Is What You Measure What You Expect? Factors Affecting Smartphone-Based Mobile Network Measurement

Lei Xue*, Xiaobo Ma†*, Xiapu Luo*§, Le Yu*, Shuai Wang*, Ting Chen‡*
*Department of Computing, The Hong Kong Polytechnic University
†MOE KLINNS Lab, Xi'an Jiaotong University
‡Cybersecurity Research Center, University of Electronic Science and Technology of China
*{cslxue,csxluo,cslyu,csswang}@comp.polyu.edu.hk, †xma.cs@xjtu.edu.cn, ‡brokendragon@uestc.edu.cn

***Abstract*—Many apps have been developed to measure the performance of mobile networks. Unfortunately, their measurement results may not be what users expect, because the results could be biased by various factors and the apps' descriptions may confuse users. Although a few recent studies pointed out several factors, they missed other important factors and lacked of fine-grained analysis on the factors and measurement apps. Moreover, none has studied whether or not the descriptions of such apps will mislead users. In this paper, we conduct the *first* systematic study of the factors that could bias the result from measurement apps and their descriptions. We identify new factors, revisit known factors, and propose a novel approach with new tools to discover these factors in proprietary apps. We also develop a new measurement app named `MobiScope` for demonstrating how to mitigate the negative effects of these factors. Furthermore, we construct enhanced descriptions for measurement apps to provide users more information about what is measured. The extensive experimental results illustrate the negative effects of various factors, the improvement in performance measurement brought by `MobiScope`, and the clarity of the enhanced descriptions.**

I. Introduction

The widespread adoption of smartphone and the prosperity of mobile apps demand stable and high-speed mobile networks. Hence, how to measure the performance of mobile network has recently attracted a lot of attention from governments, academia, and industry [1], [2]. Many measurement apps have been developed and published in Google Play or Apple Store [3]. Unfortunately, the measurement results from apps may not be what users expect because of two reasons.

First, various factors could affect the measurement results, and not all app developers are network measurement experts being aware of such factors. For example, according to RFC 2681 [4], the round-trip time (RTT) reported by an app is determined by the *host times*, which include the timestamp just prior to sending the packet and that right after receiving the response packet. In contrast, the network RTT is calculated using the *wire times*, which refer to the time when the packet leaves the smartphone's network interface (NIC) and the time when the corresponding response packet arrives at the smartphone's NIC. It has been shown that the difference between *host times* and *wire times* is not ignorable [5].

Second, users could misunderstand the results from the mobile measurement apps because their descriptions may be ambiguous. For example, although many apps claim to be able to measure *RTT*, they refer to different time intervals, such as, RTTs derived from host times/wire times, time-to-first-byte with/without DNS resolution.

Although a few recent studies pointed out some factors (e.g., Dalvik virtual machine) that may affect the measurement [5]–[7], they have several limitations, such as, missing other important factors (e.g., implementation patterns), conducting only coarse-grained analysis, and lack of evaluating off-the-shelf measurement apps. Please refer to Section VII for detailed differences between previous studies and our investigation. Moreover, none has studied whether or not the descriptions of measurement apps will mislead users.

In this paper, we conduct the *first* systematic study of the factors that could bias the result from measurement apps and their descriptions. It is non-trivial to accomplish this study because the measurement process involves intricate factors from apps, OS, and network protocols. Moreover, it is difficult and time-consuming to understand how an app performs the measurement, not to mention that most apps are proprietary.

We examine Android system, apps, and network protocols to identify new factors and revisit known factors (Sec. II), and perform extensive experiments to quantify their effect (Sec. V). Android system is selected because it has occupied more than 81% market share [8]. To discover these factors in measurement apps, we develop two tools, namely `AppDissector`, a static bytecode analyzer, and `AppTracer`, a dynamic trace analyzer (Sec. III). We also design `MobiScope`, a measurement app for demonstrating how to mitigate the negative effects of various factors (Sec. IV).

Furthermore, we construct enhanced descriptions for measurement apps to provide users more information about what is measured by leveraging the static and dynamic analysis of measurement apps. User studies have been performed to assess whether the original and the enhanced descriptions make users understand what the apps measure (Sec. VI).

Our major contributions are summarized as follows:

(1) We conduct the *first* systematic study of the factors that could bias the result from measurement apps by identifying new factors, revisiting known factors, and quantifying the negative effects through extensive experiments.
(2) We propose a novel approach combining static bytecode analysis and dynamic trace analysis, and develop practical tools to facilitate discovering these factors in apps.
(3) We develop `MobiScope`, a new measurement app that

§ The corresponding author.

adopts various techniques to mitigate those factors' negative effects. It will be released after the paper is published.

(4) We perform the *first* examination on the clarity of measurement apps' descriptions, and construct enhanced descriptions to inform users what is measured.

Roadmap. Sec. II elaborates the classes of factors. Sec. III details `AppDissector` and `AppTracer` for inspecting the factors. Sec. IV describes `MobiScope`. We present experiments in Sec. V, and report user studies in Sec. VI. After introducing related work in Sec. VII, we conclude in Sec. VIII.

II. Factors Affecting Measurement Results

We classify the factors that could bias the measurement into three categories, namely implementation patterns (Sec. II-A), Android architecture and configurations (Sec. II-B), and network protocols (Sec. II-C). We identify new factors in category 1 and 2, including implementation patterns for HTTP/TCP based measurement, monitoring and power management mechanisms in Android. Moreover, we revisit known factors, including multiple-layer nature of Android in category 2 and mobile and WiFi protocols in category 3. More precisely, in contrast to previous study [5], we conduct a fine-grained analysis on the effect of each layer and examine the new Android runtime. For category 3, we examine the protocols' effect on the measurement result and suggest solutions.

We use the process of RTT measurement, one primitive measurement task [4], to explain the effect of various factors. These factors have similar effect on the capacity and throughput measurement, because they will introduce non-negligible time gap to back-to-back packets used for measuring these metrics [9]. Note that although the effect of some factors could be mitigated using statistical algorithms (e.g., outlier detection), eliminating the effect of other factors requires app re-design or modification (e.g., those in Sec. II-A).

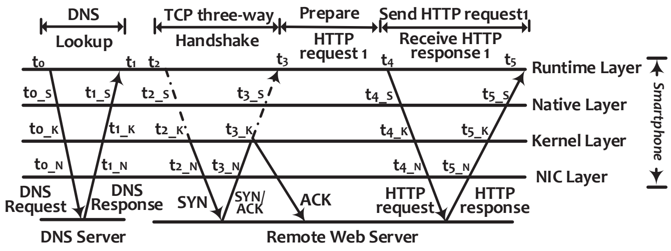


Fig. 1. An example of RTT measurement conducted on an Android smartphone. We use dash-dot line to connect t_2 and t_{2_k} because the TCP SYN packet is sent by the kernel but triggered by the function at the runtime layer. Similarly, the dash-dot line connecting t_{3_k} and t_3 indicates that the TCP SYN/ACK packet is received by the kernel without forwarding to the runtime layer but the system will notify the function at that layer.

Fig.1 shows the measurement process with three stages, namely, (1) DNS lookup; (2) TCP three-way handshaking; and (3) preparing and sending HTTP request 1 to a remote web server, and receiving HTTP response 1. Fig. 1 also depicts the multiple layers of Android, including runtime, system (i.e., the user space of Android's customized Linux), kernel, and network interface (NIC). In Fig. 1, t_i denotes the timestamp obtained at the runtime layer. t_{i_S}, t_{i_K}, and t_{i_N} (i=0…9) represent the timestamps recorded at system/kernel/NIC layer, respectively. According to RFC2681, t_i, t_{i_S}, and t_{i_K} are *host times*, referring to the timestamps acquired right before sending a request and those obtained just after receiving the response at different layers. t_{i_N} is the wire time, including the time when a packet leaves the NIC and the time when the response packet arrives at the NIC.

A. Category 1: Implementation Patterns

Although various apps claim measuring the same metric (e.g., RTT), they may adopt different implementation patterns that lead to different results. We summarize common patterns used by apps for measuring RTT in Fig.2. Generally, an app first obtains a timestamp (i.e., $tStart$), and then performs the measurement. Finally, it records another timestamp (i.e., $tEnd$) and computes RTT as $tEnd - tStart$. Note that we only examine HTTP (or TCP) based RTT measurement, because most measurement apps support them and it is easy for users to find a web server for measurement, not to mention that HTTP is widely used by various apps [10].

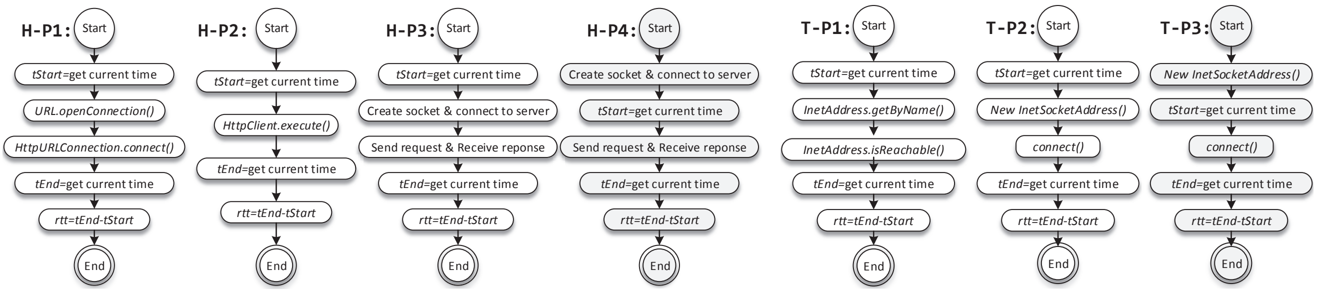
HTTP-based RTT measurement: H-P1 measures $t_3 - t_0$ if resolving the destination's IP is required. Otherwise, it outputs $t_3 - t_2$. More precisely, after $tStart$ is gained, an instance of `URLConnection` is created through *openConnection()* in the class `java.net.URL`. Then, *connect()* in the class `java.net.HttpURLConnection` is called before getting $tEnd$. Note that the Java method *connect()* calls the native method *getaddrinfo()* to perform DNS lookup, and then invokes the native method *connect()* to create a TCP connection.

H-P2 measures $t_5 - t_0$ if IP resolving is needed. Otherwise, it estimates $t_5 - t_2$. Specifically, between getting $tStart$ and $tEnd$, it employs *execute()* in the class `org.apache.http.client.HttpClient` to send an HTTP HEAD request and receive the corresponding HTTP response. Note that before sending an HTTP request and receiving the HTTP response, *execute()* calls the native methods *getaddrinfo()* and *connect()* to establish a TCP connection.

H-P3 measures $t_5 - t_2$ using methods in the class `java.net.Socket`. The whole process includes establishing the TCP connection, constructing an HTTP request, sending the HTTP request, receiving the HTTP response, and parsing the HTTP response.

H-P4 estimates $t_5 - t_4$, which is the correct network RTT measured by HTTP. To minimize the difference between *host time* and *wire time* [4], $tStart$ is captured just before sending the request encapsulated in *one* packet, and $tEnd$ is recorded right after receiving the first packet of the response.

TCP-based RTT measurement: T-P1 measures $t_3 - t_0$ if IP resolving is required. Otherwise, it yields $t_3 - t_2$ or a much larger value depending on the implementation of *isReachable()*. More precisely, after using *getByName()* in the class `java.net.InetAddress` to resolve the IP, it invokes *isReachable()* in the same class for checking whether the destination is reachable. According to official documentation [11], *isReachable()* first uses ICMP to test the reachability, and returns to TCP (i.e., exploit TCP three-way handshaking for RTT measurement) if ICMP method fails. Therefore, if ICMP packets are dropped by a router, which is common in today's Internet, the measured value would be much larger than the



(a) HTTP ping. (b) TCP ping

Fig. 2. Implementation patterns for HTTP-based and TCP-based RTT measurement.

real RTT. Although we find that the current implementation of *isReachable()* does not realize the ICMP-based probing, it is *not* recommended to use this method for measuring RTT in case the ICMP-based probing is realized in future version.

T-P2 invokes two Java methods: *InetSocketAddress()* in the class `java.net.InetSocketAddress` and *connect()* in the class `java.net.Socket`. Since the former method will conduct DNS lookup, **T-P2** measures $t_3 - t_0$ if IP resolving is needed. Otherwise, it outputs $t_3 - t_2$. To accurately measure network RTT, we suggest recording $tStart$ and $tEnd$ right before and after invoking *connect()* to avoid the additional delay due to DNS lookup, as shown in **T-P3** in Fig.2(b).

B. Category 2: Android Architecture and Configurations

Multiple-layer nature of Android. Since apps run within the runtime, which is a Linux process, packets sent from apps would be delayed at all layers. Since Android 5.0, the default runtime is changed from the Dalvik virtual machine (DVM) to the new Android runtime (ART) for better performance [12]. In particular, apps will be compiled into native code before execution. Note that 51.6% Android devices are still using DVM [13]. Although Li et al. found that DVM may introduce considerable delay [5], they neither conduct fine-grained analysis on the delay caused by different layers nor study ART.

Monitoring mechanisms. Several monitoring mechanisms can be used in Android for monitoring packets. *libpcap* captures packets in the kernel through BPF filter. *Netfilter* allows users to register packet handlers and enables *iptables* to inspect packets that match pre-specified rules. Using *iptables*, *VpnService* [14] allows apps to redirect traffic to a tunnel. It has been employed to capture packets and conduct measurement [15]. All these mechanisms will introduce additional delay if they inspect the measurement packets.

Power management. Android has an aggressive power management strategy but provides a mechanism called *wake locks* [16] that empowers apps to keep the device awake. These mechanisms have an indirect impact on measurement results because they affect the parameters of PSM (Power Save Mode) adopted by WiFi interface, which introduces additional delay.

C. Category 3: Network Protocols

The effect of network protocols on measurement is mainly due to their state transitions, because previous studies reveal that the state transitions will introduce noticeable delays [6], [17], [18]. We revisit these factors because to what extent it may bias the measurement result remains unknown.

Cellular Network. The RRC (Radio Resource Control) protocol of cellular networks influences power consumption and network performance [6], [17]. Typically, 3G has three main RRC states (i.e., `IDLE`, `FACH` and `DCH`), while LTE has two main states (`IDLE` and `CONNECTED`). Typically, packets transmitted by cellular interface in the `DCH` (3G) or `CONNECTED` (LTE) state experience shortest delay. Therefore, when the cellular interface is in the states other than `DCH` or `CONNECTED`, RTT measurement will suffer from additional delays due to the state transition.

WiFi Network. PSM allows WiFi interfaces to sleep for an integer number of beacon intervals (i.e., *ListenInterval*), and then wake up to detect the presence of its buffered frames in the access point (AP) [7]. If no frames are detected, the interface continues to sleep for *ListenInterval* beacon intervals. Otherwise, it wakes up to retrieve the buffered frames one by one. When none of the buffered frames are left, the WiFi interface goes back to sleep again. Pyles et al. found that different PSM algorithms lead to different delays [18].

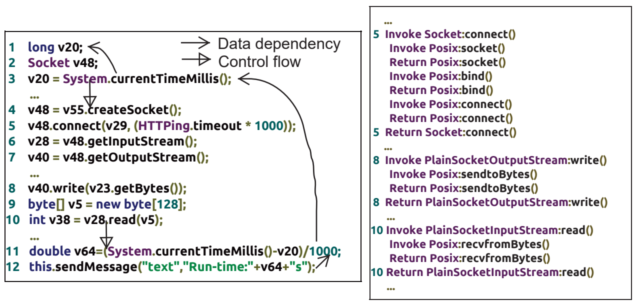
III. Is An App Affected By These Factors?

We combine static bytecode analysis and dynamic trace analysis to inspect whether an app uses any implementation pattern in Fig. 2, and whether it adopts any approaches to mitigate the negative effect of the factors described in Sec. II. We further develop two tools (i.e., `AppDissector` and `AppTracer`) to facilitate this inspection. Given a measurement app, `AppDissector` locates its measurement code and checks: (**C1**) whether it uses native code to perform the measurement for avoiding the effect of runtime; (**C2**) whether it employs *VpnService* to handle packets; (**C3**) whether it requests *wake lock* and *Wi-Fi lock* to mitigate the effect of power saving mechanisms. `AppTracer` collects information of method calls in Android framework, system libraries, and system calls to construct the cross-layer method call graph and determine whether the app uses *libpcap* or *iptables*.

A. Static Bytecode Analysis

Pre-processing. Given an app, `AppDissector` uses `VulHunter` [19] to its abstract syntax trees(AST), inter-procedure control flow graph(ICFG), method call graph, and system dependency graph, and utilizes `IccTA` [20] to find the target of each intent because an app's components can communicate through intents.

Then, we look for the entry of the measurement process. If the entry is a UI component, we locate its callback functions



```
1   long v20;
2   Socket v48;
3   v20 = System.currentTimeMillis();
    ...
4   v48 = v55.createSocket();
5   v48.connect(v29, (HTTPing.timeout * 1000));
6   v28 = v48.getInputStream();
7   v40 = v48.getOutputStream();
    ...
8   v40.write(v23.getBytes());
9   byte[] v5 = new byte[128];
10  int v38 = v28.read(v5);
    ...
11  double v64=(System.currentTimeMillis()-v20)/1000;
12  this.sendMessage("text","Run-time:"+v64+"s");
```

(a) Static bytecode analysis.

```
   ...
5  Invoke Socket:connect()
    Invoke Posix:socket()
    Return Posix:socket()
    Invoke Posix:bind()
    Return Posix:bind()
    Invoke Posix:connect()
    Return Posix:connect()
5  Return Socket:connect()
   ...
8  Invoke PlainSocketOutputStream:write()
    Invoke Posix:sendtoBytes()
    Return Posix:sendtoBytes()
8  Return PlainSocketOutputStream:write()
    ...
10 Invoke PlainSocketInputStream:read()
    Invoke Posix:recvfromBytes()
    Return Posix:recvfromBytes()
10 Return PlainSocketInputStream:read()
    ...
```

(b) Dynamic trace analysis.

Fig. 3. A snippet of the static and dynamic analysis results of `HTTPing`.

from two sources, because they may start the measurement process. First, we parse the layout file to find the callback functions because they can be set in it. Second, since an app can get a UI component in code and register event listeners, we first obtain the UI component's ID and then enumerate all statements using this ID as parameters. After that, we search the AST of each event listener to locate the callback functions.

Locating measurement code. The measurement code includes the measurement result related code for deriving the result and the measurement procedure related code for conducting the measurement. We first look for the former by locating the variables/fields representing the measurement results and then performing *backward slicing* and *chopping* [21]. More precisely, we conduct *backward slicing* to identify the statements that influence the computation of measurement results. We perform the depth-first traversal from the statement that gets the measurement results following the data dependency relation. The traversal stops at local declaration statement because it defines a new variable without depending on other statements. All statements on the paths are saved in W. Then, we perform *chopping* to identify the statements that use the variables defined in W. For each statement in W, we check the variable defined in it. If the variable is used by another statement s, we add s into W. Finally, all statements in W are regarded as the measurement result related code.

After that, we leverage the statements in W to look for the measurement procedure related code. More precisely, we locate the first and the last statements without considering local declaration statements, and then traverse the control flow graph to find the paths between them. All statements on the paths are saved in W'. We output the statements in W and W', and regard the statements that are not included in W as the measurement procedure related code.

We use `HTTPing`, whose snippet is shown in Fig.3(a), as an example to illustrate the procedure. In `HTTPing`, the measurement result $v64$ is calculated at line 11. To know how this result is generated, we add line 11 into W and perform *backward slicing* from it. The depth-first traversal stops at the local declaration statement (i.e., line 1). We also add lines 1 and 3 to W. To know how the measurement result is used, we perform *chopping* on each statement in W. As $v64$ used in line 12 (i.e., show user the measurement result), we add line 12 into W. Finally, W contains lines 1, 3, 11, 12. We regard them as the measurement result related code. Then, we traverse from line 3. The traversal stops at line 12. Lines 3-12 are put in W'. Since lines 4-10 are not included in W, we regard them as the measurement procedure related code.

C1: We traverse the ICFG from the entry of the measurement process to identify statements invoking native methods that affect the measurement results. If found, we use dynamic trace analysis to confirm whether the native methods perform the measurement or not, because our static bytecode analysis module currently cannot handle native codes.

C2: We check whether the app requests the permission `BIND_VPN_SERVICE` in `AndroidManifest.xml` and invokes *android.net.VpnService.establish()* to create a VPN interface. If so, the app uses *VpnService*.

C3: An app can keep the screen on by calling *Window.addFlags()* with `FLAG_KEEP_SCREEN_ON` or setting the attribute `android:keepScreenOn` to "true" in the layout file. We look for them by inspecting the ASTs and the layout file. To lock the Wi-Fi, the app must request `WAKE_LOCK` permission and invoke *WifiLock.acquire()*. We discover it by parsing the manifest file and checking the ASTs.

B. `AppTracer`: Dynamic Trace Analysis

It is non-trivial to design a tool running in smartphone to collect information about method calls across layers. We accomplish it by developing `AppTracer` based on `valgrind` [22], whose architecture is shown in Fig.4. The tracers trace method invocations and returns at the corresponding layers, and the trace analyzer generates the control flow information based on the logs generated by the tracers.

DVM Runtime Tracer. The information of each invoked DVM function can be collected from *dvmMethodTraceAdd()* if Android's profiling framework is enabled [23]. Therefore, the DVM runtime tracer wraps this function for tracing functions at the DVM layer. When a measurement app is launched, we enable Android's profiling framework, and then the DVM runtime tracer collects the information of method invocation and return in the wrapper function of *dvmMethodTraceAdd()*.

ART Runtime Tracer. `AppTracer` supports ART. Since the functions *Trace::MethodEntered()* and *Trace::MethodExited()* are called when each method is invoked and returned, respectively, the ART runtime tracer obtains the entering and exiting events of the involved methods by wrapping both functions in `libart.so` and enabling Android's profiling framework.

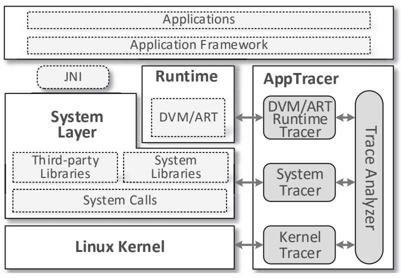


Fig. 4. Architecture of `AppTracer`

System Tracer. It collects information about system library functions and system calls. By using `valgrind` to get the instruction level information and the addresses of all functions in loaded libraries, it obtains the function invocation and return information by matching function addresses and the target addresses of jump instructions. Since `valgrind` can obtain

TABLE I
THE IMPLEMENTATION PATTERNS OF 10 MEASUREMENT APPS UNDER EXAMINATION. `PING` REFERS TO USING THE DEFAULT `PING` TOOL IN ANDROID.

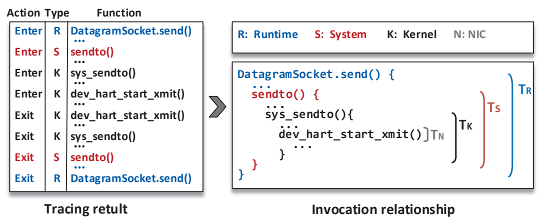
App	Fing[2]	MobiPerf[2]	Netalyzr	WiFi Speed Test[1]	Internet Speed Test[1]	Ping&DNS[1,2]	PingTools	he.net	HTTPing	Httping Tool
Implementation	`ping`	H-P1, T-P1, `ping`	T-P2	T-P1	T-P2	T-P2, `ping`	H-P2, T-P2, `ping`	`ping`	H-P3	T-P3[3]

[1] `WiFi Speed Test` and `Ping&DNS` provide option to prevent screen off; `Internet Speed Test` keeps screen on during the measurement process.
[2] `Fing`, `MobiPerf` and `Ping&DNS` acquire the WAKE_LOCK permission in their `AndroidManifest.xml` file.
[3] Its implementation is similar to the pattern T-P3 with difference that it sends HTTP request before getting $tEnd$.

the number of each system call when it is invoked and returns, we maintain the relationship between the number and the name of each system call to identify all system calls involved.

Kernel Tracer. It utilizes the *function_graph* tracer of *ftrace* to record the flow of every function call in the kernel and then constructs the function call graph according to the traces generated by the *function_graph* tracer.

Trace Analyzer. It constructs the cross-layer control flow by exploiting the order of entering and existing a function, which are recorded by tracers at different layers. For example, Fig.5 shows the tracing results generated by `AppTracer`. Since these functions are not executed in parallel, we reconstruct the cross-layer dynamic control flow as shown in the sub-figure on the right side of Fig.5 according to the entering and exiting orders of all invoked functions at different layers.



```
Enter  R  DatagramSocket.send()
          ...
Enter  S  sendto()
          ...
Enter  K  sys_sendto()
          ...
Enter  K  dev_hart_start_xmit()
          ...
Exit   K  dev_hart_start_xmit()
          ...
Exit   K  sys_sendto()
          ...
Exit   S  sendto()
          ...
Exit   R  DatagramSocket.send()
```

```
DatagramSocket.send() {
  ...
  sendto() {
    ...
    sys_sendto(){
      ...
      dev_hart_start_xmit()
        ...
      }
  }
}
```

Fig. 5. Example of reconstructing invocation relationship from tracing result.

With the cross-layer control flow, we further correlate it to the result of static analysis. For example, from the dynamic analysis results of `HTTPing` shown in Fig.3(b), we can match Java methods in Fig.3(a) to JNI functions in `libcore_io_Posix.cpp`. Therefore, by combining the results of static and dynamic analysis, we can learn that the RTT measured by `HTTPing` includes constructing TCP connection, sending HTTP request, receiving HTTP response. In other words, it follows the H-P3 pattern in Fig.2.

To check whether an app uses *iptables*, `AppTracer` calls *"iptables -L"* to list all *Netfilter* rules. Since *libpcap* captures packets through `PF_PACKET` socket, `AppTracer` hooks the function *socket()* to monitor whether `PF_PACKET` socket is created. If so, `AppTracer` checks whether the socket is used for receiving packets. Based on the result, `AppTracer` knows whether the app uses *libpcap* to capture packets.

IV. MobiScope

We develop `MobiScope` to demonstrate how to mitigate the effects of the factors in Section II. To mitigate the effect of multiple layers (i.e., shorten the difference between *host time* and *wire time* [4]), `MobiScope`'s two major modules (i.e., `kping` and `kband`) are realized in kernel. Although installing kernel modules requires root privilege, we develop them for users requiring highly accurate measurement results.

Using ICMP packets to measure RTT, `kping` first constructs an echo request packet in kernel and stores it in the structure *sk_buff*, and then sends it out by calling kernel function *dev_hard_start_xmit()*, which is the entry of device driver [24]. It uses *ktime_get_ts()* to obtain the sending timestamp with nanosecond resolution. `kping` registers a *Netfilter* function to receive echo reply packet. Before the measurement, `kping` calls the function *net_enable_timestamp()* to enable *sk_buff* timestamping so that each packet's arriving timestamp will be stored in the field *tstamp* of structure *sk_buff* by the NIC driver. To measure capacity or throughput, `kband` sends packet trains through *dev_hard_start_xmit()* and receives packets through another registered *Netfilter* function. It records the timestamps by calling *ktime_get_ts()*.

Although `MobiScope` uses *Netfilter*, we minimize its impact on the measurement by taking two measures. First, invoking functions in the device driver to send packets for excluding the impact of *Netfilter*. Second, manipulating the *Netfilter* chains to register the packet receiving function at the first position for mitigating the impact of other *Netfilter* rules.

To mitigate the effect of power management and NIC state transition, `MobiScope` acquires WakeLock and WifiLock to keep the device and WiFi radio awake, respectively. It also sends several packets before starting the measurement to assure that the WiFi state or the RRC state is at the awake state and the `DCH`(3G)/`CONNECTED`(LTE) state, individually.

`MobiScope` also includes modules that realize patterns **H-P4** and **T-P3** for conducting HTTP and TCP based RTT measurement at system layer.

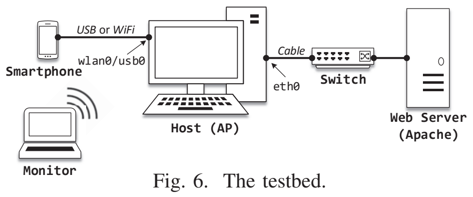


Fig. 6. The testbed.

V. Experiments

A. Testbed

As shown in Fig.6, the smartphone communicates with an Apache2 web server via a host. The host offers the smartphone two access methods, namely, USB tethering and WiFi (i.e., an AP), and it uses Linux traffic control (TC) and `netem` to emulate various capacity and additional delay. USB tethering offers stable wired connection with very low latency. In contrast, WiFi channel may have variable and large delay due to contention and signal variance. Hence, when examining factors except WiFi state transitions, we connect the smartphone to the host via USB tethering to avoid unexpected noise. Otherwise, we use WiFi access. Besides collecting timestamps in Android, we also record the timing information of packet transmissions at all possible vantage points in the testbed.

We have two smartphones: Samsung S3 with Exynos 4412 Quad and Murata M2322007 WiFi module, and LG Nexus 5 with Qualcomm MSM8974 Snapdragon 800 and Broadcom BCM4339 WiFi module. The LG Nexus 5 runs either official

Android 4.4 or Android 6.0 for DVM or ART. The Samsung S3 runs CM-11.0 (based on Android 4.4) or CM-13.0 (based on Android 6.0), because AOSP builds only support Nexus devices. The apps under investigation including Fing, MobiPerf, Netalyzr, WiFi Speed Test, Internet Speed Test, Ping&DNS, PingTools, he.net, HTTPing, and Httping Tool. These tools can be found at [3]. Their implementation patterns identified by our analysis are summarized in Table I.

B. Effect of Factors in Category 1

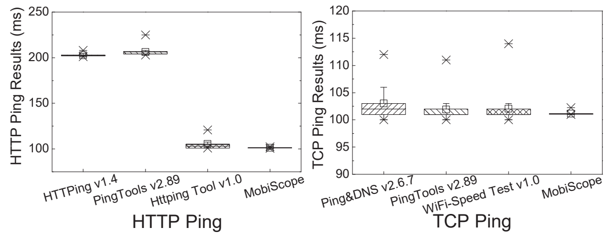


Fig. 7. HTTP/TCP-based RTT measurement by different ping apps.

In this experiment, we add 100ms delay to the USB connection between the smartphone and the host. Then we run the measurement applications sequentially and repeat for 50 times. Thus each app returns 50 measurement results for comparison. The left subfigure of Fig.7 shows the results of HTTP ping from `HTTPing`, `PingToos`, `Httping Tool`, and `MobiScope`. The right subfigure of Fig.7 illustrates the results of TCP ping from `Ping&DNS`, `PingTools`, `WiFi-Speed Test`, and `MobiScope`. Since all apps use the default `ping` to conduct ICMP ping, we compare it and `MoboScope` in Section V-E.

For HTTP ping, the results from `HTTPing` and `PingTools` are around *twice* of the real RTT. It is because `HTTPing` and `PingTools` use the patterns **H-P2** and **H-P3** respectively. Hence, their results include the time for establishing TCP connection and the time for sending request and receiving response. Although `Httping tool` claims using HTTP request and response to measure RTT, its implementation is similar to pattern **T-P3** except that it sends HTTP request *before* getting $tEnd$. Therefore, its result is similar to that from `MobiScope` but has larger variance.

For TCP ping, `Ping&DNS` and `PingTools` adopt patten **T-P2** and `WiFi Speed Test` follows pattern **T-P1**. Their results are similar to that from `MobiScope`. The reason is we use the web server's IP address instead of domain name so that apps do not need to perform DNS lookup. In other words, they measure $t_3 - t_2$ in Fig.1. And this is also the reason why `HTTPing` and `PingTools` get the similar results.

Summary. Implementation patterns may obviously bias the measurement result. Developers had better describe the implementation patterns of apps to avoid confusing the users.

C. Effect of Factors in Category 2

Measuring the delays at different layers. To profile the time for delivering a packet across different layers, we construct the cross-layer method call graph using `AppTracer`, and then select methods as the entries of different layers. Note that we use the execution time of an entry function to approximate the time used to deliver a packet at the corresponding layer.

To minimize the noise due to timestamp acquiring functions, we get the timestamps used for profiling a Java method from the system layer instead of the runtime layer. More precisely, given a Java method, we obtain the timestamps right before and after its execution by modifying *dvmInterpret()* (in `libdvm.so`) and *ArtMethod.Invoke()* (in `libart.so`) for DVM and ART, respectively. Moreover, we invoke the Java method through Java reflection so that this method will be called and returned in *dvmInterpret()* or *ArtMethod.Invoke()*, individually. Then, at runtime and system layer, we acquire timestamps through *gettimeofday()*. At kernel and device layer, we get them through kernel function *ktime_get_ts()*.

Delay due to Android Architecture. To profile the time consumed for delivering a UDP packet across different layers, we select *DatagramSocket.send()*, *sendto()*, *sys_sendto()* and *dev_hard_start_xmit()* as the entries of runtime/system/kernel/NIC layer, as shown in Fig.5. *sys_sendto()* and *dev_hard_start_xmit()* are kernel functions. The former is the kernel implementation of *sendto()* while the latter delivers the packet to the NIC driver. Moreover, the delays at runtime/system/kernel/NIC layer correspond to T_R/T_S/T_K/T_N in Fig.5.

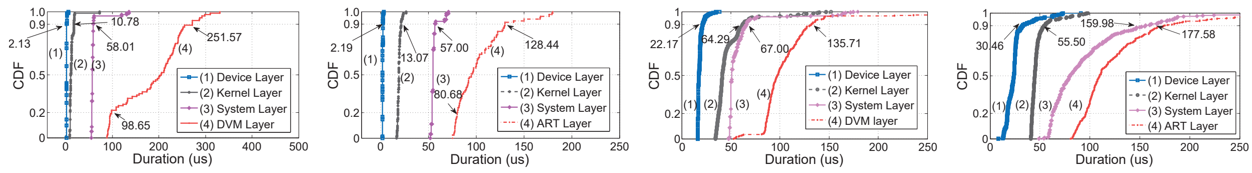
Fig.8 shows the CDF of the time consumed at different layers for sending a UDP packet from apps to network. This experiment is repeated for 100 times. We can see that the delays at the device, kernel and system layers are relatively stable and small. For example, Fig.8(a) shows that 90% delays are less than 2.13us/10.78us/58.01us at the device/kernel/system layers for Android 4.4. Similarity, Fig.8(b) illustrates that 90% delays are less than 2.19us/13.07us/57.00us at the device/kernel/system layers for Android 6.0.

DVM and ART introduce longer delay than other layers, and their values are not stable. Fig.8(a) demonstrates that in DVM 20% delays are less than 98.68us and 70% delays are in the range of [98.68,251.57]us. ART usually causes shorter delay than DVM. Fig.8(b) shows that in ART 20% delays are smaller than 80.68us and 70% delays are in the range of [80.68,128.44]us. The unstable delays caused by the runtime may be due to the kernel's process scheduling.

Delay due to monitoring mechanisms. We send UDP packets of [628, 1428] bytes from the runtime layer or the system layer, and measure the throughput with/without certain monitoring mechanisms. Table II shows that *libpcap* and *Netfilter* significantly degrade the throughput due to additional delays.

Netfilter. We insert 10 string matching rules into *iptables* before conducting the experiments. Table II shows that the *Netfilter* rules result in throughput degradation no matter the measurement is conducted at the runtime or the system layer. For example, in LG Nexus 5 running Android 6.0, the throughput can achieve 136.40Mb/s and 215.37Mb/s at the runtime and the system layer, respectively, if the 1428-byte packets are used. However, if the *Netfilter* rules are applied, the throughput drops to 68.42Mb/s and 75.03Mb/s, individually.

Libpcap. Since *Tcpdump* uses *Libpcap* to capture packets, we turn it on or off for getting the results with or without *Tcpdump*. Table II illustrates that *Tcpdump* also brings obvious overhead to the packet transmission. For example, the throughput measured by 1428-byte packets at the Android



(a) LG Nexus 5 (Android 4.4). (b) LG Nexus 5 (Android 6.0). (c) Samsung S3 (Android 4.4) (d) Samsung S3 (Android 6.0)

Fig. 8. Time consumed for sending a UDP packet across different layers.

TABLE II
PACKET TRANSMISSION CAPACITY MEASURED WITH DIFFERENT CONFIGURATIONS (NORMAL/WITH LIBPCAP/WITH 10 NETFILTER RULES).

System		Android 4.4		Android 6.0	
Device		Samsung S3 (Mb/s)	LG Nexus 5 (Mb/s)	Samsung S3 (Mb/s)	LG Nexus 5 (Mb/s)
Runtime layer	600 byte	54.99±6.1 / 14.86±5.3 / 12.57±4.4	68.04±5.4 / 44.98±5.3 / 32.87±5.3	37.84±4.1 / 9.00±4.3 / 7.08±3.3	65.03±5.7 / 45.35±4.5 / 26.22±5.1
	1400 byte	87.84±8.3 / 26.56±5.2 / 27.98±3.2	134.45±8.1 / 75.21±9.2 / 70.28±8.3	75.35±9.1 / 11.12±3.2 / 15.51±4.1	136.40±9.2 / 77.41±9.3 / 68.42±7.2
System layer	600 byte	90.36±8.9 / 41.40±4.4 / 37.70±5.2	112.46±7.3 / 60.94±7.1 / 32.30±8.9	32.89±8.3 / 15.48±3.3 / 11.77±3.9	72.53±7.3 / 49.42±6.2 / 38.56±7.9
	1400 byte	167.03±9.2 / 56.92±5.3 / 66.91±4.9	294.11±13 / 127.35±9.2 / 73.90±9.3	84.17±7.8 / 34.85±5.1 / 28.50±4.6	215.37±10 / 85.60±4.9 / 75.03±4.6

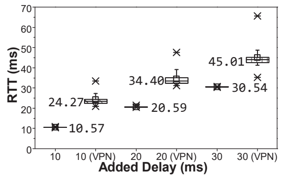


Fig. 9. RTT measured with and without *VPNservice*.

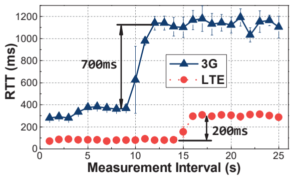


Fig. 10. RTT measured with different intervals (Cellular).

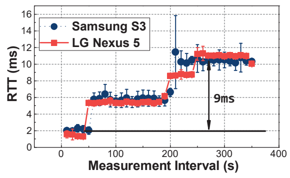


Fig. 11. RTT measured with different intervals (WiFi).

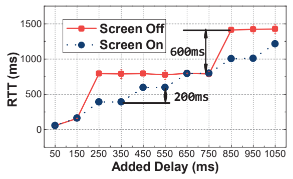


Fig. 12. RTT measured with different added delays (WiFi).

4.4/6.0 runtime layer in LG Nexus 5 is 134.45/136.40Mb/s without *Tcpdump*, but the value drops to 75.21/77.41Mb/s when *Tcpdump* is used. Its effect is less than that of *Netfilter*.
***VpnService*.** We launch *LocalVPN* to measure the effect of *VpnService*. Since preliminary experiment shows that *VpnService* will significantly delay or drop packets, we only evaluate the additional delay caused by it. More precisely, to emulate different network conditions, we add [10,20,30]ms delay and use `MobiScope` to measure RTT 100 times. Fig.9 shows that *VpnService* causes large additional delay to the results. The differences between the mean values of the measurement results with/without VPN is around 14ms.
Summary: The cross-layer nature of Android brings non-negligible delay to the measurement results for both DVM and ART. Note that packets sent across layers experience different delays and conducting measurement in kernel suffers less delay. Besides, the popular monitoring mechanisms also introduce obvious delay to the measurement results. Note due to its close relation with Wifi state transition, the power management experiment is put in the subsection below.

D. Effect of Factors in Category 3

We run `MobiScope` to perform ICMP-based RTT measurement at the system layer with different time intervals for estimating the delay due to NIC state transition.
Delay due to cellular state transition. As shown in Fig.10, when the smartphone connects to 3G network, the measured RTTs are substantially inflated when the measurement interval exceeds 10s. Since there is no background traffic generated during the measurement period, we can infer that the RRC state switches from DCH to IDLE when the idle time exceeds 10s. Similarity, we can learn that the RRC transition from IDLE to DCH consumes around 700 ms.

When the smartphone connects to LTE network, Fig.10 illustrates that the measured RTTs are increased by 200ms when the measurement intervals (i.e., idle time) become larger than 15s. Therefore, we can infer that the timeout value of the CONNECTED state is 15s and the delay introduced by state switch from IDLE to CONNECTED is more than 200ms. Fig.10 also shows that the state transition of either LTE or 3G can cause significant delay to the measurement results, and the delay resulted from 3G state transition is more than three times of the delay brought by LTE state transition.
Delay due to WiFi state transition. We describe the result of sending packets and that of receiving packets, individually.
Packet sending. Fig.11 shows the measured RTT with different measurement intervals. For Samsung S3, the RTT values roughly center around 2ms, 6ms and 10ms, which are separated at intervals of 50ms and 200ms. For Nexus 5, the RTT values roughly center around 2ms, 5ms, 8ms, and 11ms. Moreover, they stay almost constant within a range of measurement intervals, and abruptly increase when the interval reaches a certain value (i.e., 50ms, 200ms, and 250ms). Both Samsung S3 and Nexus 5 devices have a state transition when the measurement interval is 200ms.

By analyzing the captured 802.11 frames, we find that after 200ms of inactivity (i.e., *ListenInterval*=200ms) the WiFi interface goes to sleep. Therefore, if the measurement interval exceeds 200ms, the WiFi interface goes to sleep after finishing an RTT measurement and then wakes up for the next measurement. Before conducting a new measurement after wake up, the WiFi interface needs some time to send a null data frame (NDF) with power management bit set to 0 to retrieve frames buffered at the AP. We can see that such additional time does not exist when measurement intervals are less than 200ms. Moreover, the jumps at 50ms for Samsung S3 and both 50ms and 250ms for Nexus 5 suggest other state transitions. Due to the lack of the WiFi driver's source code, we could just conjecture that they have proprietary PSM mechanisms, and we will investigate it in future work.

For TCP ping, the tools `Ping&DNS` and `PingTools` adopt T-P2 implementation pattern and `WiFi Speed Test` is implemented based on T-P1 pattern. As we represent the web server in the test bed using IP address instead of domain name, the result of these three tools are $t_3 - t_2$ in Fig.1.

Packet receiving. Fig. 12 shows the RTT values when the screen is on and off, respectively. By analyzing the captured 802.11 frames, we find that the WiFi interface adopts two different PSM schemes when screen is on and off. Specifically, when the screen is on, the WiFi interface wakes up with a time interval of 200ms to check buffered packets in the AP. When the screen is off, such a time interval is 600ms.

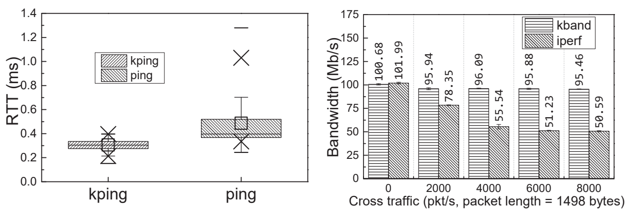
Summary: The NIC (i.e., 3G, LTE, WiFi) state transition can introduce obvious delays to the measurement. LTE may cause less delay than 3G, and different WiFi chipsets have different effects. Moreover, the effects are not the same when the smartphone is at different status (i.e., screen on/off).

E. Evaluation of `kping` and `kband`

In this experiment, we connect the smartphone to the host via USB tethering and limit the bandwidth to 100Mbs. Setting the server as the destination, we run `kping` and *ping* to measure the RTT. Fig.13(a) shows that the RTTs measured by `kping` center around 0.3ms with an upper bound of 0.4ms whereas the RTTs measured by *ping* are highly dispersed and fluctuate between 0.3ms and 1.3ms.

To evaluate the accuracy of capacity measurement, we run `kband` and `iperf` (native program) on the smartphone, both of which send 1498-byte UDP packets to the server for measurement. Moreover, to generate cross traffic, we run D-ITG [25] in the smartphone, which sends UDP packets through raw socket at the system layer. The rate of cross traffic ranges from 0pkt/s to 8000pkt/s with an incremental step of 2000pkt/s. Fig.13(b) shows the capacity measured by `iperf` and `kband` with varying cross traffic from the smartphone to the server. We observe that when the volume of the cross traffic increases both `iperf`'s and `kband`'s accuracy of capacity measurement decreases. However, compared with `iperf`, `kband` is robust to cross traffic when measuring capacity. For example, when the cross traffic reaches 8000pkt/s, `kband` can still achieve a high accuracy with only 4.54% underestimation whereas `iperf` underestimates the capacity up to 49.54%.

Summary. Conducting measurement in the kernel layer can obtain more accurate and stable results. Moreover, it is more robust to cross traffic than the measurement at above layers.



(a) `kping` vs. `ping`. (b) `kband` vs. `iperf`.

Fig. 13. Evaluation of `kping` and `kband`.

VI. Examining Descriptions of Measurement Apps

Since ambiguous descriptions may confuse users what is measured, we construct enhanced descriptions, each of which comprises a statement based on the static and dynamic analysis and a figure showing how the measurement is performed. To evaluate the clarity of the official description and the enhanced one, we design questionnaires for 10 measurement apps [3] and conduct user studies.

A. Questionnaire design

Fig.14 shows a snippet of the questionnaire of one particular app. The figure shows how the app actually conducts the measurement. The official description and the enhanced one are presented in random orders unknown to respondents. In other words, DESC1/DESC2 could be the official one or the enhanced one. Such randomness is to avoid biased result due to direct or indirect exposure of the official description. Note that the official description is fetched from Google Play. For each description, we ask respondents whether it is clear. They can only select one answer from "Abs YES", "YES", "Neutral", "NO", "Abs NO", resulting in integer scores from five to one, respectively. We recruit 29 respondents with knowledge of computer networks from three cities because without such knowledge, a respondent may not understand the descriptions.

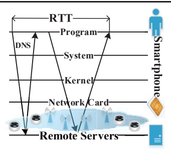
Apps	Procedure of the ping task	DESC1	Is DESC1 clear?	DESC2	Is DESC2 clear?
Ping&DNS	 RTT Program DNS System Kernel Network Card Remote Servers Smartphone	Ping a server (via ICMP over IPv4 or IPv6 and TCP), DNS lookup (with geographical lookup of IP addresses)	☐ Abs YES ☐ YES ☐ Neutral ☐ NO ☐ Abs NO	Ping a server using TCP packets and the RTT result contains the time consumed to perform one DNS lookup and build one TCP connection	☐ Abs YES ☐ YES ☐ Neutral ☐ NO ☐ Abs NO

Fig. 14. A snippet of the questionnaire.

B. Result analysis

We first investigate the clarity of official descriptions. For the 290 answers (10 answers in each questionnaire), 25.8% (i.e., (13+62)/290) of them are "Abs YES"/"YES" (i.e., the respondents think that the official description is clear). 7.9% (i.e., 23/290) and 27.9% (i.e., 81/290) of them are "Abs NO" and "NO", respectively. For the enhanced descriptions, 71.7% (i.e., (97+111)/290) of the answers are "Abs YES"/"YES" and only 5.2% (i.e., (5+10)/290) of them are "Abs NO"/"NO".

We contrast the official description with the enhanced one derived from static and dynamic analysis. We find that the description of `Httping Tool` contradicts its implementation because it claims to measure HTTP latency, but its implementation measures the latency for establishing a TCP connection. The descriptions of most apps are ambiguous due to limited details. For example, `PingTools`'s description only has a simple sentence "ICMP, TCP and HTTP ping" without any details. Note that the result of its HTTP ping could be *twice* of the result of its TCP ping, because it adopts **H-P2** for HTTP ping and **T-P2** for TCP ping. Such difference will confuse users. The description of `HTTPing`, in our opinion, is the most accurate and complete. Although `Netalyzr`'s description is simple, it clearly explains the meaning of each measurement result. That is also a good practice.

VII. Related Work

Many apps for mobile network measurement have been proposed [2], [26]–[28]. However, most of them conduct the measurement *without* considering the affecting factors. Although a few studies pointed out some factors, there is a lack of a systematic investigation on them. For example, studies on the effect of mobile network protocols and WiFi on performance [7], [29]–[32] neither examine the effect on measurement apps nor take into account apps' implementation patterns and Android architecture and configurations. Note that all factors under our investigation have non-negligible impact on the result of mobile network measurement.

The most closely related work is [5]. Different from ours, they only studied the effect of DVM [5] and conducted coarse-grained analysis. For example, they attribute the additional delay of HTTP-based RTT measurement to DVM and kernel, neglecting apps' implementation patterns. Moreover, their analysis has two limitations that may bias their results. First, they monitor packets using *TcpDump*, which will introduce obvious delay as shown in Tab. II. Second, they use the timestamps of wireless frames to approximate the time when the request (or response) packet leaves (or reaches) the smartphone without considering the contention of WiFi channel, which will also bring additional delay.

Recent studies on profiling app performance [33]–[35] aim at improving user experience rather than mobile network measurement. QoE Doctor [33] diagnoses the user-perceived latency of Android apps by correlating user interaction events, network traffic and RRC states. Panappticon [34] identifies the critical paths in user transactions, and locate performance problems by capturing specified events at the user/kernel layers. AppInsight locates performance bottlenecks through critical paths in user transactions. However, it focuses on Windows mobile apps, and needs to modify the binaries [35].

VIII. Conclusion and Future Work

We conduct the *first* systematic investigation on why the measurement result from apps may not be what users expect. First, we identify new factors, revisit known factors, and propose a novel approach as well as two tools to discover these factors in proprietary apps. Moreover, we perform extensive experiments to quantify the negative effects of these factors, and develop `MobiScope` for demonstrating how to mitigate such effects. Second, we find that the measurement apps' descriptions may be ambiguous and confuse users. We construct enhanced descriptions to provide users more information on what is measured. The user studies show the improvement of the enhanced descriptions. This research sheds light on creating better mobile measurement apps and conducting expected network measurement in apps. In future work, we will improve our tools as the process is still semi-automated, and examine more apps with large-scale user studies.

IX. Acknowledgment

This work is supported in part by the Hong Kong GRF/ECS (PolyU 5389/13E, PolyU 152279/16E), the National Natural Science Foundation of China (61602371, 61202396), the HKPolyU Research Grant (No. G-UA3X), Shenzhen City Science and Technology R&D Fund (No. JCYJ20150630115257892), Natural Science Basic Research Plan in Shanxi Province (2016JQ6034), China Postdoctoral Science Foundation (No. 2015M582663), and the Fundamental Research Funds for the Central Universities of China.

References

[1] "Measuring broadband america mobile broadband services," https://goo.gl/3MmsSP, 2014.
[2] U. Goel, M. Wittie, K. Claffy, and A. Le, "Survey of end-to-end mobile network measurement testbeds, tools, and services," *Communications Surveys Tutorials, IEEE*, vol. 18, no. 1, pp. 105–123, 2016.
[3] "The measurement tools under evaluation," https://goo.gl/MGtoAO.
[4] G. Almes and S. Kalidindi, "Rfc 2681: A round-trip delay metric for ippm," Sept. 1999.
[5] W. Li, R. Mok, D. Wu, and R. Chang, "On the accuracy of smartphone-based mobile network measurement," in *Proc. IEEE INFOCOM*, 2015.
[6] S. Rosen, H. Luo, Q. A. Chen, M. Mao, J. Hui, A. Drake, and K. Lau, "Discovering fine-grained rrc state dynamics and performance impacts in cellular networks," in *Proc. ACM MobiCom*, 2014.
[7] H. Han, Y. Liu, G. Shen, Y. Zhang, and Q. Li, "Dozyap: power-efficient wi-fi tethering," in *Proc. ACM MobiSys*, 2012, p. 2012.
[8] I. Corporate, "Worldwide smartphone market will see the first single-digit growth year on record," https://goo.gl/1bJ1lt, Dec. 2015.
[9] A. Morton, "Rfc 7497: Rate measurement test protocol problem statement and requirements," Apr. 2015.
[10] H. Yao, G. Ranjan, A. Tongaonkar, Y. Liao, and Z. M. Mao, "Samples: Self adaptive mining of persistent lexical snippets for classifying mobile application traffic," in *Proc. MobiCom*, 2015.
[11] "isreachable," https://goo.gl/p3sMxA.
[12] "ART and Dalvik," https://source.android.com/devices/tech/dalvik/.
[13] "Platform versions dashboards," https://goo.gl/YRW9II, July 2016.
[14] "Vpnservice," https://goo.gl/2tqRTm.
[15] A. Razaghpanah, N. Vallina-Rodriguez, S. Sundaresan, C. Kreibich, P. Gill, M. Allman, and V. Paxson, "Haystack: In situ mobile traffic analysis in user space," http://arxiv.org/pdf/1510.01419v1.pdf, 2015.
[16] "Keeping the device awake," https://goo.gl/Kb0QZE.
[17] A. Gerber, S. Sen, and O. Spatscheck, "A call for more energy-efficient apps," *AT&T Labs Research*, 2011.
[18] 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.
[19] C. Qian, X. Luo, Y. Le, and G. Gu, "Vulhunter: Toward discovering vulnerabilities in android applications," *IEEE Micro*, vol. 35, no. 1, 2015.
[20] L. Li, A. Bartel, T. F. D. A. Bissyande, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: detecting inter-component privacy leaks in android apps," in *Proc. ICSE*, 2015.
[21] A. Lanzi, M. I. Sharif, and W. Lee, "K-tracer: A system for extracting kernel malware behavior." in *proc. NDSS*, 2009.
[22] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan notices*, 2007.
[23] L. Xue, C. Qian, and X. Luo, "Androidperf: A cross-layer profiling system for android applications," in *Proc. IEEE IWQoS*, 2015.
[24] L. Xue, X. Luo, and Y. Shao, "ktrxer: A portable toolkit for reliable internet probing," in *Proc. IEEE IWQoS*, 2014.
[25] "D-itg," http://traffic.comics.unina.it/software/ITG/.
[26] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson, "Netalyzr: Illuminating the edge network," in *Proc. ACM IMC*, 2010.
[27] A. Nikravesh, H. Yao, S. Xu, D. Choffnes, and Z. M. Mao, "Mobilyzer: An open platform for controllable mobile network measurements," in *Proc. MobiSys*, 2015.
[28] M. Wittie, B. Stone-Gross, K. Almeroth, and E. Belding, "Mist: Cellular data network measurement for mobile applications," in *Proc. ICST BROADNETS*, 2007.
[29] 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.
[30] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin, "A first look at traffic on smartphones," in *Proc. ACM IMC*, 2010.
[31] S. Deng and H. Balakrishnan, "Traffic-aware techniques to reduce 3g/lte wireless energy consumption," in *Proc. ACM CoNEXT*, 2012.
[32] L. Sun, R. K. Sheshadri, W. Zheng, and D. Koutsonikolas, "Modeling wifi active power/energy consumption in smartphones," in *Proc. IEEE ICDCS*, 2014.
[33] Q. Chen, H. Luo, S. Rosen, Z. Mao, K. Iyer, J. Hui, K. Sontineni, and K. Lau, "Qoe doctor: Diagnosing mobile app qoe with automated ui control and cross-layer analysis," in *Proc. ACM IMC*, 2014.
[34] L. Zhang, D. Bild, R. Dick, Z. Mao, and P. Dinda, "Panappticon: event-based tracing to measure mobile application and platform performance," in *Proc. CODES+ISSS*, 2013.
[35] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh, "Appinsight: Mobile app performance monitoring in the wild." in *Proc. OSDI*, 2012.