Demystifying and Puncturing the Inflated Delay in Smartphone-based WiFi Network Measurement

Weichao Li[1], Daoyuan Wu[2], Rocky K. C. Chang[1], and Ricky K. P. Mok[3] †
[1]The Hong Kong Polytechnic University, [2]Singapore Management University, [3]CAIDA/UCSD
{csweicli|csrchang}@comp.polyu.edu.hk[1], dywu.2015@smu.edu.sg[2],
cs.rickymok@connect.polyu.hk[3]

ABSTRACT

Using network measurement apps has become a very effective approach to crowdsourcing WiFi network performance data. However, these apps usually measure the *user-level* performance metrics instead of the *network-level* performance which is important for diagnosing performance problems. In this paper we report for the first time that a major source of measurement noises comes from the periodical SDIO (Secure Digital Input Output) bus sleep inside the phone. The additional latency introduced by SDIO and Power Saving Mode can inflate and unstablize network delay measurement significantly. We carefully design and implement a scheme to wake up the phone for delay measurement by sending just enough warm-up and background traffic. Our evaluation results show that the overall median delay overheads can be kept within 3ms, regardless of the actual network delay.

CCS Concepts

•Networks → Network measurement; Mobile networks; Wireless local area networks;

Keywords

Network measurement; Accuracy; WiFi; Android

1. INTRODUCTION

To manage and operate a reliable WiFi network, it is necessary to monitor the WiFi network performance, such as network delay, effectively and accurately. However, network delay cannot be obtained directly from the common wireless access points (APs). Moreover, existing passive monitoring methods, including sniffer-based methods [40, 14, 25] and methods based on AP hacking [33, 28, 29], are not flexible and scalable to deploy.

A promising approach to monitor mobile network performance is to use smartphone apps which have gained popularity in recent years. These speedtest-like services are available on Android [1, 2, 10], iOS [8, 9], and Windows Phone [7, 11]. Besides, the research community has deployed measurement apps (e.g., Netalyzr [6], MobiPerf [4], and the recent MopEye [5]) to analyze and diagnose various network performance issues [35, 36, 26, 38].

These measurement apps usually measure user-level performance, because they are implemented in the user space. While they are useful for characterizing the user experience, they cannot be used to reliably infer the network-level performance which is important for many reasons, such as for operators to know their network performance and for users to diagnose whether the network or their phone is responsible for performance degradation. As shown in [23], the network round-trip times (nRTTs) measured by apps in WiFi networks are all inflated, and the amount of inflation varies across smartphone models, measurement methods, and the actual nRTT. As CDN and cloud services continue to reduce the end-to-end delay, this delay inflation will significantly over-estimate the actual nRTT. Although it is shown that part of the delay inflation can be mitigated, the remaining overhead, which is not negligible, is still not understood.

In this paper, as the first main contribution, we discover and demonstrate that the energy-saving mechanisms—the Secure Digital Input Output (SDIO) bus sleeping and the IEEE 802.11 Power Saving Mode (PSM)—are the main sources of the delay inflation. Specifically, the SDIO bus sleeping inflates the nRTT internally, whereas the PSM externally, between the phone and AP. To the best of our knowledge, this paper is the first to report the effect of SDIO bus sleeping on the nRTT measurement. Our analysis also shows that the delay inflation is dependent on the WiFi chipset utilized by the smartphone. Therefore, two different smartphones may obtain quite different nRTTs for same network path.

In our second contribution, we propose to mitigate the

†This work was done when the author worked at the Hong Kong Polytechnic University.

We thank Dr. Dan Pei for shepherding our paper and the anonymous reviewers for their valuable comments. This work is partially supported by a grant (ref. no. G-YBAK) from The Hong Kong Polytechnic University and a grant (ref. no. H-ZL17) from the Joint Universities Computer Centre of Hong Kong.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CoNEXT '16, December 12-15, 2016, Irvine, CA, USA
© 2016 ACM. ISBN 978-1-4503-4292-6/16/12. . . $15.00
DOI: http://dx.doi.org/10.1145/2999572.2999595

impact of the energy-saving mechanisms by enforcing the smartphones to operate in the wake-up mode during the delay measurement. Our approach is the first to mitigate the effect of SDIO bus sleeping. Although the impact of PSM have been studied by many works [13, 17, 30, 31], their focus is more on scheduling the packets on the AP side to achieve a better balance between energy consumption and network delay. Sui et al. [34] propose `ping2` to measure the network delay from the server side and try to avoid the additional delays introduced by the energy-saving mechanisms. However, `ping2` can be used only for network paths with short nRTT and cannot remove the inflations completely, because, when nRTT is long, the device could fall back to the inactive state again before it receives the response packet and starts the second ping.

We have implemented our approach in AcuteMon, an Android app prototype run on unrooted phones and requiring no system customization, such as kernel recompilation and customized ROM. We carefully design AcuteMon, so that the packets sent out to keep the phone in the wake-up state is minimal. We have conducted testbed experiments to validate our approach using five smartphones equipped with different WiFi chipsets and Android systems. The median additional delay is kept within 3ms for most of the tests, regardless of the actual nRTTs, and the delay variations are smaller than 3ms. To the best of our knowledge, this is the most accurate delay measurement that can be achieved on Android phones to this date without hacking the underlying system.

In §2 next, we first introduce our approach to evaluate the accuracy of the nRTT measurement. In §3, we illustrate the delay inflation problems caused by SDIO and PSM, and perform root-cause analysis. We describe AcuteMon in §4 and evaluate its performance. After highlighting the related works in §5, we conclude this paper in §6.

2. BACKGROUND

2.1 A model for nRTT measurement

The nRTT measurement can be inflated inside or outside the phones. For the internal factors, we use *delay overhead*, denoted by Δd, to evaluate the (in)accuracy of delay measurement [22, 23]. As shown in Fig. 1, the reported *user-level RTT* is computed by $d_u = t_u^i - t_u^o$, where t_u^o is the send time of the probe packet and t_u^i is the receive time of the response packet at the measurement app. However, due to various processing layers in the system, they are delivered to the air at t_n^o and received at t_n^i. Therefore, the delay overhead is computed as

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

Here we do not consider the processing delay introduced by the server, because the additional delay is in microsecond level for TCP data packets [24]. Since the delay overhead is accumulated at various layers of the system, it is necessary to record the timestamp at each vantage point to understand the sources of the overhead. Besides the timestamps reported by the app (t_u^o and t_u^i), Fig. 1 also shows the timestamps

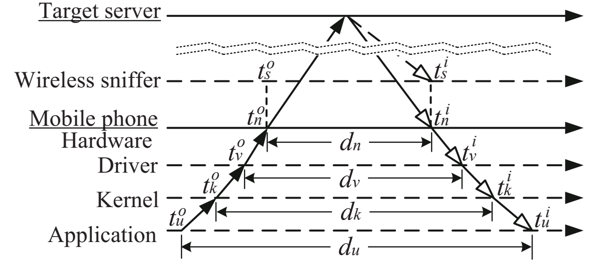


Figure 1: Measurement flow for Android apps.

in kernel space (t_k^o, t_k^i), WNIC (wireless NIC) driver (t_v^o, t_v^i), and network adapter (t_n^o, t_n^i). The kernel timestamps can be recorded with `bpf` and `libpcap`. Since the system usually does not provide any API to obtain the timestamps in the driver, we modify the driver to obtain them (see §3.2.1). As for t_n^o and t_n^i, we can estimate them through external wireless sniffers for WiFi networks [23]. In addition to d_u and d_n, we define *kernel-level RTT* as $d_k = t_k^i - t_k^o$ and *driver-level RTT* as $d_v = t_v^i - t_v^o$. To clarify where and how the delay inflation occurs, we also break down the overall delay overhead into three types of overheads:

- *User-kernel overhead*: $\Delta d_{u-k} = d_u - d_k$.
- *Kernel-driver overhead*: $\Delta d_{k-v} = d_k - d_v$.
- *Driver-phy overhead*: $\Delta d_{v-n} = d_v - d_n$.

Similarly, we can define *kernel-phy overhead* (Δd_{k-n}) as the difference between d_k and d_n ($= d_k - d_n$) or the sum of Δd_{k-v} and Δd_{v-n}.

Based on [23] and others, Δd_{u-k} is now better understood and can be mitigated. In particular, the overhead incurred in the DVM (i.e., Δd_{u-k}) can be mitigated by executing a pre-compiled native C program. In this paper, we focus on understanding and mitigating the inflated delay between PHY and the kernel space, because mitigating only the user-kernel overhead is not sufficient to obtain accurate nRTT.

Besides, the IEEE 802.11 PSM (see §3.2.2) can inflate the nRTT between an AP and smartphone, because the AP does not always send the received packets to the smartphone immediately. Although this additional delay looks like an external factor, we still treat it delay overhead, mainly because this inflation occurs only when the device is in dormancy and can vary across different smartphone models and at different times. Therefore, excluding this PSM-induced delay can facilitate a more accurate and stable nRTT.

2.2 A multiple-sniffer testbed in WiFi network

Our testbed, as shown in Figure 2, consists of a measurement server, which is equipped with a 1.86GHz Intel Core 2 Duo processor (E6320) and 2GB memory, and an IEEE 802.11g compatible wireless AP, NETGEAR WNDR3800. In §4.3 and §4.4, a load generator and a load server are added to the testbed to generate cross traffic. The three sniffers are wire-connected to the measurement server through the AP and a switch. The sniffers are three desktop PCs equipped with Intel 7260 WNIC. In particular, the smartphone is connected to one of the sniffer via USB cable, so that `adb` (Android Debug Bridge) and automation scripts can be run on the phone.

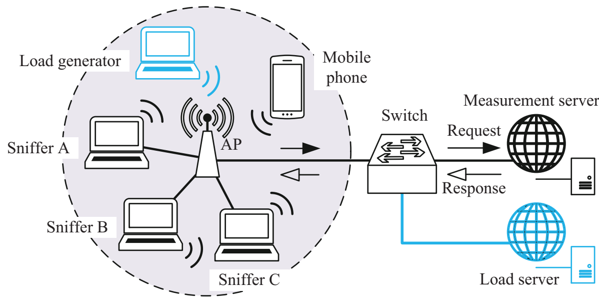


Figure 2: The testbed setup where the packet sniffers, smartphone, and wireless AP are placed within a distance of 0.5m.

3. DELAY OVERHEAD CAUSED BY ENERGY-SAVING MECHANISMS

3.1 Effect of packet sending interval

Our root-cause analysis begins with an ICMP ping experiment conducted in the testbed described in §2.2. We run a `ping` program through `adb shell` for 100 times with two packet sending intervals, a small interval of 10ms and larger default of 1s, measuring the nRTTs between the smartphones under test and the measurement server. In this paper, we test five different smartphones in Table 1. For the root-cause analysis, we use only Google Nexus 4 and Nexus 5, because they employ the WNIC chipsets manufactured by Qualcomm (WCN 3660) and Broadcom (BCM 4339), respectively. As most smartphones employ the WNIC chipsets provided by these two manufacturers [32], and these chipsets usually share the same source code for the same manufacturer, any delay overhead caused by these chipsets and drivers could be captured by these two phones. Given the fact that the average latency of broadband service for each monitored ISP in US ranges from 14ms to 52ms [16], we set the nRTT to 30ms and 60ms with `tc` command on the server side to emulate the real Internet environment.

Table 1: The smartphones used in the testbed evaluation.

Models	Ver.	CPU (core)	RAM	WNIC†
Google Nexus 5	4.4.2	2.26GHz (4)	2GB	BCM4339
Google Nexus 4	4.4.4	1.5GHz (4)	2GB	WCN3660
HTC One	4.2.2	1.7GHz (4)	2GB	WCN3680
Sony Xperia J	4.0.4	1GHz (1)	512MB	BCM4330
Samsung Grand	4.1.2	1.2GHz (2)	1GB	BCM4329

Note: †: WNIC model numbers with prefix BCM/WCN are manufactured by Broadcom/Qualcomm, respectively.

Table 2 summarizes the result of the multi-layer delay measurement. The measurement for both phones consistently report smaller and consistent d_u when the packet sending interval is small. Moreover, the RTTs captured by `tcpdump` (i.e., d_k) and the external wireless sniffers (i.e., d_n) show that both are very close to d_u. However, as the sending interval is increased to 1s, both phones are observed with significant delay increase. In particular, Nexus 4 shows two different patterns. With the emulated RTTs of 60ms, the nRTT for Nexus 4 is inflated mainly in the network. But when the emulated RTT is 30ms, the inflation occurs inside and outside the phone. On the other hand, Nexus 5's nRTT is mainly inflated inside the phone. The reasons will be given in §3.2.2.

Table 2: RTTs measured at different layers (mean with 95% confidence interval in ms).

Phone	RTT	Intv.	d_u	d_k	d_n
Google Nexus 4	30ms	10ms	33.16 ±0.96	32.46 ±0.04	31.29 ±0.35
		1s	48.15 ±3.88	48.10 ±3.88	42.58 ±4.28
	60ms	10ms	63.91 ±0.73	63.86 ±0.73	62.32 ±0.46
		1s	136.33 ±7.64	136.66 ±7.66	130.03 ±7.52
Google Nexus 5	30ms	10ms	33.38 ±0.58	33.27 ±0.59	31.22 ±0.45
		1s	43.21 ±1.29	43.03 ±1.29	31.78 ±1.01
	60ms	10ms	64.18 ±0.68	64.08 ±0.67	61.61 ±0.35
		1s	81.98 ±2.04	81.83 ±2.05	62.35 ±0.42

To better visualize the distribution of the delay overheads, we employ box-and-whisker plots to present Δd_{k-n} in Figure 3(a) and 3(c) and Δd_{u-k} in Figure 3(b) and 3(d). In each plot, the mark inside the box is the median and the top and bottom are the 75th and 25th percentile. The upper and lower whiskers are the maximum and minimum, respectively, after excluding the outliers. Figure 3(a) and 3(c) clearly show that Nexus 4 and 5 experience comparably small Δd_{k-n}, which are smaller than ~4ms, when the packet sending interval is small. With the interval of 1s, Nexus 5 has a much larger Δd_{k-n} than Nexus 4 (~18ms vs. ~6ms in median for emulated RTT of 60ms, and ~12ms vs. ~6ms for emulated RTT of 30ms). On the other hand, since Δd_{u-k} is very close to 0 for both Nexus 4 and 5, Δd_{u-k} is not a major source of delay inflation. Moreover, some values of Δd_{u-k} are negative due to the low resolution of the reported `ping` results. For example, ping on Nexus 4 reports RTT in integer when RTTs are larger than 100ms. Therefore, the fractional part could be truncated and the round-down RTT could be smaller than the `tcpdump` measurement.

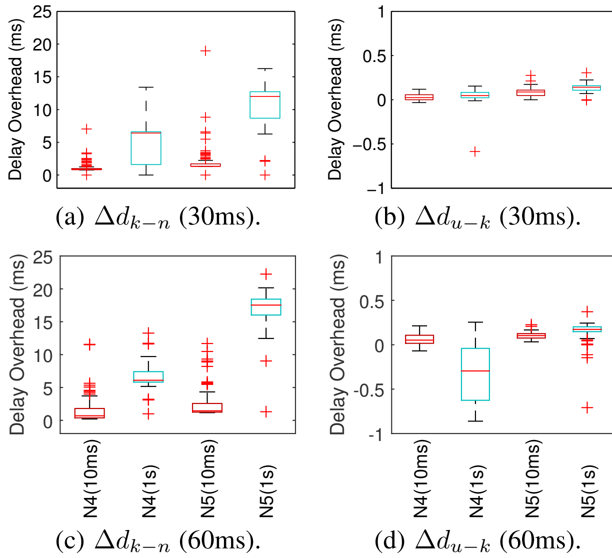
3.2 Root cause analysis

3.2.1 *Effect of SDIO bus sleep*

We first analyze the source code of the WNIC driver in Nexus 5. The WiFi chipset used by Nexus 5 (Broadcom BCM 4339) connects to the system through SDIO bus and adopts the "bcmdhd" driver [1]. This driver also supports other Broadcom WNIC chipsets, such as BCM 4329, 4330, 4335, and others. Therefore, the finding here is also applicable to other phones equipped with Broadcom WiFi chipset, especially those with FullMAC MLME (MAC Sublayer Management Entity).

We trace the function calls in the packet sending and receiving directions. For packet sending, the kernel function `dev_queue_xmit()` maps to driver function `dhd_start_xmit()`. As shown in Figure 4(a), this function further

[1] In "`drivers/net/wireless/bcmdhd`".

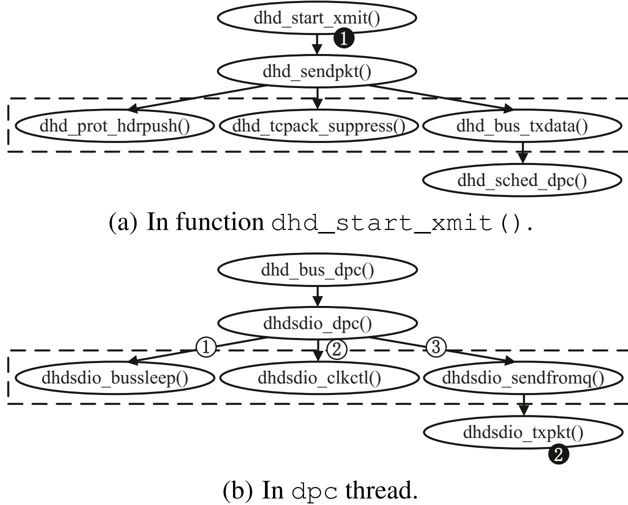


(a) Δd_{k-n} (30ms). (b) Δd_{u-k} (30ms).

(c) Δd_{k-n} (60ms). (d) Δd_{u-k} (60ms).

Figure 3: Kernel-phy delay overhead (Δd_{k-n}) and User-kernel delay overhead (Δd_{u-k}) for Google Nexus 4 and 5 when the emulated RTTs are 30ms and 60ms, respectively.

calls `dhd_sched_dpc()`, registering a packet sending task in a kernel thread, `dpc`. The `dpc` maintains a `while(1)` loop in its sub-function `dhdsdio_dpc()`. Before the `dpc` thread can send packets (③ in Figure 4(b)), `dhdsdio_dpc()` needs to check the status of SDIO bus and the readiness of backplane clock (① and ②, respectively). Eventually, function `dhdsdio_txpkt()` is executed, and the data are written to the bus.

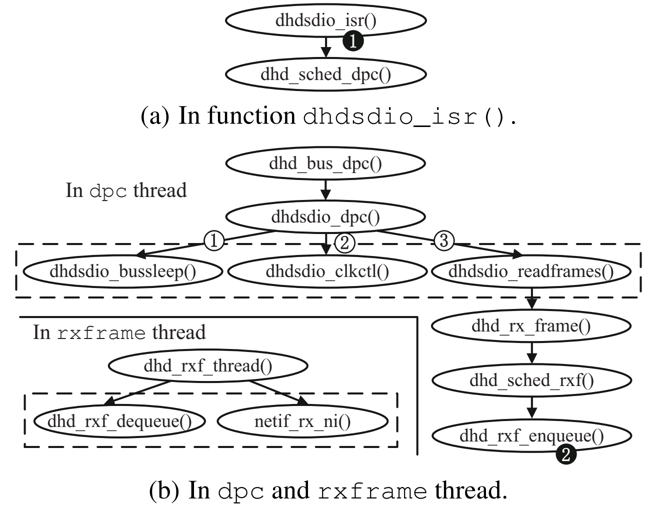


(a) In function `dhd_start_xmit()`.

(b) In `dpc` thread.

Figure 4: Key WNIC driver functions for packet sending.

For packet receiving, the `dpc` thread is also responsible for data processing. As shown in Figure 5(a), the interrupt handling function `dhdsdio_isr()` first registers a task in the `dpc` thread. Similar to packet sending, the `dpc` thread also needs to check the status of SDIO bus and backplane clock (i.e., ① and ② in Figure 5(b)), and then receives frames from the bus with function `dhdsdio_readframes()`. After the frames are queued using `dhd_rxf_enqueue()`, another kernel thread `rxframe` invokes `netif_rx_ni()` to deliver the packets to the system.



(a) In function `dhdsdio_isr()`.

(b) In `dpc` and `rxframe` thread.

Figure 5: Key WNIC driver functions for packet receiving.

We enable the driver debug message by re-compiling the Android kernel. The kernel log information shows that the driver puts the SDIO bus into sleeping state frequently if the data transmission rate is not high. When there is a packet sending request or a packet arrival interrupt, it takes time for the driver to bring up the bus. We modify the source code by timestamping at the entrances of functions `dhd_start_xmit()` and `dhdsdio_txpkt()` (i.e., ❶ and ❷ in Figure 4), so that we can measure the delay when the driver sends out a packet, denoted by d_{vsend}. Similarly, the delay d_{vrecv} for the driver to receive a packet can be measured by timestamping `dhdsdio_isr()` and `dhd_rxf_enqueue()` (i.e., ❶ and ❷ in Figure 5). We re-build the kernel to measure d_{vsend} and d_{vrecv} in Nexus 5.

To evaluate the effect of bus sleep, we also disable the sleep feature in `dhdsdio_bussleep()`. Table 3 presents the minimum, mean, and maximum values of d_{vsend} (d_{vrecv}) when sending out (receiving) 100 ICMP packets with the two packet sending intervals (10ms and 1s). After disabling the bus sleep feature, both d_{vsend} and d_{vrecv} drop closely to 1ms, regardless of the packet transmission rate. Otherwise, the mean value can be as high as 14ms when the packet sending interval is 1s. As a result, we have verified that the SDIO bus sleep is the main component in d_{k-n}.

Table 3: d_{vsend} and d_{vrecv} measured by Nexus 5 with SDIO bus sleep mode enabled or disabled (in ms). The delay for waking up the bus could go up to ∼14ms.

Type	Bus sleep	Packet interval	Min	Mean	Max
d_{vsend}	Enabled	10ms	0.096	0.321	10.184
		1000ms	0.139	10.151	13.547
	Disabled	10ms	0.092	0.229	0.836
		1000ms	0.139	0.720	0.858
d_{vrecv}	Enabled	10ms	0.314	1.635	2.827
		1000ms	0.368	12.754	14.224
	Disabled	10ms	0.311	1.589	2.651
		1000ms	0.362	1.756	2.088

We also investigate the criteria that trigger the bus sleep mode. Our driver analysis shows that the driver maintains a counter *idlecount*. For every *dhd_watchdog_ms*, whose

default value is 10ms, the driver increases this counter by 1 if the hardware is idle. When the counter reaches a threshold $idletime$ (5 by default), the driver will instruct the bus to sleep. Therefore, the default idle period is 50ms. Both $idletime$ and $dhd_watchdog_ms$ are configurable when the driver is loaded. Our experiments also confirm that the idle period (demotion timer, T_{is}) for Nexus 5 is 50ms.

The "wcnss" driver used by Qualcomm WNIC chipsets shares similar mechanism, although the chipsets connect to the system via SMD interface instead of SDIO. To simplify our presentation, we refer also this energy-saving mechanism to SDIO bus sleep mode.

3.2.2 Effect of Power Save Mode

The PSM allows WNIC to switch from *active* state (a.k.a. Constantly Awake Mode, CAM) to *sleep* state (a.k.a. Power Save Mode, PSM) in order to reduce energy consumption and prolong battery lifetime. In PSM, the WiFi station (STA) and AP agree upon a *listen interval*, which is the number of *beacon intervals* that the STA will ignore before turning on the receiver. Right before the end of the listen interval, the STA wakes up and listens for the beacon frame. Only when there are packets buffered on the AP will the STA stay in CAM to receive them.

PSM can be further classified into *static PSM* and *adaptive PSM*. As static PSM could lead to RTT round-up effect and degrade network performance [19], adaptive PSM is usually adopted by smartphones today [15, 30, 31]. The adaptive PSM allows STA to stay in CAM for a pre-defined idle period (PSM timeout, T_{ip}), preventing the STA from immediately going to sleep after data transmission.

However, adaptive PSM could still inflate the nRTT. Let d_p and d_n be the measured and actual network delay, respectively. If d_n is larger than T_{ip}, the STA has already turned off its receiver when the response packet arrives at the AP. Only after the STA listens to the beacon frame can it change the state to CAM and receive the packet. As a result, d_n could be inflated by up to $I_B * (L + 1)$, where I_B is the *beacon interval* with a value of 100 TUs (Time Units, 1.024ms per TU), and L is the listen interval.

We measure the PSM timeout value (T_{ip}) by carefully sending out packets with increased packet sending interval. Table 4 shows that T_{ip} is smartphone-dependent. As an extreme case, Nexus 4 enters PSM in 40ms when the WNIC is idle. Therefore, there is a higher possibility for Nexus 4 to report an inaccurate result when measuring a network path longer than 40ms. This is why Nexus 4 is observed with significant increase of network delay when the emulated RTT is set to 60ms in §3.1. On the other hand, the values of the listen interval determines how much nRTT can be inflated by PSM. Although STA announces a default listen interval during the *association period* (1 for "wcnss" driver and 10 for "bcmhd" driver by default), we find that the smartphones do not adopt this default value. The actual listen interval for the phones are all 0, meaning that its length is 1 beacon cycle which is 102.4ms. Thus, the adaptive PSM can inflate the nRTT by over 100ms.

Table 4: PSM timeout values (T_{ip}) and initial listen intervals (L) of the smartphones under test.

Phone	T_{ip}	L (associated)	L (actual)
Google Nexus 4	$\sim$40ms	1	0
Google Nexus 5	$\sim$205ms	10	0
Samsung Grand	$\sim$45ms	10	0
HTC One	$\sim$400ms	1	0
Sony Xperia J	$\sim$210ms	10	0

4. A BETTER PRACTICE

In this section, we present an effective approach to mitigate the delay overhead in the WiFi network measurement. We have implemented our approach in AcuteMon, an Android app prototype running on unrooted phones and requiring no system modification and customization, such as kernel recompilation and customized ROM. Although AcuteMon is designed mainly for WiFi networks, it can be easily extended to cellular environment, mitigating the effect of RRC (Radio Resource Control) state transition.

4.1 Implementation details

As illustrated in Figure 6, AcuteMon consists of two concurrent threads—background traffic thread (BT) and measurement thread (MT). The goal of the BT is to keep the smartphone in the wake-up state (i.e., SDIO bus awake mode and CAM in WiFi) during nRTT measurement. The process starts with a *warm-up phase* where the BT sends a warm-up packet to a warm-up server. According to §3, a smartphone enters CAM immediately when sending out packets, but it has to wait a period of time (promotion delay, T_{prom}) for the SDIO bus to wake up. After being activated, it remains in the SDIO bus awake mode for T_{is} and CAM for T_{ip}, where T_{is} and T_{ip} are the demotion timeout value for SDIO bus sleep and PSM, respectively. Therefore, the warm-up time d_{pre} should meet $T_{prom} < d_{pre} < \min(T_{is}, T_{ip})$.

In the subsequent *measurement phase*, the BT periodically sends lightweight background traffic with an inter-packet interval of d_b. With a proper choice of d_b ($d_b < \min(T_{is}, T_{ip})$), the background traffic can reset the state demotion timers to prevent any state demotion. Here we assign both d_{pre} and d_b to 20ms. Our evaluation in §4.2 shows that the empirical values work effectively. For example, d_b of 20ms is also appropriate for the smartphones employing "wcnss" driver. AcuteMon ignores the response packets of warm-up and background packets by setting the TTL (time-to-live) value of the warm-up packets to 1, so that the packets will be dropped at the first-hop router.

The MT, on the other hand, sends K measurement probes to measure the nRTT between the phone and a target server. In the current version, AcuteMon uses TCP control messages (TCP SYN/ACK packets) and TCP data packets (HTTP request and response) to measure nRTT to any TCP servers. The implementation can be easily extended to UDP and ICMP packets. We implement the MT as a pre-compiled C binary (instead of running within the Android runtime) to mitigate the user-kernel delay overheads [23].

In our prototype of AcuteMon, d_{pre} and d_b were assigned with empirical values. Although they work well in our testbed evaluation (§4.2-§4.4), they could be inappropriate for some

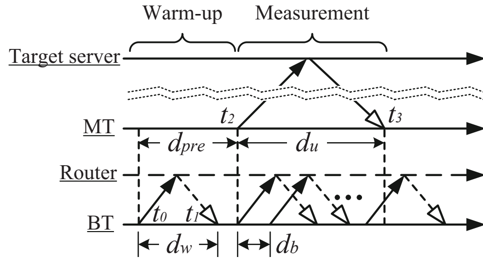


Figure 6: Measurement process of AcuteMon.

smartphone models, because both T_{is} and T_{ip} are tunable. However, inferring the actual T_{is} and T_{ip} of a particular smartphone is challenging. A simple solution is training the program to obtain suitable values. Another possible workaround is to collect the configurations by modelling and building a database. This will be our future work.

AcuteMon consumes very low battery, because it sends out very few additional packets in the measurement phase, and will not affect the energy-saving mechanisms when there are no measurement tasks. Moreover, the warm-up and background packet(s) are dropped in the first hop and will not burden the remaining part of a network path. Supposing that AcuteMon sends five probe packets ($K = 5$) to measure a path with nRTT of 100ms, the BT will send only around 25 packets to the gateway for the nRTT measurement. Our evaluation in §4.4 shows that the impact of the background traffic on the measurement results is negligible.

4.2 Performance evaluation

We evaluate the performance of AcuteMon in the testbed mentioned in §3.1. For each test, we run AcuteMon on the smartphone to measure the nRTT between the phone and the measurement server by sending out 100 TCP probes ($K = 100$). We introduce additional delays on the server side to emulate four different nRTTs: 20ms, 50ms, 85ms, and 135ms. Since the experiments are performed in an ideal environment without other mobile devices and cross traffic, introducing additional delays on the server side can be considered as controlling the length of the network path.

4.2.1 Actual nRTT

Table 5 presents the means and 95% confidence intervals of the actual nRTTs (d_n) measured by the external sniffers. For all five smartphones, d_ns are very close to their emulated values. In fact, no significant nRTT inflation can be observed, and most of the deviations are kept within 3ms, implying that the measurement packets have not been delayed at the AP. Our further analysis of the raw `pcap` files also confirms that no PSM activity can be detected when the smartphone receives response packets. Compared with the results shown in Table 2, AcuteMon successfully prevents the smartphones from entering PSM.

4.2.2 Delay overheads

Next we analyze the measurement accuracy of AcuteMon in terms of delay overhead. We use box plots to present Δd_{u-k} and Δd_{k-n} introduced by AcuteMon in Figure 7. On the x-axis, we use (u) and (k) after the RTT to denote Δd_{u-k} and Δd_{k-n}, respectively. Due to the page limit, we only show three of the five smartphones tested, because the rest have very similar results.

The previous study [23] has shown that executing the measurement logic as a native Linux program can mitigate the delay overheads caused in the DVM. The measurement results support this claim. We observe very small Δd_{u-k} for all the phones, most of which are smaller than 0.5ms. Even for the two smartphones with relatively low hardware configurations (i.e., Sony Xperia J and Samsung Grand), their Δd_{u-k}s are smaller than 1ms.

On the other hand, Δd_{k-n} accounts for the majority of the delay overhead. Although Δd_{k-n} is much larger than Δd_{u-k}, their medians are all less than 2ms, and the upper bounds are still less than 3ms (except for Sony Xperia J, whose upper bounds can be 4ms). For the smartphones equipped with Qualcomm's WNIC chipsets (Google Nexus 4 and HTC One), Figure 7(c) shows that the medians of Δd_{k-n} can be as small as ∼0.8ms. As a result, the overall median delay overheads are kept within 3ms.

Another important observation is that the delay overheads for AcuteMon are independent of nRTTs, and the values of the overheads are much more stable. Therefore, the true value can be obtained by performing calibration.

Table 5: The actual nRTTs (d_n) measured by external sniffers (mean with 95% confidence interval, in ms).

Phone	Emulated RTT (ms)			
	20	50	85	135
Google Nexus 5	22.461 ±0.545	51.683 ±0.168	87.198 ±0.387	137.090 ±0.320
Sony Xperia J	21.584 ±0.184	51.597 ±0.149	86.868 ±0.275	136.79 ±0.178
Samsung Grand	22.020 ±0.382	52.614 ±0.485	86.675 ±0.177	137.0 ±0.217
Google Nexus 4	21.680 ±0.181	51.673 ±0.202	86.888 ±0.358	137.98 ±1.101
HTC One	21.874 ±0.200	51.786 ±0.198	86.810 ±0.192	136.850 ±0.154

4.3 Comparison with other tools

We compare AcuteMon with three other popular tools, including ICMP ping, `httping` [18], and MobiPerf, on Google Nexus 5. Since we have evaluated the delay overhead caused by AcuteMon in §4.2.2, here we compare only the RTTs measured by these tools in the same testbed. For httping, we download and cross-compile its source code, so that it can run on Android. As for MobiPerf, it supports three measurement methods: 1) invoking the ping program, employing Java classes 2) `InetAddress`, and 3) `HttpURLConnection`. In fact, the second and third methods are very similar, both of which utilize TCP control messages (SYN/RST vs. SYN/SYN ACK). Since MobiPerf cannot configure the number of probe packets, we implement its second method in our own test app, called Java ping.

In comparing the four tools, we introduce an additional delay of 30ms on the server side. During the measurement, each tool sends out 100 probes ($K = 100$). We consider

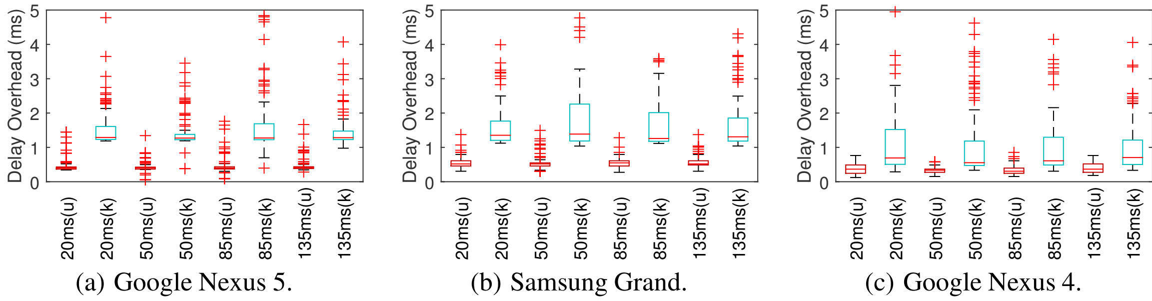


Figure 7: Box plots of Δd_{u-k} and Δd_{k-n} obtained by AcuteMon.

two scenarios, with and without cross traffic. The objective of introducing cross traffic is to congest the WiFi network. We use `iPerf` [3] to generate a large volume of data between a wireless load generator and a fixed load server. The load generator establishes 10 connections to the server, and each connection sends out UDP packets at a sending rate of 2.5Mbps. According to [37], the actual throughput for UDP traffic in an IEEE 802.11g WLAN is usually smaller than 20Mbps. Therefore, the introduced cross traffic can overload the network. In our test, we find that the maximum throughput is only around 10Mbps.

Figure 8 plots the CDF of the RTTs measured by the four tools. The figure clearly shows that AcuteMon outperforms the other three significantly in both scenarios. When the network is free of cross traffic, almost 90% of the RTTs measured by AcuteMon are smaller than 35ms. The differences between AcuteMon and the other three are almost larger than 10ms. In the congested network case, the measured RTTs are all increased, and AcuteMon still gives the smallest measured RTTs.

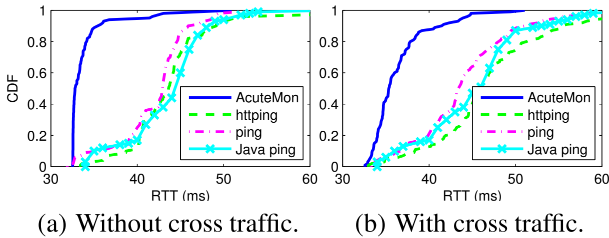


Figure 8: CDF plots of RTT measured by AcuteMon and other popular measurement tools.

4.4 Effect of background traffic

We further study whether the background traffic sent by AcuteMon will affect the measurement results in a congested WiFi network. We apply the same settings of emulated RTT and cross traffic described in §4.3 to the testbed. For a fair comparison, we disable the SDIO bus sleep function by modifying the driver, so that AcuteMon will stay in the wake-up mode without sending background traffic. Moreover, since the emulated RTT (30ms) is smaller than Nexus 5's PSM timeout value (~205ms), the smartphone will also stay in the CAM during the measurement. As shown in Figure 9, the difference between with and without background traffic for AcuteMon is very small, showing that the effect of sending just enough background traffic is slight. Compared to the result obtained when the network is not congested, the RTT increase here is mainly caused by the cross traffic, but not the background traffic.

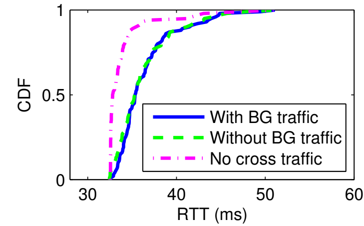


Figure 9: CDF plot of RTTs measured by AcuteMon with and without sending background traffic.

5. RELATED WORKS

The performance of Android system has been well studied and evaluated in the past, especially on the performance impact of DVM on Android apps [27], and the comparison between native C/JNI and Java applications in DVM [12, 21]. However, these works seldom study the relationship between system delay and network delay measurement. Although some other studies apply cross-layer analysis [39] and intercept system events [20], they do not analyze the network behavior systematically nor study the device driver. Li et al. [23] study the delay overhead in WiFi network and propose to mitigate the user-level overhead by using native C implementation. In this paper, we systemically investigate the delay overhead caused by energy-saving mechanisms for WiFi networks.

6. CONCLUSION

In this paper we considered the problem of measuring network RTT from smartphones. We presented AcuteMon, a network measurement app in Android, to mitigate all major sources of delay inflation for WiFi networks. We first reported and demonstrated that the energy-saving mechanisms (SDIO and PSM) employed in smartphones are the main sources of the delay inflation. Based on our driver code analysis and empirical evaluations, we proposed to keep the phone in the wake-up state during the delay measurement through a carefully timed sending of warm-up and periodic background traffic. Our testbed experiment results show that our approach can effectively mitigate the delay overheads and achieve very accurate network RTT.

7. REFERENCES

[1] Internet Speed Test 3G,4G,Wifi on Google Play. https://play.google.com/store/apps/details?id=uk.co.broadbandspeedchecker.

[2] Internet Speed Test on Google Play. https://play.google.com/store/apps/details?id=pl.speedtest.android.

[3] iPerf - the ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/.

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

[5] MopEye: Speed test & sniffer. https://play.google.com/store/apps/details?id=com.mopeye.

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

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

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

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

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

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

[12] 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.

[13] Y. Bejerano, J. Ferragut, K. Guo, V. Gupta, C. Gutterman, T. Nandagopal, and G. Zussman. Scalable WiFi multicast services for very large groups. In *Proc. ICNP*, 2013.

[14] Y.-C. Cheng, J. Bellardo, P. Benkö, A. C. Snoeren, G. M. Voelker, and S. Savage. Jigsaw: Solving the puzzle of enterprise 802.11 analysis. In *Proc. ACM SIGCOMM*, 2006.

[15] N. Ding, A. Pathak, D. Koutsonikolas, C. Shepard, Y. Hu, and L. Zhong. Realizing the full potential of PSM using proxying. In *Proc. IEEE INFOCOM*, 2012.

[16] FCC. 2015 measuring broadband America fixed report. https://www.fcc.gov/reports-research/reports/measuring-broadband-america/measuring-broadband-america-2015, 2015.

[17] Y. He and R. Yuan. A novel scheduled power saving mechanism for 802.11 wireless LANs. *Mobile Computing, IEEE Transactions on*, 8(10):1368–1383, Oct 2009.

[18] F. Heusden. httping. http://www.vanheusden.com/httping/.

[19] R. Krashinsky and H. Balakrishnan. Minimizing energy for wireless web access with bounded slowdown. *Wireless Network*, 11:135–148, Jan. 2005.

[20] J.-C. Kuester and A. Bauer. Monitoring real android malware. In *Proc. Runtime Verification*, 2015.

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

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

[23] W. Li, R. Mok, D. Wu, and R. Chang. On the accuracy of smartphone-based mobile network measurement. In *Proc. IEEE INFOCOM*, 2015.

[24] X. Luo, E. Chan, and R. Chang. Design and implementation of TCP data probes for reliable network path monitoring. In *Proc. USENIX ATC*, 2009.

[25] 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.

[26] A. Nikravesh, D. Choffnes, E. Katz-Bassett, Z. Mao, and M. Welsh. Mobile network performance from user devices: A longitudinal, multidimensional analysis. In *Proc. PAM*, 2014.

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

[28] A. Patro, S. Govindan, and S. Banerjee. Observing home wireless experience through WiFi APs. In *Proc. ACM MobiCom*, 2013.

[29] C. Pei, Y. Zhao, G. Chen, R. Tang, Y. Mengy, M. Ma, K. Ling, and D. Pei. WiFi can be the weakest link of round trip network latency in the wild. In *Proc. IEEE INFOCOM*, 2016.

[30] A. J. Pyles, X. Qi, G. Zhou, M. Keally, and X. Liu. SAPSM: Smart adaptive 802.11 PSM for smartphones. In *Proc. ACM UbiComp*, 2012.

[31] E. Rozner, V. Navda, R. Ramjee, and S. Rayanchu. NAPman: Network-assisted power management for Wifi devices. In *Proc. ACM MobiSys*, 2010.

[32] B. Shaffer. Broadcom and Qualcomm battle for WLAN IC leadership. https://technology.ihs.com/517658/broadcom-and-qualcomm-battle-for-wlan-ic-leadership, November 2014.

[33] V. Shrivastava, S. Rayanchu, S. Banerjee, and K. Papagiannaki. PIE in the sky: Online passive interference estimation for enterprise WLANs. In *Proc. USENIX NSDI*, 2011.

[34] K. Sui, M. Zhou, D. Liu, M. Ma, D. Pei, Y. Zhao, Z. Li, and T. Moscibroda. Characterizing and improving WiFi latency in large-scale operational networks. In *Proc. ACM MobiSys*, 2016.

[35] N. Vallina-Rodriguez, N.Weaver, C. Kreibich, and V. Paxson. Netalyzr for Android: Challenges and opportunities. In *Proc. Workshop on Active Internet Measurements (AIMS)*, 2014.

[36] N. Vallina-Rodriguez, S. Sundaresan, C. Kreibich, N. Weaver, and V. Paxson. Beyond the radio: Illuminating the higher layers of mobile networks. In *Proc. ACM MobiSys*, 2015.

[37] A. L. Wijesinha, Y. tae Song, M. Krishnan, V. Mathur, J. Ahn, and V. Shyamasundar. Throughput measurement for UDP traffic in an IEEE 802.11g WLAN. In *Proc. SNPD/SAWN*, 2005.

[38] D. Wu, W. Li, R. Chang, and D. Gao. MopEye: Monitoring per-app network performance with zero measurement traffic. In *Proc. ACM CoNEXT Student Workshop*, 2015.

[39] L. Xue, C. Qian, and X. Luo. Androidperf: A cross-layer profiling system for Android applications. In *Proc. IEEE IWQoS*, 2015.

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