BASTION: A Security Enforcement Network Stack for Container Networks

Jaehyun Nam, Seungsoo Lee, and Hyunmin Seo, *KAIST;* Phil Porras and Vinod Yegneswaran, *SRI International;* Seungwon Shin, *KAIST*

https://www.usenix.org/conference/atc20/presentation/nam

This paper is included in the Proceedings of the 2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

Open access to the Proceedings of the 2020 USENIX Annual Technical Conference is sponsored by USENIX.

BASTION: A Security Enforcement Network Stack for Container Networks

Jaehyun Nam[†]*, Seungsoo Lee*[†]*, Hyunmin Seo*[†]*, Phillip Porras*[‡]*,*
Vinod Yegneswaran[‡]*, and Seungwon Shin*[†]
KAIST, Daejeon, Korea[†]*, SRI International, CA, USA*[‡]

Abstract

In this work, we conduct a security analysis of container networks, identifying a number of concerns that arise from the exposure of unnecessary network operations by containerized applications and discuss their implications. We then present a new high-performance security enforcement network stack, called BASTION, which extends the container hosting platform with an intelligent container-aware communication sandbox. BASTION introduces (*i*) a *network visibility service* that provides fine-grained control over the visible network topology per container application, and (*ii*) a *traffic visibility service*, which securely isolates and forwards inter-container traffic in a point-to-point manner, preventing the exposure of this traffic to other peer containers. 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.

1 Introduction

Among the leading trends in virtualization is that of containerized application deployment at industrial scales across private and public cloud infrastructures. For example, Google has been a significant adopter of container-based software deployment using its container orchestrater, *Kubernetes* [26] to spawn more than two billion containers per week [17]. Yelp uses containers to migrate their code onto AWS, and launches more than one million containers per day [56]. Netflix spawns more than 3 million containers per week within Amazon EC2 using its Titus container management platform [25].

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. For example, Tripwire's container security report [50] 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. In recognition of such risks, several efforts [10, 14, 36] 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 [1], Seccomp [40], and SELinux [41] can provide much stronger isolation of containers by preventing various system resource abuses. In fact, several commercial products introduce container security frameworks [2, 44, 51], which can monitor containers at runtime and impose dynamic policy controls.

However, while there continues to emerge a variety of approaches to secure containerized applications, less attention has been paid to bounding these applications' access to the container network. Specifically, there has been significant adoption of containers as microservices [31], in which containers are used to create complex cloud and data-center services. Although current container platforms often utilize IP-based access control to restrict each container's network interactions, there are limitations in such controls that offer opportunities for significant container abuse.

This paper begins by discussing several of the challenges that arise from the current reliance on the host OS network stack and virtual networking features to provide robust container-network security policies. The paper will present five examples of inherent limitations that arise in using the Host OS network stack to manage the communications of container ecosystems as they are deployed today. Informed by these existing limitations, we introduce BASTION, a new extension to container network stack isolation and protection. BASTION instantiates a security network stack per container, offering isolation, performance efficiency, and a fine-grained network security policy specification that implements the least privileged network access for each container. This approach also provides better network policy scalability in network pol-

icy management as the number for hosted containers increases, and greater dynamic control of the container ecosystem as containers are dynamically instantiated and removed.

BASTION is composed of a manager and per-container network stacks. The manager solicits network and policy information from active containers, and deploys a security enforcement network stack into each container. Then, in the network stack, all security enforcement is conducted through two major security services: *a network visibility service* and *a traffic visibility service*. Based on a set of inter-container dependencies, the network visibility service mediates the container discovery process, filtering out any access to containers and hosts that are deemed irrelevant given the dependency map. The traffic visibility service controls network traffic between the container and other peer containers, while also verifying the sources of the traffic. This service enables traffic to flow among the containers through an efficient forwarding mechanism that also isolates network communications between senders and recipients. Whenever there is any change in container environments, the manager dynamically updates the network stack of each container with no service interruption.

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:

- A security assessment of container networks, illustrating security challenges that arise in current container network stacks and security mechanisms.
- The introduction of a novel security-enforcement network stack for containers, which restricts the network visibility of containers and isolates network traffic among peer containers with high performance.
- The presentation of the prototype system, BASTION, including an analysis of how it addresses network security challenges in current container environments.

2 Background and Motivation

Here, we provide the background of container networks and identify how the underlying architectural limitations of current network security services impact container environments.

2.1 Current Container Networks

Docker Platform: Docker [12] uses bridge networks to provide inter-container connectivity, by default. 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` [34].

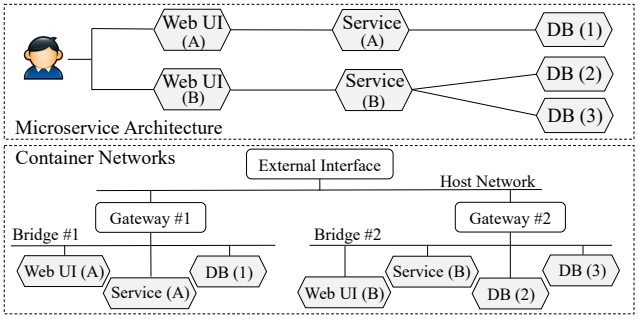


Figure 1: Overview of Docker Bridge Networking. Upper panel: a conceptual microservice architecture involving two independent services. Lower panel: separate bridged networks are instantiated to manage container network flows.

Kubernetes Orchestration System: Kubernetes [26] supports the management of large numbers of Docker containers across multiple nodes (e.g., physical *host* servers), enabling cross-host container applications to work as a logical unit. Thus, while Docker uses bridge networks for containers within the same host (node), Kubernetes uses various overlay networks (e.g., Flannel [11], Weave [55], Calico [49]) to provide inter-container connectivity across multiple nodes. For example, in the Weave overlay network [55], each node has a special bridge interface, called `weave`, to connect all local containers. The weave bridges, run at each node, 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 [49] similarly applies network and security policies using `iptables`. In the case that operators want further security enforcement, they may use Cilium [7], a security extension that conducts API-aware access control (e.g., HTTP method) by redirecting all network traffic to its containerized security service (envoy).

Network-privileged Containers: Besides the typical use of containers, there are special cases in which an operator wants to directly expose containerized services using the host IP address (e.g., HAProxy [8], OpenVPN [28], and MemSQL [30]). 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*.

2.2 Challenges in Container Networks

While current container platforms mostly utilize OS-level 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, which motivate the BASTION design.

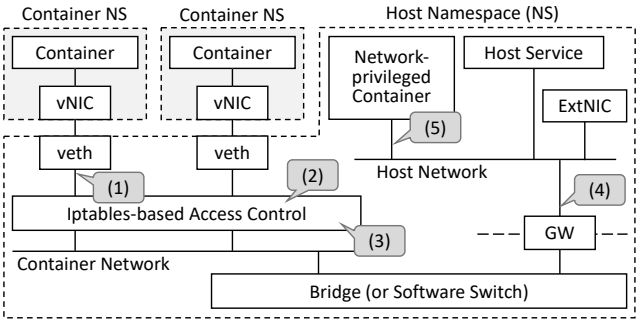


Figure 2: Five critical challenges in container networks: (1) Loss of container context, (2) Limitations of IP-based access controls, (3) Network policy explosion (performance degradation), (4) Unrestricted host access, and (5) No restriction on network-privileged containers.

(1) Loss of container context: As shown in Figure 2, each container has its own virtual interface, but this is only visible inside of the container. 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. Unfortunately, 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 network namespace, which means that the packet already flows into a container network. Hence, all decisions for further security inspection (e.g., source verification and network flow control) and packet forwarding should be solely made based on the packet header information, and a malicious container can directly forge packets on behalf of any other containers, allowing lateral attacks and traffic poisoning when any container is compromised.

(2) Limitations of IP-based access controls: The primary method for imposing network flow control among container platforms is through `iptables`, an IP-based access control provided by a Linux kernel. However, the IP addresses of containers can be dynamic, and adjustments are then required whenever containers are spun up and down. Thus, it can be a challenge to specify security policies for containers in terms of both performance and security, since these policies must be updated whenever containers are re-created, and the policy tables of `iptables` should be also locked during policy updates. Furthermore, although operators enforce various security policies, container networks are still vulnerable to layer-2 attacks, due to the limited scope of `iptables`.

(3) Network policy explosion: Finer grained network policies inherently require larger sets of network policies. Further, since each container may require different policies, the overall number of policies will tend to increase with the heterogeneity and size of the container ecosystem. Unfortunately, `iptables` is a centralized mechanism for all network interfaces in the host, which results in monolithic network rules that can be daunting to manage and at worst produce

```
root@container1-95986c685-qzkpw:~# route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric
default         10.40.0.0       0.0.0.0         UG    0
10.32.0.0       0.0.0.0         255.240.0.0     U     0
root@container1-95986c685-qzkpw:~# nmap -sS 10.40.0.0

PORT   STATE SERVICE
80/tcp open  http
MAC Address: 7A:5C:1D:87:4D:19 (Unknown)

root@container1-95986c685-qzkpw:~# wget -q 10.40.0.0 80
root@container1-95986c685-qzkpw:~# ls
index.html
root@container1-95986c685-qzkpw:~# cat index.html
Host web service
```

Figure 3: Host service access through the gateway IP address of a container network. A container scans and accesses the services running in the host without any restriction.

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state U
26: eth0@if27: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1
```

(a) Network interfaces visible by a general container

```
2: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pf
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdis
4: datapath: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1376 qdisc
6: weave: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1376 qdisc no
9: vethwe-datapath@vethwe-bridge: <BROADCAST,MULTICAST,UP,LOW
10: vethwe-bridge@vethwe-datapath: <BROADCAST,MULTICAST,UP,LO
11: vxlan-6784: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 65485 q
13: vethweplac1ab3a@if12: <BROADCAST,MULTICAST,UP,LOWER_UP,M-
```

(b) Network interfaces visible by a network-privileged container

Figure 4: Network visibility according to container privileges: upper panel - a general container sees only its own network interfaces, lower panel - a network-privileged container shares the network namespace with a host; thus, it can see all network interfaces in the host.

a network policy explosion (the number of security policies will rapidly increase as a large number of containers are deployed). Consequentially, if the number of security policies in `iptables` increases beyond hundreds, the container ecosystem may face a significant performance degradation [37].

(4) Unrestricted host access: Each container network has a gateway interface for external accesses, which is 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. In Kubernetes, containers can even access all other hosts (nodes) through the gateway IP addresses assigned to them. If a service running in a host opens a certain network port, as shown in Figure 3, a container can directly access the service through the gateway IP address. In the worst case, a malicious container can exploit the service in a manner that can subvert/harm the availability of the host.

(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 with respect to operational isolation. As shown in Figure 4, network-privileged containers can access not only the host network interfaces, but can also monitor all network traffic from deployed containers in the

Network Threats	Docker [12]	Flannel [11]	WeaveNet [55]	Calico [49]	Open vSwitch [27]	Cilium [7]	BASTION
L2 attack (e.g., ARP Spoofing)	✔	✔	✔	✘	▲	✘	✘
Traffic Eavesdropping	✔	✔	✔	✘	▲	✘	✘
L3/L4 attack (e.g., IP Spoofing)	✔	✔	✔	✔	▲	▲	✘
Host Service Access	✔	✔	✔	✘	▲	▲	✘
Host Network Namespace Abuse	✔	✔	✔	✔	✔	✔	✘

Table 1: Potential of network attacks 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 application of network security policies. Infeasible (✘): network attack is always blocked.

host 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 secure policy configuration for the containers by themselves.

2.3 Assumptions and Threat Model

Assumptions: Consider the case of containers connected to each other in order 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) to perform a remote hijacking of an Internet-accessible container application that is operating as a part of a microservice, using published container vulnerabilities [45,52]. For example, even certain images provided by the official Docker hub include known vulnerabilities [47]. Given this, 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 [33] and modifying Linux capabilities within a container [53], and defense techniques based on status inspection of namespaces [24]. 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) in this paper.

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`).

2.4 Limitations of Container Network Interface Plugins

Here, we briefly discuss the limitations of current container networking plugins. Table 1 presents the feasibility of network threats that abuse the above security challenges.

Docker, Flannel, WeaveNet: Docker [12], Flannel [11], WeaveNet [55] 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 2.2 and are vulnerable to all network threats presented in Table 1.

Calico: Calico [49] 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.). 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) [27] 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 and this could be used to prevent spoofing attacks. However, one critical concern is that OVS does not support a `NOT operation`, meaning 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 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 be able to 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 provides no protection in the case of network-privileged containers.

Cilium: Cilium [7] 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 provides support for a range of network policies (e.g., identity and label-based policies), which can block accesses to specific containers or hosts, the feasibil-

ity of such network threats depend on the operator and deployment considerations. For example, even though an operator carefully defines network policies to restrict service-to-service communications based on container identities, containers may still conduct L3/4-based lateral attacks to neighbors in the same service. Network-privileged containers are beyond its threat model, meaning that 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 Section2.2. Unlike existing container network interface plugins that rely on operator-defined network policies to protect containers from various network threats, BASTION automatically discovers inter-container dependencies from container platforms, and provides an intelligent container-aware communication sandbox that protects inter-container communications. In the following section, we will describe BASTION in greater detail.

3 BASTION Design

As we discussed in Section 2.2, many of the security limitation that arise from the use of the OS network stack to service container process are less well studied than other container security mechanisms. To address these limitations, we begin by identifying the design considerations that BASTION addresses, followed by a presentation of its design.

R1: *Container-aware least privilege communications enforcement.* A container's connectivity should be a function of the interdependencies between itself and those containers whose communications are required to compose a service.

R2: *Scalable and fine-grained network policy expression.* Network policy expression and enforcement performance within the container network should scale well to the dynamism and size of modern host container topologies.

R3: *Policy control over intra-container communications.* While the gateway interface plays as a key role in the communications with external networks, the network stack should filter out the direct access of the gateway interface to prevent the abuse of the host namespace.

R4: *Policy enforcement for network-privileged containers.* Network policy enforcement should be capable of fine-grained access control over network-privilege-enabled containers that share the host network namespace for the direct access of the host interfaces.

R5: *Unauthorized eavesdropping and spoofing prevention.* Communication mediation should prevent access to third-party packets (i.e., eavesdropping) and false packet production (i.e., preventing both ARP spoofing and traffic injection among local containers).

R6: *Competitive performance that scales well with any container topology.* The network stack should deliver low latency and high throughput communications while securing container networks.

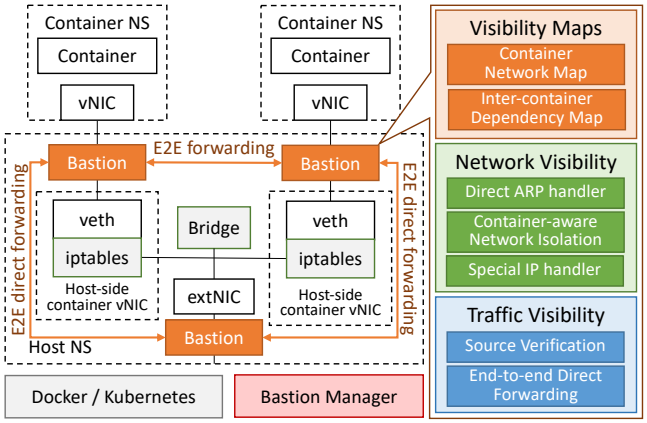


Figure 5: BASTION Architecture Overview. Orange box: BASTION network stack. Red box: manager that maintains the global view of container networks. Green box: network visibility service that restricts container reachability. Blue box: traffic visibility service that controls inter-container traffic while concealing irrelevant traffic from containers.

3.1 Architectural Overview

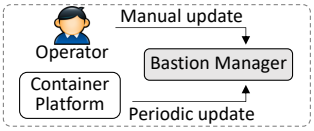
BASTION represents the opposite spectrum of prior container network stack designs, which implement network policy enforcement in a centralized manner. BASTION implements a decentralized, per-container, network stack. That is, all BASTION security enforcement occurs before a container's packets are delivered into the container network. This approach enables BASTION to provide individualized control over the network traffic coming from each container, mitigating the security challenges discussed in Section 2.2.

Figure 5 illustrates the overall architecture of BASTION. BASTION is composed of a manager, which maintains the global network view of all containers with their security dependencies, and per-container network stacks that include two security services (i.e., network and traffic visibility services). A BASTION network stack maintains the container network map for the corresponding container, which includes the network information of all reachable containers that have peer dependencies (e.g., microservice composition), and an inter-container dependency map, including the security policies on dependent containers only (R2).

When packets arrive at the BASTION network stack, the network visibility service proactively filters any discovery processes of irrelevant containers by dealing with ARP requests based on the container network map (R1, R5), and restricts the communications between containers according to security policies specified in the inter-container dependency map (R1). In addition, a special IP-handler restricts unauthorized access to special IP addresses (e.g., gateway IP addresses) (R3). The traffic visibility service conducts secure packet-forwarding between containers. This service first verifies the packets with the identity of the container (R4-5), directly passing packets

< Container Network Map >

ContainerID	Network	ContainerSet	Interface	IP address	MAC address
WebApp-X1	WebService	WebApp	vethweplf964e8	10.32.0.2	96:0e:73:ef:86:fe
WebApp-X2	WebService	WebApp	vethweplb89dc35	10.32.0.3	6e:81:0f:a7:db:c7
Service-Y1	WebService	Service	vethweplb957e84	10.32.0.4	D6:bc:7b:20:32:c5
Database-Z1	WebService	Database	vethweplc5ee33c	10.32.0.5	42:a0:ae:b7:f5:97



< Inter-container Dependency Map >

Source	Destination	Policy
WebApp	Service	Any
WebApp	Database	TCP:3306
Service	Database	TCP:3307

Figure 6: BASTION computes a container network map that captures the network interface attributes for each hosted container, and an inter-container dependency map that indicates the links and dependencies between containers.

from the source container to the destination containers using their interface information (R6). Since this direct packet forwarding occurs at the network interface level, packets are no longer passed through the container network (host-side), eliminating any chance for unauthorized network traffic exposure (even to network-privileged ones) (R4-5).

In terms of cross-host inter-container communications, a specialized BASTION network stack is utilized at the external interface of each node. It only maintains the container network map for all containers deployed in each node since all security decisions are already made at the network stack of each container. Thus, when it receives packets from other nodes, it simply conducts a secure forwarding from the external interface to destination containers. Overlay network composition among hosts (nodes) are beyond the coverage of BASTION; thus, it utilizes existing overlay networks (e.g., WeaveNet over IPSec). BASTION also retains the existing mechanisms of container platforms to handle inbound traffic from external networks.

3.2 BASTION Manager

The BASTION manager performs two primary roles. It collects the network information of all active containers from container platforms and manages the BASTION network stacks deployed to the active containers.

(1) Container Collection. The BASTION manager first maintains two hash maps (i.e., a global container network map and the inter-container dependency map for all containers) for the security evaluation of each container. As shown in Figure 6, BASTION uses a container platform to retrieve the network information for all containers, and to build the inter-container dependency map by extracting the dependencies among containers based on the retrieved information and their ingress/egress security policies. In addition, because containers can be dynamically spun up and down, the manager periodically retrieves containers' network information to update the maps. While a notification-based mechanism would provide greater efficiency, a polling-based mechanism was selected to provide a transparent and compatible solution that can be integrated with already-deployed container environments without any required modifications.

Algorithm 1 Extracting Inter-Container Dependencies

```
1:  Input: C, which is the set of all active containers
2:  for each container u ∈ C do
3:      for each container v ∈ C where u ≠ v do
4:          if v ∈ u.explicitDependents then
5:              p_uv = u.EgressPolicies ∩ v.IngressPolicies
6:              add v into u.dependencyMap with p_uv
7:          else if u.containerSet ≠ v.containerSet then
8:              for each service pair s ∈ S_set(v.ContainerSet) do
9:                  p_us = u.EgressPolicies ∩ s.Port
10:                 add s.IP into u.dependencyMap with p_us
11:     for each service pair s ∈ S_microservice do
12:         p_us = u.EgressPolicies ∩ s.Port
13:         add s.IP into u.dependencyMap with p_us
```

Extracting inter-container dependencies: BASTION automatically extracts dependencies among containers. To do this, a container network model is defined, in which a microservice is composed of one or more container sets, and each container set has one or more containers that play the same role (due to scaling and load-balancing). Each container set exposes internal service ports to communicate with other container sets, while a microservice exposes global service ports to redirect accesses from the outside world to some of the internal service ports. We then define four constraints for implicit dependencies in inter-container communications: (1) containers with the same container set are not granted interconnectivity, (2) containers in different container sets only communicate via internal service ports (explicitly exposed by configurations), (3) containers that are unrelated to each other may talk through global service ports, (4) all other communications are not allowed by default. Based on the container network model, all inter-container dependencies are extracted using Algorithm 1.

Discovering inter-container dependencies: As no container network can be made secure without proper network policies that restrict communications to the minimum required access, BASTION also discovers inter-container dependencies not explicitly defined by a container operator. During the flow control of inter-container traffic, BASTION produces network logs that capture the network accesses from/to containers. At the same time, it compares these logs with the inter-container dependency map, classifying them into three cases: *legitimate accesses, missing policies, and excessive policies*.

If the pair of observed containers are not in the precomputed inter-container dependency map, BASTION considers that there is either a missing network policy or an invalid access. Then, it informs an operator to review the specific flows to determine whether to produce a missing network policy. In addition, it identifies network 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. In these cases, BASTION informs an operator to review the specific policy that may require to be updated in the current configuration.

(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 interfaces with the container network and inter-container dependency maps. With respect to map size, each container only requires a part of the network information to communicate with dependent neighbors. Thus, to reduce the size of security services, BASTION filters irrelevant information per container. The manager also performs change detection of inter-container dependencies, automatically updating the maps in the network stacks of the corresponding containers.

3.3 Network Visibility Service

The network visibility service restricts unnecessary connectivity among containers and between containers and external hosts. To do this, the following three security components are introduced to handle container discovery, inter-container communications, gateway/service-IP accesses, respectively.

3.3.1 Direct ARP Handler

For inter-container networking, container discovery is the first step to identify other containers (communication targets). Containers use ARP requests to identify the necessary network information (i.e., MAC addresses) of target containers. 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 scan. Indeed, they offer no mechanism to control non-IP-based communications.

BASTION's direct ARP handler 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 in the container network map, and sends the reply back to the source container (while no ARP requests flow into the container network). If not, it drops the request.

3.3.2 Inter-container Communications Handler

Although the direct ARP handler 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, to further restrict the reachability of containers, a second component implements container-aware network isolation.

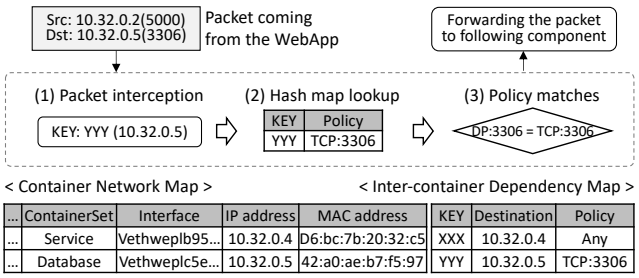


Figure 7: Workflow of container-aware network isolation. The WebApp container accesses a service of the Database container in a container network shown in Figure 6, and the container-aware network isolation in the WebApp's network stack inspects their dependency and security policies.

To illustrate how BASTION implements container-aware network isolation, we consider the example in Figure 6, which illustrates an interdependence between WebApp and Database containers. In mediating the WebApp's packets, as shown in Figure 7, BASTION first checks the dependency between the WebApp and the 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 WebApp has a dependency on the destination - in this case the Database. The connection is allowed if matched to the policy for the Database, otherwise it is dropped.

BASTION implements a per-container rule partitioning strategy, which simplifies rule conflict evaluation, as only the container-relevant rules are considered, at deployment and evaluation times. In addition, it offers a minimized policy enforcement performance impact, as the match set is container-specific rather than host-global (as occurs with `iptables`). This approach offers an inherent key advantage over host global network policy rule enforcement as the number of containers increases. A natural strategy for managing large sets of global (host layer) network security rules is to apply a global rule optimization algorithm (e.g., aggregating the rules into a reduced set). Unfortunately, as containers are dynamically spun up and down, particularly within large orchestrated container ecosystems, their corresponding security rules would also require frequent updating. In such situations, global rule optimization could prove less effective and even be a new performance bottleneck over our rule partitioning strategy.

3.3.3 Gateway and Service-IP Handler

In container environments, it is possible for a subverted container to exploit the gateway to probe services within the host OS. To address this concern, BASTION's gateway-IP handler 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 accesses by checking if both IP and MAC addresses be-

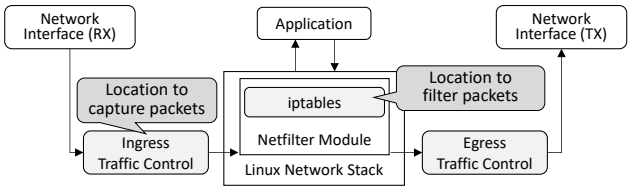


Figure 8: An illustration of the network packet processing sequence performed within the Linux kernel. While packets are filtered after delivered into the network stack, those can be still exposed during packet capture with nothing missed.

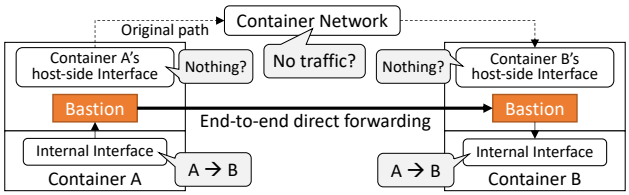


Figure 9: An illustration of how BASTION implements end-to-end direct packet forwarding to bypass exposure of intra-container traffic to other containers.

long to the gateway. It would be also possible that a network flow might access the gateways of other container networks, since these gateways are connected to the host network as well. Hence, the gateway-IP handler also filters unauthorized host accesses by comparing packets with the other gateways.

In Kubernetes environments, there is another special IP address, called a service IP address that is a virtual IP address used for the redirection to actual containers. Unfortunately, since service IP addresses do not belong to container networks, they can be simply considered as 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, all inter-container communications can be conducted with the existent IP addresses, and the other security components can process packets as intended.

3.4 Traffic Visibility Service

The traffic visibility service provides point-to-point integrity and confidentiality among container network flows. Here, we present how BASTION hides irrelevant traffic from containers using two security components: source verification and end-to-end direct forwarding.

3.4.1 Source Verification

To precisely track the actual source of inter-container traffic, BASTION leverages the kernel metadata of incoming packets (e.g., ingress network interface index). The BASTION network stack of each container statically contains the network information (i.e., IP/MAC addresses and the metadata of container-side and host-side interfaces) of the corresponding container, and BASTION verifies the incoming traffic by comparing not only the packet header information but also its metadata to the container's information embedded in the BASTION network stack. If either the packet header information or the metadata is not matched with the container network information, BASTION identifies the incoming traffic as spoofed and drops it. Furthermore, even though network-privileged containers can inject spoofed packets into other containers, BASTION will drop their spoofed packets since the packet metadata would not be matched with the container network information. As a result, BASTION can effectively eliminate the spectrum of disruption and spoofing threats.

3.4.2 End-to-end Direct Forwarding

Current network stacks cannot prevent the exposure of inter-container traffic from other containers as the filter position is behind the capture point, as illustrated in Figure 8. Thus, if a malicious container has a capability to 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 directly monitor the network traffic of others with no need to redirect the traffic.

To implement least-privilege traffic exposure, BASTION provides an end-to-end direct forwarding component. As shown in Figure 9, this component performs direct packet delivery between source and destination containers in the network interface level, bypassing not only their original network stacks (container-side) but also bridge interfaces (host-side); thus, it can prevent eavesdropping by peer containers. 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. If the destination is a container in the same node, BASTION directly injects the packet stream into the destination container. If the destination is a container in another node, BASTION injects the packet to the external interface of a host. 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. This traffic isolation prevents any traffic disclosure by other containers, preventing even network-privileged containers to view third-party traffic.

4 Implementation

We implement BASTION with 2.2K lines of C code and 5.1K lines of Python code on the Linux 4.16 kernel, which include two subsystems: a manager, and a network stack.

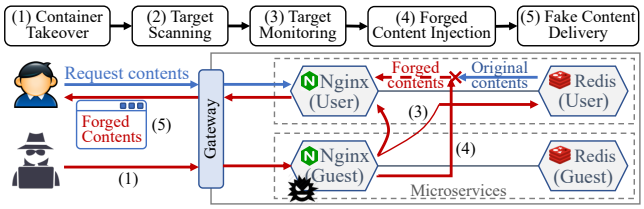


Figure 10: An example attack scenario within a Kubernetes environment. A compromised container from one service conducts a series of network attacks to hijack communications between other containers in a peer service.

BASTION Manager: For container collection, the manager periodically captures the attributes (e.g., NetworkSettings) of active containers from the Docker engine and the Kubernetes API server. Especially, in terms of explicit inter-container dependencies, it utilizes specific keywords (e.g., "*link*" and "*depends_on*"). In the case of Kubernetes, it does not have a way to explicitly define inter-container dependencies; thus, the manager utilizes `labels` to define explicit inter-container dependencies (e.g., "*dependencies: container A*"). In terms of network security policies, it extracts "*ingress*" and "*egress*" network security policies from `iptables` in a host and Kubernetes.

BASTION Network Stack: The security enforcement network stack for each container is implemented using eBPF [22] and XDP [19, 21], and the security services in the network stack inspect raw incoming packets in the `xdp_md` structure provided by XDP. During the inspection, they look up two hash maps (i.e., the container network and inter-container dependency maps), and these maps are synchronized with the maps in the corresponding management thread of the manager using BPF syscalls. Then, they use three types of XDP actions: 'XDP_TX' sends a packet back to the incoming container (the direct ARP handler), 'XDP_REDIRECT' injects a packet into the transmit queue of a destination (the end-to-end direct forwarding), and 'XDP_DROP' drops packets.

5 Security Evaluation

This section introduces a scenario that abuses the security holes in the current container network with real containers, and demonstrates how BASTION mitigates network attacks.

5.1 Scenario Validation

Figure 10 illustrates two independent services that are deployed along with common microservices [20, 54] in a Kubernetes environment. One is a service for legitimate users, and the other is a service for guest users. These services use Nginx [35] and Redis [38] container images retrieved from Docker Hub [13]. In this scenario, an attacker forges legitimate user requests, after infiltrating into the public-facing Nginx server by exploiting web application vulnerabilities.

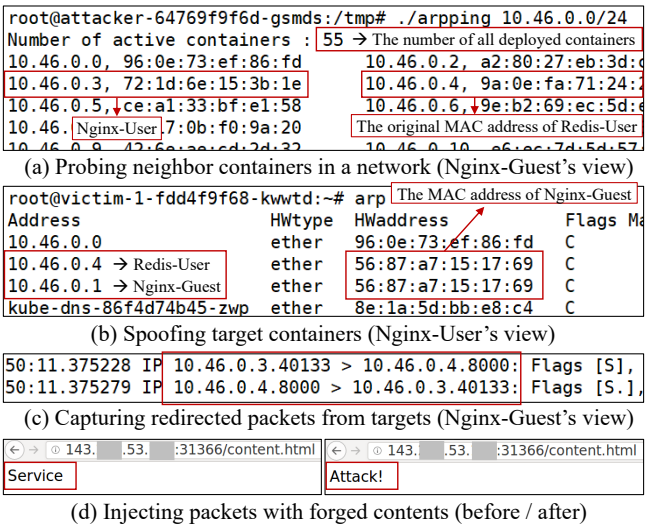


Figure 11: Screenshots demonstrating the attack scenario in a Kubernetes environment between two services.

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 11-(a). Then, the attacker injects fake ARP responses into the network to make all traffic between the Nginx-User and the Redis-User containers passes through the Nginx-Guest. As shown in Figure 11-(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 11-(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 back to the user instead of the original ones (Figure 11-(d)). In the end, the user receives forged contents as the attacker intended.

5.2 Effectiveness of Security Functions

Here, we focus on validating the effectiveness of BASTION against a range of network-oriented attacks. *For the following experiments, we disabled some of BASTION's security functions to show the before and after differences.*

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 12, BASTION's

```
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:c
root@attacker-64769f9f6d-gsmds:/tmp#
```

Figure 12: An illustration of neighbor container discovery. Our ARP handler 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)

Figure 13: 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 (only the reverse direction is intentionally filtered for illustration).

direct ARP handler and container-aware network isolation reduce the reachability of each container based on its container dependencies (R1). As a result, the infected container (i.e., Nginx-Guest in Figure 10) has only one dependent container (i.e., Redis-Guest in Figure 10), and BASTION ensures that the container observes only its gateway and that dependent.

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 13 illustrates the utility of BASTION's direct forwarding. The upper panel, Figure 13-(a), shows the visible network traffic of a target container (i.e., Nginx-User) after spoofing the container without direct forwarding. The lower panel, Figure 13-(b), demonstrates the use of direct forwarding. When direct forwarding is applied, the only visible traffic from a given interface is that of traffic involving the container itself (R4, R5). *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 fully apply end-to-end forwarding for all traffic, the attacker will see no traffic between them.

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

```
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)

Figure 14: 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.

from the attacker and victim perspectives. In the following example, we enable source verification only at the IP-level and allow an attacker to conduct ARP spoofing attacks.

Figure 14-A illustrates cases without source verification. Here, the attacker spoofs the Nginx-User 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 (R5).

6 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 an experimental testbed comprising of three machines to construct a Kubernetes environment with the Weave 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` [18] and `iperf` [23] were respectively used to measure round-trip latencies and TCP stream throughputs.

6.1 Network Stack Deployment Overhead

BASTION periodically retrieves the container information from container platforms (1 second in our case) and deploys the network stacks for newly detected containers.

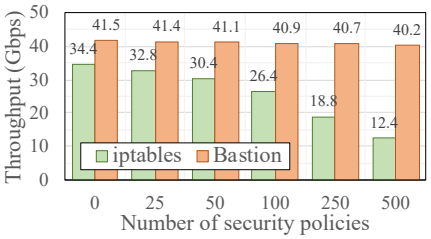


Figure 15: Inter-container throughput variations with the increasing number of security policies within a host.

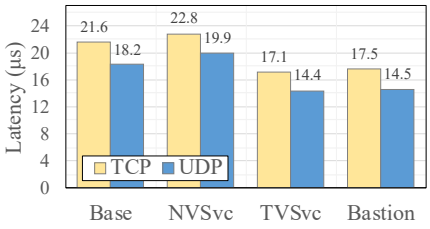


Figure 16: Inter-container latency measurements with different combinations of BASTION's services within a host.

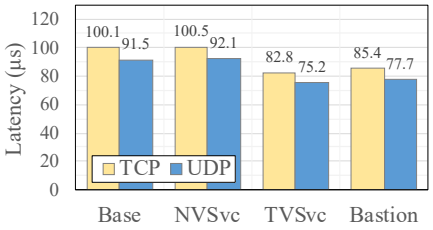


Figure 17: Inter-container latency measurements with different combinations of BASTION's services across hosts.

To see how long it takes to deploy a new network stack, we measured the deployment time while creating 100 containers. The result shows that it took 13.03 μs on average, meaning that BASTION's network stack would be deployed almost right after a new container is detected, while it can take a couple of seconds for containers to initialize their services (i.e., pulling container images from repositories, configuring container isolation, and starting services).

6.2 Security Policy Inspection Overhead

We compared the matching overheads with both `iptables`-based access control and BASTION, and Figure 15 shows the TCP throughputs with different numbers of security policies. For a fair comparison, we defined the same number of policies to each container for the overhead measurements of BASTION.

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` 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 15, 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) (R2). Such performance gains stem from BASTION's matching process optimized for containers, which comprises of a hash-based policy lookup for specific destinations and their port matches while there is no need to match source IP addresses and ports. Note that BASTION's performance gain with no security policy is because of the end-to-end direct forwarding that bypasses the host-side Linux network stack.

6.3 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 security services, which were 21.6μs and 18.2μs for TCP and UDP packets respectively. When we applied BASTION's network visibility service, the latencies slightly increased by 5.7% and 9.3% due to the newly applied security functions requiring additional packet processing to derive the reachability check between containers. When we applied BASTION's traffic visibility service, the overall latencies were noticeably improved by 26.3% because our secure forwarding 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 (R6). Table 2 also shows that the overall throughput of BASTION was improved by 20.6% compared to that of the base case within a host.

Throughput (Gbps)	Base	Network Visibility	Traffic Visibility	Bastion
Within a host	34.4	33.7	41.8	41.5
Across hosts	4.28	4.23	4.91	4.83

Table 2: Inter-container throughput measurements for the base case (no security) and BASTION's services.

6.4 Performance: Cross-Host Deployment

Next, we measured the latencies and throughput for cross-host container deployments. Figure 17 illustrates the measurement results with different combinations of BASTION's security services. Compared to the intra-host measurements, the overall latencies significantly increased due to 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 network-visibility service receded (less than 1%). Next, when we introduced BASTION's traffic-visibility service, the latencies were reduced by 21.3%

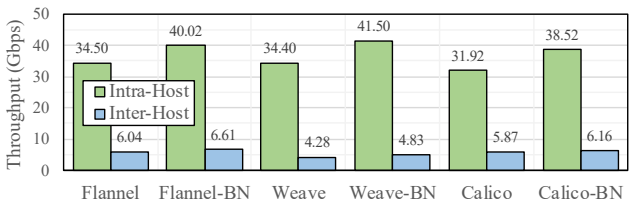


Figure 18: Throughput comparison with different types of container networks. BN = (when BASTION is deployed).

on average because our secure forwarding directly passed network packets from the source container to the destination container via the external interfaces. Finally, when we applied all security services, the latencies decreased by 17.7%, a significant improvement compared to the base case (R6). These latency improvements translated to a cross-host throughput improvement of 12.9%, as shown in Table 2.

6.5 Performance: Networking Plugins

Lastly, we compared the throughput variations in different types of container networks when BASTION is deployed. Figure 18 shows the TCP-stream throughputs between intra-host and inter-host containers in three container networks (i.e., Flannel, WeaveNet, and Calico). From the results, we see 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 through BASTION. In terms of the inter-host throughputs, we also see the performance improvements (9.4%, 12.9%, and 4.9% respectively) through BASTION. In sum, we ascertain the fact that BASTION can provide not only further network isolation and security enforcement but also better performance in various container networks.

7 Related Work

Container Security Analysis. There have been several efforts [15, 16, 24, 32, 33, 53] that have analyzed the security issues of container implementations. For example, Dua et al. [15] analyzed various container implementations, concluding that they are yet insecure from the perspective of the filesystem, network, and memory isolation. More specifically, Jian et al. [24] 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 [43, 45, 47, 48, 52] of container security focuses on container images. Shu et al. [43] and Tak et al. [47,48] 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. Rather, BASTION focuses on container networks.

Container Security and Isolation. Bacis et al. introduced DockerPolicyModules (DPM) [4] that allow Docker image maintainers to specify and ship SELinux policies within their images. Sun et al. [46] proposed security namespaces that enable containers to independently define security policies and apply them to a limited scope of processes. SCONE [3] presented a secure container mechanism for Docker containers by isolating them inside of SGX enclaves. LightVM [29] wraps containers in lightweight virtual machines (VMs). X-Containers [42] isolate containers that have the same concerns together on top of separate library OSes. These efforts are complementary to the network-focused objectives of BASTION, and could be combined to deliver security services that span both the system and networking services.

Container Network Security. Most container network solutions [57, 58] 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 [5], Comb et al. [9] and Chelladhurai et al. [6] analyzed Docker container security, finding that Docker is vulnerable to ARP spoofing and MAC flooding attacks in default settings. Our work extends these results by identifying 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 [39,49,55] have adopted `iptables`, Cilium [7] provides its own API-aware security mechanisms for L3/4/7 policies. As we discussed previously, while Cilium pursues API-level network security filtering to define and enforce both network- and application-layer security policies, BASTION fundamentally redesigns a network stack per container to construct an inherently secure container networking system.

8 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 network stack, and addressed these challenges by presenting BASTION, an intelligent communication sandbox for securing container-network communications, using Linux kernel features. In this work, we raise awareness of several security problems that lie within today's container networks, and offer security services for halting these problems in real-world container deployments.

Acknowledgement

We would like to thank all the anonymous reviewers of the program committee for their valuable insights on the paper. This work was partially funded by the National Science Foundation (NSF) under Grant No. CNS-1514503.

References

[1] AppArmor. https://gitlab.com/apparmor.

[2] Aqua. https://www.aquasec.com.

[3] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the Symposium on Operating Systems Design and Implementation*. USENIX, 2016.

[4] Enrico Bacis, Simone Mutti, Steven Capelli, and Stefano Paraboschi. DockerPolicyModules: Mandatory Access Control for Docker Containers. In *Proceedings of the Conference on Communications and Network Security*. IEEE, 2015.

[5] Thanh Bui. Analysis of Docker Security. *arXiv preprint arXiv:1501.02967*, 2015.

[6] Jeeva Chelladhurai, Pethuru Raj Chelliah, and Sathish Alampalayam Kumar. Securing Docker Containers from Denial of Service (DoS) Attacks. In *Proceedings of the International Conference on Services Computing*. IEEE, 2016.

[7] Cilium. API-aware Networking and Security. https://cilium.io.

[8] CiscoCloud. HAProxy Docker Container. https://hub.docker.com/r/ciscocloud/haproxy-consul.

[9] Theo Combe, Antony Martin, and Roberto Di Pietro. To Docker or Not to Docker: A Security Perspective. *Proceedings of the Cloud Computing*, 2016.

[10] CoreOS. Clair. https://coreos.com/clair/docs/latest.

[11] CoreOS. Flannel. https://coreos.com/flannel.

[12] Docker. https://www.docker.com.

[13] Docker. Docker Hub. https://hub.docker.com.

[14] Docker. Docker Security Scanning. https://docs.docker.com/v17.12/docker-cloud/builds/image-scan.

[15] Rajdeep Dua, A Reddy Raja, and Dharmesh Kakadia. Virtualization vs Containerization to support PaaS. In *Proceedings of the International Conference on Cloud Engineering*. IEEE, 2014.

[16] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds. In *Proceedings of the International Conference on Dependable Systems and Networks*. IEEE, 2017.

[17] Google. Containers at Google. https://cloud.google.com/containers.

[18] Hewlett Packard Enterprise. Netperf. https://github.com/HewlettPackard/netperf.

[19] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The eXpress data path: fast programmable packet processing in the operating system kernel. In *Proceedings of the International Conference on emerging Networking EXperiments and Technologies*. ACM, 2018.

[20] Instana. Stan's Robot Shop - A Sample Microservice Application. https://www.instana.com/blog/stans-robot-shop-sample-microservice-application.

[21] IO Visor Project. eXpress Data Path. https://www.iovisor.org/technology/xdp.

[22] IO Visor Project. extended Berkeley Packet Filter. https://www.iovisor.org/technology/ebpf.

[23] iPerf. Network bandwidth measurement tool. https://iperf.fr.

[24] Zhiqiang Jian and Long Chen. A Defense Method against Docker Escape Attack. In *Proceedings of the International Conference on Cryptography, Security and Privacy*. ACM, 2017.

[25] Amit Joshi, Andrew Leung, Corin Dwyer, Fabio Kung, Sargun Dhillon, Tomasz Bak, Andrew Spyker, and Tim Bozarth. Titus, the Netflix container management platform, is now open source. *Netflix Technology Blog*, 2018.

[26] Kubernetes. https://kubernetes.io.

[27] Linux Foundation cOLLABORATIVE Project. Open vSwitch. https://www.openvswitch.org.

[28] Mace. Docker OpenVPN container. https://hub.docker.com/r/mace/openvpn-as.

[29] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) than your Container. In *Proceedings of the Symposium on Operating Systems Principles*. ACM, 2017.

[30] MemSQL. Docker MemSQL container. https://hub.docker.com/_/memsq.

[31] Microservice Architecture. https://microservices.io/patterns/microservices.html.

[32] Amr A Mohallel, Julian M Bass, and Ali Dehghantaha. Experimenting with Docker: Linux Container and BaseOS Attack Surfaces. In *Proceedings of the International Conference on Information Society*. IEEE, 2016.

[33] NCCGroup. Abusing Privileged and Unprivileged Linux Containers. https://www.nccgroup.trust/uk/our-research/abusing-privileged-and-unprivileged-linux-containers.

[34] Netfilter and IPtables. https://www.netfilter.org.

[35] Nginx. Nginx Docker Container. https://hub.docker.com/_/nginx.

[36] RedHat. Atomic Scan - Container Vulnerability Detection. https://developers.redhat.com/blog/2016/05/02/introducing-atomic-scan-container-vulnerability-detection.

[37] RedHat. Benchmarking nftables. https://developers.redhat.com/blog/2017/04/11/benchmarking-nftables.

[38] Redis. Redis Docker Container. https://hub.docker.com/_/redis.

[39] Romana. Romana v2.0. https://romana.io.

[40] Seccomp sandbox. http://man7.org/linux/man-pages/man2/seccomp.2.html.

[41] SELinux Project. http://selinuxproject.org/page/Main_Page.

[42] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019.

[43] Rui Shu, Xiaohui Gu, and William Enck. A Study of Security Vulnerabilities on Docker Hub. In *Proceedings of the Conference on Data and Application Security and Privacy*. ACM, 2017.

[44] StackRox. https://www.stackrox.com.

[45] StackRox. Breaking Bad: Detecting real world container exploits. https://www.stackrox.com/post/2018/03/breaking-bad-detecting-real-world-container-exploits.

[46] Yuqiong Sun, David Safford, Mimi Zohar, Dimitrios Pendarakis, Zhongshu Gu, and Trent Jaeger. Security Namespace: Making Linux Security Frameworks Available to Containers. In *Proceedings of the Security Symposium*. USENIX, 2018.

[47] Byungchul Tak, Canturk Isci, Sastry Duri, Nilton Bila, Shripad Nadgowda, and James Doran. Understanding Security Implications of Using Containers in the Cloud. In *Proceedings of the Annual Technical Conference*. USENIX, 2017.

[48] Byungchul Tak, Hyekyung Kim, Sahil Suneja, Canturk Isci, and Prabhakar Kudva. Security Analysis of Container Images Using Cloud Analytics Framework. In *International Conference on Web Services*. Springer, 2018.

[49] Tigera. Project Calico. https://www.projectcalico.org.

[50] Tripwire. State of Container Security Report. https://www.tripwire.com/state-of-security/devops/organizations-container-security-incident.

[51] TwistLock. https://www.twistlock.com.

[52] TwistLock. A Busybox autocompletion vulnerability. https://www.twistlock.com/2017/11/20/cve-2017-16544-busybox-autocompletion-vulnerability.

[53] TwistLock. Escaping Docker container using waitid. https://www.twistlock.com/2017/12/27/escaping-docker-container-using-waitid-cve-2017-5123.

[54] Weaveworks. Sock Shop - A Microservices Demo Application. https://microservices-demo.github.io.

[55] Weaveworks. Weave Net. https://www.weave.works/oss/net.

[56] Yelp. How Yelp Runs Millions of Tests Every Day. https://engineeringblog.yelp.com/2017/04/how-yelp-runs-millions-of-tests-every-day.html.

[57] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, KK Ramakrishnan, and Timothy Wood. OpenNetVM: A platform for high performance network service chains. In *Proceedings of the workshop on Hot topics in Middleboxes and Network Function Virtualization*. ACM, 2016.

[58] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. Slim: OS Kernel Support for a Low-Overhead Container Overlay Network. In *Proceedings in the Symposium on Networked Systems Design and Implementation*. USENIX, 2019.