*Journal of The Korea Institute of Information Security & Cryptology* ISSN 1598-3986(Print)
VOL.33, NO.5, Oct. 2023 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2023.33.5.775

# 비 문서화 명령어 탐색 퍼저의 명령어 실행 정보 기반 길이 결정 방법*

이 유 석,[1†] 송 원 준[2‡]
[1,2]강원대학교 (학생, 교수)

# A Method of Instruction Length Determination Based on Execution Information in Undocumented Instruction Fuzzer*

Yoo-seok Lee,[1†] Won-jun Song[2‡]
[1,2]Kangwon National University (Undergraduate student, Professor)

## 요 약

프로세서 기술의 발전은 ISA 확장의 가속화와 마이크로아키텍처의 복잡성을 증가시켰으며, 이에 따라서 다양한 프로세서 검증 기법에 대한 관심이 계속해서 증가하고 있다. 최근 비 문서화 명령어를 찾기 위한 다양한 퍼징 기술이 소개되고 있으며, 본 연구에서는 기존의 비 문서화 퍼징 기술의 문제점과 Intel 및 AMD의 최신 프로세서에서의 오류사례를 소개한다. 특히, 퍼저가 CPU 명령어 길이를 잘못 판단하여 발생하는 긍정오류 문제의 원인을 분석하고, 명령어 실행 정보 기반의 길이 결정 기법을 제안함으로써 정확도를 향상시킨다.

## ABSTRACT

As processor technology advances, it has accelerated ISA extensions and increased the complexity of micro-architectures, leading to a continued rise in the importance of processor validation techniques. Recently, various fuzzing techniques have been introduced to discover undocumented instructions, and this study highlights the shortcomings of existing undocumented instruction fuzzing techniques and presents our observation on error cases in the latest processors from Intel and AMD. In particular, we analyzes the causes of false positives resulting from the fuzzer incorrectly judging CPU instruction length and proposes the length determination technique based on instruction execution information to improve accuracy.

**Keywords:** Fuzzing, Undocumented Instruction, x86-64 architecture

## I. 서 론

현대 컴퓨팅 환경에서 가상화, 보안, 인공지능 등의 다양한 요구가 증가함에 따라서 프로세서의 ISA (Instruction Set Architecture) 확장과 마이크로아키텍처의 개선이 지속적으로 이루어지고 있다. 이러한 개선은 마이크로 아키텍처의 복잡도 증가와 이에 따른 다양한 프로세서 보안 취약점 문제를 발생시켰다. 성능 개선 목적으로 설계된 예측 실행과 캐싱 기술의 발전은 마이크로아키텍처 수준의 다양한 부 채널 공격들에 취약할 수 있음이 밝혀졌고[1,2,3,4], ISA의 새로운 명령어 확장 역시 보안 공격에 활용되기도 한다. 예를 들면, Intel TSX는 하드웨어 기반의 트랙젝션 메모리

Received(09. 12. 2023), Modified(10. 11. 2023), Accepted(10. 11. 2023)
* 본 연구는 부분적으로 2020년도 강원대학교 대학회계 학술연구 조성비로 연구하였음 그리고 이 성과는 부분적으로 2021년도 정부의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(NRF-2021R1F1A1064009)
* 본 논문은 2023년도 한국정보보호학회 하계학술대회에서 발표한 우수논문을 개선 및 확장한 것임.
† 주저자, dldbtjr1999@kangwon.ac.kr
‡ 교신저자, wjsong@kangwon.ac.kr(Corresponding author)

(transactional memory operation)를 지원하기 위해서 확장되었지만 ASLR(Address Space Layout Randomization)을 우회하는데 악용되었고[5], SIMD (Single Instruction Multiple Data) 명령어 확장 집합인 AVX (Advanced Vector Extensions)의 데이터 이동 명령어를 악용하여 데이터 유출 공격에 사용되기도 했다[6]. 또한, Intel의 보안 확장인 AES-NI의 취약 명령어를 사용하여, AES 키 값을 추출하거나[7], 가상화 기술 확장 명령어의 취약점으로 권한 상승 공격에 악용되었다[8].

신뢰도 높은 프로세서의 구현과 설계를 검증하기 위해서 형식 검증(formal verification)[19,20]과 기능 검증(functional verification)의 다양한 방법들이 제안되어왔다. 특히, 기능 검증을 위해서 ISA 명세를 바탕으로 구현된 RTL(Register Transfer Level) 설계에 대한 다양한 차등 테스팅(differential testing) 기법들이 주로 연구되어 오고 있다[9,10,11]. RTL 설계를 바탕으로 구현된 상용 프로세서에서, ISA에는 명세 되지 않아있지만 프로세서 내부에 구현된 명령어들을 비 문서화 명령어(undocumented instruction) 라고 한다. 이러한 명령어들은 디버깅 혹은 NDA 등의 다양한 목적을 위해서 존재할 수 있으며, 메모리 보호 기법들을 우회하거나[12], 권한 상승 등의 위협적인 보안 공격에 대한 연구들 역시 발표되고 있다[13]. 명세 되지 않은 명령어들에 대한 보안 위협이 계속해서 증가됨에 따라서, 최근에는 ISA 명세에 대한 기능 검증 뿐만 아니라 비 문서화 명령어의 검증 기술에 대한 연구가 활발히 진행되고 있다[14,15,16,17,18].

본 논문에서는 기존에 알려진 비 문서화 명령어를 탐지하는 방법이 갖는 문제점을 상용 AMD 및 Intel 프로세서에서 검증하고, 탐지 방식에서 긍정 오류(false positive)가 발생하는 원인을 자세히 분석한다. 특히, CISC(Complex Instruction Set Computer) 기반의 x86 아키텍처는 명령어의 길이가 비 균일한 특징을 가지기 때문에 선행 연구는 비 문서화 명령어의 길이를 결정하기 위해서, 공통적으로 하드웨어 예외(exception) 기반의 결정 기법을 사용한다. 하지만 이 기법은 최신 Intel 및 AMD 프로세서에서는 올바르게 명령어의 길이를 판단하지 못하는 문제가 존재하며, 본 연구에서는 이러한 문제를 해결하기 위해서 프로그램 카운터 (Program Counter:PC)와 하드웨어 성능 카운터 (H/W Performance Counter)등의 실제 하드웨어 수준에서 실행된 정보를 활용하여 명령어 길이 결정의 정확도를 향상시키는 방법을 제안한다.

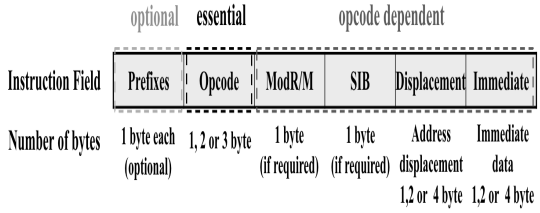


Fig. 1. x86-64 Instruction Format

## II. 배경 지식

### 2.1 x86-64 명령어 형식

x86-64 아키텍처는 CISC 체계의 프로세서로서, 단일 명령어가 여러 개의 저수준 연산 실행을 통해 복잡한 계산을 수행할 수 있도록 하는 특징을 갖는다. 또한, 고정된 길이의 명령어를 사용하는 RISC (Reduced Instruction Set Computer) 체계와 다르게 1~15Byte의 길이를 갖는 가변 길이 명령어를 사용한다.

RISC와 다르게 CISC는 비 균일 디코딩(non-uniform decoding)을 수행하며, Fig. 1.과 같이 x86-64명령어는 6개의 필드 Prefix, Opcode, ModR/M, SIB, Displacement, Immediate으로 구성될 수 있다. Opcode는 명령어의 핵심 동작을 알 수 있게 하는 필드이며, 반드시 존재해야 한다. ModR/M, SIB는 명령어가 사용하는 레지스터와 메모리 정보를 명시하는 필드이다. 마지막으로, Displacement와 Immediate는 피연산자에서 상수 값이나 메모리 오프셋(offset)을 표현하는 필드이다.

두 명령어 "MOV %RAX, 0x1"와 "MOV %RSI, 0xA"을 예시로 설명한다면, 두 명령어는 공통적으로 레지스터에 Immediate 값을 이동시키는 명령어(i.e., MOV)로서 0xC7의 Opcode 값을 가진다. 또한, MOV 명령어는 피연산자로 레지스터와 Immediate 값이 사용되기 때문에, ModR/M과 Immediate 필드가 필요하다. "MOV %RAX, 0x1"은 RAX 레지스터를 사용하기 때문에 ModR/M 필드는 C0의 값을 가지며 Immediate 필드는 0x1의 값을 가지지만, 0xC7의 Opcode갖는

Table 1. UMIP bit-related Instruction Table

| Instruction | opcode | Description |
|---|---|---|
| SGDT m | 0f 01 /0 | Store GDTR to operand |
| SIDT m | 0f 01 /1 | Store IDTR to operand |
| SLDT m | 0f 01 /0 | Stores segment selector from LDTR in operand |
| STR r/m | 0f 00 /1 | Stores segment selector from TR in operand |
| SMSW r/m | 0f 01 /4 | Store machine status word to operand |

MOV 명령어는 Immediate 값으로 4바이트를 가질 수 있으므로 “MOV %RAX, 0x1”은 “48C7C001000000”으로 인코딩 된다. 앞선 설명과 마찬가지로 “MOV %RSI, 0xA”는 RSI 레지스터를 뜻하는 ModR/M 필드 값 C6과 Immediate 필드 값 0x0A를 가지기 때문에, “48C7C60A000000”으로 표현할 수 있다[24].

### 2.2 비 문서화 명령어 탐색 퍼저

비 문서화 명령어 탐색은 ISA에 명시되어 있지 않지만, 프로세서 내부에 구현이 되어있는 명령어들을 찾는 절차를 의미한다. 비 문서화 명령어임을 결정하기 위해서는 유효한 명령어들이 명세된 ISA 데이터 시트를 활용하여, 반대로 명세되지 않은 명령어들을 찾는데 활용할 수 있다. 이를, 디스어셈블러를 활용하여 자동화 탐색을 수행하는 퍼저(fuzzer)설계에 활용하는 다양한 선행 연구들이 발표되어왔다.

CISC 체계에 존재하는 비 문서화 명령어를 찾는 최초의 연구로는 2018년에 Domas Christoper의 Sandsifter가 발표되었다[16]. 이후 RISC 체계에서는 2020년 Dofferhoff의 iScanU가 발표 되었다[18]. 두 연구 모두 비 문서화 명령어를 결정하기 위해서 디스어셈블러를 활용하여 차등 테스팅을 공통적으로 사용하였다. 특히, 모든 명령어의 길이가 동일한 RISC 체계에서와 다르게, 비 균일 디코딩을 수행하는 CISC 체계에서는 퍼저가 비 문서화 명령어의 길이와 탐색 범위를 결정하는 기술이 요구된다.

탐색 문제는 CISC에서의 1~15byte의 가변 길이 명령어의 경우에 해당 범위는 $2^{120}$이 되어 탐색 범위가 매우 넓다는 문제가 존재한다. 이를 해결하기 위해 DFS(Depth First Search) 방식을 통해서 탐색 범위를 줄이는 방법이 제안되었다[16]. 해당 기법으로 $2^{120}$에 해당하는 탐색 범위를 약 10억개까지 효과적으로 줄였으며, 다양한 비 문서화 명령어를 제한된 시간에 찾아내었다.

비 문서화 명령어의 길이 결정을 위해서는 권한이 다른 두 개의 메모리 페이지(page)의 경계를 활용하여, 하드웨어 예외기반의 길이 결정 방법이 제안되었다[16]. 예를 들면, 실행 권한이 존재하는 페이지와 존재하지 않는 페이지의 경계에 명령어를 위치 시켜 실행함으로써 발생하는 예외를 분류하여, 비 문서화 명령어의 길이를 결정한다.

이후 CISC 체계의 비 문서화 명령어 탐색 시간을 더욱 단축하기 위해서, 탐색 범위를 효율적으로 결정하는 UISFuzz[17]와 Skipscan[14] 방식이 고안되었다. 2019년 Xixing Li의 UISFuzz는 명령어 길이 결정에 영향을 미치지 않는 Immediate와 Displacement 영역을 DFS로 탐색하지 않고, 명령어 데이터 베이스를 이용하여 명령어의 유효성을 재확인하며 속도와 신뢰성을 향상시켰다. 또한, 2023년에 발표된 Guang Wang의 Skipscan은 Prefix와 Opcode 조합을 구조적으로 접근하여, 유효한 조합을 찾는 방식을 사용했다. 그리고 UISFuzz와 마찬가지로 명령어 길이에 영향을 미치지 않는 Immediate와 Displacement 영역의 탐색을 건너뛰는 방식으로, 새로운 비 문서화 명령어를 효과적으로 탐색하였다. Skipscan과 UISFuzz는 기존의 탐색 방법을 크게 개선하였지만, 디스어셈블러와 차등 테스팅을 통해서 비 문서화 명령어를 결정하는 방식과 하드웨어 예외 정보를 바탕으로 명령어의 길이를 결정하는 방식의 개선은 연구하지 않았다.

### 2.3 디스어셈블러 신뢰도

비 문서화 명령어 탐색 퍼저는 비 문서화 명령어를 판단하기 위해서 생성된 후보 명령어를 프로세서와 디스어셈블러에 각각 실행한 후, 그 결과들에 대한 비교를 통해서 결정한다. 퍼저에 사용되는 디스어셈블러의 신뢰도는 이러한 차등 테스팅에서 매우 중요하게 작용한다. 하지만 디스어셈블러 (혹은 디코더)의 비 일관성에 관한 연구가

Table 2. Exception Handling Table

| Length Determination | Condition | | Description |
|---|---|---|---|
| | Exception | Exception Address | |
| On going | #PF | Page Boundary | The length of the instruction is not determined yet |
| Done | #PF | Not Page Boundary | Determined as a valid instruction |
| Done | #PG | Don't Care | Determined as a privileged instruction |
| Done | #UD | Don't Care | Determined as an invalid instruction |
| Done | #BP | Don't Care | Determined as a valid instruction |

진행되었고[22,23], Sandsifter에서 사용된 디스어셈블러 Capstone의 신뢰도가 낮은 모습을 보였다. 그와 반대로 Intel 사의 디스어셈블러 XED는 가장 높은 신뢰도를 보였다.

### 2.4 UMIP 비트 관련 명령어

X86-64 아키텍처의 CR4 (Control Register 4) 레지스터의 UMIP (User Mode Instruction Prevention) 비트는 Intel 7세대 , AMD Zen 아키텍쳐 기반 프로세서에서 처음 도입되었다. UMIP 비트는 유저모드에서 사용되던 Table 1.의 명령어들의 유저모드 사용 제한에 사용되었다.

SGDT와 SLDT 명령어는 각각 GDTR(Global Description Table Register)과 LDTR(Local Description Table Register)를 지정된 오퍼랜드에 저장시키는 명령어이다. GDTR과 LDTR은 각각 GDT와 LDT의 주소를 담고 있는 레지스터로 크기가 10byte이다. GDT와 LDT는 보호모드에서 사용되는 테이블로 베이스 주소, 세그먼트 제한 길이 들의 정보를 담고 있으며, GDT는 커널공간에서 LDT는 유저공간에서 사용된다.

SIDT 명령어는 IDTR(Interrupt Descriptor Table Register)를 오퍼랜드에 저장하는 명령어이다. IDT는 인터럽트가 발생했을 때 처리해주는 함수의 루틴을 포함하고 있는 테이블로써, 시스템상에 존재하는 프로세스 별로 독립적인 개별 테이블을 갖는다.

SMSW 명령어는 CR0 레지스터의 0 비트에서 15 비트까지의 영역, 즉 MSW (Machine Status Word)를 오퍼랜드의 저장하는 명령어이다. MSW는 시스템 상태와 관련된 비트를 저장하며, 주로 시스템모드와 가상 메모리 관련 시스템의 동작을 제어한다.

STR 명령어는 TR(Task Register)의 세그먼트 셀렉터를 오퍼랜드에 저장하는 명령어이다. TR은 태스크 스위치 및 보호모드 전환과 관련된 정보를 저장하는데 사용되는 레지스터이다.

## III. 하드웨어 예외 기반 명령어 길이 결정 기법 및 비 문서화 명령어 판단 기준

Sandsifter는 메모리 경계를 사용해서, 하드웨어 예외 기반 명령어 길이 결정 기법을 사용하여 효율적인 탐색을 진행하였다. 그 결과 CISC의 탐색 범위인 $2^{120}$개를 약 10억 개로 줄여내는 성과를 보였다[4,5]. 하드웨어 예외 기반 길이 결정 기법은 물리적으로 연속적인 두 개의 메모리 페이지의 경계에 길이가 15byte인 후보 byte를 위치시키고, 1byte씩 움직이며 실행할 때 발생하는 하드웨어 예외 정보를 바탕으로 길이를 판단하는 방식이다. 예를 들면, 4KB의 크기에 해당하는 메모리 페이지 $M_1$, $M_2$를 Fig. 2.와 같이 연속적으로 할당한다. 그리고 $M_1$에는 읽기, 쓰기, 실행 권한을 $M_2$에는 읽기, 쓰기 권한만을 준다. 이후 후보 byte가 생성되면, Fig. 2.의 (1)과 같이 후보 byte의 첫 번째 바이트만을

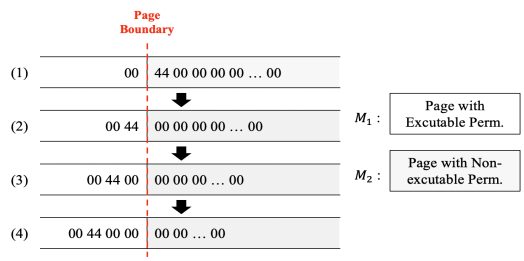


Fig. 2. Exception based Instruction Length Determination

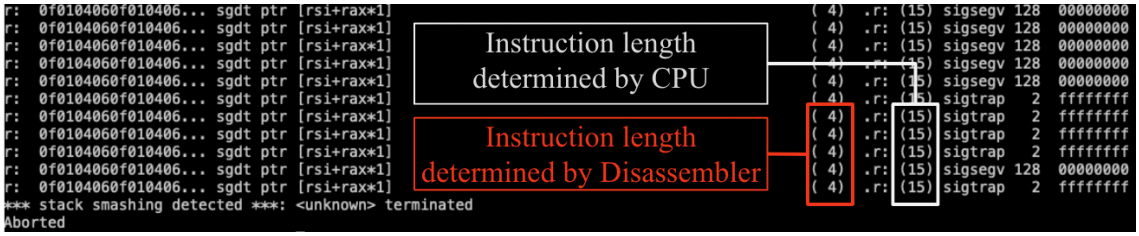


Fig. 3. Failed Case of Sandsifter

$M_1$ 영역 끝에 위치시키고, 나머지 바이트는 $M_2$ 영역 앞에서부터 순서대로 배치한다. 후보 byte를 실행시키기 전 모든 GPR(General Purpose Register)를 0으로 초기화 및 후보 byte 실행 후 무조건 예외가 발생하도록 설정한다. 이후, 후보 byte를 실행했을 때 발생한 예외와 예외 발생 주소를 바탕으로, Table 2.에 기술된 분류를 비교하며 길이 결정 여부를 판단하면서 동작을 이어나간다. 만약 명령어를 구성하는 모든 바이트가 실행 가능한 영역 $M_1$에 위치하지 않는다면, 프로세서는 #PF(페이지 폴트) 예외를 발생시킨다. 하드웨어 예외 기반 길이 결정 기법은 #PF가 발생하게 되면 Fig. 2.의 (2), (3), (4)와 같이 한 바이트를 왼쪽으로 이동시키면서 페이지 폴트 예외가 페이지 경계 부분에서 발생하지 않을 때까지 반복한다. 실행 후 예외가 $M_2$ 첫 번째 주소인 메모리 경계 부분에서 발생한 #PF 외의 다른 예외가 발생하면 메모리 경계 기법은 후보 byte중 메모리 $M_1$에 위치해 있는 byte 수 만큼 해당 명령어의 길이 및 하나의 명령어로 인식하고 명령어 길이 결정을 끝마치게 된다. 이후 Sandsifter는 수신된 마지막 예외의 정보와 디스어셈블러의 명령어 디코딩 결과를 종합하여 비 문서화 명령어의 여부를 판단한다. 하드웨어 기반의 명령어 길이 결정 기법은 후보 byte를 실행하여 발생하는 예외를 분류하여, 명령어의 길이와 명령어의 구성 byte를 판단한다.

본 연구에서는 단순하고 효과적인 방식처럼 보이는 예외 기반의 명령어 길이 결정 기법이 최신 프로세서를 대상으로 길이 결정에 긍정 오류를 보이는 한계가 존재하는 것을 관찰하였다.

## IV. 메모리 경계 기법 한계

본 장에서는 선행 연구가 갖는 예외 기반 명령어 길이 결정 기법의 한계를 보이는 실험을 진행한다. 특히, 실험에서 사용된 퍼저는 기존 Sandsifter가 사용한 Capstone 디스어셈블러 대신 신뢰도가 높은 Intel XED 디스어셈블러를 사용하였다[22,23]. 이를 통해 비 문서화 명령어 퍼징 결과에 대한 신뢰성을 높이고, 기존 연구에 사용된 명령어 길이 결정 기법으로 최신 아키텍처에서 검증 실험을 수행한다.

### 4.1 실험 환경

실험 환경으로는 Linux kernel version 5.15.0 그리고 Sandsifter의 기존 디스어셈블러 Capstone 버전 대신 Intel XED 디스어셈블러로 교체된 버전을 사용하였다[25]. 프로세서는 CISC 체계의 대표적인 Intel(13세대 i9-13900K)과 AMD(Zen 4 Ryzen 9 5950X)의 최신 프로세서를 대상으로 실험을 진행하였다.

### 4.2 실험 방법론 및 관찰

Intel과 AMD 프로세서에서 하드웨어 예외 기반 명령어 길이 결정 기법으로 탐색을 진행하였을 때, UMIP 관련 명령어인 SGDT 명령어에서 메모리 경계 기법을 통한 명령어 길이 측정을 실패하는 것을 관찰하였다. 또한, SGDT 명령어와 같이 UMIP 관련 명령어인 SLDT, STR, SIDT, SMSW 명령어들에 대해서도, 길이 결정이 실패하는 것을 관찰하였다. Fig. 3.에서 Sandsifter가 길이 결정을 잘못 판단하는 상황이며, 디스어셈블러와 CPU가 판단한 길이가 다른 것을 볼 수 있다.

Sandsifter는 원활한 퍼저의 동작과 분석을 위해서 널 포인터(0x0) 접근에 대한 탐지를 활성화하거나 비활성화할 수 있다. 예외 기반 명령어 길이 결정의 오류가 널 포인터에 대한 접근 권한과 관련 있는 현상인지 확인하기 위해서, 해당 널 포인터 접근의 활성화 또는 비활성화로 나누어 실험을 진행하였다. SGDT, SIDT, SLDT 명령어들은 메모리 접근이 가능한 명령어 들이며 Sandsifter에 널 포인터 접

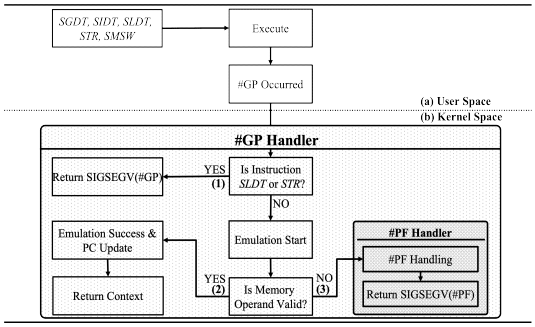


Fig. 4. A Block Diagram of Exception Control Flow of UMIP bit-related Instruction

근 허용을 활성화 상태로 실행하였을 때 명령어 길이 측정 실패를 관찰하였고, 널 포인터 접근을 비활성화 상태로 실행하였을 때는 명령어 길이 측정 성공을 관찰하였다.

SMSW와 STR 명령어는 오퍼랜드가 메모리를 참조하는 경우에는 널 포인터 접근이 활성화 상태로 실행하였을 때 명령어 길이 측정이 실패하는 것을 관찰하였고, 널 포인터 옵션을 비활성화 상태로 실행하였을 때는 명령어 길이 측정이 성공하는 것을 관찰하였다. 하지만 SMSW와 STR명령어는 오퍼랜드에 메모리 접근이 아닌, 목적 레지스터를 지정할 수 있는데 이때는 널 포인터 옵션과 상관없이 명령어 길이 측정에 실패하였다. 따라서, 단순 널 포인터 접근의 유무에 의존하는 현상이 아니었음을 알 수 있었다.

### 4.3 예외 기반 명령어 길이 결정 오류 및 원인 분석

Sandsifter가 명령어 길이 측정에 실패한 명령어들은 UMIP 관련 명령어들이며, 명령어 길이 측정 실패의 원인은 운영체제인 리눅스 커널에 존재한다. UMIP 비트가 도입되기 전의 프로세서에서, 유저 모드에서 SGDT, SIDT, SMSW, STR, SLDT 명령어들을 사용하는 다양한 프로그램이 존재하였다. 하지만 UMIP 비트 도입 이후 해당 명령어들은 더 이상 유저모드에서 사용을 할 수 없었다. 그래서 리눅스 커널은 이러한 프로그램을 지원해주기 위해 Table 1.에 있는 모든 UMIP 관련 명령어를 커널 수준에서 소프트웨어 에뮬레이션을 수행해 주었다. 그래서 해당 명령어를 실행하면 권한 부족으로 발생되는 #GP 예외를 커널 내부의 예외 핸들러에서 캐치하여, 유저 공간의 프로그램에 전달하지 않는다. 따라서 예외 기반으로 길이를 결정하는 Sandsifter는 Table 2.의 분류와 비교를 수행하지 못하게 되어 명령어 길이 결정을 하지 못한다. 대신, 커널은 마치 #GP 예외 없이 수행되는 것처럼 커널 수준에서 해당 명령어를 에뮬레이션을 함으로서, 유저 공간에 존재하는 프로그램에게는 정상적으로 실행되는 것처럼 보이는 환상을 제공한다.

예를 들면, Sandsifter의 하드웨어 예외 기반 명령어 길이 결정 기법은 실행 가능 영역에 위치해 있는 후보 byte를 실행하여 발생하는 예외를 Table 2.를 기준으로 명령어 길이를 결정한다. 이 방식은 기본적으로 실행 가능 영역에 위치한 후보 byte가 유효하던 유효하지 않던 실행한다면 무조건 예외가 발생되게 구성이 되어있다. 따라서, Table 2.와 같은 기준을 세워 명령어의 유효성 및 길이를 판단 할 수 있었다.

하지만 리눅스 커널이 UMIP 관련 명령어를 에뮬레이션하면서 기존에 발생하였던 #GP 예외는 사라지게 되고, 명령어가 실행된 것처럼 PC값에 명령어 길이 만큼의 값이 더해지고 정상적인 실행 흐름이 반환되게 된다.

그러면 프로세서는 문제없이 명령어를 실행한 것으로 인식하고, 자연스럽게 다음 byte를 디코더로 패치 해 올 것이다. 하지만 다음 byte는 실행 불가능한 영역에 위치해 있는 byte이기 때문에 #PF 예외가 페이지 경계에서 발생하게 된다. 이때 Sandsifter가 인식할 수 있는 예외가 발생하였기 때문에 메모리 경계 기법은 이를 인식하여 명령어의 길이를 Table 2.의 기준에 맞춰 판단하고, 페이지 경계에서 발생한 #PF 예외는 명령어 길이 측정이 끝나지 않았음을 나타내기 때문에 계속해서 명령어 길이 측정을 진행하게 되고 명령어 길이 측정에 실패하게 된다.

결과적으로, UMIP 관련 명령어에 대한 커널 수준에서의 에뮬레이션이 예외 기반의 명령어 길이 결정 방식에 영향을 미치게 되는 것이다. 하지만 앞선 실험에서 Sandsifter에 널 포인터 접근을 비활성화 상태로 실행하였을 때 SGDT, SIDT, SMSW는 길이 측정이 잘 되는 것을 볼 수 있다.

이는 Sandsifter는 후보 byte를 실행하기 전에 모든 GPR(General Purpose Register)을 0으로 초기화 하기 때문에 SGDT, SIDT, SMSW는 에뮬레이션 될 때 커널이 내부적으로 정의한 더미 값(예: 0xFFFFFFFFFFFE0000)이 주소 0 즉 널 포인터에 저장이 되는데, 이때 널 포인터에 쓰기 권

**Algorithm 1** PC & Exception-based Instruction Length Determination algorithm

```
1  : Current Length ← 0
2  :  for all Current Length ∈ Instruction Max Length do
3  :      Execute Candidate byte for Current Length
4  :      if Exception is #GP && Address is page boundary || PC not updated
5  :      then
6  :          Continue Loop
7  :      else
8  :          break Loop
9  :      end if
10 :  end for
```

Fig. 5. PC & Exception based Instruction Length Determination Algorithm

**Algorithm 2** PMU & Exception-based Instruction Length Determination algorithm

```
1  : Current Length ← 0
2  : PMU Event Counter ← 0
3  :  for all Current Length ∈ Instruction Max Length do
4  :      Start counting PMU Event Counter
5  :      Execute Candidate byte for Current Length
6  :      End counting PMU Event Counter
7  :      if Exception is #GP && Address is page boundary
          || PMU Event Counter not updated
8  :      then
9  :          continue Loop
10 :      else
11 :          break Loop
12 :      end if
13 :  end for
```

Fig. 6. PMU & Exception based Instruction Length Determination Algorithm

한 없이 값을 저장하게 되면 Fig. 4.의 (3) 처럼 명령어 에뮬레이션이 중단되고 실행 흐름이 #PF 핸들러에 넘어가게 된다. 그러면 #PF가 핸들링 되면서 유저 공간에 있는 Sandsifter가 널포인터에서 #PF가 발생했다는 것을 알 수 있기 때문에 명령어 길이 측정을 성공할 수 있었던 것이다. 따라서, 만약 GPR을 0이 아닌 다른 커널 영역의 가상주소로 초기화를 하게 되면, 길이 측정이 실패하게 된다.

AMD 환경에서는 Intel 환경에서와 다르게 SLDT와 STR에서도 길이 결정의 실패를 관측하였고, 이의 원인이 Linux kernel version의 차이였다. AMD는 Linux kernel version 5.15.0이 설치되어 있는데 해당 버전에서는 Intel 시스템의 kernel version 5.4.0과 달리 모든 UMIP관련 명령어를 에뮬레이션 해주었기 때문이다. 그래서 AMD에서 SLDT와 STR을 실행하였을 때 Intel과 마찬가지로 #GP 예외를 인식하지 못하였고 길이 결정에 실패한 것이였다. 이를 인지하고 AMD 시스템의 kernel version을

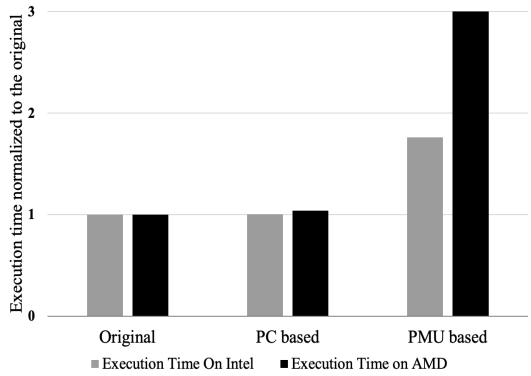


Fig. 7. Execution Time Overhead on Both Intel and AMD

5.15.0에서 5.4.0으로 교체 후 실험을 진행하였고, 결과는 Intel과 동일하게 관찰되었다. 결과적으로, 하드웨어 예외 기반 명령어 길이 결정 방법은 UMIP 관련 명령어에 대한 커널 에뮬레이션뿐만 아니라, 커널 버전에 따라서도 긍정 오류가 발생함을 관찰하였다.

### 4.4 하드웨어 예외 기반 명령어 길이 결정 기법의 한계

분석에 따르면 Sandsifter의 메모리 경계 기법은 한정적인 정보로 명령어를 판단하는 방식으로 예외적인 상황에 대처하지 못하는 한계를 보여줬다. Sandsifter의 명령어 길이 측정 실패 사례에 따르면, 예외와 주소만을 핸들링하는 것으로는 후보 byte의 길이를 명확하게 측정할 수 없었다. 한정적인 정보로 후보 byte의 길이를 측정하는 메모리 경계 기법의 한계를 극복하기 위해서는 더 고차원적인 기법이 필요하다. 예외 정보 외의 추가적인 정보를 분석하여 길이를 판단하거나, 아니면 더욱 직관적인 정보를 이용하여 후보 byte의 길이를 결정하는 방식을 사용하는 것이 적절해 보인다. 따라서, 본 연구에서는 하드웨어 예외뿐만 아니라, 명령어의 실행 정보를 바탕으로 명령어 길이 결정 기법의 정확성을 높이는 방법을 제안한다.

## V. 실행 정보 기반 명령어 길이 결정 기법

하드웨어 예외 기반 명령어 길이 결정 기법에서 예외 정보만으로 후보 byte의 길이를 제대로 측정해

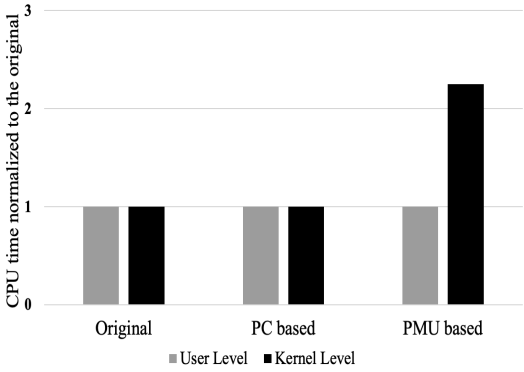


Fig. 8. User Level and Kernel Level Overhead in CPU Time on Intel

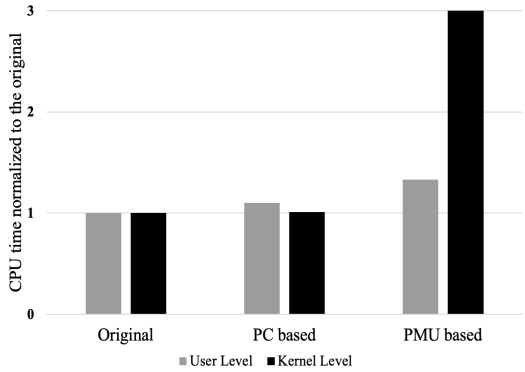


Fig. 9. User Level and Kernel Level Overhead in CPU Time on AMD

내지 못하는 한계를 극복하기 위해 PC값과 Performance Counter를 사용하여 Instruction Counter의 값 변화를 추가적으로 관찰하는 방식을 제안한다. 유저 공간에서 UMIP 관련 명령어가 실행될 때 권한 부족으로 인한 #GP 예외가 발생한다. 하지만 커널의 UMIP 관련 명령어 에뮬레이션으로 유저 공간은 #GP 예외를 인식할 수 없고, 이것이 많은 선행 연구에서 사용하고 있는 예외 기반 명령어 길이 결정 기법의 한계이다. 따라서, 본 연구에서는 예외 정보뿐만 아니라, 실제 하드웨어적으로 실행된 명령어의 정보를 관측할 수 있는 하드웨어 성능 카운터와 프로그램 카운터를 이용하여 명령어 길이를 결정한다. 두 방식 모두 하드웨어가 제공하는 정보를 기반으로 길이를 결정하는 방식이기 때문에, 신뢰성이 높은 방식이다.

### 5.1 PC(Program Counter) 기반 명령어 길이 결정 방식

커널이 에뮬레이션을 완료하고 Fig. 4.의 (2)와 같이 유저 공간에 실행 흐름을 반환할 때 유저 공간은 #GP 예외를 인식하지 못하여도 PC값의 변화를 인식할 수 있으며, 이는 유저 공간에서 하나의 명령어가 실행되었음을 간접적으로 알 수 있다. 그래서 기존 조건문에서 "PC not updated"의 조건을 추가하여 명령어가 커널 영역에서 에뮬레이션 되어 예외가 전달되지 않더라도, 유저 공간에서 이를 인식하고 대처할 수 있도록 Fig. 5.와 같이 새로운 기법을 제안한다.

### 5.2 PMU(Performance Monitoring Unit) 기반 명령어 길이 결정 방식

PC값의 변화 관측 방식뿐만 아니라 Performance Counter의 Instruction Counter 이벤트의 측정 또한 유저 공간에서 명령어의 실행 여부를 추적할 수 있기 때문에, Fig. 6.과 같이 PMU 기반 명령어 길이 결정 기법을 추가적으로 제안한다. 대부분의 Intel 및 AMD 프로세서는 은퇴 명령어(retired instruction)에 대해서 측정을 할 수 있는 이벤트를 제공하며, 해당 이벤트는 커널이 에뮬레이션 해주는 UMIP 관련 명령어 또한 문제없이 측정되는 것을 확인하였다. 그렇기에 PMU 기반 명령어 길이 결정 기법은 커널이 명령어를 에뮬레이션하는 상황에 대해서도 유효하였다.

## VI. 성능 평가

PC 기반 명령어 길이 결정 기법과 PMU 기반 명령어 길이 결정 기법을 각각 Intel(13세대 i9-13900k)와 AMD(Zen 4 Ryzen 9 5950X)시스템에서 성능을 측정하였다. 성능 측정 범위를 기존 기법이 UMIP 관련 명령어에 대한 길이 결정을 정확하게 측정하지 못하여 퍼저가 정상 작동하지 않으므로, UMIP 명령어 직전까지로 제한하였다. 그리고 성능은 Linux Perf tool을 사용하여 퍼저의 전체적인 실행을 기준으로 실행 시간(execution time)과 CPU 시간을 측정하였다. 실행 시간은 기존 기법을 기준(값 : 1)으로 PC 기반 기법과 PMU 기반 기법의 값을 환산하였다. Fig. 7.에 따르면

Intel 시스템에서 퍼저의 전체적인 실행시간은 PC 기반 기법이 기존 기법에 비해 약 0.4%의 성능이 저하가 되었으며, PMU 기법은 기존 기법에 비해 약 75%의 성능이 저하되었다. AMD 시스템에서 실행 시간은 PC기반 기법이 기존 기법에 비해 약 3% 성능이 저하되었으며, PMU 기반 기법은 약 200%의 성능이 저하되었다. PMU 기반 기법의 성능 하락이 두 시스템에서 상당히 높게 측정되는 이유는 PMU가 커널모드에서만 사용이 가능하기 때문에 유저모드에서 사용하려면 syscall로 구현된 인터페이스를 이용해야 되기 때문이다. Fig. 8, 9.의 결과와 같이 Execution Time에서 PMU 기반 기법의 Kernel 영역 비율이 기존 기법보다 2~3배가량 증가하기 때문에 높은 성능 저하가 발생하는 것이며, 이는 하드웨어 PMU의 사용은 리눅스 커널 내에 구현되어 있는 Perf. subsystem 코드의 수행이 요구되고, 이를 사용하기 위한 syscall의 오버헤드가 가중되어 성능 저하가 발생된 것으로 분석되었다.

또한 새롭게 추가한 PC값 및 Instruction Counter의 조건이 기존 정상적으로 작동하였던 조건인 #GP, #UD, #BP 예외 상황에는 영향을 미치지 말아야 한다. 왜냐하면 새롭게 추가한 조건은 명령어 에뮬레이션 상황을 퍼저가 인식할 수 있도록 돕는 특수 상황의 조건이며, 해당 특수 상황의 조건이 기존의 작동방식에 영향을 준다면 또 다른 문제가 야기될 가능성이 있기 때문이다. 하지만 추가한 두 조건들은 #GP, #UD, #BP 예외에서 필요충분조건 관계이기 때문에 서로 영향을 미치지 않는다. 새롭게 제안한 실행 정보 기반 명령어 결정 기법은 UMIP와 같은 예외를 숨기고 명령어가 실행한 것처럼 보여지는 커널 에뮬레이션 상황에 대한 특수 상황 조건일 뿐 프로세서에 존재 혹은 숨어있는 모든 명령어의 길이를 완벽하게 분석하는 범용적인 조건이 아니다.

## VII. 결 론

비 문서화 명령어를 탐색하는 퍼저의 핵심 기법인 하드웨어 예외 기반 명령어 길이 결정 기법이 예외적인 상황을 처리하지 못하여 명령어의 길이를 정확하게 측정하지 못하는 한계를 보여주었다. 이는 하드웨어 예외 기반 명령어 길이 결정 기법이 단일적인 정보로 명령어의 길이를 판단하였기 때문이다. 이러한 한계는 프로세서에 존재하는 문서화 명령어와 비 문서화 명령어를 제대로 탐색할 수 없으며, 비 문서화 명령어 탐색 신뢰성에도 큰 영향을 미칠 수 있다. 그렇기에 본 논문에서는 하드웨어 예외 기반 명령어 길이 결정 기법의 한계를 정밀 분석하고 예외적인 상황을 처리할 수 있도록 하는 실행 정보 기반 명령어 길이 결정 기법을 제시하였다. 하지만 이는 프로세서에 존재하는 모든 명령어를 정확하게 분석할 수 있는 범용적인 기법이 아니며, 이러한 한계를 극복하여 프로세서에 존재하는 명령어를 더욱 신뢰도 높게 탐색하기 위해서는 더 고차원적이고 수준 높은 기법이 필요할 것으로 보인다.

## References


[1] Lipp M, Schwarz M, Gruss D, Prescher T, Haas W, Mangard S, Kocher P, Genkin D, Yarom Y and Hamburg M, “MeltdownL Reading Kernel Memory from User Space,” 27th USENIX Ssecurity Symposium (USENIX Security 27), pp. 972-990, Jan. 2018

[2] Kocher P, Horn J, Fogh A, Genkin D, Gruss D, Haas W, Hamburg M, Lipp M, Mangard S, Prescher T and Schwarz M, “Spectre attacks: Exploiting speculative execution,” Communications of the ACM, pp. 93-101, Jun. 2020

[3] S. van Schaik, M. Minkin, A. Kwong, D. Genkin and Y. Yarom, ″CacheOut: Leaking Data on Intel CPUs via Cache Evictions,″ 2021 IEEE Symposium on Security and Privacy (SP), pp. 339-354, May. 2021

[4] Lipp M, Gruss D and Schwarz M. “{AMD} prefetch attacks through power and time,” 31st USENIX Security Symposium (USENIX Security 22), pp. 643-660, Aug. 2022

[5] Jang Y, Lee S and Kim T. “Breaking kernel address space layout randomization with intel tsx,” 2016 ACM SIGSAC Conference on Computer and Communications

Security, pp.380-392, Oct. 2016
〔6〕 Moghimi D. "Downfall: Exploiting Speculative Data Gathering," 32nd USENIX Security Symposium (USENIX Security 23), pp. 7179-7193, Aug. 2023
〔7〕 Murdock K, Oswald D, Garcia FD, Van Bulck J, Gruss D and Piessens F. "Plundervolt: Software-based fault injection attacks against Intel SGX," 2020 IEEE Symposium on Security and Privacy (SP), pp. 1466-1482, May. 2020
〔8〕 Dunlap G. "The Intel SYSRET privilege escalation," Xen Project, pp.1-7, Jun. 2012
〔9〕 Hur J, Song S, Kwon D, Baek E, Kim J and Lee B, "Difuzzrtl: Differential fuzz testing to find cpu bugs," 2021 IEEE Symposium on Security and Privacy (SP), pp. 1286-1303, May. 2021
〔10〕 Xu J, Liu Y, He S, Lin H, Zhou Y and Wang C, "{MorFuzz}: Fuzzing Processor via Runtime Instruction Morphing enhanced Synchronizable Co-simulation," 32nd USENIX Security Symposium (USENIX Security 23), pp. 1307-1324, Aug. 2023
〔11〕 Kande R, Crump A, Persyn G, Jauernig P, Sadeghi AR, Tyagi A and Rajendran J, "{TheHuzz}: Instruction Fuzzing of Processors Using {Golden-Reference} Models for Finding {Software-Exploitable} Vulnerabilities," 31st USENIX Security Symposium (USENIX Security 22),pp. 3219-3236, Aug. 2022
〔12〕 Ermolov M, Sklyarov D and Goryachy M. "Undocumented x86 instructions to control the CPU at the micro-architecture level in modern Intel processors," Journal of Computer Virology and Hacking Techniques, pp. 1-5, Aug. 2022
〔13〕 C. Domas, "Hardware backdoors in x86 cpus," Black Hat, pp. 1-14, Jul. 2018
〔14〕 Wang G, Zhu Z, Cheng X and Meng D, "A High-coverage and Efficient Instruc- tion-level Testing Approach for x86 Processors," IEEE Transactions on Computers, pp. 3203-3217, Jun. 2023
〔15〕 Y. Wang, P. Liu, W. Wang, X. Wang, and Y. Jiang, "On a consistency testing model and strategy for revealing risc processor's dark instructions and vulnerabilities," IEEE Transactions on Computers, pp.1586-1597 Jul. 2021
〔16〕 C. Domas, "Breaking the x86 isa," in Black Hat, Jun. 2017
〔17〕 Li X, Wu Z, Wei Q, and Wu H. "Uisfuzz: An efficient fuzzing method for cpu undocumented instruction searching," IEEE Access, pp. 149224-149236. Oct. 2019
〔18〕 Dofferhoff R, Göebel M, Rietveld K and Van Der Kouwe E. "iscanu: A portable scanner for undocumented instr- uctions on risc processors," 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 306-317, Jun. 2020
〔19〕 S. Drzevitzky, "Proof-Carrying Hardware: Runtime Formal Verification for Secure Dynamic Reconfiguration," 2010 International Conference on Field Programmable Logic and Applications, pp. 255-258 Aug. 2010
〔20〕 Xiaolong Guo, Raj Gautam Dutta, Yier Jin, Farimah Farahmandi, and Prabhat Mishra, "Pre-silicon security verification and validation: a formal

perspective," Proceedings of the 52nd Annual Design Automation Conference (DAC '15). Association for Computing Machinery, pp. 1-6, Jun. 2015
[21] PALEARI, Roberto, et al, "N-version disassembly: differential testing of x86 disassemblers," Proceedings of the 19th international symposium on Software testing and analysis, pp. 265-274, Jul. 2010
[22] WOODRUFF, William; CARROLL, Niki; PETERS, Sebastiaan., "Differential analysis of x86-64 instruction decoders," 2021 IEEE Security and Privacy Workshops (SPW), pp. 152-161, May. 2021
[23] WANG, Guang, et al. "Differential testing of x86 instruction decoders with instruction operand inferring algorithm," 2021 IEEE 39th International Conference on Computer Design (ICCD), pp. 196-203, Oct. 2021
[24] INTEL, "Intel® 64 and IA-32 Architectures Software Developer's Manual", https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, Jul. 2022
[25] Github, "XED Sandsifter", https://github.com/cattius/sandsifter.git, Feb. 2023

## 〈저자소개〉

이 유 석 (Yoo-seok Lee) 학생회원
2018년 3월~현재: 강원대학교 컴퓨터공학과 학사과정
〈관심분야〉 정보보호, 컴퓨터구조, 시스템보안

송 원 준 (Won-jun Song) 종신회원
2012년 2월: 성균관대학교 전자전기컴퓨터 공학부 졸업
2014년 2월: 한국과학기술원 석사
2018년 2월: 한국과학기술원 박사
〈관심분야〉 시스템보안, 컴퓨터구조, 운영체제