O'REILLY®

Linux Observability with BPF

BPF로 리눅스 관측 가능성 향상하기

성능 분석과 네트워킹을 위한 고급 프로그래밍

데이비드 칼라베라, 로렌초 폰타나 지음
류광 옮김

한빛미디어 Hanbit Media, Inc.

Linux Observability with BPF

BPF로 리눅스 관측 가능성 향상하기

| 표지 설명 |

앞표지의 동물은 예전에는 큰소쩍새(*Otus bakkamoena*)로 분류했던 순다 소쩍새(Sunda scops owl)로, 학명은 *Otus lempiji*이다. 소쩍새는 머리에 귀뿔깃이 난, 올빼미과의 소형 조류이다. 순다 소쩍새는 동남아시아에 서식하며, 싱가포르 소쩍새(Singapore scops owl)라고 부르기도 한다. 원래 순다 소쩍새는 숲에 살았지만, 이제는 도시화되어 공원에서 살기도 한다.

순다 소쩍새는 전체적으로 밝은 갈색에 검은 줄무늬와 반점이 있다. 키는 약 20cm, 무게는 약 170g이다. 초봄에 암컷이 나무 구멍에 두세 개의 알을 낳는다. 주로 곤충(특히 딱정벌레)을 먹고 살지만, 설치류나 도마뱀, 작은 새를 사냥하기도 한다.

특징적인 날카로운 울음소리 때문에 흔히 비명 올빼미(screech owl) 또는 귀신 소쩍새라고 부른다. 점점 높아져서 고음에 이르는 울음을 매 10초 간격으로 반복할 수 있다.

오라일리 책들의 앞표지에 나온 동물 중 다수는 멸종 위기이다. 이들은 모두 이 세상에서 소중한 존재이다.

표지 그림은 『British Birds』의 흑백 판화에 기초해 수지 위비엇Suzy Wiviott이 그렸다.

BPF로 리눅스 관측 가능성 향상하기

성능 분석과 네트워킹을 위한 고급 프로그래밍

초판 1쇄 발행 2020년 5월 1일

지은이 데이비드 칼라베라, 로렌초 폰타나 / **옮긴이** 류광 / **펴낸이** 김태헌
펴낸곳 한빛미디어(주) / **주소** 서울시 서대문구 연희로2길 62 한빛미디어(주) IT출판부
전화 02-325-5544 / **팩스** 02-336-7124
등록 1999년 6월 24일 제25100-2017-000058호 / **ISBN** 979-11-6224-305-3 93000

총괄 전정아 / **책임편집** 박지영 / **기획** 최현우 / **편집** 정지수 / **교정** 오현숙
디자인 표지 이아란 내지 김연정 조판 이경숙
영업 김형진, 김진불, 조유미 / **마케팅** 박상용, 송경석, 조수현, 이행은, 홍혜은 / **제작** 박성우, 김정우

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오. 잘못된 책은 구입하신 서점에서 교환해드립니다. 책값은 뒤표지에 표시되어 있습니다.

한빛미디어 홈페이지 www.hanbit.co.kr / 이메일 ask@hanbit.co.kr

지금 하지 않으면 할 수 없는 일이 있습니다.
책으로 펴내고 싶은 아이디어나 원고를 메일(writer@hanbit.co.kr)로 보내주세요.
한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

Linux Observability with BPF

BPF로 리눅스 관측 가능성 향상하기

O'REILLY® 한빛미디어 Hanbit Media, Inc.

지은이 · 옮긴이 소개

지은이 **데이비드 칼라베라** David Calavera

Netlify의 CTO이다. 도커의 개발 관리자로 일했으며, Runc와 Go, BCC 도구들을 비롯해 여러 오픈소스 프로젝트에 기여했다. 데이비드는 도커 프로젝트들과 관련해서 도커 플러그인 생태계를 만들고 관리한 것으로 이름을 알렸다. 불꽃 그래프와 성능 최적화에 크나큰 애착을 지니고 있다.

지은이 **로렌초 폰타나** Lorenzo Fontana

Sysdig 사의 오픈소스 팀에서 클라우드 네이티브 컴퓨팅 파운데이션 프로젝트인 Falco를 개발한다. Falco는 커널 모듈과 eBPF를 이용해서 컨테이너 런타임 보안 및 비정상 검출을 수행하는 제품이다. 분산 시스템, 소프트웨어 정의 네트워크(SDN), 리눅스 커널, 성능 분석에 열정을 지니고 있다.

옮긴이 **류광**

25년여의 번역 경력을 가진 전문 번역가로, 도널드 커누스 교수의 『컴퓨터 프로그래밍의 예술』(*The Art of Computer Programming*) 시리즈와 스티븐스의 『UNIX 고급 프로그래밍』(*Advanced Programming in UNIX Environment*) 제2판 및 제3판, 『Game Programming Gems』 시리즈를 포함해 80권 이상의 다양한 IT 전문서를 번역했다.

옮긴이의 말

C++ 프로그래밍 언어를 만든 비야네 스트롭스트룹은 "우리의 문명은 소프트웨어를 바탕으로 돌아간다"라고 말한 적이 있습니다. 현대 문명을 지탱하는 컴퓨팅 기반구조에서 큰 자리를 차지하고 있는 것이 리눅스 커널입니다. 우리 삶에 직 · 간접적으로 영향을 미치는 수많은 소프트웨어가 리눅스 커널에 의존하며, 어떤 운영체제이든 '커널'은 극도의 성능과 보안이 요구되는 영역입니다. 그런 만큼 리눅스 커널은 아무나 건드릴 수 없는, 소수의 전문가에게만 허용되는 영역으로 간주됩니다. 그 점을 생각하면, 평범한 개발자도 스크립팅과 VM을 통해서 편하고 안전하게 리눅스 커널 안에서 원하는 코드를 실행하게 하는 BPF는 참으로 흥미로운 기술입니다. 추천사에서 제시 프래즐이 "할렐루야!"를 외친 것도 이해가 됩니다. 이 책을 읽고 독자 여러분도 프래즐만큼이나 BPF에 열광하게 되면 좋겠습니다.

이 책에는 독자의 이해를 돕는 다양한 예제가 나옵니다. 번역하면서 발견한 사소한 오류(콜론이나 세미콜론 누락 등)는 직접 수정했고, 코드나 본문에 직접 반영하기가 마땅치 않은 사항은 역주로 언급해 두었습니다. 특히 예제 실행에 필요한 개발 도구나 라이브러리, 패키지를 저자들이 구체적으로 언급하지 않은 경우가 종종 있었습니다. 단, 어느 정도 경험 있는 독자를 대상으로 한 책이니만큼 독자가 충분히 짐작할 만한 것들(이를테면 golang 패키지 설치 등)은 굳이 역주를 달지 않았습니다. 제 웹사이트(*http://occamsrazr.net*)에 이 책을 위한 페이지가 있으니, 예제 관련 문제점을 알려주시면 함께 고민하고 해결책을 찾아보겠습니다. *번역서 정보* 페이지에서 링크를 찾으시면 됩니다. 오타 · 오역 보고와 의견, 제안도 환영합니다.

그리 두껍지 않은 책이지만, 이 책의 출간에는 많은 분의 노력이 필요했습니다. 저에게 번역을 맡겨 주신 한빛미디어 최현우 부장님과 번역 및 교정 과정을 능숙하게 이끌어 주신 정지수 편집자님, 제 마음에 딱 들게 책을 조판해 주신 이경숙 디자이너님을 비롯해 모든 관련자분께 감사드립니다. 마지막으로, 2020년 봄 가족의 건강을 지키면서 교정까지 깔끔하게 끝낸 아내 오현숙에게 그저 고맙고 사랑한다는 말을 전합니다.

재미있게 읽으시길!

류광

추천사

프로그래머인(그리고 샌님임을 스스로 인정하는) 나는 다양한 커널의 최신 변경 사항과 컴퓨팅 분야의 연구 성과를 따라잡는 데 열심이다. 리눅스의 BPF(Berkeley Packet Filter)와 XDP(Express Data Path)를 접하고 나는 바로 사랑에 빠졌다. 너무나 멋진 도구인 BPF와 XDP를 주제로 한 이 책 덕분에 더 많은 사람이 프로젝트에 이들을 사용하게 되리라 생각하니 기쁘기 짝이 없다.

내 배경과 이 커널 인터페이스들을 사랑하게 된 이유를 좀 더 자세히 이야기해 보겠다. 나는 이 책의 저자 중 한 명인 데이비드 칼라베라와 함께 도커Docker의 핵심 개발 관리자(maintainer)로 일했다. 도커를 잘 모르는 독자를 위해 잠깐 언급하자면, 도커는 컨테이너에 대해 다양한 필터링과 라우팅 논리를 수행하기 위해 `iptables`를 셸로 실행한다. 내가 도커 프로젝트에 기여한 첫 패치는 CentOS에서 특정 버전의 `iptables`가 명령줄 플래그들의 차이 때문에 작업에 실패하는 버그를 잡는 것이었다. 도커에는 이와 비슷한 괴상한 문제점들이 많이 있었으며, 꼭 도커가 아니더라도 자신의 소프트웨어에서 어떤 외부 도구를 셸로 실행해 본 사람이라면 비슷한 문제점을 겪어보았을 것이다. 그 문제점 외에, 애초에 `iptables`는 하나의 호스트에 수천 가지 규칙을 적용해야 하는 상황에 맞게 만들어진 것이 아니라서 도커에서 그런 식으로 사용하기에는 성능 측면에서 문제가 있었다.

그러다가 나는 BPF와 XDP에 관한 이야기를 들었는데, 내 귀에는 마치 음악과도 같았다. 이제는 또 다른 버그 때문에 `iptables`가 남긴 상처에서 피를 흘릴 필요가 없다! 심지어 커널 공동체는 `iptables`를 BPF로 대체하는 프로젝트도 진행 중이다(*https://oreil.ly/cuqTy*)! 할렐루야! 또한 컨테이너 네트워킹을 위한 도구인 Cilium(*https://cilium.io*)은 내부적으로 BPF와 XDP를 사용한다.

그리고 그것이 전부가 아니다! BPF에는 `iptables` 대용으로 사용하는 것 이외에도 아주 많은 용도가 있다. BPF를 이용하면 임의의 시스템 호출 또는 커널 함수를 추적할 수 있으며, 임의의 사용자 공간 프로그램도 추적할 수 있다. bpftrace(*https://github.com/iovisor/bpftrace*)는 리눅스에서 DTrace와 비슷한 기능을 제공하는 명령줄 도구이다. 이 도구를 이

용하면 예를 들어 열린 파일들과 그것을 연 프로세스들을 추적하거나, 프로그램이 요청한 시스템 호출들을 세거나, OOM 킬러를 추적하는 등의 다양한 추적 작업을 수행할 수 있다. 간단히 말해서, 이 도구가 있으면 리눅스 시스템의 내부 상황을 완전히 파악할 BPF와 XDP는 Cloudflare(*https://oreil.ly/OZdmj*)와 페이스북(*https://oreil.ly/wrM5-*)의 부하 분산기에서 DDoS 공격을 방지하는 용도로도 쓰인다. XDP가 패킷 폐기 작업에 왜 그렇게 뛰어난지는 이 책의 제7장에 나오므로 여기서 미리 누설하지는 않겠다.

나는 쿠버네티스 공동체에서 이 책의 또 다른 저자 로렌초 폰타나와 안면을 트는 영광을 누렸다. 그가 만든 `kubectl-trace`(*https://oreil.ly/Ot7kq*)를 이용하면 커스텀 추적 프로그램을 쿠버네티스 클러스터 안에서 손쉽게 실행할 수 있다.

개인적으로 내가 제일 선호하는 BPF 활용법은, 어떤 소프트웨어의 성능에 관한 주장이 거짓이었음을, 또는 그 소프트웨어가 시스템 호출들을 필요 이상으로 많이 호출한다는 점을 커스텀 추적기를 이용해서 폭로하는 것이다. 누군가의 과대선전이나 오류를 구체적인 자료로 증명하는 것만큼 짜릿한 일도 없을 것이다. 이 책에 BPF를 이용해서 추적 프로그램을 작성하는 방법이 나오니 여러분도 나처럼 할 수 있다. 무엇보다도 BPF의 매력은 사건이 발생하는 바로 그 장소(커널 안)에서 직접 자료를 취합하거나 분류할 수 있다는 점이다. 이는 수집한 자료를 손실 가능성 있는 대기열을 거쳐 사용자 공간에 보낸 후 거기서 취합, 처리해야 하는 기존 도구들에 비한 BPF의 확실한 장점이다.

나는 내 경력의 절반을 개발자를 위한 도구를 만드는 데 보냈다. 최고의 도구들은 적절한 인터페이스를 통해서 자신의 내부를 여러분 같은 개발자에게 드러내며, 그럼으로써 사용자들이 도구 작성자가 상상도 못 한 방식으로 도구를 활용하게 한다. 리처드 파인먼은 "나는 뭔가의 이름을 아는 것과 그것을 이해하는 것은 다른 문제임을 아주 일찍 깨달았다."라고 말한 적이 있다. BPF라는 이름만 들어본 독자라면, 이 책을 통해서 BPF가 얼마나 유용할지도 알게 될 것이다.

내가 이 책을 마음에 들어 하는 이유 하나는 이 책이 BPF를 이용해서 여러분이 스스로 새로운 도구를 만드는 데 필요한 지식을 제공한다는 점이다. 이 책을 읽고 예제들을 공부하면서 자

신만의 도구들을 만들다 보면 BPF를 마음대로 활용하는 능력을 갖출 것이며, 몇몇 도구는 아주 유용해서 실제로 일상 업무에서 자주 사용하게 될 것이다. 그냥 BPF를 아는 것에서 그치지 말고 BPF를 "이해하는" 경지로 나아가길 권한다. 이 책을 충실하게 공부한다면, BPF로 만들 수 있는 무궁무진한 도구들의 세계로 넘어갈 수 있을 것이다.

BPF 공동체는 계속해서 발전하고 있다. 이 활기찬 공동체가 더욱 성장해서 더 많은 사람이 BPF의 위력을 자신의 것으로 삼았으면 좋겠다. 이 책의 독자가 무엇을 만들어 낼지 기대가 크다. 괴상한 소프트웨어 버그를 추적하는 스크립트일 수도 있고, 커스텀 방화벽일 수도 있고, 어쩌면 적외선 복호화(infra decoding; *https://lwn.net/Articles/759188*) 도구일 수도 있겠다. 뭔가 만들어 냈다면 모두 알 수 있도록 소식 전해주시길!

제시 프래즐 Jessie Frazelle

2015년 데이비드는 컨테이너를 유명하게 만든 회사인 도커 사에서 핵심 개발자로 일하고 있었다. 매일 그는 한편으로는 도커 공동체를 돕고, 한편으로는 도커 프로젝트를 키워나갔다. 그의 업무 중 일부는 도커 공동체의 구성원들이 보낸 수많은 풀 요청(PR)을 검토하는 것이었다. 또한 그는 도커가 모든 종류의 시나리오에서 잘 돌아가게 하는 문제도 담당했는데, 그런 시나리오 중에는 임의의 시점에서 수천 개의 컨테이너를 실행하고 제공할 수 있을 정도로 고성능이 요구되는 상황도 포함된다.

도커의 성능 문제를 진단하기 위해 우리는 **불꽃 그래프**(flame graph)라는 고급 시각화 기법을 사용했다. 이 기법은 자료를 좀 더 쉽게 탐색하는 데 도움이 된다. Go 프로그래밍 언어로 만든 응용 프로그램에는 HTTP 종점(endpoint)이 내장되어 있어서 성능을 측정하고 자료를 추출하기가 아주 쉽다. 또한 그러한 자료로 그래프를 만드는 것도 간단하다. 데이비드는 Go의 프로파일링 기능과 그 기능으로 수집한 자료로 불꽃 그래프를 만드는 방법에 관한 글을 썼다. 도커의 성능 자료 수집 방식에서 한 가지 주의할 점은, 도커의 프로파일러가 기본적으로 꺼져 있다는 것이다. 그래서 성능 문제를 디버깅하려면 먼저 도커를 재시작해야 한다. 그런데 서비스를 재시작하면 수집하고자 했던 관련 자료도 사라지며, 따라서 추적하고자 하는 사건이 일어날 때까지 기다려야 한다. 도커 불꽃 그래프에 관한 글에서 데이비드는 도커의 성능 측정에 그러한 재시작이 꼭 필요하지만, 반드시 그런 방식이어야 하는 것은 아님을 언급했다. 그러한 점을 깨닫고 그는 임의의 응용 프로그램의 성능 자료를 수집하고 분석하는 다양한 기술들을 연구하기 시작했으며, 급기야는 BPF를 알게 되었다.

한편, 데이비드와는 멀리 떨어진 곳에서 로렌초는 리눅스 커널의 내부를 좀 더 본격적으로 공부하는 방법을 모색하다가, BPF를 중점적으로 공부하면서 그와 관련된 리눅스 커널 하위 시스템들을 살펴보는 것이 효과적임을 깨달았다. 2년 후 그는 InfluxData 사에서 자신의 업무에 (구체적으로는 InfluxCloud의 자료 소화 속도를 높이는 데) BPF를 적용할 수 있게 되었다. 현재 로렌초는 BPF 공동체와 IOVisor에 참여하며, Falco에서 BPF를 이용해서 컨테이너와 리눅스의 실행 시점 보안을 수행하는 도구인 Sysdig를 개발하고 있다.

지난 수년간 우리는 쿠버네티스 클러스터의 사용량 자료 수집에서부터 네트워크 소통량 정책 관리까지 다양한 시나리오에서 BPF를 사용했다. BPF를 활용하면서, 그리고 브렌던 그레그Brendan Gregg와 알렉세이 스타로보이토프Alexei Starovoitov 같은 기술 선구자들이나 Cilium, 페이스북 같은 기업들의 여러 블로그 글을 읽으면서 우리는 BPF를 속속들이 파악하게 되었다. 그들의 글과 출판물은 당시 우리에게 아주 큰 도움이 되었을 뿐만 아니라, 이 책을 저술할 때 중요한 참고 자료로도 쓰였다.

여러 자료를 읽으면서 우리는 BPF에 관해 뭔가 배울 것이 생길 때마다 수많은 블로그 글과 매뉴얼 페이지, 기타 웹 페이지들을 오가는 것이 그리 효율적이지 않다는 점을 깨달았다. 이 책은 다음 세대의 BPF 애호가들이 이 멋진 기술을 좀 더 손쉽게 배울 수 있도록 웹에 흩어져 있는 지식을 한 장소로 모으려는 시도의 산물이다.

우리는 우리가 알고 있는 것을 9개의 장(chapter)으로 나누어 저술했다. 이 장들은 BPF로 무엇을 어떻게 할 수 있는지 알려준다. 참고서나 지침서로서 따로 읽어도 되는 장들도 있지만, BPF를 처음 접하는 독자라면 9개의 장을 순서대로 모두 읽길 권한다. 그러면 먼저 BPF의 핵심 개념들을 익힌 다음 좀 더 구체적인 여러 응용 방법으로 나아갈 수 있다.

관측 가능성과 성능 분석 분야의 전문가이든, 아니면 자신의 실무용 시스템에 관해 아직 풀지 못한 의문의 답을 구하는 새로운 기법을 모색하는 개발자이든, 이 책에서 뭔가 새로운 것을 배울 수 있을 것이다.

예제 코드

이 책의 예제 코드는 *https://oreil.ly/lbpf-repo*에서 내려받을 수 있다.

감사의 말

책을 쓰는 것이 생각보다 힘들었지만, 이 책의 저술은 우리가 지금까지 해온 활동 중 가장 보람 있는 축에 속할 것이다. 우리는 수많은 낮과 밤을 이 책에 쏟아부었으며, 동료들과 가족, 친구, 강아지들의 도움이 없었다면 저술을 끝내지 못했을 것이다. 로렌초는 긴 저술 기간 동안 참을성 있게 기다려 준 여자 친구 데버라 페이스Debora Pace와 아들 리카르도Riccardo에 감사한다. 또한 조언을 제공하고 특히 XDP와 코드 검사에 관한 글을 써준 친구 레오나르도 디 도나토Leonardo Di Donato에게 감사한다.

데이비드는 아내 로빈 민스Robin Means에 무한한 감사의 마음을 보낸다. 로빈은 여러 장의 초안과 이 책의 출발점이 된 초고를 감수했으며, 지난 수년간 데이비드의 여러 저술 작업을 도왔다. 또한 실제보다 더 근사하게 들리도록 지어낸 가짜 영어 단어들에 웃어주었다.

우리 두 필자는 eBPF와 BPF를 만들고 개선한 모든 이에게 큰 감사의 뜻을 표한다. 리눅스 커널의 개선에 계속해서 기여하고 그럼으로써 eBPF와 그 공동체를 가능하게 한 데이비드 밀러David Miller와 알렉세이 스타로보이토프에게 감사한다. BPF에 대한 열정을 나눠주고 사람들이 eBPF를 좀 더 손쉽게 사용할 수 있도록 하는 도구들을 개발한 브렌던 그레그에게 감사한다. 우리를 지지하고 수많은 조언을 제공했으며 bpftrace, gobpf, kubectl-trace, BCC를 만들고 발전시킨 IOVisor 그룹에 감사한다. 영감을 주는 작업들, 특히 libbpf와 도구 기반 구조를 만든 대니얼 보크먼Daniel Borkmann에게 감사한다. 추천사를 써주었으며 우리 두 필자와 수천 명의 개발자에게 영감을 준 제시 프래즐에게 감사한다. 더 바랄 것이 없는 최고의 감수자인 제롬 페타초니Jérôme Petazzoni에게 감사한다. 그의 질문 덕분에 우리는 이 책의 여러 부분과 코드 예제에 관한 접근 방식을 다시 생각하게 되었다.

그리고 수천 명의 리눅스 커널 기여자들, 특히 질문/답변, 패치, 제안으로 BPF 메일링 리스트에 활발하게 참여한 모든 이에게 감사한다. 마지막으로, 편집자 존 데빈스John Devins와 멜리사 포터Melissa Potter를 비롯해 표지를 만들고, 원고를 검토하고, 이 책을 우리가 개발자 경력에서 만들어 낸 그 무엇보다도 더 전문적으로 보이게 만든 분들을 포함해 이 책의 출판에 관여한 오라일리의 모든 분께 감사한다.

CONTENTS

CONTENTS

CONTENTS

제1장 소개

지난 수십 년간 컴퓨팅 시스템은 계속해서 복잡해졌다. 소프트웨어의 작동을 분석, 추론하는 작업에서 다양한 업무 분야가 파생되었는데, 그 분야들은 모두 복잡한 시스템에 대한 통찰을 얻는 난제를 해결하려 한다는 공통점이 있다. 소프트웨어에 대한 가시성을 획득하는 접근 방식 하나는 컴퓨팅 시스템 안에서 실행되는 모든 응용 프로그램이 생성한 기록 자료, 즉 로그log를 수집하고 분석하는 것이다. 로그에는 많은 정보가 담겨 있다. 로그들을 살펴보면 응용 프로그램이 정확히 어떻게 작동하고 있는지를 상당히 정확하게 알 수 있다. 그러나 로그에는 애초에 응용 프로그램을 만든 개발자가 제공하기로 한 정보만 담겨 있다는 한계도 있다. 그래서 임의의 시스템에서 로그로부터 추가적인 정보를 수집하는 것이 프로그램을 역컴파일(decompile)해서 그 실행 흐름을 파악하는 것만큼이나 어려운 일이 되기도 한다. 또 다른 인기 있는 접근 방식은 계량(metrics)으로 얻은 측정치들을 이용해서 프로그램의 행동 방식을 추론하는 것이다. 측정치들은 로그와 자료 형식이 다르다. 로그는 명시적인 자료를 제공하는 반면, 계량은 특정 시점(time point)에서 응용 프로그램의 행동 방식을 측정한 자료들을 취합한 결과를 제공한다.

비교적 최근 등장한 **관측 가능성**(observability; 또는 가관측성) 접근 방식은 이 문제에 앞의 두 방식과는 다른 각도에서 접근한다. 일반적으로 관측 가능성은 주어진 임의의 시스템에 대해 임의의 복잡한 질문을 던지고 그 응답을 얻을 수 있는 능력으로 정의된다. 관측 가능성과 로그, 측정치 취합의 주된 차이점은 각각이 수집하는 자료에 있다. 관측 가능성을 활용한다는 것이 임의의 시점에서 임의의 질문에 대한 답을 얻을 필요가 있다는 뜻이라고 할 때, 자료로부

터 응용 프로그램의 행동을 추론하려면 시스템이 생성하는 모든 자료를 수집해야 하며, 그러면서도 질문에 대한 답을 얻어야 할 필요가 있을 때만 그 자료를 취합해야 한다.

『안티프래질』(와이즈베리, 2013) 같은 베스트셀러를 여러 권 쓴 나심 니콜라스 탈레브가 유행시킨 **검은 백조**(black swan; 또는 흑조)라는 용어는 일어나리라고 기대하지 않았지만 실제로 일어났을 때 커다란 결과를 야기하는, 그리고 돌이켜 보면 얼마든지 일어날 수 있는 일임을 깨닫게 되는 사건을 뜻한다. 또 다른 저서 『블랙 스완』(동녘사이언스, 2008)에서 그는 관련 자료를 확보하는 것이 그런 드문 사건의 위험을 완화하는 데 도움이 되는 이유를 설명했다. 소프트웨어 공학에서는 검은 백조 사건이 생각보다 자주, 그리고 반드시 일어난다. 그런 종류의 사건을 애초에 방지하는 것은 사실상 불가능하므로, 유일한 방책은 그런 사건이 발생했을 때 업무 시스템에 치명적인 피해가 생기지 않도록 대응하는 데 필요한 정보를 최대한 많이 모으는 것이다. 관측 가능성은 강건한 시스템을 구축하고 미래의 검은 백조 사건을 완화하는 데 도움이 되는데, 이는 관측 가능성을 위해서는 언젠가 질문하게 될 임의의 질문에 대한 답을 구하는 데 필요한 모든 자료를 수집해야 하기 때문이다.

리눅스 컨테이너Linux container는 리눅스 커널의 여러 기능 위에 놓인 하나의 추상화 층으로, 컴퓨터 안에서 실행되는 프로세스들을 격리(isolation)하고 관리하는 역할을 한다. 전통적으로 커널kernel은 자원 관리를 담당하며, 작업(task)의 격리와 보안도 책임진다. 리눅스에서 컨테이너가 의존하는 커널의 주된 기능은 이름공간(namespace)과 흔히 cgroup으로 줄여 쓰는 제어 그룹(control group)이다. 이름공간은 작업들을 서로 격리하는 구성요소로, 한 이름공간 안에서 실행되는 작업은 마치 운영체제 안에서 자기 혼자만 실행되는 것처럼 느끼게 된다. cgroup은 자원 관리를 제공하는 구성요소이다. 운영의 관점에서 cgroup은 CPU나 디스크 I/O, 네트워크 등 임의의 자원 사용을 세밀하게 제어하는 수단이라 할 수 있다. 지난 10여 년간 리눅스 컨테이너가 인기를 끌면서 소프트웨어 공학자들이 대규모 분산 시스템과 계산 플랫폼(compute platform)을 설계하는 방식이 변했다. 클라우드 서비스 같은 '다중 입주(multitenant)' 컴퓨팅은 전적으로 커널의 이런 기능들에 의존해서 성장했다.

그런데 컴퓨팅 시스템이 리눅스 커널의 저수준 기능에 이처럼 크게 의존하면, 관측 가능성을 위해서는 이전과는 다른 새로운 복잡성과 정보까지 고려해서 시스템을 설계해야 한다. 커널은 사건 주도적(event-driven) 시스템이다. 즉, 커널의 모든 일은 흔히 이벤트라고 부르는 사건 표현에 기초해서 정의되고 실행된다. 파일을 여는 것도 일종의 이벤트이고, CPU가 임의

의 명령을 수행하는 것도 이벤트이고, 네트워크 패킷 하나를 받는 것도 이벤트이다. 이 책의 주제인 BPF(Berkeley Packet Filter; 버클리 패킷 필터)는 그러한 새로운 종류의 정보를 조사하는 능력을 제공하는 커널의 한 하위 시스템(subsystem)이다. BPF를 이용하면 커널이 임의의 이벤트를 발생했을 때 안전하게 실행되는 프로그램을 작성할 수 있다. 또한 BPF는 잘못된 BPF 프로그램 때문에 시스템이 폭주(crashing)하거나 오작동하는 일을 방지하는 안전 보장 장치도 갖추고 있다. BPF 덕분에, 컨테이너 기반의 새로운 플랫폼들을 시스템 개발자가 관찰하고 운영하는 데 도움이 되는 새로운 종류의 도구들을 만들 수 있게 되었다.

이 책은 임의의 컴퓨팅 시스템의 관측 가능성을 높이는 데 도움이 되는 BPF의 여러 강력한 기능을 제시한다. 또한 다양한 프로그래밍 언어로 BPF 프로그램을 작성하는 방법도 설명한다. 독자의 편의를 위해 이 책의 예제 코드를 모두 이 책을 위한 깃허브GitHub 저장소(*https://oreil.ly/lbpf-repo*)에 올려두었으니 적극 활용하기 바란다.

BPF의 기술적인 측면으로 들어가기 전에, 이 모든 일이 어떻게 일어났는지부터 살펴보자.

1.1 BPF의 역사

1992년에 스티븐 매캔Steven McCanne과 밴 제이컵슨Van Jacobson이 「The BSD Packet Filter: A New Architecture for User-Level Packet Capture」라는 논문을 썼다. 이 논문에서 저자들은 당시 최고 수준의 패킷 필터보다 20배 빠르게 작동하는 Unix 커널용 네트워크 패킷 필터의 구현을 설명했다. 패킷 필터packet filter는 시스템의 네트워크를 감시하는 응용 프로그램에게 커널이 정보를 직접 제공하는 데 특화된 소프트웨어이다. 응용 프로그램은 커널이 제공한 정보에 기초해서 패킷의 처리 방식을 결정한다. BPF는 패킷 필터링을 다음 두 가지 방식으로 혁신했다.

- 레지스터register 기반 CPU와 효율적으로 작동하도록 설계된 새 VM(virtual machine; 가상 기계)을 도입했다.
- 모든 패킷 정보를 복사하지 않고도 패킷들을 필터링할 수 있는 응용 프로그램별 버퍼를 사용했다. 이 덕분에 BPF가 의사결정을 내리는 데 필요한 자료의 양이 최소화되었다.

이러한 극적인 개선 덕분에 점차 모든 Unix 시스템이 메모리를 더 많이 사용하면서도 성능은 더 느린 기존 구현을 버리고 BPF를 네트워크 패킷 필터링에 채용했다. 저자들의 구현이 지금도 리눅스 커널을 비롯해 Unix 커널에서 파생된 여러 커널에 쓰이고 있다.

2014년 초반에는 알렉세이 스타로보이토프Alexei Starovoitov가 확장된 BPF 구현을 소개했다. 이 새 설계는 현대적인 하드웨어에 최적화된 것으로, 기존 BPF 해석기(interpreter)보다 빠른 명령어 집합을 산출한다. 또한 BPF VM의 레지스터 수도 늘었는데, 예전에는 32비트 레지스터 두 개뿐이었지만 확장 버전은 64비트 레지스터가 열 개이다. 레지스터가 늘고 비트 수도 증가해서 개발자가 함수 매개변수들로 더 많은 정보를 교환할 수 있게 되었으며, 결과적으로 좀 더 복잡한 프로그램을 작성할 수 있게 되었다. 이러한 변화와 기타 여러 개선점 덕분에, 흔히 eBPF라고 표기하는 확장된(extended) BPF는 기존 BPF 구현보다 최대 4배 빠르게 작동한다.

원래 이 새 구현은 네트워크 필터를 처리하는 내부 BPF 명령어 집합을 최적화하는 것이 목표였다. 당시 BPF는 여전히 커널 공간(kernel space; 또는 커널 영역)으로 한정되었으며, 사용자 공간(user space; 또는 사용자 영역)에서 커널이 처리할 BPF 필터를 작성할 수 있는 프로그램은 그리 많지 않았다. 이후 장들에서 설명할 tcpdump와 seccomp 등이 그런 프로그램이다. 오늘날에도 이 프로그램들은 여전히 구형 BPF 해석기를 위한 바이트코드를 생성하지만, 커널이 그 명령들을 그보다 훨씬 개선된 내부 표현으로 번역한다.

2014년 6월에 드디어 확장 BPF를 사용자 공간에서도 사용할 수 있게 되었는데, 돌이켜 보면 이때가 바로 BPF의 미래를 바꾼 변곡점이었다. 이 변경을 도입한 패치에서 알렉세이는 이렇게 말했다. "이 패치 집합은 eBPF의 잠재력을 보여준다."

이후 BPF는 최상위 커널 하위 시스템이 되었으며, 네트워킹 스택 이외의 곳에서도 쓰이기 시작했다. BPF 프로그램은 안전성(safety)과 안정성(stability)에 크게 강조를 둔 커널 모듈과 비슷한 모습이 되었다. 단, 커널 모듈과는 달리 BPF 프로그램은 커널을 다시 컴파일하지 않아도 되며, 폭주하는 일 없이 실행이 종료됨을 보장한다.

이러한 필수적인 안전 보장을 제공하는 것은 다음 장에서 좀 더 이야기할 BPF 검증기(verifier)이다. BPF 검증기는 그 어떤 BPF 프로그램이라도 폭주 없이 종료됨을 보장하며, 프로그램이 주어진 범위 밖의 메모리에 접근하지 못하는 하는 역할도 한다. 이러한 장점에는 어느 정도의 대가가 따르는데, 우선 프로그램의 최대 크기가 제한되며, 루프 반복 횟수도 제한된다. 이는 나쁜 BPF 프로그램 때문에 시스템의 메모리가 소진되는 일을 방지하기 위한 것이다.

사용자 공간에서도 BPF에 접근할 수 있게 하는 패치가 적용되면서 커널 개발자들은 `bpf`라고 하는 새로운 시스템 호출(system call, syscall)도 추가했다. 이 새 시스템 호출이 사용자 공간과 커널 사이의 주된 통신 통로로 쓰인다. 이 시스템 호출을 BPF 프로그램 및 맵에 사용하는 방법을 이 책의 제2장과 제3장에서 논의할 것이다.

BPF 맵은 커널과 사용자 공간 사이의 주된 자료 교환 수단이 된다. 제2장에서는 이 특화된 자료 구조를 이용해서 커널에서 자료를 수집하고 이미 커널 안에서 실행 중인 BPF 프로그램에 정보를 보내는 방법을 살펴본다.

이 책은 eBPF, 즉 확장 BPF(extended BPF)를 다룬다. 특별한 언급이 없는 한 이후의 BPF는 모두 eBPF를 뜻한다.[1] 첫 확장 버전이 나오고 5년이 흐르면서 BPF는 크게 진화했다. 이 책은 BPF 프로그램과 BPF 맵의 진화뿐만 아니라 이 진화에 영향을 받은 커널 시스템의 변화도 상세히 소개한다.

1.2 구조

커널 안에 놓인 BPF의 구조는 매혹적이다. 이 책 전반에서 구조의 세부사항들을 좀 더 자세히 살펴볼 것이므로, 여기서는 전체적인 구조를 간략하게나마 개괄하고 넘어간다.

앞에서 언급했듯이 BPF는 격리된 환경에서 코드 명령문을 실행하는 고도로 진보된 VM이다. 어떤 면에서 BPF는 고수준 프로그래밍 언어로부터 컴파일된 기계 코드를 실행하는 특화된 프로그램인 JVM(Java Virtual Machine; 자바 가상 기계)과 비슷하다. LLVM이 BPF를 지원하므로(GCC도 조만간 BPF를 지원할 계획이다) C 코드를 BPF 명령문들로 컴파일하는 것이 가능하다. 컴파일된 코드(BPF 프로그램)는 BPF 검증기를 거친다. 검증기는 코드가 커널에서 실행해도 안전한지 확인한다. 덕분에 커널 폭주를 일으킬 만한 코드가 걸러진다. 코드가 안전하다면 BPF 프로그램이 커널에 적재(loading)된다. 리눅스 커널은 BPF 명령문들을 위한 JIT(just-in-time) 컴파일러도 갖추고 있다. 이 JIT 컴파일러는 프로그램이 검증된 직후 BPF 바이트코드를 기계어 코드로 변환함으로써 실행 시점에서 그러한 변환을 수행하는 부담

1 eBPF와 구분하기 위해 원래의 BPF를 cBPF, 즉 'classic(고전적)' BPF라고 부르기도 한다 — 옮긴이(이하 이 책의 모든 각주는 역자 주이며, '옮긴이' 표시는 생략합니다).

을 줄인다. 이러한 구조의 한 가지 흥미로운 측면은 시스템을 재시작하지 않고도 BPF 프로그램을 커널에 올릴 수 있다는 것이다. 필요할 때 언제라도 적재할 수 있으며, 또한 적절한 초기화 스크립트를 이용해서 시스템 시작 시 BPF 프로그램들을 자동으로 적재하는 것도 가능하다.

주어진 BPF 프로그램을 실행하려면 커널은 먼저 그 프로그램을 붙일 실행 지점(execution point)을 알아야 한다. 커널에 BPF 프로그램을 붙일 수 있는 지점은 여러 개이며 커널이 갱신되면서 계속 늘어나고 있다. 부착 가능한 실행 지점은 BPF 프로그램의 유형(종류)에 따라 결정된다. BPF 프로그램의 여러 유형은 다음 장에서 논의한다. 실행 지점이 결정되면 커널은 그 프로그램이 커널로부터 자료를 받는 데 사용할 특별한 보조 함수들을 활성화한다. 이에 의해 실행 지점과 BPF 프로그램이 더욱 단단히 묶이게 된다.

BPF 구조의 마지막 구성요소는 커널과 사용자 공간이 자료를 공유하는 데 사용하는 BPF 맵map이다. 맵은 제3장에서 논의한다. BPF 맵은 자료 공유를 위한 양방향 자료 구조이다. 즉, 커널과 사용자 공간 모두 이 맵을 읽고 쓸 수 있다. BPF에는 여러 자료 구조가 쓰이는데, 단순한 배열에서부터 해시 맵, 그리고 BPF 프로그램 전체를 저장할 수 있는 특화된 맵 등 다양하다.

이 책 전반에서 BPF 구조의 모든 구성요소를 좀 더 자세히 살펴볼 것이다. 또한 BPF의 확장성과 자료 공유 능력의 장점을 스택 추적 분석에서 네트워크 필터링, 런타임 격리 같은 다양한 주제들과 함께 이야기한다.

1.3 결론

우리 필자들은 독자가 일상 업무에서 리눅스 하위 시스템을 다루는 데 필요한 기본적인 BPF 개념들에 익숙해지는 데 도움을 주기 위해 이 책을 썼다. BPF는 여전히 발전 중인 기술이며, 이 책을 쓰는 동안에도 새로운 개념과 패러다임이 생기고 발전한다. BPF의 기본 구성요소들을 확실하게 설명하고 정리함으로써 BPF에 대한 여러분의 지식을 확장하는 데 이 책이 도움이 되었으면 좋겠다.

다음 장인 제2장에서는 BPF 프로그램의 구조를 살펴보고 커널이 BPF 프로그램을 실행하는 과정을 설명한다. 또한 커널 안에 BPF 프로그램을 붙이는 지점들도 이야기한다. 제2장에서 여러분은 BPF 프로그램이 어떤 자료를 어떻게 사용할 수 있는지 알게 될 것이다.

제2장

생애 첫 BPF 프로그램

BPF VM은 커널이 발생한 이벤트에 반응해서 명령을 실행하는 능력을 갖추고 있다. 그런데 모든 BPF 프로그램이 커널이 발생한 모든 이벤트에 반응할 수 있는 것은 아니다. 하나의 프로그램을 BPF VM에 적재할 때는 그 프로그램의 유형(type)을 지정해야 한다. 커널은 프로그램의 유형을 보고 그 프로그램에 대해 발생할 이벤트를 결정한다. 또한, BPF 검증기가 그 프로그램이 사용해도 되도록 허용하는 보조 함수들 역시 이 프로그램 유형에 따라 달라진다. 간단히 말해서 프로그램의 유형을 선택하는 것은 프로그램이 구현할 인터페이스를 선택하는 것에 해당한다. 적절한 인터페이스를 구현하는 BPF 프로그램은 항상 적절한 종류의 자료에 접근하게 된다. 특히, 어떤 인터페이스를 구현하느냐에 따라 프로그램이 네트워크 패킷에 직접 접근할 수 있는지가 결정된다.

이번 장은 여러분이 생애 첫 BPF 프로그램을 작성하는 방법을 보여준다. 또한, 여러분이 만들 수 있는(이 책을 쓰는 시점을 기준으로) 여러 BPF 프로그램 유형도 소개한다. 지난 몇 년간 커널 개발자들은 BPF 프로그램을 붙일 수 있는 다수의 진입점을 커널에 추가했다. 이 작업은 아직 끝나지 않았으며, 커널 개발자들은 BPF를 활용하는 새로운 방법을 계속해서 찾고 있다. BPF로 할 수 있는 일이 어떤 것인지 여러분이 감을 잡게 하자는 취지로, 이번 장에서는 가장 유용한 프로그램 유형들에 초점을 둔다. 이후의 장들에서 좀 더 다양한 유형의 BPF 프로그램들을 만나게 될 것이다.

이번 장에서는 또한 BPF 프로그램의 실행에서 BPF 검증기가 차지하는 역할도 설명한다. 이 검증기는 여러분이 작성한 코드가 커널 안에서 실행하기에 안전한지 확인하며, 메모리가 소

진되거나 커널이 갑자기 폭주하는 등의 원치 않은 결과가 발생하지 않도록 프로그램을 작성하는 데 도움이 되는 정보를 제공한다. 그 부분은 잠시 후에 다시 이야기하고, 먼저 간단한 BPF 프로그램을 직접 작성해 보자.

2.1 BPF 프로그램 작성

BPF 프로그램을 만드는 가장 흔한 방법은 C 언어의 부분집합에 해당하는 언어로 소스 코드를 작성하고 그것을 LLVM으로 컴파일하는 것이다. LLVM은 다양한 종류의 바이트코드를 산출할 수 있는 범용 컴파일러이다. 지금 예에서 LLVM는 BPF 어셈블리 코드(나중에 커널에 적재할)를 생성하는 역할을 한다. BPF 어셈블리는 이 책에서 자세히 다루지 않는다. 이 문제를 두고 필자들이 오랜 시간 논의를 했는데, BPF 어셈블리 언어에 지면을 할애하느니 특정 상황에서 사용할 수 있는 구체적인 예제들을 제시하는 게 더 낫다는 결론을 내렸다. BPF 어셈블리에 관해서는 웹이나 BPF 매뉴얼 페이지에서 여러 참고 자료를 어렵지 않게 찾을 수 있다. 이후의 장들에서 짧은 BPF 어셈블리 예제들이 가끔 등장하긴 하지만, C보다 어셈블리가 더 나은 경우에 한해서일 뿐이다. 예를 들어 커널로 들어오는 시스템 호출들을 제어하는 seccomp 필터가 그런 경우인데, seccomp에 관해서는 제8장에서 좀 더 이야기한다.

커널은 `bpf`라는 시스템 호출을 제공한다. 이 함수의 기본 용도는 컴파일된 BPF 프로그램을 BPF VM에 적재하는 것이지만, 그 외에도 여러 용도가 있다. 이후의 장들에서 여러 용례를 만나게 될 것이다. 커널은 또한 BPF 프로그램의 적재를 추상화해 주는 여러 편의용 보조 수단을 제공한다. 이번 장의 첫 예제 코드인 "Hello World" BPF 프로그램도 그런 보조 함수(helper) 몇 가지를 사용한다.

```
#include <linux/bpf.h>
#define SEC(NAME) __attribute__((section(NAME), used))

static int (*bpf_trace_printk)(const char *fmt, int fmt_size,
                               ...) = (void *)BPF_FUNC_trace_printk;

SEC("tracepoint/syscalls/sys_enter_execve")
int bpf_prog(void *ctx) {
  char msg[] = "Hello, BPF World!";
```

```
    bpf_trace_printk(msg, sizeof(msg));
    return 0;
}

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

이 첫 번째 프로그램에 몇 가지 흥미로운 개념이 담겨 있다. 우선, 이 프로그램은 `SEC` 매크로를 이용해서 섹션 특성(section attribute)을 설정하는데, 이 특성은 BPF VM이 이 프로그램을 실행할 시점을 정의한다. 좀 더 구체적으로, 이 예제는 `execve` 시스템 호출의 추적점(tracepoint)이 검출되었을 때 BPF VM이 이 BPF 프로그램을 실행해야 함을 `SEC` 매크로로 지정한다. 추적점은 커널의 이진 코드 안에 있는 정적인 표식(mark)으로, 개발자는 추적점들을 이용해서 커널의 실행 흐름 안의 특정 지점에 자신이 원하는 코드를 주입한다. 추적점에 관해서는 제4장에서 좀 더 이야기할 것이다. 일단 지금은 `execve`가 다른 프로그램을 실행하는 명령이라는 점만 기억하기 바란다. 결과적으로, 한 프로그램이 다른 프로그램을 실행하는 상황을 커널이 포착할 때마다 `Hello, BPF World!`라는 메시지가 출력된다.

소스 코드의 마지막 줄은 이 프로그램의 사용권(license)을 명시한 것이다. 리눅스 커널은 GPL 사용권을 따르므로, 오직 GPL을 따르는 프로그램만 커널에 적재할 수 있다. 만일 GPL 이외의 사용권을 설정하면 커널은 프로그램의 적재를 거부한다. 프로그램 본문에는 `bpf_trace_printk`라는 함수가 쓰였는데, 이 함수는 커널 추적 로그(tracing log)에 메시지를 기록한다. 이 로그는 */sys/kernel/debug/tracing/trace_pipe*에 있다.

이제 이 첫 번째 프로그램을 Clang으로 컴파일해서 유효한 ELF 이진 파일을 만든다. ELF는 리눅스 커널이 적재할 수 있는 이진 실행 파일 형식이다. 앞의 예제 코드를 `bpf_program.c`라는 이름의 소스 파일로 저장했다고 할 때, 이를 컴파일하는 명령[1]은 다음과 같다.

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

독자의 편의를 위해 이 책의 깃허브 저장소(*https://oreil.ly/lbpf-repo*)에 예제 코드와 함께 BPF 프로그램 컴파일용 셸 스크립트 또는 Makefil을 올려두었으니, 이 Clang 컴파일 명령을

1 명령줄이나 셸, 터미널과 관련해서 '명령'은 개별 실행 파일 또는 셸 내장 명령을 뜻하기도 하고, 지금 예처럼 개별 명령과 각종 매개변수 및 옵션의 조합을 뜻하거나 심지어 입출력 재지정이나 파이프 연결까지 포함한 명령줄 전체를 뜻하기도 한다. 특별히 혼동할 여지가 없는 한 이 책에서는 구분 없이 그냥 '명령'으로 표기한다.

일일이 외워 둘 필요는 없다.

이제 첫 BPF 프로그램의 컴파일을 마쳤다. 다음으로 할 일은 컴파일된 코드를 커널에 적재하는 것이다. 앞에서 언급했듯이, 커널은 프로그램의 컴파일과 적재(load) 과정을 추상화하는 보조 함수들을 제공한다. 지금 필요한 것은 *bpf_load.h* 헤더 파일에 선언된 `load_bpf_file`이라는 함수인데, 이 함수는 주어진 이진 파일을 커널에 적재한다. 다음은 이 함수를 이용해서 앞에서 만든 이진 파일을 커널에 적재하는 프로그램이다. 앞의 BPF 프로그램과 마찬가지로, 이 예제 코드 역시 이 책의 깃허브 저장소에 있다.

```
#include <stdio.h>
#include <uapi/linux/bpf.h>
#include "bpf_load.h"

int main(int argc, char **argv) {
  if (load_bpf_file("bpf_program.o") != 0) {
    printf("The kernel didn't load the BPF program\n");
    return -1;
  }

  read_trace_pipe();

  return 0;
}
```

다음은 이 소스 코드를 컴파일, 링크해서 ELF 실행 파일을 만드는 셸 스크립트이다. 이 프로그램은 BPF VM에 적재할 것이 아니므로 `-target` 옵션은 지정할 필요가 없다. 각종 디렉터리를 지정한 것 외에, `load_bpf_file` 함수의 정의가 있는 *bpf_load.c* 파일도 함께 컴파일, 링크함을 주의하기 바란다. 여러 파일이 관여하는 빌드 작업을 수행할 때는 이처럼 스크립트를 사용하는 것이 편하다.[2]

```
TOOLS=/kernel-src/tools
SAMPLES=/kernel-src/samples/bpf
clang -o loader -l elf -l bpf \
```

2 이 스크립트는 커널 소스 트리가 */kernel-src* 디렉터리에 있다고 가정한다. 이 예제뿐만 아니라 이 책의 대부분의 예제는 독자가 원서 깃허브 저장소의 README.md에 나온 절차에 따라 예제 실행 환경을 구성했다고 가정을 깔고 있다. 참고로 이 책을 번역하는 현재(2020년 2월) 원서 깃허브 저장소는 아직 완전하지 않다. 일부 예제 코드 및 관련 파일들이 누락되었고 컴파일 오류나 실행 오류를 발생하는 코드도 있는데, 저자들이 그 사실을 알고 있으며 조금씩 해결하는 중이다.

```
-I${SAMPLES} \
-I${TOOLS}/lib \
-I${TOOLS}/perf \
-I${TOOLS}/include \
${SAMPLES}/bpf_load.c \
loader.c
```

이 스크립트를 실행해서 컴파일 및 링크가 잘 진행되었다면 *loader*라는 실행 파일이 생긴다. 이 실행 파일은 반드시 `sudo`로 실행해야 한다(이를테면 `sudo ./loader`). `sudo`는 현재 사용자에게 일시적으로 루트 권한을 부여하는 리눅스 명령이다. 대부분의 BPF 프로그램은 오직 루트 권한을 가진 사용자만 커널에 올릴 수 있으므로, `sudo` 없이 이 프로그램을 실행하면 오류 메시지가 나온다.

이 프로그램을 실행하면, 여러분이 컴퓨터로 아무 일도 하지 않았는데도 몇 초 후에 `Hello, BPF World!` 메시지가 나타나기 시작할 것이다. 이는 배경에서 실행되는 프로그램들이 다른 프로그램들을 실행하기 때문이다.

프로그램을 종료하면 터미널에 더 이상 메시지가 나타나지 않는다. BPF 프로그램은 그것을 적재한 프로그램이 종료되면 즉시 VM에서 제거된다. 적재 프로그램을 종료해도 BPF 프로그램이 계속 남아 있게 만드는 방법도 있다. 사실 이 책의 목적에서 BPF 프로그램은 다른 프로세스의 실행 여부와는 무관하게 배경에서 계속 실행되면서 시스템의 자료를 수집해야 하므로 이는 중요한 문제이다. 그렇지만 지금 너무 여러 가지를 이야기하면 부담이 될 것이므로, 그 방법은 나중에 때가 되면 이야기하기로 한다.

이렇게 해서 아주 간단한 BPF 프로그램을 만들고 실행해 보았다. 그럼 여러분이 작성할 수 있는 다양한 종류의 BPF 프로그램들을 살펴보자. 이들을 통해서 리눅스 커널 안의 여러 하위 시스템들에 접근할 수 있다.

2.2 BPF 프로그램 유형

아주 깔끔하게 나누어지지는 않지만, 이번 절에서 다루는 프로그램 유형들은 그 목적에 따라 크게 두 범주로 나뉜다.

첫 범주는 **추적**(tracing)이다. BPF 프로그램 중에는 시스템 안에서 어떤 일이 일어나는지 파악하는 데 도움이 되는 것들이 많다. 이 범주의 프로그램은 시스템과 바탕 하드웨어의 작동 방식에 관한 직접적인 정보를 제공한다. 이 범주의 프로그램은 특정 프로그램과 관련된 메모리 영역에 접근할 수 있으며, 실행 중인 프로세스에서 실행 추적 정보(execution trace; 또는 실행 자취)를 추출할 수 있다. 또한 각각의 특정 프로세스에 할당된 자원들에도 직접 접근할 수 있다. 이를테면 파일 서술자(file descriptor)나 CPU, 메모리 사용량을 바로 알 수 있다.

둘째 범주는 **네트워킹**networking이다. 이 범주의 프로그램들로는 시스템의 네트워크 소통량[3]을 조사하거나 조작할[4] 수 있다. 예를 들어 네트워크 인터페이스에서 온 패킷들을 걸러내거나(필터링), 그런 패킷들을 완전히 폐기할 수도 있다. 이 범주의 프로그램들은 커널 안에서 네트워킹의 여러 처리 단계 중 어디에 부착하느냐에 따라 그 유형이 세분된다. 각 세부 유형에는 나름의 장단점이 있다. 예를 들어 네트워크 드라이버가 패킷을 받은 즉시 발생하는 네트워크 이벤트에 붙인 BPF 프로그램은 효율적이긴 하지만, 패킷에 관해서는 많은 정보를 얻지 못한다. 그 시점에서는 커널이 패킷에 관해 아는 것이 별로 없기 때문이다. 반대로, 패킷이 사용자 공간으로 넘어가기 직전에 발생하는 네트워크 이벤트에 BPF 프로그램을 부착할 수도 있다. 그런 프로그램은 좀 더 풍부한 정보를 근거로 결정을 내릴 수 있지만, 대신 패킷을 완전히 처리하는 데 든 비용(더 일찍 패킷을 폐기했다면 피할 수 있었던)을 허비하게 된다.

아래의 프로그램 유형들은 이 두 범주로 구분된 것이 아니다. 이들은 그냥 커널에 추가된 시간순으로 나열된 것이다. 또한, 여러분에게 유용할 만한 유형들만 개별적으로 소개하고, 잘 쓰이지 않는 유형들은 이번 절 끝에서 간단하게만 언급한다. 만일 여기서 자세히 다루지 않는 프로그램 유형들에 관심이 있다면 해당 매뉴얼 페이지(`man 2 bpf` 또는 *https://oreil.ly/qXl0F*)을 참고하기 바란다.

2.2.1 소켓 필터 프로그램

`BPF_PROG_TYPE_SOCKET_FILTER`는 리눅스 커널에 처음 추가된 프로그램 유형이다. 원(raw)

---

3 '소통량'은 traffic을 옮긴 것으로, 비록 '~량'으로 끝나긴 하지만 네트워크로 주고받은 정보의 양뿐만 아니라 그러한 정보 자체 또는 정보를 주고받는 행위까지도 아우르는 용어이다.

4 이 책에서 조작은 (대체로 악의적인 목적으로) 뭔가를 위조하거나 변조하는 造作이 아니라 어떤 장치나 대상을 다루는 것을 뜻하는 操作이다.

소켓에 부착되는 이 유형의 프로그램은 그 소켓이 처리한 모든 패킷에 접근할 수 있다. 소켓 필터 프로그램이 그 패킷들의 내용이나 수신 주소를 변경하지는 못한다. 이 프로그램 유형은 전적으로 관측 가능성을 위한 것이다. 이 유형의 프로그램이 받는 메타자료(metadata)에는 네트워크 스택에 관한 정보(이를테면 패킷 전달에 쓰이는 프로토콜 종류)가 포함되어 있다.

소켓 필터링과 기타 네트워크 프로그램에 관해서는 제6장에서 자세히 논의한다.

2.2.2 kprobe 프로그램

추적(tracing)을 다루는 제4장에서 이야기하겠지만, kprobe는 커널의 특정 호출 지점(call point)들에 동적으로 부착할 수 있는 함수들을 아우르는 용어이다. kprobe 유형의 BPF 프로그램은 kprobe 처리기(handler)로 작동한다. 이 유형을 나타내는 식별자(열거형 값)는 `BPF_PROG_TYPE_KPROBE`이다. BPF VM은 주어진 kprobe 프로그램이 언제라도 안전하게 실행됨을 보장하는데, 이는 전통적인 kprobe 모듈에는 없는 장점이다. 단, kprobe가 커널 안의 안정적인(stable) 진입점이라고 간주되지는 않음을 유념해야 한다. 커널 버전이 바뀌면 특정 진입점의 지원 여부가 달라질 수 있으므로, 여러분이 작성한 kprobe BPF 프로그램이 현재 실행 중인 커널의 버전과 호환되는지 확인할 필요가 있다.

kprobe 유형의 BPF 프로그램을 작성할 때는 그 프로그램이 함수 호출의 첫 명령으로서 실행될 것인지 아니면 호출이 완료되는 시점에서 실행될 것인지를 설정해야 한다. 이 설정은 첫 예제 프로그램처럼 프로그램 소스 코드의 섹션 헤더 부분에서 섹션 특성으로 지정한다. 예를 들어 커널이 시스템 호출 `exec`를 실행할 때 전달된 인수들을 조사하고 싶다면 섹션 헤더 `SEC("kprobe/sys_exec")`를 지정하면 된다. 또는, `exec` 시스템 호출의 반환값을 조사하고 싶으면 섹션 헤더 `SEC("kretprobe/sys_exec")`를 지정한다.

kprobe는 이후의 장들에서 좀 더 자세히 다룬다. kprobe는 BPF를 이용한 추적 작업의 필수 요소이다.

2.2.3 추적점 프로그램

이 유형의 BPF 프로그램은 커널이 제공하는 추적점 처리기(tracepoint handler)에 부착된

O'REILLY® Linux Observability with BPF

유능한 엔지니어로 레벨업할 BPF 완벽 활용 가이드

이 책은 시스템 엔지니어를 위한 리눅스 커널의 BPF VM에 관한 전문 지식을 제공합니다. BPF 프로그램의 수명 주기를 자세히 설명할 뿐만 아니라, 커널에서 벌어지는 사건들을 감시, 추적, 관찰하는 코드를 주입해 커널의 행동을 좀 더 안전하고 안정적으로 관찰하고 수정하는 방법을 알려줍니다. C, Go, 파이썬으로 작성된 다양한 예제 코드로 BPF의 필수 개념을 익히고 나면 일상 업무를 좀 더 효율적으로 수행하게 됨은 물론 성능 최적화, 네트워킹, 보안에 관한 기본기도 향상될 것입니다.

- BPF 맵으로 커널과 사용자 공간 통신 채널 확립하기
- BCC 프레임워크로 추적 프로그램 작성하기
- 고품질 BPF 기반 도구인 BPFTool, BPFTrace, eBPF Exporter 활용법 알아보기
- 네트워크의 소통량(traffic)을 분석하고 메시지 전달 제어하기
- 고성능 패킷 처리기인 XDP 프로그램 알아보기
- 리눅스 커널 보안 능력(capability)과 seccomp로 필터 작성하기

만약 20년 전에 까다로운 시스템의 문제 해결 방법을 내게 물었다면, tcpdump와 strace를 알려줬을 것입니다. 하지만 이제는 그냥 이 책을 건네겠습니다.

제롬 페타초니, Tiny Shell Script LLC 설립자

관련 도서

처음 배우는 암호화	bash를 활용한 사이버 보안 운영

예제 소스 github.com/bpftools/linux-observability-with-bpf

보안 / 해킹

9 791162 243053 93000

ISBN 979-11-6224-305-3

정가 22,000원

O'REILLY® 한빛미디어 Hanbit Media, Inc.