On the Accuracy of Smartphone-based Mobile Network Measurement

Weichao Li, Ricky K. P. Mok, Daoyuan Wu, and Rocky K. C. Chang
Department of Computing
The Hong Kong Polytechnic University
Email: csweicli|cskpmok|csdwu|csrchang@comp.polyu.edu.hk

***Abstract*—As most of mobile apps rely on network connections for their operations, measuring and understanding the performance of mobile networks is becoming very important for end users and operators. Despite the availability of many measurement apps, their measurement accuracy has not received sufficient scrutiny. In this paper, we appraise the accuracy of smartphone-based network performance measurement using the Android platform and the network round-trip time as the metric. We use a multiple-sniffer testbed to overcome the challenge of obtaining a complete trace for acquiring the required timestamps. Our experiment results show that the RTTs measured by the apps are all inflated, ranging from a few milliseconds (ms) to tens of milliseconds. Moreover, the 95% confidence interval can be as high as 2.4ms. A finer-grained analysis reveals that the delay inflation can be introduced both in the Dalvik VM (DVM) and below the Linux kernel. The in-DVM overhead can be mitigated but the other cannot be. Finally, we propose and implement a native app which uses HTTP messages for network measurement, and the delay inflation can be kept under 5ms for almost all cases.**

I. Introduction

Mobile devices, notably smartphones and tablets, have already become essential parts of our daily lives because of their mobility and rich functionalities. Due to their limited computational power and storage, they rely on network access to offload intensive computation tasks to remote servers or cloud. Moreover, the offloading approach can save energy, thus extending the battery lifespan [18], [21]. Tongaonkar et al. find that 84% of apps require permission of Internet access [28] from a pool of 55K Android apps randomly picked from the official Android app market. Therefore, understanding mobile network performance is critical for providing good quality of experience to users. For example, recent performance studies characterize LTE networks [16] and optimize mobile application performance [29]. The data collected by `Speedtest.net` is used for comparing the performance between cellular and WiFi networks [26].

The importance of monitoring mobile network quality motivates a number of studies on network performance measurement. These measurement works are conducted on mobile devices using browsers or measurement apps. The browser-based measurement is similar to speedtest for desktop in that the measurement is conducted through mobile browsers [1], [6]. A more popular approach is using measurement apps on smartphones, such as [2], [3], [5], [9] for Android, [7], [8] for iOS, and [4], [10] for Windows Phone. In particular, the Ookla speedtest app [9] has recorded over 10 million downloads in the Android app market. These measurement apps can measure network round-trip time (RTT) and upload/download throughput. Some of them can even perform traceroute, measure DNS performance, and characterize HTTP caching behavior [3].

Despite the availability of many measurement apps, their measurement accuracy has not received sufficient scrutiny. In this paper, we appraise the accuracy of smartphone-based network performance measurement. We focus on the RTT measurement, because it is the most available atomic metric. Moreover, we consider only Android smartphones and the measurement-app approach. Similar studies for iOS and others will be our future work. We first identify three implementation models for measurement apps: *Native ping* (commands external to Java), *Inet ping* (using network-related classes in Java/Android with TCP SYN/RST packets), and *HTTP ping* (using HTTP-based Java classes with TCP data packets). For the purpose of evaluation, we develop a dedicated measurement app for each model.

A major challenge in measuring their accuracy is setting up a reliable testbed environment to obtain accurate timestamps when the packets are just sent out to and received from the air. Unlike fixed network measurement, a single sniffer is not able to capture all the packets because of frequent missing frames. By employing multiple sniffers, we are able to merge partial traces into an almost complete trace. The entire process requires us to resolve the synchronization issues for the smartphone and sniffers, recover the timestamps, and investigate the impact of clock skew between the smartphone and sniffers on the results.

We have conducted experiments on the testbed using three Android phones with different configurations installed with the three measurement apps. Although the experiments are conducted in a WiFi network, part of the results can also be applied to cellular network. Below is a summary of our findings which, to our best knowledge, have not been reported before.

1) (Highly inflated RTT measurement) The experiment results show that the RTTs measured by the apps are all inflated for all three phones, ranging from a few milliseconds to tens of milliseconds (ms), Moreover, the 95% confidence interval can be as high as 2.4ms. Although there is a wide range of latency performance in mobile networks, the cloud infrastructure continues

to help reduce the network latency for many apps. For example, the median RTT from University of Connecticut to Akamai-Hartford servers is only 8.5ms [12]. Therefore, the RTT inflation introduced by the measurement apps is too significant to ignore.

2) (Causes for the RTT inflation) By obtaining the timestamps when the probe and response packets transit in the Android kernel, we are able to conduct a finer-grained analysis on the inflated delay. Our analysis reveals that the delay inflation can occur both in the Dalvik VM (DVM) and below the kernel. We also identify that the delay inflation introduced by the DVM is asymmetric for packet sending and receiving, but it can be mitigated. However, another part of delay inflation that occurs between the kernel and hardware/driver cannot be easily evaded.
3) (Mitigating the RTT inflation) Based on the cause analysis, a promising approach to mitigating the delay inflation is to bypass the DVM. We therefore implement the core measurement logic into a native C program and invoke it through an external system call in the app. Experiment results show that the delay inflation can be kept under 5ms for most of the cases. Moreover, employing TCP data packets, instead of TCP control messages, as measurement probes, can further minimize the kernel-hardware overhead.

The remainder of the paper is organized as follows. In §II, we first introduce our approach to measuring the accuracy of three main methods in a testbed. In §III, we detail the different aspects of our testbed setup, including the use of multiple sniffers to obtain a complete trace for acquiring timestamp information. §IV reports the evaluation results obtained from the testbed. We then propose in §V a measurement method for mitigating the delay inflation. After highlighting the related works in §VI, we conclude the paper in §VII.

II. Our approach

Carrying out network measurement with mobile devices is much more challenging than the desktop environment, even though the core methodology is similar in both cases. A major difference is the operating system architecture. Android measurement apps usually run in a virtual machine. The apps could therefore encounter a larger system overhead in sending probe packets and receiving response packets, thus inflating the actual network delay. In this paper, we consider the accuracy of the network RTT measurement, because it is most available, and it can be used for obtaining other performance metrics, such as jitter, available bandwidth, and capacity.

A. Measuring the delay overhead

To evaluate the accuracy of Android measurement apps, we use the *delay overhead* defined in [20], which is the difference between the measured and the actual network delay. Considering a simple probe-response scenario in Fig. 1, a measurement app sends out a probe packet at time t_u^o to a web server (or other types of target). The probe packet elicits a response packet from the server, arriving at the measurement app at time t_u^i. The measurement app thus uses d_u (= $t_u^i - t_u^o$) as the network RTT. Obviously, this measured RTT is generally larger than the actual RTT d_n (= $t_n^i - t_n^o$), where t_n^o (t_n^i) is the time for the probe (response) packet to leave (arrive at) the smartphone. The delay overhead is therefore defined as

$$\Delta d = d_u - d_n = (t_u^i - t_u^o) - (t_n^i - t_n^o). \qquad (1)$$

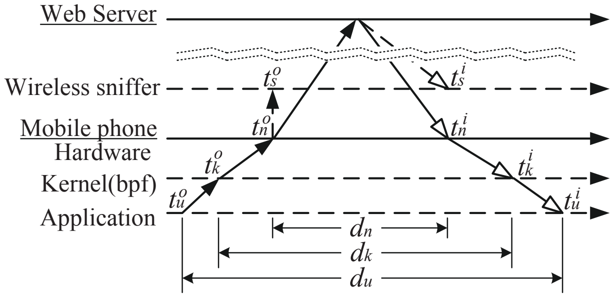


Fig. 1: Measurement flow for Android apps.

There are three possible factors contributing to the delay overhead: (i) the timing accuracy of the outgoing and receiving packets, (ii) the delay for Android to propagate the probes to the kernel and network stack, and the delay for delivering the responses to the app, and (iii) the delay for the hardware (wireless network adaptor) to send and receive packets.

For factor (i), Android provides several timing functions, such as `System.nanoTime()` and `System.currentTimeMillis()`. Although these two functions have different resolutions (ns vs. ms) and map to different POSIX functions `clock_gettime()` and `gettimeofday()`, they share the same back-end function `clock_gettime()` through vsyscall according to POSIX.1-2008 [27]. Giucastro tested the granularity and performance of the two functions on some Android phones, and found that the average cost for executing a such timing function is about 1μs [15]. Considering that the network delay is usually at ms level, the overhead of calling the timing functions is negligible.

We will therefore focus on the other two factors. To further quantify them, we also include two other timestamps t_k^i and t_k^o which are obtained when the packets are at the kernel. While we could obtain the kernel timestamps using `tcpdump` (see §III-A), it is much more challenging to obtain the two network timestamps t_n^o and t_n^i. In wired network, these two timestamps can be easily obtained by placing an external packet sniffer to capture the packets diverted from a network tap, because the fixed network is more reliable (i.e., the packets are seldom dropped by the sniffer), and the measurand and the sniffer can be easily time-synchronized.

In wireless network, a single wireless sniffer is not reliable enough to capture all the packets in the air (see §III-A). Using multiple sniffers, however, requires a careful trace merging and timestamp recovery. Moreover, as Android phones do not support PTP, synchronizing the clocks between the external sniffer and the phone is difficult. Another concern is due to the mechanism of FullMAC MLME (MAC Sublayer Management Entity) in which all 802.11 wireless frames are first transformed into IEEE 802.3 Ethernet frames before being delivered to the kernel. Such transformation could further increase the delay overhead. We will explain how we tackle these issues in §III.

B. Building measurement apps in Android

Android provides several interfaces or APIs that can be utilized for implementing a network measurement app without rooting the devices. We have studied the RTT measurement methods employed by a number of Android apps, such as MobiPerf [2], Netalyzr [3], and Ookla Speedtest [9], by inspecting their codes and the packets exchanged between the Android phone and servers. They can be classified into the following three methods.

Native ping. This method executes external shell commands through a Java `Runtime` class. A measurement app can directly invoke the `ping` program, which is located at a default location `/system/bin`, to perform ICMP-based RTT measurements. The `ping` program sends/receives the ICMP Echo messages on behalf of the measurement app and returns the measurement results. Although the `ping` program can only provide the resolution of 1ms or 0.1ms, it is the only way to handle ICMP packets without modifying the Android framework. Other than `ping` program, we find that executing any pre-compiled C program packaged with the app is also feasible.

Inet ping. The measurement app can also employ the network related classes provided by Android or Java. For example, the method `isReachable` of class `java.net.InetAddress` sends TCP SYN packets on port 7 (Echo) to a remote host[1] to elicit TCP SYN ACKs (if the port is open) or TCP RST packets. Therefore it can be utilized for implementing a TCP-based ping app. Besides, classes `java.net.Socket` and `java.net.DatagramPacket` can be used for respective TCP and UDP throughput tests.

HTTP ping. HTTP-based classes, such as class `java.net.HttpURLConnection`, can also be used for implementing a measurement app. Here the outgoing and incoming packets are complete HTTP request and response messages. Unlike Inet ping, any web server can serve as a remote destination without the need of setting up additional measurement servers. However, recording sending time after the TCP three-way handshake is required to avoid including the delay of connection establishment into the measurement.

To test the performance of these methods, we have developed a test app for each method. To minimize the workload of the test apps put on the phone, we compute all RTT estimates offline. For Native ping, the test app only parses and saves the output from the `ping` program without any further calculation. The test apps for Inet ping and HTTP ping employ the `InetAddress` and `HttpURLConnection` classes, respectively. We simply log the timestamps of packet sending and receiving events with the system time function `System.currentTimeMillis()` or `System.nanoTime()`. For HTTP ping, we limit the size of HTTP request/response to no larger than 300 bytes, so that each message can be sent in a single TCP packet.

III. A MULTIPLE-SNIFFER TESTBED

Fig. 2 shows the testbed which consists of a measurement server, which is equipped with a 1.86GHz Intel Core 2 Duo processor (E6320) and 2GB memory, and a IEEE 802.11g wireless AP, Netgear WNDR3800. We use three Android phones—Google Nexus 5, HTC One, and Sony Xperia J—to conduct the experiments. Their detailed hardware configurations and OS versions are listed in Table I. We choose these phones for their diverse hardware capability which may produce different results. The OS versions cover the latest 4.4 and two older versions. We have also rooted the phones, so that they can run the cross-compiled version of `tcpdump` through `adb` (Android Debug Bridge) and scripts.

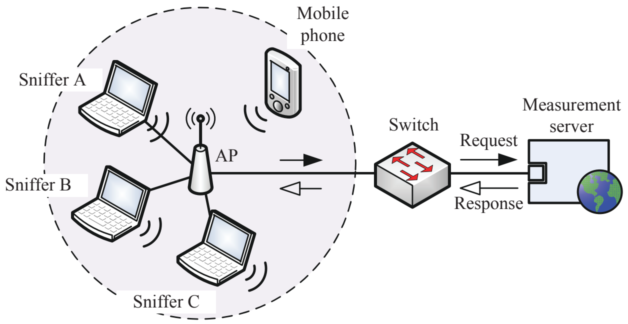


Fig. 2: The testbed setup where the packet sniffers, mobile phone, and wireless AP are placed within a distance of 0.5m.

TABLE I: The mobile phones used in the experiment.

Models	OS Ver.	Hardware specifications
Google Nexus 5	4.4.2	Qualia MSM8974 Snapdragon 800 CPU (quad-core 2.26GHz), 2GB MEM
HTC One	4.2.2	Qualia APQ8064T Snapdragon 600 CPU (quad-core 1.7GHz), 2GB MEM
Sony Xperia J	4.0.4	Qualia MSM7227A CPU (1GHz), 512M MEM

The three external packet sniffers are run on IBM T43 laptops running Ubuntu 12.04. We also wire-connect the sniffers to the AP, so that they can be controlled through SSH. We run the three test apps, each of which implements one of the three methodologies mentioned in §II-B, one by one on each phone. These apps send probes to the measurement server to elicit response packets and record the timestamps. We introduce an additional delay on the server side to simulate four different RTTs: 20ms, 50ms, 85ms, and 135ms. To avoid the RTT being affected by packet retransmission, we ensure there are no

[1] Although the official documentation says the method first tries ICMP and falls back to TCP when it fails, we find that the ICMP option has not been implemented.

probe losses during the measurement. Since wireless packet capturing could result in frame loss or duplication, we repeat each experiment for 100 times. Meanwhile, `tcpdump` is running in the background on the phone to obtain the kernel timestamps t_k^i and t_k^o. The impact of running `tcpdump` is negligible, because the traffic volume in each experiment is very small (<2 packet/s).

A. Wireless packet capturing

We use the packet capturing method described in [30]. We enable the monitor mode and promiscuous mode in the wireless network adaptors of the sniffers to capture the wireless frames (including the IEEE 802.11 header, physical layer header, and higher-layer protocols' information) using `tcpdump`. To simplify the decoding of wireless frames, we also disable the security options, such as WPA. We have not performed clock synchronization among the sniffers, because hardware timestamping is not supported and software timestamping cannot meet our requirement. We use the method to be described in §III-B to evade the clock drift offline.

We employ three sniffers, because a single sniffer will miss many packets [30], [24]. Although we put the AP, mobile phone, and the sniffers very close together (within a distance of 0.5m), we still find random frame losses and duplications in the captured traces. To ensure the completeness of a packet trace, Serrano et al. proposed to use multiple sniffers to merge the individual traces [24]. In our case, after merging the packet traces from the three sniffers, the trace completeness can reach to more than 99%.

B. Trace merging and time recovering

To merge the incomplete traces together, we first assign a trace as the main trace and others as reference traces. Then the missing frames in the main trace can be identified after comparing all traces. Finally, we insert the missing frames to the correct locations in the main trace and adjust their timestamps, so that they are coherent to the local frames. The most challenging part in this procedure is to accurately recover the timestamps of the missing frames. A most straightforward approach is to synchronize the sniffers, but the results do not meet our expectation. We therefore use reference frames (e.g., beacon frames) for "frame-level synchronization." However, considering the timestamp variation when a system reports its current time, simply performing linear translation between two reference frames [22] could lead to fitting errors. Accordingly, we employ a linear regression algorithm to take care of the time fluctuations and clock skews present in the sniffers.

Our algorithm first obtains the clock skews between each pair of the sniffers by applying linear regression to the beacon frames, as beacon frames are observed with the smallest time fluctuations by all sniffers. The clock skew is re-calculated for every set of data which is collected in around 180s. Our measurement results show that the clock skew progresses linearly during such a short period. When computing the timestamps of missing frames, we treat beacon frames as reference frames for the same reason.

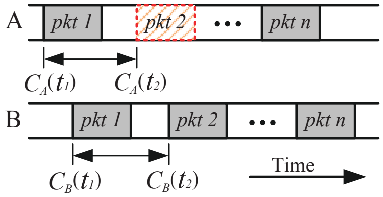


Fig. 3: Procedure of trace merging and time recovery.

Fig. 3 shows the procedure of how we recover the timestamp of a missing frame. Suppose that we want to recover a lost frame *pkt2* in the main trace A from the reference trace B. Let $C_A(t)$ and $C_B(t)$ be the times reported by sniffers A and B at time t. We denote the clock skew of A relative to B by $\delta_{\{A,B\}}(t) = C'_A(t) - C'_B(t)$, where $C'_A(t) \equiv dC_A(t)/dt$ and $C'_B(t) \equiv dC_B(t)/dt$, $\forall t \geq 0$. To recover the timestamp $C_A(t_2)$, we make use of the closest beacon frame as the reference frame, which is *pkt1* in Fig. 3, in both traces:

$$C_A(t_2) = C_A(t_1) + (C_B(t_2) - C_B(t_1)) + \int_{t_1}^{t_2} \delta_{\{A,B\}}(t)dt. \tag{2}$$

Since $\int_{t_1}^{t_2} \delta_{\{A,B\}}(t)dt = \delta_{\{A,B\}}(t) \times (t_2 - t_1)$, we have

$$C_A(t_2) = C_A(t_1) + (C_B(t_2) - C_B(t_1)) + \delta_{\{A,B\}}(t) \times (t_2 - t_1). \tag{3}$$

As the interval between two consecutive beacon frames is around 100ms, the missing frame is separated from the closest beacon frame by no more than 50ms. Given such short period of time and the typical clock skew for computer grade crystals, $t_2 - t_1$ can be replaced by $C_B(t_2) - C_B(t_1)$. Therefore, we can recover $C_A(t_2)$ with

$$C_A(t_2) \approx C_A(t_1) + (C_B(t_2) - C_B(t_1)) \times (1 + \delta_{\{A,B\}}(t)). \tag{4}$$

C. Clock skew handling

External sniffers and phones are also running different clocks. As t_s^o and t_s^i are measured from outside, we would like to know whether the RTTs estimated by the sniffers are comparable to the phones'. Similar to §III-B, let $C_p(t)$ and $C_s(t)$ be the times reported by the phone and sniffer at time t, and $\delta_{\{p,s\}}(t)$ the clock skew between the phone and the sniffer. For a time interval (t_1,t_2), the difference of the measured duration $\Delta D_{\{p,s\}}$ is

$$\begin{aligned} \Delta D_{\{p,s\}} &= (C_p(t_2) - C_p(t_1)) - (C_s(t_2) - C_s(t_1)). & (5) \\ &= \int_{t_1}^{t_2} \delta(t)dt. & (6) \end{aligned}$$

We have tested several Android phones and wireless sniffers. The clock skews among them are all within the range of ±100 ppm (parts per million). For an end-to-end network path, the RTT is usually tens to hundreds milliseconds [13]. Taking 100ms as an example, the measured RTT difference could be smaller than 10μs, which is small enough to ignore. Therefore, the delay overhead in Eqn. (1) can be computed by

$$\Delta d \approx (t_u^i - t_u^o) - (t_s^i - t_s^o). \tag{7}$$

IV. Evaluation

Table II presents the means and 95% confidence intervals of the delay overheads measured for the three test apps (methods) and four emulated RTTs. Compared with the RTTs observed by the external sniffers, the RTTs measured by the apps are inflated significantly for all three phones. The delay overheads can range from a few milliseconds to tens of milliseconds, and the 95% confidence interval can be as high as 2.4ms. The inflated RTT measurement is too significant to ignore, considering the network delay today is getting smaller due to the prevalence of CDNs and cloud services. For example, the median RTT from University of Connecticut to Akamai-Hartford servers is only 8.5ms [12].

TABLE II: Delay overheads measured when `System.currentTimeMillis()` is used (mean with 95% confidence interval, in ms).

	Phone*	Emulated RTT (ms)			
		20	50	85	135
Native ping	G	7.700 ±2.331	6.028 ±0.811	14.078 ±0.684	13.963 ±0.691
	H	2.108 ±0.726	1.177 ±0.292	5.179 ±0.564	0.849 ±0.281
	S	6.779 ±1.129	7.840 ±0.932	9.999 ±1.039	8.387 ±1.191
Inet ping	G	11.931 ±1.063	12.514 ±0.779	16.211 ±0.833	15.874 ±0.787
	H	7.243 ±1.907	7.470 ±0.815	8.551 ±2.413	7.060 ±0.821
	S	13.822 ±1.327	12.223 ±1.142	12.814 ±1.146	12.511 ±1.055
HTTP ping	G	6.481 ±0.855	7.651 ±0.963	9.156 ±0.703	10.790 ±0.911
	H	6.566 ±0.588	7.151 ±0.957	7.222 ±1.041	6.675 ±0.739
	S	11.206 ±0.947	11.153 ±0.855	11.805 ±0.987	12.987 ±1.312

Note ∗: G for Google Nexus 5, H for HTC One, and S for Sony Xperia J.

A. Overview

As summarized in Table II, each test app (method) suffers from high delay inflation during the RTT measurement. To investigate the distribution of the delay overheads, we plot the probability densities of the nine sets of measurement results in Fig. 4. Each subplot includes the measurements by the four emulated RTT cases for one app (method) and one phone.

1) Native ping: For phone G, two different patterns can be observed. When the emulated RTTs are 20ms and 50ms (short RTTs), the distributions of the delay overheads almost coincide, and the peak is at around 4.8ms. But when the RTT increases to 85ms and 135ms (long RTTs), most of the delay overheads occur at ∼16ms. Phone S also has two different distribution patterns: one for 20ms and the other for other three cases. The delay overheads for the 20ms case are mainly located at ∼2.4ms, but it spreads over a larger range (from 3ms to 14ms) than the other three cases. The delay overheads for phone H is unexpectedly small (concentrated at 0.5∼0.7ms), which is the only case that the delay overheads are smaller than 1ms. Interestingly, delay *deflation* can be observed for all three phones. We conjecture that it is due to the coarse resolution of measurement results that `ping` program can provide.

2) Inet ping: Similar to Native ping, phone G also has two different patterns. The overheads are concentrated at ∼14ms for the cases of 20ms and 50ms, and ∼18ms for the other two. For phone H, a bimodal distribution can be observed for all four cases with peak values at 3.5ms and 9ms. At the first glance, phone S has a more complicated distribution, but in fact it consists of two types of bimodal distribution: peak values of 8.5ms and 18ms for short RTTs, and peak values of 6ms and 13.8ms for long RTTs.

3) HTTP ping: Both phones H and S have relatively more consistent delay overheads, most of which occur at 5.8ms or 10ms. However, there are still two patterns for phone G: a bimodal distribution for short RTTs and a unimodal distribution for long RTTs.

To sum up, the same Android phone has different performance for different measurement methods. Although it is hard to say which method is the best, HTTP ping and Native ping exhibit comparatively smaller delay overheads for most of the cases. By comparing the results from Inet ping and HTTP ping which use TCP SYN/RST packets and TCP data packets, respectively, we find that establishing a new TCP connection may incur a higher delay overhead than processing content in an existing connection. Generally speaking, most of the delay overheads are observed as RTT-independent, except for phone G for which the delay overheads correlate with the RTTs: smaller (larger) overhead for short (long) RTTs.

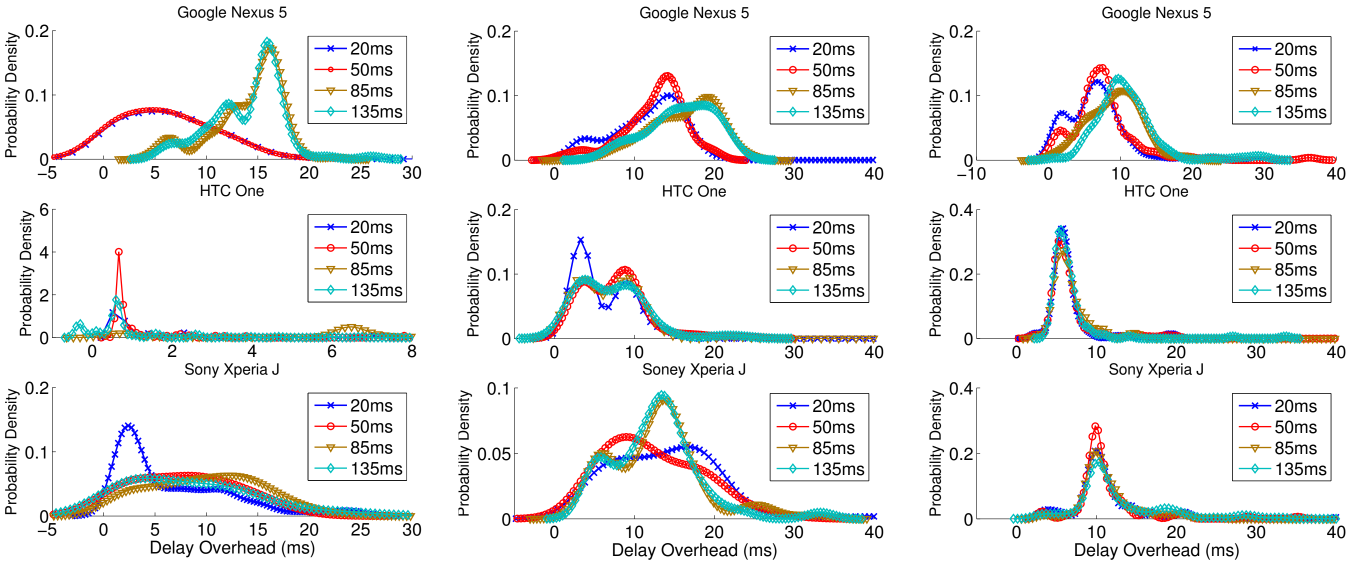
B. Effect of timing functions

The results presented in Table II and Fig. 4 are measured when function `System.currentTimeMillis()` is used. Since it is reported that this function could have coarse granularity (such as ∼15ms) in some OS [20], we also implement the test apps with the more precise `System.nanoTime()` for the purpose of comparison. We perform experiments with the same setting described in §III, and link the results together with the ones that obtained by `System.currentTimeMillis()`. To better visualize the effect of the two timing functions, we use box plots to present the data in Fig. 5. In each box-and-whisker plot, the top and bottom of the box are given by the 75th and 25th percentile, and the mark inside is the median. The upper and lower whiskers are the maximum and minimum, respectively, after excluding the outliers. The outliers above the upper whiskers are those exceeding 1.5 of the upper quartile, and those below the minimum are less than 1.5 of the lower quartile.

We only present the data of phone G in detail, since the other two phones have the similar results. The figures show that the delay overheads measured by `System.nanoTime()` is similar to those by `System.currentTimeMillis()`. Considering the relatively large delay inflation, the overhead of executing a timing function is therefore not a key factor to consider for measurement accuracy.

C. Explaining the delay overheads

To locate where the overheads are introduced, we dissect the round-trip delay overheads into several components. Back

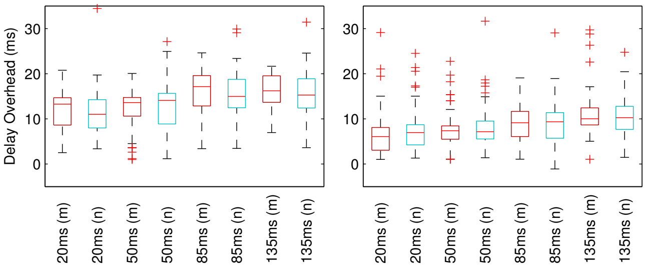


(a) Native ping.

(b) Inet ping.

(c) HTTP ping.

Fig. 4: PDF plots of the delay overheads (by apps and phones).



(a) Inet ping.

(b) HTTP ping.

Fig. 5: Delay overhead comparison in box plot for phone G (red/m for `System.currentTimeMillis()`, and cyan/n for `System.nanoTime()`).

to the packet sending and receiving processes in Fig. 1, a packet needs to be delivered to the Linux kernel before it reaches the network (for the outgoing direction) or the app (for the incoming direction). Supposing that the outgoing and incoming packets arrive at the kernel at times t_k^o and t_k^i respectively, we then calculate the *kernel delay overhead* Δd_k occurred between the kernel and air medium:

$$\Delta d_k = d_k - d_n = (t_k^i - t_k^o) - (t_n^i - t_n^o). \qquad (8)$$

Similarly, the *app delay overhead* Δd_u that takes place between the app and kernel can be computed as

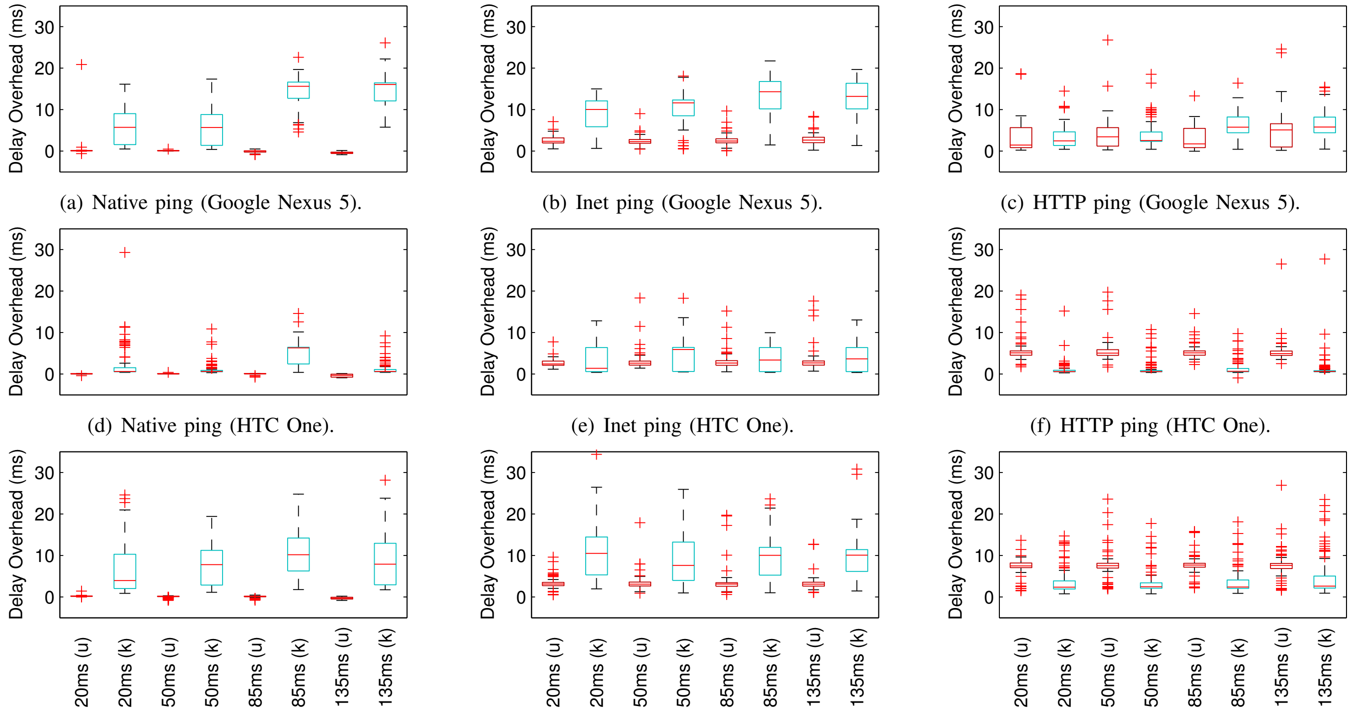
$$\Delta d_u = d_u - d_k = (t_u^i - t_u^o) - (t_k^i - t_k^o). \qquad (9)$$

By calculating these two types of delay overheads, we can identify the place where the delay overheads are introduced. Note that the two overhead components are independent, because the magnitude of Δd_k depends on the performance of hardware/driver of the network interface, whereas Δd_u the performance of the Android system. Although our evaluation in this paper is based on IEEE 802.11g network, the analysis of Δd_u is still valid when the mobile network is changed to others, such as HSPA and LTE.

As described in §III, during our previous experiments, we also run `tcpdump` in the background on those three test phones, which allows us to obtain t_k^o and t_k^i in the kernel space with `bpf` and `libpcap`. We then calculate and plot the two types of delay overheads in box plot in Fig. 6. Figs. 6(a), 6(d), and 6(g) for Native ping clearly show that Δd_u for all three phones is very close to 0, suggesting that the packets are mainly delayed between the kernel and physical link. Similar to Native ping, Δd_k for Inet ping contributes the majority of the total delay overheads, as shown in Fig. 6(b), 6(d), and 6(f), except that the layer above the kernel space adds 2ms to 4ms more delay. As for HTTP ping, phones H and S experience much larger Δd_u than Δd_k (around 5.5ms vs. 0.6ms and 7.5ms vs. 2.5ms, respectively, as shown in Figs. 6(f) and 6(i)). In particular, phone G has a totally different pattern from H and S, having Δd_u relatively close to Δd_k. Only when the network RTT increases to 85ms and 135ms will Δd_k overtake Δd_u.

Fig. 6 also indicates that the inconstancy of the delay overheads (Δd) is chiefly caused by Δd_k. Since the network adaptor on the phone cannot explicitly support hardware timer or SoftMAC, we are unable to further decompose Δd_k without hacking the firmware. Therefore, it is not possible to identify whether the delay is introduced by the driver or hardware. What we can only say is that the physical link or driver react slower to ICMP and TCP SYN/RST messages than TCP data packets.

Our analysis shows that Native ping introduces nearly no overhead to the application but Inet ping and HTTP ping will. Note that the major difference between Native ping and the others is the measurement execution manner: external system call vs. in app. In the external system call, the external `ping`



(a) Native ping (Google Nexus 5).

(b) Inet ping (Google Nexus 5).

(c) HTTP ping (Google Nexus 5).

(d) Native ping (HTC One).

(e) Inet ping (HTC One).

(f) HTTP ping (HTC One).

(g) Native ping (Sony Xperia J).

(h) Inet ping (Sony Xperia J).

(i) HTTP ping (Sony Xperia J).

Fig. 6: Box plots for the app delay overheads (Δd_u, red) and kernel delay overheads (Δd_k, cyan).

runs as a native Linux program, whereas the app in the in-app approach is implemented in Java APIs and runs as an instance of the DVM. In fact, invoking a Java API usually involves several more function calls. For each additional call, DVM needs to consume more bytecode instructions (e.g., pushing parameters into virtual registers). Moreover, network-related Java APIs are finally mapped to the bionic C library, which is equivalent to the BSD's standard C library, through Java Native Interface (JNI). Due to the extra translation, JNI could also lower the performance. Therefore, performing network measurement within an app could result in more delay than a native Linux program.

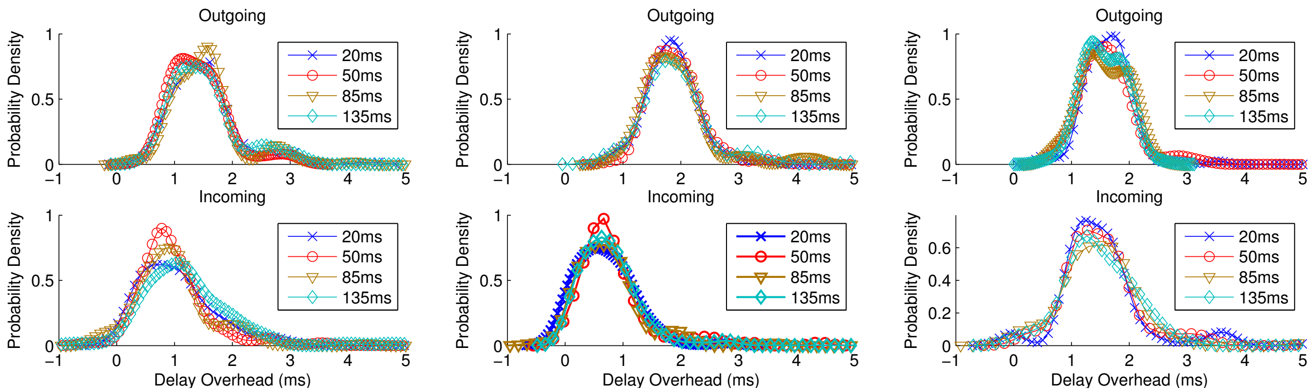
Another noticeable observation is that for Inet ping and HTTP ping Δd_u slightly increases from phone G to phone H and from phone H to phone J, which is the same ranking of the hardware performance of the phones. As the measurement operation for the two apps takes place in the DVM, we can infer that a phone with more powerful computation capability generally has a smaller Δd_u.

D. Incoming vs. outgoing

Running `tcpdump` also allows us to analyze the (a)symmetry of the delay overheads occurring in the app. Since Android uses the same clock source of the underlying Linux system, the timestamps recorded by the measurement apps and `tcpdump` are comparable. Therefore, we can measure the outgoing app delay overhead $\Delta d_u^o = t_k^o - t_u^o$, and the incoming delay overhead $\Delta d_u^i = t_u^i - t_k^i$. We plot the probability density of the overheads per direction in Fig. 7 for Inet ping and in Fig. 8 for HTTP ping. Note that we cannot analyze Native ping, because the external `ping` program does not provide the packet send and receive times.

Both Fig. 7 and Fig. 8 show significant delay asymmetry. For example, for Inet ping, establishing a TCP connection costs more time in the outgoing direction. The disparity can be larger than 1ms for phone H. On the other hand, the majority part of the app delay overhead occurs when receiving and processing the HTTP messages for HTTP ping. Altogether, the bidirectional delay overheads follow a unimodal distribution. The only exception is phone G, which follows a bimodal distribution in the incoming direction for HTTP ping (as shown in Fig. 8(a)).

We believe the asymmetry of Δd_u is caused by the interpretation execution nature of DVM. For example, the method `isReachable` of class `java.net.InetAddress` used in Inet ping involves several functions defined in `libcore/io/IoBridge.java`, such as `IoBridge.socket()`, `IoBridge.bind()`, and `IoBridge.connect()`, which can be further traced to three native functions (`socket()`, `bind()`, and `connect()`) in `libcore/io/Posix.java`. By analyzing the source code of the Android framework, we can find that the native functions are bridged to the functions defined in `libcore_io_Posix.cpp` and finally mapped to the functions defined in the bionic libc library (`sys/socket.h`). To sum up, after recording the sending time t_u^o in the app, a series of function calls are executed, and eventually the kernel sends out a TCP SYN packet at time t_k^o, which is captured by `tcpdump`. However, for the reverse

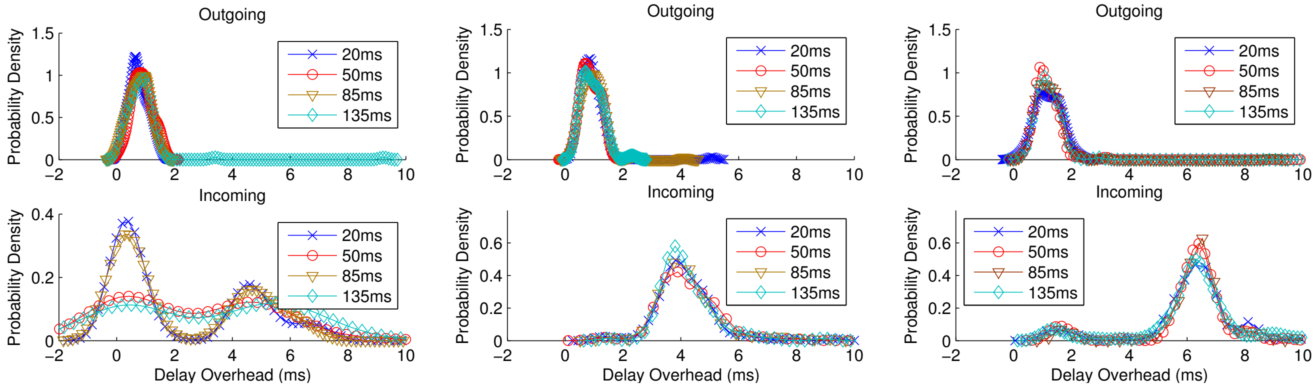


(a) Google Nexus 5.

(b) HTC One.

(c) Sony Xperia J.

Fig. 7: PDF plots of the delay overhead asymmetry (Inet ping).



(a) Google Nexus 5.

(b) HTC One.

(c) Sony Xperia J.

Fig. 8: PDF plots of the delay overhead asymmetry (HTTP ping).

direction, when the kernel receives the TCP RST packet, it just needs to notify the DVM without any further operations, thus incurring less overhead in the incoming path.

V. A Better Practice

An important observation gleaned from the analysis of the overhead components in the last section is the negligible Δd_u for Native ping. To validate whether bypassing the DVM can mitigate the delay overhead, we implement a simple C socket program which supports RTT measurements with TCP SYN/RST packets and HTTP GET request/response messages. Similar to HTTP ping, we limit the size of HTTP messages to no more than 300 bytes, so that each message can be transmitted in one TCP packet. We employ `clock_gettime()` to record the send and receive timestamps. After cross-compilation, the executable binary is packed into a test app, called *External ping*. This app can invoke the binary through the Java class `Runtime`. We test the app with the same settings described in §III and compute Δd_u based on Eqn. (9).

We compare Δd_u measured by External ping to the other two in-DVM apps in Table III. We only present the results obtained by phone G, because the other two phones have similar characteristics. As expected, Δd_u drops after employing the external system call, with a decrease of 1.6ms $\sim$ 2.2ms for Inet ping and 1.9ms $\sim$ 3.2ms for HTTP ping. Besides, the overheads are more stable with the confidence intervals smaller than 0.2ms. We also find that the HTTP ping introduces 0.4ms $\sim$ 0.5ms more delay than Inet ping. The additional delay is due to the fact that HTTP messages need to be further processed in the user space, but handling TCP SYN/RST packets can be completed within the kernel.

TABLE III: A comparison of Δd_u (Ext) for external C socket program and in-DVM measurement (App) (mean with 95% confidence interval, in ms).

	Type	Emulated RTT (ms)			
		20	50	85	135
Inet ping	App	2.946 ±0.695	2.443 ±0.200	2.637 ±0.251	2.828 ±0.236
	Ext	0.736 ±0.121	0.794 ±0.139	0.798 ±0.154	0.830 ±0.134
HTTP ping	App	3.312 ±0.663	3.824 ±0.721	3.157 ±0.540	4.542 ±0.834
	Ext	1.095 ±0.075	1.246 ±0.098	1.289 ±0.112	1.365 ±0.186

We advocate using HTTP messages for the best practice measurement on Android, because the kernel delay overheads for handling HTTP messages are more stable than for TCP control messages (as shown in Fig. 6). Our modification of External ping does not require root privilege, thus facilitating a wide deployment of the app. By repeating the measurements and computing the mean or median, the delay overhead can be kept within 5ms or even lower. Although External ping cannot completely remove the delay overhead, the measurement results it produces are much closer to the real network RTTs.

VI. Related works

A most closely related work is [20], which appraised the accuracy of browser-based measurement methods in fixed network. However, their methodology cannot be applied straightly to the mobile network measurement. Other measurement studies based on smartphones users include [14], [16], [17], [25]. In particular, a simple logger was employed in [14] to collect the network usage information from Android and Windows Mobile users, whereas LiveLab [25] measured wireless networks in iOS. In [16] and [17], the performance of 4G LTE and 3G networks was evaluated using `4GTest` and `3GTest`. These existing apps are designed with more concern on privacy issues or energy consumption, but their accuracy has not received any attention.

In the system performance area, several studies evaluated the performance of JNI or DVM. For example, Oh et al. investigated the performance impact of DVM on Android apps [23]. Batyuk et al. compared the performance between native C and Java applications for identical tasks [11], and showed that native C applications can be up to 30 times faster than running Java in DVM. But their work drew conclusions from Android emulator and Linux x86 platform. Lee and Jeon also carried out similar study for five algorithms [19] and found that JNI communication delays were about 0.15ms. These works mainly focused on the performance comparison of specific algorithms but do not study the relationship between system delay and network delay measurement.

VII. Conclusions

In this paper, we appraised the accuracy of measurement apps in Android phones. We overcame the main challenge of obtaining accurate packet timestamps from the wireless medium and setup a reliable wireless testbed. We found that the RTTs measured by the apps are significantly inflated. After conducting careful investigations, we identified the delay overhead introduced by the DVM is not negligible and symmetric in the send and receive directions. Finally, we proposed to mitigate the delay overhead by implementing a native measurement app using HTTP messages for measurement. The results showed that the delay overhead can be reduced to less than 5ms. We believe the improvement can provide more accurate understanding about the real network status for those who care more about the network quality than the user-perceived performance. In the future, we plan to extend our work to other mobile platforms, such as iOS and Windows Phone.

Acknowledgment

We thank all four anonymous reviewers for their valuable comments. This work is partially supported by an ITSP Tier-2 project grant (ref. no. GHP/027/11) from the Innovation Technology Fund in Hong Kong.

References

[1] Mobile Speed Test.com. http://www.mobilespeedtest.com/.

[2] MobiPerf on Google Play. https://play.google.com/store/apps/details?id=com.mobiperf.

[3] Netalyzr on Google Play. https://play.google.com/store/apps/details?id=edu.berkeley.icsi.netalyzr.android.

[4] Network Speed Test on Windows Store. http://www.windowsphone.com/en-us/store/app/network-speed-test/9b9ae06b-2961-41ef-987d-b09567cffe70.

[5] SpeedChecker on Google Play. https://play.google.com/store/apps/details?id=uk.co.broadbandspeedchecker.

[6] SpeedOf.Me Lite. http://speedof.me/m/.

[7] Speedtest X HD WiFi & Mobile Speed Test on App Store. https://itunes.apple.com/us/app/speedtest-x-hd-wifi-mobile/id366593092.

[8] Speedtest.net on App Store. https://itunes.apple.com/us/app/speedtest.net-mobile-speed/id300704847.

[9] Speedtest.net on Google Play. https://play.google.com/store/apps/details?id=org.zwanoo.android.speedtest.

[10] Speedtest.net on Windows Store. http://www.windowsphone.com/en-us/store/app/speedtest-net/4fcd4de1-050b-44dc-b123-a786808eb49b.

[11] L. Batyuk, A.-D. Schmidt, H.-G. Schmidt, A. Camtepe, and S. Albayrak. Developing and benchmarking native Linux applications on Android. In *Proc. Mobilware*, 2009.

[12] X. Chen, R. Jin, K. Suh, B. Wang, and W. Wei. Network performance of smart mobile handhelds in a university campus WiFi network. In *Proc. ACM/USENIX IMC*, 2012.

[13] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin. A first look at traffic on smartphones. In *Proc. ACM/USENIX IMC*, 2010.

[14] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin. Diversity in smartphone usage. In *Proc. ACM MobiSys*, 2010.

[15] S. Giucastro. Getting high precision timing on Android. http://www.gamasutra.com/view/feature/171774/getting_high_precision_timing_on_.php.

[16] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A close examination of performance and power characteristics of 4G LTE networks. In *Proc. ACM MobiSys*, 2012.

[17] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl. Anatomizing application performance differences on smartphones. In *Proc. ACM MobiSys*, 2010.

[18] K. Lee, J. Lee, Y. Yi, I. Rhee, and S. Chong. Mobile data offloading: How much can WiFi deliver? In *Proc. ACM CoNEXT*, 2010.

[19] S. Lee and J. W. Jeon. Evaluating performance of Android platform using native C for embedded systems. In *Proc. IEEE ICCAS*, 2010.

[20] W. Li, R. Mok, R. Chang, and W. Fok. Appraising the delay accuracy in browser-based network measurement. In *Proc. ACM/USENIX IMC*, 2013.

[21] Z. Li, C. Wang, and R. Xu. Computation offloading to save energy on handheld devices: A partition scheme. In *Proc. ACM CASES*, 2001.

[22] R. Mahajan, M. Rodrig, D. Wetherall, and J. Zahorjan. Analyzing the MAC-level behavior of wireless networks in the wild. In *Proc. ACM SIGCOMM*, 2006.

[23] H.-S. Oh, B.-J. Kim, H.-K. Choi, and S.-M. Moon. Evaluation of Android Dalvik virtual machine. In *Proc. JTRES*, 2012.

[24] P. Serrano, M. Zink, and J. Kurose. Assessing the fidelity of COTS 802.11 sniffers. In *Proc. IEEE INFOCOM*, 2009.

[25] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum. Livelab: Measuring wireless networks and smartphone users in the field. *SIGMETRICS Perform. Eval. Rev.*, 38(3):15–20, Jan. 2011.

[26] J. Sommers and P. Barford. Cell vs. WiFi: On the performance of metro area mobile connections. In *Proc. ACM/USENIX IMC*, 2012.

[27] The IEEE and The Open Group. IEEE Std 1003.1-2008. http://pubs.opengroup.org/onlinepubs/9699919799/.

[28] A. Tongaonkar, S. Dai, A. Nucci, and D. Song. Understanding mobile app usage patterns using in-app advertisements. In *Proc. PAM*, 2013.

[29] Q. Xu. *Optimizing Mobile Application Performance through Network Infrastructure Aware Adaptation*. PhD thesis, University of Michigan, 2013.

[30] J. Yeo, M. Youssef, and A. Agrawala. A framework for wireless LAN monitoring and its applications. In *Proc. ACM WiSe*, 2004.