Demons in the Shared Kernel: Abstract Resource Attacks Against OS-level Virtualization

Nanzi Yang*
Xidian University
Xi'an, China

Wenbo Shen*†
Zhejiang University
Key Laboratory of Blockchain and Cyberspace Governance of Zhejiang Province
Hangzhou, China

Jinku Li
Xidian University
Xi'an, China

Yutian Yang
Zhejiang University
Hangzhou, China

Kangjie Lu
University of Minnesota, Twin Cities
Minneapolis, USA

Jietao Xiao
Xidian University
Xi'an, China

Tianyu Zhou
Zhejiang University
Hangzhou, China

Chenggang Qin
Ant Group
Hangzhou, China

Wang Yu
Ant Group
Hangzhou, China

Jianfeng Ma
Xidian University
Xi'an, China

Kui Ren
Zhejiang University
Hangzhou, China

ABSTRACT

Due to its faster start-up speed and better resource utilization efficiency, OS-level virtualization has been widely adopted and has become a fundamental technology in cloud computing. Compared to hardware virtualization, OS-level virtualization leverages the shared-kernel design to achieve high efficiency and runs multiple user-space instances (a.k.a., containers) on the shared kernel.

However, in this paper, we reveal a new attack surface that is intrinsic to OS-level virtualization, affecting Linux, FreeBSD, and Fuchsia. The root cause is that the shared-kernel design in OS-level virtualization results containers in sharing thousands of kernel variables and data structures directly and indirectly. Without exploiting any kernel vulnerabilities, a non-privileged container can easily exhaust the shared kernel variables and data structure instances to cause DoS attacks against other containers. Compared with the physical resources, these kernel variables or data structure instances (termed *abstract resources*) are more prevalent but under-protected.

To show the importance of confining abstract resources, we conduct abstract resource attacks that target different aspects of the OS kernel. The results show that attacking abstract resources is highly practical and critical. We further conduct a systematic analysis to identify vulnerable abstract resources in the Linux kernel, which successfully detects 1,010 abstract resources and 501 of them can be repeatedly consumed dynamically. We also conduct the attacking experiments in the self-deployed shared-kernel container environments on the top 4 cloud vendors. The results show that all environments are vulnerable to abstract resource attacks. We conclude that containing abstract resources is hard and give out multiple strategies for mitigating the risks.

*Co-first authors.
†Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

© 2021 Association for Computing Machinery.

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3484744

CCS CONCEPTS

• **Security and privacy → Virtualization and security**.

KEYWORDS

OS-level Virtualization; Shared Kernel; Abstract Resource Attack

ACM Reference Format:
Nanzi Yang, Wenbo Shen, Jinku Li, Yutian Yang, Kangjie Lu, Jietao Xiao, Tianyu Zhou, Chenggang Qin, Wang Yu, Jianfeng Ma, and Kui Ren. 2021. 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 (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3460120.3484744

1 INTRODUCTION

Operating-system-level virtualization (a.k.a., OS-level virtualization) allows multiple self-contained and isolated user-space environments to run on the same kernel [67]. Compared to hardware virtualization (i.e., virtual machines), OS-level virtualization eliminates the burden of maintaining an operating system kernel for each user-space instance and thus has a faster start-up speed and

better resource utilization efficiency. Therefore, OS-level virtualization has been widely adopted in recent years and has become a fundamental technology in cloud computing. The user-space instances in OS-level virtualization are named as *jails* in FreeBSD [33], *Zones* in Solaris [59], and *containers*[1] in Linux [67].

Despite its high efficiency, OS-level virtualization also introduces multiple security concerns. First, OS-level virtualization is vulnerable to kernel vulnerabilities due to the shared kernel [40]. As a result, it cannot isolate kernel bugs. Once the shared kernel is compromised, all user-space instances (referred to as *containers*) lose isolation and protection. Moreover, researchers recently questioned the isolation of container techniques, such as information leaks [22], covert channels [24], and out-of-band workloads that break control groups [23].

However, in this paper, we reveal *a new attack surface that is intrinsic to OS-level virtualization.* Compared to hardware virtualization, OS-level virtualization leverages the shared-kernel design to achieve high efficiency. In a typical OS-level virtualization environment, containers run on the same OS kernel and request various services via 300+ system calls. Notice that the underlying OS kernel contains hundreds of thousands of variables and data structure instances to provide services for containers. As a result, *these containers are directly and indirectly sharing these kernel variables and data structure instances.*

Unfortunately, these shared kernel variables and data structure instances are new attack surfaces in OS-level virtualization. Without exploiting any vulnerabilities, a non-privileged container can easily exhaust certain kernel variables and data structure instances, causing DoS attacks in OS-level virtualization environments. As a result, even other containers have enough physical resources, with the kernel critical variables or data structure instances being exhausted, they still cannot perform any meaningful tasks. Compared with the physical resources supported by the real hardware, we regard these kernel variables or data structure instances as *abstract resources* and the exhaustion attacks on these resources as *abstract resource attacks.*

Though abstract resources can be exploited for DoS attacks, they are often under-protected. The kernel and container developers focus more on protecting physical resources rather than abstract resources. For example, the Linux kernel provides control groups to restrict the resource usages for each container instance. However, among 13 control groups, 12 of them are for physical resources, restricting the usages of CPU, memory, storage, and IO. Only the PIDs control group is designed for limiting the abstract resource `pid`. As a result, hundreds of container-shared abstract resources do not have any restrictions, such as the global dirty ratio, open-file structs, and pseudo-terminal structs, which makes them vulnerable to DoS attacks.

To show the criticality of confining abstract resources on OS-level virtualization, we conduct attacks using Docker containers on the Linux kernel, targeting abstract resources on different aspects of the operating system services, including process management, memory management, storage management, and IO management. Our experiments show that attacking abstract resources is highly practical and critical—it can easily disable new program execution, slow down the memory writes by 97.3%, crash all file-open related operations, and deny all new SSH connections. Even worse, it affects all aspects of OS services. Moreover, experiments also demonstrate that other than Linux, FreeBSD and Fuchsia are also vulnerable to abstract resource attacks.

It is unfortunate that even though abstract resources are critical, they are inherently hard to contain for several fundamental reasons. First, it is impractical to enumerate all possible abstract resources in operating system kernels. Different from the few physical resource types, abstract resource types in the kernel are many and various. Second, it is fairly easy to form conditions leading to abstract resource exhaustion. When implementing new features in the kernel, developers are often concerned about the physical resource consumption while paying much less attention to abstract resource consumption. Moreover, the OS kernel has complex data and path dependencies, leading to various ways to exhaust abstract resources in the kernel.

Therefore, we design and implement a tool based on LLVM to identify vulnerable abstract resources in the Linux kernel systematically. We propose new techniques to identify the shareable abstract resources and analyze their container controllability. We apply our tool to the latest Linux kernel and detect 1,010 abstract resources. 501 of them can be repeatedly consumed dynamically. From the detected abstract resources, we pick 7 resources that affect each aspect of OS services based on our familiarity (i.e., we know the impacts of exhausting that resource). We further conduct the attacking experiments on these selected resources in the shared-kernel container environments deployed on the top 4 cloud vendors, including AWS, MS Azure, Google Cloud, and Alibaba Cloud. The results show that all environments are vulnerable to our attacks. At last, we give out multiple strategies for mitigating the risks of abstract resource attacks.

The contributions of this paper are as follows:

- **New Attack Surface:** We reveal a new attack surface that is intrinsic to OS-level virtualization. We propose a new attack called *abstract resource attack*. We demonstrate that the abstract resource attack is highly practical and is a broad class of attacks that affect Linux, FreeBSD, and Fuchsia.
- **Systematic Analysis:** We design and implement a static analysis tool based on LLVM to identify vulnerable abstract resources in the Linux kernel. We propose and implement novel techniques, including configuration-based analysis and container-controllability analysis. Our tool detects 501 abstract resources that can be dynamically and repeatedly triggered in the Linux kernel.
- **Practical Evaluation:** We evaluate 7 abstract resource attacks in the self-deployed shared-kernel container environments on AWS, MS Azure, Google Cloud, and Alibaba Cloud. All environments are vulnerable to abstract resource attacks. [2] In particular, two environments are vulnerable to 6 attacks, one environment is vulnerable to 5 attacks, and the other is vulnerable to 4 attacks. We responsibly disclosed our findings to all cloud vendors. All of them confirmed the identified problems.

[1]In this paper, we use the container to refer to the self-contained user-space execution environment that shares the kernel of the host system.

[2]Current public cloud vendors do not provide the shared-kernel containers to different users directly. Containers in public cloud are usually isolated by virtual machines.

- **Community Impact:** We plan to open-source our tool and the identified abstract resources at https://github.com/ZJU-SEC/AbstractResourceAttack, so that they can help the Linux kernel community and the container community to identify the weak spots of resource isolation in OS-level virtualization.

2 BACKGROUND

OS-level virtualization relies on the underlying OS kernel for resource isolation and containment. More specifically, the Linux kernel provides namespaces for resource isolation and control groups for resource containment.

2.1 Linux Namespaces

Linux namespaces provide process-level resource isolation. Currently, Linux namespaces are divided into 8 types. According to their release time, we list them as follows:

- **Mount** for file system isolation;
- **UTS** for hostname and domain name isolation;
- **IPC** for IPC and message queue isolation;
- **PID** for process ID isolation;
- **Network** for network resource isolation;
- **User** for UID/GID isolation;
- **Cgroup** for control group isolation;
- **Time** for clock time isolation.

A process can be assigned to different namespaces of different types. But for each type, it can only belong to one namespace. By default, a process is in the same namespaces as its parent. It can be added to a new namespace during process creation by passing specific flags, or during process running by calling the `setns` system call. Ideally, only processes within the same namespace can share the namespace isolated resources. Resources are thus isolated across namespaces. As a result, running out of an isolated resource in one namespace does not affect processes in other namespaces. Such a design inherently requires that the namespace mechanism correctly and thoroughly contains the resources.

However, there still exist hundreds of types of abstract resources that are not included by namespaces. The large attacking surface still exists even with the protection of namespaces. One may argue to isolate all the abstract resources using namespaces. This is however impractical: the huge number and flexibility of abstract resources make the solution unacceptable due to huge code changes and high performance overhead.

2.2 Linux Control Groups

On the other hand, Linux control groups are used to limit resource usages. A control group accounts for resources used by all processes within that control group. Control groups are organized as a tree structure, where resources accounted for children are also accounted for their parents. The limits on resource usages are also enforced recursively on the tree so that resource usages in a control group should not exceed the limits of all its ancestors.

Control groups mainly manage hardware resources like CPU, memory, storage, IO, and etc. There are two versions of control groups, namely v1 and v2. The main difference is that control group v1 can have a tree hierarchy for each type of resources while control group v2 has only one hierarchy. The implementation of resource accounting and resource usage limiting has little difference between v1 and v2, though. Control group v1 is currently used by default because it is more stable and provides control over more resources. It manages 13 types of resources while v2 supports only 9 resource types until now [44]. More specifically, among the 13 types of resources, 5 of them are for CPU accounting, including `cpu, cpuacct, cpuset, freezer, perf_event`; 3 of them are for memory, including `memory, hugetlb, rdma`; `blkio` is for storage; and 3 are for IO, including `devices, net_cls, net_prio`. Only `PIDs` control group are for the abstract resource of PID.

While limiting the usages of shared abstract resources in container processes can mitigate DoS attacks, it is again impractical to extend control groups to include all abstract resources. Accounting resources and enforcing limits on so many types of resources will introduce unacceptable overhead.

3 ABSTRACT RESOURCE ATTACKS

In this section, we first clarify the threat model and assumptions. Next, we discuss weaknesses in the container isolation. Finally, we show that abstract resource attacks also work on FreeBSD and Fuchsia kernels.

Threat model and assumptions. In this paper, as we are targeting OS-level virtualization, we assume the containers are running on the same shared kernel. Containers enforce state-of-the-art protection and follow the most security practices in deployment. More specifically, containers are running as different non-root users with all capabilities dropped. While the kernel is enforcing as many namespaces and control groups as possible for the container. Moreover, the kernel is also using seccomp to block sensitive system calls. We further assume that the kernel has no bugs and all security mechanisms are working properly.

On the other side, the attacker controls one container and attempts to disrupt other containers running on the same kernel. The attacker can run any code within the container and call seccomp allowed system calls. However, he/she is not allowed to exploit kernel vulnerabilities. Furthermore, the attacker is in a non-privileged container as a non-root user, with no capabilities at all. Finally, the attacker is not allowed to escalate the privilege or regain any capabilities. In the following, we show that due to shared abstract resources in the kernel, even such an attacker still can launch DoS attacks to other containers.

3.1 Weaknesses in OS-level Virtualization

In OS-level virtualization, containers are directly and indirectly sharing thousands of kernel abstract resources, which makes them vulnerable to resource-exhaustion attacks. We leverage an example in the Linux kernel to illustrate the details. Figure 1 shows the global variable `nr_files` and function `alloc_empty_file` in the Linux kernel. `alloc_empty_file` allocates `struct file` (line 17). For each allocated `struct file`, `nr_files` accounts it by increasing the counter (line 19). In the host Linux kernel, the total number of `struct file` is limited by `files_stat.max_files` (line 13). If the limit is reached, the `alloc_empty_file` returns an error (line 23). However, the Linux kernel does not provide any namespaces or control groups to isolate or limit `nr_files`. As a result, `nr_files` is directly controllable to all containers—any allocation of `struct`

```
static struct percpu_counter nr_files __cacheline_aligned_in_smp;

static long get_nr_files(void)
{
  return percpu_counter_read_positive(&nr_files);
}

struct file *alloc_empty_file(int flags, const struct cred *cred)
{
    static long old_max;
    struct file *f;

    if (get_nr_files() >= files_stat.max_files &&
    ↪  !capable(CAP_SYS_ADMIN)) {
        ...
        goto over;
    }
    f = __alloc_file(flags, cred);
    if (!IS_ERR(f))
        percpu_counter_inc(&nr_files);
    ...
over:
    ...
    return ERR_PTR(-ENFILE);
}
```

Figure 1: Linux kernel source of `nr_files`. `nr_files` is a global variable shared by all containers. For each allocated `struct file`, `nr_files` increases by 1 (line 19).

`file` from any container increases the same shared global variable `nr_files`.

Such a sharing of `nr_files` leads to a new attack. In Linux, everything is a file. So many operations, such as file open, process creation, pipe creation, new network connection creation, even the timer creation (`timerfd_create`) and event generation (`eventfd`), increase `nr_files`. A malicious container can pop `nr_files` to its upper limit easily. Actually, in our experiment, the quota of `nr_files` can be quickly exhausted in several seconds. Consequently, all operations that consume `struct file` will fail. The impact is severe: the victim-container cannot even run a command (as it needs to open a command file) or `exec` a new binary, leading to program crashes. From the above example, we find that even the container has enough physical resources, such as CPU or memory, it still cannot run any new programs without the quota in `nr_files`.

To demonstrate that abstract resource attacks affect all kernel functionalities, we present one abstract resource attack for each aspect of the Linux kernel functionalities, including process, memory, storage, and IO management [21]. In this section, we present the attack results on the local test environments and defer the attack results of the top 4 vendors to §5.

For the local test environment setup, the test machine has the Intel Core i5 CPU, with 8 GB memory and 500 GB HDD, and it runs Ubuntu 18.04 with Linux kernel v5.3.1. We refer to it as the *host-machine*. On the host-machine, we set up two docker containers using Docker 18.06.0-ce, and use them as *attacker-container* and *victim-container*, respectively. We set up both containers following the docker security best practices [9, 12, 30], which is running them in different non-root users, dropping all capabilities, enabling namespaces and control groups, and applying seccomp system call blocking, as discussed in the threat model.

3.2 Attacks on Process Management

To implement process management, the Linux kernel has introduced a series of abstract resources, such as process-control-block `struct task_struct`, `pid`, `state` and various data structures to support the derived entities, such as `struct thread_info` for thread, `struct rq` runqueues for scheduling, `struct shm_info` and `struct msginfo` for inter-process communication (IPC), `struct spinlock` and `struct semaphores` for synchronization. In fact, process management in Linux introduces thousands of abstract resources. In the following, we introduce the attack against `struct idr` as an example.

```
struct pid *alloc_pid(struct pid_namespace *ns, pid_t *set_tid,
↪  size_t set_tid_size)
{
    ...
    nr = idr_alloc_cyclic(&tmp->idr, NULL, pid_min, pid_max,
    ↪  GFP_ATOMIC);
    ...
    if (nr < 0) {
        retval = (nr == -ENOSPC) ? -EAGAIN : nr;
        goto out_free;
    }
    pid->numbers[i].nr = nr;
    ...
}
```

Figure 2: Linux kernel source of `idr` allocation. `idr_alloc_cyclic` checks the idr against `pid_max`, returning a negative number idr if goes beyond.

3.2.1 Attacking `idr` of PID. The Linux kernel introduces `struct idr` for integer ID management. Process management also uses `idr` for the pid allocation. Figure 2 shows the `alloc_pid` function, which calls `idr_alloc_cyclic` to get a new pid. `idr_alloc_cyclic`, in turn, checks `pid_max` during the idr allocation and return a negative error code if the idr grows beyond `pid_max`. Later we will show that even with PID namespace and PIDs control group enabled, `idr` can still be regarded as a globally shared resource for all processes. Similar to the fork bomb, a malicious container process can repeatedly fork to exhaust all `idr`. As a result, all containers on the shared-kernel cannot create any new processes or threads.

In our experiments, the attacker-container spawns processes repeatedly by calling the fork system call. As a result, in the victim-container, all operations related to new-process creation fail with an error of "Resource temporarily unavailable". Even root users on the host-machine suffer from the same failure.

3.2.2 The effectiveness of the PID namespace. Linux v2.6.24 introduces the PID namespace, which provides processes an independent set of PIDs from other PID namespaces [47]. However, in the PID namespace implementation, the Linux kernel allocates an extra PID in the root PID namespace for any PID allocated in other PID namespaces, so that all PIDs in the other PID namespaces can be mapped to the root PID namespace. In other words, the root PID namespace is still globally shared. As a result, even the attacker-container is in a separated PID namespace, its PID allocation still exhausts the PID in the root PID namespace, causing the new-process-create failures on both the victim-container and host-machine. Therefore, even with the PID namespace enabled, containers are still vulnerable to the above idr-exhaustion attack.

3.2.3 The effectiveness of the PIDs control group. The PIDs control group was also introduced recently in Linux v4.3 [44]. It is used to limit the total number of PIDs that are allocated in one

control group. More specifically, the PIDs control group checks against the process limit during the process forking, and returns an error and aborts the forking if the total process number in the PIDs control group (pids_cgroup->counter) reaches the upper limit (pids_cgroup->limit). PIDs control group is effective in defending against direct forking. However, it only charges the pid number to the current process. Similar to the work-delegation approach in [23], the attacker-container can trick the kernel to fork a large number of kernel threads, such as frequent aborting to cause the kernel to spawn interrupt-handling threads. In this way, the idr is exhausted by kernel threads, which bypasses the restriction enforced by the PIDs control group.

3.3 Attacks on Memory Management

The Linux kernel introduces various kernel data structures, such as mm_struct for holding all the memory-related information of a process, and vm_area_struct for representing the virtual memory area. Moreover, to improve the reading and writing efficiency, the Linux kernel also uses the memory as buffers to cache certain data. Besides, it also introduces the write-back scheme, in which the writing is done only to the memory. The dirty memory pages will be written to the disk later by the kernel thread. Using the write-back scheme, the caller only needs to write to the memory, and it does not need to wait for the time-consuming disk-IO operations to finish (i.e., write-through), which significantly improves the write performance. However, we find that the kernel does not isolate or restrict the dirty memory area usages, giving the attacker chances to exhaust all dirty memory, which slows down other containers significantly. Next, we discuss the attack on dirty memory.

3.3.1 Attacking dirty_throttle_control *memory dirty ratio.* The Linux kernel introduces the dirty_throttle_control struct for dirty-area control, which uses the dirty field to represent the whole kernel-space dirty ratio. Whenever the dirty value is too high, the kernel wakes up background threads to sync the dirty memory to disk. However, in the meantime, as the dirty ratio is too high, the kernel blocks the write-back and converts all writes to write-through, which slows down the write performance dramatically.

Unfortunately, the kernel does not provide any isolation for the memory dirty ratio. Any process can impact the global memory dirty ratio. In our attack, the attacker-container uses the dd command to generate files, which quickly occupies all dirty memory, reaching the memory dirty ratio limit. As a result, all writes from the host-machine or the victim-container are converted to write-through, which dramatically downgrades the performance. In our experiments, the performance of command dd if=/dev/zero of=/mnt/test bs=1M count=1024 on the victim-container drops from 1.2 GB/s to 32.6 MB/s due to the attack, resulting in 97.3% slow down. Besides, even the privileged root user on the host-machine also has a 96.1% performance downgrade.

Note that the currently Linux kernel has no namespaces related to memory management, and memory control groups are used to limit the memory usage instead of the memory dirty ratio. Therefore, it cannot defend against the attacks on memory dirty ratio.

3.4 Attacks on Storage Management

The operating system kernel abstracts the disk or other secondary storage as the file and introduces various file-related abstract resources. In fact, the storage management in the Linux kernel is complicated, which involves thousands of functions and data structures. In our experiment, we find that 133 storage-related abstract resources are reachable from container processes. Unfortunately, the kernel does not provide any namespaces or control groups to isolate or restrict the usage of these abstract resources. As a result, the attacker-container can exhaust these abstract resources to launch DoS attacks against other containers on the shared kernel. Next, we illustrate how a malicious container can exploit the file limit variable nr_files for the DoS attacks.

3.4.1 Attacking nr_files*.* As mentioned in §3.1, nr_files is a global variable in the Linux kernel, which counts the total number of opened files in the kernel. More specifically, for each allocated struct file, the kernel increases nr_files by one, as shown in lines 17-19 of Figure 1. Unfortunately, nr_files is shared among all processes. It is neither isolated by namespaces nor restricted by any control groups. As a result, the attacker-container can easily exhaust nr_files to achieve DoS attacks. To verify the feasibility of this attack, our attack-container spawns hundreds of processes, each of which opens 1,024 files. Consequently, nr_files reaches its limit. As a result, on both the host-machine and the victim-container, all file-open operations fail, and the kernel issues a warning of "Too many open files in system."

Our attack confirms that even with a few hundred of processes, the attacker is able to exhaust nr_files. While for usability, PIDs control group usually allows thousands of processes. Therefore, even with the PIDs control group enabled, the attacker-container can still DoS-attack nr_files successfully. Even worse, nr_files is shared among all processes including root and non-root processes. Therefore, not only are the non-privileged container processes impacted, the root process on the host-machine cannot perform any file-open operations either.

3.5 Attacks on IO Management

The IO management is an essential part of an operating system. For management convenience, the Linux kernel abstracts IO devices into /dev files and introduces abstract resources, such as tty_struct, to implement the IO device management. Similar to the previous cases, these abstract resources are not isolated or limited by any namespaces or control groups, thus it leads to new attacks. In the following, we introduce the attacks against pty_count, which causes DoS to the SSH connection.

3.5.1 Attacking pty_count*.* The Linux kernel abstracts the pseudo-terminal (abbreviated as pty) to /dev/ptmx and /dev/pts [46]. At the meantime, kernel also uses a global variable called pty_count to count the total number of the opened pseudo-terminal, which increases by one for each time /dev/ptmx is opened, as shown in line 6 of Figure 3. However, the kernel does not provide any namespaces or control groups to isolate or limit pty_count usages. Consequently, the attacker can easily exhaust the pty_count.

In our experiments, the attacker keeps opening /dev/ptmx in the container to trigger ptmx_open, which calls devpts_new_index and

```
static atomic_t pty_count = ATOMIC_INIT(0);

int devpts_new_index(struct pts_fs_info *fsi)
{
  int index = -ENOSPC;
  if (atomic_inc_return(&pty_count) >= (pty_limit -
  ↪  (fsi->mount_opts.reserve ? 0 : pty_reserve)))
    goto out;
    ...
  return index;
}

static int ptmx_open(struct inode *inode, struct file *filp)
{
    ...
    index = devpts_new_index(fsi);
    ...
}
```

Figure 3: Linux kernel source of `pty_count` usage. `pty_count` is global atomic variables, shared by all containers on the same kernel.

increases `pty_count`. In a couple of seconds, the `pty_count` limit is reached, and all the following `ptmx_open` operations fail. The consequences are severe as pty devices are widely used by various applications such as SSH connection. As a result, all SSH connection attempts to any other container fail due to the failed pseudo-terminal-open. Even worse, the host-machine cannot start any new containers, as the connections to a new container are denied due to the same error.

3.6 Attacking FreeBSD and Fuchsia Kernels

The root cause of abstract resource attacks is the shared kernel data (i.e., abstract resources). Next, we demonstrate that the shared kernel data also makes both the FreeBSD and the Fuchsia vulnerable to abstract resource attacks.

Attacking FreeBSD. In FreeBSD kernel, following similar resources in the Linux kernel, we manually identified 5 shared globally abstract resources, namely, `dp_dirty_total`, `numvnodes`, `openfiles`, `pid`, and `pty`. Our experiments further confirm that the former two can be DoS attacked, while the latter three are limited by `rctl` per-jail.

The experiments are conducted on the FreeBSD 13.0-RELEASE with Ezjail-admin v3.4.2 running in a virtual machine with Intel Core i5 processor, 8GB memory, and 40GB hard disk. Ezjail [53] is a jail administration framework. The ezjail commands provide a simple way to create multiple jails using FreeBSD's jail system. Jails here are similar to the containers on the Linux. We set up two jails following the FreeBSD's handbook [18] and use `rctl` [54] to limit per Jail's resources. We use these two jails as the `attacker-jail` and the `victim-jail`, which is similar to the container setup in §3.1.

For the dirty counter `dp_dirty_total`, ZFS in FreeBSD introduces the `dsl_pool` struct for recording the data of each ZFS pool. The `dsl_pool` struct uses the `dp_dirty_total` field to represent the whole ZFS pool dirty data. When the `dp_dirty_total` reaches the limit of `zfs_dirty_data_max`, ZFS delays the upcoming writing and waits for the dirty data to be synchronized to the disk. Unfortunately, FreeBSD does not provide any isolation for the `dp_dirty_total`. In the attacker-jail, we run the command `dd if=/dev/zero of=/mnt/test bs=1M count=1024` (same with the one in §3.3) to exhaust the dirty total `dp_dirty_total`. As a result, the victim-jail has a 46% IO performance downgrade.

For the `numvnodes`, FreeBSD uses a `vnode` struct to represent a file system entity, such as a file or a directory. FreeBSD also keeps a global variable `numvnodes` to record the total number of `vnode` in the whole kernel. And the limit is in `maxvnodes`. In the experiments, we can easily exhaust the host-machine's `numvnodes` and reach the `maxvnodes` limit by repeatedly creating directories in the attacker-jail.

Attacking Fuchsia. Fuchsia uses the Zircon kernel, which introduces the concept of *handle* to allow user-space programs to reference kernel objects [19]. Zircon maintains a global data structure called `gHandleTableArena` for allocating all handles. The limit for handles in the kernel is in `kMaxHandleCount`. Handles are used very frequently in Zircon. Surprisingly, we find that the creation of handles is not restricted. We further confirm this problem on the Fuchsia emulator. A user with basic rights [20] (similar to capabilities in the Linux) can repeatedly create handles to exhaust all handles, which leads to the whole system crash. We report this problem to the Fuchsia developers. They have confirmed this problem, and plan to fix the problem after identifying more attack vectors to local DoS.

3.7 Summary

From the above discussions, it is easy to see that abstract resource attacks are highly practical and the consequences are severe. What makes things worse is that abstract resources are pretty common in the Linux kernel, affecting every aspect of Linux functionalities. Furthermore, abstract resource attack is intrinsic to OS-level virtualization. It also works on FreeBSD and Fuchsia kernels.

4 STATIC ANALYSIS OF CONTAINER-EXHAUSTIBLE ABSTRACT RESOURCES

As mentioned before, abstract resources are critical to containers. On the other side, there are thousands of abstract resources, which makes it virtually impossible to enumerate all of them. In this paper, we take an initial step to identify exhaustible abstract resources shared by containers.

Challenges. We need to resolve two challenges. First, it is challenging to identify meaningful abstract resources, especially those that are shared in the kernel. An abstract resource in the Linux kernel can be a variable or a data structure instance. However, not all variables or data structure instances are meaningful abstract resources. We need to find the abstract resources that are critical to the OS functionalities. Moreover, the identified abstract resources need to be shared between containers so that one container can exhaust these resources to attack other containers. Unfortunately, there is no documentation regarding shareable abstract resources. To address this challenge, we propose *configuration-based analysis* and *access-based analysis* to identify various shared abstract resources in the Linux kernel.

Second, it is challenging to decide if the container can exhaust a specific abstract resource. Different from regular user-space programs, resource accesses from a container face more restrictions, such as namespaces, control groups, and seccomp. Moreover, as each container runs in a separate user, its resource consumption is also restricted by the per-user limitation. Thus the simple reachability analysis to the resource consumption sites cannot tell the

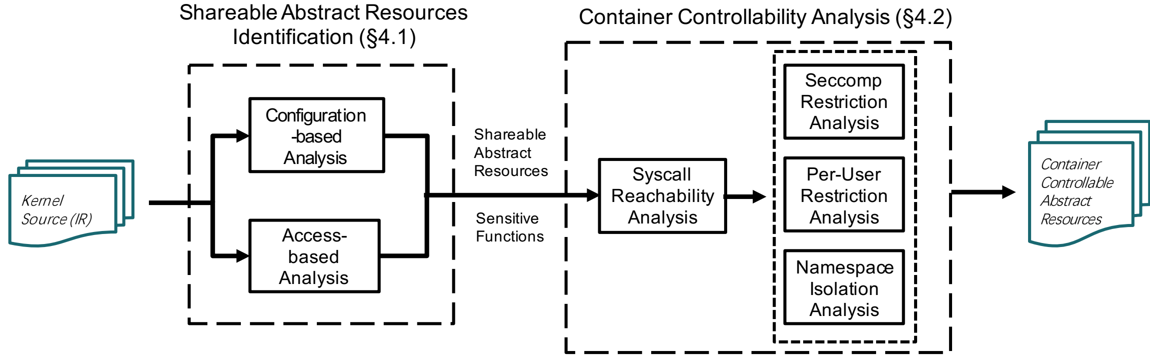

Figure 4: The architecture of the analysis tool.

controllability of the container on an abstract resource. For example, for abstract resources that are isolated by namespaces, even though the container can consume these abstract resources, it still may not affect other containers due to the namespace isolation. Therefore, to overcome this challenge, we propose *container controllability analysis*, which includes seccomp restriction analysis, per-user restriction analysis, and namespace isolation analysis, to further filter container-exhaustible resources.

Figure 4 shows the architecture of our tool, which automatically identifies container-exhaustible abstract resources. The analysis tool takes kernel source IR as the input. It first identifies all the kernel shareable abstract resources using configuration-based analysis and access-based analysis in §4.1. Then, it conducts the syscall reachability analysis and container restriction analysis in §4.2, which includes seccomp, per-user and namespace restriction analysis, to analyze the container controllability over these abstract resources. Moreover, we give out the analysis results in §4.3.

4.1 Identification of Kernel Shareable Abstract Resources

As mentioned before, it is challenging to identify meaningful abstract resources from thousands of kernel variables and data structure instances. Even harder, to make sure these abstract resources are directly or indirectly shared between containers, we need to narrow them down to the shareable kernel abstract resources.

To overcome this challenge, we leverage kernel programming paradigms and propose configuration-based analysis and access-based analysis to identify kernel shareable resources.

4.1.1 Configuration-based Analysis. The Linux kernel provides `sysctl` interfaces under `/proc/sys` to allow user-space programs to configure kernel parameters [49]. Our key observation is that most of these sysctl configurations are used for abstract resource limiting, such as limiting the file number `fs.file-nr` or memory huge pages `vm.nr_hugepages`. As a result, all containers are sharing the same global limit specified by sysctl configurations. Such sysctl configurations offer important clues about the abstract resources that are shareable between containers.

Based on the above observation, we propose to identify the shareable kernel abstract resources using the sysctl configurations, termed as the configuration-based analysis, which consists of three

① identify sysctl struct

```
1  static struct ctl_table fs_table[] = {
2    ...
3    {
4      .procname     = "file-nr",
5      .data         = &files_stat,
6      .proc_handler = proc_nr_files,
7    },
8    ...
9  }
10
11 int proc_nr_files(...)
12 {
13   files_stat.nr_files = get_nr_files();
14   ...
15 }
16
17 static long get_nr_files(void)
18 {
19   return percpu_counter_read_positive(&nr_files);
20 }
21
22 struct file *alloc_empty_file(int flags, ...)
23 {
24   ...
25   if (get_nr_files() >= files_stat.max_files &&
   ↪     !capable(CAP_SYS_ADMIN)) {
26       ...
27       goto over;
28   }
29   ...
30 }
```

② identify critical variable

③ check critical variable usages

Figure 5: The sysctl data structures in Linux kernel.

basic steps. First, it uses the specific sysctl data types to identify all sysctl-related data structures. These data structures contain the configurable sysctl kernel parameters. Second, the sysctl data structure usually contains the function that displays the sysctl value in `/proc/sys/` folder. Therefore, by analyzing that function, we are able to pinpoint the exact variable for this kernel parameter. Finally, if a kernel parameter is used for restricting resource consumption, its corresponding variable should appear in comparison instructions. Therefore, we follow the use-def chain to check the usages of the identified variable and mark it as an abstract resource if it is used in a comparison instruction.

We design and implement an inter-procedural analysis pass in LLVM. We use an example in Figure 5 to illustrate the details. Specifically, the Linux kernel uses the type `struct ctl_table` to configure sysctl kernel parameters, such as the file system configurations in `fs_table` shown in line 1 of Figure 5. Therefore, the pass first

traverses all kernel global variables to collect all `struct ctl_table` variables, such as `fs_table` in Figure 5.

Second, `fs_table` uses the function pointer in `proc_handler` to display the parameter in the `/proc/sys/` file system. Therefore, from the `proc_handler` field, the pass follows its points-to and launches an inter-procedural analysis to obtain the exact variable, whose value is displayed in the sysctl configuration interface. As shown in line 19 of Figure 5, our pass marks `nr_files` as the critical variable.

Third, our pass checks all usages of identified critical variables. If one critical variable is used in a comparison instruction (i.e., `icmp` in LLVM IR), our pass records the locations and marks this variable as the abstract resource. For example, `nr_files` is used for comparison in line 25 of Figure 5. Our pass further detects that if the comparison fails, an error is returned in lines 25 and 27. Therefore, our pass marks `nr_files` as an abstract resource. By analyzing all `struct ctl_table` structures, our pass gets a collection of abstract resources.

4.1.2 Access-based Analysis. Besides `sysctl` configurations, the Linux kernel also uses lock or atomic mechanism to protect the concurrently-accessed resources. Therefore, we propose to use concurrent accesses as an indication to identify a set of shareable abstract resources.

As the race condition and concurrency analysis is an old topic, we adopt the existing lockset detection approaches [5, 68]. If the lock is taken on a field of a data structure, we mark this data structure as an abstract resource and add this function into the sensitive function set. Moreover, if a variable is modified quantitatively between the lock and unlock functions, we also mark it as the abstract resource.

Besides the lock/unlock, we observe that `atomic` and `percpu counter` are also used to protect concurrently-accessed data, such as `percpu_counter_inc` (line 19 in Figure 1) and `atomic_inc_return` (line 6 in Figure 3). Therefore, we implement a pass to analyze all atomic and percpu counter usages. Our pass first analyzes the function parameters, and adds all functions with `struct atomic_t`, `struct atomic64_t`, and `struct percpu_counter` parameters to an atomic/percpu function set. Second, our pass traverses all statements in all kernel functions to check all usages of atomic/percpu functions. If a variable is passed to an atomic/percpu function, we mark it as an abstract resource.

During the implementation, we find that the LLVM linker merges structure types that have the same memory layout, such as `typedef struct {int counter;} atomic_t` and `typedef struct {uid_t val;} kuid_t`. The reason is that `uid_t` is of type `unsigned int`, which has the same size as `int`. Therefore, the LLVM linker merges them and mis-uses `kuid_t` for `atomic_t`. To address this problem, we trace the LLVM linker and find that the `get` method in `lib/Linker/IRMover.cpp` compares a new type with existing types and merges them if the memory layout is same. Therefore, we disable the merging by commenting out the comparing and merging code.

4.2 Container-Controllability Analysis

With identified abstract resources, we propose container controllability analysis to make sure that the container can actually consume those abstract resources. Our idea of the container controllability analysis is two-fold. First, we need to make sure the abstract resource consumption sites from §4.1 can be reached by the container processes. To achieve this, we perform the traditional backward control-flow analysis based on the kernel control flow graph, in which indirect calls are resolved based on struct-type [42, 70]. If there are no paths from system call entries to the abstract resource consumption sites, we mark this abstract resource unreachable from the container.

Second, note that reachability analysis alone is not enough, we need to further make sure that there are no additional container-specific restrictions on the path. In other words, we need to check if there are any restriction checks on the paths to make sure that the container can exhaust these abstract resources. As mentioned before, different from user-space programs, the container faces more restrictions such as seccomp, namespaces, control groups as well as per-user resource limitations. Since our reachability analysis is standard, in the following, we focus on the restriction analysis.

Seccomp Restriction Analysis. Seccomp is a mechanism used for system-call filtering. Our restriction analysis against seccomp is as follows. In our implementation, we use Docker default seccomp profile [15], which blocks more than 50 system calls. Among all the paths from system call entries to the resource consumption sites, we filter out paths that originate from any blocked system calls.

Per-User Restriction Analysis. In a real deployment, the containers are usually running as different users. Thus, the resource consumption from each container is also restricted by the per-user resource quotas. For example, Linux provides the user-limits command `ulimit` for limiting resource consumption of a specific user [50]. While the underlying implementation of ulimit is using rlimit [39, 45] to set multiple per-user resource quotas.

Besides ulimit, Linux also provides interfaces that allow users to leverage PAM (Pluggable Authentication Module) [63] to deploy per-user quotas. The PAM uses the `setup_limits` function [64] to set per-user resource quotas, which calls `setrlimit` to configure multiple rlimit constraints. For the resources limited by ulimit, rlimit and the PAM, the attacker-container cannot consume beyond the per-user quotas. As a result, it cannot fully control those abstract resources to launch DoS attacks. As both the ulimit and the PAM use rlimit to set per-user resource quotas, we need to analyze rlimit and filter out the abstract resources restricted by it.

For rlimit analysis, our key observation is that a rlimit value is usually specified in `struct rlimit` or `struct rlimit64`. Therefore, we first traverse the kernel IR to identify all variables that are loaded from `struct rlimit` or `struct rlimit64`. And then, we perform data-flow analysis to follow all the propagation and usages of these variables and mark those functions if they are used in any comparison instructions. In these functions, rlimit is checked to limit certain resources. We consider those resources not exhaustible by the attacker-container, therefore we filter out the paths based on these functions. Our tool identifies 40 functions that check rlimit.

Namespace Isolation Analysis. As mentioned before, the Linux kernel introduces namespaces for resource isolation. For a namespace isolated resource, the Linux kernel creates a "copy" for it under each namespace so that the modification in one namespace does not affect other namespaces. Therefore, to confirm container controllability, we need to make sure that those abstract resources are not protected by namespaces. Here, the challenge is that even though Linux has documentation about namespaces, there are no specifications about which abstract resources are isolated by namespaces.

Table 1: Summary of static analysis results. Res. is short for resources; Reachable is container-reachable abstract resources. Limited is per-user and namespace limited resources. Manual is manually filtered out resources. CC Res. means container controllable resources.

Res. Type	Reachable	Limited	Manual	CC Res.
Proc.	526	72	136	318
Mem.	110	10	26	74
Storage	256	57	66	133
IO	952	203	264	485
Total	1,844	342	492	1,010

Therefore, we propose namespace isolation analysis to identify the abstract resources protected by namespaces systematically.

Our key observation is that for a namespace-isolated resource, the corresponding data structure has a pointer field that points to the namespace it belongs to. Therefore, our tool first traverses all fields of each data structure type in the kernel. If the type has a namespace pointer, we mark it as an isolated resource. Second, for the identified isolated resources, our tool uses it to filter the shared abstract resources identified in §4.1.

Note that some namespace-isolated resources may still be vulnerable to abstract resource attacks due to the mapping between different namespaces. As mentioned in §3.2.2, `idr` is isolated by `pid_namespace->idr`. However, each `idr` allocated in a non-root PID namespace is mapped to a new `idr` in the root PID namespace, so that the root namespace can manage it. As a result, the root PID namespace is globally shared by all containers in all PID namespaces. Therefore, it is still vulnerable to the `idr` exhaustion attacks. In our analysis, we manually filter out these resources.

4.3 Analysis Results

We implement our analysis tool with about 2,500 lines of C++ code in LLVM 12.0. The Linux kernel IR is generated based on the latest Linux stable version v5.10 with `defconfig`. The results are shown in Table 1. In particular, by applying the configuration-based analysis and the access-based analysis, together with the reachability analysis from system calls and the seccomp restriction analysis, our tool identifies 1,844 shared abstract resources that are reachable by containers.

Resource Filtering. With the per-user quota restriction and the namespace isolation analysis, our tool finds 342 resources that are limited by the rlimit or have pointers pointing to namespace structures. Those resources either have a limit check on the path or get namespaced.

We further conduct a manual analysis. Specifically, for every resource R in the identified abstract resources, we walk through all the detected modifications of R or the fields of R. If the modification is not quantitative, such as being assigned with boolean, enumeration, or string types, we mark this modification as non-quantitative. If all the modifications to R and the fields of R are non-quantitative, we mark R as non-exhaustible. Our manual analysis identifies 492 abstract resources that are non-exhaustible, as shown in Table 1. After manual analysis, there are still 1,010 abstract resources remaining.

Table 2: Summary of validation results. Res. Dir. is the directory of resources. The drivers either have hardware support, or have no hardware support. No. is resources number, Repeatedly means the resource consumption can be repeatedly triggered.

<table>
<tr><th colspan="2">Res. Dir.</th><th>No.</th><th>Repeatedly</th><th>True Postive</th></tr>
<tr><td colspan="2">Non-Driver</td><td>700</td><td>389</td><td>55.6%</td></tr>
<tr><td rowspan="2">Driver</td><td>Have HW</td><td>218</td><td>112</td><td>51.4%</td></tr>
<tr><td>No HW</td><td>92</td><td>-</td><td>-</td></tr>
</table>

Dynamic Validation. To further validate the dynamic exhaustion of these 1,010 resources, we develop a dynamic validation method for resource consumption. For each resource, we first obtain its consumption sites and the triggered system calls from the above controllability analysis. After that, we instrument those consumption sites to monitor the actual resource consumption. Next, we execute the test cases of the corresponding triggered system calls to repeatedly trigger the consumption and record the results. We leverage 1,156 test cases from the Linux Test Project (LTP) [14] and develop 177 new ones to cover more cases. We also develop scripts to automate the above steps.

We applied our dynamic validation method to test the consumption of all 1,010 resources. The results are summarized in Table 2. For the 1,010 detected resources, 700 of them are not in the driver folder, while the other 310 resources are in the driver folder, as shown in Table 2. For the 700 non-driver resources, 389 of them can be repeatedly triggered dynamically, leading to a true positive rate of 55.6%. The resources in the driver folder need to be handled specially for two reasons. First, drivers are specific to the hardware. Without the corresponding hardware, the driver code cannot be triggered dynamically. Our key observation is that most hardware-supported drivers expose specific interfaces under `/dev` or `/sys/class` folders. Based on this observation, we remove 92 resources in drivers that are not supported by our hardware. Second, the test cases provided by LTP might not cover a specific driver. To resolve this problem, we modify the LTP test cases and develop new test cases for the drivers. Among the 218 driver resources, 112 of them can be repeatedly triggered, leading to a true positive rate of 51.4%, as shown in Table 2.

Identifying container-exhaustible abstract resources is a very challenging task, as it requires the domain knowledge to trigger the exhaustion of abstract resources and it needs to assess the impacts when these resources are exhausted. In this paper, we conduct a preliminary analysis. Note that a thorough analysis and risk assessment needs help from the Linux kernel and the container community. Therefore, we plan to open source our tool and the detected abstract resources. We think it will help the Linux kernel and the container community to identify the weak spots of resource isolation and develop robust resource containment schemes.

5 ABSTRACT RESOURCE ATTACKS ON CLOUD PLATFORMS

In this section, we further evaluate abstract resource attacks on the container environments of public cloud vendors. We first present the environment setup and then give out the results.

Table 3: Summary of the abstract resources chosen from analysis results.

OS Service	Resource Name	Identification	Consumption Function	Syscall
Process	PID idr	Access	alloc_pid()	fork()
Memory	dirty ratio	Access	balance_dirty_pages()	write()
Storage	inode	Access	__ext4_new_inode()	creat()
	nr_files	Configuration	alloc_empty_file()	open()
IO	pty_count	Configuration	devpts_new_index()	open()
	netns_ct->count	Access	__nf_conntrack_alloc()	connect()
	random entropy	Access	extract_entropy()	read()

5.1 Environment Setup and Ethical Considerations

To evaluate the effectiveness of the abstract resource attack, we set up the container environments on both local and cloud platforms. The local test environment has been presented in §3.1.

Ethical Considerations. For the cloud platforms, we intend to minimize the impact of our attacks on other cloud users as much as possible. Therefore, we use a dedicated virtual server, e.g., AWS EC2, Azure VM, Google GCE, and Alibaba ECS, to conduct the experiments. In addition, we ensure that we are the only user of that server.

Moreover, most container users leverage the container orchestration systems to deploy and manage containers [36]. Therefore, we choose the most popular one named Kubernetes and leverage cloud vendors' Kubernetes services to deploy two docker containers (i.e., the attacker-container and the victim-container) on the virtual server. For strong isolation, we apply different *Kubernetes namespaces* [37] for the attacker-container and the victim-container. As mentioned in §4.2, containers are also subjected to per-user quota restrictions. To enforce the per-user quotas in our experiments, we run the attacker-container and the victim-container in separate users with the per-user quota enforced. We also discuss restrictions that can be deployed by the PAM in §6.

Amazon AWS. For the container services, we use Elastic Kubernetes Service (EKS) [2] to deploy two container instances on an EC2 instance. The EC2 instance contains 4 CPUs, 8 GB memory, and 20 GB SSD disk. During the container deployment, we surprisingly find that the "Amazon EKS default pod security policy" uses `eks.privileged` as the default pod security policy [3]. Note that this policy allows containers to run as a privileged user and also allows privilege escalation as well as host network accesses.

To better demonstrate the effectiveness of our proposed attack, we adopt a stronger security policy from our local test environment to EKS containers, which runs containers in non-root users, drops all privileges, enables all namespaces and control groups, and uses docker seccomp profile [15] to block 50+ sensitive system calls including `ptrace`, `pivot_root`, etc. And we apply the same security policy for both the attacker-container and the victim-container.

MS Azure. We use Azure Kubernetes Service (AKS) [51] to deploy two container instances on an Azure virtual machine. The Azure VM contains 2 CPUs, 8 GB memory, and 120 GB disk. To improve the security of the deployed containers, Azure provides best practices for pod security policy in AKS [52], which runs a container in the non-root user by setting `runAsUser:1000` in `yaml` file, and it denies privilege escalation by setting `allowPrivilegeEscalation: false`. However, it still adds two capabilities, i.e., `CAP_NET_ADMIN` and `CAP_SYS_TIME`, and does not enforce seccomp.

Same as the AWS settings, we adopt a tighter security policy for containers on AKS. In addition to the best practices suggestions (i.e., non-root user and disallowing privilege escalation), we run AKS containers in non-root users, drop all capabilities, enable all namespaces and control groups, and use docker seccomp profile [15] to block 50+ sensitive system calls. And we apply the same security policy for both the attacker-container and the victim-container.

Google Cloud. For the container services, we choose the Kubernetes and use Google Kubernetes Engine (GKE) [27] to deploy two container instances on a Google Compute Engine instance [28]. The Google Compute Engine (GCE) instance we use contains 4 CPUs, 16 GB memory, and 100 GB SSD. More specifically, we apply one GCE instance and deploy two containers (i.e., the attacker-container and the victim-container) based on the regular runtime on that GCE instance.

For the container deployment, we follow the GKS container setup wizard. Google Cloud provides best practices for operating containers [29], which suggests avoiding privileged containers. Therefore, in `securityContext` of the `yaml` configuration file, we disallow the privileged escalation, run the container as a non-privileged user, and drop all the capabilities. The GKS setup wizard enables 6 namespaces and 13 control groups by default. Besides, we apply the docker default seccomp profile to filter out sensitive system calls.

Furthermore, the GKE also offers Google's secure container runtime—gVisor [31], which leverages a user-space kernel named Sentry, to serve the system calls from applications. Sentry calls about 50 system calls of the host machine to provide services as needed. gVisor is regarded as a secure sand-boxed runtime for containers [31]. For the container deployment based on gVisor, all its security settings (including non-privileged user, dropping capabilities) are the same as the GKE docker runtime settings.

Alibaba Cloud. For the container services, Alibaba Cloud provides Elastic Container Instance, Container Service for Kubernetes, Container Registry, and Alibaba Cloud Service Mesh [1]. We use the Container Service for Kubernetes to deploy two container instances on an Elastic Computing Service (ECS) instance. The ECS instance contains 4 CPUs, 16 GB memory, and 120 GB SSD disk. For container security, we follow the official guide for container service deployment [11], which runs containers with non-root user by setting `runAsUser` to 1000. However, it does not disallow privilege escalation and does not enforce seccomp and SELinux either.

We adopt a stronger security policy, which is the same as previous ones. We run containers in non-root users, drop all capabilities, enable all namespaces and control groups, and use docker seccomp profile [15] to block sensitive system calls. And we apply the same security policy for both the attacker-container and the victim-container.

5.2 Selection of Abstract Resources

To conduct the attacks, we need to select meaningful abstract resources. To demonstrate the effectiveness of abstract resource attacks, we want to select abstract resources that affect each aspect of the operating system services, including process management, memory management, storage management, and IO management. Therefore, we first classify all the identified resources into these four categories, i.e., for process, for memory, for storage, and for IO management, according to their declaration locations. Then, we pick at least one resource from each category based on our domain knowledge, i.e., we know the impacts of resource exhaustion.

Eventually, we select 7 abstract resources covering all four aspects, as shown in Table 3. The resource names are listed in the second column of Table 3. Among the selected abstract resources, 5 of them (i.e., PID idr, dirty ratio, inode, netns_ct->count, and random entropy) are identified by the access-based analysis, and the other 2 (i.e., nr_files and pty_count) are identified by the configuration-based analysis, as shown in the third column of Table 3. We also list the resource consumption functions in the fourth column and the system calls we can use to trigger the attacks in the last column of Table 3.

5.3 Attacking Results on Cloud Platforms

As mentioned in the previous session, we set up 5 test environments for our proposed attack, including the ones on local, AWS, Azure, Google Cloud, and Alibaba Cloud. For each test environment, we set up two containers with tight security policies, as the *attacker-container* and *victim-container*. The attacker-container launches attacks targeting certain abstract resources. We use the above 7 selected abstract resources to launch the attacks. A benchmark is running on both the victim-container and the host-machine to measures their performance downgrade under abstract resource attacks. The results are shown in Table 4.

The PID `idr` attack. The PID `idr` attack and its root cause have been detailed in §3.2.1. For the PID attack on the vendors, all victim containers and even the host-machine in Local, AWS, Azure, and Google test environments cannot fork new processes. The victim containers even get evicted. Alibaba Cloud is not vulnerable to the PID attack.

The dirty ratio attack. The dirty ratio attack has been discussed in §3.3.1. Without the attack, the IO performance is regarded as 100%. Under the dirty ratio attack, the IO performance of victim-container on AWS, Azure, and Alibaba Cloud drop to 6.3%, 1.2%, 6.7%, respectively. Even worse, the host-machine is also vulnerable to this attack, while its IO performance drops to 8.3%on AWS and drops to 8.6% on Alibaba Cloud. Here MS Azure does not provide any access to the host machine, so we cannot get Azure host IO performance. Google Cloud is not vulnerable to the dirty ratio attack.

The inode attack. In the inode attack, the victim-container keeps allocating inode structures. Unfortunately, the mount namespace does not isolate the inode. Neither the Linux kernel provides any inode related control groups. As a result, all inodes on the partition are exhausted. All operations consuming inodes fail, including the ones from the victim-container or the host machine. In our experiments, Alibaba Cloud is vulnerable to the inode attack. The victim-container even gets evicted. Moreover, the host-machine cannot create any new files either.

The `nr_files` attack. The `nr_files` attack has been discussed in §3.4.1. `nr_files` is globally shared by all containers. There are no namespaces or control groups to limit its usages. With `nr_files` quota exhausted, various operations fail, including file open, executing a new program, pipe creation, socket creation, and the timer creation, as everything in Linux is a file. Our experiment shows all of the top 4 vendors are vulnerable to the `nr_files` attack.

The `pty_count` attack. The `pty_count` attack has been discussed in §3.5.1, which uses up all open pseudo-terminals quota. As a result, all operations that need to open a new pseudo-terminal fail, such as SSH connections. Unfortunately, all of the top 4 vendors are vulnerable to the `pty_count` attack.

The `netns_ct->count` attack. Netfilter in the Linux kernel provides connection tracking functionalities, which keeps track of all logical network connections [66]. While the total connection has a limit, and it is counted by `struct netns_ct->count` [34]. Both the host machine and the containers need to maintain the connections. Even though the containers are in the different `net` namespace, all of their connections need to consume the `init_net.ct.count` [35] of the init net namespace of the host machine. Therefore, if one can generate a large number of TCP connections in a short time, it can consume all quota of `init_net.ct.count`, causing Netfilter malfunction. In our experiments, the attacker-container can exhaust `init_net.ct.count` in a few seconds, which causes random packet dropping. Again, all environments of the top 4 vendors are vulnerable to the `struct netns_ct->count` attack.

The random entropy attack. In the Linux kernel, every read to the `/dev/random` consumes the random entropy. Whenever the random entropy drops below a threshold, the Linux kernel blocks read operations to `/dev/random` and waits for the entropy to increase [41]. As there are no namespace or control groups to isolate the random entropy, the attacker-container can easily consume all random entropy by repeatedly reading `/dev/random`, and lead to benign reads blocked. The latest Linux kernel v5.10 fixed this issue by redirecting `/dev/random` reads to `/dev/urandom`. However, both Azure and Alibaba Cloud are vulnerable to this attack.

5.4 Attacking gVisor

We also conduct the 7 resource attacks on gVisor. To set up gVisor environment, we select runsc, instead of runc, as the container runtime in Google Kubernetes Engine (GKE), as mentioned in §5.1. Among the 7 attacks, two attacks, i.e., `nr_files` attack and `netns_ct->count` attack, still work in the gVisor environment. In the following, we present our analysis to show why these two attacks work on gVisor.

For the `nr_files`, gVisor uses Sentry to serve syscalls and Gofer to handle different types of IO for the Sentry. Sentry intercepts

Table 4: Summary of the attack results on different environments. "Y" indicates a successful attack, "-" indicates a failed attack.

Abstract Resources	Local	AWS	Azure	Google	Alibaba	Attacking Results
PID idr	Y	Y	Y	Y	-	Fork fail, victim container is evicted
dirty ratio	Y	Y	Y	-	Y	IO performance down for over 90%
inode	Y	-	-	-	Y	Victim container gets evicted
nr_files	Y	Y	Y	Y	Y	Operations requiring open-new-file fail
pty_count	Y	Y	Y	Y	Y	New SSH connections are rejected
netns_ct->count	Y	Y	Y	Y	Y	Random packets dropping
random entropy	Y	-	Y	-	Y	/dev/random read blocked

the open syscall from the container and sends the request to Gofer. On the other side, Gofer handles the request by calling the openat syscall of the host OS. Eventually, the openat syscall on the host OS triggers the alloc_empty_file function, which consumes the nr_files. In this way, the attacker in gVisor is able to exhaust the nr_files of the host machine.

For the netns_ct->count, Sentry intercepts the connect syscall and uses its own network stack to forward the data packets to the veth-peer network card created in the host. The veth-peer is attached to the virtual bridge in the host. When a network frame is forwarded via a virtual bridge, the netfilter on the host is triggered to call the nf_conntrack_alloc function, which in turn consumes the netns_ct->count. Therefore, attackers in gVisor still can exhaust the netns_ct->count of the host machine.

5.5 Summary

For the self-deployed shared-kernel container environments, two of them are vulnerable to 6 attacks, one is vulnerable to 5 attacks, and the other one is vulnerable to 4 attacks. Surprisingly, gVisor runtime is also vulnerable to 2 attacks—the nr_files attack and the netns_ct->count attack. We have reported these attacks to all the four vendors. All of them confirmed that the problems exist in their shared-kernel container environments.

Though the top vendors use virtual machines to isolate the containers for different tenants, abstract resource attack is still practical for several reasons. First, as demonstrated on Linux, FreeBSD, and Fuchsia, abstract resource attack is intrinsic to OS-level virtualization and thus is a broad class of attacks. Second, inexperienced users may not understand the risks of the shared-kernel and may use containers for sand-boxing [62]. Our paper would help to improve the awareness of the risks. Third, even within the same tenant, the competing teams might attack each other by exploiting abstract resources. Therefore, it is still necessary to monitor and mitigate such attacks.

6 MITIGATION DISCUSSIONS

In this paper, we reveal that other than physical resources, containers are also sharing the abstract resources of the underlying running kernel. These abstract resources are easy to attack and the consequences are severe. In the following, we give out multiple strategies for mitigating the risks introduced by abstract resources.

Using PAM for per-user quota restrictions. As mentioned in §4.2, the Linux kernel provides interfaces allowing the user to load user-customized PAM. PAM is able to limit 18 resources, 5 of which are for abstract resources, including maxlogin/maxsyslogins, nofile, nproc, and sigpending [48]. From our communication with the cloud vendors, we are not aware that any cloud vendors adopt PAM. Therefore, it is suggested to use PAM for certain abstract resource restrictions.

Using VM for strong isolation. For the security-critical applications, we suggest not using the multi-tenancy container environments. Stronger isolation schemes, such as the virtual machine-based virtualization, are more preferable.

Using Monitoring Tools. We recommend to use the monitoring tools for Kubernetes clusters, such as Falco [61], to monitor the resource consumption of containers. For sensitive abstract resources such as nr_files and inode, users should customize their own rules to monitor specific resource consumption in the system.

Improving current isolation design. For the existing namespaces, such as PID namespace, due to the mapping to the root namespace design, it cannot defend against the resource exhaustion attacks. As detailed in §3.2.2, the Linux kernel allocates an extra idr in the root PID namespace for any idr allocated in other PID namespaces. As a result, the root PID namespace is still globally shared. The attacker can still easily exhaust the PID in the root PID namespace, causing DoS attacks. For the similar reason, nf-conntrack count netns_ct->count can be attacked even if it is isolated by network namespace. Therefore, Linux community needs to review the namespace design, eliminating the namespace dependencies to improve the isolation.

New kernel containment mechanisms. The Linux kernel community and the container community need to put more effort into the protection of abstract resources. Actually, we report this problem to the Docker security team. The feedback is that "Linux containers can only use available kernel isolation mechanisms. If there are no kernel mechanisms to control the limit, the container cannot do anything to restrict it". Therefore, we first need a thorough analysis of all container shareable abstract resources, so that we can understand and more importantly, clear up their data dependencies. This requires comprehensive kernel domain knowledge and substantial kernel code changes. Moreover, the Linux kernel is not initially designed for supporting OS-level virtualization. Its resource isolation and containment are incomplete. Therefore, new namespace and control groups are needed.

More restrictive system call blocking. From the container side, currently, even with seccomp enforced, the applications in the containers can still access about 250 system calls. Before we understand the data dependency of those system calls, it is suggested to enforce a more strict seccomp profile to block more unnecessary system calls. The container users can use techniques in [13, 25, 26, 38, 55]

to get tighter seccomp profiles, to reduce the potential of abstract resource attacks.

7 RELATED WORK

In this section, we present the studies that are related to virtualization, resource isolation, and container security.

7.1 Virtualization Techniques

There are two mainstream virtualization techniques used in the cloud environment, VM-based virtualization and OS-level virtualization. Compared with the VM-based virtualization, OS-level virtualization is becoming popular for enabling full application capability with light-weight virtualization. To fully understand the performance advantages, researchers have conducted a series of studies. Felter et al. show that Docker can achieve better performance than KVM in all cases by using a set of benchmarks covering multiple resources [17]. Joy et al. make a comparison between Linux containers and virtual machines in terms of performance and scalability [32]. Zhang et al. show that the container has better performance than virtual machines in big data environment [69].

All these works demonstrate that OS-level virtualization has better performance than traditional VM-based virtualization. However, none of them pay attention to the potential influence of underlying kernel abstract resources. Our paper reveals the new attack surfaces introduced by abstract resources.

7.2 Resource Isolation

Linux uses capabilities [43] to prohibit processes without certain capabilities from accessing resource instances of corresponding types. Researchers have proposed approaches that are based on Linux capabilities, such as Wedge [7], Capsicum [65], and ACES [10]. These works enforce more fine-grained capability control to mitigate memory corruption attacks. However, they cannot defend against our DoS attacks which exhaust accessible shared resources.

Memory address space isolation [56] is a typical resource space isolation scheme, which avoids memory address resource from being exhausted. Linux namespaces [47] isolate 8 types of resources listed in §2.1. These schemes can isolate only limited types of resources. Resource containers [6] propose to extend monolithic kernel to isolate system resources and account for resources at thread-level, which is similar to control groups. Using resource containers to protect all abstract resources is impractical due to the large performance overhead. EdgeOS [57] deploys OS with strong isolation for edge clouds. However, adopting a micro-kernel without hardware supporting introduces more overhead than a monolithic kernel. Faasm[58] uses software-fault isolation (SFI) for memory isolation while uses namespaces to isolate the network resource space in server-less computing. However, most shared resources are still exposed to the threat of DoS attacks.

7.3 Container Security

Besides resource isolation, there are studies on container security. Gao et al. find that information leaks from `/proc` or `/sys` can be exploited to launch power attacks [22]. While the same research group also conducts five attacks to generate out-of-band workloads to break the resource constraints of Linux control groups [23]. However, they mainly focus on information leakage problem or attacking physical resources such as CPU, IO, not abstract resources.

Lin et al. show that containers cannot isolate kernel vulnerabilities [40]. Another work uses static analysis to analyze Docker's code in order to find differences between the vulnerable and the patched code [16]. However, these works focus on existing vulnerabilities and exploits. On the contrary, our work introduces new attacks targeting the shared abstract resources.

There are also works on securing containers. Lei et al. propose a container security mechanism called SPEAKER to reduce the application's available system calls inside container [38]. Sun et al. develop security namespaces that provide the security policy isolation for each container[60]. Another work uses Intel SGX to secure containers [4], which provides a small trusted computing base with low-performance overhead. Brady et al. implement a security assessment system of container images [8]. However, containers in all of these works still rely on the kernel for various services and thus are still vulnerable to abstract resource attacks.

8 CONCLUSION

In this paper, we reveal a new attack surface introduced by the shared-kernel in OS-level virtualization. The containers are directly and indirectly sharing thousands of abstract resources, which can be exhausted easily to cause DoS attacks against other containers. To show the importance of confining abstract resources, we have conducted abstract resource attacks, targeting abstract resources on different aspects of the operating system kernel. The results show that attacking abstract resources is highly practical and critical.

Abstract resources are inherently hard to contain. To understand the attack surfaces, we take an initial trial by conducting a systematic analysis to identify vulnerable abstract resources in the Linux kernel. Our tool successfully detects 501 dynamically triggered abstract resources, in which we pick 7 ones and conduct the attacking experiments in the self-deployed shared-kernel container environments on the top 4 cloud vendors. The results show that all environments are vulnerable to our attacks. As a mitigation, we provide several suggestions for container users and developers to reduce the risks.

ACKNOWLEDGMENTS

The authors would like to thank all reviewers for the insightful comments. Those comments helped to re-shape this paper. This work is partially supported by the National Natural Science Foundation of China (Grants No. 62002317, 62032021, and 61772236), by the National Key R&D Program of China (Grant No. 2020AAA0107700), by the Key R&D Program of Shaanxi Province of China (Grant No. 2019ZDLGY12-06), by the Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang (Grant No. 2018R01005), and by the Ant Group Funds for Security Research.

REFERENCES

[1] Alibaba. 2020. Alibaba Cloud. https://us.alibabacloud.com/.
[2] Amazon. 2020. Containers on AWS. https://aws.amazon.com/containers.
[3] Amazon. 2020. Pod security policy. https://docs.aws.amazon.com/eks/latest/userguide/pod-security-policy.html.
[4] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. 2016. {SCONE}: Secure linux containers with intel {SGX}. In *12th*

{USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16). USENIX Association, 689–703.

[5] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. 2019. Effective static analysis of concurrency use-after-free bugs in Linux device drivers. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. USENIX Association, 255–268.

[6] Gaurav Banga, Peter Druschel, and Jeffrey C Mogul. 1999. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*. USENIX Association, 45–58.

[7] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting applications into reduced-privilege compartments. In *5th USENIX Symposium on Networked Systems Design & Implementation,NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*. USENIX Association, 309–322.

[8] Kelly Brady, Seung Moon, Tuan Nguyen, and Joel Coffman. 2020. Docker container security in cloud computing. In *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, 975–980.

[9] Thanh Bui. 2015. Analysis of docker security. *arXiv preprint arXiv:1501.02967* (2015). http://arxiv.org/abs/1501.02967

[10] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. 2018. {ACES}: Automatic compartments for embedded systems. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. USENIX Association, 65–82.

[11] Alibaba Cloud. 2020. Pod security policy. https://www.alibabacloud.com/help/doc-detail/149547.html.

[12] Theo Combe, Antony Martin, and Roberto Di Pietro. 2016. To docker or not to docker: A security perspective. *IEEE Cloud Computing* 3, 5 (2016), 54–62.

[13] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P Kemerlis. 2020. Sysfilter: Automated system call filtering for commodity software. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*. USENIX Association, 459–474.

[14] LTP Developers. 2021. Linux Test Project. https://linux-test-project.github.io/.

[15] Docker. 2020. Seccomp security profiles for Docker. https://docs.docker.com/engine/security/seccomp/.

[16] Ana Duarte and Nuno Antunes. 2018. An empirical study of docker vulnerabilities and of static code analysis applicability. In *2018 Eighth Latin-American Symposium on Dependable Computing (LADC)*. IEEE, 27–36.

[17] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. 2015. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE Computer Society, 171–172.

[18] FreeBSD. 2021. freeBSD handbook. https://docs.freebsd.org/en/books/handbook/jails/.

[19] Fuchsia. 2020. Zircon handles. https://fuchsia.dev/fuchsia-src/concepts/kernel/handles.

[20] Fuchsia. 2020. ZX RIGHTS BASIC. https://fuchsia.dev/fuchsia-src/concepts/kernel/rights#zx_rights_basic.

[21] Peter B Galvin, Greg Gagne, Abraham Silberschatz, et al. 2003. *Operating system concepts*. John Wiley & Sons.

[22] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. 2017. ContainerLeaks: Emerging security threats of information leakages in container clouds. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society, 237–248.

[23] Xing Gao, Zhongshu Gu, Zhengfa Li, Hani Jamjoom, and Cong Wang. 2019. Houdini's Escape: Breaking the Resource Rein of Linux Control Groups. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. ACM, 1073–1086.

[24] Xing Gao, Benjamin Steenkamer, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. 2018. A study on the security implications of information leakages in container clouds. *IEEE Transactions on Dependable and Secure Computing* 18, 1 (2018), 174–191.

[25] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. 2020. Confine: Automated system call policy generation for container attack surface reduction. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*. USENIX Association, 443–458.

[26] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal system call specialization for attack surface reduction. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. USENIX Association, 1749–1766.

[27] Google. 2020. GKE quick start. https://cloud.google.com/kubernetes-engine/docs/quickstart.

[28] Google. 2020. google compute engine of Containers. https://cloud.google.com/compute/docs/containers.

[29] Google. 2021. Best practices for operating containers. https://cloud.google.com/kubernetes-engine/docs/best-practices/enterprise-multitenancy.

[30] Aaron Grattafiori. 2016. Understanding and hardening linux containers. *Whitepaper, NCC Group* (2016).

[31] 2020 The gVisor Authors. 2020. What is gVisor. https://gvisor.dev/docs.

[32] Ann Mary Joy. 2015. Performance comparison between linux containers and virtual machines. In *2015 International Conference on Advances in Computer Engineering and Applications*. 342–346.

[33] Poul-Henning Kamp and Robert NM Watson. 2000. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, Vol. 43. 116.

[34] Linux Kenrnel. 2020. Kernel source - nf-conntrack-core.c. https://elixir.bootlin.com/linux/v5.10/source/net/netfilter/nf_conntrack_core.c#L1480.

[35] Linux Kernel. 2020. Kernel source - nf-conntrack-standalone.c. https://elixir.bootlin.com/linux/v5.10/source/net/netfilter/nf_conntrack_standalone.c#L614.

[36] Kubernetes. 2020. Kubernetes. https://kubernetes.io/.

[37] Kubernetes. 2020. Kubernetes Namespaces. https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/.

[38] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. 2017. SPEAKER: Split-phase execution of application containers. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (Lecture Notes in Computer Science, Vol. 10327)*. Springer, 230–251.

[39] GNU C Library. 2021. ulmit source code. https://sourceware.org/git/?p=glibc.git;a=blob_plain;f=sysdeps/posix/ulimit.c.

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

[41] Linux. 2020. random read kernel function. https://elixir.bootlin.com/linux/v5.3.1/source/drivers/char/random.c#L1948.

[42] Kangjie Lu and Hong Hu. 2019. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1867–1881.

[43] Linux man-pages project. 2020. capabilities(7) — Linux manual page. https://man7.org/linux/man-pages/man7/capabilities.7.html.

[44] Linux man-pages project. 2020. cgroups - Linux control groups. http://man7.org/linux/man-pages/man7/cgroups.7.html.

[45] Linux man-pages project. 2020. getrlimit man page. https://man7.org/linux/man-pages/man2/getrlimit.2.html.

[46] Linux man-pages project. 2020. Linux pty. https://man7.org/linux/man-pages/man7/pty.7.html.

[47] Linux man-pages project. 2020. namespace - Linux Namespace. https://man7.org/linux/man-pages/man7/namespaces.7.html.

[48] Linux man-pages project. 2020. PAM limits conf man page. https://www.man7.org/linux/man-pages/man5/limits.conf.5.html.

[49] Linux man pages project. 2020. sysctl man page. https://man7.org/linux/man-pages/man8/sysctl.8.html.

[50] Linux man-pages project. 2020. ulimit man page. https://man7.org/linux/man-pages/man3/ulimit.3.html.

[51] Microsoft. 2020. Containers on Azure. https://azure.microsoft.com/en-us/product-categories/containers/.

[52] Microsoft. 2020. Security policy on Azure. https://docs.microsoft.com/azure/aks/developer-best-practices-pod-security.

[53] FreeBSD Manual Pages. 2021. ezjail man page. https://www.freebsd.org/cgi/man.cgi?query=ezjail.

[54] FreeBSD Manual Pages. 2021. rctl man page. https://www.freebsd.org/cgi/man.cgi?query=rctl&sektion=8.

[55] Shankara Pailoor, Xinyu Wang, Hovav Shacham, and Isil Dillig. 2020. Automated policy synthesis for system call sandboxing. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 135:1–135:26.

[56] James L Peterson and Abraham Silberschatz. 1985. *Operating system concepts*. Addison-Wesley Longman Publishing Co., Inc.

[57] Yuxin Ren, Guyue Liu, Vlad Nitu, Wenyuan Shao, Riley Kennedy, Gabriel Parmer, Timothy Wood, and Alain Tchana. 2020. Fine-Grained Isolation for Scalable, Dynamic, Multi-tenant Edge Clouds. In *2020 {USENIX} Annual Technical Conference ({USENIX} {ATC} 20)*. USENIX Association, 927–942.

[58] Simon Shillaker and Peter Pietzuch. 2020. Faasm: lightweight isolation for efficient stateful serverless computing. In *2020 {USENIX} Annual Technical Conference ({USENIX} {ATC} 20)*. USENIX Association, 419–433.

[59] Solaris. 2020. Solaris Zones. https://docs.oracle.com/cd/E26502_01/html/E29024/toc.html.

[60] Yuqiong Sun, David Safford, Mimi Zohar, Dimitrios Pendarakis, Zhongshu Gu, and Trent Jaeger. 2018. Security namespace: making linux security frameworks available to containers. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. USENIX Association, 1423–1439.

[61] Sysdig. 2021. Sysdig Falco. https://sysdig.com/opensource/falco/.

[62] William Viktorsson, Cristian Klein, and Johan Tordsson. 2020. Security-Performance Trade-offs of Kubernetes Container Runtimes. In *28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2020, Nice, France, November 17-19, 2020*. IEEE, 1–4. https://doi.org/10.1109/MASCOTS50786.2020.9285946

[63] Dmitry V.Levin. 2020. pam model source code. https://github.com/linux-pam/linux-pam/releases/tag/v1.5.1.

[64] Dmitry V.Levin. 2021. setup_limits source code. https://github.com/linux-pam/linux-pam/blob/v1.5.1/modules/pam_limits/pam_limits.c#L984.

[65] Robert NM Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. 2010. Capsicum: Practical Capabilities for UNIX.. In *USENIX Security Symposium*, Vol. 46. USENIX Association, 2. https://doi.org/10.1109/MASCOTS50786.2020.9285946

[66] Wikipedia. 2020. Connection tracking. https://en.wikipedia.org/wiki/Netfilter#Connection_tracking.

[67] Wikipedia. 2020. OS-level virtualization. https://en.wikipedia.org/wiki/OS-level_virtualization.

[68] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. 2018. Precise and scalable detection of double-fetch bugs in OS kernels. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 661–678. https://doi.org/10.1109/SP.2018.00017

[69] Qi Zhang, Ling Liu, Calton Pu, Qiwei Dou, Liren Wu, and Wei Zhou. 2018. A comparative study of containers and virtual machines in big data environment. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE Computer Society, 178–185. https://doi.org/10.1109/CLOUD.2018.00030

[70] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. 2019. Pex: A permission check analysis framework for linux kernel. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. USENIX Association, 1205–1220.