논문 번역본

Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications

- '[]' : 설명 보충

## Intro.

확장 버클리 패킷 필터(EBPF)는 리눅스 커널 내부의 명령어 세트이자 실행 환경이다.
런타임에 수정, 상호 작용 및 커널 프로그램 가능성을 가능하게 한다. eBPF는 빠른 패킷 처리를 위해 NIC에 더 가깝게 패킷을 처리하는 커널 네트워크 계층인 eXpress Data Path(XDP)를 프로그램하는 데 사용될 수 있다. 개발자는 C 또는 P4 언어로 프로그램을 작성한 다음 eBPF 명령어로 컴파일할 수 있으며, 이는 커널 또는 프로그램 가능한 장치(예: SmartNIC)에 의해 처리될 수 있다. eBPF는 2014년에 도입된 이후 페이스북, 클라우드플레어, 넷로놈과 같은 주요 기업들에 의해 빠르게 채택되었다.
사용 사례에는 네트워크 모니터링, 네트워크 트래픽 조작, 로드 밸런싱, 시스템 프로파일링 등이 포함된다. 이 작업은 eBPF와 XDP의 주요 이론적이고 근본적인 측면을 다루는 비전문가 청중에게 eBPF를 제시할 뿐만 아니라 두 기술의 일반적인 작동과 사용에 대한 통찰력을 제공하기 위해 간단한 예를 독자에게 소개하는 것을 목표로 한다.
CCS 개념: • 네트워크 → 프로그래밍 인터페이스; 중간 상자 / 네트워크 어플라이언스; 엔드 노드; 추가 주요 단어 및 구문: 컴퓨터 네트워킹, 패킷 처리, 네트워크 기능

## 1 INTRODUCTION

인터넷 트래픽의 증가와 데이터 센터 네트워크에서 제공되는 서비스의 복잡성으로 인해 그 어느 때보다 높은 패킷 처리 속도가 요구되고 있다. 또한 서비스 요구의 역동성으로 인해 네트워크는 적절한 수준의 서비스 품질을 유지하고 사용 가능한 자원을 효율적으로 사용하기 위해 빠르게 적응해야 한다. 그러나 컴퓨터 네트워크는 전통적으로 정적인 방식으로 개발되어 통신 프로토콜 구현을 네트워크 장치의 하드웨어에 내장하여 현재 요구에 적응하는 것이 어렵다. 최근 몇 년 동안 네트워크에 더 많은 프로그래밍 가능성을 추가하기 위한 몇 가지 제안이 제안되었다.
그중에서 SDN[28, 43] 및 NFV[48] 패러다임, POF[61], P4[14], 그리고 보다 최근에는 확장된 버클리 패킷 필터(eBPF) 및 eXpress Data Path(XDP) 리눅스 커널 네트워크 계층을 강조할 수 있다. 이 작업에서는 주제에 대한 배경이 독자에게 거의 또는 전혀 없다고 가정하여 eBPF 및 XDP를 제시한다. eBPF는 리눅스 커널 내부에 명령어 세트와 실행 환경을 제공한다. 커널에서 패킷 처리를 수정하는 데 사용되며 네트워크 장치의 프로그래밍도 허용한다. 개발자는 제한된 C 언어로 애플리케이션을 작성한 다음 코드를 eBPF 명령어로 컴파일한다. 결과적으로 생성된 eBPF 코드는 커널에서 또는 SmartNIC과 같은 프로그래밍 가능한 장치에 의해 처리될 수 있다.
XDP는 리눅스 네트워크 스택의 최하위 계층이다[31].
이것은 개발자들이 패킷을 처리하는 프로그램들을 리눅스 커널에 설치할 수 있게 해준다. 이러한 프로그램들은 들어오는 패킷마다 호출된다. XDP는 빠른 패킷 처리 애플리케이션들을 위해 설계되는 동시에 프로그래밍 가능성을 향상시킨다. 게다가 커널 소스 코드를 수정하지 않고도 이러한 프로그램들을 추가하거나 수정하는 것이 가능하다.
eBPF 프로그램은 커널의 재컴파일을 필요로 하지 않고 런타임에 (프로그래밍 가능한) 커널 연산을 수정한다.
eBPF와 XDP의 중요성은 업계와 학계 모두에서 2014년 리눅스 커널에 도입된 이후 빠른 채택으로 강조된다.
이들의 사용 사례는 네트워크 모니터링, 네트워크 트래픽 처리, 로드 밸런싱, 운영 체제 인사이트 등의 작업을 포함할 정도로 빠르게 성장했다. 이미 페이스북[26], 넷로놈[10], CloudFlare[11] 등의 프로젝트에 eBPF를 사용하고 있는 회사들도 여럿 있다.

우리는 이 튜토리얼을 다음과 같이 구성했다. 이 섹션의 나머지 부분은 독자에게 원래 BPF를 소개한다.
1.2절에서는 eBPF 기계의 아키텍처에 대해 설명한다. 2절에서는 eBPF 시스템에 대해 설명한다. 3절에서는 eBPF의 구조, 사용 가능한 프로그램 유형, 사용 가능한 Map 및 사용 방법, 사용 가능한 Map 유형, Helper 기능 및 libpf 라이브러리와의 사용자 공간으로부터의 상호 작용과 같은 eBPF 프로그램의 측면을 설명한다. 4절에서는 eBPF가 어떻게 Hook를 사용하는지 설명하고 그 중 두 가지인 XDP와 TC를 제시한다.

### 1.2 BPF

커널 내 패킷 필터[50]에 대한 이전 연구에서 영감을 받아 1992년 스티븐 매캔과 반 제이콥슨이 유닉스 BSD 시스템의 커널에서 패킷 필터링을 수행하는 솔루션으로 버클리 패킷 필터[46]를 제안했다.
명령어 집합과 해당 언어로 작성된 프로그램을 실행하기 위한 가상 머신(VM)으로 구성되었다.
처음에는 응용 프로그램의 바이트코드가 사용자 공간에서 커널로 전달되었고, 그곳에서 보안을 보장하고 커널 충돌을 방지하기 위해 점검되었다. 검증을 통과한 후, 시스템은 프로그램을 소켓에 부착하고 도착하는 패킷마다 실행하였다. 커널에서 사용자가 제공한 프로그램을 안전하게 실행하는 능력은 BPF의 훌륭한 설계 선택임이 입증되었다.
BPF의 또 다른 중요한 요소는 간단하고 잘 정의된 명령어 세트였다. 게다가 커널에는 BPF를 위한 JIT(Just-In-Time) 컴파일 엔진이 존재하였다. 이 모든 요소들은 도구의 우수한 성능을 위한 기본이었다.

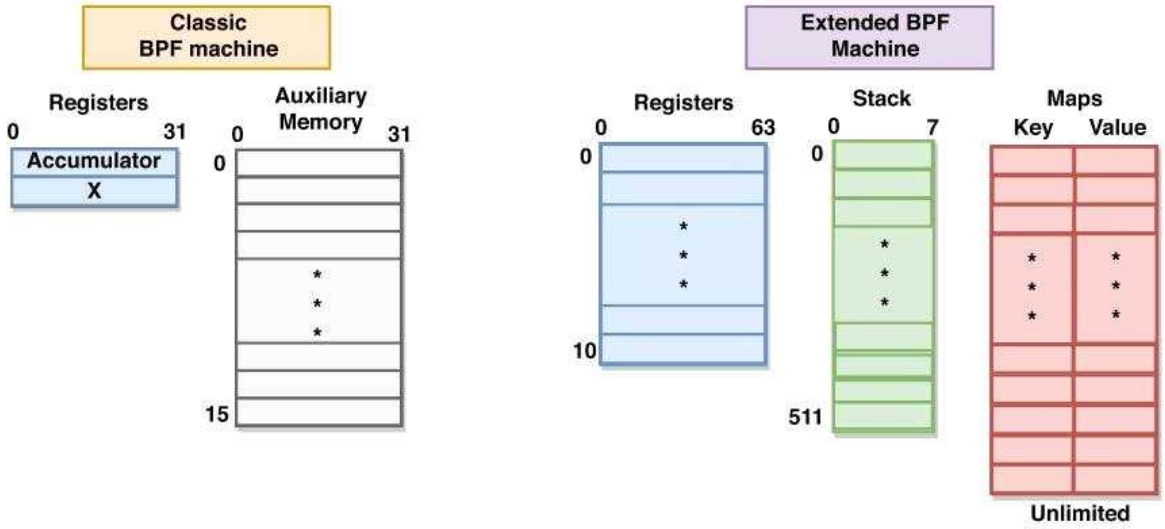


Fig. 1. BPF and eBPF processors.

BPF는 바이트코드 명령어 이외에도 패킷 기반 메모리 모델(처리된 패킷에서 암묵적으로 로드 명령어가 만들어진다), 두 개의 레지스터, 즉 누적기(A)와 인덱스 레지스터(X), 암시적 프로그램 카운터, 임시 보조 메모리를 정의한다.
그림 1의 왼쪽은 BPF 머신 아키텍처를 나타낸다. 리눅스 커널은 버전 2.5부터 BPF를 지원했다. 2011년 BPF 인터프리터가 동적 변환기로 수정될 때까지 BPF 코드에 큰 변화가 없었다[25]. BPF 바이트 코드를 해석하는 대신 커널은 BPF 프로그램을 x86 명령어로 직접 번역할 수 있게 되었다.
BPF를 사용하는 가장 중요한 도구 중 하나는 tcpdump 도구에 의해 사용되는 libpcap 라이브러리이다. 패킷을 캡처하기 위해 tcpdump를 사용할 때, 사용자는 해당 표현식과 일치하는 패킷만 실제로 캡처되도록 패킷 필터링 식을 설정할 수 있다. 예를 들어, "ip and tcp"라는 표현식은 TCP 전송 계층 프로토콜을 포함하는 모든 IPv4 패킷을 캡처한다. 이 표현식은 컴파일러에 의해 BPF 바이트코드로 축소될 수 있다. BPF man 페이지[35]를 기반으로 하는 코드 1은 TCP 세그먼트만 캡처하도록 패킷을 필터링하는 BPF 프로그램이다. 니모닉은 명확성을 위해 확장되었다.

**Code 1** BPF program example based on [35] to only allow IPv4 TCP segments.

```
1  load 2 bytes @ [12]
2  jump equal #0x800 jump true 3 jump false 6
3  load 1 byte @ [23]
4  jump equal #6  jump true 5 jump false 6
5  return #-1
6  return #0
```

기본적으로 코드 1이 하는 일은 다음과 같다:
• 명령 (1): 프레임의 오프셋(12)에서 2 바이트(16 비트)를 누적기에 로드한다. 오프셋(12)은 이더넷 프레임에서의 패킷 타입을 나타낸다.
• 명령 (2): 누적기 값을 IPv4에 대한 EtherType 값인 0x800과 비교합니다. 결과가 참이면 프로그램 카운터가 명령 (3)으로 점프하고(jumptrue) 그렇지 않으면 명령 (6)으로 점프합니다.
• 명령 (3): 프레임의 오프셋(23)을 바이트로서 누산기에 로드한다. 오프셋(23)은 IPv4 패킷의 프로토콜 필드를 나타낸다. 카운트는 이더넷 프레임의 처음부터이다.
• 명령 (4): 값을 상수 6(TCP 세그먼트에 대한 IPv4 패킷 프로토콜 필드의 값)과 비교한다. 참이면 명령어 (5)로 이동하고, 그렇지 않으면 명령어 (6)로 이동한다.

패킷 필터링 프로그램은 결과를 반환할 때까지 실행되며, 이는 일반적으로 부울(Boolean)이다. 0이 아닌 다른 값을 반환하면(명령어 (5)) 패킷이 필터와 일치했음을 의미하는 반면, 0을 반환하면(명령어 (6)) 패킷이 필터와 일치하지 않음을 나타내므로 폐기된다.

## 1.2 Extended BPF

BPF가 패킷 필터링에 매우 유용했지만, 커뮤니티는 커널을 도구화하는 기능으로부터 다른 영역들도 이익을 얻을 수 있다는 것을 깨닫게 되었다. 범용 커널 가상 머신으로 변환하기 위해[21], BPF 머신과 전체 아키텍처에 많은 개선 사항이 도입되었다. 이 새로운 버전은 eBPF(확장 BPF) 또는 간단히 BPF로 불리는 반면, 원래의 반복[BPF를 의미하는 것으로 보임]은 cBPF(classic BPF)가 되었다. eBPF는 리눅스 커널 버전 3.15에서 도입되었다. 이 섹션의 내용은 eBPF 사양 [60]에 기초한다.

그림 1의 오른쪽은 eBPF 엔진을 보여준다. 레지스터의 수는 2개에서 11개로 증가했고(이 중 10개는 쓰기 레지스터), 레지스터 폭은 32비트에서 64비트로 바뀌었고, 명령어 세트는 64비트가 되었으며, 새로운 엔진은 512바이트의 스택을 갖게 되었다. 맵이라고 불리는 전역 데이터 저장소도 포함되어 프로그램이 실행 사이에 데이터를 지속하고 서로 및 사용자 공간과 정보를 공유할 수 있게 해주었다. 커널 내부에서 실행되는 함수를 호출하는 옵션도 추가되었는데, 헬퍼 함수[60]이라고 불렸다. cBPF에서는 프로그램에서 참과 거짓의 경우에 대한 점프를 정의할 필요가 있었다. eBPF에서는 참 점프만 정의하면 되고, 거짓 점프는 프로그램의 실행 순서를 따른다(jump-fall-through라고 불림).

eBPF의 명령어 세트 아키텍처(ISA)는 함수 호출을 포함하도록 업데이트되었다. 이러한 호출들은 C 호출 규약을 따른다. 파라미터들은 네이티브 하드웨어에서 발생하는 것과 같이 레지스터들을 통해 함수들로 전달된다. 이것은 eBPF 함수 호출을 하나의 하드웨어 명령어에 매핑하는 것을 허용하여 오버헤드가 거의 발생하지 않는다. eBPF는 헬퍼 함수들을 가능하게 하기 위해 이 기능을 사용하여 프로그램들이 시스템 호출들을 하고 스토리지(지도들)들을 조작할 수 있게 한다. eBPF 가상 머신은 동적 로딩 및 프로그램 재장전을 지원한다. 이러한 방식으로, 프로그램들은 필요에 따라 런타임에 변경되거나, 수정되거나, 또는 다시 로딩될 수 있다.

표 1은 각 eBPF 레지스터의 기능을 설명한다. 레지스터 r0은 계산이 끝날 때 패킷의 포워딩에서 어떤 액션이 취해질 것인지를 나타내는 함수 리턴 값을 저장한다. 레지스터 r10은 유일한 읽기 전용 레지스터이며, BPF 스택에 대한 어드레스를 저장한다. eBPF가 C 호출 규약을 따르므로 인수는 함수에 레지스터 값으로 전달된다.

따라서 레지스

터 r1-r5는 이러한 목적을 위해 예약되는 반면, 레지스터 r6-r9는 함수 호출 사이에 그 값이 보존된다.



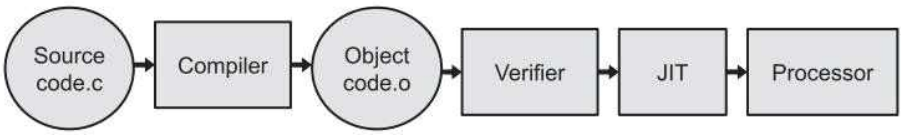


Fig. 2. eBPF Workflow.

## 2 EBPF SYSTEM

eBPF 시스템은 개발된 애플리케이션의 소스코드를 컴파일, 검증, 실행하기 위한 일련의 구성요소들로 구성된다. 이 절에서는 이들 각각에 대한 보다 자세한 내용을 설명한다.

### 2.1 Overview

eBPF 시스템의 일반적인 워크플로우는 그림 2에 도시되어 있다. eBPF 프로그램은 높은 수준의 언어(주로 제한된 C)로 작성된다. clang 컴파일러는 이를 ELF/object 코드로 변환한다. ELF eBPF 로더는 특별한 시스템 호출을 사용하여 커널에 삽입할 수 있다. 이 과정에서 검증자는 프로그램을 분석하고 승인이 나면 커널은 동적 변환(JIT)을 수행한다. 프로그램은 하드웨어로 오프로드될 수 있으며, 그렇지 않으면 프로세서 자체에 의해 실행된다.

### 2.2 Compiler

버전 3.7부터 LLVM 컴파일러 모음은 eBPF 플랫폼에 대한 백엔드를 가지고 있다.

C의 부분 집합[C 언어중 일부만 사용 가능하다는 뜻으로 보임]에서 eBPF 프로그램을 개발하고 clang 컴파일러를 통해 eBPF 형식의 실행 코드를 생성할 수 있다.

이 C의 부분 집합에서는 일부 syscall과 라이브러리가 제외되지만 eBPF Map 을 조작하고 다른 일반적인 작업을 수행할 수 있는 도우미 기능을 제공한다. 이러한 제한에 대한 부분적인 해결책은 섹션 9의 뒷부분에 제시되어 있다.

주요 제한 사항은 다음과 같다:

- eBPF는 C 언어 라이브러리의 서브셋만을 사용할 수 있다. 예를 들어, printf() 함수는 사용할 수 없다;
- 정적이 아닌 전역 변수는 허용되지 않는다;
- 유계 루프만 허용된다.
- 스택 공간은 512바이트로 제한된다..

또한 IOVisor[17, 32]와 같은 오픈 소스 프로젝트와 VMWare[68]에서 eBPF용 P4[14] 컴파일러를 구현하려는 노력이 있다.

이전 버전은 이미 존재하지만 아직 준비가 되지 않았다. 또한 BPF 컴파일러 모음(BCC) 프로젝트[7]을 통해 표준 C 코드에 대한 추가 추상화를 통해 eBPF 프로그램의 쓰기 및 상호 작용을 용이하게 할 수 있을 뿐만 아니라 eBPF와의 사용자 공간 상호 작용을 위한 메인 업스트림 라이브러리인 libbpf(섹션 3.6)도 가능하다.

### 2.3 Verifier

운영 체제의 무결성과 보안을 보장하기 위해, 커널은 시스템에 로딩되는 eBPF 명령어들의 정적 프로그램 분석을 수행하는 검증기를 사용한다. 그것의 구현은 커널 소스 코드에서 kernel/bpf/verifier.c에서 이용 가능하다.
무엇보다도, 검증기는 프로그램이 허용된 것보다 큰지(현재 제한은 106개의 명령어), 프로그램이 종료되는지 여부, 메모리 주소가 프로그램에 허용된 메모리 범위 내에 있는지 여부, 실행 경로의 깊이가 어느 정도인지 확인한다. 코드가 컴파일된 후, 그리고 프로그램을 데이터 평면에 로드하는 과정에서 호출된다. Miller[49]에 의해 좋은 개요가 제시된다.
검증기는 두 개의 path를 사용하여 프로그램을 거부할지 여부를 결정한다. 첫 번째 path에서는 깊이 우선 검색을 사용하여 프로그램 명령어를 DAG(Directed Acyclic Graph)로 파싱할 수 있는지 확인한다. 역방향 점프가 없거나 미리 정의된 크기 루프만 있는 eBPF 프로그램은 DAG로 합성되어 종료를 보장한다. 또한 DAG는 도달할 수 없는 명령어를 확인하고 최악의 경우 실행 시간을 계산하는 데 유용하다.
두 번째 path는 프로그램의 첫 번째 명령어에서 가능한 모든 경로를 탐색한다. 상태 머신을 생성하여 상태가 올바른 동작을 나타내는지 확인하고 이미 확인한 동작에 대한 기록도 유지한다[59]. 검증기는 가지치기를 위해 이미 확인한 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.
eBPF 검증기에 대해서도 두 가지 점을 더 언급할 가치가 있다. 첫 번째는 일부 eBPF 함수는 GPL 호환 라이선스를 가진 프로그램에서만 호출이 가능하다는 점과 관련이 있다.
그 때문에 검증자는 프로그램이 사용하는 기능의 라이센스와 프로그램의 라이센스가 호환되는지 검사하고 호환되지 않으면 프로그램을 거부한다.
마지막으로, 검증기는 커널의 무결성과 보안을 보장하기 위해 로컬 변수와 패킷 경계를 넘어서는 메모리 액세스를 허용하지 않는다. 패킷의 임의의 바이트에 액세스하기 위해서는 항상 경계 검사를 수행할 필요가 있다(이후 5.2.1절에서 나타낸 바와 같이). 그러나 패킷의 저장 공간이 변경되지 않는 한, 각 바이트는 한 번만 확인하면 된다. 이러한 방식으로 프로그램을 분석하는 동안 검증기는 패킷에 대한 모든 메모리 액세스가 검사된 주소에 있음을 보장한다. eBPF 프로그램이 이러한 유형의 검사를 수행하지 않으면 검증기는 이를 거부하여 커널에 로드할 수 없다[31].

## 3 EBPF PROGRAMS

eBPF는 성능 분석, 패킷 필터링 및 트래픽 분류와 같은 여러 종류의 응용 프로그램을 구현할 수 있다. eBPF 시스템 전체의 가장 큰 장점은 리눅스 커널 내부에서 유연하고 안전한 프로그래밍 가능 환경을 제공한다는 것이다. 예를 들어, eBPF 프로그램은 런타임 동안 로드 및 수정할 수 있으며 kprobe, perf 이벤트, 소켓 및 라우팅 테이블과 같은 커널 요소와 상호 작용할 수 있다[44].
그러나 eBPF 프로그램이 사용할 수 있는 하위 시스템과 기능은 커널에서 어디에 로드되는지, 즉 프로그램 유형에 따라 정의되는 어떤 계층 또는 하위 시스템에 연결되는지에 따라 달라진다. 이 절에서는 다양한 eBPF 프로그램 유형에 대해 논의하고 MAP이라고 불리는 KEY-VALUE 저장 데이터 구조를 제시하며 eBPF 프로그램이 사용할 수 있는 몇 가지 도우미 기능도 보여준다.

### 3.1 How and when are eBPF Programs Executed?

eBPF 프로그램을 실행하기 위해서는 먼저 맞춤형 프로그래밍이 가능한 인터페이스에 연결할 필요가 있다. 이 인터페이스를 후크(hook)라고 부른다. 후크는 특정 이벤트에 대한 프로그램 등록을 허용한다.
4절에서는 eBPF 프로그램을 연결할 수 있는 두 개의 리눅스 커널 후크인 XDP와 TC에 대해 설명한다.
eBPF 프로그램은 자신이 등록한 이벤트가 있을 때마다 실행된다. 컴퓨터 네트워킹에서 일반적인 이벤트는 패킷을 전송하거나 수신하는 것이다.

### 3.2 Program Types

각각의 eBPF 프로그램에는 종류가 있는데, 이것은 세 가지 중요한 측면들을 결정한다: 그것에 전달되는 입력은 무엇인가 (its context), 그것이 어떤 도우미 기능을 사용하도록 허용되는가, 그리고 그것이 어떤 커널 훅에 부착될 것인가. 예를 들어, 많은 종류의 eBPF 프로그램들 중 두 가지는 소켓 필터와 트레이싱[이벤트 추적같은 것을 생각하면 된다]이다.
소켓 필터 프로그램의 입력 파라미터는 소켓 버퍼로서 커널에 의해 생성되지만 L2 및 L3 정보가 제거된 패킷 메타데이터를 포함한다.
그러나 trace 프로그램은 레지스터 값들의 집합을 수신한다.
또한 이 두 종류에 대해 사용 가능한 헬퍼 함수들의 부분집합들은 비록 공통 범용 함수들의 중복이 존재하지만 동일하지 않다.
지원되는 프로그램 유형은 헤더 파일 linux/bpf.h에서 enum bpf_prog_type에 의해 정의된다.
버전 5.3-rc6에서 커널은 총 25개의 유효한 다른 프로그램 유형을 제공하며, 그 중 일부는 아래에 나열되어 있다:
• BPF_PROG_TYPE_SOCKET_FILTER: 소켓 필터링을 수행하는 프로그램;

• BPF_PROG_TYPE_SCHED_CLS: TC 계층에서 트래픽 분류를 수행하는 프로그램;
• BPF_PROG_TYPE_SCHED_ACT: TC 계층에 액션을 추가하는 프로그램;
• BPF_PROG_TYPE_XDP: eXpress Data Path 후크에 연결할 프로그램;
• BPF_PROG_TYPE_LWT_{IN, OUT 또는 XMIT: 계층-3 터널을 위한 프로그램;
• BPF_PROG_TYPE_SOKET_OPS: 재전송 타임아웃, 패시브/액티브 연결 설정 등과 같은 소켓 작업을 캐치하고 설정하는 프로그램;
• BPF_PROG_TYPE_SKB: 소켓 버퍼 및 소켓 파라미터(IP 주소, 포트 등)에 액세스하고 소켓 간 패킷 리디렉션을 수행하는 프로그램;
• BPF_PROG_TYPE_FLOW_DISSECTOR: 플로우 해부, 즉 네트워크 패킷 헤더에서 중요한 데이터를 찾는 프로그램.
이러한 프로그램 유형은 이 작업의 초점인 네트워킹과 관련이 있다. 그러나 커널 추적/모니터링을 위한 다른 프로그램 유형(예: BPF_PROG_TYPE_PERF_EVENT, BPF_PROG_TYPE_KPROBE 및 BPF_PROG_TYPE_TRACEPOINT), cgroup(예: BPF_PROG_TYPE_CGROUP_SKB 및 BPF_PROG_TYPE_CGROUP_SOK) 및 기타 [44, 51]이 있다. 지원되는 프로그램 유형의 전체 목록은 커널 소스 코드에서 $git grep -W 'bpf_prog_type {' include/uapi/linux/bphf] 명령으로 직접 얻을 수 있다

## 3.3 Maps

맵은 eBPF 프로그램이 사용할 수 있는 일반적인 키-값 저장소이다. 키와 값은 이진 블롭으로 처리되어 사용자 정의 데이터 구조와 유형을 저장할 수 있으며, 이들의 크기는 맵 정의 시 알려져야 한다. 맵은 bpf 시스템 호출을 사용하여 생성되며,
맵의 파일 디스크립터를 통해 맵 조작을 가능하게 한다. 이것은 명령어 BPF_MAP_CREATE(enum bpf_cmd로 정의됨)와 추가 매개변수가 있는 bpf_attr 결합을 bpf 시스템 호출에 전달함으로써 수행된다:

bpf( BPF_MAP_CREATE, &bpf_attr, size of (bpf_attr)).

이 경우 bpf_attr에 다음과 같은 속성을 설정해야 한다:
(1) map_type: 생성할 맵의 유형;
(2) key_size: 키를 저장할 바이트 수;
(3) value_size: 값을 저장할 바이트 수;
(4) max_entries: 맵의 행 수.
사용자 공간 프로세스는 여러 개의 맵을 생성할 수 있으며, 사용자 공간 프로세스와 커널에 로드된 eBPF 프로그램 모두에서 액세스할 수 있으므로 두 환경 간의 데이터 교환이 가능하다.
맵에 액세스하기 위해 eBPF 프로그램은 맵 ELF 섹션에서 구조 bpf_map_def (libpf로 정의됨)의 특별한 전역 변수를 선언해야 한다.
로드 프로세스 동안 파일 로더는 위의 syscall을 사용하여 선언된 맵을 생성하고 파일 디스크립터를 프로그램에 전달하며, 나중에 이는 런타임에 사용할 검증자에 의해 실제 포인터로 변환된다. 코드 2는 맵네임(mapname)인 BPF_PROG_TYPE_ARY MAP이 선언된 예를 보여준다.

3.3.1 Map 유형.

eBPF 프로그램에는 여러 가지 다양한 Map 유형이 있으며, 이들은 linux/bpf.h와 같은 enum bpf_map_type에 정의되어 있다. 각 Map 유형은 다른 동작을 제공하는데, 그 중 일부는 일반적으로 사용되는 반면 다른 것들은 특정한 사용 사례를 가지고 있다.
eBPF Map 코드들의 예들은 커널에 의해 제공된다. 커널 버전 5.3-rc6은 총 24개의 유효한 상이한 맵 타입들을 나열한다. 그들 중 일부는
• BPF_MAP_TYPE_ARRAY: 상위 수준의 프로그래밍 언어 배열에서와 같이 입력이 숫자에 의해 인덱싱되는 맵이다.
항목을 쿼리하는 입력이 주소인 RAM 모델을 따른다.
• BPF_MAP_TYPE_PROG_ARY: eBPF 프로그램들에 대한 참조들을 저장하는 맵이다. 그것의 사용은 예를 들어 특정 상황들을 처리하기 위한 서브 프로그램들의 호출을 허용한다.
• BPF_MAP_TYPE_HASH: 해시 함수를 사용하여 엔트리를 저장한다.
• BPF_MAP_TYPE_PERCPU_HASH: BPF_MAP_TYPE_HASH와 유사한 맵. 각 프로세서 코어에 대해 해시 테이블을 생성할 수 있다.
• BPF_MAP_TYPE_LRU_HASH: 해시 함수를 사용하여 엔트리를 저장하는 맵이다. 테이블이 가득 차면 요소 LRU를 제거하는 정책, 즉 제거할 요소가 가장 오래 사용된 것이다.
• BPF_MAP_TYPE_LRU_PERCPU_HASH: LRU 제거 정책으로 각 프로세서 코어에 대한 해시 테이블을 생성할 수 있다.
• BPF_MAP_TYPE_PERCPU_ARY: BPF_MAP_TYPE_ARY와 유사한 맵. 각 프로세서 코어에 대해 어레이를 생성할 수 있다.
• BPF_MAP_TYPE_LPM_TRIE: LPM(Longest-prefix match) trie.
• BPF_MAP_TYPE_ARY_OF_MAPS: eBPF 맵에 대한 참조를 저장하는 배열이다.
• BPF_MAP_TYPE_HASH_OF_MAPS: eBPF 맵에 대한 참조를 저장하는 해시 테이블.
• BPF_MAP_TYPE_DEVMAP: 네트워크 장치의 읽기 참조를 저장한다.
• BPF_MAP_TYPE_SOKMAP: 소켓 참조를 저장한다. 예를 들어 소켓 redi287 rection을 구현하는 데 사용할 수 있다.
• BPF_MAP_TYPE_QUE: 큐와 유사한 동작을 갖는 맵.
• BPF_MAP_TYPE_STACK: 스택과 유사한 동작을 갖는 맵.

지원되는 모든 맵 유형의 목록은 다음 명령을 사용하여 커널 소스 코드에서 직접 얻을 수 있다:
$ git grep -W 'bpf_map_type {' 포함/uapi/linux/bpf.h
3.3.2 map와 map pinning의 수명
모든 eBPF 객체(프로그램, 지도 및 디버그 정보)는 커널에 의해 유지되는 참조 카운터(refcnt)를 갖는다[63]. 사용자 공간 프로세스가 bpf_create_map()이라는 호출을 갖는 map를 생성할 때, 커널은 지도 refcnt를 1로 초기화한다. 그러면 커널은 map를 사용하는 새로운 eBPF 프로그램이 로드될 때마다 map refcnt를 증가시키고, 둘 중 하나가 닫힐 때마다 감소시킨다. map refcnt는 또한 그것을 생성한 프로세스가 종료(또는 충돌)할 때 감소될 것이다. refcnt가 0이 되면, 메모리 프리가 트리거되어 카운터와 관련된 eBPF 객체가 파괴된다. 이 플로우는 간단한 방법으로 eBPF map의 수명을 나타낸다.
방금 설명한 체계는 여러 프로그램 간에 동일한 eBPF map를 한 번에 공유할 수 있게 해준다. 그것은 그것의 상위 프로세스나 그것을 사용하는 일부 eBPF 프로그램이 활성화되어 있는 한 map를 활성화한다.
그러나 eBPF 지도를 활성화하는 또 다른 방법은 지도 고정을 하는 것이다.
사용자 공간 프로세스는 /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 맵이 고정되면 커널은 refcnt를 증가시켜 어떤 프로그램도 사용하지 않음에도 불구하고 생존할 수 있게 한다.
마찬가지로, Map이 고정되지 않을 때, Map의 refcnt는 감소하고, 만약 Map가 사용되지 않는다면, 다시 Map은 파괴될 수 있다.
맵 pinning은 사용자 공간으로부터의 bpf() 시스템 호출(명령어 BPF_OBJ_PIN), libpf(섹션 3.6), bpftool(섹션 6.2), iproute2(섹션 6.1)에서 제공하는 특수 맵 구조를 사용하여 여러 가지 방법으로 수행할 수 있다.
bpf_elf_map이라고 하는 이 대체 구조는 커널에서 제공하는 구조와 호환되며 bpf_map_def 대신 eBPF 프로그램에서 사용할 수 있다.
맵 범위를 정의하는 데 사용할 수 있는 pinning과 같은 추가 구성원을 노출한다.
이 필드는 PIN_GLOBAL_NS, PIN_OBject_NS 및 PIN_NONE의 세 가지 값을 수신할 수 있다.
PIN_OBject_NS로 생성된 맵은 이를 선언한 프로그램에 고유한 로컬 스코프를 가진다.
결과적으로 동일한 선언을 가진 맵은 다른 프로그램에 공존할 수 있다. 이 경우 해당 맵에 해당하는 노드를 저장하기 위해 BPF 파일 시스템에 특정 디렉터리가 생성된다. PIN_OBject_GLOBAL 값을 사용하면 글로벌 스코프로 맵이 생성되어 여러 프로그램에서 공유할 수 있다. 이 맵은 의사 파일 시스템의 디렉터리 글로벌에 항목을 수신한다.
PIN_NONE은 해당 맵을 파일 시스템에서 고정해서는 안 되며, 다른 응용 프로그램과 공유할 수 없도록 표시한다. 마지막으로 BPF 파일 시스템에서 해당 파일을 제거하면 맵의 고정을 해제할 수 있다. 이 제거는 syscall unlink()를 사용하여 수행할 수 있다.
3.3.3 잠금 메모리.
eBPF Map은 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. 이 제한은 ulimit -l <size>로 변경할 수 있다.

## 3.4 Helper functions

eBPF는 cBPF와 여러 가지 점에서 차이가 있는데, 하나는 프로그램이 소위 헬퍼 함수들을 호출하도록 허용하는 기능이다. 이것들은 Map, 라우팅 테이블, 터널링 메커니즘 등과 같은 각 후크의 컨텍스트와 다른 커널 시설 및 구조와의 상호 작용을 가능하게 하는 커널 인프라스트럭처에 의해 제공되는 특수한 기능들이다.
헬퍼 함수가 수행하는 작업에는 Map와의 상호 작용, 패킷 수정, 커널 트레이스에 대한 메시지 인쇄 등이 포함된다.
프로그램 유형이 많고 각각 특정 실행 컨텍스트를 가지므로 특정 함수가 호출할 수 있는 함수 목록은 커널이 구현하는 모든 헬퍼 함수의 부분집합을 나타내며, 이는 프로그램이 첨부된 후크에 따라 달라진다. 예를 들어 함수 bpf_xdp_adjust_tail()은 패킷의 마지막 바이트를 제거하는 데 사용되어 패킷의 크기를 효과적으로 줄일 수 있다. 그러나 이름에서 알 수 있듯이 XDP 후크에서만 사용할 수 있다. BCC 프로젝트는 각 프로그램 유형에 대한 헬퍼 함수 목록을 유지 관리한다[8].
eBPF 프로그램이 사용할 수 있는 헬퍼 함수는 커널이 제공하고 구현하는 리스트로 제한된다. 커널 모듈을 통한 확장은 허용되지 않기 때문에, 새로운 헬퍼 함수의 추가는 커널 소스 코드에 대한 확장을 통해서만 수행될 수 있다.
새로운 함수는 입력 파라미터의 최대 수를 5로 제한하면서 eBPF 프로그램 상의 모든 함수가 공유하는 호출 규약을 따라야 한다. 파라미터 통과는 레지스터 r1-r5의 사용을 통해 이루어지며, 스택과의 상호 작용은 필요하지 않다.
사용 가능한 헬퍼 함수의 수는 많고 새로운 커널 버전에 따라 지속적으로 증가한다. 버전 5.3-rc6은 총 109개의 그러한 함수들을 제공한다. 이들 중 일부는 아래에 강조되어 있다:
• bpf_map_delete_elem, bpf_map_update_elem, bpf_map_lookup_elem: 각각 맵에서 요소를 제거, 설치 또는 업데이트하고 검색하는 데 사용된다;
• bpf_get_prandom_u32: 32비트 의사 random 값을 반환한다;
• bpf_l4_csum_replace, bpf_l3_csum_replace: 각각 레이어-4 및 레이어-3 체크섬을 다시 계산하는 데 사용된다;
• bpf_ktime_get_ns: 시스템 부팅 후 시간(나노초)을 반환한다;
• bpf_redirect, bpf_redirect_map: 패킷을 다른 네트워크 장치로 리디렉션하는 기능.
두 번째는 특별한 방향전환 맵을 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: 사용자 공간 호출에서 sockopt ()를 get/setsockopt ()를 get/setsockopt로 설정하는 기능과 유사하다.
• bpf_get_local_storage: 로컬 저장 영역에 대한 포인터를 반환한다. 프로그램 유형에 따라 병렬로 실행되는 여러 프로그램 인스턴스 간에 이 영역을 공유할 수 있다.

헬퍼 함수들의 선언들은 커널 소스 코드의 디렉토리 tools/test/selftest/bpf 에 포함된 여러 헤더 파일들에 걸쳐 퍼져 있다.
그러나 그들 대부분은 bpf_helpers.h에 있다. 엔디안니스 변환을 수행하는 일부 일반적인 연산들은 bpf_endian.h에 의해 선언되며, 동일한 폴더에 위치한다. 이 파일은 예를 들어 bpf_ntohs() 및 bpf_htons()와 같은 잘 알려진 함수들의 BPF 호환 버전들을 제공한다.
앞에서 설명했듯이 커널은 이러한 기능들의 구현을 제공하며 eBPF 프로그램은 .c 대응어가 필요 없이 서명을 포함하는 헤더 파일에 대해서만 컴파일하면 된다.
이것은 -I 플래그를 사용하여 clang하도록 커널 소스 코드의 이러한 파일들에 대한 경로를 전달함으로써 수행될 수 있다.
그러나 필요한 헤더 파일의 로컬 복사본을 만들고 커널 트리에 대한 컴파일을 피할 수도 있으므로 코드의 컴파일 및 배포가 더 쉬워진다.
3.4.1 테일 콜.[tail call]
eBPF 프로그램은 테일 콜을 통해 다음에 실행할 다른 프로그램을 호출하여 발신자에게 다시는 돌아가지 않을 수 있다.
이들은 예를 들어 복잡한 프로그램을 단순화하고 프로그램의 동적 체인을 구축하는 데 사용될 수 있다[62]. 테일 콜은 긴 점프로 구현되며, 현재 스택 프레임을 재사용하여 새로운 스택 프레임을 생성하는 것을 방지하여 함수 콜과 비교할 때 오버헤드를 최소화한다. 테일 콜의 사용은 (i) eBPF 프로그램의 참조를 저장하기 위해 프로그램 어레이(BPF_MAP_TYPE_PROG_ARRAY)라고 하는 특수화된 맵을 사용하고, (ii) 테일 콜을 실행하기 위해 도우미 함수(bpf_tail_call)를 사용하는 것을 포함한다.
프로그램 배열은 키-값 쌍으로 사용자 공간에 의해 채워질 수 있는데, 여기서 값은 eBPF 프로그램들의 파일 서술자들이다.
도우미 함수는 세 가지 인수를 받는데, 컨텍스트는 프로그램 배열 맵에 대한 참조이며, 룩업 키 테일 호출에는 몇 가지 제한이 있다.
테일 호출의 체인은 루프를 형성할 수 있기 때문에, 현재 무한 루프를 피하기 위해 테일 호출의 최대 수는 32개로 제한된다. 또한 eBPF는 동일한 유형의 프로그램만 테일 호출을 허용한다. 호출자의 것(JITed 또는 해석됨)과 일치해야 하는 번역 유형도 마찬가지이다.

## 3.5 Return Codes

eBPF 프로그램이 반환하는 코드는 프로그램 종류에 따라 의미와 가치가 달라진다.
예를 들어, XDP 프로그램(섹션 4.2)은 bpf.h의 enum xdp_action에 의해 정의된 처리 후에 패킷으로 무엇을 해야 하는지에 대한 평결을 반환한다.
TC 리턴 코드(섹션 4.3)는 유사한 의미를 갖지만 다른 열거형을 사용한다.
그러나 소켓 필터는 반송 코드를 사용하여 스택에 전달할 패킷 길이를 나타내므로 패킷을 트리밍하거나 아예 폐기할 수도 있다.

## 3.6 Interaction from User Space with libbpf

커널이 사용자 공간으로부터 eBPF 프레임워크와 상호 작용하기 위해 bpf() syscall을 노출시키지만,
그것은 많은 목적을 위한 단일 도구이기 때문에 다소 복잡하다. libpf[39]는 좀 더 사용자 친화적인 API를 제공하는데,
이것은 커널 커뮤니티에 의해 개발된 사용자 공간 라이브러리이다. 이것은 커널 소스 코드의 tool/lib/bpf에서 이용할 수 있으며,
커널로부터 대응하는 파일들을 미러링하는 GitHub[40]의 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 tool/test/selftest/bpf에서 사용할 수 있는 몇 가지 예는 이 라이브러리의 사용 사례를 보여주며 좋은 시작점이 될 수 있다. 또한 GitHub의 독립 실행형 버전에는 커널 소스에 대한 컴파일 없이 libpf를 프로젝트에 통합하는 방법에 대한 자세한 지침이 있다.
이 API는 bpf() 시스템 호출의 직접적인 몇 가지 래퍼를 포함하며 eBPF 시스템과의 상호 작용을 돕기 위해 여러 구조를 노출한다.
예를 들어, 사용자는 각각 구조 bpf_map, struct bpf_program 및 struct bpf_object를 사용하여 Map, 프로그램 및 객체 파일에 대한 정보를 처리할 수 있다.
이러한 객체 유사 유형에는 각각 특정 게터와 세터가 있으며, 그 이름에는 구조의 이름으로 시작하고 이중 밑줄과 수행할 작업의 선언적 이름이 뒤따른다. 다음 단락에는 이러한 각 객체 유형에 대해 가장 일반적인 몇 가지 함수가 나열되어 있다.
eBPF 프로그램이 포함된 .c 파일을 clang으로 컴파일한 후 생성된 객체 파일에는 각 프로그램에 해당하는 여러 ELF 섹션이 포함된다.

사용자 공간 프로그램은 bpf_object_* 함수 제품군을 사용하여 이러한 파일과 상호 작용할 수 있다. 일부 예에는 다음이 포함된다:
• bpf_object__open 및 bpf_object_open_xattr: 개체 파일을 읽고 struct bpf_object에 포인터를 반환한다.
_xattr 버전에서는 프로그램 유형을 지정할 수 있다;
• bpf_object__load 및 bpf_object_load_xattr: 프로그램을 struct bpf_object에서 커널로 로드한다.
_xattr 버전을 사용하면 원하는 로그 레벨을 지정할 수 있다;
• bpf_object__pin_maps: 객체 파일에서 모든 맵의 피닝을 처리할 수 있다;
• bpf_object_for_each_program: 객체 파일에서 각 프로그램을 반복하는 매크로;
• bpf_object__find_program_by_title: BPF 프로그램의 섹션 이름을 기준으로 핸들을 반환한다.

상기 함수들 중 일부는 또한 각각의 대응물(무부하, 닫기, 언핀)을 갖는다.

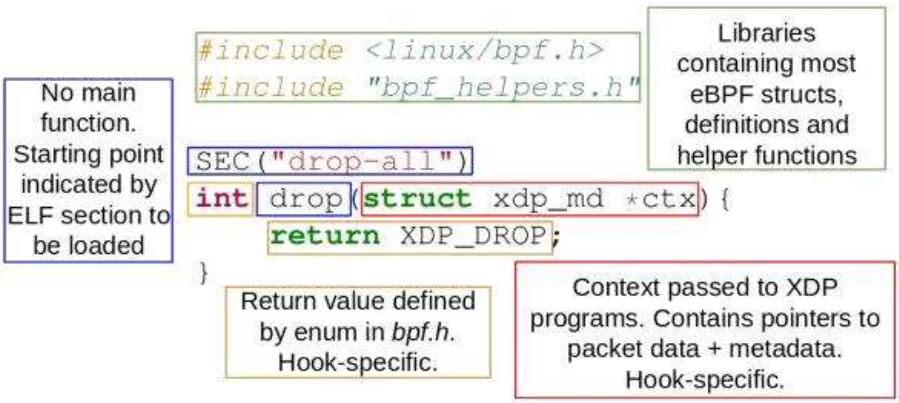


Fig. 3. Dropworld example illustrating the structure of an eBPF program.

• bpf_program__load: 커널에 주어진 프로그램을 로드한다;
• bpf_program__fd: BPF 프로그램에 대한 파일 디스크립터를 반환한다. • bpf_program__pin: 주어진 프로그램을 특정 파일 경로로 고정한다;
• bpf_program__set_ifindex: 지도 및 프로그램을 오프로드할 장치의 ifindex를 설정한다.

마지막으로 bpf_map__*로 시작하는 함수는 생성, 정보 검색, 재사용, 피닝 등 맵 객체에 대한 작업을 지원한다.
• bpf_map__def: 기본 지도 정보(유형, 크기 등)를 반환한다;
• bpf_map__reuse_fd: 새 프로그램을 로드할 때 기존 맵을 재사용할 수 있다;
• bpf_map__resize: 맵에서 허용되는 최대 항목 수를 변경하는 데 사용된다;
• bpf_map__fd: 지정된 맵에 대한 파일 설명자를 반환한다;
• bpf_map_for_each: 객체 파일의 모든 맵에 대해 반복하는 매크로;
이 섹션에서는 사용 가능한 모든 기능에 대해 광범위하게 설명할 것이 아니라 독자들에게 libpf가 제공하는 설비들을 살짝 보여준다.
API에서 누락된 부분들은 perf 버퍼들과의 상호작용, 전처리 도우미들 등을 포함한다.
표시된 함수들의 전체 서명뿐만 아니라 사용 가능한 모든 호출들의 전체 목록은 소스 코드의 libpf.h 헤더 파일을 확인하기 바란다.
libpf를 사용한 예제 코드는 5절의 뒷부분에 표시될 것이다.

### 3.7 Basic Program Structure

그림 3은 eBPF 프로그램의 기본 구조를 보여준다.
수신된 모든 패킷을 드롭하는 간단한 XDP 프로그램을 제시한다. 이에 대한 더 자세한 내용은 4절에서 다룰 것이다. 라이브러리 linux/bpf.h는 추가 헤더 파일이 필요한 트래픽 제어(TC) 및 perf와 같은 특정 하위 시스템을 제외하고 eBPF 프로그램이 사용하는 모든 구조 및 상수 정의를 가지고 있다.
원칙적으로 모든 eBPF 프로그램은 이 파일을 포함해야 한다.
리턴 값들과 프로그램들이 수신하는 입력 파라미터는 그들이 첨부할 후크에 따라 달라진다.
그림 3과 같이 XDP 프로그램들은 구조 xdp_md에 대한 포인터를 수신하며, 이에 대해서는 4.2.1절에서 자세히 설명한다. 다른 후크들의 프로그램들은 서로 다른 컨텍스트 구조를 수신한다.다른 후크에 대한 프로그램은 섹션 5를 참조하라.
표시된 프로그램은 표준 C 프로그램에서 일반적으로 주 기능을 포함하지 않는다.
프로그램의 시작점은 ELF 개체 파일에서 해당 섹션으로 표시된다.
컴파일이 되면 표시되는 프로그램은 default .text 섹션에 위치하게 된다. 섹션 5는 사용자 정의 섹션의 정의를 보여준다.

아래에서는 그림 3의 프로그램 예제의 객체 파일과 디셈블러 출력을 보여준다.

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

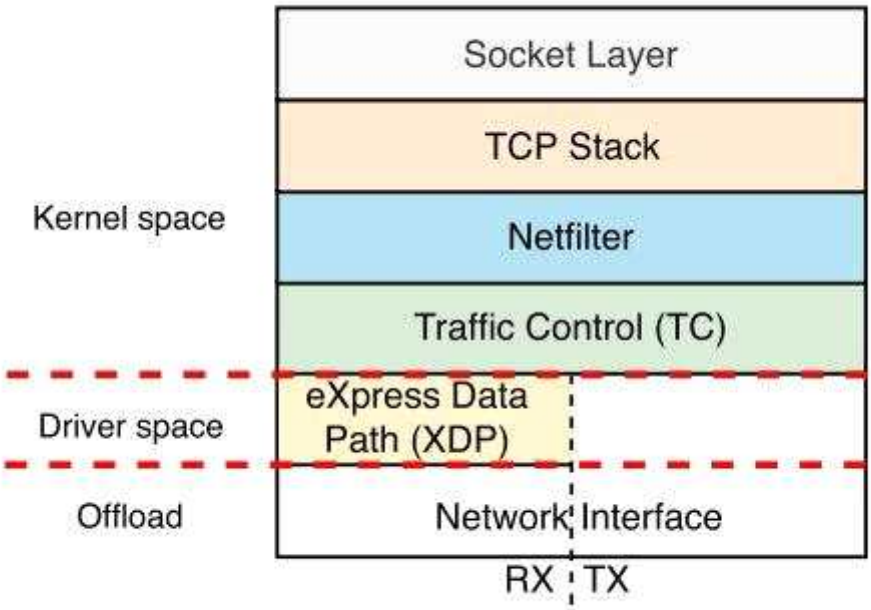


Fig. 4. Linux kernel network stack.

첫 번째 명령어는 r0을 등록하기 위해 1(XDP_DROP)을 쓴다. 표 1에서 r0은 eBPF 프로그램의 반환 값을 포함한다는 것을 기억하라. 두 번째 명령어는 그냥 빠져나간다. 이제 독자가 eBPF 프로그램의 기본 프로그램 구조를 이해했으니, 다음 절에서는 XDP를 다룬다.

## 4 NETWORK HOOKS

컴퓨터 네트워킹에서 후크는 운영 체제에서 호출 전 또는 실행 중에 패킷을 가로채는 데 사용된다. 리눅스 커널은 eBPF 프로그램을 부착할 수 있는 여러 후크를 노출하므로 데이터 수집과 사용자 정의 이벤트 처리가 가능하다.
리눅스 커널에는 많은 후크 포인트가 있지만 네트워킹 하위 시스템에 존재하는 두 가지, 즉 eXDP와 TC에 초점을 맞출 것이다. 이들을 함께 사용하여 RX와 TX 모두에서 NIC에 가까운 패킷을 처리할 수 있으므로 많은 네트워크 응용 프로그램을 개발할 수 있다. 이 절에서는 eBPF를 사용하여 이 두 후크를 프로그래밍하는 방법과 각 후크에 프로그램을 로드하는 방법을 설명한다.

### 4.1 Kernel's Networking Layers

OS로 들어오는 패킷들은 그림 4와 같이 커널에서 여러 계층에 의해 처리된다. 이 계층들은 소켓 계층, TCP 스택, Netfilter, Tcp(트래픽 제어), eXDP(eXpress Data Path), NIC이다. 사용자 공간 응용 프로그램으로 향하는 패킷들은 이 모든 계층을 거치며, 이 과정에서 Netfilter 계층에 상주하는 iptables와 같은 모듈에 의해 가로채여 수정할 수 있다.
앞서 설명한 바와 같이 eBPF 프로그램은 커널 내부의 여러 곳에 부착할 수 있어 패킷 망글링 및 필터링이 가능하다

### 4.2 eXpress Data Path

eXDP(eXpress Data Path)는 리눅스 커널 네트워크 스택의 최하위 계층이다. 이는 디바이스의 네트워크 드라이버 내부의 RX 경로에만 존재하며, OS에 의해 메모리 할당이 이루어지기도 전에 네트워크 스택의 가장 이른 시점에서 패킷 처리를 허용한다. eBPF 프로그램을 부착할 수 있는 후크(hook)를 노출한다[31].
이 후크에서 프로그램은 들어오는 패킷에 대한 빠른 결정을 내릴 수 있고 또한 임의의 수정을 수행하여 커널 내부의 처리에 의해 부과되는 추가적인 오버헤드를 피할 수 있다. 이것은 DDoS 공격의 완화와 같은 애플리케이션에 대한 성능 속도 측면에서 XDP를 최상의 후크로 만든다.

패킷을 처리한 후, XDP 프로그램은 액션을 반환하는데, 이는 프로그램 종료 후에 패킷에 무엇을 해야 하는지에 대한 최종 평결을 나타낸다.

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. |

4.2.1 XDP Input Context

XDP 프로그램이 보여주는 컨텍스트는 커널에 의해 전달되는 단일 입력 파라미터에 의해 정의된다.

이것은 bpf.h로 정의되는 xdp_md 유형의 구조이며, 여기서 코드 3으로 재현된다. 프로그램 실행 시에 데이터와 data_end 필드는 각각 패킷 데이터의 시작과 끝에 대한 포인터를 포함한다. 이 값들은 5절에서 자세히 설명한 바와 같이 패킷 액세스를 안내하는 데 사용되어야 한다. 구조 내부의 세 번째 값은 data_meta 포인터로, XDP 프로그램이 다른 계층과 패킷 메타데이터를 교환할 때 자유롭게 사용할 수 있는 메모리 영역의 주소를 유지한다.

마지막 두 필드는 각각 패킷을 수신한 인터페이스와 해당 RX 큐의 인덱스를 유지한다. 이 두 값에 액세스할 때 BPF 코드는 커널 내부에서 다시 작성되어 실제로 해당 값을 유지하는 커널 구조 구조 xdp_rxq_info에 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 유형은 부호가 없는 정규 32바이트 정수이다. 메모리 주소를 올바르게 사용하려면 프로그램이 먼저 포인터 값을 캐스트해야 하며, 이는 거의 모든 XDP 프로그램의 시작 부분에 존재하는 다음 코드 스니펫을 통해 이루어진다:

여기서, ctx는 위에 도시된 xdp_md 형식의 XDP 프로그램의 입력이다.

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

4.2.2 XDP Actions.

표 2는 가능한 모든 XDP 액션, 그 값 및 그 설명을 나열한다.

동작은 프로그램 리턴 코드로 지정되며, 이는 eBPF 프로그램이 종료되기 직전 레지스터 r0에 저장된다.

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 액션은 XDP 프로그램이 (i) 다른 NIC(물리적 또는 가상적), (ii) 추가 처리를 위한 다른 CPU, 또는 (iii) 사용자 공간 처리를 위한 AF_XDP 소켓으로 패킷을 리디렉션할 수 있게 한다.

다른 것들과 달리 이 액션은 리디렉션 대상을 지정하기 위한 매개 변수를 필요로 한다.

이것은 두 가지 헬퍼 함수 중 하나인 bpf_redirect() 또는 bpf_redirect_map()을 통해 이루어진다. 전자는 타겟의 인터페이스 인덱스를 수신하고 네트워크 디바이스에 집중한다. 후자는 보다 일반적인 대안으로, 최종 타겟을 검색하기 위해 보조 맵에 대한 룩업을 수행하며, 이는 넷 디바이스 또는 CPU 둘 다일 수 있다. 패킷 전송을 일괄 처리하여 bpf_redirect()에 비해 훨씬 우수한 성능을 제공하고, 사용자 및 커널 공간에서 맵 엔트리를 동적으로 수정할 수 있기 때문에 두 번째 옵션이 권장된다.

4.2.3 XDP 작동 모드.

향상된 성능을 위해 XDP에 연결된 eBPF 프로그램은 디바이스 드라이버 수준에서 패킷 처리 파이프라인을 변경하며, 이는 관련

네트워크 드라이버에 의한 명시적인 지원을 필요로 한다. i40e, nfp, mlx* 및 ixgbe 계열과 같은 고속 디바이스를 위한 일부 드라이버는 이미 이러한 기능을 가지고 있다. 이러한 드라이버와 호환되는 디바이스에서 XDP 프로그램은 운영 체제에 의해 처리되기도 전에 드라이버에 의해 직접 실행된다. 이를 XDP Native mode라고 한다. BCC Project [9]는 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. 이 모드에서 XDP 실행은 네이티브 실행을 에뮬레이트하면서 운영 체제 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[47].
시스템은 eBPF 프로그램을 로드할 때 이 두 가지 모드 중에서 자동으로 선택합니다.
한번 로드되면 다음 절과 같이 ip tool을 이용하여 동작모드를 확인할 수 있다.
또 다른 동작 모드인 XDP Offload가 있다. 이름에서 알 수 있듯이 eBPF 프로그램은 호환 가능한 프로그래밍 가능 NIC(섹션 7.2.1)로 오프로드되어 다른 두 모드와 비교했을 때 훨씬 더 우수한 성능을 발휘한다. 이 모드는 프로그램을 로드할 때 명시적으로 표시되어야 한다.

4.2.4 XDP 및 XDP 오프로드 예.
XDP 프로그램을 컴파일하고 로드하는 방법을 설명하기 위해 그림 3의 예제를 사용할 것이다.
간단한 XDP 프로그램으로 모든 패킷이 네트워크 인터페이스에 도착하면 바로 드롭한다.
예제를 dropworld.c 파일에 저장한 후, 코드는 clang 컴파일러를 사용하여 ELF 객체 파일로 컴파일할 수 있다:
$ clang - target bpf -O2 -c dropworld.c -o dropworld.o
ip 도구는 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 XDP 후크의 인터페이스에 부착되었는지 확인할 수도 있다.
$ ip link show dev [DEV]
DEV: <BROADCAST, MULTICAST, UP, LOWER_UP> mtu 1500 xdp qdisc
mq state UP 모드 기본 그룹 기본 qlen 1000
link/ether 00:16:3d:13:08:80 brd fff:ff:ff:ff:ff prog/xdpid 27 tag f95672269956c10d jited
첫 번째 출력 행의 키워드 xdp는 XDP 네이티브 모드에서 XDP 프로그램이 인터페이스에 연결되어 있음을 나타낸다.

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. |

다른 가능한 출력은 xdp generic과 xdp offload일 수 있다.
프로그램은 매개 변수를 off로 전달함으로써 인터페이스에서 제거될 수도 있다:
# ip link set dev [DEV] xdp off
이 마지막 예제에서 eBPF 프로그램은 드라이버가 CPU를 사용하여 XDP 후크에서 실행되었다.
프로그램을 오프로드하기 위해서는 이전과 동일한 방법을 사용하되 xdpoffload 파라미터를 ip link set 명령으로 전달할 수 있다:
# ip-force link set dev [DEV] xdpoffload object dropworld.o sec.text
이전과 마찬가지로 프로그램을 제거하려면 다음 명령을 실행하라:
# 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 리눅스 상에서 트래픽 제어 정책들을 실행하는 역할을 한다. 그 안에서 네트워크 관리자는 시스템에 존재하는 다양한 패킷 큐들에 대해 서로 다른 큐잉 규율들(qdisc)을 구성할 수 있을 뿐만 아니라 패킷들을 거부하거나 수정하기 위한 필터들을 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.
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:
# tcqdisc 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 기능적 eBPF 프로그램과 XDP 계층과의 상호 작용에 대한 예는 GitHub[67]의 저장소를 확인하라.

### 4.4 Comparison Between XDP and TC

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.
그러나 후자는 struct__sk_buff를 통해 사용 가능한 추가 파싱된 데이터의 이점을 얻을 수 있으며 TX에서 가장 낮은 계층인 입력 트래픽과 출력 트래픽 모두에 대해 eBPF 프로그램을 실행할 수 있다.

## 5 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. 그러나 후자는 struct__sk_buff를 통해 사용 가능한 추가 파싱된 데이터의 이점을 얻을 수 있으며 TX에서 가장 낮은 계층인 입력 트래픽과 출력 트래픽 모두에 대해 eBPF 프로그램을 실행할 수 있다.

### 5.1 TCP Filter

이 절에서는 독자가 eBPF 프로그램에 익숙해지도록 돕기 위해 코드에 대한 심층적인 설명과 함께 몇 가지 예를 제시한다. 첫 번째 예는 코드 1의 BPF 예와 유사하게 IPv4 TCP 세그먼트만 허용하는 프로그램을 설명한다. 두 번째 예에서는 libppf를 통해 사용자와 커널 공간 사이의 상호 작용을 보여주고, 세 번째 예에서는 XDP와 TC 계층의 프로그램이 어떻게 함께 통계를 수집할 수 있는지 보여준다. 마지막으로, 우리는 독자에게 몇 가지 더 외부적인 예를 제시한다.

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.
이 프로그램은 XDP 후크에 로드되도록 설계되었으므로 4.2.1절에서 설명한 대로 함수의 입력 파라미터는 xdp_md 형식이어야 한다.
처리되는 패킷의 바이트는 data와 data_end 포인터에 의해 구분되며, 이는 패킷에 액세스하기 위해 프로그램 전체에서 사용되어야 한다. 이 두 값에 대한 형식 변환은 표준이므로 패킷 데이터에 액세스하는 모든 eBPF 프로그램의 시작 부분에 라인 9와 10을

사용해야 한다. 데이터를 사용하여 헤더의 파싱은 리눅스에서 제공하는 표준 헤더 파일로 수행할 수 있다.
그러나 다른 일반적인 패킷 파싱 리눅스 프로그램과의 주된 차이점은 실제로 프로토콜 헤더 데이터에 액세스하기 전에 바인딩 검사가 필요하다는 것이다.
커널 검증자는 엄격한 메모리 바인딩 검사를 수행하므로(섹션 2.3) 패킷 데이터에 대한 모든 액세스는 경계 검사가 포함된 if 문(라인 14와 21)으로 덮어야 한다. 패킷의 저장 공간을 수정하는 도우미 함수를 사용하지 않는 한(예: bpf_xdp_adjust_head()) 각 바이트는 한 번만 확인하면 된다. 그런 경우 이러한 함수를 호출한 후 모든 검사를 다시 수행해야 한다. eBPF 프로그램이 이러한 유형의 검사를 수행하지 않으면 로드 시간 동안 검증자에 의해 거부되어 커널에 로드되지 않는다.
패킷의 프로토콜 번호를 추출한 후 프로그램은 TCP 프로토콜에 해당하는지 확인하고 스택(라인 26)을 통과하도록 허용한다. 그렇지 않으면 패킷은 그냥 드롭된다(라인 28).
5.1.2 eBPF Bytecode. 컴파일 시 clang은 eBPF 명령어로 객체 파일을 생성하고, 이 파일은 커널에 로딩될 수 있다. 2.3절에서 설명된 바와 같이, 검증기는 프로그램에 기초하여 DAG를 생성한다.
그림 5는 이 예에서 각각의 DAG를 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

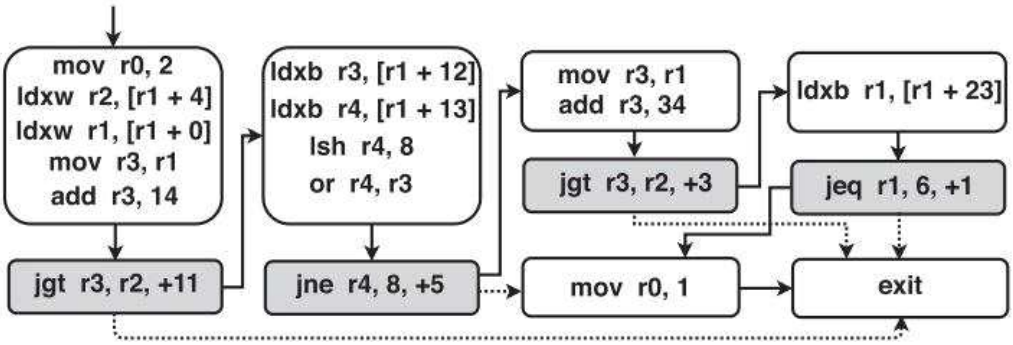


Fig. 5. Directed acyclic graph example of Code 4.

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

```
1  #include <linux/bpf.h>
2  #include <linux/if_ether.h>
3  #include <linux/ip.h>
4  #include <linux/tcp.h>
5  #include <linux/in.h>
6  #include "bpf_endian.h"
7
8  int isTCP( struct xdp_md *ctx ) {
9      void *data_end = (void *)(long) ctx->data_end;
10     void *data_begin = (void *)(long) ctx->data;
11     struct ethhdr* eth = data_begin;
12
13     // Check packet's size
14     if(eth + 1 > data_end)
15       return XDP_PASS;
16
17     // Check if Ethernet frame has IPv4 packet
18     if (eth->h_proto == bpf_htons( ETH_P_IP )) {
19         struct iphdr *ipv4 = (struct iphdr *)( ((void*)eth) + ETH_HLEN );
20
21         if(ipv4 + 1 > data_end)
22           return XDP_PASS;
23
24         // Check if IPv4 packet contains a TCP segment
25         if (ipv4->protocol == IPPROTO_TCP)
26           return XDP_PASS;
27     }
28     return XDP_DROP;
29 }
```

이 eBPF 프로그램을 더 잘 이해하려면 레지스터 r1은 메인 메모리에 저장된 입력 컨텍스트에 대한 포인터로 시작하고 레지스터 r0은 리턴 값을 저장한다는 것을 기억하라(표 1).

첫 번째 노드에서, 첫 번째 어셈블러 명령어는 r0(표 1부터)을 2(XDP_PASS)로 설정한다(표 2와 같이).
또한, 로드 명령어는 입력이 통과한 상태에서 패킷의 시작과 끝에 대한 참조를 계산한다(행 10-12).
그런 다음 이더넷 헤더의 경계를 검사하여 나중에 유효한 메모리 접근을 보장한다(행 14-16).
검사가 실패하면, 첫 번째 점프 명령어는 흐름을 종료 명령어로 전환하여 프로그램을 종료한다.
그렇지 않으면, 이더넷 헤더의 바이트 12와 13이 로드되며(0부터 계산), 이것이 이더넷 타입 필드이다. 그런 다음, 바이트 스왑이 수행되어 엔디안니스가 설정된다(행 19).
두 번째 점프 명령어는 이더넷 유형이 0x0800인지를 비교한다(19번 라인). 그런 다음 IP 헤더의 경계를 확인하여 나중에 유효한 메모리 접근을 보장한다(다시, 검증자에 의해 요구됨)(21~23번 라인). 다음, IPv4 패킷에서 IP 프로토콜 필드를 추출한 다음, 프로그램은 TCP 프로토콜(값 6)(라인 26)(DAG의 마지막 회색 배경 노드)인지를 확인한다.
마지막으로 레지스터(r0)에 수락을 나타내는 값 2(XDP_PASS)가 로드되었기 때문에 패킷은 커널로 전달된다. IP 패킷에 TCP 헤더가 포함되어 있지 않은 경우 값 1(XDP_DROP)이 r0에 로드된다. 마지막 명령어는 코드가 종료됨을 알려준다.

## 5.2 User and Kernel Space Interaction

다음으로, 우리는 samples/bpf 디렉토리에 있는 커널 소스 코드에서 직접 추출한 xdp1 예제를 제시한다.
그것은 두 부분으로 나뉜다: xdp1_kern.c는 컴파일되어 커널에 로드되는 실제 eBPF 프로그램이고, xdp1_user.c는 eBPF 프로그램을 커널에 로드하고 맵을 통해 상호 작용하는 사용자 공간 대응물이다. 우리는 각각의 코드를 보여주고 아래에서 그것들에 대해 따로 논의한다.
5.2.1 커널 공간.
파일 xdp1_kern.c[54]는 XDP 후크의 각 패킷을 처리하고, 해당 IP 프로토콜 번호를 추출하고, rxcnt라는 CPU별 어레이 맵을 사용하여 프로토콜당 수신되는 패킷 수를 카운트하고, 최종적으로 모든 수신 트래픽을 드롭하는 eBPF 프로그램을 포함한다.

```
1  /* Copyright (c) 2016 PLUMgrid
2   *
3   * This program is free software; you can redistribute it and/or
4   * modify it under the terms of version 2 of the GNU General Public
5   * License as published by the Free Software Foundation.
6   */
8  #include <uapi/linux/bpf.h>
9  #include <linux/in.h>
10 #include <linux/if_ether.h>
11 #include <linux/if_packet.h>
12 #include <linux/if_vlan.h>
13 #include <linux/ip.h>
14 #include <linux/ipv6.h>
15 #include "bpf_helpers.h"
16
17 struct bpf_map_def SEC("maps") rxcnt = {
18     .type = BPF_MAP_TYPE_PERCPU_ARRAY,
19     .key_size = sizeof(u32),
20     .value_size = sizeof(long),
21     .max_entries = 256,
22 };
23
24 static int parse_ipv4(void *data, u64 nh_off, void *data_end) {
25     struct iphdr *iph = data + nh_off;
26
27     if (iph + 1 > data_end)
28         return 0;
29     return iph->protocol;
30 }
31
32 static int parse_ipv6(void *data, u64 nh_off, void *data_end) {
33     struct ipv6hdr *ip6h = data + nh_off;
34
35     if (ip6h + 1 > data_end)
36         return 0;
37     return ip6h->nexthdr;
38 }
39
40 SEC("xdp1")
41 int xdp_prog1(struct xdp_md *ctx) {
42     void *data_end = (void *)(long)ctx->data_end;
43     void *data = (void *)(long)ctx->data;
44     struct ethhdr *eth = data;
45     int rc = XDP_DROP;
46     long *value;
47     u16 h_proto;
48     u64 nh_off;
49     u32 ipproto;
50
51     nh_off = sizeof(*eth);
52     if (data + nh_off > data_end)
53         return rc;
54
55     h_proto = eth->h_proto;
56
57     if (h_proto == htons(ETH_P_8021Q) || h_proto == htons(ETH_P_8021AD)) {
58         struct vlan_hdr *vhdr;
59
60         vhdr = data + nh_off;
61         nh_off += sizeof(struct vlan_hdr);
62         if (data + nh_off > data_end)
63             return rc;
64         h_proto = vhdr->h_vlan_encapsulated_proto;
65     }
66     if (h_proto == htons(ETH_P_8021Q) || h_proto == htons(ETH_P_8021AD)) {
67         struct vlan_hdr *vhdr;
68
69         vhdr = data + nh_off;
70         nh_off += sizeof(struct vlan_hdr);
71         if (data + nh_off > data_end)
72             return rc;
73         h_proto = vhdr->h_vlan_encapsulated_proto;
74     }
75
76     if (h_proto == htons(ETH_P_IP))
77         ipproto = parse_ipv4(data, nh_off, data_end);
78     else if (h_proto == htons(ETH_P_IPV6))
79         ipproto = parse_ipv6(data, nh_off, data_end);
80     else
81         ipproto = 0;
82
83     value = bpf_map_lookup_elem(&rxcnt, &ipproto);
84     if (value)
85         *value += 1;
86
87     return rc;
88 }
89 char _license[] SEC("license") = "GPL";
```

첫 번째로 지적해야 할 점은 C 소스 파일이 많은 eBPF 프로그램을 포함할 수 있다는 것이다.
컴파일러가 생성한 ELF 파일에서 고유한 섹션으로 분리되어 있다.
해당 함수(Line 40) 위의 섹션 레이블은 생성된 개체 파일에 있는 프로그램을 포함할 ELF 섹션의 이름을 컴파일러에게 나타낸다.
이 정보는 커널에 코드를 로드하는 동안 시스템이 어떤 ELF 섹션을 로드해야 할지 알 수 있도록 하기 위해 필요하다.
섹션 레이블은 맵(Line 17)과 프로그램 라이센스(Line 89) 선언 시에도 사용되는데, 이 둘은 모두 고정된 값을 갖는다.
검증자는 라이센스 섹션을 사용하여 사용자가 사용할 수 있는 도우미 함수를 결정하는데, 이 중 일부는 GPL 호환 라이센스를 선언하는 프로그램으로 제한되기 때문이다. 앞의 예와 유사하게 프로그램은 패킷 헤더를 IPv4 또는 IPv6 헤더까지 파싱한다.
Layer-4 프로토콜 유형(Line 77 및 79)을 결정한 후 프로그램은 룩업 헬퍼 함수(Line 83)를 사용하여 해당 프로토콜에 대한

카운터를 검색한다. 이 함수는 맵에 저장된 현재 값이 존재하는 경우 포인터를 반환하거나, 존재하지 않는 경우 NULL로 반환한다. 이 주소는 맵 업데이트 작업 없이 저장된 데이터를 직접 변경하는 데 사용할 수 있다. 마지막으로 프로그램은 현재 패킷에 대해 XDP 후크가 수행해야 하는 작업을 반환하며, 이 경우 패킷은 항상 XDP_DROP이므로 패킷을 폐기해야 함을 나타낸다.
5.2.2 사용자 공간.
커널에 의해 수집되고 맵 rxcnt에 저장된 통계는 xdp1_user.c[55] 파일로 구현된 사용자 공간 응용 프로그램에 의해 쿼리된다. 간결성을 위해, 우리는 이 프로그램의 의미 있는 부분들만 아래에서 강조한다.
먼저, 프로그램을 커널에 로드하기 위해 섹션 4.2.4와 같은 iproute2의 기능을 사용하는 대신, 이 예제에서는 libpf(섹션 3.6)를 기반으로 사용자 정의 로더를 구현한다. 이 방법은 xdp1_kern.c의 프로그램이 커널에 로드되는 방식에 대한 더 높은 수준의 제어를 제공하며, 이는 사용자 공간 응용 프로그램에서 프로그램적으로 수정될 수 있다. 이러한 경우, libpf.h와 bpf.h는 사용자 공간에서 eBPF 시스템과의 상호 작용을 허용하기 위해 포함된다:

```
1   // SPDX-License-Identifier: GPL-2.0-only
2   /* Copyright (c) 2016 PLUMgrid
3    */
21  #include "bpf/bpf.h"
22  #include "bpf/libbpf.h"
```

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

```
73  struct bpf_prog_load_attr prog_load_attr = {
74      .prog_type        = BPF_PROG_TYPE_XDP,
75  };
```

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

그런 다음 이 구조는 XDP 후크에 프로그램을 로드하는 데 사용된다. 성공의 경우 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.

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

eBPF 프로그램을 커널에 로딩한 후에, rxcnt 맵에 대한 참조를 얻는다. 함수 bpf_map_next는 프로그램에서 선언된 맵들의 리스트에 대한 반복기를 반환한다. 이 경우 선언된 맵이 하나뿐이므로, 그 반복기의 값은 그것을 참조하는 파일 디스크립터를 얻는 데 사용될 수 있다. libpf.h 라이브러리는 맵 리스트의 이름 또는 인덱스로 맵 디스크립터를 얻는 다른 함수들도 제공한다.

```
118 map = bpf_map__next(NULL, obj);
119 if (!map) {
120     printf("finding a map in obj file failed\n");
121     return 1;
122 }
123 map_fd = bpf_map__fd(map);
```

마지막으로 eBPF 프로그램은 인터페이스에 연결할 준비가 되어 있으며, 사용자 공간 응용 프로그램은 poll_stats() 함수 내부의 무한 루프에 들어갈 수 있다.

```
130 if (bpf_set_link_xdp_fd(ifindex, prog_fd, xdp_flags) < 0) {
131     printf("link set xdp fd failed\n");
132     return 1;
133 }
134
135 poll_stats(map_fd, 2);
```

poll 함수 poll_stats()는 map rxcnt의 파일 기술자를 이용하여 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 어레이 맵이므로 저장된 데이터는 실제로 여러 CPU에 걸쳐 퍼져 있다. 도우미 bpf_num_possible_cpus(37번 라인)는 프로그램이 사용한 CPU의 수를 검색하고, 이는 각 CPU(값)의 데이터를 저장할 어레이의 크기를 설정하는 데 사용된다. 그런 다음 무한 루프는 최근에 수집된 통계를 검색하기 위해 전체 맵을 수시로 조회한다. 이것은 bpf_map_get_next_key 반복기 함수를 사용하여 수행되며, 한 번에 하나의 맵 키를 생성하여 맵의 모든 항목을 순서대로 반복할 수 있다. 커널 측 프로그램에서 키는 IP 프로토콜 번호이므로 선언된 맵 크기(256)가 주어지면 모든 키를 통해 반복하는 것은 가능한 모든 IP 프로토콜 번호를 통해 반복하는 것에 해당한다. 사용자 공간 버전인 bpf_map_lookup_elem은 모든 CPU에서 해당 키와 관련된 맵 값을 실제로 동시에 읽는 데 사용되며(Line 49), 이 값은 값 배열의 도우미 함수에 의해 저장된다. 그런 다음 이 값을 추가하여 전체 통계(Line 51)를 얻고, 얻은 값이 마지막에 표시된 값보다 크면 그 차이가 표준 출력으로 인쇄되어 샘플링 간격 동안 해당 프로토콜 번호를 가진 패킷이 몇 개 수신되었는지 사용자에게 보여준다.
이 예는 eBPF 프로그램과 지도를 통해 사용자와 커널 공간이 어떻게 상호 작용할 수 있는지 보여준다.
빠른 경로(커널) 상의 모든 데이터 수집 및 패킷 처리는 최소의 최적화된 eBPF 프로그램에 의해 실행되는 반면, 에이전트는 느린 경로(사용자 공간) 상에서 주기적으로 데이터를 검색하고 이에 기초한 액션, 예를 들어 사용자에게 디스플레이할 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 별개의 eBPF 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 지도에 저장하고 RX와 TX를 모두 추적한다.
이 예는 iproute2(섹션 6.1)가 제공하는 몇 가지 고유한 시설, 맞춤형 사용자 공간 프로그램 없이 프로그램을 로드할 수 있는 방법, 그리고 다양한 계층의 프로그램이 지도를 통해 상호 작용할 수 있는 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};

    stats = bpf_map_lookup_elem(&trackers, key);
    if(stats){
        if(is_rx){
            stats->rx_cnt++;
            stats->rx_bytes += bytes;
        }else{
            stats->tx_cnt++;
            stats->tx_bytes += bytes;
        }
    }else{
        if(is_rx){
            newstats.rx_cnt = 1;
            newstats.rx_bytes = bytes;
        }else{
            newstats.tx_cnt = 1;
            newstats.tx_bytes = bytes;
        }

        bpf_map_update_elem(&trackers, key, &newstats, BPF_NOEXIST);
    }
}

SEC("rx")
int track_rx(struct xdp_md *ctx)
{
    void *data_end = (void *)(long)ctx->data_end;
    void *data = (void *)(long)ctx->data;
    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, 맵들은 사용자 정의 구조들을 처리할 수도 있다.
예를 들어, 통신 통계가 저장될 추적자 맵(27번 라인)에 대해, 구조 쌍 및 구조 통계(15번 라인 및 20번 라인)가 각각 키 및 값으로서 여기서 사용된다. 통계는 인터페이스에 의해 보여지는 각각의 고유한 IPv4 주소 쌍에 대해 추적되는 RX 및 TX 패킷 및 바이트 수에 따라 구성된다.
Map의 정의는 앞의 예와 다르다. bpf_elf_map 구조는 iproute2에 의해 사용되므로 헤더 파일 iproute2/bpf_elf.h를 포함하며, 이는 iproute2를 소스에서 설치할 때 시스템에 추가된다.
약간 다른 필드 이름 외에 추가적인 고정 필드도 포함하며, 3.3.2절에서 설명한 대로 Map의 범위를 정의하는 데 사용할 수 있다. 이 필드에 대한 값 설정은 각각의 개별 복사본 대신 두 프로그램이 공유하는 단일 추적기 맵이 있어야 한다고 결정한다.
이 동일한 소스 파일은 두 개의 서로 다른 ELF 섹션에서 사용되는 두 프로그램을 유지한다. 섹션 rx(81번 라인)에는 XDP 프로그램이 있어 패킷이 시스템에 도착하면 바로 처리한다.
parse_ipv4 (Line 35) 함수는 패킷에서 소스 및 IP 주소를 추출하는 데 필요한 모든 파싱을 실행한다.
패킷 데이터와 주소 정보를 배치하는 버퍼에 대한 포인터 외에 입력 파라미터로서 부울 값도 받는다.
이것은 이것이 들어오는 패킷인지 나가는 패킷인지를 나타낸다. 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 반면, 소스 IP는 원격이다. TX에서는 그 반대이다.
패킷에서 IP 쌍이 추출되면 함수 update_stats로 전달되어 그에 따라 추적기 맵을 업데이트한다. 또한 보이는 패킷이 들어오는 것인지 나가는 것인지를 나타내는 매개 변수 is_rx를 수신한다. 함수는 해당 주소 쌍(59번 라인)에 대한 이전 통계를 검색하고 항목이 이미 있는지 확인한다. 그렇지 않은 경우 BPF_NOEXT 플래그(77번 라인)와 함께 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.
이 프로그램은 XDP_PASS 코드를 반환하는 것을 종료하여 패킷을 스택에 정상적으로 전달한다.
TX에는 XDP 계층이 없으므로 NIC로 패킷을 전송하기 직전에 처리할 수 있는 가장 가까운 것은 TC 계층에 eBPF 프로그램을 연결하는 것이다. 섹션 tx의 프로그램이 바로 그 역할을 한다.

## 5.4 Other Examples

위에 나타낸 예들은 eBPF 프로그램들의 개발 동안 고려되어야 하는 몇 가지 기본적인 측면들을 보여준다. 그들은 또한 사용자 공간 프로그램들과 상호 작용하는 방법을 보여준다.

리눅스 커널: 다른 유용한 예들은 리눅스 커널의 소스 코드에 의해 제공되며, samples/bpf, tools/testing/selftests/bpf의 두 개의 별개의 디렉터리에 위치한다.
둘 다 커널 스택의 다양한 기능과 훅의 사용을 보여주는 프로그램들을 포함하고 있으며, 각각의 새로운 버전의 커널에 새로운 프로그램이 추가된다. 첫 번째 디렉터리에 있는 대부분의 예들은 두 개의 별개의 파일로 나뉘는데, 하나는 프로그램을 로드하기 위한 사용자 공간 코드로 되어 있고 다른 하나는 커널에 로드하기 위한 eBPF 프로그램의 구현으로 되어 있다. 일반적으로 이 폴더의 예들은 다양한 작업에 유용할 수 있는 독립 실행형 프로젝트를 나타낸다. 두 번째 디렉터리의 예들은 커널 개발 중에 기능 테스트를 실행하기 위한 기초로 사용된다. 커널에 들어가는 실제 eBPF 프로그램들은 progs/하위 디렉터리 내부에 배치되며, 루트의 나머지 파일들은 사용자 공간 프로그램과 이를 로드하는 데 사용되는 스크립트에 해당한다.

XDP 프로젝트: 커널의 예제를 넘어 공식 XDP 튜토리얼 [70]에서도 XDP 후크의 다양한 기능을 자세히 설명하는 단계별 지침이 포함된 예제를 제시한다.

L4 로드 밸런서: Netronome은 L4 로드 밸런서를 구현하는 l4lb라는 XDP 프로그램의 코드를 제공한다[53]. 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 맵에서 키로 사용된다.
이 eBPF 맵은 프로그램이 패킷을 리디렉션할 수 있는 사용 가능한 서버의 주소로 채워진다. 이 프로그램은 맵의 데이터와 함께 외부 IP 헤더를 확장하고 삽입한다. 그런 다음 패킷은 해당 서버로 전달된다. 이 프로그램은 SmartNIC으로 오프로드되어 CPU 사이클을 절약할 수도 있다.
프로그래밍 가능한 수신 측 스케일링: 수신 측 스케일링(RSS)을 사용하면 CPU 코어에 의해 처리될 수신 패킷을 매핑할 수 있다. 이는 멀티프로세서 시스템에서 여러 CPU에 걸쳐 네트워크 트래픽을 분산시키는 데 중요하다. RSS 기법은 많은 네트워크 어댑터에서 여러 큐의 사용을 통해 패킷의 계산을 별개의 CPU 세트에 분산시키기 위해 사용된다.
그러나, 구현들은 일반적으로 독점적이고 하드웨어 기반이어서, 프로그래밍가능성이 거의 또는 전혀 허용되지 않는다.
eBPF 프로그램을 사용하여, 패킷 분배는 맵 값들 또는 로딩된 eBPF 프로그램의 완전한 대체를 통해 요구에 따라 수정될 수 있다[53].

## 6 TOOLS

이 절에서는 eBPF 프로그램을 개발하고 디버깅하는 데 유용할 수 있는 몇 가지 도구를 소개한다.

### 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 사용자 공간 프로그램 없이 eBPF 코드를 커널에 로드하는 대안적인 방법들을 제공한다.
ip와 tc 도구를 모두 사용하는 방법에 대한 예제는 이 본문에서 이전에 보여주었다. 또한 iproute2는 eBPF 시스템과 상호 작용하는 고유한 인터페이스를 가지고 있으며 eBPF 맵 할당 범위를 지정하는 기능과 같은 추가 기능을 제공한다. 불행히도 추가적인 기능으로 인해 libbpf 로더와 호환되지 않는다. 예를 들어 iproute2를 사용할 때 맵은 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 Map의 범위를 정의하는 데 사용되는 피닝과 같은 여분의 필드를 가지고 있다(섹션 3.3.2에서 설명됨). 따라서 iproute2는 libpf 프로그램을 로드할 수 있지만 libpf가 여분의 기능을 처리하는 방법을 모르기 때문에 그 반대는 사실이 아니다[2]

### 6.2 bpftool

bpftool [15]는 eBPF 프로그램을 로드하고, 맵을 만들고 조작하며, eBPF 프로그램과 맵에 대한 정보를 수집할 수 있는 사용자 공간 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 safford
• 런타임에 Map의 내용을 나열하고 인쇄할 수 있다:
# bpftool map
# bpftool map dump id <map id>
• 또한 프로그램 로드, 수행을 포함한 일부 관리 작업을 수행할 수 있다
Map 값을 검색하거나 업데이트합니다. 마지막 항목의 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 -Dropworld.o

### 6.4 BPF Compiler Collection

오픈 소스 프로젝트 BPF 컴파일러 모음(BCC)[7]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 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 기원이며 가장 인기 있는 플랫폼이며, 가장 활발한 개발이 이 텍스트의 주요 초점이다.
커널 내부의 eBPF 애플리케이션들은 네트워크 처리에 제한될 뿐만 아니라 커널 계측 및 모니터링에도 적용될 수 있다.
따라서, 오늘날 eBPF 프로그램들은 리눅스 인트로스펙션 및 성능 분석을 수행하기 위한 Import 툴셋을 나타내며, IOVisor[32]와 같은 중요한 오픈 소스 프로젝트들은 이러한 전면에서 많은 솔루션들을 제공한다.
7.1.2 Userspace BPF.
UBSPF(Userspace BPF)[65]는 eBPF 프로세서를 사용자 공간에서 실행되도록 적응시키는 오픈 소스 프로젝트이다. uBPF 프로젝트는 eBPF 인터프리터와 JIT 컴파일러의 복사본을 가지고 있어 사용자가 사용자 공간에서 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 러스트[52]에서 eBPF VM의 대체 구현을 제공한다.
uBPF는 사용자 공간에서 실행되기 때문에 언롤링 루프를 필요로 하는 것처럼 eBPF 검증기에 의해 부과되는 제한을 받지 않는다. 그러나 uBPF는 커널의 eBPF 시스템과 달리 맵에 대한 네이티브 지원을 제공하지 않으며, 어떤 헬퍼 기능도 구현하지 않는다. 그럼에도 불구하고, 다음에 논의할 BPFabric 프로그래밍 가능 가상 스위치의 일부로 사용한 Reference [34]에서와 같이 이러한 기능을 쉽게 지원하도록 확장할 수 있다.
7.1.3 BPFabric.
OpenFlow의 한계를 다루기 위해 [34]는 BPFabric이라는 새로운 SDN 아키텍처를 제안한다. BPFabric은 uBPF의 수정된 버전을 사용하여 데이터 평면이 eBPF 명령으로 프로그래밍될 수 있도록 하는 스위치 아키텍처이다.
eBPF 프로그램은 동적으로 수정될 수 있으므로 임의 프로토콜의 파싱 및 eBPF 맵을 사용하여 상태를 저장할 수 있다. 새로운 기능에 지원을 추가하고 원격 측정, 통계 수집 및 패킷 추적과 같은 서비스를 제공하기 위해 새로운 도우미 기능이 개발될 수 있다.
제어 평면은 사우스바운드 API를 통해 데이터 평면과 통신하는 에이전트를 호스팅한다. 에이전트는 (i) 스위치의 거동을 변경하고, (ii) 패킷을 수신하고, (iii) 이벤트를 보고하고, (iv) 테이블 엔트리를 읽고 업데이트하는 역할을 한다. 에이전트는 컨트롤러로부터 컴파일된 코드를 수신할 때, eBPF ELF Loader를 사용하여 스위치 파이프라인을 수정한다.
eBPF Loader는 검증자를 호출하여 코드를 확인한다. 성공 시 필요한 eBPF 테이블의 할당을 수행하고 수신된 바이트 코드를 스위치 고유의 형식으로 변환해야 한다.
패킷을 수신할 때, eBPF 프로세서는 이전에 프로그램된 eBPF 명령어들을 패킷에 실행한다.

### 7.2 Hardware

7.2.1 Netronome SmartNICs 스마트 네트워크 인터페이스 카드(SmartNICs)는 런타임에 기능을 수정하여 다양한 작동 모드를 구현할 수 있는 네트워크 장치이다. 이 장치들은 단순히 연결과 기본 계층-2 및 물리 계층 처리를 제공하는 대신 전용 코어와 메모리를 사용하여 사용자 정의 패킷 처리를 실행하여 컴퓨터의 CPU에서 많은 주기를 해방시킬 수 있다. 우리가 아는 한, 현재 시장에서 이러한 종류의 제품을 제공하는 유일한 회사는 Netronome이다.
네트워크 관리자는 예를 들어 P4 또는 eBPF에서 패킷 처리 루틴을 정의하고 이러한 프로그램을 넷로놈 카드에 로드할 수 있으며, 넷로놈 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.
대응하는 펌웨어 버전을 사용하면 TC 및 XDP 계층에서 실행되는 eBPF 프로그램이 성능 향상을 위해 이들 장치로 원활하게 오프로드될 수 있다. 오프로딩을 수행하는 데 필요한 대부분의 드라이버와 코드는 이미 업스트림 커널의 일부이다.
뛰어난 처리 능력을 감안할 때, SmartNIC는 저지연 및 고속 워크로드를 위한 좋은 대안이 되었다. 일부 기능은 하드웨어에서 직접 구현될 수 있기 때문에, 운영 체제의 네트워크 스택을 올라갈 필요조차 없이 패킷을 수정하여 네트워크로 다시 보낼 수 있다.

더욱이, 프로그램은 즉시 수정 및 업데이트될 수 있다. 이러한 장치에 대한 전형적인 사용 사례는 초기 패킷 필터링, 레이트 리미터, DDoS 완화 작업, 로드 밸런싱, RSS 작업, 패킷 스위칭 등이다.

## 8 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[29]. 이 기술을 사용하면 네트워크 관리자가 네트워크의 특정 지점에서 패킷에 대해 수행할 다른 작업을 지정할 수 있다. 소스 라우팅 헤더(SRH)라는 특수 필드가 있는 MPLS 레이블 또는 IPv6 프로토콜을 사용하면 각 패킷이 세그먼트라고 하는 라우팅 및 처리 작업의 순서가 지정된 목록으로 캡슐화된다. 패킷이 네트워크를 통해 이동할 때 활성화된 디바이스는 세그먼트 목록을 처리하고 그에 의해 지정된 작업을 수행한다. 이를 통해 예를 들어 다른 네트워크 기능의 구현이 가능하다[4]. 리눅스 커널은 버전 4.10 이후로 이미 IPv6를 통한 세그먼트 라우팅을 지원하지만 라우팅 및 레이블이 지정된 패킷의 송수신과 같은 몇 가지 처리 옵션만 사용한다.
참조 [24]는 새로운 세그먼트의 생성 및 사양을 보다 일반적이고 유연하게 만들기 위해 eBPF 프레임워크를 사용했다.
새로운 헬퍼 기능의 추가를 통해 저자는 eBPF 프로그램 형태의 세그먼트 구현을 허용하도록 리눅스 커널을 확장했다.
따라서 새로운 네트워크 기능이 쉽게 개발되고 리눅스 라우팅 규칙과 연관되어 IPv6 환경을 통한 세그먼트 기반 라우팅과 통합하기가 더 쉬워진다.
다른 컨텍스트에서 데이터 평면에 프로그램 가능성을 추가하기 위해 eBPF 프레임워크를 사용하는 여러 다른 프로젝트가 있다.
InKeV[5]는 가상 데이터 센터 네트워크의 데이터 경로를 수정하기 위해 eBPF 프로그램을 사용하는 네트워크 가상화 플랫폼이다.
OpenFlow 고정 매칭 문제를 해결하기 위해 참조 [33]은 다양한 매칭 필드를 제공하기 위해 eBPF 프로그램을 사용할 것을 제안한다. 참조 [64]는 Open vSwitch 가상 스위치에 대한 확장 가능한 데이터 경로 아키텍처를 생성하기 위해 eBPF를 사용한다.
참조 [12]는 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 중복(decomputing)된다. 참조 [6]에서, eBPF 프로그램은 패킷 복제 동작을 제어한다.
클라우드플레어(Cloudflare)와 같은 일부 회사는 이미 운영 환경에서 eBPF를 사용하는데,
이들은 서비스 거부 공격 완화, 로드 밸런싱을 위해 XDP를 사용하고 소켓 필터링 및 디스패치를 위해 상위 계층에서 eBPF를 사용한다[45].
또 다른 예는 Katran[26]이라고 불리는 eBPF를 기반으로 L4 로드 밸런서를 개발한 페이스북(Facebook)에서 나온다. 또한 넷플릭스(Netflix)는 성능 모니터링 및 시스템 프로파일링을 위해 eBPF를 사용해 왔다[36].
더욱이 eBPF 기술은 컨테이너 네트워킹에서 중요한 역할을 해오고 있다. 컨테이너 간 통신을 위해 리눅스에서 일반적으로 사용되는 veth 인터페이스 타입은 커널 4.14[41]에서 Native XDP를 지원받았다.
Cilium [20] 오픈소스 프로젝트는 네트워크 헤더보다 높은 수준의 세부사항을 인지하고 eBPF를 광범위하게 사용하여 마이크로서비스 응용 프로그램에 네트워킹 및 보안을 제공한다.
Cilium은 eBPF 프로그램을 컨테이너에 로드함으로써 컨테이너별 보안 및 네트워킹 정책을 적용할 수 있다. Weave Scope는 한동안 eBPF를 활용하여 Kubernetes 클러스터에서 TCP 연결을 추적하고 있는 또 다른 프로젝트이다[69]. Project Calico도 올해 초 eBPF를 기반으로 컨테이너 네트워킹을 위한 새로운 데이터 평면이 개발되고 있다고 발표했다[57].
마지막으로, eBPF 기반 오픈 소스 프로젝트도 최근에 등장하고 있다. Cilium[18]은 컨테이너 네트워크 및 마이크로서비스 애플리케이션에서 보안을 제공하는 프로젝트로 IOVisor[32]가 있다.
프로젝트는 이전에 논의된 BCC[7] 및 uBPF[65] 외에도 Go 언어를 사용하여 eBPF 시스템과 상호 작용할 수 있는 gobpf[30], ply[56] 및 커널 인트로스펙션을 위한 bpftrace[16]와 같은 여러 eBPF 기반 하위 프로젝트를 유지한다.

## 9 LIMITATIOUS 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 설명한다. 여기서 설명하는 일부 해결 방법은 참조 [3]에서 제공되는 XDP 프로젝트 개발 저장소에서 찾을 수 있다.

### 9.1 Subset of C Language Libraries

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

## 9.2 Non-static Global Variables

현재 eBPF 프로그램은 정적 전역 변수만 지원한다[13]. 대안은 BPF_MAP_TYPE_PERCPU_ARY 맵을 사용하는 것이다. 이 맵은 프로그램 실행 중에 단일 엔트리로 임시 데이터를 저장하는 데 사용할 수 있는 사용자 정의 크기의 비공유 메모리 공간을 예약한다[19].

## 9.3 Loops

eBPF 검증기는 루프가 모든 프로그램을 확실히 끝내는 것을 허용하지 않았다. 이 한계를 우회하기 위해 채택된 첫 번째 기법은 clang 컴파일러의 루프 언롤링 명령어를 사용하여 루프를 독립적인 명령어들의 반복되는 시퀀스로 다시 쓰는 것이었다. 이 기법은 컴파일 시 반복 횟수를 결정할 수 있어야만 사용할 수 있기 때문에 이 한계를 부분적으로 해결한다. 이 외에도 최종 프로그램에서 명령어의 수를 증가시키는 부작용이 있다.
아래 코드 스니펫은 컴파일러에게 for 루프를 언롤하도록 지시하는 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 loops 로 대체되었다. 이 릴리스 이후로 검증자는 루프가 타임아웃 내에 종료되는지 여부를 먼저 확인하지 않고 루프를 포함하는 모든 eBPF 프로그램을 거부하지 않는다.
유계 loops 는 존 패스트벤드(John Fastabend)에 의해 제안되었으며 2018년 리눅스 plumbers 회의에서
BPF 마이크로 컨퍼런스에서 발표되었다.
이 기술은 검증기를 통해 모델링된 간단한 루프를 사용할 수 있게 한다. 루프의 거동을 유도 변수를 통해 분석하고 유도 변수를 사용한 메모리 액세스가 메모리 어드레스 범위에 속하는지 확인한다. 검증기 상의 유계 루프에 대한 구현 세부 사항은 참조 [27]에서 확인할 수 있다.

## 9.4 Limited Stack Space

C 프로그램의 로컬 변수들은 eBPF 프로그램으로 변환된 후 스택에 저장된다. 스택 공간이 단지 512 바이트로 제한되므로, 프로그램의 모든 로컬 변수들을 변환 후 저장하기에 충분하지 않을 수 있다. 채택된 해결 방법은 전역 변수에 대해 동일하게 사용된다:
스택 공간이 충분하지 않을 때 BPF_MAP_TYPE_PERCPU_ARY를 보조 버퍼로 사용하여 일부 로컬 변수를 저장한다[19].

## 9.5 Complex Applications

Miano et al. [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 프로그램들은 후크들과 연관되기 때문에, 그들은 수동 이벤트-기반 모델을 따른다.
이것은 예를 들어, (현재) 사용자 공간 애플리케이션에 의해 수행되어야 할, 능동 네트워크 측정들과 같은 비동기 태스크들을 수행하는 것을 어렵게 한다.
더욱이, eBPF 프로그램들을 위한 제어 평면을 구현하기 위한 기존의 도구들(예컨대, libpf 및 bpf syscall)은 매우 기본적이며, 주로 데이터 교환을 위해 맵 구조에 의존한다.
그러나, Cilium이나 BCC와 같은 프로젝트들은 그러한 점에서 진보를 나타낸다.
마지막으로, eBPF 프로그램의 최대 명령어 수는 4096개로 제한되었고, 이것은 복잡한 네트워크 기능의 개발을 어렵게 했다.
이 한계를 극복하는 한 가지 방법은 프로그램을 여러 개의 하위 프로그램으로 분할하고 bpf_tail_call() 도우미 기능을 사용하여 하나의 하위 프로그램에서 다른 하위 프로그램으로 점프하는 것이었다. 이 기술은 한 모듈에서 다른 모듈로 점프할 때 낮은 오버헤드로 각 모듈이 다른 기능(분석, 분류 또는 필드 수정)을 수행하는 느슨하게 결합된 모듈의 집합으로서 네트워크 서비스의 개발을 가능하게 한다.
그러나 허용되는 중첩된 꼬리 호출의 최대 수는 32개이다. 2019년 4월, 최대 명령어 수가 4096개에서 100만 개로 증가하여 꼬리 호출[1] 없이 더 큰 eBPF 프로그램을 실행할 수 있게 되었다.

# 10 COMPARISON WITH SIMILAR TECHNOLOGIES

이 작업을 통해 볼 수 있듯이 eBPF는 네트워킹 플랫폼에 프로그래밍 가능성을 추가하는 강력한 도구이며 XDP 후크를 통해 리눅스 환경에서 빠른 패킷 처리를 달성할 수 있는 대안이다. 이 절에서는 eBPF를 널리 사용되는 일부 기술인 P4, 클릭, 넷맵 및 데이터 평면 개발 키트(DPDK)와 비교하여 프로그래밍 가능하고 고속 데이터 평면 환경에 어떻게 적합한지 논의한다

## 10.1 Programmable Data Planes

P4 언어[14]는 프로그래밍 가능한 스위치를 위한 맞춤형 데이터 평면의 정의를 가능하게 하도록 제안되었다.
널리 채택됨에 따라 P4 지원 디바이스는 물리적 스위치 및 가상 스위치 모두의 기능을 정의하면서 완전히 프로그래밍 가능한 코어 네트워크를 구성할 수 있다. 동일한 전면에서, BPFabric은 가상화 환경 또는 심지어 서버 중심 네트워크에 적용 가능한 프로그래밍 가능한 가상 스위치의 동작을 정의하기 위해 eBPF를 그 언어로 사용한다.
그러나 eBPF의 주요 강점은 통신의 가장자리에 적용하는 것이다.
리눅스 커널에서는 P4가 설계된 것과 다른 사용 사례인 통신 엔드포인트에서 데이터 평면 기능을 정의할 수 있다. 스위칭과 라우팅은 P4에 의해 커버되는 코어 네트워크에 의해 수행되지만 네트워크 프로토콜 스택의 상당 부분은 엔드포인트에서 구현된다. eBPF를 사용하면 이 남은 부분을 모니터링하고 수정 및 재구성할 수 있으므로 완전한 네트워크 프로그래밍 가능성을 제공하는 데 필수적인 도구가 된다.
[37]을 클릭하면 리눅스 커널 내부에서 패킷을 처리할 수 있는 사용자 정의 네트워크 요소를 만들 수도 있다.
응용 프로그램은 요소라고 하는 많은 구성 요소의 구성을 통해 생성될 수 있으며, 이는 클릭 특정 함수 호출을 사용하여 C++ 코드로도 구현될 수도 있다.
클릭 응용 프로그램의 정의는 사용되는 요소의 종류와 이들 간의 상호 연결을 지정하는 구성 파일을 통해 이루어지며, 패킷 처리 그래프를 나타낸다. 이 사양은 이후 사용자 공간이나 커널로 컴파일될 수 있다. 클릭 커널 모듈은 네트워크 장치에 가까운 패킷을 캡처하고 eBPF로 XDP 후크에서 수행할 수 있는 것과 유사하게 ToHost 요소를 사용하여 커널 스택으로 전달할 수 있다.
Li et al. [38] build on Click을 사용하여 FPGA에서 네트워크 기능을 생성하고 실행할 수 있는 ClickNP 프레임워크를 만들어 모든 패킷 크기에 대해 40Gbps 처리량을 달성한다.
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

## 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[42] 및 NetMap[58]과 같은 잘 알려진 기술은 폴 모드에서 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 후크에서 맞춤형 패킷 처리를 허용함으로써 가능하다.
이 후크를 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 회피한다.
DPDK에 비해 XDP 후크의 장점 중 하나는 커널에 존재하는 기존 네트워크 설비(예: 라우팅 테이블)를 사용할 수 있다는 것인데, 이는 DPDK의 경우와 같이 사용자 공간에서 처리를 할 때 처음부터 다시 구현해야 한다. 또한 XDP는 커널에서 처리하기 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 OS에서 볼 수 있는 NIC를 완전히 제어할 필요가 없기 때문에 디바이스 공유도 용이하다. 동시에 OS의 추상화 계층 아래에 처리가 숨겨져 있기 때문에 프로그램은 호스트의 다른 응용 프로그램에 대해 투명하다.
리소스 사용 측면에서 이벤트 기반 특성 때문에 비지 폴링으로 소중한 CPU 사이클을 투자하는 것을 피한다.
최근 두 기술 간의 성능 비교에 따르면 DPDK는 여전히 더 높은 대역폭에 도달하지만(5개의 코어로 패킷을 드롭할 때 XDP의 경우 100Mpps에 비해 115Mpps) XDP보다 훨씬 높은 CPU 사용률을 보인다[31].
마지막으로, 프로그램들은 원자적(atomically..?)으로 대체될 수 있으며,
패킷 처리에 대한 온디맨드 변경을 위한 더 큰 유연성을 제공한다.
따라서, eBPF 및 DPDK는 본 섹션 및 섹션 9에서 논의된 바와 같이, 그들 자신의 제한 및 이점들 모두를 갖는 고속 패킷 처리에 대한 상이한 접근법들을 제공한다.
비록 둘 모두의 통합이 DPDK 상의 AF_XDP 소켓들 및 대응하는 폴 모드 드라이버를 사용함으로써 또는 DPDK에 의해 제공되는 librtebpf 라이브러리를 사용함으로써 달성될 수 있지만, 실제 사용 사례에 따라 기술의 최상의 선택이 달라질 수 있다[23].

## 11 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 후크에 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.

## REFERENCES


[1] 2019. bpf: Increase Complexity Limit and Maximum Program Size. Retrieved from https://git.kernel.org/pub/scm/linux/kernel/git/davem/net-next.git/commit/?id=c04c0d2b968ac45d6ef020316808ef6c82325a82.

[2] 2019. libbpf Unification and Golang Bindings. Discussion Summary, Linux Kernel Developers' bpfconf 2019. Retrieved from http://vger.kernel.org/bpfconf2019.html#session-4.
[3] 2019. XDP Project Repository. Retrieved from https://github.com/xdp-project/xdp-project.
[4] Ahmed Abdelsalam, Francois Clad, Clarence Filsfils, Stefano Salsano, Giuseppe Siracusano, and Luca Veltri. 2017. Implementation of virtual network function chaining through segment routing in a linux-based NFV infrastructure. In Proceedings of the 2017 IEEE Conference on Network Softwarization: Softwarization Sustaining a Hyper-Connected World: en Route to 5G (NetSoft'17). IEEE, Los Alamitos, CA, 1-5.
DOI:https://doi.org/10.1109/NETSOFT.2017.8004208arXiv:1702.05157

[5] Zaafar Ahmed, Muhammad Hamad Alizai, and Affan A. Syed. 2018. InKeV: In-kernel distributed network virtual-ization for DCN. SIGCOMM Comput. Commun. Rev. 46, 3, Article 4 (Jul. 2018), 6 pages.
DOI:https://doi.org/10.1145/3243157.3243161
[6] S. Baidya, Y. Chen, and M. Levorato. 2018. eBPF-based content and computation-aware communication for real-time edge computing. In Proceedings of the INFOCOM IEEE Conference on Computer Communications Workshops (INFOCOMWORKSHOPS'18). IEEE, Los Alamitos, CA, 865-870.
DOI:https://doi.org/10.1109/INFCOMW.2018.8407006
[7] BCC. 2019. BPF Compiler Collection. Retrieved from https://github.com/iovisor/bcc.
[8] BCC. 2019. BPF Program Types. Retrieved from https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md#program-types.
[9] BCC. 2019. XDP Compatible Drivers. Retrieved from https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md#xdp.
[10] David Beckett, Jaco Joubert, and Simon Horman. 2018. Host Dataplane Acceleration (HDA).
[11] Gilberto Bertin. 2017. XDP in practice: Integrating XDP into our DDoS mitigation pipeline. In Proceedings of the Netdev 2.1 Technical Conference on Linux Networking. 1-5.
[12] Matteo Bertrone, Sebastiano Miano, Fulvio Risso, and Massimo Tumolo. 2018. Accelerating linux security with eBPF iptables. In Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos (SIGCOMM'18). ACM, New York, NY, 108-110.
DOI:https://doi.org/10.1145/3234200.3234228
[13] Daniel Borkmann. 2019. bpf, Libbpf: Support Global Data/bss/rodata Sections. Retrieved from https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=d859900c4c56.
[14] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, 1136 Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming protocol-independent packet processors. SIGCOMM Comput. Commun. Rev. 44, 3 (Jul. 2014), 87-95.
DOI:https://doi.org/10.1145/2656877.2656890
[15] Autores bpftool. 2018. Manual Bpftool. Retrieved from https://elixir.bootlin.com/linux/v4.18-rc1/source/tools/bpf/bpftool/Documentation/bpftool.rst.
[16] bpftrace. 2019. High-level Tracing Language for Linux eBPF. Retrieved from https://github.com/iovisor/bpftrace.
[17] Mihai Budiu. 2015. Compiling p4 to ebpf. Retrieved from https://github.com/iovisor/bcc/tree/master/src/cc/frontends/p4.
[18] Cilium. 2018. Cilium 1.0: Bringing the BPF Revolution to Kubernetes Networking and Security. Retrieved from

https://cilium.io/blog/2018/04/24/cilium-10/.
[19] Cilium. 2019. BPF and XDP Reference Guide. Retrieved September 9, 2019 from https://cilium.readthedocs.io/en/latest/bpf/.

[20] Cilium. 2019. Cilium: API-aware Networking and Security. Retrieved September 9, 2019 from https://cilium.io/.
[21] Jonathan Corbet. 2014. BPF: The Universal In-kernel Virtual Machine. Retrieved from https://lwn.net/Articles/599755/.
[22] DPDK. 2019. AF_XDP Poll Mode Driver. Retrieved from https://doc.dpdk.org/guides/nics/af_xdp.html.
[23] DPDK. 2019. Berkeley Packet Filter Library. Retrieved from https://doc.dpdk.org/guides/prog_guide/bpf_lib.html.
[24] Fabien Duchene, Mathieu Jadin, and Olivier Bonaventure. 2018. Exploring various use cases for IPv6 segment routing.In Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos (SIGCOMM'18). ACM, New York, NY, 129-131. DOI:https://doi.org/10.1145/3234200.3234213
[25] Eric Dumazet. 2011. A JIT for Packet Filters. Retrieved from https://lwn.net/Articles/437981/.
[26] Facebook. 2018. Katran Source Code Repository. Retrieved from https://github.com/facebookincubator/katran.
[27] John Fastabend. 2018. [RFC PATCH 00/16] bpf, Bounded Loop Support Work in Progress. Retrieved from https://lwn.net/ml/netdev/20180601092646.15353.28269.stgit@john-Precision-Tower-5810/.
[28] Nick Feamster, Jennifer Rexford, and Ellen Zegura. 2014. The road to SDN: An intellectual history of programmable networks. ACM SIGCOMM Comput. Commun. Rev. 44, 2 (2014), 87-98.
[29] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir. 2018. Segment Routing Architecture. RFC 8402. RFC Editor.
[30] gobpf. 2019. Go Bindings for Creating BPF Programs. Retrieved from https://github.com/iovisor/gobpf.
[31] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The eXpress data path: Fast programmable packet processing in the operating system kernel. In Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT'18). ACM, New York, NY, 54-66. DOI:https://doi.org/10.1145/3281411.3281443
[32] IOvisor. 2019. Iovisor Project. Retrieved March 29, 2019 from www.iovisor.org.
[33] S. Jouet, R. Cziva, and D. P. Pezaros. 2015. Arbitrary packet matching in OpenFlow. In Proceedings of the 16th Interח1170 national Conference on High Performance Switching and Routing (HPSR'15). IEEE, Los Alamitos, CA, 1-6. DOI:https://doi.org/10.1109/HPSR.2015.7483106