Application-Specific Packet Capturing using Kernel Probes

Byungjoon Lee*, Seong Moon* and Youngseok Lee†

*Electronics and Telecommunications Research Institute
Daejeon, 305-700, Republic of Korea
{bjlee, seong}@etri.re.kr

†Chungnam National University
Daejeon, 305-764, Republic of Korea
lee@cnu.ac.kr

Abstract—When we reverse-engineer unknown protocols or analyze the Internet traffic, it is critical to capture complete traffic traces generated by a target application. Besides, to prove the accuracy of Internet traffic classification algorithms of the traffic monitoring system usually located in the middle of the network, it is highly required to retain traffic traces associated with the related application. Therefore, in this paper, we present an application-specific packet capturing method at end hosts, which is based on the dynamic kernel probing technique. From the experiments it is shown that the proposed method is useful for creating per-application complete traffic traces without performance degradation.

I. INTRODUCTION

To reverse-engineer the unknown protocol of an application or to analyze the application traffic, it is required to have a complete traffic trace generated by the application. Besides, to assess the accuracy of a traffic classification algorithm of the traffic monitoring system usually located in the middle of network, it is necessary to have the full traffic traces where the portion of each application is known. Otherwise, it becomes difficult to assess the accuracy of the classification algorithm. Moreover, a classification method itself is affected by the quality of a traffic trace especially when a classification method is implemented by machine-learning algorithm because the noise in the training-set might put a negative impact to the training result.

Traditionally, an application-specific traffic trace has been created by capturing packets generated by running the applications within a closed network. Though the method is simple, it is getting harder to be applied because most of the popular contemporary network applications are highly distributed inherently (e.g. P2P applications), which require more complex testbed network to be built. Moreover, the method has another disadvantage that it is impossible to remove the packets related to the maintenance of the network from the traces without the aid of offline analysis. The problem of the offline analysis is that it takes too much time, and there is no systematic way to prove the correctness.

Classification methods which are commonly used in the middle of the networks are also not suitable for the purpose because of the false-positives and false-negatives: As widely known, port-based simple traffic classification method cannot capture all the packets of an application because lots of recent Internet applications use dynamic port numbers as well as multiple connections. Other classification methods [1] [2] [3] [4] which use communication patterns, statistical patterns, or payload signatures also share the same problem in that they cannot perfectly amortize the false-positives and false-negatives.

Therefore, in this paper, we propose an application-specific packet trace acquiring method at the end host that captures all the Internet traffic generated by the application, given its name. The proposed method is based on a dynamic kernel probing technique called Kprobes [5] supported by the Linux kernel 2.6.0 or higher. Kernel probes like Kprobes enable the administrator of the system to monitor the activities of the kernel functions in detail by intercepting calls to kernel functions and investigating the passed parameters and return values. As the probes can be inserted to the running system dynamically, the proposed method could be easily deployed to or removed from a system without any rebooting or re-compilation process.

Kprobes provides three kinds of probes: Kprobe, Jprobe, and Kretprobe. Kprobe is a plain kernel probe. Jprobe is a specialized Kprobe in inspecting arguments passed to the kernel functions. Kretprobe is for monitoring the values returned by kernel functions. The suggested method uses Jprobe and Kretprobe for implementation.

The rest of this paper is organized as follows. Section II outlines the architecture of the proposed method and explains the implementation details of the key components. Section III provides the performance evaluation result of the proposed architecture. Section IV concludes this paper.

II. PROPOSED APPLICATION-SPECIFIC CAPTURING METHOD

A. Architecture

The architecture of the proposed method, called ACAP (Application-specific packet CAPture), is shown in Fig. 1. As depicted, ACAP is composed of one kernel module (ACAP_Probe) and two user processes (ACAP_Collector and ACAP_Capturer).
ACAP_Probe is a kernel module which inserts probes into the kernel functions. Its role is to extract 5-tuple (protocol number, source IP address, destination IP address, source port number and destination port number) information of packets sent or received by a specific application and export the information to ACAP_Collector with the name of the application. The name of the application to be monitored is given when ACAP_Probe is inserted to the kernel.

ACAP_Collector is a UDP server which receives 6-tuple information (5-tuple plus the name of the application) sent by ACAP_Probe and maintains the information in a hash table.

B. ACAP_Probe

As shown in Fig. 2, Linux TCP/IP networking kernel is divided into three layers: IP layer, INET socket layer and BSD socket layer. Though every IP packet is processed by a set of IP layer functions which are much simpler than the upper layer functions, we chose to hook INET socket layer functions instead because the application name (or the socket number) associated with a packet is determined when the packet arrives at the upper half of the INET socket layer [7]. For example, not until the tcp_recvmsg() reads the packet from the underlying queue, the socket to which the packet should be delivered is not decided. Once a socket is determined, the name of the application associated with the packet is easily retrieved by reading current->comm array.

Therefore, ACAP_Probe inserts two Kprobes probes for each of following two INET layer functions: inet_sendmsg() and sock_common_recvmsg(). Per each of the functions, one Jprobe probe and one Kretprobe probe is inserted. Whenever the function is called, the Jprobe probe is executed before the real function call to save the pointers to the socket and message structures passed to the kernel function as arguments. The Kretprobe probe is executed after the hooked kernel function is executed. It checks first if the kernel function is executed without any errors. If there is no error, the probe extracts the 6-tuple information of the packet which incurred the call. The information is extracted from the socket and message structures which were previously saved by Jprobe probe.

The extracted information is maintained in the hash table of ACAP_Probe. The table is introduced to reduce traffic between ACAP_Probe and ACAP_Collector. Only when a new entry is inserted to the table, the 6-tuple information of the entry is assembled into a UDP packet and sent to ACAP_Collector using netif_rx(). As netif_rx() injects a packet into the upper layer of the network drivers directly [7], the communication between ACAP_Probe and ACAP_Collector is not seen by any packet capturing processes including ACAP_Capturer or any other PCAP programs such as tcpdump.

ACAP_Probe is dynamically attached to and removed from the Linux kernel using insmod and rmmod commands. For example, we execute following command to capture only skype traffic, where 'acap_probe.ko' is the name of compiled ACAP_Probe kernel module and 'appname' is the application name to capture.

\$insmod acap_probe.ko appname='skype'

C. ACAP_Collector and ACAP_Capturer

Whenever receives 6-tuple information from ACAP_Probe, ACAP_Collector saves it into the internal hash table. ACAP_Capturer queries the table to check whether a captured packet is associated with a specific internet application.

As depicted in Fig 3, ACAP_Capturer pushes every captured packet into an internal queue first. The queued packet is taken from the queue after a specific amount of time (QUEUE_TIME) and compared to the ACAP_Collector hash table. If a matched entry is found, the packet is saved into a PCAP file.

The delay QUEUE_TIME is introduced because it is possible to have no matching entry in the ACAP_Collector hash table when the packet is captured because the packet is not processed by the networking kernel yet. By delaying packet processing for QUEUE_TIME, we can make the networking kernel finish packet processing before the ACAP_Collector hash table is queried. Therefore, we can safely conclude
that a packet is not from the application which is being captured if corresponding 6-tuple information is not found in ACAP_Collector hash table.

The matched packets are saved into a PCAP file, which could be used as an input to popular traffic analysis toolkits such as tcpdump and Ethereal [6] for facilitating further analysis of the captured trace.

III. PERFORMANCE EVALUATION

A. Accuracy

We tested the correctness of ACAP implementation in two ways. Firstly, we tested ACAP using a program which sends/receives UDP/TCP packets of the random size from/to the external server with 1000 randomly selected port numbers and records the total volume of the UDP/TCP payloads.

To prevent packets from being dropped at the capturing process, we controlled the throughput of the program below 10Mbps. Though there is no limitation in ACAP implementation on interface speed, we introduced the regulation to facilitate the verification process because the random drop/retransmission of packets shall make the total volume of UDP/TCP payloads captured by ACAP greater than that of actually transmitted payloads.

After capturing the traffic using ACAP, we recalculated the total volume of the UDP/TCP payloads using the captured traffic file and compared with the pre-recorded figures. From the experiment it is observed that ACAP guarantees 100% of accuracy.

Secondly, we tested ACAP against skype. Skype [8] is IP-telephony software based on well-known P2P protocol KaZaA. We chose skype because it uses both UDP and TCP protocol and creates a lot of flows for its operation.

For the test, we created traffic capture files using both ACAP and tcpdump while skype was running and compared the files in following three steps: Firstly, we did an off-line analysis on the tcpdump file to extract a set of flows which are suspected as non-skype flows. Secondly, we obtained another set of flows which are in tcpdump files but not in ACAP files. Finally, we compared both sets of flows. The experiment revealed that both sets of flows are exactly the same. We also checked if there are flows which only belong to ACAP files. We verified that there are no such flows.

B. Impact on Throughput

As it is expected that kernel probing delays normal packet processing procedures, it is important to correctly assess the overhead.

To estimate the impact on the throughput, we executed iperf [9] 100 times to measure the performance of transferring 697Mbytes file from Linux node (iperf client) to Windows XP node (iperf server) which are connected by 1Gbps NICs and a 1Gbps switching hub. The Linux node is a LG LM70 notebook, equipped with 2Gbytes memory and 80Gbytes hard disk, and Intel Pentium M 1.86GHz processor. The Windows XP node has 1Gbytes memory, Intel Pentium D 3.0GHz CPU and 250Gbytes hard disk. Experiment for ACAP is done at the Linux node. The test result is given in Fig 4.

In Fig 4, (a) is a performance measurement result without ACAP_Probe, (b) is a result with ACAP_Probe and without ACAP_Collector and ACAP_Capturer, and (c) is a result with ACAP_Probe, ACAP_Collector and ACAP_Capturer. (d) is a performance measurement result of tcpdump.

As shown in Fig. 4, no notable differences among experiment cases are observed. Therefore, we conclude that ACAP does not put any noticeable impact on the network performance if enough memory is provided. It means that we are able to deploy ACAP to end nodes without any degradation of network performance which is perceived by users of the nodes because ACAP_Probe does not incur any significant performance degradation of the Linux networking kernel.
### TABLE I

<table>
<thead>
<tr>
<th>QUEUE_TIME</th>
<th>CPU</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Utilization</td>
<td>1sec</td>
<td>5-12%</td>
</tr>
<tr>
<td></td>
<td>2sec</td>
<td>5-12%</td>
</tr>
<tr>
<td></td>
<td>4sec</td>
<td>5-12%</td>
</tr>
<tr>
<td></td>
<td>10sec</td>
<td>5-12%</td>
</tr>
<tr>
<td></td>
<td>160sec</td>
<td>5-12%</td>
</tr>
</tbody>
</table>

**C. System Utilization**

As we expected that the memory and CPU utilization would vary with different settings of QUEUE_TIME, we have tested ACAP_Capturer with QUEUE_TIME set to 1, 2, 4, 10 and 160 seconds. The result is explained in Table 1. The memory utilization ratio was recorded as the maximum value during the experiments.

As shown in Table 1, the CPU utilization is not affected by the QUEUE_TIME. Yet, the memory which is consumed by ACAP_Capturer increases linearly by the increase of the QUEUE_TIME.

When QUEUE_TIME is set to 160 seconds, the memory utilization of ACAP_Capturer approaches 60%. After 50% of memory is used, the average throughput starts to degrade gradually as shown in Fig 5. However, we believe that setting QUEUE_TIME to 160 seconds is obviously unrealistic because there is no reason to delay processing of packets about 3 minutes. Therefore, we concluded that the system utilization of ACAP is kept low where QUEUE_TIME value is reasonably chosen.

**IV. Conclusion**

In this paper, we have proposed an application-specific packet capturing method based on the kernel probing technique that could collect the complete packet trace of the target application at the end host. The kernel probes enable administrators to hook specific kernel functions. With the probes, administrators can inspect the arguments passed to them, and values returned from them. Therefore, it is possible to extract 5-tuple information of the packets and the name of the application which generates them while they pass through the networking layer of the kernel. The extracted information has been used to capture the traffic generated from a specific internet application only.

The software architecture described in this paper has been implemented on the Linux kernel probing technology called Kprobes. Through experiments, we have shown that the application-specific packet capturing function can be achieved without interrupting the kernel operation and without degrading the networking performance.

The proposed method can be easily extended to capture all the traffic generated from a specific host. In that case, traffic statistics report per each application which is running on the host can be easily obtained. We believe that the extension shall be very useful to researchers who invent and test traffic classification algorithms.

**ACKNOWLEDGMENT**

This work was partly supported by the IT R&D program of MKEIITA (2008-F-016-02, CASFI).

**REFERENCES**