MopEye: Opportunistic Monitoring of Per-app Mobile Network Performance

Daoyuan Wu, ***Singapore Management University;*** **Rocky K. C. Chang, Weichao Li, and Eric K. T. Cheng,** ***The Hong Kong Polytechnic University;*** **Debin Gao,** ***Singapore Management University***

https://www.usenix.org/conference/atc17/technical-sessions/presentation/wu

This paper is included in the Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17).

July 12–14, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-38-6

Open access to the Proceedings of the 2017 USENIX Annual Technical Conference is sponsored by USENIX.

MopEye: Opportunistic Monitoring of Per-app Mobile Network Performance

Daoyuan Wu[1*], Rocky K. C. Chang[2], Weichao Li[2], Eric K. T. Cheng[2], and Debin Gao[1]

[1]Singapore Management University
[2]The Hong Kong Polytechnic University

https://mopeye.github.io[†]

Abstract

Crowdsourcing mobile user's network performance has become an effective way of understanding and improving mobile network performance and user quality-of-experience. However, the current measurement method is still based on the landline measurement paradigm in which a measurement app measures the path to fixed (measurement or web) servers. In this work, we introduce a new paradigm of measuring *per-app* mobile network performance. We design and implement MopEye, an Android app to measure network round-trip delay for each app whenever there is app traffic. This opportunistic measurement can be conducted automatically without user intervention. Therefore, it can facilitate a large-scale and long-term crowdsourcing of mobile network performance. In the course of implementing MopEye, we have overcome a suite of challenges to make the continuous latency monitoring lightweight and accurate. We have deployed MopEye to Google Play for an IRB-approved crowdsourcing study in a period of ten months, which obtains over five million measurements from 6,266 Android apps on 2,351 smartphones. The analysis reveals a number of new findings on the per-app network performance and mobile DNS performance.

1 Introduction

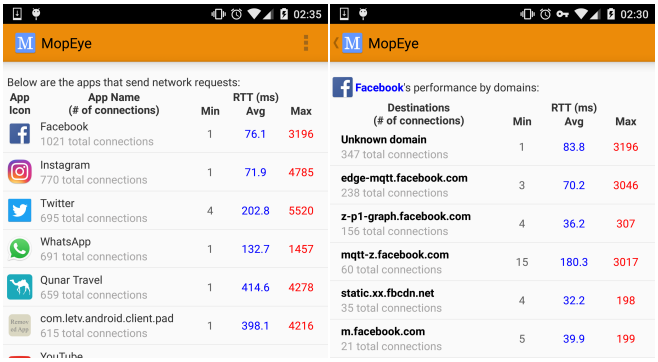
In recent years, a number of crowdsourcing platforms using smartphone apps are deployed to measure mobile network performance. MobiPerf [5] and Netalyzr [7] on Android, for example, enable users to measure a number of network performance metrics between their smartphones and remote endpoints. Using these uncoordinated network measurement performed by end users to obtain accurate and meaningful insights is still under active research [40]. Related to that, a number of speedtest services are provided for Android [13, 16], iOS [14, 15], and Windows Phone users [8, 17].

The existing mobile measurement apps, however, are still based on the landline measurement paradigm. They actively send probe packets to user-specified remote endpoints or measurement servers (e.g., M-Lab servers). Due to the diverse locations of various servers and user mobility, such landline measurement will not correlate well with the user's experience. In this paper, we propose to measure mobile network performance for each app (i.e., from user's smartphone to the app server). The *per-app* measurement not only reflects user's experience with the app but also helps diagnose application-specific problems. An effective approach to per-app measurement is to perform the measurement only when there is app traffic. Since this opportunistic measurement can be conducted automatically without user's intervention, it can facilitate a large-scale and long-term crowdsourcing of mobile network performance.

In this paper, we utilize the `VpnService` API available on Android 4.0+ [20] to implement opportunistic measurement of per-app network performance in MopEye (MObile Performance Eye), our Android measurement app. Figure 1 shows the two main interfaces in MopEye. With the `VpnService` interface, MopEye can *passively* capture the traffic initiated by all apps and forward them *actively* to the remote app servers using socket calls. Based on the `connect()` socket calls, it can estimate the round-trip time (RTT) for each app. Therefore, the measurement incurs zero network overhead, and the RTT can accurately reflect the network delay experienced by each app. Moreover, MopEye can be deployed easily, because it does not need the root privilege which is required for `tcpdump`-based passive measurement. It is also very easy to operate. Users are only required to

*Half of the work by this author was performed at The Hong Kong Polytechnic University.

[†]We thank Dr. Ada Gavrilovska 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, a grant (ref. no. H-ZL17) from the Joint Universities Computer Centre of Hong Kong, and the Singapore National Research Foundation under NCR Award Number NRF2014NCR-NCR001-012.



(a) An all-app view. (b) An individual-app view.

Figure 1: MopEye's two major user interfaces.

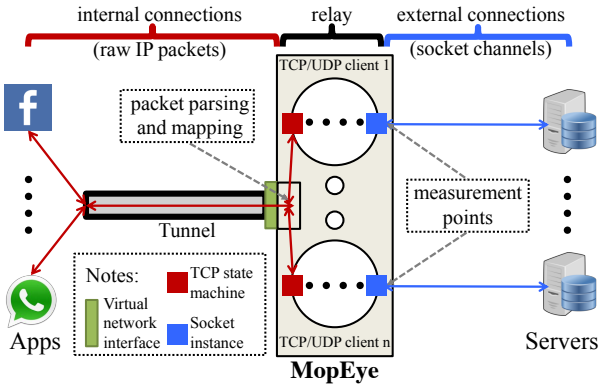


Figure 2: An overview of MopEye.

consent to enabling MopEye's VPN interface once. After that, MopEye performs the measurement opportunistically and autonomously.

The main challenge in the design and implementation of MopEye is to mitigate the impact on other apps by performing *fast* packet relaying. However, our design choices are constrained by two important restrictions: no relaying using a remote VPN server and no raw sockets which require the root privilege. To satisfy the constraints, we build our own user-space TCP/IP stack to perform packet relaying between the VPN tunnel packets and those in the socket connections. In particular, we have identified and overcome a number of serious performance degradation issues in the entire packet-relaying process. Another challenge is to obtain high measurement accuracy. Based on our evaluation, MopEye's mean RTT measurement deviates from `tcpdump`'s results by at most 1ms. Besides that, our evaluation also shows that MopEye incurs very low overhead on the throughput, battery consumption, and CPU usage.

We have deployed MopEye to Google Play [6] for an IRB-approved crowdsourcing study since May 2016. We have so far[1] attracted 4,014 user installs from 126 countries and collected the *first* large-scale per-app measurement dataset comprising 5,252,758 RTT measurements from 6,266 Android apps on 2,351 smartphones[2]. An analysis of these crowdsourced data reveals a number of new findings on the per-app and DNS network performance experienced by real users under different network types and ISPs in the wild. We also perform several case studies to diagnose the performance issues in Whatsapp, India's largest 4G ISP, and two American cellular ISPs.

2 Design of MopEye

In this section, we present an overview of MopEye and its main components. We defer the implementation details and performance enhancement to the next section.

[1]By the time of our submission on 7 February 2017.

[2]Note that many users use daily apps such as Facebook and Whatsapp. Thus, there is a large common app space among different phones.

2.1 MopEye Overview

Figure 2 presents a high-level design of MopEye. There are three main steps for MopEye to use an app's traffic to opportunistically measure the network RTT. For the outgoing traffic, MopEye first captures an app's packets through a tunnel, relays the captured packet to an external TCP connection or UDP association with a remote server, and sends the packets to the server. In the last step, MopEye calculates the time between the app's SYN and SYN/ACK packets to measure the RTT. The RTT measurement for UDP apps is similar (i.e., between query and response messages). In the following we describe each step in more details.

2.2 Packet Capturing, Parsing, and Mapping

We leverage Android's `VpnService` APIs to build a virtual network interface (green box in Figure 2) to intercept all traffic initiated from any app on the smartphone. It also receives server-initiated traffic, but for the sake of simplicity we do not discuss this traffic direction in this paper.

Android's `VpnService` APIs leverage the `TUN` virtual network device (`/dev/tun` on Android or `/dev/net/tun` on some UNIX systems) to capture packets. Figure 3 illustrates MopEye's packet capturing and relaying mechanisms for the incoming and outgoing traffic. Once MopEye builds a `TUN` interface (i.e., `mInterface` in the figure), the `TUN` device driver will capture and deliver all outgoing app packets to this interface. MopEye then obtains these packets using `mInterface`'s input stream. It is worth noting that the packets captured here are all IP packets, because a `TUN` device is essentially a virtual point-to-point IP link. MopEye parses the captured packets to obtain the IP addresses and port numbers for packet relaying.

To support per-app measurement, MopEye must also determine to which app a captured packet belongs. Although there is no API support for this socket-to-app mapping function, we find that four pseudo files in the

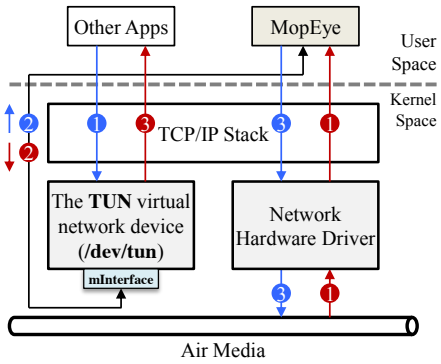


Figure 3: MopEye's packet capturing and relaying for incoming flow (red) and outgoing flow (blue). The black link represents a bi-directional flow.

`proc` filesystem (`/proc/net/tcp6|tcp|udp|udp6`) store each TCP/UDP connection's local and remote IP addresses and ports, as well as the corresponding app's UID which is a unique ID for each installed app. Moreover, using Android's `PackageManager` APIs, MopEye obtains the app's name from its UID. To reduce the overhead of this procedure, MopEye performs this operation only for the SYN packets, and the resolved names and socket addresses are cached for the subsequent data packets. Furthermore, we will present in §3.3 a new mechanism to significantly minimize the mapping overhead even for SYN packets. As for UDP packets, MopEye currently supports only DNS measurement (though it relays all UDP packets). Since DNS is system-wide, MopEye does not need to map UDP packets for now.

2.3 Packet Relaying

Relaying packets between apps and their servers efficiently is the most challenging task in the design and implementation of MopEye. Our solution to this problem is shaped by the three main considerations below.

- **Measurement objective** Since our goal is to measure the RTT between a user's smartphone and the app servers, we cannot rely on a remote VPN server to relay the application packets to their servers. Therefore, we require MopEye to relay packets within the smartphone.
- **Running on unrooted phones** Our another objective is to run MopEye on unrooted phones. Using raw sockets to relay packets to the servers is therefore not an option. Instead, MopEye must relay packets via the regular TCP/UDP sockets for the external connections. We have implemented both TCP and UDP packet relays. Due to the page limit, we describe only the TCP relay from now on.
- **User-space TCP stack** As a result of using regular TCP socket, MopEye will not be able to access the information in the TCB (Transmission Control Block [11]), such as the TCP sequence and acknowledgement numbers, from the external connections. Therefore, MopEye must create its own user-space TCP stack (in the form of TCP state machine) for the internal connections. We refer the packets transmitted in the internal and external connections to as *tunnel packets* and *socket packets*, respectively.

Splicing the two connections To relay packets in a TCP connection, MopEye "splices" the internal connection terminated by MopEye's TCP state machine and the external connection initiated by MopEye's TCP socket. Our approach is to link the state machine and the socket with two-way referencing. That is, we create a TCP client object that wraps the socket instance and include a reference to the state machine. The state machine also maintains a reference to the corresponding TCP client.

Processing tunnel packets MopEye processes the tunnel packets according to RFC 793 [11]. The processing logics for different TCP packets are summarized as follows.

- *TCP SYN:* Upon receiving a SYN packet, MopEye creates a TCP client object and uses its socket instance to perform handshake with the remote server. Only after establishing the external connection can MopEye complete the handshake with the app.
- *TCP Data:* MopEye places the data from tunnel packets to a socket write buffer and triggers a socket write event for the socket instance to handle.
- *Pure ACK:* MopEye discards pure ACK packets, because there is no need to relay them to the socket channel.
- *TCP FIN:* MopEye updates the TCP state to half closed and generates an ACK packet to the app. Meanwhile, it triggers a half-close write event for the socket instance to handle.
- *TCP RST:* MopEye closes the external socket connection and removes the corresponding TCP client object from the cached TCP client list.

Processing socket packets To handle concurrent socket instances, MopEye uses *non-blocking* `SocketChannel` APIs to communicate with the remote app servers. In particular, it uses a socket selector [32] to listen for read and write events, and handles them as follows.

- *Socket Read:* Upon detecting a read event, MopEye retrieves the incoming data from the read buffer and constructs data packets for the internal connection. In §3.4, we propose a method to improve the performance of this step. However, if this read event is for a socket close/reset, MopEye generates a FIN/RESET packet for the internal connection.
- *Socket Write:* Upon detecting a write event, the socket instance sends all the data in the write buffer to the remote server and instructs the corresponding TCP state machine to generate an ACK packet to the app. However, if this write event is for half-close, MopEye closes the external connection and generates a FIN packet to the app.

2.4 Measurement Methods

Obtaining accurate per-app RTT measurement using MopEye faces more challenges than that using the traditional active measurement apps, such as MobiPerf [5] and Ookla Speedtest [16]. There are two main challenges.

C1: Since MopEye has no control on the relayed packets, it cannot execute pre-negotiated measurement logic as in active measurement apps. This challenge is further exacerbated due to the lack of TCB information for correlating packets for measurement.

C2: Unlike other apps that have a relatively "clean" measurement environment, the performance and accuracy of MopEye can be easily affected by measurement noises, because it has to relay packets for all applications in the phone.

To address challenge C1, we identify and correlate the correct packets for computing the RTT. Among the four types of TCP socket calls (i.e., `connect()`, `read()`, `write()`, and `close()`), our evaluation using `tcpdump` shows that the `connect()` call always accurately corresponds to a single round of packets, i.e., the SYN and SYN-ACK pair. That is, invoking a `connect()` call will immediately send out a SYN packet, and the call returns just after receiving a SYN-ACK packet. In contrast, a `read()`/`write()` call may involve multiple rounds of packet exchanges, and a `close()` call may not always elicit an ACK packet from the server.

However, it is difficult for MopEye to obtain the post-`connect()` timestamp accurately due to C2. Since MopEye uses non-blocking `SocketChannel` APIs to relay packets, it has to wait for the system's notification for a received ACK. This event-based notification can introduce an additional delay up to several milliseconds if there are other pending socket events (e.g., read/write or `VpnService`'s incoming packets). We resolve this inaccuracy problem by temporarily setting the socket into blocking mode for each `connect()` call. That is, MopEye runs a `connect()` call in a temporary new thread, which we call `socket-connect` thread. Once the connection is established, MopEye resumes the non-blocking mode and switches back to the main thread listening for read and write events. As a result, MopEye can obtain an accurate post-`connect()` timestamp for the RTT measurement and, at the same time, provides efficient packet relaying. As will be explained in §3, the temporary `socket-connect` threads also give us several other benefits for optimizing MopEye's performance.

Besides the TCP-based measurement, MopEye also supports DNS. Measuring the RTT for DNS is quite straightforward. We can obtain it by measuring the time between `send()` and `receive()` UDP socket calls, which correspond to DNS query and reply, respectively. However, obtaining an accurate post-`receive()` timestamp is still difficult because of C2. We adopt a similar solution by setting up a temporary thread for a blocking-mode measurement, except that this time we run the whole DNS processing, including DNS parsing and socket initialization, in the temporary thread (instead of just doing so for the `connect()` call as in the TCP measurement). This is because DNS is an application-layer protocol built upon UDP, and processing it should not block the main thread of `VpnService`.

3 Implementation and Enhancements

We have implemented MopEye in 11,786 LOC and deployed it to Google Play [6] on 16 May 2016 for a crowdsourcing measurement study[3]. Figure 4 presents the architecture of MopEye. It has three major components or core threads (created by our `MopEyeService` that extends the Android `VpnService` class). The `TunReader` and `TunWriter` threads handle read/write for the VPN tunnel, whereas the `MainWorker` thread is responsible for all the packet processing (i.e., packet parsing, mapping, and relaying) and RTT measurement.

In this section, we will detail how we solve the challenges of implementing `TunReader`, `TunWriter`, and `MainWorker`, particularly our methods of enhancing MopEye's performance. For better reading and quick reference, we include the subsection numbers in the corresponding components in Figure 4. Among them, §3.1 and §3.5.2 present solutions generic to all VPN-based apps on Android, whereas the rest can benefit various VPN-based traffic inspection systems on different OSes.

3.1 Zero-delay Packet Retrieval from the VPN Tunnel

Reading packets from the VPN tunnel is straightforward, but it is very challenging to fast-retrieve the packets under the existing Android VPN programming paradigm. To illustrate this problem, we use a code snippet from ToyVpn [19], a representative VPN client in the official Android SDK sample code. The code[4] shows a 100ms sleep before executing each `read()` call. The purpose of this sleep is to reduce CPU cycles for data reading. Therefore, the sleep period is determined by the tradeoff between CPU consumption and packet retrieval delay.

We are not aware of any solution addressing this delayed VPN read problem. The ToyVpn example [19] implements an "intelligent" sleeping algorithm to partially mitigate this problem. The basic idea is to stop sleeping when detecting consecutive packet reads. The recently proposed Haystack [42] adopts a similar idea, but the system performance is not acceptable, e.g., achieving

[3] IRB approval was obtained from Singapore Management University on 9 October 2015 under application IRB-15-093-A077(1015).

[4] Due to the page limit, we skip the code here and refer interested readers to `http://tinyurl.com/ToyVPN`.

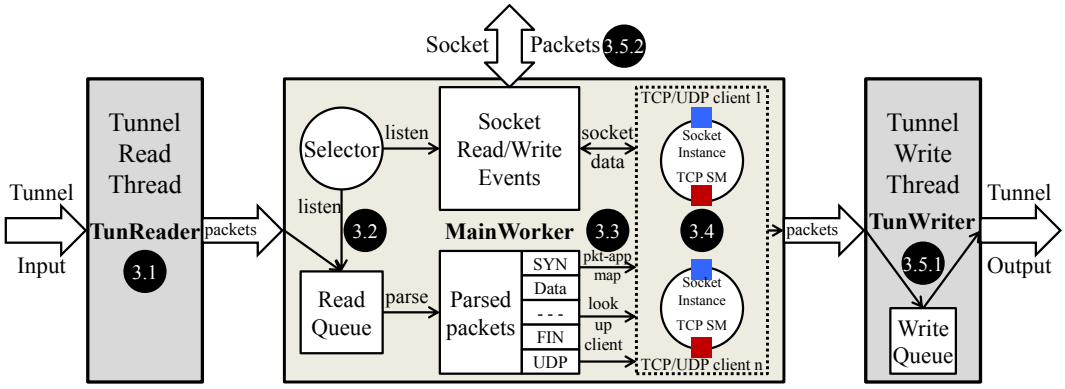


Figure 4: The architecture of MopEye.

only 17.2Mbps throughput from a 73Mbps upload link. PrivacyGuard [46], another system using `VpnService`, simply sets the sleep interval to 20ms.

We propose to fundamentally solve this problem by putting the VPN `read()` API into a blocking mode. That is, each `in.read()` call will be blocked until a packet is retrieved from the tunnel. This will effectively relieve the CPU from checking for data continuously. As a result, we must run the VPN `read()` API in a dedicated thread, i.e., `TunReader` in MopEye, and the retrieved packets will be put in a read queue shown in Figure 4.

Unfortunately, there is no API provided for setting the blocking mode of the VPN interface's file descriptor until Android 5.0. To implement our idea also for Android 4.0 to 4.4, we propose the following two solutions. First, at the native code level, we can invoke the `fcntl()` API with the `F_SETFL` command to set the blocking mode. Second, we can leverage Java reflection to invoke a non-API function called `setBlocking` in the unexported `libcore.io.IoUtils` class. We verify that this private function exists on Android from its inception.

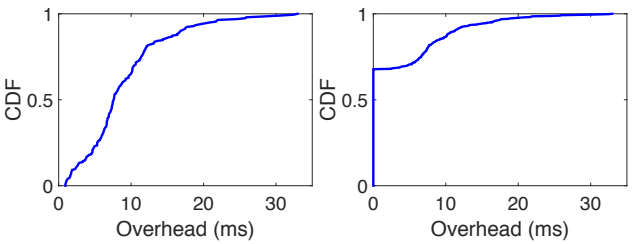
Although we can achieve zero-delay packet retrieval, there is a side effect of not being able to *timely* stop the `TunReader` thread in a blocking mode. We have tried the `Thread.interrupt()` API, but it does not work because in the absence of incoming packets the `read()` call will be blocked. To address this issue, we send a dummy packet to the VPN tunnel to release the blocked `read()` call. The dummy packet can be sent by MopEye itself for Android versions below 5.0. For Android 5.0+, however, MopEye no longer has the capability of letting its own packets go through the VPN tunnel due to the need of calling `addDisallowedApplication(mopeye)` to improve the performance (see §3.5.2). The only solution is to trigger a network request from other apps. After careful consideration, we use Android DownloadManager APIs [3] to stably trigger dummy download requests.

3.2 Monitoring Selector and Read Queue

As shown in Figure 4, we use a socket selector to listen for non-blocking read/write events from each socket instance and a read queue for receiving tunnel packets from `TunReader`. Being implemented as a single thread, `MainWorker`, however, cannot monitor both the socket selector and the tunnel read queue at the same time. To circumvent this problem, we leverage the existing `select()` waiting point to also monitor the read queue. That is, `TunReader` will issue a `Selector.wakeup()` event whenever it adds a new packet to the read queue. As a result, when the selector is woken up, MopEye will check for both socket and tunnel events, because either could have activated the selector. Moreover, to process the events timely, we interleave the code for checking these two types of events.

3.3 Lazy Packet-to-App Mapping

As presented in §2.2, MopEye performs a packet-to-app mapping for SYN packets in order to obtain per-app network performance. Our evaluation, however, shows that such mapping is expensive. Figure 5(a) shows the cumulative distribution function (CDF) of the overhead for parsing `/proc/net/tcp6|tcp` for each SYN packet. The experiment was performed on a Nexus 6 phone, containing 196 samples, and in the experiment we browsed a list of websites using the Chrome app. Over 75% of the samples required more than 5ms for the parsing; over 10% of them needed even more than 15ms. Furthermore, the overhead will increase with the number of active connections in the system.



(a) Before the lazy mapping. (b) After the lazy mapping.

Figure 5: CDF plots of packet-to-app overhead per packet.

We propose a *lazy* mapping mechanism to address this problem. First, we defer the mapping from the main thread to each temporary `socket-connect` thread mentioned in §2.4. Moreover, the mapping is performed

only after the connection is established or failed, thus not affecting the timely TCP handshake on the application side. Second and more importantly, we develop an efficient mapping algorithm that performs less `proc` file parsing. Our mapping algorithm is based on the observation that for multiple concurrent `socket-connect` threads, it is sufficient to let only one thread perform the parsing. Other threads just check and/or sleep to wait for the working thread to retrieve the mappings for them. We choose the sleep period of 50ms which is sufficiently large when compared with the parsing overhead in Figure 5(a). The evaluation results show that such a *lazy* mapping algorithm is very useful for scenarios like web browsing. For a total of 481 temporary `socket-connect` threads in a web browsing scenario, only 155 of them need to perform parsing. Moreover, the algorithm helps avoid the mapping overhead in the other 326 threads, i.e., achieving 67.8% mitigation rate as shown in Figure 5(b). Besides improving the mapping performance, it also helps reduce the CPU overhead.

Haystack [42] briefly mentions that they use *cache* to minimize the mapping overhead. However, cache-based mechanism could cause *inaccurate* packet-to-app mapping results. For example, both the Facebook app and accessing Facebook by Chrome may use the same server IP and port, but their mappings are different. This problem is more noticeable for advertisement modules since the same library may be embedded in many different mobile apps. Therefore, in order to obtain an accurate mapping, we use our own lazy mapping mechanism instead of the traditional cache-based mechanism.

3.4 Tuning TCP Performance

Besides implementing the basic user-space TCP/IP stack presented in §2.3, we have identified and tuned the following performance issues for fast packet relaying.

Maximum segment size (MSS) To maximize the throughput of the internal connections, MopEye sets the MSS option to 1460 bytes in the SYN/ACK packet and sends 1500-byte IP packets to the apps.

Receive window size Another factor affecting TCP throughput is the TCP receive window. MopEye assigns the maximum of 65,535 bytes to each MopEye's socket write and read buffer. We could also use the TCP window scale option [10] to further increase the throughput but have not done so, because the existing receive window is already big enough for achieving good performance and a bigger window size will increase the buffer memory.

No congestion and flow control Since no packet loss and reordering is expected in the VPN tunnel, MopEye forwards the data packets continuously to the app without waiting for the ACKs. Moreover, upon receiving a FIN/RST packet, MopEye stops the packet forwarding immediately.

Minimizing the use of expensive calls We try to minimize the use of expensive calls during the packet processing. For example, we discover that the `register()` call [1] for registering the socket selector can sometimes be very expensive. MopEye therefore executes this call in the `socket-connect` thread *only after* completing the handshaking for the internal connection. Other examples include never performing database operations in the main thread and always avoiding the debug log output.

3.5 Fast Dispatching of Tunnel and Socket Packets

3.5.1 Dispatching Packets to the VPN Tunnel

We observe that writing packets to the tunnel is *not* always fast, partially because multiple writing threads (e.g., `MainWorker` and individual `socket-connect` thread) share only one tunnel. We use the experimental results obtained from two writing schemes in Table 1 to illustrate this problem.

- *directWrite*: Writing is performed whenever there are packets to be sent to the tunnel.
- *queueWrite*: As illustrated in Figure 4, the packets are first put in a queue. A separate writing thread is used to output the packets. This scheme is currently adopted by MopEye.

	directWrite	queueWrite	oldPut	newPut
Total	1,244	2,161	810	5,321
0∼1ms	1,202	2,147	763	5,317
1∼2ms	30	12	39	1
2∼5ms	7	2	7	1
5∼10ms	3	0	1	2
>10ms	2	0	0	0

Table 1: Delay of writing packets to the VPN tunnel under four different writing schemes.

According to Table 1, the queueWrite scheme performs much better than the directWrite scheme. Among a total of 1,244 samples in the directWrite testing, we encounter 42 large writing overheads (i.e., those larger than 1ms). The corresponding result for the queueWrite testing is only 14 out of 2,161 samples. In particular, there are five extremely large overheads (i.e., those larger than 5ms) in the directWrite samples, two of which are even over 20ms. While there are still 14 overheads of 1∼5ms for queueWrite, they do not affect the performance of `MainWorker`, because they are performed by the dedicated `TunWriter` thread.

Although the queueWrite scheme significantly reduces the writing overhead, it introduces the overhead of packet enqueuing. We find that a traditional enqueuing scheme, denoted by oldPut, has large overheads. Among the 810 oldPut samples in Table 1, 47 have an overhead larger than 1ms. Our testing shows that most of the overheads between 1∼5ms are due to the

queue's `wait-notify` delay. When there are no packets in the queue, `TunWriter` goes to sleep by calling `queue.wait()` and is woken up by `queue.notify()`. We design a new enqueuing algorithm, denoted by newPut, to mitigate such delays. The basic idea is to let `TunWriter` perform more rounds of queue checking before going to `wait()`. Specifically, we design a sleep counter to systemize this process:

- The counter is initialized to 0 and is reset to 0 each time being woken up from `wait()`.
- When there are no packets in the queue, the counter increments for every round of checking and decrements (e.g., dividing by 2) whenever detecting a nonempty queue.
- `TunWriter` sleeps only when the counter reaches a threshold.

The newPut column in Table 1 shows the effectiveness of our algorithm. Out of the 5,321 samples, only four contain 1∼5ms overheads. Compared with the oldPut scheme, the percentage of large overheads drops from 5.69% to only 0.075%. It is worth noting that the remaining two large overheads of 5∼10ms are likely due to thread competition. We also observe that such competition effect is significantly reduced, because the enqueuing operation (at the microsecond level) is much faster than tunnel writing (at the 0.1ms level).

3.5.2 Dispatching of Socket Packets

When MopEye relays packets to the external connection, a delay overhead which could be up to several milliseconds comes from the `VpnService.protect(socket)` method [18]. Before establishing socket connections with remote app servers, MopEye must call the `protect(socket)` method to ensure that the socket packets will be sent directly to the underlying network. Without this method, the socket packets will be directed back to the VPN tunnel, thus creating a data loop.

Our solution is to replace the socket-wide `protect()` API with the application-wide `addDisallowedApplication()` API. By adding MopEye into the list of VPN-disallowed applications, we do not need to invoke `protect(socket)` for each socket client. Moreover, since MopEye just needs to call `addDisallowedApplication(mopeye)` once, the call is best invoked during the initialization of MopEye to avoid impact on `MainWorker`. The limitation of this solution is that `addDisallowedApplication()` is newly introduced in Android 5.0. For older versions, MopEye still has to call `protect(socket)`. Our mitigation method is to put `protect(socket)` in each `socket-connect` thread. In this way, only the performance of the SYN packet will be affected but not the subsequent data. Furthermore, this issue will be of less importance as more devices are upgraded to Android 5.0+, currently with over 60% of devices [2].

4 Evaluation

In this section, we present two sets of evaluation results. The first is on the measurement accuracy and overhead of MopEye, and the second is a set of crowdsourcing measurement results from 2,351 active users over nine months.

4.1 Measurement Accuracy and Overhead

4.1.1 Measurement Accuracy

The first evaluation we perform is on the accuracy of RTT measurement of MopEye. In addition to the standalone measurement, we also compare MopEye with MobiPerf v3.4.0 (the latest version at the time of our evaluation), which makes active network measurements using the state-of-the-art Mobilyzer library [40]. For a fair comparison, we use MobiPerf's HTTP ping measurement [37] because, like MopEye, it also uses SYN-ACK for the RTT measurement. For each destination, we use its raw IP address instead of the domain name so that MobiPerf's accuracy will not be interfered by DNS queries. Moreover, each result is presented by the mean of ten independent runs (MobiPerf does not provide detailed results of each run). We also run `tcpdump` to provide the reference measurement results.

Destinations	MopEye (mean, in ms)			MobiPerf (mean, in ms)		
	`tcp dump`	Mop Eye*	δ	`tcp dump`	Mobi Perf	δ
Google (`216.58.221.132`)	4.26	4	**0**	4.29	16.4	**12.11**
	4.47	5.5	**1.03**	4.35	18.5	**14.15**
	5.32	5	**0**	4.85	18	**13.15**
Facebook (`31.13.79.251`)	36.55	37	**0.45**	36.39	59.5	**23.11**
	36.55	37	**0.45**	36.72	55.2	**18.48**
	38.54	38.5	**0**	46.10	63.2	**17.10**
Dropbox (`108.160.166.126`)	284.85	284.5	**0**	361.76	409.7	**47.94**
	390.94	391	**0.06**	388.94	411.5	**22.56**
	513.78	513.5	**0**	395.87	475.2	**79.33**

* We round MopEye's μs-level results to *ms*-level, e.g., 4.135ms to 4ms.

Table 2: Measurement accuracy of MopEye and MobiPerf.

Table 2 presents three sets of results for Google, Facebook, and Dropbox, which experience RTTs on different scales. The differences between the RTT measurement of MopEye/MobiPerf and that of `tcpdump` are denoted by δ. The results clearly show that MopEye has a much better accuracy than MobiPerf—MopEye's measurement deviates from that of `tcpdump` by at most 1ms, whereas MobiPerf's deviations range from 12ms to 79ms. By assessing MobiPerf's code[5], we identify three factors responsible for MopEye's higher accuracy, including using the low-level socket call and the nanosecond-level timestamp method, and most importantly, putting the timing function just before and after the socket call. We refer interested readers to our previous poster version [48] for more details.

[5] `http://tinyurl.com/PingTask`, where HTTP ping starts from the line 438.

Throughput	Baseline	MopEye	Δ	Haystack	Δ
Download	24.47	24.01	**0.46**	20.19	**4.28**
Upload	25.97	25.08	**0.89**	6.79	**19.18**

Table 3: The download and upload throughput overhead of MopEye and Haystack.

4.1.2 Measurement Overhead

To measure the overhead introduced by MopEye, we first measure the additional delay introduced to the connection establishment and data transmission in other apps when MopEye is running. For the connection time, we implement a simple tool that invokes `connect()` to measure the time taken with and without MopEye. For data packets, we use the popular Ookla Speedtest app [16] to measure the latency. Both experiments are repeatedly executed on a Nexus 4 running Android 5.0. With a 95% confidence interval, the mean delay overhead of a round of SYN and SYN/ACK packets is 3.26∼4.27ms and that of data packets is 1.22∼2.18ms. Considering that the median of all 714,675 LTE RTTs in our dataset is 76ms, the delay overhead is acceptable.

Another important metric is the download and upload throughput overhead. We compare MopEye with Haystack [42] v1.0.0.8 (the latest version at the time of our evaluation), which uses the `VpnService` API to detect privacy leaks in app traffic. For a fair comparison, we do not enable Haystack's TLS traffic analysis for all its experiments. We use the Ookla Speedtest app as the reference tool to measure the throughput with and without MopEye/Haystack. All three experiments are repeatedly conducted in a dedicated WiFi network which provides very strong signal strength and stable throughput at around 25Mbps for both download and upload links.

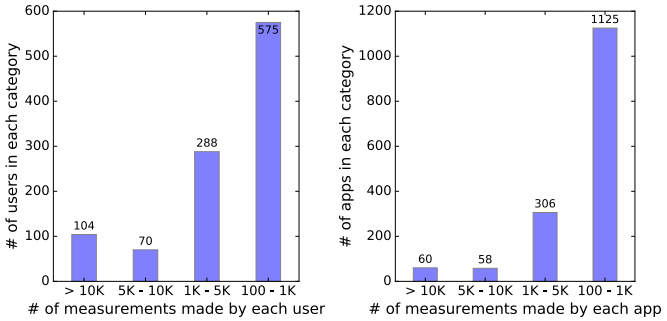
Table 3 presents the throughput results with Δ denoting the difference from our baseline using Speedtest. The results clearly show that MopEye achieves a much better throughput performance than Haystack. MopEye's throughput deviates from the baseline by less than 1Mbps, whereas that for Haystack ranges from 4Mbps (for the download link) to 19Mbps (for the upload link). In particular, we find that Haystack's throughput degrades significantly (e.g., 11.63Mbps for the download and 3.74Mbps for the upload) if we do not restart it for the next run. Therefore, in order to obtain the Haystack results in Table 3, we reset Haystack's VPN interface before each test. We attribute our superior results to the major challenges addressed in §3.

4.1.3 Resource Consumption Overhead

We now summarize the resource consumption overhead of MopEye and Haystack with a Nexus 6 playing a high-definition YouTube video for around one hour. According to Table 4, MopEye's resource consumption overhead is lower than that of Haystack in terms of CPU, battery, and memory. In particular, the CPU overhead with Haystack is over 9%, mainly because Haystack has to keep executing the VPN `read()` regardless there are app packets to be relayed or not. Moreover, we argue that the 1% battery overhead of MopEye is not contributed only by MopEye, because, with MopEye enabled, YouTube is no longer considered using the network interface by the system battery benchmark.

Scenario	Resource	Overhead	
		MopEye	Haystack
Playing a 58-minute high-definition (1080p) YouTube video	CPU	2.74%	9.56%
	Battery	1%	2%
	Memory	12MB	148MB

Table 4: The resource overhead of MopEye and Haystack.



(a) By user. (b) By app.

Figure 6: Number of measurements performed by each user/app that contribute to at least 100 measurements.

4.2 Crowdsourcing Measurement Results

Our MopEye deployment on Google Play has attracted 4,014 user installs from 126 countries since May 2016. In this section, we first describe the dataset used in this paper and then present our measurement analysis to underline the value of MopEye's opportunistic per-app measurement.

4.2.1 Dataset Statistics

By deploying MopEye for over ten months, to the best of our knowledge, we have collected the *first* large-scale per-app measurement dataset. Our analysis in this paper is based on the MopEye data received between its launch on 25 May 2016 and 3 January 2017. Our dataset covers a wide spectrum of devices, countries, and apps, and includes over 5 million RTT measurements.

User/Device coverage: The dataset includes a total of 2,351 devices that performed at least one measurement. Figure 6(a) shows the number of measurements performed by 1,037 devices each of which conducted at least 100 measurements. Although most of them are in the range of 100–1K, 462 of them (45%) contribute from 1K to more than 10K measurements each. This shows a significant number of consistently active users. Moreover, these user devices cover 922 different phone models, manufactueres of which include Samsung, HTC, LG, Motorola, Huawei, XiaoMi, and others. This evidences that MopEye can support a wide range of Android phones in the market.

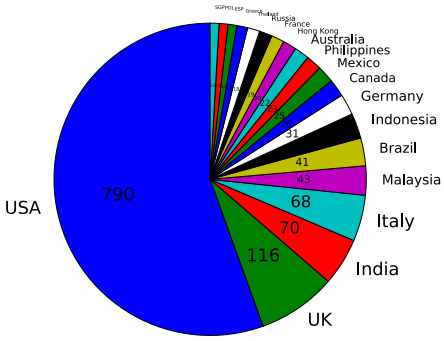


Figure 7: Distribution of the top 20 MopEye user countries.

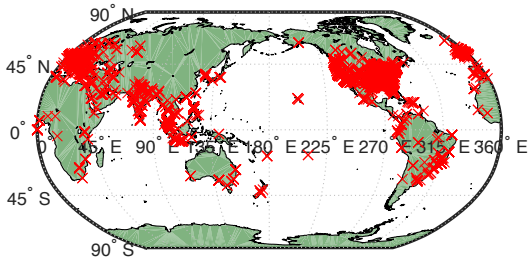
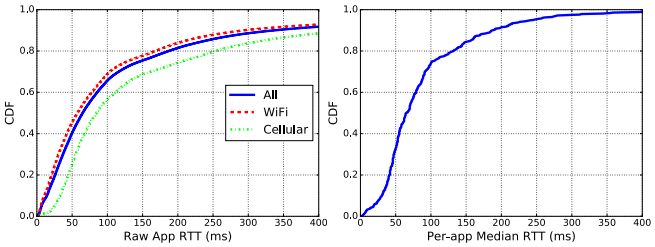


Figure 8: Locations of conducting the MopEye measurement.

Country distribution: Users in our dataset come from 114 countries worldwide. Figure 7 shows the distribution of the top 20 user countries, including the United States (790 users), United Kingdom (116 users), India (70 users), and Italy (68 users). Moreover, Figure 8 plots 6,987 geographical locations where the MopEye measurements were conducted. The figure visually shows that our dataset covers a large populated area, notably the North America, Europe, India, coastal regions of South America, Southeast Asia, and the Pacific Rim.

Applications measured: This dataset includes measurement on 6,266 apps. Figure 6(b) shows the distribution of the number of RTT measurements performed by each app that contributes at least 100 measurements, with a total of such 1,549 apps. Similar to Figure 6(a), most of them contribute 100–1K measurements, and 424 of them have between 1K and more than 10K measurements. The most popular (in terms of the number of times being measured) apps include social networking apps such as Facebook, Instagram, and WeChat, and system built-in apps such as YouTube and Google Play.

Measurements collected: The dataset contains a total number of 5,252,758 RTT measurements. Among them, 3,576,931 are measurements for TCP connections used by the apps, and the remaining 1,675,827 are for DNS measurements. Altogether they cover 106,182 destination IP addresses, 35,351 destination server domains, 2,427 destination server ports, and 943+ DNS servers. The most accessed domain is `graph.facebook.com` with 142,873 connections.



(a) All apps' raw RTTs. (b) Top 424 apps' median RTTs.

Figure 9: CDF plots of apps' raw RTTs and median RTTs.

4.2.2 Per-app Measurement Analysis

We now present the 3,576,931 per-app measurement results, which characterize the network performance experienced by different apps under different network types and ISPs in the wild. We envision ways of using the analysis results to improve the mobile network performance. For example, we reported our measurement results of WeChat to help Tencent (developer of the WeChat app) solve a misconfiguration problem [48].

Overall results. We first present the overall app performance in our dataset by plotting the distribution of apps' raw and median RTTs in Figure 9. Figure 9(a) shows the CDF plot of all 6,266 apps' raw RTTs, in which we further distinguish between WiFi and cellular access. Overall, the performance experienced by mobile users is good with a median RTT of 65ms (i.e., the value at the 0.5 line in Figure 9(a)). Moreover, ∼40% of the RTTs are below 50ms and ∼60% of the RTTs are below 100ms. However, we can still observe ∼20% of them suffering from relatively long RTTs (>200ms), and ∼10% at exceedingly long RTT (>400ms). In this dataset, WiFi shows superior performance than that on cellular networks. The median RTTs for WiFi, cellular networks (including 2G, 3G, and LTE), and LTE alone are 58ms, 84ms, and 76ms, respectively.

Figure 9(b) plots the median RTT distribution of 424 apps that have more than 1K measurements each (see Figure 6(b)). We choose the median over the mean value because the median is less affected by RTT outliers. The dataset also has enough measurements for each app, making the median a reliable measure. The figure shows that more than 70% of the apps experience less than 100ms in their RTTs. However, there are ∼10% of the apps suffering from more than 200ms of RTT.

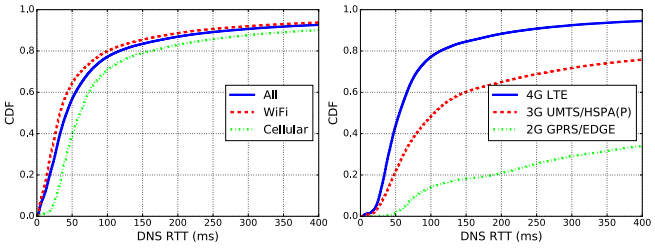
Representative apps' performance. We next study the network performance of representative apps that are frequently used in our daily life. Table 5 lists 16 such apps in five categories. For each app, we present its total number of RTT measurements and the median RTT. Most of these apps exhibit very good network performance. For example, Instagram, WeChat, Google Play Store, YouTube, and Amazon have a median RTT below 60ms. We also notice that the median RTT of Whatsapp is larger than 100ms. Next we present two case studies in more depth.

Category	Apps	# RTT	Median RTT
Social	Facebook	215,769	61ms
	Instagram	38,640	50.5ms
	Weibo	28,905	43ms
	Twitter	11,407	56ms
Commu-nication	WeChat	61,804	36ms
	Facebook Messenger	42,408	42ms
	Whatsapp	32,372	133ms
	Skype	16,264	76ms
Google	Google Play Store	100,115	48ms
	Google Play services	60,805	37ms
	Google Search	35,858	45ms
	Google Map	19,996	38ms
Video	YouTube	99,895	32ms
	Netflix	28,302	33ms
Shopping	Amazon	18,313	59ms
	Ebay	16,114	70ms

Table 5: Network performance of 16 representative apps.

Case 1: The vast majority of *.whatsapp.net domains do not perform well in many networks. Whatsapp employs a total of 334 `whatsapp.net` domains as its server domains, but the median RTT of all these domain traffic is as high as 261ms. Specifically, the median RTTs for all, except three, are larger than 200ms. The median RTTs for those three domains (starting with `mme`, `mmg`, or `pps`) are less than 100ms. According to our analysis, the three domains are deployed in the Facebook CDN, whereas the other 331 domains are with SoftLayer Technologies, a server hosting provider. Furthermore, we analyze the median RTTs on these 331 `whatsapp.net` domains in 20 most accessed networks (11 WiFi and 9 LTE networks) that have at least 100 measurements each. The results show that only two networks can achieve less than 100ms of RTT (77.5ms for a WiFi network and 56ms for the Verizon 4G network), six networks in the 100–200ms interval, eight networks in between 200ms and 300ms, and four networks with RTTs over 300ms. Moreover, our manual Ping tests from Singapore and Hong Kong to those domains report a latency of ∼250ms. All of the above show that there is much room for Whatsapp to improve their `whatsapp.net` network performance.

Case 2: Jio, India's largest 4G ISP, fails to provide acceptable performance to many app domains. In the course of analyzing the Whatsapp case, we find that Jio provides poor performance to many app server domains. Among all the ten 4G ISPs with more than 10K measurements, Jio is the only one that has a median RTT larger than 100ms. The median RTT of its 76,717 RTT measurements is as high as 281ms. Considering that the median RTT of its DNS measurements is only 59ms, the root cause lies very likely in its LTE core network. Moreover, our analysis of 115 domains (that have 100+ measurements each) in Jio finds that only 19 domains' median RTTs are less than 100ms, whereas the median RTTs of 67 domains are over 200ms, 57 domains over 300ms, and 24 domains even over 400ms. We further confirm that Jio's poor performance is not due to the performance of the app servers. It is because out of the 71 domains that have 100+ measurements each in both Jio and non-Jio LTE networks, 63 of them have much better latency (138ms less than Jio on average) with non-Jio LTE networks.



(a) All results. (b) Cellular results.

Figure 10: CDF plots of DNS measurement results.

4.2.3 DNS Measurement Analysis

Next we analyze the 1,675,827 DNS measurements received from 943+ WiFi and cellular DNS servers.

Overall results. Figure 10(a) shows the CDF plot of all measured DNS RTTs. According to the overall distribution, the DNS performance for mobile networks in the wild is good with a median of 42ms, and around 80% of DNS RTTs are less than 100ms. The DNS RTTs are in fact much better than the per-app performance by comparing Figure 10(a) with Figure 9(a). For example, 80% of per-app RTTs are less than 200ms, two times higher than DNS. This is probably because ISPs usually deploy local DNS servers. Additionally, we notice that WiFi's DNS RTTs are consistently lower than the overall results with a median of only 33ms; whereas that of cellular networks is 61ms. This indicates that the first-hop performance of WiFi is generally better than cellular networks.

We plot the detailed results for 2G, 3G, and 4G cellular networks in Figure 10(b). The CDF plots show clearly the performance difference among the three. More specifically, the median DNS RTT of 4G is 56ms; whereas that of 3G and 2G are as high as 105ms and 755ms, respectively. Most of the devices in our measurement use 4G—around 80% of DNS RTTs come from 4G. This also explains why the CDF plot for 4G DNS RTTs is close to that of all cellular RTTs.

Major 4G ISPs' DNS performance. We now take a closer look at the DNS performance of major 4G ISPs. Table 6 lists the performance of 15 LTE operators that have most DNS RTTs in our dataset. First, we notice that there is no clear correlation between the country and DNS performance. For example, the performance of most American ISPs, three Hong Kong ISPs, and two Malaysia ISPs are similar. Second, the majority of 4G ISPs provide good DNS performance with the median RTTs in 40-60ms. The only three outliers are the good-performer Singtel, and the poor-performers Cricket and U.S. Cellular. To gain a better understanding, we further study these three ISPs along with Verizon, a representative of other ISPs.

ISP Name	Country	# RTT	Median RTT
Verizon	America	80,227	46ms
Jio 4G	India	52,397	59ms
AT&T	America	51,421	53ms
Singtel	Singapore	34,609	27ms
Boost Mobile	America	21,854	50ms
Sprint	America	20,878	51ms
3	HK (China)	14,354	53ms
MetroPCS	America	13,282	60ms
T-Mobile	America	9,084	45ms
CMHK	HK (China)	5,820	50ms
Celcom	Malaysia	4,120	56ms
CSL	HK (China)	3,099	61ms
Cricket	America	2,822	93ms
Maxis	Malaysia	2,419	40ms
U.S. Cellular	America	1,988	76ms

Table 6: DNS performance of 15 LTE 4G operators.

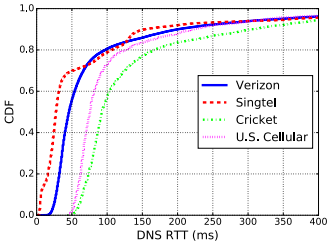


Figure 11: CDF plots for DNS performance of four LTE ISPs.

Figure 11 presents the DNS RTT distribution of the four selected ISPs with the Verizon plot as the baseline. The plots show that Singtel has an outstanding first-hop performance with 5,084 DNS RTTs less than 10ms (14.7% of its total RTTs), whereas Verizon has less than 1% of its DNS RTTs below 10ms. This is mainly because Singtel has deployed the latest upgrade of LTE, Tri-band 4G+ [21], countrywide [12]. On the other hand, the DNS performance of Cricket and U.S. Cellular clearly is worse than the baseline. In particular, the minimum RTTs of Cricket and U.S. Cellular are around 43ms, much higher than the best performance of Singtel and Verizon. They are probably using the pre-4G or near-4G implementations, because we find that around half of their DNS RTTs (64% of Cricket and 45% of U.S. Cellular) are still from non-LTE networks.

Key Takeaway: MopEye enables a large-scale deployment of per-app measurements in the wild, which help understand and diagnose the network quality of app providers and mobile networks at different granularity.

5 Related Work

Many measurement tools have been proposed to understand mobile network performance. They could be classified into crowdsourcing measurement apps (e.g., [30, 28, 25, 40]) and controlled testbeds (e.g., [47, 35, 24]). They study 3G/4G networks' RRC (Radio Resource Control) state dynamics [41, 28, 44], analyze the behaviors of cellular networks [27, 49, 33, 29], measure mobile network performance and reliability [40, 35, 22, 23, 45], and perform other measurements [37, 38]. MopEye belongs to the domain of crowdsourcing measurements. Using the `VpnService` API to perform passive network measurement, MopEye is the first app that provides per-app network performance on unrooted phones without user intervention. With MopEye, we also provide the first report of large-scale per-app network measurements.

Recently, researchers are interested in utilizing the `VpnService` API for different purposes. Nearly all of them focus on detecting privacy leakage [26] by relaying and intercepting mobile apps' traffic either in the smartphone [46, 42] or at a remote VPN server [36, 43]. Two recent works [34, 39] use a remote VPN server to identify traffic differentiation and optimize traffic volume in cellular networks. MopEye is different from all these related works in that we leverage the `VpnService` API for per-app network performance measurement. Indeed, MopEye is the first and the only one on the market that provides per-app measurement for end users. Moreover, our solutions for tackling the delayed VPN read problem (§3.1) and mitigating the VPN `protect()` delay (§3.5.2) can benefit all VPN-based apps, such as OpenVPN [9].

Due to the traffic-interception capability of `VpnService` APIs, it is important for VPN-based apps to preserve users' privacy in their design. Unfortunately, many VPN apps on the market fail to do so according to a recent study [31]. The majority of them use remote VPN servers for traffic relay, but not always in a secure fashion (e.g., no encryption for the tunnel to VPN servers, or no tunneling for DNS traffic). In contrast, our MopEye adopts the local phone-side traffic forwarding scheme, without additional risks associated with VPN servers, such as leaking user traffic. Further, unlike PrivacyGuard [46] and Haystack that perform traffic *content* inspection, MopEye makes no such attempt, let alone the TLS interception performed by those two. This may be an important factor contributing to a much higher number of MopEye installs than Haystack, which reached only 1.5K installs by the end of March in 2017 [4].

6 Conclusion

In this paper we proposed MopEye, a novel measurement app to monitor per-app network performance on unrooted smartphones. By leveraging the `VpnService` API on Android to intercept all network traffic, MopEye was able to opportunistically measure each app for its network RTT without network overhead and user intervention. We overcame a number of challenges to achieve a fast packet relaying and an accurate measurement in MopEye. We have deployed MopEye to Google Play for an IRB-approved crowdsourcing study for over ten months. By collecting and analyzing the first large-scale per-app measurement dataset, we discovered a number of new findings on the per-app and DNS network performance experienced by real users in the wild. We plan to further improve MopEye (e.g., supporting more metrics beyond RTT), and release more analysis results for app developers and ISPs to optimize their performance.

References

[1] AbstractSelectableChannel selector() — Android Developers. http://developer.android.com/reference/java/nio/channels/spi/AbstractSelectableChannel.html#register(java.nio.channels.Selector,int,java.lang.Object).

[2] Dashboards — Android Developers. https://developer.android.com/about/dashboards/index.html.

[3] DownloadManager — Android Developers. https://developer.android.com/reference/android/app/DownloadManager.html.

[4] Lumen Privacy Monitor has reached 1.5K installs on Google Play! https://twitter.com/lumen_app/status/845230899226689537.

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

[6] MopEye on Google Play. https://play.google.com/store/apps/details?id=com.mopeye.

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

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

[9] OpenVPN - Open Source VPN. https://openvpn.net/.

[10] RFC 1323 - TCP Extensions for High Performance. http://tools.ietf.org/html/rfc1323.

[11] RFC 793 - Transmission Control Protocol. https://tools.ietf.org/html/rfc793.

[12] Singtel's 4G Network Deployment History. https://www.singtel.com/personal/i/4g/why-singtel.

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

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

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

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

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

[18] The protect() API in VpnService — Android Developers. http://developer.android.com/reference/android/net/VpnService.html#protect(int).

[19] ToyVpn. https://github.com/android/platform_development/tree/master/samples/ToyVpn.

[20] VpnService — Android Developers. http://developer.android.com/reference/android/net/VpnService.html.

[21] What are 4G, 4G+ and Tri-band 4G+? https://www.singtel.com/personal/i/4g/support.

[22] BALTRUNAS, D., ELMOKASHFI, A., AND KVALBEIN, A. Measuring the reliability of mobile broadband networks. In *Proc. ACM IMC* (2014).

[23] BALTRUNAS, D., ELMOKASHFI, A., AND KVALBEIN, A. Dissecting packet loss in mobile broadband networks from the edge. In *Proc. IEEE INFOCOM* (2015).

[24] CHEN, Q., LUO, H., ROSEN, S., MAO, Z., IYER, K., HUI, J., SONTINENI, K., AND LAU, K. QoE Doctor: Diagnosing mobile app QoE with automated UI control and cross-layer analysis. In *Proc. ACM IMC* (2014).

[25] DENG, S., NETRAVALI, R., SIVARAMAN, A., AND BALAKRISHNAN, H. WiFi, LTE, or both?: Measuring multi-homed wireless Internet performance. In *Proc. ACM IMC* (2014).

[26] ENCK, W., GILBERT, P., CHUN, B., COX, L., JUNG, J., MCDANIEL, P., AND SHETH, A. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. Usenix OSDI* (2010).

[27] FALAKI, H., LYMBEROPOULOS, D., MAHAJAN, R., KANDULA, S., AND ESTRIN, D. A first look at traffic on smartphones. In *Proc. ACM IMC* (2010).

[28] HUANG, J., QIAN, F., GERBER, A., MAO, Z., SEN, S., AND SPATSCHECK, O. A close examination of performance and power characteristics of 4G LTE networks. In *Proc. ACM MobiSys* (2012).

[29] HUANG, J., QIAN, F., GUO, Y., ZHOU, Y., XU, Q., MAO, Z., SEN, S., AND SPATSCHECK, O. An in-depth study of LTE: Effect of network protocol and application behavior on performance. In *Proc. ACM SIGCOMM* (2013).

[30] HUANG, J., XU, Q., TIWANA, B., MAO, Z., ZHANG, M., AND BAHL, P. Anatomizing application performance differences on smartphones. In *Proc. ACM MobiSys* (2010).

[31] IKRAM, M., VALLINA-RODRIGUEZ, N., SENEVIRATNE, S., KAAFAR, M. A., AND PAXSON, V. An analysis of the privacy and security risks of Android VPN permission-enabled apps. In *Proc. ACM IMC* (2016).

[32] JENKOV, J. Java NIO Selector. http://tutorials.jenkov.com/java-nio/selectors.html.

[33] JIANG, H., LIU, Z., WANG, Y., LEE, K., AND RHEE, I. Understanding bufferbloat in cellular networks. In *Proc. ACM CellNet* (2011).

[34] KAKHKI, A., RAZAGHPANAH, A., LI, A., KOO, H., GOLANI, R., CHOFFNES, D., GILL, P., AND MISLOVE, A. Identifying traffic differentiation in mobile networks. In *Proc. ACM IMC* (2015).

[35] KVALBEIN, A., BALTRUNAS, D., EVENSEN, K., XIANG, J., ELMOKASHFI, A., AND OLIVEIRA, S. The Nornet Edge platform for mobile broadband measurements. In *Computer Networks* (2014).

[36] LE, A., VARMARKEN, J., LANGHOFF, S., SHUBA, A., GJOKA, M., AND MARKOPOULOU, A. AntMonitor: A system for monitoring from mobile devices. In *ACM SIGCOMM Workshop on Crowdsourcing and Crowdsharing of Big Internet Data (C2BID)* (2015).

[37] LI, W., MOK, R., WU, D., AND CHANG, R. On the accuracy of smartphone-based mobile network measurement. In *Proc. IEEE INFOCOM* (2015).

[38] LI, W., WU, D., CHANG, R., AND MOK, R. K. P. Demystifying and puncturing the inflated delay in smartphone-based WiFi network measurement. In *Proc. ACM CoNEXT* (2016).

[39] LI, Z., WANG, W., XU, T., ZHONG, X., LI, X.-Y., LIU, Y., WILSON, C., AND ZHAO, B. Y. Exploring cross-application cellular traffic optimization with Baidu TrafficGuard. In *Proc. USENIX NSDI* (2016).

[40] NIKRAVESH, A., YAO, H., XU, S., CHOFFNES, D., AND MAO, Z. Mobilyzer: An open platform for controllable mobile network measurements. In *Proc. ACM MobiSys* (2015).

[41] QIAN, F., WANG, Z., GERBER, A., MAO, Z., SEN, S., AND SPATSCHECK, O. Characterizing radio resource allocation for 3G networks. In *Proc. ACM IMC* (2010).

[42] RAZAGHPANAH, A., VALLINA-RODRIGUEZ, N., SUNDARESAN, S., KREIBICH, C., GILL, P., ALLMAN, M., AND PAXSON, V. Haystack: In situ mobile traffic analysis in user space. *CoRR abs/1510.01419* (2015).

[43] REN, J., RAO, A., LINDORFER, M., LEGOUT, A., AND CHOFFNES, D. R. ReCon: Revealing and controlling PII leaks in mobile network traffic. In *Proc. ACM MobiSys* (2016).

[44] ROSEN, S., LUO, H., CHEN, Q., MAO, Z., HUI, J., DRAKE, A., AND LAU, K. Discovering fine-grained RRC state dynamics and performance impacts in cellular networks. In *Proc. ACM MobiCom* (2014).

[45] RULA, J. P., AND BUSTAMANTE, F. E. Behind the curtain: Cellular DNS and content replica selection. In *Proc. ACM IMC* (2014).

[46] SONG, Y., AND HENGARTNER, U. PrivacyGuard: A VPN-based platform to detect information leakage on Android devices. In *ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)* (2015).

[47] WEI, X., GOMEZ, L., NEAMTIU, I., AND FALOUTSOS, M. ProfileDroid: multi-layer profiling of Android applications. In *Proc. ACM MobiCom* (2012).

[48] WU, D., LI, W., CHANG, R., AND GAO, D. MopEye: Monitoring per-app network performance with zero measurement traffic. In *Proc. ACM CoNEXT Student Workshop* (2015).

[49] XU, Q., ERMAN, J., GERBER, A., MAO, Z., PANG, J., AND VENKATARAMAN, S. Identifying diverse usage behaviors of smartphone apps. In *Proc. ACM IMC* (2011).