시스템 프로그래밍 과제 2

2020320135 김윤서 (2 free days used)

1. Netfilter

개요 및 주요 특징

Netfilter는 리눅스 커널에 내장된 패킷 필터링 프레임워크로, 패킷 수신부터 전송까지의 경로 중 여러 지점에 개입하여 필터링, 주소 변환(NAT), 로깅, 리디렉션 등의 다양한 작업을 수행할 수 있도록 지원한다. 기존의 iptables, nftables와 같은 사용자 공간 도구들이 Netfilter 프레임워크를 기반으로 작동하며, 이와 별개로 커널 모듈을 직접 구현하여 고유한 패킷 처리 로직을 삽입하는 것도 가능하다.

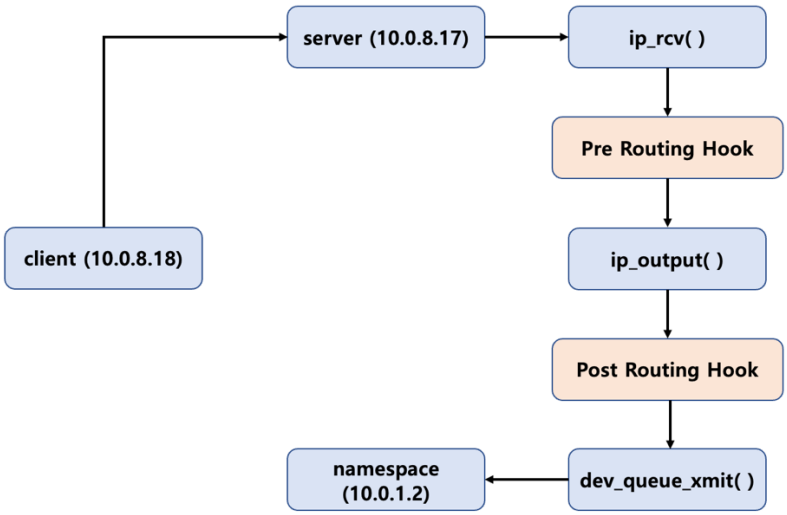
Hook Point 설명

Hook Point	위치	설명
PREROUTING	수신 패킷이 라우팅 결정 전에 도착	목적지 주소 기반 라우팅 이전에 패킷 처리 (예: NAT)
INPUT	로컬 소켓으로 들어가기 전	대상이 로컬 시스템인 패킷 처리
FORWARD	패킷이 다른 인터페이스로 전달될 때	라우팅된 패킷 중 로컬 목적지가 아닌 경우 처리
OUTPUT	로컬 프로세스에서 생성한 패킷	로컬에서 생성된 패킷이 외부로 나가기 전 처리
POSTROUTING	라우팅 후 실제 전송 직전	소스 NAT 등의 처리를 위해 전송 전 마지막 단계에서 개입

커널 모듈로서의 동작 방식

커널 모듈로 Netfilter를 활용할 경우, nf_register_net_hook() 함수를 통해 원하는 Hook Point에 콜백 함수를 등록한다. 이때 사용하는 nf_hook_ops 구조체는 우선순위, 프로토콜, 훅 지점, 그리고 처리 함수에 대한 정보를 담고 있다. 등록된 콜백 함수는 패킷이 해당 지점에 도달할 때마다 호출되며, 반환값을 통해 패킷의 운명을 결정한다. 패킷을 그대로 전달할지(ACCEPT), 삭제할지(DROP), 사용자 공간으로 넘길지(QUEUE), 혹은 다른 곳에서 처리하게 할지(STOLEN) 등을 설정할 수 있다. 이러한 구조는 모듈 단위의 유연한 패킷 처리를 가능하게 하지만, 동시에 성능 상의 몇 가지 제약도 존재한다.

Routing 파이프라인'



성능적 특성과 한계

Netfilter는 IP 계층 이후에 개입하는 구조이기 때문에 이미 커널 네트워크 스택의 일정 부분을 통과한 이후에야 처리가 가능하다. 이는 불필요한 패킷에 대해서도 CPU 자원이 낭비될 수 있다는 의미이며, 초당 수만 개 이상의 패킷이 발생하는 고성능 환경에서는 이 오버헤드가 무시하기 어렵다. 특히 NFQUEUE와 같이 사용자 공간과 커널 공간 간의 컨텍스트 스위칭이 발생하는 모드는 지연(latency)을 유발할 수 있으며, 실시간성이 요구되는 시스템에는 적합하지 않다. 또한 Netfilter는 early drop이 불가능하기 때문에, 악성 패킷을 완전히 커널에 진입시키기 전에 차단하는 구조를 가지지 못한다는 단점도 존재한다.

이러한 한계로 인해 최근에는 Netfilter보다 하위 계층에서, 즉 네트워크 드라이버 수준에서 바로 패킷을 처리할 수 있는 eBPF 기반의 XDP가 대안으로 주목받고 있다. Netfilter는 구조적으로는 완성도가 높고 범용적인 인터페이스를 제공하지만, 성능 최적화가 중요한 상황에서는 더 낮은 계층에서의 처리 기술이 요구된다. 본 프로젝트에서는 이러한 관점에서 Netfilter 기반 리디렉션과 eBPF(XDP) 기반 리디렉션의 성능 차이를 비교하고, 실질적인 CPU 사용률과 처리 효율성을 분석하고자 한다.

2. eBPF(Extended Berkeley Packet Filter)

개요 및 주요특징

eBPF(Extended Berkeley Packet Filter)는 리눅스 커널 내에서 사용자가 정의한 프로그램을 커널 공간에서 안전하게 실행할 수 있도록 하는 고성능의 확장 가능한 프레임워크이다. 원래는 패킷 필터링 용도로 설계되었지만, 최근에는 트레이싱, 모니터링, 보안, 시스템 콜 필터링 등 다양한 영역으로 확장되며 커널 기능을 사용자 정의 코드로 확장할 수 있는 강력한 도구로 자리잡았다.

기존에는 커널 기능을 변경하려면 커널 모듈을 직접 작성하거나 패치를 통해 커널을 재컴파일해야 했지만, eBPF 는 이러한 과정 없이도 런타임 시점에 검증된 코드를 삽입할 수 있어 유연성과 안정성을 동시에 갖춘다. 특히, 네트워킹 분야에서는 XDP 와 결합되어 드라이버 계층에서 빠르게 패킷을 처리할 수 있어 Netfilter 를 대체하는 기술로 주목받고 있다. eBPF 의 가장 큰 특징 중 하나는 커널 내의 특정 Hook Point 에 프로그램을 동적으로 attach 할 수 있다는 점이다. 이러한 프로그램은 커널에 의해 미리 정의된 인터페이스에 맞게 작성되어야 하며, eBPF verifier 에 의해 안전성이 검증된 뒤에 실행된다.

eBPF 구성요소

구성 요소	설명
eBPF 프로그램	BPF bytecode 로 컴파일되어 커널에 로드되는 로직
Verifier	커널 내에서 프로그램의 안정성, 루프 제한, 메모리 접근 안전성 등을 검증
Helper 함수	커널 내부 기능 호출을 위한 인터페이스
Map	커널과 사용자 공간 간 공유 가능한 데이터 구조 (hash, array 등)
Attach Point	프로그램이 연결되는 커널의 이벤트 지점 (e.g., XDP, tc, tracepoint 등)

커널 내 동작방식

eBPF 프로그램은 일반적으로 C 언어로 작성된 뒤, LLVM 과 Clang 을 사용하여 BPF 바이트코드로 컴파일된다. 이 바이트코드는 리눅스 커널에 로드되어 실행되며, 로드 시점에서 커널 내부의 Verifier 에 의해 철저한 정적 분석을 거친다. Verifier 는 프로그램의 안전성과 안정성을 보장하기 위한 핵심 구성 요소로, 모든 실행 경로에 종료 조건이 있는지, 루프가 제한되어 있는지, 포인터나 메모리 접근이 안전하게 이루어지는지를 검사한다.

검증에 통과한 프로그램은 커널 내부의 특정 이벤트 지점, 즉 Attach Point 에 연결된다. 이 attach 지점은 XDP, tc, tracepoint, kprobe, cgroup 등 다양한 위치일 수 있으며, 해당 이벤트가 발생할 때마다 eBPF 프로그램이 자동으로 실행된다. 예를 들어, 네트워크 패킷 수신 시점의 드라이버 계층에 attach 된 eBPF 프로그램은 패킷 도착과 동시에 실행되어, drop, redirect, pass 등의 처리를 수행할 수 있다.

eBPF 프로그램은 단순히 커널 내부 상태를 읽거나 쓰는 것이 아니라, Helper 함수를 호출하여 다양한 커널 기능에 접근할 수 있으며, Map 이라는 특수한 자료구조를 사용하여 사용자 공간과 데이터를 공유하거나 상태를 저장할 수 있다. Map 은 해시 테이블, 배열, LRU 캐시 등 다양한 형태를 지원하며, 사용자 공간에서도 실시간으로 접근 가능하기 때문에, eBPF 를 통한 네트워크 모니터링이나 정책 제어가 효율적으로 이루어진다. 결과적으로 eBPF 는 사용자 정의 코드를 안전하게 커널 공간에서 실행할 수 있도록 하여, 기존의 커널 모듈 방식보다 훨씬 유연하고 경량화된 방식으로 시스템 동작을 확장할 수 있게 한다. 특히 네트워크 분야에서는 eBPF 프로그램을 통해 패킷 처리, 로깅, 리디렉션 등을 고성능으로 수행할 수 있어, XDP 와 결합되어 커널 네트워크 처리 성능의 새로운 패러다임을 제시하고 있다.

성능적 특성과 장점

eBPF 는 커널 공간에서 직접 실행되기 때문에 context switching 비용 없이 높은 성능을 달성할 수 있다. 특히 XDP 와 함께 사용할 경우, 네트워크 드라이버에서 패킷을 수신하자마자 처리할 수 있으므로, 불필요한 커널 경로를 거치지 않고도 패킷을 드롭하거나 리디렉션할 수 있다. 이는 Netfilter 대비 빠른 응답성과 낮은 CPU 사용률을 보장한다.

또한, eBPF 는 프로그램의 로딩과 실행이 분리되어 있어 동적으로 수정 및 업데이트가 가능하며, 사용자 공간에서 map 을 통해 프로그램 상태를 실시간으로 확인하고 제어할 수 있다. 이러한 유연성과 성능 덕분에 대규모 트래픽 환경이나 실시간 네트워크 제어가 필요한 상황에서 매우 적합하다.

3. XDP(eXpress Data Path)

개요 및 주요특징

XDP(eXpress Data Path)는 리눅스 커널에서 eBPF 프로그램을 가장 낮은 네트워크 계층, 즉 네트워크 드라이버 레벨에서 직접 실행할 수 있도록 설계된 고성능 패킷 처리 프레임워크이다. 기존 Netfilter 나 tc 계층에서는 패킷이 이미 커널 네트워크 스택의 일부를 통과한 이후에야 필터링이나 리디렉션이 가능했으나, XDP 는 NIC(네트워크 인터페이스 카드)와 드라이버 사이에서 바로 실행되므로 패킷을 조기에 처리하고 불필요한 커널 리소스를 사용하지 않는 것이 가능하다.

XDP 는 주로 고속 패킷 필터링, DDoS 방어, 로드밸런싱, 리디렉션 등 초고성능 네트워크 처리가 요구되는 분야에서 사용된다. 패킷을 수신하자마자 즉시 드롭하거나 특정 인터페이스로 리디렉션함으로써 사용자 공간에 도달하지 않고도 처리를 완료할 수 있어, CPU 사용률 절감 및 처리 지연(latency) 감소 측면에서 매우 효과적이다.

XDP 는 eBPF 를 기반으로 구현되며, 프로그램은 C 로 작성 후 LLVM/Clang 으로 컴파일되어 BPF 바이트코드로 변환된다. 로딩된 코드는 NIC 드라이버 수준에서 attach 되며, Verifier 를 거쳐 안전성이 보장된 경우에만 실행된다. 네트워크 트래픽의 전처리 단계에서 빠르고 정확하게 개입할 수 있다는 점에서 Netfilter 대비 성능상 큰 이점을 가진다.

XDP 모드별 차이

XDP 모드	위치	장점	단점 및 제약
Native	NIC 드라이버 내부	가장 빠름, context switching 없음	드라이버 지원 필요, 디버깅 난이도 높음
Generic	커널의 softnet 계층 이후	대부분의 NIC 에서 작동 가능	성능 제한, SKB 생성 이후라 early drop 불가
Offload	NIC 자체에 BPF offload	CPU 부담 없음, 하드웨어 처리	NIC 하드웨어 지원 필요, 구현 복잡

XDP 동작 흐름 및 처리 방식

XDP 는 네트워크 인터페이스 카드(NIC)에서 패킷을 수신하는 즉시, 커널 네트워크 스택에 진입하기 전에 패킷을 가로채어 처리하는 방식으로 동작한다. 이는 기존의 Netfilter 나 tc 계층보다 훨씬 앞선 지점에서 패킷을 처리할 수 있도록 하여, 불필요한 커널 경로를 차단하고 처리 효율을 극대화하는 데에 목적이 있다. 사용자는 C 언어로 작성된 eBPF 프로그램을 XDP 에 attach 하여 패킷의 처리를 제어할 수 있으며, 프로그램 내부에서 특정 반환값을 통해 패킷의 처리 방향을 지정하게 된다.

가장 대표적인 반환값은 XDP_DROP 으로, 이는 수신된 패킷을 즉시 폐기하여 커널 스택에 도달하지 않도록 하는 방식이다. 이는 DDoS 방어와 같이 불필요하거나 악의적인 트래픽을 빠르게 차단할 때 효과적이다. 반면, XDP_PASS 는 패킷을 일반적인 커널 네트워크 스택으로 전달하여 기존 TCP/IP 처리 경로를 그대로 따르도록 한다. XDP_TX 는 수신한 패킷을 같은 인터페이스를 통해 그대로 반송하는 처리 방식으로, 에코 서버와 같은 구조에 유용하다. 마지막으로 XDP_REDIRECT 는 패킷을 특정한 다른 인터페이스나 네임스페이스로 전달하는 것으로, 로드밸런싱이나 네트워크 리디렉션 용도에 사용된다.

이러한 동작 방식은 단순한 패킷 필터링을 넘어 실시간 트래픽 제어, 경량화된 네트워크 스위칭, 그리고 커널 개입 최소화를 통한 고성능 처리를 가능하게 한다. 특히, 패킷이 커널 버퍼에 복사되기 전 처리되기 때문에 불필요한 메모리 접근도 줄일 수 있으며, CPU 사용률 절감에도 크게 기여한다. 본 프로젝트에서는 이러한 XDP 의 특성을 활용하여, 클라이언트에서 서버로 전송되는 패킷을 XDP 프로그램을 통해 네임스페이스 내부 인터페이스로 직접 리디렉션하고, 이를 Netfilter 방식과 비교하여 처리 효율과 자원 사용 측면에서의 차이를 실험적으로 분석한다.

장점 및 활용 사례

XDP 의 가장 큰 장점은 패킷 처리의 최전방 위치에서 동작하기 때문에 지연을 최소화하고, CPU 오버헤드를 획기적으로 줄일 수 있다는 점이다. 전통적인 커널 경로를 거치지 않고도 필요한 처리를 빠르게 수행할 수 있어 DDoS 방어, 대용량 로드밸런싱, 인라인 패킷 수정 등의 분야에서 널리 사용된다.

또한, bpf_redirect() 등의 helper 함수를 통해 다른 인터페이스나 네임스페이스로 패킷을 전송할 수 있으며, 이를 통해 실시간 리디렉션 기능 구현이 가능하다. 단, Native 모드에서는 드라이버 지원이 필요하고 디버깅이 어려운 점이 있으므로 실험 환경에서는 이를 고려한 설계가 요구된다.

본 프로젝트에서는 eBPF XDP 를 사용하여 클라이언트에서 수신된 패킷을 직접 네임스페이스 내의 서버 인터페이스로 리디렉션하고, Netfilter 기반 구현과 CPU 자원 소모 측면에서 비교 분석을 수행한다.

소스코드

1. Netfilter - redir_module.c

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
#include <linux/ip.h>
#include <linux/udp.h>
#include <linux/skbuff.h>
#include <linux/netdevice.h>

static unsigned int prerouting_hook(void *priv, struct sk_buff *skb, const struct nf_hook_state *state);
static unsigned int postrouting_hook(void *priv, struct sk_buff *skb, const struct nf_hook_state *state);


// 원래 목적지: 10.0.8.17:8080
// 리다이렉션: 10.0.1.2:8083
#define TARGET_IP   0x0A000811  // 10.0.8.17 (hex: 0A000811)
#define REDIR_IP    0x0A000102  // 10.0.1.2  (hex: 0A000102)
#define TARGET_PORT 8080
#define REDIR_PORT  8083

static void calculate_ip_checksum(struct iphdr *iph)
{
    unsigned short *ip_header = (unsigned short *)iph;
    unsigned int checksum = 0;
    int i;

    iph->check = 0;

    for (i = 0; i < sizeof(struct iphdr) / 2; i++)
        checksum += ntohs(ip_header[i]);

    checksum = (checksum & 0xFFFF) + (checksum >> 16);
    checksum += (checksum >> 16);

    iph->check = htons(~checksum);
}
```

설명

이 코드는 Netfilter 훅 함수인 prerouting_hook과 postrouting_hook의 선언부를 포함하고 있으며, 네트워크 패킷의 PREROUTING과 POSTROUTING 단계에서 호출될 함수들이다. 리디렉션을 위한 기준으로 TARGET_IP와 TARGET_PORT가 정의되어 있으며, 이는 원래 목적지인 10.0.8.17:8080을 나타낸다. 변경될 목적지인 10.0.1.2:8083은 REDIR_IP와 REDIR_PORT로 정의되어 있다.

또한 calculate_ip_checksum 함수는 IP 헤더의 체크섬을 재계산하는 함수로, IP 헤더 내용을 16비트 단위로 더한 뒤 오버플로우를 정리하고 보수를 취해 새로운 체크섬 값을 계산한다. 이 함수는 IP 헤더를 수정한 후 정상적인 전송을 위해 반드시 호출되어야 한다.

```c
// PREROUTING 훅: 목적지 리다이렉션
static unsigned int prerouting_hook(void *priv, struct sk_buff *skb, const struct nf_hook_state *state) {
    struct iphdr *iph;
    struct udphdr *udph;
    unsigned int src_ip, dst_ip, src_port, dst_port;

    if (!skb)
    return NF_ACCEPT;

    iph = ip_hdr(skb);
    if (!iph || iph->protocol != IPPROTO_UDP)
    return NF_ACCEPT;

    udph = (struct udphdr *)((__u32 *)iph + iph->ihl);
    if (!udph)
    return NF_ACCEPT;

    src_ip = ntohl(iph->saddr);
    dst_ip = ntohl(iph->daddr);
    src_port = ntohs(udph->source);
    dst_port = ntohs(udph->dest);

    if (iph->daddr == htonl(TARGET_IP) && udph->dest == htons(TARGET_PORT)) {
        printk(KERN_INFO "[NETFILTER] FORWARD:  UDP:%pI4:%u;%pI4:%u\n", &iph->saddr, src_port, &iph->daddr, dst_port);

        iph->daddr = htonl(REDIR_IP);
        udph->dest = htons(REDIR_PORT);

        udph->check = 0;
        skb->csum = 0;
        skb->ip_summed = CHECKSUM_NONE;
        calculate_ip_checksum(iph);

        printk(KERN_INFO "[NETFILTER] MODIFIED: UDP:%pI4:%u;%pI4:%u\n", &iph->saddr, src_port, &iph->daddr, REDIR_PORT);
        printk(KERN_INFO "[NETFILTER] FORWARDING: UDP:%pI4:%u -> %pI4:%u\n", &iph->saddr, src_port, &iph->daddr, REDIR_PORT);
    }

return NF_ACCEPT;
}
```

설명

이 코드는 Netfilter의 PREROUTING 훅에서 실행되는 함수로, 들어오는 UDP 패킷을 검사하고 특정 조건을 만족하면 목적지 IP와 포트를 변경한다. 먼저 skb가 유효한지 확인한 뒤, IP 헤더를 가져오고 프로토콜이 UDP인지 확인한다. 이후 UDP 헤더를 추출하고, 출발지와 목적지의 IP 및 포트 번호를 읽어온다.

만약 패킷의 목적지 IP와 포트가 TARGET_IP와 TARGET_PORT(10.0.8.17:8080)에 해당한다면, 해당 패킷의 목적지를 REDIR_IP와 REDIR_PORT(10.0.1.2:8083)로 수정한다. 이 과정에서 UDP 체크섬과 skb 관련 필드를 초기화하고, calculate_ip_checksum() 함수를 호출해 IP 체크섬을 다시 계산한다. 또한 패킷 수정 전과 후의 정보를 printk()를 통해 커널 로그로 출력하여 디버깅에 활용할 수 있도록 했다. 최종적으로 모든 패킷은 NF_ACCEPT를 반환하여 커널 네트워크 스택의 다음 단계로 계속 전달된다.

```c
// POSTROUTING 훅: 서버에서 응답 보낼 때
static unsigned int postrouting_hook(void *priv, struct sk_buff *skb, const struct nf_hook_state *state) {
    struct iphdr *iph;
    struct udphdr *udph;
    unsigned int src_ip, dst_ip, src_port, dst_port;

    if (!skb)
        return NF_ACCEPT;

    iph = ip_hdr(skb);
    if (!iph || iph->protocol != IPPROTO_UDP)
        return NF_ACCEPT;

    udph = (struct udphdr *)((__u32 *)iph + iph->ihl);
    if (!udph)
        return NF_ACCEPT;

    src_ip = ntohl(iph->saddr);
    dst_ip = ntohl(iph->daddr);
    src_port = ntohs(udph->source);
    dst_port = ntohs(udph->dest);

    // 서버에서 응답 (10.0.1.2 → 10.0.8.*)
    if (iph->saddr == htonl(REDIR_IP) && (iph->daddr & htonl(0xFFFFFF00)) == htonl(0x0A000800)) {
        printk(KERN_INFO "[NETFILTER] POSTROUTING: UDP:%pI4:%u;%pI4:%u\n", &iph->saddr, src_port, &iph->daddr, dst_port);
    }

    return NF_ACCEPT;
}
```

설명

이 코드는 Netfilter의 POSTROUTING 훅에서 실행되는 함수로, 서버가 클라이언트로 응답을 보낼 때 패킷 정보를 확인하는 용도로 사용된다. 먼저 skb가 유효한지 검사한 후, IP 헤더를 추출하고 프로토콜이 UDP인지 확인한다. 그 후 UDP 헤더를 얻고, 출발지와 목적지의 IP 및 포트 번호를 가져온다.

패킷의 출발지 IP가 REDIR_IP(10.0.1.2)이고, 목적지 IP가 10.0.8.0/24 대역에 속하는 경우에만 printk()를 통해 패킷의 송수신 정보를 커널 로그에 출력한다. 이는 리디렉션된 서버 응답이 클라이언트로 나가는 과정을 추적하는 데 활용된다. 이 함수는 패킷의 내용을 변경하지 않고 모든 패킷을 그대로 통과시키며, NF_ACCEPT를 반환하여 이후 처리를 계속 진행시킨다.

```c
// Netfilter 등록
static struct nf_hook_ops nf_ops[] = {
    {
        .hook     = prerouting_hook,
        .hooknum  = NF_INET_PRE_ROUTING,
        .pf       = PF_INET,
        .priority = NF_IP_PRI_FIRST
    },
    {
        .hook     = postrouting_hook,
        .hooknum  = NF_INET_POST_ROUTING,
        .pf       = PF_INET,
        .priority = NF_IP_PRI_FIRST
    }
};

static int __init redir_init(void) {
    nf_register_net_hooks(&init_net, nf_ops, ARRAY_SIZE(nf_ops));
    printk(KERN_INFO "[NETFILTER] 리다이렉션 모듈 로딩됨\n");
    return 0;
}

static void __exit redir_exit(void) {
    nf_unregister_net_hooks(&init_net, nf_ops, ARRAY_SIZE(nf_ops));
    printk(KERN_INFO "[NETFILTER] 리다이렉션 모듈 언로드됨\n");
}

module_init(redir_init);
module_exit(redir_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("KYS");
MODULE_DESCRIPTION("Netfilter UDP Redirect Module");
```

설명

이 코드는 Netfilter 훅 함수를 커널에 등록하고 해제하는 역할을 하는 커널 모듈 초기화 및 종료 함수이다. nf_hook_ops 구조체 배열인 nf_ops에는 두 개의 훅이 정의되어 있으며, 각각 PREROUTING과 POSTROUTING 위치에 prerouting_hook과 postrouting_hook 함수를 등록한다. 이들은 IPv4 프로토콜에 대해 가장 높은 우선순위로 동작하도록 설정되어 있다.

모듈이 로드될 때 호출되는 redir_init() 함수는 nf_register_net_hooks()를 통해 위의 훅들을 커널에 등록하며, 모듈 로딩 메시지를 커널 로그에 출력한다. 반대로, 모듈이 언로드될 때 호출되는 redir_exit() 함수는 nf_unregister_net_hooks()를 호출하여 등록된 훅을 해제하고, 언로드 메시지를 출력한다.

마지막으로 module_init()과 module_exit() 매크로를 통해 커널이 해당 초기화 및 종료 함수를 자동으로 인식하도록 설정하며, 모듈 정보에는 라이선스, 작성자, 설명이 포함되어 있다. 이 구조를 통해 Netfilter 기반의 리다이렉션 기능을 커널 모듈 형태로 간편하게 추가하고 제거할 수 있다.

2. eBPF XDP – redir_ebpf.c

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

#define htons(x)    __builtin_bswap16(x)
#define htonl(x)    __builtin_bswap32(x)

#define ETH_P_IP 0x0800   // IPv4

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

    struct ethhdr *eth = data;
    if ((void*)(eth + 1) > data_end) return XDP_PASS;

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

    struct iphdr *ip = (void *)(eth + 1);
    if ((void*)(ip + 1) > data_end) return XDP_PASS;

    if (ip->protocol != IPPROTO_UDP)
        return XDP_PASS;

    struct udphdr *udp = (void *)ip + (ip->ihl * 4);
    if ((void*)(udp + 1) > data_end) return XDP_PASS;

    __u16 orig_port = udp->dest;
    __u32 orig_daddr = ip->daddr;

    // 목적 포트가 8080이 아닌 경우 무시
    if (orig_port != htons(8080))
        return XDP_PASS;

    // 원래 목적지 로그 출력
    bpf_printk("Original: dst_ip=%x, dst_port=%d", orig_daddr, __builtin_bswap16(orig_port));
```

설명

이 코드는 XDP 프로그램으로, 네트워크 인터페이스 카드(NIC)에서 수신된 패킷을 커널 네트워크 스택으로 전달되기 전에 처리하기 위해 작성되었다. SEC("xdp")는 해당 함수가 XDP Hook에 연결된다는 것을 나타낸다. 먼저 ctx로부터 패킷의 시작 주소(data)와 끝 주소(data_end)를 얻고, 이를 기반으로 패킷 내부의 이더넷 헤더, IP 헤더, UDP 헤더에 접근한다. 이때 각 헤더에 접근하기 전에 data_end를 넘지 않는지 확인하여 out-of-bound 접근을 방지한다.

이더넷 헤더에서 EtherType이 IPv4(0x0800)인지 확인하고, IP 헤더의 프로토콜이 UDP인지 검사한다. 조건을 만족할 경우, IP 헤더의 길이를 고려해 UDP 헤더 위치를 계산하고, UDP 헤더도 유효한지 확인한다. 그 후 목적지 포트를 확인하여 8080 포트가 아닌 경우에는 XDP_PASS를 반환해 패킷을 그대로 통과시킨다. 만약 포트가 8080이라면, 원래의 목적지 IP와 포트를 bpf_printk를 통해 커널 로그로 출력한다.

```
    // 포트, IP 수정
    udp->dest = htons(8083);
    ip->daddr = htonl(0x0A000102);  // 10.0.1.2

    // UDP 체크섬 무효화
    udp->check = 0;

    // IP 체크섬 재계산
    ip->check = 0;
    __u32 csum = 0;
    __u16 *ip_hdr = (__u16 *)ip;
    for (int i = 0; i < sizeof(*ip) / 2; i++)
        csum += ip_hdr[i];
    while (csum >> 16)
        csum = (csum & 0xFFFF) + (csum >> 16);
    ip->check = ~csum;

    // 변경된 값 로그 출력
    bpf_printk("Modified: dst_ip=%x, dst_port=%d, ip_csum=0x%x", ip->daddr, __builtin_bswap16(udp->dest), ip->check);

    return XDP_PASS;
}

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

설명

이 부분에서는 UDP 패킷의 목적지 정보를 직접 수정하여 리디렉션을 수행한다. 먼저, UDP 헤더의 목적지 포트를 8083으로 변경하고, IP 헤더의 목적지 주소도 0x0A000102(10.0.1.2)로 수정한다. 이후 UDP 체크섬은 0으로 설정해 무효화하며, IP 헤더의 체크섬은 수동으로 재계산한다.

IP 체크섬 재계산은 16비트 단위로 IP 헤더를 순회하며 합산한 뒤, 오버플로우를 처리하여 최종적으로 보수 연산을 통해 설정한다. 변경된 목적지 IP, 포트, 재계산된 IP 체크섬은 bpf_printk를 통해 커널 로그로 출력된다.

마지막으로 XDP_PASS를 반환하여 수정된 패킷을 커널 네트워크 스택으로 정상적으로 전달한다. 라이선스는 "GPL"로 명시되어 있어, 이 프로그램이 커널에 로드 가능한 오픈소스 eBPF 프로그램임을 나타낸다.

실험 설명

클라이언트부터 서버까지의 패킷 경로

본 실험에서 클라이언트는 10.0.8.18 IP를 가진 사용자 프로세스로, 10.0.8.17:8080으로 데이터를 전송하도록 구성되어 있다. 하지만 이 목적지는 실제 서버가 아니며, Netfilter 또는 XDP 프로그램을 통해 목적지가 10.0.1.2:8083인 내부 네임스페이스 서버로 리디렉션되도록 설정되어 있다.

Netfilter 실험의 경우, 커널 모듈이 PREROUTING 훅에서 패킷을 가로채어 목적지 IP와 포트를 변경하고, POSTROUTING 훅에서는 서버의 응답 패킷을 로그로 기록한다. 이때 리디렉션된 패킷은 커널 네트워크 스택을 통과한 뒤 veth 인터페이스를 통해 네임스페이스 내부 서버에 도달한다.

XDP 실험의 경우에는, 패킷이 NIC를 통해 수신되자마자 드라이버 수준에서 eBPF 프로그램이 실행되어 목적지 정보를 수정한다. 수정된 패킷은 커널 스택을 거쳐 마찬가지로 네임스페이스 내부의 서버로 전달된다. 서버는 응답을 생성하여 다시 클라이언트 쪽으로 송신하게 되고, 이때 응답은 별도로 수정되지 않으며 그대로 전송된다.

결과적으로, 두 방식 모두 클라이언트는 10.0.8.17:8080에 데이터를 보냈다고 인식하지만, 실제 패킷은 10.0.1.2:8083에 위치한 서버로 전달되어 처리된다.

실험 측정 및 비교 대상

본 실험에서는 클라이언트로부터 서버까지 전송되는 UDP 트래픽에 대해 두 가지 리디렉션 방식, 즉 Netfilter 기반 방식과 XDP 기반 방식을 적용한 후, 각각의 CPU 사용률을 비교 분석한다. 이를 통해 두 방식의 처리 효율성과 시스템 자원 소모 측면에서의 차이를 확인하는 것이 실험의 주요 목적이다.

실험은 동일한 조건 하에서 진행된다. 클라이언트는 약 512MB의 파일을 전송하며, 리디렉션 방식만 Netfilter 또는 XDP로 각각 변경한다. 클라이언트는 항상 10.0.8.17:8080을 목적지로 설정하지만, 리디렉션 로직을 통해 실제 서버는 10.0.1.2:8083으로 변경된다. 이를 통해 사용자 입장에서의 네트워크 흐름은 동일하게 유지되면서도, 내부적으로 서로 다른 리디렉션 방식이 적용된다.

CPU 사용률은 mpstat를 이용해 측정되며, 일정 주기로 CPU의 전체 사용량, 사용자 영역(%usr), 커널 영역(%sys), 유휴 시간(%idle) 등을 기록한다. 측정은 클라이언트 트래픽이 전송되는 전체 구간 동안 수행되며, 실험 결과로는 평균 CPU 사용률과 리디렉션 수행 중의 부하 양상이 주요 비교 지표가 된다. 특히 리디렉션 처리 위치에 따른 커널 영역의 CPU 점유율 변화가 실험의 핵심 분석 대상이다.

이러한 비교를 통해, 보다 낮은 계층에서 실행되는 XDP 방식이 리디렉션 과정에서 CPU 사용률을 얼마나 절감할 수 있는지를 정량적으로 파악하고자 한다. Netfilter는 IP 계층 이후에 패킷을 처리하기 때문에 커널 네트워크 스택의 여러 경로를 거치게 되며, 이로 인해 상대적으로 CPU 자원이 더 많이 소모될 수 있다. 반면 XDP는 네트워크 드라이버 단계에서 즉시 패킷을 처리하므로 오버헤드를 최소화할 수 있다는 이론을 실험을 통해 실증적으로 검증한다.

실험 환경

환경변수

본 실험은 UTM 가상 머신 환경에서 Ubuntu 리눅스 시스템을 기반으로 구성되었으며, 리눅스 커널 버전은 5.15.0-136-generic을 사용하였다. 실험 대상은 Netfilter 기반 커널 모듈과 eBPF 기반 XDP 프로그램으로, 동일한 네트워크 트래픽에 대해 두 방식의 패킷 리디렉션을 수행하고 이 과정에서의 CPU 자원 사용률을 비교하였다.

네트워크는 가상 인터페이스와 네임스페이스를 활용하여 구성되었다. 클라이언트는 10.0.8.18, 초기 목적지는 10.0.8.17:8080으로 설정되었으며, 실제 서버는 네임스페이스 내부에 존재하는 10.0.1.2:8083 주소를 사용하였다. 서버는 독립된 네트워크 네임스페이스(sslab-ns) 안에서 실행되며, 가상 인터페이스 veth-host와 veth-ns를 통해 호스트와 연결된다. 이때 setup_netsh.sh 스크립트를 통해 veth 페어 생성, IP 할당, 경로 설정 및 IP 포워딩 기능을 활성화하였다.

클라이언트는 Python 스크립트를 통해 약 512MB의 파일을 UDP 프로토콜로 전송하며, 서버는 수신 후 해당 데이터를 처리한다. 리디렉션은 Netfilter 또는 XDP 중 하나의 방식으로 설정되며, 각각의 경우에 대해 동일한 전송 테스트가 수행된다.

Netfilter의 경우, 직접 작성한 커널 모듈을 통해 PREROUTING 훅에서 목적지 IP와 포트를 수정하고, POSTROUTING 훅에서는 응답 패킷의 정보를 확인하여 로그로 출력하였다. XDP 방식은 eBPF 프로그램을 NIC에 attach하여 드라이버 수준에서 패킷을 처리하며, IP와 포트 변경, 체크섬 재계산을 모두 드라이버 내부에서 수행하였다.

CPU 사용률 측정은 mpstat 명령어를 통해 수행되었으며, 전송이 시작되기 전부터 종료될 때까지의 CPU 사용 상태를 1초 간격으로 기록하여 리디렉션 처리의 부하를 실시간으로 분석할 수 있도록 하였다. 전체적인 실험은 클라이언트와 서버 사이의 트래픽 흐름, 패킷 처리 위치, 커널 개입 범위, 그리고 측정 결과를 종합적으로 비교 분석하는 방식으로 진행되었다.

netfilter

```
kys@2020320135:~/sp_project2$ make
sudo insmod redir_module.ko
make -C /lib/modules/5.15.0-136-generic/build M=/home/kys/sp_project2 modules
make[1]: Entering directory '/usr/src/linux-headers-5.15.0-136-generic'
make[1]: Leaving directory '/usr/src/linux-headers-5.15.0-136-generic'
kys@2020320135:~/sp_project2$ sudo lsmod | grep redir
redir_module           16384  0
```

eBPF

```
kys@2020320135:~/sp_project2$ sudo bpftool net show
xdp:
enp0s1(2) driver id 66

tc:

flow_dissector:

netfilter:
```

실험 결과

1. netfilter

client

```
kys@2020320135vm2:~/sp_project2$ python3 client.py --host 10.0.8.17 --port 8080
[Client] 시작됨: 서버로 연결 시도 중
[Client] Trial 1 시작
[Client] 소켓 바인딩 완료 - 포트 10000
[Client] 서버로 /upload 전송
[Client] 데이터 전송 시작 (총 536870912 bytes)
[Client] 데이터 전송 완료, END 전송
[Client] ACK 응답 대기 중...
[Client] Trial 1: 성공 ✅ 1463.65 ms
[Client] Trial 1 종료

[Client] Trial 2 시작
[Client] 소켓 바인딩 완료 - 포트 10001
[Client] 서버로 /upload 전송
[Client] 데이터 전송 시작 (총 536870912 bytes)
[Client] 데이터 전송 완료, END 전송
[Client] ACK 응답 대기 중...
[Client] Trial 2: 성공 ✅ 1447.59 ms
[Client] Trial 2 종료

[Client] Trial 3 시작
[Client] 소켓 바인딩 완료 - 포트 10002
[Client] 서버로 /upload 전송
[Client] 데이터 전송 시작 (총 536870912 bytes)
[Client] 데이터 전송 완료, END 전송
[Client] ACK 응답 대기 중...
[Client] Trial 3: 성공 ✅ 1446.57 ms
[Client] Trial 3 종료

[Client] Trial 4 시작
[Client] 소켓 바인딩 완료 - 포트 10003
[Client] 서버로 /upload 전송
[Client] 데이터 전송 시작 (총 536870912 bytes)
[Client] 데이터 전송 완료, END 전송
[Client] ACK 응답 대기 중...
[Client] Trial 4: 성공 ✅ 1442.49 ms
[Client] Trial 4 종료

[Client] Trial 5 시작
[Client] 소켓 바인딩 완료 - 포트 10004
[Client] 서버로 /upload 전송
[Client] 데이터 전송 시작 (총 536870912 bytes)
[Client] 데이터 전송 완료, END 전송
[Client] ACK 응답 대기 중...
[Client] Trial 5: 성공 ✅ 1449.04 ms
[Client] Trial 5 종료
```

server

```
root@2020320135:/home/kys/sp_project2# python3 server.py --host 10.0.1.2 --port 8083
[SSLAB] UDP Server listening on 10.0.1.2:8083
[SSLAB] START from ('10.0.8.18', 10000)
[SSLAB] END from ('10.0.8.18', 10000) | Packets: 17632 | Duration: 1459.04 ms
[SSLAB] START from ('10.0.8.18', 10001)
[SSLAB] END from ('10.0.8.18', 10001) | Packets: 17632 | Duration: 1445.99 ms
[SSLAB] START from ('10.0.8.18', 10002)
[SSLAB] END from ('10.0.8.18', 10002) | Packets: 17632 | Duration: 1445.91 ms
[SSLAB] START from ('10.0.8.18', 10003)
[SSLAB] END from ('10.0.8.18', 10003) | Packets: 17632 | Duration: 1441.50 ms
[SSLAB] START from ('10.0.8.18', 10004)
[SSLAB] END from ('10.0.8.18', 10004) | Packets: 17633 | Duration: 1448.34 ms
[SSLAB] All 5 trials complete. Stopping mpstat and exiting.
[SSLAB] mpstat output:
Linux 5.15.0-136-generic (2020320135)   06/15/2025      _aarch64_       (2 CPU)

02:00:28 PM  CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest  %gnice   %idle
02:00:29 PM  all   13.48    0.00   14.04    0.56    0.00   34.27    0.00    0.00    0.00   37.64
02:00:30 PM  all   15.93    0.00    9.34    0.00    0.00   37.36    0.00    0.00    0.00   37.36
02:00:31 PM  all   19.55    0.00    9.50    0.00    0.00   32.96    0.00    0.00    0.00   37.99
02:00:32 PM  all   15.47    0.00    7.73    0.00    0.00   36.46    0.00    0.00    0.00   40.33
02:00:33 PM  all   17.93    0.00   13.04    0.00    0.00   33.15    0.00    0.00    0.00   35.87
02:00:34 PM  all   21.67    0.00   11.67    0.00    0.00   32.22    0.00    0.00    0.00   34.44
02:00:35 PM  all   15.47    0.00    6.63    0.00    0.00   37.02    0.00    0.00    0.00   40.88
Average:     all   17.08    0.00   10.28    0.08    0.00   34.78    0.00    0.00    0.00   37.79
```

log

```
[Sat Jun 14 16:14:26 2025] [NETFILTER] FORWARD:  UDP:10.0.8.18:10004;10.0.8.17:8080
[Sat Jun 14 16:14:26 2025] [NETFILTER] MODIFIED: UDP:10.0.8.18:10004;10.0.1.2:8083
[Sat Jun 14 16:14:26 2025] [NETFILTER] FORWARDING: UDP:10.0.8.18:10004 -> 10.0.1.2:8083
[Sat Jun 14 16:14:26 2025] [NETFILTER] POSTROUTING: UDP:10.0.1.2:8083;10.0.8.18:10004
```

2. eBPF

client

```
kys@2020320135vm2:~/sp_project2$ python3 client.py --host 10.0.8.17 --port 8080
[Client] 시작됨: 서버로 연결 시도 중
[Client] Trial 1 시작
[Client] 소켓 바인딩 완료 - 포트 10000
[Client] 서버로 /upload 전송
[Client] 데이터 전송 시작 (총 536870912 bytes)
[Client] 데이터 전송 완료, END 전송
[Client] ACK 응답 대기 중...
[Client] Trial 1: 성공 ✅ 1465.48 ms
[Client] Trial 1 종료

[Client] Trial 2 시작
[Client] 소켓 바인딩 완료 - 포트 10001
[Client] 서버로 /upload 전송
[Client] 데이터 전송 시작 (총 536870912 bytes)
[Client] 데이터 전송 완료, END 전송
[Client] ACK 응답 대기 중...
[Client] Trial 2: 성공 ✅ 1477.86 ms
[Client] Trial 2 종료

[Client] Trial 3 시작
[Client] 소켓 바인딩 완료 - 포트 10002
[Client] 서버로 /upload 전송
[Client] 데이터 전송 시작 (총 536870912 bytes)
[Client] 데이터 전송 완료, END 전송
[Client] ACK 응답 대기 중...
[Client] Trial 3: 성공 ✅ 1475.13 ms
[Client] Trial 3 종료

[Client] Trial 4 시작
[Client] 소켓 바인딩 완료 - 포트 10003
[Client] 서버로 /upload 전송
[Client] 데이터 전송 시작 (총 536870912 bytes)
[Client] 데이터 전송 완료, END 전송
[Client] ACK 응답 대기 중...
[Client] Trial 4: 성공 ✅ 1478.91 ms
[Client] Trial 4 종료

[Client] Trial 5 시작
[Client] 소켓 바인딩 완료 - 포트 10004
[Client] 서버로 /upload 전송
[Client] 데이터 전송 시작 (총 536870912 bytes)
[Client] 데이터 전송 완료, END 전송
[Client] ACK 응답 대기 중...
[Client] Trial 5: 성공 ✅ 1481.49 ms
[Client] Trial 5 종료
```

server

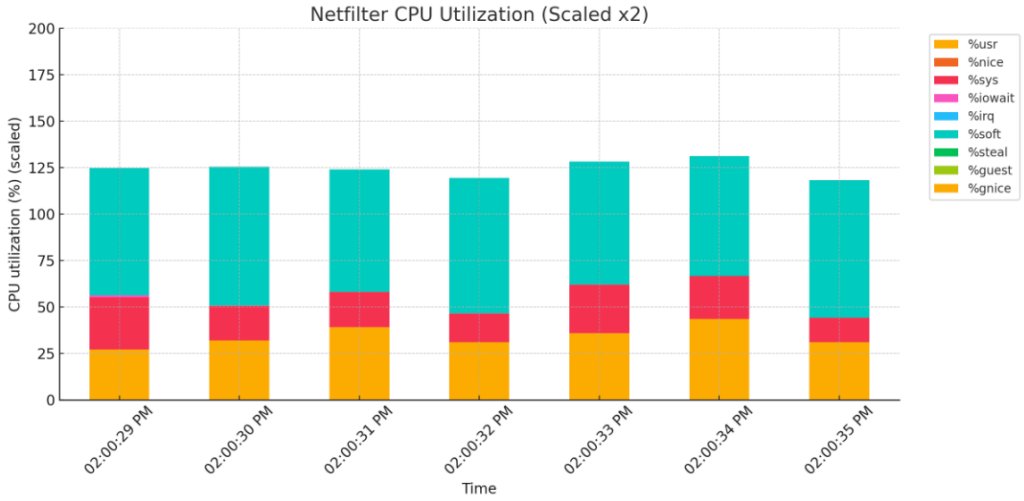
```
kys@2020320135:~/sp_project2$ sudo ip netns exec sslab-ns bash
root@2020320135:/home/kys/sp_project2# python3 server.py --host 10.0.1.2 --port 8083
[SSLAB] UDP Server listening on 10.0.1.2:8083
[SSLAB] START from ('10.0.8.18', 10000)
[SSLAB] END from ('10.0.8.18', 10000) | Packets: 370256 | Duration: 1558.03 ms
[SSLAB] START from ('10.0.8.18', 10001)
[SSLAB] END from ('10.0.8.18', 10001) | Packets: 370256 | Duration: 1528.13 ms
[SSLAB] START from ('10.0.8.18', 10002)
[SSLAB] END from ('10.0.8.18', 10002) | Packets: 370256 | Duration: 1665.56 ms
[SSLAB] START from ('10.0.8.18', 10003)
[SSLAB] END from ('10.0.8.18', 10003) | Packets: 370256 | Duration: 1625.32 ms
[SSLAB] START from ('10.0.8.18', 10004)
[SSLAB] END from ('10.0.8.18', 10004) | Packets: 370256 | Duration: 2128.28 ms
[SSLAB] All 5 trials complete. Stopping mpstat and exiting.
[SSLAB] mpstat output:
Linux 5.15.0-136-generic (2020320135)   06/15/2025      _aarch64_       (2 CPU)

02:09:30 PM  CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest  %gnice   %idle
02:09:31 PM  all   17.20    0.00   13.44    0.00    0.00   24.73    0.00    0.00    0.00   44.62
02:09:32 PM  all   19.38    0.00   13.12    0.00    0.00   16.25    0.00    0.00    0.00   51.25
02:09:33 PM  all   19.28    0.00   12.65    0.00    0.00   15.06    0.00    0.00    0.00   53.01
02:09:34 PM  all   20.83    0.00   13.69    0.60    0.00   18.45    0.00    0.00    0.00   46.43
02:09:35 PM  all   16.37    0.00   12.28    0.00    0.00   21.05    0.00    0.00    0.00   50.29
02:09:36 PM  all   17.54    0.00   12.28    0.00    0.00   16.96    0.00    0.00    0.00   53.22
02:09:37 PM  all   17.03    0.00    7.69    0.00    0.00   15.93    0.00    0.00    0.00   59.34
02:09:38 PM  all   18.71    0.00    8.39    0.00    0.00   12.90    0.00    0.00    0.00   60.00
Average:     all   18.25    0.00   11.70    0.07    0.00   17.81    0.00    0.00    0.00   52.17
```

log

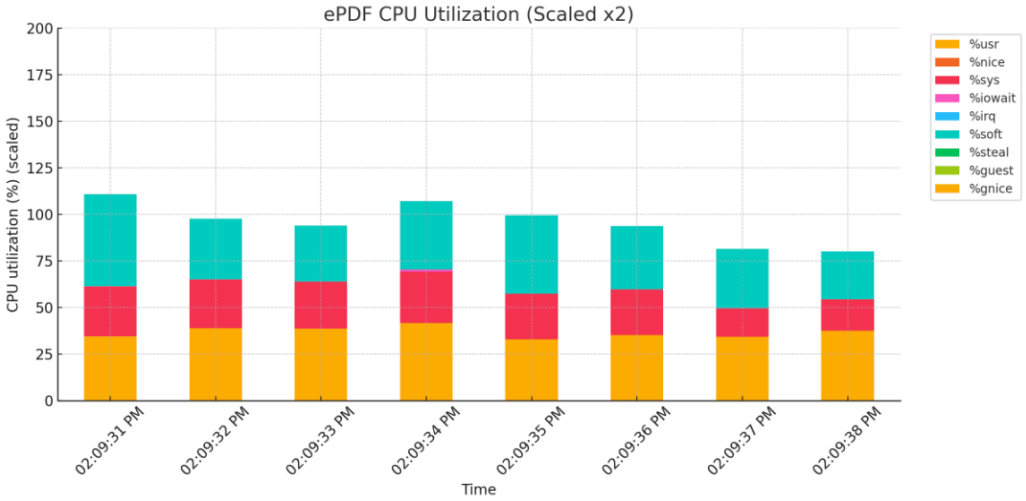
```
kys@2020320135:~/sp_project2$ sudo cat /sys/kernel/debug/tracing/trace_pipe
          <idle>-0       [001] ..s..  6515.700697: bpf_trace_printk: Original: dst_ip=1108000a, dst_port=8080
          <idle>-0       [001] ..s..  6515.700726: bpf_trace_printk: Modified: dst_ip=201000a, dst_port=8083, ip_csum=0xc4a8
          <idle>-0       [001] ..s..  6515.700746: bpf_trace_printk: Original: dst_ip=1108000a, dst_port=8080
          <idle>-0       [001] ..s..  6515.700747: bpf_trace_printk: Modified: dst_ip=201000a, dst_port=8083, ip_csum=0x20a3
          <idle>-0       [001] ..s..  6515.700749: bpf_trace_printk: Original: dst_ip=1108000a, dst_port=8080
          <idle>-0       [001] ..s..  6515.700749: bpf_trace_printk: Modified: dst_ip=201000a, dst_port=8083, ip_csum=0x1fa3
          <idle>-0       [001] ..s..  6515.700750: bpf_trace_printk: Original: dst_ip=1108000a, dst_port=8080
          <idle>-0       [001] ..s..  6515.700751: bpf_trace_printk: Modified: dst_ip=201000a, dst_port=8083, ip_csum=0x1ea3
          <idle>-0       [001] ..s..  6515.700752: bpf_trace_printk: Original: dst_ip=1108000a, dst_port=8080
          <idle>-0       [001] ..s..  6515.700752: bpf_trace_printk: Modified: dst_ip=201000a, dst_port=8083, ip_csum=0x1da3
          <idle>-0       [001] ..s..  6515.700756: bpf_trace_printk: Original: dst_ip=1108000a, dst_port=8080
          <idle>-0       [001] ..s..  6515.700756: bpf_trace_printk: Modified: dst_ip=201000a, dst_port=8083, ip_csum=0x1ca3
          <idle>-0       [001] ..s..  6515.700758: bpf_trace_printk: Original: dst_ip=1108000a, dst_port=8080
          <idle>-0       [001] ..s..  6515.700758: bpf_trace_printk: Modified: dst_ip=201000a, dst_port=8083, ip_csum=0x1ba3
```

실험 결과

netfilter



eBPF



비교 분석

그래프를 보면 Netfilter 기반 리디렉션과 XDP(eBPF) 기반 리디렉션은 CPU 자원 사용 패턴에서 뚜렷한 차이를 보인다.

먼저 Netfilter 그래프에서는 전체적으로 CPU 사용률이 높게 나타나며, 특히 %sys(시스템 영역, 커널 공간에서의 사용률)와 %softirq 의 비율이 상당히 높은 수준을 유지하고 있다. 이는 Netfilter 가 커널 네트워크 스택의 PREROUTING 이후 단계에서 작동하며, 소켓 버퍼(skb) 기반으로 패킷을 처리하기 때문에 커널 내부에서 많은 처리 비용이 발생함을 의미한다. 또한 커널 공간에서 수행되는 작업이 많을수록 context switching 과 인터럽트 처리도 늘어나며, 이는 softirq 의 상승으로도 나타난다.

반면, XDP 기반 리디렉션을 나타낸 eBPF 그래프에서는 전반적인 CPU 사용률이 낮고, 특히 %sys 와 %softirq 사용량이 Netfilter 에 비해 확연히 줄어든 모습이다. 이는 XDP 가 NIC 드라이버 수준에서 가장 빠르게 패킷을 처리하기 때문에, 이후의 커널 네트워크 계층을 거의 통과하지 않고 리디렉션을 마칠 수 있기 때문이다. 즉, skb 를 생성하지 않으며, IP 계층 이상의 프로토콜 처리도 생략할 수 있어 커널 개입 자체가 매우 제한적이고 그만큼 CPU 오버헤드도 낮게 유지된다.

또한 두 그래프 모두 %usr 사용률은 비슷한 수준으로 나타났는데, 이는 사용자 공간에서 수행되는 파일 송수신 작업 자체는 리디렉션 방식에 상관없이 동일하게 작동했기 때문이다. 하지만 리디렉션 처리가 커널 내부에서 얼마나 많은 연산을 요구하는지는 %sys 및 %softirq 항목에서 명확히 드러난다.

요약하면, Netfilter 는 커널 스택을 모두 통과한 후 패킷을 처리하기 때문에 CPU 리소스를 더 많이 사용하며, 특히 커널 영역 부하가 크다. 반면 XDP 는 패킷을 수신하자마자 즉시 처리하기 때문에 커널 부하가 적고, CPU 효율이 뛰어나다. 따라서 실시간성, 고성능 네트워크 처리가 요구되는 환경에서는 XDP 기반 리디렉션이 더욱 적합한 방식임을 실험을 통해 확인할 수 있다.

어려웠던 점과 해결 방법

실험을 진행하면서 관찰된 한 가지 주요한 점은 Netfilter 방식과 eBPF(XDP) 방식 간에 수신된 패킷 개수에 일정한 차이가 존재했다는 것이다. 동일한 환경에서 동일한 크기의 데이터를 전송하였음에도 불구하고, Netfilter 경로를 사용할 때 수신된 총 패킷 수가 XDP 방식에 비해 다소 적게 측정되었다. 이를 해결하기 위해 여러 가지 네트워크 커널 파라미터를 조정하였다.

```
# 수신 버퍼 최대값을 128MB로 설정
sudo sysctl -w net.core.rmem_max=134217728
sudo sysctl -w net.core.rmem_default=134217728

# NIC → 커널 대기열 확장
sudo sysctl -w net.core.netdev_max_backlog=250000
```

설정을 조정한 이후에도 Netfilter와 XDP 간의 수신 패킷 수에는 여전히 약간의 차이가 있었다. 단순히 버퍼 크기나 수신 큐 설정을 늘리는 것으로는 완전히 해결되지 않았고, 이는 Netfilter의 구조적인 한계에서 비롯된 것으로 보인다.

실험에서는 Netfilter 훅에서 패킷의 목적지 IP와 포트를 변경한 뒤, IP 체크섬을 직접 수동으로 계산하여 업데이트하였다. 이는 헤더 정보가 변경되었기 때문에 필수적인 처리였으며, 계산은 16비트 단위의 덧셈과 보수 연산을 이용하여 구현되었다. 또한 UDP 체크섬은 udp->check = 0;으로 명시적으로 무효화하였는데, 이는 로컬 네트워크 환경에서 체크섬 검사가 필요하지 않고, 커널이 이를 생략하도록 하기 위한 조치였다. 이러한 처리는 실험 목적에 맞춰 CPU 자원 소모를 중심으로 리디렉션 방식의 차이를 비교하는 데 초점을 맞춘 것이다. 체크섬 처리 방식에 따라 일부 패킷이 내부적으로 무시되었을 가능성도 고려되어 tcpdump를 통해 인터페이스 단의 패킷 흐름을 관찰했으나, 눈에 띄는 드롭 로그나 비정상 패킷은 확인되지 않았다.

그럼에도 불구하고 본 실험의 핵심 목적은 전체 전송량을 동일하게 유지하는 것이 아니라, 리디렉션 처리 방식에 따라 CPU 자원이 어떻게 사용되는지를 비교하는 데 있었다. 특히 결과에서 주목한 부분은 CPU 전체 사용률뿐만 아니라, 소프트웨어 인터럽트 처리 비율(%soft)의 차이가 매우 뚜렷하게 나타났다는 점이다. Netfilter에서는 커널 내부에서 발생하는 소프트웨어 인터럽트 비중이 높게 유지된 반면, XDP는 드라이버 수준에서 패킷을 조기에 처리하므로 softirq 부하가 현저히 낮게 나타났다. 이는 XDP가 커널 개입을 최소화하여 효율적으로 작동한다는 이론적 장점이 실험에서도 확인되었음을 의미한다.

결론적으로, 수신 패킷 수에 약간의 차이는 존재하였으나, CPU 점유율 및 softirq 사용량이라는 핵심 지표에서 의미 있는 차이가 명확히 관찰되었기 때문에, 실험 결과는 본래의 목적에 부합하며 유의미하다고 판단되어 그대로 분석에 활용하였다.

이번 실험을 통해 알게 된 점

이번 실험을 통해, 동일한 목적을 가진 패킷 리디렉션 작업이라도 어떤 계층에서 처리하느냐에 따라 시스템 자원 소모에 큰 차이가 발생한다는 점을 명확히 확인할 수 있었다. Netfilter와 eBPF(XDP)는 모두 리눅스 커널 내에서 네트워크 패킷을 조작할 수 있는 강력한 프레임워크지만, 동작 위치와 처리 방식의 차이로 인해 CPU 활용 효율에서 상당한 차이를 보였다.

특히 주목할 만한 점은 CPU 사용률 중 softirq(%soft)의 비중이었다. Netfilter는 커널 네트워크 스택 내부, 즉 상대적으로 높은 계층에서 작동하기 때문에 패킷이 여러 커널 경로를 거쳐 처리되며, 이 과정에서 소프트웨어 인터럽트가 자주 발생하고 CPU 리소스가 더 많이 소모되었다. 반면, XDP는 네트워크 드라이버 수준에서 패킷을 조기에 처리함으로써 이러한 오버헤드를 최소화하였고, 결과적으로 낮은 CPU 점유율과 효율적인 인터럽트 처리를 가능하게 했다.

또한 실험 중 수신 패킷 수에서 약간의 차이가 발생했음에도 불구하고, 실험의 본질이 "데이터 전송 성공률"이 아니라 "처리 방식에 따른 커널 자원 사용 효율성"에 있다는 점을 명확히 이해하고 판단할 수 있었다. 이를 통해 실험의 목적을 정확히 설정하는 것이 결과 해석에 있어 얼마나 중요한지를 다시금 깨달았다.

종합적으로 이번 실험은 시스템 수준 네트워크 최적화를 고려할 때 단순한 기능 구현만이 아닌, 처리 계층과 커널 부하에 대한 고려가 필수적임을 보여주었으며, 실제 환경에서 어떤 방식이 더 적합한지를 판단하는 데 있어 실질적인 기준을 제공해 주는 경험이 되었다. 또한 실험 과정에서 다양한 커널 파라미터를 조정하고 디버깅을 수행하면서, 커널 네트워크 처리 구조와 성능 병목에 대한 이해도도 함께 높일 수 있었다.