kTRxer: A Portable Toolkit for Reliable Internet Probing

Lei Xue[†], Xiapu Luo[†‡§], and Yuru Shao[†]
Department of Computing, The Hong Kong Polytechnic University[†]
The Hong Kong Polytechnic University Shenzhen Research Institute[‡]
{cslxue,csxluo,csyshao}@comp.polyu.edu.hk

***Abstract*—Being one of the primitives of Internet measurement and security scanning, active probing has numerous applications. While the majority of existing probing tools were designed for PCs/servers, the wide adoption of mobile devices and embedded systems bring new requirements and challenges to active probing, for example, the limited resources in those devices may affect active probing's accuracy and efficiency. However, few research studies examine such impact. In this paper, we fill the gap by investigating the effect of resource-limited devices on common packet sending/receiving techniques used by probing tools and proposing *kTRxer*, a toolkit that can be run in many devices to help probing tools achieve better accuracy and efficiency. *kTRxer* mitigates the negative effect from devices by keeping away from noise sources and achieves portability by avoiding modifying specific device drivers. We have implemented *kTRxer* with 5489 lines of C codes and conducted extensive evaluation on three platforms, including PC, broadband router, and smartphone. The experimental results show that *kTRxer* can achieve up to 10 times transmission rate and introduce much less delay noise than existing approaches.**

I. Introduction

Active probing has been widely used in numerous applications, such as network path performance measurement [1], network fault diagnosis [2], vulnerability scanning [3], to name a few. While existing probing tools were usually designed for PCs/servers that have sufficient resources, the significant increase of mobile devices and embedded systems adoption introduces new requirements and challenges to active probing. More precisely, conducting measurement or scanning in new contexts, such as mobile network and home network, usually requires running probing tools in resource-limited devices, like smartphone [4] or broadband routers [5], [6]. However, few research studies examine the effect of limited resources on the accuracy and the efficiency of active probing and how to mitigate the negative effect.

Probing tools usually involve three basic operations, including sending customized probes, receiving responses, and reacting after processing responses. The accuracy of active probing will be affected by whether probes can be transmitted at predefined time. For example, since `socket` functions, such as *send()* and *sendto()*, just put the message on buffer and the real transmission is handled by an operating system(OS)'s TCP/IP stack [7], a probing tool may not know whether the OS really sends out the packet. As another example, while tools for measuring network capacity or available bandwidth usually require that the probe packets should be sent back-to-back [8], [9], the OS may delay the transmission of some probe packets due to the competition of CPU/memory/network resources from applications, thus biasing the measurement results.

The efficiency of active probing refers to the capability of making full use of the network resources. To conduct large-scale scanning, it is desirable to reach the maximal sending rate of a network interface (NIC). For example, `ZMap` employs `raw socket` to achieve fast internet-wide scanning [3]. However, Section III shows that `raw socket` cannot achieve high efficiency in resource-limited devices.

In this paper, we examine the effect of resource-limited devices on common packet sending/receiving techniques (e.g., `socket` and `raw socket` used by probing tools and identify four kinds of limitations in existing approaches. First, the accuracy will be affected by many factors, such as the resource competition from other processes, memory copy from user space to kernel space, interception of other kernel modules (e.g., iptables), to name a few. Second, the efficiency is low in the presence of cross traffic or in a resource-limited device; Third, although some techniques like `netmap` [10] can achieve high efficiency by directly manipulating NIC's buffer, they do not have good portability because they need to modify NIC drivers. Fourth, some techniques are not flexible. For example, `socket` supports neither customizing the whole packet nor processing packets destined to IP addresses not owned by the device.

To solve the above issues, we propose *kTRxer*, a portable toolkit for helping probing tools achieve better accuracy and efficiency. *kTRxer* mitigates the negative effect from devices by keeping away from noise sources and achieves portability by avoiding modifying device drivers. Our major contributions are threefold. First, we explore the design space and select the most suitable techniques for *kTRxer*. For example, to send probes, *kTRxer* first constructs all necessary data structures for packet transmission in Linux kernel and then invokes the function that will call NIC's transmission function directly. By doing so, *kTRxer* can not only achieve high efficiency but also escape from many noise sources, such as interception of iptables, the long chain of system calls, etc. As another example, to mitigate the delay of memory copy, *kTRxer* creates two virtual devices for mapping the user space to the kernel space. Second, we have implemented *kTRxer* with 5,489 lines of C codes. It consists of a Linux kernel module for sending and receiving probes and a user interface module offering a set of APIs to facilitate building probing tools on top

[§] The corresponding author.

978–1–4799-4852-9/14/$31.00 ©2014 IEEE

of *kTRxer*. All modules can be easily migrated to different devices running Linux as demonstrated in Section III. Third, we conduct extensive experiments to evaluate *kTRxer* on three platforms (i.e., PC, broadband router, and smartphone) in the presence/absence of cross traffic. The experimental results show that *kTRxer* can achieve up to 10 times transmission rate and introduce much less delay noise than existing approaches. Moreover, the overhead incurred by *kTRxer* is low.

The remainder of this paper is organized as follows. Section II describes the design of *kTRxer* and Section III reports the experiment results. After introducing the related work in Section IV, we conclude the paper in Section V.

II. kTRxer

In this section, we first present *kTRxer*'s design goals. After introducing *kTRxer*'s architecture in Section II-B, we will describe how *kTRxer*'s major modules are designed to achieve the goals in the remaining subsections.

A. Design goals

To facilitate the implementation of accurate and efficient probing tools in different devices, we have the following design goals for *kTRxer*.

1) It should mitigate the delay noise from the device as much as possible to achieve high accuracy.
2) It should improve the efficiency as high as a NIC can support.
3) It can be easily migrated to various devices running different Linux distributions.
4) It should be flexible to support common probing requirements(e.g., customize probes and manipulate responses).

B. Architecture

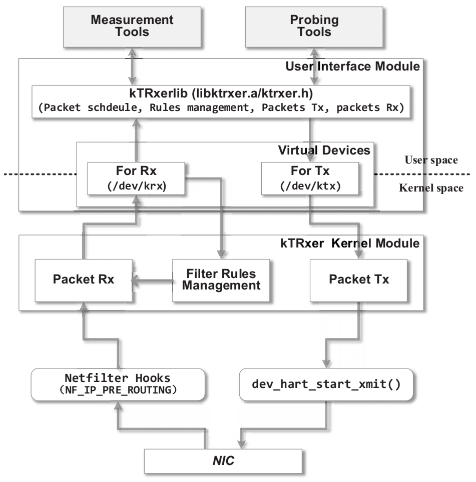


Fig. 1. The architecture of kTRxer.

Fig.1 illustrates the *kTRxer*'s architecture that consists of two major components: a loadable kernel module that comprises of a packet transmission component (i.e., Packet Rx) and a packet capture component (i.e., Packet Tx), and a user interface module that provides rich interfaces for upper-layer probing tools to send and schedule probes, and to receive and filter responses. When using *kTRxer*, a probing tool first delivers probes and the related operation commands to *kTRxer*'s kernel module through its user interface module. Then, *kTRxer* will schedule the packet transmissions and send the received responses to the probing tool.

This architecture takes into account the design goals. First, implementing the packet sending and receiving functions in a kernel module empowers *kTRxer* to quickly dispatch probes through the Packet Tx component and capture response packets through the Packet Rx component. In other words, this kernel module can not only avoid some delay noise but also improve the efficiency. We will elaborate more on how these two components can further mitigate the delay noise in Section II-C and II-D, respectively. To shorten the delay of exchanging data between the user interface module and the kernel module, *kTRxer* creates two virtual devices for mapping the memory in user space and that in kernel space so that the two modules can communicate data without using memory copy. Second, the kernel module can be easily migrated to different devices running various Linux distributions. We have tested *kTRxer* on a PC running Ubuntu, a broadband router running OpenWRT, and a smartphone running Android.

Third, *kTRxer* allows probing tools to fully customize the probes and then invoke the user-space library to deliver the probes to the kernel module through virtual device for transmission. They can also supply filter rules to the Packet Rx component for capturing specific response packets. Section II-E details the user interface module. It is worth noting that the separated packet sending and receiving components allow users to develop complex probing logic on top of *kTRxer*. For example, a measurement/probing tool can first send some probes, and then react after processing the received responses. Note that many packet generators do not provide such functionality because they usually just send fixed packets as quickly as possible and do not care about response packets.

C. Packet Tx

To achieve the first and the second goals, we fist analyze the process flows of packet transmission in Linux, as shown in Fig. 2, and then select the most suitable techniques. If `socket` is used to send probes, a probing tool cannot tell whether the packets are dispatched or discarded. Furthermore, the processing by different functions from the user space to the kernel space may introduce unpredictable but significant delay to packet transmissions [10]. In other words, it is difficult to precisely control the sending time of probes. More precisely, a probe's payload will first be copied from the user space to the kernel space and a socket buffer structure (i.e., `sk_buff`) will be created to record information describing how the packet is represented in the kernel. Then it will be processed by transport layer functions, for example *tcp_sendmsg()*, which will divide the message into several segments according to the maximum segment size (MSS).

Before the packet is sent to the function *dev_queue_xmit()*, its route is first determined and then *ip_fragment()* is called to perform packet fragmentation if necessary. Moreover, the packet will also go through two *netfilter* [11] hook points (`NF_INET_LOACL_OUT` and `NF_INET_POST_ROUTINT`). Note that handlers hooking these two points will process the packet and introduce additional delay. Even the function *dev_queue_xmit()* does not immediately send out the packet. Instead, it inserts the packet into a queue for transmission and then invokes *dev_hart_start_xmit()*. The comments in Linux kernel mention that *dev_queue_xmit()* does not guarantee the packet will be transmitted by NIC, because it may be dropped due to congestion or traffic shaping. The function *dev_hart_start_xmit()* will further invoke NIC's transmission function through the function pointer *ndo_start_xmit()*.

Compared with `socket`-based approaches, using `raw socket` can avoid the delay introduced by the transport layer functions. However, packets sent by `raw socket` may still be postponed due to the IP layer functions and the device queue management if device supports priority queuing. To avoid the delays introduced by these functions, `netmap` inserts the packet into NIC's buffer and then invokes NIC's transmission function to send packets. Although this approach minimizes the delay introduced by Linux's TCP/IP stack, it is not easy to migrate `netmap` to other devices, because it requires modifying NIC's driver.

Our solution consists of two steps. First, *kTRxer* prepares all necessary data structures (e.g., a linked list of *sk_buff*) in the kernel according to the input. By doing so, *kTRxer* can avoid the delay introduced by the processing functions at upper layers, such as functions for preparing headers, iptables' handlers, and `tc`'s methods, and allow the user to fully customize the packets. Second, *kTRxer* invokes the last general function (i.e., *dev_hart_start_xmit()*) supported by all NICs to transmit the packet. This approach ensures that *kTRxer* can be easily migrated to different Linux distributions.

Since sometimes probes need to be sent at scheduled time, *kTRxer* performs the scheduling in kernel module instead of user space for achieving higher accuracy. Two kinds of timers can be used in kernel space: *timer* and *hrtimer*. The resolution of *timer* in Linux kernel is a jiffy, which depends on the value of `HZ` defined in Linux and may not be the same in different distributions, for example, it is 1 millisecond on i386 and 10 milliseconds on most embedded platforms [12]. In contrast, the resolution of *hrtimer* is 1 nanosecond and therefore *kTRxer* uses it for scheduling packet transmission.

D. Packet Rx

With the design goals and the requirements of active probing in mind, we examine the process flow of receiving packets to identify most suitable approach to handle incoming packets. A packet handler could be located at three places [13]: NIC driver, protocol handlers, and *netfilter* [11] hooks. Although adding handling functions in NIC driver may achieve the best efficiency, it requires the modification of NIC driver, thus having portability issues.

Registering a protocol handler also allows us to capture incoming packets. However, since the system will send packets to all matching protocol handlers, not only our protocol handler but also others (e.g., the default IP handler) will receive and process the response packets, thus wasting the resources. *kTRxer* requires to be the first handler to process specified incoming packets (e.g., drop, modify, extract required content) and it only delivers required content to the upper layer. Therefore, we did not use this approach. Similarly, we did not adopt methods designed for passively capturing packets (i.e., *libpcap*, *PF_RING* [14]), because they also do not support such kind of packet manipulations.

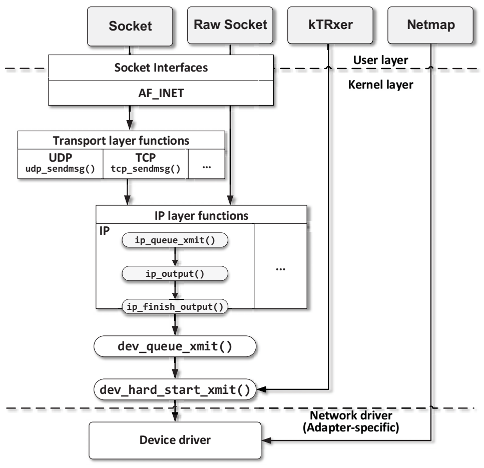


Fig. 2. The process flow of packet transmission.

kTRxer registers *netfilter* hooks to handle incoming packets. While *netfilter* has five hook points, *kTRxer* registers its hook function on the point `NF_IP_PRE_PROUTING` because of two reasons. First, `NF_IP_PRE_ROUTING` is the first hook point for handling arriving packets so that *kTRxer* can process the packet as soon as possible. Second, the hooks on `NF_IP_PRE_ROUTING` can get all the packets reaching the network device including those that are not sent to the local host. It allows *kTRxer* to easily support Internet probing/measurement using fake IPs. Since each hook point may have multiple handlers with different priority, we implement new hook register/unregister functions to ensure that *kTRxer* will be the first to process incoming packets by exploiting the data structure of handlers.

The Packet Rx component will consult the Filter Rules Management module on how to process incoming packets. The latter maintains a rule table, whose entry specifies the operations for packets matching certain rules. Each rule maps a 5-tuple (source IP address, destination IP address, source port, destination port, protocol) to one operation. Currently, *kTRxer* provides four operation types, including capture the packet, drop the packet, capture and then drop the packet, and forward the packet.

Besides manipulating packets, another requirement is to assign accurate timestamp to each packet. To be portable, *kTRxer* relies on Linux kernel to obtain the timestamp. More precisely, *sk_buffer* contains a timestamp parameter that records the timestap when the packet arrives at the network device. In Linux kernel 2.4, the parameter is set by default. Since kernel 2.6, we can call *net_enable_timestamp()* to instruct the system to record the timestamp. As a result, the timestamp obtained by *kTRxer* is the same as that recorded by *libpcap*, which is much more accurate than that obtained in the user space.

E. User interface

The user interface module is composed of two components: two virtual devices for efficiently exchanging data between a probing tool and *kTRxer*'s kernel module, and a library along with the header file (i.e., `libktrxer.a` and `ktrxer.h`) for building tools on top of *kTRxer*.

The virtual devices */dev/ktx* and */dev/krx* employ memory map (mmap) instead of memory copy to exchange data for shortening the delay due to data movement. Employing memory map saves one memory copy between the user space and the kernel space. Device */dev/ktx* is used for delivering customized probes to the kernel module for transmission and */dev/krx* is used for fetching captured packets and handling filter rules. They work independently and will not interfere with each other. When *kTRxer*'s kernel module is loaded, it will create these two virtual devices and register a set of operations for them. *kTRxer* also provides APIs for a probing tool to deliver probes through *libktrxer.a* and *ktrxer.h*

III. Evaluation

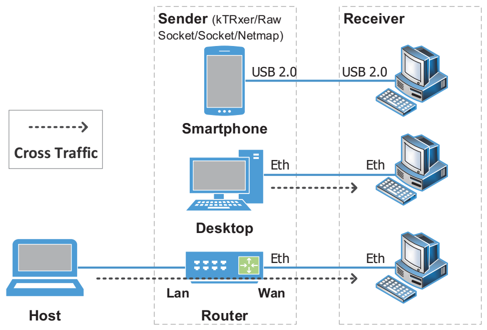


Fig. 3. The experiment setup

We have implemented *kTRxer* in 5489 lines of C codes (counted by CLOC), which include 4837 lines for the kernel module and 652 lines for the user interface module. We conduct extensive experiments on three different platforms (i.e., PC, router, and smartphone) to evaluate *kTRxer*. The PC, running Ubuntu 12.04 LTS, is equipped with Intel 3.4 GHz i7-4770 CPU, 16G memory, and 1Gbps NIC. The router, running OpenWRT, is a Buffalo WZR-HP-AG3000H model with an Atheros AR7161 680MHz CPU, 128M RAM, 32M Flash, and AR8319 ethernet adapters. The Sumsang Vibrant smartphone, running Android 4.2, has a 1GHz ARM Cortex-A8 CPU and 512M memory.

The experiments were designed to answer the following questions:

1) Can *kTRxer* transmit probes as quickly as possible in the presence/absence of cross traffic? How about other techniques?
2) Can *kTRxer* accurately send probes at scheduled time? How about other techniques?

Besides answering these questions, we also confirm that *k-TRxer* can obtain the same timestamp for incoming packets as libpcap does.

We implement four probing tools, which are based on *netmap*, *kTRxer*, `socket`, and `raw socket`, respectively. Note that we only run the *netmap*-based probing tool on the PC because *netmap* does not provide modified drivers for the smartphone and the router. Other probing tools are tested on these platforms. Fig 3 shows the experiment setup. In all scenarios, we run the probing tools on different senders, which send probes to the same receiver, and capture the packets on the receiver side.

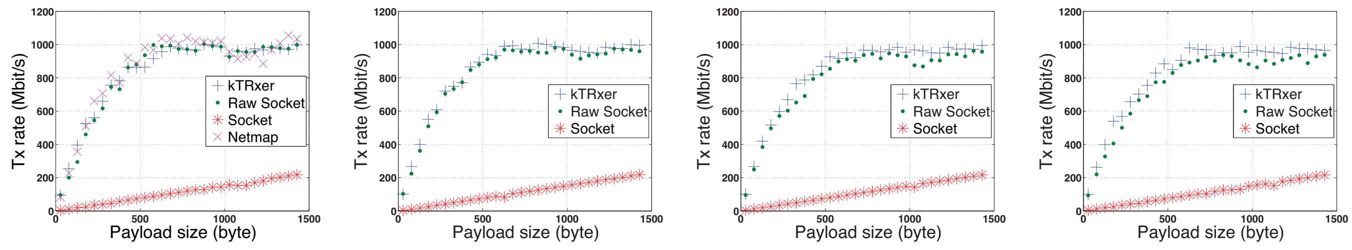
We connect the smartphone to the receiver using a USB 2.0 cable and leverage the USB reverse tethering technique [15] to build a network connection between them. When running tests on PC, we directly connect the PC to the receiver using a cable and the cross traffic generated by the PC goes through the same path as probes (i.e., both of them are sent to the receiver). To test *kTRxer* on the router, we connect the router's WAN interface to the receiver, and one of its LAN interfaces to a host. To examine the worst case, we let the host send cross traffic to the receiver. Thus, both probes and the cross traffic will go through the same WAN interface. We use *D-ITG* [16] to generate the cross traffic.

A. Transmission rate

We let probing tools send 100 UDP packets as a whole. The payload size ranges from 28 bytes to 1,428 bytes, with an increment of 50 bytes. In the receiver side, we capture these packets and calculate the elapsed time between the first packet and the last packet, then compute the transmission rate. We run such test 10 times on each platform under different cross traffic conditions, and draw the median value in Fig 4-6.

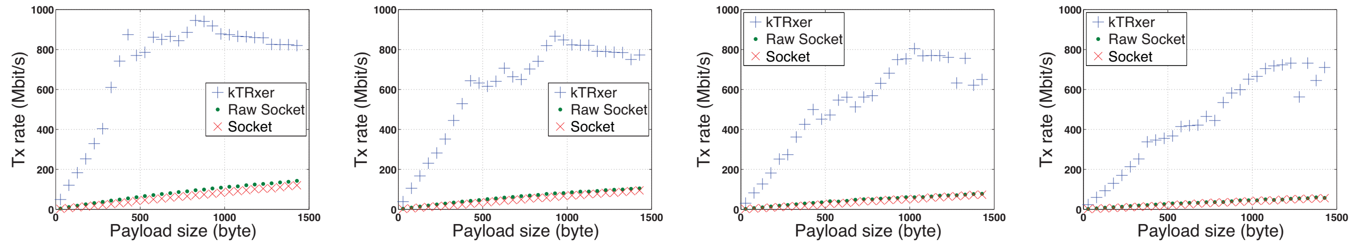
PC: We found that the NIC needs to be re-initialized after running *netmap*'s pkt-gen program, and therefore D-ITG cannot be run simultaneously. Hence, *netmap* only has the result in the absence of cross traffic. Without cross traffic, *netmap* can achieve the highest transmission rate. It is expectable because *netmap* can avoid many delay noises by inserting probes into NIC's buffer directly. *xTRxer* is a little better than `raw socket`. However, the differences among kTRxer, raw socket and netmap are *small*. When the payload size is over 500 bytes, all except the `socket`-based method can reach the maximum transmission rate (i.e., 1,000 Mbs) of the 1G NIC. The implication is that using portable approaches like *kTRxer* or `raw socket` with large packet size can achieve the maximal transmission rate in device with sufficient resource. `socket`'s performance is obviously the worst, only about one fifths of other three tools.

We also run the probing tools in the presence of cross traffic, whose sending rate is {12,000, 24,000, 36,000} packets per second (pps) and the packet size uniformly distributes in the [100, 1400] bytes. As shown in Fig 4(b) to Fig 4(d), *kTRxer* is better than `raw socket` and `socket` in these cases and the difference between *kTRxer* and `raw socket` grows when the cross traffic become heavier. In particular, *kTRxer*'s



(a) $r = 0$ pps (b) $r = 12,000$ pps (c) $r = 24,000$ pps (d) $r = 36,000$ pps

Fig. 4. Transmission rate of different probing tools on the PC under different cross-traffic rate (r).



(a) $r = 0$ pps (b) $r = 6,000$ pps (c) $r = 12,000$ pps (d) $r = 18,000$ pps

Fig. 5. Transmission rate of different probing tools on the router under different cross-traffic rate (r).

transmission rate can still reach the limit of 1G NIC even with heavy cross traffic while `raw socket`'s transmission rate decreases when there is heavy cross traffic. Moreover, `socket` still has the worse performance.

Router: Fig 5 shows the transmission rates of *kTRxer*, `raw socket`, and `socket` with/without cross traffic. Different from the results on PC, *kTRxer* is much better than `raw socket` and `socket` in all cases. Although the transmission rates of all methods decrease with the increase of cross-traffic rate, the advantage of *kTRxer* is still obvious. By contrast, the difference between the performance of `raw socket` and that of `socket` declines with the increase of cross-traffic rate. It may be due to the effect from a low-end CPU and the process of *iptables* in the router, both of which will introduce additional delay to all packets. *kTRxer* suffers less from them because it prepares all necessary data structures for transmission in kernel and then immediately calls the NIC's transmission function to dispatch the probes.

Smartphone: Since D-ITG is not available for Android, we cannot evaluate *kTRxer*'s performance under cross traffic on smartphone and leave it to future work. Since Android application cannot invoke raw socket directly, for a fair comparison, we run all probing tools on Android's Linux platform. We expect that using socket within an Android application to send probes will lead to worse performance because of the additional noises introduced by the Dalvik virtual machine. Fig 6 shows the transmission rates of different methods obtained on smartphone. All methods can result in higher transmission rate by sending larger probes. *kTRxer*'s performance is still much better than that of `raw socket` and `socket`, reaching the highest Tx rate nearly 120 Mbit/s. `raw socket` is still better than `socket` and its highest Tx rate is about 50 Mbit/s.

B. Packet scheduling

To evaluate each method's capability of sending packets at scheduled time, we instruct probing tools to send 10 packets and let the interval between consecutive packets be 10ms. Then, we calculate the difference between measured interval between packets and predefined interval. The smaller the difference is, the better the method is.

Table I show the median and the interquartile range (IQR) [17] of the absolute difference between the measured interval and scheduled interval in the absence of cross traffic. It is obvious that in differnet devices *kTRxer* leads to much smaller median and IQR of the difference than `socket` and `raw socket` do, meaning that *kTRxer* can control the transmission of probes actually and stably. Particularly, the median difference caused by *kTRxer* is usually one tenth of that caused by `socket` and `raw socket`. Meanwhile, the results do not have many changes with different data sizes.

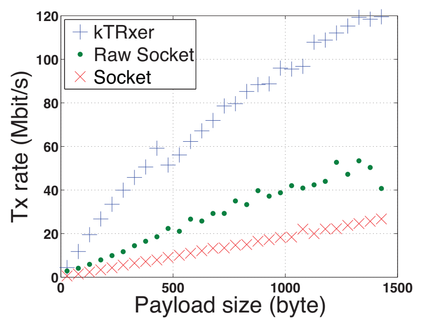


Fig. 6. Transmission rate of different probing tools on smartphone without cross traffic.

IV. Related work

A plethora of active probing tools have been proposed [18]. After manually analyzing 21 open-source tools in [18], we

TABLE I
MEDIAN/IQR (IN MICROSECONDS) OF |(SCHEDULED INTERVAL) - (MEASURED INTERVAL)|, WHERE SCHEDULED INTERVAL= 10MS

Platform	Tool	Data Size (byte)						
		100	300	500	700	900	1100	1300
Desktop	kTRxer	21/27	10/25	17/42	15/30	8/33	5/31	11/36
	Raw Socket	138/34	135/41	131/41	141/7	132/78	135/78	136/68
	Socket	147/56	126/41	137/42	141/62	140/51	134/50	129/15
Router	kTRxer	8/15	8/10	9/9	5/7	8/9	10/15	10/18
	Raw Socket	72/39	70/34	81/23	82/37	82/61	81/51	89/54
	Socket	162/46	147/52	151/49	153/50	147/51	137/37	151/44
Smartphone	kTRxer	37/124	61/119	64/139	71/151	76/114	59/124	78/113
	Raw Socket	125/102	125/202	127/149	126/82	129/141	123/110	129/180
	Socket	609/263	624/335	379/377	365/199	326/142	373/259	195/498

found that 10 tools just use `socket` and the remaining tools use `raw socket` if necessary. However, few tool examines how to make full use of the NIC and how to avoid the negative effect from the system.

Recently, some packet generators are optimized to make full use of the NIC. For example, *Brute* [19] uses `raw socket` and *pktgen* [20] invokes *ndo_start_xmit()* to generate high-speed traffic. There are three major differences between *pktgen* and *kTRxer*. First, since *pktgen* aims at dispatching packets as quickly as possible for testing network equipments, it randomly generates meaningless packets and sends them out. In contrast, since *kTRxer* targets on helping probing tools achieve better accuracy and efficiency in resource-limited devices, it provides simple interfaces to facilitate the communication between itself and probing tools, and send the packets out by invoking *dev_hard_start_xmit()*. Second, since *pktgen* randomly generates packets, it does not care about whether the packet transmission succeeds or not. In contrast, *kTRxer* will retransmit a probe if the transmission fails and will notify the probing tools if all retransmissions fail. Third, *pktgen* does not handle response packets while *kTRxer* will process them. *KUTE* [21] uses similar techniques like *pktgen*.

Mobile devices and embedded systems introduce new requirements and challenges to active probing. *SamKnows* [5] and *BISmark* [6] run measurement tools in routers with customized *OpenWRT*. However, the results may be biased if there is heavy cross traffic. Similarly, *SmartProbe*, a Java tool for estimating network capacity in smartphone, uses socket to send probes [4]. Its results may be affected by the system [4]. We demonstrated the previous version of *kTRxer* in [22] and have significantly improved it since then.

V. CONCLUSION

Performing active probing in resource-limited devices introduces new requirements and challenges due to the non-negligible effect of the system on the commonly used packet sending/receiving techniques. To mitigate such negative effect, we propose *kTRxer*, a portable toolkit for helping probing tools achieve better accuracy and efficiency in resource-limited devices. We have implemented *kTRxer* and the extensive experimental results obtained from three different devices shows that *kTRxer* can achieve much higher transmission rate and introduce much less delay noise than existing approaches.

VI. ACKNOWLEDGMENT

This work is supported in part by the CCF-Tencent Open Research Fund, the Hong Kong GRF (No. PolyU 5389/13E), the National Natural Science Foundation of China (No. 61202396), Shenzhen City Special Fund for Strategic Emerging Industries (No. JCYJ20120830153030584), and the Open Fund of Key Lab of Digital Signal and Image Processing of Guangdong Province.

REFERENCES

[1] X. Luo, E. Chan, and R. Chang, “Design and implementation of TCP data probes for reliable and metric-rich network path monitoring,” in *Proc. USENIX Annual Tech. Conf.*, 2009.
[2] L. Quan, J. Heidemann, and Y. Pradkin, “Trinocular: Understanding internet reliability through adaptive probing,” in *Proc. ACM SIGCOMM*, 2013.
[3] Z. Durumeric, E. Wustrow, and J. Halderman, “Zmap: Fast internet-wide scanning and its security applications,” in *Proc. USENIX SECURITY*, 2013.
[4] F. Disperati, D. Grassini, E. Gregori, A. Improta, L. Lenzini, D. Pellegrino, and N. Redini, “Smartprobe: a bottleneck capacity estimation tool for smartphones,” in *Proc. IEEE GreenCom*, 2013.
[5] “Samknows,” http://www.samknows.com.
[6] “Bismark,” http://www.projectbismark.net.
[7] W. Stevens, B. Fenner, and A. Rudoff, *Unix Network Programming: The Sockets Networking API*, 2003, vol. 1.
[8] C. Dovrolis, P. Ramanathan, and D. Moore, “Packet dispersion techniques and a capacity-estimation methodology,” *IEEE/ACM Trans. Netw.*, vol. 12, no. 6, 2004.
[9] M. Jain and C. Dovrolis, “End-to-end available bandwidth: Measurement methodology, dynamics, and relation with TCP throughput,” *IEEE/ACM Trans. Netw.*, vol. 11, no. 4, pp. 537–549, 2003.
[10] L. Rizzo, “netmap: a novel framework for fast packet i/o,” in *Proc. USENIX ATC*, 2012.
[11] “Netfilter,” http://www.netfilter.org.
[12] “High resolution timers,” http://elinux.org/High_Resolution_Timers.
[13] S. Seth and M. Venkatesulu, *TCP/IP Architecture, Design and Implementation in Linux*. Wiley, 2008.
[14] “Pf_ring,” http://www.ntop.org/products/pf_ring/.
[15] “Usb reverse tethering,” http://forum.xda-developers.com/showthread.php?t=2287494, 2013.
[16] A. Botta, A. Dainotti, and A. Pescape, “A tool for the generation of realistic network workload for emerging networking scenarios,” *Computer Networks*, vol. 56, no. 15, 2012.
[17] P. Huber and E. Ronchetti, *Robust Statistics*, 2nd ed. Wiley, 2009.
[18] “Performance measurement tools taxonomy,” http://www.caida.org/tools/taxonomy/performance.xml.
[19] N. Secchi, R. Bonelli, S. Giordano, and G. Procissi, “Brute: A high performance and extensible traffic generator,” in *Proc. SPECTS*, 2005.
[20] D. Turull, “pktgen,” http://people.kth.se/ danieltt/pktgen/.
[21] S. Zander, D. Kennedy, and G. Armitage, “Kute - a high performance kernel-based udp traffic engine,” Swinburne University of Technolog, Tech. Rep., 2005.
[22] L. Xue, R. Mok, and R. Chang, “Omware: An open measurement ware for stable residential broadband measurement (Demo),” in *Proc. ACM SIGCOMM*, 2013.