Experiences of Landing Machine Learning onto Market-Scale Mobile Malware Detection

Liangyi Gong[1], Zhenhua Li[1*], Feng Qian[2], Zifan Zhang[1,3]
Qi Alfred Chen[4], Zhiyun Qian[5], Hao Lin[1], Yunhao Liu[1,6]
[1]Tsinghua University [2]University of Minnesota, Twin Cities [3]Tencent Co. LTD
[4]University of California, Irvine [5]University of California, Riverside [6]Michigan State University

Abstract

App markets, being crucial and critical for today's mobile ecosystem, have also become a natural malware delivery channel since they actually "lend credibility" to malicious apps. In the past decade, machine learning (ML) techniques have been explored for automated, robust malware detection. Unfortunately, to date, we have yet to see an ML-based malware detection solution deployed at market scales. To better understand the real-world challenges, we conduct a collaborative study with a major Android app market (T-Market) offering us large-scale ground-truth data. Our study shows that the key to successfully developing such systems is manifold, including *feature selection/engineering*, *app analysis speed*, *developer engagement*, and *model evolution*. Failure in any of the above aspects would lead to the "wooden barrel effect" of the entire system. We discuss our careful design choices as well as our first-hand deployment experiences in building such an ML-powered malware detection system. We implement our design and examine its effectiveness in the T-Market for over one year, using a single commodity server to vet ~10K apps every day. The evaluation results show that this design achieves an overall precision of 98% and recall of 96% with an average per-app scan time of 1.3 minutes.

CCS Concepts: • **Security and privacy → Mobile platform security**; **Malware and its mitigation**.

ACM Reference Format:
Liangyi Gong, Zhenhua Li, Feng Qian, Zifan Zhang, Qi Alfred Chen, Zhiyun Qian, Hao Lin, Yunhao Liu. 2020. Experiences of Landing Machine Learning onto Market-Scale Mobile Malware Detection . In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3342195.3387530

*Corresponding author.

EuroSys '20, April 27–30, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387530

1 Introduction

App markets such as Google Play and Amazon AppStore play an important role in today's mobile ecosystem, through which the majority of mobile apps are published, updated, and distributed to users. On the flip side, the markets have also become a convenient channel to spread malware. Even worse, most attackers prefer to use the channel due to the fact that when an app is published in a well-known app market, it "lends credibility" to the app. Additionally, mobile devices are often pre-configured to allow apps to be installed from only app markets by default [50]. Hence, automated market-scale malware detection is necessary by all major commercial app markets today [18].

In the past decade, machine learning (ML) techniques have been widely explored for malware detections, since they do not rely on specific rules so that they are considered more automated and robust. There exist a plethora of ML-based techniques in the literature, from simple fingerprint-based antivirus checking [50], static code inspection [7], to sophisticated dynamic behavior analysis [42]. Unfortunately, we have not yet seen any report on the effectiveness of such solutions when they are applied at a market scale.

In this paper, we present our experiences on building and deploying an ML-powered solution. Together with a major Android app market, *i.e.,* Tencent App Market[1] or T-Market for short, we are able to obtain full access to the large-scale ground-truth data of apps (both published and rejected) and their corresponding labels. By comprehensively analyzing the data and existing ML-based malware detection solutions, we show that the key challenges lie in multiple aspects: *feature selection*, *feature engineering*, *app analysis speed*, *developer engagement*, and ML *model evolution* over time.

More importantly, we note that failure in any of the above aspects would lead to the "wooden barrel effect" [40] (and thus vain endeavors). For example, feature selection affects not only the detection accuracy but also the time it takes to analyze an app (both of which have stringent requirements on a market-scale solution). Further, feature engineering determines the detection robustness and the difficulty of model evolution. Additionally, both detection accuracy and speed impact developers' engagement in app submissions, which is crucial to the prosperity of a commercial app market.

[1]https://sj.qq.com/myapp/

In order to build a desirable ML-powered malware detection system at market scales, we choose to **focus on a lightweight and scalable feature extraction design: *API-centric dynamic analysis***, which monitors Android API invocations at an app's runtime to achieve high analysis speed. Given that Android SDK APIs provide almost all functions for a typical app, they remain the de facto feature choice in nearly all previous studies [2, 31, 33, 39]. Indeed, our study shows that in many cases, using more complex features does not bring a noticeably higher detection accuracy [1, 42, 46].

Next, we take **a principled, data-driven approach for the concrete selection of (API) features**, noticing that the current Android SDK provides >50,000 APIs. The rationale is threefold. First, the number of selected APIs has unnegligible impact on the dynamic analysis time (up to 50× difference). Second, compared to monitoring all 50K APIs, strategically monitoring a smaller number of APIs yields a better detection accuracy, possibly due to the reduced likelihood of over-fitting. Third, APIs identified from different sources complement each other; judiciously combining them helps considerably improve the detection accuracy. Based on the above considerations, we select a total number of 426 key APIs as ML features, and employ the lightweight *random forest* ML algorithm (among a variety of ML algorithms), which provides the best detection accuracy (96.8% precision and 93.7% recall) in our large-scale dataset from T-Market.

To further improve the detection accuracy, we take **an adversary's perspective to identify hidden features**. The limitation of purely relying on Android APIs for malware detection is that, to achieve the functionality of certain APIs, an attacker can bypass the API invocations and use other mechanisms such as Java reflection and intents (Android's IPC mechanism). From our dataset, we observe both mechanisms have been employed by malicious apps to hide their certain features. To account for this, we also capture the requested permissions and the used intents in dynamic analysis. The captured "indirect" features are then combined with the API invocation features to obtain a more complete picture of an app's runtime behavior. After applying this enhancement, the detection precision and recall are further improved to 98.6% and 96.7% over our dataset.

Having derived the desired features, we shift our focus to improving the runtime performance of app analysis. We **architect the app emulation system to efficiently run on powerful x86 servers** (§5.1). To boost the runtime performance, we run the native x86 port of Android OS [21] and translate apps' native code from ARM to x86 (using the state-of-the-art dynamic binary translation framework developed by Intel [22]). Compared to the full-system binary translation of ARM-based Android OS and apps (using Google's QEMU-based Android emulator), we can achieve 70% reduction of app execution time. The system also becomes more reliable through app crash detection and emulator fallback.

False positives and negatives are generally inevitable for ML-based systems. In APICHECKER, false positive apps (as complained by developers) and false negative apps (as reported by end users) are both manually analyzed but in distinct manners. We choose to actively avoid the former (on a daily basis), as it significantly increases the burden of manual intervention to address developers' complaints. Usually, ~90% of the flagged malicious apps are updated apps that can be quickly vetted based on their previous versions, so the totally required manual inspection is practically affordable. In contrast, we can hardly avoid the latter and thus only conduct manual analysis upon user reports. Fortunately, we observe that **the existence of a small number of false negative apps in fact has little effect on the regular operation of T-Market**. Manual inspection shows that 87% of the sampled false negative apps barely use the key APIs we select to monitor, and thus have fairly simple functionalities without posing a great security threat to end users.

We embodied all above efforts into a real-world system called APICHECKER, and it has been operational at T-Market since Mar. 2018. Running on a single commodity server, it can vet ~10K newly submitted apps every day. APICHECKER takes 1.3 minutes on average to scan a submitted app, which is generally acceptable to the developers (considering that the typical per-app scan time is ~5 minutes in Google Play [26]). The overall achieved precision and recall have been above 98% and 96% since its first deployment, owing to **our automatically updating the ML model with new apps and novel Android SDK APIs (if any) on a monthly basis** (§5.3). As the model evolves, the number of selected key APIs only slightly fluctuates between 425 and 432.

Overall, this study showcases several key design decisions we make towards implementing, deploying, and operating a production market-scale mobile malware detection system. To our knowledge, this is the first study at such a large-scale. At a high level, our experiences indicate that machine learning, despite being a powerful building block for mobile security, should be strategically applied by considering multiple dimensions of accuracy, performance, generalizability, long-term robustness, and deployability. Because of these considerations, for example, we make a judicious decision of not using deep learning for APICHECKER but random forest, which achieves a comparable classification accuracy while owning other advantages of minor training time, lightweight operations, and good interpretability. We also demonstrate a systematic approach for feature construction, a procedure that is perhaps more important than the ML model selection itself in the context of automated security analysis.

Dataset and tool release. The list of our selected 426 key framework APIs is available at https://apichecker.github.io/. We will also release our analysis logs, as well as our implemented efficient emulator, to the research community once we get approvals from T-Market.

Ethical consideration. All analysis tasks in this study comply with the agreement established between T-Market and the developers who publish their apps to T-Market. To protect developers' privacy, when referring to individual apps as examples or case studies, we anonymize their names.

2 Background and Motivation

In this section, we first introduce the basics of Android app development for a better understanding of mobile malware detection. Then, we describe the malware defense mechanisms employed by T-Market and other app markets, as well as the associated challenges.

Android app development basics. Android is an open-source operating system based on the Linux kernel and ARM architecture. An Android app is usually written in Java, compiled into Dex (Dalvik Executable) bytecode, and then compressed and deployed as an Android Package (APK) archive. Specifically, an APK file contains the configuration file, the compiled code, and some other files. In particular, the configuration file `AndroidManifest.xml` stores the meta-data (*e.g.,* package name, permissions requested, Android components declaration and dependencies), and the compiled code `classes.dex` contains the information of API usage. An app runs in its own Dalvik or ART (Android Runtime) virtual machine instance, and the virtual machine instance runs as a Linux process with a unique UNIX user ID and some group IDs corresponding to the permissions, so that the access for an app to system resources is strictly limited. An app must request permissions to sensitive user data and certain system features in the Android configuration file (`AndroidManifest.xml`). Such a security model in Android ensures that each app is running in a sandbox.

For Android app development, rich APIs are provided by the Android framework (which contains the entire feature set of the Android OS) to enable an app's interactions with other apps and with the system. In order to ensure the integrity of an app, the app needs to request permissions in its configuration file for sensitive user data and certain system features. Only when these permissions are granted by the system (and sometimes even the users) can the corresponding code (and APIs) be executed. In addition, inter-process communications (IPCs) in Android are implemented by a mechanism called Binder, which are employed to perform remote procedure calls (RPCs) from one Java process to another; this mechanism also enables framework APIs to interact with Android system services. A developer just needs to pass a message-like object, such as an Intent, to fulfill an IPC through certain APIs. Through the above mechanisms, most functions of the Android system are exposed to developers. On the flip side, although an app runs in a sandbox, the richness of APIs has been exploited by attackers to develop malicious apps, which pose a persistent threat to the Android ecosystem.

Current market-level defense mechanisms and challenges. In today's app markets, malicious apps use various means (*e.g.,* repackaging and update attack) to disguise themselves as benign [25], induce users to download and install from the app markets, and thus conduct malicious behaviors that are difficult to detect. To combat such threats, Google launched its proprietary "Bouncer" system [26] in 2012 to scan apps uploaded to Google Play, reducing the number of malicious apps on the platform by 40%. However, due to the existence of a great many third-party app markets without such malware detection, malicious apps still managed to spread.

In order to deeply and practically explore the issue, in this paper we collaborate with T-Market, a third-party app market that has released over 6M apps since its launch in 2012, with over 30M APKs being downloaded by 20M users every day. To protect its users, T-Market reviews both new and updated apps submitted from developers on a daily basis.

To accurately determine the malice of hosted apps, T-Market introduced in 2014 a sophisticated app review process mainly consisting of 1) fingerprint-based antivirus checking, 2) expert-informed API inspection, and 3) user-report-driven manual examination. First, antivirus checking inspects apps against virus fingerprints [50] from antivirus service companies including Symantec, Kaspersky, Norton, McAfee, *etc.* , and those collected by T-Market itself. Second, API inspection identifies malicious apps by monitoring the invocations of a group of selected APIs in the app code. These APIs are selected by security experts based on their experiences, where the intuition is that certain invocation patterns (combinations and orders) of these APIs imply potential security threats [1]. Third, after these two steps, there are still false positives and false negatives. T-Market currently relies on developers to report false positives, and end users to report false negatives. For these reported cases, manual inspection is then performed to determine the malice.

In this detection process, the fingerprint-based antivirus checking can only detect known malware samples. Thus, the most critical defense task, detecting zero-day malware, mainly relies on the API inspection and manual inspection steps [20]. As manual inspection is slow (it may take a couple of days to analyze an app), T-Market is highly interested in improving the API inspection step and achieving a comparable performance as the "Bouncer" system. In particular, it wishes to know whether today's popular ML techniques are capable of achieving the target, which are commonly expected to be more automated and robust since they do not rely on specific rules (coming from security experts).

3 Related Work

For Android app development, rich APIs are provided by the Android framework to enable an app's interactions with other apps and with the system. This section reviews prior API-based malware detection solutions using ML techniques,

API Selection Strategy	Related Work	Analysis Method	Analysis Time per App	# APIs Used	# Apps Studied	Precision, Recall
Statistical Correlations	Sharma *et al.* [35]	static	- -	35	1,600	91.2%, 97.5%
	DroidAPIMiner [1]	static	25 sec	169	~20K	- -
Restrictive Permissions	Stowaway [15]	static	- -	1,259	964	- -
	DroidMat [43]	static	- -	- -	1,738	96.7%, 87.4%
	Yang *et al.* [46]	dynamic	1080 sec	19	~27K	92.8%, 84.9%
Sensitive Operations	RiskRanker [20]	static	41 sec	- -	~118K	- -
	DroidCat [9]	semi-dynamic	354 sec	27	~34K	97.5%, 97.3%
	IntelliDroid [42]	static + dynamic	138.4 sec	228	2,326	- -
	Droid–Sec [49]	static + dynamic	- -	64	250	- -
	DroidDolphin [44]	dynamic	1020 sec	25	64K	90%, 82%
Hybrid	DREBIN [6]	static	10 sec	- -	~128K	- -
	APICHECKER	**dynamic**	**78 sec**	**426**	**~500K**	**98.6%, 96.7%**

Table 1. Representative Android malware researches that detect malicious apps by studying a selected set of potentially useful framework APIs. "- -" means unknown.

in terms of both static and dynamic approaches of gathering an app's API usage information. We also compare existing solutions with our work from different perspectives.

Static analysis. The static API usage information can be directly extracted from an app's APK. After that, machine learning or rule-based methods can be applied to determine the malice of apps. For instance, Sharma *et al.* [35] extract from 1,600 apps 35 APIs that are correlated with the malice of apps, and combine Naive Bayesian and kNN classifiers to achieve 91.2% precision and 97.5% recall for malware detection. Other representative work includes DroidAPIMiner [1], Stowaway [15], DroidMat [43], Droid–Sec [49], RiskRanker [20], and DREBIN [6].

DroidAPIMiner [1] extracts critical APIs according to their usage frequencies, and compares the performance of four machine learning classifiers where kNN achieves the best performance with 99% accuracy and 2.2% false positive rate. In terms of analysis speed, it requires 25 seconds on average to classify an APK file. Stowaway [15] is an automated static analysis tool that extracts from 964 apps 1,259 APIs with *restrictive permission*, based on which a "permission map" is built for apps' over-privilege detection. The same strategy is also adopted by DroidMat [43] to determine the malice of 1,738 apps. DroidMat [43] employs the k-means algorithm to enhance the malware classification model (kNN).

RiskRanker [20] performs a two-order risk analysis to assess 118K apps (taking around 41 seconds per app), and eventually reports 3,281 risky apps by identifying certain rules of seemingly innocent API uses that may well be indicators of malware. Further, Droid-Sec [49] proposes an ML-based method that utilizes 64 sensitive APIs extracted from static analysis and 18 behaviors from dynamic analysis of 250 Android apps for malware detection. It also presents a comparison between the deep learning model and other five classic machine learning models, and achieves the highest accuracy of 96.0% using the deep belief network.

Adopting a *hybrid* strategy, DREBIN [6] gathers numerous features from 129K apps, including permission-restricted APIs, suspicious APIs (that may relate to sensitive operations), requested permissions, network addresses, and so on. It then takes 10 seconds on average to collect features on a real device, and identifies certain patterns of input features with the support vector machines (SVM) classifier. DREBIN does not report the gathered APIs or the detection accuracy.

Dynamic analysis. Owing to its immunity to code obfuscation and dynamic code loading, dynamic analysis can provide a deeper and more complete view of apps' behaviors. For instance, Yang *et al.* [46] develop a dynamic app behavior inspection platform by examining the run-time use of 19 APIs that are restricted by three special types of permissions with regard to obtaining device/system information, accessing the network, and charging from the user's account; they examine each app for around 18 minutes and use a SVM model to achieve 92.8% precision and 84.9% recall. There exist a wide range of similar dynamic analysis systems in the literature, such as DroidDolphin [44], IntelliDroid [42], TaintDroid [14], and DroidCat [9].

DroidDolphin [44] builds a dynamic analysis framework based on big data analysis and the SVM machine learning algorithm by checking the use of 25 APIs and 13 types of sensitive operations of an app in around 17 minutes, and achieves 90% precision and 82% recall. IntelliDroid [42] extracts from 2,326 apps specific API call paths and sensitive event-chains with respect to 228 "targeted" APIs that may facilitate sensitive operations identified by TaintDroid [14]. Based on the extracted information, it detects malware through dynamic analysis in an average of 138.4 seconds per app. Moreover, DroidCat [9] leverages 122 behavioral features (including manually picked APIs, inter-component communication, and

potentially risky sources/sinks) together with a random forest classification model, achieving as high as 97.5% precision and 97.3% recall. For each app, the average time consumption for feature computation and testing is 354 seconds. Unfortunately, it is unable to deal with the case of dynamic code loading, thus substantially degrading its generality.

Given that dynamic analysis often requires considerable time, some studies employ methods to reduce the detection time, such as only inspecting the ~370 Linux system calls underlying the 50K APIs to cut the time cost [38]. These methods are typically quite ad-hoc, and oftentimes sacrifice the detection accuracy due to the loss of APIs' expressiveness of apps' semantics.

Comparison with our work. Despite adopting a similar *API-centric analysis approach*, our work differs from existing studies in several aspects: First, the scale of our measurement study (in terms of the number of studied apps) is much larger. Second, we bring innovations to API selection, and identify hidden features to further boost the accuracy. Third, we optimize the dynamic execution (emulation) infrastructure to significantly reduce the app analysis time and meanwhile guarantee the runtime reliability. Fourth, we manage to commercially deploy our system and timely update the ML model. In a nutshell, our work provides the first practical and comprehensive solution to ML-based malware detection at market scales with commercial deployment results reported.

4 Collaborative Study

In this section, we take a principled big-data-driven approach to study the ML-powered malware detection, using a ground-truth dataset (§4.1) of ~500K apps from T-Market. Moreover, we build an API-centric dynamic analysis engine (§4.2) to hook API invocations at an app's runtime and examine different classification models. After that, we report the basic study results in §4.3, which effectively guide us on the feature selection (§4.4) and feature engineering (§4.5).

4.1 App Dataset

Our dataset contains 501,971 new and updated apps submitted to T-Market in 10 months (from March to December 2017). Note that ~85% of the apps in our dataset are updated apps initially submitted to T-Market as early as in 2014. Moreover, APKs with the same package name but different MD5 hash codes are taken as different apps. In particular, T-Market provides a malice label (Malicious or Benign) for each app by a rather sophisticated and effective app review process, as introduced in §2. These labels are obtained with at least four state-of-the-art fingerprint-based antivirus checking[2], empirical API inspection, and manual examination triggered by developers and users' feedback. Consequently, despite containing a trivial portion of falsely-labeled apps, our dataset is generally able to provide credible, unbiased ground truth. In total, there are 463,273 benign apps and 38,698 malicious apps in the dataset; the malicious apps are not released to users but quarantined in T-Market's database.

[2]In detail, the false positive rate claimed by each antivirus checking engines is less than 5%. When they all label an app as malicious, T-Market takes the app as malicious; else, T-Market manually examines the app. Consequently, the falsely-labeled apps in our training set should not exceed $(1 - 95\%)^4$.

4.2 Dynamic Analysis Engine

In order to intercept and log the run-time invocations of framework APIs, we construct a dynamic analysis engine based on Google's official Android emulator [5], as well as the Xposed hooking framework [32] which can intercept a target API before it is actually invoked. Meanwhile, we explore each app using the *Monkey* UI exerciser [4] that can generate UI event streams at both application and system levels. In addition, we apply nine mainstream machine learning algorithms as listed in Table 2. They include Naive Bayes (NB), CART decision tree, logistic regression (LR), *k*-nearest neighbor (kNN), support vector machine (SVM), gradient boosting decision tree (GBDT), artifical neural network (ANN), deep neural network (DNN), and random forest (RF). We select them because they have been utilized in existing studies and systems [1, 6, 9, 35, 43, 44, 46, 49]. We apply them to the collected logs to derive an appropriate classification model for determining the malice of apps. Here a key design decision is to use *provably mature* program analysis (*i.e.,* a well-received tool chain for app behavior analysis, with respect to app emulation [5], API hooking [37], and UI testing [3]) and machine learning techniques as our building blocks, since our engine will be part of the commercial system, and it is supposed to accommodate all apps hosted by the app market.

App emulation environment. We deploy Google's Android emulators on a cluster of (16) commodity x86 servers (HP ProLiant DL-380 running Ubuntu 16.04 LTS 64-bit), each of which is equipped with a 5×4-core Xeon CPU @ 2.50 GHz and 256-GB DDR memory. On each server, we run 16 emulators on 16 cores concurrently and the remaining 4 cores are used for task scheduling, status monitoring, and information logging. Then, the ~500K apps are parallelly run on the emulators with the Android debug bridge (*adb*) tool. Specifically, for each app, we sequentially execute *adb* commands to automatically install the app, run the *Monkey* UI exerciser, record the running logs, uninstall the app, and clear up the residual data. We measure the overall emulation time of an app using only the execution time of *Monkey*. Moreover, using Xposed we can not only intercept the invocations of target APIs (and record their names and parameters), but also implement the callback interface to perform additional operations (*e.g.,* hooking a certain `Activity`, and tampering with the return values to bypass user login or simulate a real device's behaviors), to facilitate the emulation.

During the emulation, we notice that some malicious apps attempt to recognize whether they are running on emulators so

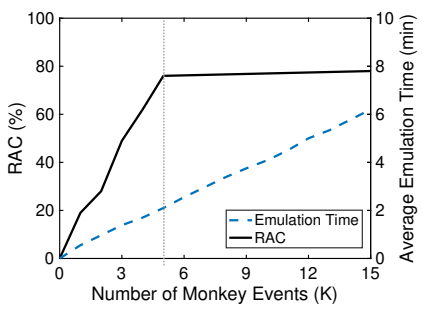

Figure 1. Relationship among # *Monkey* events, RAC, and emulation time.

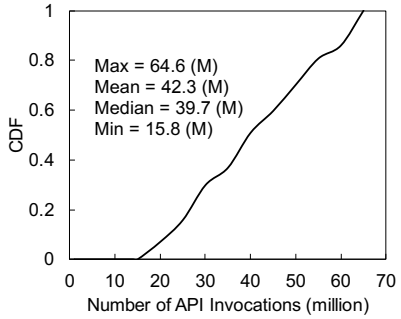

Figure 2. CDF of the number of API invocations when emulating one app.

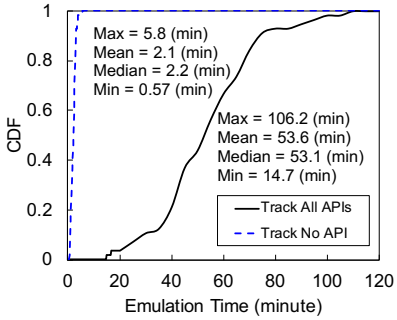

Figure 3. Time consumption for tracking all APIs and no API.

as to suppress their malicious activities. They usually examine the static configurations of the Android system, the dynamic time intervals of user inputs, and the sensor data of the user device to identify the existence of emulators. To prevent such detections, we make fourfold improvements to our emulators as follows:

- First, we change the default configurations of emulators, including their identities (`IMEI` and `IMSI`), `PRODUCT`/`MODEL` types, and network properties (*e.g.,* the TCP information maintained in `/proc/net/tcp`).
- Second, we tune the execution parameters `throttle` and `pct-touch` of *Monkey*, which respectively control the input interval and the percentage of touch events in all inputs, to make the generated UI events look more realistic. Specifically, `throttle` is set to 500 ms (the average interval of human inputs), and `pct-touch` is set within 50%~80% according to the app type, since touch events usually dominate common users' daily inputs.
- Third, we replay traces of sensor data (*e.g.,* accelerometer, gyroscope) collected from a number of real smartphones on our emulators to improve their authenticity [19].
- Finally, we obfuscate the relevant libraries of Xposed and change the return values of certain methods (for instance, `getInstalledApplications`) of the very important `PackageManager` class, so that the studied apps can hardly detect the existence of Xposed.

In order to quantify the effect of our enhanced emulation environment, we construct a controlled experiment with the same sample set of apps running on top of real Android devices (Google Nexus 6), the original emulator, and our enhanced emulator. Specifically, we run an unbiased subset (1%) of our dataset (the ~500K apps) in the three environments. We observe that on the original emulator only 86.6% of apps invoke the same number of APIs as (they invoke) on the real Android devices, while on our enhanced emulator as many as 98.6% of apps invoke the same number of APIs as on the real Android devices. The remainder (1.4%) invoke fewer APIs due to their requirement of real-time data from special sensors (*e.g.,* microphone), which currently cannot be generated by our emulation environment. This demonstrates the efficacy of our improvements to the app emulation environment.

UI exploration methodology. Our dynamic analysis engine needs to achieve a high UI coverage to emulate as many user activities as possible. Initially, we used `Activity` coverage as the main metric of UI coverage [27], as each Android app specifies its possible `Activity` objects in its configuration file (`AndroidManifest.xml`). Nonetheless, this metric is overly pessimistic because it takes into account some `Activities` that are not actually referenced by the code.

Hence, to figure out the ratio of specified `Activities` that are actually referenced by an Android app, we write a script to automatically scan the configuration file and the code of each non-obfuscated APK in our dataset. The scanning results indicate that on average, only 88% of specified `Activities` are actually referenced. Furthermore, we define a more accurate metric, Referred Activity Coverage (RAC), to quantify the UI coverage. RAC is the ratio between the number of detected (actually used) `Activities` during an app's emulation and the number of app's referenced `Activities`.

The actually used `Activities` during an app's emulation can be detected by Xposed [32]. Quantitatively, we find that executing ~100K *Monkey* events can achieve almost 86% RAC on average – executing more *Monkey* events can hardly increase the RAC. However, it requires 2,142 seconds (35.7 minutes) on average to execute 100K *Monkey* events. Such a long emulation time is unacceptably long to both app store operators and developers in practice (considering that Google Bouncer only requires less than 5 minutes to analyze each app submission [26]). To address this problem, we need to carefully balance the effectiveness (in terms of RAC) and the efficiency (in terms of emulation time).

Figure 1 shows the average RAC achieved with increasing number of *Monkey* events. In detail, we notice that as the emulation time increases, the average RAC quickly increases

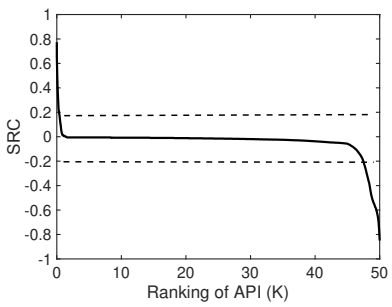

Figure 4. Ranking of the ~50K framework APIs in terms of their *SRC*s.

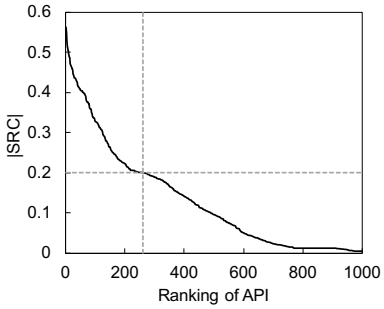

Figure 5. Top-1K APIs (that are not seldom invoked) in terms of $|SRC|$.

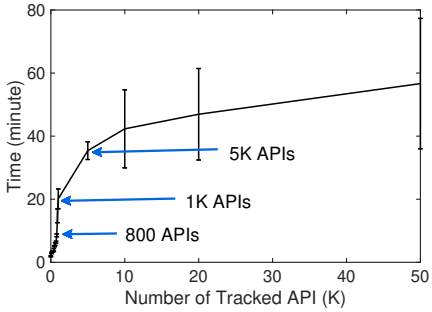

Figure 6. Time consumption for tracking top-n correlated APIs.

to 76.5% within 126 seconds on average. Afterwards, the increase is rather flat – even spending ~250 seconds to generate 10K *Monkey* events merely increases the RAC by 1.5% on average. As a result, we choose to run the emulation for 126 seconds (= 2.1 minutes, corresponding to 5K *Monkey* events), so as to achieve a decent RAC (76.5%). In other words, we choose to sacrifice a small fraction (9.5%) of RAC to reduce a large fraction (94%) of the emulation time compared to executing 100K *Monkey* events.

In addition, Figure 2 depicts how many API invocations occur during the emulation of one app. In general, 5K *Monkey* events trigger tens of millions of API invocations, *i.e.,* one *Monkey* event triggers an average of 8,460 API invocations, indicating the intensive usage of APIs when an Android app is running. Furthermore, as illustrated in Figure 3, when we run an app without tracking any APIs, the time consumption is only 2.1 minutes on average. But when we track the usage of all the ~50K APIs, the time consumption significantly increases to an average of 53.6 minutes due to the overhead of intercepting the huge number of API calls. Obviously, tracking all APIs would be time-wise infeasible. We investigate how to judiciously select a subset of APIs in §4.4.

Machine learning classification model. Like most existing work as surveyed in §3, we adopt machine learning to classify an app as malware or non-malware. During the emulation of each app from our dataset, the invocation data of the tracked APIs (API calls' names and parameters) is logged. We employ *One-Hot* encoding to convert the log to a feature vector comprising a total of n bits, where n is the total number of our tracked APIs. Each bit corresponds to a tracked API – if the API is invoked, the corresponding bit is set as 1; otherwise it is 0. All the feature vectors of our studied apps are used as the training and test sets (which are of course disjoint).

There are three key metrics to evaluate a machine learning model: precision, recall, and training time. The precision and recall are defined as: $precision = \frac{TP}{TP+FP}$ and $recall = \frac{TP}{TP+FN}$, where true positive (TP) denotes the number of apps correctly classified as malware, while false positive (FP) and false negative (FN) indicate the number of apps mistakenly identified as malicious and benign, respectively [29].

Following common practices, we calculate the precision and recall to evaluate the malware detection. In total, we use scikit-learn [23] to realize nine mainstream machine learning models (*cf.* §4.2). In our experiments, the hyperparameters of each model are configured based on our domain knowledge.

Further, when evaluating an ML model, we adopt 10-fold cross-validation to mitigate possible *data leakage* [24] in the training and testing stages. Here *data leakage* means that the training set gains access to the test set, *i.e.,* identical or similar data exist in both sets, thus leading to exaggerated evaluation results. Compared to a single random train/test split, 10-fold cross-validation enables us to obtain less biased results by training and testing the model with several different train/test splits. Meanwhile, for each iteration of the cross-validation, we remove duplicate feature vectors in the training and test sets from the test set. Additionally, we examine the percentage of *cloned* apps (*i.e.,* apps having the same package names but different MD5 hash codes, which could also lead to data leakage) in the training and test sets, and find it to be fairly small (<1%). The concrete model configurations are released at https://apichecker.github.io/.

4.3 Understanding Tradeoffs for API Selection

We use the above dynamic analysis engine to evaluate the tradeoff between API (feature) selection and malware detection time/detection accuracy. First, we rank all the ~50K APIs by their correlations with the malice of apps [30]. Next, we acquire a series of insights with regard to the analysis time, ML-based classification model, malware detection accuracy, and API selection strategies.

APIs' correlations with the malice of apps. The correlation between an API and the malice of apps is an objective metric of statistical measurements [35]. In this paper, we utilize the *Spearman's rank correlation coefficient* (*SRC* [30]) to evaluate the statistical correlation. Using our dynamic analysis results, we calculate the *SRC* value of each API with the malice of apps, and all the extracted APIs are ranked by

Models	Precision (50K / 426)	Recall (50K / 426)	Training Time (50K / 426)
Naive Bayes [35]	60.4% / 64.1%	59.6% / 63.6%	3.6 min / 1.7 sec
Logistic Regression [49]	81.2% / 89.9%	70.3% / 72.4%	10.4 min / 4.5 sec
SVM [6, 44, 46]	87.9% / 96.2%	71.6% / 80.1%	~27K min / 13K sec
GBDT	88.4% / 96.2%	74.3% / 77.9%	364 min / 174 sec
kNN [35, 43]	86.5% / 95.3%	83.7% / 93.3%	~1.8K min / 821 sec
CART [1]	87.6% / 94.3%	84.3% / 93.7%	11.6 min / 5.8 sec
ANN [49]	90.8% / 96.0%	89.9% / 93.4%	~1.2K min / 563 sec
DNN	91.5% / 96.4%	90.9% / 93.7%	~1.9K min / 944 sec
Random Forest [9]	91.6% / **96.8**%	90.2% / **93.7**%	29.1 min / 14.4 sec

Table 2. Performance and overhead of different classification models when we track 50K *vs.* 426 key APIs.

their *SRC* values in descending order in Figure 4. We find there are 247 APIs whose $SRC \geq 0.2$, and 2,536 APIs whose $SRC \leq -0.2$. Note that when $|SRC|$ of a feature is smaller than 0.2, it is considered to have a very weak or no relationship with the malice of apps [36]; in other words, when $|SRC| \geq 0.2$, the correlation is considered non-trivial.

Among the above 247 APIs whose *SRC* is greater than 0.2, we find that some of them are correlated with each other with regard to certain apps, thereby being somewhat redundant when used to analyze these apps. Nevertheless, these APIs often complement each other in terms of functionality rather than being mutually replaceable. Thus, they are in fact beneficial and necessary to our analysis. Moreover, as we carefully examine the 2,536 APIs whose *SRC* is smaller than −0.2, we notice that most of them are *seldom* invoked by the apps in our dataset. Here we empirically take *seldom* as being invoked by fewer than 0.1% apps in our dataset. Using these infrequently invoked APIs as features may bring over-fitting problems to machine learning, and thus we have to neglect these APIs when conducting API selection. On the other hand, we do observe that 13 APIs whose $SRC \leq -0.2$ are frequently invoked by most apps to perform common Android operations like file I/O; such APIs are still taken into account in our analysis. In detail, Figure 5 shows the top-1K framework APIs that are not seldom invoked in terms of their $|SRC|$s with the malice of apps. From the figure we find there are 260 APIs (247 APIs with $SRC \geq 0.2$ and 13 APIs with $SRC \leq -0.2$, referred to as **`Set-C`**) that possess a non-trivial $|SRC|$.

Analysis time. Figure 6 shows the relationship between the number of tracked APIs (n) and the analysis time (t), when we prioritize tracking highly correlated APIs (with the malice of apps) that are not seldom invoked.

To understand the statistical relationship demonstrated in Figure 6, we propose a complex tri-modal distribution and find it can fit the data well. Concretely, t first linearly grows with n when $n \in [1, 800)$, with the associated APIs being used with moderate frequency, more likely by malware due to their high *SRC*s. Then, t polynomially grows when $n \in [800, 1K]$ due to the enrollment of APIs which are heavily used by both malware and non-malware, and therefore they are less expressive in terms of characterizing malicious behaviors. Finally, t logarithmically grows when $n > 1K$ because the newly added APIs have low invocation frequencies. We use the following tri-modal distribution to fit $t(n)$:

$$t = \begin{cases} a_1 \cdot n + b_1, & n \in [1, 800); \\ a_2 \cdot n^{b_2}, & n \in [800, 1K]; \\ a_3 \cdot log(n) + b_3, & n > 1K. \end{cases} \quad (1)$$

where $a_1 = 0.006$, $b_1 = 2.06$, $a_2 = 10^{-9}$, $b_2 = 3.44$, $a_3 = 6.4$, and $b_3 = -43.36$. Also, we adopt the *coefficient of determination* [34] (denoted as R^2, ranging from 0 to 1) to measure the closeness between the measured data and the three fitting equations. We calculate $R_1^2 = 0.96$, $R_2^2 = 0.99$, and $R_3^2 = 0.99$, which are very close to 1.0 (the perfect fitting).

The above result indicates that the relationship between APIs' *SRC* and their invocation frequency (and thus the incurred dynamic analysis overhead) is rather complex. Quantitatively understanding it helps better balance the tradeoff between analysis time and detection accuracy. Regarding the former, Figure 6 indicates that with our dynamic analysis engine, up the top-490 APIs can be tracked to achieve an average detection time of less than 5 minutes per app. We explore the latter (the accuracy dimension) shortly.

Machine learning models. We evaluate the impact of the nine machine learning models introduced in §4.2, assuming all 50K APIs are tracked. The performance and overhead of each classifier are listed in Table 2. We find that these classifiers exhibit significant differences in terms of precision, recall, and in particular training time; however, no single classifier achieves the best in all the three key metrics. Therefore, we choose to adopt the classifier that makes the best balance among the three metrics, *i.e.,* Random Forest, which yields the best precision, the better recall, and acceptable training time. In addition, RF is also known to have a good generalization ability [28]. As a matter of fact, when we only track top-1K or top-490 APIs (refer to Figure 6) to collect data, our best choice is still the random forest classifier.

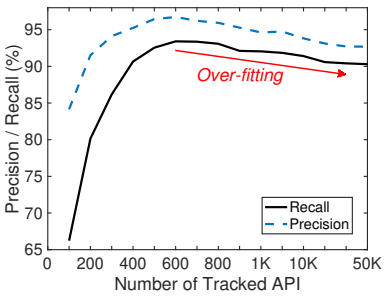

Figure 7. Efficacy for tracking top-*n* correlated APIs respectively.

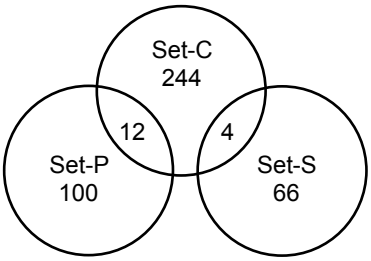

Figure 8. Number of APIs in **`Set-C`**, **`Set-P`**, **`Set-S`** and their overlaps.

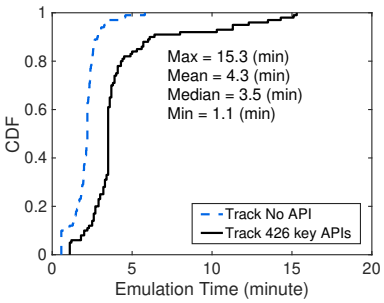

Figure 9. Time consumption for tracking 426 key APIs.

Malware detection accuracy. Tracking all the 50K APIs, despite being the most time-consuming, is supposed to produce the best accuracy. Figure 7 shows the detection accuracy achieved by tracking the top-*n* correlated APIs based on the random forest classifier. The precision/recall achieved by tracking 50K APIs, top-1K correlated APIs, and top-490 correlated APIs is 91.6%/90.2%, 94.7%/92.0%, and 96.3%/92.4%, respectively. Somewhat to our surprise, (strategically) tracking fewer APIs can result in better precision and recall than tracking all the 50K APIs. Delving deeper, we find this counter-intuitive observation stems from the fact that most APIs are sparsely or rarely invoked by Android apps, and thus too many features cause over-fitting of the trained model. In other words, tracking fewer APIs can bring benefits in terms of both runtime performance and detection accuracy.

4.4 Key API Selection Strategy

We now detail our principled API selection approach that consists of four steps.

Step 1. Selecting APIs with the highest correlation with malware (`Set-C`). Our analysis in §4.3 indicates that tracking the top-260 highly correlated APIs (**`Set-C`**) can achieve 93.5% precision and 82.1% recall.

Step 2. Selecting APIs that relate to restrictive permissions (`Set-P`). To protect the privacy/security of user information, an app needs to request permissions before obtaining certain information or fulfilling certain functions [43]. Android permissions are classified into three protection levels [41]: normal, signature, and dangerous. The APIs protected by dangerous-level permissions and those by signature-level permissions are oftentimes relevant to sensitive user data (such as camera, SMS, and location data), which are thus crucial for malware detection. We take advantage of the Axplorer [12] and PScout [13] tools to select the APIs related to restrictive permissions, and we get a total of 112 APIs (referred to as **`Set-P`**). By solely tracking the 112 APIs, we achieve 95.1% precision but rather low (71.3%) recall.

Step 3. Selecting APIs that perform sensitive operations (`Set-S`). The third strategy selects APIs that perform sensitive operations. Different from permissions, there is no "official" definition of sensitive operations. Based on previous work, we find five categories of sensitive operations commonly exploited for conducting attacks: (1) APIs that can lead to privilege escalation, *e.g.,* shell command execution APIs [25], (2) APIs for database operations and file read/write, which are commonly used in privacy leakage attacks [14], (3) APIs operating on key Android components, *e.g.,* those for creating an Android window or overlay, which are used in attacks such as Activity hijacking [10], (4) cryptographic operation APIs, which are commonly used in ransomware attacks [48], and (5) APIs for dynamic code loading, which can load malicious payloads at runtime and perform attacks such as update attack [25]. Based on our domain knowledge, we identify 70 APIs (referred to as **`Set-S`**) that are relevant to the above sensitive operations. By solely tracking these 70 APIs, we achieve 95% precision but poor (70.1%) recall for malware detection.

Step 4. Combining the above. The last step is combining the above strategies, *i.e.,* **`Set-P`**$\bigcup$**`Set-S`**$\bigcup$**`Set-C`**, leading to a total of 426 key APIs. Intuitively, doing so jointly considers both the statistical observations of the data and adversaries' general intentions based on domain knowledge. Note that only 16 overlapped APIs exist among the three sets as shown in Figure 8, indicating that the three sets tend to be orthogonal.

Figure 9 shows that when we only track the 426 key APIs, the per-app time consumption becomes 4.3 minutes on average, which is much shorter than 53.6 minutes (the average time consumption when we track all the 50K APIs), and close to 2.1 minutes (the average time consumption when we do not track any APIs), on our dynamic analysis engine. In §5.1, we will further reduce this detection time by engineering the underlying dynamic analysis engine.

Further, we evaluate the precision and recall of malware detection using the 426 key APIs with the nine mainstream classifiers. As listed in Table 2, random forest still exhibits the highest precision (96.8%) and recall (93.7%), and its training (14.4 seconds) is much faster than that of the more complex classifiers (such as DNN and SVM). In comparison, solely tracking the top-426 highly correlated APIs (extending **`Set-C`**, also using random forest) results in 95.2% precision

and 90.6% recall, as shown in Figure 7. This confirms that the hybrid strategy is better than an individual strategy.

It is worth noting that our selected 426 key APIs might not be the optimal set of APIs with the highest detection accuracy. We do not exhaustively search for such an "optimality" due to the unacceptably large search space. Nevertheless, our proposed API selection strategy is easy to execute, and demonstrates good results in our real-world deployment (§5.2).

4.5 Further Enriching the Feature Space

By examining the dynamic analysis results of ~500K apps, we notice that purely relying on framework APIs to detect malware is inherently limited: an attacker can bypass certain key API invocations by other mechanisms. In practice, two alternative methods can trigger the action of a given framework API without explicitly invoking it: (1) internal/hidden APIs, which can be triggered using special methods such as Java reflection, (2) intents, a key IPC mechanism of Android, which can help an app request another app/service to perform sensitive actions on behalf of it, as well as to monitor/intercept system-level events or broadcasts [16]. We observe that both cases are actively exploited to hide the usage of certain APIs by malicious apps. Hence, these "concealed" API invocations become hidden features that deserve further explorations.

Fortunately, we find that this limitation can be effectively mitigated without incurring any dynamic analysis overhead. Specifically, we add two auxiliary features to help unveil concealed API invocations: the requested permissions and the used intents. Note that the requested permissions are prerequisites of invoking internal/hidden APIs – to our knowledge there is no way for the application to bypass it [16]. The two auxiliary features can be collected by analyzing the app metadata and the parameters of the intent-related framework APIs that are already tracked in **`Set-S`**. Specifically, we add one feature per permission/intent to the existing feature vector.

Figure 10 shows that combining permissions and the 426 key APIs ("A+P") can increase the recall from 93.7% to 96.5%, while combining intents and the 426 key APIs ("A+I") can increase the recall to 94.8%. Interestingly, using permissions and intents alone ("P+I") can also achieve sound performance (97.5% precision and 94.6% recall), implying that these two mechanisms are heavily used by today's Android malware. Finally, in comparison to purely using the 426 key APIs ("A"), jointly leveraging all three feature categories ("A+P+I") achieves the best performance, *i.e.,* increasing the precision from 96.8% to 98.6%, recall from 93.7% to 96.7%, and *F1-score* from 95.2% to 97.6%. Here *F1-score* is the harmonic mean of precision and recall: $2 \times \frac{precision \times recall}{precision + recall}$.

5 System Development

Guided by our above study results in §4, we implement and deploy the APICHECKER system to automatically detect malicious apps submitted to T-Market. In §5.1 we introduce our optimization to the emulation environment for more efficient dynamic analysis. Then we present the deployment of APICHECKER and its "in-the-wild" performance in §5.2. We discuss some other practical aspects of the system in §5.4.

5.1 Emulation Environment Optimization

Having derived the desired features, we shift our focus to enhancing the runtime performance of APICHECKER. To begin with, we notice that the default Google Android device emulator [5] has suboptimal performance due to its heavyweight, full-system emulation built on top of QEMU [11]. This may not be an issue for performing in-lab analysis as we did in §4. However, in a real-world production environment where a large number of apps need to be examined at scale, a long detection delay may negatively affect the experiences and incentives of app developers, as well as increase the infrastructural cost of app market operators.

In Figure 9 we notice that for 30% apps, our original dynamic analysis engine (§4.2) requires more than 5 minutes, a typical turnaround time of Google Bouncer [26], to scan each of them. To boost the scanning performance, in addition to various optimizations introduced in §4 at the detection engine level, we make system-level optimizations to the underlying emulation environment. We architect a lightweight emulation system that efficiently runs the Android OS and apps on powerful commodity x86 servers. First, as for the Android OS we use Android-x86 [21], an open-source x86 port of the original ARM-based Android OS. As a result, the OS-level performance degradation caused by the ISA gap between ARM and x86 is mostly avoided. Meanwhile, to support apps that use Android's *native* libraries, we utilize the state-of-the-art *dynamic binary translation* framework developed by Intel (Houdini [22]) to translate the apps' ARM instructions into x86 instructions, given that most native libraries in Android are based on ARM ISA instead of x86 [47].

Our lightweight Android emulator works with *Monkey* and the Xposed hooking tool, and runs on top of a physical x86 server with a 5×4-core Xeon CPU @ 2.50 GHz and 256-GB DDR memory. To fully utilize the hardware resources, we run multiple emulators in parallel on the server, with each emulator bound to a CPU core. In detail, 16 emulators run on 16 cores concurrently, and the remaining 4 cores are used for task scheduling, status monitoring, and information logging.

Although our lightweight analysis engine substantially outpaces our original engine, its compatibility to Android apps slightly decreases. By customizing the `SystemServer` service in Android-x86, once an app hangs or crashes during the emulation process, an exception message will be automatically reported to the 4 cores (among the 20 cores on the physical server) used for task scheduling, status monitoring, and information logging. From the collected reports, we observe that a very small portion (< 1%) of apps cannot run successfully on the lightweight engine, mainly because of the compatibility issues stemming from Android-x86 and

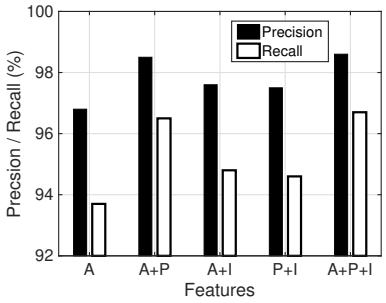

Figure 10. Benefits of auxiliary features (A: key APIs, P: permissions, I: intents).

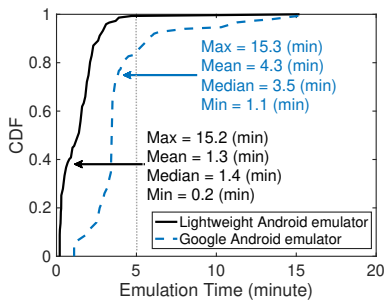

Figure 11. Time consumption of Google emulator and our lightweight emulator.

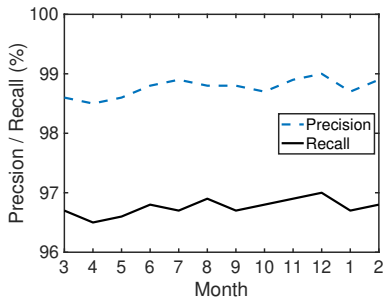

Figure 12. Online performance of APICHECKER over 12 months, from March 2018 to February 2019.

Intel Houdini. For these incompatible apps, we roll back to the default Google Android emulator to successfully analyze them. This apparently takes longer analysis time, but ensures the reliability of APICHECKER—all submitted apps can be tested on our production system.

The above custom infrastructure enables APICHECKER to analyze apps much more efficiently, saving around 70% of the detection time compared to using the default Google Android emulator under the same hardware and detection engine configurations without any detection accuracy loss (we have taken into account the time cost of dealing with the incompatible apps). We evaluate the per-app dynamic analysis time using the default Google Android emulator and our lightweight emulator on the same physical x86 server. Here we only track 426 key APIs. Figure 11 shows that on the same server, our lightweight emulator considerably saves the analysis time per app. The average analysis time per-app is as short as 1.3 minutes, compared to 4.3 minutes of the Google Android emulator.

5.2 System Deployment and Performance

We apply the optimized emulator to APICHECKER and evaluate its performance in T-Market. In particular, we discover a handful of key features play a more important role and discuss them in detail. Moreover, accompanying the emergence of new apps and the upgrade of Android SDK, the selected set of key APIs requires continuous evolution, which prompts us to periodically update our detection model.

Integration to a real app market. APICHECKER was deployed at T-Market and has been running since March 2018. It checks around 10K apps per day using a single commodity server with 16 emulators running concurrently on 16 cores (refer to §4.2 and §5.1 for the detailed hardware/software configurations).

Specifically, it takes an app's APK file as input, and then installs it on an idle Android emulator. Next, APICHECKER executes the app and meanwhile logs a wide range of information as its features (§4). Finally, the random forest classifier determines the malice of the app based on the feature vector. The overall process costs 1.92 minutes on average where the app security analysis consumes 1.4 minutes. In the subsequent 12 months (from March 2018 to February 2019), APICHECKER detected around 2.4K suspicious apps every month. To evaluate the detection accuracy of APICHECKER in the production environment, we inspect submitted apps based on other components in T-Market's app review process (the signature-based preliminary detection, T-Market's manual inspection based on users' and developers' feedback, see §2) as well as our own manual examination. This inspection process can ensure very high precision and recall but involves high manual efforts. The results indicate that over the 12 months, the per-month precision is over 98% (min: 98.5%, max: 99.0%) and the recall is over 96% (min: 96.5%, max: 97.0%), as shown in Figure 12.

As mentioned in §4.1, T-Market has a sophisticated app review process with very high precision and recall. This gives us the ability to deeply understand the <2% false positive and <4% false negative incurred by APICHECKER. First, we find that the 2% false positive (apps) have all used quite a few top-ranking features exemplified in Figure 13. In other words, these apps look similar to malicious apps in terms of API, permission, and/or intent usage. Specifically, among the ~10K apps submitted to T-Market per day, there are on average 800 apps flagged as malicious by APICHECKER. Among them, usually more than 90% are updated apps that can be quickly vetted based on their previous versions. Hence, seeking out the false positives only requires manual inspection on fewer than 80 apps on average, incurring an acceptable overhead to T-Market. Consequently, we choose to actively avoid the false positive apps by manual work every day. In contrast, the 4% false negative reported by users have seldom used the top-ranking features. We manually inspect the detection logs, and find that most (87%) of the false negative apps barely use the 426 key APIs we track. In fact, these apps usually have fairly simple functionalities such as displaying advertisements or accessing certain websites, which are commonly considered

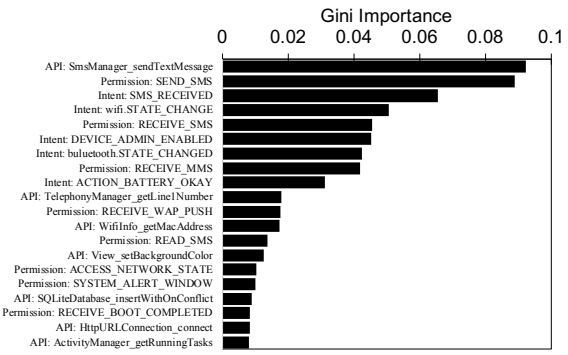

Figure 13. Top-ranking important features for API-based Android malware detection.

to be "mild" security threats to end users. Consequently, we choose to passively mitigate the false negative apps when receiving users' complaints upon specific apps.

Important features. Among all the key features we extracted from the dataset of ~500K apps, we asked which features play the most important roles in malware detection. In Figure 13 we list the Gini [8] indices of the top-20 most important features, including 7 key APIs, 8 requested permissions, and 5 used intents. Here we use the Gini index to quantify the importance of the features due to its suitability for our trained random forest model. In terms of functionality, these 20 key features can be classified into three categories:

- Attempting to acquire privacy-sensitive information of user devices such as SMS message (*e.g.,* the `SmsManager_sendTextMessage` API[3]), phone number (*e.g.,* the `TelephonyManager_getLine1Number` permission), and MAC address (*e.g.,* the `WifiInfo_getMacAddress` API).
- Tracking or intercepting system-level events such as critical activities of devices (*e.g.,* the `RECEIVE_BOOT_COMPLETED` permission), changes of network status (*e.g.,* the `wifi.STATE_CHANGE` intent), and granting privileges (*e.g.,* the `DEVICE_ADMIN_ENABLED` intent).
- Enabling certain types of attacks such as overlay-based attacks [45] (*e.g.,* requiring the `SYSTEM_ALERT_WINDOW` permission to launch "cloak and dagger" attacks [17]).

5.3 System Evolution

During the APICHECKER's operation, we note that it is necessary to periodically update our selected set of key APIs; accordingly, the total number of key APIs is not constant (initially it was 426). This is because new apps are continuously added into T-Market's database, and the entire API base also evolves as Android SDK is updated every several months.

Currently, the period of our updating the key APIs (and retraining the classification model) is empirically configured as one month. The retraining dataset consists of the original dataset acquired from T-Market (§4.1) and the subsequent new apps submitted to T-Market. The malice labels of new apps are flagged by both APICHECKER and manual inspection, bearing no false positives while a small number of false negatives. Moreover, our methodology of selecting the key APIs stays unchanged as described in §4.4.

Figure 14 shows that from March 2018 to February 2019, the number of our selected key APIs only slightly fluctuates between 425 and 432. Hence, the per-app detection time remains stable over the 12 months of deployment. Note that this retraining process is taken into account in Figure 12 where we report the online results. As shown, changes in the key API set only bring mild impacts on the online detection precision, recall, and F1-score, which are 98.5%~98.9%, 96.5%~96.9%, and 97.5%~97.9%, respectively. Overall, the above results indicate that APICHECKER is robust to API evolution.

5.4 Discussion on Other Aspects

During the operation and maintenance of APICHECKER, we have also acquired insights or experiences on other aspects of the system, such as its robustness to possible knowledgeable attackers in the future, its applicability to other app markets who do not wish to implement our customized infrastructure, and its extensibility to other app markets.

How robust is APICHECKER to sophisticated attackers? We write a script to scan the source code of Android SDK (level 27) to know the *actual* coverage of our selected key APIs. We find that although the 426 key APIs take a tiny portion (0.85%) in the 50K framework APIs, there are 4,816 other APIs (9.6%) whose implementation relies on them (*i.e.,* the internal realization of these 4,816 APIs use the 426 key APIs), thus adding up to 5,242 (10.5%) APIs. Consequently, if knowledgeable attackers want to evade our detection by adapting the usage of APIs, they will have to "reimplement" a considerable number of APIs with the same functions to replace the concerned framework APIs and meanwhile carefully avoid the usage of our added auxiliary features (§4.5). This is highly difficult and tedious, if not impossible. A more practical approach for attackers is to use Android NDK. But it also raises the bar of malware development. Also, heavy usage of NDK for common features itself is an indicator of potential malicious behaviors.

Can we further reduce the key API set? API selection is known to be crucial to the tradeoffs among detection accuracy, analysis time, and system resource consumption. Figure 15 shows that among the 426 key APIs, the majority have a relatively small impact on the detection accuracy, but a large impact on the analysis time (see Figure 6). In Figure 15, the

[3]Here we use the short alias `SmsManager_sendTextMessage` to represent the actual API name `android.telephony.SmsManager.sendTextMessage`. We also use short aliases for other mentioned APIs.

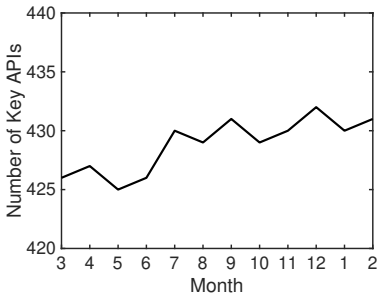

Figure 14. Evolution of the number of our selected key APIs over 12 months, from March 2018 to February 2019.

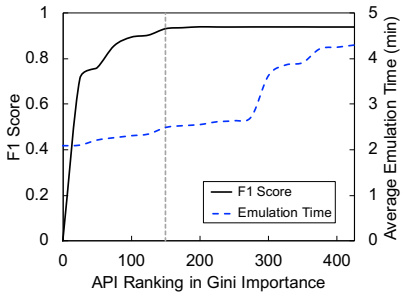

Figure 15. Detection accuracy and analysis time when we only track the top-k ($k \in [1, 426]$) important key APIs.

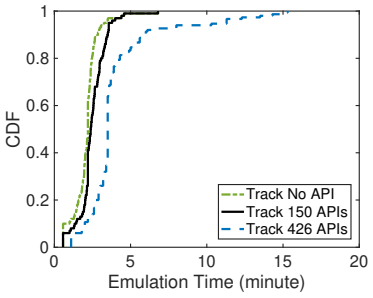

Figure 16. CDF of time consumption when we track no API, 150 APIs and 426 key APIs.

results make us wonder if we can further use small detection accuracy loss to trade for large detection time improvement. We find that APICHECKER using only the top-150 high-ranking key APIs can achieve similar detection performance (98.3% precision and 96.6% recall) compared to that using all 426 key APIs (98.6% precision and 96.7% recall), while the analysis time per app is considerably reduced to 2.5 minutes on average, as shown in Figure 16. This makes it feasible to run Google's Android emulator on even low-end PCs or VM instances for timely malware detection.

Can APICHECKER be used by other app markets? It is not difficult to apply our methodology to other app stores, as it only requires the APK files and some ground-truth data for training purposes. In addition, large app markets can possibly distribute their trained models to smaller markets, who thus do not need to train their own models. Besides, APICHECKER only uses mature program analysis and machine learning techniques, which are easy to implement, deploy, and maintain.

6 Conclusion and Future Work

Machine learning (ML)-based mobile malware detection has been promising in the last decade. Nonetheless, till now we have not seen its realistic solutions for large-scale app markets, which are pivotal to the success of today's mobile ecosystem. In order to figure out and overcome the real-world challenges, we collaborate with a major Android app market to implement, deploy, and maintain an effective and efficient ML-powered malware detection system. It detects Android malware by examining their run-time usage of a small set of strategically selected APIs, enhanced by additional features such as the requested permissions and the used intents. It has been operational at our collaborated market for over one year with several system-level optimizations, such as our customized fast emulation engine, automated model evolution, and affordable false positive/negative mitigation. We hope our measurement findings, mechanism designs, deployment experience, and data release will contribute to the community.

In the future, we plan to explore potential opportunities to further improve our work. First, we note that the UI coverage of *Monkey* could be a bottleneck of our detection due to its critical impact on feature extraction. To make our automated UI exploration mechanism more robust and effective, we wish to incorporate sophisticated software testing techniques such as fuzzing. Moreover, in our current design we leverage a bit vector to encode extracted features, which is lightweight and efficient in practice, but could lose certain feature information (*e.g.,* API invocation frequency) and lead to over-fitting. Therefore, we would like to experiment with other encoding techniques such as histogram that are able to retain more abundant feature information.

Acknowledgments

We sincerely thank our shepherd Prof. Jon Crowcroft and the anonymous reviewers for their valuable feedback. We also appreciate Weizhi Li, Yang Li, Zipeng Wu, and Hai Long for their contributions to the data collection and system deployment of APICHECKER. This work is supported in part by the National Key R&D Program of China under grant 2018YFB1004700, the National Natural Science Foundation of China (NSFC) under grants 61822205, 61902211, 61632020 and 61632013, and the Beijing National Research Center for Information Science and Technology (BNRist).

References

[1] Yousra Aafer, Wenliang Du, and Heng Yin. 2013. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In *Proc. of SecureComm*.

[2] Mohammed Alzaylaee, Suleiman Yerima, and Sakir Sezer. 2017. Improving Dynamic Analysis of Android Apps Using Hybrid Test Input Generation. In *Proc. of IEEE Cyber Security*.

[3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M Memon. 2014. MobiGUITAR: Automated Model-based Testing of Mobile Apps. *IEEE Software* 32 (2014).

[4] Android.com. 2008. UI/Application Exerciser Monkey in Android Studio. https://developer.android.com/studio/test/monkey.html.

[5] Android.com. 2020. Android Emulator. https://developer.android.com/studio/run/emulator.

[6] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *Proc. of NDSS*.

[7] Arzt, Steven and Rasthofer, Siegfried and Fritz, Christian and Bodden, Eric and Bartel, Alexandre and Klein, Jacques and Le Traon, Yves and Octeau, Damien and McDaniel, Patrick. 2014. Flowdroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proc. of ACM PLDI*.

[8] Leo Breiman, Jerome H. Friedman, Richard. Olshen, and Charles J. Stone. 1984. Classification and Regression Trees. *Encyclopedia of Ecology* 40, 3 (1984), 582–588.

[9] Haipeng Cai, Na Meng, Barbara Ryder, and Daphne Yao. 2019. Droidcat: Effective Android Malware Detection and Categorization via App-level Profiling. *IEEE Transactions on Information Forensics and Security* 14, 6 (2019), 1455–1470.

[10] Qi Alfred Chen, Zhiyun Qian, and Z. Morley Mao. 2014. Peeking into Your App without Actually Seeing it: UI State Inference and Novel Android Attacks. In *Proc. of USENIX Security*.

[11] Fan Dang, Zhenhua Li, Yunhao Liu, Ennan Zhai, Qi Alfred Chen, Tianyin Xu, Yan Chen, and Jingyu Yang. 2019. Understanding Fileless Attacks on Linux-based IoT Devices with HoneyCloud. In *Proc. of ACM MobiSys*.

[12] Erik Derr. 2017. axplorer. https://github.com/reddr/axplorer.

[13] dlgroupuoft. 2018. PScout. https://github.com/dlgroupuoft/PScout.

[14] William Enck, Peter Gilbert, Seungyeop Han, et al. 2014. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Transactions on Computer Systems* 32, 2 (2014).

[15] Adrienne Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android Permissions Demystified. In *Proc. of ACM CCS*.

[16] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. 2011. Permission Re-Delegation: Attacks and Defenses. In *Proc. of USENIX Security*.

[17] Yanick Fratantonio, Chenxiong Qian, Simon Chung, and Wenke Lee. 2017. Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop. In *Proc. of IEEE S&P*.

[18] Bin Fu, Jialiu Lin, Li Lei, Christos Faloutsos, Jason Hong, and Norman Sadeh. 2013. Why People Hate Your App: Making Sense of User Feedback in a Mobile App Store. In *Proc. of ACM KDD*.

[19] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. 2013. Reran: Timing and Touch-sensitive Record and Replay for Android. In *Proc. of IEEE ICSE*.

[20] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. 2012. Riskranker: Scalable and Accurate Zero-day Android Malware Detection. In *Proc. of ACM MobiSys*.

[21] Google Group. 2009. Android-x86 – Porting Android to x86. http://www.android-x86.org/.

[22] Chih-Wei Huang. 2016. Intel Houdini. https://osdn.net/projects/android-x86/scm/git/vendor-intel-houdini/.

[23] INRIA. 2010. Scikit-learn. http://scikit-learn.org/stable/index.html.

[24] Jason Brownlee. 2016. Data Leakage in Machine Learning. https://machinelearningmastery.com/data-leakage-machine-learning/.

[25] Xuxian Jiang and Yajin Zhou. 2012. Dissecting Android Malware: Characterization and Evolution. In *Proc. of IEEE S&P*.

[26] O. Jon and M. Charlie. 2012. Dissecting the Android Bouncer. https://jon.oberheide.org/files/summercon12-bouncer.pdf.

[27] Jemin Lee and Hyungshin Kim. 2016. QDroid: Mobile Application Quality Analyzer for App Market Curators. *Mobile Information Systems* 2016, 1 (2016), 1–11.

[28] Hong Bo Li, Wei Wang, Hong Wei Ding, and Jin Dong. 2010. Trees Weighting Random Forest Method for Classifying High-dimensional Noisy Data. In *Proc. of IEEE ICEBE*.

[29] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon J. Ross, and Gianluca Stringhini. 2017. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. In *Proc. of NDSS*.

[30] John H. McDonald. 2019. Spearman Rank Correlation. http://www.biostathandbook.com/spearman.html.

[31] Naser Peiravian and Xingquan Zhu. 2013. Machine Learning for Android Malware Detection Using Permission and API Calls. In *Proc. of IEEE ICTAI*.

[32] Rovo89. 2016. XposedBridge. https://github.com/rovo89/XposedBridge/wiki/Development-tutorial.

[33] Arshad Saba, Ali Shah Munam, Khan Abid, and Ahmed Mansoor. 2016. Android Malware Detection & Protection: A Survey. *International Journal of Advanced Computer Science and Applications* 7, 2 (2016).

[34] Schulz, Michèle N and Landström, Jens and Hubbard, Roderick E. 2013. MTSA–A Matlab Program to Fit Thermal Shift Data. *Analytical biochemistry* 433, 1 (2013), 43–47.

[35] Akanksha Sharma and Subrat Kumar Dash. 2014. Mining API Calls and Permissions for Android Malware Detection. In *Proc. of CANS*.

[36] Simone Silvestri, Rahul Urgaonkar, Murtaza Zafer, and Bong Jun Ko. 2018. A Framework for the Inference of Sensing Measurements based on Correlation. *ACM Transactions on Sensor Networks* 15, 1 (2018).

[37] Mingshen Sun, Min Zheng, John CS Lui, and Xuxian Jiang. 2014. Design and Implementation of an Android Host-based Intrusion Prevention System. In *Proc. of IEEE ACSAC*.

[38] Kimberly Tam, Salahuddin J. Khan, Aristide Fattori, and Lorenzo Cavallaro. 2015. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Proc. of NDSS*.

[39] Guanhong Tao, Zibin Zheng, Ziying Guo, and Michael R. Lyu. 2018. MalPat: Mining Patterns of Malicious and Benign Android Apps via Permission-Related APIs. *IEEE Transactions on Reliability* 67 (2018).

[40] Dali Wang, Zhifen Lin, Ting Wang, Xiruo Ding, and Ying Liu. 2017. An Analogous Wood Barrel Theory to Explain the Occurrence of Hormesis: A Case Study of Sulfonamides and Erythromycin on Escherichia Coli Growth. *PloS One* 12, 7 (2017).

[41] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. 2012. Permission Evolution in the Android Ecosystem. In *Proc. of IEEE ACSAC*.

[42] M. Wong and D. Lie. 2016. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware. In *Proc. of NDSS*.

[43] D. Wu, C. Mao, T. Wei, H. Lee, and K. Wu. 2012. Droidmat: Android Malware Detection through Manifest and API Calls Tracing. In *Proc. of IEEE Asia JCIS*.

[44] WenChieh Wu and ShihHao Hung. 2014. DroidDolphin: A Dynamic Android Malware Detection Framework Using Big Data and Machine Learning. In *Proc. of ACM RACS*.

[45] Yuxuan Yan, Zhenhua Li, Qi Alfred Chen, Christo Wilson, Tianyin Xu, Ennan Zhai, Yong Li, and Yunhao Liu. 2019. Understanding and Detecting Overlay-based Android Malware at Market Scales. In *Proc. of ACM MobiSys*.

[46] Ming Yang, Shan Wang, Zhen Ling, Yaowen Liu, and Zhenyu Ni. 2017. Detection of Malicious Behavior in Android Apps through API Calls and Permission Uses Analysis. *Concurrency and Computation: Practice and Experience* 29, 19 (2017).

[47] Qifan Yang, Zhenhua Li, Yunhao Liu, Hai Long, Yuanchao Huang, Jiaming He, Tianyin Xu, and Ennan Zhai. 2019. Mobile Gaming on Personal Computers with Direct Android Emulation. In *Proc. of ACM MobiCom*.

[48] Tianda Yang, Yu Yang, Kai Qian, Dan Chia-Tien Lo, Ying Qian, and Lixin Tao. 2015. Automated Detection and Analysis for Android Ransomware. In *Proc. of IEEE HPCC*.

[49] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. 2014. Droid-sec: Deep Learning in Android Malware Detection. *ACM SIGCOMM Computer Communication Review* 44, 4 (2014), 371–372.

[50] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. 2012. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proc. of NDSS*.