Towards A Contextual and Scalable Automated-testing Service for Mobile Apps

Li Lyna Zhang[⋆‡], Chieh-Jan Mike Liang[‡], Wei Zhang[⋆‡], Enhong Chen[⋆]
[⋆]University of Science and Technology of China [‡]Microsoft Research

ABSTRACT

As app quality is a deciding factor for user base growth, many automated testing services are available to reduce app developers' burden. However, we argue that these existing services do not sufficiently bring real-world contexts into app testing, which reduces the visibility into how an unreleased app would perform in the wild. In fact, this is a challenging problem that current emulator-based or device-based testing services cannot properly or scalably address. This paper envisions a split-execution model for building automated and contextual testing services for mobile apps. This model allows the service to evolve over time, by adopting new algorithms and recruiting new physical devices. Finally, preliminary results from a prototype demonstrate the potential and feasibility of our proposed architecture.

CCS Concepts

•Software and its engineering → Software testing and debugging;

Keywords

App testing; Split Execution; Device labs; Contextual Fuzzing

1. INTRODUCTION

Reducing developers' burden on achieving testing coverage can drive the mobile app quality, especially that app complexities have generally grown beyond what manual testing can reasonably cover [17]. As such, the demand for app testing services on the market has increased significantly in the past few years. These services [6, 23, 27] offer a controlled environment for repetitive testing, with a reasonable cost and turnaround time.

Yet current realizations of automated testing have shortcomings in efficiently scaling the *system size* and the *testing dimension*. One popular approach is to use mobile device emulators [12] as the app execution container. Unfortunately, while emulator instances can be easily started to scale the system size, they cannot accurately emulate many real-world contexts [4]. This is crucial because the ubiquity of mobile devices implies that mobile apps can encounter contexts of diverse device specifications, various geo-locations, wide spectrum of network conditions, etc. Another approach is physical device labs, or pools of shared mobile devices that developers can access remotely [26, 27] or locally [18]. While device labs offer the realism w.r.t. device configurations, it is relatively expensive to purchase and maintain devices of different hardware configurations and operating systems. And, certain real-world contexts such as network conditions and geo-locations are fixed to the device lab. Finally, there are efforts proposing application-layer record-and-reply [14, 15] to simplify repetitive testing involving sensor inputs, but they cannot adequately capture certain contexts such as network transmissions.

This paper rethinks "*how automated app testing services should evolve to scalably achieve realism of the vast testing space of real-world contexts*". We aim to provide time-series logs for developers to quantify real app behavior in the wild, without actually releasing app binaries to the public. From our own experience with app testing [10, 19], we highlight the following **design principles**:

First, while a vertical system model is easy to build, a horizontal model that decouples computation (e.g., algorithms and resources) and app execution containers would better incorporate rapid innovations from the community. Specifically, the community continuously advances on techniques, and the industry periodically releases new mobile devices and software. Decoupling enables the automated testing service to evolve by independently adopting new testing algorithms and growing the device pool. This design conceptually resembles the narrow-waist of the IP stack.

Second, to make real-world contexts accessible for testing, code blocks that access or interact with real-world contexts must be pinned to physical devices in the wild. While there are efforts emulating real-world contexts in software [19], the realism still cannot match that of physical devices. Interestingly, these code blocks are mostly implemented in the OS and exposed through system APIs, e.g., sensors, GPS, and networking stack. Furthermore, context-insensitive APIs can be serviced inside the emulator, and they do not need to run on the physical device. This split-execution design also prevents disclosing unreleased apps from public eyes [9].

Third, the overhead for maintaining and operating the testing service should be low [17]. For instance, adding physical devices (which could be retired smartphones) to the pool should not require device rooting, or anything more than in-

HotMobile '17, February 21-22, 2017, Sonoma, CA, USA
 ISBN 978-1-4503-4907-9/17/02…$15.00
DOI: http://dx.doi.org/10.1145/3032970.3032972

stalling a simple agent app. In addition, developers should not need to modify their app binaries.

Realizing these design principles, however, requires the infrastructure support. Fortunately, we can leverage the popular "cloud-and-device" infrastructure. Conceptually, a single test session can benefit from cloud for heavy computation. Then, with devices sitting at the last mile of Internet consisting of access points and cellular connections, mobile devices can capture the real-world realism for the test session.

This work contributes to the above vision of evolving automated app testing. From our previous experience in app testing [10, 19], we are motivated by pain points from both testing service providers (e.g., Microsoft) and app developers. To this end, our prototype, *RainDrops*, builds on the narrow-waist architecture to realize split-execution for contextual app testing.

2. BACKGROUND AND MOTIVATIONS

2.1 Real-world Contexts Relevant to Apps

Testing coverage should consider various real-world contexts that mobile apps experience in the wild.

Device Diversity. Mobile devices have a wide range of computation and memory capabilities, display sizes, OS versions, etc. These differences can be reflected in the app behavior. For example, on memory-limited devices, mobile apps that consume too much memory could experience out-of-memory errors, or have their own background services be killed by the OS at an arbitrary time. In fact, Android's well-known fragmentation problem further motivates the need to consider device diversity during testing.

Network Environment Diversity. 83% of apps in Google Play use network access [24]. And, crawling OpenSignal [3] suggests that there are more than 500 distinctive combinations of network properties such as provider, bandwidth, latency, and loss. Liang et al. [19] reported an interesting location bug where one social app's crashing frequency is proportional to the network latency. Unfortunately, with limited resources, app developers cannot easily gain insights into how network conditions would impact the app behavior.

Geo-location Diversity. More than 45% of apps in Google Play request permissions for geo-location data [24], and apps may run at different locations throughout the day. Failure to consider geo-locations during testing can lead to false negatives. For example, we observed that one social app is brittle in countries outside US (which is confirmed by user comments posted online).

2.2 Existing Gaps in Testing Coverage

Table 1 summarizes shortcomings that current automated testing practices have in achieving testing coverage.

Testing with Emulators. Being a software-based solution, emulators are relatively cost-efficient in spawning and setting up new instances.

Emulators cannot sufficiently cover several aspects of testing dimensions relevant to mobile apps – *(i) Device Diversity*: emulators typically present a "standard" device type, so they cannot perfectly capture the hardware-specific (e.g., I/O) and system-specific (e.g., device drivers) characteristics. For example, there is a poor support for emulating orientation, accelerometers and geo-location [4]. *(ii) Network Environment Diversity*: while some emulators can perform traffic shaping to emulate different network conditions [20], they typically follow measurements in the lab, rather than reflecting real-world conditions. *(iii) Geo-location Diversity*: some emulators have a custom GPS driver that can fake coordinates as desired [12, 20]. However, changing coordinates alone might not be sufficient, as many others parameters are also relevant, e.g., corresponding network conditions and GPS signal strength.

API categories	Real-world contexts
Geo-location	Coordinates
Network	Latency, packet loss, bandwidth
Sensors	Environment dynamics
File I/O	Storage medium
Device info	Device ID
Others	Encryption engine

Table 2: Categories of context-sensitive system APIs that RainDrops offloads to physical devices.

Testing with Physical Devices. Industries and research communities have invested significant efforts into building real-world device labs [7, 16]. Device labs can be viewed as racks of mobile devices wire-connected to some centralized management service, and developers can remotely start testing jobs.

Device labs cannot sufficiently support several aspects of testing dimensions relevant to mobile apps – *(i) Device Diversity*: the cost of growing and maintaining physical device pools can hinder the scalability. *(ii) Network Environment Diversity*: while most device labs have both Wi-Fi and cellular connectivity, network conditions are fixed to the lab setup. *(iii) Geo-location Diversity*: since device labs are situated at some fixed locations, they do not have the flexibility necessary to provide geo-location diversity.

2.3 Potential of Idle Mobile Devices in the Wild

We now discuss the potential and feasibility of idle mobile devices as a main building block for the contextual testing service. One example of idle devices could be retired ones that are simply stored in drawers at home [21, 28].

The rapid smartphone upgrade cycle implies that a large fraction of idle mobile devices are less than two generations old [8]. These devices have reasonably up-to-date hardware configurations, and they can still run modern operating systems. For example, Android Jelly Bean 4.1 was reported to have 26.5% of market share 2 years after its initial release date [22].

Furthermore, since idle devices are scattered in the real world, different devices exhibit different last-mile link conditions and geo-location coordinates. For example, 3,997 different models of Android devices – with more than 250 screen resolution sizes – contributed to the OpenSignal database hundreds of distinct network conditions and geo-locations during a recent six-month period.

3. PROPOSED DESIGN AND CHALLENGES

3.1 Narrow-waist Architecture

Fig 1 illustrates the proposed narrow-waist architecture. Coordinator separates and coordinates two types of execution containers: the cloud and the physical device. The former has computation capabilities to run **(i)** emulators and

	Emulators	Device labs	RainDrops
Device diversity for testing	**Low**: "Standard" profiles are tested	**High**	**Medium**: CPU and memory are partially emulated
Network env diversity for testing	**Medium**: "Standard" profiles are tested	**Low**: Fixed to the device lab setup	**High**
Geo diversity for testing	**Medium**: Timing properties are not emulated	**Low**: Fixed to the device lab location	**High**
Sensing diversity for testing	**Low**: Simplistic sensing data models are used	**High**	**High**
System scalability	**High**	**Low**: High costs in purchasing and maintaining devices	**High**

Table 1: Comparisons of different approaches for automated app testing services.

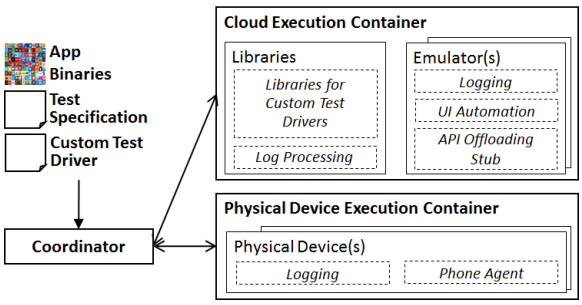


Figure 1: Architectural overview of RainDrops.

(ii) submitted custom test drivers that may call built-in libraries. The latter runs a light-weight Phone Agent app to receive and execute offloading commands. Table 2 lists types of system APIs that Phone Agent can service.

In contrast to the traditional testing practice, RainDrops takes advantage of the narrow-waist architecture to carefully split the app execution load between the cloud container and the physical device. Such a *split-execution* model is feasible because mobile apps typically have three distinctively functional blocks: UI layouts, user-mode computation, and kernel-mode system APIs. Importantly, a subset of system APIs (c.f. Table 2) enables mobile apps to act on real-world events and contexts, e.g., network transmissions, sensing, etc. If these context-sensitive system API calls are serviced on a physical device, the app effectively acts on that device's real-world contexts. Furthermore, RainDrops runs all other code blocks (including UI) inside the emulator so that the cloud can drive the app automation. The transfer of execution happens over an overloading channel between a pair of emulator and physical device.

The proposed architecture promotes three desirable characteristics. First, the computation is decoupled from the physical device pool, so the testing service can evolve by incorporating new libraries for the custom test driver, and organically growing the device pool. Second, by offloading only known context-sensitive system APIs to the physical device, the Phone Agent app can have a straight-forward design. In addition, since operating systems rarely introduce significant changes to published APIs, the Phone Agent app does not impose high maintenance overhead. Third, high app confidentiality is guaranteed by not hosting the entire app on the physical device, especially the UI.

Finally, we note that app developers do not need to modify their app binaries. Since RainDrops offloads only system API calls to the physical device, RainDrops can be implemented at the OS level in the emulator. On the device side, the Phone Agent app does not require rooting the device, as it only calls standard system APIs.

3.2 Challenges

The split-execution model introduces following challenges to the automated app testing service.

Dealing with Internet Uncertainties. Since the offloading channel is established over the Internet, offloading can violate certain timing expectations that app developers assume if system APIs were executed locally.

First, offloading inevitably introduces Internet latencies to the end-to-end system API call. For instance, app developers can have some expectations on the communication with backend servers, expressed in the form of time-outs. However, if the offloading channel significantly slows down for any reason, the developer can see more time-outs than expected. Even for system APIs without time-outs (e.g., getting device ID), the additional latency is present in time-series logs produced.

Second, Internet can exhibit packet losses, and this translates into data losses for sensors with periodic sampling. This problem is similar to reliable data streaming over Internet. While TCP can address the losses, it adds additional delays and bandwidth overhead undesirable for slow overloading channel.

Fusing Logs From Heterogeneous Execution Containers. With the split-execution, parts of the app execution take place in different execution containers. Fusing logs from these heterogeneous sources can be challenging.

First, different execution containers can have heterogeneous properties. For example, as the emulator cannot perfectly emulate CPUs, the emulator might be faster or slower than the physical device. In this case, naively merging pieces of time-series logs according to execution timestamps might not yield right results. Second, since execution containers maintain an offloading channel over the Internet, artifacts due to offloading are considered to be noise and should be carefully removed.

Lowering Test Turnaround Time. A number of factors typically limit the test turnaround time. For one, many automated services run random UI exploration to achieve UI coverage, which has been shown to be very inefficient [13].

Another factor is the test space explosion from introducing real-world contexts into automated app testing. In RainDrops, each physical device represents one set of real-world contexts (c.f. Table 2): {`geo-location`, `network`, `sensors`, ...}. Since the range of values for each context can be large, the test space from combinations of real-world contexts can

be infeasible to completely explore. For instance, if it takes 10 minutes to automate an app under a set of real-world contexts, RainDrops could only finish six testing sessions in an machine-hour.

Standard Programming Interfaces. To give developers control over how their apps should be tested, the testing service should have standard programming interfaces to expose certain service functionalities.

4. FEASIBILITY AND PROTOTYPE

For the feasibility study, we have been prototyping an automated testing service for Android apps (c.f. Fig 1), RainDrops. This section discusses our efforts.

4.1 Testing Session Planning

App developers submit via the cloud portal: **(1)** an Android app binary, **(2)** a test specification file, and **(3)** an optional custom test driver.

The test specification is a XML file describing sets of real-world contexts to test: geo-locations, network conditions, device configuration, OS version, etc. According to the test specification, Coordinator initializes an emulator instance and finds a physical device that can provide the requested contexts. For this matching, we note that each physical device runs a Phone Agent app that registers the device capabilities and contexts with Coordinator. Finally, the emulator instance establishes a TCP connection to the selected device. We choose Genymotion [12] as it is one of the fastest Android emulators on the market.

App developers optionally submit a custom test driver, to override the default planning of random UI exploration. This test driver can directly call functions in libraries hosted by the cloud execution container, e.g., Android app binary decompiler [1], control/data flow analysis tools, UI automator [2], etc. We note that new libraries can be added.

As an example, one custom test driver that we have built implements *targeted UI automation*. Since most mobile app package formats are compiled to an intermediate byte code (e.g., the Java bytecode format in the case of Android), decompilation can recover the program structure such as variables and methods. Then, combining data flow analysis and UI metadata can construct UI flow graph (UIFG), which shows transitions among app UI pages. Compared to naive random exploration, UIFG can guide automated testing to efficiently traverse all app pages.

4.2 Split-Execution on Apps

After loading app binaries in the emulator, RainDrops exercises the app UI either randomly or as instructed by the custom test driver. Split-execution occurs as the app execution moves across the boundary of selected system APIs (c.f. Table 2).

Intercepting System API Calls. The emulator relies on Xposed [5] to modify the Android system process, Zygote, to add redirections before and after pre-specified API functions. Table 2 lists types of context-sensitive system APIs that RainDrops intercepts. Upon an interception, the emulator records arguments, and then its Remote Invocation Stub forwards them over the offloading channel. During the offloading, the system API call is blocked by Xposed.

Offloading System API Calls. Fig 2 illustrates offloading. Phone Agent on the physical device waits for offloading

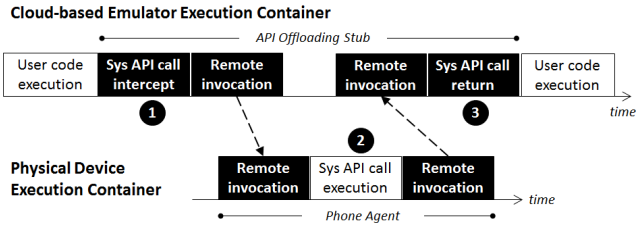


Figure 2: Intercepting and offloading an app's system API calls.

commands. Offloading is serviced by calling the system API specified in the command. Then, the return value is sent back to the emulator over the same offloading channel.

We note that offloading system APIs over Internet can lose certain timing guarantees that app developers assume system APIs to have, especially considering the Internet fluctuation over time. To this end, we are currently investigating the following two strategies:

First, many sensors (e.g., accelerometer) allow apps to register for periodic data updates. To minimize the impact of Internet fluctuations on supposedly periodic events, the emulator prefetches and buffers a certain amount of sensor data prior to the testing session. Then, at each periodic time instance, the emulator feeds the app one data point from the buffer. At the same time, the device still continuously streams new sensing data points. To determine sensors to prefetch and buffer, RainDrops statically analyzes the decompiled app binaries. Finally, we note that, since all sensor streams need to be temporally aligned, prefetching happens for all requested sensors at the same time.

Second, some system APIs have a developer-specified timeout threshold, and an example is HTTP requests issued to get content updates from remote servers. The additional latencies due to API offloading can trigger unexpected timeouts. RainDrops handles this situation by intercepting developer's attempts to set the timeout threshold, and increase it before passing to the OS.

4.3 Aggregation of Logs From Heterogeneous Execution Containers

Due to split-execution, both the emulator and the physical device need to log app performance counters. Presenting an unified view requires RainDrops to aggregate logs from these two heterogeneous sources, as described in §3.2. Fig 2 illustrates steps of offloading. Conceptually, RainDrops should present data relevant to app execution (i.e., white boxes), not noises due to the offloading overhead (i.e., black boxes). Then, RainDrops needs to minimize the differences due to heterogeneous sources. The rest of this section discusses our current implementation.

Step 1: Noise Removal on Emulator Logs. We are exploring two methods to minimize the presence of noise (i.e., black boxes in Fig 2) in the feedback to app developers.

The first method relies on event timestamps for noise removal. Specifically, Remote Invocation Stub records timestamps of system call interception and completion, and the time period between these two timestamps are discarded. While this method is simple to implement, it might not well handle the case of multi-threaded apps. For instance, if thread #1's system API offloading overlaps with thread #2's

user code execution, it is difficult to separate the resource utilization of each thread at a time instance.

The second method under consideration is *execution slow-down*. Namely, the app execution is slowed down while the Internet-induced offloading remains the same. Although execution slow-down has benefited various testing scenarios [11, 29], our goal is to reduce the ratio of offloading in the time-series log, which in turn increases the size ratio of white boxes to black boxes in Fig 2. Effectively, this creates the illustration that the offloading delay is sufficiently short to be reasonably ignored. To achieve execution slow-down, we are investigating ways to clock down the CPU.

Step 2: Log Segmentation. Next, RainDrops determines temporal boundaries of user code execution and system API execution. This step can be done by having Remote Invocation Stub and Phone Agent record event timestamps.

Step 3: Piece-wise Merging. After noise removal in step 1, the Internet-induced offloading delay should be minimized. And, the emulator log should have a "gap" where the app makes the system API call. Intuitively, this gap can be filled by merging the time period system API execution on the physical device (which is determined in Step 2). However, since raw logs are generated on heterogeneous devices, piece-wise merging requires considerations below.

First, directly taking absolute values from logs does not work, as different execution containers may already be using different levels of (base) resource utilization before the system API call. So, on the physical device, RainDrops considers only the resource utilization increase due to servicing the system API call.

Second, it is possible that the gap's starting point, end point, or even the size do not match those of the physical device log. Therefore, we currently transform the physical device log by stretching or compressing, as necessary.

5. PRELIMINARY EVALUATION

This section addresses the following questions: **(1)** Are logs from RainDrops sufficiently accurate? **(2)** What are resource costs for an idle device in the wild to join RainDrops? **(3)** How has RainDrops helped developers in the real world so far?

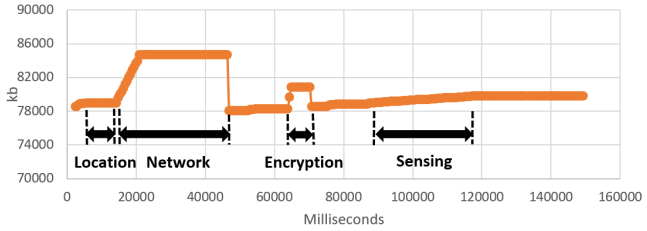
5.1 Methodology

We implement the RainDrops with Genymotion emulators hosted on a Windows 8 Server (with quad-score 2.5 GHz CPU), and two popular models of Android smartphones: Nexus 6 and Galaxy S5. Our device pool currently spans over locations in Beijing and Seattle. The offloading channel uses JSON as the message format.

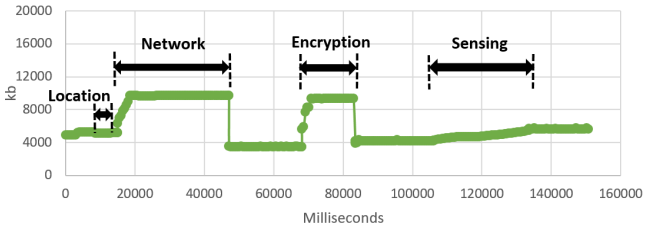
To generate reproducible workloads, we develop an Android benchmarking app, RainTester, to perform four sequenced tasks: requesting geo-location updates, posting HTTP requests to a popular weather server, encrypting a large piece of data, and requesting accelerometer data. We log the relevant performance counters – *CPU*: utilization and user time, *memory*: private dirty, and *network*: HTTP request-reply latency and incoming/outgoing packet rate.

5.2 Are Logs from RainDrops Sufficiently Accurate?

To show that RainDrops can provide accurate logs as developer feedback, we use the traditional device-based testing as the comparison baseline. And, we run the RainTester app under different real-world contexts. Fig 3(a) and Fig 3(b) show the private dirty memory consumption while running RainTester with the traditional testing of real devices and RainDrops, respectively. The fact that both curves exhibit a very similar trend suggests RainDrops can provide sufficiently accurate logs.



(a) Memory usage logged by traditional testing of real devices



(b) Memory usage logged by RainDrops

Figure 3: App behavior logged by RainDrops closely resembles that logged by traditional testing of real devices. Through event timestamping, we annotate both figures with the RainTester app's system API calls.

	CPU user time	Private dirty memory
Location	0.973	0.898
Networking	0.803	0.876
Encryption	0.806	0.641
Sensing	0.608	0.964

Table 3: We quantify the testing accuracy of by calculating the similarity score (r) for RainTester. A score close to 1 would suggest that RainDrops logs are very similar to traditional testing of real devices.

We further examine this observation by quantifying the similarity of curve features captured by RainDrops and the comparison baseline. First, since emulators cannot perfectly emulate the CPU clock speed, we align two time-series logs by applying Dynamic Time Wrapping (DTW) [25] in the time axis. Then, we compute the Pearson correlation coefficient (r) to quantify the log similarity. Since DTW does not change curve features, r can reflect the similarity with a score between -1 (i.e., total negative correlation) and 1 (i.e., total positive correlation).

Table 3 shows the r score w.r.t. both CPU user time and private dirty memory from running the RainTester app. Most r values are above 0.8, which indicates logs from RainDrops closely resemble those from traditional device-based testing. Furthermore, we note that encryption exhibits the lowest similarity score on private dirty memory in Table 3. This is due to Phone Agent consuming some memory to cache the content to be encrypted. In addition, sensing exhibits the lowest similarity score on CPU user time. That

	Idle wait	Execution
CPU util (%)	0.2	1.3
Private dirty memory (KB)	29488.1	33700.0
Network (KB/s)	0.2	10.5
Power (mw)	36.8	158.7

Table 4: Overhead of Phone Agent on Galaxy S5. (1)*Idle Wait*: waiting for offloading commands. (2)*Execution*: servicing offloading commands.

Table 5: RainDrops quickly provides feedback on app behavior w.r.t. different real-world contexts.

(a) Avg. object download time for five apps

	Beijing (ms)	Seattle (ms)
#1	228	514
#2	268	507
#3	283	549
#4	513	319
#5	463	876

(b) Normalized test duration for App #1

	Time (Normalized)
802.11n	*1*
3G	1.14
GPRS	2.59

is because asynchronous sensor callbacks frequently overlap with other Phone Agent activities.

5.3 Is Overhead from Phone Agent Feasible?

Ideally, Phone Agent should be light-weight to minimize any undesirable artifacts being introduced to testing. Table 4 shows its maximum resource overhead as measured on a Galaxy S5 smartphone. We organize measurements by the two states of Phone Agent. While empirical results show that Phone Agent consumes little resources, it is interesting that the memory consumption is higher than expected, especially during idle wait. This is because Phone Agent consumes some resources to (1) periodically process and stream sensor data that our testing app, RainTester, registers for, and (2) periodically send keep-alive messages to Coordinator.

5.4 Case Studies of App Problems Reported

Geo-location Factor. While a lot of apps depend on the geo-location to dynamically present contents (e.g., news and weather apps), adding location-aware code can introduce opportunities for errors, especially that developers cannot quickly get feedback from worldwide locations. One observation is that developers typically do not think about how the app responsiveness changes with the user location. Table 5(a) shows how the average object download time (msec) varies from Beijing to Seattle for five production apps. Interestingly, the download time is always longer in Seattle because the backend service has more Seattle-related contents available. Our findings have motivated developers to implement aggressive caching and background refresh.

Network Factor. Developers typically assume reasonably good network conditions between their apps and backend services. This assumption might not hold in the wild, and lossy or slow network conditions can impact apps that are not properly tuned for it. We illustrate this case study with app #1. Table 5(b) shows how one production app's (normalized) test session completion time changes under different network conditions. This feedback successfully help the app developers to adopt image resolution scaling to minimize the load time variance.

6. CONCLUSION

This paper takes the first step towards evolving automated app testing services to scalably achieve test coverage and realism. To this end, we demonstrate the potential of app split-execution, and show the feasibility through a narrow-waist architecture that combines emulators in the cloud and physical devices in the wild.

7. REFERENCES

[1] Androguard. https://code.google.com/p/androguard/.
[2] Appium. http://appium.io/.
[3] OpenSignal. https://opensignal.com/products/.
[4] Rely on Real Emulators vs Devices. http://testdroid.com/testdroid/5901/rely-only-on-real-emulators-vs-devices.
[5] Xposed. http://repo.xposed.info/.
[6] Applause. Applause. http://applause.com.
[7] Apple. TestFlight.
[8] Business Insider. iPhone Users Upgrade A Lot More Often Than Android Users, 2015.
[9] Centercode. Keep The Beta Tests Confidential. 2013.
[10] R. Chandra, B. Karlsson, N. D. Lane, C.-J. M. Liang, S. Nath, J. Padhye, L. R. Sivalingam, and F. Zhao. How to Smash the Next Billion Mobile App Bugs? *GetMobile*, 19, 2015.
[11] C. Curtsinger and E. D. Berger. COZ: Finding Code that Counts with Causal Profiling. In *ATC*. USENIX, 2016.
[12] Genymobile. Genymotion. http://genymotion.com.
[13] P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung. Vision: Automated Security Validation of Mobile Apps at App Markets. In *MCS*, 2011.
[14] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. RERAN: Timing- and Touch-Sensitive Record and Replay for Android. In *ICSE*. IEEE, 2013.
[15] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. MobiPlay: A Remote Execution Based Record-and-Replay Tool for Mobile Applications. In *ICSE*. ACM, 2016.
[16] Google. Firebase Test Lab for Android.
[17] P. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo. Understanding the Test Automation Culture of App Developers. In *IEEE*. ICST, 2015.
[18] LabUp! OpenDeviceLab.com - Access a Variety of Devices. Worldwide. For Free.
[19] C.-J. M. Liang, N. D. Lane, N. Brouwers, L. Zhang, B. F. Karlsson, H. Liu, Y. Liu, J. Tang, X. Shan, R. Chandra, and F. Zhao. Caiipa: Automated Large-scale Mobile App Testing through Contextual Fuzzing. In *MobiCom*. ACM, 2014.
[20] Microsoft. Windows Phone Emulator.
[21] Mikael Ricknas. Most Old Mobile Phones End Up in Drawers. Computer World, 2008.
[22] OpenSignal. Android Fragmentation Visualized, 2014.
[23] Perfecto Mobile. A Single Platform for Your Digital App Quality Needs. http://perfectomobile.com.
[24] Pew Research Center. Apps Permissions in the Google Play Store. http://www.pewinternet.org/2015/11/10/apps-permissions-in-the-google-play-store/, 2015.
[25] C. A. Ratanamahatana and E. Keogh. Everything You Know about Dynamic Time Warping is Wrong. In *KDD*. ACM, 2004.
[26] Sauce Labs. Sauce Labs. http://saucelabs.com.
[27] Testin Inc. Automated App Testing on Real Devices.
[28] Yahoo! Finance. New Study Finds $47 Billion Worth Of Cell Phones Gathering Dust, 2014.
[29] T. Yoshida, H. Yamada, and K. Knon. Using a Virtual Machine Monitor to Slow Down CPU Speed for Embedded Time-Sensitive Software Testing. In *ACS*. IPSJ, 2009.