Virtual Device Farms for Mobile App Testing at Scale

A Pursuit for Fidelity, Efficiency, and Accessibility

Hao Lin[†], Jiaxing Qiu[†*], Hongyi Wang[†*], Zhenhua Li[†✉], Liangyi Gong[‡]
Di Gao[†], Yunhao Liu[†], Feng Qian[§], Zhao Zhang[*], Ping Yang[*], Tianyin Xu[¶]
[†]Tsinghua University [*]ByteDance Inc. [‡]CNIC of CAS [§]University of Southern California [¶]UIUC

ABSTRACT

Virtual devices based on device emulation have been widely used in lab research of mobile app testing for their efficiency and low cost. However, it remains controversial to use virtual devices for app testing in industry, given the inherent difficulties of high-fidelity emulation across diverse mobile systems and devices. Hence, mobile app companies still rely on physical device farms or services like AWS Device Farm.

This paper presents our effort to analyze, improve, and effectively use virtual devices for large-scale testing of mobile apps like Douyin. Our goal is to understand the fidelity of virtual devices and to explore how to better utilize virtual devices to improve the efficiency and accessibility of mobile app testing in industrial settings. Our study is conducted on a massive commercial testing infrastructure that deploys a physical device farm and its virtualized counterpart.

We show that high-fidelity app testing can be achieved by sensible design and implementation of virtual device farms. With that, we find that major discrepancies are no longer caused by commonly believed factors like hardware heterogeneity and system customizations, but due to non-standard, uncoordinated, and occasionally defective vendor-specific services and drivers, as well as defense mechanisms against malicious app behavior. We present effective solutions to address those problems to significantly improve testing fidelity. We also share our experiences of using virtual device farms to substantially improve the efficiency and accessibility of mobile app testing, without compromising safety.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Virtual machines**.

KEYWORDS

Mobile App Testing; Virtualization; Fidelity; Efficiency.

ACM MobiCom '23, October 2–6, 2023, Madrid, Spain

ACM ISBN 978-1-4503-9990-6/23/10.
https://doi.org/10.1145/3570361.3613259

ACM Reference Format:
Hao Lin, Jiaxing Qiu, Hongyi Wang, Zhenhua Li, Liangyi Gong, Di Gao, Yunhao Liu, Feng Qian, Zhao Zhang, Ping Yang, Tianyin Xu. 2023. Virtual Device Farms for Mobile App Testing at Scale: A Pursuit for Fidelity, Efficiency, and Accessibility. In *The 29th Annual International Conference on Mobile Computing and Networking (ACM MobiCom '23), October 2–6, 2023, Madrid, Spain.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3570361.3613259

1 INTRODUCTION

Mobile app testing, especially for Android, is known to be challenging due to the openness of the ecosystem—Android phone vendors extensively customize hardware and system components, leading to significant behavioral differences across hundreds of phone models released every year. As a result, comprehensive testing of mobile apps against different device models incurs high cost for purchasing and operating phones. Today, mobile app companies either build their own device farms or pay for cloud services like AWS Device Farm [1] and Google Firebase Test Lab [31]. For example, Douyin [18], a mobile app with 1.5 billion users, is tested on a large-scale physical device farm consisting of ~6,000 phones, operated by a dedicated team of 15 engineers.

A natural direction to reduce the operation cost of continuous mobile app testing is to use *virtual device farms*, where mobile devices are emulated on commodity servers using virtualization techniques [22, 78]. Virtual device farms are scalable, elastic, and cost-effective. Virtual devices are much easier to manage compared with physical devices. For example, it is easier to repair or replace hardware components of a server, compared with a mobile device. In addition, virtualization enables useful features for testing and debugging, such as instrumentation [51], snapshot [17], and memory introspection [16, 76] not offered by physical devices.

In fact, virtual devices have been extensively used by lab research of mobile app testing [10, 12, 15, 15, 17, 19, 23–25, 34, 44, 52, 55, 61–63, 70–72, 74, 77]. While being a common practice in research papers on software testing, it remains controversial to use virtual devices for mobile app testing across diverse device models in industry. The main concern comes from the inherent difficulties of high-fidelity device emulation across the diverse, opaque, and ever-growing customized devices—the discrepancies between the

physical and virtual devices may impair test results, leading to both escapes of bugs and false alarms [33, 67, 68]. For mobile apps with a global user base, seemingly small-numbered discrepancies could have magnified impacts.

In this paper, we confront the essential problems of using virtual device farms for large-scale, continuous mobile app testing in industrial settings. We present an effort to analyze, improve, and effectively use virtual devices for mobile app testing at scale, across thousands of different mobile systems and device models. Our goal is to (1) quantitatively understand the fidelity of virtual devices and its impact on test results, and (2) explore how to better utilize virtual device farms in today's industrial mobile app testing infrastructures to drastically improve the efficiency and accessibility of large-scale mobile app testing.

Understanding virtual devices for testing. We analyze the real-world test results of of Douyin and nine other global-scale mobile apps developed by Douyin's partners from Jan. 1 to Mar. 31 in 2022. The testing takes a total of 618 hours on 103 versions of the ten apps. For each app version, we run automated tests on both the physical and the virtual device farms. We then perform comparative analysis on the test results from the virtual and the physical devices to measure fidelity and understand discrepancies.

Our study shows that with sensible design, implementation, and configuration, virtual device farms can achieve high-fidelity mobile app testing for diverse device models. Specifically, the vast majority (92.4%) of test failures (caused by native crash, Java/Kotlin exception, app-not-responding error, and so forth) that occurred on physical devices can be reproduced on virtual devices, with only 1.8% false positives (test failures that only manifest on virtual devices). We analyze the root causes of discrepant test results in depth, including both false negatives (test failures that cannot be reproduced on virtual devices) and false positives.

Counterintuitively, the device emulator's lack of support for vendor-specific hardware is not a major root cause of discrepancies; on the contrary, defective drivers of common hardware contribute to 27.6% of the discrepancies. Note that Android HAL does define standard hardware types [3], but provides no abstractions for vendor-specific hardware. Thus, mobile apps rarely access vendor-specific hardware, except for a small number of vendor apps. However, bugs in common hardware drivers (e.g., for MediaTek SoCs) caused many failures on physical devices, but not on virtual devices. The reason is that virtual devices do not use those drivers which depend on proprietary and undocumented hardware specifications such as register functions and MMIO behaviors.

Vendor-specific Android framework customizations incur few discrepancies either; however, vendor-specific system services are the main root causes of discrepancies (46.2% false negatives and 87.8% false positives). The compatibility of Android framework customizations is largely attributed to specifications enforced by Android CTS (Compatibility Test Suite) and VTS (Vendor Test Suite). However, CTS and VTS do not check interfaces between stakeholders. As a common case, add-on system services (typically for graphics acceleration and enhancement) often break specifications of other stakeholders, leading to unexpected test results.

Moreover, certain types of discrepancies are specific to regional mobile app ecosystems. For example, due to a lack of well-regulated app stores like Google Play, mobile users of certain regions are more prone to malicious apps. In reaction, many phone vendors in those regions deploy very aggressive defense mechanisms to limit app behavior; however, such mechanisms cause side effects on regular apps, such as resource leaks and data corruption. Mobile app testing on related devices manifests up to 1,025× more frequent occurrences of certain failures than on other devices.

Improving virtual device fidelity. Our analysis drove a series of efforts that significantly improved the fidelity of virtual devices for app testing. Specifically, we have been focusing on addressing misalignments among stakeholders via active outreach and communication. We started by supporting vendor-specific performance optimization mechanisms and malicious app defenses on the corresponding virtual devices. For defective mechanisms or implementations that violate Android specifications, we go beyond device emulation and actively contribute bug fixes to the vendors.

A key challenge is to develop and validate bug fixes without source code, as vendor-specific components are often proprietary. We address the challenge by implementing a *dynamic binary patching* technique which enables us to effectively prototype bug fixes and validate them *before* reporting to the vendors. The binary patching is based on in-memory instruction rewriting and binary trampoline injection, with in-situ app execution contexts preserved upon failures. The technique and practice can serve as a foundation for collaborations between untrusted stakeholders.

As a result, 63% of our reported issues have been confirmed and our suggested fixes have been merged to the code bases of relevant stakeholders. Hence, the recall of mobile app testing on virtual devices increases from 92.4% to 94.7%, and the precision grows from 98.2% to 99.1%.

Continuous app testing using virtual device farms. Our efforts reshaped the mobile app testing infrastructure of Douyin. Mobile app testing no longer relies entirely on the physical device farms before app releases, which is not only costly but also hard to afford continuous testing.

Based on the virtual device farm, we developed a *continuous* mobile app testing pipeline to enable continuous integration and deployment (CI/CD) [58, 64, 69]. Figure 1 shows our

continuous app testing pipeline, compared with the early practice. The testing pipeline continuously tests every code or configuration change on virtual devices first, which can capture the vast majority of logic and functional bugs and provide prompt feedback to developers. We only use the physical device farm to test aggregated code/configuration changes to capture hardware-specific bugs upon app releases.

The continuous testing pipeline accelerates the end-to-end app development workflow by around 40%. Moreover, it reduced the test time on the physical device farm by 4×, leading to a reduction of operation cost by more than 3×.

Virtual devices as a service. We recently started to make our virtual devices an accessible service upon request, targeting app developers who cannot afford numerous physical devices. Preliminary feedback indicates that compared to developers' current testing practices using a small number of physical devices, our service helps detect 3–10× more bugs. Also, the feedback shows that our major conclusions can generalize to a broader range of apps.

Contribution. This paper makes five main contributions:

- A large-scale study of virtual devices for mobile app testing, with in-depth analysis on testing fidelity, its impacts on test results, and root causes of discrepancies.
- The design, implementation, and configuration of a high-fidelity virtual device farm for mobile app testing.
- Techniques and practices for improving virtual device fidelity for app testing based on dynamic binary patching.
- Experiences and practices of effectively using virtual device farms for continuous testing of mobile apps at scale.
- Preliminary experiences of virtual devices as a service for app developers who cannot afford physical device farms.

Research artifacts (including code and data) are available at https://Android-Emulation-Testing.github.io.

2 TESTING INFRASTRUCTURE

This section presents the design, implementation, and configuration of both the physical and virtual device farms we build and operate for Douyin. To our knowledge, we are the first to share detailed information of large-scale testing infrastructures for mobile apps with billions of users.

2.1 Physical Device Farm

It is known that Android apps behave differently on different devices due to hardware and software customizations [40, 42, 53, 73]. Without comprehensive testing, apps are subject to logic and functional bugs that disrupt user experiences.

To capture bugs of Douyin, its development team has made extensive efforts to build a massive physical device farm (distributed across the US and China), dedicated to testing the

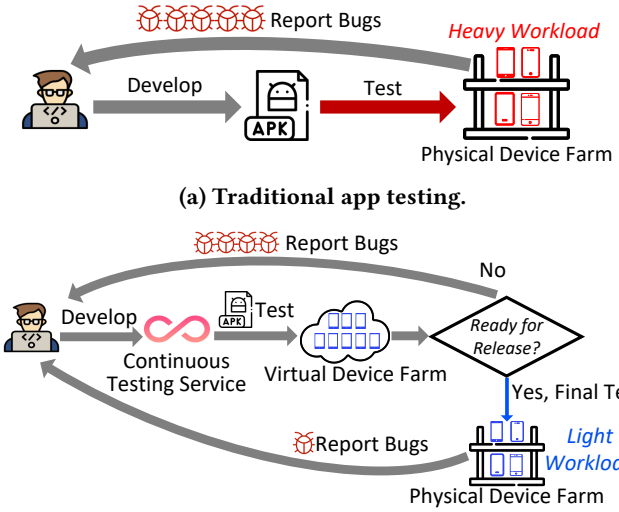

Figure 1: Mobile app testing workflow: (a) traditional app testing on physical device farms only, and (b) continuous app testing using virtual device farms.

app before its public version releases. The device farm deploys devices of the top-5000 Android phone models owned by the users of Douyin in early 2019 (one device is purchased per model). At the beginning of each year, new devices are purchased based on the statistics of model popularity and some obsolete devices are retired (obsolete devices are those owned by fewer than 1,000 users of Douyin).

As of Jan. 2022, the physical device farm operates a total of 5,918 devices. These devices run Android OSes whose versions range from 5.0 to 12; only 1.7% of them run vanilla Android. To test the app under different networks, 15% devices are equipped with SIM cards for accessing cellular networks (including 5G); the others are connected to the Internet via 600 WiFi access points (APs); each AP offers 1 Gbps of access bandwidth, supporting ~10 devices.

Operating the large fleet of physical devices is laborious. A dedicated team of 15 engineers performs daily device maintenance and system monitoring. For ease of management, all the devices in one location are connected to a centralized management platform via USB ports, which provides power supply and the ability of distributing adb shell commands.

In early 2019, Douyin spent more than 1M US dollars in building the initial device farm. The operation cost has gradually grown beyond the construction of physical infrastructures. In particular, since the physical devices run tests continuously, their average lifespan is merely 10 months before the occurrence of serious hardware failures such as battery swelling and screen wear-out. In total, it costs over 0.6M dollars per year for device upgrades and replacements, not including the yearly salaries of the maintenance team,

carrier plan expenses of cellular-capable devices, maintenance cost of WiFi networks, power usage, etc.

Cloud-based mobile app testing services such as AWS Device Farm [1] and Google Firebase Test Lab [31] are not a solution for Douyin. First, the number of available device models is insufficient. AWS Device Farm currently provides the most diverse device models, which only includes 136 models, while Douyin aims at over 5,000 models. Second, the device unit price for continuous testing can be more than 200 dollars per month, exceeding the physical farm's device unit price—100 dollars per month. Third, they pose considerable restrictions on app testing. Take AWS Device Farm as an example, the test app size needs to be less than 4 GB, the maximum number of parallel-running devices is 5, and the maximum test time is 2.5 hours, which are insufficient for conducting large-scale testing. Finally, it is hard to customize their testing mechanisms.

2.2 Virtual Device Farm

We started exploring the promise of testing mobile apps like Douyin on virtual devices in May 2021. We build a virtual device farm as a digital twin of the physical farm, also deployed across the US and China. In the virtual device farm, we run 5,918 virtual devices based on Android emulators on 395 ARM commodity servers. Each virtual device corresponds to a physical one. For each server, we use the configuration of a 64-core ARM v8.2 CPU @2.6 GHz, 128 GB of DRAM, 1 TB of NVMe SSD storage, and 1 Gbps of NIC bandwidth. We decide to deploy 395 servers based on load testing that verifies that 395 servers have sufficient resources to support 5,918 virtual devices concurrently.

Using ARM servers for virtual device farms has considerable advantages over x86 servers. First, an ARM server is ~23% cheaper than an x86 server for the same number of CPU cores and memory/storage capacity [82]. Second, the natural affinity between ARM servers and Android phones (all of which employ ARM CPUs) avoids the need for dynamic binary translation (DBT) from the ARM instructions of Android apps to the x86 instructions of server CPUs [32, 36]. In fact, few mobile apps have x86-native libraries.

Each virtual device is configured to have the same number of CPU cores, the same memory and storage sizes, and the same display resolution as its physical counterpart. We pin each virtual core to a physical core to resemble CPU locality. Although the underlying network of the virtual device is wired Ethernet, we configure its virtual network to be the same as that of the physical device, because different network types use different framework- and HAL-layer components in Android, e.g., `WiFiManager` [7] and `DcTracker` [6] are used by WiFi and cellular networks respectively.

For the software stack, the host OS is Ubuntu Server for ARM 20.04 LTS, with KVM/ARM [14] enabled to accelerate the virtualized CPU and memory. For the emulator and the guest OS, we choose the Cuttlefish variant of Google Android Emulator (or Cuttlefish GAE in short) [5] to run AOSP. Unlike the classic GAE that hooks many framework-level mechanisms into the guest OS to emulate mobile hardware (e.g., GPU and sensors), Cuttlefish GAE removes such guest modifications to ensure consistent framework-level behavior with physical devices running vanilla Android. It then multiplexes host-side hardware (e.g., storage, network, and GPU) via `virtio` [57] for high-throughput I/O; for the other hardware components (e.g., camera, microphone, and various sensors), it provides pure software emulation at the HAL.

To match vendors' app-related customizations, we also install the vendor-specific app service platforms, i.e., a collection of installable vendor apps and SDKs such as GMS (Google Mobile Services) and HMS (Huawei Mobile Services), on each virtual device based on the corresponding physical device. We do not port other system-level components (including system services and hardware drivers) since they have intricate dependencies on proprietary software and hardware that are extremely hard to emulate [54]. For example, Xiaomi's system services need access to their cloud services, which we cannot provide. Qualcomm drivers rely on the MSM Android Kernel Subsystems, and follow their own hardware specifications to implement register I/O and MMIO, which are usually proprietary and hard to emulate.

Building the virtual device farm costs ~0.2M US dollars, the vast majority of which comes from renting the ARM servers. The cost is 5× cheaper than that of building the physical device farm. Also, servers are more resilient to failures of hardware components, because they are manufactured in a modularized manner—each component can be replaced separately, as opposed to the coupled design of many mobile devices. On average, a server has a lifespan of 5+ years, significantly longer than the lifespan of mobile devices, resulting in considerable reduction of amortized cost.

Other than the construction cost, virtual device farm, as a software solution, significantly reduces operation cost. Cuttlefish GAE's support for multi-tenancy and WebRTC-based data streaming also improves the scalability and manageability of the solution. Compared with the 15-engineer team for the physical device farm, the maintenance team for the virtual device farm only consists of four engineers.

3 TESTING AND DEBUGGING TOOLS

This section introduces our tools for automated app testing and root-cause analysis of large-scale test results.

3.1 Test Case Generation

We run automated mobile app tests on each pair of physical and virtual devices. We employ an enhanced version of the Monkey UI/Application exerciser to generate input events and monitor failure occurrences during a test run. We enhance the Monkey tool based on a state-of-the-art model-based UI test technique [50] to generate effective streams of UI events (including touch and swipe). Compared to the vanilla Monkey tool which randomly generates events and often falls into invalid or cyclic actions [72], our enhanced tool increases `Activity` coverage by 2× to 3× in practice. By default, we run the tests for six hours on a pair of physical and virtual devices; running the tests longer can hardly increase `Activity` coverage. The tests terminate as long as a failure occurs on either the physical or the virtual device.

3.2 Collecting Failure Information

During the test, we monitor app failures, such as Java/Kotlin exceptions in Android Runtime, native crashes triggered by fatal signals like `SIGSEGV`, and app-not-responding (ANR) events (i.e., the app does not respond to user input or system broadcast for five seconds). Upon such events, it is vital to collect failure information for in-depth postmortem analysis. The data collection needs to meet the following requirements: (1) Failure data should be collected comprehensively to help postmortem debugging, (2) Data collection should be light-weight and storage-efficient to avoid overloading test devices or servers, and (3) Errors during the data collection should be handled reliably.

Upon a failure, we collect three common information sources: (1) Android `logcat` that contains apps' log outputs, (2) system resources like opened file descriptors, and (3) execution context like the call stacks and register values.

We also collect (4) in-situ memory dump which contains important debugging information, particularly useful for failure analysis of proprietary components. However, conventional `coredump` of process memory contains hundreds of MBs of data for a single failure event (after compression). To reduce storage overhead, we safely remove data sections that are irrelevant to the failure or can be recovered offline, e.g., JVM memory when a failure is rooted in native code, static resources (like fonts) that are recoverable, inaccessible private memory segments, and unused stack data. The pruning achieves 15× to 103× reduction of storage overhead, resulting in <3 MB memory image after `gzip` compression.

When a failure occurs, four dedicated native processes are launched to capture the four types of in-situ information described above. If one process fails, the others are not affected. We also customize the signal handler of the four processes for self recovery and logging.

3.3 Root Cause Analysis

For failures rooted in app code or standard Android components, debugging is done by inspecting the source code of the app and/or the Android system based on the captured Java/Kotlin and native call stacks. Unfortunately, discrepancy-related failures often stem from proprietary components, where source code is unavailable. For proprietary Java/Kotlin components, we use decompilation tools to inspect the code.

However, proprietary native binaries are typically not compiled with debugging information—the call stacks alone (with vendor-specific function symbols and call relations) are insufficient for root cause analysis. To debug such issues, app developers often need to inspect instructions and memory/register data involved during a proprietary vendor-specific function call. We develop a failure analysis technique based on *binary taint backtracing* [13, 75] to reconstruct the instruction and data flows related to a failure event.

Specifically, we extract binary instructions related to "taint objects" (i.e., illegal memory addresses or registers that contain such addresses), by reversely tracing the instructions that are used to pass, execute, or modify taint objects at run time. Our method works in the following three steps.

Taint object extraction. To extract the taint object(s) for a failure event, we first identify the faulty instruction (denoted as I_f) that directly results in the failure in the call stacks, and then analyze the operation (i.e., read, write, or execute) of I_f to acquire taint object(s) from operands of I_f. Specifically, if I_f is a read instruction, we mark its source operands as the taint objects. Otherwise (I_f is a write or execute instruction), its destination operands are marked as the taint objects.

Binary backtracing. Starting from the address of I_f in the offending binary, we reversely examine instructions to locate where the taint object(s) originate from, to figure out the more closely related taint object(s). Typically, this is achieved by examining whether the destination operand of a previous instruction (denoted as I_p) is a taint object, meaning that I_p has modified the taint object.

Once the examination reaches the boundary between functions, i.e., function call instructions, we jump to the caller instruction identified from the call stacks. The above operations are performed until all the proprietary functions in the call stacks have been examined; meanwhile, the taint objects (together with their historic values if possible) and the corresponding instructions are recorded.

Enriching call stacks. We enrich incomplete call stacks by incorporating the above information of taint objects and associated instructions into the corresponding proprietary function call. So, debugging can focus on failure-related instruction and data flows.

Table 1: The mobile apps and their corresponding "sub-apps" used in the paper.

ID	App	Functionality	# Sub Apps	# Users	# Releases	Test Time
1	Douyin	Video streaming, shopping, social media, map, education, etc.	31	1.5B	12	72 hours
2	Douyin Lite	Video streaming, communication, travel, photography, etc.	41	210M	12	72 hours
3	Xigua Video	Video streaming, payment, shopping, 3D gaming, etc.	24	180M	12	72 hours
4	Toutiao	News feed, shopping, web browsing, 3D gaming, etc.	33	530M	12	72 hours
5	Toutiao Lite	News feed, video streaming, security checking, payment, etc.	29	130M	12	72 hours
6	Lark	Communication, email, video conference, cloud storage, etc.	1	9.4M	5	30 hours
7	Helo	Social media, video streaming, communication, etc.	1	50M	12	72 hours
8	Fizzo Novel	E-book, shopping, 3D gaming, social media, etc.	14	10M	5	30 hours
9	Xingfu Li	E-commerce, video streaming, finance, communication, etc.	5	7.5M	12	72 hours
10	Resso Music	Music streaming, communication, social media, etc.	1	40M	9	54 hours

Table 2: The top-10 most frequent types of failures on all physical and virtual devices. App-ID is the ID in Table 1.

No.	Percent.	App-ID	Location	Exception/Signal	Root Cause
1	29.5%	1–4, 10	App	NullPointerException (Java)	Bad resource handling during activity lifecycle shifts
2	15.3%	1, 2, 4, 9	Third-party	NullPointerException (Java)	Defects in OPPO market SDK
3	7.0%	9	App	NullPointerException (Java)	Null object reference in app module
4	6.9%	4	App	NullPointerException (Java)	Attempt to cast null reference to non-null Kotlin class
5	5.7%	5	App	ClassNotFoundException (Java)	Failed resolution of app Java classes
6	5.1%	2	App	NullPointerException (Java)	Method parameter specified as non-null is null
7	4.7%	3	App	ClassCastException (Java)	Incompatible Java class casts
8	3.0%	All	App	OutOfMemoryError (Java)	Out of memory when allocating Bitmap objects
9	2.0%	8	App	NullPointerException (Java)	Method invocation on null app objects
10	1.4%	All	App	OutOfMemoryError (Java)	Out of memory when creating new threads

4 OVERALL STUDY RESULTS

We collect the test results of Douyin between Jan. 1 and Mar. 31 in 2022. For each version release of Douyin, we run automated tests (§3.1) with comparative analysis on both the physical and virtual device farms. To verify the generality of our results, we also run tests on nine other global-scale popular mobile apps upon their app releases, as listed in Table 1. The nine other apps are from commercial partners of Douyin and thus we can acquire their source code. Note that all the ten apps are independently developed. Some of them adopt the Android plugin practice [8] to modularize functionalities in "sub-apps", each coming with a standalone APK. In total, we test 180 sub-apps that cover various types.

During the three-month testing, a total of 805,423 failure events have been captured. 390,286 failure events occur on physical devices, while the other 415,137 occur on virtual devices. Among them, 2.45% are hardware-specific failures related to both the hardware devices and their system components, including hardware system services (e.g., AudioFlinger), hardware libraries (e.g., GPU render libraries), HAL libraries, and kernel drivers. The hardware-specific failures involve almost all the common mobile hardware, including the camera, GPS, Bluetooth, NFC, USB, WiFi, cellular, and GPU. The other failures are app-level failures (92.64%) and software system failures (4.91%).

Note that all the study and analysis were conducted under a well-established IRB. No personally identifiable information was collected throughout the study.

Root causes. Our root cause analysis (§3.3) reveals 873 failure types with different root causes, among which 762 occur on physical devices and 485 occur on virtual devices, with 374 types in common. Among the 374 common failure types, 99% of them have similar occurrence frequencies on both physical and virtual devices. On the other hand, the remaining 1% occur much more frequently (up to 1025×) on physical devices than on virtual devices. We will discuss the frequency discrepancy in §5.4.

Note that the top-10 most frequent failure types are the same on physical and virtual devices. Table 2 lists those types which account for 84.2% and 77.1% of the total failure events on physical and virtual devices, respectively.

We look into the failure frequency from the perspectives of both the Android system and apps. First, as Android evolves from 5.0 to 12, many system-level changes are made, which could change failure characteristics. Figure 2 shows the average failure frequency per device per test run for each Android version. We can see a decreasing trend of failure frequency on both physical and virtual devices. The reduction of failure events can be attributed to many new features and bug fixes of Android. For example, we find that the deadlock issue in the background garbage collection of ART JVM in Android

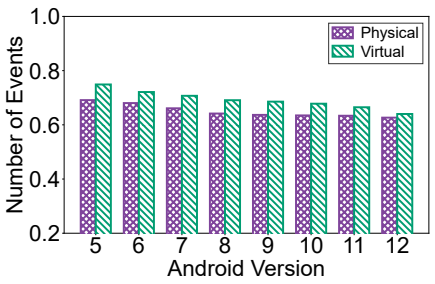

Figure 2: Average failure occurrence frequency per device per test round for different Android versions.

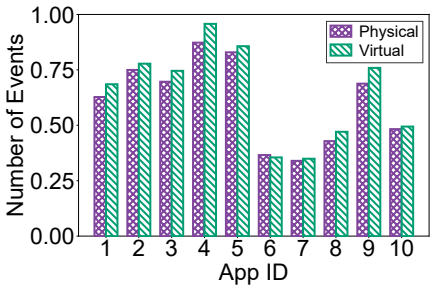

Figure 3: Average failure occurrence frequency per device per test round for each studied app.

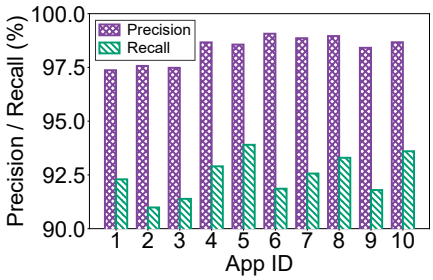

Figure 4: Precision/recall of the test results on virtual devices, relative to those on physical devices.

versions 5 and 6 [26] and the incorrect handling of page faults in the FUSE filesystem in Android versions 5–7 [28] cause a number of (2.1%) failure events in early Android versions. These issues have been fixed since Android v8.

While these bugs are only resolved in new Android versions for vanilla Android, we also observe that some phone vendors (e.g., Xiaomi) actively backport the bug fixes of new Android versions to devices that run old Android versions if possible. The backport practice partially explains why in Figure 2 the average failure frequencies on physical devices are lower than on virtual devices across Android versions—physical devices with old Android versions benefit from the backporting of vendors while virtual devices running vanilla Android do not. Lastly, Figure 3 shows the average failure frequency per device per test run for each app. We can see that failure frequencies vary significantly across apps. For example, Lark and Helo fail 50% less frequently than the other apps. The reason is that the two apps do not involve complex video processing and web browsing modules.

5 FIDELITY AND DISCREPANCIES

In this section, we present an in-depth analysis of the fidelity of virtual devices for mobile app testing. Our goals are two-fold. First, we measure the fidelity of our virtual device farm (§2.2) to assess the feasibility and promises of using virtual devices for app testing against massive, diverse physical device models in industrial settings. Second, we look into discrepant test results between the virtual and physical devices and understand their root causes. In this way, we can further improve fidelity by developing effective solutions to address the manifested discrepancies (see §6).

5.1 Overall Fidelity

With the virtual device farm described in §2.2, we observe very high fidelity of using virtual devices for mobile app testing. Among the total of 390,286 failure events that are captured on the physical devices (see §4), the vast majority (92.4%) of them are also manifested and captured on the corresponding virtual devices. Meanwhile, the false positives are low—only a small percentage (1.8%) of failure events that occurred on virtual devices are not manifested on the physical devices. So, taking the test results from the physical device farm as the ground truth, virtual devices have as high as 98.2% precision and 92.4% recall, despite hardware and OS differences. Figure 4 shows the precision and recall per app. The fidelity characteristics are similar across the apps (note that the y-axes all start from 90%).

Table 3: The 27 phone vendors and the number of device models per vendor (# Models) in our study. The country/region (Region) is where the vendor obtains the most sales revenue. C/VTS shows whether a device model is CTS/VTS-compliant.

Vendor	# Models	Region	C/VTS	Precision	Recall
Samsung	1863	US	Y	98.6%	93.2%
Xiaomi	959	China	Y	98.5%	94.3%
Huawei	901	China	Y	98.5%	91.5%
Vivo	540	China	Y	98.5%	94.7%
OPPO	291	China	Y	98.4%	90.0%
Honor	198	China	Y	98.1%	91.6%
Redmi	193	China	Y	98.3%	94.9%
Meizu	179	China	Y	95.6%	72.4%
LG	119	Korea	Y	96.8%	94.7%
Docomo	84	Japan	Y	97.1%	95.7%
Motorola	82	US	Y	96.9%	93.8%
Infinix	77	US	Y	97.4%	95.7%
Realme	66	China	Y	97.4%	94.1%
Tecno	61	S. Africa	Y	97.7%	96.4%
Google	54	US	Y	97.3%	89.6%
Lenovo	44	China	Y	96.2%	92.0%
Sony	39	Europe	Y	97.9%	89.5%
Oneplus	38	India	Y	96.1%	90.7%
Smartisan	29	China	**N**	**88.7%**	**83.0%**
Vsmart	28	Vietnam	Y	96.6%	89.2%
Asus	17	Europe	Y	96.2%	88.2%
ZTE	17	China	Y	97.6%	87.9%
Alcatel	14	US	Y	97.8%	89.9%
Blackshark	11	China	Y	97.6%	91.8%
Nubia	6	China	Y	95.8%	88.2%
Alldocube	5	China	Y	95.9%	92.6%
Blackview	3	US	Y	95.6%	89.1%

Table 4: The top-5 most frequent types of false negatives.

No.	Percent.	App-ID	Location	Exception/Signal	Root Cause
1	14.9%	All	AOSP	SIGABRT (Native)	Integer overflow during implicit conversions
2	9.1%	All	Meizu	SIGSEGV (Native)	Improper null-terminations of C/C++ strings in vendor modules
3	8.9%	All	MediaTek	SIGSEGV (Native)	Errors in MediaTek's GPU drivers
4	5.2%	All	Samsung	SIGSEGV (Native)	Array index out of bounds in vendor modules
5	4.1%	2, 3, 5–10	OPPO	SecurityException (Java)	Permission denial when querying autostart permission

Our results refute a common belief that vendor-specific hardware could largely impair the fidelity of app testing when virtual devices are used [20, 47, 48, 53, 59, 73]. It is true that vendor-specific hardware can hardly be emulated precisely. However, as shown by our results, vendor-specific hardware does not result in major discrepancies. The main reason is that Android HAL specifies the interface of standard hardware types [3] and mobile apps can only access device hardware by passing the hardware type to the HAL interface. Thus, except for a small number of vendor apps, it is non-trivial for the vast majority of apps to access vendor-specific hardware beyond Android HAL. In fact, as per our knowledge, none of the studied apps (Table 1) accessed any vendor-specific hardware and we believe that they well represent today's market apps in different app stores.

On the other hand, vendor-specific customizations on vanilla Android system components also rarely incur discrepancies. We give the credit to the active development of Android Compatibility Test Suite (CTS) and Vendor Test Suite (VTS), which are unit-level compatibility tests to ensure functional consistency of standard Android components after vendor customizations. Only CTS/VTS-compliant phone models are entitled to the "Powered by Android" trademark [30]. In Table 3, Smartisan, which is not CTS/VTS compliant, has the lowest precision and recall. For the 26 phone vendors who pass CTS/VTS, the precisions and recalls stay above 95% and 87% respectively except for Meizu (we discuss the Meizu case in §5.2). In addition, with CTS/VTS being more mature and complete in newer Android versions, virtual device fidelity is also increasing, as shown in Figure 5 (in the 11th page).

5.2 False Negatives

A false negative refers to a test failure that only manifested on a physical device. Table 4 lists the root causes of the five most frequently occurring false negatives during the tests. As shown, false negatives are mostly rooted in the native code of system components. The implication is that the discrepancies could have system-level impacts on any apps running on the virtual devices instead of individual apps. Hence, understanding them is necessary.

Surprisingly, the most frequent false negatives (accounting for 14.9%) are caused by an AOSP bug. The bug is triggered when Android initiates vendor-specific system services [29]. During the initiation of a vendor-specific system service, Android uses the system boot time to calculate a timeout threshold for the service. Regretfully, in this process, Android mistakenly assigned the 64-bit system boot time value to a 32-bit program variable without explicit type casting, resulting in an integer overflow. As a consequence, the runtime sanitizer would throw a `SIGABRT` signal and crash the foreground app (e.g., Douyin). Note that the integer overflow is only manifested when the Android device has been booted for longer than 2^{31}-1 milliseconds (≈25 days).

For most system services, the bug is not triggered because those services are initiated during system booting so the time is not long enough to trigger the integer overflow. However, certain vendor-specific services are initiated *on demand*, and thus trigger the bug. The bug affects all Android systems with versions earlier than v12. The triggering condition (a long time after booting) is hard to be caught by CTS/VTS. It is not captured on virtual devices because their AOSP system does not have on-demand vendor-specific system services.

The remaining false negatives are mostly caused by bugs in other vendor-specific system services or drivers. In particular, we find that the root causes tend to reside in non-standard functionalities in vendor-specific services, and problematic hardware drivers. For example, the second most frequent root cause (responsible for 9.1%) is a buggy null-termination of C/C++ strings in the super-resolution module of Meizu's systems, which is used for enhancing the resolution of videos. Similarly, a bug in the graphics drivers of several old MediaTek SoCs incurs 8.9% of the false negatives. The false negatives are not captured by virtual devices due to the inherent difficulties of running proprietary system services and hardware drivers in virtual devices as discussed in §2.2.

5.3 False Positives

A false positive refers to a test failure that only manifested on a virtual device. Table 5 shows the five most frequent types of false positives. We find that 87.8% of them root in the graphics subsystem of virtual devices. As a result, apps with video streaming features (e.g., Douyin, Douyin Lite, and Xigua Video) have low precisions, as shown in Figure 4.

The root causes of the graphics related failures lie in the subtle differences between the graphic resources used by Android and the emulators. Typically, to render and display a

Table 5: The top-5 most frequent types of false positives.

No.	Percent.	App-ID	Location	Exception/Signal	Root Cause
1	53.4%	All	AOSP, Emulator	SIGSEGV (Native)	Graphics resource format inconsistency
2	7.3%	All	Emulator	SIGABRT (Native)	Missing graphics buffer allocator
3	6.8%	All	AOSP, Emulator	SIGSEGV (Native)	Graphics buffer overrun (due to graphics format inconsistency)
4	5.1%	All	Third-party	SIGSEGV (Native)	Null pointer dereference in third-party media player
5	4.7%	1–5, 7–10	Emulator	SIGSEGV (Native)	Rendering issues in the graphics driver used by emulators

graphic resource like an image or a video frame, the graphics subsystem of Android would first populate a graphics buffer with the resource's raw bytes following the resource format (e.g., RGBA and YUV). During the rendering process of the resource, the graphics subsystem then reads and decodes the buffered content based on the resource format. This simplifies graphics resource sharing, because the producer and consumer of a resource only need to reach a consensus on the resource format for data sharing, avoiding sending complex metadata of the format. In this case, the correctness of the design relies on consistent interpretation of the resource format between the producer and the consumer. For physical devices, it is easy to ensure consistency, as the producer and the consumer belong to the same Android system.

On the other hand, the design caused discrepancies in the virtual devices. In order to accelerate graphics rendering, the virtual devices send graphic resources produced by Android to the host so that the host GPU can be multiplexed for graphics rendering. However, the host GPU driver's definitions of certain resource formats (e.g., the YV12 format for video frames) are different from those of Android in a number of subtle and undefined manners. For instance, the inconsistent data alignments (which are undefined in standard documents [2]) led to illegal memory access when the host attempts to read the graphics buffer based on its own interpretation of the format.

The other false positives include database errors, system service outages, and problematic 32-bit binary support. The first two are mostly caused by data corruption on the virtual disks due to crashes—virtual devices do not have power failure protection like physical devices. The last one occurs because our servers' Kunpeng 920 SoC only supports the AArch64 instruction set. Therefore, 32-bit binaries are executed via a binary compatibility layer offered by the server manufacturer, which is not sufficiently tested and sometimes incurs issues like illegal memory alignment. We expect that these false positives can be significantly reduced with ARM's removal of 32-bit support in its latest ARMv9 architecture, which forces all Android apps to provide 64-bit support.

5.4 Frequency of Discrepancies

As mentioned in §4, 1% of the failures occur more frequently (up to 1025×) on physical devices than on virtual devices. These frequent discrepancies may well conceal the high-priority issues of apps in production. Thus, it is important to understand the frequency characteristics of discrepancies.

An initial investigation reveals that the frequent discrepancies mostly occurred on the Android phones manufactured by six Chinese vendors, i.e., Xiaomi, Huawei, Vivo, OPPO, Honor, and Redmi. Delving deep into the regional phenomenon, we discover that these frequent discrepancies are rooted in these vendors' aggressive strategy for suppressing background activities of the apps under test [11]. Specifically, unlike AOSP which would normally notify a target app before killing it for a graceful termination, the Android systems customized by these vendors aggressively throttle a background app by simultaneously killing all the processes in the app's process group with no grace period. This can easily trigger resource leaks or data corruption, thus leading to more frequent failures on the physical devices than on virtual devices running AOSP.

We reached out to the concerned vendors and our close communications reveal that their aggressive strategies are purposed for inhibiting the background keep-alive behavior of malicious apps. One typical example is the dual-process co-awakening behavior—since Android independently manages the lifecycle of each process of an app, the app can lock a file in one of its process (P1), and meanwhile monitors the status of P1 by attempting to lock the same file in the other process (P2). If P1 is killed, P2 will acquire the lock and immediately revive P1. Similarly, P1 protects P2 from being killed.

According to the six vendors, malicious behaviors as such are common in many Chinese apps, and have caused a variety of undesirable user experiences like fast battery drain and poor UI responsiveness. As an effective countermeasure, the aggressive background inhibition strategy is deployed, with the cost of more frequent app failures. This dilemma largely stems from the unavailability of GMS and Google Play in the corresponding regions, where phone users often download unverified apps from third-party app markets and websites.

6 IMPROVING EMULATION FIDELITY

To improve the virtual device fidelity for mobile app testing, we have built on our insights to address various discrepancies, including those across boundaries of different components and interests of stakeholders. We did not limit

ourselves to pure technical solutions, but also coordinate stakeholders in the mobile industry and convince them to incorporate our solutions. The efforts are rewarding—we have managed to eliminate most of the discrepancies to date.

6.1 Techniques and Practices

Learning from the root cause analysis (§5), we have been taking actions to address discrepancies by improving (1) the Android emulator (§6.1.1), (2) AOSP on virtual devices (§6.1.2), and (3) vendor-specific customized Android systems on physical devices from different phone vendors (§6.1.3).

6.1.1 **Graphics resource format extension.** As discussed in §5.3, the vast majority of false positives stem from the subtle differences between the graphic resources used by Android and emulators. To address this issue, we extend the resource format within the host-side graphics library by adding Android-specific formats based on the resource definitions of Android. Hence, the host-side graphics library can correctly recognize and decode the graphics buffers populated by Android. We have also reported the issues and the fixes to the development team of GAE, who have confirmed and fixed the issue in the latest kernel branch [27].

6.1.2 **Background management strategy support.** In §5.4 we explained that the frequency discrepancies mostly stem from the aggressive background app inhibition strategies of certain vendors in their customized Android systems. After extensive discussions with the vendors, we understand that these vendors are reluctant to change their strategies in favor of their user interests and experiences. Thus, to practically reduce the frequency discrepancies, we selectively port the background management strategy to the corresponding virtual devices. We do so by customizing `ActivityManagerService` (the system service for managing app activities in Android) in the AOSP systems of the virtual devices. Specifically, to mimic vendors' strategies, when killing a background app, we first collect all the process information within the app's process group and then immediately kill all the processes.

6.1.3 **Dynamic binary patching.** To reduce false negatives (§5.2), we need a new approach for quickly prototyping and deploying our proposed fixes to the defects and bugs in specific vendors' proprietary system components. According to our experience, vendors are often not well motivated to fix seemingly app-specific issues, unless we can provide strong evidence on the root causes and offending components. To this end, we need to address the major challenges that proprietary vendor-specific components are immutable and reside mostly in the read-only `system` partition of Android. Thus, we cannot simply patch a vendor-specific component by directly replacing it without root privileges.

Our key insight for addressing this challenge is that more than 80% of the offending components in our study were in fact system or hardware libraries whose binaries are loaded into an app's address space during the app's run time. This is not a coincidence, but stems from the fact that the problematic components usually modify an app's behavior and outputs, and thus need to live in the app's memory space; otherwise, the failures in the components are most likely isolated from the app (with address space isolation). With this insight, we develop a *dynamic binary patching* technique to realize in-memory manipulation of offending components.

Upon the startup of an app, we locate the base address of the offending component's binary that we wish to patch. Combining the base address and binary taint analysis (see §3.3), we can calculate the address of the offending instructions we wish to patch in the binary's code segment. Finally, by resetting the write privilege of the corresponding memory region via the `mprotect` system call (no root privileges required), we can rewrite the original instructions or insert trampolines into the binary, so that the buggy implementations can be overwritten or bypassed. In this way, we can quickly validate the effectiveness of our proposed fixes, so as to enable productive collaborations with the vendors.

Leveraging this approach, we were able to provide effective patches for 73% false negatives. For the remaining cases, we find that they are associated with special vendor services/executables that cannot be modified by our technique. We then report the root causes and solutions to all the corresponding stakeholders including phone vendors (e.g., Huawei, Honor, and Meizu) and hardware manufacturers (e.g., MediaTek). As of Jul. 1st, 2022, we have received the stakeholders' confirmations for all the reported issues, and 63% of our suggested fixes have been adopted into their code bases. The other reports are still under review.

6.2 Evaluation

On July 1st, 2022, we have installed the latest patches released by phone vendors and hardware manufacturers (via system upgrades) on our physical devices, and applied all the software fixes to our virtual devices. Using the same testing infrastructure and methodology as in §2 and §3, we run the tests for the ten studied apps for another three months from Jul. 1st to Sep. 30th in 2022.

Results. The recall on the virtual device farm increased from 92.4% to 94.7%, and the precision increased from 98.2% to 99.1%. As shown in Figure 6, for individual apps, the false positive rate is reduced by 10%–64%, while the false negative rate is reduced by 18%–42%. In particular, for the device models that have received upgrade patches for fixing the issues we reported, their average recall has improved from 92.3% to 97.6%. Also, after applying our enhancements to the graphics

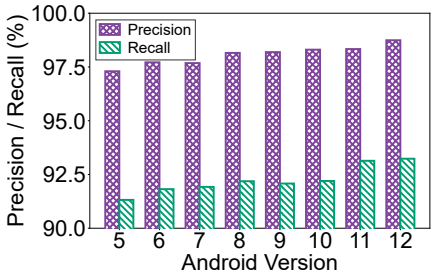

Figure 5: Precision and recall of the test results on virtual devices for each Android version.

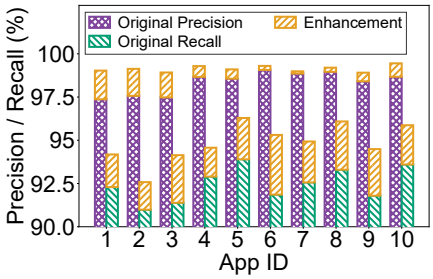

Figure 6: Test precision/recall on our virtual devices for each app, before and after enhancements.

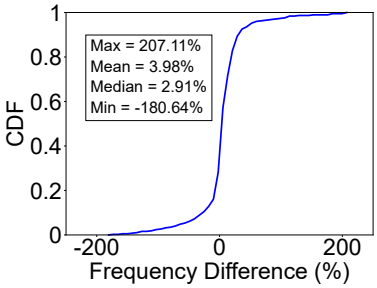

Figure 7: Failure frequency difference between virtual and physical devices after enhancements.

subsystem of virtual devices, the three video streaming apps no longer show considerable precision differences compared to the other apps.

Figure 7 shows the distribution of the frequency difference between physical and virtual devices for each type of test failure occurring during the evaluation. With our support for the background management strategy on related virtual devices, we find that the median frequency difference is now smaller than 3%, as compared to the 70% median difference before our enhancements. Also, the maximum frequency difference is reduced from 1025× to 2×, which in fact stems from a long-tail failure type that accounts for only 0.08% of failure events on physical devices.

7 CONTINUOUS MOBILE APP TESTING

With all the understandings (§5) and efforts (§6) for improving virtual device fidelity, we reshaped the testing practice of Douyin, which was entirely based on physical device farms, to modern continuous testing with virtual device farms.

7.1 Virtual Devices for Continuous Testing

Without the virtual device farm, all the tests of Douyin before its version releases were directly conducted on the physical device farm. The results were high operation cost and reduced device lifetime, making it difficult to apply modern testing practices [58, 64, 69] to enable continuous integration and deployment (CI/CD).

With high-fidelity virtual devices, we developed a continuous mobile app testing pipeline to enable continuous integration and deployment, as illustrated in Figure 1. The idea is to combine the efficiency of virtual devices and the safety of physical devices—virtual devices are used to continuously test code and configuration changes to detect the vast majority of logic and functional bugs, while physical devices are used only upon app releases as the last-level defense to capture hardware-specific issues. We do not intend to remove physical device farms, because (1) we aim at eliminating all issues before release—given the popularity of Douyin, a small number of false negatives could lead to major issues especially for specific vendors, and (2) with the virtual device farm taking most of the workloads, the cost of running tests on physical devices is minimized.

Specifically, after unit testing, code/configuration changes are automatically integrated into executables and distributed to virtual devices for automated UI-driven testing. Bugs are reported to developers when failures are identified. Before the app's public releases, we conduct testing on physical devices to uncover the remaining bugs, which are mostly related to vendor-specific system services or drivers (§5.2). Since our continuous testing has captured most app-level bugs (which account for 93% of all bugs), tests on physical devices are much less frequently interrupted by test failures.

We analyzed the continuous testing pipeline in production with the ten studied apps from Jan. 1st to Feb. 28th in 2023. The new pipeline reduces test interruptions due to test failures on the physical devices by 12×, leading to 4× reduction of test time on the physical device farm. The reduction comes from savings of data collection overhead and test restart time. Moreover, continuous testing enables us to overlap development and testing, accelerating the end-to-end app development workflow by around 40%. Last but not least, the device lifespan is lengthened by 1.5× on average, and maintenance efforts are greatly reduced, including a smaller operation team and less network and power consumption. The result is ~3× reduction in the total operation cost.

7.2 Virtual Devices as a Service

We have recently started to share the virtual device farm as a service (termed VDaaS) with interested app developers, targeting individual or startup developers who do not have sufficient resources to test their apps. To use VDaaS, developers first integrate our SDK into their apps, which provides failure information collection (§3.2) and root cause analysis (§3.3). Then, they submit the app's APK file to our service for testing with a default of six hours. After that, the test

results are returned to the developers. From Jan. 1st to Feb. 28th 2023, 28 apps have been tested on our service.

We collected feedback from the developers, which shows that VDaaS helped detect 3× to 10× more bugs than their existing practices. We analyzed false positives and negatives of VDaaS reported by developers. VDaaS achieved 99.3% precision and 96.2% recall. With our enhancements to virtual devices (§6.1.3), VDaaS was able to uncover many vendor-specific problems previously unknown to the developers.

It is encouraging that most of our findings can be generalized to a broader range of apps. Notably, we still find that vendor-specific hardware is not among the major root causes, while vendor-specific system services and drivers cause the most false negatives. However, with our fixes of graphics resource misalignment (§6.1.1), the false positives are mainly due to data corruptions of virtual disks, for which we are developing protection mechanisms based on reliable virtual disk formats like `qcow2`. For failure frequencies, we no longer observe significant discrepancies due to the support for vendor-specific background management (§6.1.2).

We are considering commercializing VDaaS by making it an accessible testing service. The adoption of VDaaS would in turn enable us to continuously improve virtual device fidelity for a wider variety of apps.

8 DISCUSSION AND FUTURE WORK

Addressing vendor-specific discrepancies. As discussed in §5.2, vendor-specific system services and drivers are still major threats to the testing fidelity of virtual devices. However, precisely emulating the complete array of vendor-specific services and drivers is impractical given the insurmountable engineering efforts. On the other hand, there is still room for reducing their impacts and further improving testing efficiency via cooperation between virtual and physical devices.

One direction is to remote the app's interactions (e.g., function call, system call, and I/O operation) with proprietary components to the corresponding physical devices. This approach will allow us to bypass the need for hardware emulation with real devices. Prior work has demonstrated the feasibility of remoting a driver's I/O operations to a physical device to achieve executing the driver in a virtual device [65, 80]. However, these approaches face several challenges in large-scale production. First, they need to modify the OSes of the physical devices for executing remoted low-level I/O operations. Second, remoting can significantly reduce performance, which could result in timeout failures. Third, the scalability of prior approaches is inherently limited by the dependent physical devices. To realize the idea in practice, we would need to explore feasibility of leveraging high-level APIs, designing more efficient remoting techniques [21, 22], and effective hardware multiplexing.

Towards cross-component compatibility tests. As discussed in §5.1–§5.3, while vendors' customizations to standard system components are not among the major root causes of discrepancies, their specific add-on services lead to the most discrepancies, especially when the add-ons silently modify app behavior or outputs. We realize that such issues are hard to detect with CTS/VTS, because they are unit-level tests focusing on the correctness and behavioral compatibility of individual components, rather than integration of multiple components from different stakeholders.

Without cross-component integration testing and analysis, it is hard to detect those critical issues that only manifest on the boundaries of components [66]. Note that cross-component testing and analysis would need the cooperation of different stakeholders. Google's recent initiative, CTS-Developer [4] that allows app developers to add CTS tests, is a good starting point.

Issues of regional mobile app ecosystems. It is believed that regional distinctions, which mainly lie in service platforms (e.g., GMS) and app markets (e.g., Google Play), are a major source of test result discrepancies [68]. In practice, we are able to avoid issues of service platforms by aligning the service platform of a virtual device with that of its physical counterpart. However, as shown in §5.4, regional differences in the app market could lead to ecosystem corruption, and thus have unexpected repercussions on stakeholders' strategies and policies, e.g., the adversarial mechanism design of certain Chinese vendors and apps. Thus, to enhance the testing fidelity on virtual devices, it is also important to pay attention to the conflicts of interest among stakeholders.

9 THREATS TO VALIDITY

Our study focuses on Douyin and its partner apps (Table 1). These apps and their sub-apps represent a variety of functionalities of popular apps. On the other hand, there is an inherent risk that our results may be specific to the studied apps and may not apply to all apps. For example, our study may not apply to vendor apps that are more likely to use vendor-specific features of system services and hardware. Also, the studied apps are developed by professional app development teams of Douyin with rigorous software engineering practices and automatic tooling. As a commercial app, Douyin has always treated compatibility as a first-class principle in its development; testing with physical device farms is such an example. We expect different characteristics of apps that are developed by individual developers or startup companies, which may have implications on the fidelity of virtual devices.

Moreover, we focus on functional correctness and detect failures with explicit symptoms such as native crash, Java/Kotlin exception, and app-not-responding error. Testing non-functional properties, such as energy, performance, and privacy and security, is very different from functional testing and our results are unlikely to generalize. For example, virtual devices are hard to emulate the energy consumption of physical devices with high fidelity, given the inherent differences between server and mobile hardware. Hardware- or device-independent abstractions for performance and energy could be useful [9, 38, 39, 41].

Lastly, our tests are automated UI-driven tests that are subject to the nondeterminism of dynamic UI elements (e.g., video and news content created by AI-based recommendation algorithms and UI layout changes upon refresh). Although we cannot avoid such nondeterminism, we have taken actions to minimize their impacts: (1) we synchronize the test execution on the physical and virtual devices, such as test time, network type, app account, etc.; (2) we run the tests long enough until a stable code coverage; and (3) we reproduced the common types of false positives and false negatives on the physical and virtual devices.

10 RELATED WORK

With mobile apps becoming the ubiquitous form of end-user software, mobile app testing has been an active topic of software testing research [10, 12, 15, 17, 19, 23–25, 34, 35, 37, 44, 45, 52, 55, 56, 62, 70, 74, 77, 79]. The majority of lab research on Android app testing relies on virtual devices to scale beyond the expensive physical devices [10, 12, 15, 19, 24, 25, 43, 49, 62, 70, 74, 77, 81]. However, it is unclear whether virtual devices could achieve reliable test results for app testing in industry settings, especially for global-scale apps like Douyin which are used by a large user base with very diverse mobile devices. Our work presents a deep analysis into virtual devices for app testing and serves as an enabler of using virtual devices for efficient CI/CD.

In fact, a number of advanced mobile testing techniques are built on top of virtualization features or capabilities [17, 34, 70–72]. For example, time-travel testing [17] uses the snapshot capability of virtual devices to checkpoint and restore app states that can help achieve high code coverage. Dynodroid [51] instruments framework services of Android inside a virtual device to monitor the reaction of apps upon input events, in order to guide test input generations. DroidScope [76] leverages the memory introspection capability of virtual devices to reconstruct OS- and Java-level activities (e.g., invoked Java VM instructions) of an app to detect malicious behavior. Some of these techniques can potentially be applied in virtual device farms for app testing at scale.

The concept of virtual device fidelity was studied by security research on malware detection [15, 23–25, 46, 55, 60, 77]. The goal is to prevent malware from recognizing that it is running on a virtual device by checking system signatures and configurations. Our work focuses on large-scale app testing which is a fundamentally different problem.

11 CONCLUSION

This paper presents our effort to analyze, improve, and effectively use virtual devices for large-scale testing of mobile apps such as Douyin. Our work shows that virtual devices can provide immense utilities for effective continuous app testing at scale, despite their inherent difficulties of high-fidelity emulation across diverse mobile systems and hardware devices. Our experiences show that, with careful design, implementation, and configuration, the essential fidelity gap can be effectively closed to achieve high-fidelity app testing. Although it is still hard to rule out all possible discrepancies, high-fidelity virtual device farms can substantially improve developer productivity as well as the sustainability of physical device farms. Furthermore, the elasticity and accessibility of virtual devices could enable new testing infrastructures as services for mobile apps, benefiting app developers who cannot afford large physical device farms.

ACKNOWLEDGEMENT

We are very grateful to our shepherd, Mary Baker, for her strong support and invaluable feedback. We thank the anonymous reviewers for their insightful comments. This work is supported in part by National Key R&D Program of China under grant 2022YFB4500703, National Natural Science Foundation of China under grants 61902211, 62202266 and 62272440, China Postdoctoral Science Foundation under grant 2022M721831, and Microsoft Research Asia under grant 100336949. Tianyin Xu is supported in part by NSF CNS-1956007 and CNS-2145295.

REFERENCES

[1] Amazon.com. AWS Device Farm, 2022. https://aws.amazon.com/device-farm/.
[2] Android.com. Android API Reference. https://developer.android.com/reference.
[3] Android.com. Android HAL Overview. https://source.android.com/docs/core/architecture/hal.
[4] Android.com. CTS-D: Developer-Powered CTS. https://source.android.com/docs/compatibility/cts/develop-cts-d.
[5] Android.com. Cuttlefish Virtual Android Devices, 2022. https://source.android.com/docs/setup/create/cuttlefish.
[6] Android.com. DcTracker Source Code, 2022. https://cs.android.com/android/platform/superproject/+/master:frameworks/opt/telephony/src/java/com/android/internal/telephony/dataconnection/DcTracker.java.
[7] Android.com. WifiManager Source Code, 2022. https://cs.android.com/android/platform/superproject/+/master:packages/

modules/Wifi/framework/java/android/net/wifi/WifiManager.java.

[8] Jianqiang Bao. *Android App-Hook and Plug-In Technology.* CRC Press, 2019.

[9] Reyhaneh Jabbarvand Behrouz, Alireza Sadeghi, Joshua Garcia, Sam Malek, and Paul Ammann. EcoDroid: An Approach for Energy-Based Ranking of Android Apps. In *Proc. of GREENS*, 2015.

[10] Haipeng Cai, Ziyi Zhang, Li Li, and Xiaoqin Fu. A Large-Scale Study of Application Incompatibilities in Android. In *Proc. of ISSTA*, 2019.

[11] Xiaomeng Chen, Abhilash Jindal, Ning Ding, Yu Charlie Hu, Maruti Gupta, and Rath Vannithamby. Smartphone Background Activities in the Wild: Origin, Energy Drain, and Optimization. In *Proc. of MobiCom*, 2015.

[12] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated Test Input Generation for Android: Are We There Yet? In *Proc. of ASE*, 2015.

[13] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P. Kemerlis. RETracer: Triaging Crashes by Reverse Execution from Partial Memory Dumps. In *Proc. of ICSE*, 2016.

[14] Christoffer Dall and Jason Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proc. of ASPLOS*, 2014.

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

[16] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *Proc. of S&P*, 2011.

[17] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. Time-Travel Testing of Android Apps. In *Proc. of ICSE*, 2020.

[18] Douyin.com. Douyin: Short Video Sharing Platform. https://www.douyin.com/.

[19] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. Large-Scale Analysis of Framework-Specific Exceptions in Android Apps. In *Proc. of ICSE*, 2018.

[20] Sadegh Farhang, Mehmet Bahadir Kirdan, Aron Laszka, and Jens Grossklags. An Empirical Study of Android Security Bulletins in Different Vendors. In *Proc. of WWW*, 2020.

[21] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Exploiting Nil-Externality for Fast Replicated Storage. In *Proc. of SOSP*, 2021.

[22] Di Gao, Hao Lin, Zhenhua Li, Chengen Huang, Yunhao Liu, Feng Qian, Liangyi Gong, and Tianyin Xu. Trinity: High-Performance Mobile Emulation through Graphics Projection. In *Proc. of OSDI*, 2022.

[23] Liangyi Gong, Zhenhua Li, Feng Qian, Zifan Zhang, Qi Alfred Chen, Zhiyun Qian, Hao Lin, and Yunhao Liu. Experiences of Landing Machine Learning onto Market-Scale Mobile Malware Detection. In *Proc. of EuroSys*, 2020.

[24] Liangyi Gong, Zhenhua Li, Hongyi Wang, Hao Lin, Xiaobo Ma, and Yunhao Liu. Overlay-Based Android Malware Detection at Market Scales: Systematically Adapting to the New Technological Landscape. *Transactions on Mobile Computing*, 21, 2022.

[25] Liangyi Gong, Hao Lin, Zhenhua Li, Feng Qian, Yang Li, Xiaobo Ma, and Yunhao Liu. Systematically Landing Machine Learning onto Market-Scale Mobile Malware Detection. *Transactions on Parallel and Distributed Systems*, 32, 2021.

[26] Google.com. Deadlock in the GC Mechanism of ART JVM. https://issuetracker.google.com/issues/37013231.

[27] Google.com. Emulator Crash when Playing YV12 Videos. https://issuetracker.google.com/issues/262255458.

[28] Google.com. Incorrect Handling of Page Faults in the FUSE Filesystem. https://issuetracker.google.com/issues/37131442.

[29] Google.com. Integer Overflow in IServiceManager. https://android.googlesource.com/platform/frameworks/native/+/16c6e7073231dc9e9a775acb5aa5c456ca38851f.

[30] Google.com. Android Bootup Marks, 2022. https://partnermarketinghub.withgoogle.com/brands/android/visual-identity/visual-identity/android-bootup-marks/.

[31] Google.com. Google Firebase Test Lab, 2022. https://firebase.google.com/products/test-lab.

[32] Google.com. libndk Dynamic Binary Translation Library, 2022. https://github.com/newbit1/libndk_translation_Module.

[33] Qualitest Mobile Testing Group. Mobile Emulators vs. Real Devices, 2022. https://qualitestgroup.com/insights/white-paper/mobile-emulators-vs-real-devices/.

[34] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. Practical GUI Testing of Android Applications Via Model Abstraction and Refinement. In *Proc. of ICSE*, 2019.

[35] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. PUMA: Programmable UI-Automation for Large-Scale Dynamic Analysis of Mobile Apps. In *Proc. of MobiSys*, 2014.

[36] Brian Hong. Sleight of ARM: Demystifying Intel Houdini, 2021.

[37] Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. Efficiently, Effectively Detecting Mobile App Bugs with AppDoctor. In *Proc. of EuroSys*, 2014.

[38] Rishabh Iyer, Jiacheng Ma, Katerina Argyraki, George Candea, and Sylvia Ratnasamy. The Case for Performance Interfaces for Hardware Accelerators. In *Proc. of HotOS*, 2023.

[39] Reyhaneh Jabbarvand, Forough Mehralian, and Sam Malek. Automated Construction of Energy Test Oracles for Android. In *Proc. of FSE/ESEC*, 2020.

[40] Wuxia Jin, Yitong Dai, Jianguo Zheng, Yu Qu, Ming Fan, Zhenyu Huang, Dezhi Huang, and Ting Liu. Dependency Facade: The Coupling and Conflicts between Android Framework and Its Customization. In *Proc. of ICSE*, 2023.

[41] Abhilash Jindal and Y Charlie Hu. Differential Energy Profiling: Energy Optimization via Diffing Similar Apps. In *Proc. of USENIX OSDI*, 2018.

[42] Taeyeon Ki, Alexander Simeonov, Bhavika Pravin Jain, Chang Min Park, Keshav Sharma, Karthik Dantu, Steven Y Ko, and Lukasz Ziarek. Reptor: Enabling API Virtualization on Android for Platform Openness. In *Proc. of MobiSys*, 2017.

[43] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. Automated Testing of Android Apps: A Systematic Literature Review. *Transactions on Reliability*, 68(1), 2018.

[44] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. DroidBot: A Lightweight UI-Guided Test Input Generator for Android. In *Proc. of ICSE*, 2017.

[45] Chieh-Jan Mike Liang, Nicholas D. Lane, Niels Brouwers, Li Zhang, Börje F. Karlsson, Hao Liu, Yan Liu, Jun Tang, Xiang Shan, Ranveer Chandra, and Feng Zhao. Caiipa: Automated Large-Scale Mobile App Testing through Contextual Fuzzing. In *Proc. of MobiCom*, 2014.

[46] Bin Liu, Suman Nath, Ramesh Govindan, and Jie Liu. DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps. In *Proc. of NSDI*, 2014.

[47] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proc. of ICSE*, 2014.

[48] Xuan Lu, Xuanzhe Liu, Huoran Li, Tao Xie, Qiaozhu Mei, Dan Hao, Gang Huang, and Feng Feng. PRADA: Prioritizing Android Devices for Apps by Mining Large-Scale Usage Data. In *Proc. of ICSE*, 2016.

[49] Chu Luo, Jorge Goncalves, Eduardo Velloso, and Vassilis Kostakos. A Survey of Context Simulation for Testing Mobile Context-Aware Applications. *Computing Surveys*, 53(1), 2020.

[50] Zhengwei Lv, Chao Peng, Zhao Zhang, Ting Su, Kai Liu, and Ping Yang. Fastbot2: Reusable Automated Model-Based GUI Testing for Android Enhanced by Reinforcement Learning. In *Proc. of ASE*, 2022.
[51] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An Input Generation System for Android Apps. In *Proc. of FSE/ESEC*, 2013.
[52] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-Objective Automated Testing for Android Applications. In *Proc. of ISSTA*, 2016.
[53] Andrea Possemato, Simone Aonzo, Davide Balzarotti, and Yanick Fratantonio. Trust, But Verify: A Longitudinal Analysis Of Android OEM Compliance and Customization. In *Proc. of S&P*, 2021.
[54] Ivan Pustogarov, Qian Wu, and David Lie. Ex-vivo dynamic analysis framework for android device drivers. In *Proc. of S&P*, 2020.
[55] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. Making Malory Behave Maliciously: Targeted Fuzzing of Android Execution Environments. In *Proc. of ICSE*, 2017.
[56] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. AppInsight: Mobile App Performance Monitoring in the Wild. In *Proc. of OSDI*, 2012.
[57] Rusty Russell. virtio: Towards a De-Facto Standard For Virtual I/O Devices. *SIGOPS Operating Systems Review*, 42(5), 2008.
[58] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. Continuous Deployment at Facebook and OANDA. In *Proc. of ICSE*, 2016.
[59] Yuru Shao, Ruowen Wang, Xun Chen, Ahemd M. Azab, and Z. Morley Mao. A Lightweight Framework for Fine-Grained Lifecycle Control of Android Applications. In *Proc. of EuroSys*, 2019.
[60] Wenna Song, Jiang Ming, Lin Jiang, Han Yan, Yi Xiang, Yuan Chen, Jianming Fu, and Guojun Peng. App's Auto-Login Function Security Testing via Android OS-Level Virtualization. In *Proc. of ICSE*, 2021.
[61] Ting Su, Lingling Fan, Sen Chen, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. Why My App Crashes? Understanding and Benchmarking Framework-Specific Exceptions of Android Apps. *Transactions on Software Engineering*, 48, 2022.
[62] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, Stochastic Model-Based GUI Testing of Android Apps. In *Proc. of FSE*, 2017.
[63] Ting Su, Jue Wang, and Zhendong Su. Benchmarking Automated GUI Testing for Android against Real-World Bugs. In *Proc. of FSE/ESEC*, 2021.
[64] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. Testing Configuration Changes in Context to Prevent Production Failures. In *Proc. of OSDI*, 2020.
[65] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: Facilitating Dynamic Analysis of Device Drivers of Mobile Systems. In *Proc. of Security*, 2018.
[66] Lilia Tang, Chaitanya Bhandari, Yongle Zhang, Anna Karanika, Shuyang Ji, Indranil Gupta, and Tianyin Xu. Fail through the Cracks: Cross-System Interaction Failures in Modern Cloud Systems. In *Proc. of EuroSys*, May 2023.
[67] BroswerStack Mobile Testing Team. Mobile Testing on Emulators and Simulators, 2022. https://www.browserstack.com/emulators-simulators.
[68] SauceLabs Mobile Testing Team. Mobile App Testing for International Markets, 2022. https://saucelabs.com/blog/mobile-app-testing-for-international-markets.
[69] Shuai Wang, Xinyu Lian, Darko Marinov, and Tianyin Xu. Test Selection for Unified Regression Testing. In *Proc. of ICSE*, 2023.
[70] Wenyu Wang, Wing Lam, and Tao Xie. An Infrastructure Approach to Improving Effectiveness of Android UI Testing Tools. In *Proc. of ISSTA*, 2021.
[71] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. An Empirical Study of Android Test Generation Tools in Industrial Cases. In *Proc. of ASE*, 2018.
[72] Wenyu Wang, Wei Yang, Tianyin Xu, and Tao Xie. Vet: Identifying and Avoiding UI Exploration Tarpits. In *Proc. of FSE/ESEC*, 2021.
[73] Lili Wei, Yepang Liu, and Shing-Chi Cheung. Taming Android Fragmentation: Characterizing and Detecting Compatibility Issues for Android Apps. In *Proc. of ASE*, 2016.
[74] Lili Wei, Yepang Liu, Shing-Chi Cheung, Huaxun Huang, Xuan Lu, and Xuanzhe Liu. Understanding and Detecting Fragmentation-Induced Compatibility Issues for Android Apps. *Transactions on Software Engineering*, 46, 2020.
[75] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. Early Detection of Configuration Errors to Reduce Failure Damage. In *Proc. of OSDI*, 2016.
[76] Lok-Kwong Yan and Heng Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proc. of Security*, 2012.
[77] Yuxuan Yan, Zhenhua Li, Qi Alfred Chen, Christo Wilson, Tianyin Xu, Ennan Zhai, Yong Li, and Yunhao Liu. Understanding and Detecting Overlay-Based Android Malware at Market Scales. In *Proc. of MobiSys*, 2019.
[78] Qifan Yang, Zhenhua Li, Yunhao Liu, Hai Long, Yuanchao Huang, Jiaming He, Tianyin Xu, and Ennan Zhai. Mobile Gaming on Personal Computers with Direct Android Emulation. In *Proc. of MobiCom*, 2019.
[79] Chao-Chun Yeh, Shih-Kun Huang, and Sung-Yen Chang. A Black-Box Based Android GUI Testing System. In *Proc. of MobiSys*, 2013.
[80] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Proc. of NDSS*, volume 14, 2014.
[81] Samer Zein, Norsaremah Salleh, and John Grundy. A Systematic Mapping Study of Mobile Application Testing Techniques. *Elsevier Journal of Systems and Software*, 117, 2016.
[82] Mary Zhang. Top 10 Cloud Service Providers Globally in 2022, 2022. https://dgtlinfra.com/top-10-cloud-service-providers-2022/.

A ARTIFACT APPENDIX

A.1 Abstract

The artifact of this paper is hosted at GitHub. The artifact contains all the (anonymized) failure data collected in our comparative study of physical and virtual device farms, and evaluation scripts that reproduce all the major tables and figures (except Table 1 whose data are inherently included in the table and thus is not additionally provided by the artifact) in the paper. We also offer our scripts in Google Colab Notebook, which enables push-button reproduction of the tables and figures without any additional runtime setup.

Artifact Claim: Our artifact contains all the data of our large-scale comparative study. You can replicate all the relevant results in Section 4 and Section 5 using the dataset we provided in our GitHub repository. Our artifact currently does not contain the detailed app failure traces after our enhancements in Section 6, and does not offer access to our proprietary virtual and physical farms for the reproduction of the study (which is highly expensive and time-consuming), due to commercial constraints.

A.2 Artifact check-list (meta-information)

- **Data set:** Our dataset is included in the repository and is automatically decompressed by the evaluation scripts.
- **Run-time environment:** You can directly reproduce all the major results via our online Google Colab Notebook without runtime setup. If you wish to run the evaluation scripts locally, the Python 3 runtime is required. All the other dependencies can be installed via a `pip3 install -r requirements.txt` command at the repository root.
- **Execution:** The execution takes approximately 2 minutes.
- **Output:** The output includes Table 2-5 and Figure 1-6 in the paper.
- **Experiments:** Follow our README in the GitHub repository or the Google Colab Notebook. If you are using Colab, the experiments should be as simple as clicking on the `Runtime—Run All` button.
- **How much disk space required (approximately)?:** None if you are using the Colab. Local evaluation requires at least 3 GB for storing the dataset.
- **How much time is needed to prepare workflow (approximately)?:** One minute.
- **How much time is needed to complete experiments (approximately)?:** Three minutes.
- **Publicly available?:** Yes.
- **Workflow framework used?** No, but scripts are provided to automate the result reproduction.
- **Code licenses (if publicly available)?:** We use the GNU General Public License v3.0 (GPLv3) license for the code.
- **Data licenses (if publicly available)?:** The data license is also GPLv3.
- **Archived (provide DOI)?:** 10.5281/zenodo.8260603.

A.3 Description

A.3.1 How to access. Choose either of the following methods to access our artifact.

- GitHub: https://github.com/Android-Emulation-Testing/emu-fidelity-ae.
- Google Colab: https://colab.research.google.com/drive/19DYtr3yrJs6aKrXXyKWrBbMsCEvw46qw?usp=sharing

The GitHub repository contains all the code and data which you can play with locally, while Google Colab offers an online interactive interface that can reproduce all the major results without environmental setup.

A.3.2 Software dependencies. Local execution of the evaluation scripts require the Python 3 runtime and the Python-Pip package management tool. All the other dependencies can be installed via a `pip3 install -r requirements.txt` command at the repository root.

A.4 Installation

Google Colab. You can directly click on the Google Colab link in your browser (Chrome, Microsoft Edge, Firefox, or Safari) to access our Colab.

Local Experiments.

- If you wish to replicate our results using the dataset in the artifact, first clone the repository from GitHub:

  `git clone` https://github.com/Android-Emulation-Testing/emu-fidelity-ae.
- Install Python 3 and Python3-Pip if your environment does not contain it.
- At the root directory of the cloned repository, execute the `pip3 install -r requirements.txt` command.

The README files in both Colab and GitHub provide a detailed description of the structure of our dataset.

A.5 Experiment workflow

You can go with either one of the following approaches for result replication depending on your specific requirements. The Colab offers a quick start to reproduce our figures and tables with a single click, but is read-only and thus cannot be modified to realize other functionalities. The local approach requires downloading the dataset and basic setup of the Python environment, but can be tailored to your specific requirements, such as building your own scripts to examine the dataset.

A.5.1 Reproduction with Google Colab. You can use the Colab to execute our evaluation scripts that produce Table 2-5 and Figure 1-6 in the paper. Table 1's data are inherently included in the table and thus is not additionally provided. The page is already executed upon accesses, but you can always re-execute it. To do this, click the `Runtime` tab at the top left of the Colab web page, and click the `Run All` button to execute the scripts.

Each figure/table's reproduction code has been organized in a separate cell in the page. Each cell can be executed independently. However, to execute any of the cells for plotting the figures or printing the tables, your should execute the top-three cells first to setup the environment and dataset.

After the execution of a cell, you should see the figures or tables displayed below the cell. The tables are printed in the form of text.

A.5.2 Local Reproduction of Figures and Tables. Having cloned the repository from GitHub, first read the `README.md` file that describes the structure of our dataset, and how to setup the environment.

Next, execute the `python3 plot.py` command at the root directory of the repository to reproduce the figures and tables. The figures will be saved to the `fig` folder at the root directory of the repository, while the tables will be printed to `stdout`, which usually should be the terminal/console you are using to run the above command.

A.5.3 Result Replication with the Dataset (Optional). You can also choose to build your own data analysis scripts to examine our dataset. We provide a detailed description of the dataset's structure in the repository's README file, which is organized as a CSV file. Basically, each row of the dataset represents a single failure (in physical or virtual devices), while each colum contains an attribute of the failure, such as the failure call stack, failure reason, and device model. You can separate failures in physical devices and virtual devices via the device model attribute, which is `virt` for virtual devices. By aggregating data over different attributes of the failures, you can replicate most of the statistics described in Section 4 and Section 5. We also offer basic data processing utility in our plotting script (`plot.py`).

A.6 Evaluation and expected results

The produced figures and tables should match there correspondences in the paper. We have marked each output figure or table with their ID in the paper. They are expected to match the figures, tables, or statistics described in Section 4, 5, 6. Note that since we do not provide detailed app failure traces after our enhancements in Section 6, Figure 5 and Figure 6 (which show enhancement effectiveness) can only be produced by our figure plotting script and cannot be derived from the dataset.