

Learning eBPF

(by Liz Rice, 2023)

JADECROSS 정환열 이사

coordinatorj@jadecross.com

2025



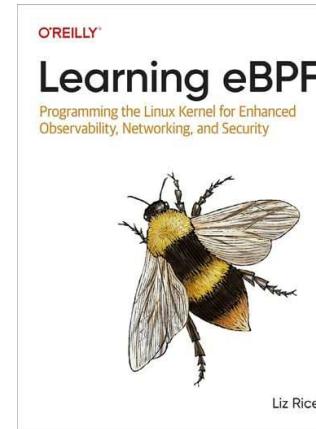
목 차

- 0. 들어가기 전에
- 1. What Is eBPF, and Why Is It Important?
- 2. eBPF's "Hello World"
- 3. Anatomy of an eBPF Program
- 4. The bpf() System Call
- 5. CO-RE, BTF, and Libbpf
- 6. The eBPF Verifier
- 7. eBPF Program and Attachment Types
- 8. eBPF for Networking
- 9. eBPF for Security
- 10. eBPF Programming
- 11. The Future Evolution of eBPF

들어가기 전에

▪ Learning eBPF PDF 다운로드

- ▶ Isovalent 의 공동 창업자인 Liz Rice 가 eBPF 기술 확인을 위해 무료로 제공
 - <https://cilium.isoivalent.com/hubfs/Learning-eBPF%20-%20Full%20book.pdf>



▪ 실습 환경

- ▶ Rocky Linux 9.6, kernel 버전 5.14.0-570.25.1.el9_6.x86_64
- ▶ PDF 책에서는 Ubuntu 환경이라 C 소스 컴파일 환경을 Rocky Linux에 맞게 조정하였음

▪ 또 다른 실습환경

- ▶ Isovalent 홈페이지에서 실습 환경을 제공하고 있으며, 간단한 정보 제공 후 이용 가능
 - <https://isovalent.com/labs/ebsf-getting-started/>

Please provide contact details to access the lab

First name*	Last name*
Company*	Work Email*
Country*	
Please Select	
<input type="checkbox"/> I would like Isovalent and Cisco to email me the offers, promotions, and the latest news regarding Isovalent and Cisco products and services. I know I can unsubscribe at any time. Visit here to learn more.	
Submit	

참고 링크

▪ eBPF 개념을 짧게 정리

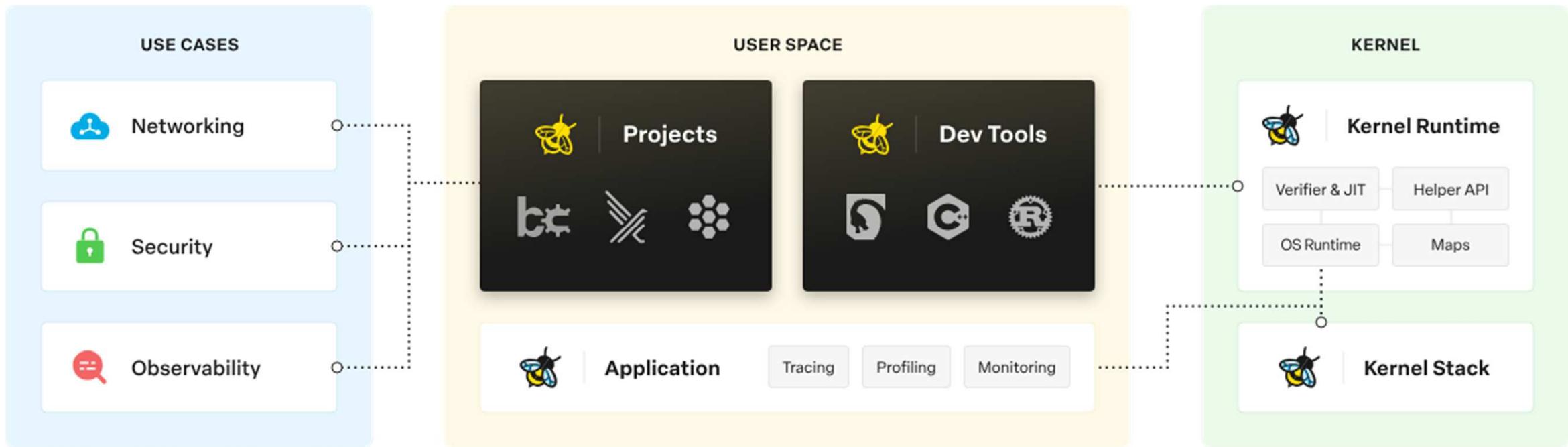
- ▶ What is eBPF?
 - <https://ebpf.io/ko-kr/what-is-ebpf/>

▪ eBPF 를 심도있게 학습

- ▶ BPF and XDP Reference Guide
 - <https://docs.cilium.io/en/latest/reference-guides/bpf/index.html#bpf-and-xdp-reference-guide>

1. What Is eBPF, and Why Is It Important?

Dynamically program the kernel for efficient networking, observability, tracing, and security



Programs are verified
to safely execute

Hook anywhere in the
kernel to modify
functionality

JIT compiler for near
native execution speed

Add OS capabilities at
runtime

What is eBPF?

- ✓ 특정 이벤트(예: 네트워크 패킷 도착, 시스템 콜 호출)가 발생했을 때만 실행됩니다.
 - system calls
 - kprobes(kernel functions), uprobes(userspace functions), fentry/fexit
 - tracepoints
 - network devices(tc/xdp), network routes, TCP congestion algorithms
 - sockets(data level)
- ✓ 평소에는 아무런 부하를 주지 않습니다.

커널 내부에 존재하는, 샌드박스 환경의 가상 머신에서 실행되는, 이벤트 기반 프로그램

- ✓ 자체적인 Instruction Set 과 Registers 를 가집니다.
- ✓ 이 덕분에 eBPF 프로그램은 CPU 아키텍처(x86, ARM)에 독립적입니다.

- ✓ Verifier라는 강력한 안전장치가 있어, 프로그램이 커널을 손상시키거나 시스템을 멈추게 하는것을 원천적으로 불가능하게 만듭니다.

- ✓ 운영체제의 가장 핵심적인 부분(Ring 0)에서 실행됩니다.
- ✓ 이는 시스템의 모든 데이터와 활동에 대한 완전한 접근 권한을 가짐을 의미합니다.

커널의 기능을 확장하고 싶어요! (1/2)

■ 선택 1: 커널 소스 코드 직접 수정

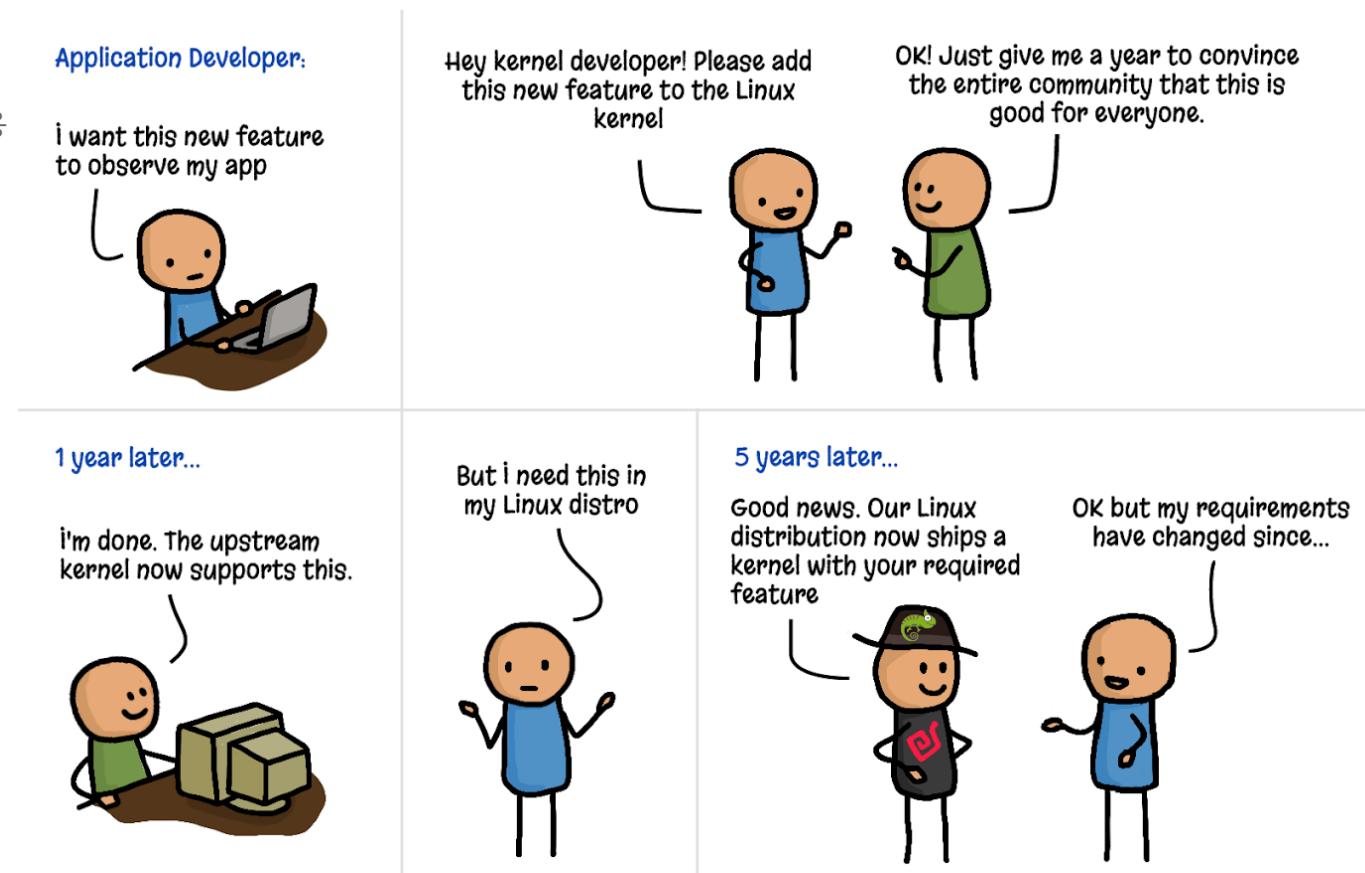
▶ 커널 소스 다운로드 → 코드 수정 → 전체 커널 컴파일 → 시스템 재부팅

▶ 단점

- 작은 실수 하나가 시스템 전체를 부팅 불능 상태로 만들 수 있음.
- 수많은 서버에 이 작업을 반복하는 것은 사실상 불가능.

▶ 결론은 커널 개발팀에서 New Feature로 커널에 포함

- 너무 오래걸림
- 더욱이 배포판 회사는 최신 커널보다는 안정화 버전 이용



커널의 기능을 확장하고 싶어요! (2/2)

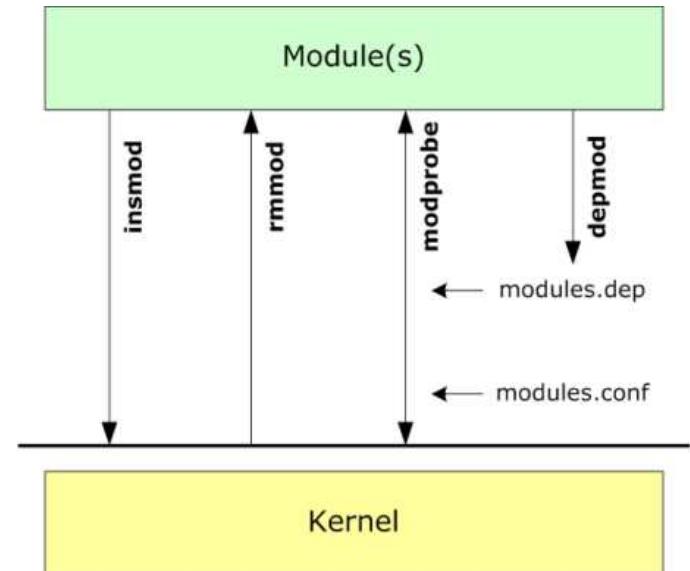
▪ 선택 2: 커널 모듈로 개발하여 동적으로 적재 및 제거

▶ 장점

- insmod 명령어로 모듈(.ko 파일)을 실행 중인 커널에 동적 로드
- rmmod 명령어로 모듈(.ko 파일)을 실행 중인 커널에서 동적으로 제거

▶ 단점

- 치명적 단점: 안전 장치 부재.
- 커널과 모든 권한을 공유하므로, 모듈의 버그는 곧 커널의 버그가 되어 시스템 전체를 다운시킵니다.



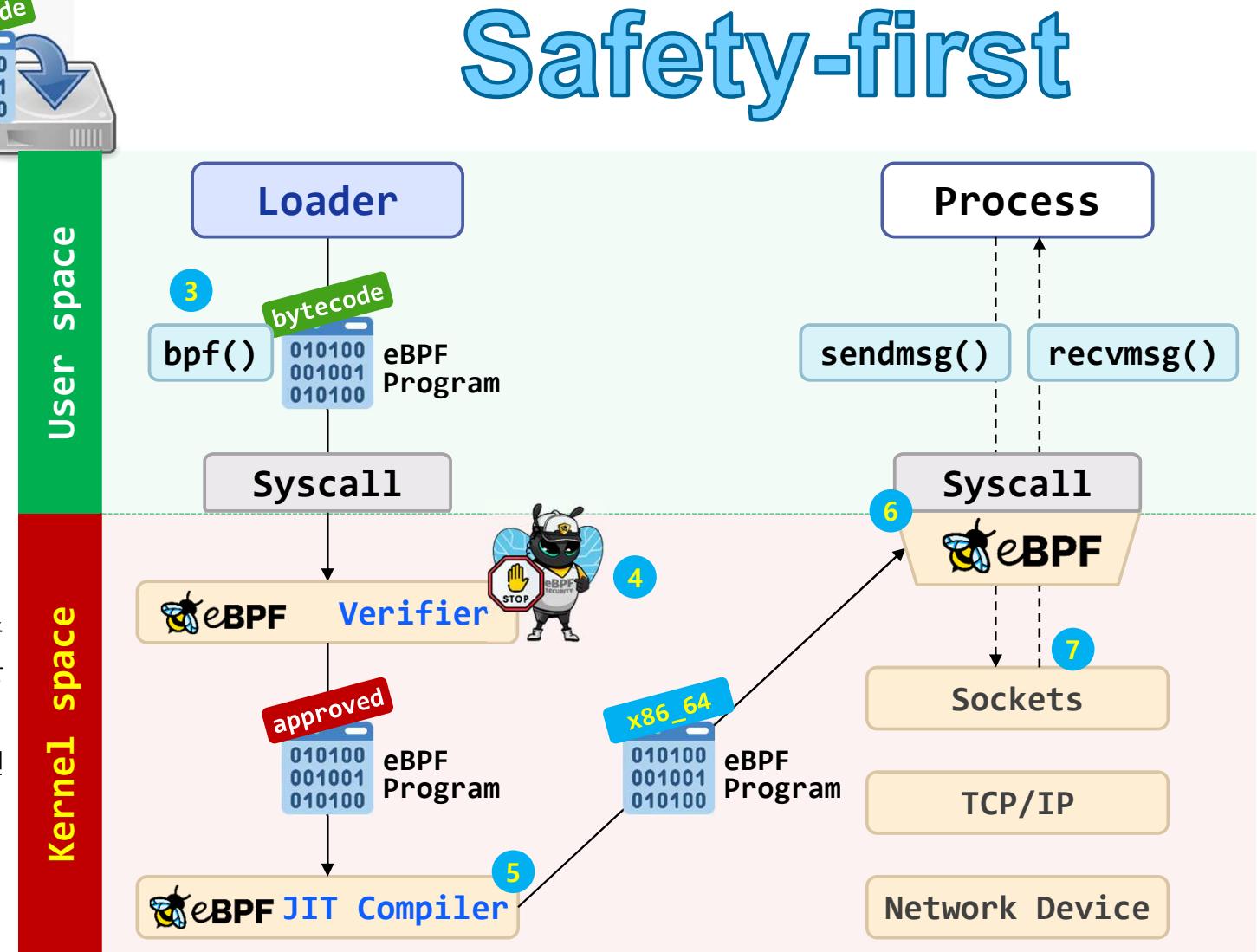
eBPF는 커널을 위한 안전하고 효율적인 샌드박스형 가상 머신 필요



eBPF 프로그램 개발 및 동작 프로세스



- ① 개발자는 C, Rust, Go와 같은 친숙한 언어로 eBPF 프로그램을 작성합니다.
- ② LLVM/Clang과 같은 특수 컴파일러가 소스 코드를 CPU 독립적인 eBPF 바이트코드로 변환합니다.
- ③ Userspace의 Loader(예: bpftool)가 bpf() 시스템 콜을 통해 바이트코드를 커널로 전달합니다.
- ④ 커널의 '검증기'가 바이트코드가 안전 규칙을 모두 준수하는지 정적 분석합니다. (가장 중요한 단계!)
- ⑤ 검증을 통과하면, JIT 컴파일러가 eBPF 바이트코드를 현재 시스템의 CPU가 직접 실행할 수 있는 네이티브 기계어 코드로 변환합니다.
- ⑥ 변환된 코드를 지정된 이벤트 흑(예: kprobe:do_sys_open)에 연결합니다.
- ⑦ 해당 이벤트가 발생하면, 부착된 네이티브 코드가 커널 내에서 직접 실행됩니다.



eBPF 핵심 구성 요소 (1/4)

▪ Verifier

- ▶ eBPF의 문지기이자 보안 분석가
- ▶ 분석 대상
 - 개발자가 작성한 소스 코드가 아닌, 컴파일된 eBPF 바이트코드
- ▶ 분석 방식
 - DAG(Directed Acyclic Graph, 방향성 비순환 그래프) 분석을 통해 프로그램의 모든 가능한 실행 경로를 시뮬레이션합니다.
- ▶ 주요 검사 목록
 - 유한 실행 보장: 프로그램이 무한 루프에 빠질 수 있는가? (백워드 점프 및 루프 횟수 제한)
 - 유효한 메모리 접근:
 - 초기화되지 않은 변수나 레지스터를 읽으려고 하는가?
 - 허용된 메모리(자신의 스택, 맵, 컨텍스트) 외의 임의의 커널 메모리에 접근하려고 하는가?
 - NULL 포인터를 역참조할 가능성이 있는가?
 - 타입 안정성: 포인터와 정수를 혼용하는 등 위험한 타입 캐스팅을 하는가?
 - 허가된 기능 호출: 오직 등록된 eBPF 헬퍼 함수만 호출하는가?

이 모든 검사를 통과해야만 프로그램 로드가 허용됩니다.



eBPF 핵심 구성 요소 (2/4)

▪ Maps

- ▶ eBPF의 두뇌와 신경망. 데이터의 저장과 통신을 담당
- ▶ 커널 내부에 존재하는 고성능 Key-Value 저장소

▪ 데이터 흐름

▶ Kernel space ↔ User space

- Kernel space → User Process
 - eBPF 프로그램이 수집한 데이터(성능 지표, 이벤트 로그)를 맵에 저장하면, 사용자 공간 애플리케이션이 이를 읽어와 시각화하거나 분석합니다.
- User Process → Kernel space
 - 사용자 공간에서 정책(예: 차단할 IP 목록)을 맵에 써넣으면, eBPF 프로그램이 이를 읽어 실시간으로 정책을 적용합니다.

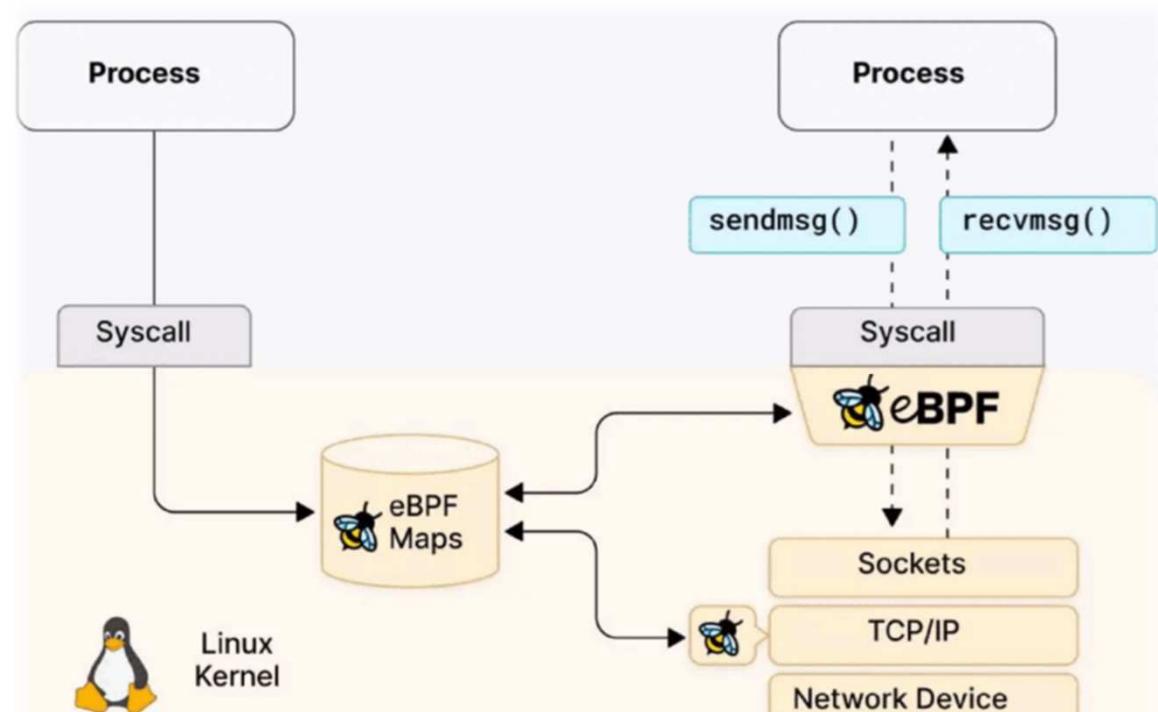
▶ eBPF 프로그램 ↔ eBPF 프로그램

- 하나의 eBPF 프로그램(예: 시스템 콜 진입점)이 맵에 상태(예: 파일 경로)를 저장하면, 다른 eBPF 프로그램(예: 시스템 콜 종료점)이 그 상태를 읽어와 후속 처리를 합니다.

▪ 종류

- ▶ 해시 테이블, 배열, 스택, 큐, 링 버퍼 등 용도에 맞는 다양한 맵 타입이 존재합니다.

eBPF 프로그램 자체는 일회성 실행 후 사라지는 '단기 기억'만 가지고 있습니다. 맵은 eBPF에게 '장기 기억'과 '소통 수단'을 제공하는 매우 중요한 존재입니다. 어떤 종류의 맵을 어떻게 설계하느냐가 eBPF 애플리케이션의 성능과 기능을 결정짓는 핵심 요소가됩니다.



eBPF 핵심 구성 요소 (3/4)

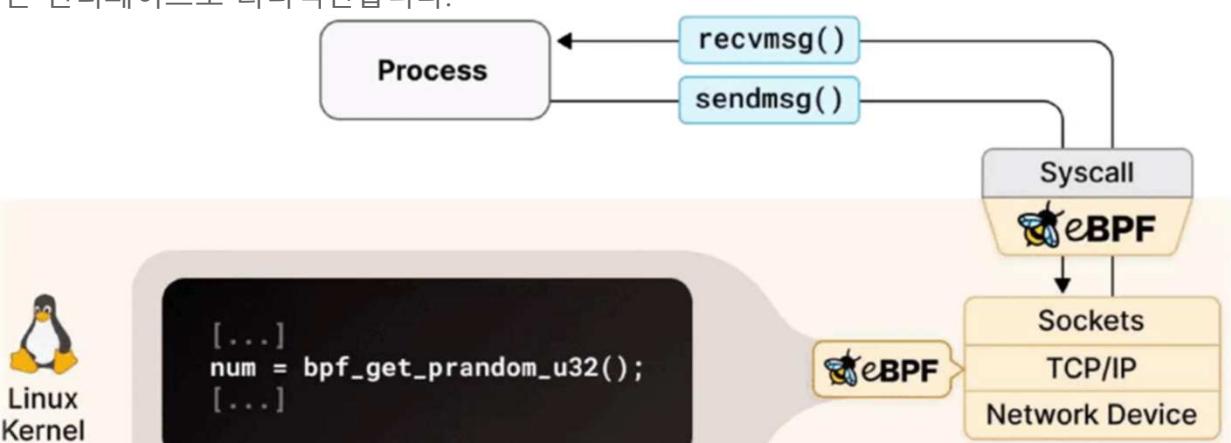
▪ Helper Functions

- ▶ eBPF 프로그램이 커널 기능에 접근하기 위한 안전하고 안정적인 API 집합
- ▶ 안전성
 - eBPF 프로그램이 커널의 아무 함수나 직접 호출하는 것을 금지합니다.
 - 이는 예측 불가능한 동작과 보안 취약점을 막기 위함입니다.
- ▶ 안정적인 인터페이스 (UAPI)
 - 커널 버전이 업그레이드되어도 헬퍼 함수의 기능과 사용법은 바뀌지 않도록 보장됩니다.
- ▶ 검증된 기능
 - 각 헬퍼 함수는 그 자체로 안전하며, 검증기는 eBPF 프로그램이 이 함수들을 올바른 타입의 인자로 호출하는지 검사합니다.
- ▶ 주요 헬퍼 함수
 - bpf_map_lookup_elem(map, &key): 맵에서 키에 해당하는 값을 조회합니다.
 - bpf_get_current_pid_tgid(): 현재 실행 중인 프로세스의 ID와 스레드 그룹 ID를 반환합니다.
 - bpf_probe_read_kernel(dst, size, src): 커널 메모리에서 안전하게 데이터를 복사합니다.
 - bpf_clone_redirect(skb, ifindex, flags): 네트워크 패킷을 복제하여 다른 인터페이스로 리디렉션합니다.

헬퍼 함수를 '커널이라는 자판기'의 버튼이라고 생각하시면
쉽습니다. 우리는 자판기 내부의 복잡한 메커니즘을 몰라도, 정해진
버튼(헬퍼 함수)을 누르면 원하는 음료수(커널 데이터나 기능)를
안전하게 얻을 수 있습니다. eBPF 프로그램은 이 버튼들 외에 다른
방법으로는 절대 자판기 내부에 손을 댈 수 없습니다. 이것이 바로
eBPF가 강력한 기능을 제공하면서도 안전을 유지하는 비결입니다.



13



eBPF 핵심 구성 요소 (4/4)

▪ Hooks

- ▶ eBPF 프로그램이 연결(attach)되어 실행을 시작하는 커널 내의 특정 지점 또는 이벤트
- ▶ eBPF 프로그램은 스스로 실행되지 않습니다. 반드시 특정 '훅'에 연결되어, 해당 이벤트가 발생했을 때 '트리거(trigger)'되어야 합니다.

▪ 주요 Hook

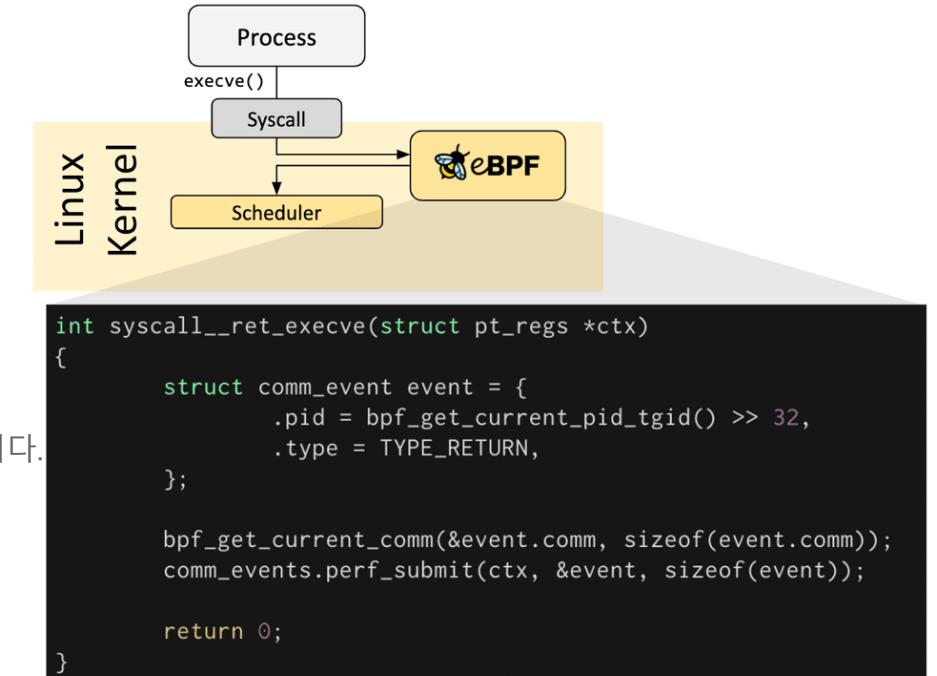
▶ 사전 정의된 Hook

- Tracepoints
 - 커널 소스 코드에 명시적으로 정의된 안정적인 추적 지점. (가장 안정적임)
- XDP (eXpress Data Path)
 - 네트워크 드라이버 레벨의 훅. 가장 먼저 패킷을 만나는 지점. (네트워킹 최고 성능)
- TC (Traffic Control)
 - 커널 네트워크 스택의 Ingress/Egress 큐에 연결. (정교한 패킷 조작)
- LSM (Linux Security Module)
 - 보안 관련 커널 결정 지점에 연결. (선제적 보안)
- cgroup
 - 특정 컨트롤 그룹에 속한 모든 프로세스의 활동에 연결. (컨테이너 단위 정책)

▶ 특정한 목적에 필요한 사전에 정의된 훅이 없다면

- 새로운 커널 프로브(kprobe) 또는 유저 프로브(uprobe)를 생성해서 커널 또는 유저 애플리케이션의 거의 모든 임의의 위치에 eBPF 프로그램을 부착할 수 있습니다.
- 커널 또는 사용자 공간 함수의 시작/종료 지점에 동적으로 연결. (가장 유연함)

*Hook은 eBPF 프로그램이라는 '플러그인'을 꽂는 '콘센트'와 같습니다.
콘센트마다 위치와 용도가 다르듯, 훅마다 연결되는 위치와 일을 수 있는 정보(컨텍스트)가 다릅니다.
어떤 문제를 해결하고 싶은가?'에 따라 가장 적절한 훅을 선택하는 것이 eBPF 프로그래밍의 첫걸음입니다.*



eBPF 상태계 (1/2)

▪ Major Infrastructure

- ▶ <https://ebpf.io/infrastructure/#major-infrastructure>



Linux Kernel

eBPF Runtime

The Linux kernel contains the eBPF runtime required to run eBPF programs. It implements the bpf(2) system call for interacting with programs, maps, BTF and various attachment points where eBPF programs can be executed from. The kernel contains a eBPF verifier in order to check programs for safety and a JIT compiler to translate programs to native machine code. User space tooling such as bpftool and libbpf are also maintained as part of the upstream kernel.



LLVM Compiler

eBPF Backend

The LLVM compiler infrastructure contains the eBPF backend required to translate programs written in a C-like syntax to eBPF instructions. LLVM generates eBPF ELF files which contain program code, map descriptions, relocation information and BTF meta data. These ELF files contain all necessary information for eBPF loaders such as libbpf to prepare and load programs into the Linux kernel. The LLVM project also contains additional developer tools such as an eBPF object file disassembler.



GCC Compiler

eBPF Backend

The GCC compiler comes with an eBPF backend starting from GCC 10. Up to that point, LLVM has been the only compiler which supports generating eBPF ELF files. The GCC port is roughly equivalent to the LLVM eBPF support. There are some missing bits of functionality but the GCC community is working to close these gaps over time. GCC also contains eBPF binutils as well as eBPF gdb support for debugging of eBPF code that is traditionally consumed by the Linux kernel. Included as part of this is an eBPF simulator for gdb.



bpftool

Command-line tool to inspect and manage eBPF objects

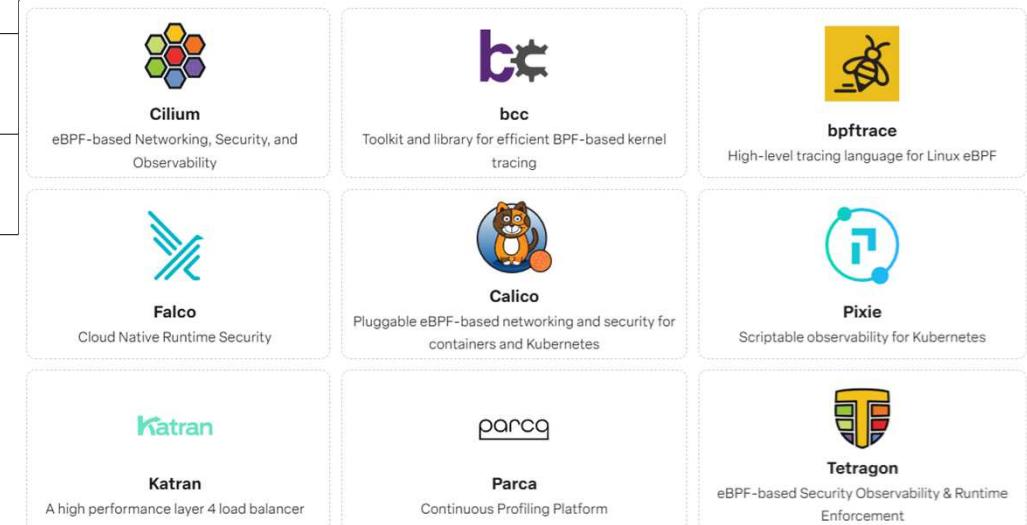
Powered by libbpf, bpftool is the reference utility to quickly inspect and manage BPF objects on a Linux system. Use it to list, dump, or load eBPF programs and maps, to generate skeletons for eBPF applications, to statically link eBPF programs from different object files, or to perform various other eBPF-related tasks.

eBPF 생태계 (2/2)

▪ Major Applications

▶ <https://ebpf.io/applications/>

카테고리	어플리케이션	설명
네트워킹 & 서비스 메시	Cilium	클라우드 네이티브 환경을 위한 네트워킹, 관찰성, 보안 플랫폼 사실상 eBPF의 킬러 애플리케이션.
	Calico	eBPF 데이터플레인을 지원하는 또 다른 인기 CNI
보안	Falco	CNCF 프로젝트. 런타임 보안 위협 탐지 도구
	Tetragon	Cilium의 하위 프로젝트 eBPF 기반 보안 관찰성 및 런타임 강화
관찰성 & 프로파일링	Parca	eBPF 기반 지속적인(Continuous) 시스템 프로파일링 도구
	Pixie	CNCF 프로젝트 쿠버네티스 애플리케이션을 위한 자동 관찰성 플랫폼
	bpftrace	고수준 추적 언어 시스템 분석가들에게 사랑받는 도구



eBPF의 진화

1993

cBPF의 탄생

"패킷 필터링"이라는 단 하나의 목적

2014

eBPF의 등장 (커널 3.18)

64비트, JIT 컴파일, 맵. "범용 가상 머신"으로의 재탄생

2015

kprobes 통합

커널 동적 트레이싱의 시대 개막. 관찰성 분야의 혁신 시작

2016

BCC 프로젝트 & Netflix

프로그래밍을 쉽게 만들고, 실제 프로덕션 환경에서 그 가치를 증명

2017

XDP & TC 통합

iptables를 뛰어넘는 고성능 네트워킹의 시작

2019

CO-RE & libbpf

"한 번 컴파일, 어디서든 실행". 이식성 문제를 해결하며 배포의 장벽을 허물다

2020

LSM 흑 통합

TOCTOU (Time-of-check to time-of-use) 공격을 막는 차세대 보안 패러다임 제시

2021

eBPF Foundation 설립

표준화와 생태계의 구심점 마련

2022

Windows eBPF 지원

리눅스를 넘어 모든 운영체제로의 확장 시작

2. eBPF's "Hello World"

첫번째 eBPF 프로그램 (1/3)

▪ BCC (BPF Compiler Collection)

- ▶ <https://github.com/iovisor/bcc>
- ▶ eBPF 프로그램 개발을 위한 "All-in-One" 프레임워크

▪ BCC is a toolkit for creating efficient kernel tracing and manipulation programs, and includes several useful tools and examples. It makes use of extended BPF (Berkeley Packet Filters), formally known as eBPF, a new feature that was first added to Linux 3.15. Much of what BCC uses requires Linux 4.1 and above.

- ▶ 프로그램 언어

구분	기능	프로그램 언어
User space	제어 로직	Python (또는 Lua/C++)
Kernel space	eBPF 프로그램	C

▶ 동작 방식

- Python 스크립트 실행 시, 내부에 포함된 C 코드를 런타임에 동적으로 컴파일하고 커널에 로드합니다.

▶ 장점

- 빠른 프로토タイ핑
 - 하나의 파일 안에서 모든 것을 해결할 수 있어 개발 속도가 빠릅니다.
- 쉬운 학습 곡선
 - 복잡한 빌드 과정 없이 Python의 편리함을 활용할 수 있습니다.

▶ 단점

- 배포의 어려움
 - 실행 환경에 LLVM/Clang 컴파일러와 커널 헤더가 필요합니다.

BCC는 마치 요리 초보자를 위한 '밀키트'와 같습니다.
 복잡한 재료 손질(컴파일, 로딩)은 BCC가 알아서 해주고,
 우리는 Python이라는 쉬운 레시피로 C라는 핵심 재료를
 요리하는 데만 집중하면 됩니다.



첫번째 eBPF 프로그램 (2/3)

▪ 개요

- ▶ 시스템에서 새로운 프로그램이 실행될 때마다(execve 시스템 콜) 터미널에 "Hello World!" 메시지를 출력

```
chapter2/hello.py
#!/usr/bin/python
from bcc import BPF

program = r"""
int hello(void *ctx) {
    bpf_trace_printk("Hello World!");
    return 0;
}
"""

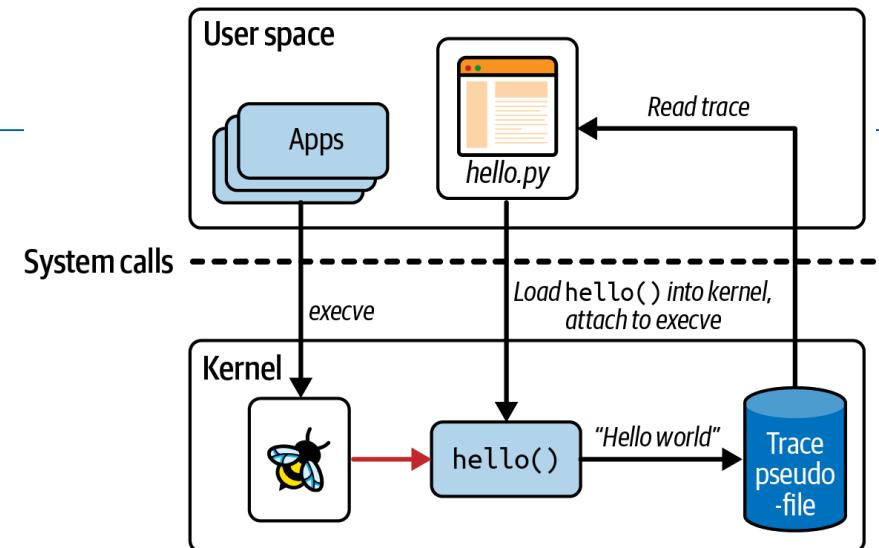
b = BPF(text=program)

syscall = b.get_syscall_fnname("execve")

b.attach_kprobe(event=syscall, fn_name="hello")

b.trace_print()
```

- 커널에서 실행되는 eBPF 프로그램 자체
 - bpf_trace_printk("Hello World!");
 - "Hello World!"메시지를 쓰기 위해 헬퍼 함수 `bpf_trace_printk()`를 사용합니다.
 - 헬퍼 함수는 eBPF를 cBPF와 구별하는 기능으로, eBPF 프로그램이 시스템과 상호 작용하기 위해 호출할 수 있는 함수 세트입니다.
 - printf와 유사하지만, 출력이 터미널이 아닌 커널의 중앙 추적 버퍼(trace pipe)로 향합니다.
 - return 0;
 - kprobe 프로그램의 반환 값은 현재 사용되지 않으므로, 관례적으로 0을 반환합니다.
- BPF 객체 생성 (컴파일 및 로드)
 - C 코드를 파싱하고 BCC 전용 매크로(예: BPF_HASH)를 실제 C 코드로 변환합니다.
 - 시스템에 설치된 clang을 호출하여 C 코드를 eBPF 바이트코드로 컴파일합니다.
 - bpf() 시스템 콜을 통해 컴파일된 바이트코드를 커널 메모리에 안전하게 로드합니다.
- 시스템 콜 이름은 아키텍처마다 다를 수 있습니다.
 - e.g., `_x64_sys_execve`, `_arm64_sys_execve`
 - 이 함수는 현재 시스템에 맞는 정확한 커널 함수 이름을 찾아주는 유ти리티입니다.
- execve 커널 함수의 시작 지점(event)에 우리가 커널에 로드한 hello eBPF 프로그램(fn_name)을 연결(attach)합니다.
 - 이제 execve가 호출될 때마다 hello 함수가 실행됩니다.
- `/sys/kernel/debug/tracing/trace_pipe` 파일을 열고, 새로운 내용이 기록될 때마다 읽어서 터미널에 출력하는 무한 루프를 실행합니다.



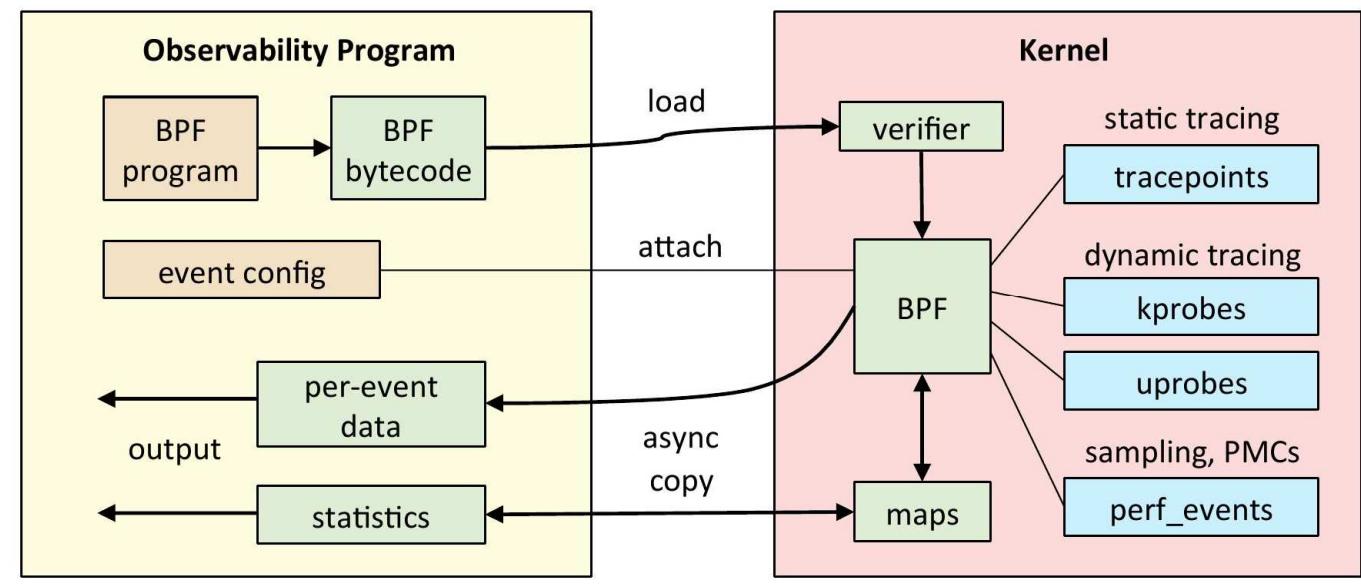
참고) kprobe

▪ kprobe

- ▶ 개발자와 시스템 관리자가 실행 중인 커널에 동적으로 후크(hook)를 삽입하여 디버깅, 성능 분석, 동작 모니터링을 할 수 있도록 해주는 강력한 메커니즘입니다.
 - 커널 소스 코드를 수정하거나 시스템을 재부팅할 필요 없이 커널의 거의 모든 지점에서 정보를 수집하거나 심지어 데이터/동작을 수정할 수 있게 해줍니다.
- ▶ kprobe는 커널 코드의 특정 지점(probe point)에 "프로브(probe)"를 배치하는 방식입니다.
 - 커널 실행이 이 프로브 지점에 도달하면, 사용자가 정의한 핸들러(handler) 함수가 실행되고, 핸들러가 완료되면 커널의 정상 실행이 재개됩니다.
- ▶ Kprobe와 eBPF
 - eBPF는 커널 내부에서 실행되는 안전하고 효율적인 사용자 정의 프로그램을 작성할 수 있게 해주며, kprobe는 이러한 eBPF 프로그램이 커널 함수의 특정 지점에 연결될 수 있는 "훅"을 제공합니다.
 - eBPF를 사용하면 커널 모듈을 작성하는 것보다 훨씬 안전하고 유연하게 커널 동작을 모니터링하고 분석할 수 있습니다.
 - libbpf와 같은 라이브러리를 통해 C로 eBPF 프로그램을 작성하고, 이를 kprobe에 연결하여 다양한 커널 트레이싱 및 디버깅 작업을 수행할 수 있습니다.

▪ Kprobe와 eBPF

- ▶ eBPF는 커널 내부에서 실행되는 안전하고 효율적인 사용자 정의 프로그램을 작성할 수 있게 해주며, kprobe는 이러한 eBPF 프로그램이 커널 함수의 특정 지점에 연결될 수 있는 "훅"을 제공합니다.
- ▶ eBPF를 사용하면 커널 모듈을 작성하는 것보다 훨씬 안전하고 유연하게 커널 동작을 모니터링하고 분석할 수 있습니다.
- ▶ libbpf와 같은 라이브러리를 통해 C로 eBPF 프로그램을 작성하고, 이를 kprobe에 연결하여 다양한 커널 트레이싱 및 디버깅 작업을 수행할 수 있습니다.



첫번째 eBPF 프로그램 (3/3)

- ✓ bcc 패키지 설치

```
[root@server1 ~]# dnf install -y bcc
```

- ✓ 실습에 사용할 소스코드를 github 에서 clone

```
[root@server1 ~]# cd; git clone https://github.com/lizrice/learning-ebpf.git
```

- ✓ ~/learning-ebpf/chapter2 디렉토리로 이동후, hello.py 실행

```
[root@server1 ~]# cd ~/learning-ebpf/chapter2
[root@server1 chapter2]# ./hello.py
b'<...>-21593 [003] ....2.1 4485.972499: bpf_trace_printk: Hello World!'
```

▶ 시스템 부팅 후 경과 시간 (타임스탬프)
 ▶ 이벤트가 발생한 CPU 코어 번호
 ▶ 이벤트를 발생시킨 프로세스의 이름

- ✓ (터미널#2) 새로운 터미널을 열고, 명령어를 실행하여 새로운 Process가 생성될때마다 <터미널#1>에 "Hello World!" 문자열이 출력되는것을 확인

```
[root@server1 ~]# ls
learning-ebpf
```

- ✓ bpf_trace_printk() 헬퍼 함수는 항상 동일한 사전 정의된 pseudo-file 위치인 /sys/kernel/debug/tracing/trace_pipe 로 출력을 보냅니다. cat을 사용하여 내용을 보고 이를 확인할 수 있습니다.

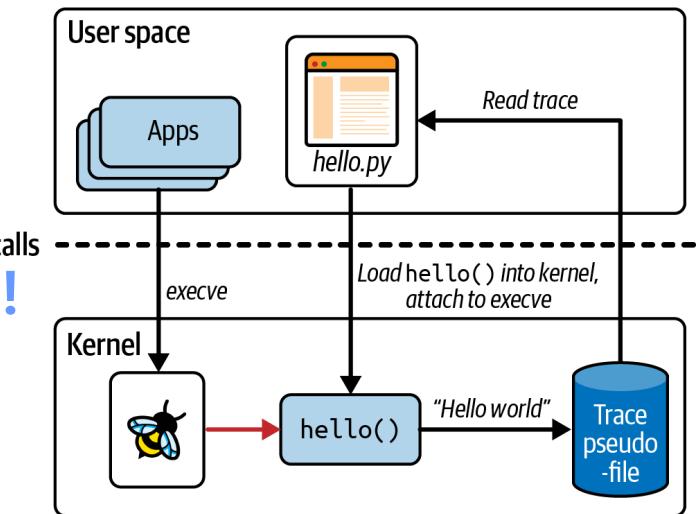
```
[root@server1 ~]# cat /sys/kernel/debug/tracing/trace_pipe
<...>-21849 [001] ....2.1 5828.471107: bpf_trace_printk: Hello World!
```

bpf_trace_printk 의 한계를 해결하기 위한 eBPF 맵 필요!!!



- ✓ (터미널#2) 새로운 터미널을 열고, 명령어를 실행하여 새로운 Process가 생성될때마다 <터미널#1>에 "Hello World!" 문자열이 출력되는것을 확인

```
[root@server1 ~]# ls
learning-ebpf
```



bpftool

▪ Linux 커널에 내장된 eBPF 서브시스템과 상호작용하기 위한 공식 유틸리티

- ▶ <https://github.com/libbpf/bpftool>
- ▶ eBPF 프로그램, 맵(map), 링크(link), 트램펄린(trampoline), 및 기타 eBPF 관련 객체를 관리하고 검사하며 디버깅하는 데 사용되는 다목적 명령줄 도구입니다.
 - 커널 버전 4.10부터 도입되었으며, eBPF 에코시스템의 발전에 따라 지속적으로 기능이 추가되고 있습니다.
 - bpftool은 주로 libbpf 라이브러리를 기반으로 구축되어 커널의 BPF 시스템 호출(syscall)과 직접 통신합니다.
 - eBPF 개발자, 시스템 관리자, 성능 분석가에게 필수적인 도구입니다.

✓ bpftool 패키지 설치하고 도움말을 확인

```
[root@server1 ~]# dnf install -y bpftool
```

✓ hello eBPF 프로그램의 정보를 확인

```
[root@server1 ~]# bpftool prog show | grep -A 6 hello
39: kprobe name hello tag f1db4e564ad5219a gpl
    loaded_at 2025-07-16T13:10:35+0900 uid 0
    xlated 104B jited 69B memlock 4096B
    btf_id 48
    pids hello.py(21587)
```



```
[root@server1 ~]# bpftool prog show id 39 --pretty
{
    "id": 39,
    "type": "kprobe",           ← kprobe 타입의 이벤트에 연결된 프로그램
    "name": "hello",
    "tag": "f1db4e564ad5219a", ← 이 프로그램을 식별하는 또 다른 식별자
    "gpl_compatible": true,
    "loaded_at": 1752639035,   ← 프로그램이 로드된 시간을 보여주는 타임스탬프
    "uid": 0,                  ← ID 0(루트)이 프로그램을 로드
    "orphaned": false,
    "bytes_xlated": 104,        ← Load 된 Bytecode 사이즈
    "jited": true,
    "bytes_jited": 69,          ← JIT-Compile된 머신코드 사이즈
    "bytes_memlock": 4096,
    "btf_id": 48,
    "pids": [
        {
            "pid": 21587,
            "comm": "hello.py"
        }
    ]
}
```

← Bpf Type Format : 이프로그램에 대한 BTF 정보 블럭

23

bpf_trace_printk 의 한계와 해결책

▪ bpf_trace_printk 의 한계

▶ 성능 저하

- 중앙 추적 버퍼는 커널 전체에서 공유되므로, 락(lock)으로 보호됩니다. 빈번한 호출은 시스템 전체에 성능 저하를 유발할 수 있습니다.

▶ 데이터 형식 제한

- 오직 포맷팅된 '문자열'만 전달 가능합니다. 구조화된 데이터(예: 숫자, 여러 필드를 가진 구조체)를 보내기 어렵습니다.

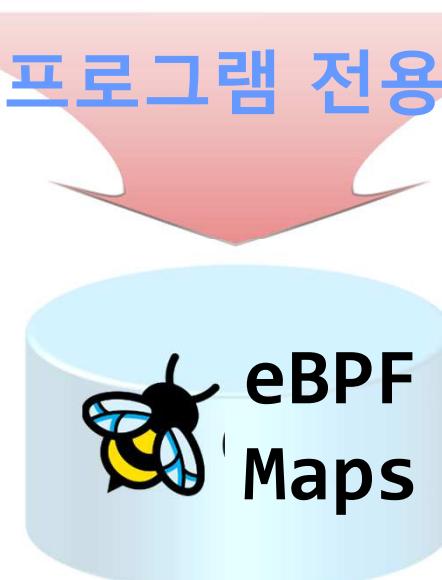
▶ 데이터 유실 가능성

- 매우 빠른 속도로 이벤트가 발생하면, 버퍼가 꽉 차서 일부 메시지가 유실될 수 있습니다.

▶ 단일 출력 지점

- 모든 eBPF 프로그램의 출력이 하나의 파일(/sys/kernel/debug/tracing/trace_pipe)로 섞여 나옵니다.

eBPF Maps 를 사용한 eBPF 프로그램 전용의 고성능 데이터 채널을 사용



2번째 eBPF 프로그램 - eBPF 맵을 이용한 데이터 집계 (1/3)

■ 개요

- ▶ 어떤 사용자가 프로그램을 몇 번 실행했는지 집계하기
- ▶ Kernel space - eBPF 프로그램
 - eBPF Map - 사용자 ID(UID)를 키(Key)로, 실행 횟수를 값(Value)으로 가지는 해시 테이블 맵을 사용합니다.
 - execve가 호출될 때마다, 현재 프로세스의 UID를 가져와 맵에 있는 해당 UID의 카운터를 1씩 증가시킵니다.
- ▶ User space - 사용자 어플리케이션
 - 주기적으로 맵의 내용을 읽어와 화면에 출력합니다.

2번째 eBPF 프로그램 - eBPF 맵을 이용한 데이터 집계 (2/3)

chapter2/hello-map.py

```

#!/usr/bin/python3
from bcc import BPF
from time import sleep

program = r"""
BPF_HASH(counter_table);

int hello(void *ctx) {
    u64 uid; u64 counter = 0; u64 *p;
    uid = bpf_get_current_uid_gid() & 0xFFFFFFFF;

    p = counter_table.lookup(&uid);
    if (p != 0) counter = *p;
    counter++;
    counter_table.update(&uid, &counter);
    return 0;
}

b = BPF(text=program)
syscall = b.get_syscall_fnname("execve")
b.attach_kprobe(event=syscall, fn_name="hello")

while True:
    sleep(2)
    s = ""
    for k,v in b["counter_table"].items():
        s += f"ID {k.value}: {v.value}\t"
    print(s)
"""

```

- BCC가 제공하는 매크로
 - counter_table이라는 이름의 해시 맵을 정의합니다.
 - 키와 값의 타입은 기본적으로 u64입니다.
- 현재 프로세스의 UID와 GID를 반환하는 헬퍼 함수.
 - 하위 32비트가 UID이므로 비트 마스킹(& 0xffffffff)을 합니다.
- uid를 키로 맵을 조회합니다.
 - 키가 존재하면 값에 대한 포인터를, 없으면 NULL을 반환합니다.
- uid 키에 counter 값을 저장(또는 덮어쓰기)합니다.
 - 이 역시 bpf_map_update_elem 헬퍼 함수로 변환됩니다.
- 참고) sys_enter 트레이스포인트
- b.attach_raw_tracepoint(tp="sys_enter", fn_name="hello")
 - 모든 시스템 호출(syscall)이 커널에 진입(enter)하는 바로 그 시점을 나타내는 커널 트레이스포인트
- while True: ... time.sleep(2)
 - 2초 간격으로 맵의 내용을 폴링(polling)하는 루프입니다.
- b["counter_table"]
 - BCC는 커널에 생성된 맵을 Python 딕셔너리처럼 만들어 줍니다.
 - b는 BPF 객체이고, counter_table은 c 코드에서 정의한 맵의 이름입니다.



2번째 eBPF 프로그램 - eBPF 맵을 이용한 데이터 집계 (3/3)

터미널 #1

✓ hello-map.py 실행

```
[root@server1 ~]# cd ~/learning-ebpf/chapter2
[root@server1 chapter2]# ./hello-map.py
```

```
ID 1000: 1
ID 1000: 1
ID 1000: 2
ID 1000: 3      ID 0: 1 ← sudo 프로세스 생성 1번, root 사용자가 ls 1번 수행
ID 1000: 4      ID 0: 1
ID 1000: 5      ID 0: 2
```

```
^CTraceback (most recent call last):
  File "/root/learning-ebpf/chapter2/hello-map.py", line 32, in <module>
    sleep(2)
KeyboardInterrupt
```



터미널 #2

✓ "ec2-user" 로 su 한후, id 를 확인합니다.

```
[root@server1 ~]# su - ec2-user
Last login: Wed Jul 16 15:15:54 KST 2025 on pts/1
```

```
[ec2-user@server1 ~]$ id
uid=1000(ec2-user) gid=1000(ec2-user) groups=1000(ec2-user),10(wheel)
```

→ 명령을 실행할 때마다 <터미널#1> 콘솔에 출력되는 내용 확인

```
[ec2-user@server1 ~]$ ls
[ec2-user@server1 ~]$ ls
[ec2-user@server1 ~]$ sudo ls
[ec2-user@server1 ~]$ ls
[ec2-user@server1 ~]$ sudo ls
```

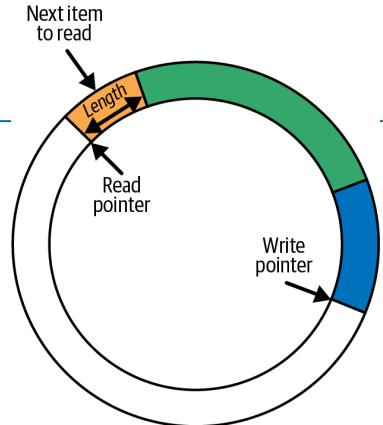
hello-map.py는 이벤트가 발생하지 않아도 2초마다 맵을 계속 읽어서 빈라인을 출력하는 비효율성이 존재합니다.

**"커널에서 이벤트가 발생했을 때만"
User space로 데이터를 보내는 것이 가장 이상적일텐데?**

3번째 eBPF 프로그램 - Perf Buffer를 이용한 이벤트 기반 통신 (1/3)

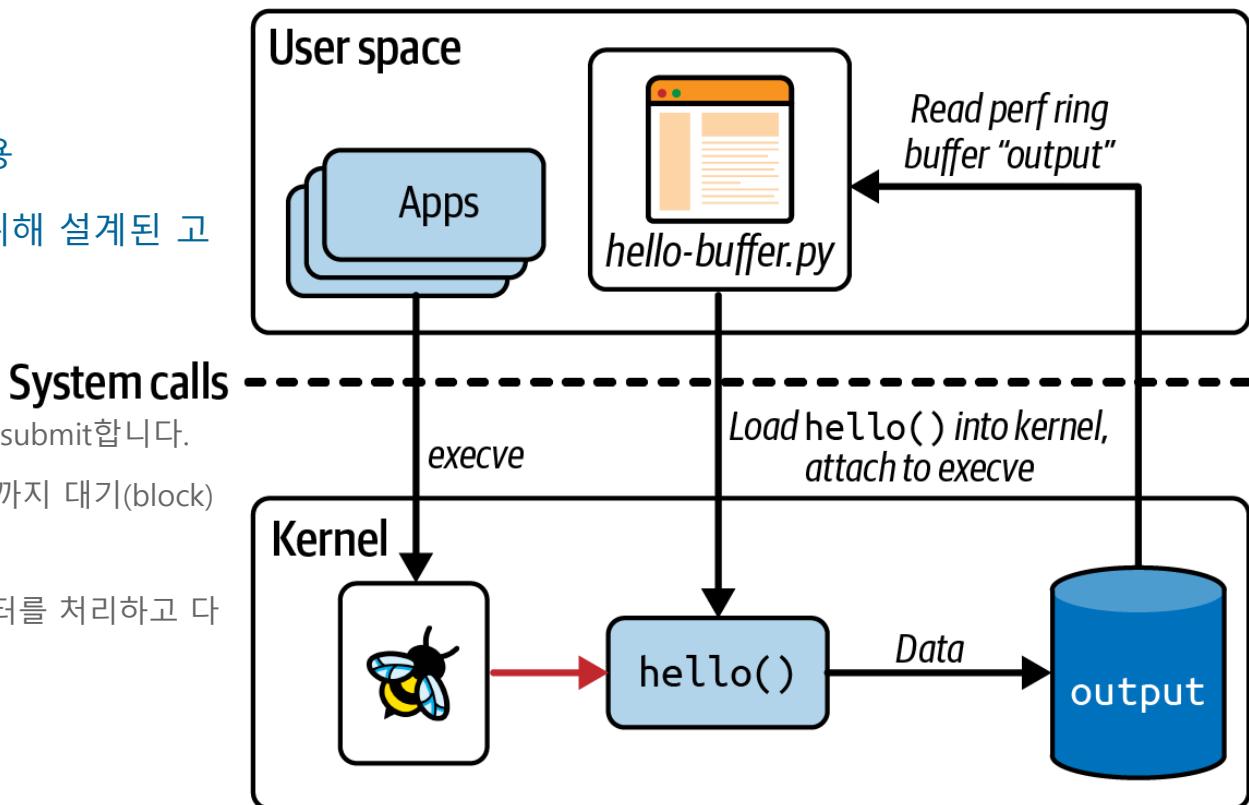
▪ 폴링 방식의 문제점

- ▶ hello-map.py는 아무 일도 일어나지 않아도 2초마다 맵을 계속 읽어야 합니다. 이는 비효율적입니다.
- ▶ "커널에서 이벤트가 발생했을 때만" 사용자 공간으로 데이터를 보내는 것이 가장 이상적입니다.



▪ Perf and Ring Buffer Maps

- ▶ BPF_MAP_TYPE_PERF_EVENT_ARRAY (줄여서 Perf Buffer) 이용
- ▶ 커널에서 User space로 데이터를 효율적으로 스트리밍하기 위해 설계된 고성능 링 버퍼(Ring Buffer) 맵입니다.
- ▶ 동작 방식
 - ① eBPF 프로그램은 이벤트가 발생하면, 보낼 데이터를 Perf Buffer에 submit합니다.
 - ② User space 어플리케이션은 이 버퍼에 새로운 데이터가 들어올 때까지 대기(block)합니다.
 - ③ 데이터가 들어오면, User space 어플리케이션 코드가 깨어나 데이터를 처리하고 다시 대기 상태로 들어갑니다.



3번째 eBPF 프로그램 - Perf Buffer를 이용한 이벤트 기반 통신 (2/3)

chapter2/hello-buffy.py

```

#!/usr/bin/python3
from bcc import BPF
program = r"""
BPF_PERF_OUTPUT(output);

struct data_t {
    int pid;  int uid;  char command[16];  char message[12];
};

int hello(void *ctx) {
    struct data_t data = {};
    char message[12] = "Hello World";
    data.pid = bpf_get_current_pid_tgid() >> 32;
    data.uid = bpf_get_current_uid_gid() & 0xFFFFFFFF;
    bpf_get_current_comm(&data.command, sizeof(data.command));
    bpf_probe_read_kernel(&data.message, sizeof(data.message), message);
    output.perf_submit(ctx, &data, sizeof(data));
    return 0;
}

b = BPF(text=program)
syscall = b.get_syscall_fnname("execve")
b.attach_kprobe(event=syscall, fn_name="hello")
def print_event(cpu, data, size):
    data = b["output"].event(data)
    print(f"{data.pid} {data.uid} {data.command.decode()}{data.message.decode()}")
b["output"].open_perf_buffer(print_event)

while True:
    b.perf_buffer_poll()
"""

```

- BCC가 제공하는 매크로
 - perf_event_output 타입의 eBPF 맵을 생성하고, output 이름으로 선언
 - 이 맵은 커널 eBPF 프로그램에서 User Process에게 비동기적으로 데이터를 전달하는데 사용
 - perf_event_output 맵은 perf event 메커니즘을 사용하여 데이터를 효율적으로 버퍼링하고 사용자 공간으로 전달합니다.
- 이 구조체는 output 맵을 통해 전송될 데이터의 형식을 결정합니다.
- void *ctx
 - kprobe의 경우 pt_regs 구조체의 포인터로, 현재 CPU 레지스터 상태등의 정보를 담고 있음
- 현재 프로세스의 PID와 TID(Thread ID)를 반환합니다.
- 현재 프로세스의 UID와 GID(Group ID)를 u64 값으로 반환합니다.
- 현재 실행 중인 프로세스의 명령 이름(task.comm)을 data.command에 복사
- 커널 공간의 메모리에서 데이터를 안전하게 읽어옵니다.
- data의 내용을 커널의 perf 버퍼에 기록하여 User space로 전송하도록 예약
- data = b["output"].event(data)
 - bcc의 perf_event_output 맵 객체(b["output"])의 event() 메소드를 호출합니다.
 - 이 메소드는 전달된 원시 data를 파이썬 객체로 역직렬화(deserialize)합니다.
 - data.pid, data.uid 등으로 접근할 수 있는 속성을 가진 객체를 생성합니다.
- output이라는 이름의 perf_event_output 맵을 열고, 커널에서 데이터가 전송될 때 호출할 콜백 함수로 print_event를 등록합니다.
- 이제 output.perf_submit으로 전송된 데이터는 print_event 함수에 의해 처리됩니다.
- 무한 루프를 실행하면서 b.perf_buffer_poll()을 계속 호출합니다.
 - 이 함수는 커널의 perf_event_output 버퍼를 주기적으로 폴링(polling)하여 새로운 이벤트(데이터)가 있는지 확인합니다.
 - 새로운 이벤트가 감지되면, 등록된 콜백 함수(print_event)를 호출하여 데이터를 처리합니다.
 - 이 루프는 Ctrl+C와 같은 인터럽트가 발생할 때까지 계속됩니다.

3번째 eBPF 프로그램 - Perf Buffer를 이용한 이벤트 기반 통신 (3/3)

터미널 #1

✓ hello-buffer.py 실행

```
[root@server1 ~]# cd ~/learning-ebpf/chapter2
[root@server1 chapter2]# ./hello-buffer.py
23487 1000 bash Hello World
23488 1000 bash Hello World
23489 0 sudo Hello World ← "sudo ls" 명령에 의해 출력
23490 0 sudo Hello World
^CTraceback (most recent call last):
  File "/root/learning-ebpf/chapter2./hello-buffer.py", line 40, in
    <module>
    b.perf_buffer_poll()
  File "/usr/lib/python3.9/site-packages/bcc/__init__.py", line 1727, in
    perf_buffer_poll
    lib.perf_reader_poll(len(readers), readers, timeout)
KeyboardInterrupt
```

터미널 #2

✓ "ec2-user"로 su 한 후, id 를 확인합니다.

```
[root@server1 ~]# su - ec2-user
Last login: Wed Jul 16 15:15:54 KST 2025 on pts/1

[ec2-user@server1 ~]$ id
uid=1000(ec2-user) gid=1000(ec2-user) groups=1000(ec2-user),10(wheel)
```

→ 명령을 실행할 때마다 <터미널#1> 콘솔에 출력되는 내용 확인

```
[ec2-user@server1 ~]$ ls
[ec2-user@server1 ~]$ sudo ls
```

3. Anatomy of an eBPF Program

LLVM Clang

▪ eBPF C 소스를 Bytecode로 컴파일하기

- ▶ eBPF 프로그램을 C 언어로 작성한 후 이를 Linux 커널에 로드하려면, C 소스 코드를 커널이 이해할 수 있는 eBPF 바이트코드(bytecode) 형식으로 변환해야 합니다. 이 변환 과정을 담당하는 핵심 도구가 바로 LLVM Clang 컴파일러입니다.

- LLVM Clang은 단순히 C 코드를 네이티브 머신 코드로 컴파일하는 것을 넘어, 특정 백엔드(backend)를 통해 다양한 중간 표현(IR)이나 다른 형식으로도 컴파일할 수 있는 유연성을 제공합니다. eBPF의 경우, Clang은 C 코드를 LLVM IR로 변환한 다음, 이 LLVM IR을 eBPF 아키텍처용 머신 코드로 변환합니다. 이 머신 코드가 바로 eBPF 바이트코드입니다.

▶ LLVM (Low Level Virtual Machine) 개요

- <https://llvm.org/>
- LLVM은 모듈식(modular)이고 재사용 가능한(reusable) 컴파일러 기술의 집합체입니다. 전통적인 컴파일러는 프론트엔드(소스 코드 파싱 및 의미 분석), 옵티마이저(중간 표현 최적화), 백엔드(타겟 아키텍처 코드 생성)로 구성되는데, LLVM은 이 각 단계를 독립적인 모듈로 분리하여 유연성을 극대화했습니다.

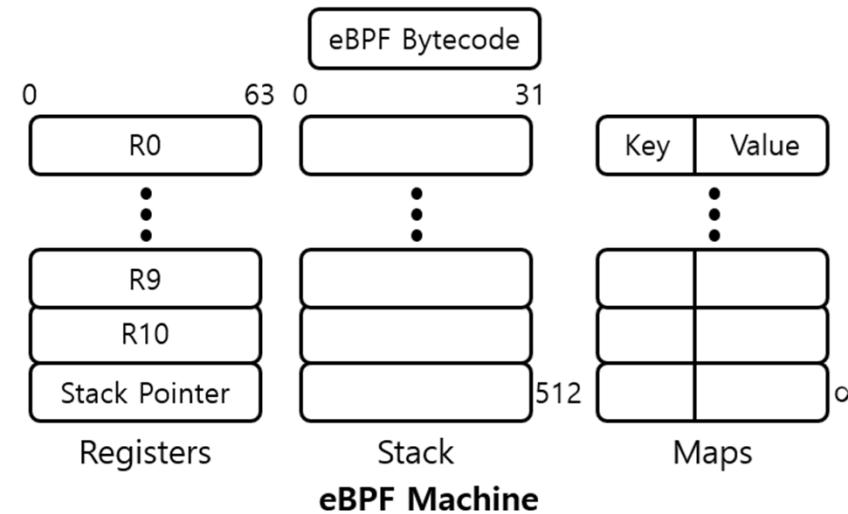


C(또는 Rust) 소스 코드는 eBPF 바이트코드로 컴파일되며,
 이는 JIT 컴파일(eBPF 프로그램이 커널에 로드될 때 한 번만 발생)되어
 네이티브 머신 코드 명령어로 변환됩니다.

eBPF Virtual Machine (1/3)

▪ eBPF 바이트코드를 실행하는 커널 내의 소프트웨어 기반 가상 머신(VM)

- ▶ 실제 하드웨어 CPU와 eBPF 프로그램 사이의 추상화 계층 역할.
 - eBPF 바이트코드는 Instruction Set으로 구성되며, 이러한 명령어는 (가상) eBPF 레지스터에서 작동합니다.
 - eBPF Instruction Set과 레지스터 모델은 일반적인 CPU 아키텍처에 깔끔하게 매핑되도록 설계되어 바이트코드에서 머신 코드로 컴파일하거나 해석하는 단계가 상당히 간단합니다.
- ▶ eBPF 프로그램이 어떤 CPU 아키텍처(x86, ARM 등)에서도 동일하게 동작하도록 보장.
- ▶ 구성 요소
 - 11개의 64비트 레지스터: 계산과 상태 저장을 위한 작업 공간.
 - 프로그램 카운터 (PC): 다음에 실행할 명령어를 가리킴.
 - 512바이트의 스택: 함수 호출 시 지역 변수나 인자를 저장하기 위한 작은 메모리 공간.



eBPF 프로그램은 CPU에서 직접 실행되는 것이 아니라, 먼저 커널에 내장된 eBPF 가상 머신 위에서 실행됩니다. 이 VM은 eBPF 프로그램에게 표준화된 실행 환경을 제공하며, 검증기가 이 VM의 규칙에 따라 프로그램의 안전성을 검사할 수 있게 해줍니다.

eBPF Virtual Machine (2/3)

▪ eBPF Registers

▶ 11개의 64비트 범용 레지스터: R0 ~ R10

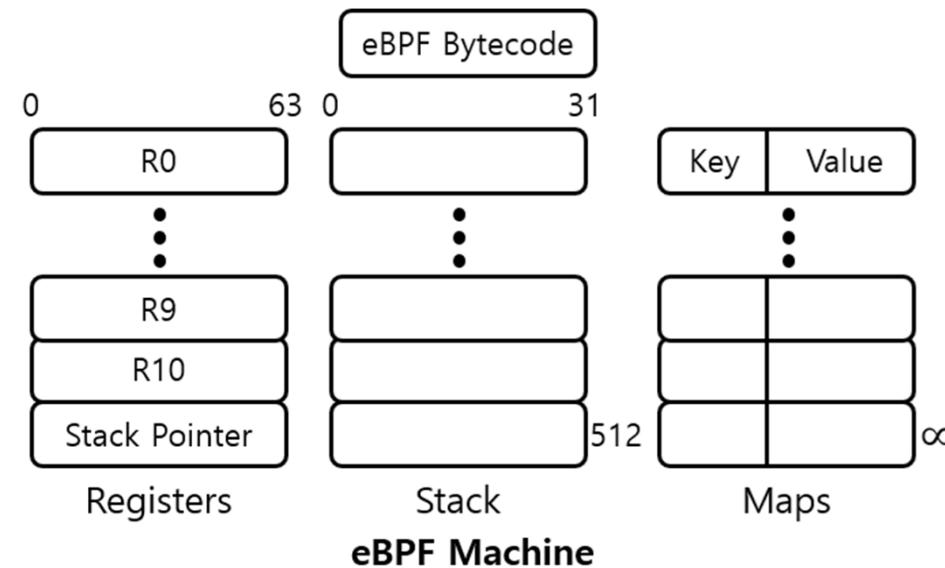
- <https://elixir.bootlin.com/linux/v5.19.17/source/include/uapi/linux/bpf.h>

레지스터	용도	
R0	Return Value	eBPF 프로그램 자체의 반환 값과 헬퍼 함수 호출의 반환 값을 저장
R1~R5	Argument Passing	헬퍼 함수를 호출할 때 인자를 전달하는 데 사용
R6~R9	Callee-saved	헬퍼 함수를 호출해도 내용이 보존됨 중요한 상태(예: 컨텍스트 포인터)를 백업하는 데 사용
R10	Stack Pointer	스택의 현재 위치를 가리키는 읽기 전용 레지스터

```
/* Register numbers */
enum {
    BPF_REG_0 = 0,
    BPF_REG_1,
    BPF_REG_2,
    BPF_REG_3,
    BPF_REG_4,
    BPF_REG_5,
    BPF_REG_6,
    BPF_REG_7,
    BPF_REG_8,
    BPF_REG_9,
    BPF_REG_10,
    __MAX_BPF_REG,
};
```

▶ eBPF의 ABI(Application Binary Interface)의 핵심

- https://en.wikipedia.org/wiki/Application_binary_interface



eBPF Virtual Machine (3/3)

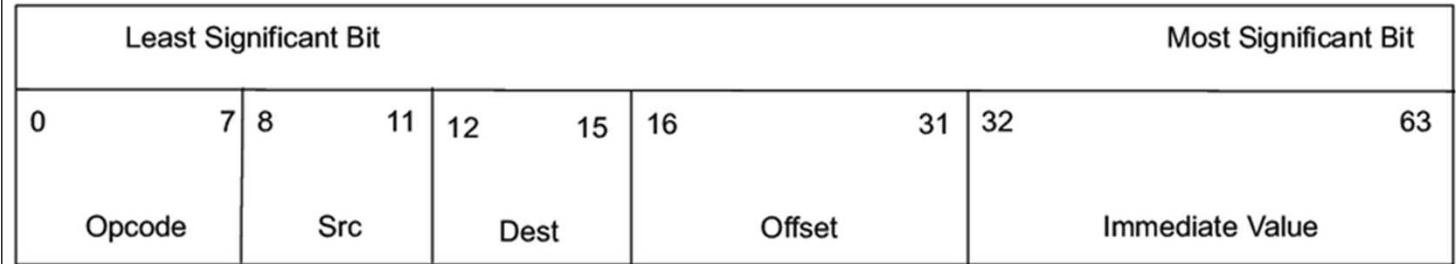
▪ eBPF Instructions

▶ eBPF instruction format

- <https://elixir.bootlin.com/linux/v5.19.17/source/include/uapi/linux/bpf.h>
- bpf_insn 이라는 구조체로 정의되어 있음

include/uapi/linux/bpf.h

```
struct bpf_insn {
    __u8 code;          // Opcode: 어떤 작업을 할지 정의
    __u8 dst_reg:4;    // Destination register
    __u8 src_reg:4;    // Source register
    __s16 off;          // Offset
    __s32 imm;          // Immediate value
};
```



▶ 명령어 종류

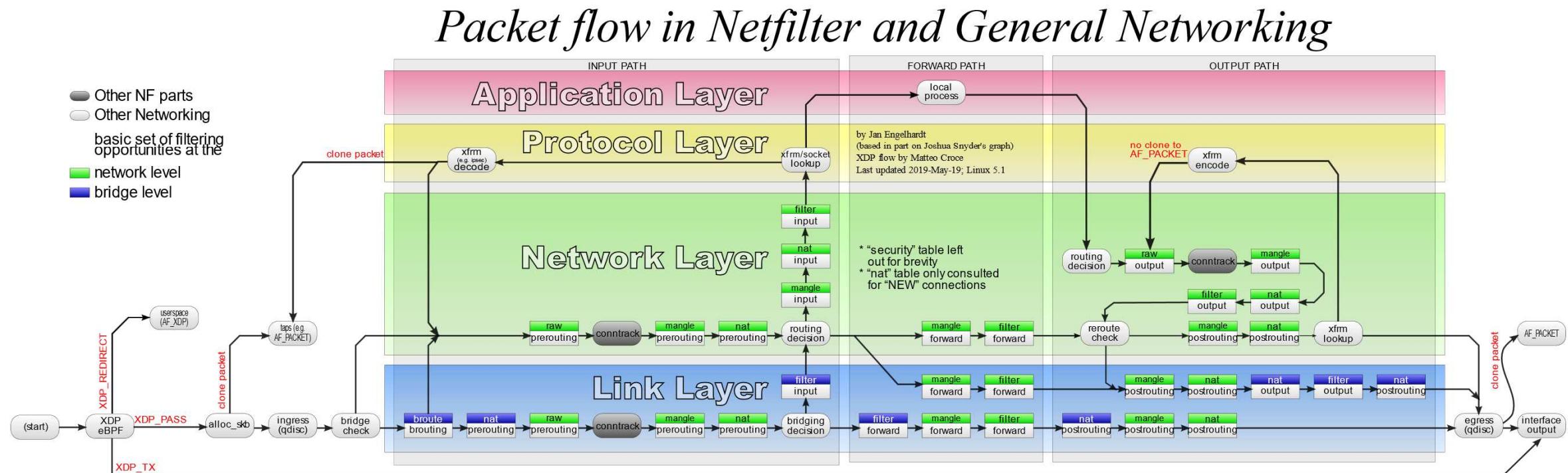
- 로딩/저장: 레지스터와 메모리 간 데이터 이동.
- 산술/논리 연산: ALU(Arithmetic Logic Unit) 연산.
- 분기: 조건에 따른 점프.

▶ 모든 eBPF 바이트코드는 이 bpf_insn 구조체의 연속으로 이루어져 있습니다.

- 하나의 명령어는 수행할 작업(opcode), 대상 레지스터, 소스 레지스터, 오프셋, 그리고 즉시 값(immediate value)으로 구성됩니다.
- 이 구조를 알면 llvm-objdump의 출력을 더 잘 이해할 수 있습니다.

▪ 리눅스 네트워크 스택

- ▶ <https://upload.wikimedia.org/wikipedia/commons/3/37/Netfilter-packet-flow.svg>



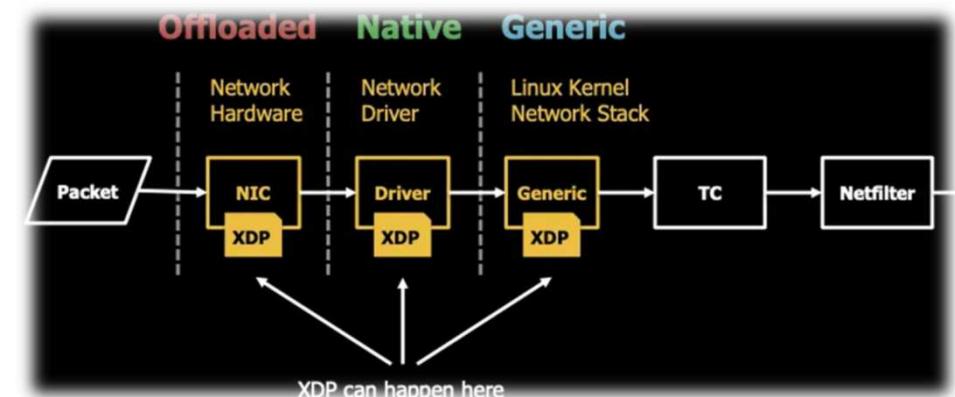
XDP (eXpress Data Path) (1/2)

▪ 고성능 패킷 처리 및 프로그래밍 가능한 데이터 경로를 제공

- ▶ 커널 내에서 패킷을 매우 이른 시점(네트워크 드라이버 수준)에서 처리할 수 있도록 하여, 기존 Linux 네트워킹 스택의 오버헤드를 줄이고 처리량을 극대화합니다.

▶ XDP의 주요 특징 및 동작 원리

- 드라이버 통합 (Driver Integration):
 - XDP는 네트워크 드라이버 내에 통합되어 작동합니다. 패킷이 NIC (Network Interface Card)로부터 수신되면, 커널의 일반적인 네트워킹 스택으로 올라가기 전에 XDP 프로그램이 실행됩니다. 이는 패킷이 소켓 버퍼(sk_buff)로 변환되기 전, 심지어 메모리 할당이 이루어지기 전인 매우 초기 단계에서 처리됨을 의미합니다. 이로 인해 불필요한 메모리 복사 및 스택 오버헤드가 크게 줄어듭니다.
- BPF 기반 (BPF-based):
 - XDP 프로그램은 eBPF (extended Berkeley Packet Filter) 언어로 작성됩니다.
- 다양한 동작 모드:
 - Native XDP
 - 네트워크 드라이버가 XDP를 직접 지원하는 모드입니다. 가장 높은 성능을 제공하며, NIC에서 직접 패킷을 처리합니다. 드라이버가 XDP API를 구현해야 합니다. (예: mlx5, ixgbe, virtio_net 등)
 - Offloaded XDP
 - 일부 고급 NIC는 XDP 프로그램을 하드웨어로 오프로드하는 기능을 제공합니다. 이 경우, 패킷 처리가 CPU가 아닌 NIC 자체에서 이루어져 CPU 부하를 더욱 줄일 수 있습니다. (예: mellanox mlx5 일부 기능)
 - Generic XDP
 - 드라이버가 XDP를 직접 지원하지 않는 경우에도 XDP를 사용할 수 있도록 커널에 일반적인 XDP 처리 로직이 구현되어 있습니다. 이 모드는 Native XDP보다 성능은 낮지만, 범용적으로 XDP를 테스트하거나 사용할 수 있게 해줍니다.



XDP (eXpress Data Path) (2/2)

▪ XDP의 장점

- ▶ XDP는 Linux 네트워킹의 패러다임을 바꾼 혁신적인 기술입니다. 패킷 처리 경로를 사용자에게 개방하고 프로그래밍 가능하게 함으로써, 기존에는 불가능했던 수준의 고성능 및 유연성을 제공합니다. 이는 클라우드 환경, 고성능 컴퓨팅, 통신 네트워크 등 현대의 복잡하고 요구사항이 많은 네트워크 환경에서 필수적인 기술로 자리매김하고 있습니다.
 - 초고속 패킷 처리: 패킷이 커널 스택의 상위 계층으로 올라가기 전에 처리되므로, 불필요한 컨텍스트 스위칭, 메모리 할당 및 복사 오버헤드를 제거하여 처리량을 대폭 향상시킵니다.
 - 낮은 지연 시간: 패킷 처리 경로가 짧아져 지연 시간이 감소합니다.
 - 프로그래밍 가능한 데이터 경로: eBPF를 통해 사용자가 직접 패킷 처리 로직을 커널 내에 구현할 수 있어 매우 유연합니다.
 - 자원 효율성: CPU 사용률을 낮추고 메모리 사용량을 최적화합니다.
 - 보안 강화: 악성 패킷을 커널 스택에 도달하기 전에 차단하여 DoS/DDoS 공격 방어에 효과적입니다.
 - 다양한 활용 사례: DDoS 완화, 로드 밸런싱, 네트워크 모니터링 및 분석, 방화벽 및 필터링, 네트워크 주소 변환 (NAT), VPN 가속

▪ XDP 개발 및 배포

- ▶ XDP 프로그램은 C 언어로 작성된 후 llvm과 clang을 사용하여 eBPF 바이트 코드로 컴파일됩니다. 컴파일된 바이트 코드는 libbpf 라이브러리와 같은 도구를 사용하여 커널에 로드되고 특정 네트워크 인터페이스에 연결됩니다.
- ▶ 기본적인 XDP 프로그램의 흐름:
 - xdp_md 구조체를 통해 패킷 데이터 및 메타데이터에 접근합니다.
 - 패킷 헤더(이더넷, IP, TCP/UDP 등)를 파싱하고 필요한 정보를 추출합니다.
 - 정의된 로직(예: 특정 포트의 패킷 드롭, 특정 IP 주소로 리다이렉트)에 따라 패킷을 처리합니다.
 - 적절한 XDP_ACTION 값을 반환합니다.

XDP_ACTION	설명
XDP_PASS	패킷을 정상적으로 통과시켜 네트워크 스택의 상위 계층(예: Netfilter, TCP/IP 스택)으로 이동
XDP_DROP	패킷을 즉시 버립니다. (방화벽 기능)
XDP_TX	패킷을 현재 수신 인터페이스로 다시 전송합니다. (로드 밸런싱 등)
XDP_REDIRECT	패킷을 다른 CPU 또는 다른 네트워크 인터페이스로 리다이렉션합니다. (고성능 포워딩)
XDP_ABORTED	프로그램 내부 오류로 패킷 처리가 중단되었음을 나타냅니다.

XDP - eBPF 프로그램 분석 (1/11)

▪ 개요

▶ 네트워크 패킷이 도착했을 때 "Hello World"라는 문자열과, 카운터를 trace pipe 기록하는

- 간단한 eBPF 프로그램을 C언어로 작성하고, LLVM Clang 으로 컴파일한 후 Bytecode를 살펴보고, bptool을 이용하여 런타임에 커널에 LOAD 하고

chapter3/hello.bpf.c

```
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

int counter = 0;

SEC("xdp")
int hello(struct xdp_md *ctx) {
    bpf_printk("Hello World %d", counter);
    counter++;
    return XDP_PASS;
}
char LICENSE[] SEC("license") = "Dual BSD/GPL";
```

- eBPF 관련 핵심 정의(예: struct bpf_insn, BPF_PROG_TYPE_XDP 등)가 포함된 표준 Linux 커널 헤더 파일입니다. eBPF 프로그램을 작성할 때 기본적으로 포함해야 하는 파일입니다.
- libbpf가 제공하는 헤더 파일로, eBPF 헬퍼 함수(예: bpf_printk, bpf_map_lookup_elem 등) 및 매크로(예: SEC)의 선언을 포함합니다. 이 헬퍼 함수들은 eBPF 프로그램이 커널 서비스를 호출할 수 있도록 해줍니다. libbpf 기반의 eBPF 프로그램을 작성할 때 흔히 사용됩니다.
- eBPF 프로그램 내에서 전역 변수는 map이 아닌 한 프로그램이 실행되는 동안 상태를 유지하지 못하고, JIT 컴파일 과정에서 최적화되거나 다른 방식으로 처리될 수 있으므로, 이 방식은 map을 사용하는 것이 더 일반적입니다. (하지만 이 간단한 예제에서는 작동합니다.)
- libbpf가 제공하는 매크로입니다.
 - 이 매크로는 다음 함수(hello)가 컴파일된 ELF 오브젝트 파일 내에서 어떤 섹션에 배치될지, 그리고 어떤 eBPF 프로그램 타입(BPF_PROG_TYPE_XDP)으로 로드될지를 컴파일러에게 알려줍니다.
- xdp: 이 프로그램이 XDP 프로그램임을 명시합니다. XDP 프로그램은 네트워크 드라이버 레벨에서 실행되며, 패킷이 네트워크 스택으로 진입하기 전에 가장 먼저 처리됩니다
- XDP 프로그램의 경우 반환 값은 패킷을 어떻게 처리할지 결정하는 XDP 액션 코드(예: XDP_PASS, XDP_DROP)여야 합니다.
- struct xdp_md *ctx: XDP 프로그램의 컨텍스트 포인터입니다.
 - xdp_md 구조체는 현재 처리 중인 네트워크 패킷에 대한 메타데이터(예: 패킷의 시작 주소, 끝 주소, 인터페이스 인덱스 등)를 포함합니다. 이 구조체를 통해 eBPF 프로그램은 패킷 데이터에 접근하고 수정할 수 있습니다.
- 커널의 디버그 메시지 버퍼(kmsg buffer)에 메시지를 출력합니다. 이는 dmesg 명령어나 /sys/kernel/debug/tracing/trace_pipe 파일을 통해 확인할 수 있습니다.
- XDP 프로그램의 반환 값
- eBPF 프로그램의 라이센스를 명시합니다.

XDP - eBPF 프로그램 분석 (2/11)

▪ clang -target bpf

- ▶ LLVM clang은 x86이나 ARM이 아닌, eBPF 가상 머신을 위한 bytecode를 생성

- ✓ ebpf C 소스파일을 컴파일하기 위해 llvm, clang 패키지 설치

```
[root@server1 ~]# dnf update -y
[root@server1 ~]# dnf install -y kernel-devel-$(uname -r)
[root@server1 ~]# dnf install -y clang llvm llvm-libs llvm-devel
[root@server1 ~]# dnf install -y man-pages
```

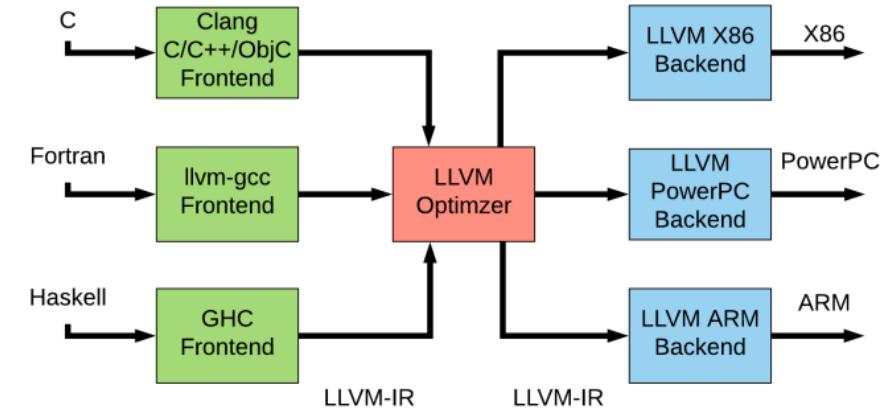
- ✓ hello.epbf.c 소스코드를 eBPF 가상 머신이 이해할 수 있는 bytecode로 컴파일하고, 컴파일된 Object 파일을 확인
 - clang은 x86이나 ARM이 아닌, eBPF 가상 머신을 위한 bytecode를 생성

```
[root@server1 ~]# cd ~/learning-ebpf/chapter3/
```

```
[root@server1 chapter3]# clang -target bpf \ ← eBPF 바이트코드를 생성하도록 지정
-I/usr/src/kernels/$(shell uname -r)/tools/lib \
-I/usr/src/kernels/$(shell uname -r)/tools/bpf/resolve_btfids/libbpf \
-g \ ← CORE eBPF 프로그램에 필요한 BTF 정보를 생성
-O2 \ ← 검증기를 통과하기 위해 필요한 최적화 레벨
-o hello.bpf.o -c hello.bpf.c
```

```
[root@server1 chapter3]# file hello.bpf.o
```

hello.bpf.o: ELF 64-bit LSB relocatable, eBPF, version 1 (SYSV), with debug_info, not stripped
 → LSB(최하위 비트) 아키텍처를 갖춘 64비트 플랫폼용 eBPF 코드를 포함하는 ELF(실행 및 링크 가능 형식) 파일임을 보여줍니다.



XDP - eBPF 프로그램 분석 (3/11)

▪ Disassembling with llvm-objdump

- ▶ llvm-objdump 유ти리티는 명령 줄에 명명된 객체 파일과 최종 링크 이미지의 내용을 인쇄합니다.

- ✓ hello.bpf.o 오브젝트파일 DisAssembling
 - llvm-objdump의 -S 옵션은 C 소스 코드와 어셈블리 코드를 함께 보여주어 분석을 용이하게 합니다.
 - C 코드 한 줄이 여러 개의 eBPF 명령어로 변환되는 것을 볼 수 있습니다.
 - 예를 들어, bpf_printk 호출을 위해 문자열 주소와 인자들을 각각 r1, r2 등에 로드한 후 call 명령어를 사용하는 것을 확인할 수 있습니다.

```
[root@server1 chapter3]# llvm-objdump -S hello.bpf.o
```

```
hello.bpf.o:      file format elf64-bpf
```

Disassembly of section xdp:

```
0000000000000000 <hello>:
;   bpf_printk("Hello World %d", counter);
 0:   18 06 00 00 00 00 00 00 00 00 00 00 00 00 00 00 r6 = 0x0 11
 2:   61 63 00 00 00 00 00 00 r3 = *(u32 *)(r6 + 0x0)
 3:   18 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 r1 = 0x0 11
 5:   b7 02 00 00 0f 00 00 00 r2 = 0xf
 6:   85 00 00 00 06 00 00 00 call 0x6
;   counter++;
 7:   61 61 00 00 00 00 00 r1 = *(u32 *)(r6 + 0x0)
 8:   07 01 00 00 01 00 00 00 r1 += 0x1
 9:   63 16 00 00 00 00 00 00 *(u32 *)(r6 + 0x0) = r1
;   return XDP_PASS;
10:   b7 00 00 00 02 00 00 00 r0 = 0x2
11:   95 00 00 00 00 00 00 00 exit
```

ebpf bytecode는 가상머신이 이해하는 Instruction Set

XDP - eBPF 프로그램 분석 (4/11)

▪ Loading & Pinning the eBPF Program into the Kernel

- ▶ bpftool을 사용하면 eBPF 프로그램을 커널에 로드할 수 있습니다.

✓ 컴파일된 오브젝트 파일에서 eBPF 프로그램을 로드하고 그것을 위치 /sys/fs/bpf/hello 에 "고정(Pinning)"합니다.

- '고정(pinning)'은 eBPF 객체(프로그램, 맵)의 생명주기를 관리하는 중요한 개념입니다.
- 파일처럼 경로를 부여함으로써, 다른 프로세스들이나 시스템 재부팅 후에도 해당 객체에 쉽게 다시 접근하고 재사용할 수 있게 해줍니다.

```
[root@server1 chapter3]# bpftool prog load hello.bpf.o /sys/fs/bpf/hello
```

↳ 로드된 프로그램을 BPF 가상 파일 시스템에 고정(pinning)합니다.

Pinning을 하지 않으면 bpftool 프로세스가 종료되면 eBPF 프로그램도 언로드 됩니다.

↳ hello.bpf.o 파일의 eBPF 바이트코드를 읽어 커널 메모리에 로드합니다.
이 과정에서 검증기가 실행됩니다.

```
[root@server1 chapter3]# ls -l /sys/fs/bpf/hello
-rw----- 1 root root 0 Jul 16 22:54 /sys/fs/bpf/hello
```

✓ bpftool 유ти리티는 커널에 로드된 모든 eBPF 프로그램을 나열할 수 있습니다

```
[root@server1 ~]# bpftool prog list | grep -A 6 hello
124: xdp name hello tag d35b94b4c0c10efb gpl
    loaded_at 2025-07-16T22:54:25+0900 uid 0
    xlated 96B jited 65B memlock 4096B map_ids 31,32
    btf_id 129
```

✓ 아래 명령어는 전부 동일한 결과를 나타냄

```
[root@server1 ~]# bpftool prog show id 124
[root@server1 ~]# bpftool prog show name hello
[root@server1 ~]# bpftool prog show tag d35b94b4c0c10efb
[root@server1 ~]# bpftool prog show pinned /sys/fs/bpf/hello
124: xdp name hello tag d35b94b4c0c10efb gpl
    loaded_at 2025-07-16T22:54:25+0900 uid 0
    xlated 96B jited 65B memlock 4096B map_ids 31,32
    btf_id 129
```

✓ eBPF 프로그램의 ID를 알면 eBPF 프로그램의 추가정보를 알 수 있습니다.

```
[root@server1 ~]# bpftool prog show id 124 --pretty
```

```
{
  "id": 124,
  "type": "xdp",
  "name": "hello",
  "tag": "d35b94b4c0c10efb",
  "gpl_compatible": true,
  "loaded_at": 1752674065,
  "uid": 0,
  "orphaned": false,
  "bytes_xlated": 96,
  "jited": true,
  "bytes_jited": 65,
  "bytes_memlock": 4096,
  "map_ids": [31,32],
  "btf_id": 129
}
```

커널이 부여한 고유 ID
프로그램 유형 (여기서는 xdp)

코드 내용 기반의 해시 값. 코드가 같으면 항상 동일.

로드된 시간을 보여주는 타임스탬프
사용자 ID 0(루트)이 프로그램을 로드

eBPF 바이트코드의 크기가 96 byte
JIT 컴파일되었고,
컴파일 결과 65바이트의 기계 코드가 나왔습니다.

ID 31와 32를 가진 BPF 맵을 참조합니다.
이 프로그램에 대한 BTF 정보 블록이 있음

XDP - eBPF 프로그램 분석 (5/11)

▪ The Translated Bytecode

- ▶ bytes_xlated 필드는 "translated" eBPF 코드의 바이트 수를 알려줍니다.
 - 이는 verifier를 통과한 후의 eBPF 바이트코드입니다. (커널에 의해 수정되었을 수 있음)

✓ bpftool을 사용하여 Verifier를 통과한 후의 bytecode를 확인

```
[root@server1 ~]# bpftool prog dump xlated name hello
int hello(struct xdp_md * ctx):
; bpf_printk("Hello World %d", counter);
 0: (18) r6 = map[id:31][0]+0
 2: (61) r3 = *(u32 *)(r6 +0)
 3: (18) r1 = map[id:32][0]+0
 5: (b7) r2 = 15
 6: (85) call bpf_trace_printk#-81552
; counter++;
 7: (61) r1 = *(u32 *)(r6 +0)
 8: (07) r1 += 1
 9: (63) *(u32 *)(r6 +0) = r1
; return XDP_PASS;
10: (b7) r0 = 2
11: (95) exit
```

**Ivm-objdump 의 출력에서 이전에 본 코드와 유사해 보이지만,
정확히 동일하지는 않습니다.**

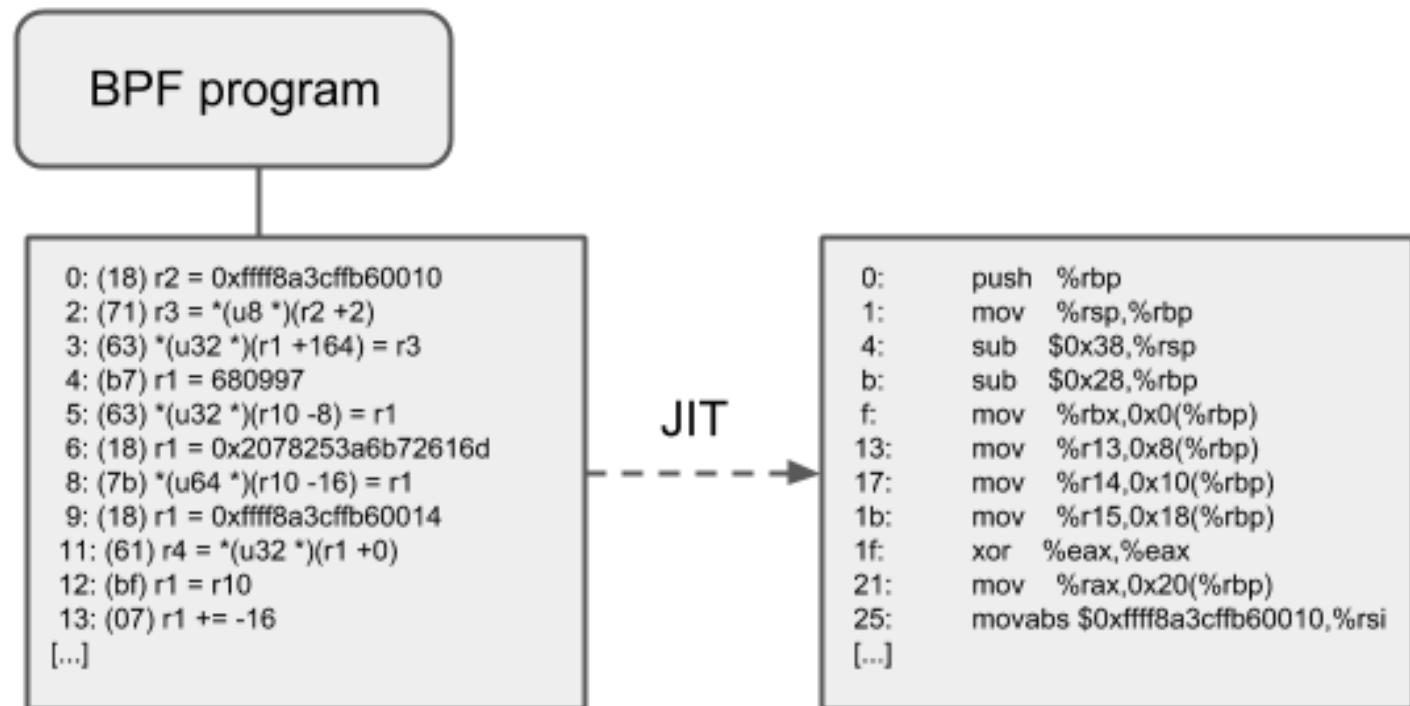
XDP - eBPF 프로그램 분석 (6/11)

▪ JIT-Compiled Machine Code

- translated bytecode는 꽤 낮은 수준이지만 아직 머신 코드는 아닙니다.
 - eBPF는 JIT 컴파일러를 사용하여 eBPF 바이트코드를 대상 CPU에서 네이티브로 실행되는 머신 코드로 변환합니다.
 - bytes_jited 필드는 변환후 eBPF 프로그램이 65바이트 길이임을 보여줍니다.
- bpftool 유틸리티는 어셈블리 언어로 이 JIT 코드의 덤프를 생성할 수 있습니다.

✓ JIT-Compile된 eBPF 프로그램의 머신코드의 어셈블리 확인

```
[root@server1 ~]# bpftool prog dump jited name hello
int hello(struct xdp_md * ctx):
0xfffffffffc03f76c8:
; bpf_printk("Hello World %d", counter);
 0:    nopl    (%rax,%rax)
 5:    nopl    (%rax)
 8:    pushq   %rbp
 9:    movq    %rsp, %rbp
 c:    pushq   %rbx
 d:    movabsq $-86562911461376, %rbx
17:    movl    (%rbx), %edx
1a:    movabsq $-108751350232824, %rdi
24:    movl    $15, %esi
29:    callq   0xfffffffffeb2ab948
; counter++;
2e:    movl    (%rbx), %edi
31:    addq    $1, %rdi
35:    movl    %edi, (%rbx)
; return XDP_PASS;
38:    movl    $2, %eax
3d:    popq    %rbx
3e:    leave
3f:    retq
40:    int3
```



XDP - eBPF 프로그램 분석 (7/11)

▪ Attaching to an Event

- ▶ hello BPF 프로그램이 커널에 로드되었지만, 아직 이벤트와 연관되지 않았으므로 아무것도 실행을 트리거하지 않습니다.
- ▶ 이벤트에 연결(attach)해야 합니다.
- ▶ BPF 프로그램 유형은, 연결하려는 이벤트 유형과 동일해야 합니다.
 - ✓ 시스템에 장착된 NIC 목록을 확인한 후, bpftool 을 이용하여 NIC 에 연결된 모든 eBPF 프로그램 확인해보면, 아직 eBPF 프로그램이 연결된 NIC는 없습니다.

```
[root@server1 ~]# ip --brief address
lo          UNKNOWN    127.0.0.1/8 ::1/128
ens160      UP         172.31.0.11/20
```

```
[root@server1 ~]# bpftool net list
xdp:
tc:
flow_dissector:
netfilter:
```

- ✓ ens160 NIC 의 XDP 이벤트에 hello eBPF 프로그램을 연결(attach)하고, 확인

```
[root@server1 ~]# bpftool net attach xdp name hello dev ens160
[root@server1 ~]# bpftool net list
xdp:
ens160(2) driver id 124
```

tc:
↑ JIT 컴파일된 eBPF 프로그램(ID:124) ens160 NIC의 XDP Hook에 연결되어 있음

```
flow_dissector:

netfilter:
[root@server1 ~]# ip link show dev ens160
2: ens160: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 xdp qdisc mq state UP mode DEFAULT group default qlen 1000
  link/ether 00:0c:29:3f:1e:16 brd ff:ff:ff:ff:ff:ff
  prog/xdp id 124 name hello tag d35b94b4c0c10efb jited
  altname enp3s0
```

XDP - eBPF 프로그램 분석 (8/11)

▪ hello XDP eBPF 프로그램 출력 확인

- ▶ hello eBPF 프로그램은 네트워크 패킷을 받을 때마다 Trace 출력을 /sys/kernel/debug/tracing/trace_pipe 파일에 기록하고 있습니다.
- ▶ 아래 명령으로 확인 가능
 - cat /sys/kernel/debug/tracing/trace_pipe
 - bpftool prog tracelog
- ✓ ens160 NIC 의 XDP 이벤트에 의해 실행되는 hello eBPF 프로그램의 출력 확인

```
[root@server1 ~]# bpftool prog tracelog
<idle>-0    [001] ...s2.. 43908.538696: bpf_trace_printk: Hello World 748
<idle>-0    [001] ...s2.. 43908.585215: bpf_trace_printk: Hello World 749
<idle>-0    [001] ...s2.. 43908.630831: bpf_trace_printk: Hello World 750
<idle>-0    [001] ...s2.. 43908.677057: bpf_trace_printk: Hello World 751
```

→ 이벤트를 발생시킨 프로세스는 idle이고, PID는 왜 0일까?



XDP 이벤트는 네트워크 패킷의 도착으로 인해 발생합니다. 이 패킷과 연관된 유저스페이스 프로세스가 없습니다. hello eBPF 프로그램이 트리거되는 시점에서 시스템은 메모리에서 패킷을 수신하는 것 외에는 패킷에 대해 아무것도 하지 않았으며 패킷이 무엇인지 또는 어디로 가는지 전혀 모릅니다.

```
[root@server1 ~]# cd ~/learning-ebpf/chapter2
[root@server1 chapter2]# ./hello.py
b'<...>-21593 [003] ....2.1 4485.972499: bpf_trace_printk: Hello World!'
```

→ 시스템 부팅 후 경과 시간 (타임스탬프)
 → 이벤트가 발생한 CPU 코어 번호
 → 이벤트를 발생시킨 프로세스의 이름과 PID

참고) 리눅스 OS의 PID 0번

▪ PID 0번은 idle 프로세스를 의미

- ▶ 이 프로세스는 매우 특별하며, 다른 일반적인 사용자 프로세스나 커널 스레드와는 다른 역할을 합니다.
 - <https://news.hada.io/topic?id=15239>
- ▶ 커널 생성 프로세스 (Kernel-Created Process):
 - PID 0번은 다른 모든 프로세스처럼 fork() 시스템 호출로 생성되는 것이 아니라, 시스템 부팅 시 커널이 직접 생성하는 최초의 프로세스(bootstrap process)입니다.
 - 따라서 /sbin/init (PID 1번)과 같은 일반적인 사용자 공간 프로세스들과는 달리, PID 0번은 커널 공간(kernel space)에서만 실행되는 커널 스레드입니다.
- ▶ 초기화 및 스케줄러 활성화:
 - 시스템 부팅 시 커널이 초기화되고 메모리 관리, 장치 드라이버 설정 등 기본적인 시스템 설정이 완료되면, PID 0번 프로세스가 생성됩니다.
 - PID 0번의 주요 역할 중 하나는 스케줄러를 초기화하고 활성화하는 것입니다. 일단 스케줄러가 활성화되면, 다른 프로세스들이 실행될 수 있는 기반이 마련됩니다.
- ▶ 유휴(Idle) 상태 관리:
 - 시스템에 실행할 프로세스가 없을 때, CPU는 유휴 상태가 됩니다. 이때 PID 0번(idle) 프로세스가 해당 CPU에서 실행됩니다.
 - 즉, CPU가 아무 작업도 하지 않고 있을 때, 실제로는 PID 0번이 실행되고 있는 것입니다.
 - 이는 CPU가 완전히 멈추는 것이 아니라, 전력 관리 기능을 통해 저전력 상태로 진입하거나, 다음 작업이 들어올 때까지 대기하는 상태를 의미합니다.
 - 따라서 각 CPU 코어마다 PID 0번의 인스턴스가 존재할 수 있으며, 이들은 해당 코어의 유휴 시간을 나타냅니다.

▪ 결론

- ▶ PID 0은 존재하며, 커널을 시작하는 스레드임.
- ▶ PID 0은 초기 커널 초기화 작업을 수행하고, 이후에는 idle 스레드로 전환됨.
- ▶ PID 0은 메모리 관리와는 관련이 없음.
- ▶ 다중 코어 시스템에서는 각 코어마다 idle 스레드가 있으며, 이들은 모두 스레드 그룹 0에 속함.

▪ 전역 변수와 맵

- ▶ C 코드의 전역 변수 int counter = 0;는 어떻게 구현될까요? → bpftool prog show 출력의 map_ids 필드가 힌트입니다.
 - Clang/LLVM 컴파일러는 eBPF 전역 변수를 eBPF 맵으로 변환합니다.
 - bpftool map dump id <map_id>로 확인:
 - .rodata 맵: bpf_printk의 포맷 문자열 "Hello World %d"와 같은 읽기 전용 데이터 저장.
 - .bss 맵: counter 변수와 같이 읽고 쓸 수 있는 전역 변수 저장.

✓ bpftool 유ти리티를 이용하여 커널에 로드된 맵을 표시

```
31: array name hello.bss flags 0x400           32: array name hello.rodata flags 0x80
      key 4B value 4B max_entries 1 memlock 8192B   key 4B value 15B max_entries 1 memlock 280B
      btf_id 129                                     btf_id 129 frozen
```

✓ bss 섹션의 내용을 확인

BTF 정보가 있을 때 (-g 옵션으로 컴파일)	BTF 정보가 없을 때 (-g 옵션으로 컴파일하지 않음)
[root@server1 ~]# bpftool map dump name hello.bss [{ "value": { ".bss": [{"counter": 4084}]} }]	[root@server1 ~]# bpftool map dump name hello.bss key: 00 00 00 00 value: 19 01 00 00 Found 1 element

✓ rodata 섹션의 내용을 확인

BTF 정보가 있을 때 (-g 옵션으로 컴파일)	BTF 정보가 없을 때 (-g 옵션으로 컴파일하지 않음)
[root@server1 ~]# bpftool map dump name hello.rodata [{ "value": { ".rodata": [{"hello.____fmt": "Hello World %d"}]} }]	[root@server1 ~]# bpftool map dump name hello.rodata key: 00 00 00 00 value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 20 25 64 00 Found 1 element

eBPF 프로그램 자체는 상태를 가질 수 없으므로, 전역 변수와 같은 상태를 저장하기 위해 맵을 활용합니다.
컴파일러가 이 과정을 자동으로 처리해주므로 개발자는 일반 C 코드처럼 전역 변수를 선언하기만 하면 됩니다.

XDP - eBPF 프로그램 분석 (10/11)

▪ Detaching the eBPF Program

- ▶ eBPF 프로그램과 이벤트의 연결을 끊습니다.
 - ▶ eBPF 프로그램은 커널에 로드된 상태로 남아있습니다.
- ✓ ens160 NIC 의 XDP 이벤트에 hello eBPF 프로그램을 끊고(detach)하고, 확인

```
[root@server1 ~]# bpftool net detach xdp dev ens160
```

```
[root@server1 ~]# bpftool net list
xdp:
```

```
tc:
```

```
flow_dissector:
```

```
netfilter:
```

```
[root@server1 ~]# ip link show dev ens160
```

```
2: ens160: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT group default qlen 1000
    link/ether 00:0c:29:3f:1e:16 brd ff:ff:ff:ff:ff:ff
    altname enp3s0
```

- ✓ 하지만 hello eBPF 프로그램은 여전히 커널에 로드되어 있습니다

```
[root@server1 ~]# bpftool prog show name hello
124: xdp  name hello  tag d35b94b4c0c10efb  gpl
        loaded_at 2025-07-16T22:54:25+0900  uid 0
        xlated 96B  jited 65B  memlock 4096B  map_ids 31,32
        btf_id 129
```

XDP - eBPF 프로그램 분석 (11/11)

▪ Unloading the eBPF Program

- ▶ bpftool prog load 의 역은 없습니다 (적어도 이 글을 쓰는 시점에서는).
- ▶ 하지만 고정된 pseudo-file을 삭제하면 커널에서 프로그램을 제거할 수 있습니다.

✓ hello eBPF 프로그램을 커널에서 제거

```
[root@server1 ~]# rm -f /sys/fs/bpf/hello
```

✓ hello eBPF 프로그램이 커널에서 제거되었기 때문에 아래 명령어는 출력이 없습니다.

```
[root@server1 ~]# bpftool prog show name hello  
[root@server1 ~]# bpftool prog list | grep hello
```

4. The bpf() System Call

bpf() System Call

▪ Userspace와 Kernel의 eBPF 엔진을 연결하는 단일 통로

- ▶ 유저스페이스 프로그램이 eBPF 프로그램을 커널에 로드하려면 bpf()라는 시스템 콜을 이용하여, eBPF 프로그램과 맵을 로드하고 상호 작용
- ▶ syscall 인터페이스는 유저스페이스 애플리케이션에서만 사용됩니다.
 - 커널에서 실행되는 eBPF 코드는 syscall을 사용하여 맵에 액세스하지 않고 헬퍼 함수를 사용하여 맵을 읽고 씁니다.

▪ bpf() 시스템콜의 시그니처

```
#include <linux/bpf.h>
int bpf(int cmd, union bpf_attr *attr, unsigned int size);
```

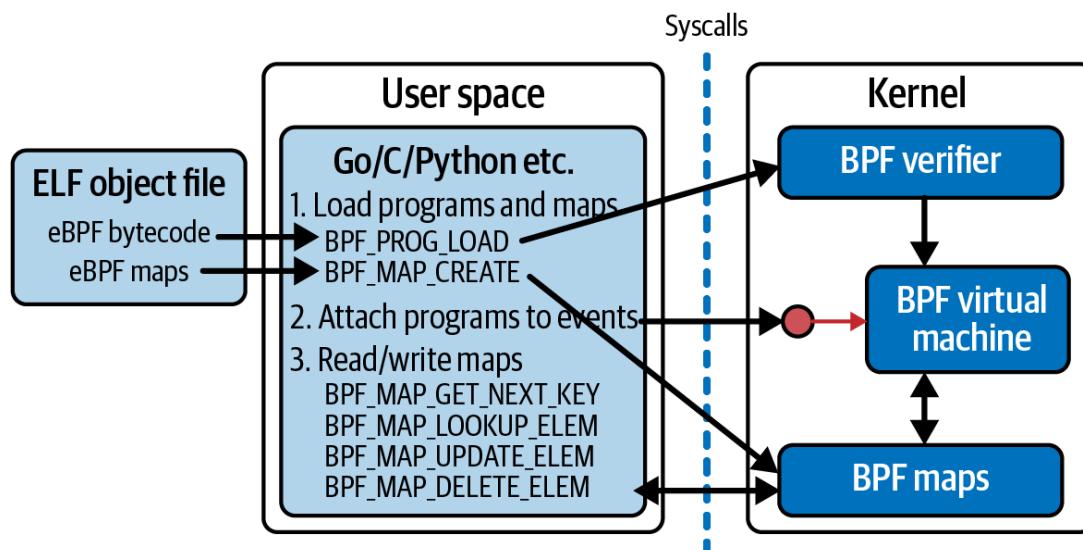
→ eBPF 프로그램과 맵을 조작하는 데 사용할 명령

→ cmd에 필요한 모든 파라미터를 담고 있는 공용체(union) 구조체의 포인터

→ attr 구조체의 크기.

→ 커널이 올바른 버전의 구조체를 사용하고 있는지 확인하기 위해 필요.

cmd	description
BPF_PROG_LOAD	eBPF 프로그램을 커널에 로드
BPF_MAP_CREATE	새로운 eBPF 맵을 생성
BPF_MAP_LOOKUP_ELEM	맵에서 특정 키에 해당하는 값을 조회
BPF_MAP_UPDATE_ELEM	맵의 특정 키에 대한 값을 생성하거나 수정
BPF_MAP_DELETE_ELEM	맵에서 특정 키-값 쌍을 삭제
BPF_PROG_ATTACH	프로그램을 특정 흙(cgroup, LSM 등)에 연결
BPF_PROG_DETACH	프로그램을 흙에서 분리
BPF_OBJ_PIN	프로그램이나 맵을 BPF 파일 시스템에 고정
BPF_OBJ_GET	고정된 경로로부터 eBPF 객체의 파일 디스크립터 GET



- 유저스페이스 프로그램은 syscall을 사용하여 커널에서 eBPF 프로그램 및 맵과 상호 작용합니다.
- 옆의 그림은 유저스페이스 코드가 eBPF 프로그램을 로드하고, 맵을 만들고, 프로그램을 이벤트에 연결하고, 맵의 키-값 쌍에 액세스하는 데 사용할 수 있는 일반적인 명령 중 일부를 간략하게 보여줍니다.

샘플 eBPF 프로그램 (1/2)

chapter4/hello-buffr-config.py

```

#!/usr/bin/python3
# -*- coding: utf-8 -*-
from bcc import BPF
import ctypes as ct
program = r"""
struct user_msg_t {
    char message[13];
};
BPF_HASH(config, u32, struct user_msg_t);
BPF_PERF_OUTPUT(output);

struct data_t {
    int pid; int uid; char command[16]; char message[12];
};

int hello(void *ctx) {
    struct data_t data = {};
    struct user_msg_t *p;
    char message[12] = "Hello World";
    data.pid = bpf_get_current_pid_tgid() >> 32;
    data.uid = bpf_get_current_uid_gid() & 0xFFFFFFFF;
    bpf_get_current_comm(&data.command, sizeof(data.command));

    p = config.lookup(&data.uid);
    if (p != 0) bpf_probe_read_kernel(&data.message, sizeof(data.message), p->message);
    else bpf_probe_read_kernel(&data.message, sizeof(data.message), message);
    output.perf_submit(ctx, &data, sizeof(data));
    return 0;
}

b = BPF(text=program)
syscall = b.get_syscall_fnname("execve")
b.attach_kprobe(event=syscall, fn_name="hello")
b["config"][ct.c_int(0)] = ct.create_string_buffer(b"Hey root!")
b["config"][ct.c_int(1000)] = ct.create_string_buffer(b"Hi ec2-user 1000!")
def print_event(cpu, data, size):
    data = b["output"].event(data)
    print(f"{data.pid} {data.uid} {data.command.decode()} {data.message.decode()}")
b["output"].open_perf_buffer(print_event)
while True:
    b.perf_buffer_poll()
"""

```

- ctypes는 C 언어 호환 데이터 타입을 제공하고 외부 함수를 호출할 수 있게 해줍니다.
- 여기서는 eBPF 맵에 데이터를 삽입할 때 C 타입으로 변환하는 데 사용됩니다.
- BPF_HASH 타입의 eBPF 맵을 생성하고 config라는 이름을 부여합니다.
- 이 맵은 사용자 공간(파이썬 스크립트)과 커널 eBPF 프로그램 간에 데이터를 공유하고, 특히 UID에 따라 다른 메시지를 동적으로 조회하는 데 사용됩니다.
- perf_event_output 타입의 eBPF 맵을 생성
- 이 맵은 커널 eBPF 프로그램에서 사용자 공간으로 비동기적으로 데이터를 전달하는 데 사용
- 커널 eBPF 프로그램에서 사용자 공간으로 전달할 데이터의 구조를 정의합니다.
- 이 구조체는 output 맵을 통해 전송될 데이터의 형식을 결정합니다.

- p = config.lookup(&data.uid);
 - config 맵에서 현재 프로세스의 UID(data.uid)를 키로 사용하여 값을 조회
- bpf_probe_read_kernel()
 - 커널 공간 메모리에서 데이터를 안전하게 읽어오는 헬퍼 함수
- data의 내용을 perf_event_output 맵(output)을 통해 사용자 공간으로 전송

- 사용자 공간에서 config eBPF 맵에 데이터를 삽입

- ✓ eBPF 프로그램은 execve 이벤트 발생 시, 현재 사용자의 UID로 config 맵을 조회.
- ✓ 만약 맵에 해당 UID에 대한 커스텀 메시지가 있으면 그 메시지를 사용하고, 없으면 기본 "Hello World" 메시지를 사용.
- ✓ 사용자 공간 Python 코드가 이 config 맵의 내용을 설정.

샘플 eBPF 프로그램 (2/2)

터미널 #1

✓ hello-buffer.py 실행

```
[root@server1 ~]# cd ~/learning-ebpf/chapter4
[root@server1 chapter2]# ./hello-buffer-config.py
42836 1000 bash Hi ec2-user
42837 1000 bash Hi ec2-user
42838    0 sudo Hey root! ← "sudo ls" 명령에 의해 출력
42839    0 sudo Hey root!
^CTraceback (most recent call last):
  File "/root/learning-ebpf/chapter4./hello-buffer-config.py", line 57, in
<module>
    b.perf_buffer_poll()
  File "/usr/lib/python3.9/site-packages/bcc/_init__.py", line 1727, in
perf_buffer_poll
    lib.perf_reader_poll(len(readers), readers, timeout)
KeyboardInterrupt
```

터미널 #2

✓ "ec2-user" 로 su 한후, id 를 확인합니다.

```
[root@server1 ~]# su - ec2-user
Last login: Wed Jul 16 15:15:54 KST 2025 on pts/1

[ec2-user@server1 ~]$ id
uid=1000(ec2-user) gid=1000(ec2-user) groups=1000(ec2-user),10(wheel)
```

→ 명령을 실행할 때마다 <터미널#1> 콘솔에 출력되는 내용 확인

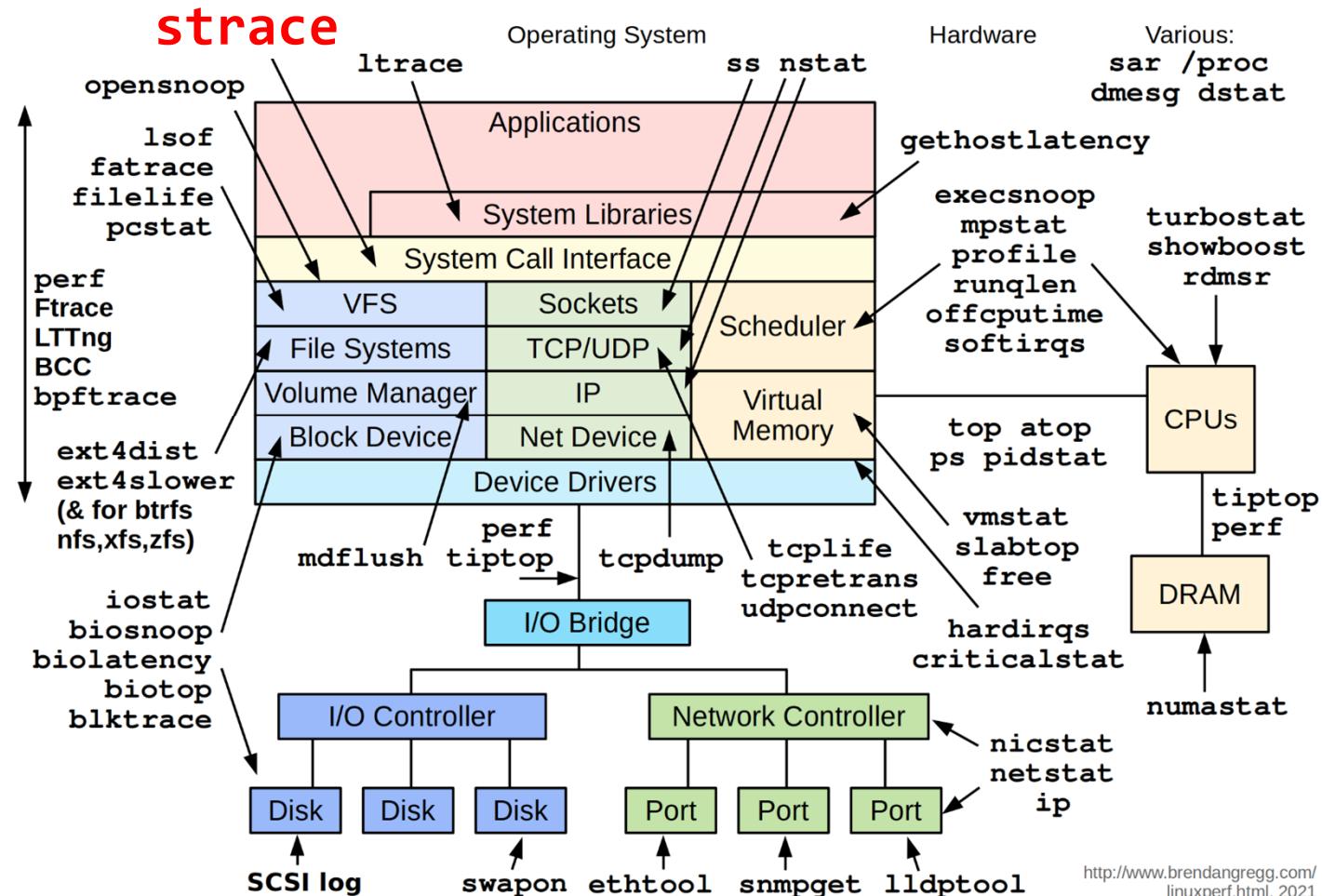
```
[ec2-user@server1 ~]$ ls
[ec2-user@server1 ~]$ sudo ls
```

참고) strace - syscall tracer

- 프로세스가 수행하는 시스템 호출 및 수신하는 시그널을 추적하는 명령줄 도구

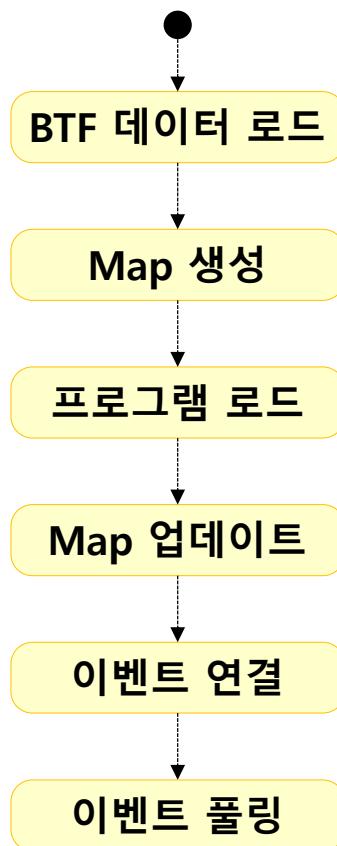
- ▶ <https://man7.org/linux/man-pages/man1/strace.1.html>
- ▶ <https://github.com/strace/strace>

Linux Performance Observability Tools



strace를 이용한 eBPF 프로그램의 bpf() 시스템 콜 분석 (1/7)

- ✓ strace 도구를 이용하여 eBPF 프로그램을 실행할 때 bpf() 시스템콜만 필터링



strace를 이용한 eBPF 프로그램의 bpf() 시스템 콜 분석 (2/7)

- **Loading BTF Data**

- ▶ CO-RE(Compile Once - Run Everywhere)를 위해, 그리고 bpftool 같은 도구가 맵과 프로그램의 구조를 이해하고 예쁘게 출력(pretty-print) 할 수 있도록, 탑 정보를 커널에 미리 로드합니다.

```
#include <linux/bpf.h>
int bpf(int cmd, union bpf_attr *attr, unsigned int size);
```

→ cmd에 필요한 모든 파라미터를 담고 있는 공용체(union) 구조체의 포인터
→ attr 구조체의 크기.
→ 커널이 올바른 버전의 구조체를 사용하고 있는지 확인하기 위해 필요.
→ eBPF 프로그램과 맵을 조작하는 데 사용할 명령

```
bpf(  
    BPF_BTF_LOAD, ← BTF (BPF Type Format) 정보를 커널에 로드  
    {  
        btf="\237\353\1\0\30\0\0\0\0\0\0\0\0\374\4\0\0\374\4\0\0\340\3\0\0\1\0\0\0\0\0\0\10"...,  
        btf_log_buf=NULL,  
        btf_size=2292,  
        btf_log_size=0,  
        btf_log_level=0  
    },  
    40  
) = 3
```

↑ 성공적으로 로드되었으며,
이 BTF 데이터를 가리키는 파일 디스크립터(fd 30) 반환됨

가장 먼저 하는 일은 '설계도(BTF)'를 커널에 올리는 것입니다.
이 설계도 정보는 나중에 맵을 만들거나 프로그램을 로드할 때 참조됩니다.
커널은 이 FD 3번을 통해 해당 설계도에 접근할 수 있습니다.

strace를 이용한 eBPF 프로그램의 bpf() 시스템 콜 분석 (3/7)

▪ Creating Maps

- ▶ 맵은 생성될 때마다 고유한 파일 디스크립터를 할당받습니다.

- 맵을 생성할 때 btf_fd를 전달하기 때문에, 연결고리가 형성되어 bpftool이 나중에 이 맵의 내용을 struct 필드 이름과 함께 예쁘게 출력할 수 있습니다.

output (PERF_EVENT_ARRAY)	config (HASH)
<pre>bpf(BPF_MAP_CREATE, { map_type=BPF_MAP_TYPE_PERF_EVENT_ARRAY, key_size=4, value_size=4, max_entries=128, map_flags=0, inner_map_fd=0, map_name="output", map_ifindex=0, btf_fd=0, btf_key_type_id=0, btf_value_type_id=0, btf_vmlinux_value_type_id=0, map_extra=0 }, 80) = 4</pre>	<pre>bpf(BPF_MAP_CREATE, { map_type=BPF_MAP_TYPE_HASH, key_size=4, value_size=13, max_entries=10240, map_flags=0, inner_map_fd=0, map_name="config", map_ifindex=0, btf_fd=3, btf_key_type_id=1, btf_value_type_id=4, btf_vmlinux_value_type_id=0, map_extra=0 }, 80) = 5</pre>

strace를 이용한 eBPF 프로그램의 bpf() 시스템 콜 분석 (4/7)

▪ Loading the Program

- ▶ bpf() 시스템콜을 이용하여 eBPF 프로그램을 커널에 로드

```
bpf(  
    BPF_PROG_LOAD,  
    {  
        prog_type=BPF_PROG_TYPE_KPROBE,  
        insn_cnt=47,  
        insns=0x7f2554d0e000,  
        license="GPL",  
        log_level=0,  
        log_size=0,  
        log_buf=NULL,  
        kern_version=KERNEL_VERSION(5, 14, 0),  
        prog_flags=0,  
        prog_name="hello",  
        prog_ifindex=0,  
        expected_attach_type=BPF_CGROUP_INET_INGRESS,  
        prog_btf_fd=3,  
        func_info_rec_size=8,  
        func_info=0x5648b0f8fc40,  
        func_info_cnt=1,  
        line_info_rec_size=16,  
        line_info=0x5648b098aa40,  
        line_info_cnt=17,  
        attach_btf_id=0,  
        attach_prog_fd=0,  
        fd_array=NULL  
    },  
    148  
) = 6
```

BPF 프로그램을 커널에 로드하는 명령
0/ BPF 프로그램이 커널의 특정 함수 진입점(kprobe)이나 리턴 지점(kretprobe)에 부착되었음
BPF Instruction의 개수
BPF Instruction(바이트코드)이 저장된 userspace 메모리 주소
BPF Verifier 로그의 상세도 수준, 0은 로그 출력이 없음을 의미합니다.

BPF 프로그램을 로드하려는 커널의 예상 버전
BPF 프로그램이 NIC에 부착될 때 사용되는 인터페이스 인덱스입니다. 0은 NIC와 관련이 없음을 의미
BPF Type Format (BTF) 정보가 포함된 파일 디스크립터입니다.
함수 정보가 저장된 사용자 공간 메모리 주소입니다.
제공된 함수 정보 레코드의 개수

eBPF 프로그램이 다른 BPF 프로그램에 Attach될 때 사용되는 BTF ID입니다. 0은 관련이 없음
다른 BPF 프로그램에 Attach될 때 사용되는 파일 디스크립터입니다. 0은 관련이 없음을 의미
특정 BPF 명령어(예: BPF_CALL)에서 참조되는 FD 배열의 주소입니다. NULL은 사용하지 않음을 의미

strace를 이용한 eBPF 프로그램의 bpf() 시스템 콜 분석 (5/7)

▪ Modifying a Map from User Space

chapter4/hello-buffr-config.py

```
b["config"][ct.c_int(0)] = ct.create_string_buffer(b"Hey root!")
```

< 현재까지의 FD >	
FD	Represents
3	BTF data
4	output perf buffer map
5	config hash table map
6	hello eBPF program

```
bpf(
    BPF_MAP_UPDATE_ELEM,
    {
        map_fd=5,
        key=0x7f25471bf190,
        value=0x7f2554682610,
        flags=BPF_ANY
    },
    32
) = 0
```

Python 코드의 `b["config"][0] = ...` 부분이 실행된 결과입니다.
사용자 공간에서 bpf() 시스템 콜을 통해 커널 내에 있는 맵의 내용을
직접 수정하고 있습니다.
이를 통해 eBPF 프로그램의 동작을 런타임에 변경할 수 있습니다.

- ✓ strace 출력에서 키나 값의 숫자 값을 알 수 없습니다. 그러나 bpftool을 사용하여 맵의 내용을 확인할 수 있습니다.

```
[root@server1 ~]# bpftool map dump name config
[{
    "key": 0,
    "value": {
        "message": "Hey root!"
    }
}, {
    "key": 1000,
    "value": {
        "message": [72, 105, 32, 101, 99, 50, 45, 117, 115, 101, 114, 32, 49]
    }
}]
```



strace를 이용한 eBPF 프로그램의 bpf() 시스템 콜 분석 (6/7)

▪ BPF Program and Map References (1/

- ▶ Userspace 프로세스가 bff() 시스템콜을 통해 eBPF 프로그램이 커널에 로드되면 FD가 반환되고, 이 FD를 통해 eBPF 프로그램 제어가 가능
- ▶ Userspace 프로세스는 eBPF 프로그램을 참조하는 FD의 소유자이며
 - 해당 프로세스가 종료되면 FD에 대한 참조카운트가 감소합니다.
 - 커널은 BPF 프로그램에 대한 참조가 남아 있지 않으면 BPF 프로그램을 제거합니다.

eBPF 프로그램을 파일 시스템에 pinning 하면 참조가 생성되어,
User Process가 종료되어도 eBPF 프로그램은 커널에서 제거되지 않습니다.

bpftrace prog load hello.bpf.o /sys/fs/bpf/hello

- ▶ 참조 카운터는 BPF 프로그램이 이를 트리거하는 후크에 연결될 때도 증가합니다.
- ▶ 참조 카운터의 동작은 BPF 프로그램 유형에 따라 달라집니다.
 - Trace에 관련된 프로그램(kprobe 및 tracepoint)은 항상 Userspace 프로세스와 연결됩니다.
 - 이러한 유형의 eBPF 프로그램의 경우 해당 프로세스가 종료되면 커널의 참조 카운터가 감소합니다.
 - 네트워크 스택 또는 cgroup 내에 연결된 eBPF 은 Userspace 프로세스와 연결되지 않습니다.
 - 이러한 유형의 eBPF 프로그램의 경우 해당 프로세스가 종료된 후에도 eBPF 프로그램은 커널에 그대로 유지됩니다.
 - 예시) ip link 명령으로 XDP 프로그램을 로드할 때

bpftrace net attach xdp name hello dev ens160

- ▶ BPF 객체의 수명
 - <https://facebookmicrosites.github.io/bpf/blog/2018/08/31/object-lifetime.html>

strace를 이용한 eBPF 프로그램의 bpf() 시스템 콜 분석 (7/7)

▪ bpf() 시스템 콜의 역할

- ▶ bpf()는 사용자 공간과 커널 eBPF 엔진 간의 유일하고 핵심적인 통신 채널입니다.
- ▶ 객체 관리: BPF_PROG_LOAD, BPF_MAP_CREATE 등의 명령어로 eBPF 프로그램과 맵의 생명주기를 관리합니다.
- ▶ 데이터 교환: BPF_MAP_*_ELEM 명령어로 커널 내 맵의 데이터를 읽고 쓸 수 있습니다.
- ▶ 추상화의 기반: libbpf, BCC, bpftool과 같은 모든 고수준 도구와 라이브러리는 내부적으로 이 bpf() 시스템 콜을 호출하여 동작합니다.
- ▶ 이벤트 연결: 일부 프로그램 유형(cgroup, LSM)은 BPF_PROG_ATTACH를 사용하지만, kprobe/tracepoint 같은 추적 도구는 perf_event_open, ioctl 등 다른 시스템 콜과 함께 사용하여 연결됩니다.

5. CO-RE, BTF, and Libbpf

eBPF의 이식성(Portability) 문제를 해결

CO-RE, BTF, and Libbpf

■ 학습 목표

- ▶ BCC의 런타임 컴파일 방식이 가진 한계를 이해합니다.
- ▶ CO-RE (Compile Once - Run Everywhere)의 개념과 필요성을 학습합니다.
- ▶ CO-RE를 구성하는 핵심 기술인 BTF, Libbpf, 컴파일러 지원에 대해 심층적으로 분석합니다.
- ▶ Libbpf와 BPF 스켈레톤을 사용한 현대적인 eBPF 애플리케이션 개발 방식을 익힙니다.

6. The eBPF Verifier

7. eBPF Program and Attachment Types

eBPF 프로그램을 커널의 어디에, 어떻게 연결할 것인가?

eBPF Program and Attachment Types (1/2)

▪ eBPF Program Type

- ▶ bfh.h 파일에 "enum bpf_prog_type" 으로 정의됨
 - <https://elixir.bootlin.com/linux/v5.19.14/source/include/uapi/linux/bpf.h#L922>
- ▶ eBPF 프로그램의 목적과 성격을 정의합니다.
 - 예: BPF_PROG_TYPE_KPROBE, BPF_PROG_TYPE_XDP
- ▶ 프로그램 유형에 따라 호출 가능한 헬퍼 함수가 결정됩니다.
- ▶ 프로그램 유형에 따라 반환 값(return code)의 의미가 달라집니다.
- ▶ 프로그램 유형에 따라 컨텍스트 인자의 구조체 타입이 결정됩니다.

▪ Context

- ▶ eBPF 프로그램에 첫 번째 인자로 전달되는 포인터.
- ▶ 이벤트가 발생한 시점의 상황 정보를 담고 있습니다.
 - kprobe: struct pt_regs * (레지스터 상태)
 - XDP: struct xdp_md * (패킷 메타데이터)
 - tracepoint: struct trace_event_raw_* * (트레이스포인트별 특정 구조체)

**eBPF Program Type은 eBPF 프로그램의 '직업'과 같습니다.
'네트워크 경찰(XDP)'인지, '시스템 감사관(kprobe)'인지에 따라
할 수 있는 일(헬퍼 함수)과 사용하는 서류 양식(컨텍스트
구조체)이 달라집니다. 검증기는 프로그램이 자신의 직업에 맞지
않는 행동을 하지 못하도록 감독합니다.**

bdf.h

```
/*
 * Note that tracing related programs such as
 * BPF_PROG_TYPE_{KPROBE,TRACEPOINT,PERF_EVENT,RAW_TRACEPOINT}
 * are not subject to a stable API since kernel internal data
 * structures can change from release to release and may
 * therefore break existing tracing BPF programs. Tracing BPF
 * programs correspond to /a/ specific kernel which is to be
 * analyzed, and not /a/ specific kernel /and/ all future ones.
 */
enum bpf_prog_type {
    BPF_PROG_TYPE_UNSPEC,
    BPF_PROG_TYPE_SOCKET_FILTER,
    BPF_PROG_TYPE_KPROBE,
    BPF_PROG_TYPE_SCHED_CLS,
    BPF_PROG_TYPE_SCHED_ACT,
    BPF_PROG_TYPE_TRACEPOINT,
    BPF_PROG_TYPE_XDP,
    BPF_PROG_TYPE_PERF_EVENT,
    BPF_PROG_TYPE_CGROUP_SKB,
    BPF_PROG_TYPE_CGROUP_SOCK,
    BPF_PROG_TYPE_LWT_IN,
    BPF_PROG_TYPE_LWT_OUT,
    BPF_PROG_TYPE_LWT_XMIT,
    BPF_PROG_TYPE SOCK_OPS,
    BPF_PROG_TYPE_SK_SKB,
    BPF_PROG_TYPE_CGROUP_DEVICE,
    BPF_PROG_TYPE_SK_MSG,
    BPF_PROG_TYPE_RAW_TRACEPOINT,
    BPF_PROG_TYPE_CGROUP_SOCK_ADDR,
    BPF_PROG_TYPE_LWT_SEG6LOCAL,
    BPF_PROG_TYPE_LIRC_MODE2,
    BPF_PROG_TYPE_SK_REUSEPORT,
    BPF_PROG_TYPE_FLOW_DISSECTOR,
    BPF_PROG_TYPE_CGROUP_SYSCTL,
    BPF_PROG_TYPE_RAW_TRACEPOINT_WRITABLE,
    BPF_PROG_TYPE_CGROUP_SOCKOPT,
    BPF_PROG_TYPE_TRACING,
    BPF_PROG_TYPE_STRUCT_OPS,
    BPF_PROG_TYPE_EXT,
    BPF_PROG_TYPE_LSM,
    BPF_PROG_TYPE_SK_LOOKUP,
    BPF_PROG_TYPE_SYSCALL, /* a program that can execute syscalls */
};
```

eBPF Program and Attachment Types (2/2)

▪ eBPF Attachment Type

- ▶ bfh.h 파일에 "enum bpf_attach_type" 으로 정의됨
 - <https://elixir.bootlin.com/linux/v5.19.14/source/include/uapi/linux/bpf.h#L957>
- ▶ eBPF 프로그램이 첨부되는 위치를 더 구체적으로 정의합니다.
- ▶ 많은 eBPF 프로그램 유형의 경우 Attachment Type은 프로그램 유형에서 유추할 수 있지만,
- ▶ 일부 프로그램 유형은 커널의 여러 다른 지점에 첨부될 수 있으므로 첨부 유형도 지정해야 합니다.

bdf.h

```
enum bpf_attach_type {
    BPF_CGROUP_INET_INGRESS,
    BPF_CGROUP_INET_EGRESS,
    BPF_CGROUP_INET_SOCK_CREATE,
    BPF_CGROUP_SOCK_OPS,
    BPF_SK_SKB_STREAM_PARSER,
    BPF_SK_SKB_STREAM_VERDICT,
    BPF_CGROUP_DEVICE,
    BPF_SK_MSG_VERDICT,
    BPF_CGROUP_INET4_BIND,
    BPF_CGROUP_INET6_BIND,
    BPF_CGROUP_INET4_CONNECT,
    BPF_CGROUP_INET6_CONNECT,
    BPF_CGROUP_INET4_POST_BIND,
    BPF_CGROUP_INET6_POST_BIND,
    BPF_CGROUP_UDP4_SENDMSG,
    BPF_CGROUP_UDP6_SENDMSG,
    BPF_LIRC_MODE2,
    BPF_FLOW_DISSECTOR,
    BPF_CGROUP_SYSCTL,
    BPF_CGROUP_UDP4_RECVMSG,
    BPF_CGROUP_UDP6_RECVMSG,
    BPF_CGROUP_GETSOCKOPT,
    BPF_CGROUP_SETSOCKOPT,
    BPF_TRACE_RAW_TP,
    BPF_TRACE_FENTRY,
    BPF_TRACE_FEXIT,
    BPF MODIFY RETURN,
    BPF LSM MAC,
    BPF TRACE ITER,
    BPF_CGROUP_INET4_GETPEERNAME,
    BPF_CGROUP_INET6_GETPEERNAME,
    BPF_CGROUP_INET4_GETSOCKNAME,
    BPF_CGROUP_INET6_GETSOCKNAME,
    BPF_XDP_DEVMAP,
    BPF_CGROUP_INET_SOCK_RELEASE,
    BPF_XDP_CPUMAP,
    BPF_SK_LOOKUP,
    BPF_XDP,
    BPF_SK_SKB_VERDICT,
    BPF_SK_REUSEPORT_SELECT,
    BPF_SK_REUSEPORT_SELECT_OR_MIGRATE,
    BPF_PERF_EVENT,
    BPF_TRACE_KPROBE_MULTI,
    __MAX_BPF_ATTACH_TYPE
};
```

2 Major Categories of Program Types

	추적 (Tracing / Probing)	네트워킹 및 보안 (Networking / Security)
목적	시스템의 동작을 관찰하고 데이터를 수집	시스템의 동작에 개입하고 변경
특징	일반적으로 시스템의 동작을 변경하지 않음 (읽기 전용) 반환 값은 보통 무시됨.	패킷을 수정, 폐기(drop), 재전송(redirect)하거나, 특정 동작을 허용/거부할 수 있음 반환 값이 시스템의 다음 행동을 결정.
종류	kprobe, tracepoint, perf_event, fentry 등	XDP, TC (Traffic Control), cgroup, LSM 등

eBPF 프로그램은 크게 두 부류로 나눌 수 있습니다.
 하나는 시스템을 몰래 지켜보는 '관찰자'이고,
 다른 하나는 시스템의 흐름에 직접 개입하는 '조정자'입니다.
 이 두 가지 분류를 이해하면 새로운 프로그램 유형을 접했을 때 그
 목적을 쉽게 파악할 수 있습니다.

Tracing 관련 eBPF 프로그램 유형

▪ Tracing 관련 eBPF 프로그램 유형

- ▶ kprobe, tracepoint, raw tracepoint, fentry/fexit probe, perf 이벤트에 첨부된 프로그램은 모두 커널의 eBPF 프로그램이 이벤트에 대한 추적 정보를 유저스페이스로 보고하는 효율적인 방법을 제공하도록 설계되었습니다.
- ▶ 이러한 추적 관련 유형은 첨부된 이벤트에 대한 응답으로 커널이 동작하는 방식에 영향을 미치지 않습니다.
- ▶ 이러한 프로그램은 때때로 "perf-related" 프로그램이라고도 합니다.
 - 예를 들어, bpftool perf sho 명령을 사용하면 다음과 같이 perf 관련 이벤트에 첨부된 프로그램을 볼 수 있습니다.

```
$ sudo bpftool perf show

pid 232272 fd 16: prog_id 392 kprobe func __x64_sys_execve offset 0
pid 232272 fd 17: prog_id 394 kprobe func do_execve offset 0
pid 232272 fd 19: prog_id 396 tracepoint sys_enter_execve
pid 232272 fd 20: prog_id 397 raw_tracepoint sched_process_exec
pid 232272 fd 21: prog_id 398 raw_tracepoint sched_process_exec
```

Tracing Type 1: kprobes and kretprobes

▪ kprobe & kretprobe

- ▶ kprobe (Kernel Probe)
 - 커널 내의 거의 모든 함수의 시작 지점에 동적으로 프로브를 설치
- ▶ kretprobe (Kernel Return Probe)
 - 함수의 종료(반환) 지점에 프로브를 설치. 함수의 실행 시간 측정이나 반환 값 확인에 유용

▪ 장점

- ▶ 매우 유연함.
- ▶ 커널의 거의 모든 함수를 추적 가능

▪ 단점

- ▶ 추적 대상 함수가 커널 버전마다 변경되거나 사라질 수 있음 (API 안정성 보장 안됨)
- ▶ 컴파일러 최적화로 함수가 인라인(inline) 처리되면 프로브를 설치할 수 없음

▪ libbpf 섹션 매크로 선언

- ▶ SEC("kprobe/function_name")
- ▶ SEC("kretprobe/function_name")
- ▶ SEC("ksyscall/syscall_name")
 - 시스템콜에 대한 아키텍처 독립적인 선언

probe는 커널 내부를 들여다보는 가장 강력하고 유연한 방법 중 하나입니다.
 마치 커널 코드의 아무 곳에나 현미경을 설치하는 것과 같습니다.
 하지만 그만큼 커널 내부 변화에 민감하기 때문에,
 안정적인 API가 아닌 내부 함수를 추적할 때는 주의가 필요합니다.

Tracing Type 2: Fentry and Fexit

▪ 개념:

- ▶ Kprobe의 현대적인 대체재 (커널 5.5+).
- ▶ BPF Trampoline 기술을 기반으로 동작.

▪ 장점

- ▶ 더 높은 성능
 - Kprobe보다 오버헤드가 훨씬 적음.
- ▶ 더 나은 인자 접근
 - fexit 프로브에서 함수의 입력 인자와 반환 값을 모두 접근할 수 있음. (kretprobe는 반환 값만 접근 가능)

▪ 단점

- ▶ 비교적 최신 커널에서만 사용 가능.

▪ libbpf 섹션 매크로 선언:

- ▶ SEC("fentry/function_name")
- ▶ SEC("fexit/function_name")

Fentry/Fexit은 Kprobe의 '개선판'입니다. 특별한 이유가 없다면, 사용 가능한 커널 버전에서는 Kprobe보다 Fentry/Fexit을 사용하는 것이 항상 더 좋습니다. 특히 함수의 입력과 출력을 모두 연관지어 분석해야 할 때 Fexit의 능력은 매우 유용합니다.

Tracing Type 3: Tracepoints

▪ 개념

- ▶ 커널 소스 코드 내에 미리 정의된, 안정적인 추적 지점.

▪ 특징

- ▶ 커널 개발자들이 중요하다고 생각하는 지점에 미리 '훅'을 심어 둔 것.
- ▶ 안정적인 API
 - 커널 버전이 바뀌어도 이름과 인자가 거의 변하지 않음.
 - kprobe보다 훨씬 안정적.
- ▶ 사용 가능한 트레이스포인트 목록
 - /sys/kernel/tracing/available_events
 - 커널 5.14 기준 2314개의 tracepoint 가 있음

▪ 단점

- ▶ 커널 개발자가 미리 만들어 둔 지점만 추적할 수 있어 Kprobe보다 유연성이 떨어짐.

▪ libbpf 섹션 매크로 선언

- ▶ SEC("tp/tracing subsystem/tracepoint name")
 - 예: SEC("tp/syscalls/sys_enter_openat")

Tracepoint는 커널이 공식적으로 제공하는 '계측용 API'입니다. Kprobe가 비공식적인 뒷길이라면, Tracepoint는 잘 닦인 대로와 같습니다. 안정성이 매우 중요할 때, 예를 들어 특정 시스템 콜의 동작을 추적할 때 가장 먼저 고려해야 할 방법입니다.

Tracing Type 4: Raw Tracepoints and BTF-enabled Tracepoints

▪ Raw Tracepoints (raw_tp):

- ▶ 일반 트레이스포인트보다 더 낮은 레벨의 데이터를 제공.
- ▶ 컨텍스트로 전달되는 인자들이 파싱되지 않은 원시(raw) 형태(주로 u64의 배열)로 전달됨.
- ▶ 성능상 약간의 이점이 있을 수 있지만, 개발자가 직접 인자를 파싱해야 하는 부담이 있음.

▪ BTF-enabled Tracepoints (tp_btf):

- ▶ 현대적인 방식. CO-RE의 핵심.
- ▶ BTF 정보를 활용하여 트레이스포인트의 인자들을 올바른 타입의 구조체 포인터로 직접 전달해 줌.
- ▶ 개발자는 더 이상 수동으로 포인터를 캐스팅하거나 오프셋을 계산할 필요가 없음.
- ▶ libbpf 섹션 매크로 선언
 - SEC("tp_btf/event_name")
 - int handle_event(struct trace_event_raw_... *ctx)

BTF 기반 트레이스포인트는 eBPF 프로그래밍을 훨씬 더 안전하고 쉽게 만들어줍니다. 커널이 BTF를 통해 컨텍스트 구조체의 타입을 정확히 알려주므로, 개발자는 타입에 맞는 구조체 포인터를 인자로 받아 필드에 바로 접근하면 됩니다. 이는 C의 타입 시스템을 eBPF 프로그래밍에 그대로 가져오는 효과를 줍니다.

Tracing Type 5: User Space Probes (Uprobes)

▪ 개념

- ▶ uprobe (User Probe)
 - 사용자 공간 애플리케이션의 함수 시작 지점에 프로브 설치.
- ▶ uretprobe (User Return Probe)
 - 사용자 공간 함수의 반환 지점에 프로브 설치

▪ 용도

- ▶ 애플리케이션 내부 동작 분석.
 - 암호화 라이브러리(OpenSSL 등)의 SSL_read/SSL_write 함수를 후킹하여 암호화되지 않은 평문 데이터 엿보기.
 - 데이터베이스 드라이버 함수를 후킹하여 SQL 쿼리 추적

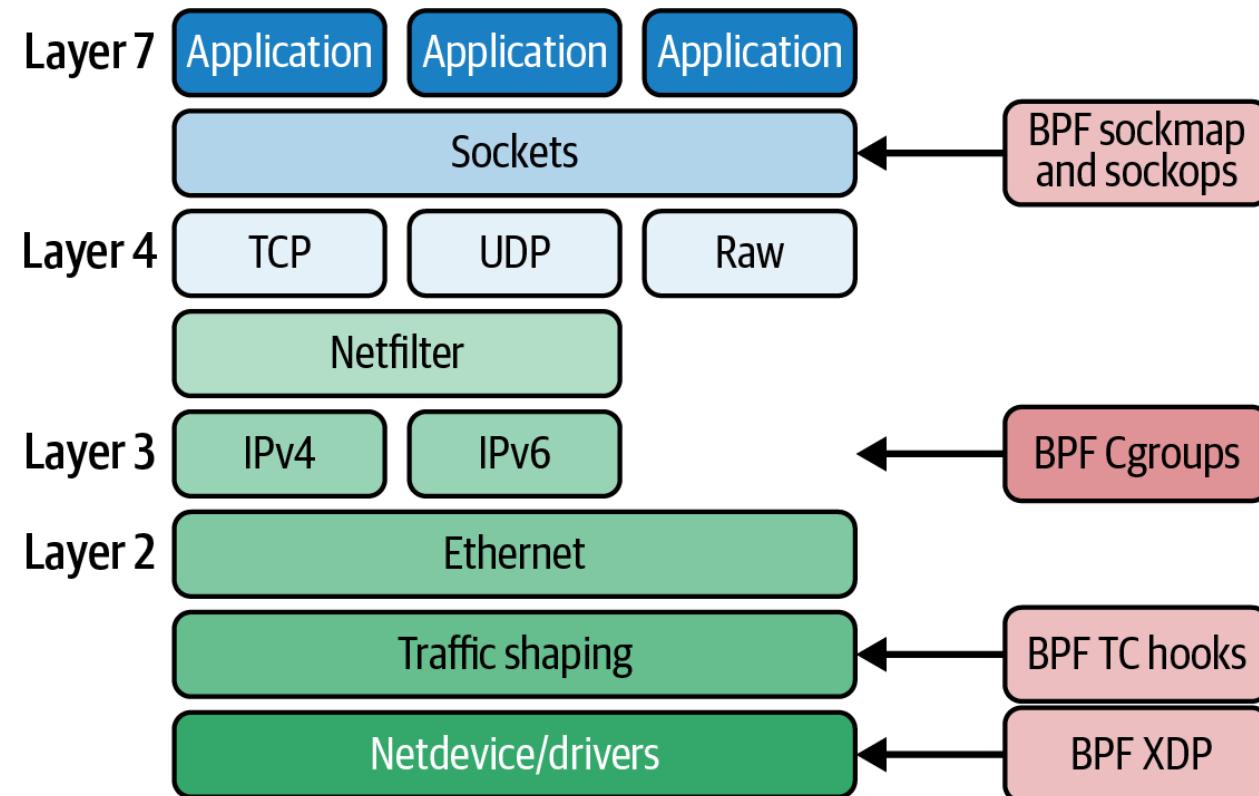
▪ libbpf 섹션 매크로 선언

- ▶ SEC("uprobe/path/to/binary:function_name")

Uprobe는 eBPF의 관찰 능력을 커널에서 사용자 공간까지 확장합니다. 이를 통해 우리는 특정 애플리케이션이 내부적으로 어떻게 동작하는지, 어떤 라이브러리 함수를 호출하는지 등을 소스 코드 없이도 분석할 수 있습니다.

Networking

- BPF 프로그램 유형은 네트워크 스택의 다양한 지점에 연결 가능



- 시스템의 동작에 개입하고 변경

- eBPF 프로그램의 반환 코드를 사용하여 커널에 네트워크 패킷에 대해 수행할 작업을 알려줍니다.
 - 여기에는 패킷을 평소처럼 처리하거나 삭제하거나 다른 대상으로 리디렉션하는 작업이 포함될 수 있습니다.
- eBPF 프로그램이 네트워크 패킷, 소켓 구성 매개변수 등을 수정하도록 허용

Networking Type 1: XDP (eXpress Data Path)

▪ 연결 지점

- ▶ 네트워크 드라이버 바로 다음, 네트워크 스택의 가장 이른 단계.

▪ 처리 대상

- ▶ Ingress(수신) 패킷만 처리.

▪ 컨텍스트

- ▶ struct xdp_md * (패킷 데이터의 포인터 등 최소한의 메타데이터).

▪ 특징

- ▶ 최고 성능: sk_buff 구조체가 생성되기 전이라 오버헤드가 극히 적음.

- ▶ 하드웨어 오프로드 가능

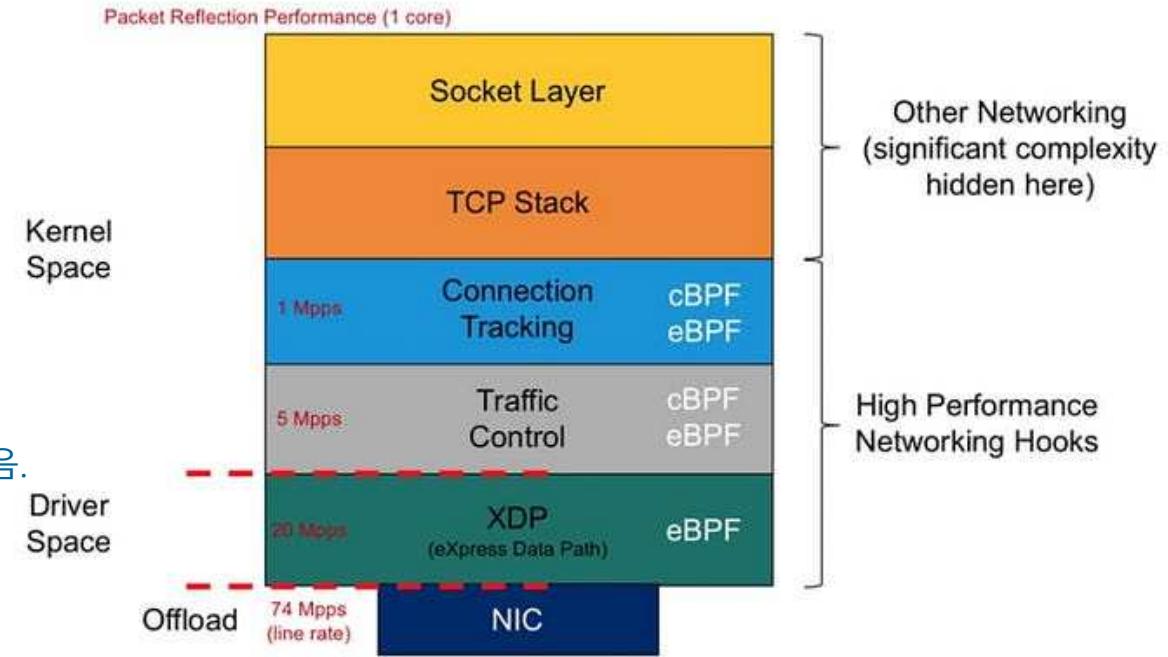
- 지원하는 NIC에서는 XDP 프로그램을 NIC 하드웨어에서 직접 실행 가능.

▪ 주요 용도

- ▶ DDoS 방어, L4 로드 밸런싱, 고성능 방화벽.

▪ libbpf 섹션 매크로 선언

- ▶ SEC("xdp")

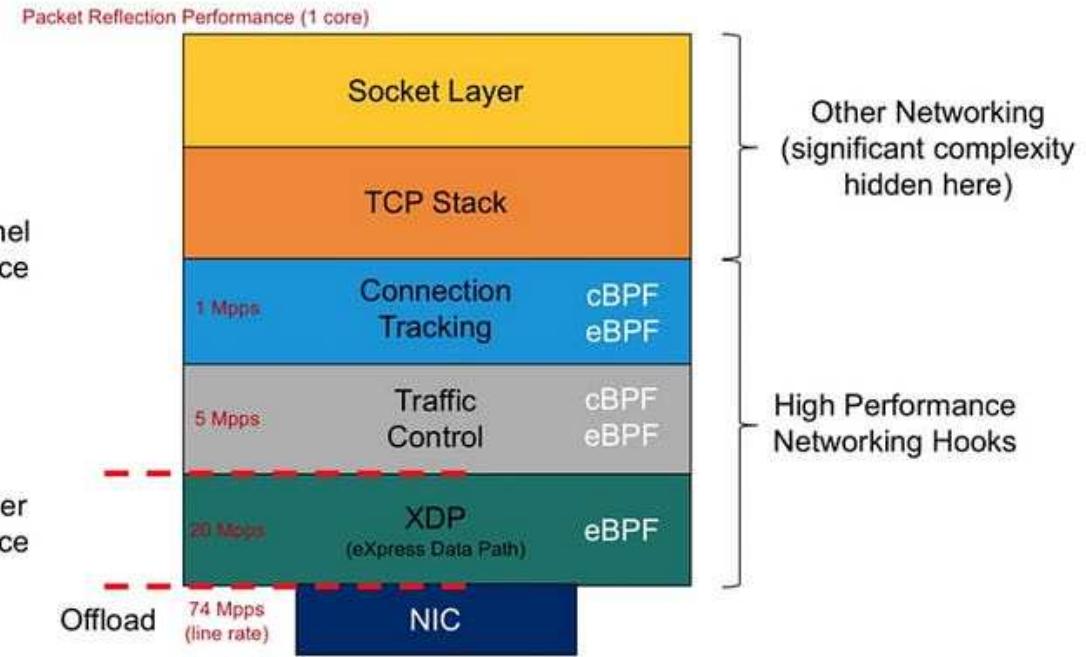


<https://medium.com/@kcl17/unlocking-network-performance-with-xdp-and-ebpf-67c712128025>

XDP는 '속도'가 생명인 작업에 사용됩니다. 커널의 복잡한 네트워크 스택을 거치기 전에 패킷을 처리하므로, 초당 수백만 개의 패킷을 처리해야 하는 시나리오에 최적화되어 있습니다.

Networking Type 2: Traffic Control (TC)

- 연결 지점
 - ▶ 커널의 트래픽 제어(TC) 계층 -> IP 계층 이후
- 처리 대상
 - ▶ Ingress(수신) 와 Egress(송신) 패킷 모두 처리 가능
- 컨텍스트
 - ▶ `struct __sk_buff *`
 - 네트워크 스택이 사용하는 핵심 데이터 구조인 `sk_buff`에 대한 포인터
- 특징
 - ▶ XDP보다 더 풍부한 정보(`sk_buff`)에 접근 가능
 - ▶ 하나의 흑에 여러 eBPF 프로그램을 체인으로 연결 가능
- 주요 용도
 - ▶ CNI, Service-Mesh 등 복잡한 네트워킹 로직 구현에 널리 사용
- libbpf 섹션 매크로 선언
 - ▶ SEC("tc")



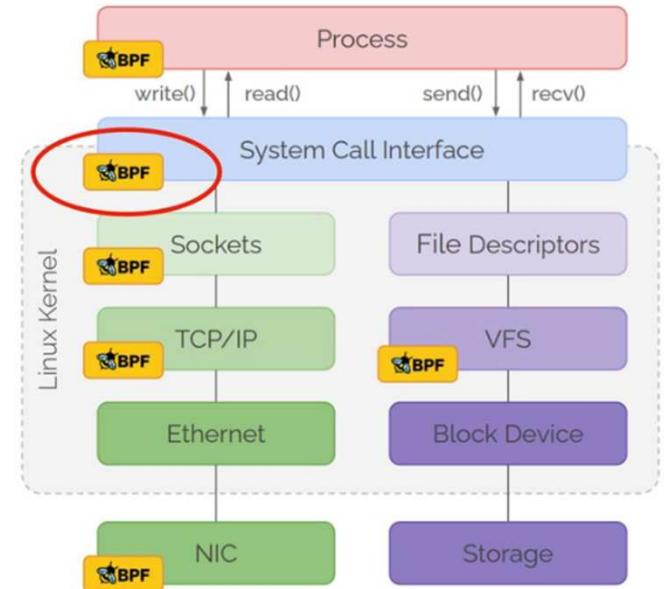
<https://medium.com/@kcl17/unlocking-network-performance-with-xdp-and-ebpf-67c712128025>

TC는 XDP보다 조금 더 상위 계층에서 동작합니다. 속도는 XDP보다 약간 느리지만, `sk_buff`라는 풍부한 컨텍스트에 접근할 수 있어 훨씬 더 정교하고 복잡한 패킷 조작 및 정책 적용이 가능합니다. Cilium과 같은 CNI가 바로 이 TC 흑을 핵심적으로 사용합니다.

Security Type 1: Cgroups

- 연결 지점
 - ▶ 특정 컨트롤 그룹(cgroup)의 경로
- 처리 대상
 - ▶ 해당 cgroup에 속한 프로세스들이 생성하는 이벤트
- 특징
 - ▶ 컨테이너 단위 정책 적용
 - 컨테이너(또는 Pod)는 cgroup으로 격리되므로, cgroup 혹은 특정 컨테이너에만 정책을 적용하는 데 이상
 - ▶ 네트워킹 외 다양한 후 제공
- 주요 용도
 - ▶ BPF_PROG_TYPE_CGROUP_SKB
 - 컨테이너의 네트워크 트래픽에 대한 정책 적용.
 - ▶ BPF_PROG_TYPE_CGROUP SOCK_ADDR
 - 컨테이너의 connect()/bind() 시스템 콜을 후킹하여 아웃바운드 연결 제어.
 - ▶ BPF_PROG_TYPE_CGROUP_DEVICE
 - 컨테이너의 장치 접근 제어.

Where is cgroup eBPF?



<https://speakerdeck.com/rueian/cilium-and-cgroup-ebpf?slide=10>

Cgroup 혹은 eBPF 정책의 범위를 '시스템 전체'에서 '특정 프로세스 그룹' 또는 '특정 컨테이너'로 좁혀주는 강력한 메커니즘입니다.
이를 통해 컨테이너별로 세분화된 네트워크 정책이나 장치 접근 정책을 효율적으로 구현할 수 있습니다.

Security Type 2: LSM (Linux Security Module)

▪ 연결 지점

- ▶ 커널의 LSM 흑. (SELinux, AppArmor가 사용하는 바로 그 흑)

▪ 특징

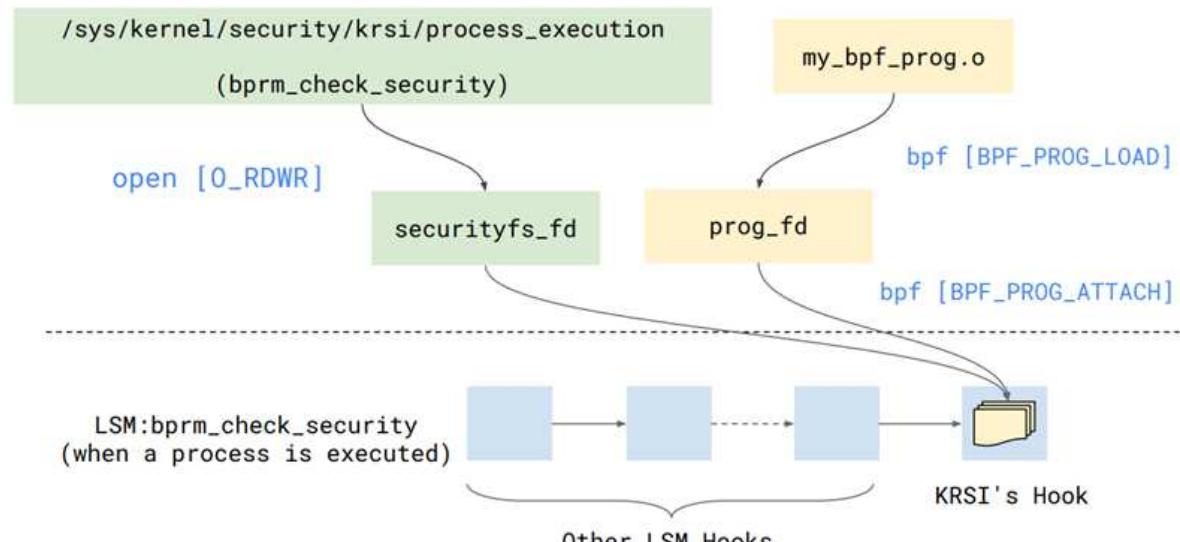
- ▶ TOCTOU(Time-of-check to time-of-use) 문제 해결
 - 시스템 콜의 인자가 커널 메모리로 복사된 이후, 하지만 실제 작업이 수행되기 직전에 실행됨.
 - 따라서 eBPF 프로그램이 검사하는 데이터와 커널이 실제로 사용하는 데이터가 동일함이 보장됨.
- ▶ 예방적 통제(Preventive Control)
 - 0이 아닌 값을 반환하여 작업을 차단할 수 있음

▪ 주요 용도

- ▶ 파일 접근 제어,
- ▶ 프로세스 권한 상승 방지 등 정교한 시스템 보안 정책 구현

▪ libbpf 섹션 매크로 선언

- ▶ SEC("lsm/hook_name")
 - 예: SEC("lsm/path_chmod")



<https://www.openeuler.org/en/blog/MrRiu/2021-01-04-openEuler-eBPF-introduce.html>

LSM 흑은 보안을 위한 eBPF의 '성배'로 불립니다. 기존의 시스템 콜 추적이 가진 TOCTOU(Time-of-check to time-of-use) 경쟁 조건 취약점을 해결하기 때문입니다. LSM 흑을 사용하면 공격자가 검사를 우회할 틈 없이, 커널이 처리할 데이터를 직접 보고 결정을 내릴 수 있어 훨씬 더 신뢰도 높은 보안 정책을 만들 수 있습니다.

요약

- eBPF의 강력함은 다양한 프로그램 유형과 전략적인 연결 지점에서 나옵니다.
- 프로그램 유형은 프로그램의 목적, 컨텍스트, 사용 가능한 헬퍼 함수, 반환 값의 의미를 결정합니다.
- 추적용 흑 (kprobe, tracepoint, fentry 등)
 - ▶ 시스템을 '읽고' 관찰하는 데 중점.
- 네트워킹/보안용 흑 (XDP, TC, cgroup, LSM 등)
 - ▶ 시스템의 동작에 '개입하고' 제어하는 데 중점.
- libbpf의 SEC() 매크로는 프로그램 유형과 연결 지점을 선언하는 표준적인 방법입니다.
- 작업에 적합한 올바른 흑을 선택하는 것이 효과적인 eBPF 프로그래밍의 핵심입니다.
- 유스케이스
 - ▶ 최고의 성능이 필요하면 XDP
 - ▶ 정교한 패킷 조작이 필요하면 TC
 - ▶ 안정적인 API가 필요하면 Tracepoint
 - ▶ 가장 안전한 보안 정책이 필요하면 LSM을 사용

8. eBPF for Networking

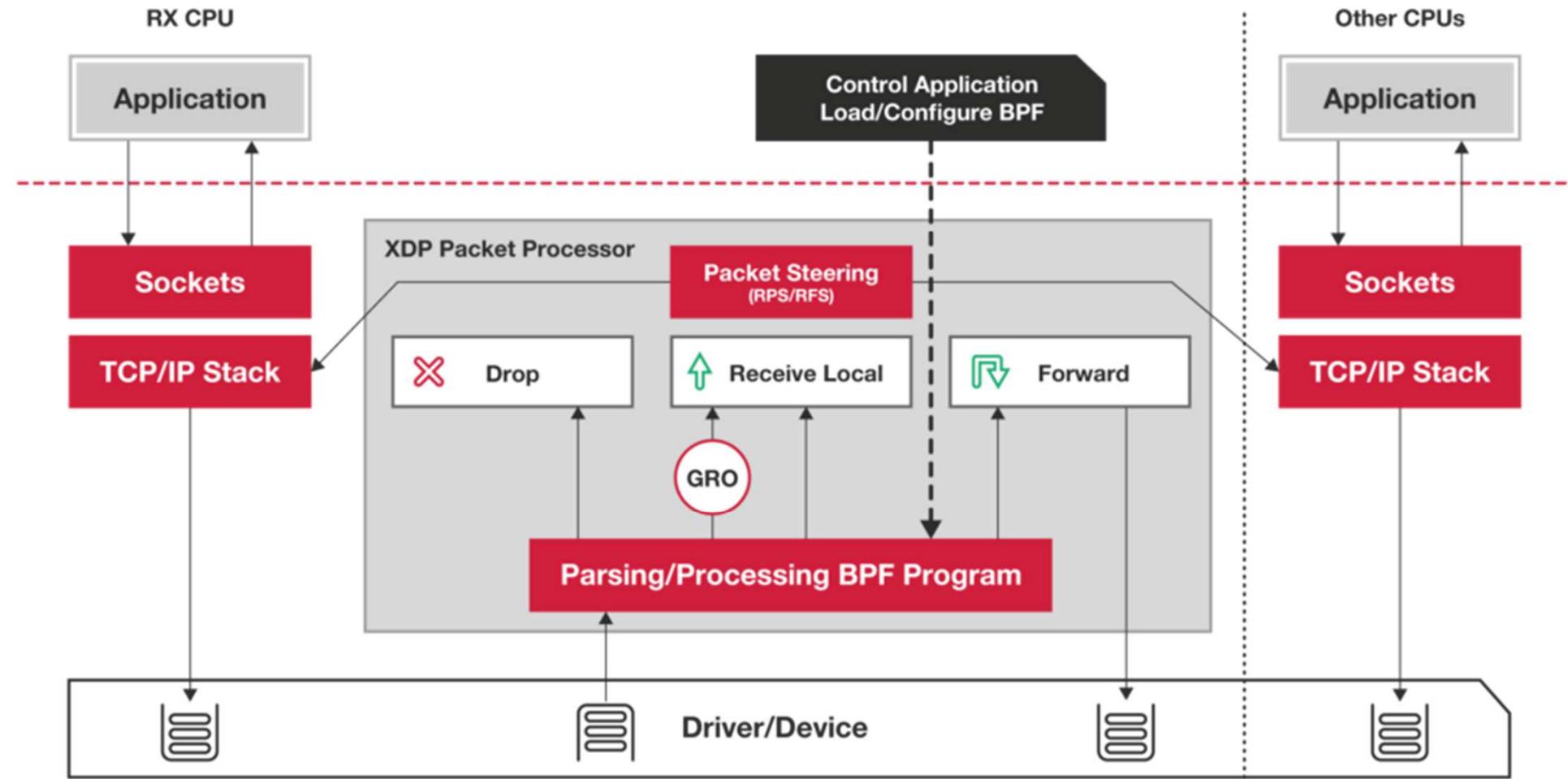
eBPF를 이용한 고성능, Programmable 네트워킹 구현

▪ 학습 목표

- ▶ eBPF가 전통적인 네트워킹 방식(iptables 등)의 한계를 어떻게 극복하는지 이해합니다.
- ▶ XDP와 TC 헥을 사용하여 패킷을 처리(파싱, 수정, 드롭, 리다이렉트)하는 구체적인 방법을 학습합니다.
- ▶ eBPF를 이용한 로드 밸런싱, 방화벽, 암호화된 트래픽 가시성 확보 등 실제 네트워킹 솔루션의 구현 원리를 분석합니다.
- ▶ 쿠버네티스 네트워킹(CNI)에서 eBPF가 어떻게 혁신을 가져오고 있는지 살펴봅니다.

XDP (Express Data Path)

- XDP는 Linux 커널에서 제공하는 기능으로, 고성능 패킷 처리를 가능하게 합니다.



Packet Drops (1/3)

▪ 패킷 처리의 기본: Packet Drops

▶ 네트워크 보안의 기본

- 원치 않는 패킷을 폐기(drop)하는 것.

▶ 적용 분야:

- 방화벽:
 - 특정 IP 주소, 포트, 프로토콜 기반으로 패킷 차단
- DDoS 방어
 - 비정상적인 대량의 트래픽을 조기에 차단
 - 예를 들어 특정 소스에서 패킷이 도착하는 속도를 추적하거나 패킷 내용의 특정 특성을 감지하여 공격자 또는 공격자 집단이 인터페이스에 트래픽을 범람시키려 하고 있다는 것을 파악하는 것일 수 있습니다.
- 패킷-오브-데스(Packet-of-death) 취약점 완화:
 - 커널을 크래시 시킬 수 있는 특정 형태의 악성 패킷을 동적으로 차단.
 - 이러한 악성 패킷을 감지하고 삭제하는 eBPF 프로그램을 동적으로 설치하여 머신에서 실행되는 애플리케이션에 영향을 미치지 않고 해당 호스트를 즉시 보호할 수 있습니다.

▶ eBPF의 역할

- XDP나 TC 혹에서 패킷의 내용을 검사하고, 정책에 따라 XDP_DROP 또는 TC_ACT_SHOT을 반환하여 패킷을 폐기.

가장 간단하지만 가장 중요한 네트워크 기능은 '버리는 것'입니다. eBPF는 이 '버리기'를 매우 효율적으로 할 수 있습니다. 특히 XDP를 사용하면 패킷이 커널의 비싼 네트워크 스택에 도달하기 전에, 드라이버 수준에서 즉시 폐기할 수 있어 DDoS 공격과 같은 대규모 공격을 방어하는 데 매우 효과적입니다.

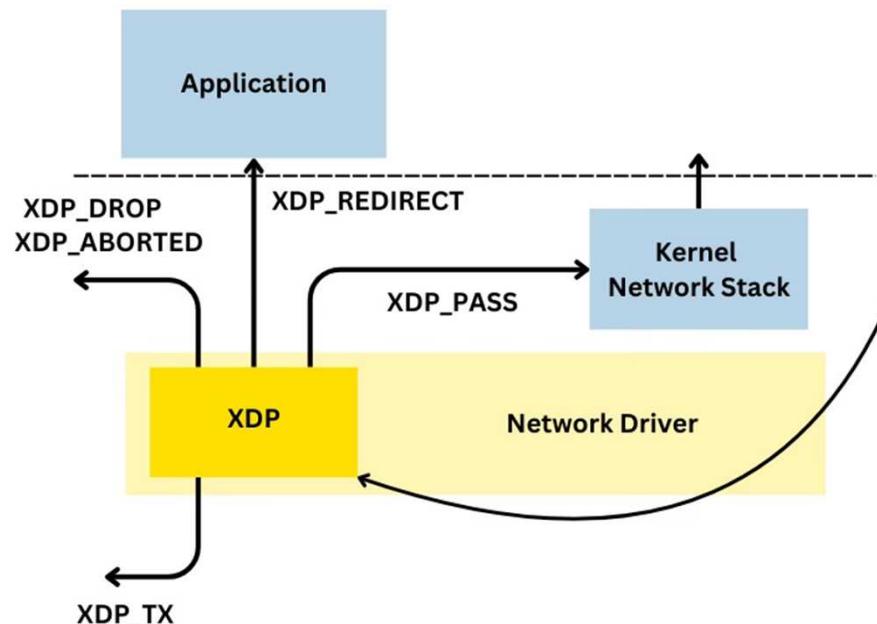
Packet Drops (2/3)

▪ XDP 프로그램은 네트워크 패킷이 도착하면 트리거됩니다.

- ▶ 이 프로그램은 패킷을 검사하고 완료되면 반환 코드를 리턴하고, 커널에게 다음에 무엇을 할지 알려주는 판결(verdict)을 내려야 합니다.

Return Code		Action
XDP_PASS	통과	패킷을 정상적으로 커널 네트워크 스택으로 전달
XDP_DROP	폐기	패킷을 조용히 버림, 성능 저하 없이 즉시 폐기
XDP_TX	전송	패킷을 수신된 동일한 네트워크 인터페이스로 다시 내보냄
XDP_REDIRECT	재전송	패킷을 다른 네트워크 인터페이스나 다른 CPU 소켓으로 전달
XDP_ABORTED	중단	XDP_DROP 과 같이 패킷을 버리지만, 비정상적인 상황임을 나타냄

이 반환 코드들이 바로 XDP 프로그램의 '제어 능력'입니다. 단순히 패킷을 통과시키거나 버리는 것을 넘어, 다른 곳으로 보내거나(redirect), 들어온 곳으로 되돌려 보내는(tx) 등 다양한 패킷 포워딩 로직을 구현할 수 있습니다.



```

#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

SEC("xdp")
int xdp_prog(struct xdp_md *ctx) {
    // ...
    // What happens to the packet?
    return 0;
}

char _license[] SEC("license") = "GPL";
  
```

<https://ebpfchirp.substack.com/p/ebpf-challenge-1-xdp-return-codes>

Packet Drops (3/3)

- 예시) 패킷을 삭제할지 여부를 결정하는 XDP 프로그램

```
SEC("xdp")
int hello(struct xdp_md *ctx) {
    bool drop;

    drop = <examine packet and decide whether to drop it>;

    if (drop)
        return XDP_DROP;
    else
        return XDP_PASS;
}
```

```
struct xdp_md {
    __u32 data;           ← 패킷 데이터의 시작 주소
    __u32 data_end;      ← 패킷 데이터의 끝 주소
    __u32 data_meta;
    /* Below access go through struct xdp_rxq_info */
    __u32 ingress_ifindex; /* rxq->dev->ifindex */
    __u32 rx_queue_index; /* rxq->queue_index */
    __u32 egress_ifindex; /* txq->dev->ifindex */
};
```

XDP Packet Parsing (1/4)

▪ 샘플 eBPF 프로그램

- ▶ ping(ICMP) 패킷을 감지할 때마다 커널 로그 출력

chapter8/hello.pbf.c

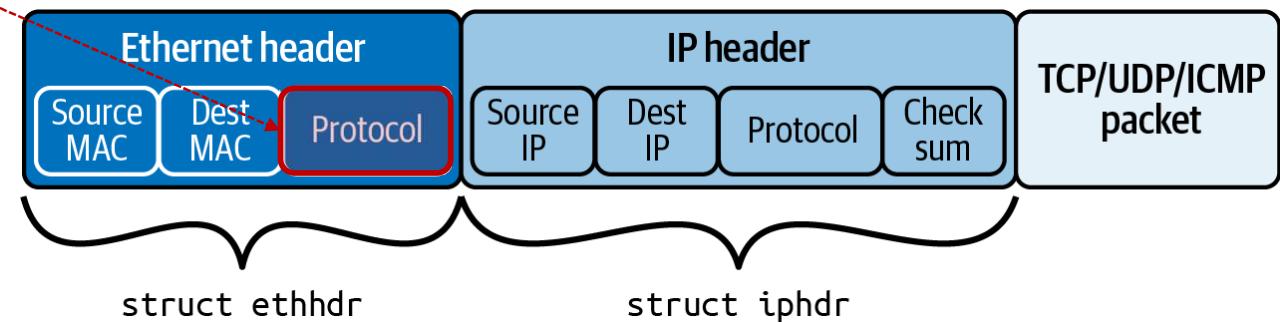
```
#include "vmlinux.h"
#include <bpf/bpf_endian.h>
#include <bpf/bpf_helpers.h>
#include "packet.h"

SEC("xdp")

int ping(struct xdp_md *ctx) {
    long protocol = lookup_protocol(ctx);
    if (protocol == 1) // ICMP
    {
        bpf_printk("Hello ping");
        // return XDP_DROP;
    }
    return XDP_PASS;
}

char LICENSE[] SEC("license") = "Dual BSD/GPL";
```

- BPF 프로그램의 섹션을 지정하는 매크로
- XDP 프로그램으로 컴파일되어야 함을 BPF 컴파일러(Clang/LLVM)에 지시
- 현재 ctx(XDP 컨텍스트)가 가리키는 패킷에서 프로토콜 정보를 추출
- ICMP 패킷이면, 커널 로그 출력



<IP 네트워크 패킷의 레이아웃>

XDP Packet Parsing (2/4)

- ✓ loopback 인터페이스의 정보를 조회하고, xdp BPF 프로그램이 연결되지 않는 것을 확인

```
[root@server1 ~]# ip link show lo
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

- ✓ chapter8 디렉토리로 이동후 make 수행

```
[root@server1 ~]# cd ~/learning-ebpf/chapter8
[root@server1 chapter8]# make
clang \
  -target bpf \
  -D __BPF_TRACING__ \
  -I/usr/src/kernels/5.14.0-570.25.1.el9_6.x86_64/tools/lib \
  -I/usr/src/kernels/5.14.0-570.25.1.el9_6.x86_64/tools/bpf/resolve_btfids \
  -Wall \
  -O2 -o hello.bpf.o -c hello.bpf.c
bpftool net detach xdp dev lo
rm -f /sys/fs/bpf/hello
bpftool prog load hello.bpf.o /sys/fs/bpf/hello
bpftool net attach xdp pinned /sys/fs/bpf/hello dev lo
```

- ✓ loopback 인터페이스의 정보를 조회하고, xdp BPF 프로그램이 연결되지 않는 것을 확인

```
[root@server1 chapter8]# ip link show lo
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 xpgeneric qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
prog/xdp id 94 name ping tag 3c91b2e4d4590b11 jited
```

Makefile

```
TARGET = hello
ARCH = $(shell uname -m | sed 's/x86_64/x86/' | sed 's/aarch64/arm64/')
BPF_OBJ = ${TARGET}:=.bpf.o

all: $(TARGET) $(BPF_OBJ)
.PHONY: all
.PHONY: $(TARGET)

$(TARGET): $(BPF_OBJ)
    bpftool net detach xdp dev lo
    rm -f /sys/fs/bpf/$(TARGET)
    bpftool prog load $(BPF_OBJ) /sys/fs/bpf/$(TARGET)
    bpftool net attach xdp pinned /sys/fs/bpf/$(TARGET) dev lo
    ↑ xdp eBPF 프로그램을 LOAD 하고 loopback 인터페이스에 연결
$(BPF_OBJ): %.o: %.c vmlinux.h
    clang \
        -target bpf \
        -D __BPF_TRACING__ \
        -I/usr/src/kernels/$(shell uname -r)/tools/lib \
        -I/usr/src/kernels/$(shell uname -r)/tools/bpf/resolve_btfids/libbpf \
        -Wall \
        -O2 -o $@ -c $<

vmlinux.h:
    bpftool btf dump file /sys/kernel/btf/vmlinux format c > vmlinux.h

clean:
    - bpftool net detach xdp dev lo
    - rm -f /sys/fs/bpf/$(TARGET)
    - rm $(BPF_OBJ)
    - tc filter delete dev parent ffff:
```

XDP Packet Parsing (3/4)

- ✓ "ping 127.0.0.1" 명령을 실행

```
[root@server1 chapter8]# ping 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.183 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.273 ms
```

- ✓ (새로운 터미널을 열고) cat /sys/kernel/tracing/trace_pipe 를 실행하여 trace pipe에서 생성된 출력을 관찰

- 1초마다 추적 라인이 두 개 생성되는 것을 확인
- 루프백 인터페이스가 ping 요청과 ping 응답을 모두 받기 때문에 초당 두 줄의 추적이 발생합니다.

```
[root@server1 ~]# cat /sys/kernel/tracing/trace_pipe
<...>-4985 [002] ...s2.1 88267.965365: bpf_trace_printk: Hello ping
<...>-4985 [002] ...s2.1 88267.965453: bpf_trace_printk: Hello ping
<...>-4985 [002] ...s2.1 88268.982810: bpf_trace_printk: Hello ping
ping-4985 [002] ...s2.1 88268.982917: bpf_trace_printk: Hello ping
```

XDP Packet Parsing (4/4)

- ✓ hello.bpf.c 소스 코드 수정 -> ICMP 패킷을 DROP

```
[root@server1 chapter8]# vi hello.bpf.c
int ping(struct xdp_md *ctx) {
    long protocol = lookup_protocol(ctx);
    if (protocol == 1) // ICMP
    {
        bpf_printk("Hello ping");
        return XDP_DROP;
    }
    return XDP_PASS;
}
```

- ✓ "make clean" 작업후 "make" 다시 수행

```
[root@server1 chapter8]# make clean; make
```

- ✓ "ping 127.0.0.1" 명령을 실행

```
[root@server1 chapter8]# ping 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.183 ms
← ICMP 패킷이 DROP 되어서 2분간 응답이 없음
```

- ✓ (새로운 터미널을 열고) cat /sys/kernel/tracing/trace_pipe 를 실행하여 trace pipe에서 생성된 출력을 관찰

- 1초마다 추적 라인이 한 개 생성되는 것을 확인
- ping 요청에 1개의 라인을 출력하고, 패킷을 DROP 하였기 때문에 2분간 응답하지 않음

```
[root@server1 ~]# cat /sys/kernel/tracing/trace_pipe
<...>-4985      [002] ...s2.1 88267.965365: bpf_trace_printk: Hello ping
```

- ✓ 다음 실습을 위해 "make clean" 수행

```
[root@server1 chapter8]# make clean
```

Load Balancing and Forwarding (1/2)

▪ 개념

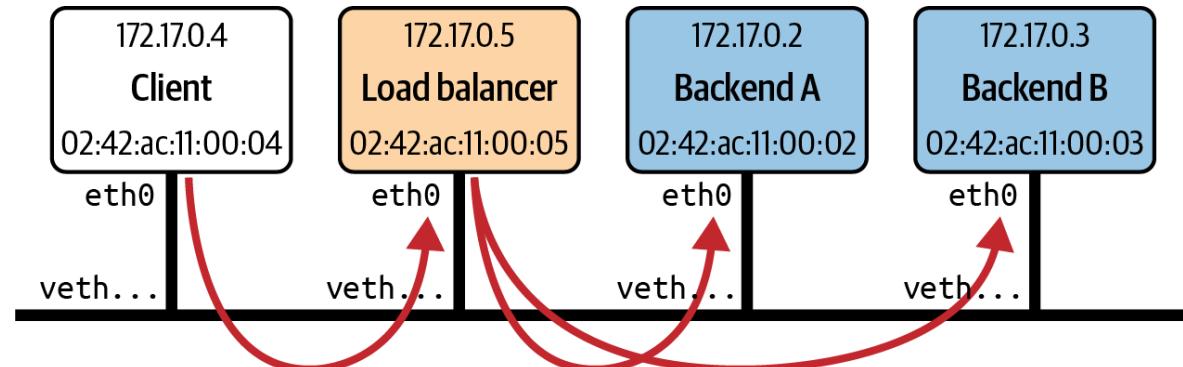
- ▶ eBPF는 패킷 내용을 수정할 수 있고, 이를 통해 정교한 패킷 포워딩과 로드 밸런싱이 가능

▪ 기본 로직

- ① 수신된 패킷의 목적지 IP/MAC 주소를 백엔드 서버의 주소로 변경.
- ② 패킷의 출발지 IP/MAC 주소를 로드 밸런서 자신의 주소로 변경 (Source NAT).
- ③ IP 헤더 체크섬을 다시 계산.
- ④ XDP_TX 또는 XDP_REDIRECT를 반환하여 수정된 패킷을 포워딩.

▪ 참고 동영상

- ▶ A Load Balancer from scratch – Liz Rice, Isovalent
 - https://youtu.be/L3_AOFSNKK8



eBPF의 진정한 힘은 '읽기'를 넘어 '쓰기'가 가능하다는 점에서 나옵니다.
 패킷의 헤더를 직접 수정함으로써, eBPF 프로그램은 커널의 라우팅
 테이블이나 conntrack 테이블을 사용하지 않고도 자신만의 라우팅 및 로드
 밸런싱 규칙을 구현할 수 있습니다.

Load Balancing and Forwarding (2/2)

```

SEC("xdp_lb")
int xdp_load_balancer(struct xdp_md *ctx) {
    void *data = (void *)(long)ctx->data;
    void *data_end = (void *)(long)ctx->data_end;

    struct ethhdr *eth = data;
    if (data + sizeof(struct ethhdr) > data_end) return XDP_ABORTED;

    if (bpf_ntohs(eth->h_proto) != ETH_P_IP) return XDP_PASS;

    struct iphdr *iph = data + sizeof(struct ethhdr);
    if (data + sizeof(struct ethhdr) + sizeof(struct iphdr) > data_end)
        return XDP_ABORTED;

    if (iph->protocol != IPPROTO_TCP) return XDP_PASS;

    if (iph->saddr == IP_ADDRESS(CLIENT)) {
        char be = BACKEND_A;
        if (bpf_get_prandom_u32() % 2)
            be = BACKEND_B;
        iph->daddr = IP_ADDRESS(be);
        eth->h_dest[5] = be;

    } else {
        iph->daddr = IP_ADDRESS(CLIENT);
        eth->h_dest[5] = CLIENT;
    }

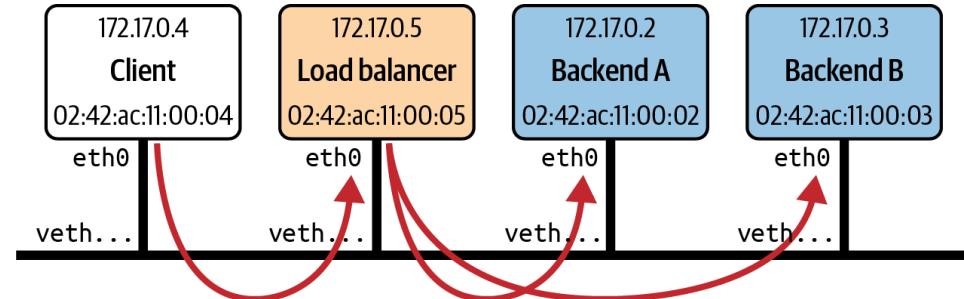
    iph->saddr = IP_ADDRESS(LB);
    eth->h_source[5] = LB;

    iph->check = iph_csum(iph);

    return XDP_TX;
}

```

이 코드는 간단한 L4 로드 밸런서의 핵심 로직입니다. 패킷의 출발지 IP를 보고 클라이언트에서 온 요청인지, 백엔드에서 온 응답인지를 판단합니다. 그 후, 패킷의 목적지와 출발지 주소를 적절히 변환(NAT)하고, 변경된 헤더에 맞게 체크섬을 다시 계산한 뒤, XDP_TX로 패킷을 다시 내보냅니다.



패킷에서 이더넷 헤더를 찾은 다음 IP 헤더를 찾습니다.

TCP 패킷만 처리하고, 그 외에 모든 패킷은 아무 일도 없었던 것처럼 스택 위로 전달

여기서 소스 IP 주소가 확인됩니다. 이 패킷이 클라이언트에서 오지 않았다면, 클라이언트로 가는 응답이라고 가정합니다.
이 코드는 백엔드 A와 B 간의 의사난수 선택을 생성합니다.
대상 IP 및 MAC 주소는 선택된 백엔드와 일치하도록 업데이트됩니다.

이것이 백엔드로부터의 응답인 경우(클라이언트로부터 오지 않았다면 여기서는 이것이 가정임)
대상 IP 및 MAC 주소가 클라이언트와 일치하도록 업데이트됩니다.

이 패킷이 어디로 가든지, 소스 주소를 업데이트하여 패킷이 로드 밸런서에서 시작된 것처럼 보이도록 해야 합니다.

IP 헤더에는 내용에 대해 계산된 체크섬이 포함되어 있으며, 소스 및 대상 IP 주소가 모두 업데이트되었으므로 체크섬도 다시 계산되어 이 패킷으로 교체되어야 합니다.

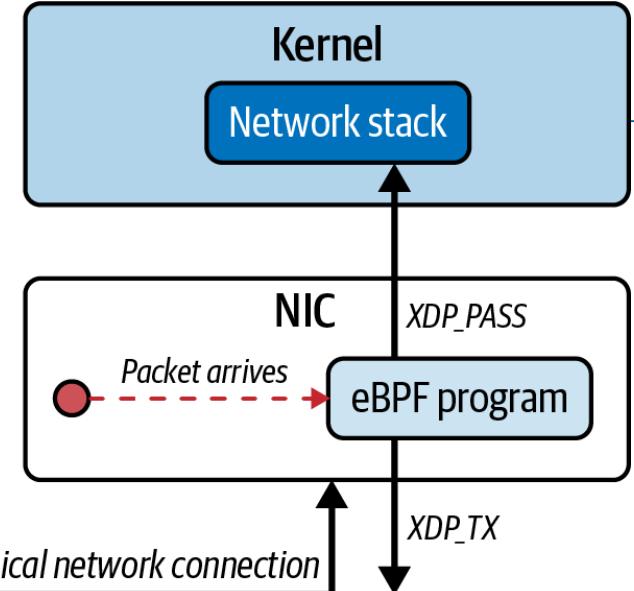
XDP Offloading

▪ XDP Offloading 개념

- ▶ eBPF XDP(eXpress Data Path) 프로그램을 CPU 대신 네트워크 인터페이스 카드(NIC) 자체에서 직접 실행하는 개념
- ▶ XDP 오프로드를 지원하는 네트워크 인터페이스 카드는 호스트 CPU에서 필요한 작업 없이 패킷을 처리, 삭제 및 재전송할 수 있습니다.

▪ XDP 실행 모드

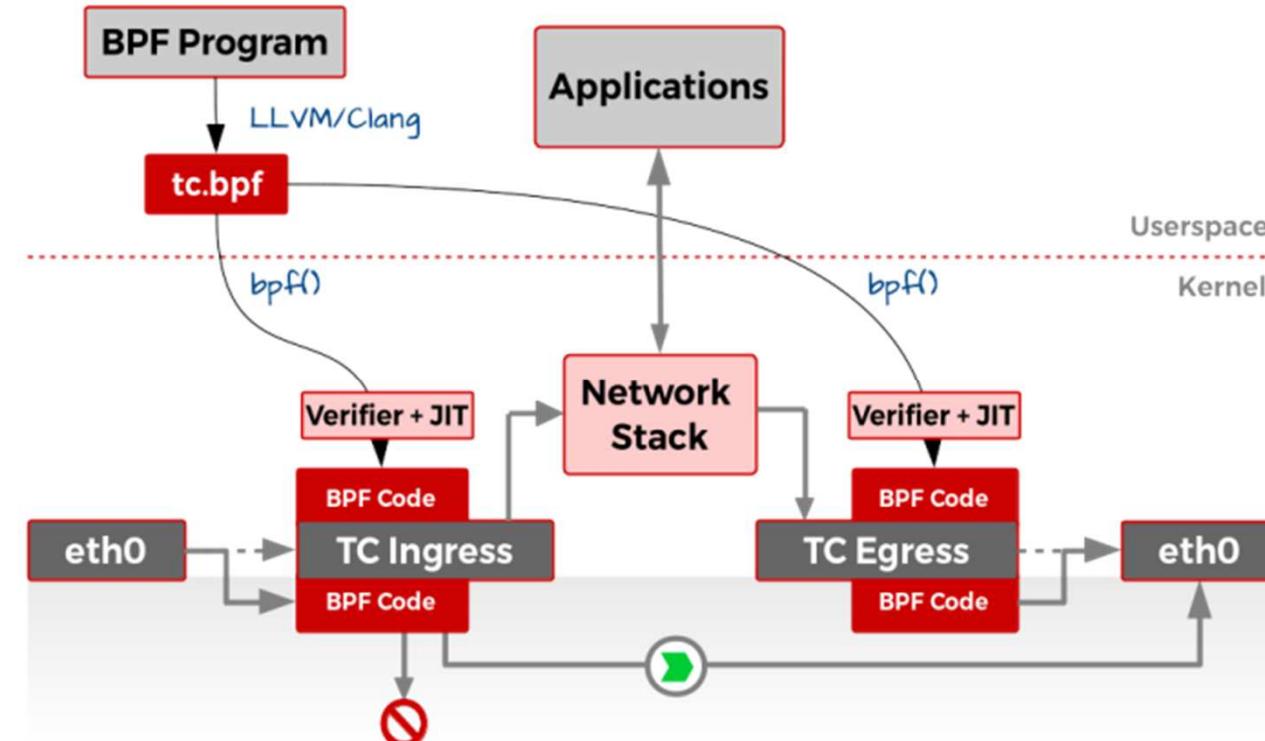
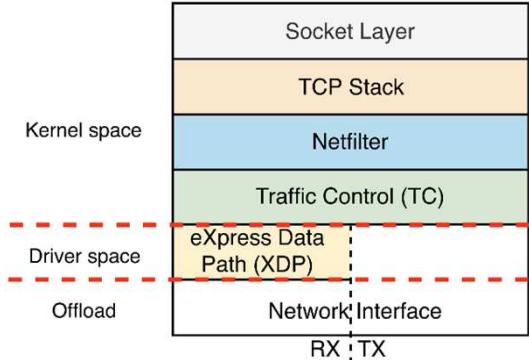
종류	개념	특징
Generic XDP (소프트웨어 에뮬레이션)	NIC 드라이버가 XDP를 직접 지원하지 않는 경우, 커널의 일반 네트워크 스택 상위 레이어에서 XDP 프로그램을 소프트웨어적으로 에뮬레이트하여 실행합니다.	가장 광범위한 하드웨어에서 작동하며, 드라이버 지원이 필요 없습니다. 하지만 커널 네트워크 스택에 진입한 후 실행되므로, XDP의 가장 큰 장점인 성능 이점은 제한적입니다. 주로 XDP 프로그램 개발 및 테스트용으로 사용됩니다.
Native XDP (드라이버 레벨)	NIC 드라이버가 XDP를 직접 지원하는 경우, 패킷이 NIC에서 수신된 직후, 드라이버의 RX 링 버퍼 단계에서 바로 eBPF XDP 프로그램이 실행됩니다.	커널 네트워크 스택의 대부분을 우회하므로, Generic XDP보다 훨씬 뛰어난 성능을 제공합니다. 대부분의 최신 고성능 NIC는 Native XDP를 지원합니다. 패킷이 메모리에 복사된 후 CPU에서 처리됩니다.
Offloaded XDP (하드웨어 오프로드)	XDP 프로그램의 바이트코드 자체가 NIC의 하드웨어로 직접 오프로드(Offload)되어 실행됩니다. 즉, 패킷이 NIC에 도착했을 때 CPU의 개입 없이 NIC 자체의 펌웨어나 전용 프로세서에서 eBPF 프로그램이 처리됩니다.	궁극의 성능 - 패킷이 NIC 와이어 속도로 처리되므로, 가능한 가장 낮은 지연 시간과 가장 높은 처리량을 제공합니다. CPU 자원을 전혀 사용하지 않습니다. CPU 부하 감소 - 패킷 처리에 CPU 자원이 소모되지 않아, CPU는 다른 애플리케이션 작업을 수행할 수 있습니다. 제한적인 하드웨어 지원 - 이 기능을 지원하는 NIC(주로 스마트 NIC 또는 DPU)가 아직 제한적입니다. Netronome, Intel, NVIDIA(Mellanox) 등 일부 제조사의 특정 NIC 모델에서 지원됩니다. 기능 제한 - NIC 하드웨어의 제약으로 인해 오프로드할 수 있는 eBPF 프로그램의 복잡성이나 사용할 수 있는 헬퍼 함수에 제한이 있을 수 있습니다. - 모든 eBPF 프로그램이 오프로드될 수 있는 것은 아닙니다



TC Traffic Control (1/5)

▪ Linux 커널의 네트워크 트래픽을 조절하고 관리

- ▶ TC를 사용하면 네트워크의 대역폭을 제한하거나, 트래픽의 우선순위를 정하거나, 패킷을 필터링할 수 있습니다.
- ▶ 주로 다음과 같은 기능을 제공합니다:
 - 대역폭 관리: 네트워크 대역폭을 조절하여 특정 애플리케이션이나 서비스가 사용하는 대역폭을 제한하거나 보장합니다.
 - 우선순위 설정: 특정 유형의 트래픽에 더 높은 우선순위를 부여하여, 중요 데이터가 지연 없이 전송될 수 있도록 합니다.
 - 패킷 필터링: 트래픽을 필터링하여 특정 패킷을 차단하거나 허용합니다.

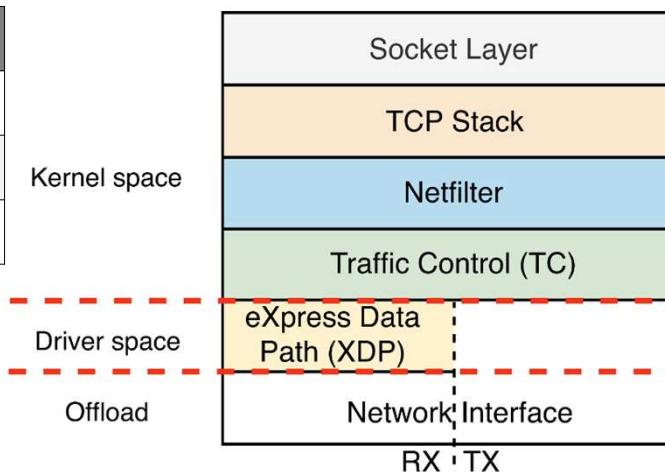


TC Traffic Control (2/5)

■ 왜 XDP로 할 수 있는 일을 TC를 이용할까?

- ▶ XDP는 Ingress 전용이라서 Egress 패킷을 제어할 수 없음
- ▶ XDP가 패킷이 도착하자마자 트리거되기 때문에 그 시점에는 패킷과 관련된 `sk_buff` 커널 데이터 구조가 없음
 - XDP가 속도에 초점을 맞춘 '특수부대'라면, TC는 풍부한 정보를 바탕으로 작전하는 '정보부대'와 같습니다. TC 혹은 `sk_buff`라는 강력한 무기를 사용할 수 있기 때문에, 컨테이너 간 통신을 제어하는 등 훨씬 복잡하고 상태 기반의 네트워킹 로직을 구현하는 데 적합합니다.
- ▶ `_sk_buff` 구조체
 - `sk_buff`는 커널 네트워크 스택의 핵심 자료구조로, 패킷 데이터뿐만 아니라 소켓 정보, 라우팅 정보 등 수많은 메타데이터를 포함
 - TC eBPF 프로그램은 이 풍부한 컨텍스트를 활용하여 더 정교한 결정을 내릴 수 있음

구분	XDP	TC
위치	드라이버 레벨	IP 스택 레벨
패킷	Ingress 전용	Ingress/Egress 양방향
컨텍스트	<code>xdp_md</code>	<code>_sk_buff</code>

`bpf.h`

```

/* user accessible mirror of in-kernel sk_buff.
 * new fields can only be added to the end of this structure */
struct __sk_buff {
    __u32 len;
    __u32 pkt_type;
    __u32 mark;
    __u32 queue_mapping;
    __u32 protocol;
    __u32 vlan_present;
    __u32 vlan_tci;
    __u32 vlan_proto;
    __u32 priority;
    __u32 ingress_ifindex;
    __u32 ifindex;
    __u32 tc_index;
    __u32 cb[5];
    __u32 hash;
    __u32 tc_classid;
    __u32 data;
    __u32 data_end;
    __u32 napi_id;
}

/* Accessed by BPF_PROG_TYPE_sk_skb types from here to ... */
__u32 family;
__u32 remote_ip4; /* Stored in network byte order */
__u32 local_ip4; /* Stored in network byte order */
__u32 remote_ip6[4]; /* Stored in network byte order */
__u32 local_ip6[4]; /* Stored in network byte order */
__u32 remote_port; /* Stored in network byte order */
__u32 local_port; /* stored in host byte order */
/* ... here. */

__u32 data_meta;
__bpf_md_ptr(struct bpf_flow_keys *, flow_keys);
__u64 tstamp;
__u32 wire_len;
__u32 gso_segs;
__bpf_md_ptr(struct bpf_sock *, sk);
__u32 gso_size;
__u8 tstamp_type;
__u32 :24; /* Padding, future use. */
__u64 hwtstamp;
};


```

TC Traffic Control (3/5)

- TC eBPF 프로그램은 패킷에 대한 후속 동작을 커널에게 지시할 수 있음

Return Code	Action
TC_ACT_SHOT	커널에게 패킷을 삭제하라고 지시
TC_ACT_UNSPEC	BPF 프로그램이 이 패킷에서 실행되지 않은 것처럼 동작
TC_ACT_OK	커널에게 패킷을 스택의 다음 계층으로 전달하라고 지시
TC_ACT_REDIRECT	패킷을 다른 네트워크 장치의 Ingress 또는 Egress로 전달

샘플

- 단순히 추적 라인을 출력하고, 커널에게 패킷을 삭제하라고 지시

```
int tc_drop(struct __sk_buff *skb) {
    bpf_trace_printk("[tc] dropping packet\n");
    return TC_ACT_SHOT;
}
```

TC Traffic Control (4/5)

■ 샘플

- ▶ ICMP(ping) 요청 패킷을 삭제

```
int tc(struct __sk_buff *skb) {
    void *data = (void *)(long)skb->data;
    void *data_end = (void *)(long)skb->data_end;

    if (is_icmp_ping_request(data, data_end)) {
        struct iphdr *iph = data + sizeof(struct ethhdr);
        struct icmphdr *icmp = data + sizeof(struct ethhdr) + sizeof(struct iphdr);
        bpf_trace_printk("[tc] ICMP request for %x type %x\n", iph->daddr, icmp->type);
        return TC_ACT_SHOT;
    }

    return TC_ACT_OK;
}
```

TC Traffic Control (5/5)

■ 샘플

- ▶ ping 요청을 식별하고 ping 응답을 Userspace 프로그램이 아닌 eBPF 프로그램이 응답

```

int tc_pingpong(struct __sk_buff *skb) {
    void *data = (void *)(long)skb->data;
    void *data_end = (void *)(long)skb->data_end;

    if (!is_icmp_ping_request(data, data_end)) {
        return TC_ACT_OK;
    }

    struct iphdr *iph = data + sizeof(struct ethhdr);
    struct icmphdr *icmp = data + sizeof(struct ethhdr) + sizeof(struct iphdr);

    swap_mac_addresses(skb);
    swap_ip_addresses(skb);

    // Change the type of the ICMP packet to 0 (ICMP Echo Reply)
    // (was 8 for ICMP Echo request)
    update_icmp_type(skb, 8, 0);

    // Redirecting a clone of the modified skb back to the interface
    // it arrived on
    bpf_clone_redirect(skb, skb->ifindex, 0);

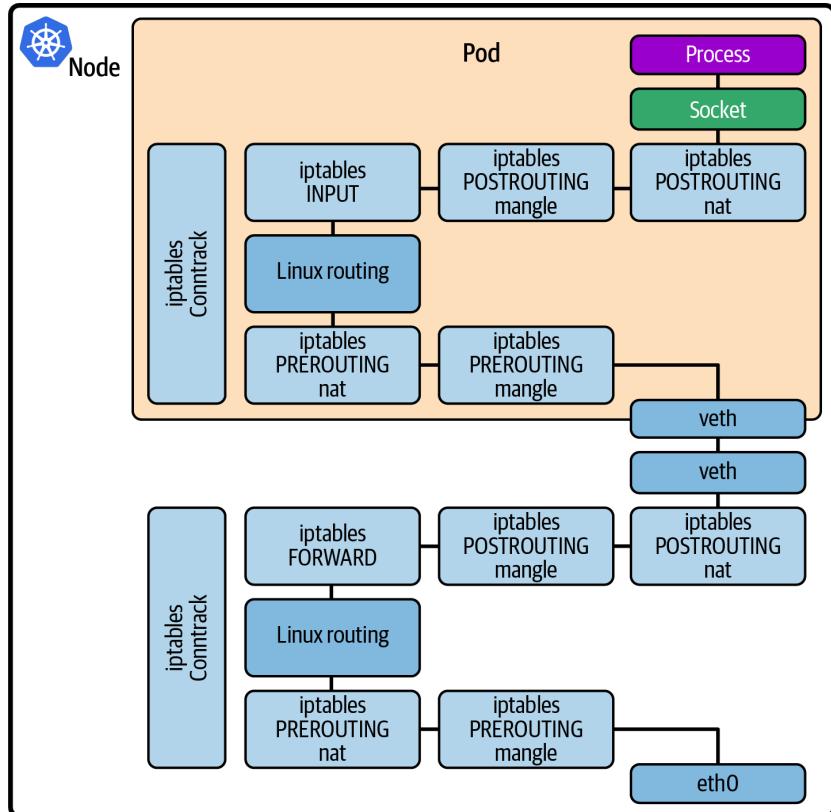
    return TC_ACT_SHOT;
}

```

이 예제는 eBPF가 어떻게 커널의 기본 동작을 완전히 대체할 수 있는지 보여줍니다. 원래 ping 응답은 전체 네트워크 스택을 거쳐 사용자 공간의 ping 프로세스까지 도달해야 생성되지만, eBPF를 사용하면 TC 계층에서 즉시 응답을 만들어 보낼 수 있습니다. 이는 커널 바이패스(kernel bypass)의 한 형태로, 지연 시간을 크게 줄일 수 있습니다.

eBPF and Kubernetes Networking (1/5)

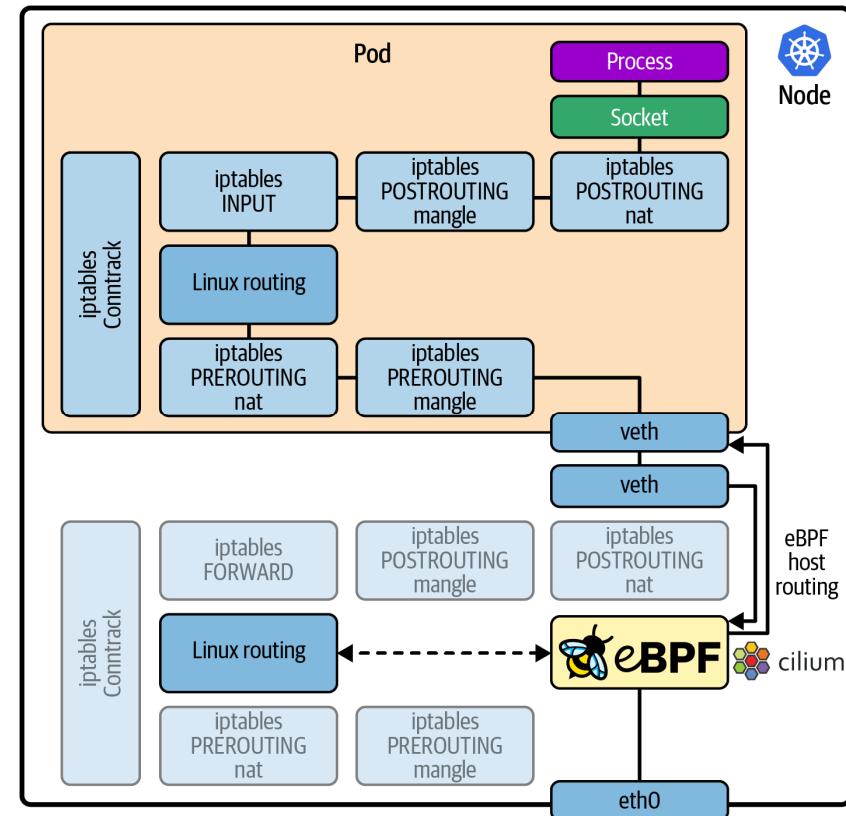
Other CNI (Calico, Flannel, ...)



머신 외부에서 애플리케이션 POD로 향하는 패킷은 호스트의 네트워크 스택을 거쳐 가상 이더넷 연결(veth)을 거쳐 POD의 네트워크 네임스페이스로 이동한 다음 다시 네트워크 스택을 통과하여 애플리케이션에 도달해야 합니다.

그 두 네트워크 스택은 동일한 커널에서 실행되므로 패킷은 실제로 동일한 처리를 두 번 거칩니다. 네트워크 패킷이 통과해야 하는 코드가 많을수록 자연 시간이 길어지므로 네트워크 경로를 단축할 수 있다면 성능이 향상될 가능성이 큽니다.

Cilium CNI (eBPF 이용)



eBPF는 iptables와 conntrack을 네트워크 규칙과 연결 추적을 관리하는 더 효율적인 솔루션으로 대체(우회)하여, 한번의 네트워크 스택을 통과하여 POD에 전달됩니다.

eBPF and Kubernetes Networking (2/5)

▪ Avoiding iptables

▪ Kubernetes CNI

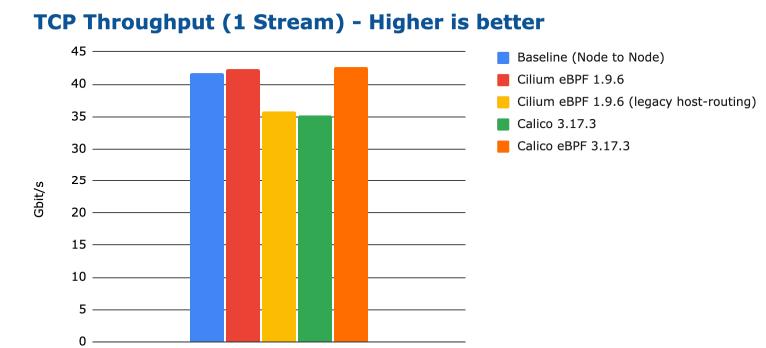
- ▶ K8S에서는 CNI를 사용하여 사용자에게 어떤 네트워킹 솔루션을 사용할지 선택할 수 있는 옵션을 제공합니다.
 - 일부 CNI 플러그인은 iptables 규칙을 사용하여 쿠버네티스에서 L3/L4 네트워크 정책을 구현합니다.
 - Service 리소스의 부하분산을 위해 kube-proxy라는 구성 요소가 필요하고, 이는 iptables 규칙을 이용합니다.

▪ iptables 문제점

- ▶ POD, Service 리소스가 추가되거나 제거될 때마다 iptables 규칙을 전체적으로 다시 작성해야 하며, 이는 대규모 성능에 영향을 미칩니다.
 - 20,000개 서비스에 대한 iptables 규칙에 대한 단일 규칙을 업데이트하는 데 5시간이 걸릴 수 있습니다.
 - <https://youtu.be/4-pawkiazEg>
- ▶ iptables 규칙 조회에 시간 소요
 - iptables를 이용하여 규칙을 찾으려면 테이블에서 선형 검색을 해야 하는데, 이는 **O(n)** 연산이고 규칙의 개수에 따라 선형적으로 증가합니다.
 - Cilium은 eBPF 해시 테이블 맵을 사용하여 해시 테이블에서 항목을 조회하고 새 항목을 삽입하는 것은 모두 대략 **O(1)** 작업이므로 훨씬 더 잘 확장됩니다.

▪ CNI Benchmark: Understanding Cilium Network Performance

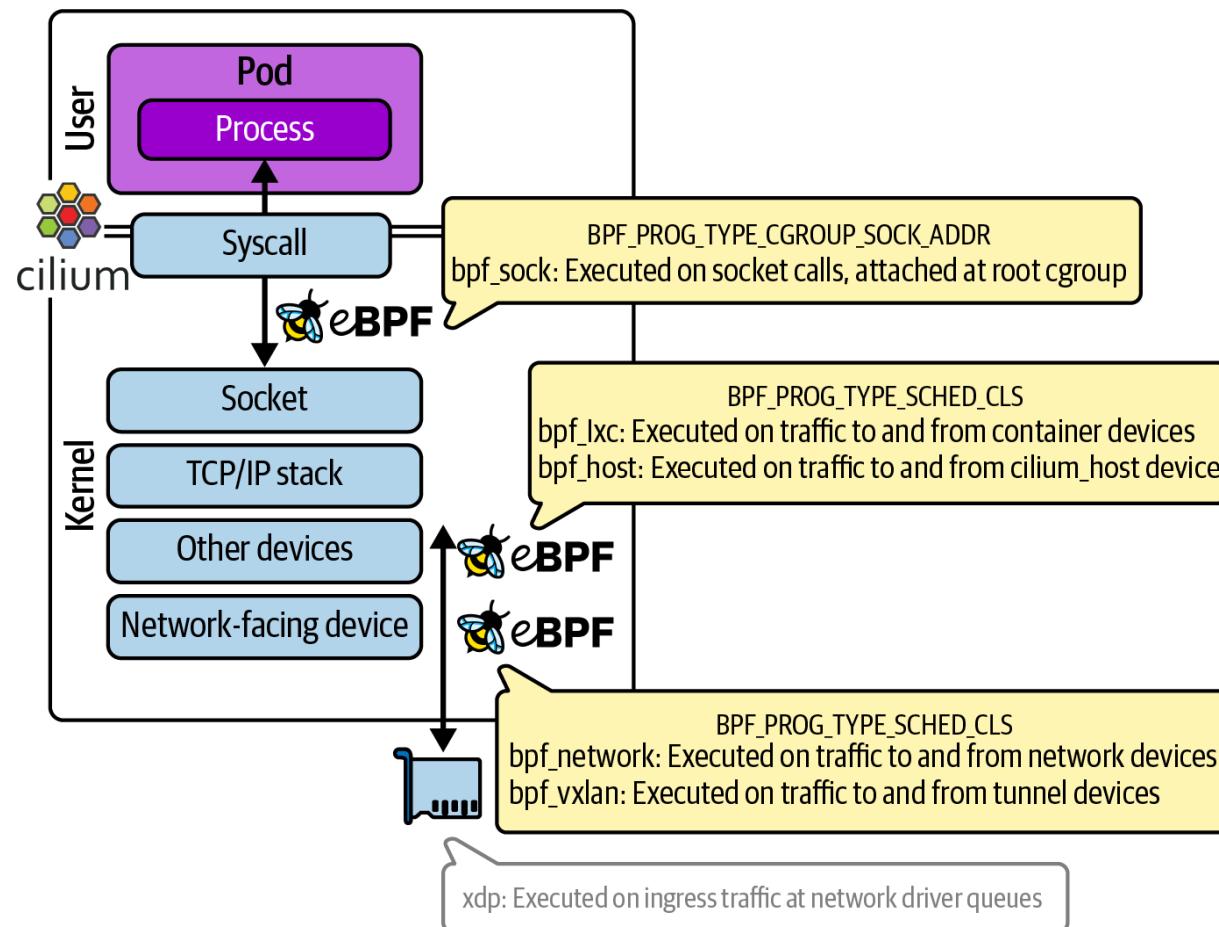
- ▶ <https://cilium.io/blog/2021/05/11/cni-benchmark/>
- ▶ eBPF 옵션이 있는 또 다른 CNI인 Calico도 iptables 대신 eBPF 구현을 선택하면 더 나은 성능을 달성한다는 것을 알 수 있습니다.
- ▶ eBPF는 확장 가능하고 동적인 Kubernetes 배포를 위한 가장 성능이 뛰어난 메커니즘을 제공합니다.



eBPF and Kubernetes Networking (3/5)

▪ Coordinated Network Programs

- ▶ Cilium과 같은 복잡한 네트워킹 구현은 단일 eBPF 프로그램으로 작성할 수 없습니다.
- ▶ Cilium은 커널의 다양한 지점에 연결되는 여러 개의 Coordinated된 eBPF 프로그램으로 구성됩니다.



eBPF and Kubernetes Networking (4/5)

▪ Network Policy Enforcement

- ▶ 쿠버네티스에서는 POD, Service 리소스의 IP 주소가 동적으로 바뀝니다.
 - IP 주소가 변경될 때마다 방화벽 규칙을 수동으로 다시 정의하는 것은 비현실적입니다.
 - 이것이 기존 방화벽이 클라우드 네이티브 환경에서 그다지 효과적이지 않은 이유입니다.
- ▶ 그래서, Kubernetes는 IP 주소가 아닌 특정 Pod에 적용된 레이블을 기반으로 방화벽 규칙을 정의하는 NetworkPolicy 리소스 개념을 지원합니다.
 - NetworkPolicy 리소스는 K8S 코어에서 제공하지 않고, 사용 중인 CNI 플러그인에 위임됩니다.
 - 일부 CNI에서는 NetworkPolicy 리소스 구현에 iptables를 이용하며, 이는 네트워크 성능저하를 동반합니다.
- ▶ Cilium CNI는 CNI는 기본 Kubernetes 정의에서 허용하는 것보다 더 정교한 NetworkPolicy 구성을 허용하는 사용자 지정 리소스를 자유롭게 구성할 수 있습니다.
 - 예를 들어 Cilium은 DNS 기반 네트워크 정책 규칙과 같은 기능을 지원하므로 IP 주소가 아닌 DNS 이름(예: "example.com")을 기반으로 트래픽을 허용할지 여부를 정의할 수 있습니다.
 - 또한 다양한 Layer 7 프로토콜에 대한 정책을 정의할 수 있습니다. 예를 들어 HTTP GET 호출에 대한 트래픽은 허용하거나 거부하지만 특정 URL에 대한 POST 호출은 허용하지 않을 수 있습니다.
 - Cilium은 Kubernetes ID를 사용하여 주어진 네트워크 정책 규칙이 적용되는지 여부를 확인합니다. 레이블이 Kubernetes에서 어떤 포드가 서비스의 일부인지 정의하는 것과 같은 방식으로 레이블은 포드에 대한 Cilium의 보안 ID도 정의합니다. 이러한 서비스 ID로 색인화된 eBPF 해시 테이블은 매우 효율적인 규칙 조회를 가능하게 합니다.

eBPF and Kubernetes Networking (5/5)

▪ Encrypted Connections

▶ Kubernetes 클러스터 내에서 트래픽이 암호화되도록 하는 가장 간단한 옵션은 투명한 암호화를 사용하는 것입니다.

- 네트워크 계층에서 완전히 이루어지고 운영 관점에서 매우 가볍기 때문에 "투명"이라고 합니다.
- 애플리케이션 자체는 암호화를 전혀 알 필요가 없으며 HTTPS 연결을 설정할 필요도 없습니다.
- 또한 이 접근 방식에는 Kubernetes에서 실행되는 추가 인프라 구성 요소가 필요하지 않습니다.

▶ 일반적으로 사용되는 커널 내 암호화 프로토콜은 IPsec과 WireGuard 두 가지가 있습니다.

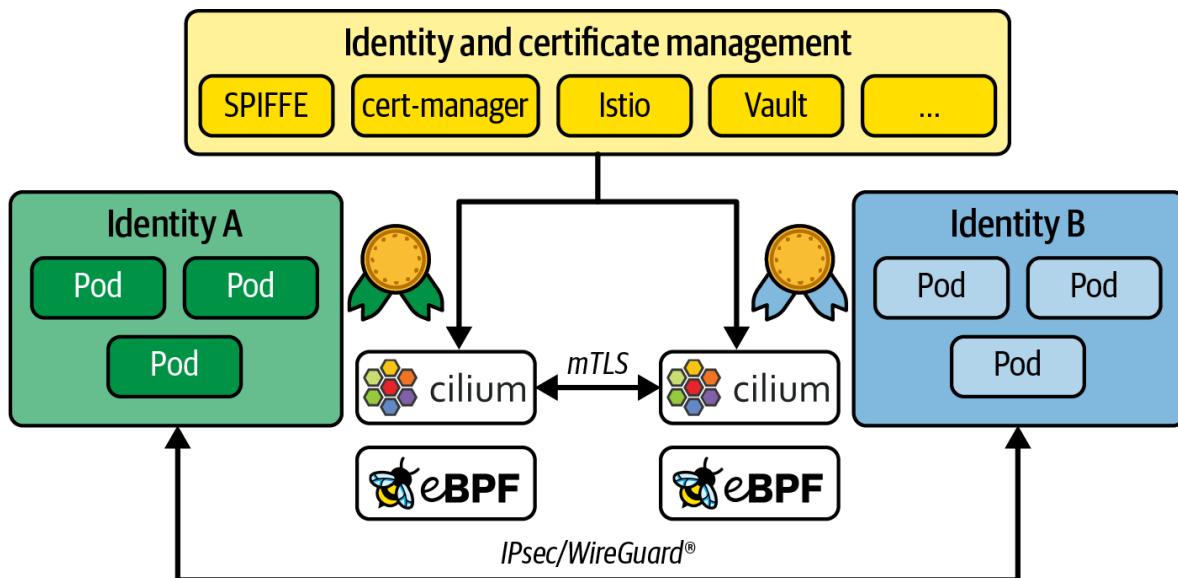
- <https://cilium.io/blog/2021/05/20/cilium-110/#wireguard>
- Cilium과 Calico CNI에서는 IPsec과 WireGuard 를 지원합니다.
- 두 머신 간에 보안 터널을 설정하고, CNI는 이 보안 터널을 통해 포드의 eBPF 엔드포인트를 연결하도록 선택할 수 있습니다.

▶ 보안 터널은 양쪽 끝의 노드 ID를 사용하여 설정됩니다.

- 투명한 암호화는 Kubernetes ID를 사용하여 클러스터의 다른 엔드포인트 간에 트래픽이 흐를 수 있는지 여부를 관리하는 NetworkPolicy와 함께 수정되지 않은 상태로 사용할 수도 있습니다.

▶ eBPF는 이제 투명한 암호화를 기반으로 하지만 초기 인증서 교환 및 엔드포인트 인증에는 TLS를 사용하는 새로운 접근 방식을 활성화하여, ID가 실행 중인 노드가 아닌 개별 애플리케이션을 나타낼 수 있도록 합니다.

- 인증 단계가 수행되면 커널 내의 IPsec 또는 WireGuard (R)를 사용하여 해당 애플리케이션 간에 흐르는 트래픽을 암호화합니다.
- 여기에는 여러 가지 이점이 있습니다. cert-manager 또는 SPIFFE/SPIRE와 같은 타사 인증서 및 ID 관리 도구가 ID 부분을 처리할 수 있으며, 네트워크가 암호화를 처리하여 애플리케이션에 완전히 투명하게 처리합니다.
- Cilium은 Kubernetes 레이블이 아닌 SPIFFE ID로 엔드포인트를 지정하는 NetworkPolicy 정의를 지원합니다. 그리고 아마도 가장 중요한 점은 이 접근 방식이 IP 패킷으로 이동하는 모든 프로토콜과 함께 사용할 수 있다는 것입니다. 이는 TCP 기반 연결에만 작동하는 mTLS보다 훨씬 뛰어납니다.



9. eBPF for Security

10. eBPF Programming

11. The Future Evolution of eBPF