실시간 다중 프로세스 트레이스 스트림 디코더 구현에 관한 연구★

김 현 철*, 김 영 수**, 김 종 현**

요 약

소프트웨어 공학 관점에서 트레이싱은 프로그램의 실행 정보를 기록하는 로깅의 특수한 형태이다. 엄청난 데이터를 실시간으로 생성하고 디코딩해야 하는 트레이서의 특징상 전용 하드웨어를 사용하는 트레이서가 많이 사용되고 있다. Intel® PT는 전용 하드웨어를 사용하여 각 하드웨어 쓰레드에서 소프트웨어 실행에 대한 모든 정보를 기록한다. 소프트웨어 실행이 완료되면 PT는 해당 소프트웨어의 트레이스 데이터를 처리하여 정확한 프로그램 흐름을 재구성할 수 있다. 하드웨어 트레이스 프로그램은 운영체제에 통합되어 사용할 수 있으나 윈도우 시스템의 경우에는 커널 개방과 같은 문제로 인하여 긴밀한 통합은 이루어지지 않고 있다. 또한, 단일 프로세스만 트레이스 할 수 있고 다중 프로세스 스트림을 트레이스 하는 방법은 제공하고 있지 않다. 본 논문에서는 이러한 단점을 극복하고자 윈도우 환경에서 다중 프로세스 스트림을 트레이스 지원이 가능하도록 기존의 PT 트레이스 프로그램을 확장하는 방안을 제안하였다.

A Study on Implementation of Real-Time Multiprocess Trace Stream Decoder

Hyuncheol Kim*, Youngsoo Kim**, Jonghyun Kim**

ABSTRACT

From a software engineering point of view, tracing is a special form of logging that records program execution informa tion. Tracers using dedicated hardware are often used because of the characteristics of tracers that need to generate and d ecode huge amounts of data in real time. Intel® PT uses proprietary hardware to record all information about software ex ecution on each hardware thread. When the software execution is completed, the PT can process the trace data of the soft ware and reconstruct the correct program flow. The hardware trace program can be integrated into the operating system, but in the case of the window system, the integration is not tight due to problems such as the kernel opening. Also, it is possible to trace only a single process and not provide a way to trace multiple process streams. In this paper, we propose a method to extend existing PT trace program to support multi - process stream traceability in Windows environment in order to overcome these disadvantages.

Key words : Multistream Decoder, Processor Trace, Software Tracing

접수일(2018년 11월 30일), 수정일(1차: 2018년 12월 17일), 게재확정일(2018년 12월 30일)

★본 논문은 2017년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임(No. 2017R1D1A3B03035922)

★본 논문은 2018년도 정부(미래창조과학부)의 재원으로 정보통신기술진흥센터의 지원을 받아 수행된 연구임(No.2016-0-00078, 맞춤형 보안서비스 제공을 위한 클라우드 기반 지능형 보안 기술 개발)

* 남서울대학교 컴퓨터소프트웨어학과 교수(주저자)

** 한국전자통신연구원(ETRI)

1. 서 론

소프트웨어 공학 관점에서 트레이싱은 프로그램의 실행 정보를 기록하는 로깅의 특수한 형태이다. 따라서 트레이싱은 일반적으로 “a technique used to understand what is going on in a system in order to debug or monitor it”으로 정의할 수 있다. 수집된 정보는 일반적으로 디버깅 용도로 사용하며, 트레이스 로그에 포함된 정보의 유형 및 세부 정보에 따라 다양한 소프트웨어 문제를 진단하는 용도로 사용될 수 있다 [1]. 트레이싱 정보는 개발 사이클과 소프트웨어 릴리스 이후에 모두 사용된다. 이벤트 로깅과 달리 이벤트의 "클래스" 또는 "이벤트 코드" 개념이 없다.

엄청난 데이터를 실시간으로 생성하고 디코딩해야 하는 트레이서의 특징상 전용 하드웨어를 사용하는 트레이서가 많이 사용되고 있으며, ARM Coresight와 Intel® PT(Processor Trace)가 하드웨어 기반의 대표적인 트레이서이다 [2].

<표 1> 로깅과 트레이싱

이벤트 로깅	트레이싱
시스템 관리자가 주로 사용	개발자가 주로 사용
프로그램 설치 오류와 같은 상위 레벨의 정보를 기록	Thrown exception과 같은 하위 레벨의 정보를 기록
명확하고 간략해야 함	중복되는 이벤트나 정보가 기록될 수 있음
표준으로 정한 형태의 결과를 요구함	Output 포맷에 제한이 없음
이벤트 로그 메시지는 종종 localized 됨	Localization과는 상관없음
신속한 이벤트/메시지를 추가가 필요 없음	트레이싱 메시지 추가가 간단해야 함

Intel® PT는 전용 하드웨어를 사용하여 각 하드웨어 쓰레드에서 소프트웨어 실행에 대한 모든 정보를 기록한다. 소프트웨어 실행이 완료되면 PT는 해당 소프트웨어의 트레이스 데이터를 처리하여 정확한 프로그램 흐름을 재구성할 수 있다 [3][4][5].

리눅스 시스템의 경우 PT를 기반으로 하는 하드웨어 트레이스 프로그램이 운영체제에 통합되어 perf와 같은 명령을 통하여 사용할 수 있다. 그러나 윈도우 시스템의 경우에는 커널 개방과 같은 문제로 인하여 프로파일링 및 디버깅 메커니즘과의 긴밀한 통합은 이루어지지 않고 있다. 이를 위해 몇몇 개인이나 단체들이 윈도우 환경에서 PT를 구현하고 있다. 그러나 perf나 윈도우 환경 모두 PT를 이용하여 단일 프로세스만 트레이스 할 수 있고 다중 프로세스 스트림을 트레이스 하는 방법은 제공하고 있지 않다 [6][7].

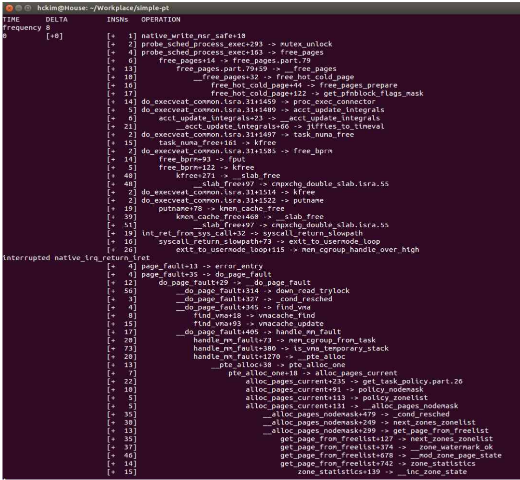
본 논문에서는 이러한 단점을 극복하고자 윈도우 환경에서 다중 프로세스 스트림을 트레이스 지원이 가능하도록 기존의 PT 트레이스 프로그램을 확장하는 방안을 제안하였다.

본 논문의 구성은 다음과 같다. 2장에서는 PT 및 관련 소프트웨어와 연관된 다양한 선행 기술들의 조사 및 분석을 수행하였다. 3장에서는 윈도우 환경에서 다중 프로세스 스트림을 트레이스 지원하는 디코더 구현에 관해서 기술하였다. 마지막으로 4장에서는 논문의 결론을 기술하였다.

2. 관련 연구

2.1 PT 개요

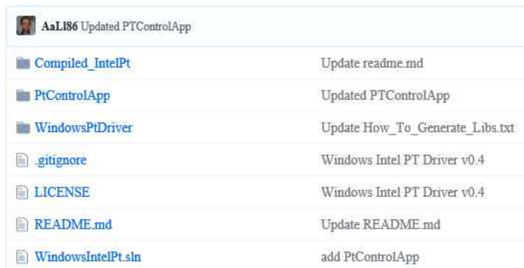
소프트웨어 트레이싱을 위해 자주 사용되는 후크(Hook) 방식과 달리, PT는 전용 하드웨어를 사용하기 때문에 시스템에 최소한의 영향 (5% 미만의 오버헤드)만 주면서 모든 소프트웨어 실행 정보를 저장할 수 있다. 또한, (그림 1)에서 나타내고 있는 바와 같이 이러한 트레이스 로그로부터 소프트웨어 실행 흐름을 복원할 수 있다. 트레이스 로그에는 프로그램 흐름 정보 (예: branch targets, branch taken/not taken indications) 및 프로그램 유발 모드 관련 정보 (예: Intel® TSX state transitions)가 포함된다. 디버거는 특정 위치로 연결되는 코드 흐름을 재구성하는 데 트레이스 로그 정보를 사용할 수 있으며 CALL 및 RET 명령어를 기반으로 스택 백 트레이스를 재구성하는 데 사용할 수도 있다 [8].



(그림 1) PT 디코드 예

현재 몇몇 단체나 개인이 윈도우 환경에서 PT 드라이버와 애플리케이션을 구현하고 있고 대표적으로 WindowsIntelPT[6] 프로그램이 있다. WindowsIntelPT 드라이버는 윈도우 환경에서 Skylake 아키텍처의 Intel 프로세서 트레이스 기능을 구현하고 있다. 즉 Skylake 아키텍처를 지원하는 CPU에서 하드웨어 기반 고성능 분기 트레이스 메커니즘을 구현하고 있다 [6].

WindowsIntelPT는 물리적 메모리에 직접 트레이스 로그를 기록하여 캐시 및 TLB 폴링을 방지하며 장시간 트레이스에 적합한 압축된 로깅 형식을 사용한다. 또한, 사용자 공간 및 커널을 포함하여 CPU 코어의 모든 분기를 추적할 수 있다.

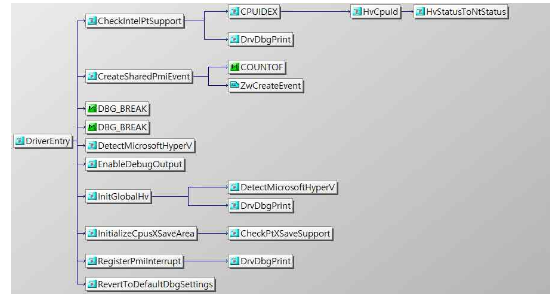


(그림 2) WindowsIntelPT S/W 구성

3. 실시간 다중 프로세서 트레이스 스트림 디코더 구현

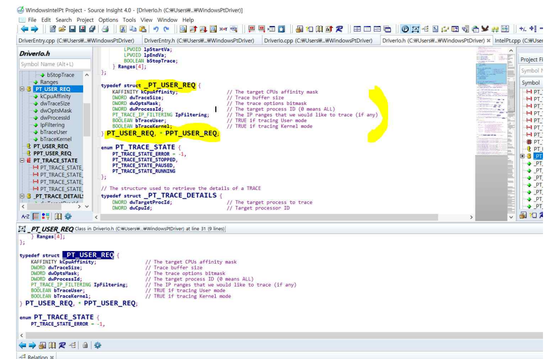
3.1 WindowsPtDrvier와 PTControlApp 분석

드라이버 개발 시 entry function은 DriverEntry()이며, 이 함수는 driver file이 메모리에 로드되는 시점에 system thread(I/O manager)에 의해서 호출되는 함수이다.



(그림 3) WindowsPTDriver 엔트리 포인트

DriverEntry 함수는 드라이버가 메모리에 로드되었을 때, 운영체제가 DriverEntry라는 이름을 갖는 함수를 먼저 찾기 때문에 무조건 이름이 DriverEntry여야 한다. 매개변수로는 PDRIVER_OBJECT->DriverObject 구조체와 PUNICODE_STRING->RegistryPath 구조체가 필요하며, PDRIVER_OBJECT->DriverObject 구조체는 드라이버를 나타내는 구조체이고, PUNICODE_STRING ->RegistryPath 구조체는 드라이버가 설치되었을 때 레지스트리의 \Registry\Machine\System\CurrentControlSet\Services\DriverName에 저장된 키값을 나타낸다.



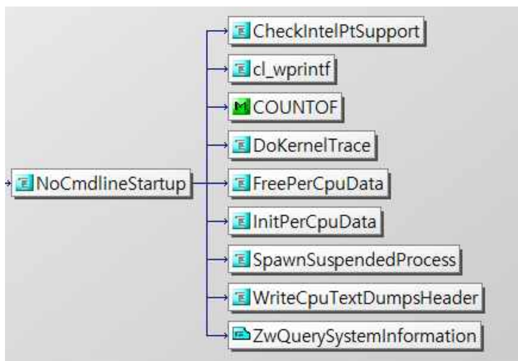
(그림 4) PT_USER_REQ 구조

WindowsIntelPT DriverEntry()에서는 다음과 같은 기능을 차례대로 제공한다.

① 프로세서 숫자 파악
② 드라이버를 위한 전체 데이터 공간 할당
③ 가상화 HyperV 지원 여부 체크
④ PT 지원 여부 체크
⑤ PMI(Performance Monitoring Interrupt) 이벤트를 만들고 인터럽트를 등록
⑥ 유저 모드 콜백함수 리스트 초기화
⑦ 유니코드 변환 초기화
⑧ IOCTL 생성 및 제어
⑨ DriverObject 생성 및 제어
⑩ DeviceIoControl 생성 및 제어
⑪ PT_USER_REQ 생성 및 제어
⑫ PT Start/Stop

3.2 다중 프로세서 트레이서 확장

본 논문에서는 (그림 5)에서와 같이 다중 프로세서 분석을 위해 NoCmdlineStartup 함수를 분석하였고 확장 방안을 연구하였다. 이는 Command line argument 없이 트레이스를 하는 함수이다. 다중 프로세스 분석을 위해서는 여러 디바이스 컨트롤을 실행해야 하고 이를 위해 PT 디바이스 핸들값 변수, 타겟 프로세스 경로 등, 여러 변수를 추가적으로 배열 선언해야 할 필요가 있다.



(그림 5) NoCmdlineStartup 함수 구조

우선 (그림 6)에서와 같이 덤프 파일을 위한 공간생성 및 초기화해야 한다. 즉 여러 프로세스의 덤프 파일 공간생성 및 초기화 코드로 바꿔야 한다. 이후 애플리케이션 정보를 가지는 전역변수의 PT 디바이스 핸들값 초기화한다.

```
// Allocate memory for the file names
lpOutBasePath = new TCHAR[MAX_PATH];
RtlZeroMemory(lpOutBasePath, MAX_PATH * sizeof(TCHAR));
```

(그림 6) 덤프파일 공간생성

(그림 7)은 여러 개의 프로세스에 대해 트레이싱을 진행할 때, 프로세스에 실행할 프로세서들의 수를 결정하게 되는 부분으로, 프로세스의 수와 이용 가능한 프로세서들 사이에서 스케줄링이 추가되어야 한다. 또한, 커널 분석을 진행하는 프로세스가 있을 때는 우선순위까지 고려하여야 한다.

```
if (sysInfo.dwNumberOfProcessors > 1) {
    // Ask how many processor to use
    wprintf(L"On how many processors would you like to run the process? [1/%i] ",
    wscanf_s(L"%i", &dwCpusCount);

    if (dwCpusCount > sysInfo.dwNumberOfProcessors) {
        wprintf(L"Invalid value, assuming all the processors as valid.\r\n");
        cpuAffinity = sysInfo.dwActiveProcessorMask;
        dwCpusCount = sysInfo.dwNumberOfProcessors;
    } else
        cpuAffinity = ((DWORD_PTR)(-1i64) >> ((sizeof(DWORD_PTR) * 8) - dwCpus
    if (FALSE)
        // If you would like to test the different affinities:
        cpuAffinity = 0xd;
    _ASSERT((sysInfo.dwActiveProcessorMask | cpuAffinity) == sysInfo.dwActiveProc
```

(그림 7) 프로세서 간 스케줄링 부분

```
// Create the PMI threads (1 per target CPU)
for (int i = 0; i < (int)dwCpusCount; i++) {
    PT_PMI_USER_CALLBACK pmiDesc = { 0 };
    HANDLE hNewThr = NULL;
    DWORD newThrId = 0;

    hNewThr = CreateThread(NULL, 0, PmiThreadProc, (LPVOID)i, CREATE
    // Register this thread and its callback
    pmiDesc.dwThrId = newThrId;
    pmiDesc.kCpuAffinity = (1i64 << i);
    pmiDesc.lpAddress = PmiCallback;
    bRetVal = DeviceIoControl(hPtDev, IOCTL_PTDRV_REGISTER_PMI_ROUTI
    if (bRetVal) {
        pCpuDescArray[i].dwPmiThrId = newThrId;
        pCpuDescArray[i].hPmiThread = hNewThr;
        ResumeThread(hNewThr);
    }
}
#pragma endregion
```

(그림 8) 성능측정 쓰레드 생성 및 등록

(그림 8)은 성능측정 스레드의 생성 및 등록을 담당하는 부분으로 여러 분석이 동시 진행될 경우, 다

중 쓰레드 생성이 필요하다. (그림 9)는 커널 테스트 모드를 나타내고 있으며 여러 트레이스가 동시 진행될 경우, 여러 주소를 찾도록 수정되어야 한다.

```
else {
    // Grab the target module base address
    SYSTEM_ALL_MODULES * pSysAllModules = NULL;
    NTSTATUS ntStatus = 0;
    CHAR modNameAnsi[0x80] = { 0 };
    sprintf_s(modNameAnsi, COUNTOF(modNameAnsi), "%S", procPath);
    ntStatus = ZwQuerySystemInformation(11, pSysAllModules, 0, &dwBytesIo);
    if (ntStatus == STATUS_INFO_LENGTH_MISMATCH) {
        pSysAllModules = (SYSTEM_ALL_MODULES*)VirtualAlloc(NULL, dwBytesIo + 64
        RtlZeroMemory(pSysAllModules, dwBytesIo);

        ntStatus = ZwQuerySystemInformation(11, pSysAllModules, dwBytesIo, &dwB
        if (ntStatus == 0) {
            // Search for the SimplePt
            for (unsigned i = 0; i < pSysAllModules->dwNumOfModules; i++) {
                SYSTEM_MODULE_INFORMATION curMod = pSysAllModules->modules[i];
                LPSTR lpTargetModName = curMod.ImageName + curMod.ModuleNameOff
                if (_stricmp(modNameAnsi, lpTargetModName) == 0) {
                    // Target module found
                    wprintf(L"Found \"%S\" kernel driver in memory.\r\n", lpTar
                    remoteModInfo.lpBaseOfDll = curMod.Base;
                    remoteModInfo.SizeOfImage = curMod.Size;
                    break;
                }
            }
        }
    } « end if ntStatus==STATUS_INFO... »
    if (pSysAllModules) VirtualFree((LPVOID)pSysAllModules, 0, MEM_RELEASE);
    ptStartStruct.bTraceKernel = TRUE;
    ptStartStruct.bTraceUser = FALSE;
} « end else »
```

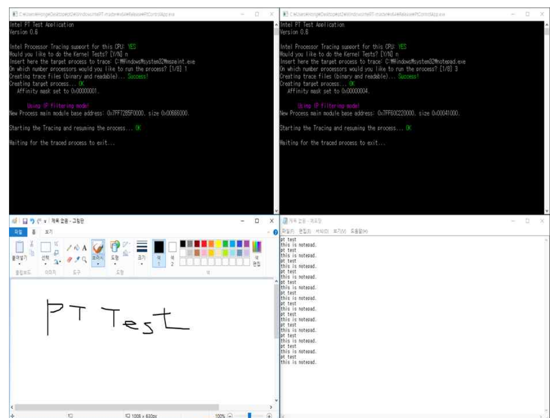
(그림 9) 커널 테스트 모드

```
#include <TlHelp32.h>
#include <tchar.h>
#include <psapi.h>
#include <process.h>

PROCESS_LIST pList = { {0},0 };
TCHAR ProcessPath[255];
TCHAR ProcessName[255];

// CreateToolhelp32Snapshot(), Process32First(), Process32Next()를 사용하여
// 현재 실행 중인 프로세서 목록을 확인한다.
// Tool Help를 사용하여 프로세스 목록, PID를 얻어 온다.

DWORD WINAPI SearchProcessListThread(LPVOID lpParam) {
    HANDLE hProcess = NULL;
    //PROCESS_LIST * processlist = ((PROCESS_LIST *)lpParam);
    PROCESSENTRY32 pe32 = { 0 };
    PROCESS_INFORMATION pi = { 0 };
    int index = 0;
    int cnt = 0;
    char buf[260] = { NULL };
    TCHAR szImagePath[MAX_PATH] = { 0, };

    /* ... */

    ZeroMemory(szImagePath, sizeof(szImagePath));

    printf("[%25s₩t%5s]₩n", "System Process", "PID");
    pe32.dwSize = sizeof(PROCESSENTRY32);

    while (1) {
        hProcess = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
        if (Process32First(hProcess, &pe32)) {
            while (Process32Next(hProcess, &pe32)) {
                cnt = 0;
                if (wcsstr(pe32.szExeFile, DEFAULT_PROCESS_NAME)) {
                    for (int j = 0; j < pList.numOfProcess; j++) {
                        if (pList.dwProcessIdList[j] == pe32.th32ProcessID) {
                            cnt = 1;
                            break;
```

(그림 10) 다중 스트림 트레이스 테스트 코드

(그림 10)은 확장된 WindowsIntelPT 드라이버를 이용하여 다중 스트림 트레이스를 테스트하는 코드 일부분이다. 테스트 프로그램에서는 createProcess 함수를 호출하면서, processPath에 ptcontrolapp.exe 경로를 넣어 전달한다. 또한, 새로운 프로세스를 감지하여 이를 전달하며, 현재 실행 중인 프로세스 리스트를 가져와 원하는 프로세스를 탐지하는 기능을 수행할 함수 또한 제공한다.

(그림 11)은 그림판과 메모장, 2개의 애플리케이션을 트레이스 하는 것을 나타내고 있고, (그림 12)는 2개의 애플리케이션을 동시에 트레이스한 결과를 덤프한 내용을 나타내고 있다.



(그림 11) 다중 스트림 트레이스 테스트

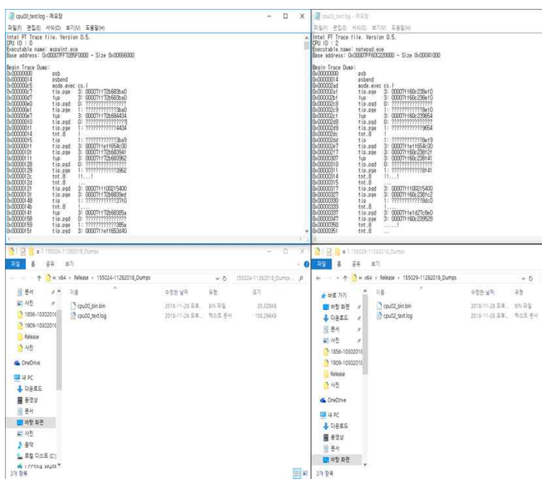
4. 결 론

Intel® PT는 전용 하드웨어를 사용하여 각 하드웨어 쓰레드에서 소프트웨어 실행에 대한 모든 정보를 기록한다. 소프트웨어 실행이 완료되면 PT는 해당 소프트웨어의 트레이스 데이터를 처리하여 정확한 프로그램 흐름을 재구성할 수 있다 [3][4][5].

리눅스 시스템의 경우 PT를 기반으로 하는 하드웨어 트레이스 프로그램이 운영체제에 통합되어 perf와 같은 명령을 통하여 사용할 수 있다. 그러나 윈도우 시스템의 경우에는 커널 개방과 같은 문제로 인하여 프로파일링 및 디버깅 메커니즘과의 긴밀한 통합은 이루어지지 않고 있다. 이를 위

해 몇몇 개인이나 단체들이 윈도우 환경에서 PT를 구현하고 있다. 그러나 perf나 윈도우 환경 모두 PT를 이용하여 단일 프로세스만 트레이스 할 수 있고 다중 프로세스 스트림을 트레이스 하는 방법은 제공하고 있지 않다.

본 논문에서는 이러한 단점을 극복하고자 윈도우 환경에서 다중 프로세스 스트림을 트레이스 지원이 가능하도록 기존의 PT 트레이스 프로그램을 확장하는 방안을 제안하였다.



(그림 12) 다중 스트림 트레이스 덤프 데이터

참고문헌

[1] Napoleon C. Paxton, “Cloud Security: A Review of Current Issues and Proposed Solutions,” International Conference on Collaboration and Internet Computing (CIC), pp. 452-455, 2016.

[2] Tahira Mahboob; Maryam Zahid; Gulnoor Ahmad, “Adopting information security techniques for cloud computing—A survey,” International Conference on Information Technology, Information Systems and Electrical Engineering (ICITISEE), pp. 7-11, 2016.

[3] Jörg Thalheim; Pramod Bhatotia; Christof Fetzer, “INSPECTOR: Data Provenance Using Intel Processor Trace (PT),” International Conference on Distributed Computing Systems (ICDCS), pp. 25-34, 2016.

[4] Khalid El Makkaoui; Abdellah Ezzati; Abderrahim Beni-Hssane; Cina Motamed, “Cloud security and privacy model for providing secure cloud services,” 2016 2nd International Conference on Cloud Computing Technologies and Applications (CloudTech), pp. 81-86, 2016.

[5] Bob Duncan; Alfred Bratterud; Andreas Happe, “Enhancing cloud security and privacy: Time for a new approach?,” International Conference on Innovative Computing Technology (INTECH), pp. 110-115, 2016.

[6] Sin-Fu Lai; Hui-Kai Su; Wen-Hsu Hsiao; Kim-Joan Chen, “Design and implementation of cloud security defense system with software defined networking technologies,” International Conference on Information and Communication Technology Convergence (ICTC), pp. 292-207, 2016.

[7] Andi Kleen, “Simple Intel CPU processor tracing on Linux,” https://github.com/andikleen/simple-pt

〔저자소개〕

김 현 철 (Hyuncheol Kim)
1990년 2월 성균관대학교 학사
1992년 2월 성균관대학교 석사
2005년 8월 성균관대학교 박사
2006년 9월 ~ 현재 남서울대학교
컴퓨터소프트웨어학과 교수
email : hckim@nsu.ac.kr

김 영 수 (Youngsoo Kim)
1998년 2월 성균관대학교 학사
2000년 2월 성균관대학교 석사
2009년 8월 성균관대학교 박사
2000년 2월 ~ 현재 한국전자통신연구원 책임연구원
email : blitzkrieg@etri.re.kr

김 종 현 (Jonghyun Kim)
2000년 2월 오클라호마 주립대 석사
2005년 2월 오클라호마 주립대 박사
1995년 ~ 1998년 삼성전자 연구원
2005년 ~ 현재 한국전자통신연구원
책임연구원
email : jhk@etri.re.k