Attacks are Forwarded: Breaking the Isolation of MicroVM-based Containers Through Operation Forwarding

Jietao Xiao and Nanzi Yang, *State Key Lab of ISN, School of Cyber Engineering, Xidian University, China;* Wenbo Shen, *Zhejiang University, China;* Jinku Li and Xin Guo, *State Key Lab of ISN, School of Cyber Engineering, Xidian University, China;* Zhiqiang Dong and Fei Xie, *Tencent Security Yunding Lab, China;* Jianfeng Ma, *State Key Lab of ISN, School of Cyber Engineering, Xidian University, China*

https://www.usenix.org/conference/usenixsecurity23/presentation/xiao-jietao

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

Attacks are Forwarded: Breaking the Isolation of MicroVM-based Containers Through Operation Forwarding

Jietao Xiao [1,*], Nanzi Yang[1,*], Wenbo Shen[2,+], Jinku Li[1,+], Xin Guo[1], Zhiqiang Dong[3], Fei Xie[3], and Jianfeng Ma[1]

[1]*State Key Lab of ISN, School of Cyber Engineering, Xidian University, China*
[2]*Zhejiang University, China*
[3]*Tencent Security Yunding Lab, China*

Abstract

People proposed to use virtualization techniques to reinforce the isolation between containers. In the design, each container runs inside a lightweight virtual machine (called microVM). MicroVM-based containers benefit from both the security of microVM and the high efficiency of the container, and thus are widely used on the public cloud.

However, in this paper, we demonstrate a new attack surface that can be exploited to break the isolation of the microVM-based container, called *operation forwarding attacks*. Our key observation is that certain operations of the microVM-based container are forwarded to host system calls and host kernel functions. The attacker can leverage the operation forwarding to exploit the host kernel's vulnerabilities and exhaust host resources. To fully understand the security risk of operation forwarding attacks, we divide the components of the microVM-based container into three layers according to their functionalities and present corresponding attacking strategies to exploit the operation forwarding of each layer. Moreover, we design eight attacks against Kata Containers and Firecracker-based containers and conduct experiments on the local environment, AWS, and Alibaba Cloud. Our results show that the attacker can trigger potential privilege escalation, downgrade 93.4% IO performance and 75.0% CPU performance of the victim container, and even crash the host. We further give security suggestions for mitigating these attacks.

1 Introduction

Containers have been widely adopted in cloud computing due to their high resource utilization efficiency. According to Cloud Native Survey 2021 from the Cloud Native Computing Foundation, 93% of respondents are currently using or planning to use containers in their productions, which has increased by 300% since 2016 [1]. Nowadays, container techniques have become the foundation of cloud computing. All leading cloud computing vendors provide container-based cloud computing services, such as Azure Container Instances and AWS Elastic Container Service.

Unfortunately, the high efficiency of containers also comes with a price. The shared kernel design cannot provide strong isolation between containers, leading to information leaks [2], out-of-band workloads breaking control groups [3], and abstract resource attacks [4]. Even worse, the attacker can exploit kernel bugs to compromise the shared kernel to attack all containers on the same kernel [5]. To mitigate security risks introduced by the shared kernel, prior works have proposed microVM-based containers, which run each container inside a lightweight virtual machine (called microVM).

The microVM-based container provides a dedicated guest kernel for the container and uses hardware virtualization to enforce strong isolation between containers. Therefore, it benefits from both the security of microVM and the high efficiency of the container, and thus it is regarded as a secure alternative to the original shared-kernel container. Therefore, leading cloud computing vendors have developed multiple microVM-based container techniques, such as Firecracker-based containers from Amazon AWS [6] and Kata Containers from Alibaba Cloud [7]. Both of them have been used heavily on the public cloud. For example, the Firecracker-based container is the runtime for AWS serverless service, including AWS Fargate [8] and Lambda [9].

Similar to traditional VMs, the microVM-based container uses a hypervisor to create and manage a microVM. On top of the VM, it adds container runtime components to create containers inside the microVM. Most system call needs of the container are served by the guest kernel of the microVM. The microVM-based container and the host machine environment can be regarded as two isolated worlds. However, we observe that certain operations of the microVM-based container are forwarded to the host kernel (termed *operation forwarding*) due to performance and functionality requirements. Unfortunately, existing research works [10–15] on the

* Co-first authors.
+ Co-corresponding authors.

security of VMs focus on vulnerabilities, side-channels, and covert-channels, ignoring the operation forwarding.

However, in this paper, we demonstrate that *the operation forwarding can be exploited to break the isolation of microVM-based containers*. We find that an attacker can leverage the operation forwarding of the microVM-based container to trigger specific host system calls and kernel functions. Next, the attacker uses these system calls and kernel functions as entries to exploit the host kernel's vulnerabilities and exhaust host resources. We term these attacks *operation forwarding attacks*.

To systematically explore the attack surface, we divide the components of the microVM-based container into three layers based on their functionalities: container runtime components, the device emulator, and host kernel components. We design three attacking strategies targeting all these three layers. To demonstrate the feasibility and practicality of these attacking strategies, we leverage a semi-automatic approach to identify attack vectors following the strategies. We further design eight attacks targeting the Firecracker-based container and Kata Containers to evaluate the impact and demonstrate the security risks of these attacks. Contributions of this work are summarized as follows.

New attack surface. We demonstrate a new attack surface of the microVM-based container called *operation forwarding attack*. Attackers can exploit the operation forwarding of microVM-based container components to trigger the host kernel's vulnerabilities and exhaust host resources, causing potential privilege escalation and DoS attacks.

Attacking strategies. We propose attacking strategies targeting all three layers of microVM-based containers, including container runtime components, the device emulator, and host kernel components. We demonstrate that the attacker can launch operation forwarding attacks through all three layers to break the isolation of the microVM-based container.

Practical attacks. We design eight attacks against Kata Containers and Firecracker-based containers and conduct experiments on the local testbed, Alibaba Cloud, and AWS. All these environments are vulnerable to our attacks. More specifically, these attacks lead to privilege escalation and cause 93.4% IO and 75.0% CPU downgrades, and 60.0% packet loss. We have reported all issues to the related security teams. They have confirmed these issues and assigned a new CVE (CVE-2022-0358) [16] to us.

2 Background

In this section, we describe the necessary background knowledge of our work, including the microVM-based container architecture and its application scenarios.

MicroVM-based container architecture. A microVM-based container runs a native container instance inside a dedicated lightweight virtual machine, which excludes unnecessary devices and guest-facing functionalities of the virtual machine to reduce the performance overhead and attack surface. As shown in Figure 1, based on the functionalities, the architecture of a microVM-based container can be divided into three layers: (1) container runtime components, (2) device emulator, and (3) host kernel components. Specifically, the container runtime components are introduced to launch and manage the microVMs and the native containers running inside the microVMs. The device emulator is a user-space program on the host that simulates various hardware devices. The host kernel components are loadable kernel modules in the host to provide virtualization functionality support. For example, Kernel-based Virtual Machine (KVM) [17] allows the host kernel to function as a hypervisor, and the vhost-net [18] module offloads the network device from the device emulator to a kernel module. Note that the microVM of a microVM-based container mainly leverages virtio [19] framework for IO virtualization, which adopts a front-end driver in the guest kernel. The front-end driver transports the IO requests from the containers to the back-end in the container runtime components and device emulator layer via virtqueues.

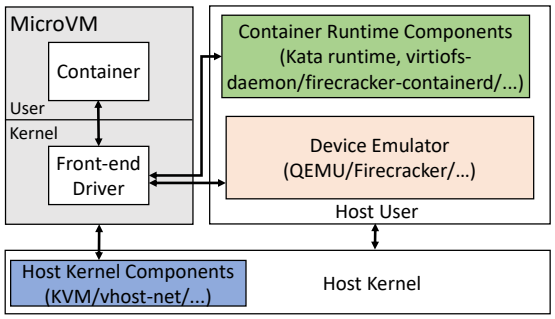


Figure 1: MicroVM-based container architecture.

Accordingly, there are two popular microVM-based containers, i.e., Kata Containers [7] and Firecracker-based container [20]. In the container runtime components layer, Kata runtime creates and manages the microVMs and isolated containers for Kata Containers, and firecracker-containerd [21] plays the same role in the Firecracker-based container. Besides, virtiofs daemon is a unique container runtime component of Kata Containers to support virtiofs in order to share containers' rootfs and volumes between the host and guest systems. As for the device emulator layer, QEMU is the default device emulator of Kata Containers, and Firecracker is naturally the device emulator of Firecracker-based container. And both Kata Containers and Firecacker-based container use KVM as the hypervisor, while only Kata Containers introduces vhost-net module for networking in the host kernel components layer.

Application scenarios. To enforce isolation and security, cloud vendors use different microVM-based containers to contain each tenant's workload. For instance, AWS (Amazon Web Services) uses the Firecracker microVM to isolate workloads in multi-tenant environments [22]. Specifically, in the AWS serverless environment, such as Lambda [9] and

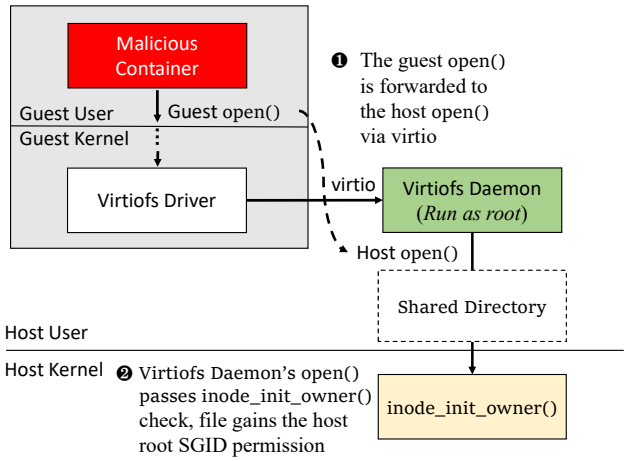


Figure 2: A real attack example in Kata Containers. Kata Container forwards the guest `open()` system call to the virtiofs daemon, which triggers host `open()` system call.

```
1  void inode_init_owner(...)
2  {
3    ...
4    if (dir && dir->i_mode & S_ISGID) {
5      ...
6      if (S_ISDIR(mode))
7        ...
8      else if ((mode & (S_ISGID | S_IXGRP)) == (S_ISGID |
       ↪ S_IXGRP) &&
9          !in_group_p(i_gid_into_mnt(mnt_userns, dir)) &&
10         !capable_wrt_inode_uidgid(mnt_userns, dir,
           ↪ CAP_FSETID))
11       mode &= ~S_ISGID;
12   } else
13     ...
14   ...
15 }
```

Figure 3: Linux kernel source of `inode_init_owner()`. It checks (line 9) and cleans the file's SGID (line 11) if the creation process does not have the same supplemental group as the directory owner.

Fargate [8], all the native containers of one tenant are running in a dedicated Firecracker microVM and managed by firecracker-containerd [23]. In contrast, Alibaba cloud uses Kata Containers to provide a multi-tenant serverless environment called ECI (Elastic Container Instance) [24]. In ECI, one tenant's workloads are confined in a separate Kata container instance, which uses QEMU as the device emulator by default. Namely, cloud vendors leverage microVM-based containers to secure different tenants' workloads in a multi-tenant environment.

Ideally, as each microVM-based container is isolated in a single virtual machine, an attack via such a container is unlikely, if not impossible, to have a security impact on the host or other containers. However, in this paper, we demonstrate that the isolation of microVM-based containers has security risks in all of its three layers, i.e., the container runtime components, the device emulator, and the host kernel components. A malicious cloud user can break the isolation of microVM-based containers, which leads to potential privilege escalation, DoS attacks, and even host crashes.

3 Motivation and Attacking Strategies

In this section, we first present the threat model and assumptions of our work, then we describe the motivation of this paper and use a real example to illustrate the details. After that, we introduce three strategies to systematically explore the attacks in each layer according to the architecture of microVM-based container (as shown in Figure 1).

3.1 Threat Model and Assumptions

The attacker aims to break the isolation of the microVM-based containers and disrupt other containers running on the same host, as the microVM-based containers are widely used by cloud vendors to provide multi-tenant container services [8, 24]. We assume that the attacker is able to create one or more microVM-based containers through a web or command-line console. The attacker can also interact with the container runtime through the console. Moreover, the attacker can run any programs within the created microVM-based containers. This assumption is reasonable as all popular cloud vendors provide cloud users with the console to allow them to control the life-cycle of containers. In addition, cloud vendors allow users to upload and execute their own programs within their containers.

On the restriction side, the attacker's microVM-based containers are limited by the most restricted security practices. There are two layers of enforcement to restrict the microVM-based container. The first layer is that the host kernel uses all popular features to isolate these microVMs. More specifically, the host kernel leverages state-of-the-art hardware virtualization (e.g., KVM with the support of Intel CPU VT-x [25]) to isolate the microVMs. At the same time, the host kernel uses seccomp [26] to block the sensitive host system calls raised by the microVMs, and it leverages control groups (cgroups) [27] to limit the microVM's resource utilization. The second layer is that the guest kernel enforces as many namespaces [28] and control groups as possible to restrict the container inside the microVM. We further assume that the host and guest kernels have no known vulnerabilities and that all security mechanisms work properly.

In this paper, we show that even with all the mentioned isolation and protection enforced, a malicious user can still attack the host or other victim containers.

3.2 Motivating Example

The microVM-based container forwards operations to the host kernel and uses the host kernel functionalities to serve its requirements. However, the operation forwarding introduces new attack surfaces that allow the attacker to break

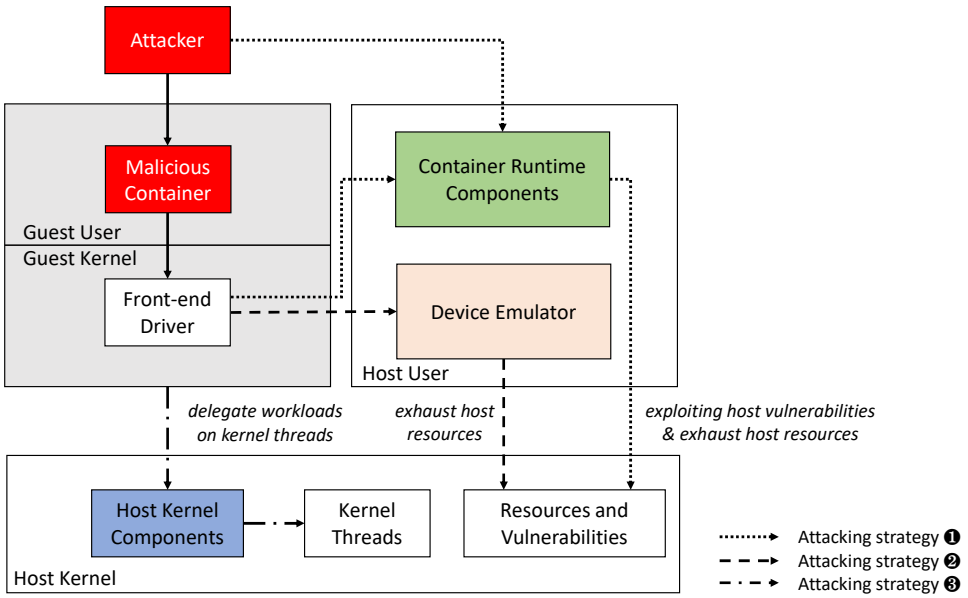


Figure 4: The overview of attacking strategies to break the isolation of microVM-based containers.

the microVM isolation. In the following, we use a real attack, which is discovered by us and confirmed by the Kata developers (with a new CVE assigned), to illustrate the attack surface.

Kata Containers leverages virtiofs to share directories and files (i.e., container rootfs image, container volume) between the host system and the container. The pass-through design of virtiofs daemon leads to an attack. To share directories between microVMs and host, virtiofs forwards the guest system calls to a host user space program (i.e., virtiofs daemon), and leverages it to perform the actual operations on the shared directories. In this example, we notice that the virtiofs forwards guest `open()` to the virtiofs daemon's `open()` while the virtiofs daemon itself has improper permission. Thus, the malicious container can leverage the virtiofs daemon's permission to bypass the check of the host kernel and create a file with host root SGID, allowing privilege escalation on the host machine.

Specifically, if a shared directory belongs to a regular user and host root group, and the SGID (Set Group ID) permission [29] bit is set for it, a malicious container can create an executable file with an unintended host SGID permission bit in the directory. Note that if the SGID permission bit is set for a directory, any files created inside the directory will get the same group ownership as the directory. As the file's SGID permission allows the file to be executed as the group that owns the file, thus a regular user can get the host root group privileges when executing the file created by the malicious container.

To launch an attack, a malicious container requests to create a new file with SGID permission bit in the shared directory, which is achieved by making an `open()` system call with `S_ISGID` [30] flag. The virtiofs driver in the guest kernel intercepts the `open()` system call and forwards it to the virtiofs daemon, which requests the host `open()` system call to serve its functionalities (❶ in Figure 2). After that, the file creation request enters the host kernel and invokes the `inode_init_owner()` function to initialize the inode of the creating file. As shown in Figure 3, the `inode_init_owner()` function determines if the SGID permission bit of the creating file needs to be cleared. To do that, it first checks if the directory has the SGID permission bit (line 4), which will be naturally passed in this case. Then it checks if the creation process has the same supplemental group as the directory owner. However, according to the virtiofs daemon's document [31], virtiofs daemon must run as the root user and has the root group in its supplemental group. As a result, the check is passed, and the file successfully keeps the SGID permission which should have been cleared.

Note that in the above process, an *operation forwarding* of a system call from guest `open()` to host `open()` happened to accomplish the task of file creation. Our key observation in this paper is that operation forwarding is possible to introduce (or forward) attacks if it is not properly used, just as shown in the real example of Figure 2.

We reported this bug to the virtiofs developer group. They confirmed the issue and fixed it by dropping the root group from the virtiofs daemons's groups list. After further evaluation by the Red Hat product team, a new CVE-2022-0358 [16] has been assigned to us.

3.3 Attacking Strategy Overview

We design three attacking strategies targeting container runtime components, device emulator, and host kernel components to launch operation forwarding attacks respectively, as

shown in Figure 4. The key idea of these strategies is leveraging the operation forwarding to make each layer's components improperly call host system calls and kernel functions. First, we show that an attacker or an attacker-controlled container can trigger the container runtime components to make host system calls to exploit host kernel's vulnerabilities and exhaust host resources (❶ in Figure 4) in §3.4. Second, we show that the malicious container can force the device emulator to invoke host system calls to exhaust host resources (❷ in Figure 4) in §3.5. Third, we demonstrate that the malicious container can forward its operations to host kernel functions running by kernel threads through host kernel components, which consumes extra CPU resources (❸ in Figure 4) in §3.6.

3.4 Attacking Container Runtime Components

The first strategy is to exploit the container runtime components to break the isolation of the microVM-based container (❶ in Figure 4). Note that the container runtime components forward guest system calls or container management requests to host system calls without rate limitations and additional security checks. A malicious container can exploit this operation forwarding at a high rate to exhaust host resources. Besides, an attacker can leverage the operation forwarding to exploit the host kernel's vulnerabilities and escalate privileges.

There are two paths that can be exploited by the attackers to launch operation forwarding attack (❶ in Figure 4). First, the attacker-controlled malicious container can make guest system calls. Then the system call requests are forwarded to the container runtime components' host system calls by the front-end driver in the guest kernel. Second, the attacker can send container management requests to the container runtime components directly via the manage console, to achieve the same goal.

By exploiting the above operation forwarding, the attackers can launch attacks to exhaust host resources. For example, the container's `write()` is forwarded to the host `write()`, which can exhaust the host dirty memory. We present more details in §4.3.1. Even worse, these components need host privileges to perform operations, and thus both of them run as the root user. As a result, the attacker can trigger host system calls with the root permission to escape the microVM-based container, as we will demonstrate in the firecracker-containerd escalation in §4.4.1.

3.5 Attacking Device Emulator

The second strategy is to exploit the device emulator (❷ in Figure 4). A malicious container can leverage operation forwarding to trigger specific guest system calls repeatedly, which consumes massive host kernel resources and causes DoS attacks.

The device emulator is a user-space program that emulates device behaviors for the microVM. To emulate the devices, it makes host system calls and leverages the host kernel to access the actual physical device. As shown in Figure 4, when the container makes system calls in the guest kernel, the guest kernel's front-end driver sends the request to the back-end device in the device emulator. After that, the device emulator invokes host system calls to serve its functionalities.

As a result, a malicious container can exhaust the host resources by repeatedly triggering the operation forwarding of device emulators. For example, the virtio block (a.k.a virtio-blk) presents a virtual hard disk for the Firecracker microVMs. Writing a large file directly to the guest's disk triggers the virtio-blk device to write the disk image file on the host, which occupies the host's dirty memory. Similarly, the virtio-net device utilizes the host kernel network stack to forward the container packets to the destinations, which can fill up the host connection tracking table. We present more details in §4.4.2 and §4.4.3.

3.6 Attacking Host Kernel Components

The third strategy is to exploit the host kernel components (❸ in Figure 4). A malicious container can use the operation forwarding to generate out-of-band workloads on the host kernel services, which consume extra host resources and cause DoS attacks.

Unlike the previous two layers, the host kernel components are kernel modules in the host kernel space, they forward the container's operations to host kernel functions. Some of these kernel functions run in the context of the kernel threads, which are scheduled to physical CPUs and consume the CPU resources. However, these kernel threads are attached to the host root cgroups. As a result, the resources consumed by these kernel threads cannot be limited by the microVM-based containers.

Specifically, KVM and vhost-net kernel modules create several kernel threads to emulate hardware devices asynchronously. For example, KVM creates `kvm-pit` kernel thread to inject the Programmable Interval Timer (PIT) [32] interrupts to the guest system. The vhost-net kernel module creates a `vhost` kernel thread to perform the network device emulation. These kernel threads will wake up to run kernel functions and consume considerable CPU resources. In the meantime, both of them are attached to the root cgroups. Thus, a microVM-based container can delegate workload on host kernel threads, consume extra host resources and break the isolation of microVM-based container.

4 Operation Forwarding Attack: Practical Case Studies

In this section, we first present our approach of identifying attack vectors. Then we use real cases to validate the identi-

Table 1: Summary of all the attacks. "K" denotes the Kata Containers environment, "F" denotes the Firecracker-based container environment, and "Container#" denotes the number of malicious containers required to achieve the effect of the attacks described in "Impact".

Case	Strategy	Trigger Func	Host Func	Container#	Impact
Virtiofs daemon escalation	❶	open()	open()	1	Create executable file with unintended SGID bit
Firecracker-containerd escalation	❶	CreateVM()	chown() & creat()	1	Change host directory's owner; empty host files; host crash
Dirty memory attack (K)	❶	write()	write()	1-2	Downgrade victim 93.4% IO performance;
Dirty memory attack (F)	❷	write()	write()	1-2	Downgrade victim 86.7% IO performance
Nf_conntrack table attack (K)	❸	connect()	tap_sendmsg()	10	55.0% of victim's network packets lost;Nginx connection timeout
Nf_conntrack table attack (F)	❷	connect()	sendmsg()	10	60.0% of victim's network packets lost;Nginx connection timeout
Vhost-net attack	❸	sendmsg()/recvmsg()	handle_tx()/handl_rx()	2	Consume 1x more resources
KVM PIT timer attack	❸	outb()	pit_do_work()	10	Downgrade victim 75.0% CPU performance

fied attack vectors and give out detailed steps for attacking microVM-based containers. The results show that our attacking strategies are practical and have severe consequences.

4.1 Identifying Attack Vectors

Our approach. We design a semi-automatic approach to identify attack vectors following strategies in §3. Our approach consists of two steps. First, we combine both dynamic and static analysis to identify operation forwarding paths from the container to the host machine. For the dynamic analysis, we run Linux Test Project (LTP) [33] test cases in the container to trigger guest system calls. In the meantime, we use `strace` command to trace container runtime components (e.g., virtiofs daemon) and device emulator processes. These traces allow us to establish the forwarding mappings between guest system calls and host system calls originating from device emulator or container runtime components processes. Moreover, the attacker may send container management requests to the container runtime components directly via gRPC. Therefore, we develop a scripting tool to statically analyze the source code of container components to identify host system calls in the gRPC implementation and build the operation forwarding paths from container management requests to host system calls. Second, we manually go through all operation forwarding paths to identify ones that miss permission checks or rate limitations. We term these forwarding paths as *sensitive paths*. To validate the practicality of exploiting sensitive paths, we then manually design attacks to trigger host vulnerabilities or exhaust host resources via these paths.

Let's use the virtiofs example in §3.2 to illustrate the above steps. We first run LTP test cases to trigger guest system calls and use `strace` for tracing the paths. As a result, we identify 25 operation forwarding paths starting from virtiofs. Second, we go through these forwarding paths and identify a sensitive path that the virtiofs daemon triggers the host system call `open()` without dropping the host root group from its supplemental group. Therefore, we follow this sensitive path and bypass the check of the host kernel to create a file with host root SGID via virtiofs daemon, which leads to privilege escalation on the host machine. Using the same methodology, we further identify seven other attacks in all three layers of the microVM-based containers, as discussed in §4.3 and §4.4.

Related techniques. Researchers have proposed multiple hypervisor fuzzing techniques to automatically uncover virtualization-related bugs [34–36]. Techniques in those works can be used to trigger paths from the container to the host machine. However, these fuzzers focus more on memory vulnerabilities such as use-after-free, stack (or heap) overflow, and segment faults, rather than operation forwarding paths between the VM and the host machine. Moreover, to identify meaningful forwarding paths, the fuzzer needs extra feedback like resource utilization and permission changes besides the code coverage to motivate program mutations, which is hard to implement.

The microVM-based container and the host machine environment can also be regarded as two isolated worlds. Therefore, sanitization vulnerabilities between SGX enclaves and the host machine [37] may also exist in the microVM-based container scenario. However, our paper focuses on a special problem that is caused by the operation forwarding, rather than the general security issues due to missing sanitization.

4.2 Experiment Setup and Result Summary

Ethical considerations. We conduct the experiments on both the local testbed and the cloud-vendor platforms. For the cloud-vendor platforms, note that our attacks may potentially affect other tenants on the same host. Therefore, we choose to use dedicated bare-metal servers of the AWS and Alibaba Cloud for experiments, which are only used by us and not shared with other tenants.

Environment setup. We set up microVM-based container environments on both the local and the cloud-vendor platforms. On the local environment, the test machine has the Intel Core i5 CPU, with 8 GB memory and 256GB SSD, and it runs deepin 20.2.2 Desktop with Linux Kernel v5.17.0. We deploy both Firecracker-based containers and Kata Containers on the local testbed. For the cloud-vendor platforms, we deploy a Firecracker-based container on an AWS m5zn.metal bare metal server with 48vCPU and 192GB memory, and it uses Amazon Linux 2 with Linux Kernel v5.10 to emulate the AWS multi-tenant environment. To represent the Alibaba Cloud multi-tenant environment, we deploy Kata Containers on an Alibaba Cloud ecs.ebmc6me.16xlarge bare metal server with 60vCPU and 192GB memory, which uses Ubuntu 20.04 LTS with Linux Kernel v5.11.

Furthermore, we follow the firecracker-containerd's doc-

umentation[1] to deploy Firecracker-based container with firecracker-containerd v1.6.3 and Firecracker v1.1.0 on both the local and the AWS bare-metal server. Note that firecracker-containerd is an incomplete open-source project which lacks necessary how-to documents. We modify its source code to activate some of its features. For example, we follow the volume test [38] to enable the container volume. As for Kata Containers, we deploy v2.4.0-alpha0 with QEMU v6.1.0 based on the guidance [39] on both the local and the Alibaba Cloud bare-metal server.

Result summary. We launch attacks and conduct case studies to validate the identified attack vectors. The overall summary of all case studies is shown in Table 1, which includes the case name (Case), the corresponding attacking strategy (Strategy), the trigger function of the operation forwarding (Trigger Func), the host function of the operation forwarding (Host Func), the number of malicious containers to launch attacks (Container#), and the corresponding impact of the attack case (Impact).

For the Kata Containers environment, besides the virtiofs daemon escalation attack in §3.2, we design three more attacks. 1) dirty memory attack. The attacker exploits the `write()` operation forwarding by virtiofs daemon (strategy ❶) to exhaust dirty memory on the host, downgrading the victim's IO performance by 93.4%. 2) nf_conntrack table attack. The malicious Kata container makes `connect()` to trigger vhost-net kernel module (strategy ❸) to call `tap_sendmsg()` to fill up the host's nf_conntrack table, which causes the victim container to drop 55.0% of network packets and fail to establish connections to typical apps (e.g., Nginx server). 3) vhost-net attack. The guest `sendmsg()` or `recvmsg()` is forwarded to host `handle_tx()` or `handle_rx()` by vhost-net (strategy ❸), which can be exploited by a malicious Kata container to consuming 1x more resources.

For the Firecracker-based container, we design four new attacks. 1) firecracker-containerd escalation. An attacker sends crafted `CreateVM()` request to the firecracker-containerd (strategy ❶). Then the firecracker-containerd invokes host `chmod()` and `creat()`, which is able to change any host directories' owner or empty any files on the host. 2) firecracker-based container dirty memory attack. A malicious Firecracker-based container leverages the virtio-blk backend device (strategy ❷) to forward guest `write()` to host `write()` to take up host dirty memory, which is able to downgrade the victim container's IO performance by 86.7%. 3) nf_conntrack table attack. A malicious container makes `connect()` to force the virtio-net backend device (strategy ❷) call `sendmsg()`, which fills up the host's nf_conntrack table and leads to 60.0% of the victim's packets loss, making the Nginx service unavailable. 4) KVM PIT timer attack (strategy ❸). A malicious Firecracker-based container uses `outb()` to make KVM call `pit_do_work()` to inject a large amount of timer interrupts into the guest system, which eventually downgrades the victim's CPU performance by 75.0%.

[1] https://github.com/firecracker-microvm/firecracker-containerd/blob/main/docs/getting-started.md

4.3 Attacks on Kata Containers

On the Kata Containers environment, we leverage three new case studies to show how to leverage the attacking strategies in §3.3 to break the isolation of Kata Containers. We conduct the experiments on the local environment and Alibaba Cloud successfully.

4.3.1 Kata Containers Dirty Memory Attack

The first case is exploiting the host's dirty memory. A malicious Kata container can leverage the virtiofs daemons to forward the `write()` system call from the guest to the host (strategy ❶ in Figure 4). One malicious container can generate a large amount of dirty memory in the host kernel by repeatedly triggering the `write()` system call to write files in the shared directory. As a result, the victim container's IO performance dramatically downgrades by 93.4%.

Root cause analysis. The size of dirty memory in the host kernel affects the file reading and writing performance. The Linux kernel introduces `dirty_backgroud_ratio` and `dirty_ratio` as the threshold of the whole kernel-space dirty memory. Whenever the size of dirty memory reaches the `dirty_backgroud_ratio`, the kernel wakes up the flusher threads to synchronize the dirty memory to the disk. Furthermore, if the size of dirty memory keeps increasing to the `dirty_ratio`, all processes' write mode will be converted from write-back to write-through. Note that the write-back mode only permits to write the file data on the memory, while the write-through mode needs to wait for the time-consuming disk IO operations to write to the disk. The write performance will dramatically downgrade if the write mode is changed from write-back to write-through.

A malicious Kata container can impact the size of the kernel's dirty memory. As shown in Figure 5, when the container calls guest `write()` to write a file in the shared directory, the write request is processed by the guest kernel's VFS (Virtual File System). Then the virtiofs driver intercepts the request and transfers the metadata of this write request to the virtiofs daemon in the host user space via virtio. Once the virtiofs daemon in the host receives the request, it triggers the host `write()` system call and writes the files in the shared directory, which increases the size of the dirty memory in the host kernel. Thus, if the attacker repeatedly generates and writes files in the container, it is possible to occupy all the dirty memory and reach the `dirty_ratio` threshold in the host kernel.

Experiments. In one malicious container, we leverage the `dd if=/dev/zero of=/mnt/test bs=1M count=4096 oflag=direct` command to create and write files. In a victim container, we run the same command to measure the IO performance before and after the attack. After that, we increase

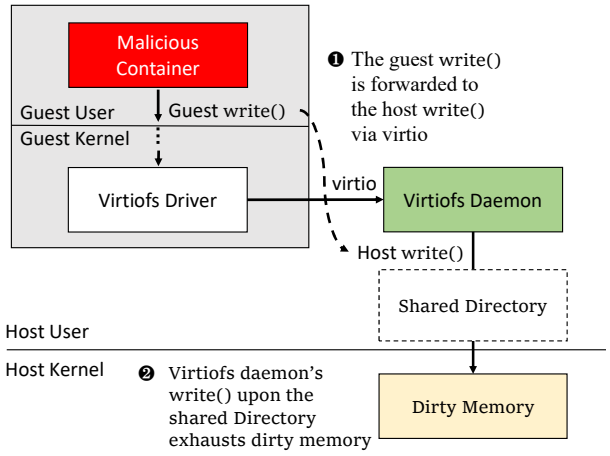


Figure 5: Operation forwarding of dirty memory attack in Kata Containers. Kata Container calls `write()` make the virtiofs daemon call host `write()` to exhaust host's dirty memory.

the number of attack containers and evaluate the relationship between the cost of the attack (i.e., the number of attackers) and the corresponding downgrades.

In our local environment, we launch attacks with 1, 5, and 10 malicious containers, respectively. As a result, the victim's downgrades of the `dd` command are 93.4%, 96.7%, and 98.2%, respectively. On the Alibaba Cloud, the machine has 192GB memory, and the `dirty_ratio` threshold is set to 20%. It needs at least two malicious containers to occupy the dirty memory and reach the `dirty_ratio` threshold. We launch attacks with 2, 5, and 10 malicious containers, and the victim's downgrades are 94.6%, 97.1%, and 98.5%, respectively.

4.3.2 Kata Containers Nf_conntrack Table Attack

The second case is the Kata Containers nf_conntrack table attack, which involves strategy ❸. In this case, the `connect()` system calls invoked by the malicious container are forwarded to the vhost-net module, which triggers `tap_sendmsg()` function in the host kernel. This function sends frames to the host kernel network stack, which will add an item in the host nf_conntrack table. Thus, an attacker can leverage this operation forwarding to fill up the host kernel's nf_conntrack table, which will eventually cause the victim container to lose 55.0% of its network packets.

Root cause analysis. It will cause random packets to drop on the host system if the host nf_conntrack table is filled up. Note that connection tracking [40] is a part of netfilter module in the Linux kernel, which maintains the state information about a connection in a memory table called nf_conntrack table. The Linux kernel introduces `nf_conntrack_max` to indicate the max number of the entries in the nf_conntrack table. If the number of entries reaches the `nf_conntrack_max` threshold, the received packets will be dropped by the netfilter module, and a new connection cannot be established successfully.

A malicious container can take up an entry of host nf_conntrack table by making `connect()` system call to establish TCP connections with other containers, as shown in Figure 6. The network packets sent by the `connect()` are encapsulated into Ethernet frames and dispatched to the virtio-net driver. Then the driver transfers the frames from the guest to the host system via virtio. After that, the host kernel's vhost-net module receives the frames from the driver, and it invokes `tun_sendmsg()` to send the frames to the tap device [41]. Finally, the frames are forwarded by the host Container Network Interface (CNI) virtual bridge [42] to other containers, which triggers the host kernel's `nf_conntrack_alloc()` function and adds an item in the host kernel's nf_conntrack table. Therefore, a malicious container is able to take advantage of this forwarding mechanism to fill up the nf_conntrack table in the host kernel, which will make the host drop packets randomly.

Experiments. We use 10 malicious Kata containers to launch attacks locally and on Alibaba Cloud. They make many TCP short connections between each other. As a result, the `vhost-net` host kernel module fills up the nf_conntrack table. Furthermore, we evaluate the attack's impact by using two sets of experiments. First, we run `ping` command in a victim container to measure the packet loss rate due to the attack, and we increase the number of malicious containers to evaluate the changes of packet loss rate. Second, we run an Nginx server in a victim container and use `ab` benchmarking tools [43] in another container to measure the performance downgrade of the Nginx server. We also increase the number of malicious containers to evaluate the relationship between the cost of the attacks and the Nginx server's performance downgrades.

Once the nf_conntrack table is filled up, it starts to drop packets randomly. Specifically, for the set of `ping` experiment, we launch attacks with 10, 15, and 20 containers, and it causes 55.0%, 57.0%, and 61.0% packet loss on the local environment, and 45.0%, 49.0%, and 50.0% packet loss on Alibaba Cloud, respectively. While for the set of Nginx experiment, once the nf_conntrack table is filled up, the `ab` command fails to establish connections to the Nginx server and it returns "`the timeout specified has expired`". The `ab` command keeps timing out regardless of the number of containers' growth in two environments.

4.3.3 Attack of the Vhost-net Kernel Module

The third case is to exploit the vhost-net kernel module. In this case, a malicious container's `sendmsg()` and `recvmsg()` system calls are forwarded to the vhost-net kernel module, which triggers the `handle_rx()` and `handle_tx()` kernel functions. Accordingly, a vhost worker thread is waked up by the vhost-net kernel module to handle the above kernel functions, which transmits and receives network packets. This process generates out-of-band workloads accounted to the host root cgroups. By generating a large network throughput, a mali-

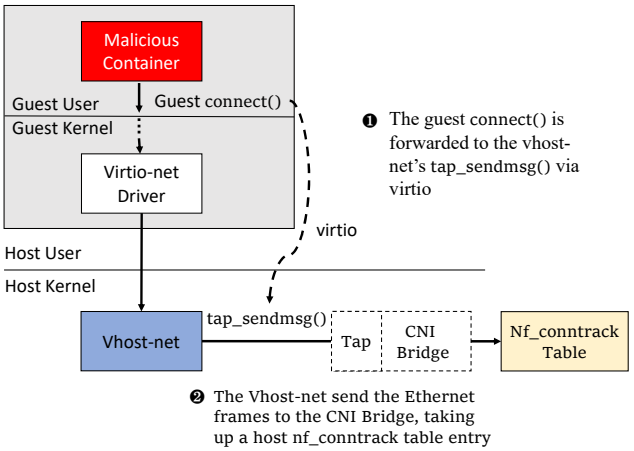


Figure 6: Operation forwarding of nf_conntrack table attack in Kata Container. Kata Container calls `connect()` make the vhost-net call `tap_sendmsg()` to forward frames to host CNI bridge, filling up host nf_conntrack table.

```
1  func (j *runcJailer) prepareBindMounts(...) error {
2      for _, m := range mounts {
3          ...
4          if stat.Mode&system call.S_IFMT == system
           ↪  call.S_IFREG {
5              err := j.bindMountFileToJail(m.HostPath,
               ↪  filepath.Join(j.RootPath(), m.HostPath))
6              ...
7          }
8      }
9      ...
10 }
11
12 func (j *runcJailer) bindMountFileToJail(src, dst string)
   ↪  error {
13     ...
14     err = os.Chown(filepath.Dir(dst), int(j.Config.UID),
       ↪  int(j.Config.GID))
15     ...
16     f, err := os.Create(dst)
17     ...
18 }
```

Figure 7: Source code of `prepareBindMounts()`. In this function, the RuncJailer joins the host path to RuncJailer's root path without a check (line 5), changes the directory's owner to RuncJailer's user (line 14), and creates an empty file (line 16).

cious Kata container is able to force the vhost worker thread in the host kernel to forward network packets, consuming 1x more resources (strategy ❸ in Figure 4).

Root cause analysis. The vhost-net worker thread created by the vhost-net kernel module consumes CPU resources to process network packets forwarding. As a kernel module, the vhost-net is introduced to improve the container's networking performance by reducing the number of system calls involved in virtio networking. It can directly leverage host kernel services to serve its functionalities by calling kernel functions instead of system calls. Specifically, a worker thread called `vhost-<owner-device-emulator-process-pid>` is created for each virtual machine. The worker thread is mainly responsible for running `handle_rx()` and `handle_tx()` kernel functions. The two kernel functions transfer network packets between the virtio-net driver and the vhost-net kernel module, which consumes CPU resources under heavy networking traffic. In the meantime, the vhost worker thread is attached to the host root cgroups. Thus the workloads will not be accounted to the microVM itself.

Experiments. We create two malicious Kata containers and use `iperf` tools to generate a large number of network throughput, and we transfer large network packets between them. Finally, each container generates workloads on two separate `vhost` kernel threads, and each kernel thread consumes 1x more CPU resources. The results are the same for the local and the cloud environments.

4.4 Attacks on Firecracker-based Container

On the Firecracker-based container environment, we leverage four case studies to show how to use the strategies in §3 to break the isolation of the Firecracker-based container. We conduct the experiments on both the local and AWS bare metal environments.

4.4.1 Firecracker-containerd Escalation

The first case of the Firecracker-based container exploits the firecracker-containerd to achieve privilege escalation and host crash. Specifically, the container management requests sent by a malicious user can trigger the firecracker-containerd to make host `chown()` and `creat()` system calls, which prepare the volume directory before starting the Firecracker microVM. Due to careless checking for the parameters of the above two system calls, the attacker can customize a crafted volume path in a container creation request, tricking the firecracker-containerd invoke these system calls with improper parameters. In addition, the firecracker-containerd runs as root user, which has two consequences. First, the firecracker-containerd changes the owner of the file's parent directory to a specific user (e.g., a regular user in the host). In other words, an attacker can change any host directory's owner, which introduces privilege escalation. Second, the file can be truncated to length 0. Thus, by emptying the host's essential system files like `ld.so`, the host cannot run any applications anymore due to the dynamic linker crash.

Root cause analysis. Due to the introduction of runc jailer [2] firecracker-containerd needs to invoke some system calls to prepare for the startup of the microVMs. Particularly, the runc jailer starts the Firecracker process in a runc container running in the host user space, which uses container isolation and limitation mechanisms to jail the Firecracker process. With the runc jailer enabled, the firecracker-containerd first creates a directory on the host as the root directory of the runc jailer. Then, the firecracker-containerd copies the Firecracker binary, guest kernel binary, microVM's root file system image, con-

[2] https://github.com/firecracker-microvm/firecracker-containerd/pull/249

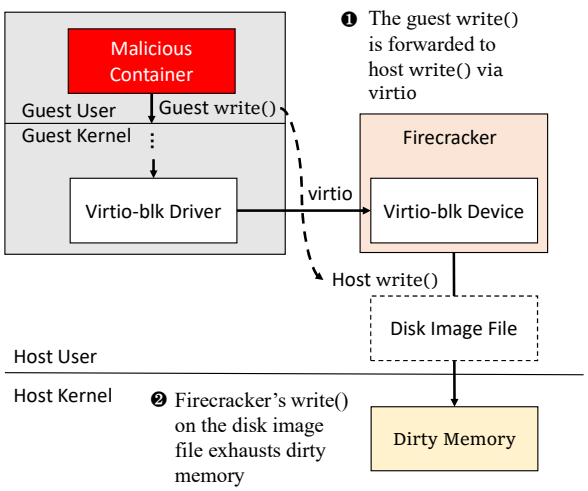


Figure 8: Operation forwarding of dirty memory attack in Firecracker-based container. Firecracker-based container calls `write()` make the Firecracker call host `write()` to exhaust host dirty memory.

tainer's rootfs image, and container volumes into the created directory, which ensures that the Firecracker process in the jailer can access them. Finally, the firecracker-containerd starts a runc jailer and executes the Firecracker binary in the jailer.

When a user creates a Firecracker-based container, he/she can specify and run the runc jailer as a host regular user. Besides, the user can specify the volume configuration. As shown in Figure 7, the `prepareBindMounts()` function copies the file and its parent directory into runc jailer's directory which is specified by the host volume path. First, the `prepareBindMounts()` function joins the jailer's root directory and volume's host path to a joined path using `filepath.Join()` (line 5). Then, in the `bindMountFileToJail()` function, the `filepath.Dir()` gets the joined path's parent directory, and the `os.Chown()` changes the directory's owner to the runc jailer's user (line 14), which is specified to a host regular user. Finally, `os.Create()` creates an empty duplicated files under the jailer's root directory (line 16). It is important to point out that if the `dst` parameter of the `os.Create()` function points to an existing file, then the file will be replaced by a new empty file.

However, with a crafted volume's host path, the result of `filepath.Join()` is possible to point to any host file outside the runc jailer's root directory. In the meantime, the `os.Chown()` function does not check whether the joined path is under the jailer's root directory or not, but simply changes the owner of the file's parent directory to the specified host's regular user. What's worse, the `os.Create()` function does not perform any checking either. Thus, if the `dst` parameter points to an existing important system file on the host (e.g., ld.so), the file will be emptied brutally. As a result, the host can not run any applications anymore due to the file crash.

Experiments. We give the details of how to launch the attack. Specifically, we specify the `m.HostPath()` to `/../../../../../../root/FILENAME` (`FILENAME` is a regular file in the `/root` directory). As shown in Figure 7, the `filePath.Join()` join the directory to `/root/FILENAME` (line 5). Then, in the `bindMountFiletoJail()` function, the `dst` parameter is `/root/FILENAME`. As the `filePath.Dir()` function gets the location of the directory where the file is located, the result of `filePath.Dir()` is `/root`. The `os.Chown()` function changes the `/root`'s owner to a specified host regular user (line 14). As a result, the host regular user can get the privileges to access the `/root` directory. Besides, we specify the `m.HostPath()` to `/../../../../../../lib/x86_64-linux-gnu/ld-2.27.so`, as shown in Figure 7, the `filePath.Join()` join the `dst` parameter as `/lib/x86_64-linux-gnu/ld-2.27.so`. As `ld-2.27.so` is an existing file in the host, the `os.Create()` truncate its size to 0 (line 16), which crashes the host as it can not run any new meaningful applications.

4.4.2 Firecracker-based Container Dirty Memory Attack

The second case is the Firecracker-based container dirty memory attack, which involves strategy ❷. A malicious Firecracker-based container can leverage the virtio block (virtio-blk) to forward the `write()` operation to host system. Similar to the dirty memory attack against Kata Containers, a malicious container can trigger this operation forwarding repeatedly and exhaust the host's dirty memory. As a result, the victim Firecracker-based container suffers 86.7% IO performance downgrade.

Root cause analysis. Similar to virtiofs daemon in Kata Containers, Firecracker-based container can also increase the size of host dirty memory with the help of virtio-blk. Firecracker-based container uses virtio-blk to mount the container's rootfs images into the microVM. As shown in Figure 8, when the malicious container generates a large number of `write()` system call to write a file in the container, the file operations are transferred as block IO requests to the virtio-blk driver in the guest kernel. Then the driver forwards the block IO requests to the virtio-blk device in the Firecracker via virtio. After that, the Firecracker makes a massive amount of host system call `write()` to modify the disk image file and quickly occupies the host's dirty memory, reaching the host kernel's `dirty_ratio` threshold. As a result, the host kernel forces all the write requests directly synchronize to the disk, which will dramatically downgrade the victim container's IO performance.

Experiments. On our local environment, we launch attacks with 1, 5, and 10 malicious containers, and there are 86.7%, 97.4%, and 98.2% slowdowns of the `dd` command in the victim containers, respectively. On the AWS environment, the machine has 192GB memory, and the `dirty_ratio` threshold

is set to 20%. It requires at least two containers to fill up the dirty memory. We conduct the experiments with 2, 5, and 10 malicious containers, and the victim suffers 70.0%, 83.1%, and 92.9% slowdowns, respectively.

4.4.3 Firecracker-based Container Nf_conntrack Table Attack

The third case is the Firecracker-based container nf_conntrack table attack, which involves strategy ❷. A malicious container can leverage the virtio-net device in Firecracker to forward the guest `connect()` system calls to host `sendmsg()`, and send frames into the host kernel network stack while writing an item in the host's nf_conntrack table. By triggering this mechanism repeatedly, a malicious container can occupy the host nf_conntrack table, causing 60.0% of the victim's packet loss.

Root cause analysis. Firecracker-based container uses the virtio-net device instead of the vhost-net kernel module as the back-end device of the virtio-net driver for networking. However, the malicious container can still leverage the virtio-net device to consume an entry of host nf_conntrack table. As shown in Figure 9, first, when the Firecracker-based container makes `connect()` system call repeatedly inside the microVM, the data-link layer takes a large number of packets from the network layer in the guest kernel. It encapsulates them into frames and transports them to the virtio-net back-end device in the Firecracker via virtio. Second, the Firecracker device emulator makes `sendmsg()` system call in the host kernel and sends frames to the `/dev/net/tap`, which is a tap device interface exposed to the host user space. After that, the frames are transported to other containers via the host CNI bridge. At the same time, the netfilter on the host calls the `nf_conntrack_alloc()` function and writes a huge number of items in the host kernel's nf_conntrack table. Thus, a malicious Firecracker-based container can generate many connections and fill up the host kernel's nf_conntrack table, making the host randomly drop packets.

Experiments. We launch attacks with 10, 15, and 20 containers. As a result, it takes up to one minute to fill up the host's nf_conntrack table. Once the nf_conntrack table is filled up, it begins to drop packets randomly. Specifically, for the `ping` experiments, the victim's packet losses are 60.0%, 58.0%, and 58.0% on the local environment, and 50.0%, 52.0%, and 55.0% on AWS. For the Nginx experiments, the Nginx server can not be accessed and a timeout error is returned.

4.4.4 KVM PIT Timer Attack

The fourth case is to exploit the programmable interval timer (PIT) [32] emulation mechanism in the KVM, which involves strategy ❸. First, the Firecracker-based container writes low-level I/O ports to create the virtual PIT timer, which KVM emulates. Second, the KVM wakes up a host kernel thread to handle `pit_do_work()` kernel function, injecting timer interrupts into the guest. Thus, a malicious Firecracker-based container can leverage this emulation mechanism to delegate workloads on the kernel thread, which consumes the host 68.0% CPU and significantly downgrades the performances of other containers up to 87.8%.

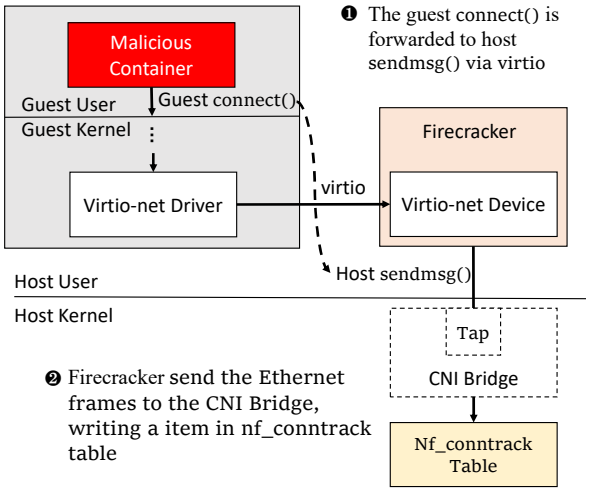


Figure 9: Operation forwarding of nf_conntrack table attack in Firecracker-based container. Firecracker-based container calls `connect()` make the Firecracker call `sendmsg()` to host CNI bridge, filling up the host nf_conntrack table.

Root cause analysis The kernel thread created by the KVM consumes CPU to inject PIT timer interrupt into the guest. The PIT (termed Intel 8253/8253) [32] is a counter that generates an output signal when it reaches a programmed count, and the output signal will trigger an interrupt. Writing `0x40-0x43` I/O ports can create a periodic PIT timer and set its period. To emulate the PIT timer in the microVM-based container, KVM leverages the host Linux kernel's high-resolution timers (hrtimers) [44] as a virtual counter. Besides, it creates a kernel thread called `kvm-pit/<owner-device-emulator-process-pid>` for each VM. Whenever the hrtimer reaches the programmed count, the callback function registered for the hrtimer queues the `pit_do_work()` into the kernel thread's work list and wakes up the thread.

If one can repeatedly write the I/O ports to create PIT timers and set a small value as the period of the timer, the kernel thread will keep scheduled to a physical CPU, and running `pit_do_work()` to inject timer interrupts into the guest. Furthermore, the kvm-pit kernel thread is attached to the root cgroups in the Linux kernel. The number of resources consumed by its workloads is accounted to the target kernel thread instead of the user-space process. As a result, a malicious microVM-based container can generate out-of-band workloads, consume host resources, and attack other microVM-based containers.

Experiments. In our experiment, we use the `outb()` function

to write the `0x43` port in the malicious container, trigger the VM exit, and make KVM wake up the kernel thread to inject timer interrupts. By generating many write requests to set the PIT timer to tick, malicious containers can force kvm-pit threads to run out-of-band workloads in the host kernel and consume 68.0% of host CPU resources. We use sysbench to measure the impact on victim containers. We launch attacks with 1, 10, and 20 containers. The victim suffers 3.3%, 75.0%, and 86.6% downgrades in the CPU benchmark. The victim suffers 5.3%, 80.1%, and 87.8% downgrades in the memory benchmark, 4.6%, 26.3%, and 64.6% downgrades in the IO read benchmark, and 5.0%, 26.7%, and 64.6% downgrades in the IO write benchmark, respectively.

4.5 Practicality Discussion

Attacks on traditional VM. The traditional VMs are also vulnerable to the operation forwarding attack, as the VM relies on the host kernel to serve its functionalities. To validate it, we choose QEMU/KVM to run the VMs and assume that the attacker fully controls the malicious VM. Then we perform our attacks in the local environment. The results show that the dirty memory attack and the nf_conntrack table attack can successfully cause DoS attacks. If we enable the vhost-net for VMs networking and utilize virtiofs to share directories between the guest and host systems, the vhost-net attack and the virtiofs daemon escalation attack can both achieve the same impact as for the microVM-based container. Surprisingly, we also find that the virtiofs daemon escalation attack can also achieve privilege escalation in 9pfs [45], which is also a pass-through file system that shares directories between host and guest, similar to virtiofs. However, as QEMU uses High Precision Event Timer (HPET) [46] to replace PIT timer, the KVM PIT timer attack cannot work in QEMU/KVM environment.

Responsible disclosure. All eight attacks presented in this work have been responsibly disclosed to the Kata Containers, Firecracker, and virtiofs teams. We summarize our reports in Table 2 of appendix A. Specifically, all attacks have been confirmed. Further, the virtiofs daemon escalation, the firecracker-containerd escalation, and the nf_conntrack table attack (Firecracker) have been fixed and the patches have been merged. While for the dirty memory attack (Kata), the nf_conntrack table attack (Kata), the vhost-net attack, the dirty memory attack (Firecracker), and the KVM PIT timer attack, the developers have given out multiple methods to mitigate these risks. We list all responses for each disclosed attack from the related teams in Figures 10 and 11.

5 Mitigation Discussion

In the following, we give multiple suggestions to mitigate the operation forwarding attacks introduced by container runtime components, the device emulator, and host kernel components, respectively. These suggestions are based on our communication with the related security teams.

5.1 Detecting Attacks via Monitor Tools

Monitor tools such as Virtual Machine Introspection (VMI) [47] can be used to detect abnormal behaviors of the microVM-based container. VMI can be leveraged to monitor the sensitive guest syscalls raised by the microVM-based containers and defend against attacks triggered by guest system calls via different rule configurations. However, the VMI technique brings extra performance overhead which may be unacceptable in certain scenarios [48].

5.2 Protecting Container Runtime Components

Jailing the container runtime components. Jailing techniques (i.e., namespaces and cgroups) can be applied on container runtime components to enforce additional resource isolation and limitation. Particularly, the mount namespace can prevent firecracker-containerd from modifying files outside its working directory, thus defeating the firecracker-containerd escalation attack. Besides, one can limit dirty memory usage of the virtiofs by adding the virtiofs daemon into cgroups.

However, ensuring container runtime's normal functionalities needs effort. For example, virtiofs community engaged in intense discussion [3] on using user namespace to start non-root virtiofs daemon without affecting users' experiences.

Disabling the sharing file system. We also suggest disabling the virtiofs file system to avoid multiple operation forwarding paths and mitigate all the vulnerabilities introduced by virtiofs. However, virtiofs is designed to offer local file system semantics and high performance. The file system performance downgrades without virtiofs in Kata Containers [49].

5.3 Protecting the Device Emulator

Using SR-IOV passthrough devices. We recommend to use Single Root I/O Virtualization (SR-IOV) [4]pass-through devices for each microVM-based container to protect the device emulator if required. In the SR-IOV scenario, each VM's I/O requests are directly handled by the physical device and bypass the host kernel for processing. Specifically, the SR-IOV devices can mitigate the Firecracker-based container nf_conntrack table attacks, as the network packets are directly processed by the virtual bridge in the SR-IOV adapter. However, it needs an SR-IOV capable adapter which may increase the cost of the infrastructure. For example, according

[3] https://patchew.org/QEMU/348d4774-bd5f-4832-bd7e-a21491fdac8d@www.fastmail.com/

[4] https://en.wikipedia.org/wiki/Single-root_input/output_virtualization

to the AWS EC2 documenation[5], users can enable SR-IOV in special EC2 instances on AWS.

Adding rate limitations. It is also suggested to limit the access rate of host resources occupied by microVM-based containers by enabling rate-limiting capabilities for specific device emulators (i.e., networking and storage devices). Specifically, the system operators can configure the Firecracker rate limiter to slow down the speed of producing dirty memory and occupying the host nf_conntrack table.

5.4 Protecting Host Kernel Components

Attaching the worker threads to proper cgroup. Container user can confine host kernel components' workloads by moving related kernel threads into the microVM-based container's cgroup. For example, if the vhost-net and KVM PIT kernel threads attach to the microVM-based container's cgroups, their workloads will be accounted and restrained properly.

Disabling functionalities of host kernel components. If needed, it can prevent `vhost-net` and `kvm-pit` kernel thread to generate workload by disabling the host's vhost-net kernel module and KVM PIT emulation. Thus, the vhost-net attack and KVM PIT timer attack would be defeated.

6 Related Work

Virtual machine security. Since virtual machines are widely used in multi-tenant cloud environments, there is extensive research on VM security. Ristenpart et al. utilize host information like network addresses to detect co-residency [10]. Later, multiple works leverage different level cache to construct covert channels [11, 14, 15, 50] for co-resident detection. Zhang et al. demonstrate that attackers can use cache and DRAM to launch side-channel attacks [12, 13, 51], which can extract information from other VMs and even make privilege escalation. Yelam et al. successfully detect co-residency in AWS serverless environments based on memory bus contention [52]. Besides, [53–56] exploit the hardware contention to cause Denial-of-Service attacks. However, they mainly target the shared hardware while ignoring that VMs' components rely on the host kernel and forward operations from guest to host.

Huang et al. demonstrate a kind of DoS attack called cascade attack, which exhausts Xen's shareable domain's processing capability to affect other co-resident VMs' performance [57]. Varadarajan et al. propose resource-freeing attack to help improve the performance of the attacker's VM [58]. Zhou et al. exploit the boost mechanism in the Xen credit scheduler to obtain extra CPU resources [59]. However, their works exploit Xen hypervisors or shared domain of Xen. However, our target is microVM-based containers based on KVM hypervisor, and we focus on operation forwarding brought by components of microVM-based container.

Container security. Besides virtual machine security, there are studies on container security. Luo et al. identify several covert channels against Docker and show related information leak attacks [60]. Gao et al. systematically identify the container's information leak channel in the `/proc` and the `/sys` directories [2]. Gao et al. demonstrate the exploiting strategies to break the resource restrictions of Linux Control groups [3]. Yang et al. reveal the abstract resources in the containers' shared kernel and related abstract resource attacks [4]. However, all these papers mainly focus on the native container but do not systematically study the potential risks in microVM-based containers.

There are also works on securing containers. Arnautov et al. secure containers with Intel SGX [61]. Lei et al. reveal a security mechanism called SPEAKER to reduce available system calls of the containers [62]. Sun et al. propose a new Linux namespace called security namespace to isolate security policies for containers [63]. However, these works mainly focus on securing native containers, which is orthogonal to our work.

7 Conclusion

In this paper, we demonstrate that the components of microVM-based container forward the user's operations to host system calls or host kernel functions, which can be exploited to break the isolation of microVM-based containers. We divide the microVM-based container's components into three layers based on their functionalities. We illustrate that attackers can leverage each layer's components to forward their operations to launch attacks. We further present four attacks against Kata Containers and Firecracker-based container, respectively, and we do the experiments on both the local testbed and cloud bare-metal physical servers. The results show that the attacks can make privilege escalation, DoS attacks, and generate out-of-band workload. Finally, we provide several suggestions for microVM-based container users and developers to mitigate operation forwarding attacks.

Acknowledgments

The authors would like to thank our shepherd and reviewers for their insightful comments. Those comments helped to reshape this paper. This work is partially supported by the National Natural Science Foundation of China (Key Program Grant No. 62232013, Grant No. 62002317), by the Foundation for Innovative Research Groups of the National Natural Science Foundation of China (Grant No. 62121001), by the Key R&D Program of Shaanxi Province of China (Grant No. 2019ZDLGY12-06), by the Innovative Research Foundation of Ship General Performance (Grant No. 25622113), and by the Tencent Security Yunding Lab.

[5] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-types.html

References

[1] CNCF. Cncf annual survey 2021. https://www.cncf.io/reports/cncf-annual-survey-2021/, 2022.

[2] Xing Gao, Benjamin Steenkamer, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. A study on the security implications of information leakages in container clouds. *IEEE Transactions on Dependable and Secure Computing*, 2018.

[3] Xing Gao, Zhongshu Gu, Zhengfa Li, Hani Jamjoom, and Cong Wang. Houdini's escape: Breaking the resource rein of linux control groups. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[4] Nanzi Yang, Wenbo Shen, Jinku Li, Yutian Yang, Kangjie Lu, Jietao Xiao, Tianyu Zhou, Chenggang Qin, Wang Yu, Jianfeng Ma, et al. Demons in the shared kernel: Abstract resource attacks against os-level virtualization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 764–778, 2021.

[5] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. A measurement study on linux container security: Attacks and countermeasures. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 418–429, 2018.

[6] Amazon. Firecracker – lightweight virtualization for serverless computing. https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing/, 2022.

[7] katacontainers. Kata containers - open source container runtime software. https://katacontainers.io/, 2022.

[8] Amazon. Aws fargate. https://aws.amazon.com/fargate/, 2022.

[9] Amazon. Aws lambda. https://aws.amazon.com/lambda/, 2022.

[10] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212, 2009.

[11] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. An exploration of l2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 29–40, 2011.

[12] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 305–316, 2012.

[13] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 990–1003, 2014.

[14] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*, pages 605–622. IEEE, 2015.

[15] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6, 2016.

[16] The MITRE Corporation. Cve-2022-0358. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-0358, 2022.

[17] KVM. Main page. https://www.linux-kvm.org/index.php?title=Main_Page&oldid=173792, 2016.

[18] Michael S. Tsirkin. vhost_net: a kernel-level virtio server. https://lwn.net/Articles/346267/, 2022.

[19] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[20] Firecracker-microvm. firecracker-based container. https://github.com/firecracker-microvm/firecracker-containerd/blob/main/README.md?plain=1#L13, 2022.

[21] Firecracker-microvm. firecracker-containerd architecture. https://github.com/firecracker-microvm/firecracker-containerd, 2022.

[22] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, pages 419–434, 2020.

[23] AWS. Aws fargate security overview whitepaper. https://d1.awsstatic.com/whitepapers/

AWS_Fargate_Security_Overview_Whitepaper.pdf, 2022.

[24] Alibaba Cloud. What is elastic container instance? https://www.alibabacloud.com/help/en/elastic-container-instance/latest/what-is-elastic-container-instance, 2022.

[25] Wikipedia. Intel virtualization (vt-x). https://en.wikipedia.org/wiki/X86_virtualization#Intel-VT-x, 2022.

[26] Wikipedia. seccomp. https://en.wikipedia.org/wiki/Seccomp, 2022.

[27] Wikipedia. Cgroups. https://en.wikipedia.org/wiki/Cgroups, 2022.

[28] Wikipedia. Linux namespaces. https://en.wikipedia.org/wiki/Linux_namespaces, 2022.

[29] byTyler Carrigan (Red Hat). Linux permissions: Suid, sgid, and sticky bit. https://www.redhat.com/sysadmin/suid-sgid-sticky-bit, 2021.

[30] Linux man-pages project. open. https://man7.org/linux/man-pages/man2/open.2.html, 2021.

[31] The QEMU Project Developers. Qemu virtio-fs shared file system daemon. https://www.qemu.org/docs/master/tools/virtiofsd.html?highlight=virtiofs, 2022.

[32] Wikipedia. Programmable interval timer. https://en.wikipedia.org/wiki/Programmable_interval_timer, 2021.

[33] Linux Test Project. Linux test project. https://linux-test-project.github.io/, 2022.

[34] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *USENIX Security Symposium*, pages 2597–2614, 2021.

[35] Andrew Henderson, Heng Yin, Guang Jin, Hao Han, and Hongmei Deng. Vdf: Targeted evolutionary fuzz testing of virtual devices. In *Research in Attacks, Intrusions, and Defenses: 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18–20, 2017, Proceedings*, pages 3–25. Springer, 2017.

[36] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Hyper-cube: High-dimensional hypervisor fuzzing. In *NDSS*, 2020.

[37] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D Garcia, and Frank Piessens. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1741–1758, 2019.

[38] Firecracker-microvm. firecracker-volume-test. https://github.com/firecracker-microvm/firecracker-containerd/blob/main/runtime/volume_integ_test.go, 2022.

[39] Kata Containers. Developer guide. https://github.com/kata-containers/kata-containers/blob/main/docs/Developer-Guide.md, 2022.

[40] Wikipedia. Connection tracking. https://en.wikipedia.org/wiki/Netfilter#Connection_tracking, 2022.

[41] Wikipedia. Tun/tap. https://en.wikipedia.org/wiki/TUN/TAP, 2021.

[42] CNI.dev. Cni bridge plugin. https://www.cni.dev/plugins/v0.7/main/bridge/, 2022.

[43] Apache. ab - apache http server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html, 2022.

[44] The kernel development community. hrtimers-subsystem for high-resolution kernel timers. https://docs.kernel.org/timers/hrtimers.html, 2022.

[45] QEMU. Qemu 9p document. https://wiki.qemu.org/Documentation/9p, 2022.

[46] Wikipedia. High precision event timer. https://en.wikipedia.org/wiki/High_Precision_Event_Timer, 2022.

[47] Tal Garfinkel, Mendel Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *Ndss*, volume 3, pages 191–206. Citeseer, 2003.

[48] Zhaofeng Yu, Lin Ye, Hongli Zhang, Dongyang Zhan, Shen Su, and Zhihong Tian. A container-oriented virtual-machine-introspection-based security monitor to secure containers in cloud computing. In *Artificial Intelligence and Security: 7th International Conference, ICAIS 2021, Dublin, Ireland, July 19–23, 2021, Proceedings, Part II 7*, pages 102–111. Springer, 2021.

[49] Bharat Kunwar. Disk i/o performance of kata containers. https://www.stackhpc.com/images/IO-Performance-of-Kata-Containers-TheNewStack.pdf, 2022.

[50] Yuval Yarom and Katrina Falkner. {FLUSH+RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack. In *23rd USENIX security symposium (USENIX security 14)*, pages 719–732, 2014.

[51] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops:{Cross-VM} row hammer attacks and privilege escalation. In *25th USENIX security symposium (USENIX Security 16)*, pages 19–35, 2016.

[52] Anil Yelam, Shibani Subbareddy, Keerthana Ganesan, Stefan Savage, and Ariana Mirian. Coresident evil: Covert communication in the cloud with lambdas. In *Proceedings of the Web Conference 2021*, pages 1005–1016, 2021.

[53] Dirk Grunwald and Soraya Ghiasi. Microarchitectural denial of service: Insuring microarchitectural fairness. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002.(MICRO-35). Proceedings.*, pages 409–418. IEEE, 2002.

[54] D Hyuk Woo and HH Lee. Analyzing performance vulnerability due to resource denial of service attack on chip multiprocessors. In *Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2007.

[55] Thomas Moscibroda Onur Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX security*, 2007.

[56] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. Dos attacks on your memory in cloud. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 253–265, 2017.

[57] Qun Huang and Patrick PC Lee. An experimental study of cascading performance interference in a virtualized environment. *ACM SIGMETRICS Performance Evaluation Review*, 40(4):43–52, 2013.

[58] Venkatanathan Varadarajan, Thawan Kooburat, Benjamin Farley, Thomas Ristenpart, and Michael M Swift. Resource-freeing attacks: improve your cloud performance (at your neighbor's expense). In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 281–292, 2012.

[59] Fangfei Zhou, Manish Goel, Peter Desnoyers, and Ravi Sundaram. Scheduler vulnerabilities and coordinated attacks in cloud computing. *Journal of Computer Security*, 21(4):533–559, 2013.

[60] Yang Luo, Wu Luo, Xiaoning Sun, Qingni Shen, Anbang Ruan, and Zhonghai Wu. Whispers between the containers: High-capacity covert channel attacks in docker. In *2016 IEEE trustcom/bigdatase/ispa*, pages 630–637. IEEE, 2016.

[61] 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 *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016.

[62] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. Speaker: Split-phase execution of application containers. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2017.

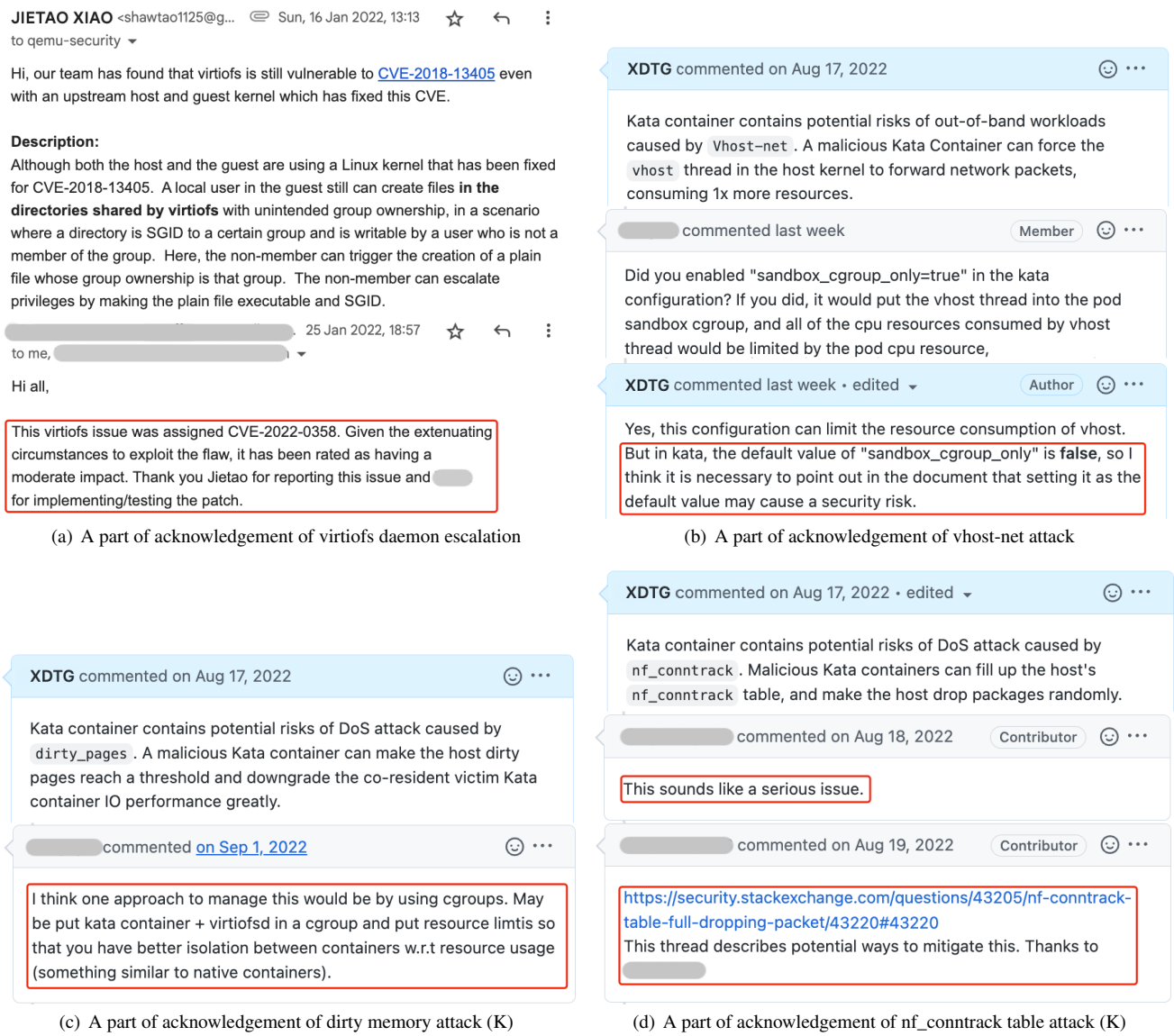
[63] Yuqiong Sun, David Safford, Mimi Zohar, Dimitrios Pendarakis, Zhongshu Gu, and Trent Jaeger. Security namespace: making linux security frameworks available to containers. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018.

A Appendix

We summarize all the attacks' reports in Table 2, including the attack case name (Case), the confirmed status of each case (Confirm Status), and whether the problem is fixed (Fix status). All the cases which we conduct in this paper have been reported through the proper channel, and all of them have been confirmed. The related organizations have fixed and merged the source code for the virtiofs daemon escalation, the firecracker-containerd escalation, and the nf_conntrack table attack (Firecracker). Further, We present the details about the disclosures of all the attack cases in Figures 10 and 11.

Table 2: Summary of all the attacks' reports. "K" denotes Kata Containers environment, "F" denotes Firecracker-based container environment.

Case	Confirm Status	Fix Status
Virtiofs daemon escalation	Confirmed	Patched
Firecracker-containerd escalation	Confirmed	Patched
Dirty memory attack (K)	Confirmed	Patch pending
Dirty memory attack (F)	Confirmed	Patch pending
Nf_conntrack table attack (K)	Confirmed	Patch pending
Nf_conntrack table attack (F)	Confirmed	Patched
Vhost-net attack	Confirmed	Patch pending
KVM PIT timer attack	Confirmed	Patch pending



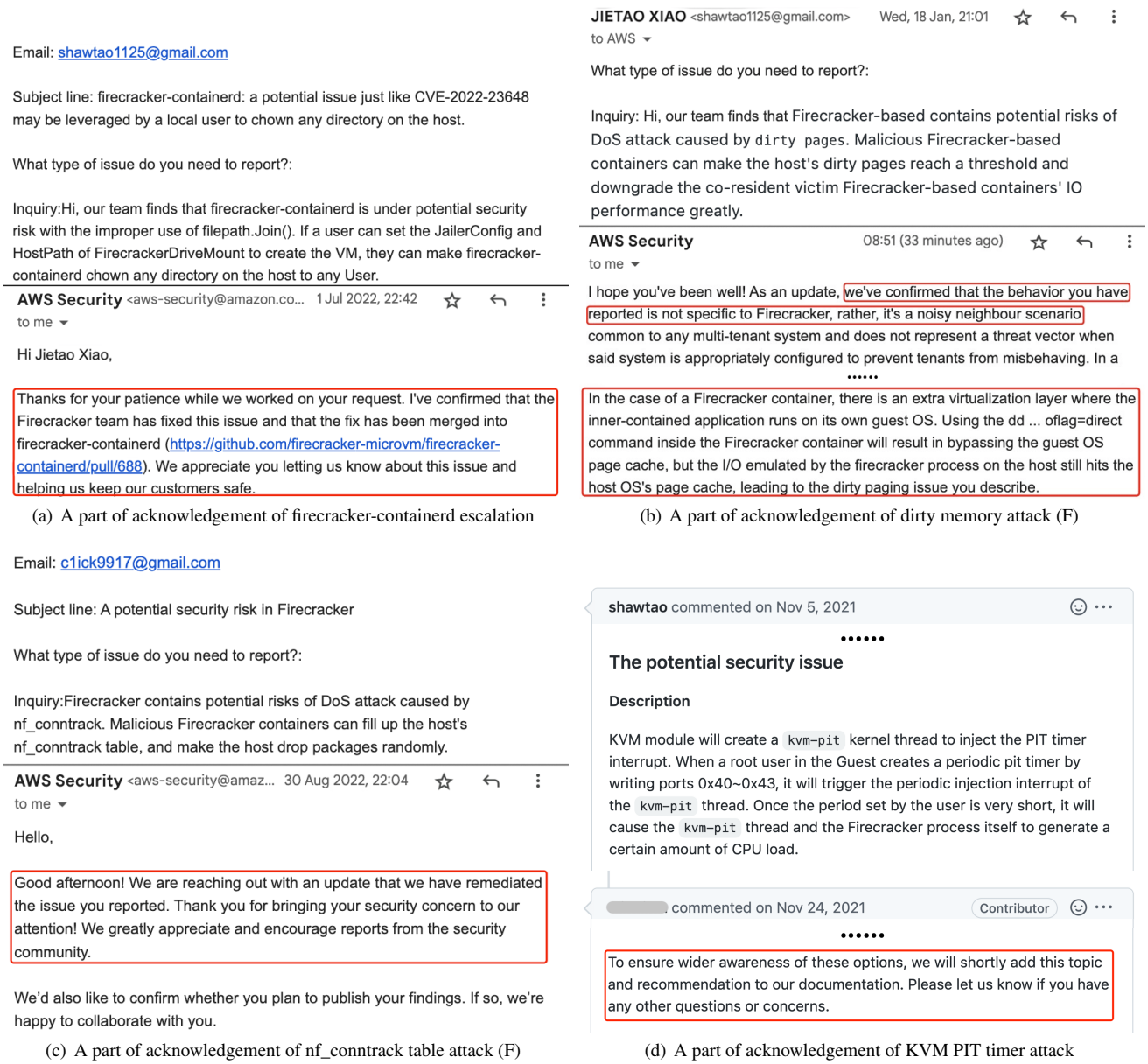
(a) A part of acknowledgement of virtiofs daemon escalation

(b) A part of acknowledgement of vhost-net attack

(c) A part of acknowledgement of dirty memory attack (K)

(d) A part of acknowledgement of nf_conntrack table attack (K)

Figure 10: The issue acknowledgments of Kata Containers



(a) A part of acknowledgement of firecracker-containerd escalation

(b) A part of acknowledgement of dirty memory attack (F)

(c) A part of acknowledgement of nf_conntrack table attack (F)

(d) A part of acknowledgement of KVM PIT timer attack

Figure 11: The issue acknowledgments of the Firecracker-based Container