공학석사 학위논문

커널 무결성 모니터링을 위한 하드웨어 설계

Hardware Architecture for Kernel Integrity Monitoring

2013년 2월

서울대학교 대학원

전기컴퓨터 공학부

이 지 훈

초 록

반도체 공정의 발달로 인해 고성능의 소형 CPU를 사용한 모바일 기기들이 등장하고 있다. 이를 기반으로 기존의 폐쇄적이던 시스템이 개방되면서 컴퓨터를 대상으로 하던 악성코드들이 모바일 기기로 넘어오고 있다. 이를 막기 위해 다양한 방법들이 연구되었으며 그 중에서 프로그램의 무결성(Integrity)을 검사하여 악성코드를 찾아내는 방법이 있다. 소프트웨어를 기반으로 하는 무결성 검사는 새로운 공격방법이 나타나면 무력해지고 고수준의 루트킷(Rootkit)을 사용하게 된다면 회피가 가능해 이를 해결하기 위해서 하드웨어나 하이퍼바이저(Hypervisor)를 사용하는 방법들이 도입되었다. 하드웨어나 하이퍼바이저를 사용한 방법들은 소프트웨어가 가지고 있던 많은 단점들이 해결되었지만 두 방법도 한계점을 가지고 있다. 하드웨어를 기반으로 하는 방법은 외부에 독립된 시스템이 필요하며 메모리 내용을 복사해 분석하는 방법으로 성능하락의 단점이 있고 하이퍼바이저를 기반으로 하는 경우는 모바일 기기에 적용하기는 하드웨어보다 간단하지만 성능손실이 발생한다는 단점이 있다. 또한 하이퍼바이저를 사용하는 방법도 커널(Kernel) 아래 소프트웨어 층을 추가하는 형태이기 때문에 취약점이 보고되고 있는 상황이다. 이러한 문제점을 해결하기 위해서 본 논문에서는 하드웨어를 기반으로 하는 커널 무결성 검증 시스템을 제안한다. 제안하는 시스템은 감시대상의 버스를 스누핑(Snooping)하는 방식으로 구현이 되었으며 기존의 연구에 비해 성능감소가 없이 더 높은 감지율을 보인다는 것을 실험을 통해 보이도록 하겠다.

주요어 : 리눅스, 커널, 무결성, 보안, 하드웨어

목 차

초 록 i

목 차 ii

제 1 장 서론 1

제 2 장 관련 연구 5

제 3 장 배경이론 7

제 1 절 Immutable Region of Linux 7

제 2 절 AMBA 2.0 9

제 4 장 가정 및 공격모델 11

제 5 장 Hardware Prototype Design 13

제 1 절 Host System 15

제 2 절 Snapshot-based System 16

제 3 절 Snoop-based System 19

제 6 장 실험결과 22

제 1 절 성능비교 22

제 2 절 Transient Attack 에 대한 감지 비교 25

제 3 절 하드웨어 Area / Power 비교 27

제 7 장 Extended Work 29

제 8 장 한계점 및 향후 연구과제 31

제 9 장 결 론 33

참 고 문 헌 34

Abstract 36

표 목차

표 1 성능측정 실험결과 23
표 2 Transient Attack 감지율 25
표 3 Hardware Module Utilization 27
표 4 XPower Analyzer 결과 28

그림 목차

그림 1 Xen 하이퍼바이저 개념도 2
그림 2 AHB 버스구조 9
그림 3 Host System 15
그림 4 Snapshot-based System 16
그림 5 DMA Period 17
그림 6 대용량의 데이터 전송 17
그림 7 Snoop-based System 19
그림 8 Snooper Module 20
그림 9 Snooper 동작 예시 21
그림 10 Copy 24
그림 11 Scale 24
그림 12 Add 24
그림 13 Triad 24
그림 14 Snapshot 주기에 따른 감지율 26
그림 15 Extended System 29

제 1 장 서론

오늘날 반도체공정 발전으로 인해 과거 일반 컴퓨터에서 사용된 CPU의 성능을 뛰어넘는 모바일 CPU가 개발되고 있다. 뛰어난 성능의 모바일용 CPU가 개발됨에 따라서 기존 기기와는 차별화된 모바일 기기의 등장이 이루어지고 있다. 과거 모바일 기기는 낮은 CPU성능으로 인해 폐쇄적이며 단순한 기능만을 제공하는 운영체제를 사용하였지만 최근 모바일 기기들은 강력한 CPU를 바탕으로 일반 컴퓨터에서 사용하는 운영체제와 유사한 환경을 제공하고 있다. 즉, 기본적으로 탑재되는 소프트웨어를 사용하지 않고 일반인이 제작한 소프트웨어를 사용하거나 무선연결을 이용해 인터넷 검색 / 인터넷 뱅킹 등 대부분의 작업이 가능해 졌다. 하지만 기존의 폐쇄적이던 환경이 개방됨에 따라서 보안위협에 노출되는 경우도 증가하고 있다. 일반 컴퓨터를 대상으로 제작되던 악성코드들이 스마트 기기를 대상으로 하는 경우가 늘어나고 있으며 이는 앞으로 더욱 증가할 것으로 예상되고 있다 [1].

늘어나는 악성코드를 방어하기 위해 다양한 백신프로그램들이 나와 있지만 악성코드 제작자들은 이런 방어기반을 회피해 자신들의 목적을 달성하고자 고수준의 공격을 하게 되는데 이때 사용하는 것이 루트킷이다. 루트킷은 사용자가 알지 못하게 자신을 숨기고 커널의 루트 권한을 얻어내는 소프트웨어 모음이다. 루트킷을 사용하게 되면 백도어, 키로깅 같은 기능을 사용할 수 있으며, 프로세스를 숨기거나 장기간의 잠복 기간 뒤에 특정 행동을 하도록 명령을 내릴 수도 있다 [2]. 루트킷은 커널영역에 숨어들어 상위 단계에서 동작하는 백신프로그램에게 잘못된 정보를 전달해 찾을 수 없도록 자신을 숨기기 때문에 신종 루트킷의 경우는 소프트웨어로 방어하기 힘든 부분이 있다.

루트킷의 주 공격대상이 되는 것은 운영체제의 핵심부분인 커널이다. 따라서 보안연구 분야에서는 커널 무결성을 보호하는 방법에 대해서도 다양하게 연구하였다. 그 중 한 가지는 커널 무결성을 보호하는 감시체계의 동작을 커널과 독립된 영역에서 할 수 있도록 하는 방법이며 2가지로 분류를 할 수 있다. 외부 하드웨어를 사용한 방법과 하이퍼바이저(hypervisor)를 사용한 방법이다.

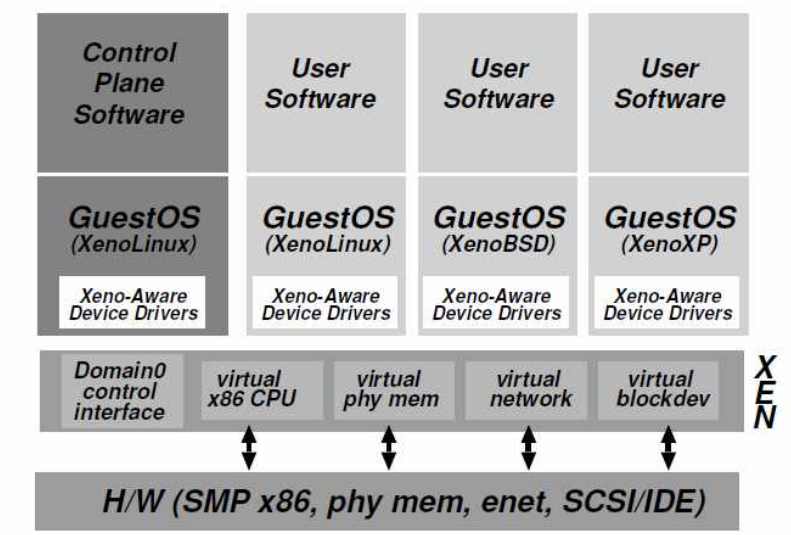

그림 1 Xen 하이퍼바이저 개념도

하이퍼바이저는 다른 말로는 Virtual Machine Monitor(VMM)로 알려져 있는데 주로 하나의 하드웨어 플랫폼에서 여러 개의 운영체제가 자원을 효율적으로 나누어 사용할 수 있도록 관리해 주는 소프트웨어이다 [3]. 그림 1의 경우 대표적인 하이퍼바이저 중 하나인 Xen을 나타낸 것이다 [4]. 하지만 하이퍼바이저도 커널 아래에 소프트웨어 층을 추가하는 형태이고 또한 그 크기가 커지면서 취약점이 나타나고 있는 상황이다 [5, 6]. 이런 단점들을 극복하기 위해서 하이퍼바이저 자체의 무결성을

보호하기 위해 하드웨어의 도움을 받는 연구들이 나타나기 시작했다 [3, 7].

하드웨어로 커널의 무결성을 보호하는 방법들은 대체적으로 시스템의 특정한 상태 스냅샷(snapshot)을 얻어와 분석하는 방법을 사용한다. 대표적으로 Copilot [2]의 경우는 특수한 PCI카드를 이용해 host system의 메모리 내용을 Direct Memory Access(DMA)를 통해 수집, 분석한다. 하이퍼바이저 자체의 무결성을 보호하려고 하였던 HyperSentry [3], HyperCheck [7]도 Intel CPU에서 제공하는 System Management Mode(SMM)를 사용하여 시스템의 상태를 얻어 분석하였다. 하지만 스냅샷을 얻어오는 방식은 메모리나 CPU의 레지스터 값 등을 읽어서 분석하기 때문에 시스템이 멈추거나 메모리 대역폭을 소비해 성능손실이 불가피하다는 단점이 존재한다. 따라서 분석을 연속적으로 할 경우 발생하는 성능손실을 최소화하기 위해서 특정 주기를 가지고 시스템의 상태를 가져오게 되는데 악성 소프트웨어가 시스템의 상태를 검사하는 주기를 알고 공격한다면 감지할 수 없다는 취약점이 존재한다. 이러한 공격을 scrubbing attack이라고 하며 HyperSentry [3]에서 언급하기로 검사 주기를 임의적으로 변경하여 공격자가 예측을 하지 못하게 만들 수 있다고 하였다. 하지만 주기의 예측 없이 임의로 짧은 시간 안에 원하는 목적을 달성하고 침입한 흔적을 지우고 사라지는 transient attack에는 취약한 면을 보인다. 즉, 기존에 방법들이 사용한 특정한 시점의 시스템 상태만을 가지고 이런 공격들을 잡아내기는 상당히 힘들다고 할 수 있다.

본 논문에서는 앞서 언급한 연구들의 단점으로 지적되었던 특정한 시점의 시스템의 상태를 얻어오면서 생기는 성능 손해와 짧은 시간 안에 공격을 하고 빠지는 경우 탐지가 힘들다는 점을 개선하기 위해서 host system의 버스를 스누핑하는 방법에 대해서 제안한다.

제안한 시스템은 host system의 버스를 지나다니는 트래픽(traffic)을 스누핑하기 위한 Snooper와 스누핑한 정보를 분석하는 감시 시스템으로 구성이 되며 Gaisler사에서 제공하는 LEON3 프로세서와 AMBA 2.0 프로토콜을 사용하는 시스템을 FPGA에서 동작 가능하도록 구현하였다. Host system의 운영체제는 LEON3 프로세서에서 동작할 수 있도록 만들어진 Snapgear 임베디드 리눅스를 사용하였다 [8].

또한 과거의 연구들과 비교 실험하기 위해서 스냅샷을 사용하는 감시 시스템을 구현하였다. 실험은 앞에서 언급한 transient attack을 하는 루트킷을 제작해 구동시킨 뒤 총 500번의 공격 중 몇 번의 공격을 감지해 내는지 실험하였으며 이로 인해 유발되는 성능 감소가 얼마만큼 되는지 측정하였다. 실험결과 100ms 주기로 host system의 스냅샷을 가지고 오는 경우에 최대 6.59%의 성능감소가 있었으며 새롭게 제안한 스누핑 방식의 경우는 성능감소가 일어나지 않았다. 또한 스누핑 방식은 transient attack을 100%를 감지하였으며 스냅샷 방식의 경우는 최악의 경우 5.8%만을 감지하였다.

본 논문의 구성은 다음과 같다. 2장에서 기존에 운영체제 커널의 무결성을 보호하기 위한 관련 연구들에 대해서 소개하겠다. 이어서 3장에서는 배경이론에 대해 설명을 하고 4장에서는 본 논문이 가정하는 상황과 공격에 대한 정의를 설명하겠다. 5장에서는 제안된 하드웨어 구조에 대해서 설명하며 6장에서는 기존의 연구와 새롭게 제안된 시스템의 성능 및 실험결과를 보여주도록 하겠다. 7장에서는 감시 영역을 하드웨어적으로 쓰기 잠금 할 수 있는 기능을 추가한 시스템을 소개하며 8장에서 본 논문에서 제안한 시스템의 단점과 향후 연구과제에 대해서 설명한 뒤 9장에서 결론을 맺도록 하겠다.

제 2 장 관련 연구

커널 무결성을 보호하기 위한 관련 연구는 크게 2가지로 분류할 수 있다. 하이퍼바이저를 사용한 방법과 하드웨어를 사용한 방법으로 나눌 수 있다.

먼저 하이퍼바이저를 사용한 방법에 대한 관련 연구를 살펴보면 SecVisor는 하이퍼바이저가 커지면서 나타나는 취약점을 극복하기 위해서 작은 하이퍼바이저를 설계하는 것을 목표로 하였다. Memory Management Unit(MMU)만을 가상화하였으며 이를 이용해 메모리에 대한 접근을 제어해 시스템 보호하는 방법을 사용하였지만 Xen에 비해 큰 성능 저하가 있는 단점을 지니고 있다 [9]. Sung-Min Lee et al. 논문에서는 VMM을 사용해 시스템을 secure / normal domain으로 분리해 보안이 필요한 소프트웨어의 경우는 secure domain에서만 실행이 가능하도록 하였다 [10]. OSck의 경우는 하이퍼바이저를 사용해 운영체제의 정적영역 보호와 함께 커널의 동적 자료구조도 감시하도록 구현하였다 [11].

하드웨어를 사용한 경우는 하이퍼바이저와 같이 하나의 소프트웨어 층을 추가하는 것이 아닌 독립된 시스템을 구축하였다. Copilot은 특수한 PCI카드를 사용해서 host system의 정적인 커널 영역의 무결성을 검증하였다. 일정한 주기마다 DMA로 미리 지정된 메모리 영역을 읽어서 분석 결과를 외부로 송신하는 방식으로 이루어진다. 분석은 스냅샷을 해시하는 방법을 사용하며 이전에 알려진 해시 값과 다르다면 외부로 전달해 사용자가 판단하도록 하였다 [2]. 하지만 이 방법에는 2가지 문제점이 있다. 첫 번째로 주기적으로 메모리의 내용을 읽어야 하기 때문에 성능의 손실이 불가피하다. 두 번째로 PCI카드를 이용해서는 CPU 레지스터의

값을 참조할 수 없다는 단점이 존재한다. 예를 들어 CR3 레지스터나 IDTR 레지스터에는 page table, Interrupt Descriptor Table(IDT)이 있는 주소가 저장되어 있다. 공격자가 이러한 점을 이용한다면 다음과 같은 공격이 가능하다. 테이블의 내용을 직접 변조하는 것이 아닌 자신이 원하는 다른 장소로 복사하고 변조한 뒤 레지스터의 값을 자신이 복사한 위치로 변경해 공격할 경우 기존의 영역은 변조되기 전의 상태와 같이 때문에 탐지하는 것이 어렵다. 이러한 문제점을 해결하기 위해서 HyperCheck에서는 네트워크 카드를 이용해 Copilot과 똑같은 기능을 하게 만들었으며 추가로 SMM을 사용해 CPU의 레지스터의 값도 알 수 있도록 개선하였다 [7]. HyperSentry는 운영체제만큼이나 커진 하이퍼바이저의 취약점을 극복하기위해서 SMM을 사용해 하이퍼바이저의 무결성을 검사하였다 [3]. Host system의 모든 동작을 멈춘 상태에서 검사를 하였고 앞선 연구들이 일정 주기를 가지고 검사를 했다면 HyperSentry는 임의의 시간에 실행할 수 있도록 만들었다. 하지만 호스트 시스템을 멈추고 검사를 진행해야 하기 때문에 성능이 감소되는 단점이 있다.

제 3 장 배경이론

제 1 절 Immutable Region of Linux

본 논문의 실험에서 무결성을 검증한 리눅스의 immutable region에 대해서 설명하도록 하겠다.

Immutable region이란 운영체제가 부팅이 된 이후로 실행 중에 어떠한 일이 있어도 변경이 되지 않는 영역을 말하며 이를 변조하는 행위는 악성코드에 의한 공격으로 판단할 수 있다. 이러한 영역에 대한 공격은 공격자로 하여금 고수준의 공격을 할 수 있게 만들어 주기 때문에 매우 중요하게 다루어져야 한다. 이 부분에 해당하는 영역은 커널의 코드영역, 시스템 콜 테이블(System Call Table), 인터럽트 테이블(Interrupt Descriptor Table)이며 본 논문에서는 3가지 영역을 immutable region으로 정하고 감시하였다.

먼저 커널 코드영역의 경우는 처음 부팅이 된 이후 변경이 되면 안 되는 대표적인 영역이다. 이 영역은 명령어의 집합으로 동작 중에 변경이 된다면 커널이 처음 컴파일된 것과 다른 방식으로 동작됨을 의미한다. 따라서 이 영역은 읽기전용 지역이며 이 영역에 쓰기연산이 일어난다면 외부에서 공격이 가해졌다고 판단할 수 있다.

두 번째로 시스템 콜 테이블은 운영체제가 동작 중에 사용하는 여러 가지 함수들의 위치를 함수 포인터 형식으로 저장해 놓은 배열이다. 커널은 동작 중에 시스템 콜 요청이 들어왔을 경우 이 테이블을 참조해서 해당 함수를 부르게 된다. 따라서 공격자는 테이블의 함수 포인터 값을 변경해 커널에서 공격자의 코드로 점프를 시킬 수 있다. 공격자의 코드를 수행한 뒤 원래의 시스템 콜을 수행한다면 사용자가 보기에는 함수가

정상적으로 동작하는 것처럼 보이게 된다. 이러한 공격방법을 hijacking 이라고 하는데 공격자가 손쉽게 자신의 코드로 제어 권을 가져올 수 있는 방법 중의 하나이다.

마지막으로 인터럽트 테이블의 경우 인터럽트가 발생할 경우 처리해야 하는 함수로 점프할 수 있도록 주소가 저장되어 있는데 만약 이 값을 변경한다면 위에서 언급한 시스템 콜과 같은 공격이 가능할 것이다. 따라서 이 부분도 처음 부팅이후 값이 변경된다면 공격이 가해졌다고 판단할 수 있다.

제 2 절 AMBA 2.0

본 논문에서 구현한 시스템은 AMBA 2.0 [12]에 기반을 둔다. AMBA 2.0규격은 다음과 같이 3가지로 나눌 수 있다. The Advanced High performance Bus(AHB), The Advanced System Bus(ASB), The Advanced Peripheral Bus(APB). 이 중에서 본 논문에서는 AHB와 APB 만을 사용하였다.

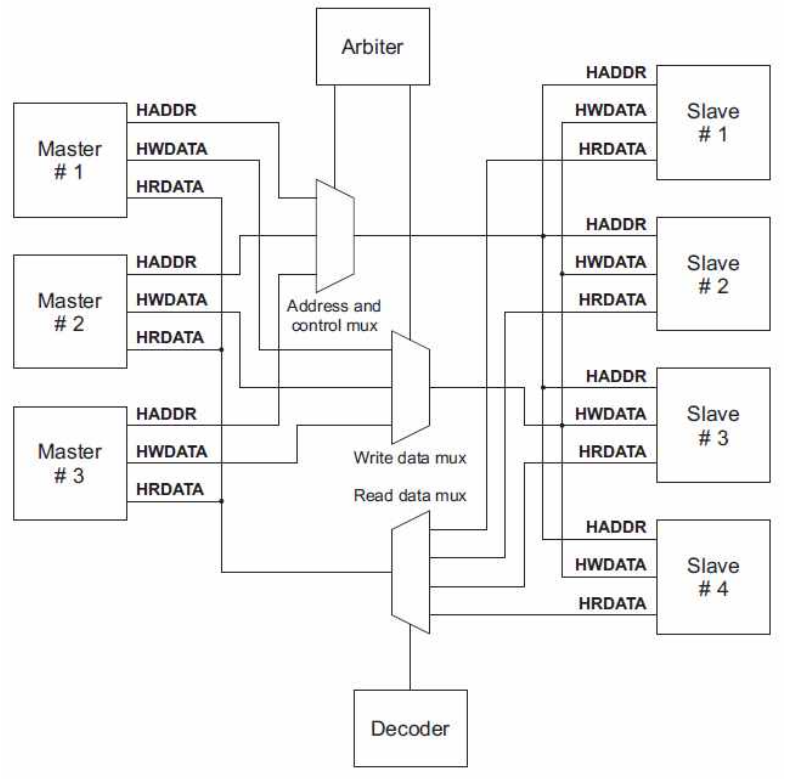

그림 2 AHB 버스구조

그림 2의 경우 3개의 AHB master와 3개의 AHB slave 간의 버스구성을 나타낸 것이다. AHB는 master와 slave로 나뉘며 master는 slave에

읽기, 쓰기명령을 내리고 slave는 자신의 주소영역에 명령이 있을 경우 이에 응답하는 형식으로 구현이 된다. 기본적으로 공유버스(shared bus) 시스템으로 구성이 되기 때문에 한 번에 단 한 개의 master만이 버스를 사용할 수 있으며 이를 통제하는 arbiter가 존재한다. Arbiter는 독자적인 알고리즘을 가지고 있어서 master간의 버스 점유권을 조절하는 역할을 담당한다. 그리고 명령이 내려지는 주소를 해석하여 해당 slave가 응답할 수 있는 신호를 만들어 낸다. Slave에서 응답하는 데이터의 경우에는 decoder가 존재하여 현재 활성화된 slave에서 전송하는 데이터만 선택해서 master로 전송을 하게 된다.

앞에서도 언급했듯이 본 논문에서 사용한 프로토콜은 공유버스 구조이기 때문에 모든 slave는 자신에게 해당하지 않아도 모든 master로부터 오는 명령을 들을 수 있다. 즉, CPU나 다른 master 장비들이 메모리영역에 쓰기 명령을 내리더라도 메모리를 제외한 다른 slave도 해당 신호를 받을 수 있다. 다만 자신이 응답해야 됨을 알리는 slave select 신호가 활성화 되지 않기 때문에 동작 하지 않는 것이다. 즉, 이를 응용해 slave에 들어오는 신호를 복제해 입력으로만 받는 모듈을 추가한다면 버스의 스누핑이 가능하다. 본 논문에서 제안하는 하드웨어 구조를 적용하기에 적합한 형태라고 할 수 있다.

제 4 장 가정 및 공격모델

본 장에서는 실험에서 가정한 공격모델에 대해서 설명하도록 하겠다. 실험에서 사용한 시스템은 이미 루트킷에 감염된 상태로 가정하였다. 따라서 공격자는 host system의 루트권한을 획득한 상태이다. 그리고 공격자는 하드웨어를 물리적으로 공격하지 않는다고 가정한다.

본 논문보다 앞서서 커널의 무결성을 검증하기 위한 방법들은 주기적으로 메모리의 내용을 얻어와 분석을 하였는데 이러한 방법이 취약하다는 것을 보이기 위해 transient attack을 가정하였다.

Transient attack은 일반적으로 시스템을 반영구적으로 변조시키는 persistent attack과는 다른 형태의 공격이다. 시스템을 공격자가 공격을 하는 시점에만 일시적으로 변조하고 목적을 달성한 뒤 원래상태로 다시 복구하는 형식의 공격방식을 말한다. 이런 유형의 공격으로는 리눅스의 소프트웨어 타이머를 이용해서 단시간동안 동작하는 커널 루트킷을 선보인 Soft-Timer Driven Transient Kernel Control Flow Attacks and Defense논문이 있다 [13]. 따라서 본 논문에서는 커널을 임의의 시간에 변조해 시스템 콜 테이블을 수정해 다른 쪽으로 hijacking하는 방식의 trasient attack을 모델로 삼았다. 리눅스에 있는 타이머를 사용해 공격대상이 되는 부분을 일정시간동안 변조한 뒤 다시 원상복구 시키는 과정을 반복적으로 수행하게 만들었다. 이런 공격이 가능한 이유는 앞 장에서 언급하였듯이 리눅스에서 사용되는 시스템 콜 함수들은 포인터로 이루어진 배열에 함수 포인터 형식으로 제공이 되기 때문이다. 따라서 배열의 값만 변조한다면 운영체제는 그 주소가 해당하는 곳으로 점프를 하고 코드를 수행하게 된다. 반복적으로 수행한 이유는 실험의 편의성을 위해서 가정했을 뿐 일반적으로는 trasient attack이 단 한번만 실행될 수도 있

고 여러 번 수행이 될 수도 있다. 만약 여러 번 수행이 된다면 스냅샷 방식으로 찾을 수 있을 것이나 단 한 번의 공격만을 하거나 혹은 아주 드물게 공격을 한다면 매우 찾아내기 힘들 것이다. 따라서 여러 번 반복적으로 수행을 하나 전체가 한 번의 공격이 아닌 각각의 공격을 다른 것으로 가정해 그 중에 몇 번이나 찾아낼 수 있는지를 실험결과로 제시하도록 하겠다.

제 5 장 Hardware Prototype Design

기존에 제시된 하드웨어, 하이퍼바이저를 이용한 방법들의 단점을 극복하고자 본 논문에서는 host system의 버스를 스누핑하여 커널의 무결성을 검증하는 하드웨어를 설계하였다. 기존의 연구와 비교하기 위해서 host system의 스냅샷을 얻어서 분석하는 시스템도 설계하였다. 스누핑과 스냅샷 두 방법 모두 감시 대상인 host system은 동일한 구성을 가지고 있으며 운영체제의 커널 또한 동일하게 설정하였다.

본 논문에서 제안된 시스템은 기존의 연구들이 사용한 스냅샷 방식을 사용하지 않기 때문에 host system의 정보를 전적으로 스누핑하는 버스의 트래픽에 의존해야 한다. 따라서 스누핑을 효율적으로 하기위해서 시스템의 디자인은 다음과 같은 것을 고려해야 한다.

어느 위치에서 버스 트래픽을 스누핑 할 것인지를 결정해야 한다. 2장에서 설명한 AMBA 2.0의 프로토콜에 따르면 master에서 나온 신호는 모든 slave에게 전송이 되고 arbiter에서는 master의 신호를 분석해 해당 slave가 알 수 있는 신호를 만들어 낸다. 따라서 arbiter에서 나온 신호를 복제하여 분석한다면 모든 master에서 모든 slave로 향하는 신호를 알 수 있다. 이로 인해서 모든 master에서 나오는 신호와 slave로 들어가는 신호를 하나씩 분석할 필요성이 사라지게 된다. 하지만 버스의 구조가 본 논문에서 사용한 공유 구조가 아닌 포인트 투 포인트(point to point) 방식이나 멀티레이어(multi-layer) 구조라면 모든 트래픽을 스누핑하기 위해서는 위와 같은 방법을 사용하기가 어렵다. 따라서 주요 부분만을 목표로 잡고 스누핑을 해야 하며 이 경우는 메모리와 버스의 연결부를 스누핑하게 된다면 모든 master로부터 메모리로 보내는 트래픽은 분석할 수 있다.

기존의 스냅샷 방식의 경우는 메모리의 내용을 복사해 분석하기 때문에 시간에 대한 제약이 비교적 적은 편이다. 하지만 버스를 스누핑하는 경우는 매 클럭 사이클마다 트래픽이 발생하므로 Snooper는 최소한 버스의 클럭과 같은 값으로 동작해야 하며 사이클 마다 발생하는 주소와 해당 주소에 대한 쓰기, 읽기 명령을 구분을 해야 한다. 하지만 이것을 소프트웨어로 구현을 하게 된다면 한 클럭당 수십 개의 명령어를 사용해 처리해야 하기 때문에 감시 시스템의 클럭이 host system에 비해서 월등히 높아져야 하는 문제가 발생하게 된다. 따라서 하드웨어에서 매 클럭마다 분석하여 관심이 없는 부분에 대한 정보는 빼고 원하는 정보만을 볼 수 있도록 구성해야 한다.

제 1 절 Host System

Host system은 LEON3 프로세서를 사용하며 SDRAM 및 기타 주변기기로 구성된다. 그림 3은 구현된 host system의 블록 다이어그램이다. LEON3 프로세서는 Gaisler에서 open source로 제공하며 SPARC V8 architecture를 기반으로 설계된 CPU이다 [8]. AMBA 2.0과 호환되는 버스 인터페이스를 제공하며 7단계의 파이프라인과 명령어, 데이터 캐시를 개별로 가지고 있는 하버드 아키텍처이고 L2 캐시는 없다. 메모리는 64MB SDRAM을 사용하였으며 외부와 통신을 위한 UART가 있고 JTAG을 사용해 디버깅 할 수 있다. 사용된 운영체제는 리눅스 커널 2.6 버전을 가지고 제작된 Gaisler사의 Snapgear 임베디드 리눅스를 사용하였다. 실험에서 사용한 prototyping board는 Huins에서 만든 RPS3000을 사용하였으며 FPGA칩은 Virtex5가 탑재되어 있다.

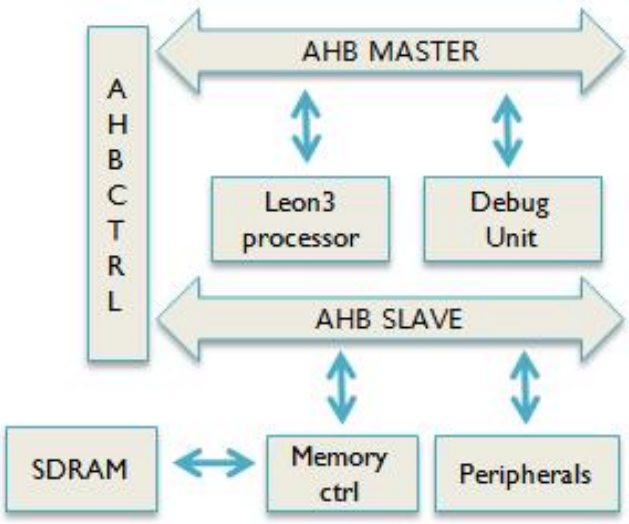

그림 3 Host system

제 2 절 Snapshot-based System

기존의 연구들과 새롭게 제안한 시스템의 성능 비교를 위해서 Copilot [2]과 유사한 방식으로 커널의 무결성을 검사하는 시스템을 구현하였다. Host system의 경우는 5장 1절에서 설명한 것과 같은 시스템을 사용하였으며 메모리의 원하는 부분을 읽어서 다른 시스템으로 전송할 수 있는 모듈이 추가되었다. Copilot의 경우는 특수한 PCI카드가 DMA 요청을 통해서 얻은 메모리의 내용을 전용 통신망을 통해서 외부로 보고를 하였지만 본 논문에서 제안한 시스템의 경우는 외부로 전달하는 것이 아닌 같은 FPGA 칩 안의 감시시스템으로 전송하도록 구현하였다. 즉, 하나의 FPGA칩 안에 host system과 감시시스템이 구현된 SoC형태이다. 그림 4는 최종적으로 완성된 snapshot-based system의 블록 다이어그램을 나타내고 있다. 1절에서 구현한 host system과 같은 2개의 시스템을 사용하였으며 두 시스템은 DMA모듈로만 연결되어 있고 같은 FPGA 칩에서 동작하지만 host system의 경우 감시시스템에 물리적으로 접근할 방법이 없다. 감시시스템의 경우는 host system의 메모리 내용을 읽을 수 있는 독립된 형태로 동작을 한다.

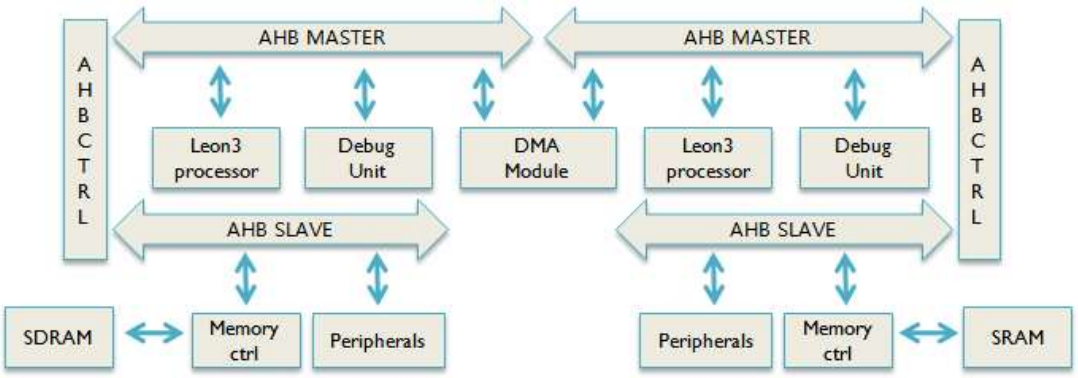

그림 4 Snapshot-based System

Host system 메모리 내용을 복사할 수 있는 DMA모듈은 단 방향으로 설계되었으며 양쪽 시스템에서 master장치로 동작을 한다. DMA 설정 레지스터는 값을 읽어올 메모리 영역을 지정하는 레지스터와 주기적으로 복사를 하거나 몇 번 시행할지 정보를 담고 있는 레지스터로 구성이 된다. 일반적인 DMA의 경우는 운영체제의 요청에 따라서 대용량의 데이터를 CPU를 사용하지 않고 전송할 수 있는 기능을 담당하지만 본 논문에서 구현한 모듈의 경우는 감시시스템에서 레지스터의 값만 정해놓는다면 설정한 주기마다 레지스터에 저장된 주소를 참조해 host system의 메모리를 읽어와 감시시스템의 메모리에 저장하는 방식으로 구현되었다.

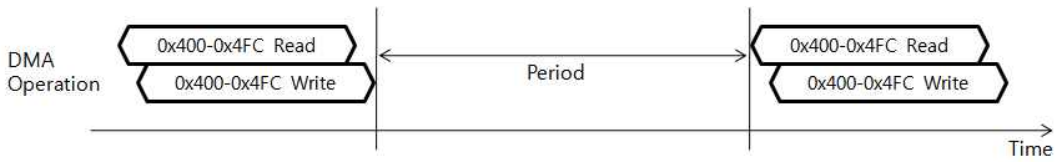

그림 5 DMA Period

그림 5는 DMA의 주기가 설정되었을 경우의 타이밍 다이어그램이다. Host system의 모든 내용의 복사가 끝난 뒤부터 설정한 주기만큼 idle상태로 있다가 다시 동일 영역을 복사하는 방법으로 동작한다.

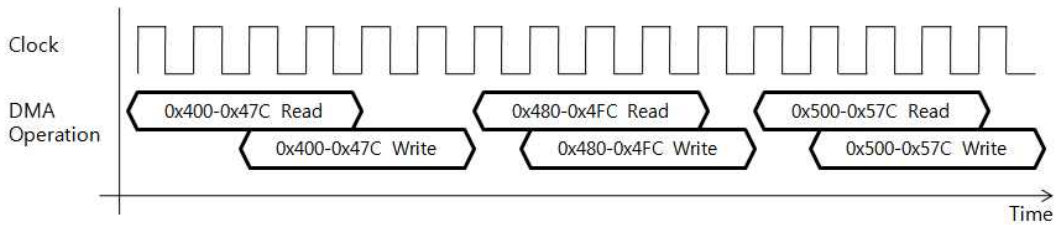

그림 6 대용량의 데이터 전송

DMA는 버스의 점유를 막기 위해서 그림 6과 같이 일정한 단위로 나누어서 전송을 하도록 구성되었다. 복사하려는 영역의 크기가 128바이트 미만일 경우는 한 번의 작업으로 끝나지만 128바이트를 넘을 경우는 이를 128바이트 단위로 나누어 전송한다. Host system에서 버스를 할당받았다 할지라도 감시시스템에서 DMA가 버스를 할당받지 못한다면 대기하는 시간이 길어져 버스를 점유하는 시간이 늘어나게 되고 이로 인해서 host system의 성능은 더 느려지는 결과를 초래한다. 따라서 DMA모듈은 내부에 FIFO를 가지고 있으며 host system에서 읽은 데이터를 저장해 감시시스템 쪽에서 버스를 할당 받기 전에 대기가 필요 없이 연속적으로 읽을 수 있게 구성하였다. 이로 인해 host system의 대기 시간은 최소한으로 줄이고 버스의 독점을 막을 수 있다.

스냅샷한 결과는 해시를 통해서 저장을 하고 이전에 저장한 검증된 값과 비교해 변조가 있었는지 찾는다. 이를 소프트웨어로 구현을 하게 된다면 매우 오랜 시간이 걸려 이를 빠르게 가속해 줄 수 있는 하드웨어 모듈을 추가로 구현하였다. 이 모듈은 본 논문이 목적으로 하는 스냅샷과 스누핑의 비교에서 주요한 부분을 차지하지 않으므로 동작 알고리즘에 대한 설명은 생략하도록 하겠다.

제 3 절 Snoop-based System

기존의 방식이 가지고 있던 단점을 해결하고자 본 논문에서 제안하는 시스템은 버스의 내용을 스누핑 하도록 구현되었다. 2장에서 설명한 AMBA2.0의 규격에서 보게 되면 slave로는 모두 동일한 신호가 전달되고 어떤 slave로 보내는 데이터인지는 각각의 slave가 자신에게 보내는 신호인지 판단해 응답을 하게 된다. 따라서 악성프로그램이 운영체제의 특정영역의 변조를 시도하려고 한다면 메모리로 가는 트래픽 중에 그 변화가 나타날 것이다. 따라서 메모리로 가는 트래픽에서 특정 패턴을 찾아낸다면 host system이 공격받았다는 점을 찾을 수 있다.

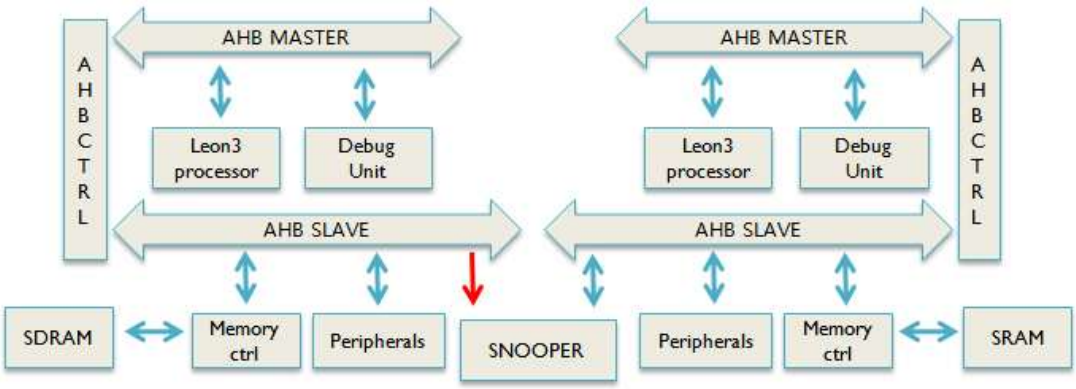

그림 7 Snoop-based System

그림 7은 구현된 snoop-based system의 블록 다이어그램이다. 그림 7에서 볼 수 있듯이 Snooper 모듈은 host system에서 slave로 전송하는 신호만을 복제해서 입력으로 받을 뿐 host system의 slave로 동작하지 않는다. 즉, 5장 2절의 snapshot-based system과 마찬가지로 host system은 Snooper를 비롯한 감시시스템에 물리적으로 접근 할 수 없다.

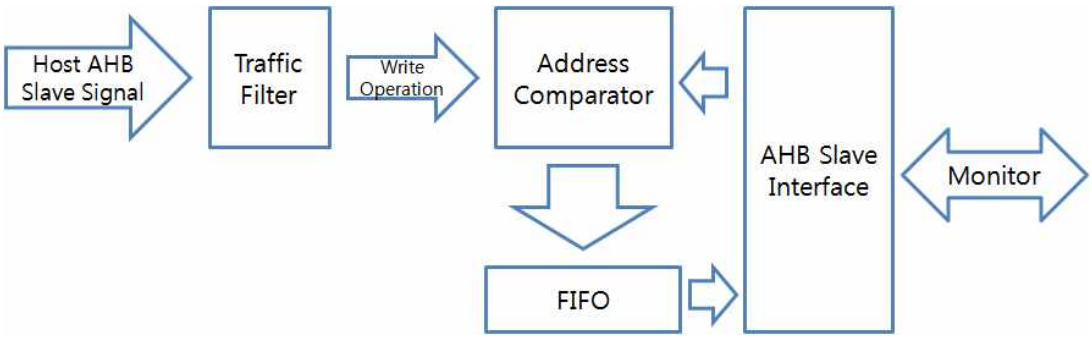

그림 8 Snooper Module

가장 핵심이 되는 Snooper모듈의 구성은 그림 8과 같다. 먼저 host system의 복제된 slave 신호를 분석한다. 쓰기 명령을 내리기 위해서는 단일 전송이나 버스트 전송 관계없이 시작주소와 함께 slave가 쓰기인지 읽기인지 구분할 수 있도록 신호를 보내게 된다. 따라서 이 시스템에서 관심이 있는 경우는 특정 주소에 대한 쓰기 명령이므로 메모리영역에 관한 읽기 명령은 필요가 없는 정보이다. 따라서 Traffic Filter에서는 지금 전송되는 신호가 idle 상태이거나 읽기 명령의 경우는 제외하고 메모리 영역에 관한 쓰기 명령만을 Address Comparator로 전달한다. 이 과정을 통해서 상당한 수의 트래픽을 제거할 수 있다.

Address Comparator는 2개의 입력을 받는데 AHB Slave Interface를 통해서는 감시하고자 하는 영역의 주소를 받으며 이는 실행 도중에도 변경이 가능하다. 다른 하나의 입력은 앞에서 설명한 Traffic Filter를 통과한 트래픽이다. 이 2개의 입력을 기반으로 쓰기명령이 사용자가 감시하고자 하는 영역에 대한 쓰기인지 판단하게 된다. 만약 감시하는 영역이 아닌 다른 영역에 대한 쓰기 명령은 버리고 나머지의 경우만 FIFO에 저장을 한다. FIFO는 단시간에 많은 쓰기명령이 들어오거나 모니터가 실시간으로 읽어 가야하는 점을 완화하기 위한 버퍼의 용도이다.

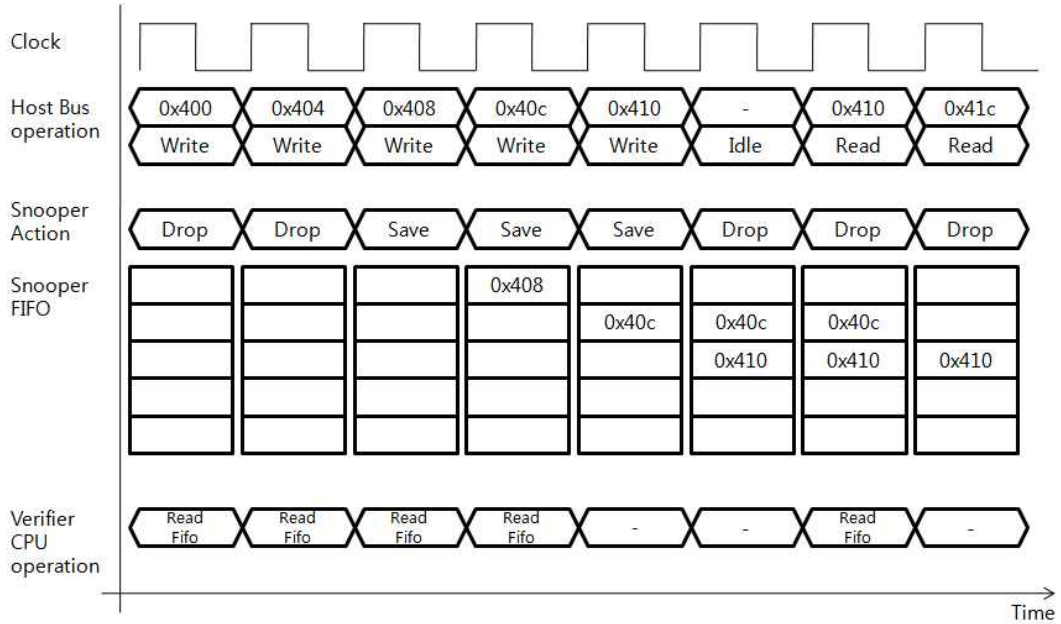

그림 9 Snooper 동작 예시

예를 들어 0x408 - 0x410번지를 감시할 영역으로 지정하였다. 그림 9에서 보는 바와 같이 1, 2번째 클럭의 경우는 쓰기 명령이지만 감시하고 있는 부분이 아니기 때문에 Traffic Filter에서 이를 걸러낸다. 하지만 3번째 클럭에서 0x408번지에 쓰기 명령이 발생하였고 이를 4번째 클럭에서 FIFO 저장한다. 감시 시스템은 FIFO에 항목이 추가되면 이를 읽어서 사용자에게 알리는 행동을 취한다. 이때 추가 사이클이 발생하게 되는데 만약 FIFO가 없다면 4, 5 번째 클럭에서 나타나는 쓰기 명령은 놓치는 상황이 벌어지게 된다. 따라서 감시 시스템이 이를 처리하는 도중 FIFO에 임시로 저장을 하게 된다. 그 다음 다시 FIFO의 내용을 읽어가게 된다. 그림 9에서는 5개의 항목을 저장할 수 있는 것으로 표시되나 실제 구현에 있어서는 16개로 하였으며 이는 RTL 코드 상에서 유연하게 조절이 가능한 값이다.

제 6 장 실험결과

본 장에서는 스냅샷, 스누핑을 사용하는 방법의 성능 비교와 transient attack 감지 실험결과를 보여주도록 하겠다.

실험에 주된 방향은 두 시스템을 사용했을 때 나타나는 성능감소가 얼마나 되는지 비교를 하고 4장에서 제시한 transient attack 공격모델에 대해 감지율이 얼마나 되는지 비교하였다. 스냅샷의 주기를 변경하는 것을 제외하고는 모든 변수를 고정하였으며 실험결과도 주기에 따라 비교분석하였다.

제 1 절 성능비교

성능을 비교하기 위해서 본 논문에서는 STREAM 벤치마크를 [14] 사용하였다. STREAM 벤치마크는 메모리의 대역폭을 측정하여 성능을 평가하게 된다. 기존의 STREAM 벤치마크는 부동소수점 연산을 가지고 있지만 open source로 제공되는 LEON3 프로세서의 경우에는 부동소수점 연산기가 포함되어있지 않다. 따라서 실험을 위해서 STREAM 벤치마크의 소스코드를 수정하여 정수연산으로 변경하고 실험을 하였다. Host system에 STREAM 벤치마크가 포함된 Snapgear 리눅스를 구동시킨 뒤 DMA를 통해 일정 주기로 커널의 내용을 복사하는 방식으로 실험하였다. DMA 모듈은 host system에서 보게 되면 LEON3 프로세서와 같은 master의 입장이기 때문에 AMBA 2.0 특징상 2개의 모듈에 버스의 권한을 나누어 주게 되고 이로 인해서 성능의 감소가 일어나게 된다.

실험에서는 벤치마크를 총 50번 시행하여 그 평균을 측정하였으며 한

번 테스트 시 사용한 배열의 크기는 20만으로 설정하였다. 스냅샷의 주기는 1초, 0.5초 0.25초 0.1초로 정하였으며 스냅샷의 크기도 1Mbtye, 100Kbtye, 10Kbyte로 변경하면서 실험을 하였다. 그 결과 각 벤치마크의 수행시간과 측정된 메모리 대역폭은 아래와 같다.

표 1 성능측정 실험결과 (단위 : MB/s)

		Copy	Scale	Add	Triad
Normal / Snooping		43.6856	37.6130	36.8754	35.8478
10Kbtye	0.1초	43.6604 (-0.057)	37.5880 (-0.066)	36.8496 (-0.070)	35.8308 (-0.047)
	0.25초	43.6701 (-0.035)	37.6042 (-0.023)	36.8707 (-0.013)	35.8402 (-0.021)
	0.5초	43.6789 (-0.015)	37.6047 (-0.022)	36.8694 (-0.016)	35.8489 (0.003)
	1초	43.6844 (-0.003)	37.6077 (-0.014)	36.8721 (-0.009)	35.8531 (-0.015)
100Kbyte	0.1초	43.4288 (-0.588)	37.4057 (-0.551)	36.6496 (-0.612)	35.6317 (-0.602)
	0.25초	43.5683 (-0.269)	37.5422 (-0.188)	36.7814 (-0.254)	35.7638 (-0.234)
	0.5초	43.6280 (-0.132)	37.5691 (-0.117)	36.8289 (-0.126)	35.8116 (-0.101)
	1초	43.6476 (-0.087)	37.5828 (-0.080)	36.8542 (-0.057)	35.8313 (-0.046)
1Mbyte	0.1초	41.8484 (-4.206)	35.8112 (-4.790)	34.8349 (-5.534)	33.4851 (-6.591)
	0.25초	42.5712 (-2.551)	36.8004 (-2.160)	36.0436 (-2.256)	34.9632 (-2.468)
	0.5초	43.1384 (-1.253)	37.2082 (-1.076)	36.5171 (-0.972)	35.3303 (-1.444)
	1초	43.4649 (-0.505)	37.4078 (-0.546)	36.6393 (-0.640)	35.6288 (-0.611)

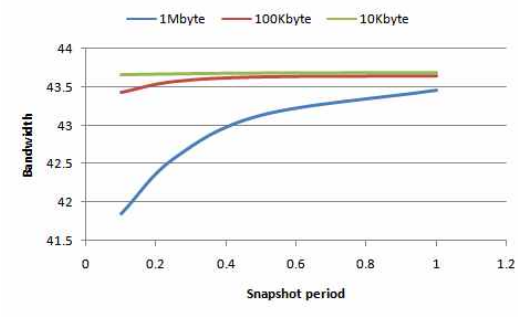

그림 10 Copy

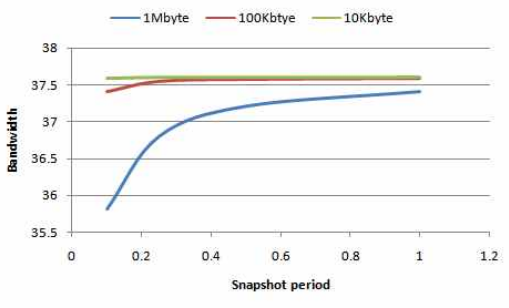

그림 11 Scale

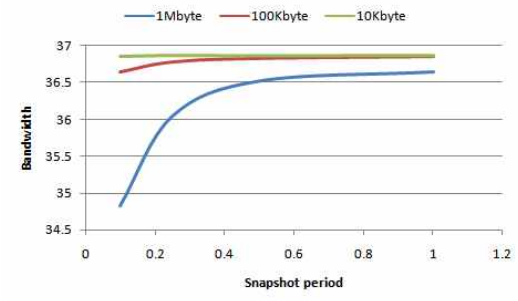

그림 12 Add

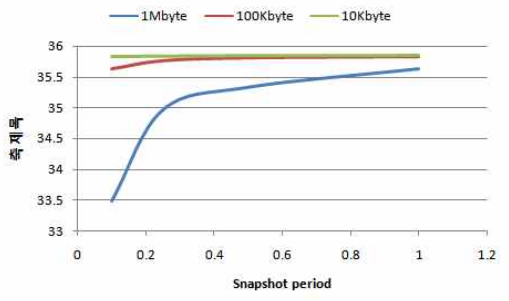

그림 13 Triad

위 결과에서 볼 수 있듯이 스냅샷의 크기가 1Mbyte일 경우 주기가 짧아질수록 성능하락폭이 매우 크게 나타나고 있다. 주기가 100ms일 경우 평균적으로 5.280%만큼이 성능이 감소하였으며 triad의 경우는 6.591%가 감소하는 것을 볼 수 있다. 이에 반해 스누핑의 경우는 host system 버스에 지나다니는 트래픽만을 볼 뿐 실제 버스를 점유하는 일이 없기 때문에 성능이 감소하지 않는다.

본 논문에서 사용한 리눅스의 immutable region의 크기는 약 1Mbyte 정도이다. 따라서 100ms의 주기로 스냅샷을 얻어올 경우 성능손실이 매우 크다는 것을 확인할 수 있다. 또한 주기가 짧아질수록 그 감소 폭은 더 커지는 것을 그래프에서 확인할 수 있으며 주기를 100ms보다 더 짧게 설정한다면 성능감소는 더욱 심할 것이다.

제 2 절 Transient Attack 에 대한 감지 비교

성능측정과 더불어 4장에서 제시한 transient attack을 얼마나 많이 감지할 수 있는지 실험하였다. 실험방법은 총 500번의 공격을 시도하고 그 중에서 몇 번을 감지하였는지를 측정하였다. 이와 동시에 커널이 변조된 상태로 있는 시간을 몇 단계로 나누어 변조된 시간과 스냅샷 주기에 따라 결과가 어떻게 나오는지도 측정하였다. 그 결과는 표 2와 같다.

표 2 Transient Attack 감지율

	100ms	500ms	1000ms	Snooper
50-100	237(47.4%)	53(10.6%)	29(5.8%)	500(100%)
100-500	497(99.4%)	96(19.2%)	46(9.2%)	500(100%)
500-1000	500(100%)	499(99.8%)	252(50.4%)	500(100%)
1000-1000	500(100%)	500(100%)	500(100%)	500(100%)

표 2의 좌측의 수치는 변조시간과 변조주기를 나타낸다. 50-100의 경우는 50ms동안 시스템 콜 테이블의 sys_read, sys_write의 함수 포인터를 변조하고 다시 원상 복귀시킨 뒤 100ms후에 다시 변조하는 것을 나타낸다. 결과에서 알 수 있듯이 커널이 변조된 시간이 스냅샷 주기보다 길어지게 되면 스누핑 방식에 비해 떨어지지 않는 성능을 보이며 스냅샷 주기와 변조시간이 같아지는 곳부터는 조금씩 감지율이 하락하는 경향을 보여주고 있다. 즉, 검사주기를 짧게 한다면 transient attack을 기존의 방법으로도 100% 감지해 낼 수 있는 것은 맞지만 6장 1절의 성능결과를 보게 된다면 성능의 손실이 크다는 결론을 내릴 수 있다.

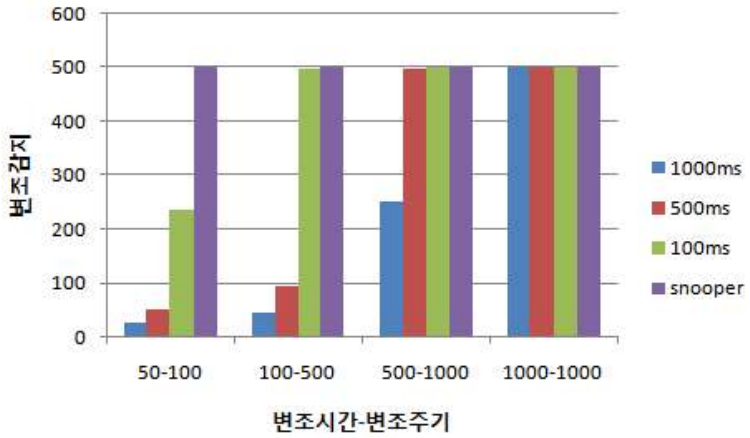

그림 14 스냅샷 주기에 따른 감지율

예를 들어 모니터링을 사용하지 않는 상황과 거의 차이가 없는 1%대의 성능감소를 보이는 500ms주기의 스냅샷 방식을 사용하게 되면 변조시간이 50ms일 경우 10%의 감지율을 보이고 있는데 이는 변조시간에 관계없이 100% 찾아내는 스누핑 방법에 비하면 아주 낮은 수치이다. 결과로 미루어 보면 50ms의 변조시간으로 공격이 이루어지는 경우 스냅샷 방식으로 90%이상 찾아내기 위해서는 50ms 미만의 주기로 스냅샷을 해야 할 것이다. 100ms에서도 약 6% 이상의 성능감소가 있었고 이 주기를 50ms로 줄이게 된다면 12% 이상의 성능감소가 유발될 것이다. 따라서 스냅샷 방식은 더 짧은 변조시간을 가지는 공격에 대해 감지하기 위해서는 주기를 극단적으로 줄여야 하는 문제점이 발생하게 된다.

제 3 절 하드웨어 Area / Power 비교

본 절에서는 FPGA로 구현된 하드웨어의 크기에 대해서 비교분석하도록 하겠다. 본 논문에서 구현한 시스템의 경우는 FPGA에서 구동이 가능하게 구현이 되었기에 실제 칩으로 만들 경우 크기가 얼마나 될지 알 수 없다. 하지만 사용된 레지스터와 LUT의 개수로 대략적인 비교가 가능하다. 두 시스템의 구현결과는 다음과 같다.

표 3 Hardware Module Utilization

	Slices	SilcesRegister	LUTs
Base	7213	7895	14198
Base+DMA+SHA	9323	11191	19145
DMA	840	1514	1956
SHA	825	1720	2402
Base+Snooper	7597	9087	14824
Snooper	509	1183	611

위 표 3에서 Base는 host system과 감시시스템의 합성결과이다. 여기에 추가로 구현한 모듈이 붙을 경우 하드웨어사용량의 변화를 나타내었다. 실제 칩으로 만들 경우 크기는 예상이 불가능하지만 위의 결과를 토대로 분석을 해본다면 Base+Snooper의 경우 Base대비 slices는 5.3%, slieces register는 15%, LUTs는 4.4%늘어난 것을 알 수 있다. One core system에 비하면 하드웨어의 사용량이 2배 가까이 증가하는 면이 있으나 기존의 하드웨어의 연구들도 외부의 프로세서를 사용한다는 측면에서는 추가의 하드웨어를 사용하는 것과 마찬가지이다. 따라서 two core system을 기준으로 비교를 한다면 스냅샷 방식에 비해 스누핑 방식이

하드웨어 오버헤드가 크지 않다는 점을 확인할 수 있다.

표 4 XPower Analyzer 결과

	Total Quiescent Power	Total Dynamic Power	Total Power
단위 (w)	3.50415	0.60892	4.1107

	Hierarchy total Power	Hierarchy CPU Power	Hierarchy Snooper Power	Hierarchy DMA Power
단위 (w)	0.35567	0.07034	0.00585	0.00573

위의 결과는 Xilinx에서 제공해 주는 XPower Analyzer를 통해서 나온 파워소모데이터이다. 이 데이터는 FPGA로 합성을 하면서 사용되는 하드웨어의 양으로 결정이 된다. Datasheet에 나온 정보를 기준으로 단순하게 계산을 하기 때문에 FPGA의 종류에 따라서 변할 수 있는 값이고 FPGA의 공정이나 최적화 여부에 따라서 다르게 결과가 나올 수 있다. 실제 칩으로 만들었을 경우와는 다른 결과가 나올 수 있지만 파워 소모량의 상대적인 비율은 판단할 수 있다.

Total Quiescent Power는 기본적으로 회로가 idle 상태로 있을 시 소비되는 전력이면 Total Dynamic Power는 동작 중에 소비할 수 있는 전력이다. Hierarchy Power는 Total Dynamic Power에서 IO와 Clock Nets에서 소비되는 전력을 뺀 값이다. 위의 값을 토대로 CPU대비 Snooper의 파워소비는 8%이며 전체에서는 1.6%를 차지한다. 기존의 스냅샷 방식과 비교했을 시에도 큰 차이가 없는 것을 볼 수 있다. 따라서 기본 시스템에 비해 약 1.6%증가한 파워를 소비하면서 기존의 방식보다 효율적으로 감지를 해낸다는 것을 알 수 있다.

제 7 장 Extended Work

앞에서 구현한 시스템의 경우는 지정한 영역에 공격이 있다고 판단될 경우 감시시스템에 보고를 하는 것만 수행한다. 하지만 사전에 지정영역에 대한 쓰기 명령을 하지 못하도록 한다면 캐시 안에서 이루어지는 trasnseint attack을 제외한 persistent attack의 경우는 메모리 내용의 변조를 막을 수 있을 것이다. 이를 위해서 메모리 컨트롤러와 버스 사이에 Snooper가 관리할 수 있는 모듈을 구현하였다.

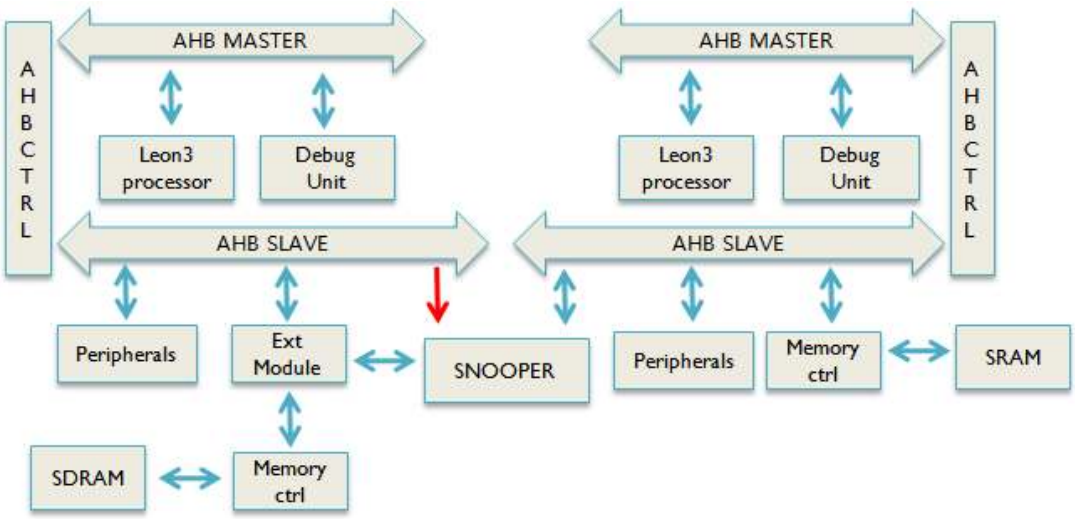

그림 15 Extended System

그림 15의 Ext Module은 사용자가 원할 경우 스누핑 하는 영역에 대한 하드웨어적인 쓰기 잠금 명령을 내릴 수 있다. 버스에서 오는 쓰기 명령을 읽기 명령으로 변환해서 메모리로 전달하며 앞에서 구현한 것과 같이 감시시스템으로도 보고한다. 따라서 이전과 같은 기능을 하면서 적어도 메모리의 내용이 변하는 것을 막을 수 있다.

실험결과 구현한 루트킷이 정상적으로 동작하지 않고 루트킷의 효과가

없음을 확인할 수 있었다. 이 방법의 경우 동적으로 변하는 영역에는 적용할 수 없지만 정적으로 그 내용이 장비의 동작이 중단될 경우까지 유지되어야 하는 경우는 충분히 적용가능하다고 생각한다.

제 8 장 한계점 및 향후 연구과제

이번 장에서는 본 논문에서 제시한 방법에 대한 한계점과 향후 연구과제에 대해 설명하도록 하겠다.

본 논문에서 감시대상으로 지정한 것은 커널의 정적인 영역으로 운영체제 동작 중에 내용이 변하지 않고 일정하게 유지돼야 하는 부분만을 대상으로 하였다. 하지만 루트킷의 공격 중에 정적인 영역을 공격하는 경우는 상당히 적은 수만이 해당할 것이다. 더욱 고수준의 공격은 커널의 동적인 영역을 목표로 삼을 것이며 이 영역은 복잡한 자료구조와 메모리에 할당되는 영역이 변한다는 특징이 있어 본 논문에서 제시한 하드웨어로 감시하기에는 다소 어려운 점이 있다. 이런 종류의 공격을 막기 위해서 휴리스틱한 방법들을 많이 사용하는데 이를 지원하기 위해서 하드웨어의 개선이 필요할 것이다. 또한 Loadable Kernel Module(LKM) 같은 경우는 운영체제의 동작 중에 필요에 따라 사용하는 커널 모듈을 변경할 수 있기 때문에 판단하기 어려운 경우도 있다. 이런 문제를 해결하기 위해서는 운영체제의 control flow를 보호하거나, 운영체제의 면밀한 분석을 통해서 보호하고자 하는 목표를 명확히 해야 할 것이다.

스누핑 방식의 경우도 CPU내부의 값을 알 수 없는 단점이 존재한다. 외부에서 감시만 할 수 있도록 설계가 되었기 때문에 CPU의 레지스터나 캐시의 내용을 변조해 공격하는 경우에는 취약하다는 단점이 있다. 예를 들어 시스템 콜 테이블을 다른 곳으로 복사하고 복사한 테이블을 변조해 사용하게 한다면 기존의 영역은 변화가 없기 때문에 발견하기 어렵다는 단점이 존재한다.

논문에서 구현에 사용한 LEON3 프로세서의 경우는 L1캐시만 가지고 있는 CPU이다. 또한 캐시정책이 write-through이기 때문에 캐시에서 변

경된 내용이 바로 메모리의 트래픽에 나타난다. 하지만 write-back을 사용하거나 L2캐시가 있다면 메모리 트래픽에 바로 나타나지 않을 것이다. 하지만 값을 변조했다가 다시 원상 복구시킨다 하더라도 캐시의 내용이 변경된 것과 같기 때문에 메모리에 쓰기 트래픽이 발생할 것이므로 늦게나마 찾을 수 있을 것이다. 하지만 이를 근본적으로 해결하기 위해서는 캐시시스템을 새롭게 디자인하는 연구가 필요할 것이다.

제 9 장 결 론

반도체 기술의 발전으로 모바일 기기들의 급속한 발전과 더불어 일반적인 컴퓨터 시스템을 대상으로 하던 악성프로그램이 모바일 기기로 넘어오고 있는 상황이다. 과거 일반적인 컴퓨터를 대상으로 악성프로그램의 공격을 막으려고 하는 시도들은 하드웨어를 기반으로 하거나 VMM을 사용하는 방법이었다. 하지만 이 방법들은 외부의 다른 시스템과 연결을 해야 하거나 성능이 감소하는 문제점을 가지고 있어 외부의 다른 시스템과 연결하기 어렵고 성능에 민감한 모바일 기기들은 이런 방법을 사용하기 어려운 점이 있다. 따라서 본 논문에서는 이러한 단점을 해결하고자 하나의 SoC 형태로 성능손실 없이 커널의 무결성을 검증하는 스누핑 기반의 하드웨어 구조를 제시하였다. 커널의 최초 구동 후 종료 시까지 변하지 않는 부분을 immutable region으로 정해 CPU와 메모리 사이에서 발생하는 쓰기 명령 트래픽을 기반으로 공격이 있었는지 판단하는 방법을 제시하였다. 그 결과 기존의 연구에 비해서 성능 감소는 낮고 감지율은 더 높은 시스템을 구성할 수 있다는 것을 보였다.

참 고 문 헌

[1] http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q3-2011.pdf.

[2] N. Petroni, T. Fraser, J. Molina, and W. Arbaugh, "Copilot - a coprocessor-based kernel runtime integrity monitor," Proceedings of the 13th conference on USENIX Security Symposium, Berkeley, CA, USA, 2004.

[3] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky, "Hypersentry: enabling stealthy in-context measurement of hypervisor integrity," Proceedings of the 17th ACM conference on Computer and Communications Security, New York, NY, USA:ACM, 2010.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt and Andrew Warfield, "Xen and the Art of Virtualization," Proceedings of the 19th ACM Symposium on Operating Systems Principles, New York, NY, USA:ACM, 2003.

[5] http://secunia.com/advisories/product/15863.

[6] http://www.cvedetails.com/vulnerabilitylist/vendor_id-6276/XEN.html.

[7] Jiang Wang, Angelos Stavrou, and Anup Ghosh, "HyperCheck: A Hardware-Assisted Integrity Monitor," Proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection, 2010.

[8] http://www.gaisler.com.

[9] Arvind Seshadri, Mark Luk, Ning Qu and Adrian Perrig, “SecVisor: A Tiny hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes,” Proceedings of 21th ACM Symposium on Operating Systems Principles, New York, NY, USA:ACM, 2007.

[10] Sung-Min Lee, Sang-bum Suh, Bokdeuk Jeong and Sangdok Mo, "A Multi-Layer Mandatory Access Control Mechanism for Mobile Devices Based on Virtualization," Proceedings of fifth IEEE Consumer Communications & Networking Conference, 2008.

[11] Owen S. Hofmann, Alan M. Dunn, Sangman Kim, Indrajit Roy and Emmett Witchel, "Ensuring Operating System Kernel Integrity with OSck," Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, New York, NY, USA:ACM, 2011.

[12] http://www.arm.com/products/system-ip/amba/amba-open-specifications.php.

[13] Jinpeng Wei, Bryan D. Payne, Jonathon Giffin and Calton Pu, “Soft-Timer Driven Transient Kernel Control Flow Attacks and Defense,” Proceedings of Annual Computer Security Applications Conference, 2008.

[14] https://www.cs.virginia.edu/stream.

Abstract

Due to the development of the semiconductor manufacturing process, many mobile devices are using high performance and small size CPU. Based on the high performance CPU, mobile devices move to the open environment from closed environment. For this reason many rootkits are targeting the mobile devices. Many researchers in the computer security, introduced many kernel integrity checking method. But software based study has a limitation, if an attacker use a new method of attack or high-level rootkits then they can avoid the software protection. Therefore, researcher adopts hardware based approaches or hypervisor approaches. These approaches solve the many problem that is the limitation of the software approaches but has limitation. Hardware based approaches have to implement external processor and that processor has independent hardware components. But it is hard to adopt that approaches to the mobile system. And adopt the hypervisor approaches are easier than hardware approaches but it cause performance overhead. Therefore, in this paper we introduce a new scheme for integrity check of the kernel. This is a hardware based approach and we snoop the host system's bus to get the illegal access to the immutable region of the kernel. Using this approach, we show the low performance overhead and high detection rate than previous works.

keywords : Linux, Kernel, Integrity, Security, Hardware

student number : 2011-20910