Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.32, NO.2, Apr. 2022 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2022.32.2.213

BPFast: 클라우드 환경을 위한 eBPF/XDP 기반 고속 네트워크 패킷 페이로드 검사 시스템*

유 명 성,[1†] 김 진 우,[2] 신 승 원,[3] 박 태 준[4‡]
[1,3]한국과학기술원 (대학원생, 교수), [2]광운대학교 (교수), [4]전남대학교 (교수)

BPFast: An eBPF/XDP-Based High-Performance Packet Payload Inspection System for Cloud Environments*

Myoung-sung You,[1†] Jin-woo Kim,[2] Seung-won Shin,[3] Tae-june Park[4‡]
[1,3]KAIST (Graduate student, Professor), [2]Kwangwoon University (Professor),
[4]Chonnam National University (Professor)

요 약

컨테이너 기술은 클라우드 환경에서 마이크로서비스의 효율적인 구축 및 운영을 가능하게 했지만, 심각한 보안 위협도 함께 가져왔다. 다수의 컨테이너가 서비스 구성을 위해 네트워크로 연결되기 때문에 공격자는 탈취한 컨테이너에서 네트워크 공격을 수행해 인접한 다른 컨테이너를 공격할 수 있다. 이러한 위협을 막기 위해 다양한 컨테이너 네트워크 보안 솔루션이 제안되었으나 성능적인 측면에서 많은 한계점을 갖고 있다. 특히 이들은 컨테이너 네트워크 보안에 필수적인 패킷 페이로드 검사 과정에서 매우 큰 네트워크 성능 저하를 일으킨다. 본 논문에서는 이러한 문제를 해결하기 위해 클라우드 환경을 위한 eBPF/XDP 기반 고속 패킷 페이로드 검사 시스템인 *BPFast*를 제안한다. *BPFast*는 별도의 유저 수준 컴포넌트 없이 커널 영역에서 컨테이너가 전송한 패킷의 헤더와 페이로드를 검사하여 컨테이너를 네트워크 공격에서부터 보호한다. 본 논문에서는 Kubernetes 환경에서 진행한 실험을 통해 *BPFast* 프로토타입이 Cilium, Istio 등 최신 솔루션보다 최대 7배 더 빠르게 동작할 수 있음을 증명했다.

ABSTRACT

Containerization, a lightweight virtualization technology, enables agile deployments of enterprise-scale microservices in modern cloud environments. However, containerization also opens a new window for adversaries who aim to disrupt the cloud environments. Since microservices are composed of multiple containers connected through a virtual network, a single compromised container can carry out network-level attacks to hijack its neighboring containers. While existing solutions protect containers against such attacks by using network access controls, they still have severe limitations in terms of performance. More specifically, they significantly degrade network performance when processing packet payloads for L7 access controls (e.g., HTTP). To address this problem, we present *BPFast*, an eBPF/XDP-based payload inspection system for containers. *BPFast* inspects headers and payloads of packets at a kernel-level without any user-level components. We evaluate a prototype of *BPFast* on a Kubernetes environment. Our results show that *BPFast* outperforms state-of-the-art solutions by up to 7x in network latency and throughput.

Keywords: Cloud security, Network security, Payload inspection, eBPF

Received(02. 08. 2022), Modified(03. 30. 2022), Accepted(03. 30. 2022)

* 본 연구는 방위사업청과 국방과학연구소가 지원하는 미래 전투체계 네트워크기술 특화연구센터 사업의 일환으로 수행되었습니다. (UD190033ED)

† 본 논문은 2021년도 한국정보보호학회 호남지부 학술대회에 발표한 우수논문을 개선 및 확장한 것임

† 주저자, famous77@kaist.ac.kr

‡ 교신저자, taejune.park@jnu.ac.kr(Corresponding author)

I. Introduction

Linux 컨테이너[1]는 경량화된 가상화 기술로 기존 가상 머신 (virtual machine)보다 빠르고 효율적인 애플리케이션 배포를 가능하게 해준다. 이러한 장점으로 인해 컨테이너는 Amazon, Google, Microsoft 등 주요 클라우드 서비스 제공업체에서 대규모 마이크로서비스 구성을 위한 핵심 기술로 사용되고 있다. 컨테이너 기술에 대한 적극적인 도입은 컨테이너로 인한 보안 위협의 증가도 함께 가져왔다. 컨테이너들은 마이크로서비스 구성을 위해 가상 네트워크로 서로 연결되어 동작한다. 그 때문에 공격자는 탈취한 컨테이너에서 TCP 세션 하이재킹[2], DNS 스푸핑[3] 등의 네트워크 공격을 수행해 인접한 다른 컨테이너를 탈취하거나 호스트 시스템을 마비시킬 수도 있다[2, 4]. 실제로 Tripwire의 컨테이너 보안 보고서에는 2018년에 인터넷 서비스 기업의 60%가 컨테이너로 인한 보안 사고를 겪은 것으로 나타나 있다[5].

이러한 위협을 막기 위해 Cilium[6], Bastion[2], Istio[7] 등 다양한 컨테이너 네트워크 보안 솔루션이 등장했다. 이들은 관리자가 지정한 네트워크 정책에 따라 컨테이너가 전송한 패킷을 필터링한다. 이를 통해 이들은 컨테이너 사이의 비 인가된 네트워크 접근을 차단하고 네트워크 공격으로부터 컨테이너를 보호한다. 하지만 현재의 솔루션들은 아직 성능적인 측면에서 여러 한계점을 갖고 있다. 대표적으로, 이들은 패킷의 페이로드 검사 과정에서 매우 큰 네트워크 성능 저하를 일으킨다. 컨테이너가 주로 HTTP REST API[8] 등 L7 프로토콜을 사용해 통신하기 때문에 L7 접근제어를 위한 패킷 페이로드 검사는 네트워크 보안에 있어 필수적이다. 하지만 본 논문에서 실험한 결과, 최신 솔루션이라도 페이로드 검사를 활성화하면 네트워크 지연이 최대 10배 이상 증가하고, 네트워크 처리량도 최대 1/9까지 줄어드는 등 컨테이너 간 통신 성능이 심각하게 감소했다. 따라서 현재 솔루션들은 AR/VR, 자율주행차, 산업용 IoT 등 네트워크 지연에 민감하고 높은 네트워크 처리량을 요구하는 서비스에는 사실상 적용하기 어렵다. 본 연구에서는 이러한 문제를 해결하기 위해 컨테이너 환경을 위한 고성능 패킷 페이로드 검사 시스템인 *BPFast*를 제안한다. 이 시스템은 Linux 커널의 최신 네트워킹 기능인 eBPF[9] 와 XDP[10]를 활용해 컨테이너 간의 비 인가된 네트워크 접근을 차단하면서 기존 솔루션보다 최대 7배 빠르게 컨테이너가 보낸 패킷의 페이로드를 검사한다. *BPFast*는 컨테이너 네트워크 접근제어를 위해 단독 사용뿐 아니라, Cilium, Bastion 등 기존 네트워크 보안 솔루션의 고속 페이로드 검사용 add-on으로써 함께 사용될 수 있다.

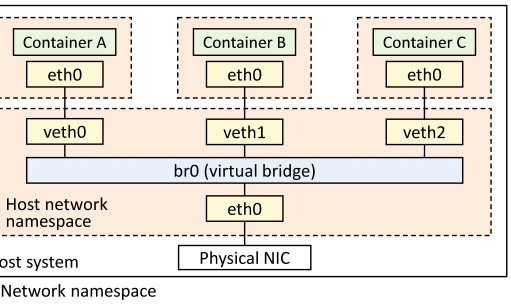


Fig. 1. A general container networking architecture of modern container platforms.

II. Background and Motivation

2.1 Linux Container

Linux 컨테이너 (container)[1]는 운영체제 수준 가상화 기술이다. 하나의 호스트 운영체제에서 실행되는 모든 컨테이너들은 호스트 운영체제의 커널을 공유해 사용하며 다양한 리눅스 커널 기능을 통해 서로 논리적으로 격리된다. 기존 가상 머신 (virtual machine) 기술은 하드웨어 자원까지 가상화하여 호스트 내에서 독립된 실행환경을 만든다. 이에 반해 컨테이너가 채용한 커널 자원 공유 모델은 컨테이너가 기존 가상 머신 기술보다 빠른 시작 시간과 효율적인 자원 활용성을 갖게 만들어줬다. 이러한 이점으로 인해 컨테이너는 마이크로서비스 구성에 있어 가상 머신보다 효율적이며, 웹 서비스, AR/VR 서비스 등 다양한 클라우드 서비스의 운영에 활발히 사용되고 있다.

2.2 Container Network Architecture

2.2.1 Network Namespace

현대의 클라우스 서비스는 대부분 마이크로서비스

형태로 구현되며 네트워크를 통해 서로 연결된 다수의 컨테이너를 통해 배포/운영된다[11]. Fig. 1은 Docker[12], OpenVZ[13] 등 대표적인 컨테이너 플랫폼에서 채택한 컨테이너 네트워킹 구조를 나타낸다. 각 컨테이너는 서로 격리된 네트워크 환경인 네트워크 네임스페이스[14]를 갖고 동작하고 각 네트워크 네임스페이스는 독립된 네트워크 자원이 (네트워크 인터페이스, 라우팅 테이블 등) 할당된다. 따라서 컨테이너는 다른 컨테이너의 네트워크 자원이나 트래픽에 직접 접근할 수 없다. 호스트 OS 역시 호스트 시스템에 연결된 물리적인 네트워크 인터페이스 카드 (NIC)에 대한 인터페이스 등 호스트의 네트워크 자원이 할당된 네트워크 네임스페이스를 가지고 있다.

2.2.2 Bridge Network

컨테이너 플랫폼은 이처럼 독립된 네트워크 네임스페이스를 갖는 컨테이너들을 연결하기 위해 기본적으로 브리지 네트워크[15]를 사용한다. 컨테이너 플랫폼은 호스트 네트워크 네임스페이스에 각 컨테이너와 1대1로 연결된 호스트 측 가상 인터페이스를 (veth) 만들고 이들을 가상 브리지로 (br0) 연결하여 컨테이너들이 통신할 수 있게 해준다. 예를 들어 Fig. 1에서 컨테이너 A가 B로 전송한 패킷은 먼저 컨테이너 A의 가상 인터페이스인 veth0로 전달되어 가상 브리지인 br0로 수신된다. 이후 br0를 통해 veth1로 전달되어 (forwarding) 컨테이너 B에서 수신된다.

이러한 컨테이너 간 통신은 다수의 컨테이너를 연결해 다양한 클라우드 서비스를 효율적으로 구성 및 관리할 수 있게 해준다. 하지만 컨테이너 간 통신은 동시에 공격자가 탈취한 컨테이너를 이용해 다른 컨테이너에 네트워크 공격을 할 수 있는 환경을 제공한다. 실제로 많은 선행 연구에서 공격자에게 탈취된 컨테이너가 ARP 스푸핑 및 TCP 세션 하이재킹과 같은 네트워크 공격을 통해 주변 다른 컨테이너뿐 아니라 호스트 시스템까지도 공격할 수 있음이 증명되었다[2, 4, 16].

2.3 eBPF/XDP

extended Berkeley Packet Filter (eBPF)[9]는 네트워크 패킷 처리를 위해 개발된 Linux 커널 내부의 가상 머신이다. 사용자가 작성한 eBPF 프로그램은 다양한 커널 내부의 다양한 네트워크 훅 (hook)에 등록되어 실행되며 커널의 네트워크 패킷 처리 과정을 변경하거나 대체하게 된다. eBPF 프로그램은 커널 영역에서 실행되기 때문에 실행 안전성을 위한 많은 제약 조건을 갖고 있으며, 프로그램 실행 전 안전성 검사 과정을 거쳐 제약 조건을 준수하지 않을 경우 실행이 불가능하다. 대표적으로 프로그램의 최대 길이가 제한되며, 고정된 길이의 반복문 (bounded loop)만 사용할 수 있다. 또한 복잡한 자료구조나 라이브러리 함수를 호출할 수 없다. 다만 eBPF 프로그램은 키-값 저장소인 eBPF 맵을 통해 유저 영역에서 실행되는 프로그램과 실시간으로 서로 정보를 교환할 수 있다.

eXpress Data Path (XDP)[10]는 eBPF 프로그램을 등록해 실행할 수 있는 커널 네트워크 훅이다. XDP에 등록된 eBPF 프로그램은 특정 네트워크 인터페이스에 패킷이 수신될 때마다 실행되어 패킷을 처리한다. 이때 등록된 eBPF 프로그램은 디바이스 드라이버 수준에서 실행되기 때문에 고속으로 동작하며, 커널 네트워크 스택보다 먼저 패킷을 처리할 수 있으며 소켓버퍼의 할당 전에 동작하기 때문에 고속으로 패킷을 처리할 수 있다.

2.4 Existing Solutions and Their Limitations

2.4.1 Existing Security Solutions

컨테이너 간의 네트워크 공격을 방지하기 위해 Cilium[6], Calico[17], BASTION[2], Istio[7], Linkerd[18] 등 많은 다양한 솔루션이 제시되었다. 이러한 솔루션들은 관리자가 지정한 네트워크 보안 정책에 따라 컨테이너가 전송하는 패킷을 필터링하여 컨테이너 사이의 비 인가된 네트워크 접근을 차단해 컨테이너를 네트워크 공격으로부터 보호한다. 예를 들어 관리자는 네트워크 정책을 생성하여 컨테이너 A가 컨테이너 B에게 TCP 80번 포트로 HTTP GET 요청만 보낼 수 있게 제한할 수 있다. 컨테이너 네트워크 보안 솔루션은 크게 컨테이너 네트워크 인터페이스 (Container Network Interface, CNI)와 서비스 메시 프레임워크 (service mesh framework)로 구분된다. Table 1은 각 솔루션의

타입 및 이들이 제공하는 네트워크 보안 서비스 및 그 구현 방법을 나타낸다.

Cilium, Calico, Bastion 등 CNI는 컨테이너 간 네트워킹을 제공하면서 컨테이너가 전송한 패킷을 호스트 네트워크 네임스페이스에서 검사한다. 구체적으로 이들은 호스트 네트워크 네임스페이스 내부의 가상 인터페이스 (veth)나 가상 브리지 (br0)에서 수신되는 패킷을 eBPF와 같은 커널 네트워킹 기능을 사용해 검사해 정책에 맞지 않는 패킷을 폐기한다. eBPF를 사용하면 패킷의 헤더는 커널 영역에서 고속으로 검사할 수 있지만, 패킷의 페이로드는 검사할 수 없다. 그 때문에 Calico와 Cilium은 HTTP 헤더 등 패킷의 페이로드를 검사할 때는 별도의 프록시로 패킷을 재전송 (redirect)해서 처리한다[22]. (Bastion은 L7 정책 지원 불가). Calico와 Cilium은 추가로 컨테이너의 패킷이 호스트 시스템 외부로 전송될 경우 해당 패킷을 IPSec[19]으로 암호화하는 기능을 제공한다.

Istio, Linkerd 와 같은 서비스 메시 프레임워크는 CNI와 함께 사용되며 컨테이너를 위한 다양한 보안 기능을 제공한다. 이들은 각 컨테이너의 네트워크 네임스페이스에 side car라고 불리는 보안 프록시를 설치해 해당 컨테이너로 송/수신되는 모든 패킷을 필터링한다. 즉 컨테이너로 전송된 패킷은 먼저 side car 프록시로 전달되고 네트워크 보안 정책에 따라 필터링된다. 이들은 mTLS[20]를 통해 컨테이너의 모든 TCP 패킷을 암호화하는 기능도 제공한다. 이 경우 side car 프록시에서 패킷의 암/복호화가 수행된다. 이러한 컨테이너 패킷 암호화는 호스트 내부의 모든 컨테이너가 전송하는 패킷을 프록시를 사용해 실시간으로 암/복호화해야 함으로 네트워크 성능을 크게 떨어뜨린다. 따라서 컨테이너 패킷 암호화는 동일 호스트 내부의 컨테이너 간 통신 (intra-host)에는 적용하지 않고 서로 다른 호스트 간의 컨테이너 통신 (inter-host)에만 적용하는 것이 일반적이다[19, 21].

Table 1. Previous work on a container network security.

	Calico/ Cilium	Bastion	Istio/ Linkerd
Type	CNI	CNI	Service mesh
L3/4 policy	eBPF	eBPF	Proxy
L7 policy	Proxy	N/A	Proxy
Encryption	IPSec	N/A	mTLS

2.4.2 Limitations of Existing Solutions

컨테이너는 주로 HTTP REST API와 같은 L7 프로토콜을 사용해 통신한다. 따라서 컨테이너 사이의 인가되지 않은 네트워크 접근을 차단하기 위해서는 패킷의 헤더뿐 아니라 패킷의 페이로드까지 검사해야 한다. 이러한 네트워크 접근제어는 컨테이너를 통해 운영되는 서비스의 운영을 침해하지 않도록 고속으로 수행되어야 한다. 하지만 기존 솔루션들은 컨테이너가 전송한 패킷의 헤더는 eBPF로 구현한 패킷 필터링 프로그램을 통해 고속으로 검사하지만, 패킷의 페이로드는 별도의 프록시로 패킷을 재전송하여 처리한다.[22] 이는 커널 영역에서 실행되어야 하는 eBPF 프로그램에선 복잡한 자료구조나 라이브러리 함수를 사용할 수 없고 프로그램의 길이도 제한되어, 패킷 페이로드 검사에 활용[23]되는 Aho-Corasick 등의 다중 패턴 매칭 알고리즘을 구현할 수 없기 때문이다.

이러한 프록시 기반 페이로드 검사는 검사가 필요한 모든 패킷을 커널 영역에서 유저 영역에서 동작 중인 프록시로 재전송해 처리한 뒤 패킷을 다시 목적지로 전송해야 한다. 즉 컨테이너가 전송한 패킷의 페이로드를 검사하는 과정에서 총 두 번의 패킷 재전송 (redirection)이 발생한다. 이러한 패킷 재전송은 컨테이너가 패킷을 전송할 때마다 발생하기 때문에 네트워크 처리량 및 통신 지연을 급격하게 떨어뜨린다. 이는 곧 컨테이너를 통해 운영되는 클라우드 서비스의 품질 저하로 이어지게 되며 자율주행차, 산업용 IoT 등 네트워크 성능에 민감한 서비스에서 심각한 문제를 초래할 수 있다.

III. System Design

3.1 Threat Model and Scope

3.1.1 Threat Model

본 논문에서는 컨테이너 플랫폼, *BPFast* 등의 보안솔루션 및 이들을 실행하는 호스트 시스템(운영체제)는 신뢰할 수 있다고 가정한다. 이와 반대로, 호스트 시스템 내에서 실행되는 컨테이너는 신뢰하지 않는다. 즉, 이미 많은 선행 연구 [2, 4, 16, 31, 32]에서 증명된 것과 같이, 공격자가 알려진 취약점을 악용해 호스트에서 실행 중인 컨테이너를 탈취할 수 있다고 가정한다. 또한 공격자는 탈취한 컨테이너를 악용해 인가되지 않은 다른 컨테이너에 네트워크를 통해 악의적으로 접근할 수 있다.

3.1.2 Scope

컨테이너 간 네트워크 접근 제어는 컨테이너 사이의 비 인가된 네트워크 접근을 차단하여 다양한 네트워크 공격을 예방한다. 본 논문에서 제안하는 *BPFast*의 설계 목표는 이전 연구와 같이 새로운 컨테이너 간 네트워크 접근 제어 기법을 개발하는 것이 아닌, 기존 연구의 단점인 느린 패킷 페이로드 검사 속도를 개선하는 것이다. 즉 본 논문의 범위는 컨테이너가 네트워크 정책을 위반하는 비 인가된 네트워크 접근 (특히 L7 정책)을 할 수 없도록 막는 것이다. 또한 컨테이너 이스케이프 (escape)[24] 등 호스트 시스템 내의 커널 및 응용 프로그램 취약점을 이용한 시스템 기반 공격은 본 논문의 범위가 아니다.

3.2 Overview

*BPFast*는 eBPF 프로그램을 사용해 컨테이너가 전송한 패킷의 헤더와 페이로드를 고속으로 검사하는 시스템이다. 기존 솔루션과 달리 패킷 검사 과정에서 별도의 프록시 서버를 사용하지 않기 때문에 네트워크 성능 저하 없이 컨테이너를 다양한 네트워크 공격으로부터 보호할 수 있다.

*BPFast*는 Fig. 2에 나와 있는 것처럼 (1) *BPFast* 매니저 (manager)와 (2) 컨테이너 별로

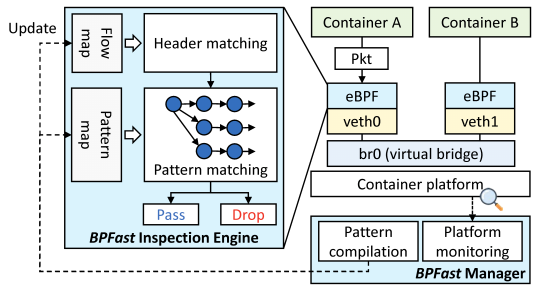


Fig. 2. An overall architecture of BPFast; This system consists of inspection engines per container (eBPF programs) and the manager.

할당된 *BPFast* 검사 엔진 (inspection engine)으로 구성된다. 매니저는 유저 영역에서 동작하는 프로세스이다. 매니저는 관리자가 입력한 네트워크 정책을 분석해 패킷 페이로드에서 검사할 문자열 패턴을 추출한 뒤 eBPF 프로그램이 처리할 수 있는 형태로 변환한다. 이후 변환된 패턴을 *BPFast* 검사 엔진 내부의 eBPF map (플로우 맵, 패턴 맵)에 저장한다. *BPFast* 검사 엔진은 eBPF 프로그램으로 각 컨테이너의 veth 인터페이스의 XDP 혹에 등록되어 커널 영역에서 실행된다. 검사 엔진은 패턴 컴파일러가 업데이트한 eBPF 맵의 내용에 따라 고속으로 컨테이너가 전송한 패킷의 헤더와 페이로드를 검사하고 이를 통해 정책을 위반하는 패킷을 폐기해 컨테이너 사이의 불법적인 네트워크 접근을 제한하여 컨테이너를 네트워크 공격으로부터 보호한다.

3.3 *BPFast* Manager

BPFast 매니저는 유저 영역에서 동작하는 프로세스로 다음과 같은 두 가지 역할을 수행한다: i) 패턴 컴파일레이션, ii) 컨테이너 플랫폼 모니터링.

3.3.1 Pattern Compilation

기존 보안 솔루션은 eBPF 프로그램을 사용해 컨테이너가 전송한 패킷의 헤더는 고속으로 검사할 수 있지만, 패킷의 페이로드는 검사할 수 없다. 이는 커널 영역에서 동작하는 eBPF 프로그램은 프로그램 길이가 제한되고, 가변 길이 반복문 및 라이브러리 함수 사용이 불가하여 패킷 페이로드를 효과적으로

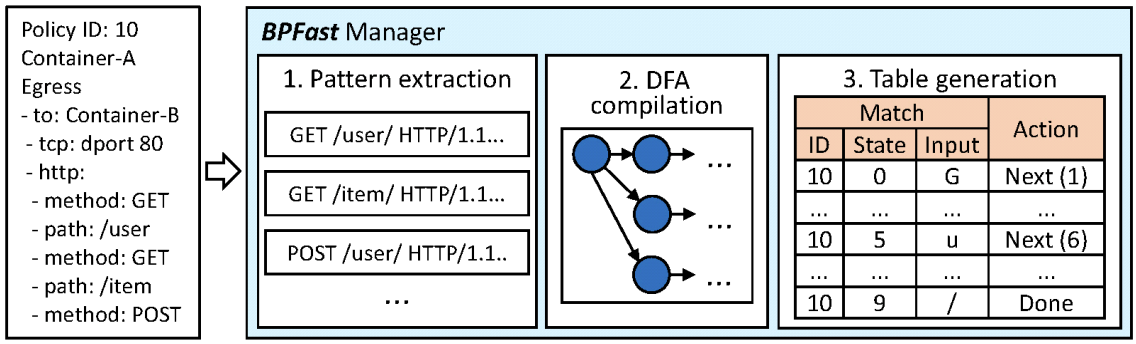


Fig. 3. The pattern compilation process of *BPFast* manager. The manager compiles a deterministic finite-state automaton (DFA) of detection patterns, taking account into common prefixes, and converts the DFA's transition table into a match-action table.

검사하기 위한 다중 패턴 매칭 알고리즘을 구현할 수 없기 때문이다 (섹션 2 참조). 본 논문에서는 이 문제를 해결하고 커널 수준의 페이로드 검사를 구현하기 위한 eBPF 맵을 활용한 오토마톤 기반의 다중 패턴 매칭 기법을 제안한다.

Fig. 3.은 *BPFast* 매니저의 패턴 컴파일 과정을 나타낸다. 네트워크 정책은 일반적으로 패킷 헤더와 관련된 L3-4 제약사항 (IP 주소, TCP 포트 번호 등)과 페이로드와 관련된 L7 제약사항 (HTTP 메소드, URL 등)으로 구성된다. 네트워크 관리자가 특정 컨테이너에 대한 네트워크 정책을 입력하면, *BPFast* 매니저는 1) 정책의 L7 제약사항에서 패킷 페이로드에서 검사할 문자열 패턴들을 추출한다. 문자열 패턴에는 정규표현식이 포함될 수 있다. 그 뒤에 매니저는 추출한 문자열들은 정해진 L7 프로토콜 (HTTP, DNS 등) 형식에 맞게 조립해 탐지 패턴들을 생성한다.

다음으로 매니저는 2) 생성한 패턴들의 공통된 접두사 (prefix)를 고려하여 해당 패턴들을 효율적으로 탐지할 수 있는 비결정적 유한 오토마톤을 생성한다. 오토마톤 생성에는 Aho-Corasick 알고리즘 등이 사용[23]될 수 있다. 이후 생성한 비결정적 유한 오토마톤을 이와 동등한 결정적 유한 오토마톤으로 변환한다. 그 뒤 매니저는 3) 생성된 오토마톤의 상태 전이표를 일반적인 네트워크 패킷 처리에 사용되는 match-action 표[25] 형태로 변환한 뒤 정책에 지정된 컨테이너 (Fig. 3.의 경우 컨테이너 A)의 *BPFast* 검사 엔진 내 패턴 맵에 저장한다. 관리자는 컨테이너마다 다수의 정책을 입력할 수 있어서, 매니저는 match-action 표를 검사 엔진에 저장할 때 정책의 ID (Fig. 3.의 경우 10) 값을 함께 저장한다. *BPFast* 매니저는 정책의 L3-4 제약사항은 곧바로 match-action table 형태로 변환하여 *BPFast* 검사 엔진의 플로우 맵에 저장한다.

BPFast 매니저가 유저 영역에서 미리 페이로드에서 탐지할 패턴들의 prefix를 고려하여 match-action 표 형태로 변환했기 때문에 *BPFast* 검사 엔진은 복잡한 자료구조나 알고리즘의 구현 없이 패킷의 페이로드를 검사할 수 있다. 구체적으로 검사 엔진은 고정된 길이 (탐지 가능한 최대 패턴 길이)의 반복문을 실행하며 페이로드의 각 바이트와 현재 상태를 키 (key)로 패턴 맵을 조회하며 상태를 전이하는 것으로 커널 영역에서 복수의 패턴을 빠르게 탐지할 수 있다.

3.3.2 Container Platform Monitoring

컨테이너를 네트워크 공격에서 보호하기 위해서 *BPFast* 매니저는 새로운 컨테이너의 생성을 감지하여 해당 컨테이너의 호스트 측 가상 인터페이스 (veth)의 XDP 훅에 *BPFast* 검사 엔진을 등록해야 한다. 또한 네트워크 정책에 변화가 생길 경우 이를 즉각 탐지하고 변경된 정책을 페이로드 엔진이 사용하는 eBPF 맵에 반영해야 한다.

이를 위해 *BPFast* 매니저는 Docker[12], Kubernetes[26] 등 컨테이너 플랫폼을 지속해서 모니터링해 컨테이너의 생성 및 삭제, 네트워크 정책변화를 감지한다. 구체적으로 매니저는 컨테이너

플랫폼이 제공하는 API[27]를 사용해 event-driven 방식으로 새로운 컨테이너의 생성을 감지하여 해당 컨테이너의 veth 인터페이스에 *BPFast* 검사 엔진을 등록한다. 마찬가지로, 특정 컨테이너에 대한 네트워크 정책에 변화가 생길 경우, 해당 컨테이너에 대한 패턴 컴파일 과정을 다시 진행하여 검사 엔진의 eBPF 맵들 (플로우 맵, 패턴 맵)의 내용을 항상 최신 상태로 유지한다.

3.4 *BPFast* Inspection Engine

BPFast 검사 엔진은 eBPF 프로그램으로 Fig. 2에 나와 있는 것처럼 각 컨테이너의 호스트 측 인터페이스 (veth)의 XDP 훅에 등록된다. 특정 컨테이너에 등록된 검사 엔진은 해당 컨테이너가 전송한 패킷이 호스트 측 인터페이스에 수신될 때마다 실행되어 네트워크 정책에 따라 패킷의 헤더와 페이로드를 검사한다.

BPFast 검사 엔진은 네트워크 정책이 저장된 두 개의 eBPF 맵 (플로우 맵, 패턴 맵)을 기반으로 동작한다. 컨테이너마다 별도의 *BPFast* 검사 엔진이 할당되며, 각 검사 엔진의 eBPF 맵 역시 컨테이너별로 독립되어 관리된다. 이러한 분산된 구조는 각 검사 엔진 인스턴스가 유지해야 하는 네트워크 정책의 수를 줄여주기 때문에 검사 엔진이 빠르게 패킷을 검사할 수 있도록 해준다. 예를 들어 컨테이너 A에 할당된 검사 엔진은 컨테이너 A의 패킷이 veth에 수신될 때만 실행되며, 검사 엔진의 eBPF 맵에는 컨테이너 A에 대한 정책만 저장된다. *BPFast* 검사 엔진은 다음과 같은 두 단계를 통해 등록된 컨테이너가 전송한 패킷을 검사한다.

3.4.1 Header Matching

헤더 매칭은 수신한 패킷이 네트워크 정책의 L3/4 제약사항을 위반하는지 검사하는 과정이다. 이를 통해 *BPFast*는 두 컨테이너 간의 비 인가된 (정책을 위반하는) L3/4 통신을 차단한다. *BPFast* 검사 엔진은 패킷의 헤더를 분석하여 플로우 맵의 내용과 비교한다. 검사 엔진이 컨테이너별로 할당되기 때문에 검사 엔진의 플로우 맵에는 해당 컨테이너와 통신이 허용된 다른 컨테이너의 IP 주소를 기준으로 TCP 포트 번호 등의 정책의 L3/4 제약사항이 저장되어 있다. 따라서 검사 엔진은 먼저 수신한 패킷의 목적지 IP 주소를 키로 플로우 맵을 조회하여 해당 패킷에 대한 L3/4 제약사항을 찾는다.

이후 검사 엔진은 찾아온 제약사항과 패킷 헤더의 필드를 하나씩 비교한다. 검사 엔진은 기본적으로 allow-list로 동작하기 때문에 가져온 L3/4 제약사항과 패킷 헤더의 내용이 일치하지 않을 경우 해당 패킷을 폐기한다. 또한 플로우 맵 조회가 실패한 경우에도 패킷을 폐기한다. 검사 엔진의 동작은 설정을 통해 deny-list (정책과 일치하는 패킷 폐기)로 변경할 수 있다.

3.4.2 Pattern Matching

패턴 매칭은 헤더 매칭을 통과한 패킷이 네트워크 정책의 L7 제약사항을 위반하는지 검사하는 과정이다. 이를 통해 *BPFast*는 두 컨테이너간의 비 인가된 L7 통신을 차단한다. 수신한 패킷이 헤더 매칭 과정을 통과하게 되면, *BPFast* 검사 엔진은 패킷의 페이로드를 검사한다. 검사 엔진은 먼저 패킷의 헤더 정보 (TCP 헤더 길이 필드 등)를 분석해 패킷에서 페이로드 부분만 따로 분리한다. 이후 엔진은 내부의 상태 머신 (state machine)을 사용해 페이로드에서 패턴 맵 내의 패턴들을 등장하는지 검사한다.

상태 머신은 3개의 레지스터 (위치, 입력, 상태 레지스터)와 5개의 상태 (READ, LOOKUP, ACTION, MATCH, MISS)로 구성된다. 상태 머신은 READ 상태에서 시작한다. READ 상태에서 상태 머신은 페이로드에서 위치 레지스터가 가리키는 곳의 1byte를 읽어 입력 레지스터에 저장한 뒤 LOOKUP 상태로 전이한다. 이때 위치 레지스터의 값이 미리 지정된 최대 길이 또는 패킷의 끝 (EOP)에 도달한 경우 MISS 상태로 전이한다. LOOKUP 상태에서는 입력 레지스터의 값과 상태 레지스터의 값을 키로 패턴 맵을 조회하여 현재 상태에 대한 액션을 가져온 뒤 ACTION 상태로 전이한다. 이때 조회가 실패할 경우 MISS 상태로 전이한다. ACTION 상태의 상태 머신은 패턴 맵에서 가져온 액션을 수행한다. 액션이 'Next (n)'일 경우 상태 레지스터의 값을 n으로 갱신한 뒤 READ 상태로 전이한다. 액션이 'Done'일 경우 MATCH 상태로 전이한다.

MATCH 상태는 상태 머신이 페이로드에서 패턴

맵 내의 특정 패턴을 찾았음을 의미하며 반대로 MISS 상태는 상태 머신이 패턴을 찾지 못했음을 의미한다. 상태 머신이 MATCH나 MISS 상태에 도달하면 *BPFast* 검사 엔진은 정책에 따라 패킷을 폐기 (drop)하거나 목적지로 전송한다 (pass).

이러한 상태 머신 기반의 페이로드 검사는 *BPFast*가 미리 컴파일한 패턴의 상태 전이표에 따라 단순히 고정된 길이의 반복문 (최대 반복 회수 = 최대 패턴 길이)을 돌면서 효율적으로 패킷의 페이로드에서 여러 패턴을 한 번에 검사할 수 있다. 또한 복잡한 자료구조나 별도의 함수 호출을 요구하지 않기 때문에 eBPF 프로그램 형태로 커널 영역에서 고속으로 동작할 수 있다.

*BPFast*의 검사 엔진은 Fig. 2에 나와있는 것처럼 독립적인 eBPF 프로그램이다. 따라서 각 컨테이너에 할당된 veth 인터페이스에 부착되어 단독 보안 솔루션으로 동작할 수 있을 뿐 아니라 다른 보안 솔루션이 컨테이너의 veth 인터페이스에 등록한eBPF 프로그램의 실행 전/후에 동작하도록 실행 순서를 조절할 수도 있다. 즉 *BPFast*는 독립적인 솔루션뿐 아니라 다른 솔루션의 페이로드 검사용 add-on으로 활용할 수 있다.

IV. Implementation

본 연구에서는 eBPF 프로그램과 python 프로그램을 통해 *BPFast*의 프로토타입을 구현했다. *BPFast*의 검사 엔진은 약 1,000라인의 eBPF 코드로 구현했으며 XDP 훅에 등록할 수 있도록 구현했다. 검사 엔진은 내부적으로 헤더 파싱 함수와 헤더 매칭 함수, 패턴 매칭 함수로 나눠서 구현했고, 패턴 매칭 함수는 미리 지정된 최대 길이만큼 3장에서 설명한 상태 머신을 실행하는 반복문으로 구현했다. 검사 엔진이 사용하는 플로우 맵과 패턴 맵은 모두 탐색 효율성을 고려해 O(1) 탐색을 보장하는 eBPF array map[9]으로 구현했다.

BPFast 매니저는 약 2,000라인의 python 프로그램으로 구현했으며 Kubernetes watcher API[27]를 통해 컨테이너 플랫폼의 변화를 실시간으로 탐지할 수 있도록 했다. 추가로 *BPFast* 매니저는 사용자로부터 YAML 또는 JSON 형식의 Kubernetes 네트워크 정책 파일을 입력받을 수 있도록 구현했다.

V. Evaluation

본 연구에서는 Kubernetes[26]로 구성한 실제 컨테이너 환경에서 *BPFast*의 페이로드 검사 성능을 측정한 뒤 Cilium, Istio등 최신 솔루션들과 비교하여 *BPFast*의 효율성을 입증했다.

5.1 Evaluation Environments

성능평가는 Intel Xeon E5-2630v4 CPU와 64GB RAM, Ubuntu 18.04 OS를 사용하는 호스트 머신에 Kubernetes 컨테이너 플랫폼을 설치하고 Cilium을 CNI로 설정한 환경에서 진행했다. 해당 환경에서 아파치 서버 컨테이너[28]와 클라이언트 컨테이너[29]를 각각 실행하고, 두 컨테이너간의 HTTP 통신을 미리 지정한 URL에 대해서만 허용하는 네트워크 정책을 Cilium (CNI), Istio (서비스 메시), *BPFast*로 각각 적용했다. 정책에 사용한 URL은 길이 10 ~ 50의 랜덤 스트링으로 생성했으며, 정책 내 URL의 수를 250에서 1,000개까지 증가시키면서 실험을 반복했다. 실험에 사용한 정책은 정규표현식 패턴을 포함하고 있지 않다. 하지만 *BPFast*와 기존 솔루션의 차이는 페이로드 검사 과정에서 패킷 재전송의 발생 유무이기 때문에 정규표현식 패턴의 유무가 실험에 큰 영향을 주지 않는지는 않는다.

HTTP 처리량 및 통신 지연 측정은 클라이언트 컨테이너에서 HTTP 성능 측정 도구인 WRK[30]를 사용해서 진행했다. 구체적으로 클라이언트 컨테이너가 12개의 스레드를 사용해 400개의 커넥션으로 서버 컨테이너에 HTTP 요청을 보내도록 설정했다.

5.2 HTTP Throughput

Fig. 4.에 나타나 있는 것처럼 L7 정책을 적용하기 전의 서버와 클라이언트 컨테이너 사이의 HTTP 처리량은 48.7Krps (requests per second)이었지만, L7 정책을 적용했을 때 솔루션의 종류와 관계없이 처리량 저하가 발생했다. 이때 세 솔루션 모두 다중 패턴 매칭 기법을 사용하기 때문에 패턴의 수와 관계없이 검사 성능은 일정했다.

1,000개의 서로 다른 패턴을 적용했을 때, 유저

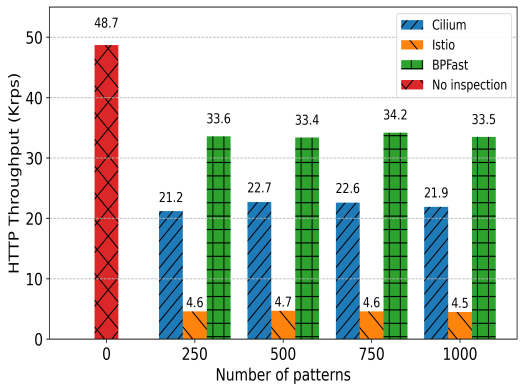


Fig. 4. HTTP throughput measurements; *BPFast* outperforms state-of-the-art solutions by up to 7x.

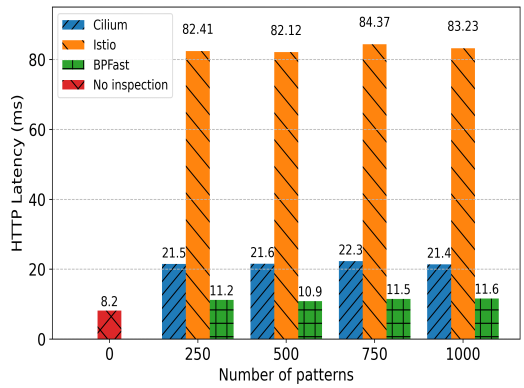


Fig. 5. HTTP latency measurements; *BPFast* achieves 7x faster HTTP latency compared to state-of-the-art solutions.

영역에서 동작하는 프록시로 패킷을 재전송해 페이로드를 검사하는 Cilium은 HTTP 처리량이 21.9Krps로 약 51% 정도 크게 감소했으며, mTLS 암호화까지 적용하는 Istio는 4.5Krps로 처리량이 1/10 수준으로 감소했다. 반면 패킷 페이로드 검사의 전 과정을 커널 영역에서 수행하는 *BPFast*는 같은 조건에서 33.5Krps의 HTTP 처리량을 보였다. 이는 Cilium보다 1.5배 Istio보다 7배 이상 높은 성능이다.

5.3 HTTP Latency

Fig. 5.는 페이로드 검사 패턴의 수에 따른 각 솔루션의 HTTP 통신 지연을 나타낸다. 페이로드 검사를 활성화하기 전 시스템의 HTTP 통신 지연은 8.2ms였지만, HTTP 처리량 실험과 마찬가지로 검사를 활성화했을 때는 모든 솔루션에서 HTTP 통신 지연 증가가 발생했다.

1,000개의 패턴을 적용했을 때 Cilium, Istio의 경우 각각 21.4ms, 83.23ms의 통신 지연을 보였다. 이는 페이로드 검사를 활성화하기 전에 비해 각각 2.6배, 10배 이상 증가한 수치이다. 이러한 결과는 패킷을 커널 영역에서 유저 영역으로 재전송하는 과정에서 패킷 복사 등으로 인해 매우 큰 성능 저하가 발생하기 때문이다. Istio의 경우 mTLS 암/복호화까지 수행해야 해서 통신 지연이 더욱 증가했다. 반면 커널 영역에서만 동작하는 *BPFast*는 같은 조건에서 11.6ms로 Cilium보다 2배, Istio보다 약 7배 이상 빠르게 동작했다.

VI. Related Work

6.1 Container Security

클라우드 환경에서 컨테이너 기술이 활발히 사용되면서 컨테이너 기술에 대한 전반적인 보안 이슈를 식별하기 위해 다양한 선행 연구들이 진행되었다. Sultan[31] 등은 컨테이너 환경에서 발생 가능한 다양한 보안 이슈를 자세히 분석하였고, 이를 해결하기 위해서 고려해야 할 사항들을 학술적인 관점에서 제시했다. Martin[32] 등은 보안 취약점 분석 관점에서 Docker 컨테이너 플랫폼을 상세히 분석했고, 다양한 취약점을 식별했고 이를 이용한 공격 시나리오를 제시했다.

6.2 Container Hardening

컨테이너 하드닝 (hardening)은 시스템 콜[33], 리눅스 캐퍼빌리티 (capability)[34] 등 컨테이너의 공격 표면 (attack surface)을 줄여 공격자로부터 컨테이너를 보호하는 기술이다. 이는 공격자에게 탈취된 컨테이너의 악의적인 행동을 제한하는데 필수적인 기술이며, 최근에는 리눅스 커널의 보안 기능을 활용해 컨테이너 하드닝을 수행하는 연구가 활발히 진행되고 있다.

SPEAKER[35]는 동적 분석 기술을 사용해

컨테이너의 시스템 콜 제한 정책 생성 과정을 간소화하고 각 컨테이너에 불필요하게 할당된 시스템 콜을 제거하여 컨테이너의 공격 표면을 줄인다. 이와 반대로 Confine[36]은 정적 분석 기술을 통해 컨테이너 실행에 필요한 시스템 콜 목록을 추출하고, 이를 바탕으로 불필요하게 할당된 시스템 콜의 호출을 제한할 수 있는 정책을 자동으로 생성해준다. 이러한 연구는 컨테이너의 시스템적 보안을 목표로 하고 있어 본 논문에서 목표로 하는 컨테이너 네트워크 보안과는 서로 다른 방향성을 가지고 있다.

6.3 Container Network Security

컨테이너들이 네트워크를 통해 연결되어 클라우드 서비스를 구성하기 때문에, 컨테이너 환경에서 발생하는 다양한 네트워크 보안 이슈를 식별하고 방지하기 위해 다양한 연구들이 진행되었다. Budigiri[37] 등은 컨테이너들의 네트워크 접근을 제한하는 컨테이너 네트워크 정책에 대해서 상세히 분석했고, 현재의 네트워크 정책 모델이 가지고 있는 취약점에 대해 제시했다. Nam[2] 등은 컨테이너 네트워킹 구조에 대해 상세히 분석했고, 공격자가 탈취한 컨테이너를 통해 인접한 다른 컨테이너를 공격할 수 있음을 실험을 통해 재현했다.

이러한 네트워크 위협으로부터 컨테이너를 보호하기 위해 다양한 보안 솔루션이 제시되었다. Cilium, Calico, Bastion 등의 CNI는 컨테이너 간 네트워킹을 제공하면서 eBPF를 사용해 고속으로 네트워크 정책을 집행하여 컨테이너를 네트워크 공격으로부터 보호한다. Istio, Linkerd 등 서비스 메시 프레임워크는 CNI 위에서 동작하면서 mTLS 암호화 등 다양한 보안 기능을 추가적으로 제공한다. 이러한 솔루션들은 모두 프록시를 사용해 L7 네트워크 정책을 처리하기 때문에 매우 큰 성능 저하를 발생시킨다. 반면 *BPFast*는 별도의 유저 수준 보안 프록시를 사용하지 않고 커널 영역에서 패킷의 페이로드를 검사하여 기존 솔루션의 단점인 느린 페이로드 검사 속도를 향상할 수 있도록 설계되었다.

VII. Conclusion

*BPFast*는 eBPF/XDP를 사용해 커널 수준에서 기존 솔루션을 상회하는 성능으로 컨테이너의 패킷 페이로드를 검사할 수 있다. 이러한 이점으로, *BPFast*는 AR/VR, 자율주행차 등 네트워크 성능에 민감한 서비스를 위해 실행되는 컨테이너의 보안성을 강화하는데 사용할 수 있다. 하지만 *BPFast*는 향후 개선할 여지가 아직 남아있다. 대표적으로 eBPF 프로그램은 명령어의 개수가 최대 4000개 제한되는데[6] 이로 인해 *BPFast*는 검사할 수 있는 패턴의 최대 길이 (최대 500 글자)가 제한되며 복잡한 정규표현식을 처리할 수 없다. 이는 페이로드 검사엔진이 반복문을 통해 오토마톤을 실행하는 형태로 패턴을 검사하는데, 긴 길이의 패턴이나 복잡한 정규 표현식의 경우 오토마톤의 길이가 길어지기 때문에 발생하는 문제이다. 하지만 이 문제가 *BPFast*의 효율성을 크게 떨어뜨리지는 않는다. *BPFast*는 페이로드에서 특정 HTTP 호스트 이름, SQL 삽입 등에 사용되는 문자열 등을 탐지할 때 사용하는 간단한 정규표현식은 모두 처리할 수 있다.

이러한 문제는 하나의 eBPF 프로그램에서 다른 eBPF 프로그램을 연쇄적으로 호출하여 실행하는 eBPF tail-call[9]을 사용하여 해결할 수 있다. 예를 들어, *BPFast* 검사 엔진을 기능 별로 여러 eBPF 프로그램으로 나눠서 구현한 뒤에 (패킷 헤더 파싱 모듈, 헤더 매칭 모듈, 페이로드 매칭 모듈 등) 이 프로그램들을 tail-call을 통해 서로 호출하는 형태로 구현한다면 명령어의 개수에 크게 제약 받지 않게 된다. 이를 통해 *BPFast* 검사 엔진은 이전보다 긴 길이의 반복문을 사용할 수 있어 더 긴 길이 패턴 및 정규표현식을 처리할 수 있다.

References

[1] A. Randal, "The ideal versus the real: Revisiting the history of virtual machines and containers," ACM Computing Surveys (CSUR), vol. 53, no. 2, pp. 1-31, Feb. 2020.

[2] J. Nam, S. Lee, H. Seo, P. Porras, V. Yegneswaran, and S. Shin, "BASTION: A Security Enforcement Network Stack for Container Networks," In Proceedings of the Annual Technical Conference. USENIX Association (ATC), pp.

81-95, Jul. 2020.
[3] F. Minna, A. Blaise, F. Rebecchi, B. Chandrasekaran, and F. Massacci, "Understanding the security implications of kubernetes networking," IEEE Security & Privacy, vol. 19, pp. 46-56, May. 2021.
[4] G. Perrone and S. P. Romano, "The docker security playground: A hands-on approach to the study of network security," In Proceedings of Principles, Systems and Applications of IP Telecommunications (IPTComm), pp. 1-8, Sep. 2021.
[5] Tripwire, "Tripwire State of Container Security Report," https://www.tripwire.com/solutions /devops/tripwire-dimensional-research-state-of-container-security-report-register, Jan. 2019.
[6] Cilium, "Cilium: security-enhanced CNI," https://cilium.io/, Feb. 2022.
[7] Istio "The Istio Service Mesh," https://istio.io/, Feb. 2022.
[8] L. Li, T. Tang and W. Chou, "A REST Service Framework for Fine-Grained Resource Management in Container-Based Cloud," In Proceedings of 2015 IEEE 8th International Conference on Cloud Computing, pp. 645-652, Jun. 2015.
[9] eBPF, "eBPF - Introduction, Tutorials & Community Resources," https://ebpf.io/, Mar. 2022.
[10] Cilium, "eBPF cGuide," https://docs.cilium.io/en/latest/bpf/, Feb. 2022.
[11] H. Kang, M. Le, and S. Tao, "Container and microservice driven design for cloud infrastructure devops," In Proceedings of 2016 IEEE International Conference on Cloud Engineering (IC2E), pp. 202-211, Apr. 2016.
[12] Docker, "Docker: Empowering App Development for Developers." https://www.docker.com/, Mar. 2021.
[13] OpenVz, "Open source container-based virtualization for Linux," https://openvz.org/, Feb. 2022.
[14] Michael Kerrisk, "ip-netns - process network namespace management," https://man7.org/linux/man-pages/man8/ip-netns.8.html, Feb. 2022.
[15] Docker, "Use bridge networks | Docker Documentation," https://docs.docker.com/network/bridge/, Feb. 2022.
[16] X. Li, Y. Chen, Z. Lin, X. Wang, and J. H. Chen, "Automatic Policy Generation for Inter-Service Access Control of Microservices," In Proceedings of 30th USENIX Security Symposium, pp. 3971-3988, Aug. 2021.
[17] Tigera, "Protect Calico - Tigera." https://www.tigera.io/project-calico/, Feb. 2022.
[18] Linkerd, "The world's lightest, fastest service mesh," https://linkerd.io/, Feb. 2022.
[19] Cilium, "IPsec Transparent Encryption,"https://docs.cilium.io/en/v1.10/gettingstarted/encryption-ipsec/, Feb. 2022.
[20] Istio, "Mutual TLS Migration," https://istio.io/latest/docs/tasks/security/authentication/mtls-migration/, Feb. 2022.
[21] Cilium, "How Cilium enhances Istio with socket-aware BPF programs," https://cilium.io/blog/2018/08/07/istio-10-cilium, Feb. 2022.
[22] Cilium, "Envoy with Cilium filter," https://github.com/cilium/proxy, Feb. 2022.
[23] R. T. El-Maghraby, N. M. Abd Elazim and A. M. Bahaa-Eldin, "A survey on deep packet inspection," In Proceedings of 2017 12th International Conference on Computer Engineering and Systems (ICCES), pp. 188-197,

Feb. 2017.

[24] Z. Jian, and L. Chen, "A defense method against docker escape attack," In Proceedings of the 2017 International Conference on Cryptography, Security and Privacy (ICCSP), pp. 142-146, Mar. 2017.

[25] P. Bosshart, G. Gibb, H. Kim, G. Varghese, N. McKeown, M. Izzard, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," ACM SIGCOMM Computer Communication Review, vol. 43, no. 4, pp. 99-110, Aug. 2013.

[26] Kubernetes, "Production-Grade Container Orchestration," https://kubernetes.io/, Feb. 2022.

[27] Kubernetes, 'Kubernetes API Concepts," https://kubernetes.io/docs/reference/using-api/api-concepts/, Feb. 2022.

[28] DockerHub, "Httpd - Official Image | Docker Hub," https://hub.docker.com/_/httpd, Feb. 2022.

[29] DockerHub, "Ubuntu - Official Image | Docker Hub," https://hub.docker.com/_/ubuntu, Feb. 2022.

[30] Will Glozer, "WRK - a HTTP benchmarking tool," https://github.com/wg/wrk, Feb. 2022.

[31] S. Sultan, I. Ahmad, and T. Dimitriou, "Container Security: Issues, Challenges, and the Road Ahead," IEEE Access, vol. 7, pp. 52976 - 52996, Apr. 2019.

[32] A. Martin, S. Raponi, T. Combe, and R. Di Pietro, "Docker Ecosystem - Vulnerability Analysis," Computer Communications, vol. 122, pp.30 - 43, Jun. 2018.

[33] Docker, "Seccomp security profiles for Docker container," https://docs.docker.com/engine/security/seccomp/, Feb. 2022.

[34] Kubernetes, "Configure a Security Context for a Pod or Container," https://kubernetes.io/docs/tasks/configure-pod-container/security-context/, Feb. 2022.

[35] L. Lei, J. Sun, K. Sun, C. Shenefiel, R. Ma, Y. Wang, and Q. Li, "Speaker: Split-phase execution of application containers," In Proceedings of International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 230-251, Jun. 2017.

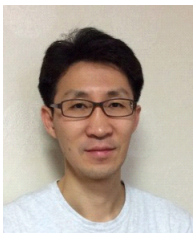
[36] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated System Call Policy Generation for Container Attack Surface Reduction," In Proceedings of International Symposium on Research in Attacks, Intrusions and Defenses (RAID), pp. 443-458, Oct. 2020.

[37] G. Budigiri, C. Baumann, J. T. M ¨ uhlberg, E. Truyen, and W. Joosen, "Network policies in kubernetes: Performance evaluation and security analysis," In proceedings of Joint European Conference on Networks and Communications & 6G Summit, pp. 407-412, Jun. 2021.

〈저자소개〉

유 명 성 (Myoung-sung You) 학생회원
2020년 2월: 충북대학교 컴퓨터공학과 졸업
2022년 2월: 한국과학기술원 정보보호대학원 석사
2022년 3월~현재: 한국과학기술원 전기및전자공학부 박사과정
〈관심분야〉 네트워크 보안, 클라우드 보안, 데이터평면 구조

김 진 우 (Jin-woo Kim) 정회원
2017년 2월: 한국과학기술원 정보보호대학원 석사
2022년 2월: 한국과학기술원 전기및전자공학부 박사
2022년 3월~현재: 광운대학교 소프트웨어융합대학 소프트웨어학부 조교수
〈관심분야〉 네트워크 보안, 클라우드 보안, SDN, 데이터평면 구조



신 승 원 (Seung-won Shin) 정회원
2000년 2월: 한국과학기술원 전기및전자공학부 석사
2013년 2월: Texas A&M University 전산학 박사
2002년~2006년: ETRI 연구원
2005년~2006년: MIT 초빙 연구원
2006년~2009년: TmaxSoft 책임 연구원
2013년~현재: 한국과학기술원 전기및전자공학부 교수
〈관심분야〉 네트워크 보안, SDN, CTI

박 태 준(Tae-june Park) 종신회원
2021년 2월: 카이스트 공학박사
2021년 3월~8월: 카이스트 연수연구원
2021년 9월~현재: 전남대학교 인공지능학부 조교수
〈관심분야〉 네트워크 보안, 데이터평면 구조