KPAT: A Kernel and Protocol Analysis Tool for Embedded Networking Devices

Ming-Hung Wang, Chia-Ming Yu, Chia-Liang Lin, Chien-Chao Tseng
Dept. Computer Science
National Chiao Tung University
Hsinchu, Taiwan, R.O.C.

Li-Hsing Yen
Dept. Computer Science and Information Engineering
National University of Kaohsiung
Kaohsiung, Taiwan, R.O.C.

***Abstract*—Sniffer tools capture protocol data. Kernel-profiling tools track function calls and events occurring in the kernel. These two types of tools help us observe external and internal behaviors of networking protocols, respectively. We need both types of data for a comprehensive view of protocol behavior. However, few tools perform these two tasks in an integrated way. We developed Kernel and Protocol Analysis Tool (KPAT). KPAT injects software probes into Linux kernel to track interested function calls and event occurrences in the kernel. Probe injection is done systematically and does not require recompiling the kernel. A module in KPAT finds the association between the tracked functions and protocol data captured by an independent sniffer. The result as an integrated log allows users to identify two-way relationship between protocol data and the execution sequence of network functions in the kernel. We successfully used KAPT to identity accurate latency of each handover phase in IEEE 802.11 wireless networks. Experimental results show that KPAT causes light overhead to the patched kernel.**

Keywords—Embedded Devices; Kernel Functions; Kernel Events; Networking Protocols; Packet Sniffers

I. INTRODUCTION

For decades, the industry and academic both have spent a great amount of efforts on developing efficient communication sub-system for embedded computation and communication devices. The communication sub-system should provide network access service with satisfactory performance to applications running on an embedded device. Therefore, it is a need for protocol design in embedded systems to reduce message delays, response times and packet losses on network. To meet this need, designers need tools for benchmarking and for the analysis of internal kernel behaviors and external protocol behaviors. A well-rounded tool should facilitate analyzing, debugging, and performance tuning processes which are important to compact timing of system prototyping and performance optimization.

Sniffer tools (such as Wireshark, tcpdump and Kismet [1-3]) are commonly used to intercept and log protocol data (e.g., frames, packets, segments), enabling an easy monitoring of contents and exchange sequences. However, sniffers do not provide a complete view of protocol behaviors because information about protocol-related activities in the kernel (such as function calls and events triggered by a particular protocol data) is absent. *Kernel-profiling tools* [8-12], on the other hand, inject software probes somewhere in the kernel to collect operation and performance information from the system. These tools, often designed for general purpose, allow us to track a certain type of system events and call sequences. However, kernel-profiling tools alone neither provide a complete view of protocol behaviors because contents and timing information of protocol data are often unavailable.

This work was motivated by the need to investigate empirically handover latency of a mobile node (MN) roaming between two access points in IEEE 802.11 wireless local area networks. For a detailed analysis, we need to know the exact timing of both kernel events and packet transmissions/receptions. For example, the probing phase of an inter-ESS layer-2 handover defined in [28] starts from the occurrence of a link-down event (an event signified by the kernel that can be logged by kernel-profiling tools) and ends with the transmission of the first authentication frame (a protocol data that can be captured by sniffer tools). It turns out that calculating the latency of the probing phase requires both external and internal behaviors of a protocol.

To provide a complete view incorporating external and internal behaviors of a protocol, this paper proposes Kernel and Protocol Analysis Tool (KPAT). KPAT injects software probes into Linux kernel to track interested function calls and event occurrences in the kernel. Probe injection is not manually done and does not require recompiling the kernel. The injected probes create a function log and an event log. An analyzer module in KPAT processes these logs and finds the association between these functions and protocol data captured by an independent sniffer. The result as an integrated log allows users to observe detailed protocol behaviors such as the execution sequence and latency of network functions, processing time and the call graph for processing a protocol data, and detailed handover latency. Experimental results indicate that KPAT causes light computation overhead to the kernel. We successfully used KPAT in the analysis of handover latency.

The remainder of this paper is organized as follows. We first review existing sniffer tools and kernel-profiling tools. Section III describes the design of KPAT in details and Section

IV presents numerical results of our experiments. The last section concludes this paper.

II. Related works

Sniffer tools such as Wireshark, tcpdump and Kismet [1-3] intercept and log protocol data, allowing users to examine the contents and the timing of protocol data. Wireshark [1], one of the most acknowledged sniffer tools, is a free and open source (license) utility for network traffic capturing and analyzing. Wireshark allows users to examine captured protocol data in every detail. This paper adopts Wireshark to work with KPAT.

Kernel-profiling tools [8-12] inject software probes into system kernel. However, different approaches differ in how the probes are injected. *Source instrumentation* approach places probes by modifying the source code of the kernel while *binary instrumentation* approach targets the object code of the kernel. *Statistical sampling* approach uses a stand-alone monitor process to probe the execution of CPU instructions periodically. This approach records the execution counts and period of each instruction and each function.

Table 1 lists several kernel-level profiling tools. LTT, KTAU and LKST all gather generic information about the kernel (average load, context switch, send signal, interrupt, exception, memory allocation and so on.) Kernprof and KFT focus on function executions in the kernel and provide a function call graph of the kernel. Kprobe, KernInst and DTrace adopt dynamic binary instrumentation mechanism, which allows a dynamic selection of locations in an object code to place probes. However, finding the address of target instruction demands the assistance of a compiler and a disassembler. To ease this task, SystemTap provides a user interface to find the addresses of target instructions and help configure Kprobe with the target function name. KLASY allows users to find all the addresses of instructions that ever access a specific data structure (by specifying the data structure name of the target), and helps configure the setting of KernInst accordingly. Oprofile uses timer interrupt to periodically probe the execution of CPU instructions (with the help of built-in performance counters in CPU). These tools provide rich profiling functions. However, none of these tools are designed to identify all kernel behaviors that associate with a particular protocol data. Furthermore, these tools do not integrate logs of kernel behaviors (function log and event log) with logs of protocol behaviors for a complete view of networking behavior.

TABLE I. Kernel-level profiling tools classified by probe-injection technique

Source Instrument	Linux Trace Toolkit (LTT) [6] Kernprof [5] Kernel Function Trace (KFT) [4] Kernel Tuning and Analysis Utilities (KTAU) [13][14] Linux Kernel State Tracer (LKST) [15]
Binary Instrument	Kprobe [16] KernInst [18] DTrace (Solaris-based) [20] SystemTap [17] Kernel Level Aspect-oriented System (KLASY) [19]
Statistical Sampling	Oprofile [21][22]

III. Kernel and Protocol Analysis Tool (KPAT) Development

KPAT works with Wireshark and assumes Linux as the target platform. Fig. 1 illustrates the system architecture of KPAT. KPAT includes three major parts: kernel tracer, network sniffer, and integrated analyzer. Kernel tracer running on the device under test (DUT) traces network-related function calls and events to generate a *kernel behavior log* (KBL). Network sniffer running on another device captures all frames exchanged on the physical network to generate a *protocol behavior log* (PBL). Integrated analyzer integrates the KBL from kernel tracer with the PBL from sniffer, and provides users with a graphic user interface (GUI) for exploration and further analysis. Kernel tracer and integrated analyzer shall be discussed in details in the following two subsections.

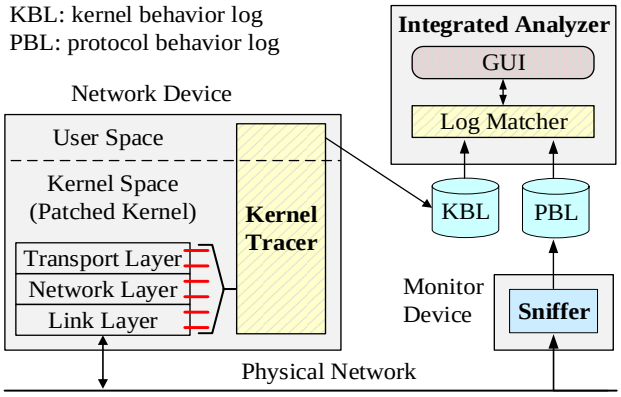

Fig. 1. System architecture of KPAT.

KBL consists of a function log and an event log. The function log keeps track of all function calls that have ever processed any protocol data. Recorded information includes the sequence of function calls, latency between function calls and function execution time. The event log records the occurrences of all network-related events such as link-down and link-up events of the network interface.

A. Kernel Tracer

Kernel tracer tracks specific function calls and particular signals raised by the kernel. Possible implementations of kernel tracer are through the use of system calls or by probe injections. System call is a fundamental interface between applications and kernel, and a probe is a piece of codes injected into kernel to monitor the execution of kernel system. Because system calls incur considerable overheads due to frequent mode switches between kernel mode and user mode, we chose probe injection.

Two possible approaches to probe injection are source instrumentation and binary instrumentation. Binary instrumentation is tightly bound to CPU architecture. In contrast, source instrumentation targets source code and thus is hardware-independent. For this reason, kernel tracer uses source instrumentation for probe injection.

Fig. 2 shows the architecture of kernel tracer. After probes have been injected into selected network functions, calling these functions will trigger probes, which in turn cause

instrument module to log the call of the function. *Event module* detects occurrences of particular network events in the kernel. Instrument module and event module are both implemented as Linux kernel modules [23-25]. Linux kernel modules can be compiled in the user space. By mounting these modules, KPAT effectively extends Linux kernel.

We can also inject probes into network applications to capture behaviors of application layer protocols. For instance, if users are interested in handover analysis, they also inject probes into DHCP client to track the actions of IP address acquisition.

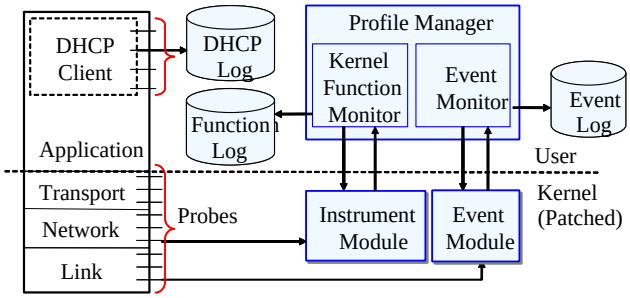

Fig. 2. Architecture of Kernel Tracer.

Probe injection can be done manually or systematically. Manual injection of probes into kernel is infeasible due to numerous target locations. Another reason for not using manual injection is that the target locations for probes vary with kernel versions. Our work thus proposes *selective auto-instrument technique*, which systematically injects probes into selected kernel functions. Linux kernel uses structure *sk_buff* (for socket buffer) [26] to keep contents and related information of each protocol data. Since all network-related functions in the kernel access this structure, the proposed auto-instrument technique finds all target functions with keyword *sk_buff*. After a compete search on kernel source code, this technique generates a patched source code. This patch allows us to inject probes without recompiling the whole kernel.

Profile manager consists of *kernel function monitor* and *event monitor*. Kernel function monitor periodically moves the function log generated by instrument module to a permanent storage whereas event monitor does the same for the event log. Profile manager also provides configurable kernel instrument (CKI) that allows users to select functions to track before each run. As Fig. 3 shows, users select functions to track by enabling or disabling the links between injected probes and instrument module.

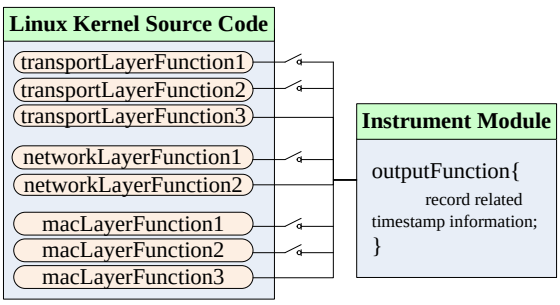

Fig. 3. Architecture of Configurable Kernel Instrument.

Kernel functions may generate considerable probing data to the function log. If instrument module uses *printk*() to output data, the incurred overhead will severely drag down kernel's performance. Therefore, the proposed instrument module writes all probing data into the function log that resides in kernel space. Profile manager then uses memory MAP technique [31] to perform a background-transfer logging that copy contents from the function log to disk storage space (Fig. 4). As this data transfer is performed only in system's idle time, kernel's performance is not significantly affected.

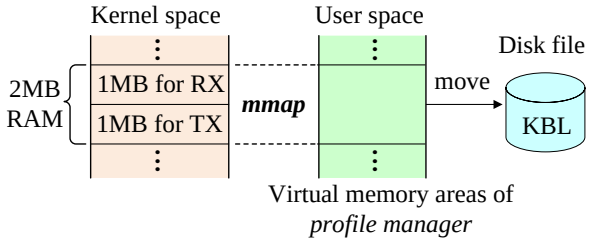

Fig. 4. *mmap* background-transfer logging mechanism.

B. *Integrated Analyzer*

Integrated analyzer provides a GUI for the presentation of integrated log information, including the processing flow and timing information of each protocol data in protocol stack and the processing time in each handover phase. As Fig. 5 illustrates, integrated analyzer consists of log matcher, call procedure analyzer, time analyzer and GUI.

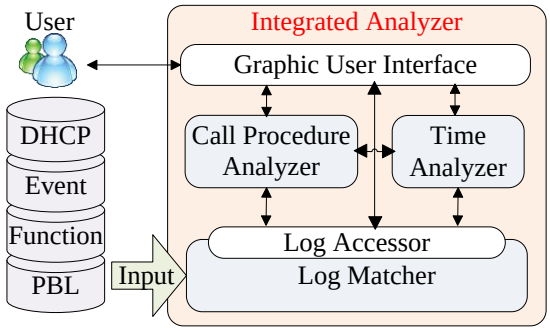

Fig. 5. Architecture of Integrated Analyzer.

1) *Log matcher*

Log matcher reads data from KBL, PBL and DHCP log. It attempts to match records and find associations between KBL and PBL. Users can select one protocol data from PBL and let log matcher find out all kernel functions that have ever processed this data. A reverse function-to-data lookup is also possible. To support these services, instrument module stores *footprints* (Fig. 6 (a)) in function log for the identification of specific protocol data. Instrument module writes footprints of a protocol data to the function log every time a target function accesses the protocol data. As a kernel function may access multiple protocol data that have already entered the system, one record in function log may contain several footprints (Fig. 6 (b)). Log matcher uses footprints from the function log and protocol headers from PBL to find all records corresponding to the same protocol data (Fig. 7).

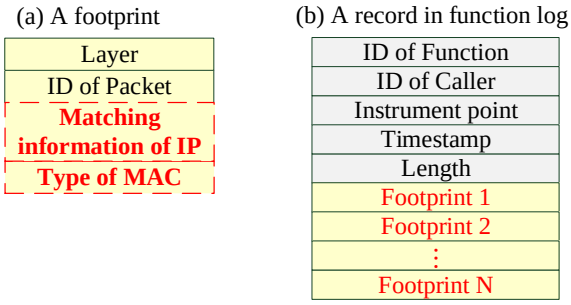

Fig. 6. (a) Format of a footprint (b) Format of a record in function log.

A footprint includes header fields of each layer (link layer, network layer, and transportation layer) to enable a layer-specific matching. In transport layer, information about IP or MAC address of a segment is generally not available to protocol functions, and port number alone does not uniquely identify this segment. As a remedy, KPAT assigns an ID that is unique in transport layer to each segment when the segment first enters the protocol stack. Since Linux kernel allocates a memory block to each protocol data for storing structure *sk_buff*, we use the memory address of *sk_buff* as the segment's transport-layer ID.

According to RFC 791 [27], the identification filed of IP header in each packet should be unique during the period of a source-destination connection. We use a 4-tuple data (destination IP address, source IP Address, identification and protocol number) to uniquely identify a layer-3 protocol data. In case that a protocol data do not include IP header (e.g., ARP or RARP), the type field of MAC header and timestamp information are used for identification.

2) Call procedure analyzer

Call procedure analyzer shows by what functions and in what sequence a selected protocol data was processed in the kernel. This is done by tracing the ID of the protocol data in footprints. Call procedure analyzer uses call graph to present hierarchical calling relationship among logged kernel functions. For instance, the call graph shown in Fig. 8 shows that the first function called by *ip_rcv*() is *ip_local_deliver*(), which then calls *tcp_v4_rcv*(). The call to *ip_rcv*() ends with *ip_rcv_finish*().

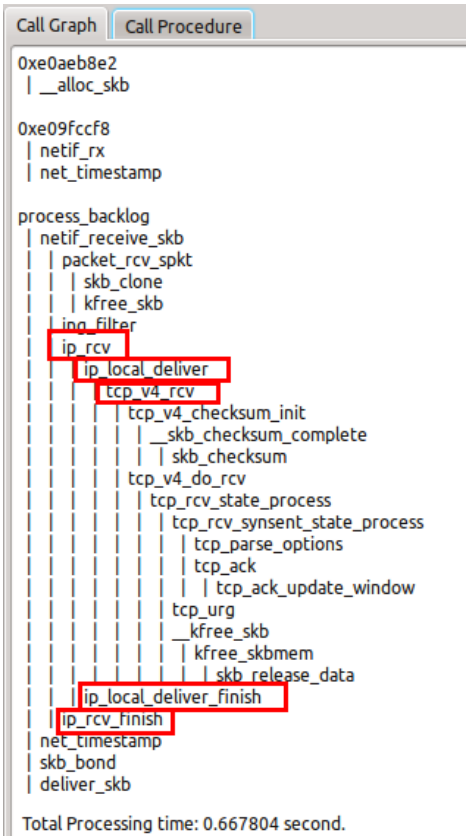

Fig. 8. Call Graph of a TCP segment.

3) Time analyzer

Time analyzer provides detailed timing information, including *protocol data processing time*, *function processing time statistic* and *handover latency*. Protocol data processing time shows the processing time of every kernel function that has processed a particular protocol data [11]. Function processing time statistic shows a list of functions that incur a processing time lower or higher than a threshold set by users. Handover latency presents the delay of each phase of a handover process.

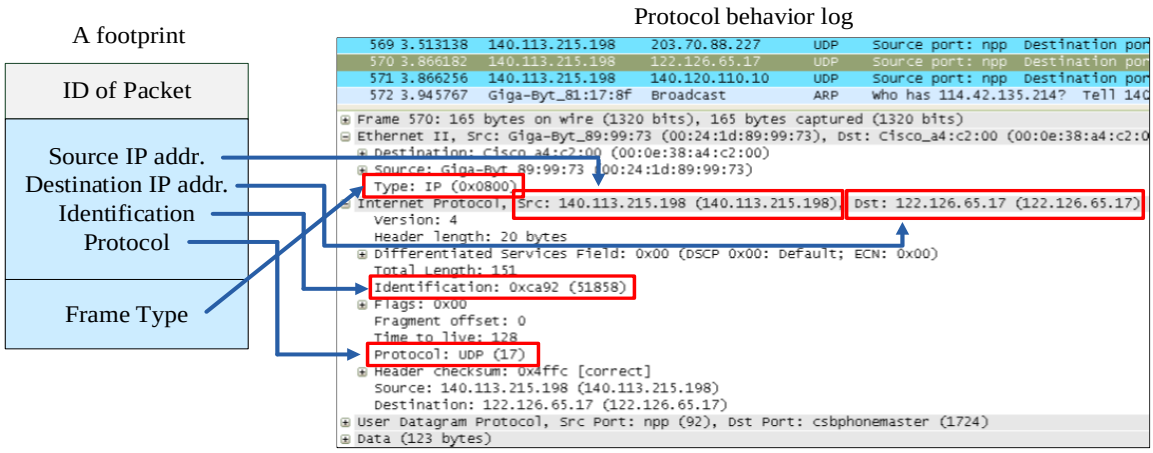

Fig. 7. The association between a footprint and a protocol data.

IV. NUMERICAL RESULTS

We conducted several experiments to study the feasibility of selective auto-instrumentation, kernel overhead, and the latency of handover process.

A. *Selective Auto-Instrumentation*

This experiment investigated whether auto-instrumentation is really needed for probe injection. We searched network-related directories (/include and /net) in Linux for *sk_buff* and all its polymorphism (alias, customized data type, and nested structure). The result shown in Table 2 reveals the number of packet processing functions and total functions. The result differs for different versions of Linux kernel. Since the number of functions to inject probes is huge, the result concludes that manual injection of probes is hardly possible.

TABLE II. FUNCTION STATISTICS IN LINUX KERNEL

	Source folder	Linux-2.6.17	Linux-2.6.21
Source files	*/include*	5492	5959
	/net	741	816
Polymorphism of *sk_buff*	*Alias*	20	20
	Customized data type	26	24
	Nested structure	367	328
Packet processing functions		6623	6813
Total functions		11175	12166

B. *Kernel Overhead*

To gauge the load imposed by KPAT on the kernel and compare the result with that of KFT [4], we measured TCP throughput with the help of iperf [29]. This experiment sets up a standalone TCP server beforehand to accept connection requests from the DUT. After the DUT connected to the TCP server, it transmitted TCP segments for throughput (called transmission rate in iperf) measurements.

Fig. 9 shows the result when we turned off the logging function of KPAT. KPAT (2.6.21-SI in the graph) yielded a result identical to that of the original kernel (2.6.21-RAW). In contrast, KFT (2.6.21-KFT) yielded a much lower transmission rate. This result indicates that TCP performance is not affected by our probes when memory logging is not performed.

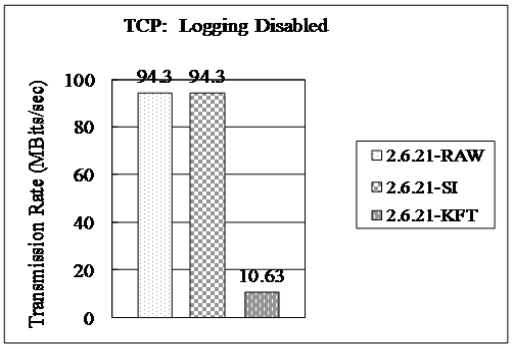

Fig. 9. TCP transmission rate – memory logging disabled.

Fig. 10 shows the result when instrument module enabled memory logging but disabled background-transfer logging. Here the performance gap between 2.6.21-RAW and 2.6.21-SI is not significant (only 0.1 Mbps). In contrast, the transmission rate of 2.6.21-KFT is much lower than those of 2.6.21-RAW and 2.6.21-SI.

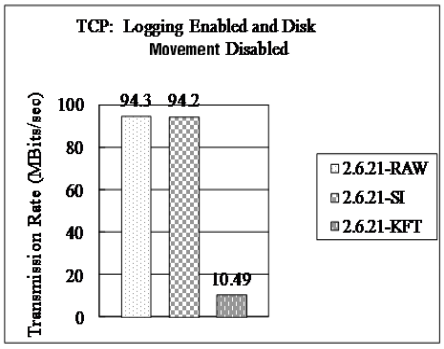

Fig. 10. TCP transmission rate - memory logging enabled and disk movement disabled.

Fig. 11 shows the result when both memory logging and background-transfer logging were enabled. The obtained transmission rate of 2.6.21-SI is only 4% lower than that of 2.6.21-RAW. In contrast, the transmission rate of 2.6.21-KFT is 94.47% lower than that of 2.6.21-RAW. These experiments show that, compared with KFT, KPAT causes a much lighter overhead to the kernel.

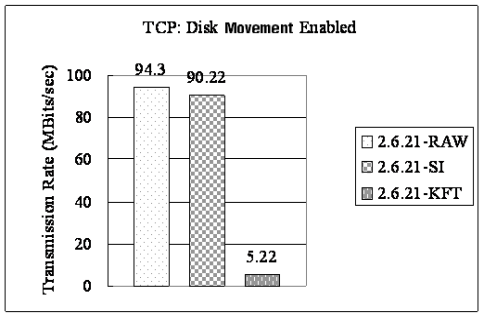

Fig. 11. TCP transmission rate - memory logging enabled and disk movement enabled.

C. *Handover Latency*

We conducted an experiment that uses KPAT to analyze handover latency (Fig. 12). This experiment sets up a VoIP communication between a mobile node, the DUT, and a corresponding node (CN). We placed the MN, a sniffer, and antennas of AP1 and AP2 in a shielding box to avoid external interference. Attenuators (attenuator1 and attenuator2) placed between access points (APs) and their antennas adjust transmission power to simulate the change of signal strength during a typical handover process.

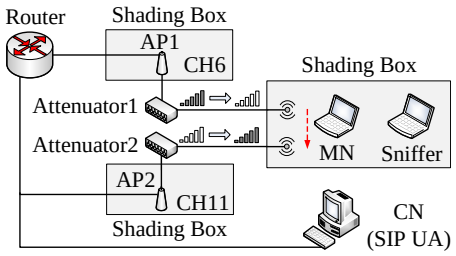

Fig. 12. Testing environment for handover latency analysis.

The MN associated with AP1 initially. The MN's handover from AP1 to AP2 was simulated by letting atenuator1 decrease the signal strength of AP1 and attenuator2 increase the signal strength of AP2. A link-down event occurred at the MN when the received signal strength dropped below a threshold. After a new link to AP2 had been established, a link-up event occurred. These two events divide different phases of the handover.

Fig. 13 shows the latency analysis of handover process generated by integrated analyzer. The integrated analyzer calculates the latency of each phase in handover process by retrieving information form the integrated log. As Fig. 13 illustrates, the handover latency is about 50 seconds. The probing phase accounts for 90% time of the entire process.

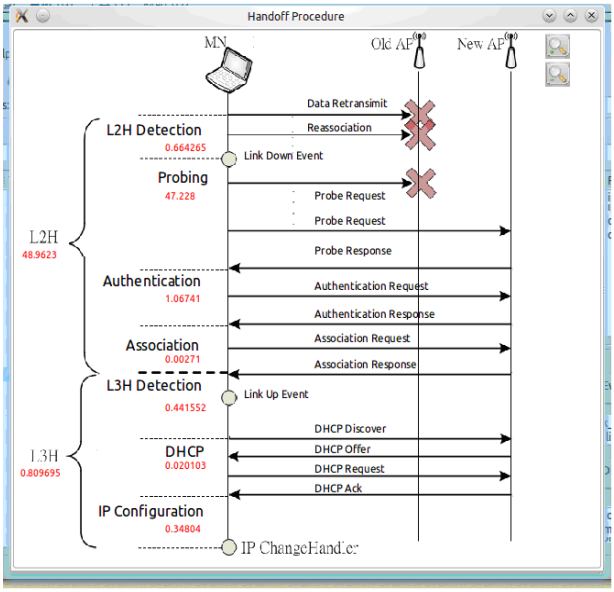

Fig. 13. Latency analysis of handover process.

V. CONCLUSIONS

We have presented the design of Kernel and Protocol Analysis Tool (KPAT). KPAT uses auto-instrument technique to systematically inject software probes into selected kernel functions without recompiling the kernel, uses Wireshark to independently collect protocol data, and uses proposed footprints as an effective technique to bind protocol data with logged functions. Consequently, KPAT allows users to observe detailed protocol behaviors such as the execution sequence and latency of network functions, processing time and the call graph for a TCP segment, and detailed handover latency. Experimental results show that KPAT imposes negligible overheads to the kernel. We successfully used KPAT to obtain detailed handover latency of mobile hosts in a wireless local area network.

ACKNOWLEDGMENT

This work was supported in part by National Science Council, Taiwan, under Grants NSC 100-2221-E-009-072-MY3, NSC 101-2221-E-009 -031 -MY3 and NSC 101-2219-E-009-028.

REFERENCES

[1] Wireshark, http://www.wireshark.org

[2] tcpdump, http://www.tcpdump.org

[3] Kismet, http://www.kismetwireless.net

[4] Kernel Function Trace, http://elinux.org/Kernel_Function_Trace

[5] Kernprof (Kernel Profiling), http://oss.sgi.com/projects/kernprof/

[6] Linux Trace Toolkit, http://www.opersys.com/ltt/

[7] Nmap project, http://nmap.org

[8] S. Best, *Linux® Debugging and Performance Tuning: Tips and Techniques*, Prentice Hall, Oct. 2005.

[9] M. Ducasse and J. Noye, "Tracing Prolog programs by source instrumentation is efficient enough," *The Journal of Logic Programming*, vol. 43, pp. 157–172, 2000.

[10] J. Levon, "Profiling in Linux HOWTO," The Linux Documentation Project, 2002.

[11] N. Joukov, A. Traeger, R. Iyer, C. P. Wright, and E. Zadok, "Operating system profiling via latency analysis," In *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2006.

[12] Y. Guo, Z. Chen, and X. Chen, "A lightweight dynamic performance monitoring framework for embedded systems," in *Proceedings of the International Conference on Embedded Software and Systems*, May 2009, pp. 256-262.

[13] A. Nataraj, A. D. Malony, S. Shende, and A. Morris, "Kernel-level measurement for integrated parallel performance views the KTAU Project," *IEEE International Conference on Cluster Computing*, pp. 1–12, 2006.

[14] KTAU, http://www.cs.uoregon.edu/research/ktau/docs.php

[15] LKST, http://lkst.sourceforge.net/

[16] R. Krishnakumar, "Kernel Korner: Kprobes - a Kernel Debugger," *Linux Journal*, vol. 2005, Issue 133, Jun. 2006.

[17] SystemTap, http://sourceware.org/systemtap/

[18] KernInst - Dynamic Kernel Instrumentation Tool Suite, http://www.paradyn.org/html/kerninst.html

[19] KLASY - Kernel Level Aspect-oriented System, Available from: http://www.csg.is.titech.ac.jp/~yanagisawa/KLASY/

[20] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic Instrumentation of Production Systems," *USENIX Annual Technical Conference*, pp. 15–28, 2004.

[21] OProfile - A System Profiler for Linux (News), http://oprofile.sourceforge.net/news/

[22] J. Levon, "OProfile Internals," 2003 [Online]. Available: http://oprofile.sourceforge.net/doc/internals/index.html

[23] Bryan Henderson, "Linux Loadable Kernel Module HOWTO," Sept. 2006 [Online]. Available: http://www.tldp.org/HOWTO/Module-HOWTO/

[24] Jongmoo Choi, "Kernel aware module verification for robust reconfigurable operating system," *Journal of Information Science and Engineering*, Vol. 23 No. 5, pp. 1339–1347, Sep. 2007.

[25] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman, *Linux Device Driver, 3rd edition*, O'Reilly, 2005

[26] K. Wehrle, F. Pahlke, H. Ritter, D. Muller, and M. Bechler, *The Linux Networking Architecture – Design and Implementation of Network Protocols in the Linux Kernel*, Prentice Hall, Apr. 2004.

[27] M. D. Rey, "Internet Protocol," IETF RFC 791, Sept. 1981.

[28] L.-H. Yen, H.-H. Chang, S.-L. Tsao, C.-C. Hung, and C.-C. Tseng, "Experimental study of mismatching ESS-subnet handoffs on IP over IEEE 802.11 WLANs," in *Eighth International Conference on Wireless and Optical Communications Networks* (*WOCN*), May 2011.

[29] Iperf, http:// sourceforge.net/projects/iperf/

[30] Memory Map, http://en.wikipedia.org/wiki/Memory_map

[31] R. Love, *Linux Kernel Development, 1st edition*, SAMS Publishing, 2003.