國立交通大學

資訊科學與工程研究所

碩 士 論 文

網路通訊協定堆疊剖析暨除錯之選擇性指令嵌入與封包關聯技術

Selective Instrumentation and Packet Association for Profiling and Debugging Network Protocol Stacks

研 究 生：李宗鴻

指導教授：曹孝櫟　教授

曾建超　教授

中 華 民 國　九 十 九 年 八 月

網路通訊協定堆疊剖析暨除錯之選擇性指令嵌入與封包關連技術

Selective Instrumentation and Packet Association for Profiling and Debugging Network Protocol Stacks

研 究 生：李宗鴻　　Student：Tsung-Hung Li

指導教授：曹孝櫟　　Advisor：Shiao-Li Tsao

曾建超　　Chien-Chao Tseng

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

June 2010

Hsinchu, Taiwan, Republic of China

中華民國九十九年八月

網路通訊協定堆疊剖析暨除錯之選擇性指令嵌入與封包關聯技術

研究生: 李宗鴻　　指導教授: 曹孝櫟 教授

曾建超 教授

國立交通大學資訊科學與工程研究所

摘 要

本研究之目的是發展選擇性指令的嵌入技術以及核心行為與網路封包的關連機制。網路通訊協定堆疊的運作，包含裝置系統內部的網路核心行為(包含核心函式的執行與核心事件的產生)，以及外顯的網路協定行為，因此，為了了解裝置的整體通訊行為，我們需要整合網路核心系統的執行紀錄，以及網路協定在裝置間交換的封包紀錄。雖然，網路封包擷取工具可以擷取封包，紀錄網路協定外顯的封包交換行為，但目前尚未有一套工具能夠有效完整地記錄核心系統內部處理封包的行為，並且將系統內部的核心行為與外顯的網路協定行為整併，以追蹤與剖析網路通訊協定堆疊的完整運作程序。

因此，本論文提出一套選擇性指令的嵌入機制，該機制可以在核心程式中選擇適當的嵌入點嵌入指令，祇記錄必要的核心行為發生的時間點，以及與封包關聯的資訊，利用這些紀錄，我們可以將核心行為的紀錄與網路封包整合，展現網通裝置的整體通訊行為，讓開發者可以剖析裝置的網路協定堆疊運作，釐清與改善網路通訊的問題，提升開發效率與通訊效能。

本論文首先於作業系統的網路通訊協定堆疊之上，設計一套追蹤網路封包處理程序的機制，該機制從核心系統的原始程式碼中，找出存取封包的函式，並且嵌入能記錄執行函式與封包相關資訊的指令於該函式內，讓開發者得以追蹤系統內部處理網路封包的行為與時間點。接著提出一套對應整體通訊行為的方法，即將核心行為記錄與網路協定行為記錄作對應，讓開發者可以進一步分析核心運作與網路封包傳遞之間的互相影響，達成整合除錯的目的。

最後利用其他手動追蹤網路封包處理程序的方法來驗證本論文尋找網路封包處理函式的準確性，以及利用網路效能測試程式，來比較本論文所提出的機制與其他核心函式追蹤工具對於作業系統處理封包速度的影響程度。實驗結果顯示，本論文所提出的追蹤封包處理程序機制能夠確實地找到網路封包處理函式，並且對於作業系統的影響程度是非常小的。另一方面，我們使用 TCP 的 Three-way-handshake 之過程，驗證本論文整併核心行為記錄與網路協定記錄的準確性。

關鍵詞：Linux、網路通訊協定堆疊、嵌入式系統、核心函式、開放原始碼、封包處理過程

Selective Instrumentation and Packet Association for Profiling and Debugging Network Protocol Stacks

Student: Tsung-Hung Li

Advisor: Dr. Shiao-Li Tsao
Dr. Chien-Chao Tseng

Institute of Computer Science and Engineering
College of Computer Science
National Chiao Tung University

Abstract

New network communication applications such as cloud computing boost the need of the small and compact interaction gadgets. Such gadgets are compact and small devices that developers have tried hard to make them more reliable and powerful. Tools like kernal profilers and packet tracer help developers to analyze the behavior of the device internally and externally, respectively.. However, there still does not have any tool that can capture the overall networking behavior and help us to investigate and debug the network protocol stack or networking protocol.

This thesis aims to develop a selective instrumentation and packet association mechanism that can automatically select network kernel functions and patch instructions to record the times functions start or termine as well as the information used to associate kernel function log with packet tracers's outputs. The selection instrumentation uses the special data-structure Linux Kernels use to maintain packets being processed to identify the network kernel functions. Experimental results show that the selective instrumentation and packet association mechanism is very effective and can indeed help to derive the overall networking behavior, including both internal network kernel operations and external communication behaviors. In the future, we could extend the work of this thesis to develop an analyzer that can fileter irrelated information and automatically thynthesize the networking behavior of interest, and help users to identy design flaws or the bottlenecks of the networking protocols or network kernels.

誌　謝

首先，感謝曾建超老師這兩年來的指導以及支持，不管是在專業知識的訓練上、相關資訊的提供以及實驗中所需要用到的設備儀器等等，老師都給予我相當多的幫助，讓我有如此豐富的資源，以專心致力於完成這篇論文。

感謝各位口試委員，包括張弘鑫老師、曹孝櫟老師以及林盈達老師，在百忙之中抽空前來參加我的碩士論文口試，並提供許多寶貴的意見，特別是曹孝櫟老師與林盈達老師對於 Linux 及嵌入式系統的開發經驗讓我有更深的體悟。

再來要感謝的是實驗室的學長們，謝謝史永健學長、黃貴笠學長、何承遠學長、何承運學長等等，在論文及口頭報告上給予我許多建設性的意見，當我遭遇困難時，學長們總是不吝於給我指導以及方向；感謝同學們：竣晨、坤穎、銘祥、奕萱、昱樺，無論是在課業或者撰寫論文的過程中，時常給予我精神上的鼓勵，讓我在交大的生活期間從來不會感到枯燥乏味；感謝學弟妹們的幫忙，讓我能專心撰寫論文，而不需要煩惱其他的雜事。

最後要感謝我的家人，感謝父母親這兩年來的支持，讓我能夠無後顧之憂，完成碩士論文，取得碩士學位。

目　錄

圖　目　錄

表　目　錄

第一章　緒論

1.1 研究動機

近年來隨著嵌入式系統(Embedded System)技術的進步與網際網路已大量建置於生活周遭，使得嵌入式網路通訊裝置也日漸普及，未來再加上雲端計算產業的發展，裝置連網會更加普遍，如今日已隨處可見的智慧型手機(Apple Iphone、Google Android)等等。而開發人員為了提升網路通訊裝置的效率、穩定性及安全性，以及提供使用者更好的服務，花費非常多的努力與時間，開發新且更好的網路通訊裝置。

然而對一個嵌入式網路通訊裝置而言，無法擁有一般個人電腦的極佳運算能力及儲存空間。因此開發人員經常關心的是(1)效能問題：如封包傳輸延遲(Packet Delivery Delay)、封包遺漏(Packet Loss)、反應時間(Response Time)，以及(2)系統內部的除錯問題：如除錯網路通訊協定堆疊的處理過程(Protocol Stack Operation Procedure)。

由於一個嵌入式網路通訊裝置的通訊行為基本上可分為兩大部分：(1)核心行為，如核心系統內部網路相關元件(Component)的行為、元件之間的訊息交換(Signaling)等，以及(2)網路協定行為，如：DHCP、TCP、UDP 等，每種網路協定皆有各自運作的行為。因此通訊裝置的效能與除錯問題會受到使用的網路類型、核心系統及網路協定等因素，而有所變化。

目前對於網路通訊裝置的通訊行為，仍然沒有一套完整的工具能夠協助開發者解決效能與除錯問題。網路封包擷取工具(Network Sniffer)取得的網路協定行為記錄(Protocol Behavior Log, PBL)，僅能協助分析網路協定行為，即網路通訊裝置的外顯網路封包類型及傳送與接收時間等，對於分析核心系統內部行為的能力較為缺乏。另一方面，現有的核心系統剖析工具(Kernel Profiler)取得的核心行為記錄(Kernel Behavior Log, KBL)，雖然可以分析核心系統內部行為，但是無法正確地分辨出網路封包的傳送與接收過程，而且無法記錄通訊行為分析所需要的網路封包相關資料。

現有的網路封包擷取工具與核心系統剖析工具也皆未能對應網路協定行為記錄與核心行為記錄，以取得網路協定行為與核心行為之間的互動與整合效果，進而剖析暨除錯核心系統的網路通訊協定堆疊。因此本論文著眼於能夠發展一套針對於網路通訊協定堆疊的剖析與除錯工具，希望能夠彌補目前工具的不足，提升網路通訊裝置的軟體開發效率與通訊效能。

1.2 研究目標

本論文的研究目標是發展一個網路通訊協定堆疊的剖析與除錯工具，讓嵌入式網路通訊裝置的開發者可以利用此工具，來剖析與除錯裝置內部的網路通訊協定堆疊，取得封包處理過程與效能資訊，其中包含有：

- 封包在網路通訊協定堆疊內部的處理過程
- 封包傳輸延遲
- 封包遺漏
- 反應時間

此外，本論文將加強功能面，以期能達到以下兩大重點：

- 我們提出一種新的追蹤網路封包處理程序的機制，該機制從核心系統的原始程式碼中，找出存取封包的函式，並且嵌入指令於該函式，記錄執行函式與封包相關資訊。之後我們可以藉由函式存取網路封包資料的順序，來追蹤系統內部處理網路封包的行為與時間點。利用這些記錄，網路通訊軟體開發者可以很容易分析網路通訊的核心系統行為，釐清與改善網路通訊的問題，藉此全面剖析與除錯網路通訊協定堆疊。

- 我們提出一種新的網路通訊裝置之整體通訊行為的分析方法，將核心行為記錄與網路協定行為記錄作對應，解決現有的核心行為記錄與網路協定記錄無法整併的問題，讓開發者可以進一步分析核心運作與網路封包傳遞之間的互相影響，清楚瞭解且分析網路通訊裝置的整體通訊行為。

1.3 章節簡介

本篇論文的章節編排與內容簡介如下：

第一章：緒論，介紹本論文的研究動機，以及期望能達成的目標。

第二章：背景知識介紹，介紹本論文所應用到的相關知識，主要是針對核心剖析技術與 Linux 相關的知識，包括網路封包資料結構、核心模組、使用者空間與核心空間的行程間通訊機制。

第三章：相關研究，介紹與本論文主題相關的工具及文獻，包含現有的核心剖析工具、網路通訊協定堆疊之分析以及核心行為記錄與網路協定記錄之整併的文獻，其中包含本實驗室先前已發展的分析工具。

第四章：網路通訊協定堆疊剖析暨除錯工具之設計與架構，分別說明本論文所提出的兩種技術：選擇性指令嵌入與封包關聯技術，與其中的設計與架構

第五章：網路通訊協定堆疊剖析暨除錯工具之實作，分別說明本論文所提出的兩種技術之實作細節，以及其中針對於 Linux 核心系統所作的修改部分。

第六章：實驗結果分析及貢獻，介紹本工具效能評估的實驗結果分析以及整體貢獻

第七章：結論與未來工作，總結本論文的結果，以及未來可繼續研究的方向。

第二章　背景知識介紹

2.1 剖析暨除錯技術與相關工具

由於現在的系統越來越複雜，因此效能微調與除錯在系統設計中，占有非常重要的角色[1][2][3][4][5]。了解現有的剖析暨除錯技術與相關工具對於系統設計可以提供很多的幫助。現有的剖析暨除錯技術根據運作模式，大略可以分成三種類型：(1)Source Instrumentation、(2)Binary Instrumentation 與(3)Statistical Sampling，以下介紹剖析暨除錯技術的三種類型：

- Source Instrumentation 運作模式為修改系統原始程式碼，將指令嵌入於系統原始程式碼，達成搜集系統運作過程的效能與執行流程等等資訊，利用於剖析與除錯。
- Binary Instrumentation 運作模式為修改系統 Object Code，將指令嵌入於 Object Code，與 Source Instrumentation 類似。然而根據修改 Object Code 的時機點，更可以將之細分成(1)靜態型(Static)與(2)動態型(Dynamic)兩種。靜態型為系統於執行前，就已經修改系統的 Object Code，執行之後即不再改變。而動態型為系統於執行的時候(Runtime)，修改已經處於記憶體內部的系統 Object Code，並不是改變實體的系統 Object Code。由於現有採用 Binary Instrumentation 技術的相關工具主要是動態型，因此之後章節提到的 Binary Instrumentation 皆是指動態型。
- Statistical Sampling 運作模式為利用一個監控程式，每隔一段固定時間，監看目前 CPU 正在執行的 Instruction，統計系統執行過程中的每一個 Instruction 或者函式執行的總次數與時間長度。

而相關工具根據剖析暨除錯的對象也可以分成兩種類型：(1)User Level 與(2)Kernel Level。User Level 剖析與除錯工具的主要對象為運作於 User Space 的應用程式，如 Firefox 等等。

目前常見的 User Level 相關工具及其研究為 GNU gprof[1][3][6]、Valgrind[7]、GCC function instrumentation[3][8]、PinOS[9]、GDB(GNU Debugger)[1][10]與 DPCL[11]等等。

Kernel Level 剖析與除錯工具的主要對象為運作於 Kernel Space 的系統程式，如 Linux 核心系統與所有 Linux 核心模組。因為本論文的主要剖析與除錯對象為核心系統，之後我們將探討 Kernel Level 剖析與除錯工具。

我們可以將現有的核心剖析暨除錯技術與相關工具分類成 Table 2-1：

Table 2-1　現有的剖析暨除錯技術與相關工具

Approach	Tool
Source Instrumentation	Linux Trace Toolkit (LTT)[12]~[18]
	Kernprof[19]
	Kernel Function Trace (KFT)[8][20][21]
	Kernel Tuning and Analysis Utilities (KTAU)[22][23]
	Linux Kernel State Tracer (LKST)[24]
Binary Instrumentation	Kprobe[25][26][27][28]
	SystemTap[29][30][31][32]
	KernInst[33][34][35]
	Kernel Level Aspect-oriented System (KLASY)[36][37]
	DTrace (Solaris-Based)[38]
Statistical Sampling	Oprofile[39][40][41]

每一個 Kernel Level 剖析暨除錯工具都有其特定之目的：

- LTT、KTAU 與 LKST

主要適用於觀察核心系統整體行為，如：Average Load、Context Switch、Send Signal、Interrupt、Exception、Memory Allocation、網路封包傳送與接收等等行為；

- Kernprof 與 KFT

主要適用於觀察核心系統全部函式的執行過程，讓使用者觀察完整的核心函式 Call Graph；

- Kprobe、KernInst 與 DTrace

主要是提供 Dynamic Binary Instrumentation 的機制，讓使用者可以自由

選擇且動態地嵌入指令於任何位置的 Object Code，取得剖析與除錯所需相關資訊。然而使用 Dynamic Binary Instrumentation 的機制時，使用者必須藉由 Compiler 或者 Disassembler 的協助，事先取得目標指令(Instruction)的記憶體位址，才能動態地嵌入指令於目標指令；

- SystemTap

主要是提供一個使用介面給使用者，讓使用者藉由 SystemTap 簡化使用 Kprobe 的困難處。使用者僅需要提供目標的函式名稱，SystemTap 負責尋找目標函式的記憶體位址，並且完成設定 Kprobe；

- KLASY

與 SystemTap 的目的相同，主要提供一個使用介面給使用者，讓使用者藉由 KLASY 簡化使用 KernInst。與 System 的不同點在於使用者可以提供目標的資料結構名稱，KLASY 則負責尋找核心內部中，所有處理目標資料結構的指令(Instruction)的記憶體位址，並且完成設定 KernInst；

- Oprofile

主要是利用 CPU 內建的 Performance Counter，每間隔一段時間產生 timer interrupt，週期性監測 CPU 目前正執行的指令(Instruction)。以統計方式，剖析核心系統行為，讓使用者觀察到巨觀的核心系統行為。

以上提到的所有工具中，與本論文的目標最為相近的是 KLASY。但 KLASY 使用 Binary Instrumentation，因此核心系統必須先經由特定的 Compiler 編譯，以及必須執行於特定的裝置。而且 KLASY 的剖析層面深入到指令(Instruction)，達成精細(Fine-grained)的剖析程度，但是也使得產生的剖析資料過於龐大，對於嵌入式裝置的開發者相當不方便。而本論文的選擇性指令嵌入技術，只記錄存取網路封包的函式與封包相關資訊。

現有的 Kernel Level 剖析暨除錯工具，已經提供許多功能強大的剖析與除錯功能。但是只能剖析核心系統內部行為，無法正確地分辨出網路封包的傳送與接收過程。而且無法記錄通訊行為分析所需要的網路封包內容資料，未能整併核心行為記錄與網路協定記錄，進而剖析暨除錯核心系統的網路通訊協定堆疊。

因此本論文無法利用現有的工具達成目標，我們必須針對網路通訊協定堆疊開發一套剖析暨除錯工具。所以我們將剖析暨除錯技術的優缺點[1][2][3]整理成 Table 2-2，選擇適合本論文的剖析暨除錯技術：

Table 2-2　剖析暨除錯技術之優缺點

Approach	**Advantage**	**Drawback**
Source Instrumentation	Less Impact More Portable Flexible	Recompilation, Reinstallation, Rebooting

Binary Instrumentation	No Source Code Required	Reverse Engineering
	No Recompilation	Instruction Set Architecture (ISA) Dependence
Statistical Sampling	Low Overhead	Statistical Approximation
	Program Changeless	OS and ISA Dependence

Source Instrumentation 的優點有：(1)與 Binary Instrumentation 相比，對系統的影響比較小(Less Impact)，(2)不需要依靠特定裝置，因此可移植性很高(More Portable)，(3)容易達成專門的需求(Flexible)。另一方面，缺點是需要重新編譯系統原始程式碼，重新安裝系統，重新啟動系統。

Binary Instrumentation 的優點有：(1)不需要系統的原始程式碼(No Source Code Required)，(2)不需要重新編譯系統原始程式碼(No Recompilation)。另一方面，缺點有：(1)需要事先藉由 Compiler 或者 Disassembler 的協助，取得目標函式的記憶體位址(Reverse Engineering)，(2)需要依靠特定的 CPU 指令集(ISA Dependence)。

Statistical Sampling 的優點有：(1)對於系統的負擔很低(Low Overhead)，(2)不需要修改系統(Program Changeless)。另一方面，缺點有：(1)得到的剖析結果為統計的近似值(Statistical Approximation)，而不是精準的數值，(2)需要依靠特定的作業系統與 CPU 指令集(OS and ISA Dependence)。

因為本論文的目標是剖析暨除錯嵌入式網路通訊裝置核心系統內部的網路通訊協定堆疊，然而嵌入式網路通訊裝置的 CPU 架構非常多樣化。如果我們選用 Binary Instrumentation，則需要針對每一種 CPU 架構調整本論文的工具。而且我們需要觀察且記錄詳細的通訊行為，如果使用 Statistical Sampling，則無法得到精準的數值。另一方面，使用 Source Instrumentation 可以輕易達成本論文的目標，針對存取網路封包的函式(Procedure Level)，因此只需要粗糙(Coarse-grained)的剖析程度，藉此縮小核心行為記錄，方便開發者使用。

結論是 Source Instrumentation 具有以下優點：(1)對系統運作行為的影響較少(Less Impact)，(2)移植性高(More Portable)，及(3)可特製需求(Flexible)。因此我們採用 Source Instrumentation 實現本論文所提出的工具，在本論文的第三章，我們將會進一步地探討採用 Source Instrumentation 的剖析與除錯相關工具。

2.2 網路封包擷取工具

網路封包擷取工具(Network Sniffer)具有其他意義相近的名稱，如：網路封包過濾器(Packet Sniffer)、網路分析器(Network Analyzer)、通訊協定分析器

(Protocol Analyzer)等等。主要的目的是在特定網路之中，擷取所有的網路封包並且記錄網路封包的相關資訊，如：封包的到達時間(Arrival Time)、擷取封包的大小等等。於擷取完成之後，解析所有封包，針對每一個封包的內容依據對應的通訊協定格式，如 Figure 2-1。

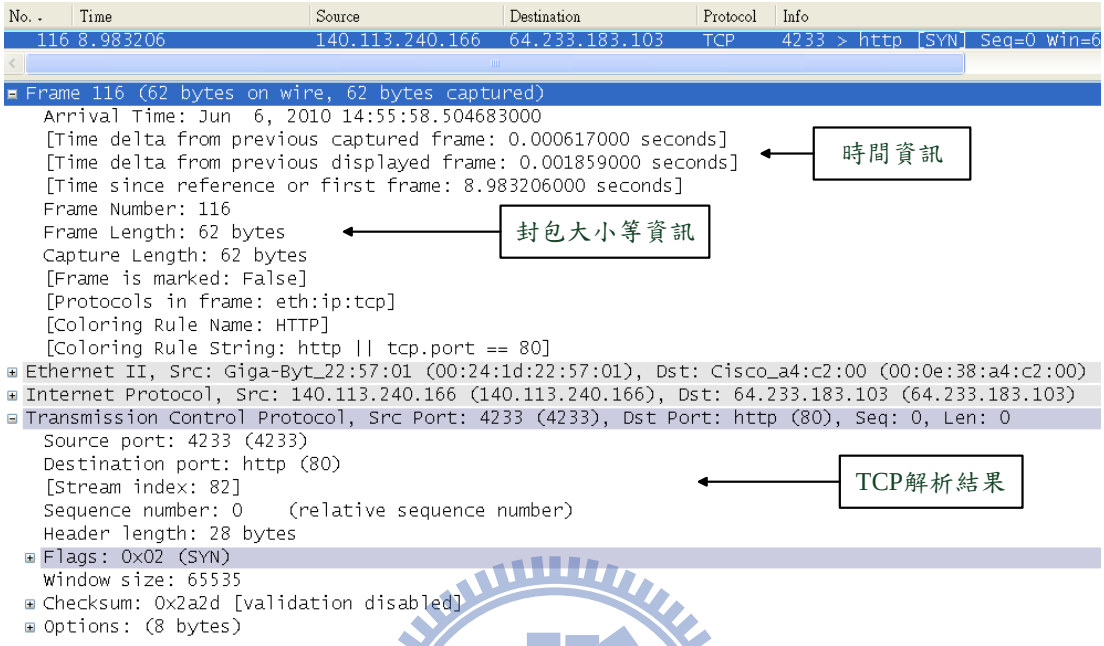

Figure 2-1　網路封包擷取工具使用畫面[42]

目前的網路封包擷取工具已經相當成熟，主要的對象包含有 Ethernet 有線網路封包、802.11 無線網路封包等等。因此無論是 Link Layer、Network Layer 或者 Transport Layer 的網路封包，皆可以利用相對應的網路協定解析所有的內容。在 Linux/Unix 平台中，網路封包擷取工具大多數是利用封包擷取函式庫(Libpcap)為基礎，達成擷取與過濾(Filter)的目的，如：Ethereal/Wireshark[42]、Kismet[43]、tcpdump[44]等等。

儘管網路封包擷取工具功能非常強大，但是也僅能分析網路通訊裝置的外顯網路封包類型及傳送與接收時間等等。對於分析核心系統內部行為的能力較為缺乏，無法剖析暨除錯核心系統的網路通訊協定堆疊。

2.3 Linux 網路封包資料結構

Linux 核心系統本身的網路功能已經相當成熟，也支援大多數的網路通訊協定，而且 Linux 擁有開放原始程式碼(Open Source)的特性，因此我們可以很容易地觀察核心系統內部的原始程式碼，比如：核心系統內部的資料儲存結構、網路通訊裝置輸入與輸出(Device I/O)行為，核心系統中斷(Interrupt)機制等等。

Linux 核心系統內部有一個重要的網路封包資料結構 *struct sk_buff*[45]，當網路封包在網路通訊協定堆疊的處理過程中，*struct sk_buff* 用於儲存網路封包內容的記憶體位址與管理網路封包的相關資訊，如 Figure 2-2。Packet data storage 即為實體儲存網路封包內容的記憶體空間，而 Management Data 即為 *struct sk_buff*

資料結構，其中最重要的資訊為儲存網路封包內容的記憶體位址與網路封包的長度(如 *head*、*data*、*tail*、*end* 和 *len* 等)，說明如下：

head 和 *end* 皆為指標，指向可以被用於儲存網路封包內容的記憶體空間；*data* 和 *tail* 也皆為指標，指向目前已經儲存網路封包內容的記憶體空間；*len* 為目前已經儲存網路封包內容的總長度。網路通訊協定堆疊中的每一個網路通訊協定可以利用這些指標與長度資訊操作網路封包內容，輕易地增加或刪去封包的 Protocol Header，讓封包在每一層網路通訊協定傳遞時，不需要重新分配記憶體空間及重複地複製封包內容，造成空間和時間的浪費，因此一個 *struct sk_buff* 將會對應於一個網路封包。

其他管理網路封包的相關資訊，如：*stamp* 為網路封包被網路卡驅動程式傳送或者接收的時間點；*dev* 為網路封包傳送或者接收的網路裝置介面；*h* 為指標指向 Transport Layer 的 Protocol Header；*nh* 為指標指向 Network Layer 的 Protocol Header；*mac* 為指標指向 Link Layer 的 Protocol Header。

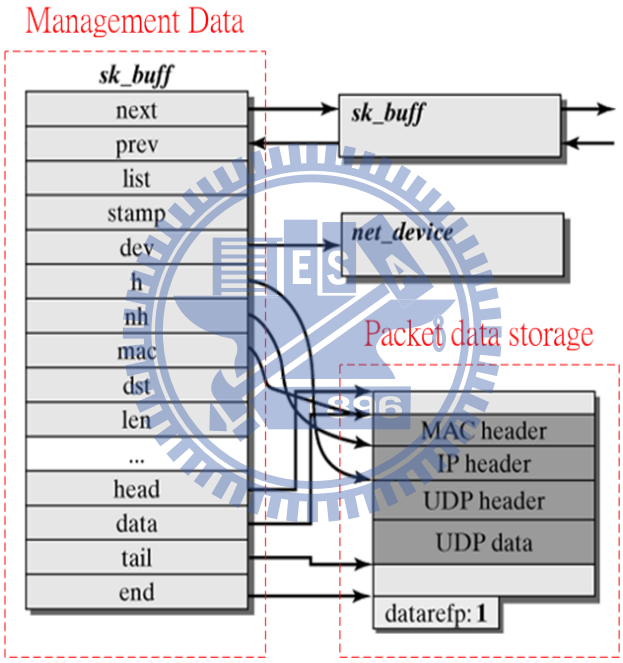

Figure 2-2　網路封包資料結構(*struct sk_buff*)

當網路封包從網路上傳送到本機端時，網路介面卡的驅動程式會負責規畫一個型態為 *struct sk_buff* 記憶空間，用於儲存接收的網路封包內容，之後網路通訊協定堆疊就可以開始處理網路封包，而驅動程式接收封包的流程，如 Figure 2-3，總共分為六個步驟[46]：

1. 驅動程式擁有一個由許多個 Packet Descriptor 組成的 Ring Buffer，而每一個 Packet Descriptor 則指向一個 *struct sk_buff*，當網路封包從網路上傳送到本機端時，網路介面卡利用 Direct Memory Access (DMA)將封包搬移到驅動程式所規劃的 *struct sk_buff*。
2. 網路介面卡啟動 Interrupt。
3. 驅動程式的 Interrupt Handler 將儲存網路介面卡相關資訊的資料結構 *struct net_device* 放入佇列(Poll_queue)中，並且啟動 Linux 中斷機制的後

半部處理(Softirq)。

4. Softirq 檢查 Poll_queue 中，目前有哪些網路介面卡擁有網路封包需要接收。
5. Softirq 向驅動程式要求 *struct sk_buff*。
6. 驅動程式將 *struct sk_buff* 從 Ring Buffer 移除後，再傳遞給 Softirq，並且重新規劃一個 *struct sk_buff* 放回 Ring Buffer。

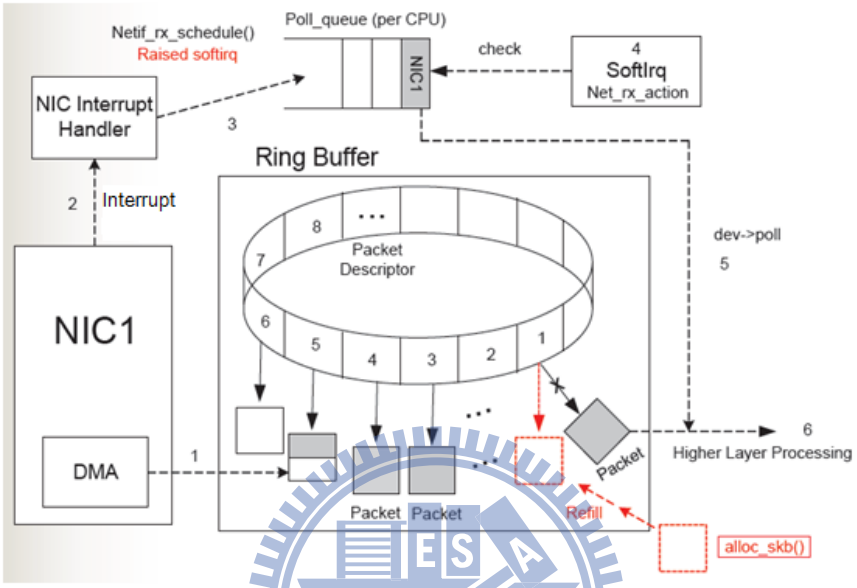

Figure 2-3　驅動程式接收網路封包的流程[46]

至於網路封包從本機端傳送到網路上的流程大致與接收的流程相同，將不再贅述，由以上所述，我們可以知道 *struct sk_buff* 對於網路通訊協定堆疊的運作是非常重要的。因此本論文得以提出選擇性指令嵌入技術，利用 *struct sk_buff*，從核心系統的原始程式碼中，找出存取網路封包的函式，並且嵌入指令，記錄處理封包的核心函式執行順序與封包關聯技術所需的封包相關資訊，幫忙開發者排除問題或找出網路通訊協定堆疊的效能瓶頸。

2.4 Linux 核心模組

Linux 核心系統是由一個基礎的核心(Base Kernel)與許多的核心模組(Linux Kernel Module)所組成，基礎核心指的是裝置啟動時需要的基本系統功能，因此在使用者編譯基礎核心時，必須將裝置啟動所需的功能，內建(Build-in)於基礎核心之中，而核心模組指的是即為未來可以動態載入的 Linux 核心模組。

當初的發展動機是在於若所有裝置的驅動程式(Device Driver)都事先內建於核心系統之中，則核心系統的規模會非常大，對於裝置的儲存空間、系統的啟動效率與記憶體使用率都會造成不良的影響，而且對於編製專屬於某一個裝置的核心系統非常不方便。因此事先只將核心系統納入最基本的功能，即為基礎核心，其餘選擇性的功能則改用可以動態載入的核心模組達成。

動態載入 Linux 核心模組主要提供一種延伸 Linux 核心系統功能的方法，不

需要重新編譯基礎核心，所以不會增加基礎核心的規模，也不需重新啟動裝置。當 Linux 核心系統需要某項功能時，才動態載入對應的核心模組，不需要此項功能時，則可以卸除此核心模組，讓 Linux 核心系統回復到原始狀態。

目前 Linux 核心模組主要利用於以下六種情形[47][48]：

1. 裝置驅動程式(Device Driver)

一個裝置驅動程式是針對特定的硬體裝置所設計，讓 Linux 核心系統利用裝置驅動程式與硬體裝置溝通，而不需要了解硬體裝置運作的任何細節。因此 Linux 核心系統想要使用任何硬體裝置，則必須擁有該硬體裝置所對應的裝置驅動程式，如：PCI 裝置有對應的驅動程式，USB 裝置也有對應的驅動程式。

2. 檔案系統驅動程式(Filesystem Driver)

一個檔案系統驅動程式是針對特定的檔案系統格式所設計，負責解讀檔案系統的內容，而檔案系統的內容通常是儲存於硬碟的資料，解讀的結果便是檔案與目錄。目前有許多種儲存檔案與目錄於硬碟的檔案系統格式，常見的格式有 Windows 系統的 NTFS、FAT 與 Linux 系統的 ext3、ext2，每一種皆有其對應的檔案系統驅動程式。

3. 系統呼叫(System Call)

User Space 的程式可以透過系統呼叫取得核心系統提供的服務，如：存取檔案、啟動一個新的 Process、關閉核心系統等等，絕大多數的標準系統呼叫被視為 Linux 核心系統必需的功能。因此內建(Built-in)於基本核心，但是使用者也可以利用 Linux 核心模組，自行開發系統呼叫。

4. 網路驅動程式(Network Driver)

一個網路驅動程式是針對特定的網路通訊協定所設計，而 Linux 網路通訊協定堆疊則是由許多個網路驅動程式所組合而成，用於傳送與接收各種網路通訊協定的網路封包，如：Linux 核心系統想要處理 IPX 的網路封包，則需要擁有 IPX 的網路驅動程式。

5. TTY 管制線(TTY Line Discipline)

TTY 管制線基本上是裝置驅動程式的擴充，協助裝置驅動程式轉換終端機裝置(Terminal Device)傳送與接收的資料格式。

6. 執行檔直譯器(Executable Interpreter)

一個執行檔直譯器負責載入與執行一個執行檔，然而執行檔的格式有許多種，因此每一種格式的執行檔皆需要對應的執行檔直譯器，才能讓執行檔執行於 Linux 核心系統，而 Linux 核心系統中，最常見的是 ELF (Executable and Linkage Format)格式。

由於本論文需要取得核心系統與網路協定互動的相關重要資訊，諸如時間標記(Timestamp)、封包相關資訊和核心函式執行順序等等，而且我們希望降低本論文對於原始 Linux 核心系統的影響。因此我們利用 Linux 核心模組的動態載入特性，當載入 Linux 核心模組時，才開始記錄核心行為，記錄完畢之後，卸載 Linux 核心模組，讓 Linux 核心系統回復到原本的狀態。

2.5 Linux 使用者空間和核心空間的行程間通訊機制

Linux 透過權限管理，將之區分成使用者空間(User Space)與核心空間(Kernel Space)，使用者執行的應用程式運行於使用者空間，而核心系統則是運行於核心空間，彼此之間不能直接存取記憶體。因此由核心系統提供訊息溝通機制，讓應用程式與核心系統之間可以交換訊息，常見的溝通機制有：ioctl[49][50]、Netlink[51]、procfs[1]、Memory MAP[49]、relayfs[52][53]。

ioctl 為 Input/Output Control，主要目的是讓應用程式透過 ioctl 傳送命令給核心系統，而核心系統會根據命令，執行相對應的動作，最後將結果回傳給應用程式。許多網路管理程式即是透過 ioctl 控制核心系統，如：iptable、ifconfig、iwconfig 等等。

Netlink 為一種 Linux 特有的 Socket，讓應用程式與核心系統具有連線的狀態，與其他的行程間通訊機制相比，Netlink 的優勢是可以讓使用者空間和核心空間進行雙向的資料傳輸，而不僅是核心空間對於使用者空間的請求回應相對訊息，所以核心系統可以主動傳輸訊息給應用程式。

procfs 為 Process File System，一個虛擬檔案系統(Virtual File System)，核心系統利用檔案的方式，讓應用程式取得核心系統內部的資訊，如：記憶體使用狀態、CPU 的資訊等等。虛擬檔案與一般檔案的不同點在於虛擬檔案的資訊是儲存於核心空間的記憶體，所以當應用程式讀取虛擬檔案時，核心系統直接將資訊從核心空間複製到使用者空間，讓應用程式得以取得資訊。但是 procfs 並未提供應用程式存取虛擬檔案所需的處理函式(Handler Function)，因此使用者必須自行撰寫處理函式，不方便使用者使用。

Memory MAP (MMAP)為一個記憶體映射機制，由於應用程式的記憶體空間(Process Address Space)為虛擬記憶體位址(Virtual Memory Address)，無法直接存取核心空間的記憶體位址。所以 MMAP 將核心系統內部的一塊記憶體空間對映到應用程式的記憶體空間，讓應用程式得以直接存取核心空間的記憶體位址，進而取得核心系統內部的資料，不需要執行額外的複製動作。但是應用程式存取核心空間的內部資料時，必須特別注意應用程式之間或中斷環境(Interrupt Context)與程序環境(Process Context)之間是否會發生 Race Condition。然而 MMAP 並未提供上述問題的解決方法，因此 MMAP 對於本論文需要取得大量且精確的資料是不方便使用的。

relayfs 為 Data Relay File System，與 procfs 相同也是一個虛擬檔案系統，但是 relayfs 的目的為提供一個高效率的資訊傳輸機制，用來傳輸大量且持續的串流資料從核心空間到使用者空間。relayfs 的架構圖如 Figure 2-4，relayfs 為每一個 CPU(cpu0)各自配置一塊記憶體空間(kbuf0)，並將 kbuf0 對應到一個管理檔案的資料結構(/cpu0)，核心系統則利用 *relay_write* 將資訊寫入 kbuf0。當 relayfs 被掛載(Mount)時，才產生虛擬檔案(/mnt/cpu0)於 User Space，因此應用程式才得以使用 MMAP 存取/mnt/cpu0，將大量且持續的串流資料傳輸到 User Space。relayfs 的設計架構可以避免多處理器的應用程式之間發生 Race Condition。另一方面，雖然 relayfs 利用 MMAP，但是內部已經實作相關程式，避免發生中斷環境與程

序環境之間發生 Race Condition 的問題，確保資料的正確性。因此我們可以利用 relayfs 達成本論文需要取得大量且精確的資料的目的。

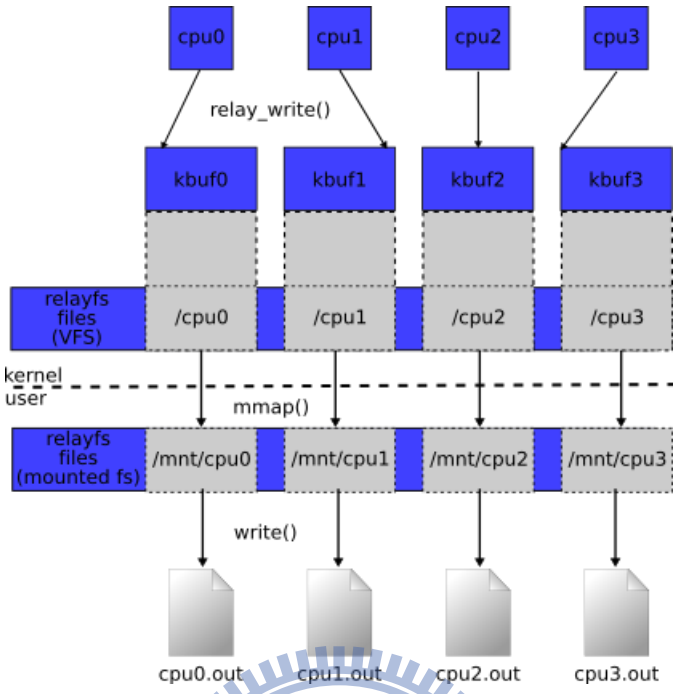

Figure 2-4　Basic Components of relayfs[53]

relayfs 更提供存取檔案所需的處理函式(Handler Function)，因此當使用者在使用 relayfs 時，不需要自行撰寫處理函式。User Space 的應用程式只需要利用存取檔案的相關系統呼叫(System Call)，如：*read*、*write* 等等，即可存取 relayfs 的虛擬檔案。

relayfs 的使用方法與 procfs 相類似，通常使用者利用 Linux 核心模組可動態載入的特性，當載入核心模組時，才配置記憶體空間與產生相關的虛擬檔案。因此 relayfs 的使用方法大多數是以核心模組的方式撰寫而成，如 Figure 2-5。

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/relay.h> /*Necessary when using the relayfs*/
static struct rchan *    chan;
int init_module(void)
{
        chan = relay_open("cpu", dir, subbuf_size, n_subbufs, &relay_callbacks);
        if(!chan)   {
                printk(KERN_ALERT "Couldn't create relay channel.\n");
                return -ENOMEM;
        }
        return 0;
}
void cleanup_module(void)
{
        if(chan)    {
                relay_close(chan);
                chan = NULL;
        }
}
```

Figure 2-5　relayfs 檔案系統之程式範例

Figure 2-5 為一個標準的 Linux 核心模組程式架構，包括必要的標頭檔案 module.h 和 kernel.h，以及模組的初始化函式 *init_module*、結束函式 *cleanup module*。因為我們將以核心模組的方式來產生 relayfs 的虛擬檔案，所以必須引入 relayfs 的標頭檔案 relay.h。之後於核心模組的初始化函式 *init_module*，執行 relayfs 提供的函式 *relay_open*，配置記憶體空間與產生虛擬檔案，而 *relay_open* 的回傳值即為管理此檔案的資料結構 *struct rchan*。最後於核心模組的結束函式 *cleanup_module*，執行 relayfs 提供的函式 *relay_close*，釋放記憶體空間與虛擬檔案，達成動態產生虛擬檔案的目的。

第三章　相關研究

3.1 Linux Kernel Profilers

現有的 Linux Kernel Profilers 非常多，因此本節將只介紹與本論文相同採用 Source Instrumentation 的代表性工具：Linux Trace Toolkit (LTT)[12]~[18]、Kernprof[19]和 Kernel Function Trace (KFT) [8][20][21]。

3.1.1 Linux Trace Toolkit (LTT) [12]~[18]

LTT 手動嵌入指令於 Linux 核心系統內部的各種重要核心事件觸發點，以剖析整體 Linux 核心系統的運作行為，例如：中斷、排程和記憶體存取等等。但是對於剖析網路通訊協定堆疊行為，LTT 僅嵌入指令於傳送網路封包與接收網路封包的核心事件觸發點，因此使用者只能得知核心系統傳送與接收封包的動作，無法詳細追蹤網路封包於網路通訊協定堆疊內部的處理過程。而且 LTT 為使用手動方式嵌入指令，所以 LTT 提供幾乎所有版本 Linux Kernel Patch[18]，讓 LTT 支援所有版本 Linux Kernel。但是嵌入式系統並不只有 Linux 而已，所以 LTT 並不方便系統開發者使用。

3.1.2 Kernprof[19]

Kernprof 利用 Gcc Compiler 提供的-pg 選項，自動嵌入指令於每一個核心函式，並且給予一個編號。在系統運作的期間，記錄每一個函式被呼叫的次數與運作時間，以及每一個函式與呼叫該函式/被該函式呼叫的函式(Parent/Child)之間的關連，產生所有函式的 Call Graph，如 Figure 3-1。

```
index % time    self  children    called     name
                0.12    6.52  405453/405453      sys_sendto [14]
[17]     4.5    0.12    6.52  405453         sock_sendmsg [17]
                0.13    6.39  405453/405453      inet_sendmsg [19]
-----------------------------------------------
                0.54    5.99  551531/551531      do_softirq [15]
[18]     4.4    0.54    5.99  551531         net_rx_action [18]
                0.12    5.87  581182/581182      ip_rcv [22]
-----------------------------------------------
                0.13    6.39  405453/405453      sock_sendmsg [17]
[19]     4.4    0.13    6.39  405453         inet_sendmsg [19]
                1.72    4.67  405453/405453      tcp_sendmsg [20]
```

Parent
Primary Function
Child

Figure 3-1　Kernprof 核心函式追蹤畫面

圖中每一個使用長虛線間隔的區塊，代表一個主要函式(Primary Function)的 Call Graph。位於 Primary Function 之上的函式皆為此 Primary Funciton 的 Parent，

位於 Primary Function 之下的函式皆為此 Primary Function 的 Child。其中的數據：index 指 Kernprof 給予 Primary Function 的編號、%time 指 Primary Function 的使用時間比、self 指 Primary Function 的總運行時間、children 指 Primary Function 呼叫 Child 的總運行時間、called 指被呼叫的總次數、name 指函式的名稱及編號。

使用者可以藉由 Kernprof 的分析結果得到核心系統內部所有函式之間的關聯性，並且剖析核心系統的效能。但是使用者依然無法藉由此分析結果，進而有效率地且正確地分辨出處理封包的核心函式執行順序，如 Figure 3-1。圖中橢圓虛線所指的 *do_softirq* 為與網路封包處理過程無關的函式。除非使用者事先追蹤過 Linux 網路通訊協定堆疊的原始碼，或者研究有關 Linux 網路通訊協定堆疊的書籍，否則無法判斷。

而且 Kernprof 對於核心系統效能剖析的準確度稍微不足，Kernprof 只能提供到 0.01 s 的準確度，網路通訊協定堆疊中的網路封包處理函式通常都需要達到 μs 的準確度，因此 Kernprof 無法達到詳細剖析網路封包於網路通訊協定堆疊內部的處理效能。

3.1.3 Kernel Function Trace (KFT) [8][20][21]

KFT 與 Kernprof 類似，KFT 利用 Gcc Compiler 提供-finstrument-functions 選項，自動嵌入指令於每一個核心函式，完整取得核心系統中核心函式的運行過程，如 Figure 3-2。其中：Entry 指進入該函式的時間點，Delta 指該函式的運行時間，如第一個函式*_local_bh_enable*，運行時間為 1μs，PID 指該函式運行時的 Process ID，PID 為-1.0 表示核心系統，Function 指該函式的名稱，Caller 指呼叫該函式的函式名稱。使用者可以利用 KFT 的結果，詳細觀察核心系統中所有的函式，並且取得每一次函式執行時間，準確度高達 μs，對於分析核心系統的效能瓶頸很有幫助。

Entry	Delta	PID	Function	Caller
3186	1	-1.0	_local_bh_enable	__do_softirq+0x1f2
3187	0	-1.0	_local_bh_enable	__do_softirq+0x1f2
3194	1	3942.0	do_softirq	smp_apic_timer_interrupt+0x111
3195	0	3942.0	do_softirq	smp_apic_timer_interrupt+0x111
3196	0	3942.0	do_softirq	smp_apic_timer_interrupt+0x111
3197	1	3942.0	smp_apic_timer_interrupt	apic_timer_interrupt+0x28
3200	0	3942.0	tcp_recvmsg	sock_common_recvmsg+0x53
3201	4	3942.0	tcp_cleanup_rbuf	tcp_recvmsg+0x1094

Figure 3-2　KFT 核心函式追蹤畫面

但是 KFT 與 Kernprof 擁有相同問題，使用者無法藉由 KFT 的分析結果，有效率且正確地分辨出處理封包的核心函式順序，如 Figure 3-2，圖中橢圓虛線所指的*_local_bh_enable*、*do_softirq* 與 *smp_apic_timer_interrupt* 即為與網路封包存取無關的函式。

KFT 與 Kernprof 除了無法正確地分辨處理封包的核心函式執行順序之外，KFT 與 Kernprof 利用 compiler 自動嵌入指令於每一個核心函式，雖然可以得到非常細部的核心系統行為，但是對於核心系統產生的負擔也很大，從 KFT 的文

件[8]中，可以得知 KFT 在 Communication 的 overhead 高達 600~1000%，因此對於網路通訊協定堆疊無法得到精準的剖析結果。

另一方面，LTT、KFT 與 Kernprof 都無法記錄本論文中封包關聯技術所需要的封包相關資訊，因此無法取得某一封包於核心系統的處理過程與時間點，也無法幫忙開發者排除問題或找出網路通訊協定堆疊的效能瓶頸。

3.2 網路通訊協定堆疊之分析

網路通訊協定堆疊之分析的議題[54][55][56]已被探討很久，除了 3.1 節所介紹的 Linux Kernel Profiler 之外，另有許多論文[37][46][48][57][58]以此為方向進行研究，本節將探討與本論文的研究目標相吻合的相關論文。

- **A dynamic aspect-oriented system for OS kernels[37]**

Yanagisawa, etc.提出一個技術-KLASY，目標為簡化使用 KernInst[34]會遭遇到的困難處。使用者使用 KernInst 時，需事先取得目標指令(Instruction)的記憶體位址，才能設定 KernInst 動態地嵌入指令於目標指令。使用者使用 KLASY 時，只需提供目標資料結構的名稱，KLASY 負責找出目標資料結構被處理的指令位址，並且完成設定 KernInst[34]且嵌入指令。該論文以網路通訊堆疊之剖析為例子，但是 KLASY 所產生的分析資料過於龐大，使用者無法有效率地追蹤處理封包的核心函式執行順序，且此技術需要特定的編譯器(Compiler)協助，對於嵌入式裝置的開發者相當不方便。

- **The performance analysis of linux networking – Packet receiving[46]**

Wenji Wu, etc.事先追蹤 Linux 核心程式碼之後，將 Linux 系統接收網路封包的過程分為三個階段，分別依序為 NIC & Device Driver、Kernel Protocol Stack 和 Data Receiving Process，並且手動嵌入指令於這三階段的部分函式內，探討內部的運作過程，以及取得實驗結果。雖然該論文並未詳細分析封包於這三階段的內部處理程序，但是找出其中的重要函式，因此本論文將會比較選擇性嵌入指令與該論文兩者找出的函式，以驗證選擇性嵌入指令的功能。

- **Design and Implementation of an Efficient and Configurable Instrument Platform for Linux Network Protocol Stack[48]**

該論文為本實驗室的曹敏峰所提出的，作者手動嵌入指令於 Linux 網路通訊協定堆疊中的每個重要函式，目標是分析 Linux 核心系統行為以及獲得網路通訊協定與 Linux 核心系統互動的相關重要資訊。

然而作者認為如果嵌入直接記錄資訊的指令於核心系統之中，缺點在於如果使用者想要變更記錄的資訊時，則必須重新嵌入指令，重新編譯系統以及重新啟動系統。因此為了改善此缺點，作者設計一個動態指令嵌入平台(Configurable Instrument Platform)，如 Figure 3-3。於此平台之中，事先嵌入於核心系統程式碼內部的指令，只是讓核心系統轉移控制權以及傳遞資訊的指令(Probe)。因此當核心系統執行到 Probe 時，即將控制權轉移到核心模組(Kernel Module)，讓核心模

組之中的記錄資訊指令，真正地記錄資訊，藉此達成動態的指令嵌入功能。

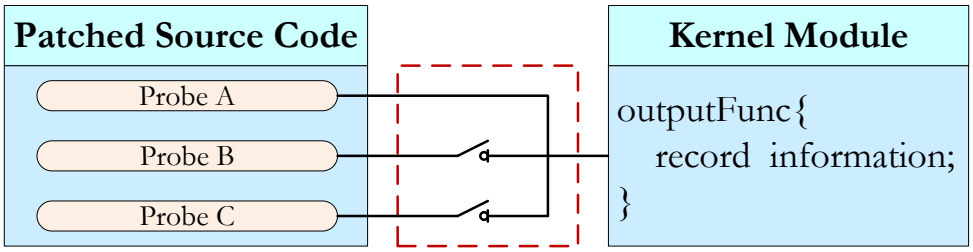

Figure 3-3　動態指令嵌入平台[48]

該論文的實驗結果顯示此平台可以有效率地取得核心系統行為以及封包相關的重要資訊，但是由於該論文只嵌入指令於網路通訊協定堆疊中的重要函式。因此無法取得封包的詳細處理程序，所以該論文還是不足以完整分析網路通訊協定堆疊。本論文參考該論文的方法然後加上與網路協定行為記錄整併的結果。

3.3 核心行為記錄與網路協定記錄之整併

- **Application-Specific Packet Capturing using Kernel Probes[28]**

Byungjoon Lee, etc.提出一個工具，目標為擷取特定應用程式所傳送與接收的網路封包，利用 Kprobes[25]於 Linux 核心系統內部取得處理網路封包的 Process 名稱與封包的 5-tuple 資料，最後對應系統內部取得的資料與 Sniffer 擷取的網路協定行為記錄，即可篩選出特定應用程式所傳送與接收的網路封包。該論文的實驗結果顯示篩選準確度很高，但是其目的為完整取得特定應用程式傳送與接收的網路封包，只需要封包的 5-tuple 即可，然而卻無法涵蓋所有網路協定，如：ICMP、ARP 等，因此無法分析 Linux 整體通訊行為。

- **Design and Implementation of a Networking Behavior Analysis Tool for Linux Operating System[59]**

該論文為本實驗室的徐勝威所提出的，目標是設計一套網路通訊裝置的評比工具，分析與衡量網路通訊裝置的通訊行為，協助開發者尋找網路通訊裝置的效能瓶頸。此工具分析的通訊行為包括：網路封包在核心系統內部的傳輸/接收過程、網路封包傳送/接收時的延遲、IP address 的取得和 Routing Table 的變化。該論文整併核心行為記錄與網路協定記錄，來達成分析與衡量通訊行為的目標。

於該論文中，整併核心行為記錄與網路協定記錄時，利用以下總共七項資訊作為參考點，如 Figure 3-4：

- Ethernet MAC Header：

  (1)Destination MAC Address、(2)Source MAC Address、(3)Type

- IP Header：

  (4)Destination IP Address、(5)Source IP Address、(6)Identification

- (7)網路封包的長度(Packet Length)

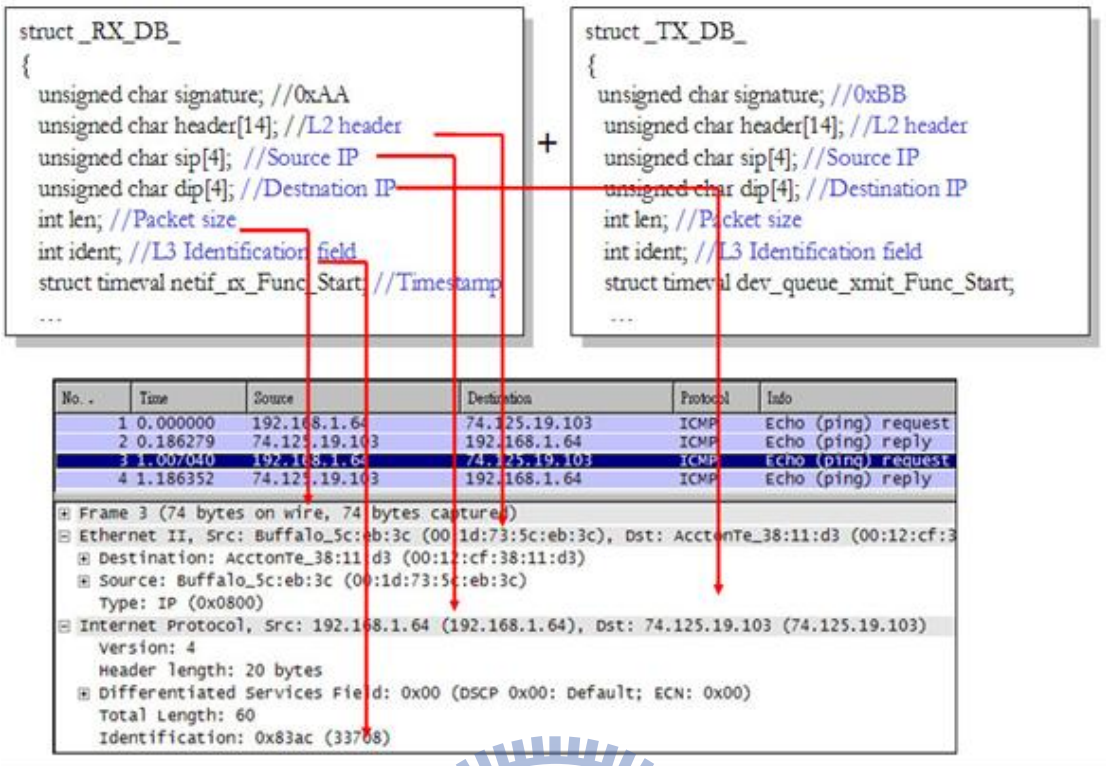

Figure 3-4　核心行為記錄與網路協定行為記錄整併[59]

然而該論文所提出的整併參考點並不是全部皆為必要的資訊，如網路封包的長度資訊對於整併是不必要的，因為有些控制類型的網路封包長度是固定的，比如 TCP 的 SYN，所以整併此類網路封包不必要使用長度資訊。另一方面，對於 Transport Layer 以上的網路協定封包，Ethernet MAC Header 的資訊是不必要的。

而且也有不足的部分，如缺少 IP Header 的 Protocol 欄位。我們從 RFC 791[60] 得知：在一段時間內，一對 Source 和 Destination 的同一種網路協定連線中，IP Header 的 Identification 必須為唯一的。因此該論文只利用 IP Header 的 Source IP、Destination IP 與 Identification 是不夠的，需要再加入 Protocol 以驗證 Identification 於不同網路協定連線之間重複的情況。

上述論文主要於整併核心行為記錄與網路協定記錄，各自有各自的考量與方法。本論文的目標為利用最少資訊於整併核心行為記錄與網路協定記錄，協助開發者正確界定造成網路封包延遲與遺漏的原因，進而分析問題及改善缺失。

第四章　網路通訊協定堆疊剖析暨除錯工具之設計與架構

本章將介紹本論文中的網路通訊協定堆疊剖析暨除錯工具。首先會介紹此工具的設計概念及設計目標，再來介紹完整的系統架構，最後介紹本工具所提出的技術，及其中的每一個元件(Component)。

4.1 設計概念與目標

網路通訊協定堆疊剖析暨除錯工具主要的用途是剖析暨除錯嵌入式網路通訊裝置核心系統內部的網路通訊協定堆疊，本工具透過指令嵌入技術取得核心系統與網路協定互動的相關重要資訊，諸如時間標記(Timestamp)、網路封包相關資訊和核心函式執行順序等等。之後將取得的核心行為記錄(Kernel Behavior Log, KBL)，與網路封包擷取工具取得的網路協定行為記錄(Protocol Behavior Log, PBL)進行整併，即可剖析暨除錯核心系統內部的網路通訊協定堆疊，更能進一步地分析網路封包傳送與接收過程的效能。

本工具的主要目標是從核心系統原始程式的執行過程之中，去剖析暨除錯網路通訊協定堆疊的運作，然而原始程式非常龐大且牽涉到太多與網路通訊協定無關的處理，因而提出兩個方法來找出相關程式片段。此兩方法為選擇性指令嵌入(Selective Instrumentation)與封包關聯(Packet Association)技術。

選擇性指令嵌入技術即是將指令嵌入於系統中的原始程式碼，因此產生兩種實作方式：手動(Manually)或自動(Automatically)，於 3.1 節介紹的 LTT 與 3.2 節介紹的動態指令嵌入平台[48]是使用手動方式。另一方面，於 3.1 節介紹的 KFT 與 Kernprof 是使用自動方式，但是這四者都無法達到本工具的目標。

由於核心系統版本一直更新與改變，連帶改變核心系統內部的核心行為，因此網路封包處理路徑也有可能改變，封包處理路徑的改變有可能是因為封包處理函式增加或者減少。如果我們選用手動方式將指令嵌入於核心系統的原始程式碼，則需要仿效 LTT 提供所有版本 Linux Kernel Patch，讓本工具達到對於 Linux Kernel 的高支援度。然而提供所有版本 Linux Kernel Patch，則需要耗費許多人力去檢驗以及嵌入指令於所有版本 Linux Kernel。所以使用手動方式以達成本工具的目標是非常沒有效率且容易出錯，因此我們決定採用自動方式，期望得出一個方法，達成自動嵌入指令於核心系統的封包處理函式。

封包關聯技術是為取得核心行為與網路協定行為之間的互動與整合效果，以剖析暨除錯核心系統內部的網路通訊協定堆疊，而現有的核心系統剖析工具與網路封包擷取工具應用於整併 KBL 與 PBL 有以下缺點：

- 核心系統剖析工具通常未考量與網路協定記錄整併，因此取得的 KBL 之中，僅包含函式的運行時間與執行順序，如 Figure 3-1 與 Figure 3-2，

因此無法整併核心行為記錄與網路協定記錄。

- 網路封包擷取工具取得的 PBL，通常只有完整的封包內容與擷取到網路封包時的時間標記(Timestamp)，如 Figure 2-1。然而在網際網路環境之下，我們必須考慮到網路封包擷取工具與核心系統剖析工具為各自獨立的事實，因此 KBL 與 PBL 之間會有些許的時間差，所以時間標記不適合作為整併的依據。

因此我們期望改進上述的缺點，達成整併核心行為記錄與網路協定行為記錄為目標。由於網路協定記錄已經包含有完整的封包內容，因此我們僅需要從核心系統內部取得封包的相關資訊，即可作為整併的參考點。本技術則希望於不影響核心系統運作的行為情況之下，從核心系統內部取得最少的封包相關資訊，達成整併 KBL 與 PBL 的目標。

4.2 系統架構圖

Figure 4-1 是本論文所提出的系統架構圖：

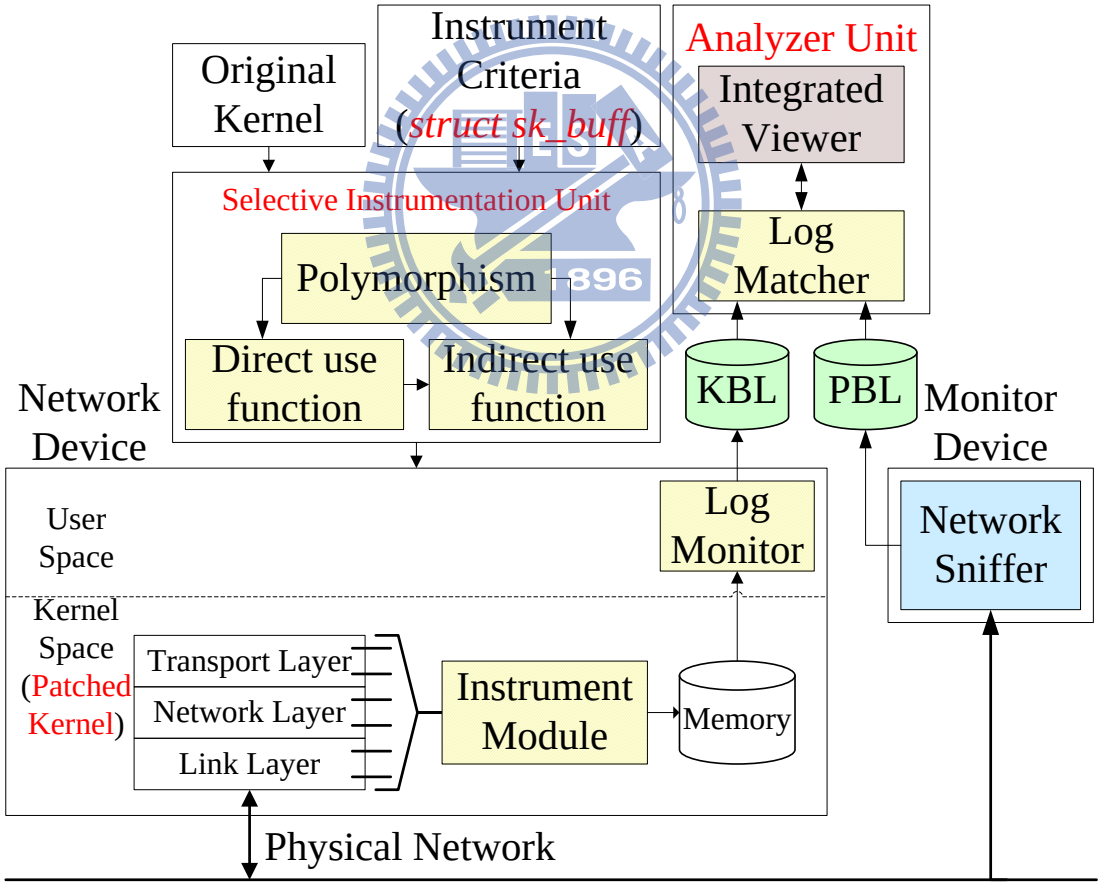

Figure 4-1　論文系統架構圖

Figure 4-1 中主要有兩大部分：

- 選擇性指令嵌入(Selective Instrumentation Unit)技術

本技術利用原始核心系統程式碼(Original Kernel)中，與網路封包處理相關的資料結構 *struct sk_buff* 作為尋找的基準點(Instrument Criteria)，先利用 Polymorphism Module 找出所有 *struct sk_buff* 的變形，再利用 Direct use function Module 與 Indirect use function Module 找出原始核心系統內部實質處理網路封包之函式，並且將剖析與除錯指令自動嵌入於函式內，以產生嵌入剖析與除錯指令的核心系統程式碼。我們編譯已經嵌入指令的核心系統程式碼，產生修改過的核心系統(Patched Kernel)並且將之安裝於 Network Device。

- 封包關聯(Packet Association)技術

本技術設計被嵌入的指令必須記錄的最少封包相關資訊及它經過處理函式的順序與時間點，此記錄稱之為 KBL。我們將實際記錄資訊的指令實作於 Instrument Module，讓 Instrument 先將 KBL 於存放 Memory 之中，避免過多的存取硬碟造成負擔。最後由 Log Monitor 從 Memory 取得 KBL，此時才存放到硬碟成實體檔案。另一方面，Monitor Device 利用封包擷取工具(Network Sniffer)擷取 PBL，可以判斷是否有遺失網路封包，以及分析網路通訊裝置的外顯網路協定行為。之後利用 Log Matcher 將 KBL 與 PBL 作對應，讓使用者清楚地觀察封包經過核心系統所留下的處理過程。最後使用 Integrated Viewer 從 KBL 與 PBL 之中，標示出特定網路封包進出核心系統與其在外的網路協定行為，協助開發者排除問題或者找出網路通訊協定堆疊的效能瓶頸。然而，Integrated Viewer 的診斷功能不包含於本論文的討論之中，我們將之列為未來工作的項目。以下將分別介紹選擇性指令嵌入技術與封包關聯技術。

4.3 選擇性指令嵌入技術之設計

本技術首先選擇觀察 Linux，主要原因是因為 Linux 作業系統本身的網路功能已經相當成熟，也支援大多數的網路通訊協定，而且 Linux 的原始程式碼是開放且容易取得，因此我們可以很容易地觀察到核心系統內部的原始程式碼。

觀察 Linux 核心系統的原始程式碼之後，發現網路封包處理函式皆擁有一個共同點，這些函式都至少傳遞或回傳一個結構型態為 *struct sk_buff* 的參數，以及宣告全域變數與區域變數的型態為 *struct sk_buff*，如 Figure 4-2。

A

```
struct sk_buff skb;
FuncA(…) {
   Access skb;
}
```

B

```
int ip_build_and_send_pkt(struct sk_buff *skb, struct sock *sk,
              __be32 saddr, __be32 daddr, struct ip_options *opt)
```

C

```
static inline struct sk_buff *tcp_recv_skb(struct sock *sk,
                                           u32 seq, u32 *off)
```

D

```
FuncB(…) {
   struct sk_buff skb;
   Access skb;
}
```

Figure 4-2　網路封包處理函式(A：全域變數，B：傳遞參數，C：回傳參數，D：區域變數)

而 *struct sk_buff* 為核心系統內部用於管理與儲存網路封包的資料結構[45]，因此得知網路封包處理函式是利用函式呼叫中傳遞與回傳 *struct sk_buff* 參數的特性來維護與交換網路封包的內容與相關資訊，所以我們可以利用 *struct sk_buff* 作為關鍵字尋找出所有的網路封包處理函式。

大部分的這類函式會直接宣告 *struct sk_buff* 作為參數或者變數，我們稱此特性為資料型態的直接引用(Direct use)。如 Figure 4-2 即是 Direct use。然而單純使用 *struct sk_buff* 作為尋找的關鍵字，會遭遇到以下兩種情況：

- Polymorphism of *sk_buff*

  由於程式語言的關係，程式開發者可以讓一種資料型態存在於多種變化形式之中，所以單純使用 *struct sk_buff* 作為尋找的字句，並未能有效且完整地尋找網路封包處理函式。

- Indirect use of *sk_buff*

  有些例外的網路封包處理函式並未直接使用 *struct sk_buff* 或其變形作為參數資料型態或者回傳資料型態，但是卻透過呼叫 Direct use function 或者被 Direct use function 呼叫的關係，間接地實質存取 *struct sk_buff*。

由於 Linux 核心系統主要是 C 語言開發而成，因此以下將以 C 語言為範例，詳細介紹這兩種情況。

4.3.1 Polymorphism of *sk_buff*

Polymorphism of *sk_buff* 的情形是 *struct sk_buff* 存在於多種變形之中，因此我們可以將其分類為以下三種：(1)別名(Alias)，(2)客製化資料型態(Customized Data Types)，(3)巢狀結構(Nested Structure)，如 Figure 4-3。

Alias

```
#define SK_BUFF_ALIAS struct sk_buff
```

Customized Data Types

```
typedef struct sk_buff SK_BUFF_TYPE;
```

Nested Structure

```
struct B    {
        …
        struct A a;
        …
};
```

```
struct A    {
        …
        struct sk_buff skb;
        …
};
```

Figure 4-3 Polymorphism of *sk_buff*

- Alias

即是利用 Macro 產生一個新的識別名稱取代 *struct sk_buff*，Compiler 會於原始程式碼進行編譯之前，執行 Preprocessor 將新的識別名稱替換還原成 *struct sk_buff*，最後才編譯原始程式碼。

- Customized Data Types

利用 *typedef* 將 *struct sk_buff* 重新定義其識別名稱，與 Alias 不同的地方是 Customized Data Types 於 Compiler 編譯程式碼時才轉換新的識別名稱成 *struct sk_buff*，而不是於 Preprocessor 階段。

- Nested Structure

將 *struct sk_buff* 與其他資料型態組合成一個新的資料型態，因此新的資料型態與 *struct sk_buff* 產生關聯，網路封包處理函式則可以透過新的資料型態存取 *struct sk_buff*。

然而實際上有可能存在 Multi-level Polymorphism of *sk_buff*，如 Figure 4-4，*struct sk_buff* 存在於 Alias，而 *struct sk_buff* 再經由 Alias 存在於 Nested Structure，因此我們將考慮 Multi-level Polymorphism of *sk_buff* 的情況。

```
Alias of sk_buff
Nested Structure  {
        …
        Alias of sk_buff
        …
};
```

Figure 4-4 Multi-level Polymorphism of *sk_buff*

所以我們必須於尋找封包處理函式之前，尋找程式原始碼內部 *struct sk_buff* 存在的多種變化形式，始能判斷函式是否至少傳遞或回傳一個結構型態為 *struct sk_buff* 或其變形的參數，以準確找出封包處理函式。而本論文將此類的封包處理函式，歸類為 Direct use function。

4.3.2 Direct / Indirect use of *sk_buff*

Indirect use of *sk_buff* 的情形是有些例外的網路封包處理函式利用 Direct use of *sk_buff* 的特性，透過呼叫 Direct use function 或者被 Direct use function 呼叫的關係，間接地實質存取 *struct sk_buff*，因此我們可以再將其細分為以下兩種情況：(1)Indirect Caller，(2)Indirect Callee，如 Figure 4-5。

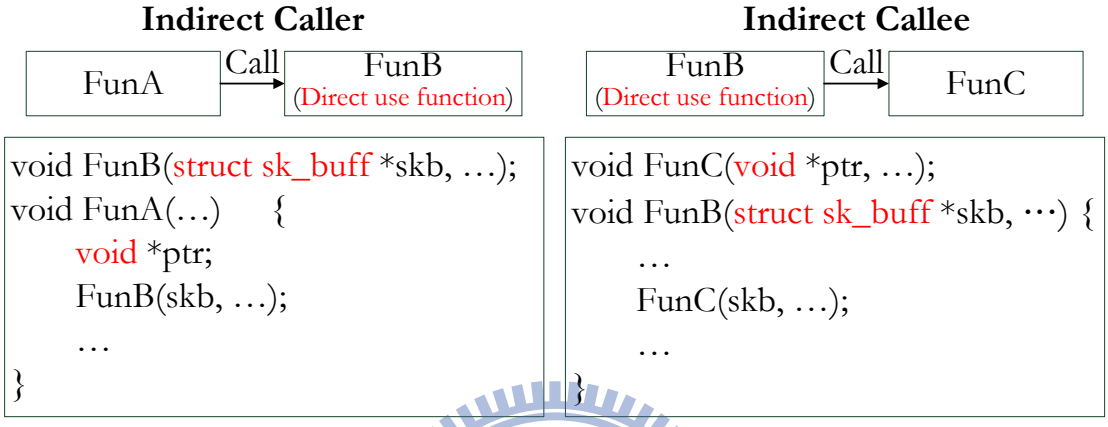

Figure 4-5 Indirect use of *sk_buff*

- Indirect Caller

  由於 Direct use of *sk_buff*，Direct use function 必須從 Caller 取得 *struct sk_buff*，始能處理網路封包，如 Figure 4-5 所表示，FunA 將 *struct sk_buff* 透過函式呼叫傳遞給 FunB，讓 FunB 得以存取 *struct sk_buff*，因此 FunA 藉由呼叫 FunB 間接存取 *struct sk_buff*。另一方面，FunA 也可能透過呼叫 FunB，讓 FunB 回傳 *struct sk_buff*。

- Indirect Callee

  Direct use function 可能透過函式呼叫將 *struct sk_buff* 傳遞給 Callee，如 Figure 4-5 所示，FunB 將 *struct sk_buff* 透過函式呼叫傳遞給 FunC，讓 FunC 得以存取 *struct sk_buff*。另一方面，FunB 也可能透過呼叫 FunC，讓 FunC 回傳 *struct sk_buff*。

所以我們必須先尋找所有的 Direct use function 之後，再透過 Direct use function 搜尋核心系統內部所有函式之間的呼叫關係，以準確找出利用 Indirect use of *sk_buff* 的網路封包處理函式，因此我們將此類函式，歸類為 Indirect use function。

然而也可能存在 Multi-level Indirect use of *sk_buff*，由於 Indirect use of *sk_buff* 分成 Indirect Caller 與 Indirect Callee，我們首先說明 Multi-level of Indirect Caller，如 Figure 4-6，FunD 為 Indirect use function，FunD 呼叫 FunB 傳遞 *struct sk_buff*，但是 FunD 卻是因為被 FunE 呼叫，從 FunE 取得 *struct sk_buff*，因此 FunE 也成為 Indirect use function。

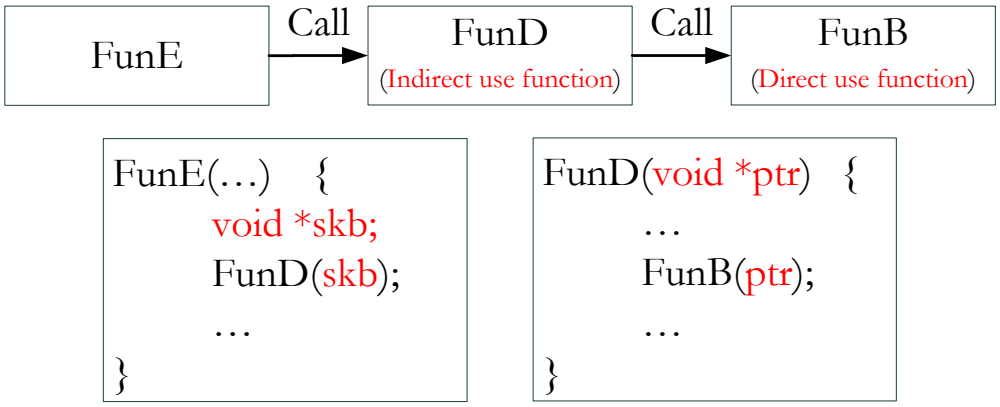

Figure 4-6 Multi-level of Indirect Caller

而 Multi-level Indirect Callee，如 Figure 4-7，FunF 為 Indirect use function，而 FunG 因為被 FunF 呼叫，從 FunF 取得 *struct sk_buff*，因此 FunG 也成為 Indirect use function。

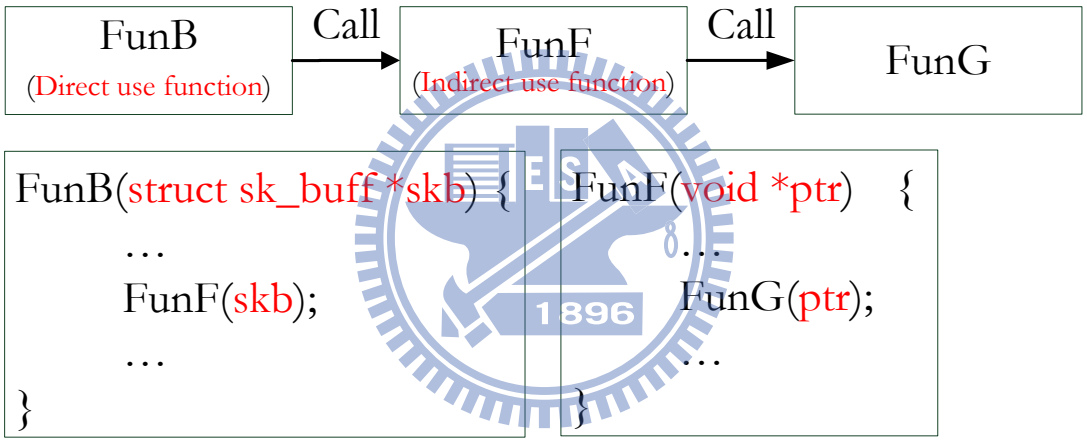

Figure 4-7 Multi-level of Indirect Callee

因此我們尋找 Indirect use function 時，也將考慮 Multi-level Indirect use of *sk_buff* 的情況，判斷函式傳遞與回傳的參數值是否與 *struct sk_buff* 有實質關聯。藉此找出所有的 Indirect use function。

結論是選擇性指令嵌入技術利用 Direcet use of *sk_buff* 的特性，並且解決 Polymorphism of *sk_buff* 與 Indirect use of *sk_buff*，自動嵌入指令於網路封包處理函式內部，得以追蹤封包處理函式的執行順序，並且記錄封包相關資訊。

4.3.3 選擇性指令嵌入技術之架構

本節將介紹選擇性指令嵌入(Selective Instrumentation)技術之架構。

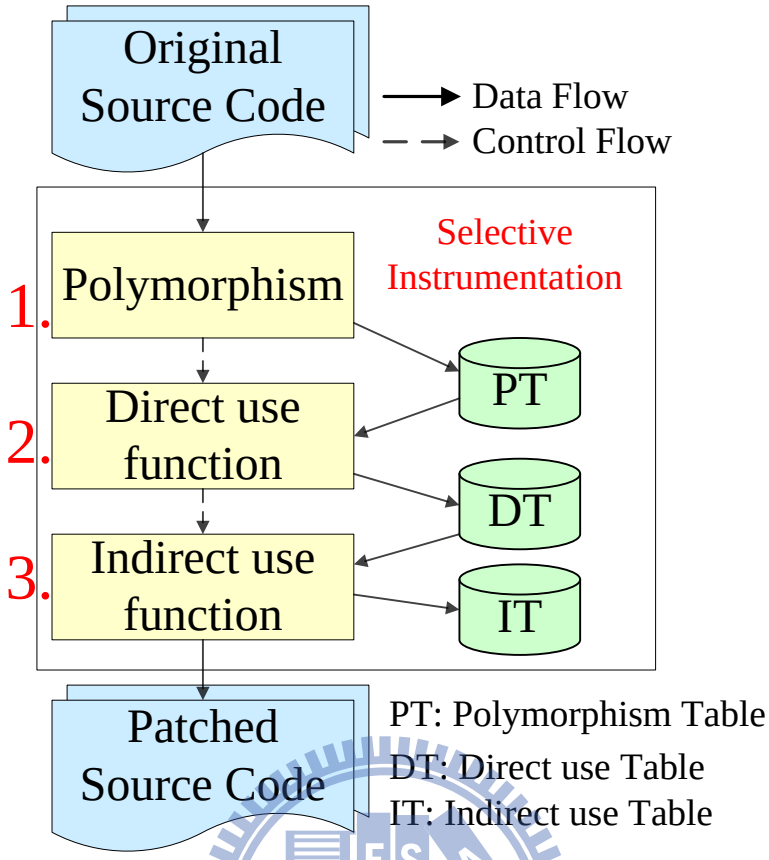

Figure 4-8　選擇性指令嵌入技術之架構

Figure 4-8 為選擇性指令嵌入技術的架構，分成三個部分：(1)Polymorphism Module 負責調查原始程式碼，以及尋找 *struct sk_buff* 的所有變形，將結果儲存於 Polymorphism Table。(2)Direct use function Module 從 Polymorphism Table 取得 *struct sk_buff* 的所有變形後，負責搜尋與嵌入指令於 Direct use function，將搜尋結果儲存於 Direct use Table。(3)Indirect use function Module 從 Direct use Table 取得所有 Direct use function 的資訊，負責搜尋與嵌入指令於 Indirect use function，最後將搜尋結果儲存於 Indirect use Table。核心系統的原始程式碼經過這三個流程之後，產生嵌入剖析與除錯指令的原始程式碼。以下將詳細介紹選擇性指令嵌入之重要元件。

- Polymorphism Module

此元件主要目的為搜尋原始程式碼，尋找 *struct sk_buff* 的所有變形，將搜尋結果儲存於 Polymorphism Table。我們利用虛擬程式碼(Pseudo Code)說明 Polymorphism Module 尋找所有變形的程序，如 Figure 4-9。搜尋所有原始程式碼，當尋找到 Polymorphism of *sk_buff*，再利用此變形繼續尋找，直到尋找完所有原始程式碼。

```
Poly(struct sk_buff)
1   For each input file
2       If find polymorphism of sk_buff
3           Then Poly(polymorphism of sk_buff)
```

Figure 4-9　Pseudo Code of Polymorphism Module

- Direct use function Module

此元件主要目的為尋找與嵌入剖析與除錯指令於 Direct use function，Direct use function Module 從 Polymorphism Table 取得 *struct sk_buff* 的所有變形之後，搜尋且嵌入指令於所有 Direct use function，最後將找到的 Direct use function 相關資訊儲存於 Direct use Table。DPPF 的相關資訊有：(1)Direct use function 的函式名稱，(2)資料型態為 *struct sk_buff* 或其變形的參數型態、名稱與位置。

- Indirect use function Module

此元件主要目的為尋找與嵌入剖析與除錯指令於 Indirect use function，Indirect use function Module 藉由 Direct use Table 取得 Direct use function 的資訊，尋找且嵌入指令於 Indirect use function，最後將結果儲存於 Indirect use Table。然而 Indirect use function 可以分為 Indirect Caller 與 Indirect Callee 兩種，因此 Indirect use function Module 將分別尋找，我們利用虛擬碼(Pseudo Code)說明 Indirect use function Module 的 Caller 與 Callee 尋找 Indirect-Caller 與 Indirect-Callee 的過程。

如 Figure 4-10，Caller 找到 Direct use function 的 Caller，而 Direct use function 的 Caller 必定為 Indirect use function。因此觀察 Indirect use function 的參數或回傳值是否與 *struct sk_buff* 有實質關連，判斷是否繼續尋找 Indirect use function。

```
Caller(Direct use function)
1   For each function
2     If find Caller of Direct use function
3       Then Indirect use function = Caller of Direct use function
4             If param. Or return of Indirect use function have association with struct sk_buff
5               Then Caller(Indirect use function)
```

Figure 4-10　Pseudo Code of Caller

如 Figure 4-11，Callee 尋找 Direct use function 的所有 function call，觀察 callee 的引數(Argument)或回傳值是否與 *struct sk_buff* 有實質關連。如果有關連，則該 callee 為 Indirect use function，以此 Indirect use function 繼續尋找 Indirect Callee。

```
Callee(Direct use function)
1   For each function call of Direct use function
2     If argu. Or return of callee have association with struct sk_buff
3       Then Indirect use function = callee
4             Callee(Indirect use function)
```

Figure 4-11　Pseudo Code of Callee

4.4 封包關聯技術之設計

本技術於核心系統內部判斷封包的類型，當封包進入傳輸層(Transport Layer)時，傳輸層的網路協定非常多種，因此無法從中找到一個具有代表性的符號作為整併的參考點，而且當封包在傳輸層時，則不會包含 MAC 或 IP Header 的資訊，所以我們將每一個封包指定一個 ID，來追蹤此封包於網路協定堆疊的運作過程。

當封包進入到網路層(Network Layer)，則封包將擁有 IP Header，我們只需要取得以下總共四項資訊作為整併的參考點：

- IP Header：

  (1)Destination IP Address、(2)Source IP Address、(3)Identification、(4)Protocol

如果封包只擁有 MAC Header，即 Layer 2.5，我們只需要取得 Ethernet MAC Header 的 Type 作為整併的參考點。由於此類型的封包像是 ARP 和 RARP 出現次數很少，所以我們只需要知道是哪一種的封包，之後再利用時間資訊與 Packet Sniffer 作整合性的分析即可。

根據以上所敘，我們設計 Footprint 來記錄每一個封包整併所需要的資訊，如 Figure 4-12。

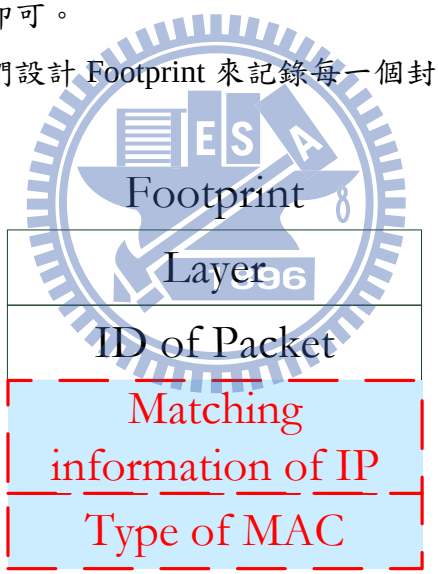

Figure 4-12　Footprint 之格式

以下將一一介紹 Footprint 格式中的每個欄位：

- *Layer*：標示此 Footprint 位於哪一層(1)L4：只有 *ID of Packet*，(2)L3：*ID of Packet* 與 IP Header 的資訊，或者(3)L2.5：*ID of Packet* 與 MAC Header 的 Type 資訊。
- *ID of Packet*：此網路封包的 ID，追蹤個別封包經過的封包處理函式執行順序。
- *Matching information of IP*：Destination IP Address、Source IP Address、Identification、Protocol
- *Type of MAC*：MAC Header 的 Type 欄位

由於 KBL 除了記錄網路封包處理函式的運行時間與執行的順序，還需要分別記錄網路封包的以上資訊，所以我們重新設計 KBL 的記錄格式，如 Figure 4-13。

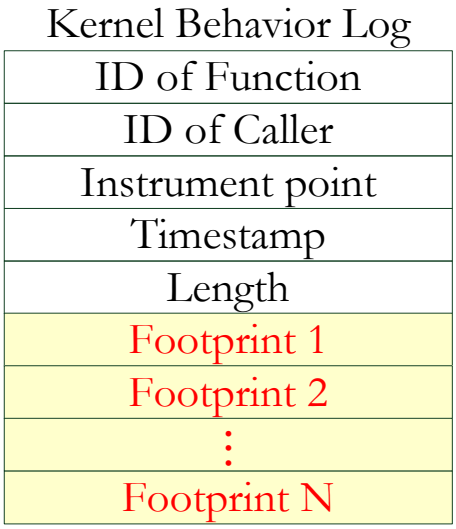

Figure 4-13　KBL 之格式

以下將一一介紹 KBL 格式中的每個欄位：

- *ID of Function*：封包處理函式的 ID
- *ID of Caller*：封包處理函式之 Caller 的 ID
- *Instrument Point*：標示此 KBL 是網路封包進入或者離開封包處理函式
- *Timestamp*：此 KBL 的時間標記
- *Length*：此 KBL 的總長度
- *Footprint*：儲存 Footprint，一個 Footprint 對應一個封包整併所需的資訊，如 Figure 4-12。因為一個封包處理函式可能同時處理多個封包，所以一個 KBL 可能包含多個 Footprints。

當網路封包進入網路封包處理函式或者從網路封包處理函式離開時，選擇性指令嵌入技術嵌入的剖析與除錯指令將會利用封包關聯技術設計的 KBL 格式，儲存所有整併 KBL 與 PBL 需要的資訊，達成本工具剖析暨除錯核心系統內部的網路通訊協定堆疊的目標，幫忙開發者排除問題或找出網路通訊協定堆疊的效能瓶頸。

4.5 總結

在本論文所使用的系統架構下，運用選擇性指令嵌入技術，解決手動指令嵌入所遇到的種種問題，包括需要耗費許多人力去檢驗，以及嵌入指令於所有版本 Linux Kernel，達成自動嵌入剖析與除錯指令於網路封包處理函式的目標。另一方面，運用封包關聯技術取得核心行為與網路協定行為之間的互動與整合效果，解決現有的核心行為記錄與網路協定記錄無法整併的問題，如此一來利用這兩種技術讓本工具達成剖析與除錯網路通訊協定堆疊的目標。

第五章　網路通訊協定堆疊剖析暨除錯工具之實作

5.1 實作環境

本工具實作於 Linux 核心系統的環境：

Linux distribution：CentOS 5.4

GUI framework：Qt 4

Compiler：GNU gcc version 4.3.4

- 選擇性指令嵌入技術之需求
  - 開發工具：Lex & Yacc
  - 開發工具版本：flex version 2.5.4 & bison version 2.3

5.2 選擇性指令嵌入技術之實作

本節將介紹我們根據第 4.3.3 節的選擇性指令嵌入技術之架構，實作本技術內部的 Polymorphism Module、Direct use function Module 與 Indirect use function Module，並且運行於 Linux 核心系統。由於該系統主要是利用 C 語言實作，然而 C 的語法規則非常複雜，無法利用簡單的字串比對程式，達成本技術尋找以及嵌入指令於函式的目的，因此我們選擇利用兩套工具：Lex & Yacc[61]來實作本技術。

因為 Lex & Yacc 主要用於實作編譯器(Compilers)及直譯器(Interpreters)，而且美國國家標準協會(ANSI)利用 Lex & Yacc 訂定 C 的語法規則標準[62][63]，所以 Lex & Yacc 非常適合於實作 C 的語法分析程式(Parser)，因此可以輕易達成本技術的目的。以下分別介紹 Polymorphism Module、Direct use function Module 與 Indirect use function Module 的實作方法。

5.2.1 Polymorphism Module 之實作

由於 Polymorphism 的三種變形，其語法規則皆不相同，因此將分別介紹：

- Alias

  因為 Alias 是由 C Preprocessor 處理，因此 ANSI 並未利用 Lex & Yacc

訂定 Alias 的標準語法規則。然而我們從 C Preprocessor 的文件[64]，得知 Alias 的語法規則為*#define* 接著一個 *identifier* 與 *token list* 直到換行，由於此語法規則較為單純，因此我們只需利用 Lex 即可實作，如 Figure 5-1。當 Lex 取得*#define* 時，即進入 Alias 的狀態，接著取得一個 *identifier*，並且執行 *define* 函式讀取 *token list*，最後判斷 *struct sk_buff* 是否存在於此 *token list*。

```
1  ^[ \t]*"#"[ \t]*"define"          { BEGIN DEFMODE; }
2  <DEFMODE>{L}({L}|{D})*  { define(); BEGIN INITIAL;}
3  <DEFMODE>\n                       { BEGIN INITIAL; }
4  define()
5     read token list
6     check if struct sk_buff exits in token list
```

Figure 5-1　Alias 的語法規則

- Customized Data Types 與 Nested Structure

因為此兩種屬於 C 語言的語法，所以 ANSI 已經訂定此兩種語法規則的標準，其中 Customized Data Types 的語法規則為 *typedef* 接著一個資料型態與一個以上的 *identifier*，而該資料型態也可以為 Nested Structure，所以我們得知此兩種變形的語法規則有重疊的部分，因此將一併實作，如 Figure 5-2。

```
1   declaration
2         : TYPEDEF type_specifier declarator_list ';'
3               { check if struct sk_buff exits in type_specifier }
4         ;
5   type_specifier
6         : TYPE_NAME
7         | struct_specifier
8         ;
9   struct_specifier
10        : STRUCT IDENTIFIER '{' struct_declaration_list '}'
11              { check if struct sk_buff exits in struct_declaration_list }
12        | STRUCT '{' struct_declaration_list '}'
13              { check if struct sk_buff exits in struct_declaration_list }
14        | STRUCT IDENTIFIER
15        ;
16  struct_declaration_list
17        : type_specifier declarator_list ';'
18        | struct_declaration_list type_specifier declarator_list ';'
19        ;
20  declarator_list
21        : declarator
22        | declarator_list ',' declarator
23        ;
24  declarator
25        : pointer IDENTIFIER
26        | IDENTIFIER
27        ;
```

Figure 5-2　Customized Data Types 與 Nested Structure 的語法規則

我們可以於 *struct_specifier* 檢查 *struct* 宣告內部是否包含 *struct sk_buff*，找出 Nested Structure，之後我們可以於 *declaration* 檢查 *typedef* 的資料結構是否為 *struct sk_buff*，找出 Customized Data Types。

5.2.2 Direct use function/Indirect use function Module 之實作

首先介紹 Direct use function Module 的實作，因為我們要找出核心系統中，傳遞或回傳結構型態為 *struct sk_buff* 之參數的所有函式，並且嵌入指令於這些函式，所以必須從核心系統的原始程式碼，找出所有函式的定義，進而找出 Direct use of function。而 C 語言的函式定義之語法規則，ANSI 也已經訂定其標準，因此我們只需利用該標準即可實作，如 Figure 5-3。

```
1   declaration_specifiers
2           : storage_class_specifier
3           | storage_class_specifier declaration_specifiers
4           | type_specifier
5           | type_specifier declaration_specifiers
6           | type_qualifier
7           | type_qualifier declaration_specifiers
8           | function_specifier
9           | function_specifier declaration_specifiers
10          ;
11  declarator
12          : pointer direct_declarator
13          | direct_declarator
14          ;
15  direct_declarator
16          : IDENTIFIER
17          | '(' declarator ')'
18          | direct_declarator '(' parameter_type_list ')'
19          | direct_declarator '(' ')'
20          ;
21  parameter_type_list
22          : parameter_list
23          | parameter_list ',' ELLIPSIS
24          ;
25  parameter_list
26          : declaration_specifiers declarator
27          | parameter_list ',' declaration_specifiers declarator
28          ;
29  function_definition
30          : declaration_specifiers declarator '{' block_item_list '}'
31                  check if struct sk_buff exit in declaration_specifiers
32                  check if struct sk_buff exit in parameter_type_list of declarator
33          ;
```

Figure 5-3　Function Definition 的語法規則

我們於 *function_definition* 檢查 *declaration_specifiers* 是否包含 *struct sk_buff*，判斷函式是否回傳資料型態為 *struct sk_buff* 的參數。接著於 *declaration* 檢查其中的 *parameter_type_list* 是否包含 *struct sk_buff*，判斷函式是否傳遞資料型態為 *struct sk_buff* 的參數，所以我們將可以判斷此函式是否為 Direct use function。

最後介紹 Indirect use function Module 的實作，因為我們要找出核心系統中，呼叫 Direct use function 或者被 Direct use function 呼叫的所有函式，並且判斷是否為 Indirect use function。所以必須從函式定義之中的敘述主體，即 Figure 5-3 中的 *block_item_list*，判斷是否為 Indirect use function。而判斷的方法已經於第 4.5.1 節的 Indirect use function Module 說明，在此不再贅述。

為了剖析與除錯網路通訊協定堆疊，我們需要測量封包於函式內部的處理時間，因此當 Direct use function Module 與 Indirect use function Module 找到封包處理函式時，需要嵌入指令於該函式的起始點(Start Point)與結束點(End Point)。一個函式只會有一個起始點，然而一個函式卻可能存在許多個結束點，因為一個函式可以於任何地方結束，並將控制權交還給 Caller Function。

在 C 語言中，擁有一個保留字(*return*)，當函式執行到 *return* 時，即交還控制權。如果函式的回傳參數其型態為 *void*，則不一定需要 *return*，當函式執行完成之後，會自動將控制權交還給 Caller Function。

因此 Direct use function Module 與 Indirect use function Module 將藉由函式回傳參數的型態以及 *return*，尋找並且嵌入指令於函式的結束點，如 Figure 5-4。

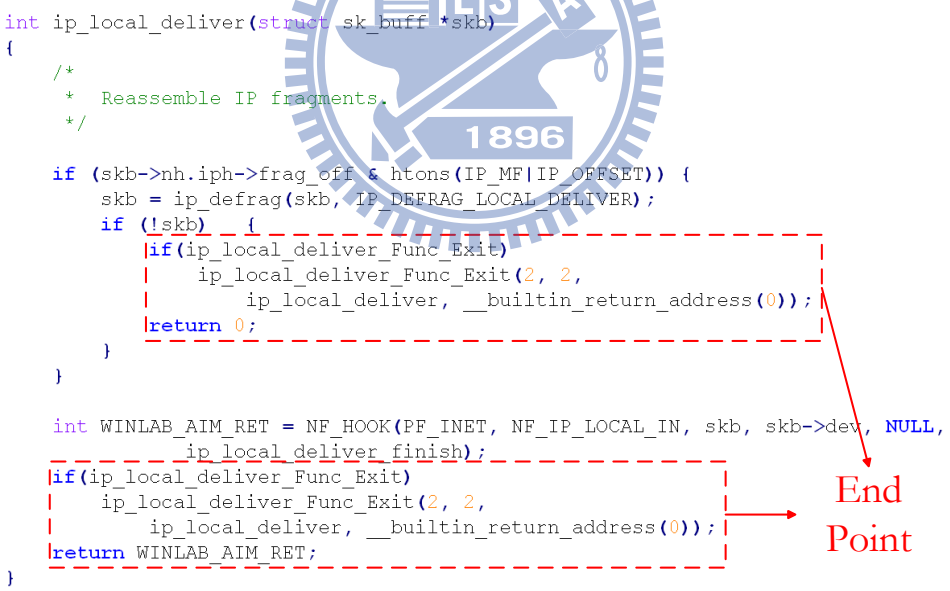

Figure 5-4　嵌入指令於函式的結束點

然而如果我們於 Linux 核心系統的原始程式碼，嵌入直接記錄資訊的指令，缺點是在於當我們想要變更輸出資訊時，必須重新執行本技術，重新編譯核心系統。

為了改善此缺點，因此本技術將建立於曹敏峰所實作的動態指令嵌入平台[48]，本論文已於 3.2 小節介紹該平台。本技術只嵌入 Probe 於封包處理函式之中，此後如果我們想要改變觀察的函式或者記錄的資訊，藉由修改核心模組，即可達成目的，不需要重新執行選擇性指令嵌入技術，以及重新編譯核心系統。

5.3 封包關聯技術之實作

本節將介紹我們根據第 4.5 節的封包關聯技術之設計實作本技術，並且運行於 Linux 核心系統。由於我們必須事先從核心系統之中取得 KBL，而本論文的選擇性指令嵌入技術，可以嵌入指令於核心系統內部所有的封包處理函式，因此我們利用該技術得到嵌入指令的核心系統，再從該核心系統取得 KBL。然而選擇性指令嵌入技術所嵌入的指令是建立於動態指令嵌入平台[48]之上，因此本技術必須實作該平台中，實際記錄資訊的核心模組，才得以記錄的最少封包相關資料及它經過處理函式的順序與時間點，再將之統整為 KBL。

5.3.1 Instrument Module 之實作

我們根據第 4.5 節的設計，實作 Instrument Module 來取得 KBL，如 Figure 4-12 與 Figure 4-13。然而 KBL 原本的設計實作於 Linux 核心系統會有困難，因此我們修改部分內容，但不影響原本的設計目的。

KBL 於 Linux 核心系統實際取得的資訊，修改的部分為 ID of Function 與 ID of Caller，我們改為取得函式與 Caller 的記憶體位址。因為 Linux 核心系統於編譯時，會產生 Kernel Symbol Table，其內部存有每一個函式與其對應的記憶體位址，如 Figure 5-5。因此我們不需要於選擇性指令嵌入技術之中，為每一個函式指定對應的 ID，並且修改原始程式碼，使得該函式取得 Caller 的 ID。因此我們利用 Gcc Compiler 的內建函式：*__builtin_return_address*，取得 Caller 的記憶體位址。如此一來，即可達成原本設計 ID of Function 與 ID of Caller 辨識執行函式的目的。

Address Function Name

```
c028fd9b T ip_append_data
c029077e T ip_send_reply
c02909c1 T ip_fragment
c029109d t ip_finish_output
c0291288 t ip_finish_output2
c02913f5 t ip_dev_loopback_xmit
c029149a T ip_generic_getfrag
c0291585 T ip_append_page
c0291a2c T ip_queue_xmit
c0291eb1 T ip_flush_pending_frames
c0291f7c T ip_build_and_send_pkt
c02921ea T ip_mc_output
c02925bc T ip_output
```

Figure 5-5　Kernel Symbol Table

KBL 中的 Footprint，修改的部分為 ID of Packet，我們改為記錄 *struct sk_buff* 的記憶體位址。因為如果要為每一個網路封包標識 ID，就必須於網路通訊協定堆疊的入口函式指定一個 ID 給每一個網路封包，然而 Linux 核心系統的網路通訊協定堆疊的入口函式不只一個，因此無法指定 ID 給每一個網路封包。最後我們觀察 Linux 核心系統於函式之間傳遞 *struct sk_buff*，是利用 Call By Reference，

所以改為記錄 *struct sk_buff* 的記憶體位址，即能達成原本利用 ID of Packet 辨識每一個網路封包的目的。

封包關聯技術的 KBL 實際儲存格式如 Figure 5-6，而 Log Matcher 即是根據此格式解析核心行為記錄，並且與 PBL 作對應，讓使用者觀察網路封包經過網路通訊協定堆疊所留下的過程。

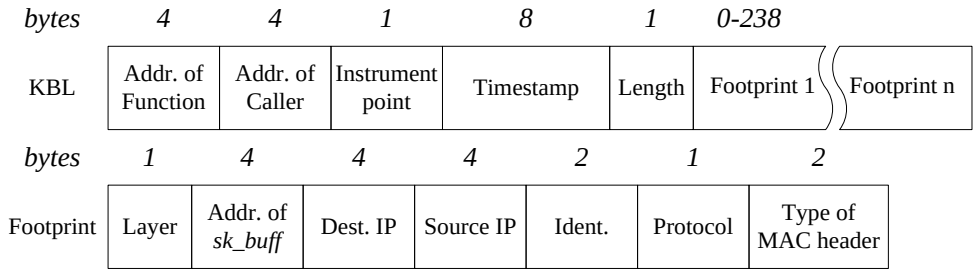

Figure 5-6　封包關聯技術實際儲存格式

然而以上修改的部分，皆需要於 Linux 核心系統中取得，再傳遞到 Instrument Module，因此 Direct use function Module 與 Indirect use function Module 將嵌入相關的指令於探針之中，之後讓 Instrument Module 取得其餘的資訊。

另一方面，由於我們必須降低本工具對於核心系統與其網路通訊協定堆疊的負擔，避免產生對於剖析與除錯結果的誤判。從先前的研究中[15][48]，得知造成系統負擔的最大因素在於記錄資訊的指令。因此我們選用對於系統負擔較小的記錄方法，即是先將資訊記錄於 Memory 之中，過一段時間之後，由 Log Monitor 將記錄的資訊從 Memory 搬移到 Disk。

因為需要將資訊記錄於記憶體之中，所以必須事先取得記憶體空間，然而我們無法於 Instrument Module 記錄資訊時動態分配記憶體空間。由於動態分配記憶體空間，將使核心系統產生 Context Switch，影響網路封包於網路通訊協定堆疊的原始處理過程，以及造成系統負擔。因此我們利用 Instrument Module 經由 relayfs 事先取得一大塊連續的記憶體空間，之後將資訊記錄於此連續的記憶體空間。

但是選擇性指令嵌入技術無法判斷尋找到的封包處理函式是運作於 Process Context 或者 Interrupt Context，因此可能會發生某一些函式正在透過 Instrument Module 記錄資訊到 Memory 時被中斷，而 Interrupt routine 之中，也會有封包處理函式透過 Instrument Module 記錄資訊到 Memory，因而產生記錄資訊混亂的情況，如 Figure 5-7。

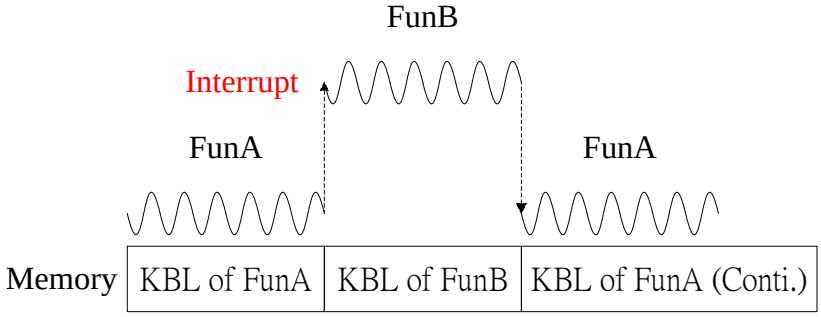

Figure 5-7　Process Context 與 Interrupt Context 的 Race Condition 情形

因此我們使用 Atomic Operation Mechanism，即是當函式記錄資訊到記憶體時關閉 Interrupt，避免產生如 Figure 5-7 的 Race Condition 情形，雖然關閉 Interrupt 會影響原本網路協定堆疊的運作，然而這是無法避免的問題，而且 KFT[8]也採用此種作法，但是我們將於第六章說明本工具對於系統的負擔，以及網路協定堆疊的影響，遠比 KFT 低。

5.3.2 Log Monitor 之實作

由於 Instrument Module 是利用 relayfs 取得記憶體空間，因此 Log Monitor 只需要利用 Linux 存取檔案的 System Call，來存取 relayfs 的虛擬檔案，即能將 Instrument Module 記錄於 Memory 的資訊搬移到 Disk，並且儲存成 KBL。

5.3.3 Log Matcher 之實作

Log Matcher 使用 Qt4 實作，讓使用者輸入整併 KBL 與 PBL 所需的封包資訊，然後根據 5.3.1 節的 Figure 5-6 之格式解析 KBL，並且找出該封包於核心系統的運作過程，如 Figure5-8。圖中方框虛線為使用 IP Header 的資訊找出該封包的運作過程。

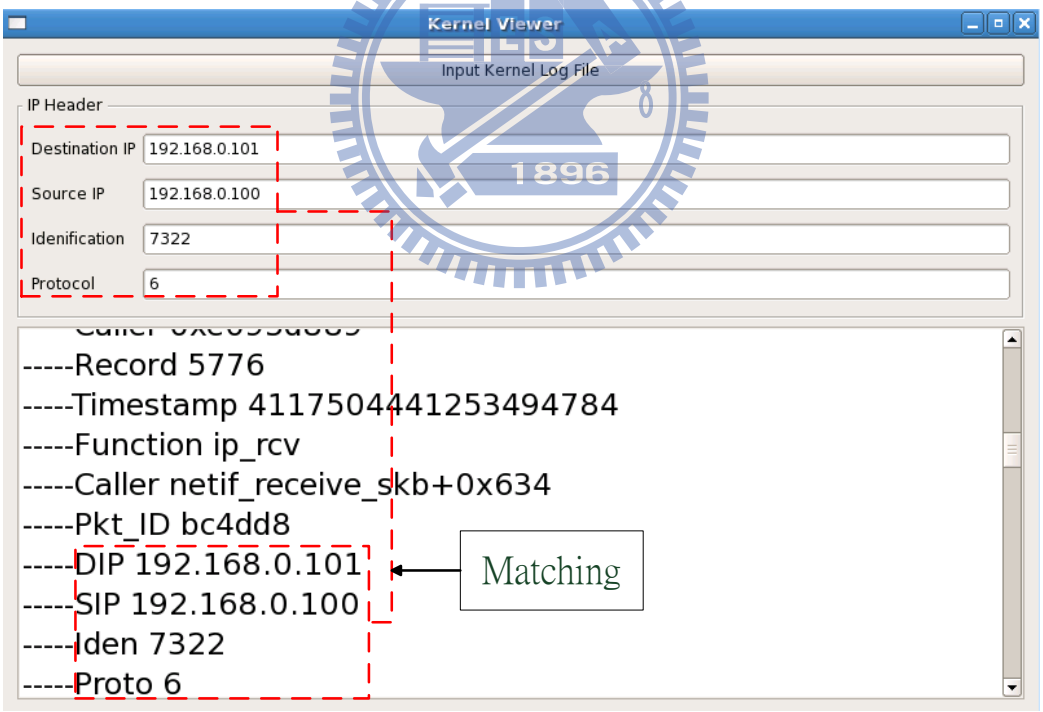

Figure 5-8　Log Matcher 之運作畫面

第六章　實驗結果與貢獻

6.1 實驗結果分析

本節將首先介紹選擇性指令嵌入技術的實驗結果，比較本論文與 KLASY、曹敏峰[48]及 Wenji Wu[46]這三篇論文找到的封包處理函式。接著介紹為本論文的剖析暨除錯工具所設計的 TCP 傳輸效能分析實驗，比較本工具與原始核心系統、KFT[8]及曹敏峰[48]，四者之間的網路效能差異。最後我們將使用 TCP 的 Three-way-handshake 來呈現本論文之 Log Matcher 整併 KBL 與 PBL 的成果。

6.1.1 選擇性指令嵌入技術的實驗結果

本技術實驗於兩種版本的 Linux 原始程式碼：2.6.17 與 2.6.21，由於原始程式碼過於龐大，我們選擇執行於網路通訊協定堆疊相關的兩個目錄底下：/include 與/net，實驗結果可見於 Table 6-1：

Table 6-1　選擇性指令嵌入技術實驗結果

	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 Types	26	24
	Nested Structure	367	328
Packet Processing Functions		6623	6813
Total Functions		11175	12166

從 Table 6-1 中，我們可以驗證當核心系統版本改變，的確連帶改變核心系統內部的核心行為，導致網路封包處理路徑也改變。Linux-2.6.21 與 Linux-2.6.17 相較之下，Linux-2.6.21 多出 190 個封包處理函式。本技術可以從核心系統的所有函式之中，篩選出封包處理函式，因此我們不需要觀察所有函式，進而減少本工具對於核心系統運作的影響

我們將本技術找到的封包處理函式與其他三篇論文：KLASY[37]、曹敏峰[48]及 Wenji Wu[46]作比較，雖然 KLASY 是分析 Linux-2.6.10 的結果，以及 Wenji Wu 是分析 Linux-2.6.12，但是不影響本實驗的比較結果，比較結果可見於 Table 6-2：(◯：該方法有找到該函式，●：該方法沒找到該函式，—：該函式不存在於該方法所分析的核心版本)

Table 6-2　選擇性指令嵌入技術與其他方法比較結果

Function Name	KLASY[37]	Wenji Wu[46]	Tsao[48]	Selective Instrumentation in Linux-2.6.17
e1000_rx_checksum	◯	●	●	●

netif_receive_skb	○	●	○	○
netif_rx	○	●	○	○
ip_rcv	○	●	○	○
ip_local_deliver	○	●	○	○
tcp_v4_rcv	○	○	○	○
tcp_rcv_established	○	○	○	○
tcp_event_data_recv	○	●	●	○
skb_copy_datagram_iovec	○	●	●	○
__kfree_skb	○	●	●	○
inet_sendmsg	●	●	○	○
tcp_sendmsg	●	●	○	○
tcp_push_one	●	●	○	○
tcp_write_xmit	●	●	○	○
tcp_connect	●	●	○	○
tcp_send_ack	●	●	○	○
tcp_send_synack	●	●	○	○
tcp_retransmit_skb	●	●	○	○
tcp_transmit_skb	●	●	○	○
udp_sendpage	●	●	○	○
udp_sendmsg	●	●	○	○
icmp_send	●	●	○	○
icmp_reply	●	●	○	○
ip_queue_xmit	●	●	○	○
ip_append_page	●	●	○	○
ip_append_data	●	●	○	○
ip_push_pending_frames	●	●	○	○
ip_output	●	●	○	○
ip_finish_output	●	●	○	○
ip_finish_output2	●	●	○	○
dev_queue_xmit	●	●	○	○
__netif_rx_schedule	●	○	○	○
net_tx_action	●	●	○	○
ip_rcv_finish	●	●	○	○
ip_local_deliver_finish	●	●	○	○
tcp_v4_do_rcv	●	○	○	○
tcp_ack	●	●	○	○
tcp_recvmsg	●	○	○	○
udp_rcv	●	○	○	○
udp_queue_rcv_skb	●	●	○	○
udp_encap_rcv	●	●	○	○
udp_recvmsg	●	●	○	○
icmp_rcv	●	●	○	○
net_rx_action	●	○	●	○

alloc_skb	●	○	—	—
tcp_rcv_state_process	●	○	●	○
sk_wait_data	●	●	●	○

從 Table 6-2 中，與曹敏峰相比之下，本技術可以找到該方法的所有函式。與 Wenji Wu 相比之下，本技術只少找到一個函式 *alloc_skb*，經過觀察我們發現該函式於 Linux-2.6.17 改名為*__alloc_skb*，且該函式屬於 Direct use of *sk_buff*，因此本技術依然可以找到該函式。與 KLASY 相比之下，本技術只比 KLASY 少找到一個封包處理函式 *e1000_rx_checksum*。然而我們發現該函式實際上為網路卡驅動程式內部的函式，所以沒有找到該函式。但是經過觀察之後，該函式屬於 Direct use of *sk_buff*，因此我們依然可以使用本技術去該驅動程式的程式碼中，找到且嵌入指令於該函式之中。

另一方面，本技術與 KLASY 都是使用自動方式嵌入指令，而且同樣使用 *struct sk_buff* 以搜尋封包處理函式，但是本技術可以找到所有的封包處理函式，因此我們將驗證 KLASY 未能找到的函式，結果可見於 Table 6-3：(P：該函式的參數屬於 Polymorphism of *sk_buff*，D：該函式屬於 Direct use of *sk_buff*，I：該函式屬於 Indirect use of *sk_buff*)。

Table 6-3　選擇性指令嵌入技術與 KLASY 比較結果

Type / Function	D	P & D
inet_sendmsg		V
tcp_sendmsg		V
tcp_push_one		V
tcp_write_xmit		V
tcp_connect		V
tcp_send_ack		V
tcp_send_synack		V
tcp_retransmit_skb		V
tcp_transmit_skb		V
udp_sendpage		V
udp_sendmsg		V
icmp_send	V	
icmp_reply	V	
ip_queue_xmit	V	
ip_append_page		V
ip_append_data		V
ip_push_pending_frames		V
ip_output	V	
ip_finish_output	V	
ip_finish_output2	V	
dev_queue_xmit	V	
__netif_rx_schedule		V
net_tx_action		V
ip_rcv_finish	V	

ip_local_deliver_finish	V	
tcp_v4_do_rcv	V	
tcp_ack	V	
tcp_recvmsg		V
udp_rcv	V	
udp_queue_rcv_skb	V	
udp_encap_rcv	V	
udp_recvmsg		V
icmp_rcv	V	
net_rx_action		V
tcp_rcv_state_process	V	
sk_wait_data		V

由於本技術擁有 Polymorphism of *sk_buff*、Direct use of *sk_buff* 及 Indirect use of *sk_buff* 的這三個概念，因此可以找到所有的封包處理函式，如 Figure 6-1 所示：*ip_queue_xmit* 屬於 D，而 *tcp_sendmsg* 的參數型態 *struct sock* 屬於 P，*tcp_sendmsg* 屬於 P & D。從以上實驗結果得知，KLASY 並未擁有本技術的三個概念，因此無法確實能夠找到所有的封包處理函式。

D：
```
int ip_queue_xmit(struct sk_buff *skb, int ipfragok)
```
P & D：
```
int tcp_sendmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg
        size_t size)
```

Figure 6-1　驗證選擇性指令嵌入技術

6.1.2 網路通訊協定堆疊剖析暨除錯工具之網路效能測試

本實驗的目的是測量本工具對於原始 Linux 核心系統的網路效能影響，以及本工具與 KFT[8]、曹敏峰[48]的效能比較。本實驗將使用一台 Linux 的 Netbook 用來模擬未來之嵌入式網路裝置，並且使用 Linux-2.6.21 版本作為測試的核心系統，Netbook 的規格表可見於 Table 6-4：

Table 6-4　Linux Netbook 規格表

Category	**Specification**
Model	Sony PCG-TR2E
CPU	Mobile Intel(R) Celeron(R) processor 800MHz
Memory	DDR SDRAM 512MB
Disk	40GB (Ultra ATA/100)
Ethernet Card	Intel 82801DB PRO/100 VE Ethernet
OS	CentOS 5.4 Final

我們利用 Iperf[65]進行 TCP 的效能測試，TCP 的測試方法為使用一台主機作為 Server，讓 Netbook 連線到 Server，進行 60 秒的傳輸，而環境配置圖，如 Figure 6-2，重覆 12 次的測試流程，去除最好與最差的傳輸速率(Transmission Rate)之後取其平均。

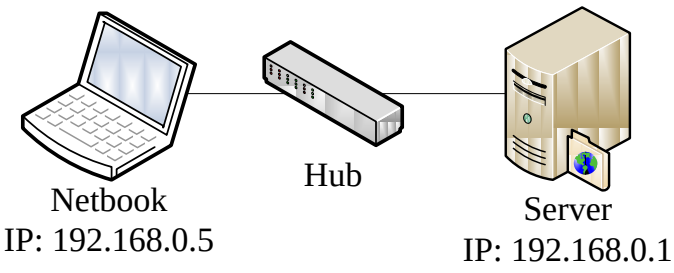

Figure 6-2　Iperf 測試之環境配置圖

本實驗之系統名稱介紹：

- 2.6.21-RAW：原始核心系統
- 2.6.21-SI：使用本工具的核心系統
- 2.6.21-KFT：使用 KFT 的核心系統
- 2.6.17-Tsao：使用曹敏峰之工具的核心系統

而且我們根據剖析暨除錯工具的運作行為，總共分為三個階段進行測試：

1. Memory Logging Disabled：關閉使用 Memory 記錄資訊的功能
2. Memory Logging Enabled and Disk Logging Disabled：開啟使用 Memory 記錄資訊，但是關閉使用 Disk 記錄資訊的功能
3. Disk Logging Enabled：開啟使用 Disk 記錄資訊的功能

以下我們將介紹 TCP 的測試結果，從 Figure 6-3 之中，我們可以得知，在 Memory Logging Disabled 的情況，2.6.21-SI、2.6.17-Tsao 與 2.6.21-RAW 的 TCP 傳輸速率是相同的，驗證儘管本工具嵌入 6813*2 = 13626 個 Probe 於核心系統中，依然不影響原始的核心系統運作行為。然而 2.6.21-KFT 的傳輸速率卻是大幅低落。

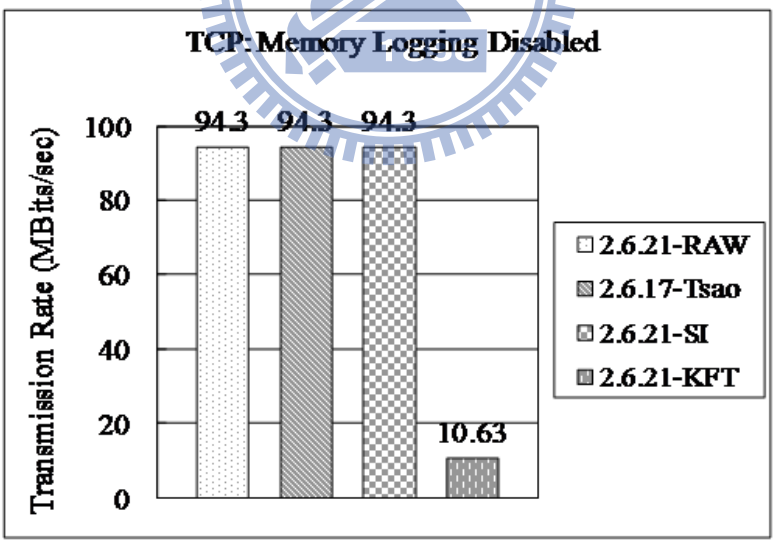

Figure 6-3　TCP 傳輸速率比較於 Memory Logging Disabled

接下來，我們可以從 Figure 6-4 得知，在 Memory Logging Enabled and Disk Logging Disabled 的情況之下，2.6.21-SI 的傳輸速率雖然降低 0.1Mbits/sec，但是與 2.6.21-RAW 及 2.6.17-Tsao 兩者傳輸速率的差別幾乎是微乎其微，因此可以證實本工具雖然於記錄資訊到 Memory 時，使用 Atomic Operation Mechanism，但是依然不影響網路傳輸的效率。至於 2.6.21-KFT 的傳輸速率依然低落。

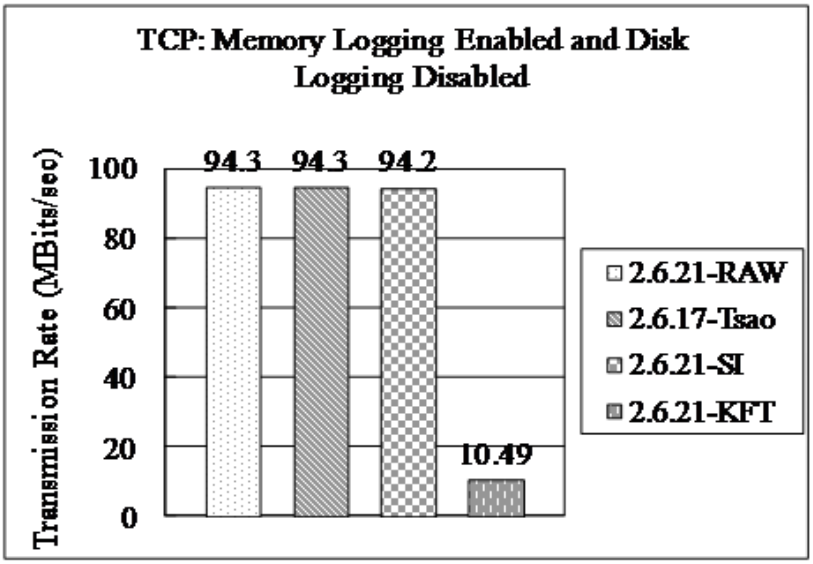

Figure 6-4　TCP 傳輸速率比較於 Memory Logging Enabled and Disk Logging Disabled

我們可以從 Figure 6-5 得知使用本工具造成 TCP 傳輸速率比曹敏峰的低 3..87Mbits/sec，因為本工具所嵌入指令比曹敏峰的多出許多，因此記錄的資料量也較大，也造成 Disk Logging 時產生較大的負擔。然而本工具的 TCP 傳輸速率與 KFT 相較之下，由於本工具只針對封包處理函式的特性，因此本工具對於核心系統負擔確實比 KFT 低許多。

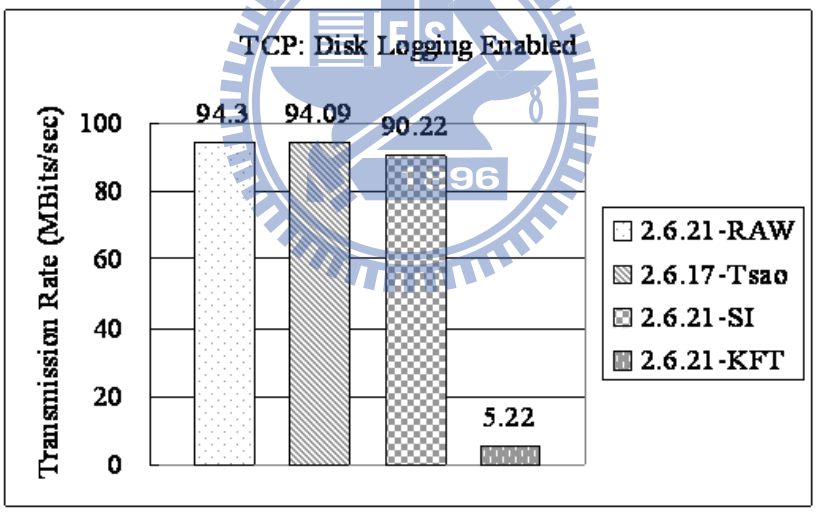

Figure 6-5　TCP 傳輸速率比較於 Disk Logging Enabled

最後，可以從 Figure 6-6 得知，在 Disk Logging Enabled 的情況，2.6.17-Tsao 的傳輸速率比未開啟 Disk 記錄功能時，稍微降低 0.21Mbits/sec，2.6.21-SI 的傳輸速率比未開啟 Disk 記錄功能時，降低 3.98Mbits/sec，然而 2.6.21-KFT 的傳輸速率更降低 5.27Mbits/sec。從以上的結果顯示在 TCP 的情況下，雖然本工具與曹敏峰比較之下，本工具的效能稍微下降，可是觀察到的函式數量更多。另一方面，與 KFT 相較之下，本工具對於核心系統的影響只有 4.33%，然而 KFT 對於核心系統的影響高達 94.47%，因此本工具確實可以準確地剖析暨除錯 Linux 網路協定堆疊。

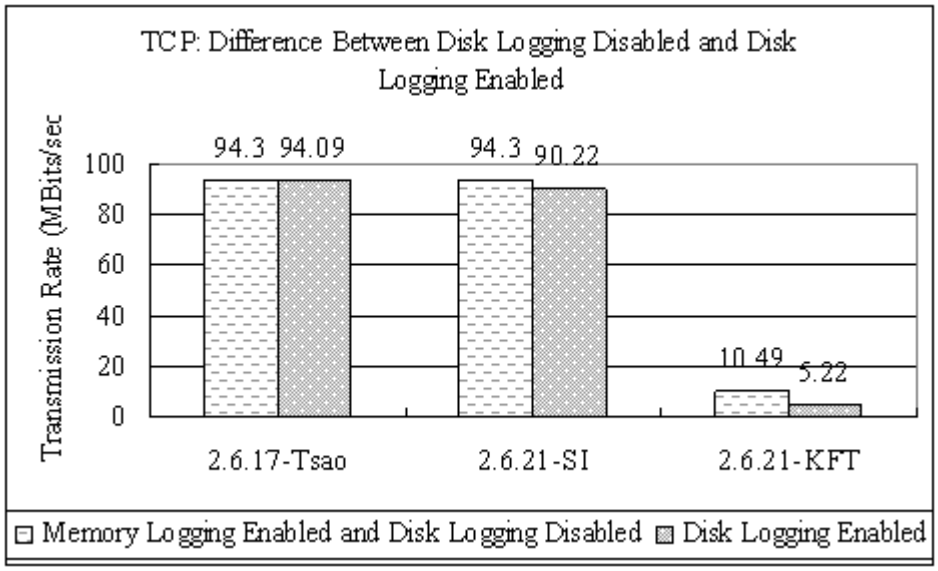

Figure 6-6　TCP 傳輸速率比較於 Memory Logging Enabled and Disk Logging Disabled 和 Disk Logging Enabled

6.1.3 核心行為記錄與網路協定記錄之整併成果

本實驗利用 TCP 的 Three-way-handshake 來驗證封包關聯技術的正確性，實驗環境配置如 Figure 6-7。Netbook 為 6.1.2 節所使用 Linux-2.6.21 作為測試的核心系統，讓 Netbook 與 TCP Server 進行 Three-way-handshake，以取得 Netbook 的 KBL，且使用 Monitor Device 執行 Wireshark，取得 TCP 之 Three-way-handshake 的 PBL。

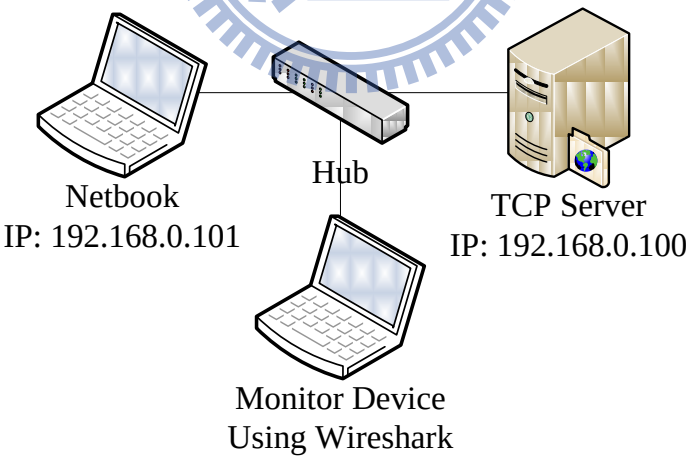

Figure 6-7　TCP 之 Three-way-handshake 測試的環境配置圖

首先，我們利用 Wireshark 從 PBL 取得此 Three-way-handshake 過程中的 SYN 封包、SYN, ACK 封包與 ACK 封包，如 Figure6-8。

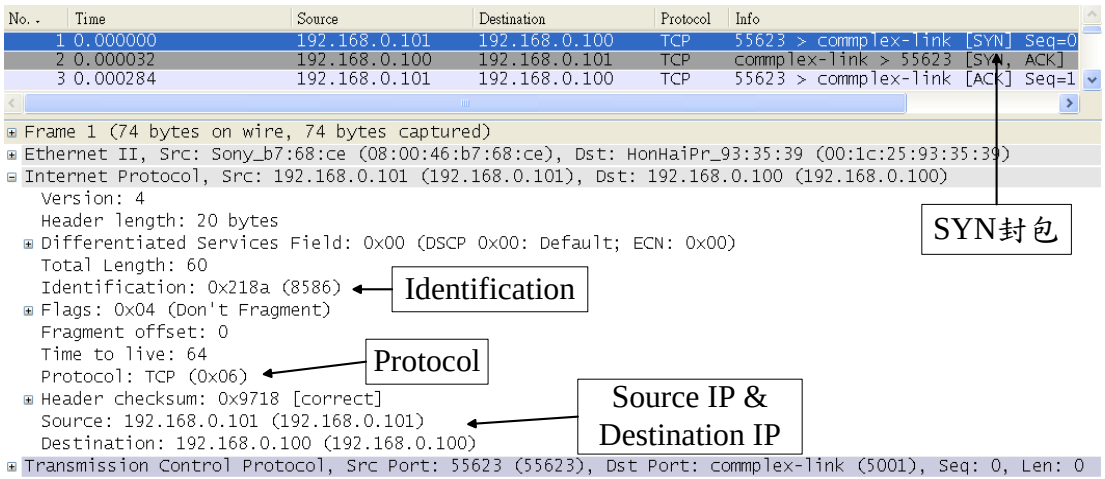

Figure 6-8　Three-way-handshake 之 SYN 封包的 PBL

此三個封包的 Destination IP、Source IP、Identification 及 Protocol 資訊如 Table 6-5。

Table 6-5　Three-way-handshake 之封包的相關資訊

	SYN	SYN, ACK	ACK
Source IP	192.168.0.101	192.168.0.100	192.168.0.101
Destination IP	192.168.0.100	192.168.0.101	192.168.0.100
Identification	8586	0	8587
Protocol	6	6	6

接著我們分別使用整併這三個封包所需的資訊輸入 Log Matcher (Kernel Viewer)與 KBL 作對應，即可得到如 Figure 6-9 的結果。

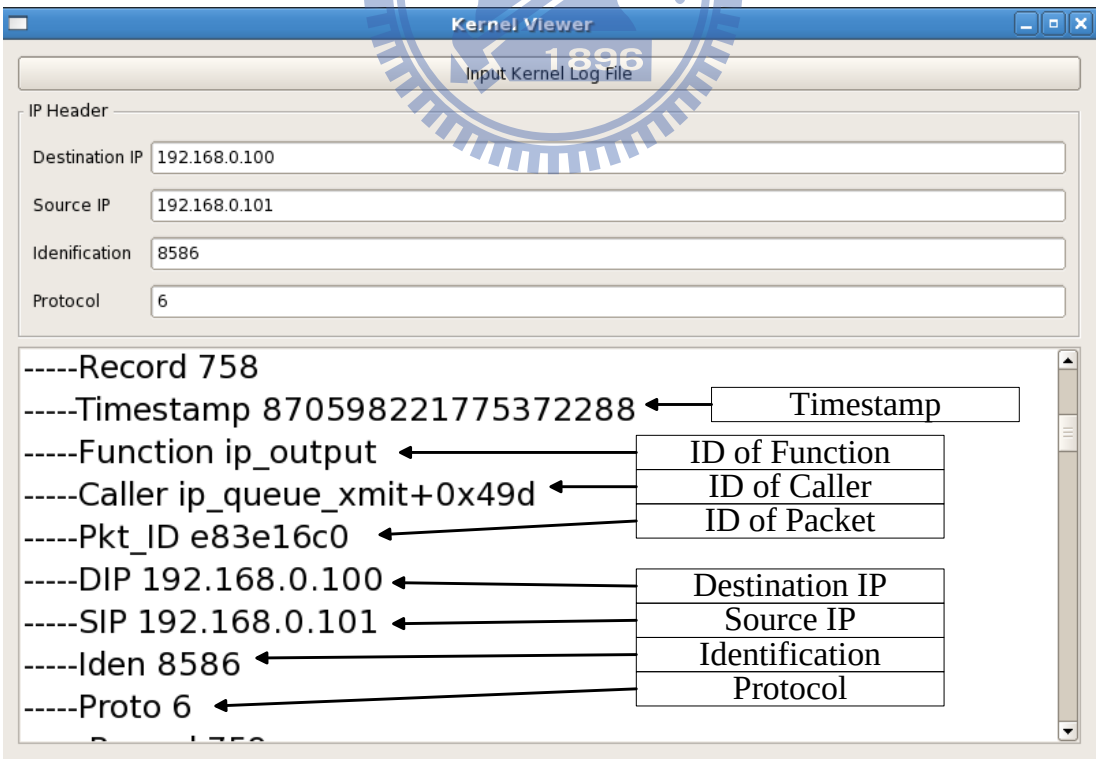

Figure 6-9　Three-way-handshake 之 SYN 封包的 KBL

從 Figure 6-9 之中，可以從 Timestamp 與 ID of Function 得知 SYN 封包經過 *ip_output* 的時間點，而從 ID of Caller 得知 *ip_output* 是被 *ip_queue_xmit* 呼叫，

並且從 ID of Packet 的欄位取得儲存 SYN 封包之 *struct sk_buff* 的記憶體位址：0xE83E16C0。因此 Log Matcher 將得以藉由此記憶體位址，以取得 SYN 封包於網路通訊協定堆疊內部的 Call Sequence，其他兩個封包也是利用相同的方式，找出該封包的 Call Sequence，如此一來我們即可得到這三個封包從建立到送出的時間長度，及所經過的所有封包處理函式，並且將 KBL 與 PBL 整併的結果整理如 Table 6 6。藉由比對網路封包擷取工具擷取到封包的時間(Arrival Time in PBL)，與本工具擷取到封包產生到送出的時間(Entry Time 和 Exit Time)，可以得知使用 Timestamp 來對應 KBL 與 PBL 是不準確的，兩者差距大約兩秒。而本技術利用 IP 的四個欄位確實可以將 KBL 與 PBL 對應，取得一個封包在核心內部經過的所有函式數量(Total Functions)與處理時間(Cost Time)。

Table 6-6　Three-way-handshake 之 KBL 與 PBL 整併結果

	SYN	SYN, ACK	ACK
Arrival Time in PBL	00:11:27.493006s	00:11:27.493050s	00:11:27:493650s
Entry Time	00:11:25.546465s	00:11:25:548464s	00:11:25.548585s
Exit Time	00:11:25:546682s	00:11:25.548577s	00:11:25.548765s
Total Functions	38	22	49
Cost Time	0.000217s	0.000113s	0.000180s

6.2 貢獻

本篇論文針對核心系統的網路通訊堆疊提出一個新的網路通訊協定堆疊剖析暨除錯工具，本工具包含一種新的追蹤網路封包處理程序的機制，利用此機制可以準確地，從核心系統的原始程式碼中，找出存取封包的函式，並且嵌入指令於該函式。

另一方面，本工具還包含一個封包關聯的技術，讓嵌入的指令可以在不影響原始核心系統的情況之下，記錄執行函式與封包相關資訊。並且利用這些核心行為記錄，與網路封包擷取工具的網路協定行為記錄作對應，以取得網路協定行為與核心行為之間的互動與整合效果，讓網路通訊軟體開發者可以很容易分析網路通訊的核心系統行為，釐清與改善網路通訊的問題，藉此全面剖析與除錯網路通訊協定堆疊。

第七章　結論與未來工作

7.1 結論

隨著嵌入式系統(Embedded System)技術的進步與網際網路已大量建置於生活周遭，嵌入式網路通訊裝置已經逐漸普遍於市面上。因此如何有效率地分析通訊裝置上的通訊行為是一個很值得深思的問題，然而目前對於通訊裝置的相關效能評估工具仍不多。使得我們必要開發一套合適的輔助工具，協助嵌入式網路通訊裝置的開發者剖析暨除錯網路通訊協定堆疊，來正確界定造成傳輸延遲與封包遺漏的原因。

本論文提出的選擇性指令嵌入技術是希望可以讓開發者，即便不熟悉核心系統的網路通訊協定堆疊，也可以輕易地找出封包處理函式，並且嵌入指令於封包處理函式。而且開發者可以經由相關的網路協定測試，追蹤系統內部處理網路封包的行為與時間點，讓開發者對於網路通訊協定堆疊有更加深入地了解。

另一方面，本論文提出的封包關聯技術是設計被嵌入的指令必須記錄的最少封包相關資料，以及它經過處理函式的順序與時間點。之後本技術將核心行為記錄與網路封包行為記錄作對應，可以清楚地標示出網路封包進出核心系統與其在外的網路協定行為。藉此取得核心行為與網路協定行為之間的互動與整合效果，讓使用者得以剖析暨除錯核心系統內部的網路通訊協定堆疊。

從本論文的實驗中，可以驗證選擇性指令嵌入技術與封包關聯技術的實用性及準確性，而且隨著嵌入式網路通訊裝置的性能不斷地提升以及核心系統越來越龐大，本論文對於尋找系統效能瓶頸和系統程式錯誤是非常有幫助的。

7.2 未來工作

我們的目標是希望可以發展一套整合核心行為與網路協定行為的網路通訊裝置之評比工具，而本論文已經成功將嵌入指令於核心系統的動作自動化，並且成功將核心行為記錄與網路協定行為記錄作對應。

可是核心行為記錄與網路協定行為記錄過於龐大，因此我們還需要一種智慧型的診斷工具，從龐大的記錄之中，找出影響網路效能的原因，進而針對原因提出建議等等。

未來診斷工具更可以配合核心事件的記錄，找出因為某些特定網路封包，造成核心系統的狀態改變，如：ARP 封包會造成 ARP Table 的改變，DHCP 封包會造成網路卡介面的 IP 改變，藉此開發者可以更加詳細剖析暨除錯網路通訊裝置的整體通訊行為。

更可以修改本論文的選擇性指令嵌入技術，使之運用於各種核心系統，並且持續改進以配合核心系統的演進。另一方面，由於本論文只針對乙太網路設計封包關聯技術，並未針對其他類型的實體網路，因此未來可以修改封包關聯技術來達成剖析暨除錯其他類型的實體網路。

Reference

[1] S. Best, “Linux® Debugging and Performance Tuning: Tips and Techniques,” Prentice Hall, Oct. 2005.
[2] 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.
[3] J. Levon, “Profiling in Linux HOWTO,” The Linux Documentation Project, 2002.
[4] N. Joukov, A. Traeger, R. Iyer, C. P. Wright, and E. Zadok, “Operating System Profiling via Latency Analysis,” 7th symposium on Operating systems design and implementation, pp. 89-102, 2006.
[5] Y. Guo, Z. Chen, and X. Chen, “A Lightweight Dynamic Performance Monitoring Framework for embedded Systems,” International Conference on Embedded Software and Systems, pp. 256-262, 2009.
[6] A. Bhatele and G. Cong, “A Selective Profiling Tool: Towards Automatic Performance Tuning,” Parallel and Distributed Processing Symposium, pp. 1-6, March 2007.
[7] Valgrind, Available from: http://valgrind.org/
[8] N. Mc Guire, “Kernel Function Instrumentation – KFT,” 2006.
[9] P. P. Bungale and C. K. Luk, “PinOS: A Programmable Framework for Whole-System Dynamic Instrumentation,” 3rd international conference on Virtual execution environments, pp. 137-147, 2007.
[10] GDB: The GNU Project Debugger, Available from: http://www.gnu.org/software/gdb/gdb.html
[11] L. DeRose and T. Hoover Jr., “The Dynamic Probe Class Library - An Infrastructure for Developing Instrumentation for Performance Tools,” 15th International Parallel & Distributed Processing Symposium, vol. 1, 2001.
[12] K. Kwon, M. Sugaya, and T. Nakajima, “Analysis of Embedded Linux using Kernel Analysis System,” International Conference on Embedded Software and Systems, pp. 417-422, 2009.
[13] M. Desnoyers and M. R. Dagenais, “Low disturbance embedded system tracing with Linux Trace Toolkit Next Generation,” Embedded Linux Conference, 2006.
[14] M. Desnoyers and M. R. Dagenais, “LTTng, Filling the Gap Between Kernel Instrumentation and a Widely Usable Kernel Tracer,” Linux Foundation Collaboration Summit, Apr. 2009.
[15] K. Yaghmour and M. R. Dagenais, “Measuring and characterizing system behavior using kernel-level event logging,” USENIX Annual Technical Conference, 2000.

[16] M. Desnoyers and M. R. Dagenais, “The LTTng tracer A low impact performance and behavior monitor for GNU Linux,” Ottawa Linux Symposium (OLS), 2006.
[17] P. Heidari, M. Desnoyers, and M. Dagenais, “Performance Analysis of Virtual Machines Through Tracing,” Canadian Conference on Electrical and Computer Engineering, pp. 261-266, 2008.
[18] LTTng Project, Available from: http://ltt.polymtl.ca/
[19] Kernprof (Kernel Profiling), Available from: http://oss.sgi.com/projects/kernprof/
[20] KFT, Available from: http://elinux.org/Kernel_Function_Trace
[21] T. Bird, “Learning the kernel and finding performance problems with kfi,” CELF International Technical Conference, 2005.
[22] 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.
[23] KTAU, Available from: http://www.cs.uoregon.edu/research/ktau/docs.php
[24] LKST, Available from: http://lkst.sourceforge.net/
[25] A. Mavinakayanahalli, P. Panchamukhi, J. Keniston, A. Keshavamurthy, and M. Hiramatsu, “Probing the Guts of Kprobes,” Ottawa Linux Symposium (OLS), 2006.
[26] R. J. Moore, ”Dynamic probes and generalized kernel hooks interface for Linux,” 4th annual Linux showcase and conference, pp. 139-145, 2000.
[27] R. Krishnakumar, “Kernel Korner: Kprobes - a Kernel Debugger,” Linux Journal, vol. 2005, Issue 133, Jun. 2006.
[28] B. Lee, S. Moon, and Y. Lee, “Application-Specific Packet Capturing using Kernel Probes,” 11th IFIP/IEEE international conference on Symposium on Integrated Network Management, pp. 303-306, 2009.
[29] SystemTap, Available from: http://sourceware.org/systemtap/
[30] V. Prasad, J. Keniston, W. Cohen, F. C. Eigler, M. Hunt, and B. Chen, “Locating System Problems Using Dynamic Instrumentation,” Ottawa Linux Symposium (OLS), July 2005.
[31] M. Mason, “Using SystemTap for Dynamic Tracing and Performance Analysis,” Ottawa Linux Symposium (OLS), 2007.
[32] F. C. Eigler, “Problem Solving With Systemtap,” Red Hat Summit, 2007.
[33] A. Tamches and B. P. Miller, “Fine-grained dynamic instrumentation of commodity operating system kernels,” 3rd symposium on Operating systems design and implementation, pp.117-130, 1999.
[34] KernInst - Dynamic Kernel Instrumentation Tool Suite, Available from: http://www.paradyn.org/html/kerninst.html
[35] A.Tamches and B. P. Miller, “Using Dynamic Kernel Instrumentation for Kernel

and Application Tuning," International Journal of High Performance Computing Applications, vol. 13, issue 3, pp. 263-276, 1999.
[36] KLASY - Kernel Level Aspect-oriented System, Available from: http://www.csg.is.titech.ac.jp/~yanagisawa/KLASY/
[37] Y. Yoshisato, K. Kenichi, and C. Shigeru, "A dynamic aspect-oriented system for OS kernels," 5th international Conference on Generative Programming and Component Engineering, pp. 69-78, 2006.
[38] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic Instrumentation of Production Systems," USENIX Annual Technical Conference, pp. 15-28, 2004.
[39] OProfile - A System Profiler for Linux (News), Available from: http://oprofile.sourceforge.net/news/
[40] J. Levon, "OProfile Internals," Available from: http://oprofile.sourceforge.net/doc/internals/index.html
[41] W. E. Cohen, "Tuning programs with OProfile," Wide Open Magazine, 2004
[42] Wireshark, Available from: http://www.wireshark.org/
[43] Kismet Wireless Network Sniffer, Available from: http://www.kismetwireless.net/
[44] F. Fuentes and D. C. Kar, "Ethereal vs. Tcpdump: a comparative study on packet sniffing tools for educational purpose," Journal of Computing Sciences in Colleges, vol. 20, pp. 169-176, April 2005.
[45] 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, April 2004.
[46] W. Wu and M. Crawford, "The performance analysis of linux networking - Packet receiving," Computer Communications, vol. 30, issue 5, pp. 1044-1057, March 2007.
[47] B. Henderson, "Linux Loadable Kernel Module HOWTO," Available from: http://www.tldp.org/HOWTO/Module-HOWTO/
[48] M. F. Tsao, "Design and Implementation of an Efficient and Configurable Instrument Platform for Linux Network Protocol Stack," WIN Lab NCTU, 2008.
[49] R. Love, "Linux Kernel Development, 1st edition," SAMS Publishing, 2003.
[50] J. Corbet, Alessandro Rubini, and Greg Kroah-Hartman, "Linux Device Drivers, 3rd Edition," O'Reilly, Feb. 2005.
[51] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov, "Linux Netlink as an IP Services Protocol," IETF RFC 3549, July 2003.
[52] T. Zanussi, K. Yaghmour, R. Wisniewski, R. Moore, and M. Dagenais, "relayfs: An Efficient Unified Approach for Transmitting Data from Kernel to User Space," Ottawa Linux Symposium (OLS), July 2003.

[53] relayfs, Available from: http://relayfs.sourceforge.net/
[54] C. Guo and S. Zheng, “Analysis and Evaluation of the TCP/IP Protocol Stack of Linux,” International Conference on Communication Technology Proceedings, vol. 1, pp. 444-453, Aug. 2000.
[55] S. Seth, “TCP/IP Architecture, Design, and Implementation in Linux,” Wiley-IEEE Computer Society Press, Dec. 2008.
[56] C. Benvenuti, “Understanding Linux Network Internals,” O’Reilly, Dec. 2005.
[57] R. Lehmann and A. Schill, “Linux TCP network stack analysis and partitioning for network processors,” 4th international symposium on Information and communication technologies, pp. 69-74, 2005.
[58] W. Wu and M. Crawford, “Potential performance bottleneck in Linux TCP,” International Journal of Communication Systems, vol. 20, issue 11, pp. 1263-1283, Nov. 2007.
[59] S. W. Hsu, “Design and Implementation of a Networking Behavior Analysis Tool for Linux Operating System,” WIN Lab NCTU, 2009.
[60] M. D. Rey, “Internet Protocol,” IETF RFC 791, Sept. 1981.
[61] J. R. Levine, T. Mason, and D. Brown, “Lex & Yacc, 2nd Edition,” O’Reilly, Oct. 1992.
[62] ANSI C grammar (Lex specification), Available from: http://www.lysator.liu.se/c/ANSI-C-grammar-l.html
[63] ANSI C grammar (Yacc specification), Available from: http://www.lysator.liu.se/c/ANSI-C-grammar-y.html
[64] The C Preprocessor, Available from: http://gcc.gnu.org/onlinedocs/cpp/
[65] Iperf, Available from: http://sourceforge.net/projects/iperf/