정 보 보 호 학 회 지
제 31 권 제1호, 2021. 2

리턴 스택 버퍼를 이용한 마이크로아키텍처 데이터 샘플링 공격

김 태 현*, 신 영 주**

요 약

마이크로아키텍처 데이터 샘플링 공격 중 하나인 Zombieload 공격은 마이크로코드 어시스트를 이용하여 물리 코어를 공유하는 다른 논리 코어가 접근하는 데이터를 읽는 공격이다. 마이크로코드 어시스트는 페이지 폴트 과정에서 로드 명령어를 수행할 때 발생하므로, Zombieload 공격은 시그널 핸들러 또는 TSX로 페이지 폴트를 처리 또는 억제한다. 그러나 시그널 핸들러에서 발생하는 잡음과 TSX를 지원하는 프로세서 수의 부족이 Zombieload 공격의 효율을 감소시킨다. 본 논문에서는 페이지 폴트를 RSB를 이용한 잘못된 추측 실행으로 처리하여, 기존의 한계점을 개선한 새로운 Zombieload 공격을 제안한다. 제안한 공격의 성능을 평가하기 위해, 실험을 통해 기존의 Zombieload 공격과 성능을 비교한다. 끝으로 제안한 공격을 막기 위해 여러 가지 방어 기법을 제시한다.

Ⅰ. 서 론

매년 프로세서는 새로운 마이크로아키텍처와 다양한 최적화 기술이 적용되어 이전의 프로세서보다 좋은 성능을 가지게 되었다. 그러나 다양한 최적화 기술들은 마이크로아키텍처 상에 여러 취약점을 야기하여 시스템 보안을 심각하게 위협하고 있다. 마이크로아키텍처 컴포넌트 중에 하나인 캐시의 취약점을 이용한 캐시 부채널 공격은 다양한 암호 알고리즘의 비밀 키를 암호 알고리즘의 결함 없이 유출하면서 시스템 보안을 위협하였다. 이 공격을 막기 위해서는 캐시를 재설계해야 하므로, 캐시 부채널 공격 [1-9]에 취약한 암호 프로그램의 코드를 재수정하여 캐시 부채널 공격으로부터 비밀 키 유출을 방지하였다.

캐시 부채널 공격 이외에도 프로세서가 명령어들을 병렬적으로 처리하게 도와주는 최적화 기술들과 캐시 부채널을 결합한 새로운 공격이 발견되면서, 해당 공격으로 인해 시스템이 보안 위협을 받게 되었다. 이 공격은 임시 실행 공격으로 불리며, 비순차 실행 또는 잘못된 추측 실행으로 임시 명령어를 실행하여 비밀 값을 유출할 수 있다. 임시 명령어로 실행된 값은 아키텍처 상태로 반영되지 않고 캐시에만 남아있으며, 캐시 부채널로 캐시에 남은 값을 유출한다. 해당 공격은 두 가지로 나뉜다.

첫 번째는 Meltdown 유형 공격 [10-16]으로 예외 상황에서 발생하는 임시 명령어들을 비순차실행으로 실행하여 캐시에 남은 값을 캐시 부채널을 이용하여 유출한다. Meltdown 유형 공격들은 페이지 폴트가 발생하는 원인에 따라 다양한 공격이 존재한다. 페이지 폴트는 페이지 테이블 엔트리 bit가 0으로 설정된 페이지에 접근하였을 때 발생하는 예외를 의미한다. 이 공격들은 U/S (User/Supervisor) bit [10], P (Present) bit [12-14]와 floating point [15]를 비롯하여 다양한 페이지 폴트를 이용하여 비밀 값을 유출하였다 [16].

두 번째는 Spectre 유형 공격 [16-23]으로 잘못된 추측 실행으로 발생한 임시 명령어들을 실행하고, 캐시 부채널을 이용해서 임시 명령어로 실행된 값을 유출하였다. 잘못된 추측 실행이 발생하는 과정은 분기 예측기 [17-22]와 memory disambiguation으로 나뉜다 [23].

다양한 방어 기법들을 도입하여 해당 공격들로부터 시스템을 보호하였다. 그러나 최근 새로운 임시 실행

이 논문은 2020년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임(No.2019-0-00533, 컴퓨터 프로세서의 구조적 보안 취약점 검증 및 공격 탐지대응)
* 광운대학교 컴퓨터공학과 (대학원생, taehyun9203@gmail.com)
** 고려대학교 정보보호학과 (조교수, syoungjoo@korea.ac.kr)

공격이 나오면서 PC와 서버들은 여전히 임시 실행 공격으로부터 안전하지 못하였다. 새로운 임시 실행 공격은 마이크로아키텍처 데이터 샘플링 공격으로, Meltdown 유형의 공격이다. 해당 공격은 이전의 Meltdown 유형 공격과 다르게 캐시가 아닌 다른 마이크로아키텍처 컴포넌트로부터 데이터를 유출하면서, 이전의 Meltdown 유형 공격의 방어 기법으로는 해당 공격을 방어하지 못하였다.

마이크로아키텍처 데이터 샘플링 공격으로 데이터를 유출하는 마이크로아키텍처 컴포넌트는 세 가지가 존재한다. Store buffer, LFB (Line Fill Buffer), load port가 이에 해당한다. 본 논문에서는 LFB에 저장된 값을 유출하는 마이크로아키텍처 데이터 샘플링 공격인 Zombieload [25]를 대상으로, 해당 공격들의 효율성을 높이는 새로운 마이크로아키텍처 샘플링 공격을 제안한다. RIDL [24]도 Zombieload처럼 LFB에 저장된 값을 유출하므로, 이름만 다를 뿐 Zombieload와 공격 방법은 동일하다. LFB는 물리 코어에서 로드 및 저장 중인 데이터를 보안 영역과 관계없이 저장하는 내부 버퍼이다. 또한 LFB는 하나의 물리 코어를 가상으로 사용하는 두 논리 코어에서도 공유해서 사용한다. 이 공격들은 LFB를 이용하여 하나의 논리 코어에서 물리 코어를 공유하는 다른 논리 코어가 로드 중인 데이터를 보안 영역과 관계없이 유출한다.

마이크로아키텍처 데이터 샘플링 공격을 포함한 이전의 Meltdown 유형 공격들은 페이지 폴트를 처리 또는 억제하고 임시 명령어로 실행된 값을 캐시 부채널을 이용하여 데이터를 유출한다. 시그널 핸들러 또는 TSX (Transaction Synchronization eXtenstion)가 해당 과정에서 사용되지만 다음과 같은 한계점을 가지고 있어 공격의 효율을 떨어뜨리는 한계점이 존재한다.

시그널 핸들러는 커널 영역에서 페이지 폴트를 처리하고, 유저 영역에 있는 공격자 프로세스에 의해 호출된다. 해당 과정은 잡음이 심하고 속도가 느리지만 모든 프로세서에서 사용할 수 있다. 반대로 TSX는 커널 영역과 유저 영역 사이에 문맥 교환 없이 프로세서의 지원을 받아 페이지 폴트를 억제하므로, 잡음이 없고 속도가 빠르다. 그러나 TSX를 지원하는 Intel 프로세서의 수가 적으므로, 모든 프로세서에서 사용할 수 없다. 해당 한계점들은 마이크로아키텍처 데이터 샘플링 공격의 효율을 떨어뜨린다.

본 논문에서는 페이지 폴트를 처리하고 캐시 부채널로 비밀 값을 유출하는 과정을 리턴 스택 버퍼를 이용한 잘못된 추측 실행으로 제안한다. 제안하는 방법은 페이지 폴트를 처리할 때 유저 영역과 커널 영역 사이에서 문맥 교환이 발생하지 않고 모든 프로세서에서 사용할 수 있으므로, 기존 마이크로아키텍처 데이터 샘플링 공격의 한계점을 보완할 수 있다.

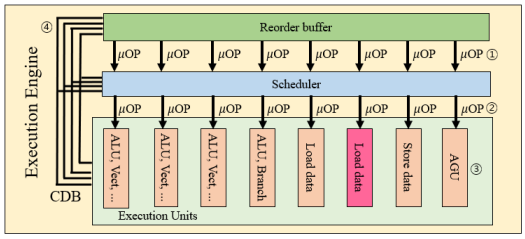
본 논문의 구성은 다음과 같다. 2 장에서는 해당 공격을 설명하기 위한 배경지식을 소개한다. 3장에서는 마이크로아키텍처 데이터 샘플링 공격과 해당 공격이 가진 한계점에 관해 설명한다. 4장에서는 제안한 공격에 사용되는 리턴 스택 버퍼에 대한 설명과 리턴 스택 버퍼를 이용한 Zombieload 공격에 대해 설명한다. 제안한 공격에 대한 구현 부분을 추가로 설명한다. 5장에서는 실험의 환경을 설명하고 실험 결과를 통해 기존의 공격과 제안한 공격의 성능을 평가한다. 6장에서는 제안한 공격을 막기 위한 방어 기법들을 소개한다. 7장에서는 관련 연구를 통해 기존의 연구와의 차이점을 소개한다. 8장으로는 결론 부분으로 논문을 요약한다.

Ⅱ. 배경지식

2.1. 비순차실행

복잡한 명령어 셋을 지원하는 CISC (Complex Instruction Set Computer) 프로세서들은 복잡한 명령어들을 간단한 마이크로 오퍼레이션으로 변환하여 실행 유닛을 통해 실행한다. 프로세서가 프로그램에 들어있는 명령어들을 순서대로만 실행한다면, 앞 순서에 있는 명령어가 실행되는 동안 뒤 순서에 있는 명령어는 실행되지 못하는 문제점이 존재한다. 특히 메모리 접근이 자주 일어나는 프로그램에서는 해당 방식으로 명령어를 수행한다면 프로그램을 실행시키는 속도는 느려진다. CISC 프로세서는 해당 문제점을 비순차실행을 사용하여 해결한다. 비순차실행을 실행하기 위해서 프로세서는 프로그램에 들어 있는 명령어들을 순서대로 마이크로 오퍼레이션으로 변환하여 FIFO (First In First Out) 버퍼인 리오더 버퍼(Reorder Buffer)에 저장한다. 리오더 버퍼에 있는 마이크로 오퍼레이션은 4가지의 상태로 존재한다 (Figure 1).

첫 번째는 이전 명령어의 의존성으로 인해 해결될 때 까지 기다리는 경우(①), 실행을 기다리는 경우(②), 실행이 완료된 경우 (③), 실행이 완료되었지만



〔그림 1〕 Four states of micro-operation

반영이 되지 않은 경우 (④)들이 있다. 반영은 실행된 결과가 아키텍처 상태로 변화된다는 것을 의미하며, 구체적으로 레지스터의 값을 변경하거나 메인 메모리에 데이터를 쓰는 과정을 의미한다.

비순차실행은 실행 유닛이 사용 가능할 때마다 리오더 버퍼에 저장된 의존성이 없는 마이크로 오퍼레이션을 미리 실행하여, 다시 리오더 버퍼에 실행된 결과 값을 저장한다. 그런 다음 프로세서는 리오더 버퍼에 저장된 순서대로 실행된 결과 값을 반영하여 순차적으로 프로그램을 실행한다. 비순차실행은 페이지 폴트 상황에서 페이지 폴트가 처리되기 전까지 명령어들을 실행하여, 처리가 완료될 때 실행된 값들은 페이지 폴트가 발생하기 이전의 값들로 되돌린다.

2.2. 추측실행

프로세서는 분기 명령어를 실행할 때 분기되는 주소가 있는 메모리에 접근해야 하므로 긴 시간이 걸리게 된다. 이러한 시간을 줄이기 위해 프로세서는 분기 명령어를 실행하기 전에 추측 실행을 이용하여 분기되는 주소를 예측하고 예측된 결과로 미리 분기하여 명령어를 미리 실행한다. 구체적으로 프로세서는 분기 명령어에 의해 분기되는 주소를 메모리에서 가져오기 전에, 분기 예측으로 분기 예측기에 있는 분기 되는 주소를 미리 분기하여 분기를 예측한다. 추측 실행은 비순차실행을 이용하여 예측된 분기 주소 이후에 실행되는 명령어를 차례대로 미리 실행하여 리오더 버퍼에 저장한다. 추측 실행 동안 프로세서가 메모리에 있는 분기된 주소와 분기 예측된 주소가 맞게 된다면, 추측 실행으로 실행된 결과 값을 그대로 사용한다. 해당 과정으로 프로세서는 분기 명령어를 실행하지 않고, 리오더 버퍼에 저장된 값을 바로 사용하므로 실행 단계부터 반영 단계까지 생략하여 성능을 개선할 수 있다. 반대로 분기 예측한 주소가 틀렸다면, 프로세서는 추측 실행된 결과 값은 되돌리고 메모리에서 가져온 분기 되는 주소를 바탕으로 분기 명령어를 실행한다.

2.3. 임시 실행 공격

임시 실행 공격은 임시 명령어로 비밀 값을 실행하여 유출하는 공격이다. 임시 명령어는 두 가지 과정에서 발생한다. 첫 번째는 페이지 폴트 상황에서 비순차실행에 의해 실행되는 명령어들을 의미한다. 두 번째는 잘못된 주소로 분기를 예측하여 추측 실행된 명령어들을 임시 명령어를 의미한다. 이러한 임시 명령어들은 페이지 폴트가 처리되거나 메모리에 저장된 분기 되는 주소를 로드할 때까지 임시 명령어들은 실행된다. 프로세서가 페이지 폴트를 처리하거나 메모리에 있는 분기되는 주소와 추측 실행으로 분기된 주소가 같지 않을 때, 임시 명령어로 실행된 값들을 되돌린다. 그 과정에서 메모리나 레지스터에는 반영되지 않지만, 캐시에는 남게 된다. 캐시 부채널을 이용하여 캐시에 남는 값을 복원하여 비밀 값을 유출한다. 해당 공격은 비순차실행을 이용한 Meltdown 유형 공격과 잘못된 추측 실행을 이용한 Spectre 유형 공격으로 분류된다.

Meltdown 유형 공격은 페이지 폴트로 발생한 임시 명령어를 이용하여 비밀 값을 유출한다. 해당 공격은 세 단계로 나뉜다. 첫 번째는 페이지 폴트를 발생시킨다 (단계 1). 페이지 폴트를 처리하는 동안 페이지 폴트에 의해 발생한 임시 명령어들을 비순차실행으로 실행한다 (단계 2). 페이지 폴트를 처리한 후 캐시 부채널로 캐시에 남겨진 값을 유출한다 (단계 3).

Spectre 유형 공격들은 잘못된 추측 실행으로 발생한 임시 명령어를 이용하여 비밀 값을 유출한다. 잘못된 추측 실행은 잘못된 분기 예측 후 해당 분기를 바탕으로 임시 명령어를 추측 실행한 것을 의미한다. 해당 공격은 네 단계로 나뉜다. 첫 번째는 잘못된 분기 예측을 위해 분기 예측기를 훈련하는 과정을 진행한다. 해당 과정은 분기 예측기에 따라 다르게 진행된다 (단계 1). 프로세서가 메모리에서 분기 되는 주소를 로드하는 동안 (단계 1)에 의해 훈련된 분기 예측기를 이용하여 잘못된 분기 예측 후 추측 실행한다 (단계 2). 프로세서는 메모리에 저장된 분기될 주소를 로드한 다음 추측 실행으로 분기된 주소와 비교한다. 비교한 값이 잘못된 경우 추측 실행된 값을 되돌린다 (단계 3). 추측 실행된 값은 메모리와 레지스터에는 반영되지 않지만,

캐시에 남게 된다. 캐시 부채널로 캐시에 남겨진 값을 복원하여 비밀 값을 유출한다 (단계 4).

Ⅲ. 마이크로아키텍처 데이터 샘플링 공격

3.1. Zombieload

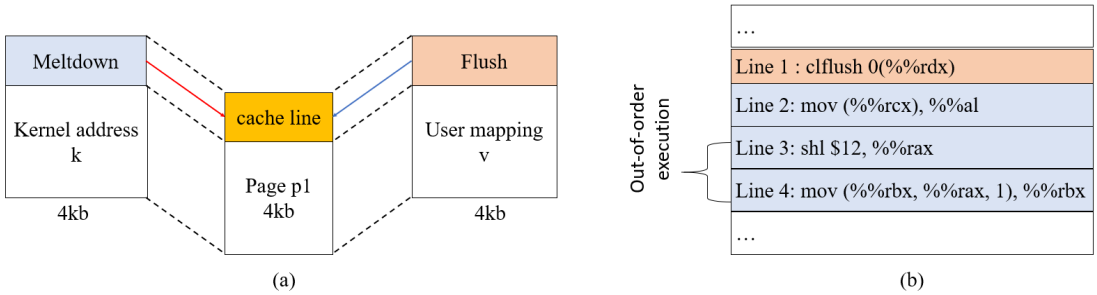
마이크로아키텍처 데이터 샘플링 공격 중 하나인 Zombieload [25]는 하나의 물리 코어를 공유하는 두 개의 논리 코어가 있는 환경에서, 마이크로코드 어시스트를 이용하여 하나의 논리 코어에서 로드 중인 데이터를 다른 논리 코어에서 유출하는 공격이다. 해당 데이터는 LFB에 저장되어 있으며 LFB는 프로세서가 로드하는 데이터가 L1 캐시에 없으면, 메모리 또는 하위 레벨 캐시로부터 데이터를 가져오는 과정에서 거쳐 가는 내부 버퍼이다. 프로세서는 로드가 완료되면 LFB는 해당 데이터를 제거한다. LFB에 저장된 데이터는 로드 중인 데이터를 의미하며, 단일 코어뿐 아니라 하나의 물리 코어를 공유하는 두 개의 논리 코어가 공유해서 사용된다.

그림 2-(a)는 Zombieload 개요이다. 물리 페이지(p1)를 할당 후, p1을 공격자가 접근할 수 있는 가상 주소인 v를 매핑한다 (mmap API 활용). 그 다음, 커널 영역에서 p1에 접근할 수 있는 가상 주소인 k를 찾아낸다. 해당 과정은 KPTI (Kernel Page Table Isolation)가 적용이 되지 않은 운영체제에서만 가능하다. Zombieload는 v가 가지고 있는 값을 캐시에서 비운 뒤, k를 Meltdown으로 로드한다. Meltdown 과정에서 마이크로코드 어시스트를 사용하여 k가 가리키는 값이 아닌 LFB에 저장된 값을 로드한다. 해당 Table에서는 생략이 되어 있지만, Meltdown으로 로드된 값을 캐시 부채널을 통해 유출한다. 그림 2-(b)는 그림 2-(a)를 코드로 나타낸 그림이다.

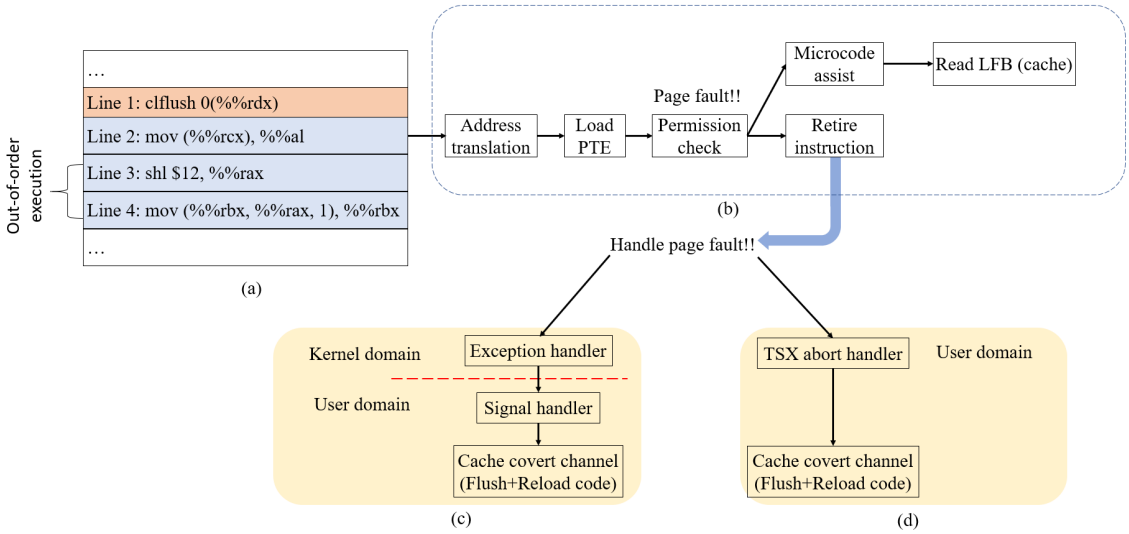
그림 3-(a)는 그림 3-(b)와 같으며 그림 3-(a) 을 이용하여 Zombieload 실행 과정을 설명한다. Line 1에서 v의 주소 값을 가지고 있는 rdx레지스터를 CLFLUSH 명령어를 사용하여 v가 가리키는 값을 캐시에서 비운다. rcx 레지스터는 k를 가지고 있으며, k가 가리키는 1바이트 값을 al 레지스터에 저장한다 (Line 2). k가 가지고 있는 한 바이트 값을 로드하기 위해 프로세서는 line 1에 의해 p1의 값을 메모리에서 가져와야 한다. 해당 과정은 아키텍처 관점에서 복잡하게 동작하므로, 그림 3-(b)로 설명한다.

프로세서는 k를 메모리에 있는 페이지 테이블을 이용하여 물리 주소로 변환한다. 물리 주소에 들어있는 페이지 테이블 엔트리를 통해 프로세서는 U bit가 0으로 설정된 페이지에 접근하였으므로 페이지 폴트 (예외)를 일으킨다. 페이지 폴트 상황에서 프로세서는 마이크로코드 어시스트를 사용하여 k가 가리키는 한 바이트 로드하는 연산은 중지한다 (Line 2 명령어 수행 중단). 대신에 LFB에 저장된 한 바이트 값을 이용하여 line 2의 로드 연산을 다시 진행한다. LFB에 저장된 값은 보안 영역과 관계없이 커널 데이터도 존재한다. Line 2의 명령어가 실행 유닛으로 실행되었다면 al 레지스터에는 LFB의 값이 저장되고, 해당 명령어가 반영되기 전에 프로세서는 비순차실행으로 al 레지스터의 값을 이용하여 line 3~4를 실행한다.

Line 3~4에 관한 내용은 Meltdown 논문 [10]과 같으므로, 간단하게 설명한다. LFB에 저장된 한 바이트 값을 가진 al 레지스터의 값을 4096바이트만큼 곱해 rax 레지스터에 저장한다 (Line 3). 해당 과정은 프리페처가 동작하지 않도록 4096바이트만큼 곱해서 저장한다. 그런 다음 공격자가 지정한 버퍼의 주소를 저장한 rbx레지스터를 이용하여 rax 레지스터값을 버퍼에



〔그림 2〕 Introduction of Zombieload



〔그림 3〕 Zombieload execution process

저장한다 (Line 4). 해당 과정은 프로세서가 페이지 폴트를 처리 또는 억제할 때 캐시에 남은 rax 레지스터값을 유출하기 위해 공격자가 제어할 수 있는 버퍼에 저장하는 과정이다. rax 레지스터값 자체로는 캐시 부채널로 유출할 수 없으므로, rbx가 가리키는 버퍼에 캐시 부채널을 이용하여 해당 버퍼에 저장된 rax 값을 유출한다. 유출된 rax 값을 다시 이용하여 al 레지스터의 값을 추론하여 최종적으로 LFB에 저장된 값을 유출한다.

프로세서는 line 3~4의 명령어들을 비순차실행으로 실행한 후, line 2의 명령어를 반영시키는 단계에서 페이지 폴트를 처리하거나 억제한다. 페이지 폴트를 처리 또는 억제한 이후 캐시 부채널로 LFB에 저장된 값을 유출한다.

Zombieload는 두 가지 방법으로 페이지 폴트를 처리 또는 억제 후 캐시 부채널로 임시 명령어로 실행된 값을 복원한다. 첫 번째는 커널 영역에 있는 예외 핸들러로 페이지 폴트를 처리 후, 공격자 프로세스에 있는 시그널 핸들러를 이용하여 캐시 부채널로 신호를 전달한다 (그림 3-(c)). 캐시 부채널로 버퍼에 저장된 rax 레지스터값을 복원하여 LFB에 저장된 값을 유출한다. 두 번째는 TSX로 페이지 폴트를 억제하는 방법이다. TSX 내부에서 페이지 폴트가 발생한 경우, 프로세서는 TSX abort 핸들러로 페이지 폴트 신호를 억제하고 캐시 부채널로 신호를 전달한다 (그림 3-(d)). 캐시 부채널로 버퍼에 저장된 rax 레지스터값을 복원하여 LFB에 저장된 값을 유출한다.

3.2. 한계점

페이지 폴트를 처리 및 억제하여 캐시 부채널로 신호를 보내주는 시그널 핸들러와 TSX는 다음과 같은 한계점으로 인해 Zombieload의 효율성을 저하한다.

시그널 핸들러는 커널 영역에서 페이지 폴트를 예외 핸들러로 처리한 뒤 유저 영역에 있는 공격자 프로세스에서 호출된다. 호출되는 과정에서 유저 영역과 커널 영역 간에 문맥 교환이 필요하다. 문맥 교환 과정에서 다른 프로세스에서 캐시를 활발하게 사용된다면 캐시에 남아있는 rax 레지스터의 값이 사라지므로, 문맥 교환 후 실행되는 캐시 부채널로 rax 값을 유출하지 못한다. 그래서 많은 프로세스가 동시에 수행되는 환경에서는 rax 레지스터를 유출하는 속도도 느리고, 해당 값을 정확하게 유출하지 못할 수 있다.

TSX는 Intel 프로세서에서 지원하는 기능으로 경쟁조건으로 발생하는 문제를 해결하기 위해 사용된다. TSX을 이용하여 코드를 실행하는 동안 코드 내부에서 페이지 폴트가 발생하면, 프로세서는 TSX abort 핸들러로 페이지 폴트를 억제하고 TSX 바깥에 있는 코드를 수행한다. 페이지 폴트를 억제하는 동안 페이지 폴트 이후에 실행될 명령어들은 비순차실행에 의해 실행되는 임시 명령어들이며, 페이지 폴트를 억제 후 실행된 결과 값은 캐시에 남아있다. Zombieload는 TSX abort 핸들러로 페이지 폴트를 억제한 이후에 캐시 부채널을 통해 rax 레지스터 값을 유출한다. TSX는 유저

영역에서 페이지 폴트를 억제하기 때문에 문맥 교환을 하지 않으므로, 시그널 핸들러 보다 빠르고 노이즈가 적은 장점이 있다. 그러나 TSX를 지원하는 프로세서의 수가 적으므로 해당 공격을 광범위하게 사용할 수 없다.

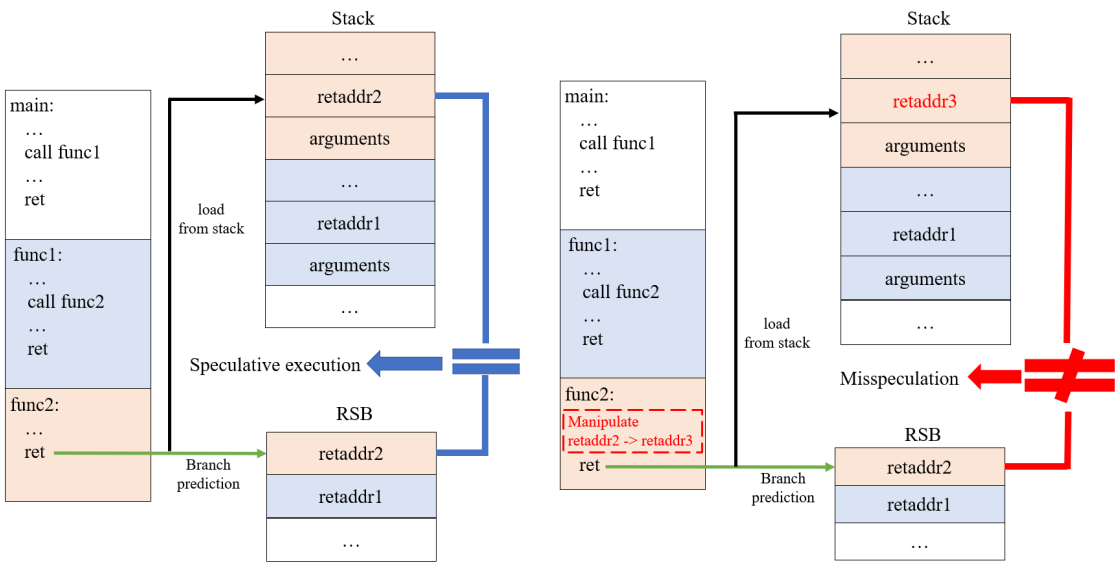
Ⅳ. 제안하는 공격 방법

4.1. 리턴 스택 버퍼 (RSB)

프로그램에서 어떤 함수를 다양한 위치에서 호출하는 경우 해당 함수에 대한 return address를 예측하기 위해, 프로세서 제조 회사들은 분기 예측기를 따로 설계하였으며 리턴 스택 버퍼가 이에 해당한다. 리턴 스택 버퍼가 저장할 수 있는 return address의 개수는 프로세서별로 다르며, 16~32개 정도 된다.

Intel 프로세서는 call 명령어로 다양한 함수들을 호출한다. call 명령어로 함수들이 호출될 때, 해당 함수의 return address와 변수들을 메모리에 있는 스택에 저장한다. 동시에 리턴 스택 버퍼에도 return address가 저장되며 해당 주소는 스택보다 빠르게 접근할 수 있다. 그림 4-(a)은 프로세서가 리턴 스택 버퍼를 사용하여 추측 실행하는 것을 보여준다. 해당 그림에서 프로세서는 call 명령어로 함수 func1을 호출 후, func1내부에서 또 다른 함수 func2을 호출한다. 이 함수들에 대한 return address는 메모리에 있는 스택과 리턴 스택 버퍼에 각각 push 된다. 프로세서는 func2를 실행하는 동안 비순차실행으로 return address를 분기하는 명령어인 ret 명령어를 이용하여 func2의 return address를 미리 실행한다. 해당 과정에서 추측 실행이 발생하며 리턴 스택 버퍼에 들어 있는 retaddr2를 참고하여 분기 예측 후 추측 실행한다. 추측 실행 후 프로세서가 ret 명령어로 스택에 있는 retaddr2를 메모리에서 로드하여 리턴 스택 버퍼에 적혀있는 retaddr2와 비교한다. 같은 값이면 프로세서는 추측 실행으로 실행된 결과 값을 반영한다. 프로세서는 스택에 있는 retaddr2를 분기하는 과정에서 발생하는 긴 사이클을 리턴 스택 버퍼를 이용한 추측 실행으로 사이클을 줄일 수 있다.

그림 4-(b)는 잘못된 추측 실행이 발생하는 경우이다. 그림 4-(a)처럼 func1과 func2 함수를 call 명령어로 호출한 뒤, func2 내부에서 func2의 return address를 변경한다 (retaddr3). 변경된 내용은 스택에만 반영되며, 리턴 스택 버퍼에 저장된 return address는 call 명령어에 의해 저장된 return address를 그대로 가지고 있다 (retaddr2). 프로세서가 비순차실행으로 ret 명령어로 func2의 return address를 실행할 때, 리턴 스택 버퍼에 들어 있는 retaddr2로 분기 예측 후 추측 실행



(a) Speculative execution (b) Misspeculation

〔그림 4〕 Speculation execution and misspeculation of RSB

한다. 프로세서가 ret 명령어로 스택에 있는 retaddr3를 메모리에서 가져올 때, 리턴 스택 버퍼에 적혀있는 retaddr2와 비교한다. 같지 않다면 프로세서는 추측 실행된 값들을 되돌리며, 스택에 저장된 return address로 분기를 시작한다. 되돌리는 과정에서 잘못된 추측 실행으로 실행된 값들은 캐시에 남아있다 [21, 22].

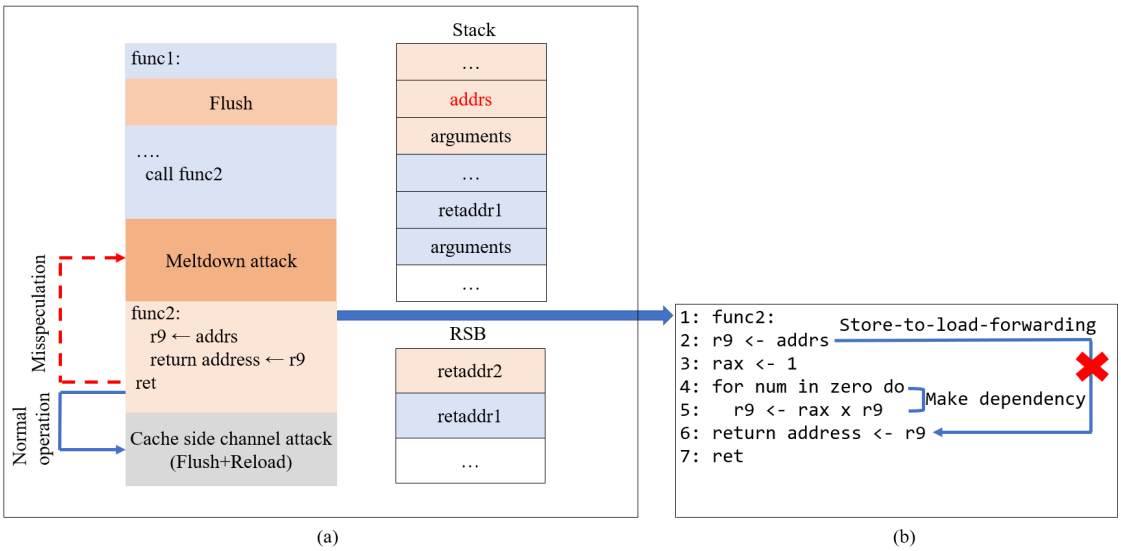
4.2. 리턴 스택 버퍼를 이용한 Zombieload 공격

4.1절에서 설명한 리턴 스택 버퍼의 잘못된 추측 실행을 이용하여 Zombieload 공격의 한계점을 보완하여 새로운 마이크로아키텍처 데이터 샘플링 공격을 제안한다. 해당 과정은 유저와 커널의 문맥 교환이 없고 모든 프로세서에서 지원하므로 시그널 핸들러와 TSX가 가진 한계점들을 보완한다.

그림 5-(a)을 이용하여 제안하는 공격에 대해 설명한다. 함수 func1에서는 그림 5-(a) 의 Flush 과정을 진행한다. Flush 이후 call 명령어로 함수 func2를 호출한다. 스택과 리턴 스택 버퍼에는 func2의 return address인 retaddr2가 push 된다. retaddr2는 call 다음 명령어를 가리키는 주소가 저장 되어 있으며, 해당 그림에서는 그림 5-(a)의 Meltdown에 해당한다. func2 내부에서 r9 레지스터 (r9)에 캐시 부채널의 주소 (addrs)를 삽입 후, retaddr2의 주소를 r9의 레지스터 주소로 바꾼다. 해당 과정으로 스택에는 addrs가 retaddr2 대신 쓰여 있다. 프로세서는 func2 내부에서 ret 명령어를 비순차실행으로 실행할 때, 리턴 스택 버퍼를 사용하여 retaddr2를 분기 예측 후 추측 실행한다. 추측 실행은 그림 5-(a)의 Meltdown을 실행하여 LFB에 저장된 값을 로드한다. 앞 과정은 프로세서가 ret 명령어로 메모리에 저장된 return address (addr)가 다르다는 것을 확인할 때 까지 진행된다. 프로세서는 스택에서 addr을 로드할 때 잘못된 추측 실행인 것을 인지하고, 추측 실행된 값을 되돌리며 되돌리는 과정에서 추측 실행된 값 (3.1절에서 설명한 rax 레지스터 값)은 캐시에 남는다. 프로세서는 ret명령어로 addrs로 분기하여, 캐시 부채널 코드를 실행한다. 3.1절에서 설명한 것처럼 캐시 부채널로 LFB에 저장된 값을 유출한다.

4.3. 구현

그림 5-(a)에서 함수 func2의 코드로는 프로세서가 리턴 스택 버퍼를 이용하여 잘못된 추측 실행을 할 수 없다. 그 이유는 프로세서는 ILP (Instruction Level Parallelism) 방식으로 명령어를 병렬적으로 실행하기 때문이다. 그러므로 프로세서는 그림 5-(a)의 func2의 코드를 mov 명령어로 캐시 부채널 공격의 주소 (r9)를 return address로 바꾸는 store 연산 후 (Line 2), 병렬적으로 ret 명령어로 func2의 return address를 로드한



〔그림 5〕 Block diagram and implementation part of the proposing attack

다 (Line 6). 프로세서는 STF (Store To load Forwarding)을 이용하여 스택에 저장된 return address를 로드하지 않고, line 2 명령어로 L1 캐시에 저장된 return address를 로드하므로 추측 실행이 발생하지 않는다.

STF을 막기 위해 r9 레지스터에 저장된 주소 (캐시 부채널 주소)를 return address에 바로 저장하지 않고 임의의 명령어를 사용하여 의존성을 부여한 후 저장한다 (그림 5-(b)). 구체적으로 곱셈 연산에 해당하는 imul 명령어를 사용하여 r9 레지스터에 저장된 주소에 1을 곱한 값을 다시 r9 레지스터에 저장하는 연산을 반복하였다 (Line 3~5). r9 레지스터가 imul 명령어에 의해 계속 사용되므로 프로세서는 STF로 r9레지스터에 저장된 주소를 ret 명령어로 사용하지 못한다 (Line 6). 그 결과 프로세서는 비순차실행으로 ret 명령어를 미리 실행하여 잘못된 추측 실행을 발생시킨다.

Ⅴ. 실 험

5.1. 리턴 스택 버퍼 (RSB)

실험은 Zombieload와 제안하는 공격이 데이터를 유출하는 속도 (Cycles per byte)와 성공률 (%)을 측정하여 두 공격의 성능을 비교한다. 해당 데이터는 공격자 논리 코어와 물리 코어를 공유하는 희생자 논리 코어에서 로드 중인 커널 데이터를 의미한다. 성공률은 공격자 논리 코어에서 n번 공격하였을 때 희생자 논리 코어가 로드한 데이터를 m번(≤n)만큼 유출할 때의 확률을 의미한다. 속도는 희생자가 로드한 한 바이트를 추출하였을 때 소모되는 프로세서 cycle를 의미한다. Cycle이 소모되는 것이 많을수록 그만큼 느리게 한 바이트를 추출하는 것을 의미하며, 소모되는 사이클이 작을수록 빠르게 한 바이트를 추출한다. 실험에 공통으로 사용되는 파라미터는 두 개이며, number와 retries이다. number는 공격의 횟수를 나타내며, retries는 공격에 실패할 경우 (LFB에서 유출된 한 바이트 값이 0인 경우) 공격을 재시도하는 횟수이다. 제안한 공격은 inst 파라미터를 추가로 사용한다. inst는 제안하는 공격에서 리턴 스택 버퍼의 잘못된 추측 실행을 발생시키기 위해 사용되는 imul 명령어의 개수를 나타내는 파라미터이다 (그림 5-(b)의 num에 해당).

5.2. 성능평가

표 1과 표 2는 Zombieload와 제안한 공격의 성공률과 속도를 비교한 것을 나타낸다. 해당 표들은 number는 1000으로 설정한 상태에서 retries와 inst의 값에 따라 변화되는 제안한 공격과 기존의 Zombieload의 성공률과 속도를 보여준다. 시그널 핸들러를 이용한 Zombieload (Signal-handler)는 커널 영역과 유저 영역의 문맥 전환과정에서 발생한 노이즈로 낮은 성공률과 느린 속도를 보여준다. TSX를 이용한 Zombieload (TSX)은 문맥 전환 없이 프로세서가 페이지 폴트를 처리하므로, Signal-handler에 비해 높은 성공률과 빠른 속도를 보여준다. 제안한 공격 (RSB)은 반복되는 명령어의 개수 (inst)가 적으면, 느린 속도와 낮은 성공률을 보여주지만 일정 수준으로 명령어 (40 ≤ inst ≤ 100)

〔표 1〕 Comparing the success rate between Zombieload attack and the proposing attack

(단위 : %)

Processor	inst / retries	RSB					TSX	Signal-handler
		10	40	70	100	130		
E3-1275-V6	10	0.77	68.86	94.5	77.84	2.32	90.07	1.86
	100	8.29	97.3	96.95	98.95	22.78	99.4	10.3
	1000	54.28	97.28	97.08	99.04	91.63	99.45	53.7
i7-7700K	10	0.91	68.51	96.02	80.12	0.82	89.95	1.92
	100	9.59	97.77	97.31	99.25	8.21	99.27	7.8
	1000	61.05	97.74	97.2	99.27	56.77	99.47	33.59
i7-8700	10	0.67	79.2	96.7	82.72	1.49	92.22	1.49
	100	7.12	98.1	97.43	99.31	14.26	99.54	6.86
	1000	49.44	98.07	97.51	99.32	74.84	99.52	36.95

〔표 2〕 Comparing the speed between Zombieload attack and the proposing attack
(단위 : 10^5 cycles / byte)

Processor	inst / retries	RSB					TSX	Signal-handler
		10	40	70	100	130		
E3-1275-V6	10	7	3	2	3	3	9	9.5
	100	70	8	2	4	20	10	92
	1000	600	8	2	4	200	10	810
i7-7700K	10	7	3	2	2	3	10	8.2
	100	70	8	2	4	30	10	100
	1000	600	8	2	4	200	10	1000
i7-8700	10	6	2	1	2	3	7	7.3
	100	60	4	1	2	20	9	85
	1000	500	5	1	2	200	9	810

가 반복되면 좋은 성능을 보여주는 것을 알 수 있다. 그러나 과도하게 많은 명령어의 개수 (inst > 130)는 반대로 느린 속도와 낮은 성공률 보여준다. 그 이유는 리오더 버퍼에 저장되는 imul 명령어의 수가 많아서, 프로세서가 비순차실행으로 ret 명령어를 미리 실행할 수 없다. 그 결과 잘못된 추측 실행이 일어나지 못한 경우이다. inst가 130개 이상일 때는 정확도가 0에 수렴하여 표 1과 표 2에서는 표시하지 않았다. 실험에서는 inst 값을 10씩 증가시켜서 측정하였으며, 속도와 정확도 면에서 30개씩 증가할 때마다 차이가 발생하여 표 1과 표 2에서는 30단위로 끊어서 표시하였다. retries가 적으면 제안한 공격은 TSX보다 낮은 성능을 보이지만, retries가 증가한 경우에는 거의 비슷한 성능을 보여주는 것을 알 수 있다. Signal-handler도 재시도 횟수를 증가할수록 성공률은 높아지지만, 제안한 공격과 TSX에 비해 속도가 느리고 성공률이 낮다는 것을 확인할 수 있다.

해당 실험 결과로 제안한 공격은 inst가 일정한 범위 내에서 retries 횟수가 1000에 가까울수록 TSX을 이용한 Zombieload와 같은 성능을 보여주고 있다. Signal handler을 이용한 Zombieload는 제안한 공격과 TSX를 이용한 Zombieload에 비해 느린 속도와 낮은 정확도를 보여주고 있다. 제안한 공격의 한계점은 inst 개수를 프로세서별로 적절하게 조절해야 하며, 반복하는 횟수도 증가시켜야만 TSX를 이용한 Zombieload 공격과 비슷한 성능이 나오는 것을 보여준다. 이러한 한계점에도 해당 공격은 모든 프로세서에서 사용할 수 있으므로, TSX을 이용한 Zombieload보다 광범위하게 사용할 수 있다는 장점이 존재한다.

Ⅵ. 방어기법

6.1. 하이퍼스레딩 기능 비활성화

하이퍼스레딩은 Intel 프로세서의 각각 물리 코어들을 두 개의 논리 코어로 가상적으로 나누어서 사용하는 기술이다. 해당 기술은 단일 코어로 프로그램을 처리하는 것보다 빠르게 프로그램을 처리할 수 있다. 두 논리 코어는 단일 코어처럼 코어 내에 존재하는 마이크로아키텍처 컴포넌트 들을 공유해서 사용한다. Zombieload와 RIDL은 하이퍼스레딩 환경에서 두 개의 논리 코어가 공유해서 사용하는 LFB에 저장된 값을 유출한다. 해당 공격을 막을 수 있는 근본적인 방어기법으로는 하이퍼스레딩 기능을 비활성화 하는 것이다. 그러면 해당 공격으로부터 논리 코어에서 로드 중인 데이터를 유출하는 것을 방지할 수 있다. 그러나 해당 기능을 비활성화하게 되면 두 개의 논리 코어가 처리하는 일을 단일 물리 코어가 처리하게 됨으로써 프로세서의 성능 저하가 있을 수 있다.

6.2. KPTI 활성화

KPTI [29, 30]은 페이지 테이블을 관리하는 레지스터인 CR3를 이용하여 커널 영역 전체를 유저 프로세스에 맵핑하지 않고 필요한 부분만 유저 프로세스에 맵핑되게 한다. 유저 영역에서 맵핑되지 않은 커널 영역에 접근하는 경우 어떠한 것도 읽을 수 없다. 유저 영역에서 커널이 제공하는 기능을 이용하기 위해서는,

CR3 레지스터를 이용하여 유저 영역에서 커널 영역으로 문맥 전환 후 이용할 수 있다. 반대인 경우에도 CR3 레지스터를 이용하여 문맥 전환을 할 수 있다. 이 과정은 유저 영역에서 직접 커널 페이지에 접근하는 것을 불가하게 하였지만, 심각한 오버헤드가 발생하는 단점이 있다. Zombieload는 커널 영역이 유저 프로세스에 맵핑된 환경에서 공격자 프로세스에 맵핑된 커널 페이지의 주소를 이용하여 Meltdown을 진행하기 때문에, KPTI가 적용된 운영체제에서는 공격할 수 없다.

6.3. LFB 데이터 비우기

마이크로아키텍처 데이터 샘플링 공격에 대한 방어 기법으로는 커널 영역에서 유저 영역으로 문맥 전환할 때 Intel 프로세서에서 지원하는 VERW 명령어로 LFB에 저장된 커널 데이터를 덮어쓴다 [32, 33]. Zombieload가 커널 데이터를 유출하기 위해서는 희생자 논리 코어에서 커널 영역에서 데이터를 로드한 상태에서 문맥 전환 후 공격자 논리 코어가 유저 영역에 있는 Zombieload 프로세스를 실행해야 한다. 해당 방어 기법으로 LFB에 저장되어 있는 데이터를 덮어쓴다면 공격 프로세스는 희생자가 로드 중인 커널 데이터를 읽을 수 없다.

Ⅶ. 관련 연구

7.1. Meltdown 유형 공격

Meltdown 유형 공격들은 페이지 폴트가 발생하는 방식에 따라 나뉘며 Meltdown [10], Foreshadow [11, 12], Lazy FP [13]이 있다.

Meltdown [10]은 유저 영역에서 커널 데이터에 접근하였을 때 발생하는 페이지 폴트를 이용하여, 프로세서가 페이지 폴트를 처리하는 동안 비순차실행에 의해 실행되는 임시 명령어를 이용하여 해당 커널 데이터를 유저 영역에서 유출한다. Meltdown에서 사용된 페이지 폴트는 페이지 테이블 엔트리에 들어있는 U/S bit에 의해 발생된다.

Foreshadow [11, 12]는 커널 모듈로 L1 캐시에 저장된 페이지의 P bit를 0으로 설정한 다음 공격을 진행한다. 유저 영역에서 해당 페이지에 접근했을 때 페이지 폴트가 발생하며, Meltdown처럼 임시 명령어를 사용하여 해당 페이지의 데이터를 유출한다. Foreshadow는 L1 캐시에 저장된 SGX enclave의 페이지를, Foreshadow-NG는 운영체제, 가상머신 등 다양한 환경에서 사용되는 L1 캐시에 저장된 페이지의 데이터를 유출하였다.

Lazy FP 공격은 하나의 물리 코어를 공유하는 두 프로세스를 이용하여 공격한다. 프로세서는 floating point 연산과 관련된 레지스터들을 문맥 전환 시 즉각적으로 바꾸지 않고 그대로 두는 경향이 있다. 해당 공격은 그러한 경향을 이용하여 희생자 프로세스는 해당 레지스터 중 하나인 SIMD (Single Instruction Multiple Data) 레지스터를 이용하여 데이터를 사용한다. 문맥 전환 후 공격자 프로세스에서 희생자 프로세스에서 사용 중인 데이터에 접근하게 된다면, 페이지 폴트가 발생한다. 페이지 폴트 후 Meltdown으로 SIMD에 들어 있는 값을 유출한다. 해당 공격은 Meltdown, Foreshadow과 다른 새로운 Meltdown 유형 공격이므로, 앞에 두 공격에 대한 방어 기법으로 방어하지 못하였다 [28-31].

세 가지 Meltdown 유형 공격의 특징들은 유출되는 경로가 캐시라는 특징이다. 마이크로아키텍처 데이터 샘플링 공격은 해당 공격들과 다르게 캐시가 아닌 다른 마이크로아키텍처 컴포넌트 상에 존재하는 데이터를 유출하므로, 기존의 Meltdown 유형 공격과 다른 공격이다.

7.2. Spectre 유형 공격

Spectre 유형 공격은 잘못된 추측 실행에 따라 나뉘며, 잘못된 추측 실행이 발생할 수 있는 경우는 4가지 경우가 있다. 앞에 3 가지 경우는 분기 예측기로 잘못된 분기 예측할 때 발생한다. Spectre 유형 공격으로 취약점이 발견된 분기 예측기는 세 가지 종류인 BPU (Branch Prediction Unit) [17, 18], BTB (Branch Target Buffer) [17, 19, 20], 리턴 스택 버퍼 [21, 22]가 존재한다. 마지막은 분기 예측기가 아닌 메모리 연산과 관련해서 발생하는 경우이며, 저장 중인 데이터를 로드할 때 발생한다.

BPU는 최근 분기 명령어의 분기되는 기록을 적어놓은 분기 예측기로, 조건 분기의 추측 실행에 사용된다. Spectre 공격자는 BPU를 이용하여 조건 분기문 밖에 있는 값을 유출하였다. 잘못된 추측 실행을 위해 이

전의 분기 기록이 계속 분기되도록 BPU를 훈련한 다음, 경계 밖에 있는 값을 조건 분기문에 삽입하여 BPU가 해당 값을 가지고 잘못된 추측 실행을 하도록 한다. 해당 이후의 과정은 임시 명령어를 이용하여 접근하고, 캐시 부채널을 이용하여 유출한다.

BTB는 이전의 명령어 결과에 의해 분기 되는 주소가 결정되는 간접 분기 명령어에 사용되는 분기 예측기이다. BTB는 간접 분기 명령어의 하위 bit와 해당 명령어에 대한 분기 되는 주소를 저장한다. 분기 되는 주소의 하위 bit를 반복적으로 사용하는 반복문에서 프로세서는 BTB를 이용하여 분기 예측한다. Spectre 공격자는 BTB를 이용하여 다른 주소 공간에 있는 값을 유출하였다. 해당 공격의 준비 과정으로는 희생자와 공격자 프로세스가 필요하다. 공격자 프로세스는 잘못된 추측 실행을 위해 희생자 프로세스에서 유출하려는 간접 분기 명령어의 주소를 확인 후, 해당 주소의 하위 bit가 같은 위치에서 간접 분기 명령어를 이용하여 비밀 값을 유출할 수 있는 주소를 삽입한다. 공격자 프로세스에서 유출하는 주소를 계속 분기하도록 설정한 뒤, 문맥 전환을 이용하여 희생자 프로세스가 실행하도록 설정한다. 희생자 프로세스는 BTB를 이용하여 공격자 프로세스에서 설정한 주소로 잘못된 추측 실행을 하여 비밀 값을 유출한다.

리턴 스택 버퍼는 return address를 분기할 때 사용되는 분기 예측기이다. 리턴 스택 버퍼에는 각 함수의 return address가 호출된 순서대로 스택 형태로 저장되어 있다. 프로세서가 리턴 스택 버퍼로 잘못된 추측 실행을 위해, Spectre 공격자는 리턴 스택 버퍼에 들어있는 return address를 오버플로, 조작 등 다양한 방법을 이용하여 리턴 스택 버퍼에 들어있는 return address와 실제 return address를 다르게 하여 잘못된 추측 실행을 발생시킨다. 오버플로인 경우는 리턴 스택 버퍼에 저장한 크기보다 더 많은 return address가 들어온다면, 리턴 스택 버퍼 중 가장 마지막에 저장된 return address는 사라진다. 리턴 스택 버퍼에서 사라진 return address를 가지고 있는 함수가 비순차실행으로 ret 명령어를 실행한다면, 리턴 스택 버퍼에 들어 있는 다른 return address를 이용하여 잘못된 추측 실행을 하게 된다. 조작은 본 논문에서 소개한 방식처럼 어셈블리 코드를 이용하여 return address를 변환한다. 그 결과 리턴 스택 버퍼와 스택에 저장되어 있는 return address가 같지 않아서 잘못된 추측 실행이 발생한 경우이다. 리턴 스택 버퍼의 잘못된 추측 실행을 이용한 Spectre 공격은 다른 주소 공간에서 사용 중인 값을 유출할 수 있다.

마지막 Spectre 공격은 분기 예측기를 사용하지 않은 경우이다 [23]. 메모리에 데이터를 저장하는 시간은 로드하는 시간보다 길게 측정된다. 만약 메모리에 저장 중인 데이터를 로드하게 된다면, memory disambiguation에 의해 저장되기 이전의 값이 로드되는 현상이 발생한다. 해당 현상은 잘못된 추측 실행으로 발생되며 Spectre 공격으로 저장되기 전 이전의 값을 유출할 수 있다.

기존의 Spectre 공격들은 잘못된 추측 실행을 정상적인 코드에 적용하여 다른 주소 공간에서 사용 중인 데이터를 유출하였다. 그러나 본 논문은 잘못된 추측 실행을 페이지 폴트를 이용한 코드에 적용하여 기존의 Meltdown 유형의 공격을 개선했다는 점에서 다른 접근이다.

7.3. 마이크로아키텍처 데이터 샘플링 공격

마이크로아키텍처 데이터 샘플링 공격은 Meltdown 유형 공격이지만, 캐시가 아닌 다른 마이크로아키텍처 컴포넌트에 존재하는 데이터를 유출하는 경우이다. 취약점이 발견된 마이크로아키텍처 컴포넌트는 세 개이며, LFB, store buffer, load port가 있다. LFB와 load port는 물리 코어를 공유하는 하나의 논리 코어가 로드 중인 데이터를 보관하고 있는 내부 버퍼에 해당이 되며, store buffer는 해당 논리 코어가 저장 중인 데이터를 보관하고 있는 버퍼이다.

RIDL [24]는 LFB와 load port를 Zombieload [25]는 LFB를 이용하여 논리 코어에서 로드 중인 데이터를 유출한다. 프로세서는 페이지 폴트 상황에서 로드하려는 데이터가 캐시에 없는 데이터인 경우, LFB 또는 load port 들어 있는 데이터를 이용해서 로드한다. 페이지 폴트 상황에서 로드된 데이터는 임시 명령어로 로드된 데이터 이므로 캐시에 남게 된다. 캐시 부채널로 캐시에 있는 데이터를 유출한다. LFB 또는 load port에 저장된 데이터는 커널 영역과 유저 영역 상관없이 저장되어 있으므로, 해당 공격들로 논리 코어가 로드 중인 커널 데이터도 유출할 수 있다.

Fallout [26]은 store buffer를 이용하여 논리 코어에서 저장 중인 데이터를 유출하거나 혹은 유효한 커널

주소를 찾는다. Store buffer는 store 연산에 사용된 데이터를 메모리에 반영되기 전까지 저장해서, store 연산과 관련 없이 잇따라 존재하는 명령어들을 실행할 수 있다. 그 결과 store 명령어가 반영될 때 까지 기다려야하는 시간을 줄여준다. Fallout으로 발견된 공격은 두 가지가 있다. 첫 번째는 WTF (Write Transient Fault) 공격으로 논리 코어가 저장 중인 데이터를 store buffer를 이용하여 유출하는 공격이다. 해당 공격은 프로세서가 로드 연산을 할 때 store buffer에 저장된 주소 중 하위 12bit를 일치한다면 해당 주소의 값을 로드하는 것을 이용한다. 논리 코어가 저장 중인 주소의 하위 12bit를 알고 있는 공격자는 하위 12bit만 같은 무작위에 의한 주소를 페이지 폴트 상황에서 Meltdown 유형 공격으로 로드한다. 로드된 값은 임시 명령어로 실행된 값이므로, 캐시에만 남게 된다. 캐시 부채널로 캐시에 남는 값을 유출한다.

두 번째는 STF을 이용한 공격이다. STF는 유효한 주소에 대해서만 발생하며, 저장 중인 주소를 바로 로드할 때 저장되는 데이터가 메모리에 반영되기 전에 로드 명령어로 바로 사용되는 기술이다. STF는 유효한 주소에 대해서만 발생한다. 해당 공격은 페이지 폴트 상황에서 특정 주소에 공격자가 알 수 있는 데이터를 저장하고 다시 로드를 진행한다. 페이지 폴트에서 실행되는 연산들은 임시 명령어들로 실행되었으므로 실행된 값은 캐시에만 남는다. 페이지 폴트를 처리 후 캐시 부채널로 캐시에 남는 값을 복원한다. 복원된 값이 공격자가 저장하는 과정에서 사용된 데이터면 STF가 사용된 것이므로 해당 주소는 유효한 주소를 의미한다. 반대면 유효한 주소가 아닌 경우이다.

기존의 공격들은 리턴 스택 버퍼를 이용한 잘못된 추측 실행을 사용하지 않고 시그널 핸들러와 TSX를 이용하여 페이지 폴트를 처리하였기 때문에, 기존의 마이크로아키텍처 데이터 샘플링 공격과 다른 새로운 공격이다.

7.4. 리턴 스택 버퍼를 이용한 Meltdown 공격 성능 개선

본 논문은 기존의 논문 [33]을 참고하여 마이크로아키텍처 데이터 샘플링 공격의 성능을 개선하였다. 기존의 논문에서는 리턴 스택 버퍼를 이용한 잘못된 추측 실행을 이용하여 Meltdown [10]의 한계점을 극복하고 새로운 Meltdown을 제안하였다. 본 논문은 해당 연구를 확장한 것으로 Meltdown을 사용하는 마이크로아키텍처 데이터 샘플링 공격 중 하나인 Zombieload의 성능을 개선하였다. 해당 연구와의 차이점은 기존 연구는 캐시에 저장된 데이터를 유출하는 Meltdown 유형 공격에 대한 성능을 개선시킬 수 있었다. 본 논문은 캐시가 아닌 다른 마이크로아키텍처 컴포넌트에 저장된 데이터를 유출하는 공격인 마이크로아키텍처 데이터 샘플링의 성능을 개선할 수 있다는 것을 보여주었다.

Ⅷ. 결론 및 고찰

본 논문은 리턴 스택 버퍼를 이용하여 마이크로아키텍처 데이터 샘플링 공격의 성능을 개선하여 새로운 마이크로아키텍처 데이터 샘플링 공격을 제안하였다. 해당 공격은 마이크로코드 어시스트와 Meltdown을 이용하여 LFB에 저장되어 있는 데이터를 유출하였다. Meltdown을 사용하기 위해 기존의 마이크로아키텍처 데이터 샘플링 공격은 두 가지 방법을 제안하였다. 첫 번째는 Meltdown에서 발생하는 페이지 폴트를 커널 영역에서 예외 핸들러로 처리하고 유저 영역에서 시그널 핸들러를 호출하여 캐시 부채널로 임시 명령어로 실행된 값을 유출하였다. 두 번째는 TSX로 프로세서가 직접 페이지 폴트를 억제 후 캐시 부채널로 임시 명령어로 실행된 값을 유출하였다. 해당 과정에서 사용되는 시그널 핸들러와 TSX는 다음과 같은 한계점이 존재하여 마이크로아키텍처 데이터 샘플링 공격의 효율성을 떨어뜨린다. 시그널 핸들러는 커널 영역과 유저 영역의 문맥 교환 때문에 속도가 느리고 잡음이 발생하며, TSX는 커널 영역과 유저 영역의 문맥 교환이 없어서 빠르고 노이즈가 없지만, 지원하는 프로세서 수가 적은 한계점이 존재한다.

본 논문은 리턴 스택 버퍼를 이용한 잘못된 추측 실행으로 기존의 한계점을 극복하고 새로운 마이크로아키텍처 데이터 샘플링 공격을 제안한다. 새로운 공격은 모든 프로세서에서 사용할 수 있으며, 커널 영역과 유저 영역의 문맥 교환이 없으므로 잡음이 적고 속도가 빠르다. 실험을 통해 기존 공격과 성능을 비교하여 제안한 공격이 모든 프로세서에서 TSX를 이용한 공격만큼 좋은 성능을 갖는다는 것을 보여주었다. 게다가 해당 공격으로부터 시스템을 보호하기 위해 다양한 방어 기법을 소개하였다.

본 논문은 마이크로아키텍처 데이터 샘플링 공격들

의 성능을 개선할 가능성을 제시하였다. 이전 연구들은 새로운 취약점을 발견하는 것에 초점을 두고 연구하였기 때문에, 해당 공격의 성능을 개선하는 연구 또한 새로운 분야이다. 본 논문의 연구는 실험을 통해 성능을 개선하였기 때문에 이론적으로 완벽하게 구현된 부분이 미흡하다. 추가로 미흡한 부분들을 개선하여 모든 프로세서에서 해당 공격을 사용할 수 있는 연구로 발전될 수 있다.

참 고 문 헌

[1] Yuval Yarom, and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In Proceedings of the 23rd USENIX Security Symposium, pp. 719-732, San Diego, CA, US, August 2014.

[2] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! a fast, cross-VM attack on AES. In Symposium on Research in Attacks, Intrusions and Defenses (RAID), pp. 299-319, Gothenburg, Sweden, September 2014.

[3] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In Proceedings of the 24th USENIX Security Symposium, pp. 897-912, Washington, DC, US, August 2015.

[4] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In Proceedings of the 2015 ACM Conference on Computer and Communications Security, pp. 1406-1418, Denver, CO, US, October 2015.

[5] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In 2015 IEEE Symposium on Security and Privacy, pp. 605-622, San Jose, CA, US, May 2015.

[6] Clementine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Kay Romer, and Stefan Mangard. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In Network and Distributed System Security Symposium, Vol. 17, pp. 8-11, San Diego, CA, US, February 2017.

[7] Daniel Gruss, Clementine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A fast and stealthy cache attack. In Proceedings of the 13rd Conference on Detection of Intrusions and Malware & Vulnerability Assessment, pp. 279-299, San Sebastian, Spain, July 2016.

[8] Osvik, Dag Arne, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In Cryptographers' track at the RSA conference, pp. 1-20, San Jose, CA, US, February 2006.

[9] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack Using Intel TSX. In Proceedings of the 26th USENIX Security Symposium, pp. 51-67, Vancouver, BC, Canada, 2017.

[10] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In Proceedings of the 27th USENIX Security Symposium, pp. 973-990, Baltimore, MD, US, August 2018.

[11] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution. In Proceedings of the 27th USENIX Security Symposium, pp. 991-1008, Baltimore, MD, US, August 2018.

[12] Intel. L1 Terminal Fault. https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-faul, August 2018.

[13] Stecklina, Julian. [RFC] x86/speculation: add

L1 Terminal Fault / Foreshadow demo. https://lkml.org/lkml/2019/1/21/606, pp. 1-6, January 2019.

[14] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. https://foreshadowattack.eu/, pp. 1-7, August 2018.

[15] Stecklina, Julian, and Thomas Prescher. Lazyfp: Leaking fpu register state using microarchitectural side-channels. arXiv preprint arXiv:1806.07480, pp. 1-6, July 2018.

[16] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In Proceedings of the 28th USENIX Security Symposium, pp. 249-266, Santa Clara, CA, US, August 2019.

[17] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In 2019 IEEE Symposium on Security and Privacy, pp. 1-19, San Francisco, CA, US, May 2019.

[18] Michael Schwarz, Martin Schwarzl, Moritz Lipp and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In European Symposium on Research in Computer Security, pp. 279-299, Cham, Luxembourg, September 2019.

[19] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In 2019 IEEE European Symposium on Security and Privacy, pp. 142-157, Stockholm, Sweden, July 2019.

[20] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: exploiting speculative execution through port contention. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 785-800, London, UK, November 2019.

[21] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In 12th USENIX Workshop on Offensive Technologies (WOOT), pp. 1-12, Baltimore, MD, US, August 2018.

[22] Giorgi Maisuradze, and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 2109-2122, Toronto, Canada, October 2018.

[23] Horn, Jann. speculative execution variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, August 2019.

[24] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In 2019 IEEE Symposium on Security and Privacy, pp. 88-105, San Francisco, CA, US, May 2019.

[25] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 753-768, London, UK, November 2019.

[26] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss1, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and

Communications Security, pp. 769-784, London, UK, November 2019.

[27] Intel. Deep Dive: Intel Analysis of L1 Terminal Fault. https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-l1-terminal-fault, August 2018.

[28] Intel. Rogue Data Cache Load / CVE-2017-5754, INTEL-SA-00088. https://software.intel.com/security-software-guidance/software-guidance/rogue-data-cache-load#mitigation, January 2018.

[29] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Cl´ementine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In International Symposium on Engineering Secure Software and Systems, pp. 161-176, Cham, Luxembourg, July 2017.

[30] Intel. L1 Terminal Fault / CVE-2018-3615, CVE-2018-3620, CVE-2018-3646 / INTEL-SA-00161. https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault, August 2018.

[31] Intel. Deep Dive: Intel Analysis of Microarchitectural Data Sampling.https://software.intel.com/security-software-guidance/insights/deep-diveintel-analysis-microarchitectural-data-sampling, May 2019.

[32] Intel. Side Channel Vulnerability MDS.https://www.intel.com/content/www/us/en/architecture-and-technology/mds.html, May 2019.

[33] Taehyun Kim, and Youngjoo Shin. Reinforcing Meltdown Attack by Using a Return Stack Buffer. IEEE Access, Vol. 7, pp. 186065-186077, 2019.

〈저 자 소 개〉

김 태 현 (Taehyun Kim)

학생회원

2018년 8월 : 광운대학교 전자공학과 학사

2020년 8월 : 광운대학교 컴퓨터공학 석사

<관심분야> 정보보호, 부채널공격

신 영 주 (Youngjoo Shin)

종신회원

2006년 2월 : 고려대학교 컴퓨터학과 학사

2008년 2월 : KAIST 전산학과 석사

2014년 8월 : KAIST 전산학과 박사

2008년 4월~2017년 2월 : 국가보안기술연구소 선임연구원

2017년 3월~2020년 8월 : 광운대학교 컴퓨터정보공학부 조교수

2020년 9월~현재 : 고려대학교 정보보호대학원 정보보호학과 조교수

<관심분야> 시스템 보안, CPU 마이크로아키텍처 취약점 분석, 클라우드 보안