공학석사 학위논문

MMU가 없는 임베디드 시스템에서의 보안 향상을 위한 가상 주소 체계 구현

2018 년 2 월

서울대학교 대학원

전기정보공학부

박 재 성

MMU가 없는 임베디드 시스템에서의 보안 향상을 위한 가상주소 체계 구현

지도교수 백 윤 흥

이 논문을 공학석사 학위논문으로 제출함
2017 년 11 월

서울대학교 대학원
전기정보공학부
박 재 성

박재성의 석사 학위논문을 인준함
2017 년 12 월

위 원 장 문 수 묵 (인)

부위원장 백 윤 흥 (인)

위 원 김 장 우 (인)

국문초록

최근 급증하고 있는 소형 기기들은 비용 절감, 소형화 등의 이유로 작은 메모리 공간을 갖고 있을 뿐 아니라 Memory Management Unit(MMU)가 누락되어 있다. 이와 같은 환경은 공격자가 code pointer 등의 값을 변환하여 쉽게 공격을 수행할 수 있기 때문에 보완이 필요하다. 본 논문은 이와 같은 공격을 방어하기 위한 가상 주소 체계를 구현하는 것을 논한다.

LLVM Compiler를 활용하여 공격에 활용될 수 있다고 판단된 대상에 대해 가상 주소 변환에 필요한 추가적인 instruction을 삽입함으로써 가상 주소 체계를 구현하였다. 또한 메모리에 저장되는 모든 개체가 아니라 공격에 활용될 수 있는 대상에 대해서만 주소 변환을 수행하여 보다 효율성을 높일 수 있었고 실험 결과 가상 주소를 활용하여 코드를 실행함에도 비교적 적은 overhead로 가상 주소가 없을 때와 동일하게 수행할 수 있었다.

주요어 : Virtual Memory System; Pointer Integrity; LLVM Compiler; Safe Stack
학 번 : 2016-20902

목 차

표 목 차

그 림 목 차

[그림 7] 주소 변환에 사용되는 Assembly ····· 12

제 1 장 개요

최근 IoT 기기, 모바일 디바이스 등이 각광을 받아 그 수가 급증함에 따라 해당 제품들의 보안성을 향상시키기 위한 시도도 끊임없이 이루어지고 있다. 이와 같은 기기들은 휴대성, 경제성 등을 강조하기 때문에 제품의 경량화와 저전력 소모에 중점을 두어 보다 단순한 구조를 지니고 있기 때문에 휴대폰이나 workstation 등과 같은 고사양 기기에 존재하는 하드웨어가 누락되어 있기도 하다. 따라서 하드웨어적인 도움을 받는 보완책을 활용할 수 있는 고사양 기기에 비해 간단한 기기들은 보완책을 온전히 활용하기 어렵기 때문에 여전히 해당 취약점으로부터 자유롭다고 보기 어렵다.

하드웨어의 도움을 받아 취약점을 방어하는 보완책의 종류 중 비교적 효과적이면서 overhead가 크지 않은 것은 바로 메모리 공간 배치를 randomize하는 ASLR(Address Space Layout Randomization)일 것이다. 실제 ASLR은 그 효용성을 인정받아 Linux나 Windows같은 운영 체제에 활용되고 있다. 그러나 randomization 기반의 방어법은 넓은 주소 공간을 전제로 한다. MMU가 존재하는 고사양 기기의 경우 가상 주소 공간을 넓게 활용할 수 있기에 ASLR을 차용할 수 있지만 임베디드 시스템과 같이 한정된 메모리를 갖고 MMU도 누락되어 있는 경우 주소 공간에서의 entropy를 높이기 힘들어 사실상 활용이 어렵다고 볼 수 있다.

최근 이와 같은 약점을 이용하여 Broadcom사의 와이파이 모듈에서의 취약점을 활용한 공격이 발표되었다. BroadPwn이라고 명명된 이 공격은 메모리 상의 포인터 값 변조를 활용하는데 공격자의 비교적 적은 개입으로도 자가 복제를 통해 쉽게 다른 기기로 전염될 수 있고 심지어 감염된 기기에서 remote shell execution까지 수행할 수 있다. 취약점이 발견된 모듈은 활용되는 저전력 모델로 대표되는 Cortex R4 코어를 활용하여 펌웨어를 실행하고 이와 같은 AP는 간단한 임베디드 시스템에서 자주

이용된다. 비록 해당 취약점이 패치를 통해 수정되었지만 BroadPwn 공격은 저사양 기기가 상대적으로 고사양 기기에 비해 보안이 취약하다는 점을 시사한다.

본 논문에서는 포인터 주소를 변조하는 방식을 활용한 공격을 방어하기 위해 가상 주소 변환 체계를 논한다. 메모리 상에 저장된 주소 값을 물리 주소 대신 변환 과정을 거친 가상 주소를 사용하여 저장하고 코드 수행 시 다시 원래 물리 주소로 변환하는 과정을 거치게 된다. 따라서 공격자가 악성 코드를 메모리 내부에 배치하고 해당 코드를 가리키는 물리 주소를 메모리 내부의 포인터 값에 저장한다 하더라도 수행 과정 시 가상 주소에서 물리 주소로 변환되는 과정을 거치기 때문에 의도하지 않은 주소를 가리키게 되어 효과적으로 방어를 수행할 수 있다. 그러나 메모리에 저장되는 모든 값에 대해 일괄적으로 주소 변환을 수행하는 것은 많은 시간이 소요되기 때문에 변환을 수행할 적절한 대상을 선정하는 것이 필요하다.

따라서 본 논문에서는 메모리에 저장되는 개체 중 공격에 이용이 될 수 있는 개체를 대상으로 LLVM compiler를 활용한 IR 분석을 통해 적절한 주소 변환 명령어가 삽입될 수 있도록 하였다. 또한 메모리에 접근할 때마다 하드웨어 MMU와 같이 매번 주소 변환 과정을 거치게 되면 실제 코드가 실행될 시 상당한 오버헤드가 발생하기 때문에 한 번 변환이 수행된 가상 주소를 다른 instruction에서 다시 활용하는 방식으로 보다 실행 시간을 줄이는데 기여하였다.

제 2 장 배경

2장에서는 본 논문에서 포인터 주소를 변조함으로써 수행하는 공격의 일종인 BroadPwn 공격에 대해 보다 자세하게 설명하고 그 방어법의 종류 중 한 가지인 ASLR에 대해 서술한다. 또한 본 논문에서 구현 시 활용한 LLVM Compiler에 대해서도 서술한다.

제 1 절 포인터 주소 변조를 통한 공격 및 방어

본 절에서는 1장에서 소개한 BroadPwn 공격에 대해 자세히 설명하도록 한다. 기본적으로 BroadPwn 공격은 와이파이 모듈을 사용하는 기기와 AP(Access Point)가 서로 연결하기 전에 수행하는 process 중 취약점이 존재하는 단계를 활용한다. 기기와 AP는 연결을 수행하기 전 다양한 종류의 request와 response를 주고받는데 그 중 association request라는 과정에서 사용되는 패킷을 변조하여 사용한다.[1] 해당 패킷을 파싱하는 wlc_bss_parse_wme_ie 함수 내부에는 current_wmm_ie라는 버퍼에 패킷 중 일부를 복사하는 과정이 있다. 그런데 해당 버퍼는 메모리 상에 44 byte만큼 할당되어 있기 때문에 패킷이 최대 담을 수 있는 데이터의 크기인 255 byte 보다 작다. 이를 활용하여 current_wmm_ie에 오버플로를 일으켜 원하는 위치의 주소와 공격을 수행할 악성 코드를 저장할 수 있다. 이 때 timer_func라는 함수에서 이용되는 struct 구조체를 변형하여 함수가 시행되면 버퍼 내부에 저장된 주소에 미리 저장한 악성 코드를 덮어쓸 수 있도록 하였다. 그 후 j_restore_cpsr이라는 함수가 불리게 되는데 해당 함수는 RAM 상의 table을 참조하여 restore_cpsr의 주소를 받아 그 주소로 이동하는 함수이다. 해당 테이블의 restore_cpsr의 주소도 마찬가지로 수정하여 악성 코드의 시작점을 가리키게 하면 덮

어 쓰는 과정 이후에 공격이 성공적으로 수행된다.

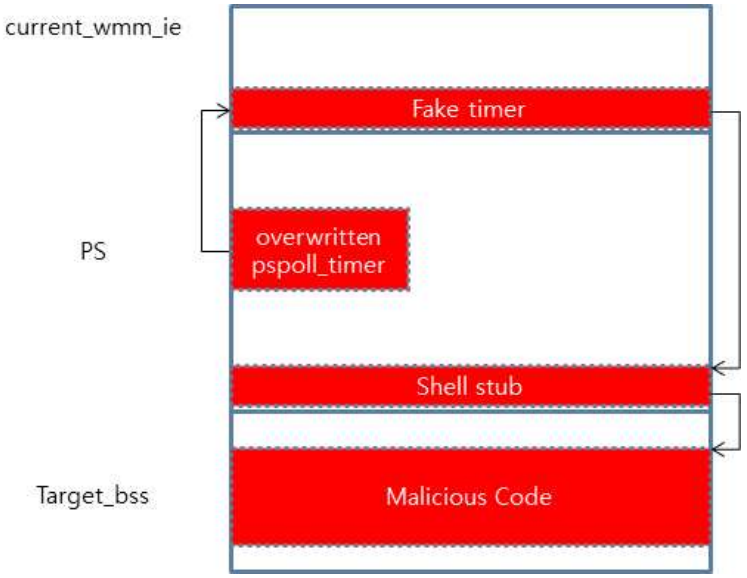

그림 1 BroadPwn 공격의 개요

Broadcom 사의 와이파이 모듈에서 위와 같은 공격이 쉽게 수행되었던 이유는 공격자가 공격을 수행할 코드를 메모리에 저장하고 해당 코드의 시작점을 쉽게 찾아내 이를 가리키도록 포인터 값을 수정할 수 있었기 때문이다. 많은 종류의 공격들이 shell code의 주소라든지 라이브러리의 주소와 같이 특정 주소를 필수적으로 알아야 수행할 수 있다. 만약 메모리 공간 내부에서 개체들의 위치가 변하지 않는다면 공격자는 binary 분석을 통해 원하는 개체의 주소를 알아낼 수 있다. 그러나 ASLR과 같이 process를 수행할 때마다 주소 공간을 무작위하게 변경한다면 공격자가 쉽게 해당 주소를 알아낼 수 없고 혹시 알아낸다 하더라도 다음 process 수행 시 해당 개체의 주소가 변경되어 잘못된 위치를 가리키게 되어 공격이 제대로 작동하지 않는다. 또한 개체를 알아내고자 하는 과정에서 혹시 잘못된 곳을 access하면 fault가 일어나 process가 종료되어 공격이 무위로 돌아가게 된다. 따라서 공격자는 process가 새로 수행될 때마다 다시 개체의 주소를 찾아내야 유효한 공격을 수행할 수 있기 때문에 공격 가능성이 굉장히 낮아진다. 이처럼 ASLR은 무작

위성에 기반을 두고 있기 때문에 효용성을 지니기 위해서는 넓은 주소 공간을 전제로 한다. 즉 MMU를 통해 넓은 가상 주소 공간을 제공할 수 있는 고사양 기기들에는 적합하지만 본 논문에서 다루고자 하는 임베디드 시스템에는 적용하기에 부적합하다. 그러나 ASLR의 핵심 중 하나는 공격에 있어 결정적인 주소 값이 원래의 위치가 아닌 엉뚱한 곳을 가리키게 하여 그 유효성을 잃게 하는 것이다. 본 논문에서는 이를 활용하여 가상 주소 변환 체계를 구현하였고 보다 자세한 내용은 3장에서 소개하도록 하겠다.

제 2 절 LLVM Compiler

LLVM은 University of Illinois에서 학술적인 목적을 지닌 프로젝트로서 시작하였다. 원래 Low Level Virtual Machine의 목적으로 진행되었지만 점점 발전하면서 현재 재사용 가능하고 모듈화 되어 있는 컴파일러 및 툴체인 관련 기술을 가리킨다.[2] 현재 LLVM은 optimizer, code generator, assembler 등 다양한 요소를 포함한 오픈 소스 컴파일러로서 성장하였다. LLVM의 핵심 라이브러리들은 소스와 타겟에 상관없는 optimizer를 제공하며 현재 사용되고 있는 다양한 종류의 CPU들이 실행할 수 있도록 code generation 기능을 수행한다.

LLVM compiler는 크게 세 가지 부분으로 구성되며 각각 frontend, LLVM optimizer, backend가 그에 해당한다. Frontend에서는 언어에 상관없이 소스 코드를 받아와 이를 Intermediate Representation(IR)로 변환한다. IR은 작성한 소스 코드의 종류와 상관없이 동일한 형태를 지니고 있기에 LLVM optimizer에서 이를 활용하여 각종 optimization을 수행할 수 있다. 본 논문에서는 공격에 방어할 수 있도록 IR을 수정할 수 있도록 LLVM compiler에 추가적으로 모듈을 작성한다. 생성된 IR을 바탕으로 backend에서는 각 지정된 타겟에서 실행할 수 있는 결과물을 생성한다.

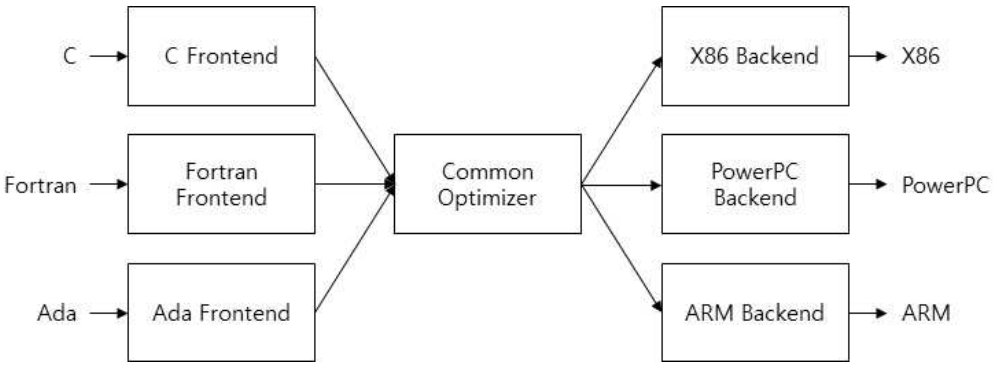

그림 2 LLVM Compiler 구조

제 3 장 가상 주소 시스템 구현

제 1 절 주소 변환의 대상 선정

본 절에서는 메모리상에 저장되는 대상 중 주소 변환을 적용할 대상에 대해 논한다. 앞선 장에서 포인터 주소 공격을 방어하기 위해 가상 주소 변환 시스템의 필요성에 대해 논한 바 있다. 그런데 하드웨어 MMU와 같이 메모리에 저장되는 모든 대상에 대해 주소 변환을 소프트웨어적으로 수행하게 되면 overhead가 굉장히 커지게 된다. 따라서 공격에 활용될 수 있는 개체에 대해서만 주소 변환을 수행하여 보다 효율성을 추구하였다.

이제 메모리에 store나 load 되는 대상 중 주소 변환이 필요 없는 개체에 대해 논해보도록 한다. 그림 3은 소프트웨어적으로 하드웨어 MMU를 구현했을 때 주어진 코드로부터 생성된 IR을 나타낸 것이다. Integer형 변수 a는 물리 주소 0x10000000번지에 저장되어 있지만 소프트웨어 MMU는 이를 미리 가상 주소 0x80000000번지에 저장해 놓았다. 따라서 전역 변수 a에 값 1을 할당하기 위해서는 가상 주소를 다시 물리 주소로 변환하여 레지스터에 넘겨준 후에 할당할 값을 저장해야 한다. 그러나 이와 같이 포인터가 아닌 단순 변수의 경우 본 논문에서 방어하고자 하는 대상이 아니기 때문에 이와 같은 과정은 생략 가능하다.

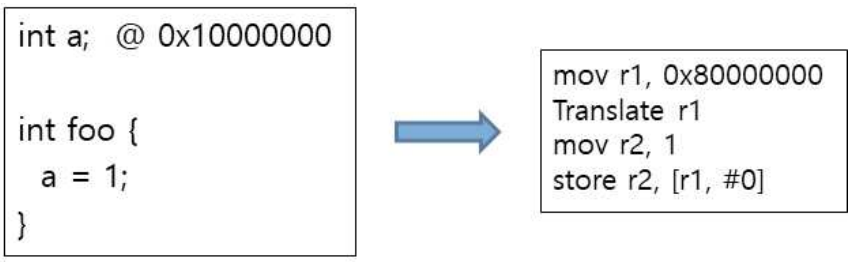

그림 3 Software MMU의 예시

또한 Safe Stack의 적용으로 주소 변환 대상을 줄일 수 있다. Safe Stack은 stack overflow 공격에 방어할 수 있도록 스택을 safe stack과 unsafe stack 두 가지 영역으로 나누는 것이다.[3] Safe stack 영역에는 return address나 register spill, 항상 안전한 방식으로 access가 이루어지는 local variable 등이 존재하고 unsafe stack에는 공격에 활용될 수 있기 때문에 안전하지 않은 변수들이 저장되게 된다. 안전하지 않은 변수들을 안전한 변수들과 따로 저장하고 둘 사이를 격리함으로써 unsafe stack 영역에서 overflow가 나더라도 safe stack 내부의 개체들은 안전하게 보호할 수 있다. Safe stack 내부에 존재하는 개체들은 정적 분석을 통해 안전한 지를 판단하여 컴파일 단계에서 미리 분류하기 때문에 runtime 시 overhead가 거의 발생하지 않는다. 따라서 언어 기반으로 접근한 Cyclone[4]이나 CCured[5] 방식에 비해 상대적으로 overhead가 적다. Safe Stack 방식은 현재 LLVM Compiler에서 모듈로써 제공하며 본 논문에서 이를 가상 주소 변환 체계에 적용함으로써 safe stack에 존재하는 개체에 대해서는 주소 변환을 수행하지 않는다.

제 2 절 가상 주소와 물리 주소간 변환

본 논문에서 활용한 Arduino Due Board의 스펙에 따르면 코드가 동작 시 활용 가능한 메모리 영역은 크게 두 가지로 나눌 수 있으며 각각 Flash Memory와 SRAM에 해당한다. Flash Memory 영역은 코드와 Read only 데이터가 저장되는 영역이고 SRAM의 경우 Read 및 Write가 모두 가능한 영역이다. Flash Memory 영역은 주소 상으로 0x80000번지부터 0x100000번지까지 512KB에 해당하며 SRAM의 경우 0x20070000번지부터 0x20088000번지까지 96KB에 해당한다. 따라서 본 논문에서는 해당하는 영역의 주소에 대해서만 가상 주소 및 물리 주소간 변환을 수행했으며 그 변환 과정은 다음과 같다.

우선 물리 주소에서 가상 주소로 변환하는 경우 그림 4와 같이 수행하였다. 물리 주소를 크게 세 부분으로 나누어 각각 12bit, 9bit, 10bit의 주소를 index로 삼아 변환에 활용하였다. 우선 Flash Memory와 SRAM의 30번째 bit가 각각 0, 1임을 이용하여 서로 다른 변환 테이블(Flash_table, SRAM_table)을 가리킬 수 있도록 하였다. 또한 각 테이블 내에서 물리 주소의 하위 11~19bit를 index 삼아 가상 주소의 상위 21bit 값을 찾아낸 후 물리 주소의 하위 10bit 값과 합쳐서 가상 주소로 변환하도록 하였다. Flash_table과 SRAM_table 내부의 entry 값을 조정하게 되면 entry에 해당하는 가상 주소를 시작으로 하위 10bit offset 만큼 이동할 수 있는 page를 만들 수 있다.

만약 각 table 내부의 entry 값을 임의로 할당하게 되면 가상 메모리 공간 내부에 1KB 단위의 페이지를 random하게 배치할 수 있다. 이와 같은 방식을 활용하면 ASLR과 비슷하게 가상 메모리 공간을 구성하게 되므로 공격에 활용될 수 있는 대상을 찾기 어렵게 만들뿐 아니라 공격자의 의도대로 가리켜야 할 주소가 잘못된 곳을 가리키게 되므로 공격이 제대로 수행될 수 없게 된다. 만약 공격자가 가상 주소를 물리 주소로 변환하지 않고 그대로 사용한다면 공격 코드가 수행되면서 아예 다른 메모리 영역에 액세스 하게 되어 Fault가 발생하기 때문에 정상적으로 실

행이 불가능하다.

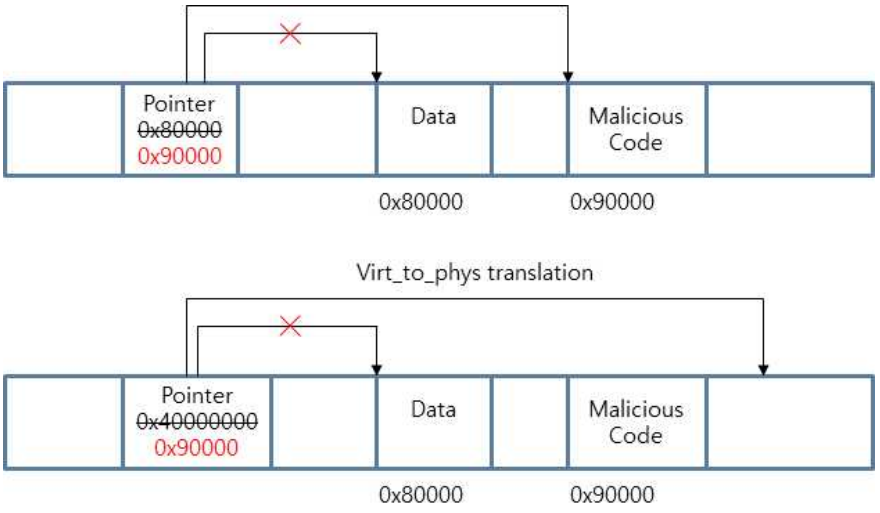

그림 4 가상 주소 활용으로 포인터 주소 공격 무효화

그림 4는 가상 주소 체계를 적용했을 때 포인터 주소 변경을 통한 공격이 기존 메모리 공간에서 수행되는 것과 비교하여 나타낸 것이다. 기존 메모리 공간의 경우 공격자가 포인터 내부에 저장되어 있는 주소를 변경하여 악성 코드가 저장되어 있는 주소를 가리키게 할 수 있다. 그러나 가상 주소 체계를 도입하면 공격자가 악성 코드가 저장되어 있는 주소로 포인터 값을 변경한다 하더라도 코드 수행 시 가상 주소를 물리 주소로 변환하는 과정을 거치기 때문에 엉뚱한 곳으로 가리키게 되어 공격이 정상적으로 수행될 수 없게 된다.

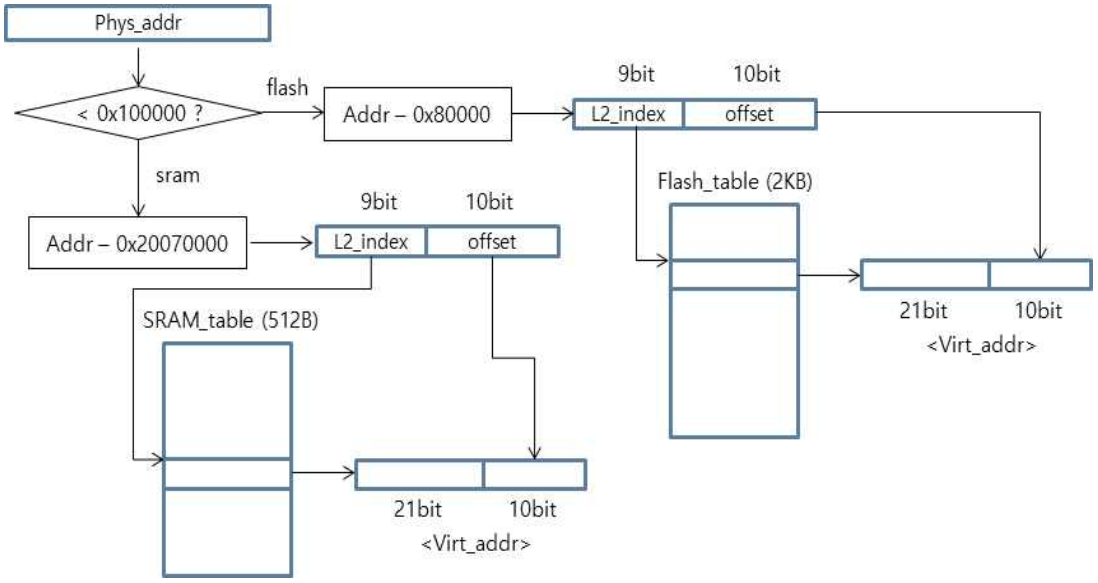

그림 5 물리 주소에서 가상 주소로 변환

가상 주소의 경우 그림 5와 같은 과정을 통해 물리 주소로 다시 변환하게 된다. 가상 주소는 물리 주소 변환 시와 마찬가지로 세 부분으로 나누어 변환을 진행한다. 20~31bit는 L1 테이블에서의 index, 11~19bit는 L2 테이블에서의 index이다. L1 테이블의 항목은 L2 테이블의 주소를 가리키게 되고, 가상 주소의 하위 10bit는 물리 주소에서 가상 주소로 변환할 때와 같이 그대로 사용하여 L2 테이블의 entry와 합쳐 물리 주소로 변환하게 된다. 그 뿐 아니라 L1 테이블의 경우 적절한 범위의 가상 주소가 입력으로 들어오지 않으면 0x0번지로 변환하여 해당 위치를 참조하게 되면 Fault가 발생하기 때문에 정확한 변환 규칙을 알지 못하면 원하는 위치의 물리 주소의 값을 이용하지 못하게 하였다.

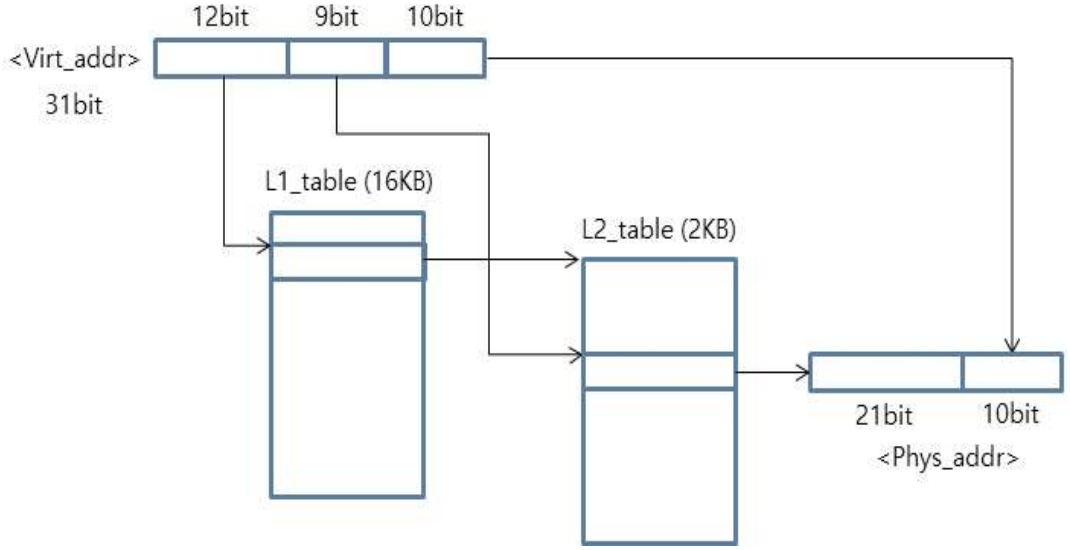

그림 6 가상 주소에서 물리 주소로 변환

위와 같은 물리 주소 및 가상 주소 간 변환은 LLVM IR 상에서 call instruction을 삽입함으로써 수행된다. Call instruction을 삽입하는 위치는 다음 절에서 자세히 소개하도록 한다. IR 상에 삽입된 call instruction은 상응하는 어셈블리로 이루어진 함수를 호출하여 앞서 말한 방법대로 주소 변환을 수행하게 된다. 이에 해당하는 어셈블리 함수는 그림 6과 같다.

```
@llvm.phys.to.virt

mov t00, <pvmtabl_l1>
lsr t01, r_physaddr, #29        ; 0x0xxxxxxx - flash, 0x2xxxxxxx - sram
ldr t02, [t00, t01, lsl #2]     ; t02 = L2 Table Base
lsr t10, r_physaddr, #OFFSET_BITS
bfc t11, t10, #PPN_L2_MASK      ; t11 = L2 Table Index
ldr t20, [t02, t11, lsl #2]     ; t20 = Physical Page #
bfi r_virtaddr, r_physaddr, t20, #OFFSET_MASK


@llvm.virt.to.phys

mov t00, <vpmtabl_l1>
lsr t01, r_virtaddr, #(OFFSET_BITS + VPN_L2_BITS)   ; t01 = L1 Table Index
ldr t02, [t00, t01, lsl #2]                         ; t02 = L2 Table Base
lsr t10, r_virtaddr, #OFFSET_BITS
bfc t11, t10, #VPN_L2_MASK                          ; t11 = L2 Table Index
ldr t20, [t02, t11, lsl #2]                         ; t20 = Physical Page #
bfi r_physaddr, r_virtaddr, t20, #OFFSET_MASK


#OFFSET_BITS = 10, #OFFSET_MASK = (1 << 10) - 1,
#PPN_L2_BITS = 9,  #PPN_L2_MASK = (1 <<  9) - 1
```

그림 7 주소 변환에 사용되는 Assembly

제 3 절 IR 코드 치환

앞선 절에서 다룬 물리 주소와 가상 주소 간 변환이 제대로 수행되어 코드가 동작하기 위해서는 IR 코드 분석을 통해 주소 변환 함수를 적절한 위치에 배치해야 한다. 이를 위해 컴파일 할 코드를 대상으로 기존과 동일한 IR을 생성한 후 이를 분석하여 주소 간 변환 함수를 삽입하였다. 또한 중복된 변환이 필요 없는 코드에 대해서는 앞서 변환된 값으로 치환시킴으로써 overhead를 감소하였다. 및 적절한 코드를 대체하여 가상 주소 시스템을 구현하였다. 본 논문에서 주소 변환을 필요로 하는 IR 구성 요소는 크게 세 가지이며 각각 load instruction, store instruction, global variable이 해당된다. 본 절에서는 각 요소마다 적용한 알고리즘에 대해 소개하도록 한다.

우선 external global과 같이 외부에서 access하고 사용하는 변수를 load 하는 경우 변환 과정을 거치지 않은 값이 저장되어 있으므로 변환 함수를 삽입 시 코드가 제대로 동작하지 못한다. 따라서 해당 global을 사용하는 instruction들을 IR 상에서 찾아내어 변환 대상으로부터 제외시켜야 한다. 이와 같은 과정은 use list를 활용하여 external global을 사용하는 store 및 load instruction을 모두 체크하여 저장한 후 추후 IR 상에서 instruction을 탐색하는 동안 해당 external global 변수를 사용하는 load나 store instruction을 만날 경우 변환 함수를 삽입하지 않도록 예외처리 해야 한다. 또한 본 논문에서는 방어하는 대상이 메모리상에 저장되는 포인터이기 때문에 load 대상이 포인터가 아닌 경우에도 예외 처리를 통해 변환 함수를 삽입하지 않도록 한다.

Load 하는 대상이 앞서 논한 제외 대상에 해당되지 않아 주소 변환 함수가 필요하다고 판단되면 해당 instruction 직후에 가상 주소에서 물리 주소로 변환하는 함수를 삽입한다. 또한 load instruction을 통해 불리는 값을 이용하는 instruction들을 앞서 external global에서와 마찬가지로 use list를 활용하여 찾아내고 변환 함수를 통해 변환된 가상 주소의 값으로 치환하여 불필요한 반복 과정을 생략할 수 있도록 하였다.

Store instruction의 경우 load instruction과 비슷한 알고리즘으로 수행된다. External global의 경우 변환 과정을 생략하고 해당 variable을 사용하는 instruction에 대해서도 예외 처리 과정을 거친다. 다만 load instruction과 다른 점은 추가하는 변환 함수의 종류와 그 위치에 있다. store instruction의 경우 물리 주소를 가상 주소로 변환하는 함수를 store instruction 직전에 치환함으로써 메모리상에 가상 주소가 저장되어 공격자가 원하는 곳으로 포인터를 가리키게 하는 공격이 어렵도록 하였다. 또한 load instruction의 경우와 동일하게 가상 주소 값을 이용하는 instruction을 use list를 통해 찾아내어 해당 값으로 치환함으로써 코드 실행에 있어 효율성을 추구하였다.

마지막으로 설명할 IR의 구성 요소는 바로 global variable이다. 이는 external로 선언되지 않은 global variable에 대해 수행되는 알고리즘인데 IR 내부의 instruction들 중 해당 global을 load하는 instruction들 때문에 필요한 과정이다. 보통 특정 값을 store하거나 load하는 것은 pair로 이루어져 있지만 전역으로 선언되고 초기 값이 존재하는 변수가 코드 내부에서 선언되는 경우 IR 상에서 store instruction으로 저장되는 것이 아니라 변수로서 IR에 존재하게 된다. 따라서 포인터 형태의 global인 경우 코드 실행 전에 미리 변환을 하여 메모리상에 저장해 놓아야 추후 해당 variable을 load하는 instruction이 수행 될 시 제대로 다시 물리 주소로 변환되는 과정을 거칠 수 있다. 해당 과정은 global variable을 load, load한 값을 가상 주소로 변환하는 함수 삽입, 변환된 값을 다시 global variable 값에 저장 하는 세 가지 instruction으로 수행할 수 있다. 위의 세 instruction을 IR의 basic block들 중 entry block에 삽입함으로써 코드가 오류 없이 제대로 실행될 수 있도록 한다.

제 4 장 실험 결과

본 논문에서는 공격으로부터 보호하고자 하는 대상에 부합하도록 MMU가 존재하지 않고 작은 메모리 공간을 지닌 Arduino Due Board를 타겟으로 선정하고 실험을 진행하였다. Arduino Due Board의 스펙은 표 1과 같다.

Experimental Environment	
Microcontroller	Atmel SAM3X8E ARM Cortex-M3 CPU
Flash Memory	512 KB
SRAM	96 KB(two banks: 64 KB and 32 KB)
Clock speed	84 MHz

표 1 Arduino Due Board Spec

가상 주소 변환은 주어진 코드를 컴파일 하는 과정에서 필요한 코드가 삽입되고 이를 구현하기 위해 LLVM compiler 내부에 module을 작성하였다. 이를 이용하여 벤치마크 프로그램을 컴파일하여 Arduino Due Board 상에서 실행시켜 원래 프로그램과 걸리는 시간을 비교하였다. 또한 컴파일 하는 과정에서 얻을 수 있는 IR을 분석하여 주소 변환 대상에 대해 제대로 변환이 수행되는 지 확인하였다.

실험에 사용한 벤치마크 프로그램은 Mibench로서 임베디드 시스템에 적합한 벤치마크를 수행할 수 있다.[6] 실험 결과는 표 2와 같다.

Program	Original(μs)	Va-trans(μs)	Overhead(%)
basic_math	5379109	5457927	1.47
qsort	61429	62227	1.30
dijkstra	1163765	1628058	39.9
patricia	47002	49119	4.50
stringsearch	1401	1442	2.93
FFT	2217779	2362224	6.51

표 2 벤치마크 결과

실험 결과 대부분의 벤치마크에서 상당히 작은 overhead를 보여주었다. 그러나 dijkstra 벤치마크의 경우 다른 벤치마크에 비해 상당히 높은 overhead를 보였는데 IR을 분석한 결과 벤치마크 실행 중 높은 비중을 차지하는 부분이 바로 loop이었다. 해당 Loop 내부에서는 주소를 변환하는 instruction의 비중이 높아 실행 과정에서 overhead가 증가한 것으로 보인다. 또한 Dijkstra 벤치마크의 소스 코드를 분석한 결과 해당 loop에서 linked list를 따라가면서 access하기 때문에 이와 같은 결과가 나온 것으로 보인다. 그러나 이와 같은 overhead도 software MMU와 같이 모든 메모리에 저장되는 대상에 대해 주소 변환을 수행하는 것에 비해서 상당히 낮은 편이다.[7]

제 5 장 결론

본 논문에서는 MMU가 존재하지 않는 저사양 임베디드 시스템에서의 가상 주소 체계를 구현하였다. 이와 같은 환경에서 이뤄지는 많은 종류의 공격에서는 코드 포인터 주소 변경을 통해 포인터가 공격자가 원하는 곳을 가리키게 하는 것이 핵심 원리로써 사용된다. 따라서 이와 같은 공격들을 방어하기 위해 메모리에 저장되는 대상 중 공격에 이용 가능한 대상만을 선별하여 주소 변환을 수행하고, safe stack을 활용하여 효율성과 안정성을 더욱 높였다. 간단하지만 효율적인 주소 체계 구현을 함으로써 대부분의 벤치마크에서 작은 성능 저하만이 발생하였다.

또한 주소 변환 과정에서 활용되는 테이블의 내부 값들을 변경하여 가상 주소 상에서 page 단위의 randomization을 수행할 수 있어 보다 공격을 효과적으로 방어할 수 있다. 그러나 본 논문에서 상정한 공격 방식인 포인터 주소 변경을 통한 공격이 아닌 다른 취약점을 활용한 공격에 대해서는 여전히 취약하므로 이에 대한 보완이 필요하다.

참 고 문 헌

[1] https://www.blackhat.com

[2] LATTNER, Chris; ADVE, Vikram. LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization. IEEE Computer Society, 2004. p. 75.

[3] Kuznetsov, Volodymyr, et al. Code-Pointer Integrity. In OSDI. Vol. 14. 2014. p. 152

[4] Grossman, Dan, et al. Cyclone: A type-safe dialect of C. C/C++ Users Journal 23.1, 2005. p. 113

[5] Necula, George C., Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. ACM SIGPLAN Notices. Vol. 37. No. 1. ACM. 2002. p. 137

[6] Guthaus, Matthew R., et al. "MiBench: A free, commercially representative embedded benchmark suite." Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on. IEEE, 2001. p. 4

[7] Choudhuri, Siddharth, and Tony Givargis. "Software virtual memory management for MMU-less embedded systems." Center for Embedded Computer Systems, 2005. p.12

Abstract

Jaesung Park

Electrical and Computer Engineering

The Graduate School

Seoul National University

With the increasing number of lightweight, simple devices, attacks on such gadgets are becoming a new threat to the users. Such devices often provides small memory space due to space and cost efficiency. Moreover, most of these devices lack Memory Management Unit(MMU) making them prone to attacks like exploiting code pointers. In this paper, we suggest a virtual address system to defend against such attacks.

We implement the system by inserting additional instructions to translate the address of objects saved in the memory by utilizing LLVM compiler. As the types of objects used in attacks exploiting code pointers can be narrowed down to a small number, we omitted the translation process for those that are not taking critical part in such exploits. By excluding redundant translation, we could not only achieve same execution result compared to the non-virtual address system, but also impose rather low overhead than software MMU.

keywords : Virtual Memory System; Pointer Integrity; LLVM Compiler; Safe Stack

Student Number **: 2016-20902**