eBPF 기반 물리 주소 추출 방법

이동영 [1], 강동현 [2]
[1] 동국대학교 컴퓨터 · AI 학과 석사과정
[2] 동국대학교 컴퓨터 · AI 학과 부교수

ehddud8339@dgu.ac.kr, donghyun@dgu.ac.kr

Methods for Extracting Physical Addresses Using eBPF

Dongyoung Lee[1], Donghyun Kang[2]
[1]Dept. of Computer Science and Artificial Intelligence, Dong-guk University
[2] Dept. of Computer Science and Artificial Intelligence, Dong-guk University

요 약

물리 주소는 메모리 내 실제 데이터 위치를 가리키며, CPU 와 하드웨어가 직접 읽고 쓰는 유일한 주소다. 운영체제는 가상 주소를 통해 물리 주소 접근을 제어하며 메모리 보호, 효율적인 관리와 보안을 제공한다. 이러한 설계로 인해 물리 주소 획득은 보안, 안전성 문제로 이어질 수 있어 보통 운영체제 코드 수정을 요구한다. eBPF 는 커널 내부에서 안전하게 사용자 정의 코드를 실행해 성능 계측, 트레이싱, 네트워킹을 확장하지만, 검증기와 보안 제약으로 물리 주소를 직접 노출하지 않는다. 본 논문에서는 eBPF 로 물리 주소를 획득하기 위한 표준 인터페이스 기반 접근과 커널 이벤트 시점에서 구조체 필드 접근의 두 가지 접근법을 소개한다.

1. 서 론

물리 주소는 메모리 내 실제 데이터의 위치로 CPU 와 하드웨어가 직접 접근하는 유일한 주소 체계이다. 운영체제는 가상 메모리를 통해 프로세스별 독립된 주소 공간을 제공하고 페이지 테이블로 이를 물리 주소와 매핑하여 보호, 격리, 효율성, 보안을 보장하기 때문에 응용 프로그램이 물리 주소를 직접 다루는 경우는 드물다. 그럼에도 불구하고 물리 주소는 디버깅, 성능 분석, 메모리 관리 최적화, 보안 연구 등 커널 수준의 정밀한 분석에 종종 필요하지만, 보안과 안정성을 이유로 기본적으로 노출되지 않아 커널 코드 수정 없이는 얻기 어렵다[1-3]. 따라서 안전하고 표준화된 방식으로 물리 주소를 추출하는 것은 연구자와 개발자에게 중요한 과제다. 예를 들어, 커널 디버깅 과정에서는 특정 가상 주소가 실제 어느 물리 페이지로 매핑되는지를 추적함으로써 메모리 누수나 중복 매핑 문제를 분석할 수 있으며, 가상화 환경에서는 게스트 물리 주소(Guest Physical Address, GPA)와 호스트 물리 주소(Host Physical Address, HPA)의 변환 관계를 검증하여 메모리 접근의 정확성과 성능을 평가할 필요가 있다. 이러한 사례들은 물리 주소 정보가 단순한 하드웨어 관리 수준을 넘어, 시스템의 신뢰성과 효율성을 확보하기 위한 필수 분석 요소임을 보여준다.

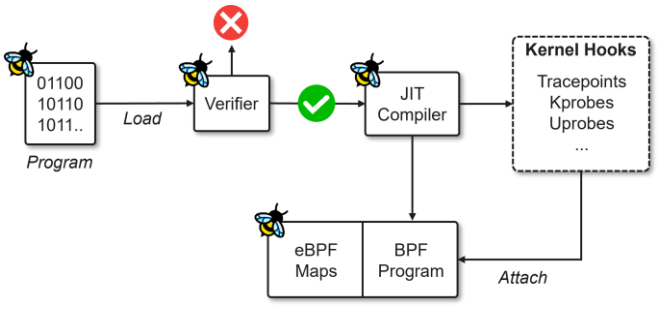

그림 1. eBPF 구조도

기존의 물리 메모리 추출 방식은 주로 커널 내부 API(get_user_pages(), page_to_phys(), virt_to_phys() 등)를 이용하여 가상 주소로부터 페이지 구조체(struct page)를 얻고, 이를 물리 주소로 변환하는 방식에 의존한다. 이러한 접근은 커널이 관리하는 메모리 매핑에 직접 접근해야 하므로, 사용자 공간에서 표준 인터페이스만으로는 수행이 불가능하다. 즉, 물리 주소를 확보하기 위해서는 커널 모듈을 작성하거나, 커널 소스를 수정하여 내부 API 를 노출시키는 절차가 불가피하다. 그러나 이러한 방법은 커널 버전에 따른 구조체 편차와 보완 정책 강화로 인한 유지보수가 어렵고, 시스템 안정성을 해칠 위험이 있다. 따라서 커널 수정 없이 안전하게 물리 주소를 관찰할 수 있는 새로

운 접근 방식이 필요하다.

이러한 배경에서 주목받는 기술이 eBPF(extended Berkeley Packet Filter)로 커널 내부에서 사용자 정의 프로그램을 안전하게 실행할 수 있도록 설계된 확장 가능한 런타임 환경이다[4]. 원래 네트워크 패킷 필터링을 위해 도입되었으나, 오늘날에는 성능 계측, 커널 이벤트 트레이싱, 보안 모니터링, 네트워킹 기능 확장 등 다양한 영역에서 활용되고 있다[5-8]. eBPF 의 장점은 별도의 커널 코드 수정이나 컴파일 없이도 실행 중인 커널에 동적으로 프로그램을 삽입할 수 있다는 점이며, 이는 커널과 하드웨어 자원의 동작을 관찰하기 위한 강력한 도구로 자리잡았다.

그러나 eBPF 의 강력한 능력은 동시에 엄격한 제약 속에서 제한적으로 제공된다. 특히 eBPF Verifier 는 프로그램의 안전성을 보장하기 위해 잠재적으로 위험한 메모리 접근을 철저히 차단한다. 이는 악의적인 프로그램이 커널 내부의 민감한 데이터를 유출하거나 시스템 안정성을 위협하지 못하도록 하기 위한 설계이다. 이러한 이유로 eBPF 는 기본적으로 물리 주소를 직접 노출하지 않으며, 사용자가 단순히 eBPF 프로그램을 작성하는 것만으로는 가상 주소로부터 물리 주소를 얻을 수 없다.

본 논문은 이러한 제약 속에서 eBPF 를 활용하여 물리 주소를 획득할 수 있는 두 가지 접근 방식을 소개한다. 첫 번째는 표준 인터페이스인 /proc/<pid>/pagemap 파일을 활용하여 가상 주소에 대응하는 페이지 프레임 번호(Page Frame Number, PFN)을 얻고, 이를 통해 물리 주소를 계산하는 방식이다. 두 번째는 커널 이벤트 시점에서 구조체 필드에 접근하는 방식이다. 이 접근법은 커널 버전 호환성과 보안 정책에 따른 제약을 고려해야 한다.

본 논문은 1 장의 서론에 이어 2 장에서 eBPF 의 구조를 살펴보고, 3 장에서 두 가지 접근 방식을 설명하며 4 장에서 결론을 도출한다.

2. eBPF (extended Berkeley Packet Filter)

eBPF 는 커널 내부에 안전하게 삽입되어 실행되는 경량 런타임 환경으로, 초기에는 네트워크 패킷 필터링을 목적으로 도입되었지만 현재는 시스템 성능 분석, 이벤트 트레이싱, 보안 모니터링, 네트워킹 기능 확장 등 다양한 영역에서 활용되고 있다. 특히 커널을 직접 수정하거나 컴파일하지 않고도 실행 중인 시스템에 동적으로 프로그램을 삽입할 수 있다는 점에서 커널 및 하드웨어 동작을 관찰하고 제어하는 강력한 도구로 자리매김하였다. 그림 1 은 eBPF 의 내부 구조를 나타내며, 크게 다섯 가지 요소로 나눌 수 있다.

BPF Program. 사용자가 작성하는 코드로, 일반적으로 C 언어를 바탕으로 작성되어 바이트 코드 형태로 커널에 로드된다. 이 프로그램은 커널 내 특정 이벤트가 발생했을 때 실행되며, 이벤트 기반의 동작을 정의하는 핵심 역할을 한다.

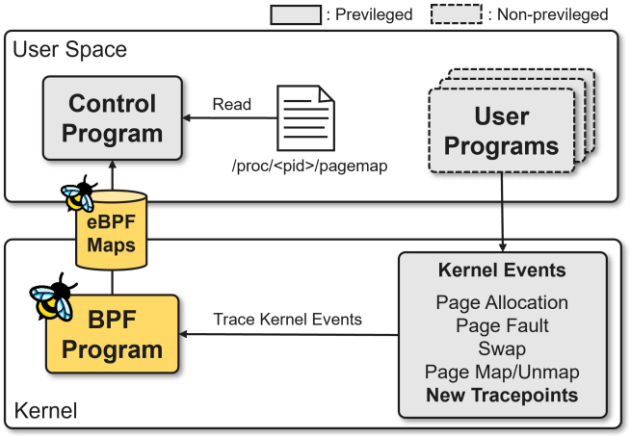

그림 2. eBPF 기반 물리 주소 추출 구조도

eBPF Verifier. 프로그램이 커널에 로드되기 전에 수행되는 정적 분석기 역할을 한다. 사용자 코드의 무한 루프 가능성, 잘못된 포인터 연산, 커널 메모리 침범 등 안정성을 해칠 수 있는 요소를 검증하고 차단한다. Verifier 단계에서 통과하지 못한 프로그램은 커널에 삽입되지 않으며, 이는 커널 안정성과 보안성을 유지하기 위한 핵심 장치다.

JIT(Just-In-Time) Compiler. Verifier 의 검증을 통과한 바이트코드를 실제 머신의 명령어 집합으로 변환해 실행한다. 이를 통해 인터프리터 방식의 성능 저하를 최소화하고, 네이티브 코드 수준의 성능을 제공한다.

Kernel Hooks. BPF 프로그램이 실제로 실행되는 지점으로, 커널 내부의 다양한 이벤트와 연결된다. 대표적으로 함수 진입/종료 지점에 붙는 Kprobes, 사용자 공간 함수 호출을 추적하는 Uprobes, 미리 정의된 커널 이벤트를 노출하는 Tracepoints 등이 있다. 이를 통해 eBPF 는 파일 시스템 호출, 메모리 관리 이벤트, 네트워크 스택 처리, 사용자 프로그램 실행 흐름 등 다양한 계층에서의 동작을 관찰하고 필요한 데이터를 수집할 수 있다.

eBPF 맵(Maps). BPF 프로그램이 수집한 데이터를 저장하고 사용자 공간과 공유하기 위한 메모리 구조다. 해시 맵(Hash), 배열(Array), LRU(Map), 링 버퍼(Ring Buffer) 등 다양한 형태를 제공하며, 이를 통해 BPF 프로그램은 지속적인 상태를 유지하거나 사용자 프로그램과 효율적으로 데이터를 교환할 수 있다. 예를 들어, 특정 프로세스의 I/O 호출을 추적하는 BPF 프로그램은 수집된 타임스탬프와 이벤트 메타데이터를 맵에 저장하고, 사용자 공간 프로그램은 이를 읽

표 1. 실험 환경

CPU	12th Intel i9-12900KF
메모리	32GB DDR4 DRAM
커널 버전	Linux 5.15

```
> sudo ./control_program
TS(ns)          PID    TGID   COMM            VADDR             PADDR
-------------------------------------------------------------------------------
  5506825673279  47300  47300 user_program    0x00007fffba869000 0x000000029c82e000
  5506825685912  47300  47300 user_program    0x00007f3176db9000 0x000000023d9fe000
  5506825698953  47300  47300 user_program    0x00007fffba92f000 0x00000005edac2000
  5506825700109  47300  47300 user_program    0x00007fffba868000 0x0000000277707000
  5506825716684  47300  47300 user_program    0x00007f3176d76000 0x000000029ee3f000
  5506825718106  47300  47300 user_program    0x00007fffba867000 0x0000000273387000
  5506825738195  47300  47300 user_program    0x00007f3176d77000 0x0000000272caf000
  5506825795155  47300  47300 user_program    0x00007f3176d75000 0x00000001eeb6d000
  5506825796823  47300  47300 user_program    0x00007f3176d73000 0x000000024428b000
  5506825825183  47300  47300 user_program    0x00007f3176d72000 0x00000002760bc000
  5506825827907  47300  47300 user_program    0x000055ccf083a000 0x00000002516a3000
  5506825828927  47300  47300 user_program    0x000055ccf083b000 0x00000002a3818000
  5527125081140  47300  47300 user_program    0x000055ccf083c000 0x000000029b7f5000

> ./user_program
location of stack   : 0x7fffba8692e4
location of heap    : 0x55ccf083a2a0

location of stack   : 0x7fffba8692e4
location of heap    : 0x55ccf083bad0
```

그림 3. user_program, control_program 실행 결과

어 분석 및 시각화를 수행할 수 있다.

이와 같은 구조는 eBPF 를 강력한 도구로 만들어 주지만, 동시에 Verifier 의 제약은 eBPF 응용에 큰 한계로 작용한다. Verifier 는 메모리 접근을 철저히 제한해 허용된 구조체나 헬퍼 함수 외에는 접근하지 못하게 하며, 특히 커널 내부의 민감한 데이터나 물리 주소는 기본적으로 차단된다. 이러한 설계는 악의적인 코드가 커널의 안정성을 해치거나 정보를 유출하는 것을 방지하는 데 효과적이지만, 연구자나 개발자가 세밀한 성능 분석이나 메모리 추적을 시도하는 데 있어 장애물이 된다. 결과적으로 eBPF 는 커널 안정성과 보안을 유지하면서도 확장성과 유연성을 제공하지만, 물리 주소 추출과 같은 민감한 작업에는 본질적으로 제약이 따른다.

3. eBPF 기반 물리 주소 추출 방법

이번 장에서는 eBPF 를 활용하여 물리 주소를 추출하는 공통적인 흐름과 이를 기반으로 한 두 가지 접근 방식을 설명한다. 그림 2 는 이러한 공통 구조를 나타내며, 일반 사용자 프로그램이 동작하면서 발생하는 커널 이벤트를 eBPF 프로그램이 추적하여 이벤트 발생 시점의 프로세스 식별자(pid)와 가상주소(vaddr)를 수집하고, 가능하다면 PFN 이나 PTE 값을 함께 확보한다. 이렇게 수집된 정보는 eBPF 맵에 저장되며, 사용자 공간의 컨트롤 프로그램이 이를 읽어 최종적인 물리 주소 계산 과정에 활용한다.

두 방법 모두 커널 이벤트 훅을 통해 pid 와 vaddr, 그리고 상황에 따라 PFN 이나 PTE 를 수집한다는 공통 절차를 따르며, 차이는 물리 주소를 산출하는 과정에서 발생한다.

표준 인터페이스 기반 추출. 커널은 일반 사용자 프로세스가 물리 주소에 직접 접근하는 것을 제한하지만, 제한된 범위 내에서 디버깅과 성능 분석을 지원하기 위해 /proc/<pid>/pagemap 과 같은 인터페이스를 제공한다. eBPF 자체는 보안상의 이유로 해당 파일을 직접 열거나 읽을 수 없으므로, 루트 권한(특권)으로 실행된 컨트롤 프로그램이 eBPF 맵에서 가져온 pid 와 vaddr 을 기반으로 /proc/<pid>/pagemap 을 조회한다. 이는 물리 주소가 민감한 커널 내부 자원에 해당하므로, 인가되지 않은 접근을 방지하고 시스템 보안을 유지하기 위함이다. 이 과정에서 해당 가상 주소에 대응하는 PFN 을 획득하고, 이를 페이지 크기 단위로 확장한 뒤 페이지 오프셋을 더해 물리 주소를 계산한다. 이와 같이 물리 주소 추출은 무분별한 접근이 아닌, 명시적으로 권한이 부여된 관리자가 수행하는 절차임을 보장한다.

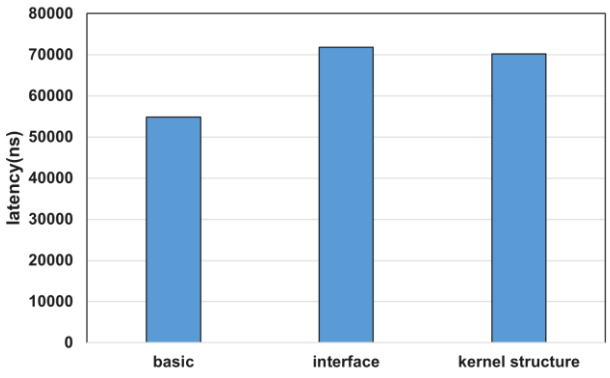

그림 4. 기본 방식과 eBPF 접근법의 latency

커널 내부 구조체 접근. eBPF 추적 메커니즘을 이용해 커널의 특정 함수나 이벤트에서 구조체 필드를 직접 읽어 PFN 이나 PTE 를 확보하는 방식이다. 예를 들어 익명(Anonymous) 페이지에서 발생한 페이지 폴트를 처리하는 함수인 do_anonymous_page 의 반환 값을 통해 PFN 값을 구할 수 있다. 그러나 대부분의 커널은 이러한 정보를 직접적으로 얻을 수 있는 함수를 커널 훅으로 제공하지 않기 때문에, 원하는 시점의 데이터를 확보하기 위해서는 커널 소스에 사용자 정의 트레이스포인트를 추가해야 하는 경우가 많다. 이 방법은 표준 인터페이스로는 얻을 수 없는 세밀한 정보를 얻을 수 있다는 장점이 있지만, 커널 수정이 필요없는 eBPF 의 장점을 해치고 보안과 운영상의 위험이 증가한다는 한계가 있다.

두 접근 방식 모두 eBPF 를 통해 커널 이벤트로부터 메타데이터를 수집하고 사용자 공간에서 이를 해석해 최종 물리 주소를 계산한다는 공통된 구조를 가진다. 그러나 표준 인터페이스 접근은 구현이 단순하고 재현성이 높다는 장점이 있지만, 접근이 제한적이고 실시간성이 부족하다는 한계가 있다. 반대로 커널 구조체 접근은 필요한 정보를 직접적으로 얻을 수 있어 정밀성 면에서는 유리하지만, 커널 버전 의존성이

크고 Verifier 제약이나 보안, 운영 리스크 때문에 실제 환경에서 적용하기 어렵다. 결국 두 방법 모두 일정한 효과를 제공하지만, 현 시점에서는 물리 시점 주소 획득 과정에서 여전히 제약이 존재한다.

4. 구현 결과 및 eBPF 오버헤드 분석

4.1 실험 환경

본 논문은 커널 이벤트 시점에서 수집한 가상 주소를 기반으로 물리 주소를 추출하고, eBPF 적용 시 발생하는 오버헤드를 분석한다.

user_program 은 페이지 폴트를 유발하기 위해 4KB 크기의 메모리를 동적으로 할당하는 함수를 반복 호출하였다. control_program 은 BPF 프로그램에서 전달된 메타데이터를 수신하여, PFN 을 확보한 경우에는 이를 기반으로 물리 주소를 계산하고 출력하였다. 가상 주소만 전달된 경우에는 표준 인터페이스를 이용하여 물리 주소를 계산하였다.

또한, 표준 인터페이스 접근 방식과 커널 구조체 접근 방식을 적용하여 user_program 의 함수 실행 시간을 측정하였으며, 이를 통해 eBPF 의 오버헤드를 정량적으로 평가하였다. 실험에 사용된 시스템 환경은 표 1 과 같다.

4.2 결과 분석

그림 3 은 user_program 과 control_program 의 실행 결과를 나타낸다. user_program 은 함수 호출 시 스택(stack) 변수와 힙(heap) 변수의 가상 주소를 출력하며, control_program 은 타임스탬프(TS), 프로세스 식별자(PID), 프로세스 그룹 식별자(TGID), 프로세스 이름(COMM), 가상 주소(VADDR), 물리 주소(PADDR)를 함께 출력한다.

그림 3 의 결과에서 확인할 수 있듯이, VADDR 의 하위 12 비트(16 진수라 3 자리)를 제외한 상위 비트가 일치하였다. 이는 커널이 페이지 단위로 페이지 폴트를 처리하기 때문에, 페이지 오프셋을 나타내는 하위 12 비트가 0 으로 초기화되었음을 의미한다. 이를 통해 제안한 방법이 커널 이벤트를 정확하게 추적하고 물리 주소를 올바르게 산출할 수 있음을 검증하였다.

그림 4 는 user_program 에서 메모리 할당 함수를 호출하고 반환되기까지의 지연 시간(Latency)를 10 회 측정한 후 평균값을 계산한 결과이다. eBPF 를 사용하지 않는 기본 방식(basic)의 평균 지연 시간은 약 55,000 ns 였다. 반면, 표준 인터페이스 기반 방법(interface)은 약 71,000 ns, 커널 구조체 기반 방법(structure)은 약 70,000 ns 로 측정되었다. 두 방식은 유사한 결과를 보였으며, 모두 기본 방식 대비 약 30% 의 추가 지연 시간이 발생하였다. 이는 eBPF Verifier 검증, 링 버퍼를 통한 데이터 전달, 커널과 사용자 공간 간 문맥 교환(Context Switch) 등에서 발생하는 오버헤드이다.

따라서 eBPF 기반 방법은 연구 및 개발 단계에서는 유용하게 활용될 수 있으나, 실시간성이나 저지연이 요구되는 환경에서는 추가적인 최적화가 필요함을 확인하였다.

5. 결 론

물리 주소는 성능 분석, 저수준 디버깅, 메모리 관리 최적화, 보안 연구 등에서 필수적인 정보이나, 보안상의 이유로 커널은 이를 직접 노출하지 않는다. eBPF 는 커널 내부 동작을 안전하게 관찰할 수 있는 유연한 도구로 주목받고 있으나, 기본적으로 물리 주소를 직접 제공하지 않는다는 한계가 존재한다.

본 논문에서는 이러한 제약 속에서 물리 주소를 획득할 수 있는 두 가지 방법을 구현하고, 실제 동작 결과 및 성능 오버헤드를 분석하였다. 두 방법 모두 정확한 물리 주소 산출이 가능했으며, 기본 방식 대비 약 30%의 지연 시간이 추가되는 것으로 나타났다.

이러한 결과는 eBPF 기반 방법이 커널 코드 수정 없이 안전하게 물리 주소 추출을 가능하게 한다는 점에서 의의가 있지만, 오버헤드 문제와 커널 버전 의존성, 보안 제약은 여전히 존재하므로, 실제 운영 환경에 적용하기 위해서는 최적화 및 보안 검증이 병행되어야 한다.

참고문헌

[1] Movall, P., Nelson, W., Wetzstein, S. “Linux Physical Memory Analysis.” *USENIX FREENIX Track*, 2005.

[2] Deng, Z., et al. “Stealthy Binary Program Instrumentation and Debugging via IPnvisible Breakpoint.” *USENIX Security Symposium*, 2013.

[3] Srinivasan, D., Subramanian, S. “Kernel Address Discovery via Virtual-Physical Mapping Attacks.” *USENIX Security Symposium*, 2021.

[4] The Linux kernel documentation. "BPF Documentation." https://docs.kernel.org/bpf/

[5] Bui, H., Frigo, M. “eUProbes: Efficient eBPF-based kernel extension using uprobes.” *OSDI*, 2022.

[6] Chen, Q., Liu, Y., Ren, F. “eXpress: Kernel Extension via eBPF for Network Function Acceleration.” *SIGCOMM*, 2023.

[7] Dwivedi, K. K., Iyer, R., Kashyap, S. “Fast, Flexible, and Practical Kernel Extensions.” *EuroSys*, 2024.

[8] Jia, J., Qin, R., Craun, M., et al. “Safe and Usable Kernel Extensions with Rax.” *OSDI*, 2025.