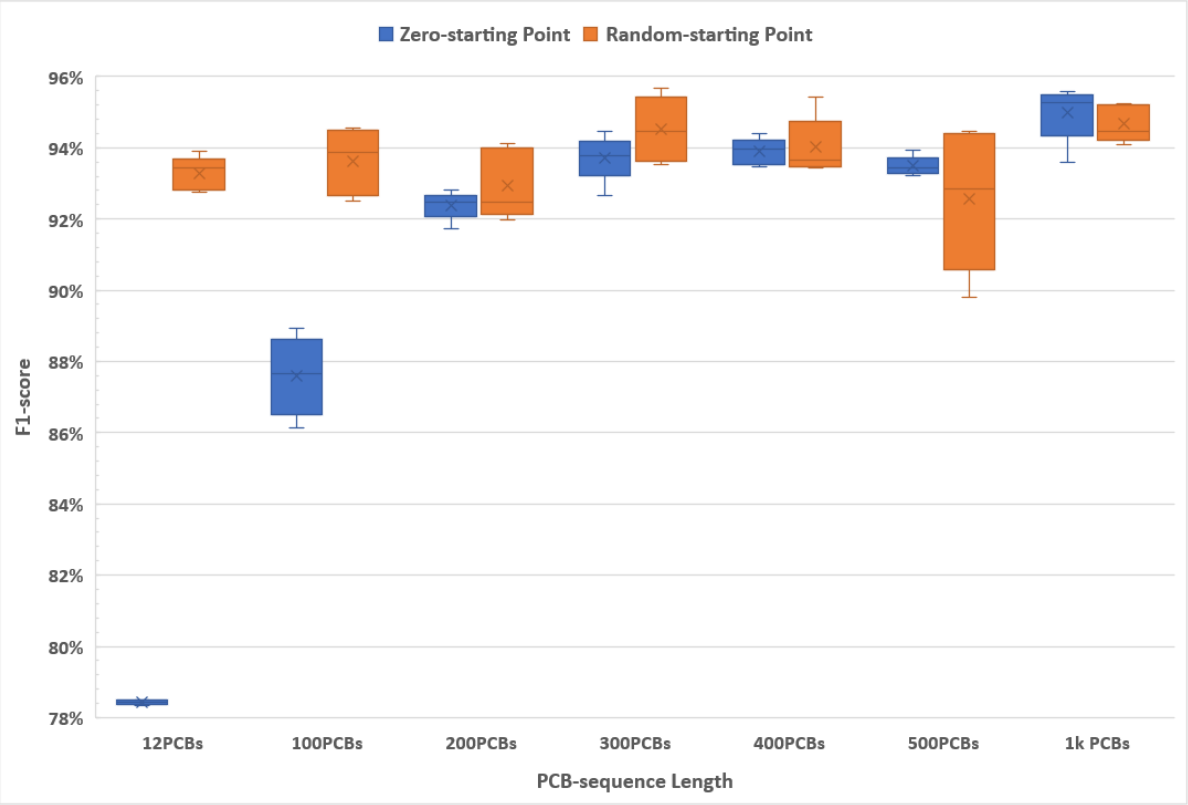


Figure 4.2: The F1-scores and their variations of the classifier models trained with the following PCB sequence setting: zero or random starting point and a sequence size of n PCBs.

Figure 4.2 shows that when training the model to identify malicious behavior using the first n PCBs of its CPU-time, the performance significantly improved by increasing sequence length. The model was able to detect some samples as early as 12 PCBs, and a sequence size of 1k PCBs improved the average F1-score to 95%. When training the classifier using the random-starting point dataset, the model was able to achieve an F1-score of 93.3% with a sequence length of 12 PCBs. Although increasing the sequence size generally did not improve classifier performance, using a much larger sequence size of 1k PCBs improved the average F1-score to 94.7%, performing similarly to 1k zero-starting point models.

Figure 4.2 also shows that random models with 100-500 PCBs had the highest variance among all models. On the other hand, the 12PCBs random classifier had higher stability at

nearly 0.4% of variance. The classifier was able to generalize better in detecting malicious behavior at different points of process execution time. This can be explained by the rich diversity of the random-starting point dataset, as well as the higher divergence between its classes, compared to the zero-starting point dataset that is limited to the first n PCBs of the process execution time only.

To further evaluate the generalizability of the random models, we trained them using the random-starting point training set and evaluated them using the expanded test set. Table 4.1 illustrates that the random models produced close results when evaluated on the expanded dataset, compared to their results on the random-starting point dataset. Moreover, classifier models trained using 12 PCBs performed slightly better (more stable and with higher F1-score) than the other models when tested on the expanded dataset. We believe this finding could be caused by the labeling approach. All sub-sequences are labeled by their original sample label, instead of labeling only sub-sequences of malicious behavior as malware. Therefore, the chance of creating false-labeled samples is higher when the PCB sequence is larger, where the actual malicious behavior can be contained in a lower number of samples which are correctly classified as malware, and the rest will be falsely labeled. Therefore, random models were able to generalize better on the expanded dataset with shorter PCB sequences.

Table 4.1: Evaluating the random models on the random-starting point versus on the expanded test set.

PCB Sequence Size	**F1-Score On Random Test Set**	**Random Test Set Size**	**F1-Score On Expanded Test Set**	**Expanded Test Set Size**
12 PCBs	$93.3\% \pm 0.42\%$	2046	$92.5\% \pm 0.29\%$	943670
100 PCBs	$93.6\% \pm 0.83\%$	2046	$92\% \pm 0.32\%$	113229
300 PCBs	$94.5\% \pm 0.82\%$	2046	$92.3\% \pm 0.71\%$	37787

Table 4.2 presents the average and standard deviation of the F1-score of the five classifier models trained and tested using the expanded dataset. The resulting models outperformed other classifiers trained using zero-starting point and random starting point datasets.

Although malware samples generally tend to hide or limit their malicious behavior, the proposed detection model was able to detect a high percent of the collected malware samples at various points of their execution time. Conducted experiments show that the RNN classifier

Table 4.2: Evaluating the classifier models trained and tested using the expanded dataset.

PCB Sequence Size	Training Set Size	Test Set Size	F1-Score
12 PCBs	1421709	943670	$95.8\% \pm 0.2\%$
100 PCBs	170543	113229	$95.5\% \pm 0.3\%$
300 PCBs	56901	37787	$95.6\% \pm 0.4\%$

performed the best with an average F1-score of 95.8% when trained using the expanded dataset with a sequence length of 12 PCBs.

4.3.2 Measuring Features Importance

To evaluate the PCB feature importance (i.e., their contribution toward model performance), we used the Permutation Feature Importance algorithm [35]. The algorithm simply measures the increase in the prediction loss after permuting feature vector j. First, we trained a classifier model using the original feature set on the expanded dataset of 12 PCBs. Then we used the test data to obtain the prediction error of the model $error_{org}$. To measure the importance score of feature j, we permutated the feature vector by randomizing its values (within [0-1)). Then, we evaluated the same classifier model using the modified test set and recorded the prediction score $error_j$. The Feature Importance score $FI_j = error_j - error_{org}$. Finally, we measured the importance score of all non-constant features and sorted them in a descending order.

Figure 4.3 shows the four groups into which PCB features were divided based on the distribution of their FI score. The sig_str is a pointer to the signal handling information struct (*signal_struct*) within the process *task_struct*. The $curr\_mm$ is a pointer to the memory management information struct (*mm_struct*), and the $sched\_str$ pointer points to the CPU scheduling information struct (*sched_info*). Features with no pointer belongs to the *task_struct* itself. It is worth mentioning that some features had common names across different structs, but did not have the same values.

Figure 4.4 illustrates two-dimensional histogram plots of malware and benign processes for the top 10% of the PCB features ranked by their FI score. Samples within the dataset had variable sequence length. Therefore, we have randomly selected a subset of a 1000 malware and a 1000 benign samples with a PCB sequence size larger than 4000 PCBs (smaller than both

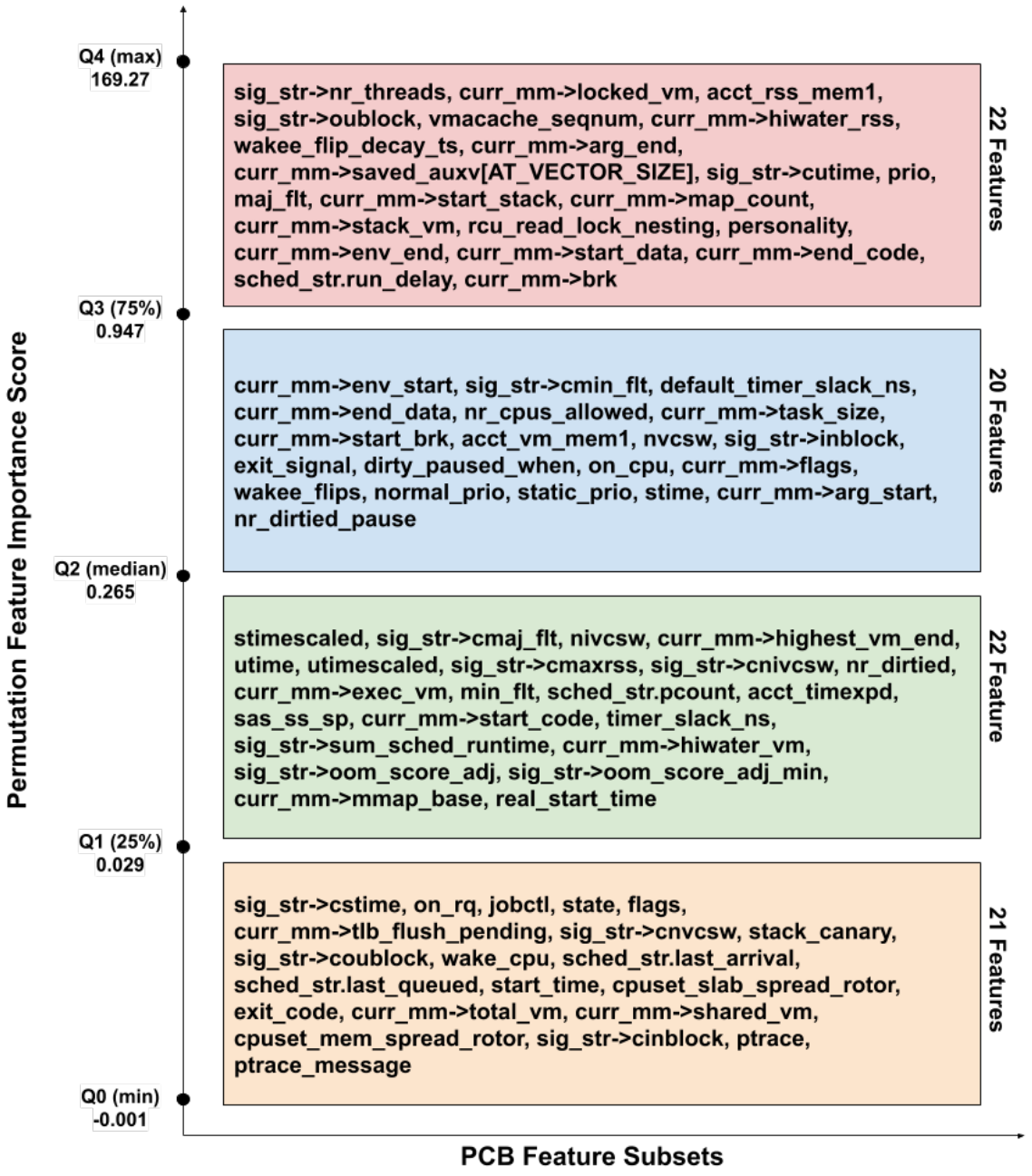


Figure 4.3: PCBs Features categories based on the Feature Importance Score (FI)

medians 4807 for the malware and 5666 for benign). All PCB sequences were then truncated to 4000 PCBs. The number of bins for the x-axis is set to the PCB sequence size to compare the PCB sequences of both groups. The y-axis is shared between the subplots for each feature. Therefore, some of the sub-figures have blank area, where one class has a wider range of values than the other.

Figure 4.4a illustrates the value of *sig_str-nr_threads* (number of threads) over time for malware and benign processes. The figure shows that malware tended to have fewer threads than benign applications throughout their execution time. The *sig_str-oublock* in figure 4.4d is the process counter for block output operations. Malware processes have shown much smaller overall values compared to benign processes.

The *acct_rss_mem1* in figure 4.4c is the cumulative counter for the Resident Set Size (RSS) that shows how much memory is allocated for the process. The figure shows that benign processes generally allocated more memory over time than malware processes. Figure 4.4e represents the *mvacache_seqnum*, that is the per-thread virtual-memory areas (VMA) cache sequence number. This *task_struct* feature is used in conjunction with another VMA sequence number within the *mm_struct* to ensure the validity of cache results. The *wakee_flip_decay_ts* in figure 4.4g represents the decay amount on switching-in (flipping) the thread to use the CPU. The variable is CPU-scheduling related and is used to prevent starvation.

The *curr_mm-locked_vm* in figure 4.4b shows the number of pages with the *PG_flag* set. This flag is used to ensure exclusive access to pages involved in I\O operations. The *curr_mm-hiwater_rss* in figure 4.4f is the thread's peak value of the RSS, for which benign processes had a larger and a wider range of values. Finally, the *curr_mm-arg_end* feature in figure 4.4h along with the *curr_mm-arg_start* feature are used to find command-line arguments of a process.

PCB features contributed differently toward making the classification decision. The top ranked features belonged to different categories: memory management, signal information, scheduling information, and others. As can be seen in figure 4.4, the most contributing features have shown differences in the overall behavior between malware and benign processes. Nevertheless, these features alone could not achieve the same performance as the model with the

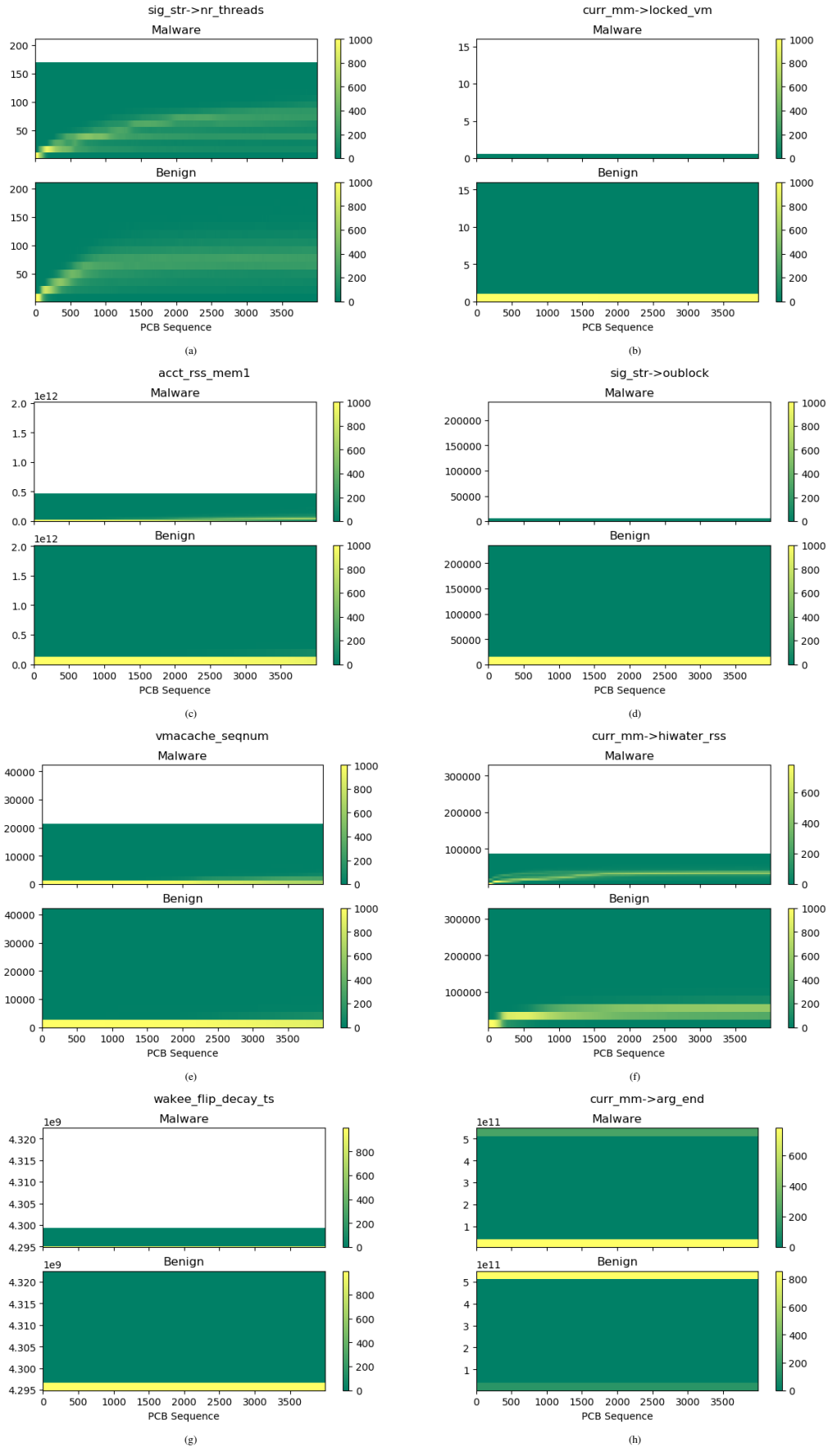


Figure 4.4: Heatmap of malware and benign processes for the top 10% features ranked by FI score.

full feature set. When training a model using the top 42 features, the model had an average F1-score of 94.5%, while training the model using the top 22 features only produced an F1-score of 94%.

4.4 Conclusion

In this chapter, we proposed a novel approach to detect Android malware using deep learning techniques on PCB information. The detection model combines CNN, LSTM, and DNN to extract important features, learn temporal correlations, and further abstract deep network findings, respectively. The structure of the detection model can be utilized with different detection window sizes, as well as PCB feature sets from other versions or other Linux-based operating systems.

The proposed detection model was able to correctly identify a large percentage of malware and benign samples at various points of their execution time using 12 PCBs only with an average F1-score of 95.8%. To the best of our knowledge, no available dynamic malware detection technique has achieved such minimal detection time. We validated our approach by training and testing the model using the PCB dataset of 2615 benign and 2502 malware infested apps. For each setting, we have trained and tested five classifier models and reported the average F1-score.

Three different dataset versions were constructed to train the detection model, the zero-starting point dataset, the random-starting point dataset, and the expanded dataset. The detection model was improved by increasing sequence size when using the zero-starting point dataset, as the behavior of malware and benign processes tend to diverge over time. The random-starting point dataset provided a higher diversity of behaviors, enabling the detection model to perform better using short PCB sequences, as well as detecting malicious behavior at various points of process execution time. The expanded dataset models achieved the best performance, with an average F1-score of 95.8% using 12 PCBs.

When building the datasets, all PCB sub-sequences were labeled based on their original sample label. However, malware may act maliciously for a limited time during their execution. We believe that our detection approach can be further improved by training the model using a

PCB dataset in which only malicious behaviors are labeled as malware, and otherwise benign, which can be costly to do.

Finally, we evaluated PCB feature importance using the Permutation Feature Importance algorithm. The feature set contained 120 features and was divided into four groups based on their score distribution. The top 10% included memory management, signals, and scheduling information. However, a classifier model trained using the top feature set was not able to improve upon the full-feature set classifier, implying that the PCB features contributed differently toward detecting malware.

Chapter 5

Conclusion & Future Work

The number and sophistication of new malware attacks targeting Android phones are continuously growing. However, the majority of Android phones are poorly or not protected against such attacks. Static malware detection techniques can provide low detection delay, but only analyze the application code and thus are vulnerable to obfuscation techniques. While dynamic detection methods can cover these limitations, they are generally time intensive and resource consuming. In this research, we propose a novel approach to detect Android malware dynamically using deep learning techniques on Process Control Block (PCB) information.

5.1 Conclusion

Employing PCB information to detect Android malware has been previously explored, where mining PCB information was triggered in the user-space, and performed for a minimum of three seconds at runtime to produce the prediction. The purpose of this dissertation was to propose a malware detection approach that can detect malicious behavior with minimal detection delay, minimal false alarms, and maximal true positives. It needs to be easily maintained and should be able to detect known malware, as well as new attacks from a variety of malware.

The efficiency and reliability of any machine learning (ML) model depends highly on the efficiency of its dataset (training and testing). Therefore, the ML model needs to be evaluated using a recent, realistic, diverse, and non-biased dataset. We did not rely on available malware datasets [16, 11, 90, 41] as they were outdated, static-based, unavailable by the time we started the research, or did not meet our sample selection criteria to produce an efficient and non-biased

sample set. Moreover, we could not directly compare our model with the work of others due to the same reasons.

To minimize detection overhead, our proposed mining approach is performed at the kernel level. To ensure the efficiency of extracted PCB sequence to represent process behavior, PCB information mining is triggered by the process context switches, and thus, is synchronized with process CPU utilization. While an average of 5577 context switches were performed by the main thread during the application runtime (25 seconds), our detection method required only 12 PCBs to perform the detection. We validated our approach using 2600 benign and 2500 malware samples and achieved an average F1-score of 95.8%.

To build an efficient, reliable, and diverse dataset, we determined criteria to select malware and benign samples. Benign samples were downloaded from the Google App Store and selected from the list of most popular Android apps across 38 categories. The chosen applications have a minimum rating score of 4.0 out of 5, have reached at least five million installs, have been updated since January 2019, and are compatible with Android versions 4.0 and later. Malware samples were collected from the VirusTotal Private API under an academic research access license. Samples were chosen if more than 40% of the malware detection engines scanned them as malware, and at least 20% of which are from the top engines. Selected samples are categorized as Trojan, Riskware, Spyware, Backdoor, Adware, Ransomware, and Exploit, and were distributed among 195 malware families.

We introduce a closed dynamic malware analysis framework to facilitate the testing of different applications concurrently on multiple physical Android phones. The framework employs *monkeyrunner* to exercise applications. It also provides a simulated internet connection over WiFi to ensure the security of local and online resources. We have used the framework to test 4020 benign and 3562 malware apps and collect their PCB information over application execution time. We have successfully completed the analysis for 2600 benign and 2500 malware samples.

The core aims of designing the PCB information mining approach were to minimize data collection overhead, as well as duplicate or missed PCB records. Additionally, collecting the PCB information of all threads running the application and not only the main thread to enable

detecting dropper malware. We implemented the PCB information mining approach using four system components. The application testing unit (UTA) and the PCB buffer reader were operating in userspace while the data collector and CPU scheduler were operating in kernel space. Mining PCB information for an application starts with passing its package name from the UTA through to the CPU scheduler. Once the application starts, the CPU scheduler appends the PCB information of all group threads to the data collector buffer at each context switch. The buffer is occasionally read by the PCB buffer reader and emptied by the data collector. At the end of application execution, the data collector destructs the package name to terminate data collection. Although the proposed approach was tested on Android phones, it can be utilized for any Linux-based operating system.

The PCB dataset comprised T PCB sequences per application, collecting a PCB sequence per thread. The average number of threads used by an application during its run time is 112 threads performing an average total of 18430 context switches. Results show that our mining approach successfully captured more than 99% of context switches for the vast majority of tested applications.

To maximize the detection performance, We introduced a detection model that combines CNN, LSTM, and DNN. The model used the PCB information of the main thread over application execution time to detect Android malware. The CNN extracts important features, eliminating the need for feature selection. The LSTM processes the data sequence and learns its temporal features. The DNN interprets the LSTM output and produces the final prediction. The structure of the detection model can be utilized with different detection window sizes, as well as PCB feature sets from other versions or other Linux-based operating systems. To train the model, we constructed three different versions from the PCB dataset; the zero-starting point dataset (sequences were truncated at n PCBs), the random-starting point dataset (sequences of n PCB starting from a random point), and the expanded dataset (slicing PCB sequences into subsequences of size n). The classifier built using the expanded dataset produced the best average F1-score of 95.8% using 12 PCBs. The second best F1-score of 94.7% was achieved using the random starting point dataset with 1000 PCBs.

To understand PCB features contribution toward detecting Android malware, we evaluated PCB feature importance using the Permutation Feature Importance algorithm. The feature set was divided into four groups based on their score distribution. The top 10% of features include the number of threads, block output operations counter, peak RSS value, and others. The top 42 features represent memory management, signals, and scheduling information. Using top-ranked features to train the classifier produced an average F1-score of 94.5% compared to 95.8% for the full-feature set classifier.

5.2 Future Work

Detecting Android malware is an ongoing race, and employing PCB information to perform the detection is a promising approach. Throughout the course of our work, we have come to believe that this approach has the potential to be improved further. Below, we briefly discuss exciting directions for future work.

Porting the solution into a stand alone Android service

In Chapter 3, we proposed a novel approach to mine PCB information over process execution time triggered by process context switches. We employed the proposed approach to collect the PCB sequences of 2600 benign and 2500 malware. In Chapter 4, we used the PCB dataset to train a classifier model to detect Android malware. Although the mining approach was performed on physical Android phones, training and testing the classifier was performed on a separate machine. Future work would be to port the detection model to run completely on the Android platform and integrate it with the proposed mining approach.

Labeling malicious behavior

In Chapter 4, we represented a new Android malware detection model that employed the time sequence of PCB information to identify malicious behavior. The best model was achieved using the expanded dataset, where PCB sequences were sliced into subsequences labeled by their original sample label (due to limited time and resources). Generally, malware does not act maliciously through its execution time. Therefore, the labeling approach can mislead the

classifier. It would be thus interesting to train the detection model using a dataset that only labels malicious behavior as malware, otherwise the sequence is labeled as benign. One possible approach is to dynamically analyze the machine code of malware samples to identify the starting and ending point of the malicious behavior (using manual or automated techniques).

Utilizing child threads in detecting dropper malware

Our proposed mining approach collects the PCB information of all threads running the Android application. We have only used PCB information on the main thread to perform the detection and did not take into consideration the possibility of malware spawning threads to carry out malicious actions. For instance, dropper malware installs and launches other types of malware to the target system. Therefore, utilizing the PCB information of the other threads in performing the detection could be interesting; however, this approach would significantly increase processing overhead due to the large number of threads used by Android applications.

Appendices

Appendix A

The Kernel-Space System Components Pseudocode

A.1 The Data Collector Pseudocode

```
#define PCBs_limit
#define PCB_buff_limit

//external functions are implemented in core.c
extern *package_name;
extern init_package_name(buf, count);
extern distruct_package_name();
extern compare_package_name(p_task_struc);

// used to control buffer reading/emptying
PCB_cnt = 0, PCB_read=0;
PCBs[PCBs_limit][PCB_buff_limit]; //PCBs buffer

//non-preemtive lock is used to manage read/write operations
define_lock(slock);
//used for proc communication
define_read_proc();
define write_proc ();
```

```
// make it reachable from other parts of the kernel
export append_PCB_record (*p_task_struct );
append_PCB_record(*p_task_struct ){
        mm_struct *curr_mm; // memory managment info
        signal_struct *sig_str; // signal handling info
        sched_info sched_str;   // CPU scheding related
        PCB_record [PCB_buff_limit];

        // if buffer is full, wait
        while(PCB_cnt >= PCBs_limit);

        // critical region to append a PCB record
        activate_lock(slock);
        // read PCB parameters
        read_task_struct (PCB_record, p_task_struct)
        read_mm_struct (PCB_record, curr_mm);
        read_signal_struct (PCB_record, sig_str);
        read_sched_info (PCB_record, sched_str);
        // append_record to PCB buffer
        append_to_PCBs_buffer(PCB_record);
        PCB_cnt += 1;
        deactivate_lock(slock);
}
send_PCB_buffer() {
        counter = 0;
        //if buffer is empty return NULL
        if (PCB_cnt ==0)
                send_NULL_buffer();
```

```
        while (counter < PCB_cnt)
                        read_record (PCBs[counter])
                        PCB_read+=1;
                        counter += 1;
}
empty_buffer() {
        cnt =0;
        // critical region to empty buffer;
        activate_lock(slock);
        // Lock emptying the buffer not the whole reading process.
        // Add PCBs appended while reading
        if (PCB_read !=PCB_cnt)
                cnt= append_missed_records (PCB_read, PCB_cnt)
        PCB_cnt = cnt ;
        deactivate_lock(slock);
}
receive_package_name( *ubuf, count){
        buf[PCB_buff_limit];
        // map the data from user-space to kernel-space.
        copy_from_user(buf, ubuf, count);

    // An empty package name  is a signal
        // to terminate the data collection process
        if (count == 1)
                distruct_package_name();
                return strlen(buf);
```

```
        // otherwise signal to start collecting PCBs
        init_package_name(buf, count);
        return strlen(buf);
}
init(){
        create_read_proc(&readOp);
        create_write_proc(&writeOp);
}
cleanup(){
        remove_read_proc(&readOp);
        remove_write_proc(&writeOp);
}
```

A.2 The modified CPU Scheduler Pseudocode

```
#define MAXSIZE
*package_name [MAXSIZE] = NULL;
// make it accessible for other kernel module
EXPORT_SYMBOL(package_name);
bool firstTime= True;
pid_t main_pid= NULL;
int init_package_name(received_pname, count){
        int i=0;
        //make sure the local buf is big enough
        if(count > MAXSIZE)
                return -1;
        package_name = received_pname;
        return 0;
}
EXPORT_SYMBOL(init_package_name);
```

```
void distruct_package_name(){
        package_name = NULL;
}
EXPORT_SYMBOL(distruct_package_name);

bool compare_package_name(*p_task_struct){
    // compare the task package_name with the received one
    // if matching set its pid as the main pid
        get_task_comm(comm, p_task_struct);
        if comm ==  package_name [0: len(comm)]{
                if firstTime {
                        main_pid = p_task_struct->pid;
                        firstTime = False;
                        }
                return 1;}
        return 0;
}
EXPORT_SYMBOL(compare_package_name);

context_switch(rq *rq, task_struct *prev, task_struct *next){
        // prepare to switch the CPU
        prepare_task_switch(rq, prev, next);
        arch_start_context_switch(prev);
        // // more preparation
        // if package_name is set
         // --> the application is being executed
        if (package_name != NULL){
```

```
            if main_pid != NULL:
                    if (main_pid == prev->tgid)
                            append_PCB_record(prev);
                    else if (compare_package_name (prev))
                            append_PCB_record(prev);
                    }
            // switch memory space
            switch_mm(prev, next);
            /* Here we just switch the register state and the stack. */
            switch_to(prev, next, prev);
            // more switching
            return finish_task_switch(prev);
}
```

Appendix B

Struct task_struct

```
struct task_struct {
        volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped
        void *stack;
        atomic_t usage;
        unsigned int flags;     /* per process flags, defined below */
        unsigned int ptrace;
#ifdef CONFIG_SMP
        struct llist_node wake_entry;
        int on_cpu;
        unsigned int wakee_flips;
        unsigned long wakee_flip_decay_ts;
        struct task_struct *last_wakee;
        int wake_cpu;
#endif
        int on_rq;
        int prio, static_prio, normal_prio;
        unsigned int rt_priority;
        const struct sched_class *sched_class;
        struct sched_entity se;
        struct sched_rt_entity rt;
```

```
#ifdef CONFIG_SCHED_HMP
        struct ravg ravg;
        /* 'init_load_pct' represents the initial
        task load assigned to children of this task*/
    u32 init_load_pct;
        u64 last_wake_ts;
        u64 last_switch_out_ts;
        u64 last_cpu_selected_ts;
        struct related_thread_group *grp;
        struct list_head grp_list;
        u64 cpu_cycles;
#endif
        struct sched_dl_entity dl;
#ifdef CONFIG_BLK_DEV_IO_TRACE
        unsigned int btrace_seq;
#endif
        unsigned int policy;
        int nr_cpus_allowed;
        cpumask_t cpus_allowed;
#ifdef CONFIG_PREEMPT_RCU
        int rcu_read_lock_nesting;
        union rcu_special rcu_read_unlock_special;
        struct list_head rcu_node_entry;
        struct rcu_node *rcu_blocked_node;
#endif /* #ifdef CONFIG_PREEMPT_RCU */

#ifdef CONFIG_SCHED_INFO
        struct sched_info sched_info;
```

```
#endif
        struct list_head tasks;
        struct mm_struct *mm, *active_mm;
        /* per-thread vma caching */
        u32 vmacache_seqnum;
        struct vm_area_struct *vmacache[VMACACHE_SIZE];
/* task state */
        int exit_state;
    int exit_code, exit_signal;
        int pdeath_signal;/*The signal sent when the parent dies*/
        unsigned long jobctl;/* JOBCTL_*, siglock protected */
        /*Used for emulating ABI behavior of previous Linux versions*/
        unsigned int personality;
        /*scheduler bits, serialized by scheduler locks */
        unsigned sched_reset_on_fork:1;
        unsigned sched_contributes_to_load:1;
        unsigned sched_migrated:1;
        unsigned :0; /* force alignment to the next boundary */
        /* unserialized, strictly 'current' */
        unsigned in_execve:1; /* bit to tell LSMs we're in execve */
        unsigned in_iowait:1;
        unsigned long atomic_flags; /* Flags needing atomic access. */
        struct restart_block restart_block;
        pid_t pid;
        pid_t tgid;
#ifdef CONFIG_CC_STACKPROTECTOR
        /* Canary value for the -fstack-protector gcc feature */
        !unsigned long stack_canary;
```

```
#endif
        /* pointers to (original) parent process, youngest child,
        younger sibling,older sibling, respectively.
        (p->father can be replaced with p->real_parent->pid)*/
        struct task_struct __rcu *real_parent; /* real parent process*/
        /* recipient of SIGCHLD, wait4() reports*/
        struct task_struct __rcu *parent;
        /*children/sibling forms the list of my natural children*/
        struct list_head children;/* list of my children*/
        struct list_head sibling;/* linkage in my parent's children list
        struct task_struct *group_leader;/* threadgroup leader */
        /* ptraced is the list of tasks this task is using ptrace on.
         * This includes both natural children and PTRACE_ATTACH targets.
         * p->ptrace_entry is p's link on the p->parent->ptraced list.*/
        struct list_head ptraced;
        struct list_head ptrace_entry;
        /* PID/PID hash table linkage. */
        struct pid_link pids[PIDTYPE_MAX];
        struct list_head thread_group;
        struct list_head thread_node;
        struct completion *vfork_done;/* for vfork() */
        int __user *set_child_tid;/* CLONE_CHILD_SETTID */
        int __user *clear_child_tid;/* CLONE_CHILD_CLEARTID */
        cputime_t utime, stime, utimescaled, stimescaled;
        cputime_t gtime;
        struct prev_cputime prev_cputime;
        unsigned long nvcsw, nivcsw; /* context switch counts */
        u64 start_time; /* monotonic time in nsec */
```

```
        u64 real_start_time;/* boot based time in nsec */
/* mm fault and swap info: this can arguably be
seen as either mm-specific or thread-specific */
        unsigned long min_flt, maj_flt;
        struct task_cputime cputime_expires;
        struct list_head cpu_timers[3];
/* process credentials */
 /* Tracer's credentials at attach */
        const struct cred __rcu *ptracer_cred;
/* objective and real subjective task credentials (COW) */
        const struct cred __rcu *real_cred;
/* effective (overridable) subjective task credentials (COW) */
        const struct cred __rcu *cred;
        char comm[TASK_COMM_LEN]; /* executable name excluding path
                - access with [gs]et_task_comm (which lock
                                        it with task_lock())
                - initialized normally by setup_new_exec */
/* file system info */
        struct nameidata *nameidata;
#ifdef CONFIG_DETECT_HUNG_TASK
/* hung task detection */
        !unsigned long last_switch_count;
#endif
/* filesystem information */
        struct fs_struct *fs;
/* open file information */
        struct files_struct *files;
/* namespaces */
```

```
        struct nsproxy *nsproxy;
/* signal handlers */
        struct signal_struct *signal;
        struct sighand_struct *sighand;
        sigset_t blocked, real_blocked;
/* restored if set_restore_sigmask() was used */
        sigset_t saved_sigmask;
        struct sigpending pending;
        unsigned long sas_ss_sp;
        size_t sas_ss_size;
        struct callback_head *task_works;
        struct audit_context *audit_context;
        struct seccomp seccomp;
/* Thread group tracking */
        u32 parent_exec_id;
    u32 self_exec_id;
/* Protection of (de-)allocation:
mm, files, fs, tty, keyrings, mems_allowed, mempolicy */
        spinlock_t alloc_lock;
        /* Protection of the PI data structures: */
        raw_spinlock_t pi_lock;
        struct wake_q_node wake_q;
/* journalling filesystem info */
        void *journal_info;
/* stacked block device info */
        struct bio_list *bio_list;
/* VM state */
        struct reclaim_state *reclaim_state;
```

```
        struct backing_dev_info *backing_dev_info;
        struct io_context *io_context;
    unsigned long ptrace_message;
        siginfo_t *last_siginfo; /* For ptrace use.  */
        struct task_io_accounting ioac;
#if defined(CONFIG_TASK_XACCT)
        u64 acct_rss_mem1;/* accumulated rss usage */
        u64 acct_vm_mem1;/* accumulated virtual memory usage */
        cputime_t acct_timexpd;/* stime + utime since last update */
#endif
#ifdef CONFIG_CPUSETS
        nodemask_t mems_allowed;/* Protected by alloc_lock */
        seqcount_t mems_allowed_seq;/* Seqence no to catch updates */
        int cpuset_mem_spread_rotor;
        int cpuset_slab_spread_rotor;
#endif
        struct rcu_head rcu;
        /*cache last used pipe for splice */
        struct pipe_inode_info *splice_pipe;
        struct page_frag task_frag;
        /* when (nr_dirtied >= nr_dirtied_pause), it's time to call
         * balance_dirty_pages() for some dirty throttling pause */
        int nr_dirtied;
        int nr_dirtied_pause;
         /* start of a write-and-pause period */
        !unsigned long dirty_paused_when;
        /*time slack values; these are used to round up poll() and
         * select() etc timeout values. These are in nanoseconds.*/
```

```
        u64 timer_slack_ns;
        u64 default_timer_slack_ns;
#ifdef CONFIG_TRACING
        /* state flags for use by tracers */
        unsigned long trace;
        /* bitmask and counter of trace recursion */
        unsigned long trace_recursion;
#endif /* CONFIG_TRACING */
        int pagefault_disabled;
/* CPU-specific state of this task */
        struct thread_struct thread;
/* * WARNING: on x86, 'thread_struct' contains a variable-sized
 * structure.  It *MUST* be at the end of 'task_struct'.
 * Do not put anything below here! */};
```

Bibliography

[1] URL: https://www.kaspersky.com/about/press-releases/2017_kaspersky-lab-detects-360000-new-malicious-files-daily.

[2] URL: https://usa.kaspersky.com/resource-center/threats/computer-viruses-vs-worms.

[3] URL: https://usa.kaspersky.com/.

[4] *1.1. What Is A Kernel Module?* URL: https://linux.die.net/lkmpg/x40.html.

[5] Yousra Aafer, Wenliang Du, and Heng Yin. "Droidapiminer: Mining api-level features for robust malware detection in android". In: *International conference on security and privacy in communication systems*. Springer. 2013, pp. 86–103.

[6] *Advanced wireless settings*. URL: https://wiki.dd-wrt.com/wiki/index.php/Advanced_wireless_settings#AP_Isolation.

[7] *Adware*. URL: https://encyclopedia.kaspersky.com/knowledge/adware/.

[8] Heba Alawneh, David Umphress, and Anthony Skjellum. "Android Malware Detection Using Neural Networks & Process Control Block Information". In: *2019 14th International Conference on Malicious and Unwanted Software (MALWARE)*. 2019, pp. 3–12. ISBN: 978-0-578-58383-9.

[9] Domenico Amalfitano et al. "MobiGUITAR: Automated model-based testing of mobile apps". In: *IEEE software* 32.5 (2014), pp. 53–59.

[10] Saswat Anand et al. “Automated concolic testing of smartphone apps”. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 2012, pp. 1–11.

[11] *Android Malware Genome Project*. URL: http://www.malgenomeproject.org/.

[12] *Android malware on Google Play adds devices to botnet*. URL: https://www.symantec.com/connect/blogs/android-malware-google-play-adds-devices-botnet-and-performs-ddos-attacks.

[13] *Android Open Source Project*. URL: https://source.android.com/.

[14] *Android/Trojan.Agent*. URL: https://blog.malwarebytes.com/detections/android-trojan-agent.

[15] Nicoló Andronio, Stefano Zanero, and Federico Maggi. “Heldroid: Dissecting and detecting mobile ransomware”. In: *International workshop on recent advances in intrusion detection*. Springer. 2015, pp. 382–404.

[16] Daniel Arp et al. “Drebin: Effective and explainable detection of android malware in your pocket.” In: *Ndss*. Vol. 14. 2014, pp. 23–26.

[17] Saba Arshad et al. “Android malware detection & protection: a survey”. In: *International Journal of Advanced Computer Science and Applications* 7.2 (2016), pp. 463–475.

[18] *AV-TEST Security Report 2018/2019*. Accessed: March 2020. July 2019. URL: https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2018-2019.pdf.

[19] Tanzirul Azim and Iulian Neamtiu. “Targeted and depth-first exploration for systematic testing of android apps”. In: *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 2013, pp. 641–660.

[20] Michael Bierma et al. “Andlantis: Large-scale Android dynamic analysis”. In: *arXiv preprint arXiv:1410.7751* (2014).

[21] Nataniel P Borges Jr, Maria Gómez, and Andreas Zeller. “Guiding app testing with mined interaction models”. In: *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. ACM. 2018, pp. 133–143.

[22] Nataniel P Borges, Maria Gómez, and Andreas Zeller. “Guiding app testing with mined interaction models”. In: *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE. 2018, pp. 133–143.

[23] Patrick Carter et al. “CuriousDroid: automated user interface interaction for android application analysis sandboxes”. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2016, pp. 231–249.

[24] Carlos A Castillo et al. “Android malware past, present, and future”. In: *White Paper of McAfee Mobile Security Working Group* 1 (2011), p. 16.

[25] Wontae Choi, George Necula, and Koushik Sen. “Guided gui testing of android apps with minimal restart and approximate learning”. In: *Acm Sigplan Notices* 48.10 (2013), pp. 623–640.

[26] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. “Automated test input generation for android: Are we there yet?(e)”. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2015, pp. 429–440.

[27] *Contributors*. URL: `https://support.virustotal.com/hc/en-us/articles/115002146809-Contributors`.

[28] *Definitive data and analysis for the mobile industry*. Apr. 2018. URL: `https://www.gsmaintelligence.com/`.

[29] Christos Douligeris and Aikaterini Mitrokotsa. “DDoS attacks and defense mechanisms: classification and state-of-the-art”. In: *Computer Networks* 44.5 (2004), pp. 643–666.

[30] William Enck, Machigar Ongtang, and Patrick McDaniel. “On lightweight mobile phone application certification”. In: *Proceedings of the 16th ACM conference on Computer and communications security*. ACM. 2009, pp. 235–245.

[31] William Enck et al. “TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones”. In: *ACM Transactions on Computer Systems (TOCS)* 32.2 (2014), p. 5.

[32] *Exploit:Android/GingerBreak*. URL: https://www.f-secure.com/v-descs/exploit_android_gingerbreak.shtml.

[33] Parvez Faruki et al. “Android security: a survey of issues, malware penetration, and defenses”. In: *IEEE communications surveys & tutorials* 17.2 (2015), pp. 998–1022.

[34] Ali Feizollah et al. “A review on feature selection in mobile malware detection”. In: *Digital investigation* 13 (2015), pp. 22–37.

[35] Aaron Fisher, Cynthia Rudin, and Francesca Dominici. “Model class reliance: Variable importance measures for any machine learning model class, from the” rashomon” perspective”. In: *arXiv preprint arXiv:1801.01489* 68 (2018).

[36] *Free antivirus protection that never quits*. URL: https://www.avast.com/en-us/index#pc.

[37] *Free Online Virus, Malware and URL Scanner*. URL: https://www.virustotal.com/en.

[38] Richard HR Hahnloser et al. “Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit”. In: *Nature* 405.6789 (2000), p. 947.

[39] Shuai Hao et al. “PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps”. In: *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. 2014, pp. 204–217.

[40] Immanuel. *What is a Logic Bomb? — How to Prevent this Logic Bomb Malware?* Dec. 2018. URL: https://antivirus.comodo.com/blog/comodo-news/logic-bomb/.

[41] ElMouatez Billah Karbab et al. “MalDozer: Automatic framework for android malware detection using deep learning”. In: *Digital Investigation* 24 (2018), S48–S59.

[42] Hahnsang Kim, Joshua Smith, and Kang G Shin. “Detecting energy-greedy anomalies and mobile malware variants”. In: *Proceedings of the 6th international conference on Mobile systems, applications, and services*. ACM. 2008, pp. 239–252.

[43] Elijah Lawal. *Shielding you from Potentially Harmful Applications*. Feb. 2017. URL: https://www.blog.google/technology/safety-security/shielding-you-potentially-harmful-applications/.

[44] *Leader in Cyber Threat Analysis and Response*. URL: https://global.ahnlab.com/site/main.do.

[45] Jin Li et al. “Significant permission identification for machine-learning-based android malware detection”. In: *IEEE Transactions on Industrial Informatics* 14.7 (2018), pp. 3216–3225.

[46] Yuanchun Li et al. “A Deep Learning based Approach to Automated Android App Testing”. In: *arXiv preprint arXiv:1901.02633* (2019).

[47] Yuanchun Li et al. “DroidBot: a lightweight UI-guided test input generator for Android”. In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE. 2017, pp. 23–26.

[48] LineageOS. *LineageOS Android Distribution*. URL: https://lineageos.org/.

[49] *List of Android Most Popular Google Play Apps*. URL: https://www.androidrank.org/android-most-popular-google-play-apps.

[50] Robert Love. *Process Scheduling*. Addison-Wesley Publishing Company, 2010.

[51] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. “Dynodroid: An input generation system for android apps”. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM. 2013, pp. 224–234.

[52] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. “Evodroid: Segmented evolutionary testing of android apps”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2014, pp. 599–609.

[53] Ke Mao, Mark Harman, and Yue Jia. “Crowd intelligence enhances automated mobile testing”. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2017, pp. 16–26.

[54] Ke Mao, Mark Harman, and Yue Jia. “Sapienz: Multi-objective automated testing for Android applications”. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 2016, pp. 94–105.

[55] McAfee. *Chinese Worm Infects Thousands of Android Phones*. Aug. 2014. URL: https://www.mcafee.com/blogs/other-blogs/mcafee-labs/chinese-worm-infects-thousands-android-phones/.

[56] Fairuz Amalina Narudin et al. “Evaluation of machine learning classifiers for mobile malware detection”. In: *Soft Computing* 20.1 (2016), pp. 343–357.

[57] *Operating System Market Share Worldwide*. July 2018. URL: https://gs.statcounter.com/os-market-share.

[58] Francisco Javier Ordóñez and Daniel Roggen. “Deep convolutional and lstm recurrent neural networks for multimodal wearable activity recognition”. In: *Sensors* 16.1 (2016), p. 115.

[59] Stelian Pilici. *How To Remove the Android Lockscreen Ransomware (with Pictures)*. Aug. 2018. URL: https://malwaretips.com/blogs/android-police-virus-removal/.

[60] Radu S Pirscoveanu et al. “Analysis of malware behavior: Type classification using machine learning”. In: *2015 International conference on cyber situational awareness, data analytics and assessment (CyberSA)*. IEEE. 2015, pp. 1–7.

[61] James Raymond. *Backdoor Virus — How to Remove a Backdoor Virus from Your System*. Nov. 2018. URL: https://antivirus.comodo.com/blog/comodo-news/backdoor-virus/.

[62] *RiskTool*. URL: https://encyclopedia.kaspersky.com/knowledge/risktool/.

[63] *Riskware:Android/SmsPay*. URL: https://www.f-secure.com/sw-desc/riskware_android_smspay.shtml.

[64] *Riskware:Android/SmsPay.variant!Online*. URL: https://www.f-secure.com/sw-desc/riskware_android_smspay_online.shtml.

[65] *Riskware:Android/Smsreg.variant!Online*. URL: https://www.f-secure.com/sw-desc/riskware_android_smsreg_online.shtml.

[66] Tara N Sainath et al. "Convolutional, long short-term memory, fully connected deep neural networks". In: *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2015, pp. 4580–4584.

[67] Tara N Sainath et al. "Learning the speech front-end with raw waveform CLDNNs". In: *Sixteenth Annual Conference of the International Speech Communication Association*. 2015.

[68] Asaf Shabtai et al. "Andromaly: a behavioral malware detection framework for android devices". In: *Journal of Intelligent Information Systems* 38.1 (2012), pp. 161–190.

[69] Farrukh Shahzad, Muhammad Shahzad, and Muddassar Farooq. "In-execution dynamic malware analysis and detection by mining information in process control blocks of Linux OS". In: *Information Sciences* 231 (2013), pp. 45–63.

[70] Farrukh Shahzad et al. "Tstructdroid: Realtime malware detection using in-execution dynamic analysis of kernel process control blocks on android". In: *National University of Computer & Emerging Sciences, Islamabad, Pakistan, Tech. Rep* (2013).

[71] Shailaja Shankar et al. *The Do You Knows of DDoS Attacks*. Mar. 2018. URL: https://securingtomorrow.mcafee.com/consumer/mobile-ddos/.

[72] Shina Sheen, R Anitha, and V Natarajan. “Android based malware detection using a multifeature collaborative decision fusion approach”. In: *Neurocomputing* 151 (2015), pp. 905–912.

[73] Adrian J Shepherd. *Second-order methods for neural networks: Fast and reliable training methods for multi-layer perceptrons*. Springer Science & Business Media, 2012. Chap. Process Managment.

[74] *SMS Trojan*. URL: https://blog.malwarebytes.com/threats/sms-trojan/.

[75] Alireza Souri and Rahil Hosseini. “A state-of-the-art survey of malware detection approaches using data mining techniques”. In: *Human-centric Computing and Information Sciences* 8.1 (2018), p. 3.

[76] Nitish Srivastava et al. “Dropout: a simple way to prevent neural networks from overfitting”. In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[77] Ting Su et al. “Guided, stochastic model-based GUI testing of Android apps”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 2017, pp. 245–256.

[78] Kimberly Tam et al. “The evolution of android malware and android analysis techniques”. In: *ACM Computing Surveys (CSUR)* 49.4 (2017), p. 76.

[79] Technical Marketing Team. *Ransomware Past, Present, and Future*. URL: https://documents.trendmicro.com/assets/wp/wp-ransomware-past-present-and-future.pdf.

[80] *Test antivirus software for Android - November 2019*. Nov. 2019. URL: https://www.av-test.org/en/antivirus/mobile-devices/android/november-2019/.

[81] *The APK Downloader for Windows, Linux and MacOS*. URL: https://raccoon.onyxbits.de/.

[82] *Trojan: Android/DroidKungFu.C Description — F-Secure Labs*. Accessed: March 2019. URL: https://www.f-secure.com/vdescs/trojan_android_droidkungfu_c.shtml.

[83] *Trojan:Android/SmsSpy*. URL: https://www.f-secure.com/v-descs/trojan_android_smsspy.shtml.

[84] *Trojanized adware family abuses accessibility service to install whatever apps it wants*. URL: https://blog.lookout.com/shedun-trojanized-adware.

[85] *UI/Application Exerciser Monkey — Android Developers*. URL: https://developer.android.com/studio/test/monkey.

[86] *VirusTotal*. URL: https://developers.virustotal.com/reference.

[87] Wenyu Wang et al. "An empirical study of android test generation tools in industrial cases". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018, pp. 738–748.

[88] Xinning Wang, Chong Li, and Dalei Song. "CrowdNet: Identifying Large-Scale Malicious Attacks Over Android Kernel Structures". In: *IEEE Access* 8 (2020), pp. 15823–15837.

[89] Yang Wang et al. "Quantitative security risk assessment of android permissions and applications". In: *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer. 2013, pp. 226–241.

[90] Fengguo Wei et al. "Deep ground truth analysis of current android malware". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2017, pp. 252–276.

[91] *Welcome to the INetSim project homepage!* URL: https://www.inetsim.org/.

[92] *Worm:Android/Samsapo*. URL: https://www.f-secure.com/v-descs/worm_android_samsapo.shtml.

[93] Dong-Jie Wu et al. “Droidmat: Android malware detection through manifest and api calls tracing”. In: *2012 Seventh Asia Joint Conference on Information Security*. IEEE. 2012, pp. 62–69.

[94] Songyang Wu et al. “Effective detection of android malware based on the usage of data flow APIs and machine learning”. In: *Information and software technology* 75 (2016), pp. 17–25.

[95] SHI Xingjian et al. “Convolutional LSTM network: A machine learning approach for precipitation nowcasting”. In: *Advances in neural information processing systems*. 2015, pp. 802–810.

[96] Wang Xinning. “Android Malware Detection through Machine Learning on Kernel Task Structure”. PhD dissertation. Auburn University, 2017.

[97] Weilin Xu, Yanjun Qi, and David Evans. “Automatically evading classifiers”. In: *Proceedings of the 2016 network and distributed systems symposium*. Vol. 10. 2016.

[98] Lok Kwong Yan and Heng Yin. “Droidscope: Seamlessly reconstructing the {OS} and dalvik semantic views for dynamic android malware analysis”. In: *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*. 2012, pp. 569–584.

[99] Ping Yan and Zheng Yan. “A survey on dynamic mobile malware detection”. In: *Software Quality Journal* 26.3 (2018), pp. 891–919.

[100] Wei Yang, Mukul R Prasad, and Tao Xie. “A grey-box approach for automated GUI-model generation of mobile applications”. In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2013, pp. 250–265.

[101] Suleiman Y Yerima et al. “A new android malware detection approach using bayesian classification”. In: *2013 IEEE 27th international conference on advanced information networking and applications (AINA)*. IEEE. 2013, pp. 121–128.

[102] Mahmood Yousefi-Azar et al. “Malytics: a malware detection scheme”. In: *IEEE Access* 6 (2018), pp. 49418–49431.

[103] Zhenlong Yuan, Yongqiang Lu, and Yibo Xue. “Droiddetector: android malware characterization and detection using deep learning”. In: *Tsinghua Science and Technology* 21.1 (2016), pp. 114–123.

[104] *Zero-day vulnerability: What it is, and how it works*. Accessed: March 2019. URL: https://us.norton.com/internetsecurity-emerging-threats-how-do-zero-day-vulnerabilities-work-30sectech.html.

[105] Min Zhao et al. “Antimaldroid: An efficient svm-based malware detection framework for android”. In: *International Conference on Information Computing and Applications*. Springer. 2011, pp. 158–166.

[106] Cong Zheng et al. “Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications”. In: *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*. 2012, pp. 93–104.

[107] Haibing Zheng et al. “Automated test input generation for android: Towards getting there in an industrial case”. In: *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE. 2017, pp. 253–262.

[108] Yajin Zhou and Xuxian Jiang. “Dissecting android malware: Characterization and evolution”. In: *2012 IEEE symposium on security and privacy*. IEEE. 2012, pp. 95–109.

[109] Dali Zhu et al. “DeepFlow: Deep learning-based malware detection by mining Android application for abnormal usage of sensitive data”. In: *2017 IEEE symposium on computers and communications (ISCC)*. IEEE. 2017, pp. 438–443.