논문 리뷰

제목	Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges and Applications
작성	[18반] 라민우

참고	(1) 논문의 전체 내용을 기반으로 해석하고 추가적인 설명(리뷰)을 더했다. (2) 필요에 따라 임의의 실험 과정을 포함했으며, 선택적인 내용 수정/삭제를 했다. (3) 논문 리뷰 발표 과정에서 참고하도록 작성했다.

==========

논문 목차

==========

==========

0 프롤로그

확장 버클리 패킷 필터(Extended Berkeley Packet Filter, eBPF)는
리눅스 내부의 명령어 집합이자 실행 환경이다.
eBPF는 런타임에 수정, 상호 작용 및 커널 프로그래밍이 가능토록 하게 한다.
eBPF는 빠른 패킷 처리를 위해, NIC에 더 가까우면서 패킷을 처리하는 커널 네트워크 계층인
XDP(eXpress Data Path)를 프로그래밍하는 데 사용할 수 있다.
개발자들은 C 또는 P4 언어로 프로그램을 작성한 다음 eBPF 명령어로 컴파일할 수 있으며,
이 명령어는 커널이나 프로그래밍이 가능한 장치(예: SmartNICs)에 의해 처리될 수 있다.
eBPF는 2014년 도입된 이후 Facebook, Cloudflare, Netronome과 같은 주요 기업에서 빠르게
채택되고 있다.

주 사용처 : 네트워크 모니터링, 네트워크 트래픽 조작, 로드 밸런싱, 시스템 프로파일링
논문의 목적 : eBPF를 전문가가 아닌 청중에게 제공 & 이론 중심적이고 근본적인 측면 전달

1.0 Introduction

전체 흐름 : 네트워크 기능을 프로그래밍하여 장치 또는 시스템에 포함시킬 수 있는가?
-> SDN,NFV,POF,P4 등이 있었고 eBPF의 XDP도 새롭게 등장한 것이다.

eBPF란?

(1) 리눅스 커널 내부에 제공하는 시스템 -> 명령어 set, 가상 실행 환경
(2) 커널에서 패킷 처리 수정/네트워크 장치에 대한 프로그래밍 가능
(3) 제한된 C언어로 응용 프로그램 작성, eBPF 명령어로 컴파일
-> ‘eBPF 코드’ -> 커널 or SmartNIC 에서 처리 가능
(4) ‘eBPF 프로그램’ 커널의 recompile 요구 X , 런타임에 커널 연산 수정 O

XDP란?

(1) 리눅스 네트워크 스택의 가장 낮은 계층
(2) 커널에서 패킷을 처리하는 프로그램을 설치할 수 있게 하며, 모든 패킷에 대해 프로그램은 호출됨
-> XDP는 Fast packet processing application 을 위해 설계됨 -> Programmability 향상
(3) 커널 소스 코드를 수정하지 않고 XDP 프로그램 추가/수정 가능
eBPF/XDP 사용 사례 : 네트워크 모니터링, 네트워크 트래픽 처리, 로드 밸런싱등...

환경
커널 버전 5.0

제공 자료
https://zenodo.org/record/3519347#.XbMxR6zMNhE
https://github.com/xdp-project/xdp-tutorial

1.1 BPF

"버클리 패킷 필터(Berkeley Packet Filter, BPF)는 커널 내 패킷 필터에 대한 이전 연구에서 영감을 받아 유닉스 BSD 시스템의 커널에서 패킷 필터링을 수행하는 솔루션으로 1992년 스티븐 매캔과 반 제이콥슨에 의해 제안됨"

구성 : 명령어 집합과 가상 머신(VM)
절차 : 응용 프로그램의 바이트코드가 사용자 공간에서 커널로 전송 -> Verifier 작동(보안&커널충돌 방지) -> 시스템은 프로그램을 소켓에 부착하고 도착하는 패킷마다 동작

BPF 핵심 요소
1. 커널에서 사용자가 제공한 프로그램을 안전하게 실행하게 함.
2. 간단하면서도 잘 정의된 명령어 set 제공.
3. 커널에 존재하는 JIT(Just-In-Time) 컴파일 엔진

BPF 제공 요소
(A) Bytecode instructions
(B) Packet-based memory model
(C) accumulator(1)
(D) index register(1)
(E) PC
(F) 임시 보조 메모리

(B) Packet-based memory model ?
수신된 패킷에 대해 eBPF에서 명시적인 데이터 load 명령어를 작성하지 않아도 커널 내부에서 자동으로 패킷 데이터를 메모리에 로드하므로 eBPF 프로그램이 해당 데이터 처리에 있어서 효율적이고 유연한 동작을 수행할 수 있다는 것이다.
(E) PC ?
레지스터가 2개 밖에 없는 BPF model에서 PC가 있다는 것이 어색할 수 있는데,
jmp 명령어와 Index register을 통해 간접적으로 실행흐름을 제어한다고 보면 된다.

+) PCAP lib도 BPF를 이용한 도구다.

1.2 eBPF

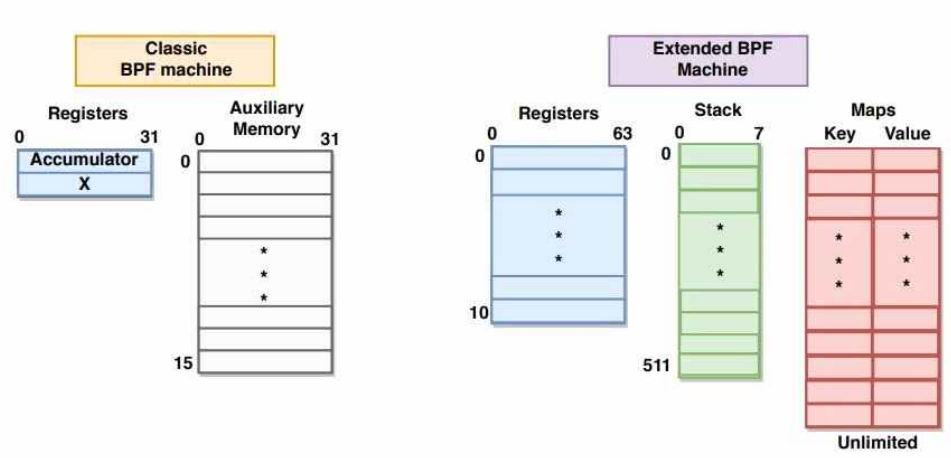


Fig. 1. BPF and eBPF processors.

cBPF -> eBPF 변화 요소
1. 레지스터 개수 증가 2개 -> 11개
2. 레지스터 폭 증가 32비트 -> 64비트
3. Stack 512 Byte
4. Map : 글로벌 데이터 저장소
   (프로그램 실행 간 데이터 유지 & 사용자 공간과의 정보 공유 기능 제공)
5. Helper function : 커널 안에서 동작하는 함수 호출하는 옵션

eBPF Program은 어떻게 runtime에 변경되거나 수정되거나 reload될 수 있을까?
- C calling convention
- 매개 변수는 레지스터를 통해 함수로 전달 (오버헤드 가능성을 줄이는 요소)
이러한 특징을 이용해서 Helper function 작동 -> 시스템 호출, Map 조작이 가능해진다.
- eBPF VM은 동적 로딩 및 프로그램 리로딩 지원

Table 1. Description of the eBPF register set.

Register	Description
r0	return value from functions and programs
r1 - r5	arguments passed to functions
r6 - r9	registers that are preserved during function calls
r10	stores frame pointer to access the stack

레지스터 r10은 유일한 읽기 전용 레지스터이며 BPF Stack 에 대한 주소를 저장한다.
레지스터 r1-r5는 C calling convention 에 따른 Parameter 전달을 위해 사용한다.
레지스터 r6-r9는 함수 호출 사이에도 값이 보존된다.

2.0 eBPF System

2.1 Overview

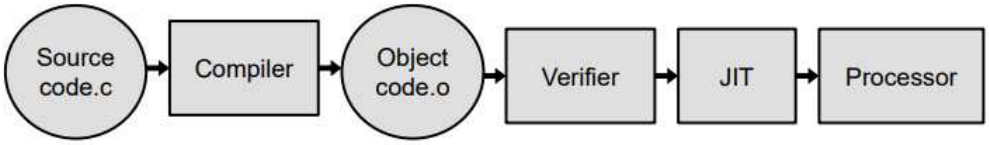


Fig. 2. eBPF Workflow.

clang 컴파일러는 소스코드를 ELF/object 코드로 변환한다.
-> ELF eBPF loader 는 특별한 시스템 호출을 사용하여 커널에 오브젝트 코드을 삽입한다.
-> Verifier는 프로그램을 분석하고, 이를 승인하면 커널은 동적 변환(JIT)을 수행한다.
-> 프로그램은 하드웨어로 offload 될 수 있으며, 그렇지 않으면 프로세서 자체(Cpu에서)에 의해 실행된다.

2.2 Compiler

버전 3.7부터는 LLVM 컴파일러 컬렉션에 eBPF 플랫폼을 위한 백엔드가 있다.
-> C코드 일부(일부 시스템콜이나 라이브러리는 커널 구조상 사용이 불가능하기 때문이다)를 사용해서 eBPF 프로그램을 개발 & clang 컴파일러를 통해 eBPF 형식의 실행 코드를 생성할 수 있게 된다.

- LLVM low level vitual machine
다양한 언어로 작성된 소스코드를 커널에서 실행될 수 있는 바이트코드로 바꿔주고 최적화하는 컴파일러 Framework 또는 라이브러리.(컴파일러의 기반)
탑재된 백앤드 기능을 이용하여 Machine Native Code를 생성해주며 JIT 컴파일을 수행할 수 있게 해준다.

- eBPF Rule
• Non-static 전역 변수 사용 X
• 무한 루프 금지
• 스택 공간은 최대 512 Byte - 보통 Map 사용을 권장한다.
 -> Stack은 프로세스마다 독립적으로 할당되는 공간인 반면, Map 은 커널 내부의 공유 메모리 공간으로 사용되기 때문이다.
 -> (chatGPT) 전역변수라는 것은 결국 메모리를 사용한다는 것이고 Map은 커널 내부에서 안전하게 관리되기 때문에 이러한 Rule이 엄격히 진행될 수 밖에 없다.

- Compiler 제작 노력들...
IOVisor, VMWare에서 eBPF용 P4 컴파일러 구현....
+) BPF 컴파일러 모음(BCC) 프로젝트 -> 표준 C 코드에 대한 추가 추상화 적용

▶ LLVM 정리
frontend, backend로 나뉘며 각각 컴파일의 역할을 나누지만, 결국엔 커널 단에서 작동할 바이트코드를 생성하는 목적에 있다.

2.3 Verifier

* 운영 체제의 무결성 및 보안을 보장
* 시스템에 로딩되는 eBPF 명령어의 정적 프로그램 분석 수행
위치 : kernel/bpf/verifier.c
Rule
 (1) 명령어 개수

(2) 프로그램이 종료되는지 여부
(3) 메모리 주소가 프로그램에 허용된 메모리 범위 내에 있는지
(4) 실행 경로가 얼마나 깊은지 여부를 확인한다. (프로그램 복잡도)
Miller (https://lwn.net/Articles/794934/)

- Verifier 작동 경로
  첫 번째 경로
  [깊이 우선 검색]
  - 프로그램 명령어를 DAG(Directed Acyclic Graph)로 파싱할 수 있는지 확인한다. 역점프가 없거나 미리 정의된 크기의 루프만 있는 eBPF 프로그램은 DAG로 합성되어 종료를 보장한다. 또한 DAG는 도달할 수 없는 명령어(그래프에는 하나의 연결된 구성 요소만 있어야함)를 확인하고 최악의 경우에는 실행 시간을 계산한다.

  두 번째 경로
  [첫 번째 명령에서 가능한 모든 경로를 탐색]
  - 상태 머신을 생성하여 상태가 올바른 동작을 나타내는지 확인하고 이미 확인한 동작에 대한 기록도 유지한다. Verifier 는 가지치기를 위해 이미 확인된 상태를 사용하므로 수행해야 할 작업량을 줄일 수 있다. 또한 분석할 경로의 최대 길이도 제한한다. 이 제한은 처음에는 64k 개의 명령어였지만 현재는 허용된 최대 프로그램 크기와 동일하다.

+) 일부 eBPF 함수는 GPL 호환 라이선스를 가진 프로그램에서만 호출이 가능하다.
이 때문에 Verifier는 프로그램이 사용하는 기능의 라이선스와 프로그램의 라이선스가 호환되는지 확인하고, 호환되지 않으면 프로그램을 거부한다. 그리고 아마 우리 프로젝트에서도 Map을 쓸 것이라면 이를 포함해야 할 것이다.
+) 커널의 무결성 및 보안을 보장하기 위해 로컬 변수와 패킷 경계를 넘어서는 메모리 액세스를 허용하지 않는다.
패킷의 임의의 바이트에 액세스하기 위해서는 항상 경계 검사를 수행해야 한다.
그러나 패킷의 저장 공간이 변경되지 않는 한 각 바이트는 한 번만 확인하면 된다.
이러한 방식으로 프로그램을 분석하는 동안 Verifier는 패킷에 대한 모든 메모리 액세스가 검사된 주소임을 보장한다.
eBPF 프로그램이 이러한 유형의 검사를 수행하지 않으면 Verifier는 이를 거부하여 커널에 로드할 수 없다.

3.0 EBPF PROGRAMS [본격적으로 eBPF Program에 대해 알아보자]

eBPF는 성능 분석, 패킷 필터링, 트래픽 분류 등 여러 종류의 응용 프로그램을 구현할 수 있다.
eBPF 시스템 전체의 가장 큰 장점은 리눅스 커널 내부에서 유연하고 안전한 프로그래밍 가능한 환경을 제공한다는 것이다.

ex) eBPF 프로그램은 런타임에 로드되고 수정될 수 있으며 kprobe, perf 이벤트, 소켓 및 라우팅 테이블과 같은 커널 요소와 상호 작용할 수 있다.

그러나 eBPF 프로그램이 사용할 수 있는 하위 시스템 및 기능은 커널에서 어디에 로드되는지, 즉 프로그램 유형에 따라 정의되는 어떤 계층 또는 하위 시스템에 연결되는지에 따라 달라진다.
- kprobe, perf ? (chatGPT)
  kprobe : 커널 내의 특정함수, 코드가 실행될 때 이벤트를 트리거하는 기능
  perf : CPU이벤트(명령어 실행), 소프트웨어 이벤트(인터럽트같은 것)등을 모니터링하는데 사용한다. (성능분석, 프로파일링)

논문의 해당 섹션에서 말하고자 하는 부분은, eBPF의 XDP 또한 다양한 프로그램 타입중에 하나고, 다양한 커널 요소들과 상호작용을 하는 프로그램을 개발할 수 있다는 것으로 보인다.

3.1 How and When Are eBPF Programs Executed?

eBPF 프로그램을 실행하기 위해서는 먼저 커스텀 프로그래밍이 가능한 인터페이스에 연결해야한다.
그리고 이러한 인터페이스를 Hook 라고 부른다. 이번에 프로젝트에서 사용할 XDP(or TC) 도 Hook 로 동작한다.
- Hook는 특정 이벤트에 대한 프로그램 등록을 허용한다.

- eBPF 프로그램은 자신이 등록한 이벤트가 있을 때마다 실행된다.
- 컴퓨터 네트워킹에서 일반적인 이벤트는 패킷을 전송하거나 수신하는 것이다.

3.2 Program Types (☆)

eBPF 프로그램은 각각 타입이 있으며 이것은 세 가지 중요한 측면을 결정한다.

(1) input
(2) Helper function
(3) Hook

ex)
socket filter program , tracing program

socket filter :
- input 매개변수는 socket buffer 이며 패킷의 메타데이터가 포함된다.
- helper function : 패킷 필터링 처리 관련

tracing:
- register 값들을 받는다.
- helper function : 시스템 이벤트 추적 & 모니터링 관련

eBPF Program Type
지원되는 프로그램 타입은 헤더 파일 linux/bpf.h에서 (enum) bpf_prog_type에 의해 정의된다.

• BPF_PROG_TYPE_SOCKET_FILTER: program to perform socket filtering;
• BPF_PROG_TYPE_SCHED_CLS: program to perform traffic classification at the TC layer;
• BPF_PROG_TYPE_SCHED_ACT: program to add actions to the TC layer;
• BPF_PROG_TYPE_XDP: program to be attached to the eXpress Data Path hook;
• BPF_PROG_TYPE_LWT_{IN, OUT or XMIT}: programs for Layer-3 tunnels;
• BPF_PROG_TYPE_SOCKET_OPS: program to catch and set socket operations such as retransmission timeouts, passive/active connection establishment etc;
• BPF_PROG_TYPE_SK_SKB: program to access socket buffers and socket parameters (IP addresses, ports, etc) and to perform packet redirection between sockets;
• BPF_PROG_TYPE_FLOW_DISSECTOR: program to do flow dissection, i.e., to find important datain network packet headers

이외에 커널 추적/모니터링을 위한 프로그램 타입
BPF_PROG_TYPE_PERF_EVENT, BPF_PROG_TYPE_KPROBE BPF_PROG_TYPE_TRACEPOINT, cgroup(리눅스 프로세스 그룹 통제,제어) (예: BPF_PROG_TYPE_CGROUP_SKB 및 BPF_PROG_TYPE_CGROUP_SOCK) 등이 있다.

지원되는 프로그램 유형의 전체 목록은 다음 명령을 사용하여 커널 소스 코드에서 직접 얻을 수 있다

“git grep -W 'bpf_prog_type {' include/uapi/linux/bpf.h”

3.3 Maps

Map은 eBPF 프로그램들이 이용할 수 있는 일반적인 키-값 구조의 저장소다.
Map은 Binary blob 으로 다뤄지고 사용자 정의된 데이터 구조나 타입의 저장이 가능하게 한다.
Map을 정의할 때는 사이즈가 반드시 나타나야한다.

Map은 BPF 시스템 호출을 사용하여 생성되며, Map의 파일 디스크립터를 통해 Map을 조작할 수 있다.

이것은 BPF_MAP_CREATE(enum bpf_cmd로 정의됨) 명령과 추가 매개 변수가 있는 bpf_attr 결합을 갖고 BPF 시스템 호출에 전달함으로써 수행된다.

>> bpf(BPF_MAP_CREATE, &bpf_attr, sizeof(bpf_attr))

bpf_attr 속성
(1) map_type: type of the map to be created
(2) key_size: number of bytes to store the key
(3) value_size: number of bytes to store the value
(4) max_entries: number of rows in the map

[추가자료] Map 종류
https://medium.com/@mazleyou/1-3-ebpf-map-%EC%84%A4%EB%AA%85-%EB%B0%8F-%EC%98%88%EC%A0%9C%EC%8B%A4%ED%96%89-0ae6b0d8e129

user-space process 는 여러 개의 Map을 생성할 수 있으며,
user-space process 와 커널에 로드된 eBPF 프로그램 모두에서 액세스할 수 있으므로 두 환경 간의 데이터 교환이 가능하다.

Map에 액세스하기 위해 eBPF 프로그램은 Map ELF 섹션에서 struct bpf_map_def 라는 특별한 전역 변수를 선언해야 한다.

During load process : file loader 는 위의 syscall 을 사용하여 선언된 Map을 생성하고
파일 디스크립터를 프로그램에 전달하며, 나중에 이는 런타임에 사용할 Verifier 에 의해 실제 포인터로 변환된다.

Code 2 Map declaration example.

```
struct bpf_map_def SEC("maps") mapname = {
    .type = BPF_MAP_TYPE_ARRAY,
    .key_size = sizeof( uint32_t ),
    .value_size = sizeof( long ),
    .max_entries = 256,
};
```

- BPF_MAP_TYPE_ARRAY: 상위 수준의 프로그래밍 언어 배열에서와 같이, 입력이 숫자에 의해 인덱싱된 Map이다.(0부터 시작하는 integer index)
- BPF_MAP_TYPE_PROG_ARRAY: eBPF 프로그램들에 대한 참조들을 저장하는 맵이다.
  예를 들어. 특정 상황들을 처리하기 위한 서브 프로그램들의 호출을 허용한다.
- BPF_MAP_TYPE_HASH: 해시 함수를 사용하여 엔트리를 저장합니다.
- BPF_MAP_TYPE_PERCPU_HASH: BPF_MAP_TYPE_HASH와 유사한 Map.
  각 프로세서 코어에 대한 해시 테이블을 생성한다.
- BPF_MAP_TYPE_LRU_HASH: 해시 함수를 사용하여 엔트리를 저장하는 Map, table이 꽉 찼을 경우, 요소인 LRU를 제거하기 위한 정책,
  (여기서 LRU 란?) 마지막으로 가장 오래 사용된 데이터를 우선적으로 제거하는 방식
- BPF_MAP_TYPE_LRU_PERCPU_HASH: 각 프로세서 코어에 대한 해시 테이블을 생성할 수 있다.
- BPF_MAP_TYPE_PERCPU_ARRAY: BPF_MAP_TYPE_ARRAY와 유사한 Map. 각 프로세서 코어에 대한 배열 생성 허용.
- BPF_MAP_TYPE_LPM_TRIE: LPM(Longest-prefix match) trie.(자료구조)
- BPF_MAP_TYPE_ARRAY_OF_MAPS: eBPF Map에 대한 참조를 저장하는 배열
- BPF_MAP_TYPE_HASH_OF_MAPS: eBPF Map에 대한 참조를 저장하는 해시 테이블.
- BPF_MAP_TYPE_DEVMAP: 네트워크 장치의 읽기 참조를 저장
- BPF_MAP_TYPE_SOCKMAP: 소켓 참조를 저장. 예를 들어 소켓 리다이렉션을 구현하는 데 사용할 수 있다.
- BPF_MAP_TYPE_QUE: 큐와 유사한 동작을 갖는 Map.
- BPF_MAP_TYPE_STACK: 스택과 유사한 동작을 갖는 Map.

지원되는 모든 Map Type 의 목록은 다음 명령을 사용하여 커널 소스 코드에서 직접 얻을 수 있다:

" $ git grep -W 'bpf_map_type {' include/uapi/linux/bpf.h "

3.3.2 지도 및 map pinning의 수명

eBPF Map의 lifecycle

모든 eBPF 객체(프로그램, Map 및 디버그 정보)는 커널에 의해 유지되는 참조 카운터(refcnt)를 가진다.
user-space process 가 bpf_create_map()이라는 호출로 Map을 생성하면,
커널은 Map refcnt를 1로 초기화한다.
그런 다음 커널은 Map을 사용하는 새로운 eBPF 프로그램이 로드될 때마다 Map refcnt를 증가시키고 그 중 하나가 닫힐 때마다 감소시킨다.
Map refcnt는 또한 그것을 생성한 프로세스가 종료(또는 충돌)할 때 감소된다.
refcnt가 0에 도달하면 'memory free' 가 트리거되어 카운터와 관련된 eBPF 객체가 파괴된다.

Map 은 어떻게 유지되는 것일까? (feat Map Pinning)

우선, 여러 프로그램 간에 동일한 eBPF Map을 한 번에 공유할 수 있다.
특정 Map 의 상위 프로세스나 그것을 사용하는 어떤 eBPF 프로그램이 동작하는 한,
그 Map은 유지한다.
그러나 eBPF Map 을 유지하는 또 다른 방법은 Map pinning 을 하는 것이다.
user-space process 는 /sys/fs/bpf/ 에 위치한 최소 커널 공간 파일 시스템인 BPF 파일 시스템에 Map (또는 다른 eBPF 개체)을 고정할 수 있다.
이 파일 시스템에 Map이 고정되면 커널은 refcnt를 증가시켜 어떤 프로그램도 사용하지 않음에도 불구하고 값을 유지할 수 있게 한다.
마찬가지로, Map이 고정되지 않을 때, Map의 refcnt는 감소하고,
사용되지 않을 경우 파괴된다.

Map pinning

Map pinning 은 user-space 로부터의 bpf() 시스템 호출(명령어 BPF_OBJ_PIN와 함께), libpf, bpftool, iproute2에서 제공되는 특수 Map 구조를 사용하여 여러 가지 방법으로 수행할 수 있다.

bpf_elf_map 이라고 하는 이 대체 구조는 커널에서 제공하는 구조와 호환되며
bpf_map_def 대신 eBPF 프로그램에서 사용할 수 있다.
이때 Map 범위를 정의하는 데 사용할 수 있는 pinning 과 같은 추가적인 멤버를 노출한다.

이 필드에서는 PIN_GLOBAL_NS, PIN_OBJ_NS 및 PIN_NONE의 세 가지 고유한 값을 받을 수 있다.
PIN_OBJ_NS로 생성된 Map은 로컬 스코프를 가지며, 이를 선언한 프로그램에서 고유하다.
결과적으로 동일한 선언을 가진 Map은 다른 프로그램에서 공존할 수 있다.
이 경우 BPF 파일 시스템에 특정 디렉토리가 생성되어 해당 노드를 저장한다.
이러한 Map 에 PIN_OBJECT_GLOBAL 값을 사용하면 글로벌 범위로 Map 이 생성된다,
이를 통해 여러 프로그램으로 공유될 수 있게 된다.

이 Map들은 pseudo-file-system 의 디렉토리 글로벌에서 엔트리를 수신할 것이다.
PIN_NONE은 Map 이 파일 시스템에서 고정되어서는 안 된다는 것을 나타내므로,
다른 응용 프로그램과 공유하는 것을 비활성화한다.
마지막으로, BPF 파일 시스템에서 Map 파일을 제거함으로써 Map 의 고정을 해제할 수 있다.
이 제거는 syscall unlink()를 사용하여 수행할 수 있다.

3.3.3 잠금 메모리

eBPF Map는 일반적으로 많은 시스템에 의해 제한되는 자원인 잠금 메모리를 사용한다.
기본 제한이 너무 낮을 수 있으며, 이로 인해 로드 시 프로그램이 거부될 수 있다.
이 제한을 극복하기 위해 잠금 메모리 제한을 충분한 제한으로 증가시키거나 심지어 완전히 제거한다.

command : ulimit -l <size>

3.4 Helper Functions

"eBPF는 cBPF와 여러 가지 점에서 차이가 있는데, 그중 하나는 프로그램이 Helper function 을 호출할 수 있도록 하는 기능이다.
Map, 라우팅 테이블, 터널링 메커니즘 등과 같은 다른 커널 구조와 각 Hook 의 컨텍스트와의 상호 작용을 할 수 있게 하기 위해 커널 자체 기반 구조에서 제공하는 특수 기능이다. "

=> Helper function 이 수행하는 작업은 Map과의 상호 작용, 패킷 수정 및 커널 Trace 에 대한 메시지 인쇄 등이 있다.

많은 프로그램 타입이 있고 각각은 특정 실행 컨텍스트를 가지고 있기 때문에,
특정 함수에 의해 호출 가능한 함수 목록은 커널에 의해 구현된 모든 Helper function의 서브셋을 나타내며, 이는 프로그램이 부착된 후크에 따라 달라진다.
예를 들어 함수 bpf_xdp_adjust_tail() 은 패킷의 마지막 바이트를 제거하는 데 사용되어 패킷의 크기를 효과적으로 줄인다. 그러나 이름에서 알 수 있듯이 XDP hook에서만 사용할 수 있다.
BCC 프로젝트는 각 프로그램 유형에 대한 Helper function 목록을 유지한다.
단, eBPF 프로그램이 사용할 수 있는 Helper function 은 커널이 제공하고 구현하는 리스트로 제한된다.

많이 복잡한데,, 간단하고 Helper function을 말하자면 아래와 같다.
-> eBPF 프로그램이 kernel 의 다양한 기능, 혹은 외부의 요소들에게 접근할 수 있도록 해주는 기능이다. 대신에 당연하게도 커널의 기능에서 무언가를 추가하거나 수정을 할 수는 없으며 단순히 값을 불러오거나 기입하는 정도로 제공되는 것이다.
-> 여기에 더 자세히 적혀있음 "https://man7.org/linux/man-pages/man7/bpf-helpers.7.html"

새로운 함수는 입력 파라미터의 최대 수를 5로 제한하면서 eBPF 프로그램 상의 모든 함수가 공유하는 호출 규약을 따라야 한다.
파라미터는 레지스터 r1-r5의 사용을 통해 사용되며, 스택과의 상호 작용은 필요로 하지 않는다.
사용 가능한 Helper function 의 수는 많고 새로운 커널 버전에 따라 지속적으로 증가한다. 버전 5.3-rc6은 총 109개의 helper 함수들을 제공한다.

- bpf_map_delete_elem, bpf_map_update_elem, bpf_map_lookup_elem:
  각각 Map에서 요소 제거, 설치-업데이트, 검색하는 데 사용
- bpf_get_prandom_u32: 32비트 pseudo random 값을 반환함
- bpf_l4_csum_replace, bpf_l3_csum_replace: 각각 L-4 및 L-3 체크섬을 다시 계산함.
- bpf_ktime_get_ns: 시스템 부팅 후 시간(나노초)을 반환함
- bpf_redirect, bpf_redirect_map: 패킷을 다른 네트워크 장치로 리디렉션하는 기능.
  두 번째는 특별한 방향전환 Map 을 통해 동적으로 장치를 지정할 수 있도록 한다;
- bpf_skb_vlan_pop, bpf_skb_vlan_push: 패킷에서 각각 VLAN 태그 제거/추가;
- bpf_getsockopt, bpf_setsockopt: user-space 호출에 대해 함수적으로 유사하다.(socket option 을 get/set 함)
- bpf_get_local_storage: 로컬 저장 영역에 대한 포인터를 반환한다.

프로그램 Type 에 따라 병렬로 실행되는 여러 프로그램들의 인스턴스 간에 이 영역을 공유할 수 있다.

Helper function 의 선언은 커널 소스 코드의 디렉토리 tools/testing/selftest/bpf에 포함된 여러 헤더 파일에 걸쳐 있다. & 대부분은 bpf_helpers.h 에 있다.

동일한 폴더에 배치되는 엔디언니스 변환을 수행하는 일부의 일반적인 연산은 bpf_endian.h 에 의해 선언된다. 이 파일은 bpf_ntohs() 및 bpf_htons()와 같은 잘 알려진 함수의 BPF 호환 버전을 제공한다.
 -> 뭔소리야?
이 부분에 대해 학습하면서 알 수 있는 재밌는 부분은, eBPF 프로그램은 이러한 함수들을 사용하기 위해 단순히 헤드 파일을 참조하여 컴파일하면 된다는 것이고, 이러한 함수들이 실제로 구현된 .c 파일이 필요없다는 것이다. 줄곧 이야기해왔지만, eBPF 의 다양한 기능들은 이미 커널 내부에 구현되어 있기 때문이다. 그리고 이것은 커널 소스 코드의 이러한 파일에 대한 경로를 -I 플래그를 사용하여 clang하도록 전달함으로써 수행할 수 있다.

+) 필요한 헤더 파일의 로컬 복사본을 만들고 커널 트리에 대한 컴파일을 피할 수 있으므로 코드의 컴파일 및 배포가 더 쉬워진다고 한다...

3.4.I Tail 호출

eBPF 프로그램은 Tail 호출을 통해 다음에 실행할 다른 프로그램을 호출할 수 있으며, caller 에게 돌아가지 않는다.
이들은 복잡한 프로그램을 단순화하고 프로그램의 동적 체인을 구축하는 데 사용될 수 있다는 것이 핵심이다.
Tail 호출은 긴 점프로 구현되며, 새로운 스택 프레임을 생성하는 것을 피하기 위해 현재 스택 프레임을 재사용하므로 함수 호출과 비교할 때 오버헤드가 최소화된다.

(추가 설명)논문 처음부분을 보면 각 프로세스마다의 무결성을 유지하기 위해서 메모리는 공유되지 않고, 각 프로세스마다 Stack 이 따로 유지된다고 했다. 그런데 갑자기 스택 프레임을 재사용한다는 것은 상식적으로 납득하기 어렵다. 왜냐하면 그러면 Stack을 공유하는 것이기 때문이다.
이해를 해보자면, 어떻게 되었든 간에 이번 프로젝트에서 여러 eBPF 프로그램을 사용해서 처리를 한다면 Tail 호출을 피하기 어렵다. 그러면 Tail 호출을 실행한 함수의 스택 프레임 자리를 호출된 프로그램이 사용하는 것이다. 따라서 Stack, Map 에 대한 프로그래밍을 잘 해야만한다.

Tail 호출의 사용
(i) eBPF 프로그램의 참조를 저장하기 위해 프로그램 array(BPF_MAP_TYPE_PROG_ARRAY)라고 하는 특수한 Map을 사용한다.
(ii) Tail 호출을 실행하기 위해 Helper function(bpf_tail_call) 사용한다.
프로그램 Array 는 키-값 쌍으로 user-space 에 의해 채워질 수 있으며,
여기서 값은 eBPF 프로그램의 파일 디스크립터이다.
Helper function 은 (컨텍스트, 프로그램 어레이 Map 에 대한 참조, lookup KEY) 세 가지 인수를 수신한다.

Tail 호출 제한 사항
(i) Tail 호출의 체인이 루프를 형성할 수 있기 때문에, 무한 루프를 피하기 위해 (논문 작성일 기준으로) Tail 호출의 최대 수는 32개로 제한된다.
(ii) eBPF는 동일한 유형의 프로그램만 Tail 호출을 허용한다.
caller 의 (JITed 또는 해석)와 일치해야 하는 변환 유형도 마찬가지이다.

3.5 Return Codes

eBPF 프로그램이 반환하는 코드는 프로그램 종류에 따라 의미와 가치(사용)가 달라진다.
예를 들어, XDP 프로그램은 처리 후(pass aground, drop, redirect 등) 패킷을 어떻게 처리해야 하는지에 대한 판단 결과를 반환한다.
물론 이것들은 bpf.h의 (enum) xdp_action으로 정의되어 있다.
TC return code 는 비슷한 의미이지만 다른 열거형을 사용한다.
반면 소켓 필터는 return code를 사용하여 스택에 전달할 패킷 길이를 나타내므로 패킷을 trim(일부 제거, 처리)하거나 아예 폐기할 수도 있다.

3.6 Interaction from User Space with libbpf

커널이 user-space 로부터 eBPF 프레임워크와 상호 작용하기 위해 bpf() syscall을 노출시키지만, 그것은 많은 목적을 위한 단일 도구이기 때문에 다소 복잡하다.
libpf는 좀 더 사용자 친화적인 API를 제공하는데, 이것은 커널 커뮤니티에 의해 개발된 user-space 라이브러리이다.
이것은 커널 소스 코드의 tool/lib/bpf에서 이용할 수 있으며, 커널로부터 대응하는 파일들을 미러링하는 GitHub의 독립형 버전으로도 배포된다.
이 라이브러리를 포함하려면 README의 단계에 따라 라이브러리를 컴파일하고 코드에 연결한다:

```
$ LIBBPF_DIR=<path-to-libbpf>/src
$ clang -I${LIBBPF_DIR}/root/usr/include/ -L${LIBBPF_DIR} myprog.c -lbpf
```

루트 디렉터리는 libpf 컴파일 시 사용되는 DESTDIR에 따라 다를 수 있다.
마지막으로 라이브러리를 C 코드에 포함시킨다.

```
# include <bpf/libpf.h>
```

디렉터리 tools/testing/selftest/bpf 에서 사용할 수 있는 몇 가지 예는 이 라이브러리의 사용 사례를 보여준다.
또한 GitHub의 독립형 버전은 커널 소스에 대한 컴파일 없이 프로젝트에 통합 libpf를 구축하는 방법에 대한 자세한 지침을 가지고 있다.
이 API는 bpf() 시스템 호출의 직접적인 몇몇 wrapper 를 포함하고 있으며, eBPF시스템과 상호작용을 하기위해 노출시킨다.
예를 들어, 사용자는 각각 struct bpf_map, struct bpf_program, struct bpf_object를 사용하여 Map, 프로그램 및 객체 파일에 대한 정보를 처리할 수 있다.
이러한 객체 유사 유형에는 각각 특정 getters와 setters가 있으며,
그 이름은 구조의 이름으로 시작하고 이중 밑줄과 수행할 액션의 선언적 이름이 뒤따른다.
다음 단락에는 이러한 객체 유형 각각에 대해 일반적인 함수 몇 가지가 나열되어 있다.

clang으로 eBPF 프로그램을 포함하는 .c 파일을 컴파일한 후 생성된 오브젝트 파일은 각 프로그램에 대응하는 여러 ELF Section 을 포함할 것이다. User-space 프로그램은 bpf_object__* 함수 계열을 사용하여 이러한 파일과 상호 작용할 수 있다.
몇 가지 예는 다음과 같다

• bpf_object__open 및 bpf_object_open_xattr: 객체 파일을 읽고 포인터를 반환함
structure bpf_object. _xattr 버전을 사용하면 프로그램 유형을 지정할 수 있음
• bpf_object__load 및 bpf_object_load_xattr: struct 구조에서 프로그램 struct bpf_object를 커널에 입력. _xattr 버전은 원하는 로그 레벨로 지정 가능
• bpf_object__pin_maps: 객체 파일에서 모든 Map의 pinning 을 처리할 수 있음
• bpf_object_for_each_program: 객체 파일에서 각 프로그램을 반복하는 매크로
• bpf_object__find_program_by_title: BPF 프로그램에 섹션 이름을 기반으로 핸들을 반환

상기 함수들 중 일부는 또한 각각의 대응물(unload, close, unpin)을 갖는다.
API에 포함된 또 다른 유용한 함수 Set은 프로그램들을 부분적으로 취급할 수 있게 한다.
예를 들어, 파일 내의 각 프로그램에 특정 액션들을 적용하기 위해 위에서 보여준 프로그램 iterator 와 함께 사용될 수 있다.
이 함수들은 bpf_program__* 시그니쳐를 가지며, 일부는 아래에 나와 있다

• bpf_program_is_<type>, bpf_program_set_<type> : 프로그램의 종류에 대해 각각 getter 와 setter 이다. 각각의 종류는 고유한 함수 쌍을 가지며,
여기서 <type>은 해당 이름(예를 들어 sched_cls, sched_xdp 등)으로 대체된다.
• bpf_program__load: 커널에 주어진 프로그램을 로드한다.
• bpf_program__fd: BPF 프로그램의 파일 디스크립터를 반환한다.
• bpf_program__pin: 주어진 프로그램을 특정 파일 경로에 고정한다.
• bpf_program__set_ifindex: Map 및 프로그램을 offload 할 장치의 ifindex를 설정한다.

bpf_map__* 으로 시작하는 함수는 생성, 정보 검색, 재사용, 고정 등에 이르기까지 Map 객체에 대한 작업을 지원한다.

• bpf_map__def: 기본 지도 정보(유형, 크기 등)를 반환한다.
• bpf_map__reuse_fd: 새 프로그램을 로드할 때 기존 Map 을 재사용할 수 있다.
• bpf_map__resize: Map 에서 허용되는 최대 항목 수를 변경하는 데 사용된다.
• bpf_map__fd: 지정된 Map 에 대한 파일 디스크립터를 반환한다.
• bpf_map_for_each: 객체 파일의 모든 Map 에 대해 반복하는 매크로

API 에 대해 소개할 때 제외된 부분은 perf 버퍼, 전처리 도우미 등과의 상호 작용을 포함한다.
모든 사용 가능한 호출의 전체 목록과 표시된 함수의 전체 시그니쳐는 소스 코드의 libpf.h 헤더 파일을 확인하면 된다.

3.7 Basic Program Structure

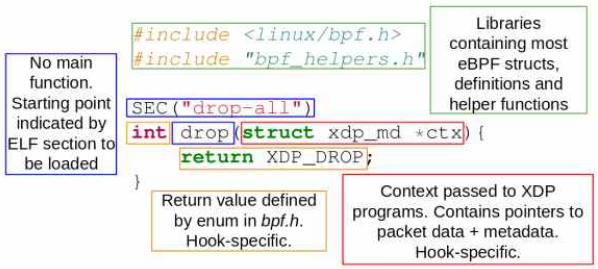


Fig. 3. Dropworld example illustrating the structure of an eBPF program.

이제부터는 본격적인 eBPF 프로그램의 기본 구조를 알아볼 것이다.
논문에 자세히 적혀 있지만, 이는 수신된 모든 패킷을 드롭하는 간단한 XDP 프로그램이다.
라이브러리 linux/bpf.h에는 추가 헤더 파일이 필요한 트래픽 제어(TC) 및 perf와 같은 특정 하위 시스템을 제외하고 eBPF 프로그램이 사용하는 모든 구조 및 상수 정의가 있다.

따라서 일반적으로 모든 eBPF 프로그램은 이 .h 파일을 포함해야 한다.
반환 값과 프로그램이 수신하는 입력 파라미터는 Hook 에 따라 달라진다.
Fig 3 을 보면, XDP 프로그램은 구조 xdp_md에 대한 포인터를 받는다.
다른 Hook 의 프로그램은 다른 컨텍스트 구조를 받는다.

+) 해당 XDP 프로그램을 잘 보면, 표준 C 프로그램에서 일반적으로 사용하는 main 함수를 포함하지 않는다. 대신 프로그램의 시작점은 ELF 오브젝트 파일에서 Section 으로 표시된다.
컴파일이 되면 표시되는 프로그램은 default .text 섹션에 위치하게 된다.

[참고]
아래에서는 Fig 3의 프로그램 예제의 객체 파일과 디셈블러 출력을 간단하게 보여준다.

```
0:       b7 00 00 00 01 00 00 00        r0 = 1
1:       95 00 00 00 00 00 00 00        exit
```

r0 = 1 (XDP_DROP)

4.0 NETWORK HOOKS

Hook 한방 이해 : “XDP는 eBPF 프로그램 타입중 하나이면서 동시에 Hook 로 동작한다.”

컴퓨터 네트워킹에서 Hook 는 OS 에서 호출 전 또는 실행 중에 패킷을 가로채는 데 사용된다.
리눅스 커널은 eBPF 프로그램을 부착할 수 있는 여러 Hook 를 노출하므로 데이터 수집과 사용자 정의 이벤트 처리가 가능하다.
리눅스 커널에는 많은 Hook 포인트가 있지만 이번 논문에서는 네트워킹 하위 시스템에 존재하는 XDP와 TC에 초점을 맞춘다.
이들을 함께 사용하여 RX와 TX 모두에서 NIC에 가까운 패킷을 처리할 수 있으므로 많은 네트워크 응용 프로그램을 개발할 수 있다.

eBPF를 사용하여 이 두 Hook 를 프로그래밍하는 방법과 각 Hook 에 프로그램을 로드하는 방법에 대해 알아보자.

4.1 Kernel’s Networking Layers

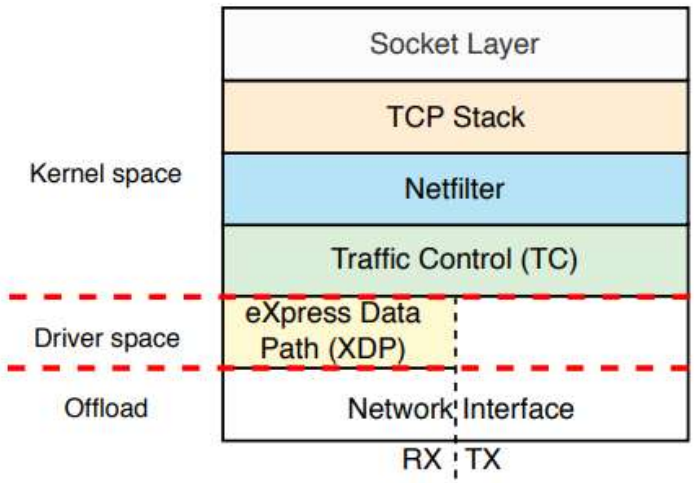


Fig. 4. Linux kernel network stack.

OS로 들어오는 패킷은 기본적으로 Fig 4 와 같이 커널에서 여러 계층에 의해 처리된다.
user-space application 으로 향하는 패킷은 이 모든 계층을 거치며 Hook 될 수 있다.
그리고 이 과정에서 Netfilter 계층에 있는 iptables와 같은 모듈에 의해 수정된다.
앞서 설명한 바와 같이, eBPF 프로그램은 커널 내부의 여러 곳에 부착할 수 있어 패킷 mangling 및 filtering이 가능하다.

4.2 eXpress Data Path

XDP는 리눅스 커널 네트워크 스택의 가장 낮은 계층이다.
이것은 RX 경로, 즉 Device 의 네트워크 드라이버 내부에만 존재하며,
OS에 의해 메모리 할당이 이루어지기도 전에 네트워크 스택의 가장 초기 지점에서 패킷 처리를 허용한다.
그리고 여기서 eBPF 프로그램을 부착할 수 있는 Hook 를 노출한다.
이 Hook 에서 프로그램은 들어오는 패킷에 대해 빠른 결정을 내릴 수 있고 패킷에 대해 임의의 수정을 수행할 수 있기 때문에, 처리하는 과정에서 발생할 수 있는 오버헤드를 피할 수 있다.
패킷을 처리한 후, XDP 프로그램은 액션을 반환하는데, 이는 프로그램 종료 후에 패킷에 무엇을 해야 하는지에 대한 최종 판단 결과를 나타낸다.

4.2.1 XDP 입력 context

XDP 프로그램에 의해 보여지는 context 는 커널에 의해 전달되는 단일 입력 파라미터에 의해 정의된다.
이것은 bpf.h로 정의되는 xdp_md 유형의 구조이며, 아래 Code 3으로 재현된다.

data, data_end : 패킷 데이터의 시작과 끝에 대한 포인터를 포함.
data_meta : data meta의 포인터로, XDP 프로그램이 다른 계층과 패킷 메타데이터를 교환할 때 사용할 수 있는 메모리 영역의 주소를 자유롭게 유지함.
ingress_ifindex, rxqueue_index : 각각 패킷을 수신한 인터페이스와 해당 RX 큐의 인덱스를 유지.

이 두 값에 액세스하면 BPF 코드가 커널 내부에 다시 작성되어 실제로 해당 값을 유지하는 커널 구조체 xdp_rxq_info 에 액세스한다.

Code 3 Declaration of `struct xdp_md` as-is from `bpf.h`

```
struct xdp_md {
        __u32 data;
        __u32 data_end;
        __u32 data_meta;
        /* Below access go through struct xdp_rxq_info */
        __u32 ingress_ifindex; /* rxq->dev->ifindex */
        __u32 rx_queue_index;  /* rxq->queue_index  */
};
```

처음 세 필드는 포인터 값을 유지하지만, C 데이터 Type 은 부호가 없는 정규 32바이트 정수이다.
따라서 메모리 주소를 올바르게 사용하려면 프로그램이 먼저 포인터 값을 cast 해야 하며, 이는 거의 모든 XDP 프로그램의 시작 부분에 존재하는 코드 snippet 을 통해 이루어진다

```
void *data_end = (void *(long)ctx->data_end;
void *data = (void *(long)ctx->data;
```

여기서, ctx는 위에 표시된 xdp_md 형식의 XDP 프로그램의 입력이다.

4.2.2 XDP Actions

Table 2 는 가능한 모든 XDP Action 과 그 값 및 그 설명이다.
동작은 프로그램 리턴 코드로 지정되며, eBPF 프로그램이 종료되기 직전 레지스터 r0에 저장된다.

Table 2. Description of XDP action set

Value	Action	Description
0	XDP_ABORTED	Error. Drop packet.
1	XDP_DROP	Drop packet.
2	XDP_PASS	Allow further processing by the kernel stack.
3	XDP_TX	Transmit from the interface it came from.
4	XDP_REDIRECT	Transmit packet from another interface.

처음 네 가지 동작은 단순 반환 값(파라미터 없음)으로, 예외를 제기하는 동안 패킷이 드롭되거나(XDP_ABORT), 조용히 드롭되거나(XDP_DROP), 커널 스택(XDP_PASS)으로 전달되거나(XDP_TX) 동일한 인터페이스를 통해 즉시 재전송되어야 함을 나타낸다.

XDP_REDIRECT Action 은 XDP 프로그램이 아래 3가지를 할 수 있게 한다.
(i) 다른 NIC(물리적 또는 가상적)
(ii) 추가 처리를 위한 다른 CPU
(iii) 사용자 공간 처리를 위한 AF_XDP 소켓으로 패킷 리디렉션

다른 것들과 달리, 이 액션은 Redirection 대상을 지정하기 위한 매개 변수를 필요로 한다.
이것은 두 가지 Helper function 중 하나인 bpf_redirect() 또는 bpf_redirect_map()을 통해 이루어진다.
전자는 타겟의 인터페이스 인덱스를 수신하고 네트워크 디바이스에 집중되어 있다.
후자는 보다 일반적인 대안으로, 최종 타겟을 검색하기 위해 보조 MAP에 대한 lookup을 수행하며, 이는 넷 디바이스 또는 CPU 둘 다일 수 있다.
.
패킷 전송을 일괄 처리하여 bpf_redirect()와 비교할 때 훨씬 더 나은 성능을 제공하고,
Map 엔트리가 사용자 및 커널 공간에서 동적으로 수정될 수 있으므로 더 나은 유연성을 제공하기 때문에 두 번째 옵션이 권장된다.

4.2.3 XDP 작동 모드

XDP에 연결된 eBPF 프로그램은 성능 향상을 위해 디바이스 드라이버 레벨에서 패킷 처리 파이프라인을 변경하며, 이는 관련 네트워크 드라이버에 의한 명시적인 지원을 필요로 한다.
(i40e, nfp, mlx* 및 ixgbe 계열과 같은 고속 장치를 위한 일부 드라이버는 이미 이러한 기능을 가지고 있다.)

이러한 드라이버와 호환되는 장치에서 XDP 프로그램은 OS 에 의해 처리되기도 전에 드라이버에 의해 직접 실행된다. -> 이를 XDP Native mode라고 한다.

BCC Project는 XDP 지원 드라이버의 최신 목록을 유지한다.
그러나 커널은 XDP Generic이라는 호환성 모드를 제공하는데, 이 모드는 드라이버 레벨에서 네이티브 지원 없이 디바이스들에 대해 XDP 프로그램 실행을 가능하게 한다.
(실제로 간단한 확인을 해볼 수 있는 부분)

이 모드에서 XDP 실행은 네이티브 실행을 에뮬레이트하면서 OS 자체에 의해 이루어진다.
이 방식은 명시적인 XDP 지원이 없는 디바이스들도 에뮬레이션을 수행하는 데
필요한 소켓 버퍼 할당 추가 단계들로 인해 성능이 저하되는 비용으로 이들에 프로그램을 부착할 수 있다.
시스템은 eBPF 프로그램을 로드할 때 이 두 가지 모드 중에서 자동으로 선택한다.
그리고 ip tool을 이용하여 프로그램의 동작 모드를 확인할 수 있다.

XDP 오프로드라는 또 다른 동작 모드가 있다. 이름에서 알 수 있듯이 eBPF 프로그램은 compatible programmable NIC로 오프로드되어 다른 두 모드에 비해 훨씬 뛰어난 성능을 발휘한다.

이 모드는 프로그램을 로드할 때 명시적으로 표시되어야 한다.

4.2.4 XDP와 XDP 오프로드 예제

XDP 프로그램을 컴파일하고 로드하는 방법을 설명하기 위해 Fig 3 의 예제를 사용해보자. 간단한 XDP 프로그램이며 이에 따르면 모든 패킷이 네트워크 인터페이스에 도착하자마자 드롭된다.

코드는 clang 컴파일러를 사용하여 ELF 객체 파일로 컴파일할 수 있다

```
$ clang - target bpf -O2 -c dropworld.c -o dropworld.o
```

ip tool은 커널에 오브젝트 파일을 로드할 수 있다.
예제 코드에서 프로그램은 섹션 태그가 없으므로 생성된 바이트코드는 ELF 오브젝트 파일의 기본 섹션(.text) 내부에 위치한다.
이 섹션은 프로그램을 로드할 때 지정되어야 한다.
-force 파라미터는 해당 인터페이스에 다른 프로그램이 로드되어 있어도 프로그램이 로드되어야 하며, 이 프로그램은 대체되어야 함을 나타낸다.
[DEV] 파라미터는 해당 인터페이스 이름으로 변경되어야 한다.

```
# ip -force link set dev [DEV] xdp obj dropworld.o sec.text
```

프로그램을 로드한 후 ip 도구를 사용하여 프로그램이 XDP Hook 의 인터페이스에 부착되었는지 확인할 수도 있다.

```
$ ip link show dev [DEV]
--------------------------------------------------------------------------------
DEV: <BROADCAST, Multicast, UP, LOWER_UP> mtu 1500 xdp qdisc
mq state UP mode DEFAULT group default qlen 1000
link/ether 00:16:3d:13:08:80 brd ff:ff:ff:ff:ff:ff
prog/xdp id 27 tag f95672269956c10d jited
--------------------------------------------------------------------------------
```

첫 번째 출력 행의 키워드 xdp는 XDP 네이티브 모드에서 XDP 프로그램이 인터페이스에 연결되어 있음을 나타낸다.
다른 가능한 출력은 xdp generic과 xdp offload일 수 있다.

프로그램을 매개 변수를 off로 전달함으로써 인터페이스에서 제거될 수도 있다:

ip link set dev [DEV] xdp 꺼짐

이 마지막 예제에서 eBPF 프로그램은 드라이버가 CPU를 사용하여 XDP Hook 에서 실행되었다.
프로그램을 오프로드하기 위해서는 이전과 동일한 방법을 사용할 수 있지만
xdpoffload 파라미터를 ip link set 명령으로 전달한다.

ip-force link set dev [DEV] xdpoffload object dropworld.o sec.text

이전과 마찬가지로 프로그램을 제거하려면 다음 명령을 실행한다.

ip link set dev [DEV] xdp offload off

4.3 Traffic Control Hook

현재 XDP 계층은 많은 응용 프로그램에 매우 적합하지만 입력 트래픽(수신 중인 패킷)만을 처리할 수 있다.
출력 트래픽(전송 패킷)을 처리하기 위해 이더넷 프레임 전체에 액세스할 수 있는 NIC에 가장 가까운 계층은 트래픽 제어(TC) 계층이다.
이 계층은 리눅스에서 트래픽 제어 정책을 실행하는 역할을 한다.
그 안에서 네트워크 관리자는 시스템에 존재하는 다양한 패킷 큐에 대해 서로 다른 큐잉 규율(qdisc)을 구성할 수 있을 뿐만 아니라 패킷을 거부하거나 수정하기 위한 필터를 추가할 수 있다.

TC는 clsact라는 특별한 큐잉 규율 타입을 가지고 있다.
eBPF 프로그램들에 의해 큐 처리 동작들을 정의할 수 있게 해주는 후크(hook)를 노출시킨다.
처리될 패킷에 대한 포인터들은 그것의 입력 컨텍스트의 일부로서 구성된 eBPF 프로그램으로 전달된다.

이 구조는 프로그램이 커널의 소켓 버퍼 내부 데이터 구조로부터 액세스하도록 허용되는 특정 필드들에 대한 UAPI이다. 그것은 구조 xdp_md와 동일한 데이터 및 data_end 포인터들을 가지지만 XDP의 경우와 비교하면 훨씬 더 많은 정보를 가지고 있다.

이것은 TC 레벨에서 커널이 프로토콜 메타데이터를 추출하기 위해 패킷을 이미 파싱했으므로 더 풍부한 컨텍스트 정보가 eBPF 프로그램으로 전달된다는 사실에 의해 설명된다. struct_sk_buff의 전체 선언은 간결함을 위해 생략되지만 include/uapi/linux/bpfh에서 볼 수 있다.

프로그램 실행 중에, 입력 패킷은 수정될 수 있고, 리턴 값은 TC에게 그것을 위해 어떤 조치를 취해야 하는지를 나타낸다. 라이브러리 linux/pkt_cls.h는 사용 가능한 리턴 값들을 정의한다. 가장 일반적인 것들은 표 3에 나열되어 있다.

4.4 Comparison between XDP and TC

Table 3. Description of TC set of actions

Value	Action	Description
0	TC_ACT_OK	Delivers the packet in the TC queue.
2	TC_ACT_SHOT	Drop packet.
-1	TC_ACT_UNSPEC	Uses standard TC action.
3	TC_ACT_PIPE	Performs the next action, if it exists.
1	TC_ACT_RECLASSIFY	Restarts the classification from the beginning.

TC 후크에 프로그램을 로드하는 것은 iproute2 패키지에서 사용할 수 있는 tc 툴을 사용하여 수행된다. 다음 명령어는 clsact qdisc를 생성하고 eBPF 프로그램을 로드하여 인터페이스 eth0에서 패킷을 처리하는 방법을 보여준다

```
# tc qdisc add device0 clact
# tc filter add device0 <direction> bpf daobj <ebpf-obj> sec <섹션>
```

<direction> 파라미터는 프로그램이 어느 방향과 연관되어야 하는지를 나타내며, 이는 입력 또는 출력이 될 수 있다. <ebpf-obj>와 <section>은 각각 컴파일된 eBPF 코드를 포함하는 파일과 프로그램을 로드할 섹션의 이름이어야 한다.

eth0에 이미 로드된 프로그램이 있는지 확인하려면 다음 명령을 사용한다:

```
# tc filter show device0 <direction>
```

TC 계층을 위한 eBPF 프로그램과 XDP 계층과의 상호작용의 예
https://github.com/racyusdelanoo/bpf-tutorial

5.0 EXAMPLES

두 후크는 DDoS 완화, 터널링 및 링크 계층 정보 처리와 같은 유사한 응용 프로그램에 사용될 수 있다. 그러나 XDP는 소켓 버퍼 할당이 발생하기 전에 실행되므로 TC의 프로그램보다 더 높은 처리량 값에 도달할 수 있다. 반면 후자는 struct __sk_buff를 통해 사용 가능한 추가 파싱된 데이터의 이점을 얻을 수 있으며 TX의 가장 낮은 계층인 입력 트래픽과 출력 트래픽 모두에 대해 eBPF 프로그램을 실행할 수 있다

5.1 TCP Filter

첫 번째 예는 TCP 세그먼트가 있는 패킷만 받아들이는 프로그램(Code 4)이다. 이는 Code 1의 예와 유사하지만, TCP 세그먼트가 없는 패킷을 드롭하기 위해 eBPF를 사용한다.

여기서는 상위 레벨 C 코드와 컴파일 후 생성된 실제 eBPF 어셈블리어와 같은 코드라는 두 가지 관점으로 제시한다.

5.1.1 C 코드

해당 프로그램은 XDP Hook 에 로드되도록 설계되었으므로, 함수의 입력 파라미터는 앞에서 논의한 바와 같이 xdp_md 형식이어야 한다.
처리되는 패킷의 바이트는 데이터와 data_end 포인터로 구분되며, 이는 패킷에 액세스하기 위해 프로그램 전체에서 사용되어야 한다.
이 두 값에 대한 형식 변환은 표준이므로 패킷 데이터에 액세스하는 모든 eBPF 프로그램의 시작 부분에 9번과 10번 행을 사용해야 한다.
데이터를 사용하여 헤더의 파싱은 리눅스에서 제공하는 표준 헤더 파일로 수행할 수 있다.
그러나 다른 일반적인 패킷 파싱 리눅스 프로그램과의 주된 차이점은 프로토콜 헤더 데이터에 실제로 액세스하기 전에 바인딩 검사가 필요하다는 것이다.
커널 Verifier는 엄격한 메모리 바인딩 검사를 수행하므로 패킷 데이터에 대한 모든 액세스는 테두리 검사(라인 14와 21)가 포함된 if 문으로 덮어야 한다.
but, 각 바이트는 단 한 번만 확인하면 된다

패킷의 저장 공간을 수정하는 헬퍼 함수들이 사용된다(예를 들어, bpf_xdp_adj_head()).
그 경우, 그러한 함수들을 호출한 후에 모든 체크를 다시 할 필요가 있다.
eBPF 프로그램이 이러한 종류의 체크를 수행하지 않으면, verifier에 의해 로드 시간 동안 거부되고 커널에 로드되지 않는다.
이 프로그램은 패킷의 프로토콜 번호를 추출한 후 TCP에 해당하는지 확인한다
프로토콜을 사용하여 스택(26번 라인)을 통과할 수 있도록 한다.
그렇지 않으면 패킷은 그냥 드롭된다(28번 라인).

5.1.2 eBPF 바이트코드

컴파일 시 clang은 eBPF 명령어로 객체 파일을 생성하고, 이 파일은 커널에 로딩될 수 있다.
Verifier 는 프로그램에 기초하여 DAG를 생성한다.
Fig 5는 이 예를 각각의 DAG에 보여준다. 각 DAG 노드는 하나 이상의 eBPF 명령어를 포함한다.
조건부 점프 노드는 두 개의 출력 라인과 밝은 회색 배경을 포함하는 노드이다.
실선은 다음 비순환 제어 흐름 그래프(ACFG) 노드를 나타낸다. 점선은 다른 ACFG 노드로의 점프를 나타낸다. 우리의 예에서, 조건부 점프 명령어에는 세 가지 종류가 있다: jgt (더 크면 점프), jne (동일하지 않으면 점프), jeq (동일하지 않으면 점프)

이 명령어 각각에 대한 마지막 숫자는 조건이 유효할 때 얼마나 많은 명령어를 점프해야 하는지를 나타낸다

Code 4 Example of a C code that checks if the packet contains an IPv4 TCP segment.

```c
#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/in.h>
#include "bpf_endian.h"

int isTCP( struct xdp_md *ctx ) {
    void *data_end = (void *)(long) ctx->data_end;
    void *data_begin = (void *)(long) ctx->data;
    struct ethhdr* eth = data_begin;

    // Check packet's size
    if(eth + 1 > data_end)
      return XDP_PASS;

    // Check if Ethernet frame has IPv4 packet
    if (eth->h_proto == bpf_htons( ETH_P_IP )) {
        struct iphdr *ipv4 = (struct iphdr *)( ((void*)eth) + ETH_HLEN );

        if(ipv4 + 1 > data_end)
          return XDP_PASS;

        // Check if IPv4 packet contains a TCP segment
        if (ipv4->protocol == IPPROTO_TCP)
          return XDP_PASS;
    }
    return XDP_DROP;
}
```

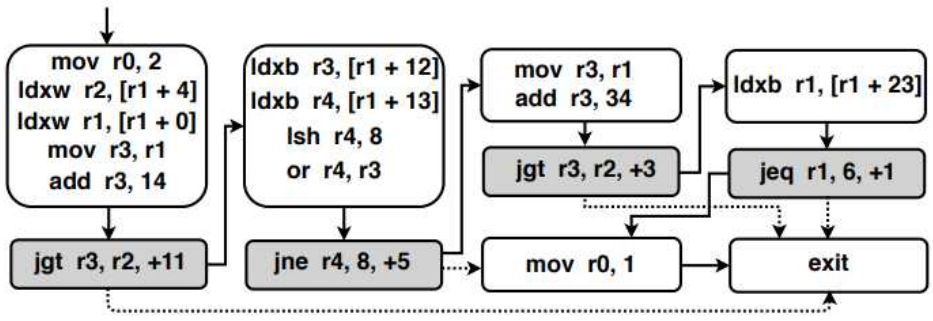


Fig. 5. Directed Acyclic Graph example of Code 4.

이 eBPF 프로그램을 더 잘 이해하려면 레지스터 r1은 메인 메모리에 저장된 입력 컨텍스트의 포인터로 시작하고 레지스터 r0은 반환 값을 저장한다(표 1).
첫 번째 노드에서 제1 어셈블러 명령어는 r0(표 1)을 2(XDP_PASS)로 설정한다(표 2). 또한 로드 명령어는 전달된 입력(라인 10-12)으로부터 패킷의 시작과 끝에 대한 참조를 계산한다.
그런 다음 이더넷 헤더의 경계를 검사하여 나중에 유효한 메모리 액세스를 보장한다(라인 14-16).

검사에 실패하면 첫 번째 점프 명령어는 흐름을 종료 명령어로 전환하여 프로그램을 종료한다. 그렇지 않으면 이더넷 헤더의 바이트 12와 13이 로드되고(0부터 카운트됨),
이것이 이더넷 타입 필드이다(라인 19). 그런 다음 바이트 스왑을 수행하여 엔디안니스를 설정한다. 두 번째 점프 명령어는 이더넷 타입이 0x0800(라인 19)인지 비교한다.
그런 다음 IP 헤더의 경계를 검사하여 나중에 유효한 메모리 액세스를 보장한다(라인 21-23). 다음 IP 프로토콜 필드는 IPv4 패킷에서 추출된 다음 프로그램은 TCP 프로토콜(값 6)(라인 26)(DAG의 마지막 회색 배경 노드)인지 확인한다.
마지막으로 레지스터(r0)에 이전에 수락을 나타내는 값 2(XDP_PASS)가 로드되었기 때문에 패킷은 커널로 전달된다. IP 패킷이 TCP 헤더를 포함하지 않으면 값 1(XDP_DROP)이 로드되어 r0에 로드된다.
마지막 명령어는 코드가 종료됨을 알려준다.

5.2 User and Kernel Space Interaction

아래는 samples/bpf 디렉토리에 있는 커널 소스 코드에서 직접 추출한 xdp1 예제를 제시한다.
(1) xdp1_kern.c는 컴파일되어 커널에 로드되는 실제 eBPF 프로그램
(2) xdp1_user.c는 eBPF 프로그램을 커널에 로드하고 맵을 통해 상호 작용하는 사용자 공간 대응물

5.2.1 커널 공간

xdp1_kern.c 파일에는 eBPF 프로그램이 포함되어 있는데,
이 프로그램은 XDP 후크에서 각 패킷을 처리하고 해당 IP 프로토콜 번호를 추출하여 rxcnt라는 CPU별 배열 Map 을 사용하여 프로토콜당 수신되는 패킷 수를 세어 최종적으로 모든 수신 트래픽을 드롭한다.

```
/* Copyright (c) 2016 PLUMgrid
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of version 2 of the GNU General Public
 * License as published by the Free Software Foundation.
 */
#include <uapi/linux/bpf.h>
#include <linux/in.h>
#include <linux/if_ether.h>
#include <linux/if_packet.h>
#include <linux/if_vlan.h>
#include <linux/ip.h>
#include <linux/ipv6.h>
#include "bpf_helpers.h"

struct bpf_map_def SEC("maps") rxcnt = {
    .type = BPF_MAP_TYPE_PERCPU_ARRAY,
    .key_size = sizeof(u32),
    .value_size = sizeof(long),
    .max_entries = 256,
};

static int parse_ipv4(void *data, u64 nh_off, void *data_end) {
    struct iphdr *iph = data + nh_off;

    if (iph + 1 > data_end)


        return 0;
    return iph->protocol;
}

static int parse_ipv6(void *data, u64 nh_off, void *data_end) {
    struct ipv6hdr *ip6h = data + nh_off;

    if (ip6h + 1 > data_end)
        return 0;
    return ip6h->nexthdr;
}

SEC("xdp1")
int xdp_prog1(struct xdp_md *ctx) {
    void *data_end = (void *)(long)ctx->data_end;
    void *data = (void *)(long)ctx->data;
    struct ethhdr *eth = data;
    int rc = XDP_DROP;
    long *value;
    u16 h_proto;
    u64 nh_off;
    u32 ipproto;

    nh_off = sizeof(*eth);
    if (data + nh_off > data_end)
        return rc;

    h_proto = eth->h_proto;

    if (h_proto == htons(ETH_P_8021Q) || h_proto == htons(ETH_P_8021AD)) {
        struct vlan_hdr *vhdr;
```

```
        vhdr = data + nh_off;
        nh_off += sizeof(struct vlan_hdr);
        if (data + nh_off > data_end)
            return rc;
        h_proto = vhdr->h_vlan_encapsulated_proto;
    }
    if (h_proto == htons(ETH_P_8021Q) || h_proto == htons(ETH_P_8021AD)) {
        struct vlan_hdr *vhdr;

        vhdr = data + nh_off;
        nh_off += sizeof(struct vlan_hdr);
        if (data + nh_off > data_end)
            return rc;
        h_proto = vhdr->h_vlan_encapsulated_proto;
    }

    if (h_proto == htons(ETH_P_IP))
        ipproto = parse_ipv4(data, nh_off, data_end);
    else if (h_proto == htons(ETH_P_IPV6))
        ipproto = parse_ipv6(data, nh_off, data_end);
    else
        ipproto = 0;

    value = bpf_map_lookup_elem(&rxcnt, &ipproto);
    if (value)

            *value += 1;

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

첫 번째로 확인해야 할 점은 C 소스 파일이 많은 eBPF 프로그램을 포함할 수 있다는 것이다.
컴파일러가 생성한 ELF 파일에서 이들은 고유한 섹션으로 분리되어 있다.

섹션 레이블은 생성된 개체 파일에 프로그램이 포함될 ELF 섹션의 이름을 컴파일러에게 나타낸다.
이 정보는 커널에 코드를 로드할 때 시스템이 어떤 ELF 섹션을 로드해야 할지 알 수 있도록 하기 위해 필요하다.
섹션 레이블은 Map 과 프로그램 라이센스 선언 시에도 사용되는데, 이들은 모두 고정된 값을 갖는다.
verifier 는 라이센스 섹션을 사용하여 사용자가 사용할 수 있는 Helper 기능을 결정하는데, 이들 중 일부는 GPL 호환 라이센스를 선언하는 프로그램으로 제한된다.

앞의 예와 유사하게 프로그램은 패킷 헤더들을 IPv4 또는 IPv6 헤더들까지 파싱한다. Layer-4 프로토콜 타입을 결정한 후, 프로그램은 lookup 헬퍼 함수(bpf_map_lookup_elem)를 사용하여 해당 프로토콜에 대한 카운터를 검색한다.

이 함수는 Map 에 저장된 현재 값이 존재하는 경우 포인터를 반환하거나, 존재하지 않는 경우 NULL로 반환한다.
이 주소는 Map 업데이트 작업 없이 저장된 데이터를 직접 변경하는 데 사용할 수 있다.
마지막으로 프로그램은 현재 패킷에 대해 XDP Hook 가 취해야 하는 작업을 반환하며, 이 경우 패킷은 항상 XDP_DROP이며 패킷을 폐기해야 함을 나타낸다.

5.2.2 사용자 공간.

커널에 의해 수집되고 맵 rxcnt에 저장된 통계는 xdp1_user.c 파일로 구현된 사용자 공간 응용 프로그램에 의해 쿼리된다.
간략하게 설명하기 위해 아래에서는 이 프로그램의 의미 있는 부분만 강조한다.
먼저 프로그램을 커널에 로드하기 위해 iproute2의 기능을 사용하는 대신, 이 예제에서는 libbpf를 기반으로 사용자 정의 로더를 구현한다.
이 방법은 xdp1_kern.c의 프로그램이 커널에 로드되는 방식에 대한 더 높은 수준의 제어를 산출하며, 이는 user-space 응용 프로그램에서 프로그램 상으로 수정될 수 있다.

이러한 경우 libbpf.h와 bpf.h는 사용자 공간에서 eBPF 시스템과의 상호 작용을 허용하기 위해 포함된다

```
// SPDX-License-Identifier: GPL-2.0-only
/* Copyright (c) 2016 PLUMgrid
 */
#include "bpf/bpf.h"
#include "bpf/libbpf.h"
```

로드될 프로그램 정보는 프로그램 유형, 프로그램을 포함하는 객체 파일 및 그것이 연관되어야 하는 인터페이스 식별자를 포함하는 bpf_prog_load_attr 구조를 통해 전달된다.

```
struct bpf_prog_load_attr prog_load_attr = {
    .prog_type        = BPF_PROG_TYPE_XDP,
};

snprintf(filename, sizeof(filename), "%s_kern.o", argv[0]);
prog_load_attr.file = filename;
```

그런 다음 이 구조는 XDP Hook에 프로그램을 로드하는 데 사용된다.
로드에 성공하면 호출 후 변수 obj 및 prog_fd는 각각 이미 로드된 코드와 그 파일 디스크립터의 상세 정보를 포함한다.
디스크립터는 커널에 현재 로드된 다른 것들로부터 프로그램을 식별하는 데 사용되며,
이는 이 프로그램과의 향후 상호 작용에 필요하다.

```
if (bpf_prog_load_xattr(&prog_load_attr, &obj, &prog_fd))
    return 1;
```

eBPF 프로그램을 커널에 로드한 후에, rxcnt 맵에 대한 참조를 얻는다.
함수 bpf_map_next는 프로그램에서 선언된 맵들의 목록에 대한 iterator를 반환한다.
이 경우 선언된 Map 이 하나뿐이므로, 그 iterator의 값은 그것을 참조하는 파일 디스크립터를 얻는 데 사용될 수 있다.
libbpf.h 라이브러리는 맵 목록에서 이름이나 인덱스로 맵 디스크립터를 얻는 다른 함수들도 제공한다.

```
map = bpf_map__next(NULL, obj);
if (!map) {
    printf("finding a map in obj file failed\n");
    return 1;
}
map_fd = bpf_map__fd(map);
```

마지막으로 eBPF 프로그램은 인터페이스에 연결할 준비가 되어 있으며, user-space 응용 프로그램은 poll_stats() 함수 내부의 무한 루프에 들어갈 수 있다

```
if (bpf_set_link_xdp_fd(ifindex, prog_fd, xdp_flags) < 0) {
    printf("link set xdp fd failed\n");
    return 1;
}

poll_stats(map_fd, 2);
```

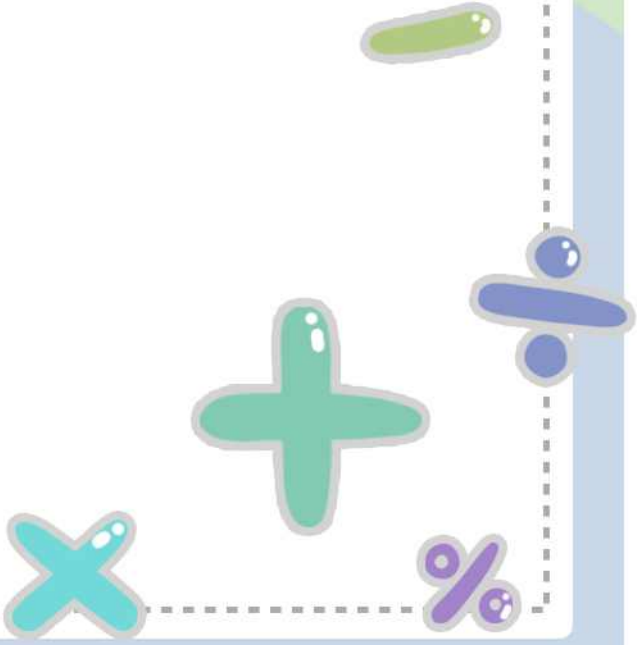
poll 함수 poll_stats()는 map rxcnt의 파일 디스크립터를 사용하여 주기적인 조회를 수행하고 지금까지 계산된 통계와 함께 기존의 모든 엔트리를 나열한다.

```c
static void poll_stats(int map_fd, int interval)
{
    unsigned int nr_cpus = bpf_num_possible_cpus();
    __u64 values[nr_cpus], prev[UINT8_MAX] = { 0 };
    int i;

    while (1) {
        __u32 key = UINT32_MAX;

        sleep(interval);

        while (bpf_map_get_next_key(map_fd, &key, &key) != -1) {
            __u64 sum = 0;

            assert(bpf_map_lookup_elem(map_fd, &key, values) == 0);
            for (i = 0; i < nr_cpus; i++)
                sum += values[i];
            if (sum > prev[key])
                printf("proto %u: %10llu pkt/s\n",
                       key, (sum - prev[key]) / interval);
            prev[key] = sum;
        }
    }
}
```

rxcnt는 CPU별 Array Map 이기 때문에 저장된 데이터는 실제로 여러 CPU에 걸쳐 퍼져 있다.
Helper인 bpf_num_possible_cpus는 프로그램이 사용하는 CPU의 수를 검색하고, 이는 각 CPU(값)의 데이터를 저장할 배열의 크기를 설정하는 데 사용된다.
그러면 무한 루프가 최근에 수집한 통계를 검색하기 위해 Map 전체를 수시로 조회한다.
이것은 한 번에 하나의 Map 키를 생성하는 bpf_map_get_next_key 반복기 함수를 사용하여 수행되며, Map 의 모든 항목을 순서대로 반복할 수 있다.
커널 측 프로그램에서 KEY 는 IP 프로토콜 번호이므로 Map 크기가 선언된(256) 경우 모든 KEY를 통해 반복하는 것은 가능한 모든 IP 프로토콜 번호를 통해 반복하는 것에 해당한다.

bpf_map_lookup_elem의 사용자 공간 버전은 모든 CPU에서 해당 KEY와 관련된 Map 값을 실제로 동시에 읽는 데 사용되며, 이 값은 값 배열의 Helper function 에 의해 저장된다.
이 값들은 전체 통계를 얻기 위해 추가되며, 얻은 값이 마지막에 본 것보다 크면 그 차이가 표준 출력으로 인쇄되어 샘플링 간격 동안 해당 특정 프로토콜 번호를 가진 패킷이 얼마나 많이 수신되었는지 사용자에게 보여준다.

이 예는 사용자와 커널 공간이 eBPF 프로그램과 맵을 통해 상호 작용할 수 있는 방법을 보여준다.
빠른 경로(커널) 상의 모든 데이터 수집 및 패킷 처리는 최소의 최적화된 eBPF 프로그램에 의해 실행되는 반면, Agent 는 느린 경로(사용자 공간) 상에서 주기적으로 데이터를 검색하고 이에 기초한 액션, 예를 들어 사용자에게 Display 할 수 있다.
이는 매우 강력하고 유용한 접근 방식으로 많은 상이한 시나리오에 적용 및 확장될 수 있다.



5.3 Cooperation between XDP and TC

다음 예제는 두 개의 개별 BPF 프로그램으로 구성되어 있는데, 하나는 XDP 계층에 연결되고 다른 하나는 TC에 연결된다.
두 개의 서로 다른 IPv4 주소 사이에서 교환되는 패킷과 바이트 수를 함께 추적하여 이러한 정보를 Map에 저장하고 RX와 TX를 모두 추적한다.
이 예제는 iproute2에서 제공하는 몇 가지 고유한 기능, 사용자 지정 user-space 프로그램 없이 프로그램을 로드하는 방법, 그리고 서로 다른 계층의 프로그램이 지도를 통해 상호 작용하는 방법을 보여준다.

```c
#include <stdbool.h>
#include <stdint.h>
#include <stdlib.h>
#include <linux/in.h>
#include <linux/bpf.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/if_ether.h>
#include <linux/pkt_cls.h>
#include <iproute2/bpf_elf.h>

#include "bpf_endian.h"
#include "bpf_helpers.h"

struct pair {
    uint32_t lip; // local IP
    uint32_t rip; // remote IP
};

struct stats {
    uint64_t tx_cnt;
    uint64_t rx_cnt;
    uint64_t tx_bytes;
    uint64_t rx_bytes;
};

struct bpf_elf_map SEC("maps") trackers = {
    .type = BPF_MAP_TYPE_HASH,
    .size_key = sizeof(struct pair),
    .size_value = sizeof(struct stats),
    .max_elem = 2048,
    .pinning = 2, // PIN_GLOBAL_NS

};

static bool parse_ipv4(bool is_rx, void* data, void* data_end, struct pair *pair){
    struct ethhdr *eth = data;
    struct iphdr *ip;

    if(data + sizeof(struct ethhdr) > data_end)
        return false;

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

    ip = data + sizeof(struct ethhdr);

    if ((void*) ip + sizeof(struct iphdr) > data_end)
        return false;

    pair->lip = is_rx ? ip->daddr : ip->saddr;
    pair->rip = is_rx ? ip->saddr : ip->daddr;

    return true;
}

static void update_stats(bool is_rx, struct pair *key, long long bytes){
    struct stats *stats, newstats = {0,0,0,0};
```

```
58
59      stats = bpf_map_lookup_elem(&trackers, key);
60      if(stats){
61          if(is_rx){
62              stats->rx_cnt++;
63              stats->rx_bytes += bytes;
64          }else{
65              stats->tx_cnt++;
66              stats->tx_bytes += bytes;
67          }
68      }else{
69          if(is_rx){
70              newstats.rx_cnt = 1;
71              newstats.rx_bytes = bytes;
72          }else{
73              newstats.tx_cnt = 1;
74              newstats.tx_bytes = bytes;
75          }
76
77          bpf_map_update_elem(&trackers, key, &newstats, BPF_NOEXIST);
78      }
79  }
80
81  SEC("rx")
82  int track_rx(struct xdp_md *ctx)
83  {
84      void *data_end = (void *)(long)ctx->data_end;
85      void *data = (void *)(long)ctx->data;
86      struct pair pair;
87
88      if(!parse_ipv4(true,data,data_end,&pair))
89          return XDP_PASS;
```

```
90
91      // Update RX statistics
92      update_stats(true,&pair,data_end-data);
93
94      return XDP_PASS;
95  }
96
97  SEC("tx")
98  int track_tx(struct __sk_buff *skb)
99  {
100     void *data_end = (void *)(long)skb->data_end;
101     void *data = (void *)(long)skb->data;
102     struct pair pair;
103
104     if(!parse_ipv4(false,data,data_end,&pair))
105         return TC_ACT_OK;
106
107     // Update TX statistics
108     update_stats(false,&pair,data_end-data);
109
110     return TC_ACT_OK;
111 }
```

표준 C 타입들 이외에도, Map 은 사용자 정의 구조들을 처리할 수도 있다.
예를 들어, 통신 통계가 저장될 trackers Map (27번 라인)에 대해, 구조 쌍 및 구조 통계(15번 라인 및 20번 라인)가 각각 Key 및 Value 로써 여기서 사용된다.
통계는 인터페이스에 의해 보여지는 각각의 고유한 IPv4 주소 쌍에 대해 추적되는 RX 및 TX 패킷 및 바이트 수에 따라 구성된다.
Map 의 정의는 앞의 예제와 다르다. bpf_elf_map 구조는 iproute2에서 사용되므로 헤더 파일 iproute2/bpf_elf.h를 포함하며, 이는 iproute2를 소스에서 설치할 때 시스템에 추가된다.
약간 다른 필드 이름 외에 추가적인 고정 필드도 포함되어 있으며, Map 의 범위를 정의하는 데 사용할 수 있다.
이 필드에 설정된 값은 각각 별도의 복사본이 아니라 두 프로그램이 공유하는 단일 추적기 Map 이

있어야 한다고 결정한다.

이 동일한 소스 파일은 두 개의 서로 다른 ELF 섹션에서 두 프로그램이 사용하는 것을 유지한다.
섹션 rx(81번 라인)에는 XDP 프로그램이 있으며, 이 프로그램은 패킷이 시스템에 도착하자마자 패킷을 처리한다.
parse_ipv4(35번 라인) 함수는 패킷에서 소스 및 IP 주소를 추출하는 데 필요한 모든 구문 분석을 실행한다.
패킷 데이터와 주소 정보를 배치하는 버퍼에 대한 포인터 외에 입력 매개 변수로서 부울 값도 받는다.
이것은 이것이 들어오는 패킷인지 나가는 패킷인지를 나타낸다.
우리는 통신하는 IP의 각 쌍을 추적하기를 원하기 때문에 양방향으로 흐르는 패킷은 맵에서 동일한 엔트리에 의해 추적되어야 한다.
따라서 프로그램은 소스 및 목적지 IP 주소를 로컬 또는 원격으로 해석한다.
RX에서는 목적지 IP가 로컬(호스트 또는 아마도 게스트)인 반면에 소스 IP는 원격이다. TX에서는 그 반대이다.
패킷에서 IP 쌍이 추출되면 함수 update_stats로 전달되어 그에 따라 추적기 Map 을 업데이트한다.
또한 보이는 패킷이 들어오는 것인지 나가는 것인지를 알려주는 매개 변수 is_rx를 수신한다. 함수는 해당 주소 쌍(59번 라인)에 대한 이전 통계를 검색하고 항목이 이미 있는지 확인한다.
그렇지 않은 경우 BPF_NOEXP 플래그(77번 라인)로 bpf_map_update_elem에 대한 호출과 함께 새 항목이 생성된다.
이 경우 호출이 실패하면 프로그램이 할 수 있는 일이 많지 않기 때문에 bpf_map_update_elem의 반환 값은 무시된다.
통계가 null이 아니면 해당 포인터(61-67번 라인)를 사용하여 값이 직접 업데이트된다.
프로그램은 XDP_PASS 코드를 반환하는 것을 종료하여 패킷을 스택으로 정상적으로 전달한다.
TX에는 XDP 계층이 없으므로 NIC로 패킷을 전송하기 직전에 처리할 수 있는 가장 가까운 방법은 TC 계층에 eBPF 프로그램을 연결하는 것이다.

섹션 tx의 프로그램이 바로 그 역할을 한다.
본질적으로 XDP 버전이 RX에 로드되는 것과 같으며 패킷을 처리하기 위해 약간의 변경만 하면 된다.
이 프로그램은 TC에 로드되기 때문에 보조 함수에 전달된 인수 외에도 입력 컨텍스트와 반환 코드가 다르다는 것에 유의해야한다.
이 예에서, 상이한 계층 상의 2개의 프로그램은 동일한 Map 을 채우기 위해 독립적으로 작동한다.
이 동일한 아이디어는 사용자 공간에 더하여 커널 내부의 프로그램들 간의 정보 공유를 가능하게 하도록 확장될 수 있다.

5.4 Other Examples

위의 예들은 eBPF 프로그램의 개발 과정에서 고려되어야 할 몇 가지 기본적인 측면들을 보여준다.
이들은 또한 user-space 프로그램과 상호 작용하는 방법을 보여준다.

리눅스 커널: 다른 유용한 예들은 리눅스 커널의 소스 코드에 의해 제공되며, 두 개의 별개의 디렉터리, 즉 samples/bpf와 tools/testing/selftests/bpf에 위치한다.
둘 다 커널 스택의 다양한 기능과 훅을 사용하는 것을 보여주는 프로그램들을 포함하며, 새로운 프로그램들은 각각의 새로운 버전의 커널에 추가된다.
첫 번째 디렉터리에 있는 대부분의 예들은 두 개의 별개의 파일로 나뉘는데, 하나는 프로그램을 로드하기 위한 사용자 공간 코드를 가지고(user.c로 끝나는 파일), 다른 하나는 커널에 로드하기 위한 eBPF 프로그램의 구현과 함께(kern.c로 끝나는 파일)이다.
일반적으로 이 폴더에 있는 예들은 다양한 작업에 유용할 수 있는 독립 실행형 프로젝트를 나타낸다.
두 번째 디렉터리에 있는 예들은 커널 개발 중에 기능 테스트를 실행하기 위한 기초로 사용된다. 커널에 들어가는 실제 eBPF 프로그램들은 progs/하위 디렉터리 내부에 배치되며, 루트에 있는 나머지 파일들은 이를 로드하는 데 사용되는 사용자 공간 프로그램과 스크립트에 해당한다.

XDP 프로젝트: 커널의 예제를 넘어 공식 XDP 튜토리얼(link 제공)에서도 XDP Hook 의 다양한 기능을 자세히 설명하는 단계별 지침이 포함된 예제를 제시한다.

L4 로드 밸런서: Netronome은 L4 로드 밸런서를 구현하는 l4lb라는 XDP 프로그램의 코드를 제공한다.
이 프로그램은 들어오는 네트워크 패킷을 처리하고 TCP 또는 UDP 포트와 함께 소스 IP 주소를 기반으로 해시 값을 계산하여 동일한 흐름의 모든 패킷이 동일한 서버에서 처리되도록 한다.
생성된 해시는 eBPF Map에서 Key 로 사용된다. 이 eBPF Map 은 프로그램이 패킷을 리디렉션할 수 있는 사용 가능한 서버의 주소로 채워진다.
이 프로그램은 Map의 데이터와 함께 외부 IP 헤더를 확장하고 삽입한다.

그런 다음 패킷은 해당 서버로 전달된다. 이 프로그램은 SmartNIC으로 오프로드되어 CPU 사이클을 절약할 수도 있다.

프로그래밍 가능 수신 측 스케일링(RSS): RSS를 사용하면 CPU 코어에 의해 처리될 수신 패킷을 매핑할 수 있다. 이것은 멀티프로세서 시스템에서 여러 CPU에 걸쳐 네트워크 트래픽을 분산시키는 데 중요하다. RSS 기술은 많은 네트워크 어댑터에서 여러 큐의 사용을 통해 패킷의 계산을 별개의 CPU 세트에 분산시키는 데 사용된다. 그러나 구현은 일반적으로 독점적이고 하드웨어 기반이므로 프로그래밍 가능성이 거의 또는 전혀 허용되지 않는다.
eBPF 프로그램을 사용하면, 패킷 분배는 Map 값을 통해 또는 로드된 eBPF 프로그램의 완전한 대체를 통해 필요에 따라 수정될 수 있다.

6.0 TOOLS

6.1 iproute2

iproute2는 커널의 네트워크를 제어, 구성 및 모니터링하기 위한 사용자 공간 도구들의 집합이다. ip와 tc는 이러한 도구들의 예이다.
둘 다 libpf 라이브러리나 bpf 시스템 호출을 사용하는 user-space 프로그램 없이 eBPF 코드를 커널에 로드하는 대안적인 방법들을 제공한다.
ip 도구와 tc 도구를 모두 사용하는 방법에 대한 예는 이전에 이 본문에서 보여주었다.
또한 iproute2는 eBPF 시스템과 상호 작용할 수 있는 고유한 인터페이스를 가지고 있으며, eBPF Map 할당 범위를 지정하는 기능과 같은 추가 기능을 제공한다.
불행히도 추가적인 기능으로 인해 libpfl 로더와 호환되지 않는다.
예를 들어 iproute2를 사용할 때 Map 은 iproute2/bpf_elf.h로 정의되는 대체 구조(bpf_elf_map)를 사용하여 선언된다. 예시적인 사용법은 다음과 같다

```
#include <linux/bpf.h>
#include <iproute2/bpf_elf.h>

struct bpf_elf_map SEC("maps") src_mac = {
    .type = BPF_MAP_TYPE_HASH,
    .size_key = 1,
    .size_value = 6,
    .max_elem = 1,
    .pinning = PIN_GLOBAL_NS,
};
```

이 구조는 libppf의 bpf_map_def와 비슷하지만 지도의 범위를 정의하는 데 사용되는 Pinning 과 같은 여분의 필드를 가지고 있다.
따라서 iproute2는 libpf 프로그램을 로드할 수 있지만 libpf가 여분의 기능을 처리하는 방법을 모르기 때문에 그 반대는 사실이 아니다.

6.2 bpftool

bpftool 은 eBPF 프로그램을 로드하고, Map 을 만들고 조작하며, eBPF 프로그램과 Map 에 대한 정보를 수집할 수 있는 user-space 디버그 유틸리티이다.
리눅스 커널 트리의 일부이며, tools/bpf/bpftool에서 사용할 수 있다.
여기서 우리는 그것의 일부 기능만을 제시한다.

• 로드된 프로그램을 다음 명령으로 나열할 수 있다
bpftool prog show
• 특정 프로그램의 지침을 인쇄하려면 다음을 사용한다
bpftool prog dump xlated id <id>
• 런타임에 Map의 내용을 나열하고 인쇄할 수 있다.
bpftool map
bpftool map dump id <map id>

• 또한 프로그램 로딩, 검색 수행 또는 Map 값 업데이트를 포함한 일부 관리 작업을 수행할 수 있다.
마지막 항목에 대한 예는 다음과 같다
bpftool map update id 1234 key 0x01 0x00 0x00 0x00 value 0x12 0x34
0x56 0x67

6.3 llvm-objdump

llvm-objdump 툴(버전 4.0 이상)은 컴파일된 바이트 코드를 사용자가 커널에 주입하기 전에 사람이 읽을 수 있는 포맷으로 변환하는 디셈블러를 제공한다.
또한 컴파일된 eBPF 파일의 ELF 섹션을 검사하는 데 유용합니다.
아래 명령어는 disassembler를 사용하는 방법을 보여준다.

```
$ llvm-objdump -S dropworld.o
```

6.4 BPF Compiler Collection

오픈 소스 프로젝트 BPF 컴파일러 모음(BCC)은 eBPF 프로그램의 개발을 용이하게 하는 것을 목표로 한다.
Python, Go와 같은 상위 언어를 사용하여 eBPF 시스템과 상호 작용하는 데 사용할 수 있는 프론트엔드 세트를 제공한다.
프로젝트에는 운영 체제에서 다양한 작업을 수행할 수 있는 BCC를 사용하여 구축된 일련의 예시적인 도구들도 있다.
이 도구들은 응용 프로그램에 의한 시스템 호출 횟수, 디스크 판독 시 경과된 시간 등의 작업을 수행할 수 있다. 이들은 eBPF 프로그램을 기반으로 하므로, 낮은 추가 오버헤드로 실제 생산 시스템을 분석하는 데 사용될 수 있다

7.0 PLATFORMS

eBPF 명령어 세트를 사용하여 리눅스 커널을 넘어 다양한 환경에 프로그래밍 가능성을 추가하는 여러 플랫폼들이 있다. 이 절에서는 소프트웨어 또는 하드웨어로 구분하여 가장 중요한 것을 제시한다.

7.1 Software

7.1.1 리눅스 커널

앞서 살펴본 바와 같이 리눅스 커널은 eBPF의 원조로서 가장 대중적인 플랫폼으로서 가장 활발한 개발이 이 본문의 주된 초점이다.
커널 내부의 eBPF 애플리케이션들은 네트워크 처리에만 국한되지 않고 커널 instrumentation 및 모니터링에도 적용될 수 있다.
따라서 오늘날 eBPF 프로그램은 리눅스 introspectation 및 성능 분석을 수행하기 위한 수입 도구 세트를 나타내며 IOVisor와 같은 중요한 오픈 소스 프로젝트는 이 전면에서 많은 솔루션을 제공한다.

7.1.2 Userspace BPF(uBPF)

uBPF 는 eBPF 프로세서를 사용자 공간에서 실행되도록 적응시키는 오픈 소스 프로젝트이다.
uBPF 프로젝트는 eBPF 인터프리터와 JIT 컴파일러의 복사본을 가지고 있어
사용자가 user-space에서 실행되는 다른 프로젝트에 eBPF 머신을 내장할 수 있다.
uBPF 소스 코드를 활용함으로써 eBPF를 사용한 프로그래밍 가능성을 다른 도구와 환경에 쉽게 추가할 수 있다.
uBPF와 유사하게 프로젝트 rBPF는 Rust 에서 eBPF VM의 대체 구현을 제공한다. uBPF는 사용자 공간에서 실행되기 때문에 언롤링 루프를 필요로 하는 것과 같은 eBPF Verifier 에 의해 부과되는 제한이 없다.
그러나 커널의 eBPF 시스템과 달리 uBPF는 Map에 대한 네이티브 지원을 제공하지 않으며
어떤 Helper 기능도 구현되지 않는다.
그럼에도 불구하고, 다음에 논의되는 BPFabric 프로그램 가능 가상 스위치의 일부로 사용한
[https://doi.org/10.1109/ANCS.2017.14] 에 의해 수행된 바와 같이 이러한 기능을 지원하도록 쉽게 확장될 수 있다.

7.1.3 BPFabric.

OpenFlow의 한계를 처리하기 위해 [https://doi.org/10.1109/ANCS.2017.14]는 BPFabric이라고 하는 새로운 SDN 아키텍처를 제안한다. BPFabric은 변형된 버전의 uBPF를 사용하여 데이터 평면이 eBPF 명령으로 프로그램되도록 하는 스위치 아키텍처이다.
eBPF 프로그램은 동적으로 수정될 수 있으므로 임의 프로토콜의 파싱 및 eBPF Map 을 사용하여 상태를 저장할 수 있다.
새로운 기능에 지원을 추가하고 원격 측정, 통계 수집 및 패킷 추적과 같은 서비스를 제공하기 위해 새로운 Helper 기능이 개발될 수 있다.
제어 평면은 Southbound API를 통해 데이터 평면과 통신하는 에이전트를 호스팅한다.
Agent 는 (i) 스위치의 동작 변경, (ii) 패킷 수신, (iii) 이벤트 보고, (iv) 테이블 엔트리 읽기 및 업데이트를 담당한다.
Agent 는 컨트롤러로부터 컴파일된 코드를 수신하면 eBPF ELF Loader를 사용하여 스위치 파이프라인을 수정한다. eBPF Loader는 코드를 확인하기 위해 Verifier 를 호출한다.
성공 시 필요한 eBPF 테이블의 할당을 수행하고 수신된 바이트 코드를 스위치별 형식으로 변환해야 한다.
패킷을 수신하면 eBPF 프로세서는 이전에 프로그래밍된 eBPF 명령어를 패킷에 실행한다. 파이프라인이 끝나면 패킷을 드롭하고 컨트롤러로 전송하며 어떤 출력 포트로 전달하거나 플러딩하는 라우팅 결정을 반환한다.

7.2 Hardware

7.2.1 Netronome SmartNICs

스마트 네트워크 인터페이스 카드(SmartNICs)는 다양한 동작 모드를 구현하기 위해 런타임에 그 기능을 수정할 수 있는 네트워크 장치이다.
이 장치들은 단순히 연결과 기본적인 계층-2 및 물리 계층 처리를 제공하는 대신 전용 코어와 메모리를 사용하여 사용자 정의 패킷 처리를 실행하여 컴퓨터의 CPU에서 많은 주기를 해방시킬 수 있다.
(논문 저자 피셜)현재 시장에서 이런 종류의 제품을 제공하는 유일한 회사는 Netronome 이다.

네트워크 관리자는 예를 들어 P4 또는 eBPF에서 패킷 처리 루틴을 정의하고 이러한 프로그램을 Netronome 카드에 로드할 수 있으며, Netronome 카드는 패킷 수신 및 전송 시 제공되는 코드를 실행한다.
서로 다른 펌웨어 버전은 서로 다른 언어를 지원하며, 이는 서버에서 장치를 재부팅하거나 제거하지 않고 SmartNIC에 로드할 수 있다.
해당 펌웨어 버전에서는 TC 및 XDP 계층에서 실행되는 eBPF 프로그램을 성능 향상을 위해 이러한 장치로 원활하게 오프로딩할 수 있다. 오프로딩을 수행하는 데 필요한 대부분의 드라이버와 코드는 이미 업스트림 커널의 일부이다.
뛰어난 처리 능력을 감안할 때 SmartNIC은 지연 시간이 짧고 속도가 빠른 워크로드를 위한 좋은 대안이 되었다. 일부 기능은 하드웨어에서 직접 구현할 수 있으므로 운영 체제의 네트워크 스택을 올라갈 필요 없이 패킷을 수정하여 네트워크로 다시 보낼 수 있다. 또한 프로그램을 수정하고 즉시 업데이트할 수 있다. 이러한 장치의 일반적인 사용 사례는 초기 패킷 필터링, 속도 제한기, DDoS 완화 작업, 로드 밸런싱, RSS 작업, 패킷 스위칭 등이다.

8.0 PROJECTS WITH eBPF

상당히 최근이지만 eBPF는 이미 많은 그룹에서 관심 조사 및 업계 주도 프로젝트에 전력을 공급하기 위해 사용되었다. 이들은 작업에서 생산 시스템 운영을 지원하는 것에 이르기까지 새로운 네트워크 서비스를 제공하는 기술에 이르기까지 다양하다. 이 절에서는 eBPF에 가능한 광범위한 적용 범위를 보여주기 위해 이들 중 일부에 대해 논의한다.

더 큰 네트워크 프로그램 가능성에 대한 최근의 요구는 세그먼트 라우팅(segment routing)과 같은 기술의 출현으로 이어졌다.
이 기술은 네트워크 관리자가 네트워크의 특정 지점에서 패킷에 대해 수행할 상이한 액션을 지정할 수 있게 한다. 소스 라우팅 헤더(SRH)라고 불리는 특수 필드를 갖는 MPLS 라벨 또는 IPv6 프로토콜을 사용하여, 각각의 패킷은 세그먼트라고 불리는 라우팅 및 처리 액션의 순서화된 목록으로 캡슐화된다.
패킷이 네트워크를 통해 이동될 때, 인에이블된 디바이스는 세그먼트 목록을 처리하고 그에 의해 지정된 액션을 수행한다.
이는 예를 들어, 상이한 네트워크 기능의 구현을 허용한다.
리눅스 커널은 버전 4.10 이후로 이미 IPv6를 통한 세그먼트 라우팅을 지원하지만, 라우팅 및 레이블이

지정된 패킷의 송수신과 같은 몇 가지 처리 옵션만 있다.

[https://doi.org/10.1145/3234200.3234213]는 eBPF 프레임워크를 사용하여 새로운 세그먼트의 생성 및 사양을 보다 일반적이고 유연하게 만들었다.
새로운 Helper 기능의 추가를 통해 저자는 eBPF 프로그램 형태의 세그먼트 구현을 허용하도록 리눅스 커널을 확장했다. 따라서 새로운 네트워크 기능을 쉽게 개발하고 리눅스 라우팅 규칙과 연관시킬 수 있어 IPv6 환경을 통한 세그먼트 기반 라우팅과 통합하기가 더 쉬워진다.
다른 컨텍스트에서 데이터 평면에 프로그램 가능성을 추가하기 위해 eBPF 프레임워크를 사용하는 여러 다른 프로젝트가 있다. InKeV는 가상 데이터 센터 네트워크의 데이터 경로를 수정하기 위해 eBPF 프로그램을 사용하는 네트워크 가상화 플랫폼이다.

OpenFlow 고정 매칭 문제를 해결하기 위해,
[https://doi.org/10.1109/HPSR.2015.7483106]은 다양한 매칭 필드를 제공하기 위해 eBPF 프로그램을 사용할 것을 제안한다. [https://doi.org/10.1145/3139645.3139657]는 eBPF를 사용하여 Open vSwitch 가상 스위치에 대한 확장 가능한 데이터 경로 아키텍처를 생성하는 데 이 기술을 사용하는 iptables 도구의 새로운 버전을 제안한다.
IoT 시나리오와 엣지 컴퓨팅 패러다임에서 센서로부터 수집된 데이터는 여러 개체에 의해 요청될 수 있다. 이 경우, 정보는 복제(dupplication)된다.

[https://doi.org/10.1109/INFCOMW.2018.8407006]에서, eBPF 프로그램은 패킷 복제 동작을 제어한다.
일부 회사는 이미 운영 환경에서 eBPF를 사용하는데, 예를 들어 클라우드플레어는 서비스 거부 공격 완화, 로드 밸런싱을 위해 XDP를 사용하고 소켓 필터링 및 디스패치를 위해 상위 계층에서 eBPF를 사용한다. 또 다른 예는 Katran이라고 불리는 eBPF를 기반으로 L4 로드 밸런서를 개발한 Facebook에서 나온다. 또한 넷플릭스는 성능 모니터링 및 시스템 프로파일링을 위해 eBPF를 사용해 왔다.

9.0 LIMITATION AND WORKAROUNDS

eBPF는 빠른 패킷 처리와 커널 프로그램 가능성을 위한 강력한 기술이다. 그러나 커널 내부에서 실행하기 위해서는 시스템 안정성 및 보안을 보장하기 위해 eBPF 프로그램에 일부 제한이 적용된다. 이 절에서는 일부 eBPF 제한 사항과 이를 극복하기 위한 해결 방법에 대해 논의한다.
여기에 설명된 일부 해결 방법은 [https://github.com/xdp-project/xdp-project]에서 제공되는 XDP 프로젝트 개발 저장소에서 찾을 수 있다.

9.1 Subset of C Language Libraries

eBPF는 제한된 수의 C언어 라이브러리를 사용하며 외부 라이브러리와의 연산을 지원하지 않는다.
이러한 한계를 극복하기 위한 대안은 보조 기능을 정의하고 사용하는 것이다.
예를 들어, eBPF 프로그램은 커널 내부에서 실행되고 eBPF에서 이 기능이 구현되지 않기 때문에 printf 기능을 사용할 수 없다. 그러나 bpf_trace_printk() Helper 기능을 사용할 수 있는데, 이는 eBPF 프로그램에서 생성된 로그 메시지를 커널 트레이스 폴더(/sys/kernel/debug/trace/trace)에 사용자 정의 출력에 따라 저장한다. 사용자는 생성된 로그를 사용하여 eBPF 프로그램의 실행에서 발생할 수 있는 오류를 분석하고 찾을 수 있다.

9.2 Non-static Global Variables

현재 eBPF 프로그램은 정적 전역 변수만을 지원한다. 대안은 BPF_MAP_TYPE_PERCPU_ARY Map을 사용하는 것이다. 이 Map은 프로그램 실행 중 한 번의 엔트리로 임시 데이터를 저장하는 데 사용할 수 있는 사용자 정의 크기의 비공유 메모리 공간을 예약한다.

9.3 Loops

eBPF Verifier 는 루프가 모든 프로그램을 확실히 끝내는 것을 허용하지 않았다.
이 제한을 우회하기 위해 채택된 첫 번째 기술은 clang 컴파일러의 루프 언롤링 명령어를 사용하여 루프를 독립적인 명령어들의 반복되는 시퀀스로 다시 쓰는 것이었다.
이 기술은 컴파일 시 반복 횟수를 결정할 수 있을 때만 사용할 수 있기 때문에 이 제한을 부분적으로 해결한다. 그 외에도 최종 프로그램의 명령어 수가 증가하는 부작용이 있다.

아래 코드 스니펫은 컴파일러에게 루프를 위해 언롤링하도록 하는 방법을 보여준다:

```
#pragma clang loop unroll (full)
for (int i=0; i<8; i++) { ... }
```

9.3.1 유계 루프.

커널 버전 5.3에서는 언롤 루프 디렉티브의 사용이 유계 루프로 대체되었다. 이 릴리스 이후로 Verifier는 루프가 타임아웃 내에 종료되는지 여부를 먼저 확인하지 않고 루프를 포함하는 모든 eBPF 프로그램을 거부하지 않는다.
유계 루프는 존 패스트벤드(John Fastabend)에 의해 제안되었으며 2018년 리눅스 배관공 회의에서 BPF 마이크로 컨퍼런스에서 발표되었다.
이 기법을 사용하면 Verifier 를 통해 모델링한 간단한 루프를 사용할 수 있다.
루프의 동작을 유도 변수를 통해 분석하고 유도 변수를 사용한 메모리 접근이 메모리 주소의 범위에 속하는지 확인한다.

Verifier 상의 경계 루프에 대한 구현 세부 사항은
[https://lwn.net/ml/netdev/20180601092646.15353.28269.stgit@john-Precision-Tower-5810/.
]에서 확인할 수 있다.

9.4 Limited Stack Space

C 프로그램의 로컬 변수는 eBPF 프로그램으로 변환된 후 스택에 저장된다.
스택 공간이 단지 512 Byte 로 제한되므로, 변환 후 프로그램의 모든 로컬 변수를 저장하기에 충분하지 않을 수 있다.
채택된 해결 방법은 글로벌 변수에 대해 동일하게 사용된다:
스택 공간이 충분하지 않을 때 BPF_MAP_TYPE_PERCPU_ARY를 일부 로컬 변수를 저장하기 위한 보조 버퍼로 사용하는 것이다.

9.5 Complex Applications

[47]은 eBPF로 복잡한 네트워크 기능을 구현할 때 직면하는 과제에 대한 심층적인 논의를 제공한다.
예를 들어, 동일한 패킷을 여러 인터페이스로 전송해야 하는 애플리케이션(예를 들어, 스위치 상의 플러드 동작)은 프로그램이 모든 인터페이스를 루프로 통과하고 패킷을 각 인터페이스에 복사해야 하기 때문에 구현하기가 어렵다.

이는 현재 이용 가능한 eBPF 메커니즘을 고려할 때 어려운 작업이다. 게다가, eBPF 프로그램은 후크(hook)와 연관되기 때문에, 수동적 이벤트 기반 모델을 따른다.
이것은 예를 들어, user-space 애플리케이션에 의해 수행되어야 할 능동적 네트워크 측정과 같은 비동기 작업을 수행하는 것을 어렵게 한다.

더욱이, eBPF 프로그램을 위한 제어 평면을 구현하기 위한 기존 도구(예를 들어, libbpf 및 bpf syscall)는 매우 기본적이며, 주로 데이터 교환을 위해 Map 구조에 의존한다.
그러나, 실리움 및 BCC와 같은 프로젝트는 그 점에서 진전을 나타낸다.

마지막으로, eBPF 프로그램의 최대 명령어 수는 4096개로 제한되어 있었으며, 이로 인해 복잡한 네트워크 기능의 개발이 어려웠다. 이 한계를 극복하기 위한 한 가지 방법은 프로그램을 여러 개의 하위 프로그램으로 분할하고 bpf_tail_call() 도우미 기능을 사용하여 한 하위 프로그램에서 다른 하위 프로그램으로 점프하는 것이었다.
이 기술은 느슨하게 결합된 모듈의 집합으로서 네트워크 서비스의 개발을 가능하게 하며, 각 모듈은 한 모듈에서 다른 모듈로 점프할 때 낮은 오버헤드로 다른 기능(분석, 분류 또는 필드 수정)을 수행한다.
그러나 허용되는 중첩 테일 호출의 최대 수는 32개이다.

2019년 4월, 최대 명령어 수가 4096개에서 100만 개로 증가하여 테일 호출없이 더 큰 eBPF 프로그램을 실행할 수 있게 되었다.

[47: complex network service with ebpf: Experience and lessons learned. In High Performance Switching and Routing (HPSR)]
[1:https://git.kernel.org/pub/scm/linux/kernel/git/davem/net-next.git/commit/?id=c04c0d2b968ac45d6ef020316808ef6c82325a82]

10.0 COMPARISON WITH SIMILAR TECHNOLOGIES

eBPF는 네트워킹 플랫폼에 프로그래밍 가능성을 추가하는 강력한 도구이며 XDP Hook를 통해 리눅스 환경에서 빠른 패킷 처리를 달성할 수 있는 대안이다.
이 절에서는 eBPF를 P4, Click, 넷맵, DPDK 등 널리 사용되는 일부 기술과 비교하여 프로그래밍 가능하고 고속 데이터 평면 환경에 어떻게 적합한지 논의한다

10.1 Programmable Data Planes

P4 언어는 프로그래밍 가능한 스위치를 위한 맞춤형 데이터 평면의 정의를 가능하게 하도록 제안되었다.
널리 채택됨에 따라 P4 지원 장치는 완전히 프로그래밍 가능한 코어 네트워크를 구성하여 물리적 스위치와 가상 스위치 모두의 기능을 정의할 수 있다.
동일한 전면에서 BPFabric은 eBPF를 언어로 사용하여 가상화 환경 또는 심지어 서버 중심 네트워크에 적용 가능한 프로그래밍 가능한 가상 스위치의 동작을 정의한다.
그러나 eBPF의 주요 강점은 통신 가장자리에 대한 응용에 있다.
리눅스 커널에서 P4용으로 설계된 것과는 다른 사용 사례인 통신 엔드포인트에서 데이터 평면 기능을 정의할 수 있다.
스위칭 및 라우팅은 P4로 덮인 코어 네트워크에 의해 수행되지만 네트워크 프로토콜 스택의 상당 부분이 엔드포인트에서 구현된다.
eBPF를 사용하면 이 나머지 부분을 모니터링, 수정 및 재구성할 수 있으므로 전체 네트워크 프로그래밍 가능성을 제공하는 데 필수적인 도구가 된다.
[37]은 또한 리눅스 커널 내부에서 패킷을 처리하기 위한 맞춤형 네트워크 요소의 생성을 가능하게 한다. 애플리케이션은 요소라고 불리는 많은 구성 요소의 구성을 통해 생성될 수 있으며, 이는 클릭 특정 함수 호출을 사용하여 C++ 코드로 구현될 수도 있다.
클릭 응용 프로그램의 정의는 사용되는 요소의 종류와 패킷 처리 그래프를 나타내는 상호 연결을 지정하는 구성 파일을 통해 이루어진다.
이 사양은 이후 user-space 또는 커널로 컴파일될 수 있다.
클릭 커널 모듈은 네트워크 장치에 가까운 패킷을 캡처하고 eBPF를 사용한 XDP Hook 에서 수행할 수 있는 것과 유사한 ToHost 요소를 사용하여 커널 스택으로 전달할 수 있다.

[https://doi.org/10.1145/2934872.2934897] build on 클릭을 통해 FPGA에서 네트워크 기능을 생성하고 실행하여 모든 패킷 크기에 대해 40Gbps 처리량을 달성할 수 있는 ClickNP 프레임워크를 만든다.
클릭 앤 클릭 NP는 라우터와 네트워크 기능을 생성하기 위해 미리 구축된 더 넓은 프리미티브 세트를 제공하지만 eBPF와 같은 수준의 커널 스택과의 통합을 제공하지는 않는다.
후자는 커널 스택의 여러 계층과 상호 작용할 수 있도록 프로그램을 첨부하여 커널의 패킷 처리 메커니즘에 대한 더 높은 수준의 제어를 제공한다.
그러나 클릭은 eBPF/XDP 인프라스트럭처를 패킷 처리 기능과 상호 작용할 수 있는 기반으로 활용하기 위해 수정될 수 있으며, 표현성을 eBPF의 성능 및 네이티브 커널 통합과 결합할 수도 있다.

[37 : The Click modular router. ACMTransactions on Computer Systems (TOCS)]

10.2 High-speed Packet Processing

10Gbps 이상의 링크를 가진 고속 네트워크에서는 패킷 간 도달 시간이 수십 나노초 정도로 낮아져 각 패킷을 처리하는 데 매우 적은 시간을 할애할 수 있다.
이러한 엄격한 요구 사항으로 인해 컨텍스트 스위치 및 인터럽트 처리와 같은 일반적인 시스템 작업에 너무 많은 비용이 소요된다.

DPDK(Data Plane Development Kit) 및 NetMap과 같은 잘 알려진 기술은 폴 모드로 작동하고 커널을 완전히 우회하여 사용자 공간에서 모든 패킷 처리를 수행함으로써 이 문제를 처리한다.
특수 드라이버를 사용하여 NIC에 의해 수신된 패킷은 사용자 공간 응용 프로그램으로 직접 전송되고 이 응용 프로그램은 이를 처리한다. 또한 DPDK 라이브러리 및 이와 함께 구축된 응용 프로그램은 일반적으로 정렬 메모리 액세스, 다중 코어 처리, 불균일 메모리 액세스(NUMA) 및 귀중한 CPU 사이클을 절약하기 위한 기타 최적화에 최적화되어 있다.

커널의 eBPF 시스템은 다른 방식으로 우수한 성능을 제공하는데,
이는 가장 낮은 커널 계층인 XDP Hook 에서 맞춤형 패킷 처리를 허용함으로써 가능하다. 이 Hook 를 통해 eBPF 응용 프로그램은 OS의 네트워크 스택을 거치지 않고도 들어오는 패킷을 파싱, 수정, 통계 수집 및 조치를 취해 패킷을 네트워크로 직접 전송할 수 있다.

이러한 방식으로, 모든 네트워크 처리를 커널에 임베딩함으로써 컨텍스트 스위치를 피할 수 있다.
DPDK에 비해 XDP Hook 의 장점 중 하나는 커널에 존재하는 기존 네트워크 설비(예: 라우팅 테이블)를 사용할 수 있다는 것인데, 이는 DPDK의 경우와 같이 사용자 공간에서 처리할 때 처음부터 다시 구현해야 한다. 또한 XDP는 커널에서 처리되기 때문에 예외 경로를 통해 패킷을 다시 주입할 필요 없이 기존 스택과 통합하여 보안을 강화하기 위해 이미 존재하는 분리 메커니즘을 통해 커널 API 안정성을 보장할 수 있다.

프로그램이 OS에 표시된 상태로 유지될 수 있는 NIC를 완전히 제어할 필요가 없기 때문에 디바이스 공유도 용이하다. 동시에 OS의 추상화 계층 아래에 처리가 숨겨져 있기 때문에 프로그램은 호스트의 다른 애플리케이션에 대해 투명하다. 리소스 사용 측면에서 이벤트 기반 특성 때문에 비지 폴링으로 귀중한 CPU 주기를 투자하는 것을 피한다. 최근 두 기술 간의 성능 비교에 따르면 DPDK는 여전히 더 높은 대역폭(코어 5개로 패킷을 드롭할 때 XDP의 경우 100 Mpps 대비 115 Mpps)에 도달하지만 XDP[31]보다 훨씬 높은 CPU 사용률을 희생한다.

마지막으로 프로그램을 원자적으로 대체할 수 있어 패킷 처리에 대한 온디맨드 변경에 대한 더 큰 유연성을 제공한다. 따라서 eBPF와 DPDK는 고속 패킷 처리에 대한 다른 접근 방식을 제공한다.
AF_XDP 소켓과 DPDK의 해당 폴 모드 드라이버를 사용하거나 DPDK가 제공하는 librte_bpf 라이브러리를 사용하여 둘 다 통합할 수 있지만 실제 사용 사례에 따라 최선의 기술 선택이 달라질 수 있다.

II.0 CONCLUSION

본 연구에서는 eBPF와 XDP를 이용한 고속 패킷 처리와 관련된 이론적, 실무적 측면의 비전을 제시하였다. 이론적 부분에서는 BPF와 eBPF 기계, 리눅스 커널에서 제공하는 eBPF 시스템의 개요, 사용 가능한 후크 및 최근 연구의 일부 결과에 대해 논의하였다. 실무적 부분에서는 eBPF와 XDP Hook 에 초점을 맞추어 예제를 제공하고 기존 도구를 보여주었다.
빠른 패킷 처리에 대한 그들의 힘을 감안할 때, 우리는 eBPF와 XDP를 사용한 새로운 연구 프로젝트의 개발에서 새로운 네트워크 기능의 개발을 위한 도구로서 또는 새로운 통신 표준 및 프로토콜의 생성과 같은 데이터 평면에서의 새로운 기능을 제공하거나 새로운 연구 프로토타입 및 네트워크 솔루션의 개발에서 큰 잠재력이 있다고 생각한다.
eBPF와 XDP는 함께 컴퓨터 네트워킹 영역에서 큰 잠재력을 가진 새로운 흥미로운 연구를 개발하는 데 도움이 될 것이다.