COMEX: Deeply Observing Application Behavior on Real Android Devices

Zeya Umayya*
IIITD
zeyau@iiitd.ac.in

Dhruv Malik*
IIITD
dhruv20373@iiitd.ac.in

Arpit Nandi*
IIITD
arpit20179@iiitd.ac.in

Akshat Kumar
IIITD
akshat21230@iiitd.ac.in

Sareena Karapoola
IIT Madras
sareena.kp@gmail.com

Sambuddho Chakravarty
IIITD
sambuddho@iiitd.ac.in

ABSTRACT

Apart from becoming an essential part of our daily lives, mobile applications add to the threat landscape drastically. The difference between benign and malicious applications keeps blurring. Analyzing APKs dynamically becomes critical as APK developers employ techniques to evade static code inspections. However, these APKs can often bypass the test environment *e.g.,* in case of emulators. Also, there are several ways with which they can even detect instrumentation of the test environment and bypass them. Thus, we present COMEX, an Android testbed that modifies the test environment minimally with checks in place to prevent evasion. It has two parts – *AXMod* and *DCoP*. AXMod module is designed to perform a detailed dynamic analysis of an APK on real mobile devices. DCoP, a data collection pipeline, generates raw dynamic analysis data for a given set of APKs in a balanced and efficient manner. COMEX provides raw data related to all possible categories, such as system, network, and hardware in a time-stamped format. We analyzed approximately a thousand APKs including equal number of benign and malware APKs which gives us ≈72M benign and ≈70M malware system calls, ≈180k packets exchanged over network and ≈545k times files accessed and ≈30M total binder transactions. Finally, we publish the source code and analysis scripts to aid further studies.

CCS CONCEPTS

• **Security and privacy** → *Malware and its mitigation.*

ACM Reference Format:
Zeya Umayya, Dhruv Malik, Arpit Nandi, Akshat Kumar, Sareena Karapoola, and Sambuddho Chakravarty. 2024. COMEX: Deeply Observing Application Behavior on Real Android Devices. In *Workshop on Cyber Security Experimentation and Test (CSET 2024), August 13, 2024, Philadelphia, PA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3675741.3675745

1 INTRODUCTION

Android has the largest market share in mobile operating systems with over 3.5 billion users [66, 78]. According to the Statista 2023 [46] report, there are around 5.7 million applications in official play stores. No wonder the diversity of these users and devices also introduces large threat surfaces through propagation of malware. MalwareBytes's [56] annual reports indicate that millions of malware and potentially malicious applications infect Android devices every year. More than 100s of malware families reported by CIC datasets [37, 39–41], exploit different components and layers of Android devices to get their desired data. Some are detectable only by studying the features at the operating system level such as ransomware or adware [23, 62]. Others can be detected by inspecting network level features as they use network resources such as Android botnets [47], or utilization of hardware level resources such as CPU and battery power [29, 61].

An increase in threat surface also demands better understanding of the current Android applications (APKs). Malware and benign APKs should be studied in a balanced manner (*i.e.,* in equal numbers) to ensure that system-wide security frameworks like malware detectors are updated promptly. An APK can be studied either statically by only inspecting the decompiled code, or dynamically by executing it on a mobile device (or emulators). Static analysis has been extensively employed by commercial anti-virus engines, and prior research [19, 22, 53, 58, 76]. Malware, as well as benign, APK developers employ techniques such as obfuscation or encryption to bypass static analysis. Some malware categories download the main malicious code at runtime, while other decrypt the code before running. Eitherways, the decompiled APK reveals no meaningful information amenable for analysing its true behavior.

Dynamic analysis has been explored on both types of test environments, *viz.* emulator and real devices. Several testbeds are available as open-source tools, while rest are research proposals. MobSF [3], AndroPyTool [43], DroidBox [1] are examples of emulator-based solutions. Such solutions require instrumentation either at APK or at the system level, may reduces the chances of correctly executing the APK under test. They often evade analysis by monitoring the test environment employing various heuristics [71]. Further, unavailability and non-functionality of some testbeds render them ineffective [1, 11, 52, 71, 77, 84]. None of the current solutions perform thorough dynamic analysis and provide data at multiple levels *i.e.,* OS, network and hardware. Thus, to aid the development of better detection solutions and understanding the malware ecosystem, testbeds are required to successfully analyse benign as well as malicious APKs at various levels.

Thus we present COMEX a testbed for dynamic analysis with modular components, *viz. AXMod* and *DCoP*. AXMod module runs an APK and extracts raw analysis data on a mobile device. DCoP

*Shared first authors

CSET 2024, August 13, 2024, Philadelphia, PA, USA

ACM ISBN 979-8-4007-0957-9/24/08
https://doi.org/10.1145/3675741.3675745

is the main data collection pipeline that analyzes individual APKs using the AXMod module. COMEX has multiple properties – (1) It has been designed to analyse APKs dynamically, on real Android devices. The analysis produces raw data related to operating system, network and the underlying hardware. APKs' behaviour can be profiled by analysing data from all such levels. COMEX also considers users' inputs for, maximising APKs' behavioural data. (2) It does not require any type of instrumentation. Although the phones are rooted, we employ known tactics to ensure that root information is hidden from the APKs. Thus APKs would less likely evade the test environment in the absence of any obvious analysis artifacts. As a result, it maximizes the chances of complete APK executions on the testbed. (3) To make the testbed compatible with multiple device vendors and models, we leverage official Android tools like adb [13], perfetto [17], and Monkey [18] *etc.*

To maximize correct APK executions in COMEX , we set a default time of 60 sec, based on an empirical timing analysis of a set of APKs in two phases. For this analysis, we leveraged `ACVTool` [2] to measure the percentage code covered for each package of an APK. The first phase involved measuring the code coverage while increasing the execution time from 10 sec to 110 sec. We observed no significant increase in code coverage after 60 sec execution. In the second phase we included user inputs with varying rates. We observed that 15 user inputs/sec is suitable for maximizing the coverage. Thus we decided to set the default execution time to 60 sec with 15 user inputs/sec (details in Section 5.2).

We perform dynamic analysis of a thousand APKs with equal number of benign and malware samples. Malware APKs are spread across seven families from years 2019, 2020 and 2021. We downloaded the APKs from diverse sources such as AndroZoo [20] and VirusShare [80] and labeled these using VirusTotal [81] reports and AVClass tool [72]. We analyze all APKs in COMEX and perform a basic analysis of the raw data collected. We report 142M system calls in total, ≈180k packets exchanged over the network, ≈545k file accesses, and ≈30M binder transactions.

To summarize, the following are the contributions of our work:

(1) We provide COMEX an Android testbed for analysing applications. It does not require any type of instrumentation for dynamic analysis, and leverages mostly existing Android official tools, so that it works with future Android versions, besides preventing evasion by APKs. It has two crucial modules – *AXMod* and *DCoP*.
   (a) AXMod, a modular component, is designed to dynamically analyze individual APKs by running them for 60 sec, based on an informed experiment of code coverage.
   (b) DCoP, the master pipeline, is designed to start a data collection campaign in a balanced manner. It provides raw data related to OS, network and hardware sources while APK is executing.
(2) We provide a detailed execution time analysis of AXMod.
(3) We also perform basic analysis of the raw data collected. We analysed 1000 APKs in COMEX , and report the statistics in Section 6.1. Our source code is public on [31].

2 ANDROID APPLICATIONS AND THEIR ANALYSIS

2.1 Background

Android uses a modified Linux kernel. Its applications are written mostly in Java/Kotlin and have a packaged structure that include assets, AndroidManifest.xml and classes.dex *etc.* [14, 27]. Analysing such APKs by decompiling the codebase is known as static analysis. In general, APKs are analysed in two ways – *statically* and *dynamically*. Static analysis involves analysing the code base of applications (*i.e.,* APKs). This requires decompiling applications and thereby extracting the features. Tools such as JADX [49], Apktool [4] *etc.*, can be used for such purposes. Static analysis provides us features like permission present in manifest file, opcodes used and their frequencies, packages mentioned in activities and intent filters. However, static information is insufficient to track malicious behavior, like downloading other malicious executables, commonly seen in malware. Thus, for monitoring an application's behavior, dynamic analysis becomes essential. This involves executing the application on Android emulators or an actual phones to analyze their behavior. Emulators *MobSF* [3], *AndroPyTool* [43] *etc.*, are some commonly used for this.

Example: Static v/s Dynamic Analysis of Malware Samples: To elaborate, upon statically analysing Trojan Dropper 2018 sample using `AndroGuard` tool [12], we extracted maximum possible features such as permissions, packages, opcodes *etc.* We observed that it has permissions related to network usage like `ACCESS_FINE_LOCA-TION`, `ACCESS_NETWORK_STATE` and `ACCESS_WIFI_STATE`, among others. But at runtime, it downloads three files, one ELF library `libcom.art.roct.so` and 2 APKs (an adware and a benign file), as reported by VirusTotal [81]. Similar static and dynamic behaviour were observed for another two Trojan Droppers from 2019 and 2020. The 2019 trojan downloaded 1 ELF library, 1 APK, 1 XML, 1 zipped file and 264 other files. The 2020 trojan downloaded two malware APKs (one adware and a trojan).

2.2 Dynamic Analysis Challenges

Dynamic analysis involves capturing raw system (*e.g.*,, system calls and IPC in the form of binder transactions), network (number of network connections, bytes send or received *etc.*) and hardware-level (battery status and CPU usage *etc.*) data.

Often dynamic analysis relies on instrumentation at various levels to capture and checkpoint app and system behavior at various levels – APKs, Android framework (by *hooking* into the different libraries present), ART-level (by changing the way runtime behaves upon execution of an APK) and kernel-level (by adding new modules or changing the signatures of existing kernel functions). Solutions rely on diverse levels of instrumentation (see Table 1). All these create more opportunities for deeper levels of raw data extraction. But they may potentially also cause APKs to evade analysis by observing the underlying changes due to instrumentation [25, 32, 60, 71]. Also, apart from instrumentation, running binaries to collect raw data on an Android device requires rooting it. Either ways, there should be strategies to deceive the APKs about underlying changes.

			Instrumentation Level				Raw Data Collection Level									
Targeted On	Work	Year	APK	Framework	ART	Kernel	OS-syscall	OS-binder txn	Network	Hardware	Rooted	RootHiding	Functional/Available	Basic Analysis	User Input
Emulator	DroidBox	2014	●	○	○	○	●	○	○	○	●	○	◑	◑	○
	mobSF	2015	●	○	○	○	●	○	○	○	●	○	●	○	○
	Dynalog	2016	●	○	○	○	●	○	○	○	●	○	○	●	●
	AndroPyTool	2018	●	○	○	○	●	○	○	○	●	○	●	●	○
	InviSeal	2023	○	●	○	●	●	○	●	○	●	○	○	○	●
Hybrid	DroidDungeon	2024	○	○	●	●	○	○	○	○	●	●	○	○	○
Real Device	Malton	2017	○	○	●	○	●	○	◑	○	●	○	◑	●	○
	Odile	2022	●	○	●	○	○	◑	○	○	○	–	◑	○	○
	Simpson *et al.*	2023	○	○	○	○	○	○	●	○	○	–	○	●	●
	COMEX	2024	○	○	○	○	●	●	●	●	●	●	●	●	●

Table 1: An overview of Android Testbeds and comparison with COMEX across multiple parameters. Hybrid: targeted for both emulator as well as real devices; Solid fill : Applies to; Half fill : Applies to but non-functional; No fill : Does not apply to; Blue colour : Desired; Gray colour : Not desired.

3 COMEX VS OTHER TESTBEDS

3.1 Android Testbed

A testbed refers to a test environment set up with some data points/samples to be tested in it. In our case, real mobile devices are the test environment and Android APKs are the test data that need to be analysed. An emulator program[1] mimics the target device's hardware and software on a workstation/server. Emulators like MobSF and AndroPyTool work on regular desktop machines running any Unix-like/Windows OS. Emulators lack real hardware and thus often mimic them using the underlying host's hardware. In the absence of such mimetic virtual abstractions, APKs that depend on the real hardware may not work as intended. Thus, we argue that a testbed with real mobile phones is necessary to capture the actual behavior of an application, for analyzing it thoroughly.

3.2 Comparison

As presented in Table 1, existing work on APK analysis can be classified into three major categories – emulator-based, real device-based, and hybrid, *i.e.,* mix of emulation and real device. Table 1 compares different solutions across multiple parameters.

First comparison is based on the level of instrumentation required, *viz.* application, framework, Android RunTime (*i.e.,* ART) and kernel-level. We also highlight if prior works considered the raw data collected from multiple sources, *e.g.,* system-level, network or hardware. Finally, other parameters are related to the availability/functionality of the testbed, anti-evasion techniques (*e.g.,* root-hiding) employed, basic analysis of the raw data, and user inputs during APK execution. The testbed could be dysfunctional in case of unmaintained or incomplete code bases. The presence of anti-evasion checks are important to ensure that the APKs do not detect that they are being monitored, and bypass the analysis. Basic analysis refers to the raw data processing to extract meaningful information such as frequency of system calls or data sent over network from the log files. The last parameter highlights if the underlying testbed considers any type of automated user inputs to the APK so as to capture as much functionality of the APK as possible.

3.2.1 Emulator based works. In this category, all proposals require at least one type of instrumentation. Most propose instrumenting the APK themselves. Application developers usually employ tamper checks that may prevent instrumented APKs from running in the emulators [71]. Most proposals involve capturing system call traces, *i.e.,* only OS level data, except for *InviSeal* [52] which collects network traffic as well. Although emulators are by default rooted, none of the solutions employ techniques to hide this information from the APKs under test. Thus, this further reduces the number of successful APK executions in the analysis environment. In our case, although the device is rooted, but we take proper measures to hide this information from the APK (see details in Section 4.1).

3.2.2 Real-Device based Work. *DroidDungeon* [71] is the only work that provides a solution applicable to both emulators, as well as real devices. Unfortunately, their sandbox is not publicly available. Further, they do not provide any means of user input, and nor is any raw data collected. *Malton* [84] instruments the ART and only provides syscalls as raw data. It does not provide any means to hide the root. Although its code is available it is not functional [55]. *Odile* [77] is another such solution which is non-functional due to missing pieces of code from its repository [2].

Although emulator-based solutions scale better, compared to real-device solutions, the evasion techniques employed by APKs (benign as well malware) make these solutions less useful. There are potential research efforts [52, 71] to make emulators pose like real-devices. Among other drawbacks, the most important is their unavailability (see rows fifth and sixth in Table 1).

Overall, Table 1 clearly shows that our solution COMEX does not require any type of instrumentation unlike [1, 3, 11, 43, 52, 63,

[1]Simulator: these let you run programs that were not made for your computers OS. Till date, only iOS has simulator but not Android.

[2]Even after recreating most of its code, it failed to instrument APKs even for less than 3MB.

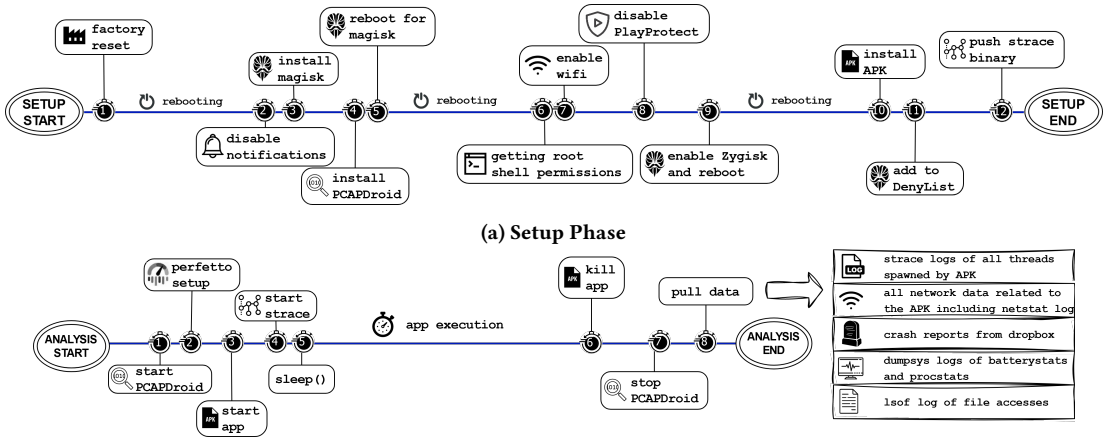


Figure 1: AXMod - Analysis Execution Module for Raw Data. Follow the numbered clocks from left. The distance between each clock is a representative of the approx time-difference between the two operations.

71, 77, 84] and it provides maximum raw data along with its basic analysis unlike others [3, 52, 71, 75, 77].

4 COMEX DESIGN DETAILS

Overview: We divide the COMEX design in two parts. The first one is an individual module, named as *AXMod*, responsible for APK analysis. The second one is responsible for collecting the per APK analysis data in a balanced manner, named as *DCoP*. Figure 1 schematically shows first part, *i.e.,* step by step process of the analysis execution module. Since this module is applied to every APK under test to get the raw data, it is the heart of the testbed. We use this module to launch a balanced data collection pipeline. We explain this pipeline in Section 4.2 with the help of Figure 2.

4.1 Analysis Execution Module (AXMod)

COMEX runs fully automated. It uses existing tools provided by the *Android Open Source Project (AOSP)* [14] so that it should work with diverse Android devices. All the operations are performed using *Android Debug Bridge (adb)* [13]. User inputs are provided using screen automation , using the `monkeyrunner` [16] tool. Figure 1a presents the setup phase of the AXMod. In the setup phase, the device is prepared for executing an APK. The figure shows the important operations performed on a timeline. The separation between two consecutive operations is proportional to the time it takes to complete the first among them (see Section 6.2).

4.1.1 Setup Phase. To have the baseline device state, it is factory reset everytime a new APK is to be analyzed ①, to wipe all user data and settings, restoring the device to its default configuration. Next, we disable the pop-up notifications and root the device using magisk APK [51] ②③. We install PCAPDroid APK [36] to capture network traffic specific to the APK under test by providing the package name of the APK④. Next, the device is rebooted ⑤ to stabilize the magisk rooting and obtain permanent root permissions ⑥. Further, wifi is enabled and the device connects to a dedicated Internet connection procured specifically for malware testing ⑦. We follow all ethical practices to ensure the safety of nearby devices (see details in Section 7.3). Google PlayProtect [44] is also disabled to bypass google's in-place security system that does not allow unwanted applications to run on the device. This suppresses pop-ups while installing APKs ⑧. Further, we leverage Zygisk's [54] DenyList feature to hide the root information from the APKs under test, to prevent evasion from the test environment ⑨. To stabilize this change, device is rebooted again. Using Ruggia *et al.*'s AntiAnalysisProofSample [5], we cross-validated that root is hidden. Finally, towards the end of setup phase, APK (under analysis) is installed ⑩ and strace [30] binary is pushed to the device for capturing system calls ⑫.

4.1.2 Analysis Phase. Once the device is set-up, analysis begins by starting the PCAPdroid ①, perfetto ②, strace ③ and AUT④. We keep the analysis running for 60sec as observed from the code coverage experiment (for details see Section 5) ⑤. At the end of analysis, all operations are stopped ⑥ ⑦ and raw data is extracted from the device and saved to local storage ⑧. The device resumes the setup phase for the next APK to be tested. We describe the data collected from the analysis in Section 6.1.2.

4.2 Data Collection Pipeline (DCoP)

The rapid growth in the variety of APKs and their samples makes it necessary for security researchers to catch up with their timely and fast analysis. Using DCoP, we launch the data collection campaign for both benign and malicious applications as shown in Figure 2. Our data collection pipeline has two properties. ***First***, it should optimize the devices present in the testbed. ***Second***, it should process APKs in a balanced manner, *i.e.,* the same number of malicious and benign APKs, at any given point in time. The number of malicious and benign APKs should be same with respect to the year. Furthermore, the number of family samples, for malware, should be same. Generally, dynamic analysis on real devices is time-consuming. It may take weeks to months for a dataset of roughly 60k APKs (30k benign and 30k malware). Hence starting with an even ratio helps

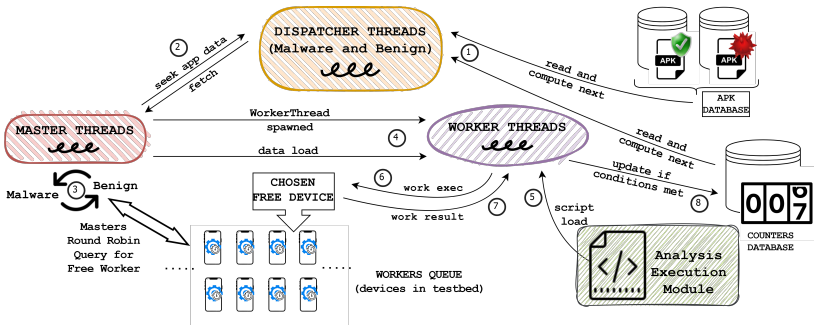


Figure 2: DCoP - Data Collection Pipeline.

where system-wide security frameworks like malware detectors are to be upgraded periodically.

To ensure the two aforementioned properties, we designed the data collection pipeline with multiple components working in sync. As shown in Figure 2, it includes three types of threads, one execution module *i.e.,* AXMod, APK database, essential counters and the physical devices, referred to as *workers*.

Work-flow: As shown in Figure 2 ①, we prepare the `APK DATABASE` with benign and malware applications (see Figure 6.1 for details) using our APK storage server. Based on the distribution of the samples in the `APK DATABASE`, another one, *viz.* `COUNTERS DATABSE`, is created. It tracks the successfully analysed APKs along with their other meta data such as APK category, family and year. It is helpful in check-pointing and restarting the pipeline whenever required.

Dispatcher threads are responsible for fetching the appropriate in-sequence APK to prepare it for master threads to work on. Thus, dispatcher threads are always ready with APKs for distribution. Dispatcher threads are also decide when the execution of the entire pipeline should halt (when the counters meet their target). In ②, dispatcher hands over the APK to master threads.

Then, the master threads for malware and benign are spawned. In ③, master threads assign the APK information received from dispatcher to an available device. There are two types of master threads namely `malware_thread` and `benign_thread`. These threads work in round-robin manner to assign APK executions to available free workers. The purpose of such execution is to ensure synchronization in the number of data points generated for malware and benign APKs at any point of time.

In ④, the master thread spawns a worker thread with an available device and loads the necessary parameters for APK data and the device info to the thread and takes the device off the free workers list. In ⑤ and ⑥, the worker thread then fetches the APK from the APK storage server, and executes the dynamic analysis script *i.e.,* AXMod (in green colour in Figure 2) on the assigned device. In ⑦, the raw data is extracted from the devices. In ⑧, worker thread checks whether the dynamic analysis data is received correctly or not, and then increases specific counter in the `COUNTERS DATABSE`, after which it returns the device to the free `WORKERS QUEUE`. Dispatchers keep track of the `COUNTERS DATABASE` and stop their execution when the counters reach their maximum values, or encounters any other possible termination condition[3], like when all APKs have been processed, is met. This results in a graceful termination of all operations.

5 APK CODE COVERAGE ANALYSIS

COMEX 's analysis phase (*i.e.,* Figure 1b) involves multiple execution steps, including the start of APK execution, followed by its termination, and raw data collection. The APK execution time is a very critical parameter of the whole testbed. To decide the experiment duration, we design some experiments and propose a time duration for Android APKs to be analysed in the test environment.

Purpose of the experiment: Dynamic analysis of an APK is performed on a device and thus is a resource consuming experiment. To maximize the utilization of device and to let an APK work fully as intenteded, we need an empirically deduced *apt* time duration. To measure the behaviour, we relied on *ACVTool* [2]. It measures the percentage of code traversed per package of the APK, during its execution. APKs have two varieties of packages, *viz.* official Android packages and third-party ones, that APK developers includes. ACVTool instruments APKs by adding code in their `smali` bytecode. The added instrumented code helps the tool to determine which methods, classes and packages have been executed and their respective percentages. To achieve this, it first decompiles the APK using apktool [4] and instruments third party and main package. Further, Android packages are instrumented using AAPT [68]. Finally, it repackages the APK and installs it on the test device. At the end of its execution it generates log files with percentage values in it. We call this experiment as code coverage analysis of APKs.

We performed code coverage analysis of 500 Android APKs *i.e.,* a mix of benign as well as malware using ACVTool [2][4]. The analysis can be divided in two parts. The first corresponds to all the experiments without providing any external user input to the APK(or GUI). And, the second corresponds to the experiments where we provide user inputs in some form to the APK.

5.1 Without User Input

To perform the experiment, we selected a set of APKs from four malware categories (*e.g.,* trojan, PUA, scareware, and adware), and applied `ACVTool` over them and executed each APK on a device for different duration, starting from 10sec up to 110sec, increasing by an interval of 10sec, every time. Each APK was executed 11 times.

[3]In case of any crashes, COMEX will restart and start processing APKs from where it stopped previously.

[4]Initially we selected 1k APKs, but due to several errors related to instrumentation and app crashes almost half of the APKs failed in this experiment [6].

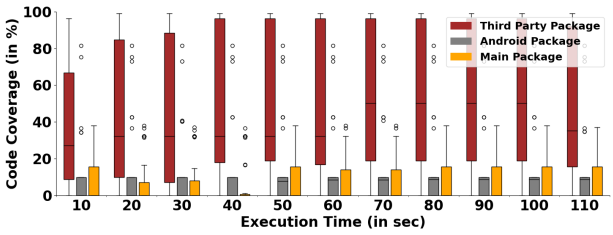


Figure 3: Code coverage of APKs without any user input.

Thus, total number of successful executions were 5.5k on the device. We present the results in Figure 3.

We show the coverage with respect to the specific package type, in an APK. X-axis shows the execution time in seconds, and Y-axis shows the percentage coverage of the respective package type. We observe that third party packages have the highest code covered among other type of packages. The observation is consistent across all APKs and concordant with prior research [85, 86, 88]. Main package has the lowest coverage. One potential reason could be that APK developers highly utilize and rely on third-party Android packages, due to the availability of pre-built solutions for common tasks. After 60sec there is little to no change in the code coverage of all package types. Thus, we selected 60sec to be the optimal time for dynamic analysis.

5.2 With User Inputs

We repeated the experiments by providing user inputs to each APK, using *Monkey* tool [18]. It is the official tool provided by the Android open source project (AOSP) for user inputs to an APK. There are alternatives such as ARES [70]. Although ARES showed increased coverage in comparison to Monkey, reported 48% APK crashes, as compared to only 11% for Monkey. Thus, to maximize APK executions, we preferred Monkey over ARES.

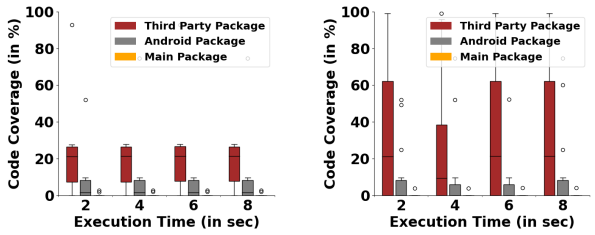
Using Monkey, we executed each APK for four separate durations *i.e.,* 2, 4, 6 and 8sec, with four user input rate variations *i.e.,* 5, 10, 15 and 20 per sec[5]. We present the analysis results in Figure 4. We observed that code coverage went up when inputs increased from 5 to 15 inputs/sec, but not significantly more when the user input was set to 20/sec. In Figure 4, we only show the significant graphs for 5 (Figure 4a) and 15 inputs/sec Figure 4b. For 15 inputs/sec, the total number of inputs goes upto 900 a minute. We selected this to be default number of inputs for our analysis. One may suggest 10sec to be sufficient based on Figure 4b, since it increases the third party coverage upto 60%. But to get the maximum possible percentage of all package types, we chose set the default time to be 60sec. Thus, we selected the default APK analysis time to be 60sec with 15 user inputs/sec[6].

6 TESTBED EVALUATION

6.1 Variety of Data Collection

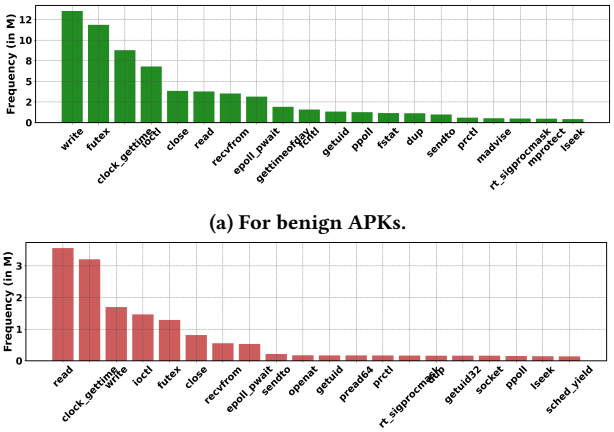
Using COMEX , we collected data for one thousand APKs.

[5]In Monkey, we kept the seed value same for each APK, to generate the same set of events during each run. Since Monkey clusters events and then spreads them across the total analysis duration, a set of five inputs may take more than one sec.
[6]AXMod has provisions for changing these parameter values



(a) At 5 user inputs/sec. **(b) At 15 user inputs/sec.**

Figure 4: Code coverage of APKs with user input.



(a) For benign APKs.

(b) For malicious APKs (family - `jiagu`).

Figure 5: Frequency distribution of top 20 system calls.

6.1.1 APKs' Description. We downloaded APKs for three different years *i.e.,* 2019, 2020 and 2021 from AndroZoo [10, 20]. APKs include equal samples for both benign as well as malware applications (500 each). We follow a two step process to label the malware APKs with their family names. Firstly, we submit each APK hash to VirusTotal [81], an online service with 60+ antivirus engines for analyzing files/URLs and producing a combined analysis report in JSON format. Secondly, these JSON reports are fed to the *AVClass2* [72] tool, that tags them corresponding to their respective family names. Thus, at the end of this labelling process, we have malware APKs spread across 7 families such as `jiagu`, `smsreg`, and `tencentprotect`. We started the data collection campaign for all the APKs mentioned here as per Figure 2, with six mobile devices. The whole campaign was executed, and at the end we analysed the raw data from each of these phones. We provide per APK processing time and its individual operations time analysis in Section 6.2.

6.1.2 Data Categories and Basic Analysis. The testbed has been designed to produce data of different categories – such as OS, network and hardware related data. We employ different methods to extract data for each category.Figure 1b shows the different logs collected at the end of analysis.

6.1.3 OS related data. We collect OS related data from `strace` logs, dropbox logs, and `perfetto` logs. This includes system calls, binder transactions and list of files accessed during an APK's execution *etc.*. The raw data for system calls contain detailed information about the system call along with its arguments values and return

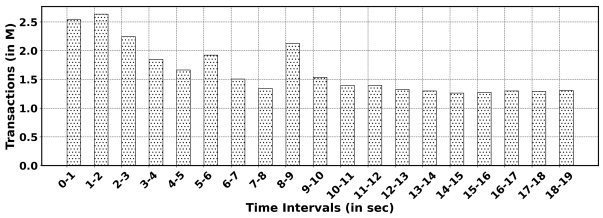
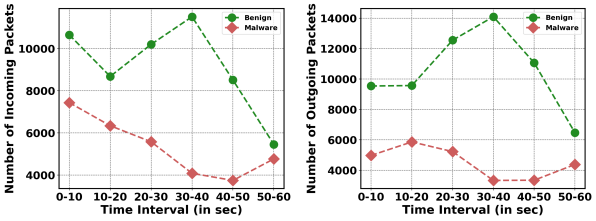


Figure 6: Frequency of binder transactions with time.



(a) Incoming Packets. **(b) Outgoing Packets.**

Figure 7: Packets exchanged over network with time.

values. We observed that while execution an APK forks multiple processes as well. We make sure to follow those forks and store separately for performing interesting analysis and obtain insights of APK working. It can be utilised in different types of analysis such as making provenance graphs [82] or semantically deriving the API calls, similar to how syscall2api [64] works. We perform a basic analysis of the `strace` logs and generate a list of system calls along with their frequency and time spent executing. Figure 5 shows the distribution of top 20 system calls along with their frequency. In total, we report 72M system calls in benign and 70M system calls in malware APKs.Figure 5b and 8 highlight that the distribution varies across malware families.

We also process `perfetto` logs and convert these into 10 different files having features such as client_tid, binder_reply_id *etc.* and saving the analysed data in tabular format. All the events in the data are timestamped. Figure 6 shows the frequency of binder transactions in first 20 seconds.

6.1.4 Network related Data. We collect network related `data` in a `pcap` format using `PCAPDroid` APK by specifying the package name of the APK under analysis. We analyse these `pcaps` using `Argus` tool [21][7]. It provides 40+ features namely `SrcId`, `SrcPkts`, `DstPkts`, `SrcBytes`, `DstBytes`, `SAppBytes`, and `DAppBytes` *etc.* [79]. In Figure 7, we present the number of bytes send and received over network in increasing time.

6.1.5 Hardware related data. We use the existing Android utility `dumpsys` [15] to collect battery status and CPU usage.

6.2 Time Analysis of COMEX

We provide a detailed time-analysis of the two phases of AXMod Figure 1). We select two APKs of sizes 10MB and 24MB, and perform their dynamic analysis on our testbed. We repeat the experiment multiple times and calculate the averages for each step. Table 2 shows that setup phase (Figure 1a) takes an average of 5.25 min

[7]CICFlowMeter [7] can also be used to extract network level features.

Setup Time (in sec)			Analysis Time (in sec)		
Steps	**10MB**	**24MB**	**Steps**	**10MB**	**24MB**
1	122.16	120.17	1	14.41	15.10
2	1.92	1.69	2	0.002	0.003
3 & 4	14.08	15.33	3	4.76	3.29
5	64.33	64.01	4	0.90	0.68
6	3.39	3.27	5	6.21	6.15
7	3.25	3.13	6	65.43	64.68
8	14.27	14.21	7	74.31	79.05
9	65.95	66.40	8	3.16	3.98
10	6.06	7.12	9	6.30	6.23
11	19.44	19.48	10	2.30	3.95
12	0.15	0.17	11	1.00	1.21
Total	315.01	314.98	Total	178.79	184.32

Table 2: COMEX - AXMod Time Analysis. Here "Steps" refer to the steps of setup (Figure 1a) and analysis phase (Figure 1b).

Data Source	Possible extraction of raw data	Enhancement to diff works
System	System calls, binder transactions, files accessed, crashes *etc.*	Zhang *et al.* [87] Mercaldo *et al.* [59] DroidScribe [33], Andromaly [73], Chimera [34], EavesDroid [83]
Network	SrcPkts, DstPkts, SrcBytes, DstBytes, SAppBytes, DAppBytes, *etc.*	Fu *et al.* [42], Heldroid [19] DroidScribe [33] Dine *et al.* [35], Andromaly [73], Hybroid [65]
Hardware	CPU utilization, battery status, *etc.*	PowerSpy [61], Andromaly [73], Cabral *et al.* [29]

Table 3: Different sources of raw data and respective applicable work.

and analysis phase (Figure 1b) takes on average 1.82 min. In the setup phase, 1st, 5th and 9th steps take ≈ 80% of the overall time (*i.e.,* factory reset, magisk and zygisk reboot.) In analysis phase, 7th and 8th step takes ≈ 78% of the overall time (*i.e.,* transfer of data from device to local storage).

6.3 Use-cases of the COMEX Data

6.3.1 Malware Detection. As presented in Table 3, malware detection approaches utilize mostly system and network level information based on the malware type of interest. For example, Zhang *et al.* [87] used only system call traces to detect malware using merely 100 benign and 325 malware samples. This work can be recreated for thousands of such samples when using COMEX . Similarly, Fu *et al.* [42] used network traffic to detect botnet malware using CICInvesAndMal-2019 dataset [38]. The dataset is very small as it contains only 5k benign and 426 malware samples. Using COMEX , the same study can be expanded to a large variety of malware as well as benign samples.

6.3.2 Malware C&C infrastructure Exploration. Malware APKs usually communicate with their command and control servers (C&C) to get a list of commands for execution, which varies from exfiltrating user data, infecting other devices to terminating their own execution [9, 28]. Such communication can be observed using our testbed. Similar pioneering work has been done for other OS such as Windows and Linux. COMEX would enable the replication of many studies such as Botminer [45], JACKSTRAWS [48], Disclosure [26], Malalert [67], and BOTection [8] for better understanding of the malware C&C infrastructure in Android.

6.4 Implementation Details

At present, we deployed COMEX using six `Motorola G40 Fusion` devices by connecting them to a host system having 32GB RAM and Intel Core-i5 processor with Ubuntu `v20.04.6` operating system (linux kernel `v5.15.0-105-generic`). The mobile devices have Android `v12` with 6GB RAM, 128GB ROM, kernel `v4.14.190-perf+` having dimensions as 2460x1080 pixels. COMEX can accommodate several more Android devices from varied vendors.

To root the devices we used `magisk v26.3` [51]. For system call tracing, we used `strace v4.23` from `strace.io` [30]. For network related data, we used `PCAPDroid v1.6.8`. To develop the analysis scripts we used python `v3.8.10`. For screen automation we use `monkeyrunner` binary. For it to work we downloaded Java `v1.8.0_402` and Jython `v2.5.3`. For per APK analysis, all the binaries need the package name of the APK, which we extract using a lightweight tool `AAPT v0.2-27.0.1` (android asset packaging tool). It is present in Android SDK `v26.1.1`. For code coverage experiments we used ACVTool `v0.2` [2]. For user inputs we used the Monkey tool [18] provided by Android officially.

7 DISCUSSION

7.1 Limitations and Challenges

• **API-level data extraction:** Collecting API level information during execution requires APK level instrumentation, which is not a desired feature of COMEX . Often APK level instrumentation is detected by the APK itself and results in evasion of the testing by it. To avoid instrumentation, solutions like `syscall2api` [64] can be explored. These solutions can be added as a part of the data analysis and does not need any changes to main design of COMEX .

• **High Setup Time:** As presented in Section 6.2, setup phase takes the maximum time in AXMod. It is related to one factory reset and two additional reboots for stabilizing the rooting process. These three steps are necessary for the AXMod to run smoothly.

• **Dependency of monkeyrunner:** `Monkeyrunner` is used for screen automation in AXMod. It involves steps such as approving root prompts from magisk APK, hidhing root using zygisk, starting pcap collection and disabling Play Protect. `Monkeyrunner` requires screen coordinates for these tasks. Thus, for a device with different screen dimension, coordinates can be fixed with a one time effort. After that, COMEX will work smoothly.

• **Scalability:** In order to make COMEX scalable for larger datasets, more resources such as mobile phones and system resource such as storage are required.

7.2 Future Work

• **Extension to iOS:** COMEX is designed only for the Android platform. To extend COMEX to other mobile operating systems *e.g.,* iOS, the AXMod part (Figure 1) needs to be modified with respect to the data extraction from different levels. *E.g.*, network level information can be collected by using applications like `Http traffic capture` [74] or `tcpdump` [57]. Frida [69] based `iOS tracer` [50] can be used to collect information on system calls. Thus, our solution can be modified to make it suitable for iOS applications.

• **Application micro-benchmarking:** In its current form, COMEX has features for collecting data at various levels. Different micro-benchmarking steps can be augmented to AXMod and then collected data using the pipeline can be used for individual application testing.

• **Testing against various mobile vendors:** We have tested on `Motorola G40 Fusion` device with Android `v12`. In future, we plan to expand it to different vendors as well as to different Android versions such as `v13` or `v14`.

7.3 Ethics

Analysing malware APKs on mobile phones require live installations and monitoring of the network traffic corresponding to its communication with command and control (C&C) servers. Thus we carefully planned our experiments using Belmont's report [24].
(1) Respect for persons: Our experiments involve analyzing applications running on mobile phones. We take utmost care to not infect other persons devices. We isolated the phones from our network so that the malware discover none (vulnerable) to infect. The phones have been procured commercially, and are dedicated to our research. Neither the researchers nor any 3rd parties have used them for their personal/non-personal communication. This ensured that no users' data could potentially exfiltrate to third-party servers.
(2) Beneficence: COMEX exhaustively analyses Android APKs and produces analysed raw data in the form of important log files. We envision that the proposed testbed's outputs *i.e.,* logs, can be used in multiple use-cases such as malware detection and application benchmarking. For example, malware detectors developed using COMEX would be beneficial to the to the scientific community and people at large. Thus the low-levels of risks (if at all) are compensated by the greater safety and security of individuals.
(3) Justice: As mentioned, we take utmost care to safeguard oblivious devices present in our institution from malware infection. These malware were not used for any other academic, financial or political pursuits. The purpose of this research is solely to design an efficient testbed for analysing Android APKs. To minimize the risk, we did not leave any malware running for too long so as to minimize the levels of exposure of sensitive users' data.
We submitted a detailed IRB proposal to the institute's board. However, we were exempted from it since as per our institute's regulations, studies with no human subject involvement do not require the approval.

8 CONCLUSION

In this work, we propose COMEX an Android testbed with real devices. COMEX does not require any instrumentation and thus suitable for analyzing APKs that often tend to bypass the underlying test environment. It has two main parts – AXMod and DCoP. AXMod is the heart of the testbed. It works on each device individually in its own virtual machine and dynamically analyzes of APKs on the device, in an automated manner. AXMod consists of two parts, setup and analysis. We performed the time analysis of AXMod. We also determine how long an APK should run so as to capture all its behavior. By analysing 500 benign and malicious APKs of various categories, we empirically concluded that 60 sec with 15 user inputs/sec, is sufficient for this purpose, Finally, using COMEX we analysed a thousand APKs from 2019-21, spanning across seven families which resulted in ≈142M system calls, ≈545k file accesses, ≈30M binder transactions, ≈180k network packets in total.

REFERENCES

[1] 2014. DroidBox. https://github.com/pjlantz/droidbox.
[2] 2020. ACVTool. https://github.com/pilgun/acvtool.
[3] 2020. Mobile-Security-Framework-MobSF. https://github.com/MobSF/Mobile-Security-Framework-MobSF.
[4] 2023. apktool. https://ibotpeaches.github.io/Apktool/.
[5] AAl-Khaser. 2024. AAl-Khaser - Android Al-Khaser. https://github.com/eurecom-s3/AAl-Khaser/tree/main/AntiAnalysisProofSample.
[6] acvtool. 2024. ACVTool Issues. https://github.com/pilgun/acvtool/issues.
[7] Canadian Institute for Cybersecurity Ahlashkari. 2021. CICFlowMeter. https://github.com/ahlashkari/CICFlowMeter/tree/master.
[8] Bushra A Alahmadi, Enrico Mariconti, Riccardo Spolaor, Gianluca Stringhini, and Ivan Martinovic. 2020. BOTection: Bot detection by building Markov Chain models of bots network behavior. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 652–664.
[9] Hui Wang Alex.Turing. 2020. The LeetHozer botnet. https://blog.netlab.360.com/the-leethozer-botnet-en/.
[10] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th international conference on mining software repositories*. 468–471.
[11] Mohammed K Alzaylaee, Suleiman Y Yerima, and Sakir Sezer. 2016. DynaLog: An automated dynamic analysis framework for characterizing android applications. In *2016 International Conference On Cyber Security And Protection Of Digital Services (Cyber Security)*. IEEE, 1–8.
[12] AndroGuard. 2023. Androguard Tool. https://androguard.readthedocs.io/en/latest/index.html.
[13] Android. 2024. Android Debug Bridge (adb). https://developer.android.com/tools/adb.
[14] Android. 2024. Android Open Source Project. https://source.android.com/.
[15] Android. 2024. Dumpsys. https://developer.android.com/tools/dumpsys.
[16] Android. 2024. monkeyrunner. https://developer.android.com/studio/test/monkeyrunner.
[17] Android. 2024. Perfetto. https://perfetto.dev/.
[18] Android. 2024. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/other-testing-tools/monkey.
[19] Nicoló Andronio, Stefano Zanero, and Federico Maggi. 2015. Heldroid: Dissecting and detecting mobile ransomware. In *Research in Attacks, Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015. Proceedings 18*. Springer, 382–404.
[20] AndroZoo. 2016. AndroZoo. https://androzoo.uni.lu/.
[21] Argus. 2024. Argus. https://openargus.org/getting-argus.
[22] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. Drebin: Effective and explainable detection of android malware in your pocket.. In *Ndss*, Vol. 14. 23–26.
[23] Avast. 2020. How to Remove Ransomware from Android Devices. https://www.avast.com/c-how-to-remove-ransomware-android.
[24] Tom L Beauchamp. 2008. The Belmont Report. *The Oxford textbook of clinical research ethics* (2008), 149–155.
[25] Stefano Berlato and Mariano Ceccato. 2020. A large-scale study on the adoption of anti-debugging and anti-tampering protections in android apps. *Journal of Information Security and Applications* 52 (2020), 102463.
[26] Leyla Bilge, Davide Balzarotti, William Robertson, Engin Kirda, and Christopher Kruegel. 2012. Disclosure: detecting botnet command and control servers through large-scale netflow analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference*. 129–138.
[27] Appdome Blog. 2024. Structure of an Android App Binary. https://www.appdome.com/how-to/devsecops-automation-mobile-cicd/appdome-basics/structure-of-an-android-app-binary-apk/.
[28] ZDNet Blog by Catalin Cimpanu. 2021. Android devices ensnared in DDoS botnet. https://www.zdnet.com/article/android-devices-ensnared-in-ddos-botnet/.
[29] Reeve Cabral, J Todd McDonald, Lee M Hively, and Ryan G Benton. 2022. Profiling cpu behavior for detection of android ransomware. In *SoutheastCon 2022*. IEEE, 690–697.
[30] Vitaly Chaykovsky. 2024. strace-linux syscall tracer. https://strace.io/.
[31] COMEX. 2024. Code Repository. https://github.com/zeya2u9/COMEX.
[32] Cryptomathic. 2022. Why Emulator Detection is Essential for Mobile Banking and Payment App Security. https://irdeto.com/denuvo/mobile-games-protection/.
[33] Santanu Kumar Dash, Guillermo Suarez-Tangil, Salahuddin Khan, Kimberly Tam, Mansour Ahmadi, Johannes Kinder, and Lorenzo Cavallaro. 2016. Droidscribe: Classifying android malware based on runtime behavior. In *2016 IEEE Security and Privacy Workshops (SPW)*. IEEE, 252–261.
[34] Angelo Schranko de Oliveira and Renato José Sassi. 2023. Chimera: an android malware detection method based on multimodal deep learning and hybrid analysis. *Authorea Preprints* (2023).
[35] Chao Ding, Nurbol Luktarhan, Bei Lu, and Wenhui Zhang. 2021. A hybrid analysis-based approach to android malware family classification. *Entropy* 23, 8 (2021), 1009.
[36] Emanuele Faranda. 2024. PCAPdroid - network monitor. https://play.google.com/store/apps/details?id=com.emanuelef.remote_capture&hl=en&gl=US&pli=1.
[37] Canadian Institute for Cybersecurity. 2017. CIC-AndMal2017. https://www.unb.ca/cic/datasets/andmal2017.html.
[38] Canadian Institute for Cybersecurity. 2019. CIC-InvesAndMal2019. https://www.unb.ca/cic/datasets/invesandmal2019.html.
[39] Canadian Institute for Cybersecurity. 2020. CCCS-CIC-AndMal-2020. https://www.unb.ca/cic/datasets/andmal2020.html.
[40] Canadian Institute for Cybersecurity. 2020. CIC-AndMal2020. https://www.unb.ca/cic/datasets/andmal2020.html.
[41] Canadian Institute for Cybersecurity. 2020. CIC-MalDroid2020. https://www.unb.ca/cic/datasets/maldroid-2020.html.
[42] Zhuoqun Fu, Mingxuan Liu, Yue Qin, Jia Zhang, Yuan Zou, Qilei Yin, Qi Li, and Haixin Duan. 2022. Encrypted malware traffic detection via graph-based network analysis. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*. 495–509.
[43] Alejandro Martín García and fabriziopicconi. 2018. AndroPyTool. https://github.com/alexMyG/AndroPyTool.
[44] Google. 2024. Google Play Help. https://developers.google.com/android/play-protect/client-protections.
[45] Guofei Gu, Roberto Perdisci, Junjie Zhang, and Wenke Lee. 2008. Botminer: Clustering analysis of network traffic for protocol-and structure-independent botnet detection. (2008).
[46] RipenApps Blog By Ishan Gupta. 2023. Mobile App Industry Statistics 2023. https://www.bankmycell.com/blog/how-many-android-users-are-there.
[47] TechTarget Blog By Katie Terrell Hanna. 2024. botnet. https://www.techtarget.com/searchsecurity/definition/botnet.
[48] Gregoire Jacob, Ralf Hund, Christopher Kruegel, and Thorsten Holz. 2011. {JACKSTRAWS}: Picking Command and Control Connections from Bot Traffic. In *20th USENIX Security Symposium (USENIX Security 11)*.
[49] JADX. 2023. Dex to Java decompiler. https://github.com/skylot/jadx.
[50] Juliangrtz. 2024. frida-iOS-syscall-tracer. https://github.com/juliangrtz/frida-iOS-syscall-tracer.
[51] Sarang Khilare. 2023. Rooting Android Device: Magisk. https://medium.com/@sarang6489/rooting-android-device-magisk-72e05793a1fb.
[52] Saurabh Kumar, Debadatta Mishra, Biswabandan Panda, and Sandeep Kumar Shukla. 2023. InviSeal: A Stealthy Dynamic Analysis Framework for Android Systems. *Digital Threats: Research and Practice* 4, 1 (2023), 1–31.
[53] Junjie Liang, Wenbo Guo, Tongbo Luo, Honavar Vasant, Gang Wang, and Xinyu Xing. 2021. Fare: Enabling fine-grained attack categorization under low-quality labeled data. In *Proceedings of The Network and Distributed System Security Symposium (NDSS)*.
[54] MagiskModule. 2024. Zygisk. https://www.magiskmodule.com/install-zygisk-and-download-best-zygisk-modules-in-2023/.
[55] Malton. 2022. Github Issues in Malton. https://github.com/rewhy/malton/issues.
[56] MalwareBytes. 2023. Crushing the two biggest threats to mobile endpoint security in 2023. https://www.malwarebytes.com/blog/business/2023/03/crushing-the-two-biggest-threats-to-mobile-endpoint\-security-in-2023.
[57] TCPDUMP manpage. 2024. TCPDUMP. https://www.tcpdump.org/manpages/tcpdump.1.html.
[58] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. 2017. Mamadroid: Detecting android malware by building markov chains of behavioral models. In *NDSS*.
[59] Francesco Mercaldo, Fabio Martinelli, Antonella Santone, et al. 2023. An Explainable Convolutional Neural Network for Dynamic Android Malware Detection. In *International Conference on Information Systems Security and Privacy*. Science and Technology Publications, Lda, 305–312.
[60] Alessio Merlo, Antonio Ruggia, Luigi Sciolla, and Luca Verderame. 2021. You shall not repackage! demystifying anti-repackaging on android. *Computers & Security* 103 (2021), 102181.
[61] Yan Michalevsky, Aaron Schulman, Gunaa Arumugam Veerapandian, Dan Boneh, and Gabi Nakibly. 2015. {PowerSpy}: Location Tracking Using Mobile Device Power Analysis. In *24th USENIX Security Symposium (USENIX Security 15)*. 785–800.
[62] Mlwarebytes. 2021. How to remove adware on an Android phone. https://www.malwarebytes.com/blog/news/2021/11/how-to-remove-adware-on-an-android-phone.
[63] Ali Muzaffar, Hani Ragab Hassen, Hind Zantout, and Michael A Lones. 2023. DroidDissector: A Static and Dynamic Analysis Tool for Android Malware Detection. In *International Conference on Applied CyberSecurity*. Springer, 3–9.
[64] Dario Nisi, Antonio Bianchi, and Yanick Fratantonio. 2019. Exploring {Syscall-Based} Semantics Reconstruction of Android Applications. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 517–531.
[65] Mohammad Reza Norouzian, Peng Xu, Claudia Eckert, and Apostolis Zarras. 2021. Hybroid: Toward Android Malware Detection and Categorization with Program Code and Network Traffic. In *International Conference on Information Security*. Springer, 259–278.

[66] Twinr Blog By Gaurav Parvadiya. 2024. A Deep Dive into Android vs iOS Market Dynamics. https://twinr.dev/blogs/android-vs-ios-market-dynamics/.

[67] Michal Piskozub, Riccardo Spolaor, and Ivan Martinovic. 2019. Malalert: Detecting malware in large-scale network traffic using statistical features. *ACM SIGMETRICS Performance Evaluation Review* 46, 3 (2019), 151–154.

[68] Android Open Source Project. 2024. https://developer.android.com/tools/aapt2.

[69] Ole André Vadla Ravnås. 2024. FRIDA. https://frida.re/.

[70] Andrea Romdhana, Alessio Merlo, Mariano Ceccato, and Paolo Tonella. 2022. Deep reinforcement learning for black-box testing of android apps. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 4 (2022), 1–29.

[71] Antonio Ruggia, Dario Nisi, Savino Dambra, Alessio Merlo, Davide Balzarotti, and Simone Aonzo. 2024. Unmasking the Veiled: A Comprehensive Analysis of Android Evasive Malware. In *ASIAACS 2024, 19th ACM ASIA Conference on Computer and Communications Security*.

[72] Silvia Sebastián and Juan Caballero. 2020. Avclass2: Massive malware tag extraction from av labels. In *Annual Computer Security Applications Conference*. 42–53.

[73] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. 2012. "Andromaly": a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems* 38, 1 (2012), 161–190.

[74] Ltd Shenzhen Chenxi Information Technology Co. 2024. Http traffic capture. https://apps.apple.com/us/app/http-traffic-capture/id1585539533.

[75] Lachlan Simpson, Kyle Millar, Adriel Cheng, Hong Gunn Chew, and Cheng-Chew Lim. 2023. A Testbed for Automating and Analysing Mobile Devices and Their Applications. In *2023 International Conference on Machine Learning and Cybernetics (ICMLC)*. IEEE, 201–208.

[76] Guillermo Suarez-Tangil, Santanu Kumar Dash, Mansour Ahmadi, Johannes Kinder, Giorgio Giacinto, and Lorenzo Cavallaro. 2017. Droidsieve: Fast and accurate classification of obfuscated android malware. In *Proceedings of the seventh ACM on conference on data and application security and privacy*. 309–320.

[77] Alain Tchana, Lavoisier Wapet, and Yérom-David Bromberg. 2022. Odile: A scalable tracing tool for non-rooted and on-device Android phones. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*. 252–262.

[78] Ash Turner. 2024. How Many Android Users Are There? https://www.bankmycell.com/blog/how-many-android-users-are-there.

[79] Ubuntu-Manuals-Argus. 2019. Argus Manpage. https://manpages.ubuntu.com/manpages/trusty/man1/ra.1.html.

[80] VirusShare. 2023. VirusShare. https://virusshare.com/.

[81] VirusTotal. 2023. VirusTotal. https://www.virustotal.com/gui/home/upload.

[82] Qi Wang, Wajih Ul Hassan, Ding Li, Kangkook Jee, Xiao Yu, Kexuan Zou, Junghwan Rhee, Zhengzhang Chen, Wei Cheng, Carl A Gunter, et al. 2020. You Are What You Do: Hunting Stealthy Malware via Data Provenance Analysis.. In *NDSS*.

[83] Quancheng Wang, Ming Tang, and Jianming Fu. 2023. EavesDroid: Eavesdropping User Behaviors via OS Side-Channels on Smartphones. *IEEE Internet of Things Journal* (2023).

[84] Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu. 2017. Malton: Towards {On-Device}{Non-Invasive} Mobile Malware Analysis for {ART}. In *26th USENIX Security Symposium (USENIX Security 17)*. 289–306.

[85] Xian Zhan, Lingling Fan, Sen Chen, Feng We, Tianming Liu, Xiapu Luo, and Yang Liu. 2021. Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1695–1707.

[86] Xian Zhan, Tianming Liu, Lingling Fan, Li Li, Sen Chen, Xiapu Luo, and Yang Liu. 2021. Research on third-party libraries in android apps: A taxonomy and systematic literature review. *IEEE Transactions on Software Engineering* 48, 10 (2021), 4181–4213.

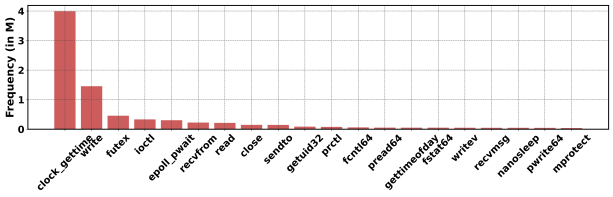
[87] Xinrun Zhang, Akshay Mathur, Lei Zhao, Safia Rahmat, Quamar Niyaz, Ahmad Javaid, and Xiaoli Yang. 2022. An early detection of android malware using system calls based machine learning model. In *Proceedings of the 17th international conference on availability, reliability and security*. 1–9.

[88] Zicheng Zhang, Wenrui Diao, Chengyu Hu, Shanqing Guo, Chaoshun Zuo, and Li Li. 2020. An empirical study of potentially malicious third-party libraries in android apps. In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. 144–154.

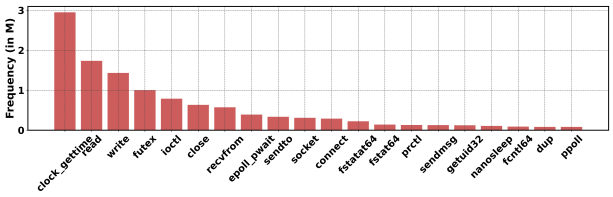
A APPENDIX

A.1 Distribution of System Calls in Malware Families

Figures 8a and 8b present the frequency distribution of top 20 system calls while running for 60 sec on a mobile device.



(a) Malware Family - `smsreg`.



(b) Malware Family - `greyware`.

Figure 8: Frequency distribution of top 20 system calls.

A.2 Challenges in COMEX Implementation

We faced multiple challenges while implementing and deploying COMEX .

- **Providing consistent power to devices:** Figure 2 shows multiple devices attached to the testbed. To function properly these devices need adequate and continuous power supply. In general, USB 3.0 ports of desktops do not deliver adequate power (less than 5 Watts) , *i.e.* lower than the rate at which the power drains from running the APKs. It is thus recommended to use an externally powered USB hub that could deliver adequate power while simultaneously allowing ADB to use the phone's USB port.
- **Non-parallelism in `monkeyrunner`:** Section 4.1 elaborates the usage of `monkeyrunner` tool with an APK on a mobile device. It does not support simultaneous execution on multiple devices, which is a requirement of DCoP module. Thus to simultaneously execute multiple `monkeyrunner` instances, a separate virtual machine (VM) is created where it runs. Now, each mobile device's USB interface needs to be port-mapped to its respective VM's virtual USB interface.

A.3 Keeping Datasets UP-to-Date

We provide methods and scripts for downloading APKs from up-to-date public sources such as AndroZoo and VirusShare in our github repository [31].

A.4 Relevance of COMEX w.r.t. APK versions

One might argue that COMEX may lose out behavioral data for an APK if multiple versions are released in the same year. But in the wild, if any APK is modified, it becomes a new test point (new APK altogether), which becomes a part of the dataset eventually. Then it is tested individually in COMEX . And thus, COMEX is not losing out any versions of the APK effectively.