DOI: 10.3745/KIPSTC.2011.18C.3.157

# 커널 기반 가상머신을 이용한 시스템 무결성 모니터링 시스템

남 현 우[†] · 박 능 수[††]


## 요 약

가상화 계층은 커널 보다 높은 권한 계층에서 수행되어 운영체제가 사용하고 있는 자원 정보를 모니터링 하는데 적합하다. 하지만 기존 가상화 기반 모니터링 시스템은 CPU나 메모리 사용률과 같은 기초적인 정보만을 제공하고 있다. 본 논문에서 메모리, 레지스터 GDT, IDT 그리고 시스템 콜과 같은 동적인 시스템 커널 객체를 모니터링하기 위하여 전가상화 방식의 모니터링 시스템을 제안한다. 모니터링 시스템을 검증하기 위해 커널의 수정 없이 바로 리눅스 커널에 적용된 전가상화 방식의 KVM을 기반으로 시스템을 구현하였다. 구현된 시스템은 KVM 내부 객체에 접근하기 위한 KvmAccess 모듈, 그리고 가상머신 모니터링 결과를 외부 모듈에서도 사용할 수 있도록 API를 제공하였다. 구현된 모니터링 시스템의 성능을 측정한 결과 1초 주기로 시스템을 모니터링을 하더라도 0.37% 정도의 CPU 점유율을 차지하여 그 성능 부하가 아주 작았다.


**키워드 : 전가상화, 모니터링 시스템, 보안, KVM, 리눅스 커널**

# System Integrity Monitoring System using Kernel-based Virtual Machine

Hyunwoo Nam[†] · Neungsoo Park[††]


## ABSTRACT

The virtualization layer is executed in higher authority layer than kernel layer and suitable for monitoring operating systems. However, existing virtualization monitoring systems provide simple information about the usage rate of CPU or memory. In this paper, the monitoring system using full virtualization technique is proposed, which can monitor virtual machine's dynamic kernel object as memory, register, GDT, IDT and system call table. To verify the monitoring system, the proposed system was implemented based on KVM(Kernel-based Virtual Machine) with full virtualization that is directly applied to linux kernel without any modification. The proposed system consists of KvmAccess module to access KVM's internal object and API to provide other external modules with monitoring result. In experiments, the CPU utilization for monitoring operations in the proposed monitering system is 0.35% when the system is monitored with 1-second period. The proposed monitoring system has a little performance degradation.


**Keywords : Full Virtualization, Monitoring System, Security, KVM, Linux Kernel**

## 1. 서 론

최근 가상화 기술은 다양한 분야에서 그 가능성을 주목하고 있으며 다양한 기술 분야에 기반 기술로서 사용되고 있다. 가상화 기술은 여러 대의 물리적 서버를 한 대의 서버로 통합함으로써 전력소비를 줄일 수 있어 그린 IT(Green IT)의 핵심 기술로 관심을 끌고 있다. 또한 컴퓨팅 자원을 논리적 단위로 관리하여 필요한 만큼의 컴퓨팅 자원을 적재적소에 할당할 수 있어 클라우드 컴퓨팅(Cloud Computing)의 기반 기술로서도 부각이 되고 있다. 이러한 기술의 발전에 부응하여 보안 시장에서도 가상화 기술을 이용한 보안 솔루션이 점점 증가하고 있는 추세이다.

가상화 계층은 커널보다 상위 권한을 가지고 있어 커널에서 사용되는 모든 자원의 접근을 완벽하게 제어가 가능하여 보안 시스템에서 유용하다. 가상화 기술은 임베디드 분야에서도 포팅작업 없이 다양한 플랫폼을 운용할 수 있는 기술로 사용되며 이종 멀티코어 임베디드 기기에서는 멀티코어 자원의 효율적 사용을 위해서 사용될 수 있을 것이다. 다양한 분야에서 가상화 기술이 탑재된 시스템들이 증가함에 따라 특화되고 전문화된 가상화 기반 모니터링 시스템의 필요성이 증대되고 있다.

※ 이 논문은 2009학년도 건국대학교의 지원에 의하여 연구되었음.
† 정 회 원 : (주)한국스마트카드 대리
†† 종신회원 : 건국대학교 컴퓨터공학부 부교수(교신저자)
논문접수 : 2011년 3월 9일
수 정 일 : 1차 2011년 3월 28일
심사완료 : 2011년 3월 28일

본 논문에서는 리눅스 커널에 적용된 KVM(Kernel-based Virtual Machine)[1][2][3] 모니터를 기반으로 구현된 가상머신 모니터링 시스템을 제안하였다. 제안된 시스템은 가상머신의 자원 사용 정보나 상태 정보들을 모니터링 하기 위한 KvmAccess 모듈과 가상머신의 모니터링 결과를 외부에서도 사용할 수 있도록 API를 제공한다.

논문의 구성은 2장에서 관련연구들을 살펴보며 3장에서 배경지식에 대해서 정리하였다. 4장에서는 논문에서 구현된 모니터링 시스템의 설계과정을 5장에서는 구현된 모니터링 시스템을 이용한 응용시스템의 구현과정에 대해 살펴보고 6장에서는 결론 및 향후연구에 대하여 정리하였다.

## 2. 관련 연구

### 2.1 모니터링 시스템

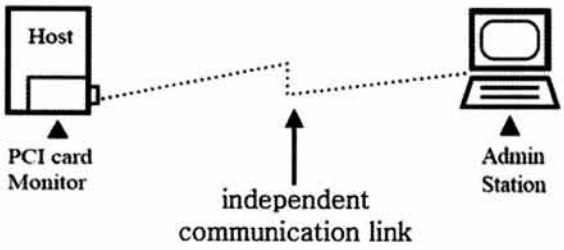
서버 관리를 위하여 시스템 관리자들은 프로세서, 메모리, I/O와 같은 여러 요소들을 관련 모니터링 도구들을 사용하여 이상 징후나 침입 탐지를 발견한다. 하지만 현재 사용되고 있는 모니터링 도구들과 IDS들은 호스트 운영체제에 의존적인 구조로 구현되었거나 루트킷과 같은 커널 레벨 공격에 취약하다는 단점이 있다. 본 절에서는 호스트 운영체제 기반이 아닌 하드웨어 기반과 가상화 기반의 모니터링 시스템에 대해서 살펴보겠다.

(1) 하드웨어 기반 모니터링 시스템

소프트웨어 기반의 보안 방법은 루트킷과 같은 공격 방법에 여전히 취약성을 가지고 있으며 많은 제약 조건들이 따르기 때문에 완벽한 모니터링 및 보안 시스템을 구축하기가 쉽지 않았다. 루트킷으로 인하여 커널 내부가 수정될 수 있었으며 커널 기반의 보안 시스템에도 공격이 가해지게 되었다. 이러한 문제를 해결하기 하드웨어 도움을 받아 커널을 모니터링 하는 연구로는 Copilot monitor(a Coprocessor-based Kernel Runtime Integrity Monitor)가 있다[4]. 이 연구는 하드웨어로 구현된 보조 프로세서를 이용하여 커널의 동적 객체의 무결성을 모니터링 하고 루트킷을 검출해내는 시스템이다.

Copilot의 시스템 구조는 (그림 1)과 같다. Copilot는 PCI 카드 형태로 구현되어 루트킷의 Signature를 메인 메모리로부터 스캐닝 하여 커널 객체의 수정 여부를 검출한다. 모니터링 대상 커널 객체는 시스템 콜 테이블과 IDT 테이블, 커널 코드 영역이 수정되었는지 여부, 그리고 숨겨진 프로세스나 소켓 등이 있다. 또한 커널 객체[5]들을 접근하기 위해서 System.map 파일을 참조하여 특정 커널 심볼의 주소를 알아내고 하드웨어에서 가상주소를 물리주소로 변환한 후 직접 물리 메모리에 접근한다.

하드웨어 기반 모니터링 방법은 루트킷과 같은 공격도구로부터 안전하며 커널에 인지되지 않고 운영할 수 있다는 장점이 있다. 하지만 커널이 수정 될 경우 이에 맞게 모니터링 하드웨어의 수정이 필요하고 하드웨어 기반에서도 메모리 접근시 데이터 버스를 점유해야 하기 때문에 전체적인 시스템의 성능은 낮아진다.

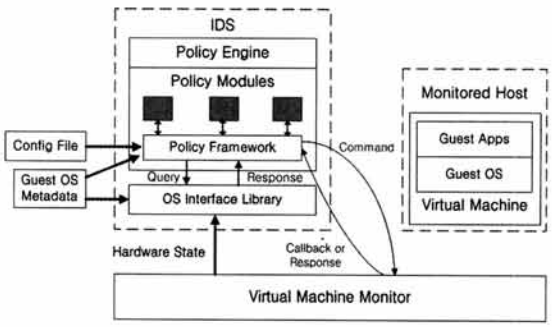



(그림 1) Copilot 시스템 구조

(2) 가상화 기반 모니터링 시스템

이 연구는 가상화 기술을 이용하여 IDS를 설계할 때 필요한 요구 사항들을 명확하게 정리하였다[6]. (그림 2)는 시스템의 구조를 보여주며 하단에 VMM 위에는 IDS가 하나의 Guest OS처럼 수행되고 있다. IDS에서는 VMM에게 하드웨어 상태 값들을 모니터링 하여 받아오며 VMM에게 받아온 값들은 OS 인터페이스 라이브러리를 통해 Policy Engine 에게 전달된다. Policy Engine은 config File와 Policy Modules들에 의해 모니터링 데이터들을 해석하고 정책에 따라 수행해야 할 명령들을 VMM에게 전달한다. 시스템에서 감시하고자 하는 대상은 우측에 있는 Monitored Host의 운영체제이다. Policy Modules은 공격자들이 ps나 공유라이브러리 또는 /proc 파일 시스템을 수정하는지, 메모리에 공격 signature가 존재하는지를 검색하거나 패킷에 악성코드가 포함 되었는지를 검출하기도 한다.

위와 같은 기능들은 기존 IDS에서 제공되고 있지만 가상화 기반에서 구현될 경우 커널 기반 공격으로부터 안전하고 IDS의 존재가 공격도구에 검출되지 않게 되는 장점을 가질 수 있다.




(그림 2) 가상화를 이용한 IDS의 설계 구조

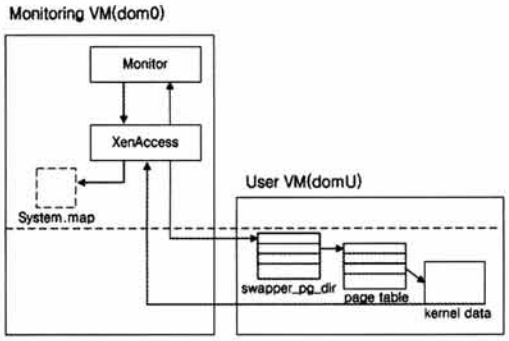
### 2.2 XenAccess

XenAccess는 오픈소스 VMM인 Xen[7][8]에서 동작중인 가상머신을 모니터링 할 수 있도록 도와주는 라이브러리이다[9]. 만약 가상머신의 운영체제가 리눅스나 윈도우즈라면 PCB(Process Control Block)와 같은 커널 내부 객체를 참조

할 수 있도록 도와주는 API를 제공하고 있다. 이 라이브러리는 Xen을 이용한 가상화 시스템에서 가상 머신을 모니터링 할 때 사용될 수 있을 것이다. 하지만 XenAccess는 반가상화 기반의 Xen을 사용하기 때문에 Guest의 운영체제를 수정해야만 한다. 따라서 Guest 운영체제의 커널 소스가 공개된 경우만 지원할 수 있다. 이는 윈도우즈와 같이 비공개 운영체제의 경우 지원할 수가 없으며 또한 Guest 커널을 Xen VMM의 요구대로 수정해야 하기 때문에 커널 버전에 따른 업그레이드를 빠르게 지원할 수 없다는 단점이 있다.

(1) Virtual Memory Introspection

XenAccess의 핵심적인 기능으로 가상머신의 메모리를 읽거나 커널 객체에 접근하는 기능이다. 단순히 메모리를 읽거나 쓰는 작업이 아니라 커널 객체에 접근하기 위해서는 (그림 3)과 같이 커널 버전에 맞는 System.map 파일을 가지고 있어야 한다. XenAccess에서 제공하고 있는 메모리 감시 API를 살펴보면 다음과 같다.




(그림 3) XenAccess에서 커널 객체 접근

• xa_access_kernel_symbol()

특정 도메인의 커널 객체를 접근할 때 사용하는 API 이다. (그림 3)은 XenAccess를 이용하여 Guest OS의 커널 객체를 접근 하는 방법을 도식화 하였다. XenAccess에게 작업을 요청하며 XenAccess는 커널 객체의 가상 주소를 해당 System.map 파일을 참조하여 커널 객체의 주소를 알아내고 페이지 테이블을 거쳐 해당 커널 데이터에 접근하게 된다. 커널 객체를 접근한 후 XenAccess로 리턴되고 결과는 Monitor에게 전달한다.

• xa_access_user_virtual_address()

특정 도메인에서 사용자 메모리의 가상 주소로 메모리를 접근 한다. 인자로는 사용자 메모리가 각 프로세스 ID에 따라 달라지게 되므로 반드시 프로세스 ID가 명시되어야 한다.

(2) Virtual Disk Monitoring

XenAccess에서 제공하는 기능으로는 메모리 감시 기능 외에 디스크 모니터링 기능이 있다. User VM에서 파일 관련 연산이 수행되면 모니터 VM에서는 파일 연산을 가로채어 파싱 작업을 통하여 모니터링 대상인지를 파악하고 처리한다.

• xadisk_set_watch() / xadisk_unset_watch()

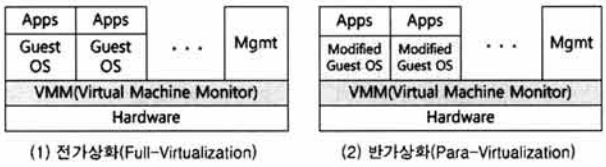
파일 시스템에서 디렉토리에 대해서 watch 포인트를 설정하거나 해제한다.

## 3. 배경지식

3.1 가상화 시스템

가상화 기술은 1960년대 IBM의 메인프레임에서부터 도입되었으며 서버의 활용도를 높이기 위한 방안으로 시작되어 현재 서버 가상화, 애플리케이션 가상화, 네트워크 가상화, 스토리지 가상화 등 많은 분야로 그 활용 범위를 넓혀가고 있다. 가상화 환경을 만들어주는 소프트웨어를 VMM(Virtual Machine Monitor) 이라고 부르며 구현 방법에 따라 여러 종류로 분류할 수 있다.

가상화는 구현 방법에 따라 다양하게 분류 되는데 크게 (그림 4)와 같이 두 종류로 나눌 수 있으며 각각의 특징에 대해서 살펴보겠다.




(그림 4) 가상화 방식에 따른 시스템 구조

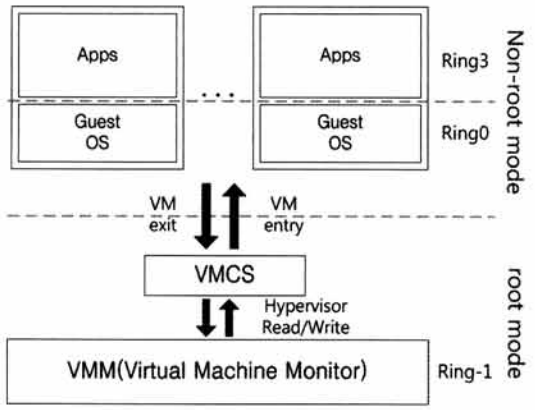
전가상화(Full-Virtualization)는 완벽하게 하드웨어를 가상화하기 때문에 커널을 수정할 필요가 없다. 하지만 x86 시스템의 경우 특권 명령어를 수행할 때 문제가 발생할 수 있다. 이러한 문제를 해결하기 위해 VMware는 명령어 코드들을 Binary Translation 하여 해결했으며 최근 CPU에서는 INTEL VT-x나 AMD-V와 같은 가상화 확장 기능을 이용하여 x86 환경에서 전가상화를 구현하기도 한다.

반가상화(Para-Virtualization)는 하드웨어를 부분적으로만 가상화 시키는 방법으로 특권 명령어나 특권 레지스터를 사용할 때에만 Hypercall을 호출하여 VMM을 수행한다. 오버헤드(Overhead)가 상대적으로 작기 때문에 네트워크나 디스크 I/O 같은 작업에서 높은 성능을 보여준다. 단점으로는 Guest의 커널을 VMM에서 요구하는 인터페이스에 맞게 수정해야 한다는 것이다. 그래서 Windows와 같이 커널이 공개되지 않는 운영체제는 반가상화 방식으로는 구현하기가 힘들다.

INTEL VT-x[10]와 AMD-V[11]는 프로세서에서 제공하는 가상화 확장 기능을 말한다. 기존 x86 기반 시스템은 Ring0~Ring3 까지 4개의 권한 모드로 구성되어 있다. 일반적으로 Ring0은 가장 높은 권한 모드로서 커널영역이 사용하며 하드웨어 설정 및 특권 명령과 특권 레지스터의 사용이 가능하다[12].

소프트웨어 기술로만 구현되었던 가상화 시스템의 경우 Ring0을 VMM이 사용하고 커널은 Ring1~Ring3을 사용하면

서 커널이 특권 레벨의 작업들을 수행할 수 없다. 따라서 커널은 특권 명령어와 관련된 소스 코드들을 모두 수정해야만 했다. 이를 해결하기 위해 INTEL과 AMD는 Ring-1 (Root mode) 권한 계층을 새롭게 추가하여 특권 명령어 사용과 관련된 문제를 해결하였다. INTEL VT-x는 (그림 5)와 같이 VMM은 Ring-1 계층에 Guest OS는 기존처럼 Ring0 계층에서 수행되어 특권 명령어를 수행하는데 더 이상 문제가 발생하지 않게 되었다.




(그림 5) INTEL VT-x에서의 가상화 계층

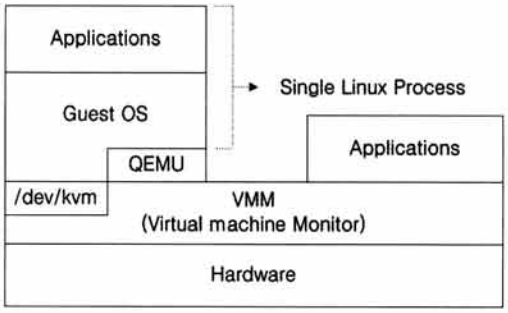
VMCS(Virtual Machine Control Structure)는 가상머신이 전환될 때마다 가상머신의 상태정보를 저장하는 공간이며 400byte 정도의 크기를 갖는다. 이는 운영체제의 PCB (Process Control Block)과 비슷한 개념으로 VMCS에는 Guest 그리고 Host의 레지스터와 가상머신의 상태와 같은 하드웨어 정보들이 기록되어 있다[13].

하드웨어 확장 기능을 사용하지 않았던 VMM에서 주소 변환 시 Guest 가상주소를 Guest 물리주소로 변환하고, 다시 Guest 물리주소를 Host 물리 머신의 주소로 변환시키는데 많은 오버헤드가 발생한다. 이를 해결하기 위해 INTEL과 AMD에서는 EPT(Extended Page Table)와 NPT(Nested Page Table)라는 가상화 환경을 위한 메모리 관리 가능을 제공하고 있다. EPT와 NPT를 사용하면 Guest의 페이지 테이블과 Host의 페이지 테이블을 동시에 참조하여 주소 변환 과정을 한 번에 처리하므로 주소변환에 소비되었던 오버헤드를 줄여줄 수 있다.

### 3.2 KVM

KVM은 리눅스 커널 2.6.20 버전부터 커널의 일부로 등장한 가상화 기법이다[1][2]. KVM은 리눅스 커널 모듈 형태로 구현되었으며 INTEL VT-x와 AMD-V와 같은 하드웨어 가상화 기능을 사용한다. KVM은 새로운 가상화 기법을 선보이고 있다.

KVM는 (그림 6)과 같이 커널 모듈형태로 구현되어 있으며 KVM 모듈이 업로드 되면 리눅스 커널은 CPU의 가상화 기능들을 초기화 한 후 VMM으로 전환 된다. 이러한 구조로 인해 KVM은 리눅스 커널에서 사용하고 있는 프로세스




(그림 6) KVM의 구조

관리, 메모리 관리, 디바이스 드라이버와 같은 많은 부분들을 직접 구현하지 않아도 된다. KVM은 외형적으로는 /dev/kvm와 같은 디바이스를 외부에 공개하고 커널의 Guest 모드를 활성화 하는데 사용한다. (그림 6)을 보면 KVM의 관점에서 보면 하나의 Guest OS는 단일 프로세스와 같다. KVM은 가상화 기능이 활성화된 리눅스 커널로 간주 할 수 있으므로 리눅스 응용 프로그램을 실행한다.

KVM 디바이스 드라이버는 read나 write 연산이 불가능하며 오직 ioctl 시스템 콜만을 사용하여 가상머신에게 메모리나 가상 하드웨어들을 할당 하게 된다. 그 외 KVM에서 I/O 가상화는 KVM에서 직접 구현하지 않고 QEMU 에뮬레이터를 사용하여 I/O 가상화를 처리하고 있다.

본 연구에서는 전가상화 방식의 KVM을 이용하여 Guest OS의 수정 없이 시스템을 모니터링 할 수 있는 가상화 기반 모니터링 시스템을 구현하고자 한다.

## 4. KVM 기반 모니터링 시스템의 설계

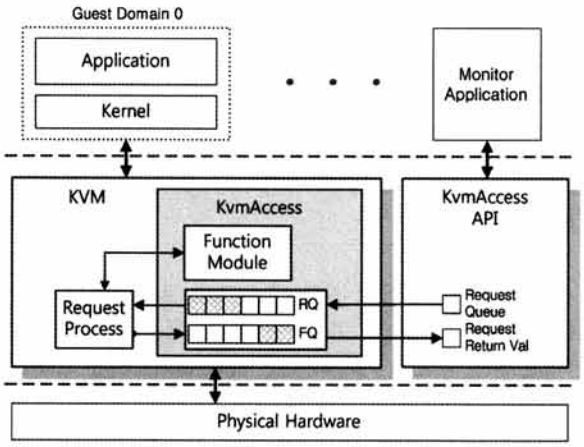
### 4.1 모니터링 시스템의 구조

(그림 7)은 모니터링 시스템의 구조를 보여주고 있으며 시스템은 크게 KvmAccess 모듈과 KvmAccess API로 나뉜다. KvmAccess 모듈은 가상머신인 게스트 영역을 모니터링 하기 위해서 KVM 내부 객체에 접근해야 할 때 사용하는 모듈이다. 그리고 KvmAccess API는 외부 모듈에게 가상머신의 모니터링 결과를 전달하고자 할 때 제공되는 API이다. 따라서 API를 이용하면 외부 모듈에서 가상머신을 모니터링 하는 기능이 필요한 응용프로그램들을 구현할 수 있게 된다.

KVM 내부에는 프로세스 요청 리스트를 검색하여 처리를 요청하는 Request Process 모듈이 있다. 이 모듈은 ka_request_queue_process() 함수이며 VM간에 스케줄링 작업이 발생할 때마다 실행되는 vcpu_load() 함수에 의해 호출된다. 이 모듈은 작업 요청이 발생할 때마다 요청 작업을 RQ(Request Queue)에서 가져와 처리하고 작업 요청 처리가 모두 끝나게 되면 처리 결과 값을 FQ(Finish Queue)에 삽입한 후 작업을 완료한다.

KvmAccess 모듈에서 Function Module은 모니터링 시스템이 제공하고 있는 기능들을 대부분 제공하고 있다. 이 모듈의 구현 방법으로는 KVM에서 EXPORT된 함수나 자료

구조들을 직접 접근하여 가상머신에서 필요한 정보들을 가져온다. Function Module에는 가상머신의 메모리, CPU의 레지스터 또는 인터럽트 테이블 등을 접근하고자 할 때 사용할 수 있는 함수들이 구현되어 있다.




(그림 7) 모니터링 시스템의 시스템 구조도

지금까지 모니터링 시스템의 내부 모듈들에 대해서 살펴보았다. 하지만 처리 과정을 살펴보면 작업 요청이 바로 처리되는 것이 아니라 작업 리스트에 등록된 후 모니터링 대상 게스트가 스케줄링 될 때 처리되므로 이는 API의 응답 속도를 느리게 하는 요소가 되었다. 따라서 현재 시스템은 특정 게스트가 실행상태가 되었을 때 작업 리스트에서 해당 게스트의 요청 작업들을 처리하는 구조로 되어 있다.

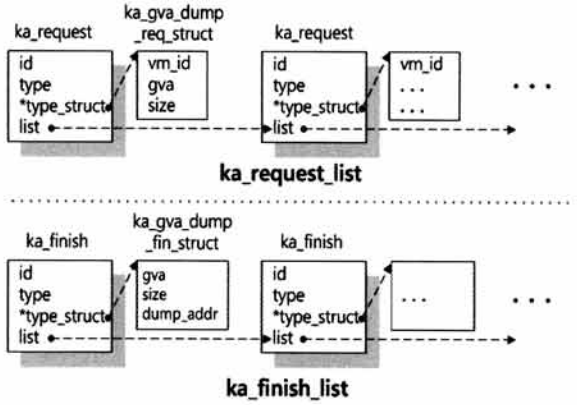
### 4.2 시스템 설계 및 구현

(그림 8)은 작업 요청 리스트와 처리 결과를 담고 있는 작업 완료 리스트의 구조를 보여 주고 있다. 각 작업 요청마다 ka_request 자료구조를 생성하며 내부에는 작업 요청마다 개별적인 id, 작업의 종류를 나타내는 type, 작업 종류에 따라 세부 내역이 담겨져 있는 type_struct 필드들이 존재한다. id는 작업 요청마다 증가하면서 부여되게 되는데 추후 작업 완료 리스트에서 id를 기준으로 요청 작업의 결과값를 찾게 된다. 그 다음으로 type과 type_struct 필드는 작업 종류에 따라 필요한 정보가 다르기 때문에 따로 세부 항목을 담는 자료구조를 생성하고 포인팅 하는 용도로 사용되고 있다.

(그림 8)에서 ka_request_list에 첫 번째 ka_request에는 ka_gva_dump_req_struct 자료구조를 가리키고 있는데 이 자료구조는 가상머신의 메모리 읽기 요청이 발생했을 때 사용된다. 작업 완료 리스트에도 이와 마찬가지로 메모리 읽기 작업 요청에 대한 결과를 ka_gva_dump_fin_struct 자료구조에 담는다. 다른 작업 요청의 경우 type_struct 필드는 작업 종류에 따라 가리키는 자료구조가 달라진다.

#### (1) 가상머신의 메모리 읽기 및 쓰기

가상머신에서 운영체제를 모니터링하기 위해서는 가장 먼




(그림 8) 작업 요청 리스트와 완료 리스트의 구조

저 가상머신의 가상 주소로 메모리를 읽어줄 수 있어야 한다. 가상머신의 메모리를 읽고 쓸 수 있다면 운영체제 내부의 자료구조들을 접근하거나 수정하여 시스템을 제어 할 수 있기 때문이다. 구현 과정으로는 작업 요청큐에 메모리 읽기 작업에 대한 요청을 등록하면 Request Process 모듈에서는 kvm_read_guest_virt()함수를 사용하여 가상머신의 메모리를 읽어준다. 읽기 작업이 완료되면 작업 완료 리스트에 결과 값을 등록하고 작업을 완료하게 된다. 이 함수는 Guest의 가상주소인 gva를 인자로 넘겨주며 해당 위치에서 메모리를 bytes 크기만큼 읽어주는 기능을 수행한다. 함수 내부의 세부 처리 과정을 정리하면 다음과 같다.

1. gva_to_gpa() 함수를 사용하여 gva를 gpa로 변환 해준다. gva_to_gpa() 함수 내부에 walk_addr() 함수는 해당 가상머신의 CR3 레지스터를 참조하여 페이징 테이블을 찾아내고 주소 변환 작업을 수행한다.
2. gpa가 구해지면 kvm_read_guest()함수를 이용하여 메모리 읽기 작업을 수행한다. 우선 gpa는 gfn_to_hva 함수를 통해 hva 주소로 변환된다.
3. 최종 변환된 hva 주소는 사용자 프로세스 공간의 메모리기 때문에 copy_from_user()함수를 이용하여 메모리를 읽어주고 작업을 완료한다.

모니터링 시스템에서 메모리를 읽는 과정은 작업 요청 리스트를 사용하여 처리되는데 처리 과정을 살펴보면 다음과 같다.

1. 작업 요청 리스트에 작업 요청을 등록한다. 메모리 덤프에 사용되는 자료구조는 ka_gva_dump_req_struct이며 읽을 주소인 gva와 크기 size를 담고 있다.
2. KVM의 vcpu_load() 함수가 호출될 때 ka_request_queue_process() 함수를 실행하여 작업 요청을 처리해준다. 이 단계에서는 gva와 size를 보고 메모리를 읽어준다.
3. 작업이 끝나면 처리 결과 값을 ka_gva_dump_fin_struct에 저장한 후 작업 완료 리스트에 등록한다.
4. 작업을 요청한 측에서는 ka_finish_get() 함수에서 작업

처리결과가 리스트에 등록될 때 까지 대기하고 있다. 작업 완료 리스트에 요청 작업의 id와 동일한 처리결과가 등록되면 값을 리스트에서 가져오고 난 후 작업을 종료하게 된다.

(2) CPU 레지스터 값 읽기

CPU의 레지스터 값은 kvm_vcpu 자료구조에 kvm_vcpu_arch 필드에 기록되어 있다. 세부 내용을 살펴보면 EAX, EBX, ECX와 같은 레지스터들은 regs 배열에 담겨져 있으며 CPU의 개수만큼 배열이 존재한다. 그 외 CPU 컨트롤 레지스터인 CR 레지스터들이 존재한다.

레지스터 읽기에 대한 요청 작업이 등록되면 KvmAccess는 해당 가상머신의 레지스터에 대한 값들을 위 kvm_vcpu_arch 자료구조에서 읽어와 ka_register_read_fin_struct 자료구조에 담게 된다.

### 4.3 KvmAccess API

모니터링 라이브러리는 외부 프로그램에서 모니터링 결과를 사용할 수 있도록 API 형태로 제공한다. 4.2 (1)에서 살펴본 메모리 읽기 API와 마찬가지로 다른 API들도 비슷한 구조로 작업 요청을 처리한다. 현재 구현된 API들은 가상머신의 메모리, 레지스터, GDT, IDT 그리고 System Call Table을 접근하거나 수정할 수 있도록 기능들을 제공하고 있다. 다음으로 본 모니터링 시스템에서 제공하는 API들에 대해 상세히 설명하도록 하겠다.

(1) 가상머신의 메모리 읽기 / 쓰기

가상머신의 메모리를 읽거나 쓰고자 할 때 사용한다. <표 1>과 같이 메모리를 읽을 때는 읽을 주소인 gva와 크기 size를 인자로 넘겨준다. 메모리에 쓰기작업을 할 때에는 추가적으로 쓰려는 데이터의 포인터를 인자로 넘겨준다.

〈표 1〉 ka_gva_read() / ka_gva_write() 함수의 원형

```
unsigned int ka_gva_read(unsigned int gva, unsigned int size);
unsigned int ka_gva_write(unsigned int gva, unsigned int *data,
unsigned int size);
```

(2) 가상 머신의 레지스터 읽기

가상머신의 레지스터를 읽고자 할 때 사용한다. <표 2>와같이 레지스터의 읽기 작업이 모두 완료되게 되면 레지스터 값들이 저장된 ka_request_register_fin_struct 자료구조의 포인터를 리턴 한다.

〈표 2〉 ka_register_read() 함수의 원형

```
struct ka_request_register_fin_struct *ka_register_read(void);
```

(3) GDT의 리스트 읽기 / 쓰기

GDT(Global Descriptor Table)는 메모리 세그먼트 영역의 시작주소와 크기, 속성을 담고 있는 디스크립터이다. <표 3>을 보면 GDT를 읽어주는 API의 경우 GDT 디스크립터를 담고 있는 자료구조를 리턴한다. GDT에 쓰기 작업을 수행할 때에는 몇 번째 디스크립터인지를 나타내는 nums와 쓰고자 하는 GDT 디스크립터 값을 인자로 넘겨준다. gdtr 레지스터는 GDT 테이블의 시작주소를 가리킨다.

〈표 3〉 ka_gdt_read() / ka_gdt_write() 함수의 원형

```
struct gdt_descriptor_struct ka_gdt_read(void);
unsigned int ka_gdt_write(int nums, struct gdt_descriptor_struct
*gdt_desc)
```

GDT 디스크립터에는 시작 주소(Base Address)와 크기정보(Size) 필드가 있으며 Type 필드에는 세그먼트 영역의 권한 정보가 들어 있다. 일반적으로 gdtr 레지스터를 참조하여 GDT 테이블의 시작 주소를 알아내지만 본 API에서는 GDT의 주소는 고정된 것이므로 매번 gdtr를 참조하여 시작 주소를 알아내지 않고 사전에 GDT의 시작 주소를 찾은 다음 접근 하였다.

(4) IDT의 리스트 읽기 / 쓰기

가상머신의 IDT(Interrupt Descriptor Table)을 읽거나 쓰고자 할 때 사용된다. <표 4>와 같이 읽기 API는 IDT 테이블의 시작 주소를 리턴하며 쓰기 API를 사용 할 때에는 쓰기 작업을 하려는 디스크립터 위치와 디스크립터 값을 인자로 넘겨준다.

〈표 4〉 ka_idt_read() / ka_idt_write() 함수의 원형

```
struct idt_descriptor_struct *ka_idt_read(void);
unsigned int ka_idt_write(int nums, struct idt_descriptor_struct
*idt_desc);
```

IDT는 인터럽트 핸들러의 디스크립터가 기록되어 있는 테이블을 말하며 idtr 레지스터가 테이블의 시작주소를 가리키고 있다. 실험에 사용한 linux-2.6.24-16 커널에서는 IDT가 0xC0410000 번지에 위치하였다.

IDT 디스크립터는 세그먼트 셀렉터 정보와 오프셋을 기록하는 필드 그리고 권한 레벨을 기록하는 DPL 필드가 있다. GDT와 마찬가지로 IDT 테이블의 시작주소로부터 해당 벡터만큼 디스크립터의 크기(8byte) 단위로 주소를 계산하여 인터럽트 디스크립터에 해당하는 메모리 주소를 구한 다음 디스크립터에 접근 하였다.

(5) 리눅스 커널 시스템 콜 핸들러 읽기 / 쓰기

시스템 콜은 사용자 프로세스에서 디스크 읽기와 같이 커널의 도움이 필요한 작업들을 처리하고자 할 때 사용하는 인터페이스이다. 시스템 콜 핸들러 읽기 API는 테이블의 시작 주소를 리턴하며 쓰기 API는 특정 벡터에 해당하는 시

〈표 5〉 ka_syscall_table_read / write() 함수의 원형

```
unsigned int *ka_syscall_table_read(void)
unsigned int ka_syscall_table_write(int nums, unsigned int
handler);
```

스템 콜 핸들러를 수정할 수 있도록 벡터를 명시하는 nums와 핸들러의 주소 값인 handler 인자로 넘겨준다.

리눅스에서 시스템 콜 테이블의 시작주소를 가리키는 sys_call_table 심볼을 찾기 위해 system.map 파일을 참조하였다. 시스템 콜 테이블은 핸들러의 주소가 32bit 시스템의 경우 4byte 크기의 엔트리로 구성된 테이블 자료구조다. 실험이 진행 되었던 시스템의 경우 시스템 콜 테이블의 시작주소는 0xC0326520 번지에 위치하였다.

### 4.4 API의 처리 속도 측정

API의 처리 속도를 측정하기 위하여 <표 6>과 같은 환경에서 실험을 하였다.

〈표 6〉 실험 환경

| | |
|---|---|
| CPU | Intel Xeon 2.66GHz(Quad-core) with VT-x |
| OS | Ubuntu 8.04 |
| KVM | KVM 8.4 |

<표 7>은 각 API에서 작업 요청에서부터 작업이 완료될 때 까지 걸리는 시간을 측정한 결과이다. 측정 방법으로는 각 API 마다 반복적으로 100번씩 수행한 다음 실제 API가 처리되는 시간만을 측정하여 평균 시간 값을 계산하였다. 측정 시간 단위는 us 이며 커널에서 TSC(Time Stamp Counter) 값을 읽어와 시간을 측정하였다. 본 측정 결과는 실제 CPU의 부하가 걸리는 부분만을 측정하였으며 작업 대기 리스트에서 대기하고 있는 시간은 포함하지 않았다.

먼저 메모리 읽기는 가상머신의 0xC0100000 번지로부터 4바이트를 읽는 작업을 했을 경우 소요시간이다. 아래의 결과들을 보면 GDT, IDT, 시스템 콜 읽기는 각 커널 객체의 메모리상의 주소를 미리 알고 작업을 수행하기 때문에 메모리 읽기에 비해 큰 시간 차이가 나지 않는 것을 확인할 수 있다.

〈표 7〉 API별 평균 작업시간

| 측정 항목 | 소요 시간 |
|---|---|
| 가상머신의 메모리 읽기(4byte) | 6.4 us |
| 가상머신의 레지스터 읽기 | 7.0 us |
| 가상머신의 GDT 읽기 | 7.8 us |
| 가상머신의 IDT 읽기 | 7.4 us |
| 가상머신의 시스템 콜 읽기 | 10.1 us |

## 5. KVM기반 객체 무결성 모니터링 시스템

본 장에서는 KVM 기반 모니터링 시스템에서 제공하는 API를 이용한 응용 시스템으로 객체 무결성 모니터링 시스템을 구현하였다. 구현된 응용시스템은 IDS와 같은 보안 관련 시스템에서 제공하는 기능들을 KvmAccess API를 사용하여 가상화 환경에서 구현하였다.

### 5.1 커널 객체의 무결성 모니터링 시스템

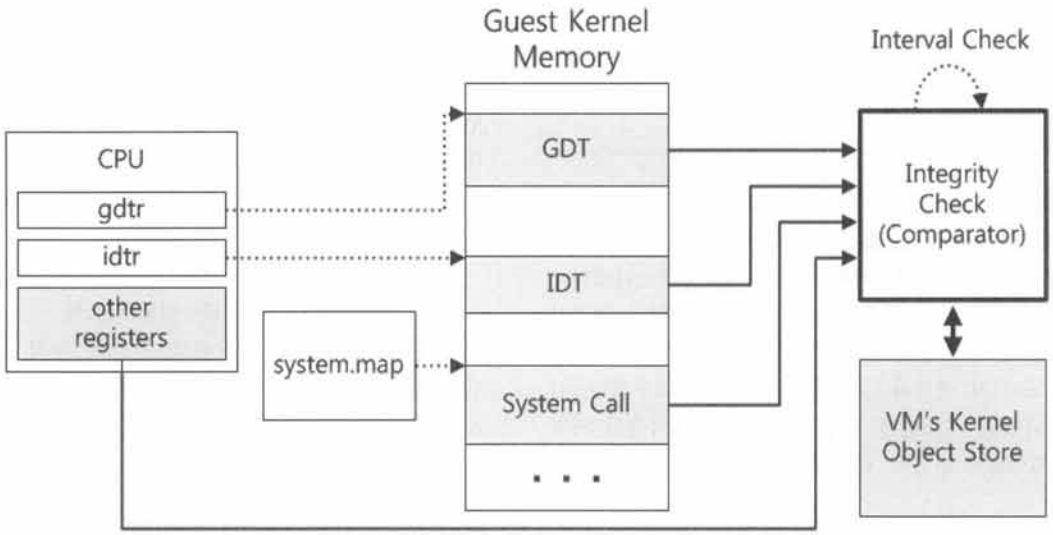
루트킷(Rookit)과 같은 커널 기반의 공격 프로그램들은 일반적으로 IDT나 시스템 콜 테이블 등을 수정 하여 후킹 공격을 시도한다. 이와 같은 공격을 방어하기 위해 보안 시스템들은 커널에서 중요한 객체들이 공격에 의해 변경되는지를 모니터링 한다. 제안하는 모니터링 시스템은 가상화 계층에서 Guest OS의 메모리를 접근하여 해당 커널 객체들을 접근하여 무결성 체크를 수행하였다.

관련연구에서 copilot 프로젝트는 추가적인 하드웨어를 사용하여 커널 메모리 영역에 접근 하였지만 본 논문에서는 추가적인 하드웨어를 필요로 하지 않는다. 또한 커널 기반에서 작동하던 모니터링 시스템의 경우 커널을 수정하거나 커널의 도움을 받아야 했지만 가상화 기반의 모니터링 시스템에서는 커널과는 독립적이어서 커널에게 인지되지 않고도 운영할 수 있다는 장점을 가지고 있다.

본 장에서는 커널 객체의 무결성을 검사하는 모니터링 시스템을 설계하고 구현하였다. 시스템은 주기적인 시간 간격으로 동작중인 커널의 변경 여부를 모니터링을 하면서 외부 공격으로 부터 커널 객체가 손상을 당했는지를 모니터링 하게 된다.

다음은 본 응용시스템에서 모니터링 하고자 하는 커널 객체들에 대해서 정리하였다.

- GDT : GDT 디스크립터는 각 세그먼트 메모리 영역에 대한 여러 속성 정보들을 기록하고 있다. 이 중 공격 프로그램들은 DPL 필드를 수정하여 사용자 영역에서도 커널 계층의 메모리 영역을 접근하게 만들거나 데이터 메모리 영역에서도 코드를 실행 시킬 수 있도록 만든 다음 공격을 시도한다.
- IDT : IDT는 인터럽트가 발생했을 때 수행되어야 할 핸들러의 정보가 기록되어 있는 테이블이다. 일반적으로 공격자들은 핸들러의 주소를 공격 코드의 시작 주소로 수정하여 해당 핸들러를 후킹하려고 한다. 보통 키보드 입력을 가로채는 키로거 같은 공격 프로그램들은 키보드 인터럽트 핸들러를 후킹하여 키 입력을 가로챌 수 있다.
- 시스템 콜 테이블 : 시스템 콜은 사용자 영역에서 커널에게 서비스를 요청할 때 사용되는 인터페이스이다. IDT와 마찬가지로 공격자들은 각 엔트리에 기록되어 있는 시스템 콜 핸들러 주소를 수정하여 후킹과 같은 공격을 시도한다.
- CR0 레지스터 : CPU를 제어하기 위해 사용되는 제어 레




(그림 9) 커널 객체 무결성 체크 시스템의 구조도

지스터이다. 레지스터에는 여러 가지 필드들이 존재하는데 WP(Write Protect) 필드가 공격 대상으로 많이 이용되고 있다. WP 필드는 페이지의 쓰기 속성을 제어하는데 값이 0이라면 페이지의 쓰기 속성 제어 기능이 비활성화되며 읽기 전용의 데이터 영역도 쓰기 작업을 가능하게 한다. 따라서 공격자들은 WP필드를 조작하여 메모리에 대한 보호 설정을 무력화 시킨다.

(그림 9)는 구현된 커널 객체 무결성 체크 시스템의 구조도이다. 시스템은 크게 가상머신에서 동작중인 커널의 객체 정보들을 저장하는 저장소인 Kernel Object Store 모듈과 일정 시간 간격으로 커널 객체들의 무결성을 체크하는 Integrity Check 모듈로 나누어 볼 수 있다. 무결성 체크 모듈은 일정 주기마다 모니터링 한 커널 객체들의 값들을 저장소에 저장된 값들과 비교한다. 만약 모니터링 커널 객체의 값이 일치하지 않는다면 커널 객체가 손상되었다는 것을 의미한다.

### 5.2 응용 시스템의 성능측정 결과

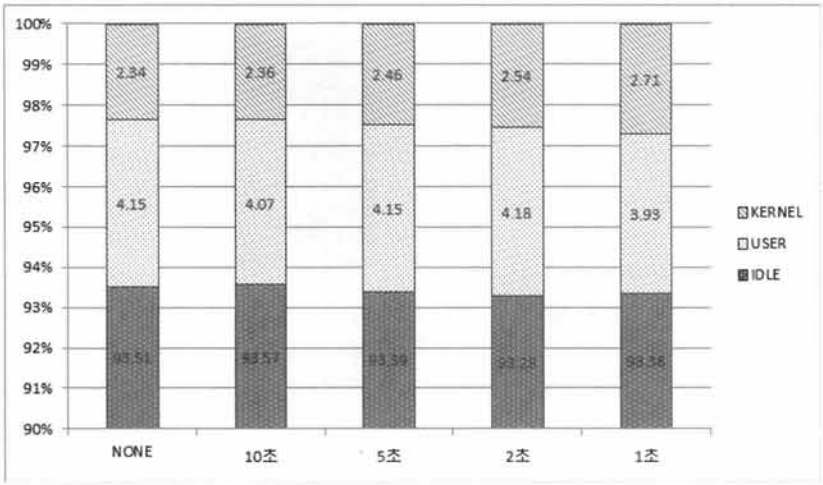
성능 평가를 위하요 <표 6>과 같은 실험 환경에서 실험을 하였다. 실험 결과인 (그림 10)을 보면 가상머신에서 웹서버를 운영하면서 커널 객체의 무결성 모니터링을 하였을 경우 발생된 CPU 점유율을 알아본 실험 결과이다. 이 실험의 목표는 실제 웹서버와 같이 서비스가 가상머신에서 운영되었을 때 가상화 기반 모니터링 시스템에서 어느 정도의 오버헤드가 발생하는지를 알아보는 것이다.

그래프에서 x축은 모니터링 주기를 나타내며 y축은 KERNEL 레벨과 USER 레벨에서 발생하는 CPU의 점유율과 IDLE 시간을 나타낸다. USER는 일반 사용자 프로그램 말하며 KERNEL은 리눅스 커널을 말한다. 구체적으로 USER에서는 웹서버가 CPU 사용률의 대부분을 차지하며 KERNEL에서는 네트워크 디바이스 드라이버에서 패킷 처리와 가상화 기반 모니터링 시스템에서 CPU 사용률의 많은 부분 차지한다. 여기서 KVM은 리눅스 커널의 모듈 형태로 동작하기 때문에 가상화 레벨에서의 커널의 성능 부하에 영향을 주게 된다.

측정 방법으로는 일정 시간 간격(1초, 2초, 5초, 10초) 단위로 모니터링 작업을 실행한 다음 5분 동안 발생한 CPU 사용률과 IDLE 시간을 측정하였다. 성능 측정 도구로는 CPU 인터럽트와 같은 이벤트 정보나 프로세스의 점유 시간을 측정해주는 mpstat을 사용하였다. 그리고 웹서버에게 고정적인 성능부하를 주기 위하여 웹서버 성능 측정도구인 siege를 사용하여 인위적인 트랙픽을 발생 시켰다. 이외 실험 정보로는 50개의 클라이언트를 이용하여 접속 요청을 발생시켰고 사용된 웹페이지의 문서는 40512 bytes 크기를 가졌다.

실험 결과를 보면 각 실험마다 동일한 트래픽을 발생시켰기 때문에 USER 레벨에서의 CPU 점유율 차이가 작은 것을 확인할 수 있다. 그리고 모니터링 주기에 따라 KVM Access 모듈이 동작하고 있는 KERNEL 레벨에서의 CPU 점유율만이 조금씩 증가하는 것을 확인할 수 있었다. 구체적으로 모니터링 시스템을 운영하지 않을 경우와 10초 간격으로 모니터링을 했을 경우 KERNEL에서의 CPU 사용률 차이는 0.02% 증가된 것을 확인할 수 있다. 또한 1초 간격으로 조밀하게 모니터링을 하게 되면 0.37%의 CPU 점유율이 증가하게 되었다.

이번 실험에서는 가상화 기반 모니터링 시스템에서 제공하는 API를 이용하여 응용 시스템을 구현하였고 성능을 측정해본 결과 적은 오버헤드로 시스템이 운영될 수 있음을 확인하였다. 따라서 논문에서 제안하고 있는 가상화 기반 모니터링 시스템이 충분한 검증을 거친다면 성능적인 측면에서는 기존의 커널 기반 IDS를 대체할 수도 있을 것으로 예상할 수 있다.




(그림 10) 웹서버와 모니터링 시스템을 운영했을 때의 CPU 사용률 측정

## 6. 결론 및 향후 연구

본 논문에서 제안된 모니터링 시스템은 가상머신의 메모리, 레지스터, GDT, IDT, 시스템 콜 테이블에 접근할 수 있도록 API를 제공하여 가상머신을 세밀하게 모니터링을 할 수 있었다. 또한 본 논문에서 구현된 사용자 API를 사용하여 커널 객체의 무결성을 모니터링 할 수 있는 응용 시스템을 구현하였다. 최종적으로 구현된 응용시스템의 성능을 측정한 결과 모니터링으로 인한 부하가 1초 주기로 모니터링을 하더라도 0.37% 정도의 CPU 점유율만을 차지하였다. 따라서 기존 보안 시스템에 적용될 가능성이 높은 것으로 예상할 수 있었다.

제안하는 가상화 기반 모니터링 시스템의 이점은 다음과 같이 정리할 수 있다. 커널 객체에 세밀한 모니터링이 가능하므로 효율적인 시스템 자원관리를 위하여 활용할 수 있을 것이다. 특히 운영체제에 의존적인 기존 모니터링 시스템과는 달리 운영체제의 제약을 받지 않고도 메모리에 대한 모든 접근이 가능하다. 또한 기존 커널 계층에서는 가상화 계층 자체를 인지하거나 접근이 불가능하기 때문에 루트킷과 같은 커널 레벨의 공격 도구로부터 공격이나 우회 시도들을 무력화 시킬 수 있다.

향후 과제로는 모니터링 시스템이 지원하는 Guest 운영체제를 리눅스 외에 Windows를 지원하게 된다면 활용도가 큰 모니터링 라이브러리로서 발전할 수 있을 것이다. 그리고 기존 보안 시스템에서 사용된 아이디어나 솔루션을 가상화 기반으로 포팅 하였을 경우 고려해야 할 설계상의 구조나 장, 단점에 대해 연구가 진행 되어야 할 것이다.

## 참 고 문 헌

[1] K. Avi, et al, "kvm: the Linux Virtual Machine Monitor", Proceedings of the Linux Symposium, Jun., 2007.

[2] M. Tim Jones, "Discover the Linux Kernel Virtual Machine", IBM Developerworks, May, 2008.

[3] http://www.linux-kvm.org

[4] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot-a Coprocessor-based Kernel Runtime Integrity Monitor", Proceedings of the 13th USENIX Security Symposium, Aug., 2004.

[5] D.P. Bovet and M.Cesati. "Understanding the Linux Kernel", O'Reilly & Associates, Inc., 3rd edition, 2005.

[6] T. Garfinkel and M. Rosenblum. "A Virtual Machine Introspection based Architecture for Intrusion Detection." Proceedings of the 2003 Network and Distributed System Symposium, 2003.

[7] David Chisnall "The Definitive Guide to the Xen Hypervisor" Prentice Hall

[8] Barham P.,et al. "Xen and Art of Virtualization.", Proceedings of the 19th ACM symposium on Operating systems principles, Oct., 2003.

[9] P. Bryan D, P, Martim D, L, Wenke, "Secure and Flexible Monitoring of Virtual Machines", Computer Security Applications Conference, Dec., 2007.

[10] Yaozu, D et al., "Extending Xen with Virtualization Technology" Intel Technology Journal, volume 10, Issue3, 2006.

[11] AMD Virtualization Technology (AMD-V). http://www.amd.com/us/products/technologies/ virtualization/Pages/amd-v.aspx.

[12] Intel Corporation, "Intel 64 and IA-32 Architectures Software Developer's Manual Vol.3 System Programming Guide," http://www.intel.com, 2006.

[13] Koichi Onoue, Yoshihiro Oyama, Akinori Yonezawa, "Control of System Calls from Outside of Virtual Machines", Proceedings of the 2008 ACM symposium on Applied computing, Mar., 2008.

남 현 우

e-mail : hw.nam@koreasmartcard.com
2007년 건국대학교 컴퓨터공학부(학사)
2009년 건국대학교 컴퓨터공학부(석사)
2009년~2010년 한국과학기술연구원 위촉 연구원
2011년~현 재 (주)한국스마트카드 대리
관심분야 : 임베디드 시스템, 가상화 시스템, RFID 등

박 능 수

e-mail : neungsoo@konkuk.ac.kr
1991년 연세대학교 전기공학과(학사)
1993년 연세대학교 전기공학과(석사)
2002년 미국 University of Southern California, 전기공학과 (컴퓨터공학)(공학박사)
2002년~2003년 삼성전자 책임연구원
2003년~2007년 건국대학교 컴퓨터공학부 조교수
2007년~현 재 건국대학교 컴퓨터공학부 부교수
관심분야 : 컴퓨터구조, 임베디드시스템, 병렬컴퓨팅, 컴퓨터보안, 멀티미디어 컴퓨팅 등