Secure Inter-Container Communications Using XDP/eBPF

Jaehyun Nam, Seungsoo Lee, Phillip Porras, Vinod Yegneswaran, and Seungwon Shin, *Member, IEEE*

***Abstract*— While the use of containerization technologies for virtual application deployment has grown at an astonishing rate, the question of the robustness of container networking has not been well scrutinized from a security perspective, even though inter-container networking is indispensable for microservices. Thus, this paper first analyzes container networks from a security perspective, discussing the implications based on their architectural limitations. Then, it presents Bastion$^+$, a secure inter-container communication bridge. Bastion$^+$ introduces (i) a *network security enforcement stack* that provides fine-grained control per container application and securely isolates inter-container traffic in a point-to-point manner. Bastion$^+$ also supports (ii) *selective security function chaining*, enabling various security functions to be chained between containers for further security inspections (e.g., deep packet inspection) according to the container's network context. Bastion$^+$ incorporates (iii) a *security policy assistant* that helps an administrator discover inter-container networking dependencies correctly. Our evaluation demonstrates how Bastion$^+$ can effectively mitigate several adversarial attacks in container networks while improving the overall performance up to 25.4% within single-host containers and 17.7% for cross-host container communications.**

***Index Terms*— Container security, network sandboxing, policy enforcement, security function chaining, XDP/eBPF.**

I. Introduction

AMONG the leading trends in virtualization is containerized application deployment at industrial scales across private and public cloud infrastructures. For example, Google has spawned more than two billion containers per week [1]. Yelp uses containers to migrate their code onto AWS and launches more than one million containers per day [2]. Netflix spawns more than three million containers per week within Amazon EC2 using its Titus container management platform [3].

With this growing attention toward the large-scale instantiation of containerized applications also comes a potential for even small security cracks within the container software ecosystem to produce hugely destructive impacts. Tripwire's container security report [4] found that 60% of organizations already had experiences of security incidents in 2018, assessing that these incidents arose primarily due to the pressures to achieve deployment speed over the risks from deploying insecure containers. Also, container hijacking for cryptocurrency mining [5] has emerged as one of the recent plagues in which computing resources, rather than user data, are being harvested en masse across the Internet. In recognition of such risks, several efforts [6], [7], [8] have arisen to help identify and warn of possible vulnerabilities in containers.

In addition, the shared kernel-resource model used by containers also introduces critical security concerns regarding the ability of the host OS to maintain isolation once a single container is infected. Indeed, many researchers (and industry) have proposed strategies to address the issue of container isolation. For example, AppArmor [9], Seccomp [10], and SELinux [11] can provide much stronger isolation of containers by preventing various system resource abuses.

However, while a variety of approaches to secure containerized applications continue to emerge, less attention has been paid to bounding these applications' access to the container network. Specifically, there has been significant adoption of containers as microservices [12], in which containers are used to create complex cloud services. Although current container platforms often utilize IP-based access control to restrict each container's network interactions, there are still limitations in such controls that offer opportunities for container abuse.

In this work, we first discuss several security challenges that arise from the current reliance on the host OS network stack and virtual networking features to provide robust container-network security. In the discussion, we present five examples of inherent limitations that arise in using the Host-OS-based networking features to manage the communications of container ecosystems as they are deployed today. Informed by these existing limitations, we then introduce Bastion$^+$, a new inter-container communication bridge. First, Bastion$^+$ instantiates a security network stack per container, offering isolation, performance efficiency, and a fine-grained network security policy enforcement that implements each container's least privileged network access. This approach also provides better network policy scalability in network policy management as the number of hosted containers increases. Second, Bastion$^+$ supports a security function chaining mechanism that allows an administrator to deploy various security functions on-demand and forces inter-container network traffic to selectively pass through the security functions according to

Manuscript received 16 October 2021; revised 22 March 2022 and 23 August 2022; accepted 10 September 2022; approved by IEEE/ACM Transactions on Networking Editor Y. Liu. This work was supported by the National Research Foundation of Korea (NRF) Grant through the Korea Government, Ministry of Science, ICT (Information and Communication Technology) and Future Planning (MSIP), under Grant 2022R1C1C1006093. *(Corresponding author: Seungwon Shin.)*

Jaehyun Nam is with the Department of Computer Engineering, Dankook University, Yongin 16890, South Korea (e-mail: jaehyun.nam@dankook.ac.kr).

Seungsoo Lee is with the Department of Computer Science and Engineering, Incheon National University, Incheon 22012, South Korea (e-mail: seungsoo@inu.ac.kr).

Phillip Porras and Vinod Yegneswaran are with the SRI International, Menlo Park, CA 94025 USA (e-mail: porras@csl.sri.com; vinod@csl.sri.com).

Seungwon Shin is with the School of Electrical Engineering, KAIST, Daejeon 34141, South Korea (e-mail: claude@kaist.ac.kr).

Digital Object Identifier 10.1109/TNET.2022.3206781

its network context for further security investigations. Lastly, Bastion$^+$ provides a security policy assistant to facilitate the identification of inter-container networking dependencies.

The paper explains how Bastion$^+$ mitigates a range of existing security challenges while also demonstrating that Bastion$^+$ can improve the overall performance up to 25.4% within the same host and 17.7% across hosts.

Contributions. Our paper contributions are as follows:

- The security assessment of container networks, illustrating security challenges in current container network stacks and security mechanisms.
- The novel security-enforcement network stack for containers, which restricts the network visibility of containers and isolates network traffic among peer containers.
- The security function chaining mechanism specialized for containers, enabling additional security inspections in inter-container communications through the function chains selectively chosen based on their network context.
- The security assistant for the policy enforcement between the containers, which helps administrators recognize inter-container networking dependencies.

II. Background and Motivation

Here, we introduce the state of container security. We then provide the background of container networks and identify how the underlying architectural limitations of current container networks impact container environments.

A. Today's Container Security Solutions

Containers are widely utilized to decompose complex production Internet services into manageable microservices. The need to harden containers to resist compromise and ensure their application integrity is of critical importance. Hence, various security solutions have been explored, broadly focusing on three aspects of the container ecosystem.

(1) Container Image Integrity. A container image is a self-contained package of software that includes everything needed to run an application (e.g., code, libraries, and configurations). Image management, tamper resistance, and configuration validation are foundational services upon which all other subsequent security features must rely. Here, two forms of image protection services have been released. One form is that of solutions like Docker Content Trust (DCT) [13], which verifies container images at image repositories (e.g., Docker Hub [14]) with the digital signatures of image owners. The second form is represented by security scanning solutions [6], [7], [8], which inspect known vulnerabilities in container images using CVE databases.

(2) Container Isolation. Once a container is deployed, three Host OS security mechanisms are used to implement application isolation and enforce least privilege access on the container application: *namespace*, *cgroups*, and *capabilities*. Since multiple containers share the same host kernel, AppArmor [9], Seccomp [10], and SELinux [11] are used for further restrictions on system resources (e.g., kernel calls). Also, there are several solutions (e.g., Lic-Sec [15] for AppArmor, Udica [16] for SELinux, Speaker [17], and Confine [18] for Seccomp) to automatically generate the security profiles of those Linux security mechanisms for containers based on container and application configurations.

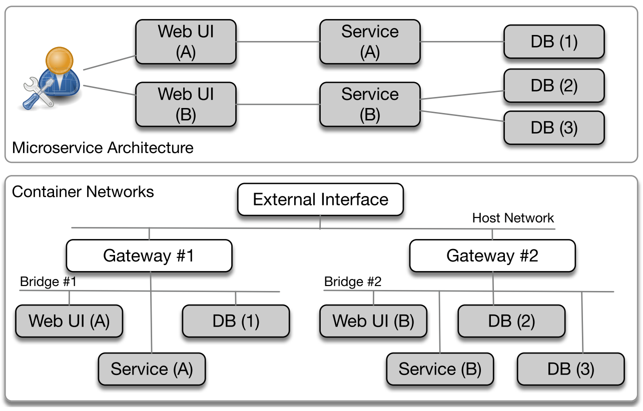


Fig. 1. Overview of docker bridge networking. Upper panel: a conceptual microservice architecture involving two independent services. Lower panel: separate bridged networks are instantiated to manage container network flows.

(3) Run-time Threat Detection. Several commercial products [19], [20], [21] have introduced container security frameworks that monitor the behavior of containers, detect runtime policy violations, and conduct anomaly detection.

This paper aims to complement the above services' protections by addressing a fourth significant aspect of container security enforcement: network security enforcement during inter-container communications. The primary service used to enforce network security policies in container networks is through ACL-based IP rules. We will discuss the limitations of these services and identify how the current underlying architectural limitations impact container environments.

B. Current Container Networks

We provide a brief overview of how current container networks are structured using two of the most prevalent container systems used today: Docker [22] and Kubernetes [23].

Docker Platform: Docker [22] is a platform for distributing and running containers. In Docker, bridge networks are used as default container networks. Independent Docker containers are, by default, connected to a bridge called `Docker0`. However, when multiple containers are created using docker-compose [24], a new bridge (network) is automatically created and assigned to manage the traffic for those containers. As an example, Figure 1 illustrates the architecture of two microservices. The microservice chains that compose a network service are shown in the upper panel, while the logical networking of the microservice containers, which are networked under separate bridges, is depicted in the lower panel. To provide network flow control, Docker applies network and security policies into bridge networks using `iptables` [25].

Kubernetes Orchestration System: Kubernetes [23] is an open-source container orchestration system that manages a large number of containers across multiple nodes and enables them to work together logically. Thus, while containers in Docker bridge networks operate within the same host (node), containers in Kubernetes are located across multiple nodes.

Kubernetes uses various overlay networks (e.g., Flannel [26], Weave [27], Calico [28]) to provide inter-container

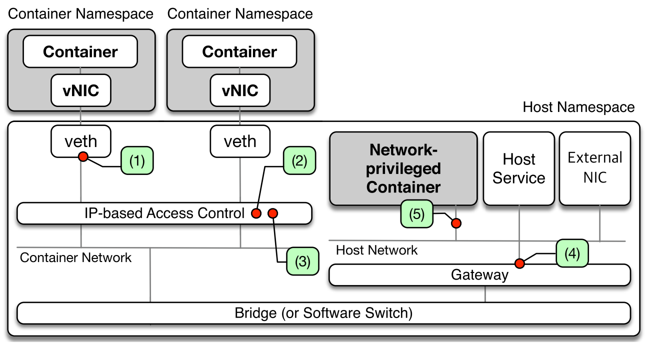


Fig. 2. Five critical challenges in container networks: (1) Limitation of packet-based source verification, (2) Limitations of IP-based access controls, (3) Lack of application-level security inspections, (4) Unrestricted host accesses, and (5) No restriction on network-privileged containers.

connectivity across multiple nodes. For example, in the Weave overlay network, each node has a special bridge interface, named `weave`, to connect local containers, and the weave bridges in all nodes are logically linked as a single network. While Kubernetes uses Docker Containers, it does not utilize Docker networking features to manage network flow control. Rather, it separately applies network policies using `iptables`. Calico [28] similarly applies network and security policies using `iptables`. If operators want further security enforcement, they may use Cilium [29], a security extension that conducts API-aware access control (e.g., HTTP method) by redirecting network traffic to its security container.

Network-privileged Containers: Besides the typical use of containers, there are special cases in which an operator wants to expose containerized services directly using the host IP address (e.g., HAProxy [30], OpenVPN [31], and MemSQL [32]). In such cases, by sharing the host namespace with a container, the container is provided access to the host network interfaces and directly exposes its services. In this work, we refer to such cases as *network-privileged containers*.

C. Challenges in Container Networks

While current container platforms utilize OS-level networking features provided by the Linux kernel to support inter-container connectivity and IP-based access control (e.g., `iptables`) to enforce container network security policies, there are significant limitations in their ability to constrain the communication privileges of today's container topologies. The following are five concerns that arise from these current OS-level architectural limitations.

(1) Limitation of packet-based source verification: Figure 2 shows that each container has its virtual interface, but this interface is only visible in the container's network namespace. Thus, container platforms effectively create a twin virtual interface corresponding to it on a host. This virtual interface is connected to the bridge, enabling connectivity with others.

However, one security-relevant problem of this design is that each packet produced by a container will lose its association with the source container at the moment that it transitions into the host networking namespace. Hence, all decisions for further security inspection and packet forwarding would be solely made based on the information in packet headers. Unfortunately, a malicious container can directly forge packets on behalf of other containers, allowing lateral attacks and traffic poisoning when any container is compromised.

(2) Limitation of IP-based access controls: The primary method for imposing network flow control among container platforms is through `iptables`, an IP-based access control mechanism provided by the Linux kernel. Unfortunately, the IP addresses of containers can be dynamic, and adjustments are then required whenever containers are spun up and down. Thus, it can be challenging to specify security policies for either case since these policies must be updated when containers are re-created. Even though operators enforce various security policies with high-level labels for containers instead of specific IP addresses, such labels are eventually converted to IP addresses, so we still have the same challenge. In addition, container networks are still vulnerable to layer-2 attacks due to the limited scope of the IP-based access control mechanism.

(3) Lack of application-level security inspections: Although the IP-based access control can restrict malicious network connectivity among containers, it cannot control malicious contents among benign containers, which raises significant concerns as a malicious container can conduct lateral attacks against dependent containers without any restriction. In addition, unlike legacy networks where we can deploy various middleboxes or virtualized network functions (VNFs) between networked entities (e.g., switches and hosts), containers communicate in multiple networks logically created using OS-level networking features within the same host. Thus, applying security functions (e.g., deep packet inspections) per logical network inside a host is difficult.

(4) Unrestricted host access: Each container network has a gateway interface for external accesses connected to the host network, as shown in Figure 2. Unfortunately, an inherent security concern arises as a container can thus access a service launched at the host-side through the gateway IP address. In Kubernetes, containers can even access all other hosts (nodes) through the gateway IP addresses. In the worst case, a malicious container can exploit the service in a manner that can subvert/harm the host's availability.

(5) No restriction on network-privileged containers: While a network-privileged container can gain a performance advantage as its traffic does not pass through additional network stacks (e.g., container networks), such a container also raises significant concerns for operational isolation. Since network-privileged containers share the same network namespace with the host, they can access not only the host network interfaces but can also monitor all network traffic from deployed containers in the host (through the virtual interfaces for the containers) and are unrestrained in their ability to inject malicious packets into container networks. Furthermore, current security solutions do not consider security policies for such containers; hence, operators must design and specify a security policy configuration for the containers by themselves.

III. Security Analysis of Container Networks

This section explores the attack surfaces in container networks and introduces an example scenario that illustrates the viability of the network threats abusing the attack surfaces.

A. Assumptions and Threat Model

Assumptions: Consider the case of containers connected to operate as microservices using Docker or Kubernetes network configurations. Let us assume that an attacker possesses enough skill (e.g., gaining a remote shell to execute arbitrary commands inside a container [34], [35]) to perform a remote hijacking of an Internet-accessible container application operating as part of a microservice using published container vulnerabilities [36], [37], [38]. Note that there is no particular difference between typical and containerized applications. The only difference is that containerized applications are the typical applications packaged by containerization techniques, meaning that any vulnerabilities in typical applications are also in the corresponding containers. With this fact, we consider what an attacker may do after getting into the subverted container.

Threat Model: The scope of threat models considered in this work focuses on network-based lateral attacks launched from a compromised container rather than system-based attacks that may occur within a container. Unlike network-based attacks, system-based attacks have been actively explored in other work, such as abusing privileged and unprivileged containers [39] and modifying Linux capabilities within a container [40], and defense techniques based on status inspection of namespaces [41]. Thus, we believe that an operator would properly deploy containers with system-wide security policies, and we, therefore, do not consider system-wide threats (e.g., attacks against the host kernel).

Here, a specific attack case involves one in which a compromised container is employed "as is" as the launching point for these lateral attacks, where *no privilege escalation* is required within the container to conduct further exploitation. Also, an attacker can acquire a base understanding of the compromised container's network configuration by investigating several system files (e.g., `/proc/net/arp`, `/proc/net/route`) and environment variables and may download malicious binaries to the `/tmp` directory in a container, as this directory has global read or write permissions for all processes.

B. Attack Surfaces in Container Networks

Attacks against container networks can be categorized into two main classes: (i) attacks that reveal topology information of the container networks (topology visibility attacks) and (ii) attacks that perform illicit monitoring or modifying of network traffic within container networks (traffic visibility attacks).

In the topology visibility attacks, *network probing* and *host exploit* attacks are possible in current container networks. Network probing attacks are performed by probing containers using TCP/IP packets (e.g., TCP-flag-based scans) or via ARP requests. Although an operator can adopt IP-based access controls to avoid scanning, an attacker may still employ an ARP scan method (bypassing iptables-like protections). The host exploit attack accesses the host upon which the compromised container runs. Accessible host services are scanned for, which may later be exploited to compromise the host further. Container systems employ different network namespaces to isolate different networks. However, container systems also use a bridge interface to integrate different network namespaces, and an attacker can abuse this bridge to access the host.

In the traffic visibility attacks, first, the ARP *poisoning attack* can overwrite ARP caches in the containers. By sending fake ARP responses, the attacker can redirect network traffic between a target container and the gateway of the target network (or another target container) to their containers. Second, the attacker can capture the network traffic between a container and the gateway (or another container) through *packet sniffing* and extract sensitive information (e.g., user credentials, tokens, and even confidential files). Third, the attacker can inject malicious packets into a target container (*IP spoofing attack*). The two remaining cases involve TCP session manipulation. In the fourth attack, one can disrupt existing sessions by injecting fake packets with proper SEQ and ACK numbers because one can observe the SEQ and ACK numbers of the sessions through sniffing TCP packets. Even when an attacker does not know the SEQ and ACK numbers, such an attack remains possible using a predictive attack [42]. An attacker can terminate existing sessions by injecting a TCP packet with the RST flag (*TCP reset attack*). Fifth, attackers can create fake sessions with other containers or external entities (*fake session attack*). An attacker can observe network traffic toward a target container and then reverse the injected packet target to the external connection point rather than the victim container.

C. Limitations of Container Network Plugins

Here, we briefly discuss the limitations of current container network interface plugins. Table I presents the feasibility of network threats discussed in Section II-C.

Docker, Flannel, WeaveNet: Docker [22], Flannel [26], WeaveNet [27] operate on bridge-based L2 forwarding, which is tightly coupled with the networking features and the IP-based access control provided by the host OS. Hence, they have the same security challenges discussed in Section II-C and are vulnerable to all network threats in Table I.

Calico: Calico [28] employs IP-in-IP-based L3 routing and uses a single MAC address (`EE:EE:EE:EE:EE:EE`) for all containers, which makes L2 attacks infeasible. However, it remains vulnerable to L3/4 attacks (e.g., TCP SYN floods, DNS reflection attacks, ICMP spoofing attacks etc.) since Calico mainly focuses on packet routing and IP-based access control while paying less attention to security mechanisms that protect against L3/4 spoofing attacks. In addition, while the host-service abuse is infeasible because Calico uses a virtual gateway IP address (`169.254.1.1`) for all containers, it does not provide security mechanisms that guard against the host network namespace abuse.

Open vSwitch: Open vSwitch (OVS) [33] provides more flexible networking features than the host OS; thus, it might be viewed as an alternate solution for bolstering container network security. OVS can derive which virtual port a container is connected to, which could be used to prevent spoofing attacks. However, one critical concern is that OVS does not support a `NOT operation`. It means that we need to install all possible flow rules from each container to other containers, which at least contain (the virtual port and the MAC/IP addresses of a source container, the IP address and the service port of

TABLE I
POTENTIAL OF NETWORK THREATS ACROSS CONTAINER NETWORK INTERFACE PLUGINS. FEASIBLE (●): NETWORK ATTACK CAN BE SUCCESSFULLY EXECUTED OVER THE CONTAINER NETWORK INTERFACE PLUGIN. PROBABLE (▲): NETWORK ATTACK REMAINS POSSIBLE, BUT MAY BE BLOCKED WITH APPROPRIATE NETWORK SECURITY POLICIES. INFEASIBLE (✕): NETWORK ATTACK IS ALWAYS BLOCKED

Network Threats	Docker [22]	Flannel [26]	WeaveNet [27]	Calico [28]	Open vSwitch [33]	Cilium [29]	Bastion$^+$
L2 attack (e.g., ARP Spoofing)	●	●	●	✕	▲	✕	✕
Traffic Eavesdropping	●	●	●	✕	▲	✕	✕
L3/L4 attack (e.g., IP Spoofing)	●	●	●	●	▲	▲	✕
Host Service Access	●	●	●	✕	▲	▲	✕
Host Network Namespace Abuse	●	●	●	●	●	●	✕

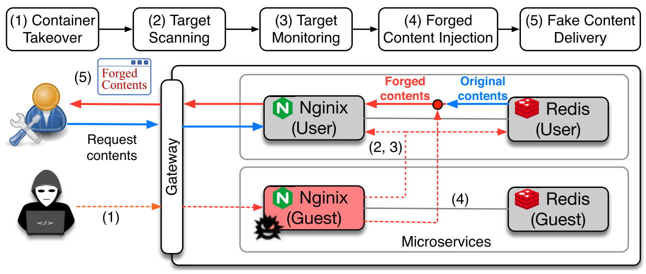


Fig. 3. An example attack scenario within a Kubernetes environment. A compromised container from one service conducts a series of network attacks to hijack communications between other containers in a peer service.

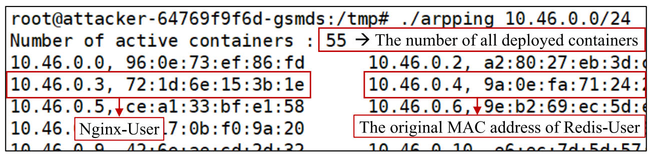
a destination one) matching fields for source verification and spoofing attack prevention. In addition, frequent rule updates are inevitable (as in the case of iptables) whenever containers are spun up and down. While OVS may block unauthorized host IP address accesses, it still allows containers to access host services using gateway IP addresses since OVS is located at the host network namespace. Unfortunately, OVS would still need a large number of security policies against all possible host accesses from each container. In addition, OVS does not protect network-privileged containers.

Cilium: Cilium [29] operates at the L3 routing level and provides advanced network security mechanisms for implementing L3-7 firewalls. In addition, L2 attacks are not feasible, as in the case of Calico. However, other network threats remain possible. Although Cilium supports a range of network policies (e.g., identity and label-based policies), which can block accesses to specific containers or hosts, the feasibility of such network threats depends on the operator and deployment considerations. If operators carefully investigate their container network and apply security policies against various network threats, the network threats might be infeasible. If not, some of the network threats are still available. Network-privileged containers are beyond its threat model; thus, Cilium is still vulnerable to them.

Bastion$^+$: Bastion$^+$ is designed as a transparent container-network security extension that protects against diverse security challenges discussed in Section II-C, which enables the same security functionalities against the network threats to the container networks (we will describe it in the next section).

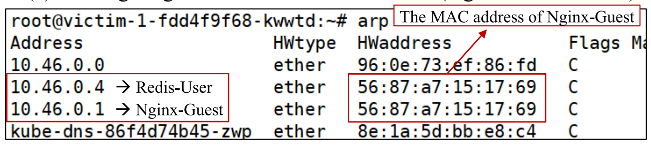
D. Attack Scenario Example

Figure 3 illustrates two independent services deployed along with common microservices [43], [44] in a Kubernetes environment. One is a service for legitimate users, and the other is a service for guest users. These services use the official Nginx [45] and Redis [46] container images retrieved from



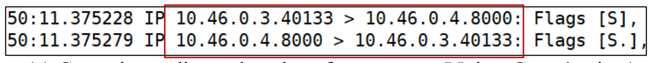
```
root@attacker-64769f9f6d-gsmds:/tmp# ./arpping 10.46.0.0/24
Number of active containers : 55 → The number of all deployed containers
10.46.0.0, 96:0e:73:ef:86:fd         10.46.0.2, a2:80:27:eb:3d:
10.46.0.3, 72:1d:6e:15:3b:1e         10.46.0.4, 9a:0e:fa:71:24:
10.46.0.5, ce:a1:33:bf:e1:58         10.46.0.6, 9e:b2:69:ec:5d:
10.46.      7:0b:f0:9a:20            The original MAC address of Redis-User
```

(a) Probing neighbor containers in a network (Nginx-Guest's view)



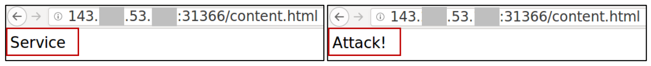
```
root@victim-1-fdd4f9f68-kwwtd:~# arp   The MAC address of Nginx-Guest
Address                     HWtype  HWaddress           Flags Ma
10.46.0.0                   ether   96:0e:73:ef:86:fd   C
10.46.0.4 → Redis-User      ether   56:87:a7:15:17:69   C
10.46.0.1 → Nginx-Guest     ether   56:87:a7:15:17:69   C
kube-dns-86f4d74b45-zwp     ether   8e:1a:5d:bb:e8:c4   C
```

(b) Spoofing target containers (Nginx-User's view)



```
50:11.375228 IP 10.46.0.3.40133 > 10.46.0.4.8000: Flags [S],
50:11.375279 IP 10.46.0.4.8000 > 10.46.0.3.40133: Flags [S.],
```

(c) Capturing redirected packets from targets (Nginx-Guest's view)



```
143.[redacted].53.[redacted]:31366/content.html    143.[redacted].53.[redacted]:31366/content.html
Service                                            Attack!
```

(d) Injecting packets with forged contents (before / after)

Fig. 4. Screenshots demonstrating the attack scenario in a Kubernetes environment between two services.

Docker Hub [14]. In this scenario, an attacker forges legitimate user requests after infiltrating into the public-facing Nginx server.

In this attack kill chain, the attacker leverages three network-based attacks to compromise the Nginx-Guest container and successfully execute a man-in-the-middle attack. In the first step, he discovers active containers around the network through ARP-based scanning. Since all containers are connected to an overlay network and ARP packets are not filtered by `iptables`, the attacker can easily collect the network information of containers, as shown in Figure 4-(a). Then, the attacker injects fake ARP responses into the network to make all traffic between the Nginx-User and the Redis-User containers pass through the Nginx-Guest. As shown in Figure 4-(b), we can see that the MAC address of the Redis-User in the ARP table of the Nginx-User is replaced with that of the Nginx-Guest, and the attacker monitors all traffic between the Nginx-User and the Redis-User (Figure 4-(c)). Lastly, the attacker replaces the response for the legitimate user with forged contents by internally dropping the packets delivered from the Redis-User and injecting forged packets. Then, the Nginx-User returns the forged contents to the user instead of the original ones (Figure 4-(d)). In the end, the user receives the forged contents as the attacker intended.

IV. BASTION$^+$ DESIGN

This section introduces a new inter-container communication bridge, Bastion$^+$, and discusses how its components address the limitations discussed in Section II-C.

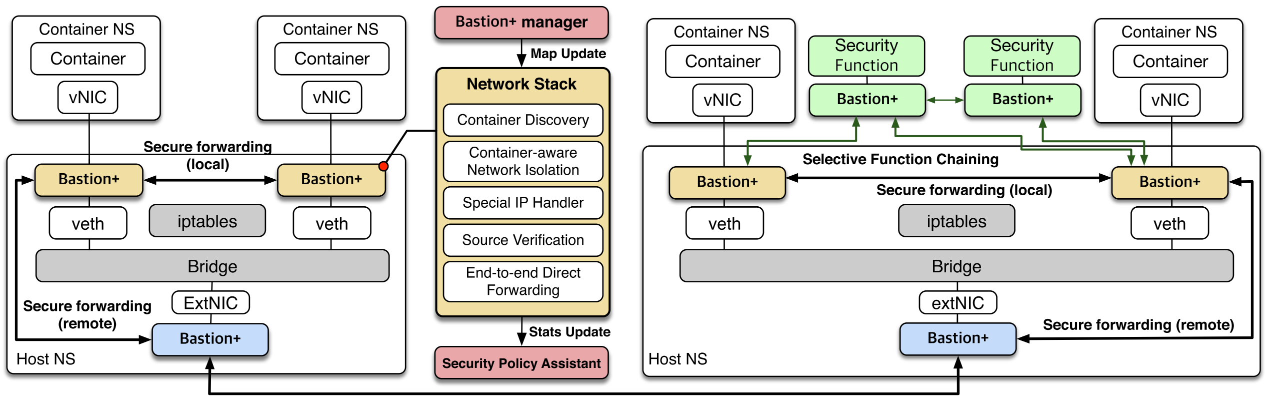


Fig. 5. Bastio$^+$ Architecture Overview. Orange box: Bastio$^+$ network security enforcement stack for containers. Red box: manager that maintains the global view of container networks and security policy assistant that discovers network policy misconfigurations. Green box: selective security function chaining that enables application-level security inspections against inter-container network flows. Blue box: Bastio$^+$ network stack for inter-host communications.

A. Architectural Overview

As illustrated in Figure 5, Bastion$^+$ is composed of three parts: a *manager*, which maintains the global network view of all containers with their inter-container dependencies, *per-container network stacks* where all Bastion$^+$ security enforcement occurs before a container's packets are delivered into the container network and *chained security functions*, which are deployed to enable further security inspections (e.g., content-based access controls) against inter-container network traffic.

When packets arrive at the Bastion$^+$ network stack, it proactively filters any discovery processes of irrelevant containers by dealing with ARP requests based on the container network map. It restricts the communications between containers according to security policies specified in the inter-container dependency map. In addition, it restricts unauthorized access to special IP addresses (e.g., gateway IP addresses). Also, it conducts secure packet-forwarding between containers by directly passing packets from source containers to destination containers while verifying if the packets are forged. If packets need to pass through security functions (e.g., network intrusion detection/prevention systems and web application firewalls), Bastion$^+$ forwards them into the corresponding security functions. Then, once the security functions do deep packet inspections, the packets are delivered to the destination containers. Since all direct packet forwarding occurs at the network interface level, packets are not passed through the container network (host-side), eliminating any chance for unauthorized network traffic exposure.

In terms of inter-container communications across hosts (nodes), a specialized Bastion$^+$ network stack is utilized at the external interface of each node, conducting a secure forwarding from the external interface to destination containers as all security decisions are already made at each container. Bastion$^+$ retains the existing mechanisms of container platforms to handle inbound traffic from external networks.

B. Bastio$^+$ Manager

The Bastion$^+$ manager performs three primary roles: container network information collection, network stack management, and security function management.

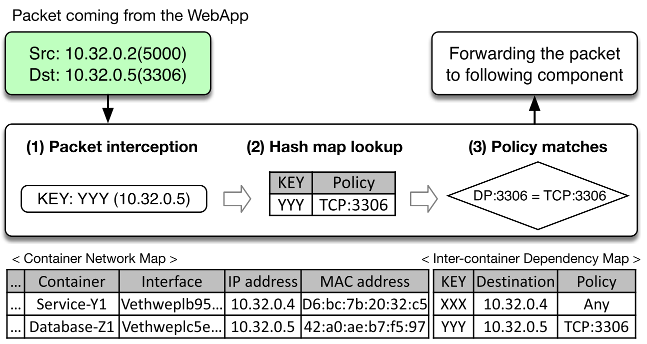
(1) Container Network Information Collection. The manager has a global container network map and an inter-container dependency map for all containers. It directly communicates with container platforms to retrieve the network information (virtual network interface, IP and MAC addresses) for all containers and to build the inter-container dependency map by extracting the dependencies (source $\rightarrow$ protocol://destination:{port $|$ any}) among containers based on the retrieved information and their ingress/egress security policies. In addition, as containers can be dynamically spun up and down, the manager watches any changes in container platforms (event-driven) and updates the maps in run-time.

(2) Network Stack Management. The manager maintains the Bastion$^+$ secure network stack for each container. For newly spawned containers, it installs the network stacks at their network interfaces and updates the container network and inter-container dependency maps in the network stacks. Each container only requires a part of the network information to communicate with dependent neighbors with respect to map size. Thus, to reduce the size of security services, The manager filters irrelevant information per container. Then, whenever there are changes in inter-container dependencies, it automatically updates the maps for the corresponding containers.

(3) Security Function Management. The manager deploys various security functions (e.g., intrusion detection and prevention systems and web firewalls) per host and attaches Bastion$^+$ network stacks for those security functions. Then, according to function chaining policies (i.e., the order of functions according to the network context (e.g., protocol and destination port)), the manager updates the container network map in the Bastion$^+$ network stacks of containers and security functions to keep passing inter-container network traffic through the security functions. We will explain the details in Section IV-D.

C. Network Security Enforcement Stack

The Bastion$^+$ network stack restricts inter-container communications from two points of view: network visibility and traffic visibility. For network visibility, it restricts unnecessary connectivity among containers and between containers and external hosts. For traffic visibility, it provides point-to-point integrity and confidentiality among container network



KEY	Policy
YYY	TCP:3306

...	Container	Interface	IP address	MAC address
...	Service-Y1	Vethweplb95...	10.32.0.4	D6:bc:7b:20:32:c5
...	Database-Z1	Vethweplc5e...	10.32.0.5	42:a0:ae:b7:f5:97

KEY	Destination	Policy
XXX	10.32.0.4	Any
YYY	10.32.0.5	TCP:3306

Fig. 6. Workflow of container-aware network isolation. The WebApp container accesses the database container, and the container-aware network isolation in the WebApp's network stack inspects the dependency on the database.

flows. Here, we present each component that addresses those points.

1) Container Discovery: For inter-container networking, container discovery is the first step to identify other containers (communication targets). Containers use ARP requests to identify target containers' necessary network information (i.e., MAC addresses). Unfortunately, this discovery process can be exploited to scan all containers connected to the same network by malicious containers, as current network stacks do not prevent ARP scans. Indeed, they offer no mechanism to control non-IP-based communications.

Bastion$^+$ filters out any unnecessary container discovery that does not pertain to the present container's dependency map. When a container sends an ARP request, the handler intercepts the request before it is broadcasted, verifying if the source container has a dependency on the destination container. This analysis is done using the inter-container dependency map. If accessible, the handler generates an ARP reply with the MAC address of the destination container and sends the reply back to the source container. If not, it drops the request.

2) Container-Aware Network Isolation: Even though container discovery prevents containers from performing unbounded topology discovery, its coverage is limited to container-level isolation. It does not address malicious accesses among dependent containers. Hence, Bastion$^+$ implements container-aware network isolation to restrict the reachability of containers further.

When packets arrive at the Bastion$^+$ network stack of a source container, as shown in Figure 6, Bastion$^+$ first checks the dependency between the source and its destination by examining the inter-container dependency map using the destination IP address as a key. If any policies exist in the map, it concludes that the source has a dependency on the destination. Then, Bastion$^+$ matches the packets to the policies for the destination container, and the connection is allowed if matched. Otherwise, the packets are dropped.

3) Gateway and Service-IP Handing: In container environments, a subverted container can exploit the gateway to probe services within the host OS. To address this concern, Bastion$^+$ filters direct host accesses. When a network connection targets non-local container addresses, it includes the gateway MAC address and the IP address of the actual destination. Based on this fact, the gateway-IP handler blocks any direct host access by checking if both IP and MAC addresses belong to the gateway. It would also be possible that a network flow might access the gateways of other container networks since they are connected to the host network. Hence, the gateway-IP handler also filters unauthorized host accesses by comparing packets with the other gateways.

In Kubernetes environments, another special IP address, called a service IP address, is used to redirect actual containers. Unfortunately, since service IP addresses are virtual IP addresses that do not belong to container networks, they can be considered external IP addresses. Thus, Bastion$^+$ additionally extracts the pairs of {service IP address, port} and {corresponding container IP address, port} from Kubernetes and maintains a service map in each Bastion$^+$ network stack. Then, when a container sends a packet with a service IP address and port, the service-IP handler overwrites the service IP address and port to an actual container IP address and port according to the service map. As a result, the other security components can process packets as intended.

4) Source Verification: One problem within the current network stack is that all security enforcement and packet forwarding rely on packet-header information. Thus, a malicious container can submit packets that match the identity of target containers. Doing so can redirect traffic of a target container to itself (e.g., ARP spoofing attack). Also, the malicious container can modify the traffic passing through it (e.g., Man-In-the-Middle attack) or inject forged packets to disrupt another container's existing sessions (e.g., TCP RST attack).

Considering those cases, Bastion$^+$ leverages the predefined network information in each network stack and a network session map to precisely track the actual source of inter-container traffic. The Bastion$^+$ network stack of each container statically contains the corresponding container's network information (i.e., IP/MAC addresses), and Bastion$^+$ verifies the incoming traffic by comparing its packet header information to the container's information embedded in the Bastion$^+$ network stack. If the packet header information is not matched with the container's network information, Bastion$^+$ identifies the incoming traffic as spoofed and drops it. Furthermore, since network-privileged containers can inject spoofed packets into other containers, Bastion$^+$ maintains the session information of the incoming traffic in the network session map, which is globally maintained across Bastion$^+$ network stacks. Thus, if some packets arrive at the Bastion$^+$ network stack and their session information is not in the global network session map, Bastion$^+$ considers them spoofed packets and drops them immediately. As a result, Bastion$^+$ can effectively eliminate the spectrum of disruption and spoofing threats.

5) End-to-End Direct Forwarding: Current network stacks cannot prevent the exposure of inter-container traffic from other containers. If a malicious container can redirect the traffic of a target container to itself, it can monitor the traffic without restriction. In the case of network-privileged containers, they have the full visibility of all container networks: they can monitor the traffic of others without any traffic redirection.

To implement least-privilege traffic exposure, as shown in Figure 7, Bastion$^+$ performs direct packet delivery between

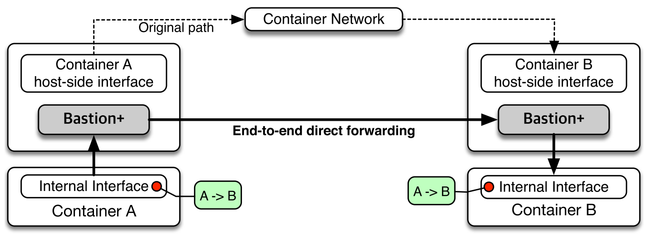


Fig. 7. An illustration of how Bastio^{+} implements end-to-end direct packet forwarding to bypass exposure of intra-container traffic to other containers.

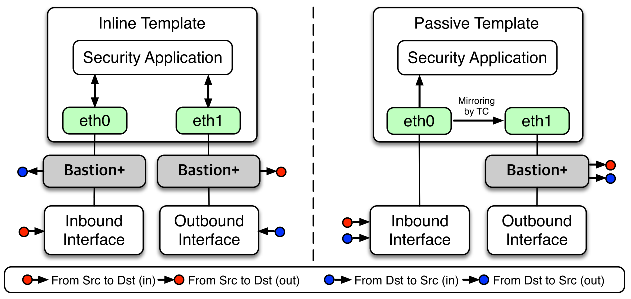


Fig. 8. Two types of security function templates: inline and passive templates. Any security applications can run as usual based on the inbound and outbound interfaces. while Bastio^{+} takes in charge of all network flow controls.

source and destination containers at the network interface level, bypassing not only their original network stacks (host-side) but also bridge interfaces. As soon as Bastion^{+} receives an incoming network connection from a container, it retrieves the interface information of a destination from the container network map. Bastion^{+} directly injects the packet stream into the destination container if the destination is a container in the same node. If the destination is a container in another node, Bastion^{+} injects the packet to the external interface of a node. Then, once the special Bastion^{+} network stack of the external interface at the target node receives the packet, it directly injects the packet stream into the destination container.

As capturing network traffic happens *in the ingress and egress traffic controls*, no traffic would be visible at the traffic controls with Bastion^{+} as it handles all network traffic at the network interface level (before the traffic controls). Thus, this traffic isolation prevents any traffic disclosure by other containers and even network-privileged containers.

D. Security Function Chaining

As discussed in Section II-C, current security solutions for container networks have limitations in security inspections at the level of applications. Here, we introduce security function chaining for further application-level inspections.

1) Challenges in Security Function Chaining: While network service chaining has been widely used in software-defined networking (SDN) and network function virtualization (NFV) because of the flexibility of network flow controls, it has not been well adopted into container networks due to their architectural limitations. Multiple microservices are deployed inside hosts, and they have their logical networks with different IP address spaces. Hence, it is difficult to either deploy security services per microservice due to the resource limits of hosts or deploy shared security service instance per host due to the unreachability among the different microservices.

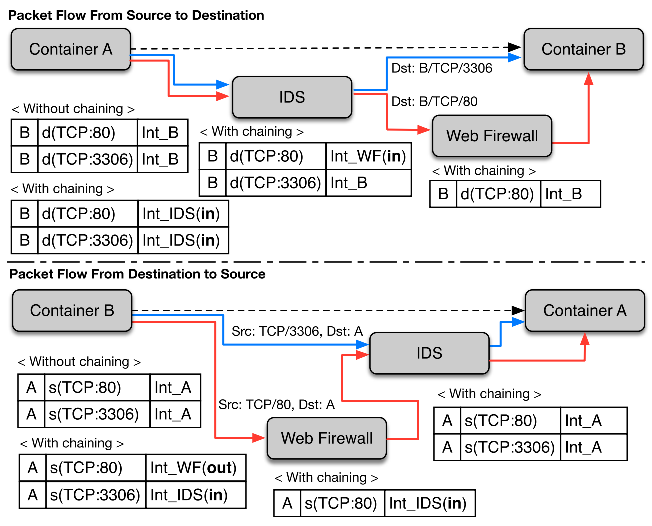
To address those challenges, Bastion^{+} introduces a way to deploy common security functions per host, while providing resource efficiency. Then, to address the network unreachability issue, Bastion^{+} directly leverages the kernel features rather than implementing these services as extension to the network processing pipeline. By utilizing the end-to-end direct forwarding, Bastion^{+} forces network flows through security functions at the kernel side, bypassing the reachability issue.

2) Security Function Deployment: Bastion^{+} provides two templates for inline and passive security functions, as shown in Figure 8. Each function has two network interfaces for inbound and outbound traffic. In the case of the inline template, Bastion^{+} network stacks are attached at both interfaces since the direction of packet flows is important to the inline functions, especially for session management. On the other hand, the passive template has one Bastion^{+} network stack attached to the outbound interface while all traffic is mirrored from the inbound interface to the outbound interface. Then, operators build security functions with legacy security applications based on those templates according to the types of applications. Then, Bastion^{+} deploys those functions to each host.

3) Selective Security Function Chaining: Bastion^{+} enables selective security function chaining according to security function chaining policies, which define the order of functions based on the network context. Here, we explain how the function chaining works with the example illustrated in Figure 9. In this example, we have two containers (Container A and B) and two security functions (IDS and Web Firewall). When Container A sends a packet to Container B, Bastion^{+} directly forwards the packet to Container B. However, the packet flows differently when chaining policies are applied.

Let us assume that an administrator applies two chaining policies to Container A: (i) make the TCP/80 sessions toward Container B pass the IDS and the Web Firewall and (ii) make the TCP/3306 sessions toward Container B pass the IDS. Then, Bastion^{+} updates the interface information for Container B in the container network and dependency maps of Container A and the security functions according to those chaining policies. First, Bastion^{+} updates the destination interface for both TCP/80 and TCP/3306 toward Container B to the inbound interface of the IDS in the maps of Container A. Second, in the outbound-side network stack for the IDS, Bastion^{+} updates the interface for the TCP/80 to the inbound interface of the Web Firewall while it updates the interface for the TCP/3306 to that of Container B. Lastly, in the outbound-side network stack for the Web Firewall, Bastion^{+} updates the interface of the TCP/3306 to that of Container B. As a result, packets can pass through either the IDS and the Web Firewall or the IDS only according to the protocol and port information.

In the case of the opposite direction (from Container B to Container A), most of the map updates are similar. The only difference is the direction. First, Bastion^{+} updates the interface for the TCP/80 from Container B to the outbound interface of the Web Firewall for the session management in the network stack of Container B while it updates the interface for the



< Without chaining >

B	d(TCP:80)	Int_B
B	d(TCP:3306)	Int_B

< With chaining >

B	d(TCP:80)	Int_IDS(**in**)
B	d(TCP:3306)	Int_IDS(**in**)

< With chaining >

B	d(TCP:80)	Int_WF(**in**)
B	d(TCP:3306)	Int_B

< With chaining >

B	d(TCP:80)	Int_B

< Without chaining >

A	s(TCP:80)	Int_A
A	s(TCP:3306)	Int_A

< With chaining >

A	s(TCP:80)	Int_WF(**out**)
A	s(TCP:3306)	Int_IDS(**in**)

< With chaining >

A	s(TCP:80)	Int_A
A	s(TCP:3306)	Int_A

< With chaining >

A	s(TCP:80)	Int_IDS(**in**)

Fig. 9. Selective network flow redirections according to service chaining policies. The TCP/80 session between container A and B passes through the IDS and the web firewall while the TCP/3306 session passes through the IDS only.

TCP/3306 from Container B to the inbound interface of the IDS as the IDS is a passive function. Then, in the inbound-side network stack of the Web Firewall, Bastion$^+$ updates the interface for the TCP/80 from Container B to the inbound interface of the IDS. Lastly, in the outbound-side network stack of the IDS, it updates the interface for TCP/80 and TCP/3306 to that of Container A.

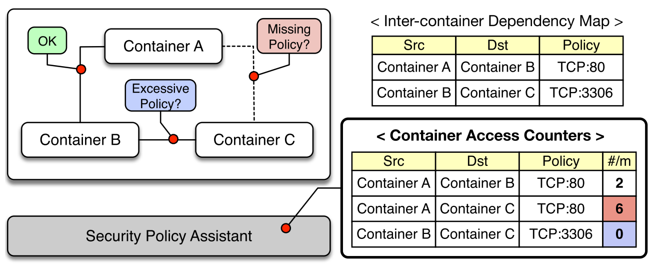
E. Security Policy Assistant

While Bastion$^+$ restricts the ability of attackers through its security components, no container network can be made secure without proper security policies that restrict communications to the minimum required access.

Bastion$^+$ provides a security policy assistant that can help identify the security policies that a container operator should consider, as illustrated in Figure 10. Bastion$^+$ produces flow monitoring statistics that automatically capture the observed container accesses during the inter-container traffic flow control. In parallel, it compares these statistics with the inter-container dependency map, classifying them into three cases: legitimate accesses, missing policies, and excessive policies. If the two containers are in both the inter-container dependency map and there are connections between them, we consider their security policies well-defined. Otherwise, the operator may consider either a missing security policy or an excess policy, which differs from the actual flow statistics.

The security policy assistant effectively directs the operator to review specific flows to determine whether to produce missing network security policies in the current operating configuration. In addition, it identifies policies for which no flows have been encountered. Such cases may represent an over-specification of policies that enable unnecessary flows for the container network's operations.

While the security policy assistant does not directly correct security policies, its feedback can be used as vital insights to help the operator confirm (or improve) the current network



< Inter-container Dependency Map >

Src	Dst	Policy
Container A	Container B	TCP:80
Container B	Container C	TCP:3306

< Container Access Counters >

Src	Dst	Policy	#/m
Container A	Container B	TCP:80	2
Container A	Container C	TCP:80	6
Container B	Container C	TCP:3306	0

Fig. 10. Security policy assistant to discover inter-container dependencies and detect possible configuration errors.

security policies. For example, the network policies for large microservices may be relatively complicated to validate, and operators may overlook policies that are broader than required. In such cases, the security policy assistant can offer an iterative stream of operational information that can help the operator maintain and improve even complex network policies for a large number of containers.

V. Implementation

We implement a prototype of Bastion$^+$ with 2.1K lines of C code and 12.5K lines of Go code on the Linux kernel v4.16.

Bastion$^+$ Manager: For container network information collection, the manager watches the attribute changes (e.g., NetworkSettings) of active containers from the Docker engine and the Kubernetes API server. In addition, Bastion$^+$ uses the label-based configurations for container deployments and *ingress*/*egress* network security policies in Kubernetes to generate inter-container dependencies.

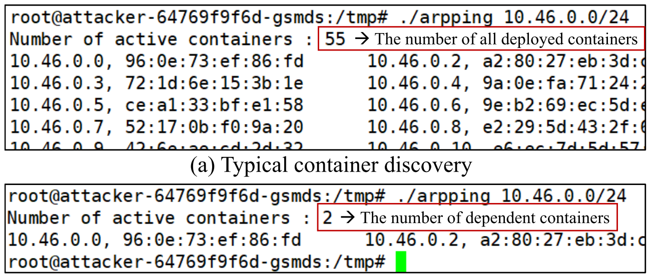
Network Security Enforcement Stack: Each container's security enforcement network stack is implemented using eBPF [47] and XDP [48], [49]. During the inspection, the network stack looks up two hash maps (i.e., the container network and inter-container dependency maps), which are synchronized with the Bastion$^+$ manager. Then, they employ XDP actions to send a packet back to the incoming container, inject a packet into a destination, and drop a packet.

Security Function Chaining: Bastion$^+$ deploys security applications (for evaluation, we deployed Snort IDS [50] and Suricata IDS/IPS [51]) by using its templates and as daemon containers for each host according to the types of the applications (i.e., inline vs. passive). For passive functions, Bastion$^+$ internally configures the traffic controls (using "*tc qdisc*" and "*tc filter*") to mirror incoming traffic to the outbound interface.

Security Policy Assistant: To monitor the network flows between containers, Each security stack maintains an eBPF map shared with the security policy assistant and increments the counters for network flows after removing the randomness (e.g., a source port) of the network flows. Whenever a container is terminated, the corresponding eBPF map is removed.

VI. Security Evaluation

This section demonstrates how Bastion$^+$ mitigates network attacks based on the attack scenario, shown in Section III-D, that abuses the security holes in the current container network.



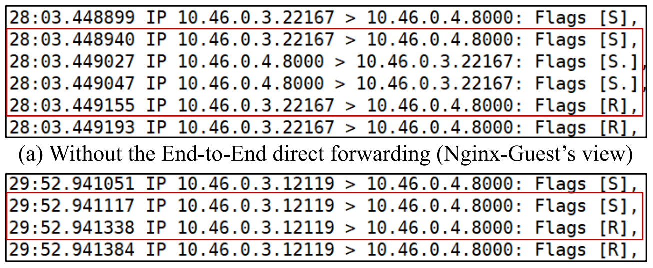
```
root@attacker-64769f9f6d-gsmds:/tmp# ./arpping 10.46.0.0/24
Number of active containers : 55 → The number of all deployed containers
10.46.0.0, 96:0e:73:ef:86:fd      10.46.0.2, a2:80:27:eb:3d:
10.46.0.3, 72:1d:6e:15:3b:1e      10.46.0.4, 9a:0e:fa:71:24:
10.46.0.5, ce:a1:33:bf:e1:58      10.46.0.6, 9e:b2:69:ec:5d:
10.46.0.7, 52:17:0b:f0:9a:20      10.46.0.8, e2:29:5d:43:2f:
10.46.0.9, 42:6e:ae:cd:2d:32      10.46.0.10, e6:ec:7d:5d:57
```

(a) Typical container discovery

```
root@attacker-64769f9f6d-gsmds:/tmp# ./arpping 10.46.0.0/24
Number of active containers : 2 → The number of dependent containers
10.46.0.0, 96:0e:73:ef:86:fd      10.46.0.2, a2:80:27:eb:3d:
root@attacker-64769f9f6d-gsmds:/tmp#
```

(b) Restricted container discovery

Fig. 11. An illustration of neighbor container discovery: upper panel - an attacker can discover all peer containers, lower panel - our container discovery and container-aware flow control only allow the inter-dependent containers to be shown.



```
28:03.448899 IP 10.46.0.3.22167 > 10.46.0.4.8000: Flags [S],
28:03.448940 IP 10.46.0.3.22167 > 10.46.0.4.8000: Flags [S],
28:03.449027 IP 10.46.0.4.8000 > 10.46.0.3.22167: Flags [S.],
28:03.449047 IP 10.46.0.4.8000 > 10.46.0.3.22167: Flags [S.],
28:03.449155 IP 10.46.0.3.22167 > 10.46.0.4.8000: Flags [R],
28:03.449193 IP 10.46.0.3.22167 > 10.46.0.4.8000: Flags [R],
```

(a) Without the End-to-End direct forwarding (Nginx-Guest's view)

```
29:52.941051 IP 10.46.0.3.12119 > 10.46.0.4.8000: Flags [S],
29:52.941117 IP 10.46.0.3.12119 > 10.46.0.4.8000: Flags [S],
29:52.941338 IP 10.46.0.3.12119 > 10.46.0.4.8000: Flags [R],
29:52.941384 IP 10.46.0.3.12119 > 10.46.0.4.8000: Flags [R],
```

(b) With the End-to-End direct forwarding (Nginx-Guest's view)

Fig. 12. Restricting traffic visibility: upper panel - an attacker can see the traffic of the spoofed target container without end-to-end forwarding, lower panel - the attacker cannot see response traffic with end-to-end forwarding.

A. Container Discovery

When a compromised container is used to conduct peer discovery to locate other containers, as shown in Figure 11-(a), the current container network stack allows an attacker to discover all neighboring containers. On the other hand, as shown in Figure 11-(b), Bastion$^+$'s container discovery and container-aware network isolation reduce the reachability of each container based on its inter-container dependencies. The infected container (i.e., Nginx-Guest in Figure 3) has only a limited number of dependent containers (i.e., Redis-Guest in Figure 3), and Bastion$^+$ ensures that the container observes only its gateway and the dependents. In sum, our container discovery can protect containers from L2 attacks by limiting the topology visibility and controlling all L2 packets (i.e., no broadcast to containers and no response from containers).

B. Passive Packet Monitoring

As discussed previously, a compromised container may be able to sniff the network traffic of a target container. Further, when an attacker compromises a "network-privileged" container, the attacker is provided access to all network traffic with no restriction. Bastion$^+$ mitigates these concerns by implementing end-to-end direct container traffic forwarding.

Figure 12 illustrates the utility of Bastion$^+$'s direct forwarding. The upper panel, Figure 12-(a), shows the visible network traffic of a target container (i.e., Nginx-User in Figure 3) after spoofing the container without direct forwarding. The lower panel, Figure 12-(b), demonstrates the use of direct forwarding. When direct forwarding is applied, the only visible traffic from a given interface is traffic involving the container itself. *To highlight the differences, we intentionally make the flow from a source to a destination visible.* As a result, while the attacker can observe the source-to-destination flow, he can no longer observe the traffic in the reverse direction. If we entirely apply end-to-end forwarding for all traffic, the attacker will see no traffic between them. In sum, Bastion$^+$ does not allow containers and even host-privileged containers that abuse the host network namespace to eavesdrop third-party traffic.

C. Active Packet Injection

Network-based attacks frequently rely on spoofed packet injection techniques to send malicious packets to target containers. Bastion$^+$ prevents these attacks by performing explicit source verification. To illustrate its impact, we demonstrate before and after cases from the attacker and victim perspectives. In the following example, we enable source verification only and allow an attacker to conduct ARP spoofing attacks.

Figure 13-(A) illustrates cases without source verification. The attacker spoofs the Nginx-User (in Figure 3) and receives the traffic of the Nginx-User. Further, the attacker injects RST packets to terminate the session of the Nginx-User. As soon as the attacker injects the RST packets, as shown in panel (A-2), the Nginx-User receives the injected RST packets (see the received times of the RST packets), causing its session to be immediately terminated. This situation is remedied with explicit source verification. Although the attacker tries to inject RST packets, as shown in panel (B-2), the RST packets are rejected by the source verification component and prevented from reaching the Nginx-User. In sum, the source verification can protect containers from L3/L4 attacks (e.g., IP spoofing and TCP session hijacking attacks) even though these attacks are initiated from host-privileged containers.

VII. Performance Evaluation

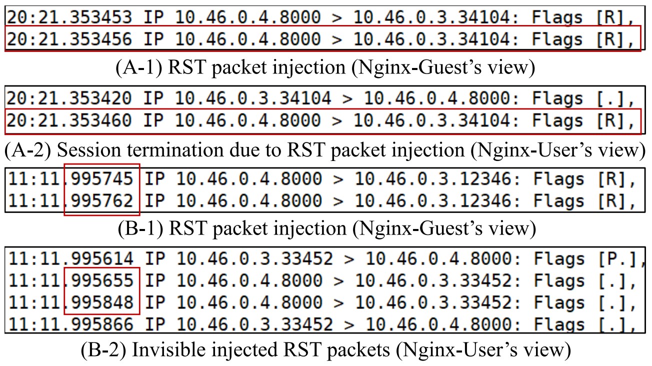
This section summarizes our measurement results of Bastion$^+$'s performance overhead with respect to latencies and throughputs between containers under various conditions.

Test Environment: We used three machines to construct a Kubernetes cluster with the WeaveNet overlay network and evaluate the Bastion$^+$ prototype. One system served as the Kubernetes master node, while the others acted as container-hosting nodes. Each system was configured with an Intel Xeon E5-2630v4 CPU, 64 GB of RAM, and an Intel 10 Gbps NIC. `netperf` [52] and `iperf` [53] were used to measure round-trip latencies and TCP stream throughputs.

A. Security Policy Inspection Overhead

We compared the matching overheads with both `iptables`-based access control and Bastion$^+$, and Figure 14 shows the TCP throughputs with different numbers of security policies. For a fair comparison on a best-effort basis, we used the same number of policies for each container.

In the case of `iptables`, security policies for all containers are maintained collectively in the host kernel. Thus, when packets arrive from containers, `iptables` first looks up the policies for the corresponding containers and inspects them individually with the incoming packets. Also, `iptables`



```
20:21.353453 IP 10.46.0.4.8000 > 10.46.0.3.34104: Flags [R],
20:21.353456 IP 10.46.0.4.8000 > 10.46.0.3.34104: Flags [R],
```

(A-1) RST packet injection (Nginx-Guest's view)

```
20:21.353420 IP 10.46.0.3.34104 > 10.46.0.4.8000: Flags [.],
20:21.353460 IP 10.46.0.4.8000 > 10.46.0.3.34104: Flags [R],
```

(A-2) Session termination due to RST packet injection (Nginx-User's view)

```
11:11.995745 IP 10.46.0.4.8000 > 10.46.0.3.12346: Flags [R],
11:11.995762 IP 10.46.0.4.8000 > 10.46.0.3.12346: Flags [R],
```

(B-1) RST packet injection (Nginx-Guest's view)

```
11:11.995614 IP 10.46.0.3.33452 > 10.46.0.4.8000: Flags [P.],
11:11.995655 IP 10.46.0.4.8000 > 10.46.0.3.33452: Flags [.],
11:11.995848 IP 10.46.0.4.8000 > 10.46.0.3.33452: Flags [.],
11:11.995866 IP 10.46.0.3.33452 > 10.46.0.4.8000: Flags [.],
```

(B-2) Invisible injected RST packets (Nginx-User's view)

Fig. 13. Restriction of packet injection. Panel A-1 shows the attacker injecting RST packets, and A-2 shows the victim session terminated by attacker's RST packet. Panel B-1 shows the trials of RST packet injections, and B-2 shows the failure of RST packet injection due to source verification.

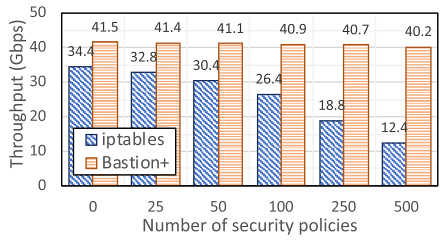


Fig. 14. Throughput variations with the increasing number of security policies.

requires a large number of field matches (at least source and destination IP addresses and ports for each policy) since it is designed for general access control. As a result, as shown in Figure 14, the throughput degraded by 23.3% with 100 policies and 64.0% with 500 policies. This trend points to a fundamental scaling challenge with the current policy enforcement approach for container networks. In contrast, the throughput degradation caused by Bastion$^+$ was barely noticeable as the number of policies increased (3.2% with 500 policies). Such performance gains stem from Bastion$^+$'s matching process optimized for containers, which comprises a hash-based policy lookup for specific destinations and their port matches (no need to match source IP addresses and ports).

B. Security Function Chaining Overhead

We measured the overheads coming from function chaining. For this, we deployed two open-source intrusion detection and prevention systems (i.e., SnortIDS [50] and SuricataIPS [51]). We used the official rules in Snort 2.9 (4K rules) for Snort IDS and the ET Open rules [54] (4K rules) for Suricata IPS.

Figure 15 shows the inter-container latencies with different combinations of security functions. With no chaining, the latencies of inter-container communications within a host were 17.5μs and 14.5μs for TCP and UDP packets, respectively. When we added the Snort IDS between the containers, the overall latencies slightly increased (+10μs on average) as the network traffic needed to pass through the SnortIDS container. However, no additional overhead for deep packet inspections was included except for the overhead of packet mirroring by traffic controls (kernel-level). When we added the Suricata IPS, the overall latencies highly increased (130.2μs for TCP

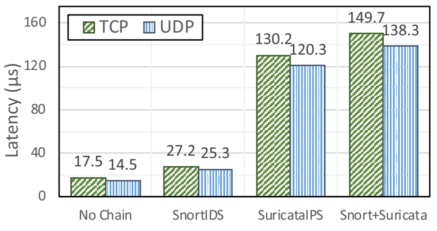


Fig. 15. Latency measurements with different Bastio$^+$ security functions.

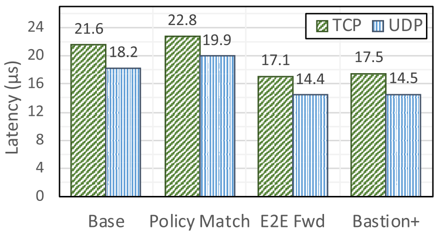


Fig. 16. Latency measurements with Bastio$^+$ components in a single host.

packets and 120.3μs for UDP packets) as all network traffic should be copied to the Suricata IPS (userspace-level), and then it should be copied to the outbound interface (kernel-level). In addition, all overheads for parsing packets and inspecting payloads increased the latencies. Note that the performance optimization of security functions is out of scope in this work. Then, when we chained both functions, the overall latencies increased up to 9.5 times compared to the baseline (no chain). However, despite significant overheads with the security functions, Bastion$^+$ provides selective security function chaining; thus, the overheads can be highly reduced by inspecting specific packets only while for non-critical packets bypassing the function chains.

C. Performance: Single-Host Deployment

Here, we evaluated latencies and throughputs between containers hosted in the same node to measure the overhead of Bastion$^+$. Figure 16 provides the round-trip latency comparison of four test cases within a single node. The base case provides latency measurements for a default configuration of two containers that interacted with no Bastion$^+$, which were 21.6μs and 18.2μs for TCP and UDP packets, respectively. When we applied Bastion$^+$'s policy matching components (i.e., container discovery and container-aware network isolation), the latencies slightly increased by 5.7% and 9.3% due to the newly applied security enforcement requiring additional packet processing to derive the reachability check between containers. When we applied Bastion$^+$'s end-to-end direct forwarding, the overall latencies were noticeably improved by 26.3% because it directly fed inter-container traffic into destination containers while bypassing the existing container networks. Finally, we observed the overall performance improvement with respect to the base case of 23.0% and 25.4% for TCP and UDP packets when all Bastion$^+$ security functions were fully applied. Figure 17 also shows that the overall throughput of Bastion$^+$ was improved by 20.6% compared to that of the base case.

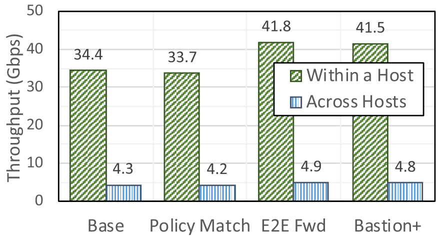


Fig. 17. Throughput measurements of the baseline and Bastio$^+$ components.

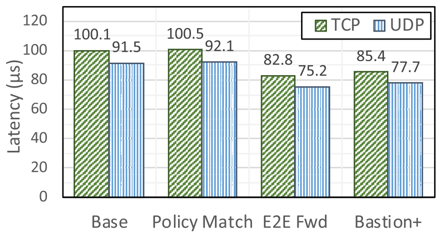


Fig. 18. Latency measurements with Bastio$^+$ components across hosts.

D. Performance: Cross-Host Deployment

Next, we measured the latencies and throughputs for cross-host container deployments. Figure 18 illustrates the measurement results with different combinations of Bastion$^+$'s security components. Compared to the intra-host measurements, the overall latencies significantly increased due to the physical link traversal and tunneling overheads between hosts; thus, the latency of the base case became 100.1μs and 91.5μs for TCP and UDP packets, respectively. Also, given the network impact, the overhead caused by Bastion$^+$'s policy matching components receded (less than 1%). Next, when we introduced Bastion$^+$'s end-to-end direct forwarding, the latencies were reduced by 21.3% because our secure forwarding directly passed network packets from the source to the destination via the external interfaces. Finally, when we applied all security components, the latencies decreased by 17.7%, significantly improving compared to the base case. These improvements translated to a cross-host throughput improvement of 12.9%, as shown in Figure 17.

E. Performance: Networking Plugins

Lastly, we compared the throughput variations in different types of container networks with/without Bastion$^+$. Figure 19 shows the TCP-stream throughputs between intra-host and inter-host containers in three container networks (i.e., Flannel, WeaveNet, and Calico). The results show that the intra-host throughputs are improved 16.0% in the Flannel network, 20.6% in the Weave network, and 20.7% in the Calico network by deploying Bastion$^+$. Regarding the inter-host throughputs, we also see the performance improvements (9.4%, 12.9%, and 4.9%, respectively) with Bastion$^+$.

F. Performance:Bastio$^+$ System

Here, we evaluated the resource usage of Bastion$^+$. For this, we divide the overall workflow into three parts: stack installation, run-time, and security policy analysis. First, we measured the CPU usage while creating 100 containers. The result shows

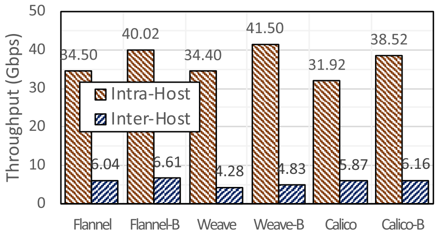


Fig. 19. Throughput comparison with different CNIs. (B = with Bastio$^+$).

that it took 13.03 μs on average for stack installation, and it consumed up to 54.7% of CPU resources in a very short time. However, while measuring the CPU usage during the performance of inter-container communications, we observe that Bastion$^+$ generally consumed 2.3% of CPU resources on average in rum time because most of the security operations are done in the kernel space and Bastion$^+$ (user-space) only manages Bastion$^+$ components. Lastly, the security policy assistant consumed less than 1% of CPU resources. As a result, we ascertain that Bastion$^+$ can provide further network isolation and security enforcement and better performance with minimal overhead costs.

VIII. Related Work

Container Security Analysis. Several efforts [39], [40], [41], [55], [56], [57] have analyzed the security issues of container implementations. For example, Dua *et al.* [55] analyzed various container implementations, concluding that they are yet insecure from filesystem, network, and memory isolation perspectives. More specifically, Jian *et al.* [41] demonstrated a Docker *escape attack*, which allows an adversary to break out of the isolation of a Docker container by exploiting a Linux kernel vulnerability. Another research area [36], [37], [58], [59], [60] of container security focuses on container images. Shu *et al.* [58] and Tak *et al.* [59], [60] have performed a large-scale vulnerability assessment of Docker images on Docker Hub and shown that many images were outdated and vulnerable. While these studies broadly point out the security issues of containers, their goals differ from our work. Instead, Bastion$^+$ focuses on container networks.

Container Security and Isolation. Bacis *et al.* introduced DockerPolicyModules (DPM) [61] that allow Docker image maintainers to specify and ship SELinux policies within their images. Sun *et al.* [62] proposed security namespaces that enable containers to independently define security policies and apply them to a limited scope of processes. SCONE [63] presented a secure container mechanism for Docker containers by isolating them inside SGX enclaves. LightVM [64] wraps containers in lightweight VMs. X-Containers [65] isolate containers that have the same concerns together on top of separate library OSes. These efforts complement the network-focused objectives of Bastion$^+$ and could be combined to deliver system- and network-wide security services.

Container Network Security. Most container network solutions [66], [67] have focused on container network performance, with little attention to fine-grained policy enforcement. A few recent studies investigated the security issues in container networks. Bui [68], Comb *et al.* [69], and Chelladhurai *et al.* [70] analyzed Docker container security. Our work extends these results by identifying the broader class of attacks, and we present system extensions that address these problems.

With respect to security policies for inter-container communications, while most solutions (e.g., Weave [27], Calico [28], and Romana [71]) have adopted `iptables`-based access control, Cilium [29] provides API-aware security mechanisms using eBPF for L3/4 policies, employing its security container for L7 policies. While Bastion$^+$ and Cilium share the use of eBPF in their implementations, their design objectives are different. Cilium pursues API-level network security filtering to define and enforce both network and application layer security policies. In contrast, Bastion$^+$ fundamentally redesigns a secure network stack per container to construct an inherently secure container networking system while also providing substantially more security features than Cilium.

IX. Conclusion

Containerization has emerged as a widely popular virtualization technology that is being aggressively deployed into large-scale enterprise and cloud environments. However, this adoption could be stifled by critical security issues, which remain understudied. We have analyzed the security challenges involved in the current container networks and addressed these challenges by presenting Bastion$^+$, an intelligent communication bridge for securing container-network communications using Linux kernel features. Bastion$^+$ restricted the network and traffic visibilities of containers with per-container fine-grained network control and container-to-container network isolation. Also, Bastion$^+$ enabled selective security function chaining according to containerized applications for further application-level inspections and helped administrators correctly configure their security policies using its security policy assistant.

References

[1] Google. *Everything at Google Runs in Containers*. Accessed: Sep. 20, 2022. [Online]. Available: https://cloud.google.com/containers

[2] Yelp. *How Yelp Runs Millions of Tests Every Day*. Accessed: Sep. 20, 2022. [Online]. Available: https://engineeringblog.yelp.com/2017/04/how-yelp-runs-millions-of-tests-every-day.html

[3] Netflix Technical Blog. *Titus, the Netflix Container Management Platform, is Now Open Source*. Accessed: Sep. 20, 2022. [Online]. Available: , [Online]. Available: https://netflixtechblog.com/titus-the-netflix-container-management-platform-is-now-open-source-f868c9fb5436

[4] Tripwire. *State of Container Security Report*. Accessed: Sep. 20, 2022. [Online]. Available: https://www.tripwire.com/state-of-security/devops/organizations-container-security-incident

[5] Security Boulevard. *Stealing Infrastructure: Cryptomining Attacks on Container Environments*. Accessed: Sep. 20, 2022. [Online]. Available: https://securityboulevard.com/2018/03/stealing-infrastructure-cryptocurrency-mining-attacks-container-environments/

[6] CoreOS. *Clair*. Accessed: Sep. 20, 2022. [Online]. Available: https://coreos.com/clair/docs/latest

[7] Docker. *Docker Security Scanning*. Accessed: Oct. 16, 2021. [Online]. Available: https://docs.docker.com/v17.12/docker-cloud/builds/image-scan

[8] RedHat. *Atomic Scan—Container Vulnerability Detection*. Accessed: Sep. 20, 2022. [Online]. Available: https://github.com/projectatomic/atomic

[9] AppArmor. *AppArmor Project*. Accessed: Sep. 20, 2022. [Online]. Available: https://apparmor.net

[10] *Seccomp*. Accessed: Sep. 20, 2022. [Online]. Available: http://man7.org/linux/man-pages/man2/seccomp.2.html

[11] *SELinux Project*. Accessed: Sep. 20, 2022. [Online]. Available: http://selinuxproject.org/page/Main_Page

[12] *Microservice*. Accessed: Sep. 20, 2022. [Online]. Available: https://microservices.io/patterns/microservices.html

[13] Docker. *Content Trust in Docker*. Accessed: Sep. 20, 2022. [Online]. Available: https://docs.docker.com/engine/security/trust/content_trust

[14] Docker. *Docker Hub*. Accessed: Sep. 20, 2022. [Online]. Available: https://hub.docker.com

[15] H. Zhu and C. Gehrmann. "Lic-Sec: An enhanced AppArmor Docker security profile generator," *J. Inf. Secur. Appl.*, vol. 61, Sep. 2021, Art. no. 102924.

[16] Udica. *SELinux Policy Generation for Containers*. Accessed: Sep. 20, 2022. [Online]. Available: https://github.com/containers/udica

[17] L. Lei *et al.*, "Speaker: Split-phase execution of application containers," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, 2017, pp. 230–251.

[18] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated system call policy generation for container attack surface reduction," in *Proc. Int. Symp. Res. Attacks, Intrusions Defenses*, 2020, pp. 443–458.

[19] *Aqua Security*. Accessed: Sep. 20, 2022. [Online]. Available: https://www.aquasec.com

[20] *StackRox*. Accessed: Sep. 20, 2022. [Online]. Available: https://www.stackrox.com

[21] *TwistLock*. Accessed: Sep. 20, 2022. [Online]. Available: https://www.twistlock.com

[22] *Docker*. Accessed: Sep. 20, 2022. [Online]. Available: https://www.docker.com

[23] *Kubernetes*. Accessed: Sep. 20, 2022. [Online]. Available: https://kubernetes.io

[24] Docker. *Compose*. Accessed: Sep. 20, 2022. [Online]. Available: https://docs.docker.com/compose/networking

[25] *Netfilter and IPtables*. Accessed: Sep. 20, 2022. [Online]. Available: https://www.netfilter.org

[26] CoreOS. *Flannel*. Accessed: Sep. 20, 2022. [Online]. Available: https://coreos.com/flannel

[27] Weaveworks. *Weave Net*. Accessed: Sep. 20, 2022. [Online]. Available: https://www.weave.works/oss/net

[28] Tigera. *Project Calico*. Accessed: Sep. 20, 2022. [Online]. Available: https://www.projectcalico.org

[29] Cilium. *API-Aware Networking and Security*. Accessed: Sep. 20, 2022. [Online]. Available: https://cilium.io

[30] CiscoCloud. *HAProxy*. Accessed: Sep. 20, 2022. [Online]. Available: https://hub.docker.com/r/ciscocloud/haproxy-consul

[31] Mace. *OpenVPN*. Accessed: Sep. 20, 2022. [Online]. Available: https://hub.docker.com/r/mace/openvpn-as

[32] MemSQL. *MemSQL*. Accessed: Sep. 20, 2022. [Online]. Available: https://hub.docker.com/_/memsq

[33] O. vSwitch. *Production Quality, Multilayer Open Virtual Switch*. Accessed: Sep. 20, 2022. [Online]. Available: https://www.openvswitch.org

[34] A. Bettini. *Vulnerability Exploitation in Docker Container Environments*. Accessed: Sep. 20, 2022. [Online]. Available: https://www.blackhat.com/eu-15/briefings.html#vulnerability-exploitation-in-docker-container-environments

[35] *LunaSec*. Accessed: Sep. 20, 2022. [Online]. Available: https://www.lunasec.io/docs/blog/log4j-zero-day

[36] TwistLock. *A Busybox AutoCompletion Vulnerability*. Accessed: Oct. 16, 2021. [Online]. Available: https://www.twistlock.com/2017/11/20/cve-2017-16544-busybox-autocompletion-vulnerability

[37] StackRox. *Breaking Bad: Detecting Real World Container Exploits*. Accessed: Oct. 16, 2021. [Online]. Available: https://www.stackrox.com/post/2018/03/breaking-bad-detecting-real-world-container-exploits

[38] Synk. *Hacking Docker Containers by Exploiting Imagemagick Vulnerabilities*. Accessed: Sep. 20, 2022. [Online]. Available: https://snyk.io/blog/hacking-docker-containers-by-exploiting-base-image-vulnerabilities

[39] *Abusing Privileged and Unprivileged Linux Containers*, NCCGroup, London, U.K., 2016.

[40] TwistLock. *Escaping Docker Container Using Waitid*. Accessed: Oct. 16, 2021. [Online]. Available: https://www.twistlock.com/2017/12/27/escaping-docker-container-using-waitid-cve-2017-5123

[41] Z. Jian and L. Chen, "A defense method against Docker escape attack," in *Proc. Int. Conf. Cryptogr., Secur. Privacy*, 2017, pp. 142–146.

[42] Z. Qian, Z. M. Mao, and Y. Xie, "Collaborative TCP sequence number inference attack: How to crack sequence number under a second," in *Proc. Conf. Comput. Commun. Secur.*, 2012, pp. 593–604.

[43] Instana. *Stan Robot Shop, A Sample Microservice Application*. Accessed: Sep. 20, 2022. [Online]. Available: https://github.com/instana/robot-shop

[44] Weaveworks. *Sock Shop—A Microservices Demo Application*. Accessed: Sep. 20, 2022. [Online]. Available: https://microservices-demo.github.io

[45] Nginx. *Nginx*. Accessed: Sep. 20, 2022. [Online]. Available: https://hub.docker.com/_/nginx

[46] Redis. *Redis*. Accessed: Sep. 20, 2022. [Online]. Available: https://hub.docker.com/_/redis

[47] IO Visor Project. *Extended Berkeley Packet Filter*. Accessed: Sep. 20, 2022. [Online]. Available: https://www.iovisor.org/technology/ebpf

[48] T. Høiland-Jørgensen *et al.*, "The eXpress data path: Fast programmable packet processing in the operating system kernel," in *Proc. Int. Conf. Emerg. Netw. EXperiments Technol.*, 2018, pp. 54–66.

[49] IO Visor Project. *eXpress Data Path*. Accessed: Sep. 20, 2022. [Online]. Available: https://www.iovisor.org/technology/xdp

[50] M. Roesch, "Snort: Lightweight intrusion detection for networks," in *Proc. Large Installation Syst. Admin. Conf.*, 1999, pp. 229–238.

[51] Suricata. *Threat Detection Engine*. Accessed: Sep. 20, 2022. [Online]. Available: https://suricata.io

[52] Hewlett Packard Enterprise. *NetPerf: Network Performance Benchmark*. Accessed: Sep. 20, 2022. [Online]. Available: https://github.com/HewlettPackard/netperf

[53] iPerf. *Network Bandwidth Measurement Tool*. Accessed: Sep. 20, 2022. [Online]. Available: https://iperf.fr

[54] proofpoint. *Emerging Threat Ruleset*. Accessed: Sep. 20, 2022. [Online]. Available: https://www.proofpoint.com/us/threat-insight/et-pro-ruleset

[55] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs containerization to support PaaS," in *Proc. Int. Conf. Cloud Eng.*, 2014, pp. 610–614.

[56] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang, "ContainerLeaks: Emerging security threats of information leakages in container clouds," in *Proc. Int. Conf. Dependable Syst. Netw.*, 2017, pp. 237–248.

[57] A. A. Mohallel, J. M. Bass, and A. Dehghantaha, "Experimenting with Docker: Linux container and BaseOS attack surfaces," in *Proc. Int. Conf. Inf. Soc.*, 2016, pp. 17–21.

[58] R. Shu, X. Gu, and W. Enck, "A study of security vulnerabilities on Docker hub," in *Proc. Conf. Data Appl. Secur. Privacy*, 2017, pp. 269–280.

[59] B. Tak, C. Isci, S. Duri, N. Bila, S. Nadgowda, and J. Doran, "Understanding security implications of using containers in the cloud," in *Proc. Annu. Tech. Conf.*, 2017, pp. 313–319.

[60] B. Tak, H. Kim, S. Suneja, C. Isci, and P. Kudva, "Security analysis of container images using cloud analytics framework," in *Proc. Int. Conf. Web Services*, 2018, pp. 116–133.

[61] E. Bacis, S. Mutti, S. Capelli, and S. Paraboschi, "DockerPolicyModules: Mandatory access control for Docker containers," in *Proc. Conf. Commun. Netw. Secur.*, 2015, pp. 749–750.

[62] Y. Sun, D. Safford, M. Zohar, D. Pendarakis, Z. Gu, and T. Jaeger, "Security namespace: Making Linux security frameworks available to containers," in *Proc. Secur. Symp.*, 2018, pp. 1423–1439.

[63] S. Arnautov *et al.*, "SCONE: Secure Linux containers with Intel SGX," in *Proc. Symp. Operating Syst. Design Implement.*, 2016, pp. 689–703.

[64] F. Manco *et al.*, "My VM is lighter (and safer) than your container," in *Proc. Symp. Operating Syst. Princ.*, 2017, pp. 218–233.

[65] Z. Shen *et al.*, "X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers," in *Proc. Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2019, pp. 121–135.

[66] W. Zhang *et al.*, "OpenNetVM: A platform for high performance network service chains," in *Proc. Workshop Hot Topics Middleboxes Netw. Function Virtualization*, 2016, pp. 26–31.

[67] D. Zhuo *et al.*, "Slim: OS kernel support for a low-overhead container overlay network," in *Proc. Symp. Netw. Syst. Design Implement.*, 2019, pp. 331–344.

[68] T. Bui, "Analysis of Docker security," 2015, *arXiv:1501.02967*.

[69] T. Combe, A. Martin, and R. D. Pietro, "To Docker or not to Docker: A security perspective," *IEEE Cloud Comput.*, vol. 3, no. 5, pp. 54–62, Oct. 2016.

[70] J. Chelladhurai, P. R. Chelliah, and S. A. Kumar, "Securing Docker containers from denial of service (DoS) attacks," in *Proc. Int. Conf. Services Comput.*, 2016, pp. 856–859.

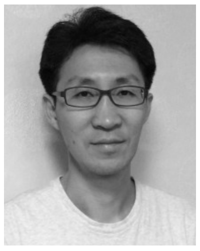
[71] Romana. *Romana V2.0*. Accessed: Oct. 16, 2021. [Online]. Available: https://romana.io

Jaehyun Nam received the B.S. degree in computer science and engineering from Sogang University, South Korea, and the M.S. and Ph.D. degrees in information security from the School of Computing, KAIST. He is an Assistant Professor with the Department of Computer Engineering, Dankook University, South Korea. His research interests include networked systems and security, and security issues in cloud and edge computing systems, including SDN, NFV, the IoT, and containers.

Seungsoo Lee received the B.S. degree in computer science from Soongsil University, South Korea, and the M.S. and Ph.D. degrees in information security from KAIST. He is an Assistant Professor with the Department of Computer Science and Engineering, Incheon National University. His research interests include network systems, network security, software-defined networking (SDN), network function virtualization (NFV), and cloud security.

Phillip Porras received the M.S. degree in computer science from the University of California, Santa Barbara, CA, USA, in 1992. He is a SRI Fellow and the Program Director of the Internet Security Group, Computer Science Laboratory, SRI International, Menlo Park, CA, USA. He has participated on numerous program committees and editorial boards, and participates on multiple commercial company technical advisory boards. He continues to publish and conduct technology development on numerous topics including intrusion detection and alarm correlation, privacy, malware analytics, active and software defined networks, and wireless security.

Vinod Yegneswaran received the A.B. degree from the University of California, Berkeley, CA, USA, in 2000, and the Ph.D. degree from the University of Wisconsin, Madison, WI, USA, in 2006, all in computer science. He is a Senior Computer Scientist with SRI International, Menlo Park, CA, USA, pursuing advanced research in network and systems security. His current research interests include SDN security, malware analysis, and anti-censorship technologies. He has served on several NSF panels and program committees of security and networking conferences, including the IEEE Security and Privacy Symposium.



Seungwon Shin (Member, IEEE) received the B.S. and M.S. degrees in electrical and computer engineering from KAIST and the Ph.D. degree in computer engineering from the Electrical and Computer Engineering Department, Texas A&M University. He is an Associate Professor with the School of Electrical Engineering, KAIST. His research interests include SDN security, the IoT security, botnet analysis/detection, darkweb analysis, and cyber threat intelligence (CTI).