Systematically Landing Machine Learning onto Market-Scale Mobile Malware Detection

Liangyi Gong, Hao Lin, Zhenhua Li, *Member, IEEE*, Feng Qian, *Member, IEEE*, Yang Li, Xiaobo Ma, *Member, IEEE*, and Yunhao Liu, *Fellow, IEEE*

Abstract—Despite being crucial to today's mobile ecosystem, app markets have meanwhile become a natural, convenient malware delivery channel as they actually "lend credibility" to malicious apps. In the past few years, machine learning (ML) techniques have been widely explored for automated, robust malware detection, but till now we have not seen an ML-based malware detection solution applied at market scales. To systematically understand the real-world challenges, we conduct a collaborative study with T-Market, a popular Android app market that offers us large-scale ground-truth data. Our study illustrates that the key to successfully developing such systems is multifold, including *feature selection and encoding*, *feature engineering and exposure*, *app analysis speed and efficacy*, *developer and user engagement*, as well as *ML model evolution*. Failure in any of the above aspects could lead to the "wooden barrel effect" of the whole system. This article presents our judicious design choices and first-hand deployment experiences in building a practical ML-powered malware detection system. It has been operational at T-Market, using a single commodity server to check ~12K apps every day, and has achieved an overall precision of 98.9 percent and recall of 98.1 percent with an average per-app scan time of 0.9 minutes.

Index Terms—Machine learning, mobile malware detection, app market, dynamic analysis, Android emulation

1 Introduction

App markets, such as Google Play, Apple App Store, and Amazon AppStore, play a critical role in today's mobile ecosystem, through which most mobile apps are published, updated, and distributed to users. On the other hand, the markets have also become a convenient channel for adversaries to spread malware. Even worse, adversaries prefer to use this channel because when an app is published in a recognized app market, it in fact "lends credibility" to the app. Moreover, mobile devices are often pre-configured to allow app installations from only app markets by default [1]. Thus, automated market-scale malware detection is necessary by all popular app markets today.

In the past few years, machine learning (ML) techniques have been extensively explored for malware detection, as they do not depend on specific rules and thus are considered more automated and robust. A variety of ML-based techniques have been proposed, from simple app fingerprint checking [1], static code inspection [2], to sophisticated dynamic behavior analysis [3]. Till now, however, we have not seen any report on the effectiveness and efficiency of such solutions being applied at a market scale.

This paper presents our systematical efforts towards building and deploying an ML-powered mobile malware detection solution. Collaborating with a popular Android app market, i.e., Tencent App Market [4] or T-Market for short, we get full access to the large-scale ground-truth data of apps (both released and rejected) and their malice labels (Benign or Malicious). By comprehensively analyzing the data and existing ML-based malware detection solutions, we unravel that the key challenges lie in multiple aspects: *feature selection and encoding*, *feature engineering and exposure* (here *exposure* means to let an app manifest its features adequately), *app analysis speed and efficacy*, *developer and user engagement*, as well as *ML model evolution* over time.

More importantly, we notice that failure in any of the above aspects could lead to the "wooden barrel effect" [5] of the entire solution. For instance, feature selection affects not only the detection accuracy but also the app analysis time (both have strict requirements on a market-scale solution). Besides, feature engineering decides the detection robustness and the difficulty of model evolution. Furthermore, both detection accuracy and analysis speed impact developers' engagement in app submissions, which is vital for the prosperity of a popular app market.

To build a desired market-scale, ML-powered malware detection system, first of all, we choose to concentrate on *a lightweight and scalable design of feature extraction*: *API-centric dynamic analysis*, which tracks Android API invocations at an app's runtime to achieve high analysis speed. Since Android SDK APIs provide almost all functions for typical apps, they are still the *de facto* feature choice in almost all

- Liangyi Gong, Hao Lin, Zhenhua Li, and Yang Li are with the School of Software and BNRist, Tsinghua University, Beijing 100084, China. E-mail: {gongliangyi, linhaomails, lizhenhua1983, liyang14thu}@gmail.com.
- Feng Qian is with the Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455 USA. E-mail: fengqian@umn.edu.
- Xiaobo Ma is with MOE Key Lab for Intelligent Networks and Network Security, School of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an 710049, China. E-mail: xma.cs@xjtu.edu.cn.
- Yunhao Liu is with the Global Innovation Exchange, Tsinghua University, Beijing 100084, China. E-mail: yunhaoliu@gmail.com.

Manuscript received 1 July 2020; revised 18 Nov. 2020; accepted 30 Nov. 2020. Date of publication 21 Dec. 2020; date of current version 11 Feb. 2021.
(Corresponding author: Zhenhua Li.)
Recommended for acceptance by P. Balaji, J. Zhai, and M. Si.
Digital Object Identifier no. 10.1109/TPDS.2020.3046092

1045-9219 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Tsinghua University. Downloaded on June 12,2022 at 13:08:09 UTC from IEEE Xplore. Restrictions apply.

prior studies [6]. Indeed, our study indicates in many cases, using more complex features can hardly bring a noticeably higher detection accuracy [3], [7], [8].

Next, we take *a principled, data-driven approach for feature selection*, noticing that Android SDK offers 50,000+ APIs. The rationale is threefold. First, the number of selected APIs has non-negligible influence on the analysis time (up to $50\times$ difference). Second, compared to tracking all 50K APIs, strategically tracking fewer APIs leads to better detection accuracy, probably because of the reduced likelihood of over-fitting. Third, APIs identified from distinct sources complement each other; carefully combining them greatly improves the detection accuracy. With the above considerations, we pick out a total number of 426 key APIs as features, and adopt the lightweight *Random Forest* ML algorithm (among a wide range of off-the-shelf ML algorithms), achieving 96.8 percent detection precision and 93.7 percent detection recall on the large-scale dataset from T-Market.

Moreover, we devise *a feature-embedded encoding scheme to efficiently retain the fine-grained information of key API invocations.* In the preliminary work [9], we use the traditional *One-Hot* encoding scheme [10] to represent the invocation information of key APIs, which records whether or not an API is invoked while leaving out the invocation frequency and hierarchy of key APIs. The latter fine-grained information, nevertheless, is also useful in deciding the malice of quite a few apps. Hence, we design a novel encoding scheme to efficiently represent such information by converting each API invocation into a word embedding vector [11], saving $\sim 20\times$ memory usage compared with using one-hot encoding to represent such information. After the above enhancement is applied, the detection precision and recall are improved to 97.1 and 96.9 percent, respectively.

To further boost the detection accuracy, we take *an adversary's perspective to figure out hidden features*. Purely relying on Android APIs for malware detection bears a limitation: an adversary can bypass API invocations and use other mechanisms like Java reflection and intents (Android's IPC mechanism) to fulfill the functionality of certain APIs. In our dataset, we observe that both mechanisms have been leveraged by malicious apps to hide their certain features. To this end, we also capture the permissions requested and the intents used during dynamic analysis; such "indirect" features are then combined with API invocation features to get a more complete picture of an app's runtime behavior.

In addition, to better manifest all the aforementioned features, we *actively exploit apps' information to overcome the major shortcomings of the Monkey UI exerciser* (which we use in the preliminary work [9]): *redundant actions and action loops* [12], which could degrade the testing coverage and thus impair the feature exposure. To reduce redundant actions, we fine-tune the portions of different types of *Monkey* events according to the specific category (e.g., shopping or news feed) of an app. To mitigate action loops, we strategically exploit an app's UI layout structure and component information as heuristics for more comprehensive activity exploration. With the above optimizations, the detection time is shortened by 15 percent on average, and the detection precision and recall are improved to 98.9 and 98.1 percent, respectively.

Having acquired the desired features, we shift our focus to accelerating the app analysis. We *architect the app emulation system to let it run efficiently on x86 servers* (Section 5.1). Specifically, we run the native Android-x86 OS [13] and translate apps' native code from ARM to x86 with Houdini, the state-of-the-art dynamic binary translation (DBT) tool developed by Intel [14]. Also, to optimize graphic rendering for apps, we adopt GPU-assisted acceleration to intercept the "micro" graphic instructions (that are decomposed and reconstructed from OpenGL instructions by the graphic driver) and execute them on x86 servers' dedicated GPUs. The above efforts reduce the app execution time by $\sim 77\%$.

On the other hand, we notice that some apps could recognize the emulation environment and silence their malicious activities. To address this, in the preliminary work [9] our solutions mostly involve parameter configuration and library obfuscation before app emulation. However, these are insufficient when encountered with apps' careful examination of system properties and user behaviors. To account for this, we further make *twofold in-situ efforts during app emulation*: 1) *hooking apps' certain operations* that can recognize emulation and applying defensive intervention, and 2) *adaptively tuning the action frequency of* Monkey once the emulated app enters a "silent" stage. With such in-situ improvements, only 0.4 percent apps exhibit fewer features than on real devices; the percentage is remarkably smaller than that (1.4 percent) without in-situ improvements.

False positives/negatives can hardly be avoided in ML-based systems. In our solution, false positive apps (that are complained by developers) and false negative apps (that are reported by end users) are both manually analyzed but in distinct ways. We choose to actively avoid the former (on a daily basis), since it essentially increases the burden of manual intervention to deal with developers' complaints. In detail, $\sim 90\%$ of the flagged malicious apps are updated apps, which can be quickly checked based on their previous versions, and thus the totally required manual inspection is affordable in practice. On the contrary, for the latter we only conduct manual analysis upon user reports, because we observe that *the existence of a small portion of false negative apps in fact brings little impact on the regular operation of T-Market*. Manual inspection illustrates that 87 percent of the sampled false negative apps seldom use the key APIs we select to track; hence, they have fairly simple functionalities and will not pose a great security threat to end users.

All the above efforts have been implemented in the real-world system APICHECKER, whose workflow is shown in Fig. 1. Upon an app submission, it first leverages DBT to enable the app's efficiently running on x86 environments (①). Then, the enhanced *Monkey* exerciser automates the app execution to trigger various app behaviors (②). To prevent adversaries' detection evasion, we also integrate multiple defensive interventions (③). Meanwhile, the internal Xposed [23] intercepts and logs API invocations (④⑤). Based on the app's metadata (⑥), our analysis engine extracts carefully-selected features and encodes them in a feature-embedded manner (⑦⑧). Using a random forest classifier, APICHECKER is then able to determine the app's malice. Finally, we take different measures to cope with possible false positives and negatives (⑨). In real deployment, we notice that several steps above are in fact independent and thus can be carried out in parallel. Thus, we convert the monolithic back-to-back

Authorized licensed use limited to: Tsinghua University. Downloaded on June 12,2022 at 13:08:09 UTC from IEEE Xplore. Restrictions apply.

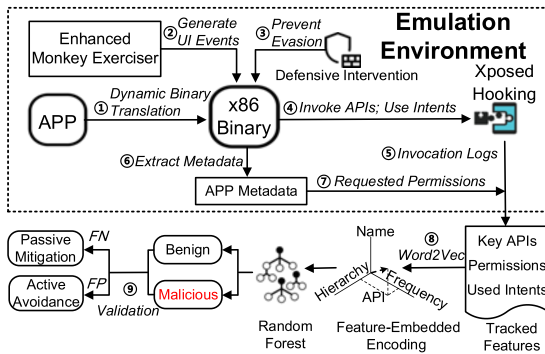


Fig. 1. Architectural overview of APIChecker.

execution into a loosely-coupled pipeline to further boost performance.

APIChecker was first launched at T-Market in March 2018 [9] and has been continually optimized. It can currently examine ~12K newly submitted apps per day on a single commodity server (with 20 cores), and take an average of 0.9 minutes to scan a submitted app. This is basically acceptable to the developers, given that the typical per-app scan time is nearly 5 minutes in Google Play [24]. Recently, the detection precision and recall have been around 99 and 98 percent, respectively, because *the ML model is automatically updated every month with new apps and novel Android SDK APIs* (Section 5.2). As the model evolves, the amount of selected key APIs just slightly fluctuates between 425 and 432.

*Data and Tool Availability.*We have released at https://apichecker.github.io/ the list of our selected key Android SDK APIs, part of our analysis logs, and our implemented efficient emulator.

2 Motivation

Nowadays in app markets, mobile malware leverages various techniques (e.g., repackaging and update attack) to conceal its malice [25], entices users to download and install from the app markets, and then carries out malicious operations on user devices that are hard to detect. To combat these threats, Google launched a proprietary malware detection system called "Bouncer" [24] in 2012 on Google Play, reducing the number of malicious apps on the platform by 40 percent. However, a lack of such detection on numerous third-party app markets has allowed malware continue to spread. To have an in-depth and practical understanding of the issue, in this paper we collaborate with a third-party app market called T-Market that has released over 6M apps since 2012, with more than 30M APKs being downloaded by 20M users every day.

To detect malware, T-Market reviews new and updated apps submitted from developers on a daily basis. Specifically, T-Market introduced a sophisticated app review process that is mainly a conglomeration of three techniques: 1) fingerprint-based antivirus checking, 2) empirical API inspection, and 3) user-report-driven manual examination. First, antivirus checking vets apps against virus fingerprint databases [1] provided by antivirus services such as Symantec, Kaspersky, Norton, and McAfee, as well as those collected by T-Market itself. Second, API inspection identifies malware by monitoring the invocations of a set of specified APIs through static code analysis. The API set is selected by security experts, based on their experiences that certain invocation patterns of APIs hint potential security threats [7]. Finally, after the above steps, false results can still exist. T-Market currently depends on developers and users to report false positives and false negatives, respectively. For reported cases, manual examination is then performed to decide the results.

In this review process, the fingerprint-based antivirus checking is only able to detect known malware samples, while leaving the most critical defense task, detecting zero-day malware, to the subsequent API and manual inspection steps [18]. As manual inspection is slow (up to several days of analysis time for an app), T-Market is especially interested in enhancing the API inspection step and achieving a similar performance as the "Bouncer" system. In particular, as today's popular ML techniques are often expected to be more automated and robust without dependence on specific rules (from security experts), T-Market hopes to know whether they are capable of achieving the goal.

3 Related Work

Static Analysis. Static analysis extracts API usage information from an app's APK file that contains the compiled code and configuration/resource files. After that, ML or heuristic rules can be applied to decide apps' malice. For instance, Sharma *et al.* [15] apply Naive Bayesian and kNN classifiers to 35 malice-correlated APIs extracted from 1,600 apps, to achieve 91.2 percent precision and 97.5 percent recall for malware detection.

DroidAPIMiner [7] extracts critical APIs based on their usage frequencies, and compares the performance of four ML classifiers. As a result, kNN achieves the best performance of 99 percent accuracy and 2.2 percent false positive rate, requiring 25 seconds on average to check an APK file. Stowaway [16] is an automated static analysis tool that builds a *permission map* to detect apps' over-privileged behaviors, using 1,259 APIs with *restrictive permission* extracted from 964 apps. DroidMat [17] adopts a similar strategy to determine the malice of 1,738 apps, which employs the k-means clustering to enhance the kNN classification model.

RiskRanker [18] leverages a two-order risk analysis to examine 118K apps (taking around 41 seconds for each app). Eventually 3,281 risky apps are revealed by identifying certain patterns of seemingly innocent API usages, which may in fact be indicators of malware. Further, Droid-Sec [20] proposes a mashup strategy that utilizes 64 sensitive APIs extracted through static analysis, and 18 app behaviors based on dynamic analysis of 250 Android apps. It manages to achieve the highest accuracy of 96.0 percent with the deep belief network. Adopting a *hybrid* strategy, DREBIN [22] gathers multi-dimensional features from 129K apps, including permission-restricted APIs, suspicious APIs related to sensitive operations, requested permissions, network addresses, and so forth. On a real device, it takes 10 seconds on average to extract features, and identifies certain patterns of embedded features with the support vector machine (SVM) classifier.

Dynamic Analysis. Due to its immunity to code obfuscation and dynamic code loading, dynamic analysis can reveal a

Authorized licensed use limited to: Tsinghua University. Downloaded on June 12,2022 at 13:08:09 UTC from IEEE Xplore. Restrictions apply.

deeper and more complete landscape of apps' behaviors. For instance, Yang *et al.* [8] develop a dynamic app behavior analysis platform to examine the run-time usage of 19 APIs that are related to three specific types of restrictive permissions, in terms of obtaining device/system information, network access, and charging from the user's account. Using an SVM model, they vet each app for around 18 minutes, and achieve 92.8 percent precision and 84.9 percent recall.

DroidDolphin [21] builds a dynamic analysis framework that leverages big data analysis and the SVM algorithm to check the usage of 25 APIs and 13 types of sensitive operations in around 17 minutes, achieving 90 percent precision and 82 percent recall. IntelliDroid [3] extracts specified API call paths and risky event-chains from 2,326 apps in terms of 228 target APIs that may facilitate sensitive operations uncovered by TaintDroid [26]. Based on the extracted features, it detects malware during an app's runtime in an average of 138.4 seconds. Moreover, DroidCat [19] applies a random forest classification model to 122 extracted behavioral features, achieving 97.5 percent precision and 97.3 percent recall, as well as an average per-app time cost of 354 seconds for feature computation and classification. Unfortunately, it is subject to dynamic code loading, thus substantially degrading its generalizability.

Comparison With Our Work. Despite employing a similar *API-centric analysis* scheme, our work differs from current researches (as listed in Table 1) in multiple aspects: First, our measurement scale (in terms of the number of studied apps) is much larger. Second, our API selection is innovative, and we identify hidden features and optimize UI exerciser to further boost the detection accuracy. Third, we enhance the dynamic execution (emulation) infrastructure to considerably reduce the detection time while guarantee the analysis efficacy and reliability. Fourth, we commercially deploy our system and update the ML model in the process. In total, our work develops the first practical and integrated ML-based malware detection solution at market scales with commercial deployment results reported.

4 Collaborative Study

In this section, we take a principled, data-driven approach to study challenges and strategies of building an ML-based malware detection system at real markets.

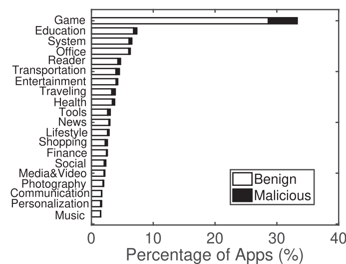


Fig. 2. Categories of apps in our dataset.

4.1 Large-Scale Dataset

Our app dataset includes ~500K apps newly submitted to T-Market within 10 months (from March to December 2017), with $\sim 85\%$ of them are updated apps initially submitted as early as in 2014. In practice, we treat APKs with the same package name but different MD5 hash codes as different apps. Also, in the dataset T-Market provides a malice label (Malicious or Benign) for each app using a sophisticated and effective app review process introduced in Section 2. The labels are acquired with at least four state-of-the-art signature-based antivirus checking [9], static API inspection, and manual examination triggered by developers and users' complaints. Hence, despite a trivial portion of falsely-labeled apps, the dataset is mostly able to provide credible, unbiased ground truth. In general, the dataset contains 463,273 benign apps and 38,698 malicious apps. Fig. 2 demonstrates the categories of apps (as classified by T-Market) in our dataset, and the malice breakdown of each category. Gaming apps account for the largest portion in our dataset ($\sim 35\%$) while each category contains $< 15\%$ of malicious app samples. Currently, the malicious ones are quarantined in T-Market's database without being released.

Ethical Concern. When conducting this study, we strictly follow the agreement established between T-Market and the developers who submit their apps. When individual apps are referred to as examples, they are anonymized for developers' privacy protection.

4.2 Dynamic Analysis Engine

In order to monitor and record the run-time invocations of framework APIs, we build a dynamic analysis engine atop of Google Android emulator [27], and the Xposed hooking framework [23] that can intercept a target API prior to its

TABLE 1
Representative Android Malware Detection Schemes That Study a Selected Set of Potentially Useful Framework APIs

API Selection Strategy	Related Work	Analysis Method	Analysis Time per App	# APIs Used	# Apps Studied	Precision, Recall
Statistical Correlations	Sharma *et al.* [15]	static	- -	35	1,600	91.2%, 97.5%
	DroidAPIMiner [7]	static	25 sec	169	~20K	- -
Restrictive Permissions	Stowaway [16]	static	- -	1,259	964	- -
	DroidMat [17]	static	- -	- -	1,738	96.7%, 87.4%
	Yang *et al.* [8]	dynamic	1080 sec	19	~27K	92.8%, 84.9%
Sensitive Operations	RiskRanker [18]	static	41 sec	- -	~118K	- -
	DroidCat [19]	semi-dynamic	354 sec	27	~34K	97.5%, 97.3%
	IntelliDroid [3]	static + dynamic	138.4 sec	228	2,326	- -
	Droid–Sec [20]	static + dynamic	- -	64	250	- -
	DroidDolphin [21]	dynamic	1020 sec	25	64K	90%, 82%
Hybrid	DREBIN [22]	static	10 sec	- -	~128K	- -
	APIChecker	**dynamic**	**66 sec**	**426**	**~500K**	**98.9%, 98.1%**

"- -" means unknown.

Authorized licensed use limited to: Tsinghua University. Downloaded on June 12,2022 at 13:08:09 UTC from IEEE Xplore. Restrictions apply.

TABLE 2
Performance and Overhead of Different ML Classification Models When 50K Versus 426 Key APIs are Tracked

Models	Precision (50K / 426)	Recall (50K / 426)	Training Time (50K / 426)
NB [15]	60.4% / 64.1%	59.6% / 63.6%	3.6 min / 1.7 sec
LR [20]	81.2% / 89.9%	70.3% / 72.4%	10.4 min / 4.5 sec
SVM [8]	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 [15]	86.5% / 95.3%	83.7% / 93.3%	~1.8K min / 821 sec
CART [7]	87.6% / 94.3%	84.3% / 93.7%	11.6 min / 5.8 sec
ANN [20]	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
RF [19]	91.6% / **96.8**%	90.2% / **93.7**%	29.1 min / 14.4 sec

invocation. Meanwhile, we automate each app with the *Monkey* UI exerciser [28] which can generate app- or system-level UI event streams. For model comparison and selection, we feed our collected logs into nine mainstream machine learning algorithms as listed in Table 2 to observe the yielded performance, including Naive Bayes (NB), CART decision tree, logistic regression (LR), *k*-nearest neighbor (kNN), support vector machine (SVM), gradient boosting decision tree (GBDT), artificial neural network (ANN), deep neural network (DNN), and random forest (RF). These models have been extensively explored in existing studies or systems [7], [8], [15], [17], [19], [20], [21], [22], and demonstrate the best performance. In building the above infrastructure, our key design decision is using *provably mature* program analysis (i.e., a well-received app analysis tool chain, in terms of emulation [27], API hooking [29], and UI testing) and ML techniques as our building blocks, as our engine will be part of a production system that is supposed to accommodate all the app market's hosted apps.

App Emulation Environment. We deploy Google Android emulators on a sever cluster containing 16 commodity x86 machines (HP ProLiant DL-380 running Ubuntu 16.04 LTS 64-bit). They are all equipped with a 5×4-core Xeon CPU@2.50 GHz and 256-GB DDR memory. On each machine, we pin 16 emulators on 16 cores to allow concurrency, while the remaining 4 cores are assigned with task scheduling, status monitoring, and data logging. We then parallelly run the ~500K apps on 16 emulators using the Android Debug Bridge (*adb*) tool. For each app, *adb* commands are sequentially executed for automatic app installation, UI event generation and execution with *Monkey*, log recording, app uninstall, and residual data purging. Meanwhile, when executing *Monkey*'s generated UI events for an app, we use Xposed not only to intercept target APIs' invocation data (including API names and parameters), but also to implement possible callback interfaces to perform instrumental operations (e.g., hooking a target `Activity`, and return customized values to bypass user login or simulate behaviors of a real device).

Furthermore, we observe that malware sometimes attempts to detect the existence of emulators to evade our detection by suppressing its malicious activities. Common practices of emulator identification mainly include examinations of system states/configurations, user behaviors, sensor data and installed packages. To prevent such detection from discovering our emulation environment, in the preliminary work [9] we make fourfold improvements as follows: First, we modify default configurations of emulators and default parameters of the `Build` class [30], including those related to device identities (`IMEI` and `IMSI`), network status (e.g., the TCP information maintained in `/proc/net/tcp`), and `PRODUCT/MODEL` types to conceal ourselves. Second, we adjust the execution parameter `throttle` of *Monkey*, which regulates the input interval to make the UI events appear more realistic. Meanwhile, we specify two flags (`ignore-crashes` and `ignore-timeouts`) to enhance *Monkey*'s resistance to response exceptions in apps [28]. Third, we replay traces of sensor data (e.g., those of accelerometer, gyroscope) collected in multiple real devices on our emulators to augment their authenticity [31]. Finally, we obfuscate libraries related to Xposed and modify the return values of certain interfaces of the `PackageManager` class, so as to hide the existence of Xposed [32].

To measure the effectiveness of our enhanced emulation environment, we carry out a controlled experiment by running the same sample set of apps on real Android devices (Google Nexus 6), the default emulator, and our enhanced emulator. The sample set is an unbiased subset (1 percent) randomly extracted from our dataset (the ~500K apps). As a result, on the default emulator only 86.6 percent of apps invoke as many APIs as (they invoke) on the real Android devices, while on our enhanced emulator 98.6 percent of apps invoke as many APIs as on the real Android devices. This result showcases the effectiveness of our optimizations to the emulator.

UI Exploration Methodology. For automatic UI exploration, our goal is to achieve a high UI coverage that can trigger as many user activities as possible. Originally, we used `Activity` coverage as our UI coverage metric [33], as possible `Activity` objects of an app have been specified in the configuration file (`AndroidManifest.xml`). However, this metric is exceedingly pessimistic given that `Activities` not referenced by the app code are taken into account in calculation. To understand the common ratio of specified `Activities` actually being referenced by an Android app, we write a script to scan and analyze the configuration file and the static app code of non-obfuscated APKs in our dataset. The scanning results showcase that only 88 percent of specified `Activities` are referenced in the code on average. To more accurately measure and quantify UI coverage, we define a new metric based on the above findings—Referred Activity Coverage (RAC), which is the ratio between the number of detected (triggered) `Activities` during an app's emulation and the number of its referenced `Activities`.

We next explore the relation between executed *Monkey* events and RAC. The triggered `Activities` during an app's emulation are detected and logged by Xposed [23]. Quantitatively, we notice that executing ~100K *Monkey* events can obtain 86 percent RAC on average, while executing more events hardly increases the recorded RAC. However, 100K *Monkey* events consume 2,142 seconds (35.7 minutes) for execution on average, which is unacceptable to both app market operators and developers in practice (given that Google Bouncer only requires nearly 5 minutes to analyze an app submission [24]). To account for this problem, carefully balancing

Authorized licensed use limited to: Tsinghua University. Downloaded on June 12,2022 at 13:08:09 UTC from IEEE Xplore. Restrictions apply.

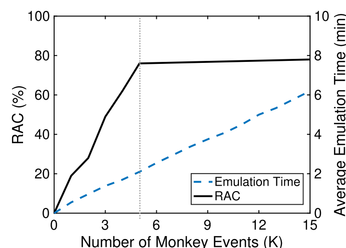


Fig. 3. Relationship among # *Monkey* events, RAC, emulating time.

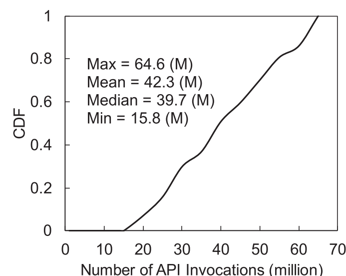


Fig. 4. CDF of the number of API invocations when emulating one app.

the effectiveness (in terms of RAC) and the efficiency (in terms of emulation time) is essential.

Fig. 3 demonstrates the average RAC with the increasing number of *Monkey* events. In detail, we notice that as the emulation time increases (proportional to the increase of *Monkey* events), the average RAC quickly rises to 76.5 percent within 126-second emulation time. Afterwards, the benefits of executing more *Monkey* events become marginal – even spending ~250 seconds for 10K events merely brings 1.5 percent increase in the RAC on average. As a result, we choose to execute 5K *Monkey* events for an app's emulation (costing 126 seconds), so as to achieve a decent RAC (76.5 percent). In other words, we sacrifice a small portion (9.5 percent) of RAC to largely reduce (94 percent) the emulation time compared to executing 100K *Monkey* events.

Additionally, Fig. 4 shows the number of detected API invocations during the emulation of one app. Quantitatively, tens of millions of API invocations are triggered during the emulation (with 5K *Monkey* events being executed). That is, one *Monkey* event would trigger an average of 8,460 APIs, indicating that API usage during an Android app's running is intensive. Furthermore, we notice that the number of tracked APIs could also greatly impact the emulation time. As shown in Fig. 5, when emulating an app without tracking any APIs, the time consumption is merely 2.1 minutes on average. But when all the ~50K APIs are tracked during emulation, the time consumption drastically increases to an average of 53.6 minutes due to the considerable overhead of intercepting a large number of APIs. Evidently, tracking all APIs would be time-wise infeasible. To practically track API usages, we judiciously investigate key API selection strategy in Section 4.4.

Machine Learning Algorithms. Similar as most existing work reviewed in Section 3, we adopt machine learning techniques to classify an app as malicious or benign. Recall that during the emulation of each app from our dataset, the invocation data of the tracked APIs (API names and parameters) are recorded. To feed the data into ML models for training, in the preliminary work [9] we employ *One-Hot*

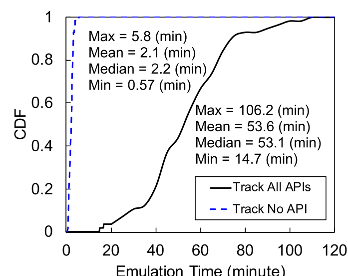


Fig. 5. Time consumption for tracking all APIs and no API.

encoding to transform the log into a feature vector of n dimensions, where n is the total number of tracked APIs and each dimension represents the invocation status of a tracked API (1 for invoked and 0 for not). All the feature vectors of our studied apps are split into the training or test sets (which are of course disjoint).

For the performance evaluation of ML models, three key metrics are commonly considered: precision, recall, and training time. The precision and recall are calculated as: $precision = \frac{TP}{TP+FP}$ and $recall = \frac{TP}{TP+FN}$, where TP (True Positive) denotes the number of apps accurately classified as malware, while FP (False Positive) and FN (False Negative) represent the number of apps falsely vetted as malicious and benign, respectively. In total, we use scikit-learn [34] to implement nine mainstream machine learning models (*cf.* Section 4.2). In our experiments, we tune the hyper-parameters of each model based on our domain knowledge.

Further, to mitigate possible *data leakage* [35] in the training and testing stages, we employ 10-fold cross-validation during model evaluation. Data leakage occurs when the training set gains access to the test set, i.e., the two sets share identical or similar data, leading to overestimated evaluation results. While one single random train/test split could easily lead to the problem, 10-fold cross-validation is able to yield less biased results by training and testing the model with multiple different train/test splits. Meanwhile, to further reduce data leakage, we remove duplicate feature vectors in the training and test sets from the test set for each iteration of the cross-validation. Additionally, examination has revealed that the ratio of *duplicate* apps (i.e., apps with the same package names but different MD5 hash codes, which could also result in data leakage) in the training and test sets is fairly small ($< 1\%$). The detailed model configurations can be found at https://apichecker.github.io/.

4.3 Understanding Tradeoffs for API Selection

Leveraging the above dynamic analysis engine, we collect apps' API invocation data to study the critical design aspects of API (feature) selection and malware detection speed/accuracy. In the market-scale context, tradeoffs between the above aspects are critical since app markets often have stringent requirements for them. Regarding this, we present our understandings and insights that effectively guide us to our practical strategies in this section.

APIs' Correlations With Apps' Malice. An API's correlation with the malice of apps is an objective metric of API statistics [15]. In this paper, we adopt the *Spearman's rank correlation coefficient* (*SRC*) [36] to evaluate the statistical correlation.

Authorized licensed use limited to: Tsinghua University. Downloaded on June 12,2022 at 13:08:09 UTC from IEEE Xplore. Restrictions apply.

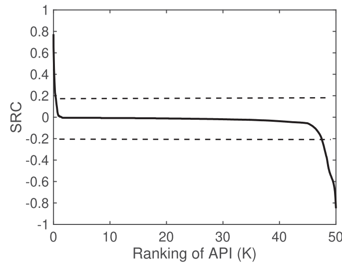


Fig. 6. Ranking of the ~50K framework APIs in terms of *SRC*s.

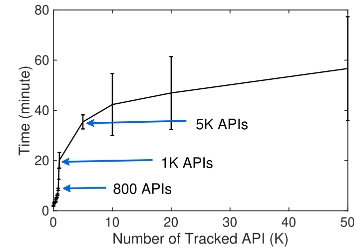


Fig. 8. Time consumption for tracking top-n correlated APIs.

With the dynamic analysis results and malice labels in the dataset, we measure *SRC* for each API regarding the malice of apps. Fig. 6 shows all the detected APIs sorted by their *SRC* values in descending order. Since the correlations (between APIs and the malice of apps) are considered non-trivial when $|SRC| \geq 0.2$ [37], we focus on these APIs and observe that there are 247 (APIs) whose $SRC \geq 0.2$, and 2,536 whose $SRC \leq -0.2$.

We next closely examine the characteristics of the above APIs and reveal several intriguing findings. Regarding the 247 APIs whose SRC is greater than 0.2, we notice that in certain apps some of these APIs are statistically correlated with each other. Therefore, it would seem somewhat redundant to include them all in app analysis. Nevertheless, we further find that these APIs often complement each other in terms of functionalities rather than being interchangeable, thereby in fact being beneficial and crucial to our analysis. Moreover, among the 2,536 APIs whose SRC is smaller than -0.2, we notice that almost all of them are *seldom* invoked by the examined apps in our dataset. Here *seldom* is empirically taken as being invoked by $\leq 0.1\%$ of the apps. Since including these rarely used APIs as features may lead to over-fitting problems in machine learning, we choose to exclude them in API selection. Nevertheless, there are 13 APIs with $SRC \leq -0.2$ that are frequently invoked by most apps to perform common system operations like disk I/O; such APIs are still retained in our analysis. In general, Fig. 7 demonstrates the top-1K framework APIs that are not seldom invoked with regard to their $|SRC|$s. From the figure we can observe that there are a total of 260 APIs (247 APIs with $SRC \geq 0.2$ and 13 APIs with $SRC \leq -0.2$, denoted as **Set-C**) that exhibit a non-trivial $|SRC|$.

Analysis Speed. As discussed in Section 4.2, the number of tracked APIs could greatly impact the analysis speed. We delve deeper into the problem, and present the relationship between the number of tracked APIs (n) and the analysis time (t) in Fig. 8, when we prioritize tracking APIs exhibiting a high correlation (with the malice of apps) that are not seldom invoked.

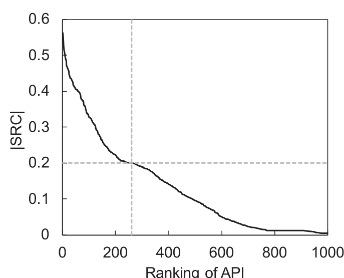


Fig. 7. Top-1K APIs in terms of $|SRC|$.

To describe the statistical relationship shown in Fig. 8, we propose a complex tri-modal distribution that can well fit the data. In detail, t first experiences linear growth when $n \in [0, 800)$, with the corresponding APIs being moderately used, more likely by malware given their high *SRC*s. Next when $n \in [800, 1K]$, t polynomially grows due to the involvement of APIs heavily used by both malware and benign apps, which are therefore less expressive regarding characterizing malware. Finally, t logarithmically grows when $n > 1K$ since the newly enrolled APIs are less frequently invoked. We use the following tri-modal distribution to describe the above characteristics:

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

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 employ the *coefficient of determination* [38] (R^2, ranging from 0 to 1) to evaluate how well the measured data can be expressed by the proposed statistical model. As a result, we have $R_1^2 = 0.96$, $R_2^2 = 0.99$, and $R_3^2 = 0.99$, which are all fairly close to 1 (the perfect fitting).

The above results demonstrate a rather complex relationship between APIs' SRC and their invocation frequency (and thus the incurred API interception overhead). A quantitative insight helps better balance the tradeoff between analysis speed (time) and detection accuracy. For the former, Fig. 8 indicates that when tracking up to the top-490 APIs, we can achieve an average analysis time of $\leq$5 minutes per app with the dynamic analysis engine. We will later explore the accuracy dimension.

Machine Learning Models. We evaluate the nine machine learning models introduced in Section 4.2 by tracking all the 50K framework APIs. The performance (in terms of precision and recall [39]) and overhead (in terms of training time) of each model are listed in Table 2. We find that the models manifest differently in the above aspects, particularly regarding training time and generalization. However, no model alone is able to surpass others in all the key metrics. Specially, we notice that the distributions of most API features in the dataset (e.g., in terms of invocation frequency) are rather skewed, resulting in generally better performance of tree-based models (GBDT and CART) and neural network models (ANN and DNN). However, such performance merits often come with a price of overfitting. Consequently, we select the classification model that can best balance them all – random forest (RF), which produces the best precision, a preferable recall, and an acceptable training time. Additionally, RF uses the ensemble learning

Authorized licensed use limited to: Tsinghua University. Downloaded on June 12,2022 at 13:08:09 UTC from IEEE Xplore. Restrictions apply.

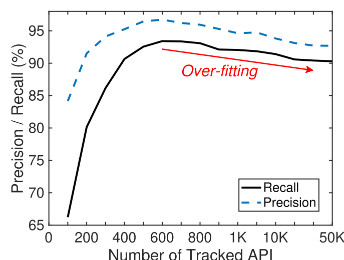


Fig. 9. Efficacy for tracking top-n correlated APIs, respectively.

technique that draws the power of multiple trained models to effectively overcome the overfitting problem and enhance its generalization ability [19]. Also, the model's simplicity (as compared to complex network models) and parallel nature of RF's internal trees make the training process rather efficient. As a matter of fact, when only the top-1K or top-490 APIs (referred to Fig. 8) are tracked in analysis, the best option is still the RF classifier.

Malware Detection Accuracy. Intuitively, tracking all the 50K APIs is supposed to yield the optimal accuracy despite its high time consumption. Fig. 9 shows the detection accuracy when tracking the top-n correlated APIs based on the RF classifier. The precision/recall obtained 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. It may be surprising that tracking fewer APIs in a strategical manner can lead to better precision and recall compared to tracking all the 50K APIs. Delving deeper, we realize that this counter-intuitive situation originates from the fact that most APIs are sparsely or seldom invoked by most apps. The enrollment of these rarely observed features could then lead to over-fitting of the trained models. As a result, tracking fewer APIs can in fact benefit both runtime speed and detection accuracy.

4.4 Key API Selection Strategy

We now describe our principled API selection strategy consisting of the following four steps.

Step 1. Selecting APIs that are most correlated with malware (**Set-C**). When tracking the top-260 correlated APIs (**Set-C**) as analyzed in Section 4.3, we can achieve 93.5 percent precision and 82.1 percent recall.

Step 2. Selecting APIs that require restrictive permissions (**Set-P**). In protection of the privacy/security of user information, an app needs to explicitly request permissions before obtaining certain information or executing certain functions [17]. There are three protection levels in Android permissions [40]: normal, signature, and dangerous. The APIs that require dangerous-level or signature-level permissions are oftentimes related to private user data (such as camera, SMS, and location data), thereby being crucial to malware detection. Leveraging the Axplorer [41] and PScout [42] static analysis tools, we pick out APIs related to restrictive permissions, and obtain a total of 112 APIs (referred to as **Set-P**). By solely tracking APIs in **Set-P**, we acquire 95.1 percent precision but rather low (71.3 percent) recall.

Step 3. Selecting APIs performing sensitive operations (**Set-S**). Unlike the permission levels, there is no "official" definition to sensitive operations. From previous work we find that five categories of operations are commonly

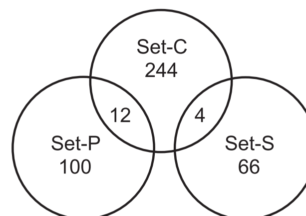


Fig. 10. Number of APIs in **Set-C**, **Set-P**, and **Set-S**.

exploited for attacks: 1) APIs that result in privilege escalation, e.g., shell command execution APIs [25], 2) APIs for database operations and file I/O, which are often used for privacy leakage attacks [26], 3) APIs that control key Android components, e.g., those for creating an Android window or overlay, which are adopted in attacks like Activity hijacking [43], 4) cryptographic operation APIs, which are frequently employed in ransomware attacks [44], and 5) APIs for dynamic code loading that load malicious payloads at runtime, enabling attacks such as update attack [25]. Combined with domain knowledge, we identify 70 APIs (referred to as **Set-S**) related to the above sensitive operations. By solely tracking these 70 APIs, we obtain 95 percent precision but poor (70.1 percent) recall.

Step 4. Combining the above. After combining the above strategies, i.e., **Set-P** $\bigcup$ **Set-S** $\bigcup$ **Set-C**, we have a total of 426 key APIs. Intuitively, doing so simultaneously considers statistical observations of the data and adversarial techniques based on domain knowledge. As shown in Fig. 10, only 16 overlapped APIs exist among the three sets, indicating an orthogonal relationship. Fig. 11 shows that when only the 426 key APIs are tracked, the per-app analysis time is 4.3 minutes on average, which is much shorter than 53.6 minutes (the average time consumption of tracking all the 50K APIs), and close to 2.1 minutes (the average time consumption tracking no APIs), on the dynamic analysis engine. In Section 5.1, we will further boost the detection speed by optimizing and engineering the underlying analysis infrastructure.

Further, we measure the detection precision and recall using the 426 key APIs with the nine mainstream ML models. As listed in Table 2, random forest still yields the highest precision (96.8 percent) and recall (93.7 percent), with its training (14.4 seconds) being much faster than that of more complex models (such as DNN and SVM). In comparison, simply tracking the top-426 correlated APIs (i.e., extending **Set-C**, also with RF) leads to 95.2 percent precision and 90.6 percent recall, as shown in Fig. 9. This confirms the advantage of the hybrid strategy over an individual strategy. It is noteworthy that selected 426 key APIs might not

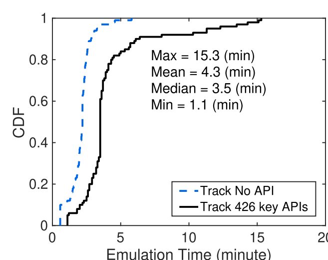


Fig. 11. Time consumption for tracking 426 key APIs.

Authorized licensed use limited to: Tsinghua University. Downloaded on June 12,2022 at 13:08:09 UTC from IEEE Xplore. Restrictions apply.

TABLE 3
Typical Call Chains of Key APIs in Logs

Call Chains of Key APIs	# of Invocations
GetDeviceId - GetSubscriberId	305200
SecretKeySpec - GetCipher - CipherInit	199395
HttpExecute - GetRunningTasks - DeleteFile	197255
GetLatitude - GetLongitude - ListDirectory	197255
CreateDirectory - DexFile - BaseDexClassLoader	45893
SetBackGroundColor - TVSetText	20931
CreateDirectory - CreateFile	14733

be the optimal API set with the best detection accuracy. We do not exhaustively pursue such an "optimal solution" for the enormous search space. Nevertheless, our proposed API selection strategy can be easily executed, and yields good results in the real-world deployment (Section 5.2).

Finally, we feel that it is difficult for adversaries to bypass our detection as we observe that a total of 5,242 APIs (10.5 percent of all framework APIs) is dependent on our 426 key APIs in implementation. Bypassing them all is highly challenging and tedious, if not impossible. Although using NDK [45] seems like a plausible alternative, heavy usage of NDK for common functions itself is an indicator of possible malicious behaviors.

4.5 Feature-Embedded Encoding Scheme

As introduced in Section 4.2, we employ the traditional One-Hot scheme to encode extracted features in the preliminary work [9]. The One-Hot encoding records the invocation status of tracked APIs, where 1 denotes that the corresponding API has been invoked and 0 means otherwise. Although this feature representation yields good detection accuracy (96.8 percent precision and 93.7 percent recall), it is inherently deficient in several aspects.

First, our original One-Hot encoding erases essential information in the features, including key APIs' invocation hierarchy and frequency, since dimensions in the One-Hot vector are orthogonal and binary. However, our observation suggests that these invocation patterns are often beneficial to malware detection [46]. As shown in Table 3, we notice that the invocations of some key APIs in malware are often invoked in a fixed order (i.e., forming a call chain), since complementary APIs need to be sequentially invoked to realize a certain function. Also, different API call chains have distinct invocation frequencies. Moreover, to retain such information, the One-Hot encoding can easily result in severe *dimension explosion*, i.e., uncontrolled growth of the feature vectors' sizes. This characteristic not only brings high memory overhead, but also is somewhat unfriendly to tree-based models (e.g., random forest) due to expensive tree split [47].

Classic approaches that can efficiently retain the fine-grained features are *word2vec* [11] and *doc2vec* [48] encoding. Compared to the bag-of-words model of One-Hot encoding, the word embedding scheme is known to be able to preserve the semantic patterns of input data [49]. However, our experiments show that none of them can achieve a desirable performance due to the skewed distribution of API invocation frequencies in some apps. An app that frequently invokes certain key APIs is often mistaken as malicious by *word2vec* (resulting in 8 percent false positive), while one that rarely uses key APIs are usually overlooked by *doc2vec* (resulting in 9 percent false negative). To tackle the issues, we devise a novel feature-embedded encoding scheme. Specifically, we first adopt *word2vec* to convert each key API into a word embedding vector using the apps' invocation logs. Given that directly using the vector as the representation of each invocation record can render frequently-used APIs being heavily repeated (henceforth suboptimal precision), we replace each original binary (corresponding to a key API) in the One-Hot vector with the associated word embedding vector. In this way, we can not only largely avoid the impact of skewed API invocations, but also achieve $\sim 20\times$ reduction in memory consumption compared to solely using the One-Hot scheme for the data. Furthermore, we evaluate the accuracy of different ML models and find that random forest still prevails with 97.1 percent precision and 96.9 percent recall.

4.6 Feature Engineering and Exposure

Hidden Feature Identification. Our examination on the dynamic analysis results of ~500K apps indicates that solely relying on Android framework APIs is problematic due to adversaries' bypassing API invocations through other mechanisms. In practice, adversaries often exploit two alternatives for triggering a target API's function without direct invocation: 1) internal/hidden APIs, which are triggered by special techniques such as Java reflection [50], 2) intents, the major IPC mechanism in Android that enables an app to delegate sensitive operations to other apps/services, and to detect system events [51]. In our dataset, both techniques are actively exploited as camouflage for the actual invocations of certain APIs in malware. So, these "covert" API invocations become hidden features requiring further explorations.

Fortunately, we can effectively and efficiently mitigate this limitation without any dynamic overhead. In detail, we add the requested permissions and the used intents as auxiliary features to help uncover hidden API invocations. This is because invoking internal/hidden APIs requires the malware to first request permissions, and as far as we know there is no practical workaround [51]. Also, we can collect the requested permissions through static analysis of apps' metadata, and monitor the used intents by checking the parameters of intent-related APIs tracked in **`Set-S`**.

As shown in Fig. 13, using permissions and key APIs as features ("A+P") can increase the precision from 97.1 to 98.6 percent, and the recall from 96.9 to 97.5 percent. Intriguingly, combining permissions and intents alone ("P+I") can also obtain sound results (98 percent precision and 96.6 percent recall), suggesting that these two mechanisms are heavily exploited by today's Android malware. Finally, compared to purely relying on the 426 key APIs ("A"), a joint consideration of all three feature categories ("A+P+I") achieves the best results, i.e., improving the precision to 98.9 percent, recall to 98.1 percent, and *F1-score* to 98.2 percent. Here *F1-score* denotes the harmonic mean of precision and recall: $2 \times \frac{precision \times recall}{precision + recall}$.

Feature Exposure. To better expose and manifest these features, we further improve our automatic UI exploration

Authorized licensed use limited to: Tsinghua University. Downloaded on June 12,2022 at 13:08:09 UTC from IEEE Xplore. Restrictions apply.

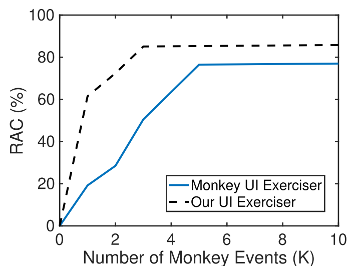


Fig. 12. Comparison of *Monkey* UI exerciser with our UI exerciser regarding RAC.

methodology targeting the *Monkey* exerciser's major deficiencies – redundant actions and action loops [12], which could lead to inefficient and ineffective UI testing. For redundant actions, we fine-tune the portions of different UI events (e.g., touch and swipe) based on the specific category of an app, since apps in the same category usually implement similar operations and action flows [52], and thus often require similar UI events to expose their features. In detail, we tune the related parameters of *Monkey*, e.g., `pct-touch` that controls the percentage of the touch event. The concrete parameter settings for a certain category are configured based on statistic analysis of the number of callback functions for different UI events (e.g., `OnClick` for the touch event) in the apps' code. Note that we perform category-level instead of app-level analysis for parameter settings since many apps are obfuscated and require high workload to analyze.

On the other hand, action loops root in the random nature of *Monkey*'s generated events, which is inherently limited due to a lack of information regarding an app's interactable UI components and visited `Activities`. To address this problem, we leverage the UI Automator [53] and Robotium [54] to obtain an app's UI layout structures containing component information as heuristics for triggering actions and `Activities`. Also, we record visited `Activities` to detect and avoid severe action loops. As shown in Fig. 12, our optimized UI exerciser can achieve a similar RAC as the original *Monkey* with only 3K UI events, leading to 15 percent reduction in detection time without any accuracy loss.

5 System Development

In this section, we present our optimization to the underlying emulation infrastructure. Then we deploy APIChecker and evaluate its real-world performance, while the model evolution is described in details.

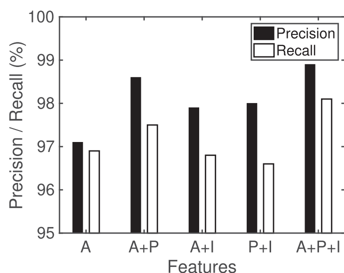


Fig. 13. Benefits of auxiliary features (A: key APIs, P: permissions, I: intents).

5.1 Emulation Infrastructure Optimization

Enhancing Runtime Performance. Having derived the desired features, we next focus on enhancing the underlying emulation infrastructure of APIChecker to improve its runtime performance. We discover that the performance of default Google Android emulator [27] is sub-optimal given its heavyweight, full-system emulation built atop of QEMU [55]. While this may satisfy the need of in-lab, controlled analysis as we perform in Section 4, a real-world production system that examines a large number of apps has much lower tolerance fo long detection delays, since they may negatively impact developers' experiences, as well as increase the infrastructure expense of market operators.

Fig. 11 shows that for 30 percent apps, the original dynamic analysis engine (Section 4.2) requires more than 5 minutes to scan each of them, which is longer than the typical detection time of Google Bouncer [24]. To boost the runtime performance, apart from multiple optimizations already introduced to APIChecker at the detection engine level (Section 4), we further make system-level optimizations to the underlying emulation infrastructure. In detail, we build a lightweight emulation system to efficiently run Android and apps on powerful commodity x86 servers. To this end, we leverage Android-x86 [13], an open-source x86 port of ARM-based Android, to largely avoid performance degradation induced by software-assisted virtualization used to address the ISA gap between ARM and x86. With Android-x86, we are then able to introduce various hardware-assisted virtualization techniques into the underlying runtime to further improve the performance, including Intel VT [56] and KVM [57] that enable our system to fully explore the power of x86 CPUs. Also, we adopt VirtIO [58], a para-virtualization technique, to realize GPU-assisted acceleration in graphic rendering (a computation-intensive job originally executed by the host CPU rather than the dedicated GPU). Specifically, we intercept guest (Android-x86) side graphic driver's "micro" instructions (that are decomposed and reconstructed from OpenGL instructions by the graphic driver) within apps' rendering pipelines, and execute them atop the GPU on the host (x86 server) side, thereby essentially outperforming the original CPU-based software rendering. Meanwhile, for apps that adopt Android's *native* ARM libraries that may not be able to run on Android-x86 [59], we integrate the state-of-the-art *dynamic binary translation* (DBT) tool developed by Intel (Houdini [14]) to translate the apps' ARM instructions into x86 instructions.

In total, combined with our enhanced *Monkey* (Section 4.6) and Xposed, our lightweight Android emulator runs atop of a physical x86 server equipped with a 5×4-core Xeon CPU@2.50 GHz and 256-GB DDR memory. To fully exploit the server's hardware resources, we concurrently run multiple emulators on it, with each emulator pinned to a CPU core. Specifically, 16 emulators run on 16 cores in parallel, with the remaining 4 cores being used for task scheduling, status monitoring, and information logging.

Despite that our lightweight emulation infrastructure substantially surpasses our original emulator in terms of runtime speed, its compatibility with Android apps slightly drops. To avoid failures of examination caused by incompatibility, we modify the `SystemServer` service in Android-x86 to report app hangs or crashes to the 4 cores used for task

Authorized licensed use limited to: Tsinghua University. Downloaded on June 12,2022 at 13:08:09 UTC from IEEE Xplore. Restrictions apply.

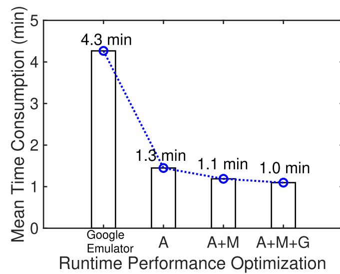


Fig. 14. Effects of optimization technologies (A:Android-x86, M: Enhanced Monkey, G:GPU).

scheduling, status monitoring, and information logging. We observe that only a trivial portion ($< 1\%$) of apps fail to run on the lightweight engine, mainly due to compatibility issues rooting in Android-x86 and DBT. For incompatible apps, we roll back to the original emulator for successful analysis. In this way we can ensure the reliability of APICHECKER despite sacrificing a small portion of detection speed—all submitted apps are analyzed on our production system.

Our customized infrastructure enables APICHECKER to efficiently analyze apps, leading to around 77 percent reduction in detection time compared to the default Android emulator under the same hardware configuration, without any accuracy loss. Tracking the 426 key APIs, we evaluate the per-app scan time with the default Android emulator and our customized emulator on the same x86 server (incompatible apps are also considered). Fig. 14 demonstrates the optimization effects of all the introduced enhancing techniques. As shown, Android-x86 and the accompanied hardware-assisted opportunities yield the most effective optimization in runtime performance, reducing 70 percent of the per-app scan time.

Preventing Detection Evasion. In Section 4.2 we have introduced the techniques we used in the preliminary work [9] to prevent emulator identification and detection evasion. Basically, our methods of concealing the emulator are static modifications to the system prior to emulation, including parameter configurations and library obfuscations. Although these improvements are able to deal with static system analysis, they are ineffective facing apps' dynamically scrutinizing system operations and user behaviors during emulation. To address this, we devise two techniques as follows to enable in-situ evasion prevention. First, we intercept calls into crucial system interfaces regarding apps' operations and system states (e.g., `LocationManager`, `NetworkInterface` and `BatteryManager`) to return similar values as in real devices.

Second, we adaptively tune the interval of *Monkey* events. The malware can take the interval as a critical indicator of emulation and detection; if the interval is smaller than a given threshold, they will suppress malicious activities (become idle). In the preliminary work [9], the interval is fixed. Hence, malicious apps could detect our emulator by simply raising the corresponding threshold. Therefore, once an app is constantly idle during emulation without responding to input events (indicating that the app may have recognized the emulation environment), we exponentially increase the interval from 500 ms to quickly reach an ideal interval. Note that we stop increasing the interval once the overall waiting is over 1 minute to avoid significant overhead. If the waiting time exceeds 1 minute when checking an app, it is then considered to be highly suspicious and submitted for further manual inspection. Also, real-world observations show that in almost all cases ($> 99\%$) this mechanism causes less than 1 second overhead to the analysis.

With these in-situ optimizations, controlled experiments show that only 0.4 percent apps invoke fewer features than on real devices, which is much smaller than that (1.4 percent) without optimizations.

5.2 System Deployment and Performance

Distributed Deployment. Originally, each app's analysis is composed of nine steps as depicted in Fig. 1, among which several steps in fact do not rely on each other and thus can be carried out in parallel. As a result, we reorganize the execution sequence of the analysis steps, as shown in Fig. 18. We convert the original monolithic back-to-back execution manner into a loosely-coupled pipeline consisting of four components – core scheduler, dynamic analysis, metadata analysis, and model classification, which work together to implement a publish-subscribe system. In detail, the core scheduler extracts submitted tasks from a global message queue and publishes it to the other components. Meanwhile, dynamic and metadata analysis channels poll for unfinished tasks and perform app emulation (corresponding to Step ①~⑤ in Fig. 1) and metadata extraction (Step ⑥⑦ in Fig. 1), respectively. When all the above analysis tasks are finished, model classification channel can then utilize the collected feature data to determine an app's malice (Step ⑧ in Fig. 1). Consequently, the per-app scan time is reduced from 1 minute to 0.9 minutes.

Integration to T-Market. APICHECKER was integrated to T-Market and has been running since March 2018. It examines about 12K apps every day with a single commodity server, atop of which 16 emulators run in parallel on 16 cores.

In detail, APICHECKER installs a submitted APK file on an idle emulator. It then automates a series of interactions with the app using our improved UI exerciser based on *Monkey*, and logs fine-grained raw API invocation data in the meantime (Section 4). Leveraging our feature encoding scheme, we convert the collected raw data into a feature vector, which is then input into the random forest classifier to determine the malice of the app. In the subsequent 27 months (from March 2018 to May 2020), APICHECKER uncovered ~5K suspicious apps (i.e., vetted as malicious by us) every month. To measure the online detection accuracy of APICHECKER, we validate its detection results against those generated by T-Market's original app review process (*cf.* Section 2) and our own manual checking. This validation process has very high precision and recall but involves heavy labor work. As shown in Fig. 15, the per-month precision during deployment is over 98 percent ($98.5\% \sim 99.1\%$) and the recall is over 96 percent ($96.5\% \sim 98.1\%$).

As introduced in Section 4.1, the original sophisticated app review process in T-Market has very high precision and recall, which enables us to obtain an in-depth understanding of the $< 2\%$ false positive and $< 4\%$ false negative. In detail, the 2 percent false positive (apps) frequently adopt key selected features, making them appear similar to

Authorized licensed use limited to: Tsinghua University. Downloaded on June 12,2022 at 13:08:09 UTC from IEEE Xplore. Restrictions apply.

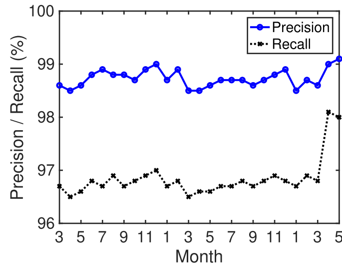


Fig. 15. Online performance of APICHECKER over 27 months, from March 2018 to May 2020.

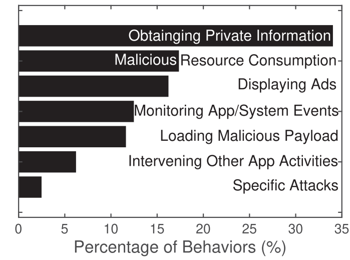


Fig. 16. Distribution of malware's malicious behaviors.

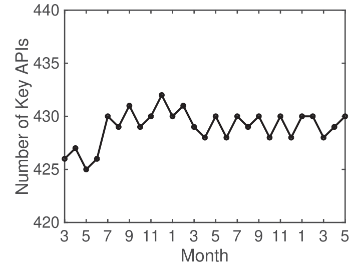


Fig. 17. Evolution of the number of our selected key APIs from March 2018 to May 2020.

malware in terms of API, permission, and/or intent usage. Specifically, there are on average 800 apps vetted as malicious by APICHECKER among all the ~10K apps submitted to T-Market each day. Fortunately, more than 90 percent of them are updated apps that can be quickly validated based on their previous versions. This leads to fewer than 80 apps that require manual efforts for validation, incurring an acceptable overhead to T-Market. Therefore, we choose to actively avoid the false positive apps through manual inspection every day. In contrast, the 4 percent false negative barely use the 426 key APIs we track according to our examination of their logs. In fact, these apps often have rather simple functionalities (e.g., displaying advertisements) that are commonly taken as "mild" security threats to end users. Consequently, for the false negative we only passively mitigate them upon users' complaints.

For other app markets, we note that integrating our system is not difficult as the mature program analysis and ML techniques in this paper are easy to implement, deploy, and maintain.

Important Features and Malicious Behaviors. To understand key features that are most crucial to our malware detection model, we explore the Gini indices of each tracked features [9], including key APIs, used intents and permissions. The Gini index is a common metric derived from a trained random forest model to quantify feature importance. We associate top ranking key features with a total of seven behaviors as follows: 1) Attempting to obtain user devices' private information such as SMS message (e.g., API `SmsManager_sendTextMessage`[1]), phone number (e.g., permission `TelephonyManager_getLine1Number`), and MAC address (e.g., API `WifiInfo_getMacAddress`); 2) Malicious resource consumption such as constantly lurking in the background that drains battery and is a common indicator of malicious activities [6] (e.g., API `Context_bindService`); 3) Displaying ads (e.g., API `WindowManager_addView`); 4) Monitoring system/app-level events such as critical device activities (e.g., permission `RECEIVE_BOOT_COMPLETED`), network changes (e.g., intent `wifi.STATE_CHANGE`), and privilege acquiring (e.g., intent `DEVICE_ADMIN_ENABLED`); 5) Loading malicious payloads (e.g., API `system_InMemoryDexClassLoader`); 6) Intervening other apps such as app hijacking [60] (e.g., API `ActivityManager_getRunningTasks`); 7) Enabling specific attacks like overlay-based attacks [61] (e.g., permission `SYSTEM_ALERT_WINDOW`).

1. `android.telephony.SmsManager.sendTextMessage` is abbreviated to `SmsManager_sendTextMessage` here. We also use similar aliases for other mentioned APIs.

Fig. 16 illustrates the distribution of the abovementioned malicious behaviors. As shown, obtaining users' private information accounts for the largest portion, while malware launching specific attacks such as overlay-based attacks is much less pervasive. For app categories, gaming apps are most likely to manifest malicious behaviors ($\sim 14\%$ are malicious), which tend to be obtaining private information, displaying ads and loading malicious payloads. In addition, office and system apps usually acquire restrictive permissions to intercept system- or app-level events.

Model Evolution. To accommodate continuously emerging apps and SDK's upgrades every several months, during operation we note that the detection model and the key API set require periodic evolution. Currently, the period of key API updates and model retraining is empirically configured as one month. The retraining dataset includes the original dataset (Section 4.1) and the subsequent newly submitted apps. The malice of new apps is determined by both APICHECKER and manual inspection, bearing no false positives and a small portion of false negatives. Moreover, our key API selection strategy stays unchanged as described in Section 4.4.

Fig. 17 shows variations in our selected key APIs' number from March 2018 to May 2020 when we deploy APICHECKER, which only slightly fluctuates between 425 and 432. Hence, the per-app detection time remains stable. Note that the model evolution process is taken into account when

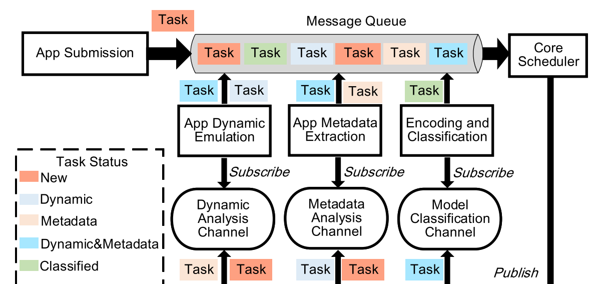


Fig. 18. APICHECKER's distributed deployment in T-Market.

Authorized licensed use limited to: Tsinghua University. Downloaded on June 12,2022 at 13:08:09 UTC from IEEE Xplore. Restrictions apply.

reporting the online performance. As a result, the fluctuation in detection accuracy caused by API set variations is moderate, leading to $98.5\% \sim 99.1\%$ precision, $96.5\% \sim 98.1\%$ recall, and $97.5\% \sim 98.6\%$ F1-score. In general, APICHECKER is robust to API evolution.

6 Conclusion

ML-based mobile malware detection has been trending in the last decade. However, till now we have yet to see a realistic solution for large-scale app markets, which are crucial to the triumph of today's mobile ecosystem. To uncover and overcome the real-world challenges, in collaboration with a major app market, we implement, deploy, and maintain an ML-based malware detection system that is both effective and efficient. By examining the runtime usage of strategically selected key APIs, as well as other auxiliary features, it has been detecting Android malware for over two years with several system-level optimizations, including our enhanced fast emulation engine, automatic model evolution, and practical false positive/negative mitigation. We hope our measurement results, system designs, deployment experiences, and data/tool release will contribute to the community.

Acknowledgment

This work was supported in part by the National Key R&D Program of China under Grant 2018YFB1004700, in part by the NSF of China under Grants 61902211, 61972313, 61822205, 61632020, and 61632013, in part by the NSF of Tianjin under Grant 18JCQNJC69900, in part by the Postdoctoral Science Fund of China under Grant 2019M663725, and in part by the BNRist. Liangyi Gong and Hao Lin are Co-primary authors.

References

[1] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets," in *Proc. 19th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2012, pp. 50–52.

[2] S. Arzt *et al.*, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2014, pp. 259–269.

[3] M. Wong and D. Lie, "IntelliDroid: A targeted input generator for the dynamic analysis of android malware," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016, pp. 21–24.

[4] Tencent app market, 2020. [Online]. Available: https://sj.qq.com/myapp/

[5] D. Wang *et al.*, "An analogous wood barrel theory to explain the occurrence of hormesis: A case study of sulfonamides and erythromycin on *Escherichia coli* growth," *PloS One*, vol. 12, no. 7, 2017, Art. no. e0181321.

[6] G. Tao, Z. Zheng, Z. Guo, and M. R. Lyu, "MalPat: Mining patterns of malicious and benign android apps via permission-related APIs," *IEEE Trans. Rel.*, vol. 67, no. 1, pp. 355–369, Mar. 2018.

[7] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-level features for robust malware detection in Android," in *Proc. ACM Int. Conf. Secur. Privacy Commun. Syst.*, 2013, pp. 86–103.

[8] M. Yang, S. Wang, Z. Ling, Y. Liu, and Z. Ni, "Detection of malicious behavior in Android apps through API calls and permission uses analysis," *Concurrency Comput., Practice Experience*, vol. 29, no. 19, 2017, Art. no. e4172.

[9] L. Gong *et al.*, "Experiences of landing machine learning onto market-scale mobile malware detection," in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, Art. no. 2.

[10] One-hot encoding, 2016. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/one-hot-encoding

[11] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *Proc. Int. Conf. Learn. Representations*, 2013.

[12] Y. L. Arnatovich, M. N. Ngo, T. H. B. Kuan, and C. Soh, "Achieving high code coverage in android UI testing via automated widget exercising," in *Proc. IEEE 23rd Asia-Pacific Softw. Eng. Conf.*, 2016, pp. 193–200.

[13] Android-x86 – Porting Android to x86, 2009. [Online]. Available: http://www.android-x86.org/

[14] Intel Houdini, 2016. [Online]. Available:https://osdn.net/projects/android-x86/scm/git/vendor-intel-houdini/

[15] A. Sharma and S. K. Dash, "Mining API calls and permissions for android malware detection," in *Proc. Int. Conf. Cryptol. Netw. Secur.*, 2014, pp. 191–205.

[16] A. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proc. 18th ACM Conf. Comput. Commun. Secur.*, 2011, pp. 627–638.

[17] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "DroidMat: Android malware detection through manifest and API calls tracing," in *Proc. IEEE 7th Asia Joint Conf. Inf. Secur.*, 2012, pp. 62–69.

[18] M. Grace *et al.*, "RiskRanker: Scalable and accurate zero-day android malware detection," in *Proc. 10th Int. Conf. Mobile Syst. Appl. Services*, 2012, pp. 281–294.

[19] H. Cai, N. Meng, B. Ryder, and D. Yao, "DroidCat: Effective android malware detection and categorization via app-level profiling," *IEEE Trans. Inf. Forensics Security*, vol. 14, no. 6, pp. 1455–1470, Jun. 2019.

[20] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue, "Droid-Sec: Deep learning in android malware detection," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 371–372, 2014.

[21] W. Wu and S. Hung, "DroidDolphin: A dynamic android malware detection framework using big data and machine learning," in *Proc. ACM Conf. Res. Adaptive Convergent Syst.*, 2014, pp. 247–252.

[22] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: Effective and explainable detection of android malware in your pocket," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 23–26.

[23] XposedBridge, 2016. [Online]. Available: https://github.com/rovo89/XposedBridge/wiki/Development-tutorial

[24] Dissecting the Android bouncer, 2012. [Online]. Available: https://jon.oberheide.org/files/summercon12-bouncer.pdf

[25] X. Jiang and Y. Zhou, "Dissecting android malware: Characterization and evolution," in *Proc. IEEE Symp. Secur. Privacy*, 2012, pp. 95–109.

[26] W. Enck *et al.*, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, 2014, Art. no. 5.

[27] Emulator, 2020. [Online]. Available: https://developer.android.com/studio/run/emulator

[28] Monkey test tool, 2008. [Online]. Available:https://developer.android.com/studio/test/monkey.html

[29] M. Sun *et al.*, "Design and implementation of an android host-based intrusion prevention system," in *Proc. 30th Annu. Comput. Secur. Appl. Conf.*, 2014, pp. 226–235.

[30] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: Hindering dynamic analysis of android malware," in *Proc. ACM 7th Eur. Workshop Syst. Secur.*, 2014, Art. no. 5.

[31] S. Dey, N. Roy, W. Xu, and S. Nelakuditi, "ACM HotMobile poster: Leveraging imperfections of sensors for fingerprinting smartphones," *ACM SIGMOBILE Mobile Comput. Commun. Rev.*, vol. 17, no. 3, pp. 21–22, 2013.

[32] Anti-hooking techniques, 2015. [Online]. Available: https://d3adend.org/blog/?p=589

[33] J. Lee and H. Kim, "QDroid: Mobile application quality analyzer for app market curators," *Mobile Inf. Syst.*, vol. 2016, pp. 1–11, 2016.

[34] Scikit-learn, 2010. [Online]. Available: http://scikit-learn.org/stable/index.html

[35] S. Kaufman, S. Rosset, C. Perlich, and O. Stitelman, "Leakage in data mining: Formulation, detection, and avoidance," *ACM Trans. Knowl. Discov. Data*, vol. 6, no. 4, pp. 1–21, 2012.

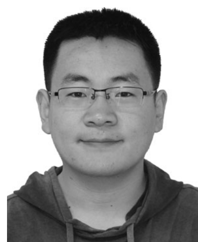
[36] J. H. Zar, "Significance testing of the spearman rank correlation coefficient," *J. Amer. Statist. Assoc.*, vol. 67, no. 339, pp. 578–580, 1972.

[37] S. Silvestri, R. Urgaonkar, M. Zafer, and B. J. Ko, "A framework for the inference of sensing measurements based on correlation," *ACM Trans. Sensor Netw.*, vol. 15, no. 1, pp. 1–28, 2018.

[38] M. N. Schulz, J. Landström, and R. E. Hubbard, "MTSA–A Matlab program to fit thermal shift data," *Analytical Biochem.*, vol. 433, no. 1, pp. 43–47, 2013.

Authorized licensed use limited to: Tsinghua University. Downloaded on June 12,2022 at 13:08:09 UTC from IEEE Xplore. Restrictions apply.

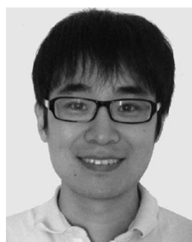
[39] X. Fan *et al.*, "BuildSenSys: Reusing building sensing data for traffic prediction with cross-domain learning," *IEEE Trans. Mobile Comput.*, to be published, doi: 10.1109/TMC.2020.2976936.
[40] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Permission evolution in the android ecosystem," in *Proc. 28th Annu. Comput. Secur. Appl. Conf.*, 2012, pp. 31–40.
[41] Axplorer, 2017. [Online]. Available:https://github.com/reddr/axplorer
[42] PScout, 2018. [Online]. Available: https://github.com/dlgroupuoft/PScout
[43] Q. A. Chen, Z. Qian, and Z. M. Mao, "Peeking into your app without actually seeing it: UI state inference and novel Android attacks," in *Proc. 23rd USENIX Conf. Secur. Symp.*, 2014, pp. 1037–1052.
[44] T. Yang, Y. Yang, K. Qian, D. C.-T. Lo, Y. Qian, and L. Tao, "Automated detection and analysis for Android ransomware," in *Proc. IEEE 17th Int. Conf. High Perform. Comput. Commun. IEEE 7th Int. Symp. Cyberspace Safety Secur. IEEE 12th Int. Conf. Embedded Softw. Syst.*, 2015, pp. 1338–1343.
[45] Android.com, "NDK," 2008. [Online]. Available: https://developer.android.com/ndk
[46] A. Arora, S. K. Peddoju, and M. Conti, "PermPair: Android malware detection using permission pairs," *IEEE Trans. Inf. Forensics Security*, vol. 15, pp. 1968–1982, Oct. 2020.
[47] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2016, pp. 785–794.
[48] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proc. 31st Int. Conf. Mach. Learn.*, 2014, pp. 1188–1196.
[49] M. Kusner, Y. Sun, N. Kolkin, and K. Weinberger, "From word embeddings to document distances," in *Proc. 32nd Int. Conf. Mach. Learn.*, 2015, pp. 957–966.
[50] A. Kharaz, S. Arshad, C. Mulliner, W. Robertson, and E. Kirda, "UNVEIL: A large-scale, automated approach to detecting ransomware," in *Proc. 25th USENIX Conf. Secur. Symp.*, 2016, pp. 757–772.
[51] A. Felt *et al.*, "Permission re-delegation: Attacks and defenses," in *Proc. 20th USENIX Conf. Secur.*, 2011, Art. no. 22.
[52] S. Hao *et al.*, "PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps," in *Proc. 12th Annu. Int. Conf. Mobile Syst. Appl. Services*, 2014, pp. 204–217.
[53] UI Automator, 2009. [Online]. Available: https://developer.android.com/training/testing/ui-automator
[54] Robotium, 2018. [Online]. Available: https://github.com/robotiumtech/robotium
[55] F. Dang *et al.*, "Understanding fileless attacks on Linux-based IoT devices with HoneyCloud," in *Proc. 17th Annu. Int. Conf. Mobile Syst. Appl. Services*, 2019, pp. 482–493.
[56] R. Uhlig *et al.*, "Intel virtualization technology," *Computer*, vol. 38, no. 5, pp. 48–56, 2005.
[57] C. Dall and J. Nieh, "KVM/ARM: The design and implementation of the Linux ARM hypervisor," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 333–348, 2014.
[58] R. Russell, "virtio: Towards a De facto standard for virtual I/O devices," *ACM SIGOPS Operating Syst. Rev.*, vol. 42, pp. 95–103, 2008.
[59] Q. Yang *et al.*, "Mobile gaming on personal computers with direct android emulation," in *Proc. 25th Annu. Int. Conf. Mobile Comput. Netw.*, 2019, Art. no. 19.
[60] V. G. Shankar and G. Somani, "Anti-Hijack: Runtime detection of malware initiated hijacking in android," *Procedia Comput. Sci.*, vol. 78, pp. 587–594, 2016.
[61] Y. Yan *et al.*, "Understanding and detecting overlay-based android malware at market scales," in *Proc. 17th Annu. Int. Conf. Mobile Syst. Appl. Services*, 2019, pp. 168–179.



Liangyi Gong received the BS and PhD degrees from the School of Computer Science and Technology, Harbin Engineering University, Harbin, China, in 2010 and 2016, respectively. He is a post-doctoral researcher with the School of Software and BNRist, Tsinghua University. His research interests include the areas of network security and mobile computing.

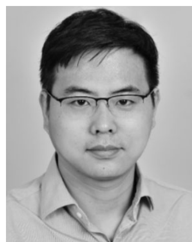
Hao Lin received the BS degree from the School of Software, Tsinghua University, Beijing, China, in 2020. He is currently working toward the PhD degree with the School of Software, Tsinghua University, Beijing, China. His research areas mainly include network measurement and mobile systems.

Zhenhua Li (Member, IEEE) received the BS and MS degrees from Nanjing University, Nanjing, China, in 2005 and 2008, respectively, and the PhD degree from Peking University, Beijing, China, in 2013, all in computer science and technology. He is currently an associate professor with the School of Software and BNRist, Tsinghua University. His research areas cover cloud computing/storage/download, big data analysis, content distribution, and mobile internet. He is a member of ACM.



Feng Qian (Member, IEEE) received the BS degree from the Shanghai Jiao Tong University, Shanghai, China, and the PhD degree from the University of Michigan, Ann Arbor, Michigan. He is an assistant professor with the Computer Science and Engineering Department, University of Minnesota, Twin Cities. Prior to joining UMN, he worked with AT&T Labs and Indiana University. His research interests cover mobile systems, AR/VR, wearable computing, real-world system measurements, and system security.

Yang Li received the BS degree from the School of Software, Tsinghua University, Beijing, China, in 2018. He is currently working toward the ME degree with the School of Software, Tsinghua University, Beijing, China. His research areas mainly include big data analysis, machine learning, cloud computing/storage, and network measurement.



Xiaobo Ma (Member, IEEE) is currently an associate professor with MOE Key Lab for Intelligent Networks and Network Security, Faculty of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an, China. He is also with Shaanxi Province Key Laboratory of Computer Network, Xi'an, China. He served as a co-chair of ACM CoNEXT 2019 Student Workshop. His research interest include cyber security.

Yunhao Liu (Fellow, IEEE) received the BS degree from Automation Department, Tsinghua University, Beijing, China, and the MS and PhD degrees in computer science and engineering from Michigan State University, East Lansing, Michigan. He is currently a professor and dean at Global Innovation Exchange, Tsinghua University. His research interests include sensor network and IoT, RFID, distributed systems, and cloud computing. He is a fellow of ACM.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: Tsinghua University. Downloaded on June 12,2022 at 13:08:09 UTC from IEEE Xplore. Restrictions apply.