Houdini's Escape: Breaking the Resource Rein of Linux Control Groups

Xing Gao
University of Memphis
xgao1@memphis.edu

Zhongshu Gu
IBM Research
zgu@us.ibm.com

Zhengfa Li
Independent Researcher
zhengfali@foxmail.com

Hani Jamjoom
IBM Research
jamjoom@us.ibm.com

Cong Wang
Old Dominion University
c1wang@odu.edu

ABSTRACT

Linux Control Groups, i.e., cgroups, are the key building blocks to enable operating-system-level containerization. The cgroups mechanism partitions processes into hierarchical groups and applies different controllers to manage system resources, including CPU, memory, block I/O, etc. Newly spawned child processes automatically copy cgroups attributes from their parents to enforce resource control. Unfortunately, inherited cgroups confinement via process creation does not always guarantee consistent and fair resource accounting. In this paper, we devise a set of exploiting strategies to generate *out-of-band* workloads via de-associating processes from their original process groups. The system resources consumed by such workloads will not be charged to the appropriate cgroups. To further demonstrate the feasibility, we present five case studies within Docker containers to demonstrate how to break the resource rein of cgroups in realistic scenarios. Even worse, by exploiting those cgroups' insufficiencies in a multi-tenant container environment, an adversarial container is able to greatly amplify the amount of consumed resources, significantly slow-down other containers on the same host, and gain extra unfair advantages on the system resources. We conduct extensive experiments on both a local testbed and an Amazon EC2 cloud dedicated server. The experimental results demonstrate that a container can consume system resources (e.g., CPU) as much as 200× of its limit, and reduce both computing and I/O performance of particular workloads in other co-resident containers by 95%.

CCS CONCEPTS

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

KEYWORDS

Container; Control Groups; Docker

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3354227

ACM Reference Format:
Xing Gao, Zhongshu Gu, Zhengfa Li, Hani Jamjoom, and Cong Wang. 2019. Houdini's Escape: Breaking the Resource Rein of Linux Control Groups. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3319535.3354227

1 INTRODUCTION

Container technology has been broadly adopted in various computation scenarios, including edge computing [1], microservice architecture [2], serverless computing [3], and commercial cloud vendors [4–6]. Compared to virtual machines, the elimination of additional abstraction layers leads to better resource utilization and improved efficiency. Thus, containers can achieve near-native performance [7, 8].

Despite performance advantages, recently container techniques also raise a number of security and privacy concerns, particularly for the resource isolation [9], privilege escalation [10–12], confused deputy attacks [13], and covert channels [14].

In the Linux kernel, the two key building blocks that enable containers' resource isolation and management are *Linux Namespaces* (i.e., namespaces) and *Linux Control Groups* (i.e., cgroups[1]). In addition, a set of security mechanisms (e.g., *Capabilities*, *SELinux*, *AppArmor*, *seccomp*, and *Security Namespace* [16]) have also been adopted or proposed to further enhance container security in deployment.

Containers depend on cgroups for resource management and control to prevent one container from draining system resources of the host. The cgroups mechanism partitions a group of processes and their children into hierarchical groups and applies different controllers to manage and limit various system resources, e.g., CPU time, computer memory, block I/O, etc. With a reasonable restriction policy, cgroups can mitigate many known denial-of-service exploits [17].

In this paper, we intend to systematically explore the methods to escape the resource control of the existing cgroups mechanism, and understand the security impacts on containers. Newly created child processes automatically inherit cgroups attributes from their parents. This mechanism guarantees that they will be confined under the same cgroups policies. To break the resource rein of cgroups, we devise a set of exploiting strategies to generate *out-of-band* workloads via processes de-associated from their originating

[1]Based on the standard terminology of cgroups kernel documentation [15], we use lower case cgroup and cgroups throughout this paper.

cgroups. These processes can be created from scratch to handle system events initiated within a cgroup. In other cases, such processes can be dormant kernel threads or system service processes that are shared across the whole system and will be activated on demand. Therefore, the corresponding consumed resources will be charged to other "victim" cgroups.

To further reveal the security risks of the insufficiencies in the existing cgroups mechanism, we conduct five case studies with Docker containers showing the steps to escape cgroup resource control in realistic system settings. In these case studies, we respectively exploit the kernel handling mechanism of exceptions, file systems and I/O devices, Linux logging systems, container engines, and handling of softirqs. We conduct experiments on a local testbed and a dedicated server in the Amazon EC2 cloud. Our experiments show that, even with multiple cgroup controllers enforced, an adversarial de-privileged container can still significantly exhaust the CPU resources or generate a large amount of I/O activities without being charged by any cgroup controllers.

Even worse, by exploiting those mechanisms in a multi-tenant container environment, an adversarial container is able to greatly amplify the amount of consumed resources. As a result of mounting multiple attacks such as denial-of-service attacks and resource-freeing attacks, the adversarial container can significantly slowdown other containers on the same host, and gain extra unfair advantages on the system resources. Our experiments demonstrate that adversaries are able to significantly affect the performance of co-located containers by controlling only a small amount of resources. For instance, a container can consume system resources (e.g., CPU) as much as 200× above its limit, and reduce both computing and I/O performance of particular benchmarks of other containers by 95%. Overall, the major contributions of this work are summarized as follows:

- We present four exploiting strategies that can cause misaccounting of system resources, thus we can escape the resource constraints enforced by cgroup controllers.
- We conduct five case studies in Docker container environments and demonstrate that it is possible to break the cgroup limit and consume significantly more resources in realistic scenarios.
- We evaluate the impacts of the proposed approaches on two testbeds with different configurations. The experimental results show the severity of the security impacts.

The rest of this paper is organized as follows. Section 2 introduces the background of control groups. Section 3 presents the strategies to escape the control of the cgroups mechanism and analyzes their root causes from the kernel perspective. Section 4 details several cases studies on containers including the threat model, attack vectors, and the effectiveness of various attacks on multi-tenant container environments. Section 5 discusses the potential mitigation from different aspects. Section 6 surveys related work and we conclude in Section 7.

2 BACKGROUND

In the Linux kernel, cgroups are the key features for managing system resources (e.g., CPU, memory, disk I/O, network, etc.) of a set of tasks and all their children. It is one of the building blocks enabling containerization. The cgroup mechanism partitions groups of processes into hierarchical groups with controlled behaviors. All child processes also inherit certain attributes (e.g., limits) from their parent, and controlled by the mechanism as well. cgroups rely on different resource controllers (or subsystems) to limit, account for, and isolate various types of system resource, including CPU time, system memory, block I/O, network bandwidth, etc. Linux containers leverage the control groups to apply resource limits to each container instance and prevent a single container from draining host resources. For the billing model in cloud computing, cgroups can also be used for assigning corresponding resources to each container and measuring their usage. Below we briefly introduce the background knowledge of cgroups hierarchy and four typical types of cgroup controller which are normally applied in the existing container environment, as well as the cgroup inheritance procedure for newly spawned processes.

2.1 cgroups Hierarchy and Controllers

In Linux, cgroups are organized in a hierarchical structure where a set of cgroups are arranged in a tree. Each task (e.g., a thread) can only be associated with exactly one cgroup in one hierarchy, but can be a member of multiple cgroups in different hierarchies. Each hierarchy then has one or more subsystems attached to it, so that a resource controller can apply per-cgroup limits on specific system resources. With the hierarchical structure, the cgroups mechanism is able to limit the total amount of resources for a group of processes (e.g., a container).

The cpu controller. The cpu controller makes the CPU as a manageable resource in two ways by scheduling the CPU leveraging the CFS (completely fair scheduler, introduced in Linux 2.6.23). The first one is to guarantee a minimum number of CPU *shares*: each group is provisioned with corresponding *shares* defining the relative weight. This policy does not limit a cgroup's CPU usage if the CPUs are free, but allocate the bandwidth in accordance with the ratio of the weight when multiple cgroups compete for the same CPU resources. For example, if one container with the *shares* 512 is running on the same core with another container with the *shares* 1,024. Then the first container will get a rough 33.3% CPU usage while the other one gets the rest 66.7%.

The cpu controller was further extended in Linux 3.2 to provide extra CPU bandwidth control by specifying a *quota* and *period*. Each group is only allowed to consume up to "quota" microseconds within each given "period" in microseconds. If the CPU bandwidth consumption of a group (tracked by a *runtime* variable) exceeds the limit, the controller will throttle the task until the next period, when the container's *runtime* is recharged to its quota. The cpu controller is widely applied in multi-tenant container environment to restrict the CPU usage of one container. If a container is setup with the *quota* equal to 50,000 and the *period* equal to 100,000, then the container can consume up to half of the total CPU cycles of one CPU core.

The cpusets controller. The `cpusets` controller provides a mechanism for constraining a set of tasks to specific CPUs and memory nodes. In multi-tenant container environments, the `cpusets` controller is leveraged to limit the workload of a container on specific cores. Each task of a container is attached to a *cpuset*, which contains

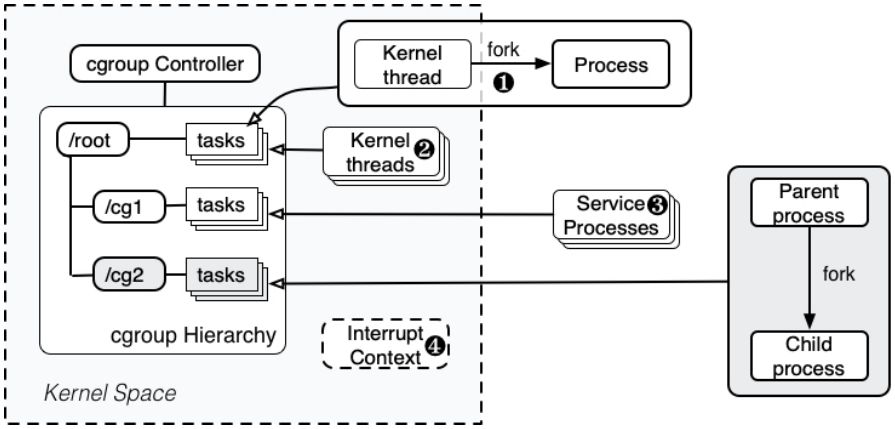


Figure 1: The overview of control groups and four exploiting strategies to generate out-of-band workloads.

a set of allowed CPUs and memory nodes. For the CPU scheduling, the scheduling of the task (via the system call `sched_setaffinity`) is filtered to those CPUs allowed by the task's *cpuset*. Any further live migration of the task is also limited to the allowed *cpuset*. Thus, the `cpusets` controller can also be used to pin one process on a specific core. The container user can also utilize user-space applications (e.g., taskset) to further set the affinities within the limit of *cpuset*.

The blkio controller. The `blkio` cgroup controls and limits access to specified block devices by applying I/O control. Two policies are available at the kernel level. The first one is a time-based division of disk policy based on proportional weight. Each cgroup is assigned with a *blkio.weight* value indicating the proportion of the disk time used by the group. The second one is a throttling policy which specifies upper limits on an I/O device.

The pid controller. The `pid` cgroup subsystem is utilized to set a certain limit on the number of tasks of a container. This is achieved by setting the maximum number of tasks in *pids.max*, and the current number of tasks is maintained in *pids.current*. The pid cgroup subsystem will stop forking or cloning a new task (e.g., returning error information) after the limit is reached (e.g., pids.current > pids.max). As a result, the pid controller is effective for defending against multiple exhaustion attacks, e.g., fork bomb.

2.2 cgroups Inheritance

One important feature of cgroups is that child processes inherit cgroups attributes from their parent processes. Every time a process creates a child process (e.g., fork or clone), it triggers the forking function in the kernel to copy the initiating process. While the newly forked process is attached to the root cgroup at the beginning, after copying the registers and other appropriate parts of the process environment (e.g., namespace), a cgroup copying function is invoked to copy parent's cgroups. Particularly, the function attaches the task to its parent cgroups by recursively going through all cgroup subsystems. As a result, after the copying procedure, the child task inherits memberships to the exact same cgroups as its parent task.

For example, if the `cpusets` resource controller sets the CPU affinity of the parent process to the second core, the newly forked child process will also be pinned on the second core. Meanwhile, if the cpu subsystem limits the CPU *quota* to 50,000 with a *period* of 100,000 on the parent cgroup, the total CPU utilization of the cgroup (including both the newly forked process and its parent) cannot exceed 50% on the second core.

3 EXPLOITING STRATEGIES

In this section, we describe four strategies to escape the resource control of the cgroups mechanism, and explain the root causes why the existing cgroups cannot track the consumed resources. As introduced above, with the hierarchical structure, the cgroups mechanism can limit the total amount of resources for a group of processes (e.g., a container). This is done by attaching resource controllers to apply per-cgroup limits on specific system resources. Besides, the inheriting mechanism in cgroups ensures that all processes and their child processes in the same cgroup could be controlled by cgroup subsystems without consuming extra system resources. However, due to the complexity of the Linux kernel and difficulty in implementing cgroups, we find that several mechanisms are not considered, and thus can be utilized to escape the constraints of existing cgroups. The key idea is to generate workloads running on processes that are not directly forked from the initiating cgroups, causing the de-association of the cgroups. Particularly, there are four strategies, as illustrated in Figure 1, could be exploited by a normal process in the user-space without root privilege to escape the control of cgroups.

3.1 Exploiting Upcalls from Kernel

In the cgroups mechanism, all kernel threads are attached to the root cgroup since a kernel thread is created by the kernel. Thus, all processes created through `fork` or `clone` by kernel threads are also attached to the same cgroup (the root cgroup) as their parents. As a result, a process inside one cgroup can exploit kernel threads as proxies to spawn new processes, and thus escape the control of cgroups. Particularly, as the strategy ❶ shown in Figure 1, a process can first trigger the kernel to initialize one kernel thread.

This kernel thread, acting as a proxy, further creates a new process. Since the kernel thread is attached to the root cgroup, the newly created process is also attached to the root cgroup. All workloads running on the newly created process will not be limited by cgroup subsystems, and thus break the resource control.

This mechanism, however, requires a user-space process to first invoke kernel functions in the kernel space, then upcall a user-space process from the kernel space. While it is natural to invoke specific kernel functions (such as system calls) from user-space, the reverse direction is not common. One feasible path is via the usermode helper API, which provides a simple interface for creating a process in the user-space by providing the name of executable and environment variables. This function first invokes a workqueue running in a kernel thread (e.g., *kworker*). The handler function for the workqueue further creates a kernel thread to start the user process. The final step, which invokes the fork function in the kernel, attaches the created user process to the kernel thread's cgroups.

The usermode helper API is used in multiple scenarios, such as loading modules, rebooting machines, generating security keys, and delivering kernel events. While triggering those activities in user-space usually requires root permission, it is still possible to invoke the API in the user-space, which is discussed in Section 4.1.

3.2 Delegating Workloads to Kernel Threads

Another way to break the constraints of cgroups by exploiting kernel threads is to delegate workloads on them, as the strategy ❷ shown in Figure 1. Again, since all kernel threads are attached to the root cgroup, the amount of resources consumed by those workloads will be accounted to the target kernel thread, instead of the initiating user-space process.

The Linux kernel runs multiple kernel threads handling various kernel functions and running kernel code in the process context. For example, *kthreadd* is the kernel thread daemon to create other kernel threads; *kworker* is introduced to handle workqueue tasks [18]; *ksoftirqd* serves softirqs; *migration* performs the migration job to move a task from one core to another; and *kswapd* manages the swap space. For those kernel threads, depending on their functions, the kernel might run only a single thread in the system (e.g., *kthreadd*), or one thread per core (e.g., *ksoftirqd*), or multiple threads per core (e.g., *kworker*). It has been constantly reported that, those kernel threads can consume a huge amount of resources due to various bugs and problems [19–22]. Thus, if a process can force kernel threads to run delegated workloads, the corresponding consumed resources will not be limited by cgroups.

3.3 Exploiting Service Processes

Besides kernel threads maintained by the kernel, a Linux server also runs multiple system processes (e.g., *systemd*) for different purposes like process management, system information logging, debugging, etc. Those processes monitor other processes and generate workloads once specific activities are triggered. Meanwhile, many user-space processes serve as the dependencies for other processes and run simultaneously to support the normal functions of other processes. If a user process can generate kernel workloads on those processes (strategy ❸ shown in Figure 1), the consumed resources will not be charged to the initiating process, and thus the cgroups mechanism can be escaped.

3.4 Exploiting Interrupt Context

The last strategy is to exploit the resource consumed in the interrupt context. The cgroup mechanism only calculates the resources consumed in the process context. Once the kernel is running in other contexts (e.g., interrupt context, as the strategy ❹ shown in Figure 1), all resources consumed will not be charged to any cgroups.

In particular, the Linux kernel services interrupts in two parts: a top half (i.e., hardware interrupts) and bottom half (i.e., software interrupts). Since a hardware interrupt might be raised anytime, the top half only performs light-weight actions by responding to hardware interrupts and then schedules (defers) the bottom half to execute. When executing an interrupt handler on the bottom half, the kernel is running in the *software interrupt context*, thus it will not charge any process for the system resources (e.g., CPU). Since kernel 3.6, the processing of softirqs (except those raised by hardware interrupt) is tied to the processes that generate them [23]. It means that all resources consumed in the softirq context will not consume any quotas of the raised process. Moreover, the execution of softirqs will preempt any workloads on the current process, and all processes will be delayed.

Furthermore, if the workloads on handling softirqs are too heavy, the kernel will offload them to the kernel thread *ksoftirqd*, which is a per-CPU (i.e., one thread per CPU) kernel thread and runs at the default process priority. Once offloaded, the handling of softirqs runs in the *process context* of *ksoftirqd*, and thus any resource consumption will be charged on the thread *ksoftirqd*. Under this scenario, it falls into the kernel thread strategy (the strategy ❷ shown in Figure 1). To conclude, if a process (referred as process *A*) is able to raise a large amount of software interrupts, the kernel will have to spend resources on handling softirqs either in *interrupt context* or the process context of *ksoftirqd*, without charging the process *A*.

4 CASE STUDIES ON CONTAINERS

In the previous section, we have discussed several potential strategies to escape the resource control of cgroups. However, in realistic container environments, exploitation is more challenging due to the existence of other co-operative security policies. In this section, we present five case studies conducted within Docker container environments to demonstrate the detailed steps of exploiting the cgroups weaknesses.

Threat model. We consider a multi-tenant container environment where multiple Docker containers belonging to different tenants share the same physical machine. The multi-tenant environment is widely adopted today in both edge and cloud platforms. The system administrators utilize cgroups to set the resource limit for each container. Each container is de-privileged, set with limited CPU time, system memory, block I/O bandwidth, and pinned to specific cores. We assume an attacker controls one container instance and attempts to exploit the insufficiencies in cgroups to (1) slow-down performance of other containers, and (2) gain unfair advantages.

Servers	Processor	RAM	Block Device	NIC	OS	Linux Kernel	Docker
Dell XPS	Intel i7-8700 (12 x 3.20GHz)	16GB	SATA (7,200 rpm)	100Mb/S	Ubuntu 16.04	4.20	18.06
EC2 Dedicated Server	Intel E5-2666 (36 x 2.9GHz)	64GB	SSD (1,000 IOPS)	10,000Mb/S	Ubuntu 18.04	4.15	18.06

Table 1: Servers used for evaluation.

Case	Strategies	Description	Impact
Exception handling	❶	Trigger user-space processes	Consume 200× more resources, DoS
Data Synchronization	❷	System-wide writenback	DoS; RFA; covert-channel
Service journald	❸	Force *journald* to log container events	Consume CPU and block device bandwidth
Container Engine	❷❸	Workloads on container engine and *kworker*	Consume 3× more resources
Softirq handling	❷❹	Workloads on *ksoftirqd* and interrupt context	Consume extra CPU

Table 2: Summary of all case studies.

Configuration. We use the Docker container to set the configuration of cgroups through the provided interfaces. Besides, Docker also ensures that containers are isolated through namespaces by default. Especially, with the *USER* namespace enabled, the root user in a container is mapped to a non-privileged user on the host. Thus, the privileged operations within containers cannot affect the host kernel. Our case studies are conducted in such de-privileged containers.

To demonstrate the effectiveness of each exploitation, we initialize a container by setting multiple cgroup configurations on an idle server, and measure the utilization of system resources on the host. In order to emulate edge and cloud environments, we select two testbeds to conduct our experiments: (1) a local machine in our lab; (2) a dedicated host in Amazon EC2. The configurations of both servers are listed in Table 1. Particularly, while our local testbed is equipped with SATA Hard Disk Drive with 7,200 rpm, we choose a much better I/O configuration on the EC2 server. The storage of the dedicated testbed is provisioned SSD with 1,000 IOPS (the default number is 400), and the throughput is about 20× better than our local testbed. Thus, the local testbed represents a lower performance node that might be deployed in an edge environment, while the powerful dedicated server can emulate a multi-tenant container cloud environment.

Ethical hacking concerns. Exploiting the cgroups will inevitably generate host-level impact, which would potentially affect the performance of all containers on the host server. Therefore, for our experiments on Amazon EC2, we choose to use a dedicated server, which is solely used by us and is not shared with other tenants. In addition, it also allows us to simulate a multi-tenant container environment and measure the system-wide impacts.

Result summary. Table 2 presents an overall summary of all case studies, their corresponding exploiting strategies, and impacts. The first case study is to exploit the exception handling mechanism in the kernel, which involves strategy ❶. We find that exceptions raised in a container can invoke user-space processes, and its consequence is that the container can consume 200× more CPU resources than the limit of cgroups. The second case is to exploit the write-back mechanism for disk data synchronization, which involves strategy ❷. A container can keep invoking global data synchronization to slow down particular I/O workloads as much as 95% on the host. The third case is to exploit system service *journald* (through strategy ❸) which generates workloads consuming CPU and block device bandwidth. The fourth case is to exploit the container engine to generate extra unaccounted workloads (about 3x) on both container engine processes (strategy ❸) and kernel threads (strategy ❷). The last case is to exploit the softirq handling mechanism to consume CPU cycles on kernel threads (strategy ❷) and interrupt context (strategy ❹).

4.1 Case 1: Exception Handling

The first case is to exploit the exception handling mechanism in the kernel. We find that it is possible to invoke the usermode helper API and further trigger a user-space process (as the strategy ❶) through exceptions. By repeatedly generating exceptions, a container can consume about 200× CPU resources than the limit, and thus significantly reduce the performance of other containers on the same host (not limited to one core) by 85% to 95%.

Detailed analysis. The Linux kernel provides a dedicated exception handler for various exceptions, including faults (e.g., divide error) and traps (e.g., overflow). The kernel maintains an *Interrupt Descriptor Table* (IDT) containing the address of each interrupt or exception handler. If a CPU raises an exception in the user mode, the corresponding handler is invoked in the kernel mode. The handler first saves registers in the kernel stack, handle the exceptions accordingly, and finally returns back to the user mode. The whole procedure runs in kernel space and in the process context that triggers the exception. Thus, it will be charged to the correct corresponding cgroups.

However, these exceptions will lead to the termination of the initial processes and raise signals. These signals will further trigger the *core dump* kernel function to generate a core dump file for debugging. The core dump code in the kernel invokes a user-space

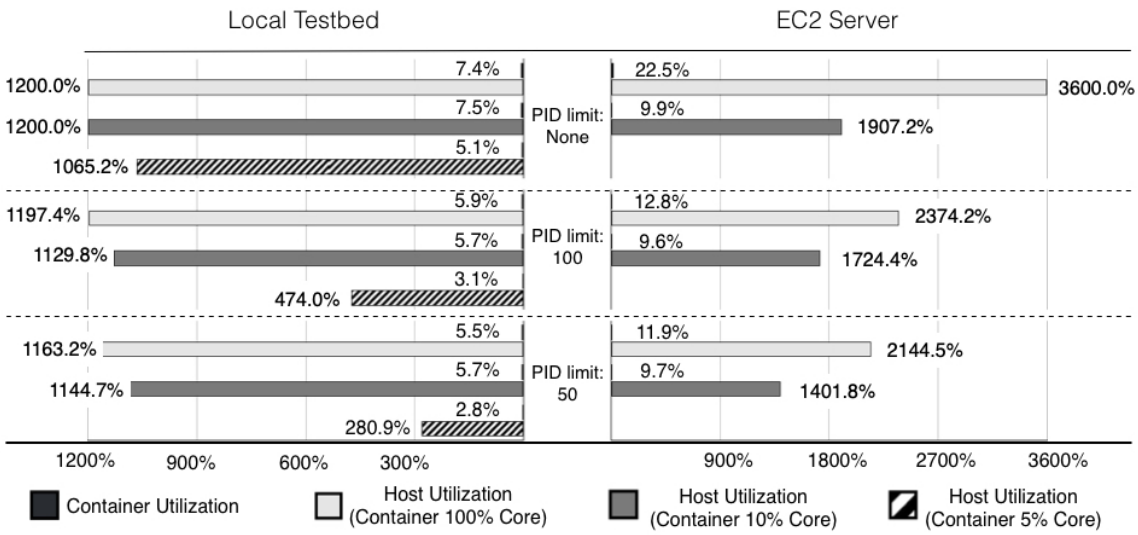


Figure 2: Workloads amplification of exception handling. The server only runs one container that keeps raising exceptions. The CPU resource used by the container is capped by the cpu controller as 100% one core, 10% of one core, and 5% of one core, respectively. A container can exhaust a server with 36 cores using only 22% CPU utilization of one core. The number of PID is further capped by the pid controller. With the number of active processes limited to 50, the container can still achieve 144× amplification for CPU resources.

application from the kernel via the usermode helper API. In Ubuntu, the default user-space core dump application is Apport, which will be triggered for every exception. As mentioned in the previous section, the system resources consumed by Apport will not be charged to the container, since the process is forked by a kernel thread, instead of a containerized process.

The newly spawned Apport instance will be scheduled by the kernel to all CPU cores for the purpose of load balancing, thus breaks the `cpusets` cgroup. Meanwhile, since the running of Apport process consumes much more resources than the lightweight exception handling (i.e., a kernel control path), if the container keeps raising exceptions, the whole CPU will be fully occupied by the Apport processes. The escaping of the `cpu` cgroup leads to a huge amplification of the system resources allocated to a container.

Workloads amplification. To investigate such impact, we launch and pin a container on one core. We set different limits of the CPU resources for the container by adjusting *period* and *quota*. The container entered into loops keeping raising exceptions. We implement several types of exceptions which are available to user-space programs. As the results are similar for different types of exception, we use the *div 0* exception as the example. The container is the only active program that runs in our testbeds. We measure the CPU usage of our testbed from the *top* command and the CPU usage of the container from the statistical tool of Docker. For the host level usage, we aggregate the CPU usage of all cores together (so the maximum usage of 12 cores is 1200%). We define the *amplification factor* as the ratio of the host's CPU utilization to the container's CPU utilization.

Figure 2 demonstrates that the usermode helper API can trigger user-space programs to significantly amplify the CPU usage of a container. On our local testbed, with only 7.4% CPU utilization on one core used by our container, the whole 12 cores are fully occupied. This problem cannot be mitigated after we reduce the CPU resources allocated to the container to only 10% core (by setting *period* to 200,000 and *quota* to 20,000). We further reduce the CPU constraint of the container to 20% core and finally limit the total utilization of 12 cores to 1,065%, giving an amplification factor of 207X. Meanwhile, while the system memory usage has increased by about 1GB, the memory usage of the container measured by Docker is only 15.58MB.

We obtain similar results from the EC2 server: a 22.5% utilization on the container is able to exhaust 36 cores. Since the CPU frequency is less powerful than our local testbed, once we limit the CPU resource of the container to 1/10 core, it can generate 1907% utilization on all 36 cores. The amplification factor is around 192X.

The `pid` controller. While the amplification requires the container to keep raising exceptions, we further use the `pid` cgroup subsystem to set a certain limit on the number of tasks of our container. Again, as demonstrated in Figure 2, the `pid` controller cannot diminish the amplification result even when the number of active processes is capped to 50, which is a very small number that might potentially cause huge usability impact on container users. The amplification factor can be reduced to 98× when we set a pid limit to 50 with only 20% CPU computing ability of one core. On the EC2 server, the amplification factor is around 144× by limiting the number of pid to 50 on a container with 10% CPU computing ability of one core.

Denial-of-service (DoS) attacks. When multiple containers run on the same core, they will share and compete for the CPU resources. The Linux CFS system allocates the CPU cycles based on the share of

Servers	Dell XPS				EC2 Dedicated Server			
	CPU	Memory	I/O Read	I/O Write	CPU	Memory	I/O Read	I/O Write
Baseline	632.5	6514.6	0.97	0.65	420.3	696.1	21.7	14.4
Exceptions (same core)	27.4	253.0	0.47	0.31	67.2	112.8	3.9	2.7
Exceptions (different cores)	35.2	291.5	0.81	0.54	76.8	129.8	1.8	1.23

Table 3: CPU based DoS attacks. The results are measured by sysbench. The unit for CPU is events per second. The units for memory and I/O benchmarks are MiB per second.

each container. The CFS system ensures complete fairness, where the container can fully utilize all resources in its slot. However, if a malicious container can produce new workloads outside its own cgroup, the CFS system will also allocate CPU cycles to those processes, thus reduce the usage of other co-resident containers. At the same time, the decreasing CPU usage might also impact other performance, such as memory and I/O throughput.

In our experiment, we measure the impact of the DoS attacks by exploiting the exception handling mechanism in the malicious container. We run two containers: one malicious container and one victim. We compare the performance of attacks with the cases that the malicious container runs normal workloads (i.e., baseline). The victim container runs different *sysbench* [24] workloads to measure the performance.

The results on both servers are illustrated in Table 3. We first set both containers on the same core with the exact same CPU shares and quotas. We find that raising exceptions (which causes core dump) can significantly reduce 95% on both CPU and memory performance, and around 17% on I/O performance on our local testbed. On the EC2 server, the number is around 85% for CPU and memory performance, 82% on the I/O performance. This is reasonable since raising exceptions causes a huge amount of user-space core dump applications that compete for the CPU cycles with the victim container.

We further change the core affinity of the malicious container by pinning the container on a different core. Although the malicious container is no longer competing CPU resources on the same core with the victim, it still shows similar results on the performance of the victim. The reason is that the main competitor for the CPU resources is not the malicious container but those triggered core dump applications.

This result demonstrates that a malicious tenant can easily utilize a container to significantly reduce the performance of all other containers on the same host and lower the quality-of-service of the service vendor, and thus potentially cause huge financial loss with little cost.

4.2 Case 2: Data Synchronization

Our second case is to exploit the writeback mechanism for disk data synchronization, which is widely adopted for performance consideration. The CPU only writes the updated data to the cache, and data is written to disk later when the cache is evicted. Our exploitation can escape cgroups since the lazy disk writeback mechanism decouples the process that initiates the I/O with the process that synchronizes the disk write. There are multiple ways to trigger the data synchronization, including periodically writeback and insufficient memory. It could also be intentionally invoked by user processes through system calls, such as `sync` (which writes back all pending modifications to the cached file data to the underlying file systems), `syncfs` (which synchronizes the file systems referred to by the open files), and `fsync` (which transfers all modified data of a file to its resident disk device). Those system calls are available to Linux containers. Particularly, we find that `sync` could be exploited to slow down system-wide I/O performance (e.g., more than 87% degradation on sequence writing), launch resource-freeing attack, and build covert channels.

Detailed analysis on sync. The first step of `sync` is to start a kernel thread, which flushes all dirty pages contained in the page cache to disk. It looks for all dirty inodes to be flushed by scanning all currently mounted file systems, and flushes the corresponding dirty pages. Since `sync` allows a process to write back all dirty buffers to disk, any I/O operations have to wait for the flushing. Even worse, the dirty buffers generated by other processes (might belong to another container) will also be forced to write back to disk.

A process within a container can repeatedly invoke the `sync` with only an insignificant amount of workloads if no I/O operation is conducted. However, at the same time, if there are I/O operations on other co-resident containers, the `sync` will write back all the dirty pages. In our experiment, we run a container that kept calling `sync`. It did not cause any extra utilization beyond the constraint of the container. However, once we run another container with some simple write operations, the `sync` leads to a huge amount of CPU wait time, which is generated by the combination of `sync` and write operations. The CPU wait time is used to indicate the time consumed for I/O waiting, and can still be used by other workloads. However, the performance of particular workloads running on other containers is significantly impacted.

`blkio` cgroup. As mentioned in Section 2.1, the `blkio` cgroup subsystem can apply I/O control to block devices. While Docker only supports limiting the relative I/O throughput by weights, the kernel actually can set an upper limit to the cgroups. We use `blkio` to apply I/O control to the container running `sync`. Unfortunately, based on the statistical tools of Docker, the I/O throughput of our container is *zero*. Thus, the `blkio` controller cannot reduce the impact of `sync`. The reason is that all the writeback workloads triggered by `sync` are handled in kernel threads, and no I/O workloads are charged to the container invoking the `sync`.

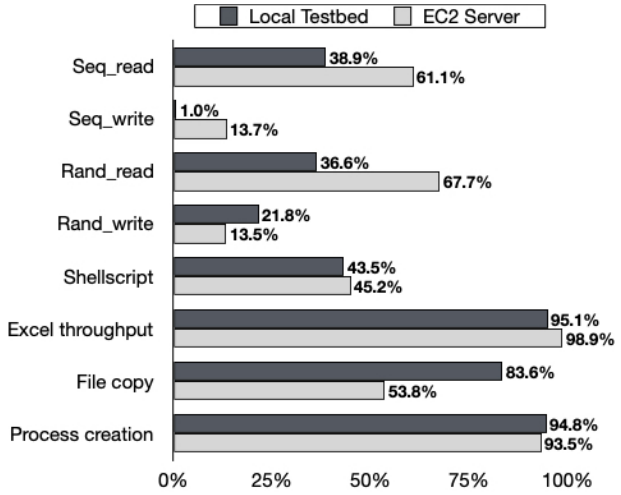
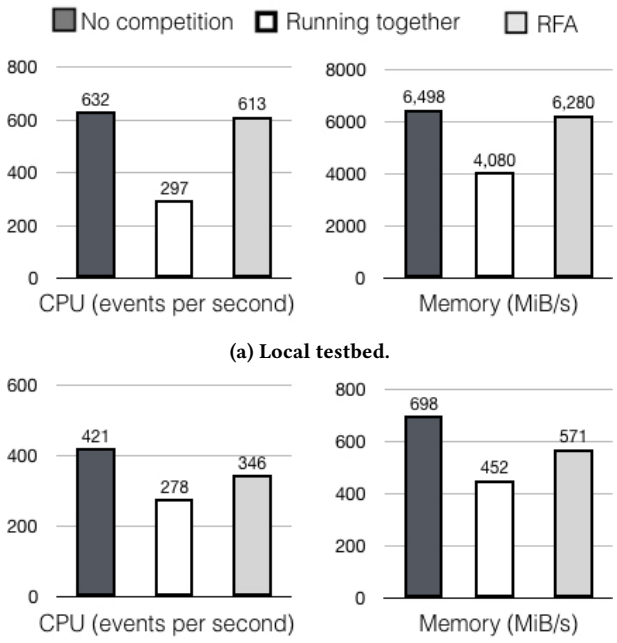


Figure 3: Performance degradation of I/O-based DoS attacks. The performance is compared with the baseline case, where an idle loop is running in the attacking container pinned in different cores.



(a) Local testbed.

(b) EC2 dedicated server.

Figure 4: Resource Freeing Attacks. The performance of the beneficiary is measured by sysbench: higher score represents better performance. With a helper mounting RFA attacks, the beneficiary can almost achieve similar performance as the case without competition.

I/O-based DoS attacks. The calling of the system call sync inside one container will keep invoking system-wide writebacks, regardless of whether the processes that issue the I/O operations are inside the container or not. In some cases, the writeback will reduce the system performance as particular workloads need to wait until the writeback finishes. To test the effectiveness, we run two containers pinned on two different cores. The only task the malicious container does is to invoke the system call sync, thus incurring no I/O operations by itself.

To measure the performance of the victim container, we run the FIO benchmark [25] inside the victim container to measure the I/O performance. In particular, we conduct four types of different FIO workloads, including sequence/random write, and sequence/random read. We also run the UnixBenchmark to test the impact on the performance other than I/O. We compute the degradation of the performance by dividing the result to the baseline case where an idle loop is running in the malicious container. The results are demonstrated in Figure 3. For UnixBenchmark, we list the workloads that have significant performance degradation. Overall, we can see that the performance of the FIO benchmark running in the victim is greatly impacted. By keep invoking sync in a malicious container, the performance of all four types I/O operations is significantly affected. For sequential write, the performance is reduced to only 2% in our local testbed, and 13% on the EC2 server. For UnixBenchmark, the performance of running shell scripts is also severely reduced to less than half. For other benchmarks, the degradation is about 5 to 10 percents.

Resource-Freeing Attack (RFA). The goal of RFA attacks [26] is to free up specific resources for the attacker's instances by competing for the bottleneck resources of the victim. In the situation of the container environment, two containers are competing for system resources such as CPU. The malicious container seeks to improve the performance of its workload (referred as the *beneficiary*) to get more system resources. Thus, the malicious container runs another lightweight program (referred as the *helper*) to free resources used by the victim container so that the *beneficiary* can gain more resources. The *helper* only consumes few system resources (thus it almost has no impact on the *beneficiary*) but can significantly impact the workloads running inside the victim container. For example, in the malicious container, a *beneficiary* program can run CPU-intensive workloads and compete for the CPU resources with the victim container on the same CPU core. The victim container runs an I/O-intensive workload so the CPU activity is bound with the frequency of I/O operations: more I/O operations make the victim container consume more CPU cycles to handle the requests. Then, the malicious one runs a *helper* program to call sync periodically, trigger writebacks, and reduce the I/O activities of the victim. As a result, the CPU usage of the victim container is reduced, and the malicious one can gain more CPU resources.

We simulate the experiments by running two containers on the same core. In the victim container, we simulate a web crawler case where the container constantly writes a web page into a new file. We measure the CPU and memory performance of the malicious container using sysbench, where a higher value represents better performance. The malicious container also regularly calls sync to trigger global writebacks. For the baseline case, only the attacker's container is active and thus there is no competition for all system resources. We then run both containers and compare the performance of the attacker's container to the baseline case. As we see

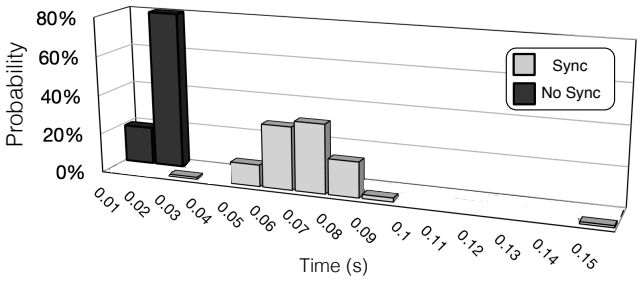


Figure 5: The distribution of the required time for opening multiple files. The grey bar represents the case for opening multiple files while running the system call sync simultaneously; The black bar represents the case without calling sync.

in Figure 4, without RFA attacks, since two containers compete for the CPU resources on the same core, the CPU performance (i.e., the white bar) is about half of the case without competition (i.e., the black bar), and the memory performance is about 1/3 of the case without competition. However, by launching the RFA attacks (i.e., the grey bar), the *beneficiary* inside the malicious container can get much better performance on both testbeds. Particularly, on our local server, the performance is almost the same as the case without competition.

Covert Channels. Finally, we demonstrate that the insufficiencies in cgroups could also be exploited by malicious attackers to build timing-based covert channels across different cores. The idea is to leverage the performance differences incurred by the disk data synchronization. We measure the time for writing ten files in one container, while running sync in another container on another core. We repeat the experiments for 100 times and present the distribution of the required time in Figure 5. We can observe the obvious timing differences for opening the files between running sync and without running sync. We build a proof-of-concept channel by leveraging the performance differences, and are able to transfer data with a rate of 2bits/s with an error rate 2%.

4.3 Case 3: System Process - Journald

Our third case is to exploit the systemd-journald service, which provides a system service to collect system logging data, including kernel log messages, system log messages (e.g., *syslog* call or Journal API), and audit records via the audit subsystem. All related activities are logged by a system process *journald*. In our case study, we find that three categories of operations in a container can force the *journald* process to log, causing 5% to 20% extra CPU utilization and an average of 2MB/s I/O throughput, which can then be exploited to impact other containers' performance.

Detailed analysis. System processes of the host are attached to cgroups that are different from the processes within containers, since they are maintained by the OS to provide system-wide functionalities. Thus, if the workloads inside containers can trigger activities for those system processes, the resource consumed by those activities will not be charged to containers' cgroups, and thus escape the resource control mechanism. However, most operations inside containers are ignored by system processes running on the host. For example, many activities of a user-space process running on the host will be recorded by the *journald*. But if the process runs within the container, those activities will be ignored. In order to record events inside containers, system administrators on the host need to change the configuration of the systemd-journald service. Particularly, Docker provides an option to enable *journald* logging.

However, we find that, even without enabling the logging option, under some particular circumstances, containers are still able to generate non-negligible workloads on the *journald* system process. In particular, we present three types of operations that lead to workloads on the system process, and thus escape the control of cgroups.

Switch user (su) command. The *su* command provides an approach to switching the ownership of a login session to root user. The action of switching to root user will be recorded in the *journald* system process. The logging contains the information of the process, users' account, and the switching of the environment. The exit of the user will also be recorded by journald service. With the *USER* namespace, a root user inside a container is mapped to an unprivileged user on the host. So a process inside the container may have full privileges inside a *USER* namespace, but it is actually de-privileged on the host. As the root user inside the container, the *su* command is able to enter into another user. Unfortunately, the activities caused by switching accounts inside a container will trigger systemd-journald service to log the related information.

Add user/group. Inside a *USER* namespace, a container user can add new groups or add new accounts inside existing groups. Those activities will also be logged by the *journald* system process on the host.

Exception. Finally, as mentioned previously, the kernel is unable to distinguish the context of raised exceptions (inside a container or not). Thus, the crash information caused by exceptions inside a container will also trigger the logging activities of the system process on the host.

All the above workloads will trigger a significant amount of event logging in the *journald*. Again, we set one container with one CPU core computing capacity to keep invoking the above commands. In our local testbed, we observe a constant 3.5% CPU utilization on *journald*, 2.5% CPU utilization on *auditd*, and 1% CPU utilization on *kauditd*. In the EC2 server, the number is much bigger due to its better I/O performance: we observe an average CPU utilization about 20%. We also find that the average I/O throughput is around 2MB/s on the *journald* process, while the I/O throughput is zero in our container. This number will increase if we assign more computing resource to the container.

DoS attacks. The logging activities in *journald* will generate a non-negligible amount of I/O operations, which lead to the resource competition with other containers. To measure the consequence, we run two containers on different cores of the host. In our malicious container, we keep switching user (i.e., su) and quitting current user (i.e., exit). In the victim container, we run the same benchmarks as described in case 2.

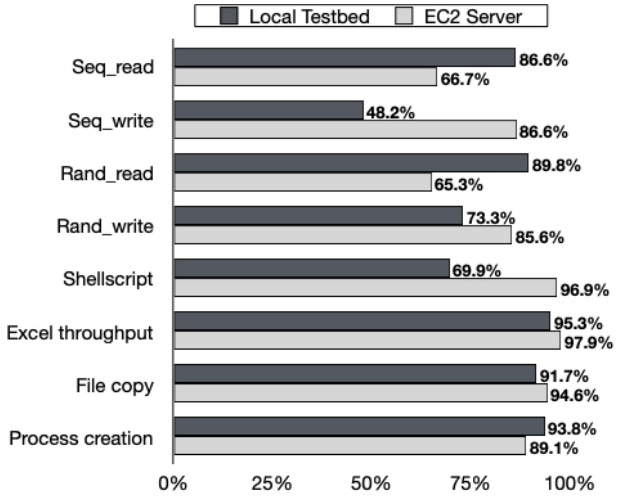


Figure 6: Performance degradation of DoS attacks exploiting journald. We compare the performance with the baseline case, where an idle loop runs in the attacking container pinned on a different core.

Figure 6 shows the results. Overall, we see system-wide performance degradation. The attack by abusing *journald* will be more effective in servers with poor I/O performance (e.g., our local testbed). As mentioned before, it can cause more than 2MB/s I/O throughput in the *journald* process. We observe it can still slowdown other containers in the EC2 dedicated server with 1,000 IOPS (the throughput is about 15MB/s). In a dedicated server with the default configuration (e.g., 400 IOPS with the throughput about 6MB/s), we can observe a more obvious result.

Residual effect. On a server with poor I/O performance, the writing workloads on the system process might surpass the I/O ability of the server. As a result, a huge amount of logging event is queued, and wait to be logged later. This will cause a residual effect: even after the container stops its workloads (e.g., *su*), the system will continue writing in the *journald* until the workloads in the queue finish. The residual effect is conspicuous in our local testbed, and last much longer than the running time of the workloads. The CPU and I/O resources are being consumed even the container is completely idle. Even worse, those writing operations will significantly affect the I/O performance of other containers and the host.

4.4 Case 4: Container Engine

The fourth case for containers is to exploit the container engine by triggering extra workloads on both kernel threads (e.g., *kworker*) and the container engine, which is required to run on the host to support and manage container instances. Particularly, the container engine runs as a privileged daemon on a system, and thus it is attached to a different cgroup as container instances. The cgroup limit on container instances will not be able to control the consumed resources on the engine. Overall in this way, we find that a container can increase the resource consumption to about three times.

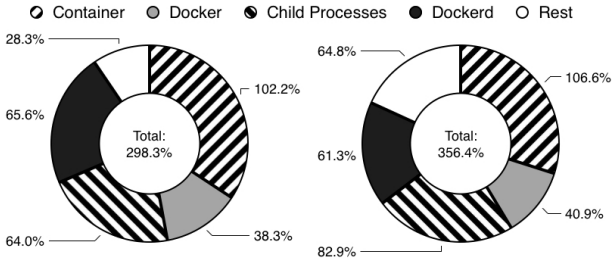


Figure 7: The CPU utilization of Docker processes by exploiting the container engine. The CPU resource of the container (i.e., `Container` in the figure) is limited as 100% of one core. However, Docker engine processes and kernel threads also consume about 200% of one core CPU resources.

Detailed analysis. We first give a brief introduction to the Docker container engine and its cgroup hierarchy. Docker creates a *Docker* cgroup containing all container instances. Each container is identified by its ID and holds its all processes created by `fork`. Theoretically, all workloads running inside a container will be charged to the container cgroup.

Besides, Docker is built on top of the Docker engine, where a daemon process (i.e. *dockerd*) runs in the background handling the management of Docker images. The Docker engine then communicates with *containerd*, a daemon to further use `runC` for running containers. The *dockerd* process has multiple child processes for each container instance. Those processes are attached to the default cgroup for all system services.

Furthermore, users mainly control and manage Docker through a command line interface (CLI) client (i.e. the *docker* process), which interacts with the Docker daemon through Docker REST API. The Docker CLI provides interfaces for users to create or execute a container. It also provides multiple commands to set the configurations on resource limitations related to underlying control groups. Similar to Docker engine processes, the Docker CLI does not belong to the container's cgroup either.

It is fairly easy to exploit the container engine to break the control of cgroups. One simple approach is to exploit the terminal subsystem. When a container user interacts with the tty device, the data first passes through the CLI process and the container daemon, and reaches the tty driver for further processing. Specifically, the data is sent to the LDISC, which connects the high-level generic interface (e.g., read, write, ioctl) and low-level device driver in the terminal system. The data is flushed to LDISC by executing work queues in the *kworker* kernel threads. As a result, all workloads on the kernel threads and all container engine processes will not be charged to the container instances.

We measure the workloads generated in container engine by repeatedly showing all loaded modules in the host in the terminal, and illustrate the results in Figure 7. Again, the utilization of the container is limited to one core (as the *Container* in Figure 7). Overall, with one core's computing power (100% utilization), the container can cause about 300% workloads on the host by abusing docker engine processes. To breakdown the usage, the *docker*

process belongs to the host's user cgroup; other docker processes belong to a system cgroup: docker.service. The rest (most of them are kernel thread *kworker* due to streaming workloads as explained in Section 3.2) belongs to the root cgroup. We also conduct similar experiments as Table 3 in Case 1. By exploiting the Docker container engine, the attacker is able to reduce the performance of CPU and memory about 15%.

4.5 Case 5: Softirq Handling

The last case is to exploit the softirq handling mechanism in the Linux kernel. The current kernel defines 11 types of softirqs. Particularly, it is common for hardware interrupt handlers to raise softirqs. While most hardware interrupts might not be directly raised by containers, container users are able to manipulate workloads on network interface generating NET softirq, or block devices generating Block softirq. The handling of those softirqs consumes CPU resources on the process context of kernel thread (e.g., *ksoftirqd*) or interrupt context.

NET softirq. Interrupt will be raised once the NIC finishes a packet transmission, and softirqs are responsible for moving packets between the driver's buffer and the networking stack. However, the overhead raised by softirqs is negligible when the traffic bandwidth is limited: previous work [27] demonstrates 1% overhead for networking traffic over 1 Gbps.

We find that, the overhead incurred by the networking traffic will be greatly amplified by the firewall system (e.g., *iptables*) on the server. The iptables, built on top of *netfilter*, provide a management layer for adding and deleting packet filtering rules. The netfilter hooks packet between the network driver and network stack. All networking packets are first inspected by filtering rules, then forwarded for further actions (e.g., forwarding, discarding, processing by the network stack). As a result, the processing of networking traffic under iptables is handled in softirq context, and thus will not be charged to the container generating or receiving the traffic. On Linux, Docker relies on configuring iptables rules to provide network isolation for containers. Particularly, it might set multiple rules for containers that provide web or networking services. The rules exist even the container is stopped. Even worse, in some circumstances, containers can make changes to the system iptables rules if the related flag is set as true. Once there is a considerable number of rules, the overhead will be non-negligible.

We measure the overhead brought by the softirq handling networking traffic under different numbers of iptables rules, as illustrated in Figure 8. Particularly, we measure the CPU usage of all *ksoftirqd* kernel threads, and the time spent on interrupt context (from the `hi` and `si` of the *top* command). On our local testbed, the capacity of NIC is 100 Mbit/s, and the networking traffic is about 20-30 Mbit/s, which is significantly smaller then the Gbps level as in [27]. We can clearly see that significant overhead is incurred by handling networking traffic, and is strongly correlated to the number of rules. When the number of rules reaches 5,000, the CPU will waste a huge amount of time on processing softirqs (around 16%), and not charge to the container which initiates the networking traffic. Once there 10,000 rules in the server, the overhead is about 40%, and most of them are concentrated on one single core.

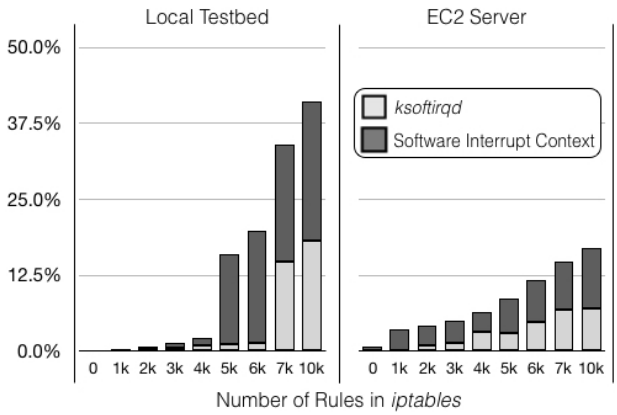


Figure 8: Overhead incurred by handling networking traffic with different numbers of *iptables* rules.

The EC2 server has a much powerful NIC with 10,000 Mbit/s capacity and much higher bandwidth compared with our local testbed. As a result, the overhead is slightly smaller compared with our local testbed. In our experiments, with the networking traffic about 300 Mbit/s, the traffic can still waste an in-negligible amount of CPU cycles. As mentioned in Section 3.4, the handling of software interrupts will either preempt current work, or consume CPU cycles in the kernel thread. As a result, all other workloads on the same core will be delayed.

BLOCK softirq. Another example of raising workloads on handling softirq is the I/O operations on block devices. The Linux kernel uses queues to store the block requests I/O, and adds new requests into queues. Once the request handling is completed, it raises softirqs, which are handled in the softirq context, to further process the queue. Then, the basic idea of escaping cgroups utilizing BLOCK softirq is similar to exploiting NET softirq.

In a container context, such workloads can be generated by keeping querying the events in the completion queue and submitting write or read operations. The impact is particularly obvious on devices with poor I/O performance. To further quantitatively measure the impact, we use a container fixed on one core running *fio* commend to do sequential reading or writing. We choose a small block size for the writing and a large size for the reading. We measure the CPU utilization of multiple kernel threads such as *kworker*. The workloads in the container are able to generate a non-negligible amount of workloads on the kernel on our local testbed, including 16.7% of workloads on the *kworker* of the same core. Besides, for the sequence I/O reading, the process of file allocation generated an additional 3.9% utilization on *jpd2* and 3.8% on *kswapd*. Finally, we also measure the degradation by exploiting *kworker*. We create workloads on the kernel thread *kworker* on the same core, and the attacker was able to cause about 10% performance loss on the victim measured by the sysbench benchmark.

5 MITIGATION DISCUSSION

In this section, we present our initial efforts and thoughts towards counter-measuring security issues raised by the insufficiencies of

the existing cgroups mechanism. As most issues involve multiple kernel subsystems, it is difficult to deploy a single solution to comprehensively and completely fix all problems. Also, resolving some issues might need a redesign of the current resource accounting system. In the following, we discuss potential mitigation methods from different perspectives.

Intuitively, cgroups should have a fine-grained accounting mechanism by considering all workloads directly or indirectly generated by a group of processes (or a container). For example, if a container invokes another user-space process through a kernel thread, the container's cgroup should be passed by the kernel thread so that it would also be copied to the new user-space process. As a result, the newly invoked process belongs to the same cgroup as the container instead of the root cgroup as the kernel thread. While such an approach can be applied to newly spawned processes with non-trivial efforts, it is difficult to handle those processes already existed in the system. Kernel threads like *kworker, ksoftirqd* are created by the kernel to handle specific kernel workloads, which might be triggered by different processes attached to different cgroups. The case of the *journald* system process is similar: it logs all related events raised by all processes, so it is unreasonable to attach the whole *journald* process to a specific cgroup. Thus, rather than changing the cgroup of those threads, a comprehensive mechanism should track the system resources of a specific amount of workloads, and charge them to the initial process. For example, Iron [27] tracks the CPU cycles for handling every networking packet and charges to the related process. However, such methods would undoubtedly require a significant amount of kernel development efforts, as well as bring a significant runtime overhead brought by instrumenting multiple kernel functions for fine-grained resource tracking.

For some workloads, another arguable concern is whether the cgroup should charge those system resources to the container or not. From the consideration of privacy, the host server should not record any sensitive information running inside a container instance. The *journald* provides specific options to enable logging activities inside a container. However, we show that, even without enabling the logging option, the host still logs multiple events for containers. The logging is conducted by the host, and thus should not charge to the container. Besides, the core dump information for the exception raised inside a container is not available to the container user. Thus, one possible method is to disable all potential logging or recording activities by distinguishing the container context. Another approach to fully addressing the problem is to build an extra cgroup subsystem specified for logging. All logging activities would be accounted by the new logging cgroup subsystem.

Finally, some issues cannot be solved even with a fine-grained accounting mechanism. For example, while the current cgroups mechanism clearly mentions that the writeback workloads are not counted, Linux kernel maintainers have started to develop new cgroup mechanisms (i.e. cgroup *v2*) that leverages both *memory* and *blkio* subsystems to track the writeback and charge containers for the dirty pages. However, a malicious container can keep calling `sync` without generating any I/O workloads. The writeback workloads are charged to the container that has I/O operations instead of the malicious one. Meanwhile, it is unfair to charge everything to the containers that invoke the data synchronization. Since simply disabling all such functions will inevitably affect the usability, a potentially feasible solution is to apply rate limit on those sensitive operations.

6 RELATED WORK

In this section, we survey some research efforts that inspire our work and highlight the differences between our work and previous research. We mainly discuss research works in the following two areas:

6.1 Virtual Machine and Container

While VM [28] has ushered in the cloud computing era, the performance is still not satisfying for its high latency and low density, despite of a large number of research efforts [29–31]. Container is becoming popular since it provides an alternative way of enabling lightweight virtualization and allows full compatibility for applications running inside. Researchers are thus curious about the performance comparison between hardware virtualization and containers. Felter et al. showed that *Docker* can achieve a higher performance than *KVM* in all cases by using a set of benchmarks covering CPU, memory, storage, and networking resources [7]. Spoiala et al. [32] also demonstrated that *Docker* outperforms *KVM* and could support real-time applications using the *Kurento Media Server* to test the performance of *WebRTC* servers. Morabito et al. [8] further compared the performance between traditional hypervisor (e.g., *KVM*) and OS-level virtualization (including both *Docker* and *LXC*) with respect to computing, storage, memory, and networks, and observed that Disk I/O is still the bottleneck of the *KVM* hypervisor. All of these works demonstrate that container-based OS-level virtualization is a more efficient solution than traditional VM-based hardware virtualization. While most previous research efforts focus on understanding the performance of containers, few attempted to investigate the effectiveness of the resource sharing of underlying kernel mechanisms. We are among the first to systematically study the performance and resource accounting problems of containers caused by insufficiencies of control groups.

Container security. Besides performance, the security of containers has also received much attention from both academia and industry. Gupta [33] first gave a brief overview of *Docker* security. Bui [34] then presented an analysis on *Docker* containers in term of the isolation and corresponding kernel security mechanisms. While those previous works claim that *Docker* containers are fairly secure with the default configuration, Lin et al. found that most of the existing exploits can successfully launch attacks from inside the container with the default configuration [17]. Grattafiori et al. [35] summarized a variety of potential vulnerabilities of containers including problems in the memory-based pseudo file systems. Gao et al. further conducted a systematical study on understanding the potential security implications of the memory-based pseudo file systems due to problems in namespaces [9, 14]. Lei et al. proposed a container security mechanism called SPEAKER to reduce the number of available system calls to applications [36]. Sun et al. [16] developed two security namespaces enabling autonomous security control for containers, and Arnautov et al. [37] proposed to secure Linux containers using Intel SGX. The misconfigured

capabilities could also be exploited to build covert channels in containers [38]. Our work on cgroups further complements previous research efforts on understanding the security of containers.

6.2 Cloud Security

Co-residence. Extensive research efforts have also been devoted to understanding the security issues of clouds, particularly multi-tenant clouds where different tenants share the same computing infrastructures. In a multi-tenant cloud environment, attackers can place malicious VMs co-resident with a target VM on the same server [39] and then launch various attacks including side-channel [40] and covert-channel attacks [41, 42]. Meanwhile, side and covert channel attacks are common approaches to verify co-residence on the same physical server. For example, cache-based covert channels [43–47] are widely adopted since multiple instances share the last-level caches on the same package. Zhang et al. further demonstrated the feasibility of launching real side-channel attacks on the cloud [48–50]. Besides the cache-based channel, other methods like memory bus [51], memory deduplication [52], core temperature [53, 54] are also effective for covert-channel construction. While multiple defense mechanisms have also been proposed [55–60], two previous works [61, 62] show that it is still practical (and cheap) to achieve co-residence in existing mainstream cloud services. Wang et al. [63] conducted a large scale measurement study on three serverless computing services and found several resource accounting issues that can be abused by tenants to run extra workloads.

Denial-of-Service attacks. Since underlying computing resources are shared among different tenants, the contention is inevitable. Varadarajan et al. proposed resource-freeing attacks [26] to free up resources used by victims so that the attacker's instances can gain extra utilization. Zhang et al. [64] investigated the impact of memory DoS attacks and showed a malicious cloud customer can cause 38× delay for an E-commerce website. For DoS attacks on I/O performance, Huang et al. [65] proposed cascading performance attacks to exhaust hypervisor's I/O processing capability. Moreover, multiple attacks [66–71] attempt to exhaust the shared infrastructure resources such as power facility so that servers in a data center are forced to shutdown. Different from all previous work, our work shows that the insufficiencies in cgroups can also be exploited to launch multiple attacks in a multi-tenant container environment.

7 CONCLUSION

In this paper, we develop a set of strategies to break the resource rein of Linux Control Groups. We demonstrate that inherited cgroups confinement via process creation does not always guarantee consistent and fair resource accounting. We can generate out-of-band workloads via processes de-associated from their original cgroups. We further present five case studies showing that it is feasible to realize these attacks in Docker container environments. By exploiting those insufficiencies of cgroups in a multi-tenant container environment, malicious containers can greatly exhaust the host's resources and launch multiple attacks, including denial-of-service attacks, resource freeing attacks, and covert-channel attacks. We conduct experiments on both a local testbed and a dedicated server in Amazon EC2 cloud, and demonstrate that a container can amplify its workloads as much as 200× above its limit, and reduce the performance of other containers by 95%.

ACKNOWLEDGMENT

We are very grateful to the anonymous reviewers for their insightful and detailed comments, which help us to improve the quality of this work.

REFERENCES

[1] Edge Computing Extend containers safely to the farthest reaches of your network. https://www.docker.com/solutions/docker-edge.
[2] Microservice Architecture. . http://microservices.io/patterns/microservices.html.
[3] Amazon Lambda. https://aws.amazon.com/lambda/.
[4] AWS Elastic Container Servicee. https://aws.amazon.com/ecs/.
[5] IBM Cloud Container Service. https://www.ibm.com/cloud/container-service.
[6] Google Kubernetes. https://cloud.google.com/kubernetes-engine/.
[7] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An Updated Performance Comparison of Virtual Machines and Linux Containers. In *IEEE ISPASS*, 2015.
[8] Roberto Morabito, Jimmy Kjällman, and Miika Komu. Hypervisors vs. Lightweight Virtualization: A Performance Comparison. In *IEEE IC2E*, 2015.
[9] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. ContainerLeaks: Emerging security threats of information leakages in container clouds. In *IEEE/IFIP DSN*, 2017.
[10] CVE-2014-6407. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6407., 2014.
[11] CVE-2014-9357. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-9357., 2014.
[12] CVE-2015-3631. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3631., 2015.
[13] Mingwei Zhang, Daniel Marino, and Petros Efstathopoulos. Harbormaster: Policy Enforcement for Containers. In *IEEE CloudCom*, 2015.
[14] 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.
[15] Control Group v2. https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html.
[16] Yuqiong Sun, David Safford, Mimi Zohar, Dimitrios Pendarakis, Zhongshu Gu, and Trent Jaeger. Security Namespace: Making Linux Security Frameworks Available to Containers. In *USENIX Security Symposium*, 2018.
[17] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. A measurement Study on Linux Container Security: Attacks and Countermeasures. In *ACM ACSAC*, 2018.
[18] Working on workqueues. https://lwn.net/Articles/403891/, 2010.
[19] kswapd taking 100% CPU. https://www.redhat.com/archives/nahant-list/2006-March/msg00033.html, 2006.
[20] Kworker, what is it and why is it hogging so much CPU? https://askubuntu.com/questions/33640/kworker-what-is-it-and-why-is-it-hogging-so-much-cpu, 2012.
[21] Why is ksoftirqd/0 process using all of my CPU? https://askubuntu.com/questions/7858/why-is-ksoftirqd-0-process-using-all-of-my-cpu, 2011.
[22] Kworker shows very high CPU usage. https://askubuntu.com/questions/806238/kworker-shows-very-high-cpu-usage, 2016.
[23] Software interrupts and realtime. https://lwn.net/Articles/520076/, 2012.
[24] sysbench - A modular, cross-platform and multi-threaded benchmark tool. http://manpages.ubuntu.com/manpages/trusty/man1/sysbench.1.html.
[25] Flexible I/O Tester. https://github.com/axboe/fio.
[26] 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 *ACM CCS*, 2012.
[27] Junaid Khalid, Eric Rozner, Wesley Felter, Cong Xu, Karthick Rajamani, Alexandre Ferreira, and Aditya Akella. Iron: Isolating Network-based CPU in Container Environments. In *USENIX NSDI 18)*, 2018.
[28] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *USENIX ATC*, 2001.
[29] Carl A Waldspurger. Memory Resource Management in VMware ESX Server. *ACM OSDI*, 2002.
[30] Andrew Whitaker, Marianne Shaw, and Steven D Gribble. Scale and Performance in the Denali Isolation Kernel. *ACM OSDI*, 2002.

[31] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) than your Container. In *ACM SOSP*, 2017.
[32] Cristian Constantin Spoiala, Alin Calinciuc, Corneliu Octavian Turcu, and Constantin Filote. Performance comparison of a WebRTC server on Docker versus Virtual Machine. In *IEEE DAS*, 2016.
[33] Udit Gupta. Comparison between security majors in virtual machine and linux containers. *arXiv preprint arXiv:1507.07816*, 2015.
[34] Thanh Bui. Analysis of Docker Security. *arXiv preprint arXiv:1501.02967*, 2015.
[35] Aaron Grattafiori. NCC Group Whitepaper: Understanding and Hardening Linux Containers, 2016.
[36] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. Speaker: Split-Phase Execution of Application Containers. In *Springer DIMVA*, 2017.
[37] 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 *USENIX OSDI*, 2016.
[38] 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 *IEEE Trustcom/BigDataSE/ISPA*, 2016.
[39] 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 *ACM CCS*, 2009.
[40] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, Dmitry Ponomarev, et al. Branchscope: A New Side-Channel Attack on Directional Branch Predictor. In *ACM ASPLOS*, 2018.
[41] Dmitry Evtyushkin and Dmitry Ponomarev. Covert Channels Through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations. In *ACM CCS*, 2016.
[42] Dean Sullivan, Orlando Arias, Travis Meade, and Yier Jin. Microarchitectural Minefields: 4k-Aliasing Covert Channel and Multi-Tenant Detection in IaaS Clouds. In *NDSS*, 2018.
[43] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology*, 2010.
[44] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. An Exploration of L2 Cache Covert Channels in Virtualized Environments. In *ACM CCSW*, 2011.
[45] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A High-Resolution Side-Channel Attack on Last-Level Cache. In *IEEE DAC*, 2016.
[46] Yuval Yarom and Katrina Falkner. FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*, 2014.
[47] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE S&P*, 2015.
[48] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *ACM CCS*, 2012.
[49] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *ACM CCS*, 2014.
[50] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *USENIX Security*, 2016.
[51] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *USENIX Security*, 2012.
[52] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. Security Implications of Memory Deduplication in a Virtualized Environment. In *IEEE/IFIP DSN*, 2013.
[53] Davide B Bartolini, Philipp Miedl, and Lothar Thiele. On the Capacity of Thermal Covert Channels in Multicores. In *ACM EuroSys*, 2016.
[54] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. Thermal Covert Channels on Multi-core Platforms. In *USENIX Security*, 2015.
[55] Yinqian Zhang and Michael K. Reiter. Düppel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud. In *ACM CCS*, 2013.
[56] Qiuyu Xiao, Michael K. Reiter, and Yinqian Zhang. Mitigating Storage Side Channels Using Statistical Privacy Mechanisms. In *ACM CCS*, 2015.
[57] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *IEEE S&P*, 2011.
[58] Sanchuan Chen, Fangfei Liu, Zeyu Mi, Yinqian Zhang, Ruby B Lee, Haibo Chen, and XiaoFeng Wang. Leveraging Hardware Transactional Memory for Cache Side-Channel Defenses. In *ACM AsiaCCS*, 2018.
[59] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. Cloudradar: A Real-Time Side-Channel Attack Detection System in Clouds. In *Springer RAID*, 2016.
[60] Ziqiao Zhou, Michael K Reiter, and Yinqian Zhang. A Software Approach to Defeating Side Channels in Last-Level Caches. In *ACM CCS*, 2016.
[61] Zhang Xu, Haining Wang, and Zhenyu Wu. A Measurement Study on Co-residence Threat inside the Cloud. In *USENIX Security*, 2015.
[62] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A Placement Vulnerability Study in Multi-Tenant Public Clouds. In *USENIX Security*, 2015.
[63] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *USENIX ATC*, 2018.
[64] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. DoS Attacks on Your Memory in Cloud. In *ACM AsiaCCS*, 2017.
[65] Qun Huang and Patrick PC Lee. An Experimental Study of Cascading Performance Interference in a Virtualized Environment. *ACM SIGMETRICS*, 2013.
[66] Zhang Xu, Haining Wang, Zichen Xu, and Xiaorui Wang. Power Attack: An Increasing Threat to Data Centers. In *NDSS*, 2014.
[67] Chao Li, Zhenhua Wang, Xiaofeng Hou, Haopeng Chen, Xiaoyao Liang, and Minyi Guo. Power Attack Defense: Securing Battery-Backed Data Centers. In *IEEE ISCA*, 2016.
[68] Xing Gao, Zhang Xu, Haining Wang, Li Li, and Xiaorui Wang. Reduced cooling redundancy: A new security vulnerability in a hot data center. In *NDSS*, 2018.
[69] Mohammad A Islam, Shaolei Ren, and Adam Wierman. Exploiting a Thermal Side Channel for Power Attacks in Multi-Tenant Data Centers. In *ACM CCS*, 2017.
[70] Mohammad A Islam, Luting Yang, Kiran Ranganath, and Shaolei Ren. Why Some Like It Loud: Timing Power Attacks in Multi-Tenant Data Centers Using an Acoustic Side Channel. *ACM SIGMETRICS*, 2018.
[71] Mohammad A Islam and Shaolei Ren. Ohm's Law in Data Centers: A Voltage Side Channel for Timing Power Attacks. In *ACM CCS*, 2018.