The Droid is in the Details: Environment-aware Evasion of Android Sandboxes

Brian Kondracki
Stony Brook University
bkondracki@cs.stonybrook.edu

Babak Amin Azad
Stony Brook University
baminazad@cs.stonybrook.edu

Najmeh Miramirkhani
Stony Brook University
nmiramirkhani@cs.stonybrook.edu

Nick Nikiforakis
Stony Brook University
nick@cs.stonybrook.edu

***Abstract*—Malware sandboxes have long been a valuable tool for detecting and analyzing malicious software. The proliferation of mobile devices and, subsequently, mobile applications, has led to a surge in the development and use of mobile device sandboxes to ensure the integrity of application marketplaces. In turn, to evade these sandboxes, malware has evolved to suspend its malicious activity when it is executed in a sandbox environment. Sophisticated malware sandboxes attempt to prevent sandbox detection by patching runtime properties indicative of malware-analysis systems.**

In this paper, we propose a set of novel mobile-sandbox-evasion techniques that we collectively refer to as "environment-aware" sandbox detection. We explore the distribution of artifacts extracted from readily available APIs in order to distinguish real user devices from sandboxes. To that end, we identify Android APIs that can be used to extract environment-related features, such as artifacts of user configurations (e.g. screen brightness), population of files on the device (e.g. number of photos and songs), and hardware sensors (e.g. presence of a step counter).

By collecting ground truth data from real users and Android sandboxes, we show that attackers can straightforwardly build a classifier capable of differentiating between real Android devices and well-known mobile sandboxes with 98.54% accuracy. Moreover, to demonstrate the inefficacy of patching APIs in sandbox environments individually, we focus on feature inconsistencies between the claimed manufacturer of a sandbox (Samsung, LG, etc.) and real devices from these manufacturers. Our findings emphasize the difficulty of creating robust sandbox environments regardless of their underlying platform being an emulated environment, or an actual mobile device. Most importantly, our work signifies the lack of protection against "environment-aware" sandbox detection in state-of-the-art mobile sandboxes which can be readily abused by mobile malware to evade detection and increase their lifespan.

I. Introduction

Mobile devices have become an integral part of our lives. With Android having over 70% of mobile market share [14], these handheld devices now hold our most sensitive personal and financial information. This data, as well as the powerful devices that house it, have increasingly become a target for attackers.

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.23056
www.ndss-symposium.org

In 2009, the security community started observing malware samples, such as, Spitmo and Zeus [33], which were equipped with a mobile counterpart used to steal two-factor authentication tokens. Mobile malware samples found at the time were most often accompanied with a desktop counterpart that would be responsible for stealing banking information. Soon after however, as mobile devices started to get more traction, mobile malware evolved. Namely, new malware surfaced that performed on-device phishing, subscribed to premium numbers, and stole sensitive information [56]. This shift to mobile-first malware, emphasized the growing need for mobile malware detection.

Unlike desktop applications, which are independently downloaded and installed from the software manufacturers' websites, mobile applications are commonly distributed through centralized application stores; such as, the Google PlayStore [9] and the iOS App Store [10] hosting 2.9 million and 1.96 million apps respectively at the time of writing [21]. This makes the parties responsible for hosting and delivering these applications the first line of defense to stop mobile malware before reaching the user's device. To that end, defenders have proposed static and dynamic analysis methods for malware detection that app stores have, to a certain extent, adopted [22], [23], [25], [30], [54], [57], [61]. Over time, however, mobile malware increased in complexity, introducing evasion techniques such as code obfuscation to hinder static analysis, and sandbox detection to bypass dynamic analysis [47], [55].

A requirement for sandbox environments is scalability and the capability to revert to a clean state to evaluate new samples. This makes Android emulators the primary option for building mobile sandboxes. Thus, there exists a large body of work (from both academic research as well as malware samples) describing techniques to fingerprint emulated environments, mostly revolving around artifacts of emulation [27], [34], [49], [51]–[53], [58]. These approaches focus on features of the emulator, such as, the CPU architecture and instruction-level behavior, performance of emulators, and the discrepancy of OS-level APIs in real vs. emulated environments.

An important distinction between desktop and mobile sandbox fingerprinting comes from the limited variation of mobile devices. These devices ship with specific hardware features (e.g., sensors, camera, and storage) as well as distinct software properties (e.g., system applications and bloatware). This makes building believable mobile-sandbox environments more challenging, as sandboxes must hide signs of emulation, mimic real-device configurations, and display signs of real-device

usage. With these details in mind, we propose "environment-aware" sandbox evasion which uses artifacts from the mobile environment to identify sandboxes, rather than the specific behavior of emulators.

To evaluate an attacker's ability to evade sandboxes based on environment artifacts, we implement an Android application that extracts statistics from an extensive list of mobile APIs (e.g. the number of contacts, the number of calls, the number of photos, etc.). We select these APIs by reviewing Android API reference documentation, focusing on those that expose environment-related information. We submit this application to 12 popular Android sandboxes (including VirusTotal) as well as multiple app stores (e.g. Google PlayStore and Samsung Galaxy Market), with the expectation that apps uploaded in app stores are also executed in mobile-specific sandboxes for malware-detection purposes. In parallel, we conduct an IRB-approved study inviting users from a crowdsourcing platform as well as our institution to participate in our experiments by downloading and installing the same app. Through this process, we were able to obtain data from 1,245 user Android devices and collect environment statistics that we later use as ground truth in our machine-learning-powered classification models.

Based on the collected information, we build a classifier that detects mobile sandbox environments with 98.54% accuracy, demonstrating the power of artifact-based classification and thereby sandbox evasion, without the need of emulation artifacts that may change every time the emulating software updates. We show that the trained models are robust to incremental sandbox patches, finding that defenders can "remove" more than 60 out of 81 features, without radically affecting the performance of the classifier.

Finally, we build manufacturer-specific models based on the settings, system applications, and factory bloat on devices of popular manufacturers. We discover that for only three sandboxes, the claimed device model matches the environment features based on the manufacturer-specific classifiers. Contrastingly, for 75% of sandboxes, we observe that our manufacturer-specific classifiers trained on data from real user devices, identify that the sandboxes are effectively lying about their make. Through our experiments, we demonstrate that a sandbox can be discovered, not just because it looks like other sandboxes (in terms of its artifacts) but also because it *does not* look like a device from the manufacturer (Samsung, LG, etc.) that it claims to be.

In summary, we make the following contributions:

- We collect data on environment artifacts from physical Android devices and mobile sandbox services which we use to identify trends and large differences in their distributions.
- We train a machine-learning classifier utilizing a wide range of Android device artifacts that is capable of effectively differentiating real Android devices from sandbox environments of top anti-malware services.
- We utilize device-hardware configuration artifacts to train an ensemble of manufacturer-specific machine learning classifiers, and use them to discover discrepancies in the Android device manufacturers reported by sandbox services, enabling manufacturer-based sandbox evasions.

II. Background

Detection and analysis of computer malware is an important, yet challenging task. One of the most vital tools for this task are malware sandboxes. In this section, we provide a brief background on sandboxes and how malware has evolved to bypass them.

A. Malware Sandboxes

Sandboxes are isolated environments where software can execute under observation. They allow defenders to examine unknown software for malicious behavior, as well as known malware in order to create behavioral signatures for use in anti-malware systems. These systems can be physical machines (known as bare-metal sandboxes) with limited network capabilities in order to contain the unwanted side effects of potentially malicious code. However, due to the vast number of software executables analyzed, virtual machines and other system emulators provide a scalable platform to create malware sandboxes.

The explosion of mobile-device adoption has led to a surge in new software in the form of mobile applications. The Google PlayStore, the largest application marketplace on the Android platform, currently has nearly 3 million applications available for download, with that number increasing every day [21]. This massive influx of new applications makes manual analysis of every single app impossible. Rather, application marketplaces and security firms turn to malware sandboxes to automate the process of finding and removing malicious applications from the ecosystem.

B. Sandbox Evasion

Evading detection from defenders is a crucial aspect of modern malware. Once malware is discovered by a sandbox, signatures are created describing the malware sample and its behaviors. These signatures are then distributed throughout the anti-malware ecosystem, severely hampering (from the attacker's point of view) the number of affected victims. Thus, attackers are always improving upon their methods to detect sandbox environments so that malware does not demonstrate its malicious behavior while under the observation of defenders.

a) Device-based Evasion: One of the most trivial methods used to detect a sandbox environment is to look for so-called "Red Pills" i.e., static values that betray the presence of sandboxes. As many sandboxes utilize virtual machines, a common check conducted by malware is to look for VM-specific system properties. These come in the form of static identifiers such as MAC addresses or serial numbers, device drivers, and system modules. Malware can easily evade sandboxes by simply checking the value of these tell-tale system properties.

Additionally, sandboxes running in an emulated environment must ensure they report realistic values for all hardware and software on the system. Inclusion or exclusion of unusual hardware or software can lead to successful malware evasion. For example, a mobile sandbox neglecting to emulate a camera module when its existence is expected, can raise a red flag for attackers.

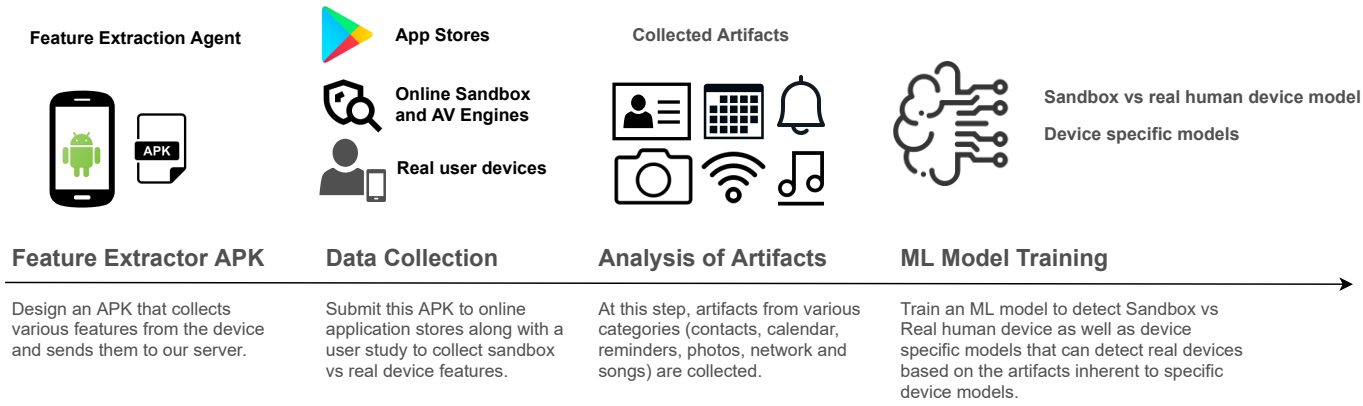

Fig. 1: *Flow of data from sources to our machine learning model.*

Sophisticated sandboxes tackle these issues by presenting realistic values for these variables (e.g. reporting a realistic MAC address or serial-number value rather than the default VM-specific values). While this raises the bar for attackers, these sandboxes can still be evaded through analysis of runtime behaviors. For instance, previous work has shown how certain code sequences have implicitly different behavior when executing on a bare-metal machine compared to an emulator [60].

b) Environment-based Evasion: Some sandboxes choose to utilize bare-metal devices rather than virtual machines and emulators. This eliminates the chance of evasion due to the detection of emulation as all hardware and software is authentic. However, many sandboxes neglect to prepare their environments to appear genuine (e.g. a Windows sandbox claiming to be a three-year old installation but showing little user activity in terms of installed programs and user files). Researchers have shown that these differences can be capitalized by malware to evade sandboxes and to even estimate the true age of a machine based on the presented wear and tear [48].

III. Data Collection

In this section, we begin with an overview of the artifacts we collect to differentiate real Android devices from sandbox environments. We then describe the Android application we designed and implemented for data collection, the sources we collect data from, and the details of our user study. Lastly, we analyze the collected data to discover trends in real Android devices compared to sandboxes, and identify differences in their distributions which would allow for the creation of a machine-learning classifier capable of differentiating between them. Details of our entire data collection and analysis pipeline is listed in Figure 1.

A. Artifact categories

To effectively evade mobile sandboxes, attackers must utilize as many sources of information as possible. This allows for the creation of a detailed model of real Android devices, which is robust against incremental improvements to sandboxes. To this end, we followed the methodology of Kurtz et al. [44] by systematically searching the Android SDK documentation for any API endpoint returning environment information. We identified a total of 81 features, split into three major artifact groups unique to mobile devices. Below, we describe these groups as well as the features belonging to each one, with a full listing of all artifacts located in Table VIII of the Appendix.

1) Wear-and-Tear Artifacts: Wear-and-tear artifacts are the natural side effect of mobile device usage. They describe the behavior of the user and can be used to estimate the age of the device. The different categories of these artifacts that we used for our study are as follows:

Photos For this feature category, we collect the number of pictures on the device, their average orientation (ratio of vertical to horizontal images), and the last time a picture was taken. Note that for photos (as well as other potentially PII-revealing artifacts listed later in this section), we collect statistics about them (such as the quantity of photos) but not the actual photos themselves.

Songs and albums For songs, we look at their total number, the number of artists, and the oldest and newest release dates.

Contacts We collect information such as the total number of contacts, as well as the number of those with a phone number or a picture associated with them. We also collect the number of "starred" contacts, custom ringtones, the total number of calls, and time of the last call. As with photos and songs, we calculate the statistics on the device (e.g. the number of contacts) and collect these statistics. We *never* extract the underlying sensitive information for obvious privacy reasons.

Calendars, events, and reminders We collect the total number of calendars, event types and their status (i.e. accepted, tentative, and canceled), and whether there is a reminder associated with these events. For reminders, we look at the reminder medium (i.e., Alarm, SMS, Email, and Default). We collect these features in bulk and not for individual calendar entries.

Packages, file system and running processes We extract the total number of packages installed on user devices. While malicious actors can go one step further and also extract individual package names and their installation dates, we chose not to do so in our study for privacy reasons. Similarly, we collect the number of documents, downloaded files, and files

listed under the `/proc` directory to obtain the number of running processes. Access to `/proc` has been discontinued since Android 6.0 and therefore our collection of that specific feature works on a best-effort basis.

Network and location For this set of features, the most notable feature that we take into consideration is the number of WiFi configurations saved on the device which, in principle, should correlate with the usage of the device (the longer users own a device, the larger the number of stored configurations on their phones). Moreover, we extract the number of ARP cache entries, which represents the number of devices on the local network that the Android device has recently communicated with.

2) Device Configuration Artifacts: Similarly to wear-and-tear artifacts, user-provided configuration options can show a degree of customization of a mobile device. When one begins to use their device, they change specific values in the device settings to meet their specific needs. By considering the values of specific configuration options, we can estimate the degree of customization of the device. However, while these configuration options indicate device usage, they are not quantifiable over long periods of time. Therefore, we discuss them separately from wear-and-tear artifacts.

We collect certain values under the device settings such as automatic update of time and time zone, data roaming, "stay on while plugged in", screen brightness level, as well as sound-related settings. One important feature in this category is the device uptime with the expectation that most users rarely turn-off/reboot their smartphones.

3) Device Hardware Artifacts: Unlike their desktop counterparts which can be assembled by users from individual components, mobile devices are inherently limited in the hardware configurations provided to users. These unique configurations of device components and settings can be used to supplement the aforementioned usage artifacts. Additionally, discrepancies in the configuration of a device compared to the reported device model can serve as a tell-tale sign of a sandbox's presence.

Our agent collects information about the device model itself such as phone manufacturer, model, and Android version. Additionally, it records the presence of certain hardware sensors, such as a heart rate monitor or a step counter.

B. Feature Collector Agent

To conduct data collection on mobile environments, we implemented an Android application which utilizes the Android SDK to gather the previously described artifacts, and transmit them to our servers. Our compiled application consists of a single APK file, allowing us to easily upload it to sandbox services for scanning or distribution to users through application marketplaces. Android applications are required to ask for permission from users to query specific APIs (e.g. contacts) and read certain configurations (e.g. saved WiFi access points). In older versions of Android, prior to 6.0 (API 23), all of the permissions would be acquired at installation time. In more recent versions, users are prompted at runtime to grant these permissions to the application. In total, our application requests five permissions: three runtime permissions, and two install-time permissions. The three runtime permissions request access to read the user's calendar, contacts, and documents on storage devices; whereas the two install-time permissions request access to read the WiFi state, and user dictionaries.

Data is transmitted from our application to our servers over a single POST request sent over an HTTPS connection. There, we store the information in our database to be used in training and validating our machine learning model. Each APK that we distribute on a marketplace or submit to a sandbox has a unique ID which is transmitted with all data in each request to our servers, and is used to distinguish data between different sources (e.g. Google Play Store vs. VirusTotal AV).

C. Collection of Features

To systematically search for all active sandbox providers, we conducted a literature review in the field of mobile sandbox development. Additionally, we utilized search engine queries for terms such as "Android Sandbox", "mobile sandbox", and "malware sandbox". This search resulted in a large collection of mobile sandboxes, ranging from open-source academic tools to commercial services. For each sandbox discovered, we upload or scan our client application. Additionally, we distribute our application on popular Android marketplaces in order to capture data on sandboxes and malware scanners used by these entities. We note that there are a number of other sandbox services we discovered in this search that were either broken, discontinued, or operated in such a way that prevented us from evaluating them. For instance, we identified 29 mobile antimalware software out of which, 17 either only performed static analysis or did not allow outbound HTTP/DNS connections to our servers. Ultimately, we tested our application on 12 working antimalware sandboxes.

To capture the state of new Android devices and build a baseline model (i.e. the distributions of the aforementioned artifacts in newly-purchased devices), we utilized Amazon Device Farm [2] and Google Firebase Test Lab [7]. These services allow developers to test their applications on a large slate of real, physical devices and pay for the time they use each device. We uploaded to each of these services an augmented version of our Android client which automatically initiates data collection without any user interaction.

a) IRB Approval and User Study: Collecting data from real Android devices required the assistance of real users. Thus, we obtained an Institutional Review Board (IRB) approval for our study. Upon providing thorough details on what data we would collect from participants as well as how we would protect participant privacy, we obtained an IRB approval on October 7, 2020.

The data we collect from users does not contain any personally identifiable information (PII). All artifacts we described previously are statistical in nature. For instance, we collect the total number of contacts on a user's device, rather than the contact information itself.

To recruit participants, we sent out emails to students of our institution asking for them to participate in our study for a chance to win a gift card. Recruitment emails were sent from our institutional email accounts, to ensure participants were confident they were not spear-phishing attempts. All

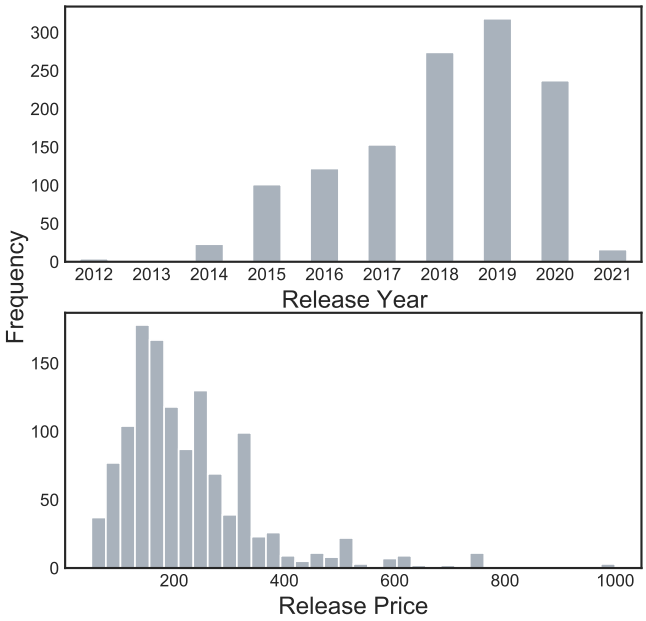

Fig. 2: *Distribution of device release dates and prices in our dataset.*

instructions were provided to participants within the recruitment email. Users were asked to download and install our application onto their Android mobile device from the Google Play Store. After providing the required permissions, we asked them to navigate to the "User Study" tab in the application and click on the "Get Token" button. At this point, our application transmits the collected artifacts to our servers, and participants receive a token which proves their participation. The total time required by participants was between 3-5 minutes.

To bolster our dataset of real user devices, we conducted the same user study with hired workers on the Microworkers [13] platform. Instructions for participants were listed on the Microworkers job listing, and participation tokens where shared using the job listing as well. Workers on this platform were paid an average of $0.75 for their participation, rather than an entry into the gift-card raffle. In total, we spent approximately $850 to complete our user study.

D. Dataset Statistics

Using our Android agent, we collected data from 1,245 real user devices, 398 baseline devices, and 210 sandbox devices. Baseline devices are real Android phones accessible through Google Firebase Test Lab [7] and Amazon Device Farm [2] services, which provide cloud-based application testing. Of real user devices, 92 originated from students of our institution, and 1,153 from the users of the Microworkers platform. Figure 2 presents the distributions of device release dates and release-time prices of devices (in US dollars) in our dataset. Here, we see a diverse range of device ages and prices. However, most devices in our dataset are on the cheaper end, which can be explained by the demographic distribution of workers on the Microworkers platform, favoring budget devices [37]. Additionally, Table I shows the total number of data points we collected from each sandbox service, as well as the number of unique devices encountered. We observe

TABLE I: *Number of samples collected from each sandbox in our dataset, as well as the number of unique devices collected from each source.*

Sandbox	# Samples	# Devices
VirusTotal AV	73	17
Comodo Sandbox	27	12
GData AV	18	9
Google PlayStore Market	15	3
eScan (mwti.net) AV	12	1
Samsung Galaxy Store Market	10	1
GetJar Market	10	4
Tgsoft AV	9	8
Amazon AppStore Market	9	2
Kaspersky AV	8	5
DrWeb AV	5	2
SandDroid Sandbox	5	2
AMAass Sandbox	3	1
JoeSandbox	3	2
BitDefender AV	2	2
Nviso Sandbox	1	1

TABLE II: *Distribution of Android device manufacturers in our real devices compared to sandboxes.*

Manufacturer	Real Devices	Sandboxes
Xiaomi	25.50%	1.16%
Samsung	23.89%	7.56%
Huawei	8.31%	5.23%
Oppo	5.97%	0.00%
Realme	5.17%	0.00%
Vivo	3.95%	2.33%
LG	3.79%	27.33%
Motorola	2.58%	0.00%
Oneplus	1.94%	0.00%
Hmd global	1.86%	0.00%
Asus	1.86%	8.14%

that the majority of sandbox services launch our application multiple times after submission. However, we find most of these repeated launches occur from one or a small group of device models, using a varied group of reported Android versions. We can infer that this differentiation is an attempt to coerce malicious behavior out of malware that may be targeting specific devices and Android versions.

We notice a large number of execution data points originating from our Android client submitted to VirusTotal as well as popular application marketplaces. VirusTotal's APK file evaluation currently uses 28 underlying antimalware checks. As a result, a single file will be analyzed and dynamically executed multiple times. Not all of the antimalware engines in VirusTotal perform dynamic analysis and one cannot choose to analyze the submitted file on a subset of engines separately. Unlike sandbox services, data points from marketplaces may be varied with some coming from the vetting procedure of the marketplace as well as real users downloading our application. To differentiate between these two scenarios, we used the ASN of the source IP address of each connection. If it belongs to the organization of the marketplace or of an organization providing computing resources (such as a public cloud), we consider that a sandbox, otherwise we consider it a real device.

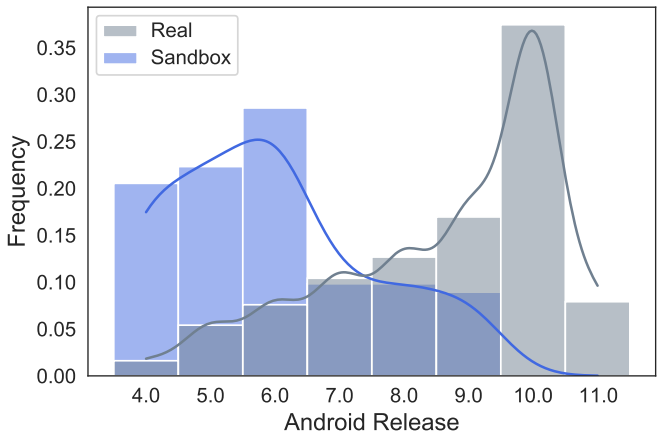

Fig. 3: *Distribution of Android versions among real user devices and sandboxes that report an Android release version.*

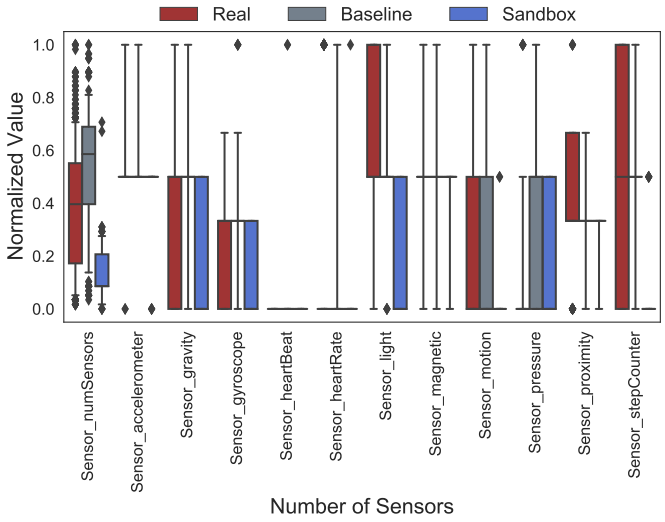

Fig. 4: *Distribution of sensor hardware in our dataset groups. Box and whiskers encompass 90% of data group values.*

Table II shows the distribution of device manufacturers among our datasets of real user devices and mobile sandboxes as reported by the Android *build* API. Here, we observe that the majority of real users own devices from popular smartphone manufacturers such as Samsung and Xiaomi. However, there is a lack of representation of these popular devices in mobile sandboxes. We find that 34.8% of sandboxes do not report any device manufacturer when requested by our application.

We also observe that this same group of sandboxes (34.8% of our dataset), do not report an Android version when requested. Figure 3 shows the distributions of reported Android software versions in both real user devices and mobile sandboxes. Most sandboxes that report an Android version claim to be running a very early release. This may be an attempt to coerce malware into revealing malicious behavior when encountering an outdated and vulnerable device. However, we see that real user devices in our dataset are more likely to run a recent release of Android.

These discrepancies are alarming as it is important for mobile-sandbox providers to follow the usage trends of real Android mobile devices in order to prevent trivial sandbox detection. "Red Pill" information such as this is powerful and non-invasive, meaning attackers can easily and stealthily acquire this information to evade sandboxes. At the same time, this type of manufacturer-based and Android-version-based evasions can be straightforwardly patched by sandboxes merely reporting the appropriate values. As such, we focus our attention on more inconspicuous information that can betray the nature of the sandbox while also requiring a greater amount of effort for sandbox providers to properly counter.

Figure 5 shows the distributions of wear-and-tear artifacts in our dataset. For increased clarity, we show 16 out of a total of 41 wear-and-tear artifacts. We find that, in most cases, real user devices produce wear-and-tear artifacts of a larger magnitude (e.g. a large number of WiFi configurations) and a larger variance than those in our sandbox and baseline datasets. This observation shows how the usage of a mobile device can be quantified from wear-and-tear artifacts, and that these artifacts can be used to reliably differentiate a mobile sandbox environment from a real user's device. We do, however, encounter some features which unexpectedly provide less classification power than we originally expected. One such example is the number of processes running on a device. Intuitively, one would expect a real device to have more running processes due to users launching many applications during typical use. Since we do not collect the actual packages installed on a device, we cannot conclusively explain this discrepancy. One possible explanation is that sandbox services have extra logging and analysis processes running in parallel with suspected malware, thus leading to inflated process counts.

Next to wear-and-tear artifacts, the relative stability of mobile device hardware compared to desktop computers allows for the use of hardware configuration artifacts in sandbox evasion. Figure 4 shows the distribution of sensors in devices of each category in our dataset. We find that mobile sandboxes are less likely to include emulated hardware such as environment sensors, creating an additional differentiating factor between artificial and genuine Android environments. We further expand upon these artifacts in Section V, where we create machine learning models for popular Android device manufacturers.

IV. Mobile Sandbox Detection

In the previous section, we described the artifacts that can be used to differentiate real Android devices from mobile sandboxes operated by appstores and anti-malware companies. Our initial analysis of data collected from real-user devices, baseline devices, and sandboxes, demonstrated that real Android devices and sandbox environments produce unique distributions for many of these artifacts, pointing to an opportunity for their differentiation. In this section, we use these artifacts as features in a machine-learning classifier that can be used by mobile malware to evade analysis by defenders.

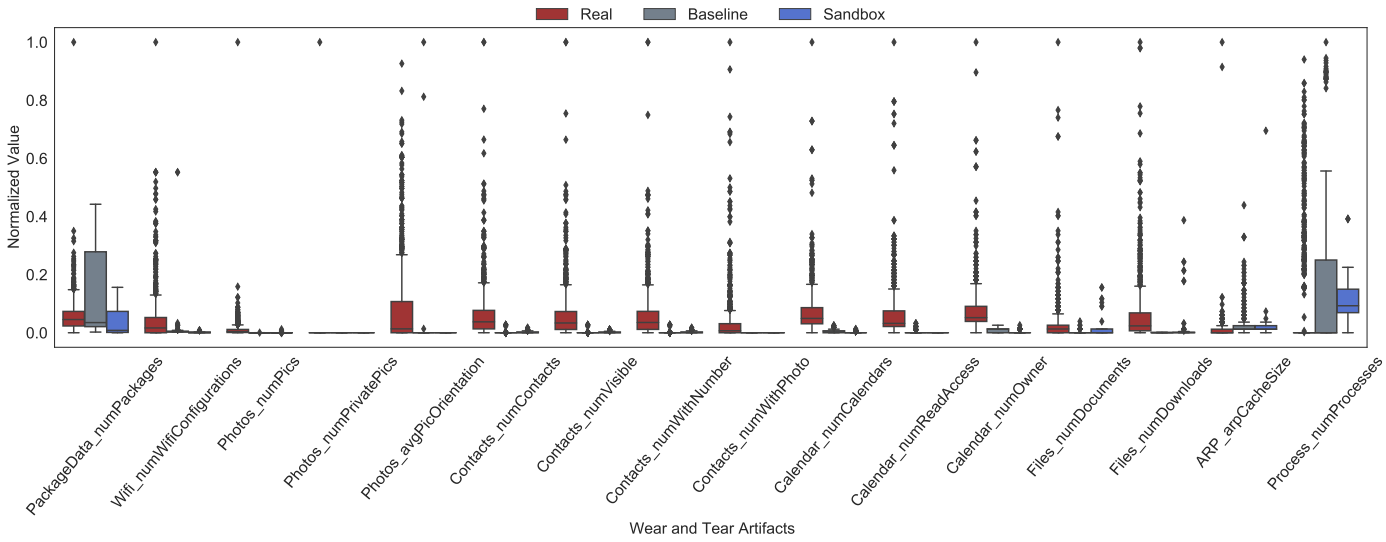

Fig. 5: *Distribution of wear and tear artifacts in our dataset groups.*

A. Classifier Setup

For an attacker implementing a classifier to detect mobile sandbox environments, simplicity of implementation and explainability of results is of the utmost importance. It is for these reasons that we chose to use a decision-tree-based, machine-learning algorithm for our classifier. Decision trees are one of the most straightforward machine-learning algorithms to implement, requiring only a series of conditional statements. This property is valuable for use in malware as this leads to a negligible increase in file size and profile. Moreover, in order to understand and react to changes made by mobile sandboxes, attackers need to visualize the importance of particular features in their classifier. By observing the splitting rules at each level of a decision tree, attackers can easily determine the decision boundaries, and make corrections as sandboxes evolve.

Before creating our classifier, we focus on the careful construction of our training and testing datasets. As described in Section III-B, we collect data from three sources: Android sandbox services, real user devices, and test devices from the services Amazon Device Farm and Google Firebase Test Lab. For brevity, we will refer to these datasets as D_{sand}, D_{user}, and D_{base}, respectively. In order to create a classifier capable of detecting various kinds of sandbox environments, from clean physical devices to sophisticated commercial sandboxes, in this section, we consider devices from D_{base} to be sandboxes.

To create our training dataset, we randomly sample 200 data points from D_{user}, and include 100 samples from both D_{sand} and D_{base}. This produces a balanced training dataset consisting of 400 total samples. We use the remaining samples from all three datasets for testing, 408 positive samples and 1,045 negative samples.

Using our compiled datasets, we evaluated the performance of a number of popular decision tree-based machine learning algorithms including: Decision Trees, Random Forest, Gradient Boosting, and Adaptive Boosting. We found that Gradient Boosting provided us with the greatest performance, while still keeping the beneficial properties of decision trees we described earlier. Gradient boosting improves upon basic decision trees by iteratively constructing trees that improve upon the weaknesses of the previous tree. This increases the overall performance without increasing the implementation complexity. Using a grid search, we find that creating a forest of 53 trees with a learning rate of 0.1 and maximum tree depth of 13, leads to the most optimal performance. Finally, we focus our attention on optimizing our classifier for a low false-negative rate. Such misclassifications can be extremely costly for malware, as displaying malicious behavior in a sandbox environment will lead to detection by antimalware services and subsequent removal from application marketplaces.

We examine the features used by each of the trees in our classifier in Table III and find that, out of the top 20 most commonly used, 13 belong to the Wear-and-Tear category. The high representation of wear-and-tear features in our classifier demonstrates the classification power of these often overlooked aspects of the Android environment. Our classifier does not rely on a small group of "red-pill" features to determine the presence of an emulated or otherwise artificial device. This adds to the robustness of our classifier compared to prior work as patches to singular API values will not be enough to bypass detection.

B. Evaluation

We evaluate our classifier on the testing set previously described. Our classifier achieves an accuracy score of 98.54%, false-positive rate of 1.58%, and false-negative rate of 0.57%. In total, our classifier produces *one* false negative, originating from D_{sand}. Further, our classifier produced a 10-fold cross validation score of 99.27% using the same testing dataset.

Figure 6 shows the Receiver Operating Characteristic curves for our classifier tested on both a balanced (1:1 real-device to sandbox ratio) and un-balanced (99:1 real-device to sandbox ratio). We note that in the real world, mobile malware is likely to encounter a much greater number of real Android devices than mobile sandboxes. We therefore report the per-

TABLE III: *The 20 most important features to our classifier based on the percentage of trees that rely on each. W&T, DH, and DC represent Wear-and-Tear, Device Hardware, and Device Configuration artifacts, respectively.*

Category	Feature	% of Trees
W&T	Process_numProcesses	88.7%
W&T	Photos_numPics	83.0%
W&T	Calendar_numOwner	75.5%
W&T	Calendar_numCalendars	73.6%
W&T	Contacts_numContacts	71.7%
DH	GyroscopeSensorReadings	64.2%
DC	Device_UpTime	64.2%
W&T	Contacts_numVisible	62.3%
DC	StayOnWhilePluggedIn	52.8%
W&T	numWifiConfigurations	45.3%
DH	DeviceTemperatureSensorReadings	45.3%
W&T	ArpCacheSize	45.3%
W&T	Events_avgEventLen	41.5%
W&T	Contacts_numWithNumber	41.5%
DH	Sensor_proximity	39.6%
W&T	Events_numTentative	37.7%
DH	LightSensorReadings	35.9%
DC	DeviceRooted	35.9%
W&T	Photos_avgPicOrientation	18.9%
W&T	Contacts_numWithPhoto	18.9%

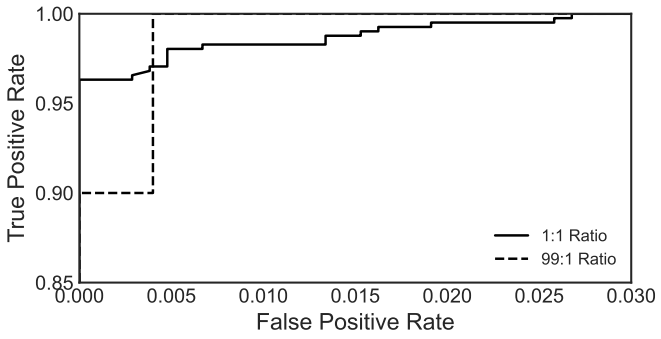

Fig. 6: *Receiver-Operator Characteristic Curves of our mobile sandbox classifier tested on datasets with a 1:1 and 99:1 real-device to sandbox ratios*

formance of our classifier on an un-balanced dataset to reduce spatial bias in our experimental outcomes [50]. We find that in both cases, our classifier is able to effectively differentiate real Android environments from artificial environments created by sandbox services, as it can achieve a true positive rate of over 99% with a false positive rate of below 3%.

An important property of mobile sandboxes is that they can constantly evolve to prevent evasion. Thus, quirks which currently betray the nature of the sandbox are not guaranteed to exist in the future. To ensure our classifier does not overfit on a small subset of powerful features, we study the decay of model performance as we iteratively remove the most important features. We determine the importance of each feature to the performance of our classifier by iteratively removing each feature, retraining, and evaluating the resulting performance. We then rank the importance of each feature by the performance dropoff of the classifier when that feature was omitted. Figure 7 shows the decrease in accuracy as well as increase in false positive and negative rates as features are

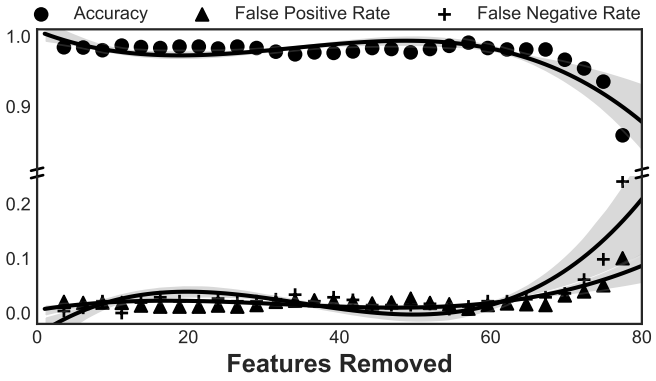

Fig. 7: *Drop-off in classifier performance when removing the most important feature and re-training.*

removed. We observe that even after removing the top 60 features, the accuracy of our classifier remains above 95%, only severely dropping once over 70 features are removed. This shows how our classifier does not rely on a small group of powerful features, but rather the entire ensemble of features as a whole.

We also record the performance of our classifier when isolating each of the three identified feature groups. This allows us to determine which specific areas sandbox services fail in their emulation of real Android devices. Table IV shows the performance of the feature groups: wear-and-tear, device-hardware properties, and device-configuration properties, as described in Section III-A.

We find that for the task of sandbox detection, device configuration and wear-and-tear artifacts perform the greatest, while a model using only device-hardware properties, though still powerful, exhibits a 2×-6× more false positives, compared to the other models. When isolating only wear-and-tear artifacts, our classifier achieves an accuracy of 97.9%. Similarly, when using only device configuration artifacts such as the state of device settings like WiFi and Bluetooth, our classifier achieves an accuracy score of 98.7%. As these two groups account for 62 features, the lack of attention to detail by sandbox services here means attackers can create powerful and robust classifiers to effectively evade detection by these services.

Finally, we measure the performance of our classifier when tested on devices of different ages and price ranges. To determine performance by device age, we divide our testing dataset into three groups: devices released before 2015, devices released between 2015 and 2018, and devices released after 2018. Additionally, to determine performance by device price, we divide our testing dataset into three additional groups: devices costing less than $250, devices costing between $250 and $750, and devices costing over $750. Note: we label each device by the release MSRP in US Dollars. We observe in Table V that the performance of our classifier does not drastically change when encountering devices of these groups. This demonstrates that while Android device hardware and software changes over time and among different price ranges, the magnitude of these changes is not large enough to affect the environment differences between real devices and mobile sandboxes.

TABLE IV: *Classifier performance when trained on artifact groups in isolation (FPR = False Positive Rate, FNR = False Negative Rate).*

Artifact Type	#Artifacts	Accuracy	FPR	FNR
Device Hardware	19	91.8%	5.1%	3.06%
Device Config.	21	98.7%	0.58%	0.73%
Wear-and-Tear	41	97.9%	0.87%	1.17%

TABLE V: *Classifier performance when tested on devices of different ages and price ranges. We consider a device to be old if it was released prior to 2015, intermediate if it was released between 2015 and 2018, and new if it was released any time after 2018.*

Device Age	Accuracy	FPR	FNR
Old	100.00%	0%	0%
Intermediate	94.8%	6.3%	0%
New	99.3%	0.8%	2.6%
Device Price	**Accuracy**	**FPR**	**FNR**
$0-$249	99.58%	0.44%	0%
$250-$749	95.54%	4.25%	5.45%
$750-$1,000	100%	0%	0%

Minimizing the number of requested permissions

Android malware is limited in its stealthiness compared to its desktop counterparts due to the fact that security-sensitive operations such as accessing certain APIs is blocked by the Android permission system. Any application that requires access to such functionality needs to declare the necessary permissions in the manifest file. This can lead to suspicion by malware scanners if too many sensitive permissions are requested. Moreover, Android permissions are divided into two main categories: install-time and runtime permissions. Android install-time permissions are listed in the manifest file, and are granted at install-time. Runtime permissions are also listed in the manifest, but prompt the user for further confirmation when the sensitive action is invoked. Depending on the declared nature of the app, these prompts could lead to the malicious behavior of the application being discovered (e.g. a weather application suddenly requesting access to the user's contact list). Therefore, it is in the best interest of the attacker to only request the permissions required for the malicious activities.

Figure 8 lists the permissions that our Android agent requests along with the number of features which rely on each permission. The features we use in our classifier extend across six different permission groups. We determine the dropoff in classifier performance when removing requested permissions by dropping features associated with each permission, retraining our classifier, and evaluating its new performance. We find that when using only features that either do not need explicitly stated permissions, or only require an install-time permission, our classifier still achieves an accuracy score of 99.08%, a false positive rate of 0.44%, and a false negative rate of 0.73%. Restricting a classifier to these features would, at the time of this writing, allow an attacker to reliably evade Android malware analysis systems without the need to include any additional permissions other than what is required for the desired malicious actions. By not requiring runtime permissions, malware can stealthily determine the current environment. Moreover, as the false negative rate remains low, attackers can be confident that a negative classification is correct and won't inadvertently give away the application's malicious behavior.

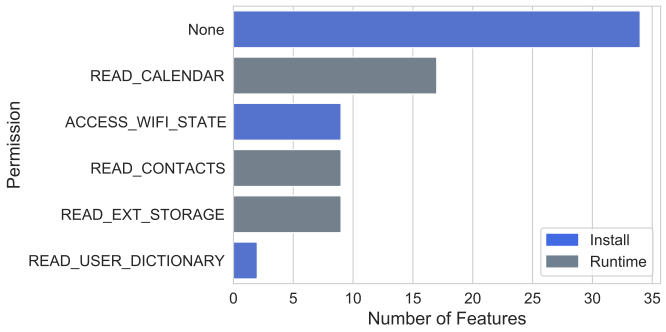

Fig. 8: *Number of artifacts collected by our Android client broken down by the permissions required to access each.*

V. Device Manufacturer Classification

In the previous section, we showed how various aspects of Android mobile devices can be used to differentiate between real and sandbox environments. We discovered that sandbox providers do not effectively emulate all aspects of real Android mobile devices and, therefore, attackers can create powerful and robust classifiers to evade detection. In this section, we explore discrepancies involving the reported manufacturer of mobile devices to determine if characteristics of the environment do not align with what is reported by the device.

We focus on a subset of artifacts of those listed in Section III-A that represent Android devices independent of the actions of users, and can provide insight into the device's manufacturer. We then show how we can use these artifacts to create machine-learning classifiers specific to each device manufacturer. Lastly, we demonstrate how attackers can use these classifiers as an additional side-channel in determining the presence of a sandbox by examining discrepancies in their output compared to the manufacturer reported by the device. Effectively, we investigate whether a sandbox can be discovered, not because it looks like other sandboxes (in terms of its artifacts) but because it *does not* look like a device from the manufacturer (e.g. Samsung, LG, etc.) that it claims to be.

A. Device Hardware Properties

Due to their form-factor and locked-down nature, mobile devices are typically limited to the hardware chosen by the manufacturer, with few options for user customizability. Thus, a particular Android device model will *always* have the same hardware configuration, regardless of the user. Moreover, hardware configurations amongst device models of the same manufacturer will likely remain consistent. Therefore, if sandbox services do not pay attention to details such as these, attackers who are aware of typical hardware configurations of a particular manufacturer can use these discrepancies to evade the sandbox.

As we described in Section III-A, we define device hardware configuration as the set of artifacts which correspond to specialized hardware components. This hardware includes environment sensors (e.g. light and gravity sensors), health trackers (e.g. heart rate and step counters), and communication modules (e.g. WiFi radios). In Section III-D, we explored the unique hardware configuration distributions between D_{user},

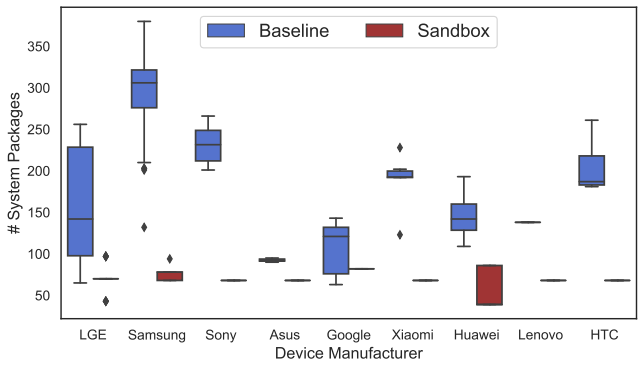

Fig. 9: *Distribution of system image packages in real Android devices compared to mobile sandboxes.*

D_{base}, and D_{sand} and showed the divide between these groups in Figure 4. Among others, we discovered that device statistics, such as, the number of each kind of environment sensors and configuration options of communication modules, can vary drastically from manufacturer to manufacturer. Additionally, this information from the hardware specifications can be acquired by attackers stealthily as they require no runtime permissions.

B. Quantifying Device Bloat

Operating system bloat is a trend common to many popular Android mobile devices. Manufacturers pre-load their devices with a tailored Android distribution containing extra, often undesired, software packages. This software, referred to as *bloatware*, is commonly bound to the device and cannot be easily removed. Different manufacturers will include various amounts of bloatware onto the system images of devices they sell. By studying the bloatware included by each manufacturer, we can determine the expected size, in number of included packages, of each manufacturer's system image.

We collect the number of packages on the image of each Android device by counting the number of entries in their `/system/app` and `/system/priv-app` directories. As described in [35], packages in these directories are pre-installed on the system image, whereas applications installed by users are located in `/data/app`. It should be noted that not all packages in these directories are stand-alone applications. Rather, many execute as background processes related to the operating system. We count all entries regardless of whether they are stand-alone applications as even packages providing background services can be linked to manufacturer bloatware.

Figure 9 shows the distributions of pre-installed packages for each device manufacturer observed in both D_{base} and D_{sand}. We can see that in the case of all manufacturers, real devices in our dataset have significantly more pre-installed packages than sandbox devices supposedly of the same manufacturer. In one specific scenario, a Google Pixel 3 XL running Android 9 in D_{base} contained 141 pre-installed packages, while the same purported device running the same version of Android in D_{sand} only had 82 pre-installed packages. This can be explained by sandbox providers stripping out unnecessary bloatware in an attempt to increase the performance of their sandbox. However, it is unlikely that a normal user will attempt to do the same on their personal device. Therefore, attackers can use this discrepancy to determine the presence of a sandbox.

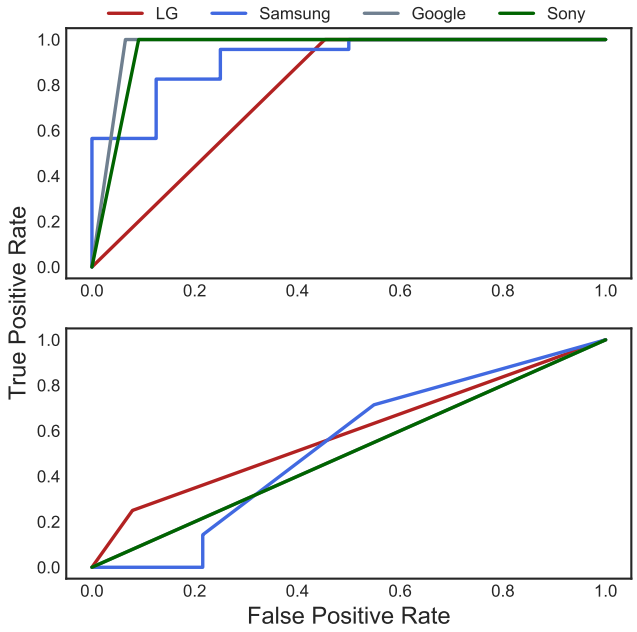

Fig. 10: *Receiver Operating curves for manufacturer-specific machine learning classifiers performing on data collected from real Android devices (top), and mobile sandbox services (bottom).*

C. Device Manufacturer Classifiers

So far, we demonstrated the differences in configurations of device hardware and pre-installed software between real Android devices and mobile sandboxes. We now describe how attackers can use these artifacts to train manufacturer-specific machine learning classifiers for the devices of top Android manufacturers, and show how their outputs can be used to pick up on discrepancies in sandbox environments for the purpose of evasion.

When deciding upon how to classify device manufacturers, we chose to create an ensemble of classifiers, each specialized in detecting a particular manufacturer, rather than one multiclass classifier. As the number of potential device manufacturers can be large, this can have a negative effect on the performance of such a classifier. Further, multiple individual classifiers can allow attackers to specifically target one or a small set of devices with greater precision.

To create manufacturer-specific classifiers utilizing the features previously described, we re-upload an augmented version of our Android application to each of the identified sandbox services, as well as Google Firebase and Amazon Test Lab. We configure our application to collect the total number of system packages as opposed to the number of user installed packages, as we did for the datasets discussed in Section IV. This allows us to observe the device bloat of Android device manufacturers rather than wear-and-tear artifacts produced by users.

We assemble training and testing datasets by assigning data points from D_{base} corresponding to each manufacturer as the positive data points, and all other samples from D_{base} as negative. Training datasets are created by randomly sampling 75% of all given positive data points, and an equal number

TABLE VI: *Percentage of samples from Android sandboxes in which the reported device manufacturer matches the output of our manufacturer classifier ensemble.*

Sandbox	Samples	% Manufacturer Match
Sandbox 1	16	100.0%
Sandbox 2	4	100.0%
Sandbox 3	4	100.0%
Sandbox 4	36	25.0%
Sandbox 5	36	20.0%
Sandbox 6	64	14.3%
Sandbox 7	60	11.1%
Sandbox 8	8	0.0%

of negative data points. All remaining samples are considered the testing set for that manufacturer. Note that we do not train classifiers for device manufacturers for which we are lacking a sufficient number of data points (at least ten total samples per device manufacturer in the training dataset). In total, we create four datasets for popular device manufacturers and use the data from D_{sand} to determine how effective mobile sandboxes are in emulating real Android devices.

Using our compiled datasets, we train four Gradient Boost decision tree classifiers, each optimized independently using Grid Search. Figure 10 shows the performance of our classifiers when testing on real Android device data and mobile sandbox data. We find that our classifier achieves consistently greater performance on data from real Android devices, while suffering on sandbox data. This, along with the device-specific feature distributions in Figure 4 and Figure 9, demonstrates that the data originating from mobile sandbox services does not match the distributions of real Android devices from the manufacturers they claim to be. Intuitively, a lower performance by our device-specific classifiers on mobile sandbox data indicates that, due to the values of the features previously described in relation to the expected distributions, the classifier determined the device was not of the reported manufacturer (leading to a "misclassification"). It is this discrepancy that attackers can leverage to determine the presence of a sandbox.

Furthermore, Table VI shows the performance of our manufacturer-specific classifiers on data from each of the sandboxes that report using a device from the same manufacturer as one of the classifiers in our ensemble. We anonymize the sandbox names to prevent direct comparisons between commercial products. Here, a sandbox performs well if the classifier for the manufacturer reported by the device outputs a positive label. In these cases, malware utilizing such a classifier would conclude that they are executing on a real device and manifest their malicious behavior, allowing the sandbox to create a signature. Conversely, sandboxes with a low manufacturer-match percentage do not sufficiently emulate the environments they claim to be. As such, using classifiers from our ensemble, attackers can observe the reported device manufacturer differing from that of the classifier output, and evade the sandbox. In our dataset, only three sandboxes achieve a 100% manufacturer match rate, while one sandbox produces discrepancies in all samples.

To demonstrate how attackers can use both of our proposed classification methods (Sandbox classifier and Device manufacturer discrepancy) in tandem to evade mobile sandbox

TABLE VII: *Verification of samples labeled as real devices (negative label) by classifier in Section IV.*

Sandbox Classifier		Manufacturer Discrepancy?	
Output	Samples	Yes	No
True Negative	378	0	378
False Negative	1	1	0

environments, we verify all samples classified as real mobile devices in Section IV. The idea is that the attacker will use the device discrepancy model after the first classifier on non-sandbox classifications to further reduce the false negatives and ensure that malware does not execute inside of a sandbox. To simulate this two-step evasion strategy, we pass the subset of the samples which report a device manufacturer matching one of our classifiers in our ensemble and determine the updated output. We focus on the samples labeled as real devices by our classifier in Section IV as these could be false negatives. Additionally, we remove the system package count feature from our manufacturer classifiers as the data from Section IV contains the number of user-installed packages. Instead we focus only on the device hardware properties as these features are consistent between both datasets.

In total, we verify 379 negative samples from Section IV whose reported manufacturer is one of the four popular manufacturers we created classifiers for, out of 1,019 total negatives. Of these samples, 378 were true negatives and 1 was a false negative from the first classifier. Table VII lists the results of this verification. We find that none of the true negatives had a discrepancy between the output of our manufacturer classifiers and the manufacturer reported by the device, while the single false negative was corrected. From the perspective of an attacker, preventing false negatives is more important than reducing false positives as mislabeling a sandbox as a real device can lead to detection and prompt removal from application marketplaces.

In summary, we conclude that it is not sufficient to patch individual Android APIs to return believable values in isolation when constructing a malware sandbox and the relationship between API values and device model is an important factor. This detail has been overlooked by the majority of sandboxes in our dataset. Our ensemble of manufacturer-specific classifiers is able to effectively identify Android devices of particular manufacturers using only their unique sensor configurations. The decrease in performance observed when testing on data from mobile sandboxes demonstrates that while sandboxes may identify themselves as a device of a particular manufacturer, their reported hardware configurations do not support those claims. This discrepancy, along with any other environment discrepancies, is enough for attackers to detect the sandbox and evade analysis.

VI. Discussion

In this section, we discuss our main findings, the environment-related weaknesses of current Android sandboxes, and suggest improvements to build sandbox environments that are more resilient to evasion. Additionally, we discuss the disclosure of our findings to sandbox services.

A. Weakness of Android Sandbox Services

Our work has shed light onto the shortcomings of mobile sandboxes in emulating the full Android environment. We stress that when creating a mobile sandbox, attention to small details is of critical importance. More specifically, contrary to desktop sandboxes, we demonstrate that it is not sufficient to patch individual APIs and the whole environment should match the details of the claimed device manufacturer and model. While we find some sandboxes are successful in emulating the hardware of real Android devices, usage-based artifacts still allow for the effective differentiation between artificial and real Android devices.

Due to this, we were able to train a machine learning classifier capable of differentiating real Android devices from sandboxes with 98.54% accuracy. More importantly, our classifier achieved a false negative rate of 0.57%, meaning malware utilizing similar techniques could evade the existing commercial and state-of-the-art sandboxes (including VirusTotal and Google PlayStore) with a very low probability of detection. Furthermore, we trained an additional ensemble of machine learning classifiers, each specialized to detect Android devices of a particular manufacturer. We demonstrated how we could use these classifiers to verify the results of our main classifier, effectively eliminating false negatives. Our findings reinforce the fact that if mobile sandboxes do not pay the required attention to every small detail of real Android devices, attackers can bypass their analysis.

B. Mitigations

Effective emulation of any computer system is not a trivial task. Attempting to do so for the purposes of malware analysis will undoubtedly lead to mistakes which attackers can use to evade detection. We therefore suggest mobile sandbox services utilize real Android devices rather than emulators. This, along with populating devices with usage-based artifacts to simulate real use will significantly raise the bar for attackers. Services such Google Firebase Test Lab and Amazon Device Farm have shown that large-scale application testing on physical devices can be achieved. However, cost may still be a limiting factor for smaller sandbox services.

An alternative method for improving mobile sandboxes while maintaining use of emulators, would be to constantly monitor the artifacts that malware collect and dynamically return realistic values to these queries. While this well-known practice is already employed by antimalware engines, our results show a gap between malware capabilities and existing defensive mechanisms when it comes to sandbox evasion. Nevertheless, the inherent limitations of this arms race ultimately leads to evasion techniques as sandbox services would constantly remain a step behind malware in determining which artifacts are of the most interest to attackers.

Our results demonstrate the power of usage-based artifacts in sandbox detection. Table III shows that 13 of the top 20 most important features to our classifier correspond to artifacts of real device usage. Sandbox providers must ensure they accurately model this aspect of the Android mobile environment to match the distributions of real user devices. In Section III-B, we show that crowd-sourcing can be used to collect usage statistics on a wide variety of real Android devices. Sandbox providers and application marketplaces can use such techniques to constantly generate up-to-date usage statistics to construct more realistic sandbox environments.

Prior work in the area of adversarial machine learning has shown that small-magnitude perturbations added to the training datasets of deep neural networks can result in significant classification errors [32]. These malicious data points are typically generated by attackers with Generative Adversarial Neural Networks (GANNs) that learn based off of the behavior of the targeted neural network. In the case of sandbox evasion that we demonstrate in this paper, attackers create machine learning classifiers using data generated from trusted sources such as physical Android devices and online sandboxes. Thus, sandbox providers who may utilize adversarial machine learning techniques to bypass sandbox evasion classifiers would simply be generating data that matches the distribution of real Android devices. We leave investigation into the use of such techniques to generate realistic Android device usage data to future work.

C. Limitations

Even though our work has uncovered valuable insights into the current state of mobile sandboxes and the weaknesses thereof, it is not without limitations. One limitation is due to the size of our real device dataset, D_{user}. In total, we collected data on 1,245 real Android devices from users from our institution as well as crowd sourcing on the Microworkers platform. While this dataset was sufficient in discovering artifacts of use in mobile devices, an even larger dataset would allow us to discover differences in artifact distributions across various populations (e.g. explore country-specific trends).

However, crowd-sourcing large, representative datasets is not a trivial process. We encountered issues performing our study on certain large crowd-sourcing platforms, due to the services' policy restricting the installation of software on workers' devices.

In order to preserve the privacy of participants in our study, we limit the artifacts we collect to only those that are statistical in nature. Since attackers are not limited in such ways, we expect malware utilizing evasion techniques such as those we presented to be even more effective. For instance, the analysis of package names and installation dates, or contact names and phone numbers is likely to result in even stronger classifiers. Although we did not include these types of features in our study, they serve as examples of minute details that attackers can and will use to evade analysis.

Lastly, this paper focuses on mobile-sandbox evasion on the Android platform. Previous work has demonstrated that environment-based artifacts can be used to track iOS users, similar to how device fingerprinting can be used to track web users [44]. Though we anticipate that the general concept of differentiating between users and sandboxes via environment artifacts will apply equally well to the iOS ecosystem, we leave the actual evaluation to future work.

D. Ethical Considerations

When developing and testing our Android sandbox detection techniques, we opted to create a machine learning classifier completely external to the Android application used to collect data. This approach, modeled after Miramirkhani et al. [48], allows us to collect data once from sandbox providers, and test various classifier hyperparameters without resubmitting our application, reducing strain on sandbox resources. Additionally, embedding the classifier into the application would require us to also package real malware to gauge real evasion success. As our application was submitted to popular Android marketplaces, we opted to remove the possibility of harm to real users who may download our application.

E. Responsible Disclosure

Our research has presented the weaknesses of mobile sandboxes, allowing for malware to effectively bypass analysis. We are in the process of reaching out to all sandbox services that we collected artifacts from and disclosing our findings. We hope that by doing so, sandbox services will begin to patch the artifacts which we currently use in our machine learning classifiers.

VII. Related Work

Sandboxes are a common means of dynamic malware analysis for both mobile and desktop. The described related work includes the study of malware samples that detect and evade sandboxes, as well as the design of resilient sandbox environments.

Mobile malware detection and sandboxes Dynamic analysis techniques and sandboxes are commonly used to study mobile malware [23], [26], [36], [64]. The sandbox environments that we studied in this paper include those from the academia [17], [61], commercial antimalware software [3]–[6], [8], [11], [12], [15], VirusTotal and popular Android application stores [1], [9], [16]. Unfortunately, many of the academic sandboxes are no longer maintained or available [30], [45], [57]. Nevertheless, there is no reason why our approach would not apply to those environments, given their described setup.

Mobile sandbox detection The main body of the work in this area has mostly focused on identifying "red pills" to tell real devices from sandboxes apart, that is, discovering specific API values and behaviors that can give away the presence of an emulated environment [27], [34], [49], [51]–[53], [58]. Spreitzenbarth et al. propose a mobile sandbox architecture and evaluate it against a dataset of Android malware samples [53]. Similarly, Gajrani et al. study the state of the art in emulator detection, and provide a pragmatic design of a sandbox environment based on emulators that is resilient against identification based on common APIs and emulator detection methods [34]. Because of their focus on masking specific API values, sensor readings and emulation detection, their system would not be resilient against our detection scheme that focuses on artifacts of device usage.

Vidas et al. worked on the detection of mobile sandboxes [58]. They propose detection mechanisms through a difference in the returned values of various Android APIs, sensors, and the detection of the underlying emulator hypervisor itself through performance measurements. Along the same line, Petsas et al. incorporate the static properties, dynamic sensor information and VM-related intricacies (QEMU specific) to detect Android emulators [51]. The authors go as far as repackaging a malware sample with their developed heuristics and demonstrate their ability to bypass the 12 existing sandbox environments.

The goal of Spreitzenbarth and Vidas et al. is similar to ours, but they focus on features that are specific to emulated environments such as the hardware performance metrics and artificial or hardcoded values returned by Android APIs (e.g., battery level, build information, etc.). Conversely, we do not focus on making a distinction between real and emulated environments. Rather, our focus is on the artifacts of real device usage compared to sandbox devices, regardless of them being based on a real or emulated setup. Moreover, we systematically analyze a large number of APIs against sandbox and real user devices and build a model capable of detecting the real usage of Android devices.

Costamagna et al. look at the returned value of certain Android APIs and check for the presence of static vs randomized values [27]. Compared to our work, while a portion of the APIs that we looked at is similar, we employ machine learning to model the distribution of values from these APIs on user (collected from real devices) versus sandbox devices.

Finally, Jing et al. proposed a framework to automatically extract heuristics to detect emulator-based Android environments [39]. In their study, the authors compare emulator-based Android sandboxes with real devices from online mobile test farms, similar to our D_{base} dataset. As a result, the focus of their work is to identify emulators in contrast to baseline real devices and produce heuristics in the form of "Red Pills". While their approach suffers from the use of real devices in online sandboxes, in our study, we consider such devices as sandboxes. Given the absence of wear-and-tear artifacts on D_{base} devices, our proposed classifiers mark them as sandboxes with high accuracy.

Our approach focuses on the mobile environment as a whole, including information about the device itself as well as user-specific attributes. We demonstrated that our classifier does not simply detect emulated devices, but is able to further distinguish the minute differences between a device thoroughly used by a real person, as opposed to an artificial sandbox environment. This significantly raises the bar for sandbox providers who must account for a much larger set of features compared to the heuristics introduced in prior work.

Desktop sandboxes The area of desktop sandbox detection and the study of evasive malware predates that of mobile sandbox detection and therefore has received more attention. Dynamic code analysis and use of sandbox environments [29], [31], [38], [42], [59], [60], [62] as well as evasive malware has been studied in [24], [41], [46]. Similarly, system configuration of the target system and its usage in sandbox detection has been the topic of significant study [19], [20], [28], [48], [63].

Most notably, Miramirkhani et al. have looked at device usage artifacts on desktop environments and showed that they can be used to identify desktop sandboxes with high accuracy [48]. They incorporate information from different

parts of the system, network, disk, browser, and registry (which they call wear-and-tear artifacts) to build a statistical model of existing sandboxes. Our work is similar to theirs in that we both incorporate usage-related artifacts to train ML models of normal vs. sandboxed environments. The main distinction of our work apart from our focus on a different platforms (i.e. mobile instead of desktop), is our effort to train manufacturer-specific ML models. Namely, we showed that attackers can capitalize on the fact that mobile devices from the same manufacturer share similar hardware and bloat, by training manufacturer-specific models to increase the accuracy of their classification and therefore their ability to evade sandboxes.

VIII. Conclusion

In this work, we explored the concept of "environment-aware" sandbox evasion where attackers can leverage artifacts present in mobile devices, to differentiate between executions in real-user devices and those in sandboxes. By analyzing tens of artifacts and their distributions, we were able to identify exactly where and how sandboxes fail in their emulation of real Android environments. We showed how attackers can straightforwardly use these artifacts to train a classifier capable of differentiating between artificial and real Android environments with 98.54% accuracy and a 0.57% false-negative rate. These results demonstrate that such a model can be used by malware to hide their activity in the presence of a sandbox, leading to a longer lifetime and greater inflicted damage upon users and systems.

To reinforce our mobile-sandbox classifier, we used device hardware configuration artifacts to create an ensemble of manufacturer-specific classifiers, each trained to detect devices of a particular manufacturer (such as, Samsung and LG). We showed that these classifiers can uncover discrepancies between the stated environment vs. the actual environment in sandboxes and can be combined with the mobile-sandbox classifier to dramatically reduce false negatives.

Our results demonstrate that mobile sandboxes are not effectively emulating real mobile-device environments, allowing mobile malware to bypass analysis. While some sandboxes have glaring weaknesses leading to trivial evasions, the majority simply overlook small details. However, these details are all that is necessary for malware to detect artificial environments. We hope that our work can provide valuable insights into the weaknesses of mobile sandboxes and can help strengthen these services against future generations of environment-aware, mobile malware.

Availability In an effort to strengthen the security of the Android ecosystem, we will release our dataset of real Android device artifacts to sandbox services and security researchers, along with our trained models. We expect that our dataset will enable sandbox services to create more realistic execution environments with manufacturer-specific, artifact distributions. All supplementary materials can be found at https://droid-in-the-details.github.io.

Acknowledgments: We thank the anonymous reviewers for their helpful feedback. This work was supported by the Office of Naval Research (ONR) under grants N00014-20-1-2720 and N00014-21-1-2159, as well as by the National Science Foundation (NSF) under grants CMMI-1842020 and CNS-2126654.

References

[1] Amazon app store. https://developer.amazon.com/apps-and-games.

[2] Aws device farm. https://aws.amazon.com/device-farm.

[3] Bitdefender antivirus. https://www.bitdefender.com/consumer/support/answer/40673/.

[4] Comodo antivirus. https://www.comodo.com/home/internet-security/submit.php.

[5] Drweb antivirus. https://vms.drweb.com/sendvirus/.

[6] escan antivirus. http://support.mwti.net/support/index.php?/Tickets/Submit.

[7] Firebase. https://firebase.google.com.

[8] Gdata - mobile security for android. https://su.gdatasoftware.com/us/sample-submission/.

[9] Google play store. https://play.google.com/store.

[10] ios appstore. https://www.apple.com/app-store/.

[11] Joesandbox cloud. https://www.joesandbox.com/.

[12] Kaspersky - threat intelligence portal. https://opentip.kaspersky.com/.

[13] Microworkers. https://microworkers.com.

[14] Mobile operating systems' market share worldwide from january 2012 to june 2021. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/.

[15] nviso antivirus. https://apkscan.nviso.be.

[16] Samsung galaxy app store. https://www.samsung.com/us/apps/galaxy-store/.

[17] Sanddroid - an automatic android application analysis system. http://sanddroid.xjtu.edu.cn/.

[18] scikit-learn. https://scikit-learn.org/stable/.

[19] A. singh, defeating darkhotel just-in-time decryption. https://labs.lastline.com/defeating-darkhotel-just-in-time-decryption, 2015.

[20] Proofpoint, ursnif banking trojan campaign ups the ante with new sandbox evasion techniques. https://www.proofpoint.com/us/threatinsight/post/ursnif-banking-trojan-campaign-sandbox-evasiontechniques, 2016.

[21] Number of apps available in the google playstore and ios app store. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores, 2020.

[22] 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.

[23] Vitor Afonso, Anatoli Kalysch, Tilo Müller, Daniela Oliveira, André Grégio, and Paulo De Geus. *Lumus: Dynamically Uncovering Evasive Android Applications: 21st International Conference, ISC*. 2018.

[24] Mohsen Ahmadi, Kevin Leach, Ryan Dougherty, Stephanie Forrest, and Westley Weimer. Mimosa: Reducing malware analysis overhead with coverings, 2021.

[25] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.

[26] Alexei Bulazel and Bülent Yener. A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web. In *Proceedings of the 1st Reversing and Offensive-Oriented Trends Symposium*, ROOTS, 2017.

[27] Valerio Costamagna, Cong Zheng, and Heqing Huang. Identifying and evading android sandbox through usage-profile based fingerprints. In *Proceedings of the First Workshop on Radical and Experiential Security*, RESEC, 2018.

[28] D Desai. Malicious documents leveraging new anti-vm & anti-sandbox techniques, 2016.

[29] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, 2008.

[30] Thomas Eder, Michael Rodler, Dieter Vymazal, and Markus Zeilinger. Ananas - a framework for analyzing android applications. *International Conference on Availability, Reliability and Security*, 2013.

[31] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM computing surveys (CSUR)*, 2008.

[32] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. Robust physical-world attacks on deep learning visual classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1625–1634, 2018.

[33] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM, 2011.

[34] Jyoti Gajrani, Jitendra Sarswat, Meenakshi Tripathi, Vijay Laxmi, M. S. Gaur, and Mauro Conti. A robust dynamic analysis system preventing sandbox detection by android malware. In *Proceedings of the 8th International Conference on Security of Information and Networks*, SIN, 2015.

[35] Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez. An analysis of pre-installed android software. *arXiv preprint arXiv:1905.02713*, 2019.

[36] Le Guan, Shijie Jia, Bo Chen, Fengwei Zhang, Bo Luo, Jingqiang Lin, Peng Liu, Xinyu Xing, and Luning Xia. Supporting transparent snapshot for bare-metal malware analysis on mobile devices. ACSAC, 2017.

[37] Matthias Hirth, Tobias Hossfeld, and Phuoc Tran-Gia. Anatomy of a crowdsourcing platform - using the example of microworkers.com. pages 322 – 329, 08 2011.

[38] Xuxian Jiang and Xinyuan Wang. "out-of-the-box" monitoring of vm-based high-interaction honeypots. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2007.

[39] Yiming Jing, Ziming Zhao, Gail-Joon Ahn, and Hongxin Hu. Morpheus: automatically generating heuristics to detect android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 216–225, 2014.

[40] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barebox: efficient malware analysis on bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011.

[41] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barecloud: bare-metal analysis-based evasive malware detection. In *23rd USENIX Security Symposium*, 2014.

[42] Alexander Küchler, Alessandro Mantovani, Yufei Han, Leyla Bilge, and Davide Balzarotti. Does every second count? time-based evolution of malware behavior in sandboxes. In *NDSS*, 2021.

[43] Alexander Küchler, Alessandro Mantovani, Yufei Han, Leyla Bilge, and Davide Balzarotti. Does every second count? time-based evolution of malware behavior in sandboxes. 2021.

[44] Andreas Kurtz, Hugo Gascon, Tobias Becker, Konrad Rieck, and Felix Freiling. Fingerprinting mobile devices using personalized configurations. *Proceedings on Privacy Enhancing Technologies*, 2016.

[45] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. v. d. Veen, and C. Platzer. Andrubis – 1,000,000 apps later: A view on current android malware behaviors. In *Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[46] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. Detecting environment-sensitive malware. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2011.

[47] L. Liu, Y. Gu, Q. Li, and P. Su. Realdroid: Large-scale evasive malware detection on "real devices". In *26th International Conference on Computer Communication and Networks (ICCCN)*, 2017.

[48] N. Miramirkhani, M. P. Appini, N. Nikiforakis, and M. Polychronakis. Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts. In *IEEE Symposium on Security and Privacy (SP)*, 2017.

[49] Sebastian Neuner, Victor van der Veen, Martina Lindorfer, Markus Huber, Georg Merzdovnik, Martin Mulazzani, and Edgar Weippl. Enter sandbox: Android sandbox comparison. *arXiv preprint arXiv:1410.7749*, 2014.

[50] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. {TESSERACT}: Eliminating experimental bias in malware classification across space and time. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 729–746, 2019.

[51] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: Hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, EuroSec, 2014.

[52] Onur Sahin, Ayse K Coskun, and Manuel Egele. Proteus: Detecting android emulators from instruction-level profiles. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2018.

[53] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: Having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC, 2013.

[54] Guillermo Suarez-Tangil, Santanu Kumar Dash, Mansour Ahmadi, Johannes Kinder, Giorgio Giacinto, and Lorenzo Cavallaro. Droidsieve: Fast and accurate classification of obfuscated android malware. CODASPY, 2017.

[55] Guillermo Suarez-Tangil and Gianluca Stringhini. Eight years of rider measurement in the android malware ecosystem. *IEEE Transactions on Dependable and Secure Computing*, 2020.

[56] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)*, 2017.

[57] Kimberly Tam, Salahuddin Khan, Aristide Fattori, and Lorenzo Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS*, 2015.

[58] Timothy Vidas and Nicolas Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS, 2014.

[59] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy*, 2007.

[60] Carsten Willems, Ralf Hund, Andreas Fobian, Dennis Felsch, Thorsten Holz, and Amit Vasudevan. Down to the bare metal: Using processor features for binary analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012.

[61] Lok Kwong Yan and Heng Yin. Droidscope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis. In *21st USENIX Security Symposium*, 2012.

[62] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

[63] Akira Yokoyama, Kou Ishii, Rui Tanabe, Yinmin Papa, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, Daisuke Inoue, Michael Brengel, Michael Backes, et al. Sandprint: Fingerprinting malware sandboxes to provide intelligence for sandbox evasion. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016.

[64] Lun-Pin Yuan, Wenjun Hu, Ting Yu, Peng Liu, and Sencun Zhu. Towards large-scale hunting for android negative-day malware. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2019.

IX. Appendix

Table VIII lists all the artifacts that we extract from mobile environments for the purpose of differentiating between user devices and mobile sandboxes.

TABLE VIII: *Full list of artifacts used in our mobile sandbox machine learning classifier.*

Artifact Name	Artifact Type
ARP	
arpCacheSize	Numeric
Accessibility	
hasAccessibility	Binary
Calendar	
numCalendars	Numeric
numOwner	Numeric
numReadAccess	Numeric
Contacts	
lastCallTime	Numeric
numContacts	Numeric
numCustomRingtone	Numeric
numSentToVoicemail	Numeric
numStarred	Numeric
numTotalCalls	Numeric
numVisible	Numeric
numWithNumber	Numeric
numWithPhoto	Numeric
Dictionary	
dictionarySize	Numeric
dictionaryNumLocales	Numeric
dictionaryAvgFrequency	Numeric
Device	
upTime	Numeric
sdkVersion	Numeric
Events	
avgEventLen	Numeric
numAllDay	Numeric
numCancelled	Numeric
numConfirmed	Numeric
numEvents	Numeric
numRecurring	Numeric
numTentative	Numeric
numWithAlarm	Numeric
numWithData	Numeric
Files	
numDocuments	Numeric
numDownloads	Numeric
PackageData	
numPackages	Numeric
Photos	
avgPicOrientation	Numeric
numPics	Numeric
numPrivatePics	Numeric
Processes	
numProcesses	Numeric
Reminders	
avgMin	Numeric
numAlert	Numeric
numDefault	Numeric
numEmail	Numeric
numReminders	Numeric
Root	
deviceRooted	Binary
Sensor	
gyroscopeSensorReadings	Numeric
lightSensorReadings	Numeric
deviceTemperatureSensorReadings	Numeric
numSensors	Numeric
gravity	Numeric
gyroscope	Numeric
heartBeat	Numeric
heartRate	Numeric
light	Numeric
magnetic	Numeric
motion	Numeric
accelerometer	Numeric
pressure	Numeric
proximity	Numeric
stepCounter	Numeric
Settings	
autoTime	Binary
autoTimeZone	Binary
bluetoothEnabled	Binary
brightness	Numeric
brightnessMode	Numeric
dataRoamingEnabled	Binary
deviceProvisioned	Binary
soundEffectsEnabled	Binary
stayOnWhilePluggedIn	Binary
usbMassStorageEnabled	Binary
vibrateWhenRinging	Binary
wifiOn	Binary
Songs	
avgNumberSongs	Numeric
avgYearLength	Numeric
numAlbums	Numeric
numUniqueArtists	Numeric
Wifi	
5ghtzBandSupported	Binary
deviceToApRTTSupported	Binary
enhancedPowerRoutingSupported	Binary
networkOffloadSupported	Binary
numWifiConfigurations	Numeric
p2pSupported	Binary
scanAvailable	Binary
tdlsSupported	Binary
wifiEnabled	Binary