디바이스 드라이버에 대한 퍼징 연구 동향

김성원[1], 오현영[2*]
[1]가천대학교 AI · 소프트웨어학과 석사과정
[2*]가천대학교 AI · 소프트웨어학과 교수

vubinstar468@gachon.ac.kr, hyoh@gachon.ac.kr

Recent Advances in Device Driver Fuzzing Research

Sung-Won Kim, Hyun-Young Oh
Dept. of AI · Software, GaChon University

요 약

디바이스 드라이버는 OS 보안의 핵심 요소로, 커널 수준에서 실행되며 중요한 공격 표면을 형성하지만 테스트가 어렵다. 본 논문은 세 가지 대표적인 연구를 비교하여, 동적 프로빙과 다양한 I/O 인터페이스를 아우르는 디바이스 드라이버 fuzzing 기법의 최신 동향을 분석한다.

1. 서론

현대 운영체제(OS)는 다양한 하드웨어 장치와 호환성을 제공하기 위해 수많은 디바이스 드라이버를 포함한다. 예를 들어 Ubuntu Linux 20.04의 드라이버 디렉터리에는 약 1300만 줄의 코드가 포함되어 있으며 전체 리눅스 소스 코드의 64.8%에 해당한다. 이처럼 방대한 코드베이스는 공격 표면을 크게 늘리며, 실제로 최근 5년간 리눅스 커널의 CVE 보고서 중 27-54%가 디바이스 드라이버 취약점이다. 디바이스 드라이버는 커널 모드에서 실행되며 주변 장치로부터 입력을 직접 처리하므로, 장치의 입력을 제대로 검증하지 못하면 시스템 전체 보안이 위협받는다. 실제 사례로 ThunderClap[1] 등 공격에서는 악의적인 PCI 장치로 시스템 메모리를 유출했으며, PS3/PS4 해킹은 USB/PCIe 장치를 통해 커널을 장악하였다. 그러나 기존 드라이버 테스트 솔루션은 한계가 있다. P2IM[2]·DICE[3] 등 일부 연구는 임베디드 펌웨어를, Charm[4]·DIFUZE[5]는 실제 장치를 필요로 하며, USBFuzz[6] 등은 수동 모델을 요구한다. 이러한 방식들은 범용적인 드라이버 fuzzing에는 적합하지 않다. 본 논문에서는 디바이스 드라이버 fuzzing의 최신 기법을 조사하고, 두 가지 과제인 동적 프로빙과 다양한 I/O 인터페이스에 대한 해결책을 다룬다. 사례 연구로 USBFuzz[6], PeriScope[7], DEVFUZZ[8]를 살펴본다.

2. 배경지식

2.1 디바이스 드라이버

디바이스 드라이버는 OS 커널의 특권 모드에서 실행되며, 하드웨어 장치와 운영체제 간의 인터페이스 역할을 한다. 드라이버는 CPU의 메모리 공간을 통해 장치 레지스터와 통신한다. 주로 메모리 매핑 I/O(MMIO)와 포트 I/O(PIO)의 두 가지 형태로 장치 레지스터를 읽고 쓰며, 대용량 데이터 전송을 위해 DMA(Direct Memory Access)를, 비동기 이벤트 처리를 위해 인터럽트(IRQ)를 사용한다. 이러한 다양한 I/O 방식이 드라이버 fuzzing에서 과제가 된다.

2.2 퍼징(fuzzing)

fuzzing은 프로그램에 무작위 또는 변형된 입력을 지속적으로 제공하여 결함을 찾는 동적 테스트 기법이다. AFL(American Fuzzy Lop)[9] 같은 도구는 coverage 기반 변이 기법을 활용하여 광범위한 소프트웨어에 효과적인 것으로 알려져 있다. 그러나 디바이스 드라이버 fuzzing 에서는 하드웨어/소프트웨어 경계 문제가 추가된다. 예를 들어 Syzkaller[10]와 같은 커널 fuzzer 는 주로 시스템 콜 경계를 다루며, 실제 장치-드라이버 간

* 교신저자

입출력 인터페이스를 모사하지 못한다. 따라서 장치 드라이버 fuzzing 은 일반 소프트웨어 fuzzing 과 달리 특별한 대응이 필요하다.

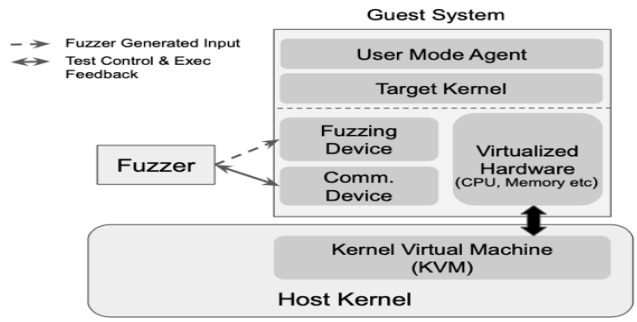
2.3 심볼릭 실행(symbolic execution)

심볼릭 실행은 프로그램을 실제 값 대신 기호(symbolic)로 실행하여 실행 경로의 조건식을 추출하고 모든 가능한 실행 경로를 탐색하는 기법이다. KLEE[11]나 S2E[12] 같은 도구는 특히 임베디드 시스템이나 폐쇄형 펌웨어에서 드라이버를 테스트할 때 유용한 것으로 평가된다. 심볼릭 실행은 입력 공간을 체계적으로 탐색할 수 있지만, 드라이버처럼 복잡하고 상태 공간이 큰 소프트웨어에는 확장성의 한계가 있다.

3. 연구사례

연구사례에서는 디바이스 드라이버 fuzzing 을 위한 대표적인 세 연구 USBFuzz, PeriScope, DEVFUZZ 의 접근 방식을 비교한다. PeriScope 는 실행 시 드라이버의 MMIO 및 DMA 접근을 후킹하여 하드웨어와의 상호작용을 동적으로 조작하며, USBFuzz 는 USB 장치의 에뮬레이션을 통해 OS 에 독립적인 fuzzing 환경을 제공한다. DEVFUZZ 는 심볼릭 실행과 정적, 동적분석을 결합해 물리 장치 없이도 프로빙과 다양한 I/O 경로를 포괄하는 자동화된 fuzzing 을 가능하게 한다.

3.1 USBFuzz[6]



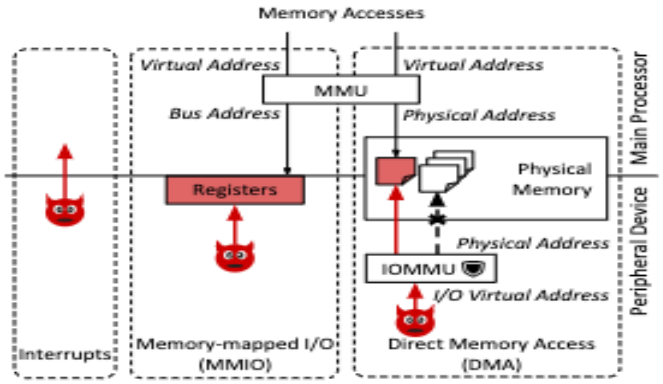
(그림 1) USBFuzz[6]의 개요

USBFuzz 는 USB 드라이버 fuzzing 을 위한 프레임워크로, USB 장치 에뮬레이션을 통해 드라이버에 난수 데이터를 공급한다. 그림 1 과 같이, USBFuzz 는 QEMU 기반의 가상 환경 위에서 작동하며, 소프트웨어 USB 호스트 컨트롤러와 USB 디바이스를 에뮬레이터 한다. 구성 요소로는 fuzzer 의 데이터를 드라이버에 전달하는 Fuzzing Device, fuzzer 와 가상 머신(VM) 간 coverage 정보를 교환하는 Communication Device, 그리고 커널 상태를 모니터링하는 User Mode Agent 로 이루어져 있다.

이 시스템은 실제 USB 장치에서 추출한 벤더/제품 ID 및 구성 정보를 기반으로 초기 설정을 수행하고, 이후 드라이버가 요청하는 데이터에 대해 fuzzer 가 생성한 난수를 응답으로 제공한다. 이 방식을 통해 USBFuzz 는 별도의 OS 커널 수정 없이 자연스럽게 입력을 공급(out-of-the-box)하여 표준에서 벗어난 입력 처리를 테스트할 수 있으며, 더 넓은 범위의 버그 탐지가 가능하도록 설계되었다. 또한 USBFuzz 는 커널 드라이버 코드의 Edge Coverage 를 AFL 방식으로 수정된 kcov 를 통해 수집하여, 비결정성을 극복하고 높은 정확도의 coverage 수집을 달성하였다. 그 결과, 리눅스에서 double-free, use-after-free, NULL pointer dereference 등 총 16 건의 심각한 memory bug 를 발견하였으며, 특히 XHCI 드라이버에서 발견된 무한 루프 메모리 할당 버그는 시스템 메모리를 완전히 고갈시키는 심각한 보안 문제였다.

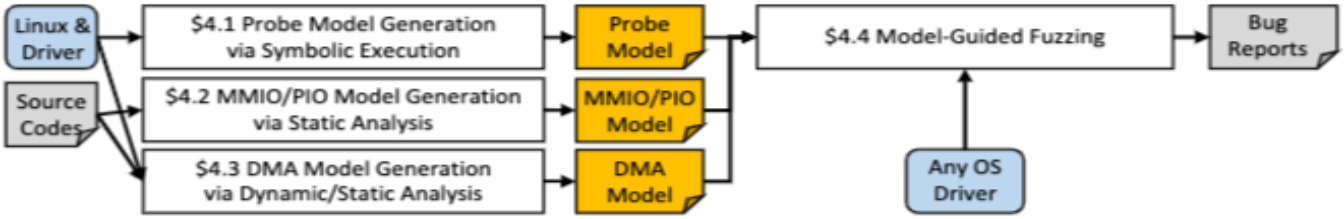
성능 측면에서도 fuzzing iteration 당 0.1-2.6 exec/sec 의 속도를 보였으며, Linux 에서 생성된 fuzzing 입력을 재활용하여 FreeBSD, macOS, Windows 환경에서도 드라이버를 성공적으로 테스트했다. 이를 통해 macOS 에서는 시스템이 예기치 않게 재부팅 되는 버그를, Windows 에서는 블루스크린을 유발하는 심각한 버그를 추가로 발견하였다.

3.2 PeriScope[7]



(그림 2) PeriScope[7]의 개요

PERISCOPE 는 하드웨어와 운영체제(OS)의 경계에서 발생하는 드라이버와 장치 간 상호작용을 정밀 분석하기 위한 프레임워크다. 그림 2 와 같이, 디바이스 드라이버는 MMIO 를 통해 주변 장치의 레지스터에 접근하거나, DMA 를 통해 물리 메모리에 접근하여 데이터를 읽고 쓴다. 이 과정에서 주변 장치는 인터럽트를 통해 CPU 에 이벤트를 알린다. PERISCOPE 는 이러한 MMIO/DMA 메모리 접근 과정에서 리눅스 커널의 페이지 폴트 처리를 후킹하여, 드라이버가 접근하려는



(그림 3) DEVFUZZ 의 개요[8]

시점에 인위적으로 페이지 폴트를 발생시킨다 미리 등록된 pre-instruction hook 을 호출하여 접근하려는 레지스터 주소와 접근 유형을 분석한 후 원래의 페이지 폴트 처리로 넘긴다. 또한 MMIO/DMA 영역이 할당될 때 관련 API 호출을 후킹하여 버퍼 목록을 관리한다.

이러한 후킹 방식의 클라이언트 모듈인 PERIFUZZ 는 드라이버가 MMIO/DMA 레지스터 값을 읽을 때 이를 가로채고, 실제 장치가 제공하는 값 대신 임의의 fuzz 값을 공급하여 메모리 손상(memory corruption)이나 이중 패치(double-fetch)와 같은 취약점을 효과적으로 탐지할 수 있다. 특히 PERIFUZZ 는 드라이버가 장치로부터 읽는 값만 조작하고 쓰는 값은 변경하지 않기 때문에, 악의적인 장치가 보내는 데이터를 모사하는 효과를 얻을 수 있다.

성능 및 적용 측면에서 PERISCOPE 는 하드웨어 및 디바이스의 종류에 무관한 범용적인 접근법으로, 가상 디바이스뿐만 아니라 실제 디바이스 드라이버에도 적용할 수 있다. 실제로 Wi-Fi 드라이버 등을 대상으로 PERIFUZZ 를 적용한 결과, 총 15 건의 취약점 중 9 건의 신규 취약점을 발견하는 성과를 거두었다.

3.3 DEVFUZZ[8]

DEVFUZZ 는 실제 물리적 장치 없이 다양한 디바이스 드라이버를 자동으로 fuzzing 하기 위해 모델 기반 접근 방식을 채택한 프레임워크다. 그림 3 과 같이 DEVFUZZ 는 우선 심볼릭 실행을 통해 드라이버의 프로빙 경로를 분석하고, 해당 경로를 성공적으로 통과하는 데 필요한 조건을 자동으로 계산하여 프로브 모델을 생성한다. 프로빙 과정에서 드라이버가 접근하는 MMIO/PIO 영역을 식별하여 이를 심볼릭 값으로 마킹하고, SMT solver 를 통해 경로의 제약식을 해결함으로써 드라이버 초기화에 필요한 레지스터 주소와 값을 자동으로 추론한다.

이후 DEVFUZZ 는 프로브 모델 뿐 아니라 정적 분석을 통해 MMIO/PIO 모델도 생성한다. 이 모델은 드라이버가 사용하는 MMIO/PIO 레지스터 값 중 특별히 의미 있는 매직 값(magic value), 경계 값(boundary condition), 특정 비트 패턴 등을 추출하여 입력 생성 범위를 효과적으로 좁히는 데 활용된다. DMA 모델은 정적 분석과 동적 실행 분석을 결합하여 구축하는데, DMA 버퍼가 할당되는 시점(dma_alloc_coherent)을 계측하고, 프로빙 단계에서 발견된 DMA 물리 주소가 MMIO 레지스터에 기록되는 순간을 추적하여 버퍼 구조와 위치를 파악한다.

이렇게 생성된 세 가지 모델(프로브, MMIO/PIO, DMA)은 AFL fuzzer 와 결합되어 실제 드라이버 fuzzing 에 활용된다. fuzzing 과정은 두 단계로 나뉜다. 첫 단계인 프로빙 단계에서는 생성된 프로브 모델만을 활용하여 드라이버 초기화가 성공적으로 이루어지도록 보장한다. 그 후 포스트 프로빙 단계에서는 고정된 프로브 모델을 기반으로 MMIO/PIO 모델과 DMA 모델을 AFL 기반 랜덤 입력과 함께 결합하여 드라이버의 다양한 동작 경로를 탐색한다. 또한 주기적으로 타이머를 이용하여 인터럽트를 인위적으로 발생시켜 인터럽트 처리 경로의 취약점도 탐색한다. 실제 평가 결과, DEVFUZZ 는 리눅스, FreeBSD, Windows 의 다양한 버스 드라이버를 대상으로 테스트를 진행해 높은 코드 coverage 를 달성하였으며, 이 과정에서 총 63 건의 버그를 발견하고 이 중 39 건의 버그가 실제로 패치 되는 성과를 거두었다.

4. 비교분석

표 1 에서는 앞서 살펴본 3 가지 디바이스 드라이버 fuzzing 연구에 대해 동적 프로빙 방식, 지원하는 하드웨어 장치 유형(Bus Types), 드라이버가 장치와 통신하는 부분에서 조작 가능한 I/O 인터페이스(Fuzzing I/O), 그리고 각 도구의 주요 특징을 중심으로 비교 분석하였다.

우선 동적 프로빙 방식에서 3 개의 프레임 워크는 명확히 구분된다. USBFuzz 는 실제 디바이스 없이 소프트웨어 기반의 에뮬레이션을 통해 동적 프로빙을 수행하며, PeriScope 는 실제 하드웨어 디바이스 연결이 필수적이다. DEVFUZZ 는 심볼릭 실행을 통해 프로빙에 필요한 조건을 자동으로 계산하여, 실제 디바이스 없이 프로빙 과정을 수행한다.

지원 가능한 버스 종류에서도 차이를 보인다. USBFuzz 는 USB 장치에 특화되어 있어 PCI/PCIe 및 I2C 버스 지원이 어렵다. 반면 PeriScope 와 DEVFUZZ 는 PCI/PCIe, I2C, USB 등 다양한 버스 환경에서 범용적으로 활용 가능하다.

	동적 프로빙 방식	버스 종류			Fuzzing I/O			특징
		PCI/PCIe	I2C	USB	PIO	MMIO	DMA	Fuzzing
USBFuzz[6]	에뮬레이션	X	X	O	O	O	O	Coverage-guided
PeriScope[7]	디바이스	O	O	O	X	O	O	후킹
DEVFUZZ[8]	심볼릭 실행	O	O	O	O	O	O	Hybrid

<표 1> 비교 분석 표

Fuzzing 가능한 I/O 인터페이스의 범위에서도 각 프레임워크 간 차이가 나타난다. USBFuzz 는 USB 프로토콜 중심으로 설계되었지만, PIO 와 MMIO, DMA 를 지원한다. PeriScope 는 MMIO 와 DMA 인터페이스를 중점적으로 지원하지만 PIO 인터페이스는 지원하지 않는다. DEVFUZZ 는 MMIO, PIO, DMA 인터페이스를 모두 지원한다.

fuzzing 방식에서도 차별점이 존재한다. USBFuzz 는 coverage-guided 방식으로 빠르고 효율적인 탐색을 지원한다. PeriScope 는 커널의 페이지 폴트 후킹을 이용한 접근으로 정밀한 메모리 접근 탐지가 가능하며, DEVFUZZ 는 심볼릭 실행과 정적 및 동적 분석을 결합한 hybrid 방식으로 구조적이고 포괄적인 fuzzing 을 수행한다.

이러한 비교분석을 통해 각 프레임워크는 서로 다른 접근 방식과 장점을 가지고 있으며, 사용자의 fuzzing 목표 및 환경에 따라 적합한 도구 선택이 가능함을 알 수 있다.

5. 결론

디바이스 드라이버는 OS 보안에서 중요한 영역이나, 하드웨어와의 상호작용으로 인해 테스트가 어렵다. 본 조사에서는 드라이버 fuzzing 의 필요성과 난제를 소개하고 주요 연구를 비교했다. USBFuzz 는 소프트웨어 USB 장치 에뮬레이션을 통해 다양한 OS 의 USB 드라이버를 fuzzing 했다. PeriScope 는 페이지 폴트 후킹을 통해 드라이버와 하드웨어 간의 통신을 실시간으로 모니터링 및 변조하는 기법을 제안했고, DEVFUZZ 는 심볼릭 실행 및 정적·동적 분석을 활용해 실제 장치 없이도 드라이버의 초기화부터 다양한 I/O 동작을 자동으로 탐색할 수 있도록 했다. 향후에는 자동화의 확장 및 심볼릭 제약 충족 알고리즘 개선이 필요하다.

사사문구

이 논문은 2025 년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구 결과 임 (No. RS-2024-00337414, SW 공급망 운영 환경에서 역공학 한계를 넘어서는 자동화된 마이크로 보안 패치 기술 개발).

참고문헌

[1] A. T. Markettos, et al. “Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals,” in Proceedings of the Network and Distributed Systems Security Symposium (NDSS), 2019.

[2] B. Feng, et al. “P2IM: Scalable and hardware- independent firmware testing via automatic peripheral interface mod- eling,” in 29th USENIX Security Symposium (USENIX Security 20), 2020, pp. 1237–1254.

[3] A. Mera, et al. “Dice: Automatic emulation of dma input channels for dynamic firmware analysis,” in 2021 IEEE Symposium on Security and Privacy (SP), 2021, pp. 1938–1954.

[4] S. M. S. Talebi, et al. “Charm: Facilitating dynamic analysis of device drivers of mobile systems,” in 27th USENIX Security Symposium (USENIX Security 18), 2018, pp. 291–307.

[5] J. Corina, et al. “Difuze: Interface aware fuzzing for kernel drivers,” in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017, pp. 2123–2138.

[6] Peng, Hui, and Mathias Payer. “{USBFuzz}: A framework for fuzzing {USB} drivers by device emulation.” 29th USENIX Security Symposium (USENIX Security 20). 2020.

[7] Song, Dokyung, et al. "Periscope: An effective probing and fuzzing framework for the hardware-os boundary." 2019 Network and Distributed Systems Security Symposium (NDSS). Internet Society, 2019.

[8] Wu, Yilun, et al. "DEVFUZZ: automatic device model-guided device driver fuzzing." 2023 IEEE Symposium on Security and Privacy (SP). IEEE, 2023.

[9] “American fuzzy lop - a security-oriented fuzzer,” Retrieved April 13, 2025. https://github.com/ google/AFL.

[10] Syzkaller is an unsupervised coverage-guided kernel fuzzer, Retrieved April 11, 2025. https: //github.com/google/syzkaller.

[11] C.Cadar, et al. “Klee:Unassistedandautomatic generation of high-coverage tests for complex systems programs,” in Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, ser. OSDI’08. USA: USENIX Associ- ation, 2008, p. 209–224.

[12] V. Chipounov, et al. “S2e: A platform for in-vivo multi-path analysis of software systems,” in Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS XVI. New York, NY, USA: Association for Computing Machinery, 2011, p. 265–278.