A Study on the Security Implications of Information Leakages in Container Clouds

Xing Gao[1], Benjamin Steenkamer[2], Zhongshu Gu[3], Mehmet Kayaalp[4], Dimitrios Pendarakis[3], Haining Wang[2]
[1]University of Memphis, [2]University of Delaware,
[3]IBM T.J. Watson Research Center, [4]University of New Hampshire
xgao1@memphis.edu, {bsteen, hnw}@udel.edu, {zgu, dimitris}@us.ibm.com, Mehmet.Kayaalp@unh.edu

Abstract—Container technology provides a lightweight operating system level virtual hosting environment. Its emergence profoundly changes the development and deployment paradigms of multi-tier distributed applications. However, due to the incomplete implementation of system resource isolation mechanisms in the Linux kernel, some security concerns still exist for multiple containers sharing an operating system kernel on a multi-tenancy container-based cloud service. In this paper, we first present the information leakage channels we discovered that are accessible within containers. Such channels expose a spectrum of system-wide host information to containers without proper resource partitioning. By exploiting such leaked host information, it becomes much easier for malicious adversaries (acting as tenants in a container cloud) to launch attacks that might impact the reliability of cloud services. We demonstrate that the information leakage channels could be exploited to infer private data, detect and verify co-residence, build covert channels, and launch more advanced cloud-based attacks. We discuss the root causes of the containers' information leakage and propose a two-stage defense approach. As demonstrated in the evaluation, our defense is effective and incurs trivial performance overhead.

Index Terms—Container, Information Leakage, Namespace.

1 Introduction

Cloud computing has been widely adopted to consolidate computing resources. Multi-tenancy is the enabling feature of cloud computing that allows computation instances from different tenants running on the same physical server. Among different types of cloud services, the multi-tenancy container cloud has recently emerged as a lightweight alternative to conventional virtual machine (VM) based cloud infrastructures. Container is an operating system (OS) level virtualization technology with multiple building blocks in the Linux kernel, including resource isolation/control techniques (e.g., *namespace* and *cgroup*) and security mechanisms (e.g., *Capabilities*, *SELinux*, *AppArmor*, and *seccomp*). By avoiding the overhead of additional abstraction layers, containers are able to achieve near-native performance and outperform VM-based systems in almost all aspects [2], [15], [34]. In addition, the advent of container management and orchestration systems, such as *Docker* and *Kubernetes*, have profoundly changed the ecosystem of building, shipping, and deploying multi-tier distributed applications on the cloud.

Despite the success of container services, there always exist security and privacy concerns for running multiple containers, presumably belonging to different tenants, on the same OS kernel. To support multi-tenancy on container clouds, we have observed on-going efforts in the Linux kernel to enforce cross-container isolation and de-privilege user-level containers. Existing container-enabling kernel features have greatly shrunk the attack surface exposed to container tenants and could restrain most existing malicious attacks. However, not all subsystems of the Linux kernel can distinguish execution contexts between a container and a host, and thus they might expose system-wide information to containerized applications. Some subsystems are considered to be of low priority for container adaptations. The rest are facing implementation difficulties for transforming their code base, and their maintainers are reluctant to accept drastic changes. In order to close these loopholes, current container runtime software and container cloud providers typically leverage access control policies to mask the user-kernel interfaces of these container-oblivious subsystems. However, such manual and temporary fixes could only cover a small fraction of the exposed attack surfaces.

In this paper, we systematically explore and identify the in-container leakage channels that may accidentally expose information of host OSes and co-resident containers. Such information leakages include host-system state information (e.g., power consumption, performance data, global kernel data, and asynchronous kernel events) and individual process execution information (e.g., process scheduling, cgroups, and process running status). The distinguishing characteristic information exposed at specific timings could help uniquely identify a physical machine. Furthermore, a malicious tenant may optimize attack strategies and maximize attack effects by acquiring the system-wide knowledge in advance. We discover these leakage channels in our local testbed on *Docker* and *LinuX Container (LXC)* and verify their (partial) existence on five public commercial multi-tenancy container cloud services.

We demonstrate that multiple security implications exist for those information leakage channels. In general, despite being mounted read-only, those channels could still be exploited by malicious container tenants to infer private data of other containers on the same physical machine, detect and verify co-residence, and build covert channels to stealthily transfer information. We present several techniques for attackers to infer co-residence by exploiting those information leakage channels. Our method is more resilient to the noise in the cloud environment compared with traditional methods like cache-based covert channels. We rank those channels based on their risk levels. We find that the system-wide value of multiple channels can be impacted by activities in container instances. Through dedicated manipulation of workloads running in containers, attackers can achieve reliable and high-speed covert channels to break the isolation mechanisms deployed in clouds. For instance, attackers can stealthily transfer bits between containers by intentionally acquiring and releasing locks without incurring networking activity. This leaked information can be observed by all the containers on the same physical machine. In order to reveal the security risks of these leakage channels, we build two covert channels employing different techniques and test their bandwidth in a real multi-tenancy cloud environment.

We further design an advanced attack, denoted as *synergistic power attack*, to exploit the seemingly innocuous information leaked through these channels. We demonstrate that such information exposure could greatly amplify the attack effects, reduce the attack costs, and simplify the attack orchestration. Power attacks have proved to be real threats to existing data centers [28], [47]. With no information of the running status of underlying cloud infrastructures, existing power attacks can only launch power-intensive workloads blindly to generate power spikes, with the hope that high spikes could trip branch circuit breakers to cause power outages. Such attacks could be costly and ineffective. However, by learning the system-wide status information, attackers can (1) pick the best timing to launch an attack, i.e., superimpose the power-intensive workload on the existing power spikes triggered by benign workloads, and (2) synchronize multiple power attacks on the same physical machine/rack by detecting proximity-residence of controlled containers. We conduct proof-of-concept experiments on one real-world container cloud service and quantitatively demonstrate that our attack is able to yield higher power spikes at a lower cost.

We further analyze in depth the root causes of these leakage channels and find that such exposures are due to the incomplete coverage of container implementation in the Linux kernel. We propose a two-stage defense mechanism to address this problem in container clouds. In particular, to defend against the *synergistic power attacks*, we design and implement a power-based namespace in the Linux kernel to partition power consumption at a finer-grained (container) level. We evaluate our power-based namespace from the perspectives of accuracy, security, and performance overhead. Our experimental results show that our system can neutralize container-based power attacks with trivial performance overhead.

The rest of this paper is organized as follows. Section 2 introduces the background of container technology and describes power attack threats on data centers. Section 3 presents the in-container leakage channels discovered by us and their leaked information. Section 4 demonstrates the construction of high-bandwidth reliable covert channels based on the leaked system-wide data. Section 5 details the synergistic power attack that leverages the leaked information through these channels. Section 6 presents a general two-stage defense mechanism and the specific design and implementation of our power-based namespace in the Linux kernel. Section 7 shows the evaluation of our defense framework from different aspects. Section 8 discusses the limitations and future work. Section 9 surveys related work, and we conclude in Section 10.

2 Background

In this section, we briefly describe the background knowledge of three topics: internals of Linux containers, multi-tenancy container cloud services, and existing power attacks in data centers.

2.1 Linux Kernel Support for Container Technology

Containers depend on multiple independent Linux kernel components to enforce isolation among user-space instances. Compared to VM-based virtualization approaches, multiple containers share the same OS kernel, thus eliminating additional performance overheads for starting and maintaining VMs. Containers have received much attention from the industry and have grown rapidly in recent years for boosting application performance, enhancing developer efficiency, and facilitating service deployment. Here we introduce two key techniques, *namespace* and *cgroup*, that enable containerization on Linux.

2.1.1 Namespace

The first namespace was introduced in the Linux kernel 2.4.19. The key idea of namespace is to isolate and virtualize system resources for a group of processes, which form a container. Each process can be associated with multiple namespaces of different types. The kernel presents a customized (based on namespace types) view of system resources to each process. The modifications to any namespaced system resources are confined within the associated namespaces, thus incurring no system-wide changes.

The current kernel has seven types of namespaces: *mount (MNT)* namespace, *UNIX timesharing system (UTS)* namespace, *PID* namespace, *network (NET)* namespace, *inter-process communications (IPC)* namespace, *USER* namespace, and *CGROUP* namespace. The *MNT* namespace isolates a set of file system mount points. In different *MNT* namespaces, processes have different views of the file system hierarchy. The *UTS* namespace allows each container to have its own host name and domain name, and thus a container could be treated as an independent node. The *PID* namespace virtualizes the process identifiers (pids). Each process has two pids: one pid within its *PID* namespace and one (globally unique) pid on the host. Processes in one container could only view processes within the same *PID* namespace. A *NET* namespace contains separate virtual network devices, IP addresses, ports, and IP routing tables. The *IPC* namespace isolates inter-process communication resources, including signals, pipes, and shared memory. The *USER* namespace was recently introduced to isolate the user and group ID number spaces. It creates a mapping between a root user inside a container to an unprivileged user on the host. Thus, a process may have full privileges inside a user namespace, but it is de-privileged on the host. The *CGROUP* namespace virtualizes the cgroup resources, and each process can only have a containerized cgroup view via *cgroupfs* mount and the */proc/self/cgroup* file.

2.1.2 Cgroup

In the Linux kernel, cgroup (i.e., control group) provides a mechanism for partitioning groups of processes (and all their children) into hierarchical groups with controlled behaviors. Containers leverage the cgroup functionality to apply per-cgroup resource limits to each container instance, thus preventing a single container from draining host resources. Such controlled resources include CPU, memory, block IO, network, etc. For the billing model in cloud computing, cgroup can also be used for assigning corresponding resources to each container and accounting for their usage. Each cgroup subsystem provides a unified *sysfs* interface to simplify the cgroup operations from the user space.

2.2 Container Cloud

With these kernel features available for resource isolation and management, the Linux kernel can provide the lightweight virtualization functionality at the OS level. More namespace and cgroup subsystems are expected to be merged into the upstream Linux kernel in the future to enhance the container security. Containerization has become a popular choice for virtual hosting in recent years with the maturity of container runtime software. *LXC* is the first complete implementation of the Linux container manager built in 2008. *Docker*, which was built upon *LXC* (now with *libcontainer*), has become the most popular container management tool in recent years. *Docker* can wrap applications and their dependencies (e.g., code, runtime, system tools, and system libraries) into an image, thus guaranteeing that application behaviors are consistent across different platforms.

A large number of cloud service providers, including Amazon ECS, IBM Bluemix, Microsoft Azure, and Google Compute Engine, have already provided container cloud services. For multi-tenancy container cloud services, containers can either run on a *bare metal* physical machine or a *virtual machine*. In both situations, containers from different tenants share the same Linux kernel with the host OS.

2.3 Covert Channels

Covert channels enable the communication between isolated entities by exploiting shared resources to break the isolation mechanisms. With the characteristics of being stealthy, covert channels can be used to retrieve sensitive information and bypass access control on standard channels. It has been widely accepted that today's cloud environment is vulnerable to covert channels even with the isolation techniques enforced by virtual machines and containers. Various techniques have been proposed to establish covert channels in multi-tenancy cloud environment. Ristenpart et al. [35] reported a bit rate of 0.2bps exploiting shared L2 data cache in a public cloud. Xu et al. [45] used last level caches between VMs to build a covert channel with a bandwidth of 3.2bps on Amazon EC2 environment. Wu et al. [42] achieved an 110 bps with an error rate of 0.75% covert channel exploiting the contention in memory bus. Masti et al. [31] successfully built covert channels by obtaining the temperature reading through the on-chip sensors with a bit rate of 12.5bps, and Bartolini et al. [11] improved the thermal-based covert channels to 50bps on nearby cores. In the cloud environment, besides the threats of leaking sensitive data, covert channels can be further abused to detect and verify co-residence, which is the prerequisite for most cloud-based attacks. By successfully transferring information via covert channels, attackers can confirm whether two instances co-locate on the same host server or not. Thus, to secure clouds by building robust defense mechanisms, discovering new types of techniques for constructing covert channels is one important step, and has been intensively studied by previous research.

2.4 Power Attacks on Data Centers

Power attacks have been demonstrated to be realistic threats to existing cloud infrastructures [28], [47]. Considering the cost of upgrading power facilities, current data centers widely adopt power oversubscription to host the maximum number of servers within the existing power supply capabilities. The safety guarantees are based on the assumption that multiple adjacent servers have a low chance of reaching peaks of power consumption simultaneously. While power oversubscription allows deploying more servers without increasing power capacity, the reduction of power redundancy increases the possibility of power outages, which might lead to forced shutdowns for servers on the same rack or on the same power distribution unit (PDU). Even normal workloads may generate power spikes that cause power outages. Facebook recently reported that it prevented 18 potential power outages within six months in 2016 [41]. The situation would have been worse if malicious adversaries intentionally drop power viruses to launch power attacks [16], [17]. The consequence of a power outage could be devastating, e.g., Delta Airlines encountered a shutdown of a power source in its data center in August 2016, which caused large-scale delays and cancellations of flights [8]. Recent research efforts [28], [47] have demonstrated that it is feasible to mount power attacks on both traditional and battery-backed data centers.

Launching a successful power attack requires three key factors: (1) gaining access to servers in the target data center by legitimately subscribing services, (2) steadily running moderate workloads to increase the power consumption of servers to their capping limits, (3) abruptly switching to power-intensive workloads to trigger power spikes. By causing a power spike in a short time window, a circuit breaker could be tripped to protect servers from physical damages caused by overcurrent or overload.

The tripping condition of a circuit breaker depends on the strength and duration of a power spike. In order to maximize the attack effects, adversaries need to run malicious workloads on a group of servers belonging to the same rack or PDU. In addition, the timing of launching attacks is also critical. If a specific set of servers (e.g., on the same rack) in a data center have already run at their peak power state, the chance of launching a successful power attack will be higher [47].

The techniques of power capping [27] have been designed to defend against power attacks. At the rack and PDU level, by monitoring the power consumption, a data center can restrict the power consumption of servers through a power-based feedback loop. At the host level, *Running Average Power Limit (RAPL)* is a technique for monitoring and limiting the power consumption for a single server. *RAPL* has been introduced by Intel since Sandy Bridge microarchitecture. It provides fine-grained CPU-level energy accounting at the microsecond level and can be used to limit the power consumption for one package.

Power capping mechanisms significantly narrow down the power attack surface, but it cannot address the problem of power oversubscription, which is the root cause of power outages in data centers. Although host-level power capping for a single server could respond immediately to power surges, the power capping mechanisms at the rack or PDU level still suffer from minute-level delays. Assuming attackers could deploy power viruses into physically adjacent servers, even if each server consumes power lower than its power capping limit, the aggregated power consumption of controlled servers altogether can still exceed the power supply capacity and trip the circuit breaker. We demonstrate in the following sections that malicious container tenants can launch *synergistic power attacks* by controlling the deployment of their power-intensive workloads and leveraging benign workloads in the background to amplify their power attacks.

3 Information Leakages in Container Clouds

As we mentioned in Section 2, the Linux kernel provides a multitude of supports to enforce resource isolation and control for the container abstraction. Such kernel mechanisms are the enabling techniques for running containers on the multi-tenancy cloud. Due to priority and difficulty levels, some components of the Linux kernel have not yet transformed to support containerization. We intend to systematically explore which parts of the kernel are left uncovered, what the root causes are, and how potential adversaries can exploit them.

3.1 Container Information Leakages

We first conduct experiments on our local Linux machines with *Docker* and *LXC* containers installed. The system is set up with default configurations, and all containers are launched with user privilege, similarly as on commercial container clouds. Linux provides two types of controlled interfaces from userspace processes to the kernel, system calls, and memory-based pseudo file systems. System calls are mainly designed for user processes to request kernel services. The system calls have strict definitions for their public interfaces and are typically backward compatible. However, memory-based pseudo file systems are more flexible for extending kernel functionalities (e.g., *ioctl*), accessing kernel data (e.g., *procfs*), and adjusting kernel parameters (e.g., *sysctl*). In addition, such pseudo file systems enable manipulating kernel data via normal file I/O operations. Linux has a number of memory-based pseudo file systems

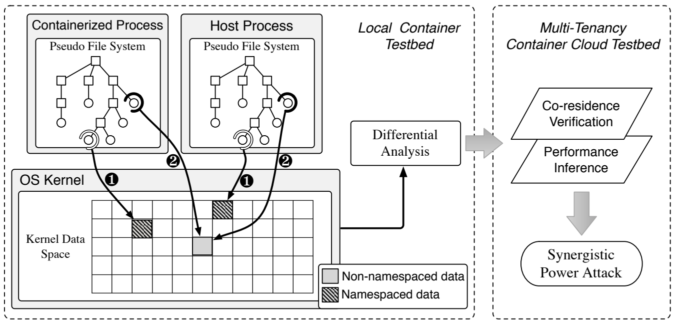


Fig. 1: The framework for information leakage detection and cloud inspection.

(e.g., *procfs*, *sysfs*, *devfs*, *securityfs*, *debugfs*, etc.) that serve the different purposes of kernel operations. We are more interested in *procfs* and *sysfs*, which are by default mounted by container runtime software.

As illustrated in the left part of Figure 1, we design a cross-validation tool to automatically discover these memory-based pseudo files that expose host information to containers. The key idea is to recursively explore all pseudo files under *procfs* and *sysfs* in two execution contexts, one running within an unprivileged container and the other running on the host. We align and reorder the files based on their file paths and then conduct pair-wise differential analysis on the contents of the same file between these two contexts. If the system resources accessed from a specific pseudo file has not been namespaced in the Linux kernel, the host and container reach the same piece of kernel data (as the case of ❷ in Figure 1). Otherwise, if properly namespaced, the container can retrieve its own private and customized kernel data (as the case of ❶ in Figure 1). Using this cross-validation tool, we can quickly identify the pseudo files (and their internal kernel data structures) that may expose system-wide host information to the container.

3.2 Leakage Channel Analysis

We list all pseudo files that may leak host information in Table 1. Those leakage channels contain different aspects of host information. Container users can retrieve kernel data structures (e.g., */proc/modules* shows the list of loaded modules), kernel events (e.g., */proc/interrupts* shows the number of interrupts per IRQ), and hardware information (e.g., */proc/cpuinfo* and */proc/meminfo* show the specification of CPU and memory, respectively). In addition, container users are able to retrieve performance statistics data through some channels. For example, containers can obtain hardware sensor data (if these sensors are available in the physical machine), such as power consumption for each package, cores, and DRAM through the *RAPL sysfs* interface, and the temperature for each core through the *Digital Temperature Sensor (DTS) sysfs* interface. Moreover, the usage of processors, memory, and disk I/O is also exposed to containers. While leaking such information seems harmless at first glance, it could be exploited by malicious adversaries to launch attacks. More detailed discussion is given in Section 5.

We further investigate the root causes of these information leakages by inspecting the kernel code (in the Linux kernel version 4.7). Generally, the leakage problems are caused

TABLE 1: LEAKAGE CHANNELS IN COMMERCIAL CONTAINER CLOUD SERVICES.

Leakage Channels	Leakage Information	Potential Vulnerability			Container Cloud Services[1]				
		Co-re	DoS	Info leak	$\mathbf{CC_1}$	$\mathbf{CC_2}$	$\mathbf{CC_3}$	$\mathbf{CC_4}$	$\mathbf{CC_5}$
`/proc/locks`	Files locked by the kernel	●	○	●	●	●	●	●	◐
`/proc/zoneinfo`	Physical RAM information	●	○	●	●	●	●	●	◐
`/proc/modules`	Loaded kernel modules information	○	○	●	●	●	●	●	●
`/proc/timer_list`	Configured clocks and timers	●	○	●	●	●	●	○	●
`/proc/sched_debug`	Task scheduler behavior	●	○	●	○	○	●	○	●
`/proc/softirqs`	Number of invoked softirq handler	●	●	●	●	●	●	●	●
`/proc/uptime`	Up and idle time	●	○	●	●	●	●	●	◐
`/proc/version`	Kernel, gcc, distribution version	○	○	●	●	●	●	●	●
`/proc/stat`	Kernel activities	●	●	●	●	●	●	●	◐
`/proc/meminfo`	Memory information	●	●	●	●	●	●	●	○
`/proc/loadavg`	CPU and IO utilization over time	●	○	●	●	●	●	●	◐
`/proc/interrupts`	Number of interrupts per IRQ	●	○	●	●	●	●	●	●
`/proc/cpuinfo`	CPU information	●	○	●	●	●	●	●	○
`/proc/schedstat`	Schedule statistics	●	○	●	●	●	●	●	◐
`/proc/sys/fs/*`	File system information	●	○	●	●	●	○	●	●
`/proc/sys/kernel/random/*`	Random number generation info	●	○	●	●	●	●	●	●
`/proc/sys/kernel/sched_domain/*`	Schedule domain info	●	○	●	●	●	●	●	●
`/proc/fs/ext4/*`	Ext4 file system info	●	○	●	●	●	●	●	●
`/sys/fs/cgroup/net_prio/*`	Priorities assigned to traffic	○	○	●	●	●	○	○	○
`/sys/devices/*`	System device information	●	●	●	●	●	●	○	○
`/sys/class/*`	System device information	○	●	●	●	●	●	○	○

by the incomplete implementation of namespaces in the kernel. To be more specific, we summarize the two main causes as follows: (1) Context checks are missing for existing namespaces, and (2) some Linux subsystems are not (fully) namespaced. We give two case studies on *net_prio.ifpriomap* and *RAPL in containers* to reveal the origins of leakages.

3.2.1 Case study I — net_prio.ifpriomap

The pseudo file *net_prio.ifpriomap* (under */sys/fs/cgroup/net_prio*) contains a map of the priorities assigned to traffic starting from processes in a cgroup and leaving the system on various interfaces. The data format is in the form of *[ifname priority]*. We find that the kernel handler function hooked at *net_prio.ifpriomap* is not aware of the *NET* namespace, and thus it discloses all network interfaces on the physical machine to the containerized applications. To be more specific, the read operation of *net_prio.ifpriomap* is handled by the function `read_priomap`. Tracing from this function, we find that it invokes `for_each_netdev_rcu` and sets the first parameter as the address of `init_net`. It iterates all network devices of the host, regardless of the *NET* namespace. Thus, from the view of a container, it can read the names of all network devices of the host.

3.2.2 Case study II — RAPL in containers

RAPL was recently introduced by Intel for setting power limits for processor packages and DRAM of a single server, which can respond at the millisecond level [21]. In the container cloud, the *sysfs* interface of RAPL, which is located at */sys/class/powercap/intel-rapl*, is accessible to containers. Therefore, it is possible for container tenants to obtain the system-wide power status of the host, including the core, DRAM, and package, through this *sysfs* interface. For example, container users can read the current energy counter in micro joules from the pseudo file *energy_uj*. The function handler of *energy_uj* in the Intel *RAPL* Linux driver is `get_energy_counter`. This function retrieves the raw energy data from the *RAPL MSR*. As namespace has not been implemented for the power data, the `energy_raw` pointer refers to the host's energy consumption data.

We further investigate the information leakage problems on container cloud services that adopt the *Docker*/*LXC* container engine. We choose five commercial public multi-tenancy container cloud services for leakage checking and present the results in Table 1. We anonymize the names ($\mathbf{CC_i}$ stands for $\mathbf{i}^{th}$ **C**ontainer **C**loud) of these container cloud services before the cloud providers patch the channels. We confirm the existence of leakage if there is an inconsistency between the results and the configuration of our container instance. The ● indicates that the channel exists in the cloud, while the ○ indicates the opposite. We find that most of the leakage channels on local machines are also available in the container cloud services. Some of them are unavailable due to the lack of support for specific hardware (e.g., Intel processor before Sandy Bridge or AMD processors that do not support RAPL). For cloud $\mathbf{CC_5}$, we find that the information of some channels is different from our local testbed, which means that the cloud vendor has customized some additional restrictions. For example, only the information about the cores and memory belonging to a tenant is available. However, those channels partially leak the host information and could still be exploited by advanced attackers. We mark them as ◐.

3.3 Inference of Co-resident Container

We further look in depth into specific cases to see whether they could be exploited to detect co-resident containers.

3.3.1 Co-residence problems in cloud settings

Co-residence is a well-known research problem in cloud security. In order to extract a victim's information, adversaries tend to move malicious instances to the same physical host with the victim. Zhang et al. have shown that it is possible for an attacker to hijack user accounts [51] and extract private keys [50] with co-resident instances. In addition, the cost of

1. The most recent check on the leakage channels was made on November 28, 2016.

achieving co-residence is rather low [40]. Co-residence still remains a problem in existing clouds, due to the intention of consolidating server resources and reducing cost. Traditional methods to verify co-residence are based on cache [48] or memory-based leakage channels [42]. The accuracy of those methods may downgrade due to the high noise in cloud settings.

3.3.2 Approaches and results of checking co-resident containers

Since containers can read the host information through the leakage channels we discovered, we tend to measure whether some channels can be used for checking container co-residence. We define three metrics, namely *uniqueness* ($\mathbb{U}$), *variation* ($\mathbb{V}$), and *manipulation* ($\mathbb{M}$) to quantitatively assess each channel's capability of inferring co-residence.

The metric $\mathbb{U}$ indicates whether this channel bestows characteristic data that can uniquely identify a host machine. It is the most important and accurate factor for determining whether two containers locate on the same host. We have found 17 leakage channels (ranked top 17 in Table 2) that satisfy this metric. Generally we can classify these channels into three groups:

1) Channels containing unique static identifiers. For example, *boot_id* under */proc/sys/kernel/random* is a random string generated at boot time and is unique for each running kernel. If two containers can read the same *boot_id*, this is a clear sign that they are running on the same host kernel. The data for channels in this group are both static and unique.

2) Channels into which container tenants can dynamically implant unique signatures. For example, from */proc/sched_debug*, container users can retrieve all active process information of the host through this interface. A tenant can launch a process with a uniquely crafted task name inside the container. From the other containers, they can verify co-residence by searching this task name in their own *sched_debug*. Similar situations apply to *timer_list* and *locks*.

3) Channels containing unique dynamic identifiers. For example, */proc/uptime* has two data fields: system up time and system idle time in seconds since booting. They are accumulated values and are unique for every host machine. Similarly, *energy_uj* in the *RAPL sysfs* interface is the accumulated energy counter in micro joules. The data read from channels in this group change at real time, but are still unique to represent a host. We rank the channels in this group based on their growth rates. A faster growth rate indicates a lower chance of duplication.

The metric $\mathbb{V}$ demonstrates whether the data change with time. With this feature available, two containers can make snapshots of this pseudo file periodically at the same time. Then they can determine co-residence by checking whether two data snapshot traces match with each other. For example, starting from the same time, we can record *MemFree* in */proc/meminfo* from two containers every second for one minute. If these two 60-point data traces match with each other, we are confident that these two containers run on the same host. Each channel contains a different capacity of information for inferring co-residence, which can be naturally measured via the *joint Shannon entropy*. We define the entropy $\mathbb{H}$ in Formula (1). Each channel C contains multiple independent data fields X_i, and n represents the number of independent data fields. Each X_i has possible values $\{x_{i1}, \cdots, x_{im}\}$. We rank the capability of revealing co-residence for the nine channels (for which $\mathbb{U}$=*False* and $\mathbb{V}$=*True*) based on their entropy results in Table 2.

TABLE 2: LEAKAGE CHANNELS FOR CO-RESIDENCE VERIFICATION.

Leakage Channels	$\mathbb{U}$	$\mathbb{V}$	$\mathbb{M}$	**Rank**
/proc/sys/kernel/random/boot_id	●	○	○	■■■■■
/sys/fs/cgroup/net_prio/net_prio.ifpriomap	●	○	○	■■■■■
/proc/sched_debug	●	●	●	■■■■■
/proc/timer_list	●	●	●	■■■■■
/proc/locks	●	●	●	■■■■■
/proc/uptime	●	●	◐	■■■■
/proc/stat	●	●	◐	■■■■
/proc/schedstat	●	●	◐	■■■■
/proc/softirqs	●	●	◐	■■■■
/proc/interrupts	●	●	◐	■■■■
/sys/devices/system/node/node#/numastat	●	●	◐	■■■■
/sys/class/powercap/.../energy_uj[2]	●	●	◐	■■■■
/sys/devices/system/.../usage[3]	●	●	◐	■■■■
/sys/devices/system/.../time[4]	●	●	◐	■■■■
/proc/sys/fs/dentry-state	●	●	◐	■■■
/proc/sys/fs/inode-nr	●	●	◐	■■■
/proc/sys/fs/file-nr	●	●	◐	■■■
/proc/zoneinfo	○	●	◐	■■
/proc/meminfo	○	●	◐	■■
/proc/fs/ext4/sda#/mb_groups	○	●	◐	■■
/sys/devices/system/node/node#/vmstat	○	●	◐	■■
/sys/devices/system/node/node#/meminfo	○	●	◐	■■
/sys/devices/platform/.../temp#_input[5]	○	●	◐	■
/proc/loadavg	○	●	◐	■
/proc/sys/kernel/random/entropy_avail	○	●	◐	■
/proc/sys/kernel/.../max_newidle_lb_cost[6]	○	●	○	■
/proc/modules	○	○	○	▫
/proc/cpuinfo	○	○	○	▫
/proc/version	○	○	○	▫
				Low High

$$\mathbb{H}[C(X_1, \cdots, X_n)] = \sum_{i=1}^{n} [- \sum_{j=1}^{m} p(x_{ij}) \log p(x_{ij})]. \quad (1)$$

The metric $\mathbb{M}$ indicates whether the container tenants can *manipulate* the data. We mark a channel ● if tenants can directly implant specially-crafted data into it. For example, we can create a timer in a program with a special task name inside a container. This task name and its associated timer will appear in */proc/timer_list*. Another container can search for this special task name in the *timer_list* to verify co-residence. We mark a channel ◐ if tenants can only indirectly influence the data in this channel. For example, an attacker can use *taskset* command to bond a computing-intensive workload to a specific core, and check the CPU utilization, power consumption, or temperature from another container. Those entries could be exploited by advanced attackers as covert channels to transmit signals.

2. /sys/class/powercap/intel-rapl:#/intel-rapl:#/energy_uj
3. /sys/devices/system/cpu/cpu#/cpuidle/state#/usage
4. /sys/devices/system/cpu/cpu#/cpuidle/state#/time
5. /sys/devices/platform/coretemp.#/hwmon/hwmon#/temp#_input
6. /proc/sys/kernel/sched_domain/cpu#/domain#/max_newidle_lb_cost

Algorithm 1 Covert Channel Protocol Based on Locks

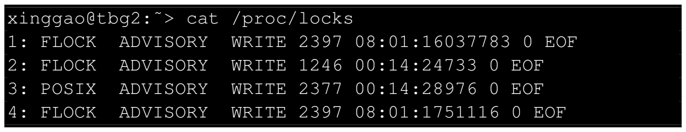
```
$N$: Number of transmissions.
$M$: Number of bits of data per transmission.
$L[M]$, $L_{signal}$, $L_{ACK}$: Locks on M data files, lock on the signal file, and lock on the ACK file.
$T_{hand}$: an amount of time for handshake.
$D_{Send}[N]$, $D_{Recv}[N]$: N transmissions to send and receive.
```

```
Sender Operations:
for amount of time $T_{hand}$ do
    Set $L[M]$, $L_{signal}$;
    Sleep for an amount of time;
    Release $L[M]$, $L_{signal}$;
end for
for amount of time $T_{hand}$ do
    Record entries in /proc/locks;
    Sleep for an amount of time;
end for
for each recorded entry in /proc/locks do
    if Number of toggles >= THRESHOLD then
        $ACK_{recevier}$ = True;
    end if
end for
```

```
Receiver Operations:
for amount of time $T_{hand}$ do
    Record entries in /proc/locks and sleep;
end for
for each recorded entry in /proc/locks do
    if Number of toggles >= THRESHOLD then
        Remember lock's device number;
    end if
end for
if Number of recorded valid locks $== M + 1$ then
    for amount of time $T_{hand}$ do
        Set $L_{ACK}$;
        Sleep for an amount of time;
        Release $L_{ACK}$;
    end for
end if
```

```
Sending Data:
for i = 0 to N - 1 do
    for j = 0 to M - 1 do
        if $D_{Send}[i][j] == 1$ then
            Set L[j];
        else
            Release L[j];
        end if
    end for
    Set $L_{signal}$;
    Sleep for an amount of time;
    Release $L_{signal}$;
    Wait for $L_{ACK}$;
end for
```

```
Receiving Data:
while connection is active do
    Record entries in /proc/locks;
    if $L_{signal}$ is presented then
        for j = 0 to M - 1 do
            if L[j] is presented then
                $D_{Recv}[i][j] = 1$;
            else
                $D_{Recv}[i][j] = 0$;
            end if
        end for
        i++;
        Set $L_{ACK}$;
        Sleep for an amount of time;
        Release $L_{ACK}$;
    end if
end while
```



```
xinggao@tbg2:~> cat /proc/locks
1: FLOCK  ADVISORY  WRITE 2397 08:01:16037783 0 EOF
2: FLOCK  ADVISORY  WRITE 1246 00:14:24733 0 EOF
3: POSIX  ADVISORY  WRITE 2377 00:14:28976 0 EOF
4: FLOCK  ADVISORY  WRITE 2397 08:01:1751116 0 EOF
```

Fig. 2: The information exposed by /proc/locks.

For those channels that do not have these $\mathbb{U}$ $\mathbb{V}$ $\mathbb{M}$ properties, we consider them hard to be exploited. For example, most servers in a cloud data center probably install the same OS distribution with the same module list. Although */proc/modules* leaks the information of loaded modules on the host, it is difficult to use this channel to infer co-resident containers.

4 Constructing Covert Channels

The channels containing the metric *manipulation* ($\mathbb{M}$) can be further exploited to build covert channels between two containers. We demonstrate that both dynamic identifiers and performance data in those information leakage channels could be abused to transfer information. In particular, we implement two channels and test their throughputs and error rates.

4.1 Covert channel based on unique dynamic identifiers

We take */proc/locks* as an example to demonstrate that covert channels can be built upon unique dynamic identifiers. The channel */proc/locks* presents an overview of the information of all locks in an OS, including lock types, corresponding processes, as well as inode numbers, as seen in Figure 2. Unfortunately, the data in this channel are not protected by namespaces. A container can obtain the information of all locks on the host server. This leakage also breaks the pid namespace since the global pids of all locks are leaked. Linux kernel maintainers partially fixed */proc/locks* in the kernel version 4.9 by masking all pids out of current pid

Algorithm 2 Covert Channel Protocol Based on /proc/meminfo

```
$Mem_{free}$: Amount of system memory currently unused by the system.
$Mem_{high}$, $Mem_{low}$: Predefined amount of memory representing a bit 1 and a bit 0, respectively.
$Limit_1$, $Limit_0$: Maximum value of $Mem_{free}$ that will be recognized as 1 and 0, respectively.
$D_{start}[M]$, $D_{send}[N]$, $D_{recv}[N]$: Handshake sequence, sending data bits, receiving data bits.
$M_{recording}[]$: the recordings of free memory at the receiver.
Sender Operations:
for i = 0 to M-1 do
    if $D_{start}[i] == 1$ then
        Allocate $Mem_{high}$;
    else
        Allocate $Mem_{low}$;
    end if
    Set memory chunk, sleep, and free;
end for
for i = 0 to N-1 do
    if $D_{send}[i] == 1$ then
        Allocate $Mem_{high}$;
    else
        Allocate $Mem_{low}$;
    end if
    Set memory chunk, sleep, and free;
end for
Receiver Operations:
Recording current value of $Mem_{free}$ to $M_{recording}$;
Obtain average $Mem_{free}$;
Calculate $Limit_1$ and $Limit_0$;
for i = 0 to $M_{recording}$.length - 1 do
    if $M_{recording}[i]$ is a local minimum and $M_{recording}[i] <= Limit_1$ then
        $D_{recv}[i]$ = 1;
    else if $M_{recording}[i]$ is a local minimum and $M_{recording}[i] <= Limit_0$ then
        $D_{recv}[i]$ = 0;
    end if
end for
```

namespace to zero. However, other information such as the number of locks and inode information are still visible within containers.

We construct a covert channel based on the */proc/locks*. Specifically, the sender can lock a file to represent 1, and release the lock to represent 0. While the locks and files are not shared among containers, the existence of locks can be checked by the receiver in */proc/locks*. By continuously checking the status of a specific lock, the information can be transferred. Additionally, multiple locks can be set simultaneously to transfer multiple bits. To build a reliable high-bandwidth covert channels, attackers need to take several factors into consideration. We detail the procedures on constructing the lock-based covert channel in Algorithm 1.

The channel */proc/locks* contains internal kernel data, thus it changes all the time. Especially in a cloud environment where noise exists, the contents might fluctuate tremendously. The first step for transmitting data blocks reliably is to handshake: the receiver needs to figure out all the locks that are used by the sender. In particular, the handshake procedure is responsible for: (1) setting up as a start point for data transmission; (2) anchoring locks for transferring a block of bits simultaneously; (3) synchronizing the sender and receiver.

We achieve the handshake by creating a specific pattern for each data lock. For a simplified case, the sender keeps getting and releasing locks in a short period time. The receiver then checks the number of toggles — locking or unlocking a file — for each data lock. If the number of toggles surpasses a specific threshold, the receiver records this lock by tracking the inode number, which is unchanged for the same lock on a specific file. At the same time, we use an extra lock to represent a transmission signal, which is used to notify the receiver about the starting of each round of transmission.

After creating the specific pattern, in theory the sender can start the transmission immediately. However, the time of the processing procedure in the receiver side is unknown and non-deterministic, especially in the case that a large amount of locks exist in the multi-tenancy cloud environment. A block of bits might be lost if the sender transmits the data too quickly. While the sender can wait a while before sending the next block of data, such a method will dramatically impact the transmission speed.

We further add an ACK lock for synchronizing the sender and receiver. After obtaining all data locks, the receiver confirms by setting the ACK lock. The detection of the ACK lock on the sender side is similar to the detection method of other data locks on the receiver side. The receiver enters into a ready state after replying with the ACK lock, and waits for the data transmission.

For data transmission, the sender sends a block of data in each round by getting or releasing data locks. For example, eight locks can represent one byte. The receiver decodes the data by checking the status of data locks. Once receiving the ACK lock, the sender starts the next round of data transmission.

4.2 Covert channel based on performance data variation

With the information leakages, container users can retrieve system-wide performance statistics data of the host server. For example, containers can obtain CPU utilization for each core by */proc/stat* and memory usage by */proc/meminfo* or */proc/vmstat*. The value of the performance data is affected by the running status of the container. Although one container only holds limited computing resource in the cloud environment, a malicious user in a container can carefully choose workloads running in the container. By generating a unique pattern in the performance channels, containers can build covert channels to transfer information. We build a

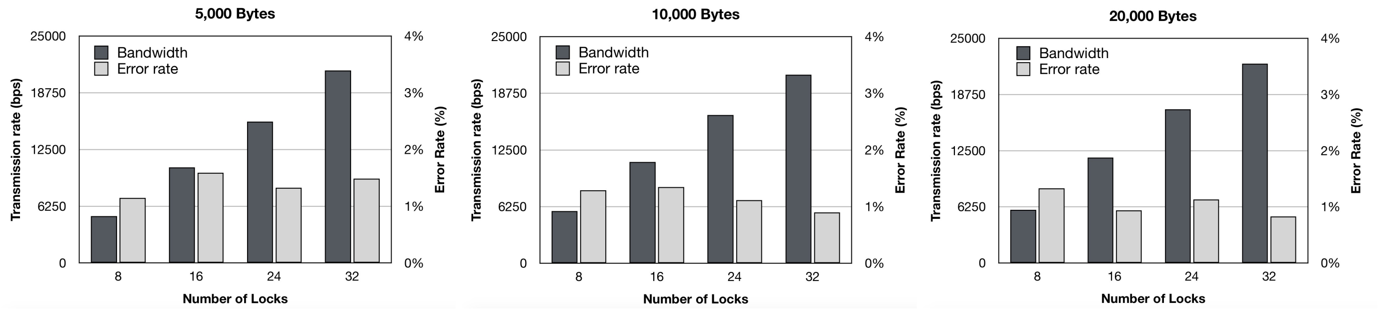


Fig. 3: Bandwidth and error rate of the lock-based covert channel.

covert channel by abusing the information of memory usage in */proc/meminfo*.

/proc/meminfo reports a large amount of valuable information about the system's memory usage, including the total amount of usable memory, the amount of physical RAM left unused by the system, and the total amount of dirty memory. Here we utilize the amount of unused memory in the system. In a cloud environment, this value might vary significantly since every container user might affect it. Such a significant variation is not desirable for building reliable covert data transmission. However, without running memory-intensive workloads, the usage might fluctuate within an acceptable range. We first record the unused memory on the server for an amount of time to get a baseline value. If the value does not change rapidly and significantly, it indicates that the covert channel can be built. Then, the sender container can allocate different amounts of memory to represent bit 1 and 0 (e.g., 100MB for bit 1 and 50MB for bit 0), which will cause the `MemFree` field in */proc/meminfo* to drop immediately. The receiver side can simply translate the data by monitoring the changes of free memory. The `MemFree` varies among three levels: around baseline case, bit 1, and bit 0. The sender would free all allocated memory after transmitting either a 1 or 0, so that the receiver side can know the end of last transmission and prepare for the next bit.

To ensure reliable transmission, a handshake is the first necessary procedure between the sender and receiver. The sender can choose to send a fixed pattern, such as eight straight bits 1, to initiate a transmission. Once the receiver side gets the handshake pattern, it will enter the data receiving mode. The detailed algorithm is listed in Algorithm 2. To reduce the impact of noise generated by the memory consumption of other containers, the receiver side will mark a data transmission once the `MemFree` falls in a predefined range. Meanwhile, increasing the amount of allocating memory can reduce the impact of noise. However, it will affect the transmission bandwidth because allocating and freeing memory consumes time. To reduce the interference from the environment, users could further design a high level reliable protocol, such as using a checksum, to ensure the integrity of the transferred data.

4.3 Experiments on a Multi-tenancy Container Cloud

To measure the performance under the environment with real noise, we choose one commercial multi-tenancy container cloud to test our covert channels. We repeatedly launch containers and verify the co-residence by checking the unique static identifiers. We make two containers located on the same physical machine.

Lock-based covert channel We test the bandwidth and error rate of the lock-based covert channel under three different sizes of data: 5,000 bytes, 10,000 bytes, and 20,000 bytes. We also choose four different numbers of locks (i.e., 8, 16, 24, 32) for measuring the performance. We empirically choose a threshold for the toggle times to ensure all locks can be correctly identified. The handshake process costs in the order of seconds. Then, all transmitted data is randomly generated. We recover all data in the receiver side and compare it with the original one. Figure 3 illustrates the experimental results on the real cloud. The left bar represents the number of bits transmitted per second, and the right bar shows the error rates.

As we see in the figure, the bandwidth of the lock-based covert channel with 8 locks is about 5,150 bps. Obviously, the bandwidth increases with more locks being used. With 32 data locks used in the channel (four bytes in each round of transmission), the bandwidth reaches 22,186 bps. Additionally, the error rate of all cases are under 2%. Those results indicate that the lock-based covert channel is highly reliable.

Memory-based covert channel For the covert channel built on */proc/meminfo*, we first set the memory allocation for bit 1 and 0 as 100,000KB and 50,000KB, respectively. We send 1,000 bits to test the performance. Then we gradually reduce the memory allocation until we fail to build the handshake. The results of bandwidth and error rate are listed in Table 3. The bandwidth is inversely proportional to the memory allocating size. However, the handshake process will fail if the allocation of memory is too small, leading to a huge amount of errors in the transmission. Finally, with allocating 65,000KB for bit 1 and 35,000KB for bit 0, we are able to achieve the bandwidth of 13.6 bps in the real cloud environment.

Comparison We show a comparison of previous approaches in terms of the bandwidth and error rate in Table 4.

TABLE 3: BANDWIDTH FOR MEMINFO BASED COVERT CHANNELS.

Bit 1 (kb)	Bit 0 (kb)	Bandwidth (bps)	Error Rate
100,000	50,000	8.79565	0.3%
90,000	45,000	9.72187	0.5%
80,000	40,000	11.0941	0.4%
70,000	35,000	12.7804	0.8%
65,000	35,000	13.603	0.5%

TABLE 4: COMPARISON AMONG DIFFERENT COVERT CHANNELS.

Method	Bandwidth (bps)	Error Rate
Lock (8 locks)	5149.9	1.14%
Meminfo	13.603	0.5%
Cache [32]	751	3.1%
Memory bus [42]	107.9±39.9	0.75%
Memory deduplication [43]	80	0%
Thermal [11]	45	1%

As demonstrated, the lock-based covert channel offers a very high data transmission rate while still keeping a low error rate. While the speed of the memory-based covert channel is limited, it can still reliably transmit data. More optimization approaches could be added to the memory-based covert channel. For example, we can use more levels to transmit multiple bits per transmission. We leave it as our future work.

It is worth noting that the covert channels built on the information leakage channels can work once two containers are co-located on the same physical server, regardless of the same CPU package or cores. Instead, a last level cache based channel only works when two instances share the same CPU package. It is also reported that memory bus based method can only work for almost 20% cases of co-residence in real cloud environment [9]. Thermal covert channels can only work when two cores are located near each other.

5 Synergistic Power Attack

At first glance, the leaked channels discovered in Section 3 seem difficult to be exploited other than as covert channels. Because both *procfs* and *sysfs* are mounted read-only inside the containers, malicious tenants can only read such information, but modification is not allowed. We argue that, by exploiting the leakage channels, attackers can make better decisions by learning the runtime status of the host machine.

In this section, we present a potential synergistic power attack in the scope of power outage threats that may impact the reliability of data centers. We demonstrate that adversaries can exploit these information leakages discovered by us to amplify the attack effects, reduce the attack costs, and facilitate attack orchestration. All experiments are conducted in a real-world container cloud.

5.1 Attack Amplification

The key to launching a successful power attack is to generate a short-time high power spike that can surpass the power facility's supply capacity. As we mentioned in 2.4, the root cause of power attacks is the wide adoption of power oversubscription, which makes it possible for power spikes to surpass the safe threshold. In addition, a rack-level power capping mechanism can only react in minute-level time granularity, leaving space for the occurrence of a short-time high power spike. In the most critical situation, the overcharging of power may trip the branch circuit breaker, cause a power outage, and finally bring down the servers. The heights of power spikes are predominantly determined by the resources that are controlled by attackers. Existing power attacks maximize the power consumption by customizing power-intensive workloads, denoted as power viruses. For example,

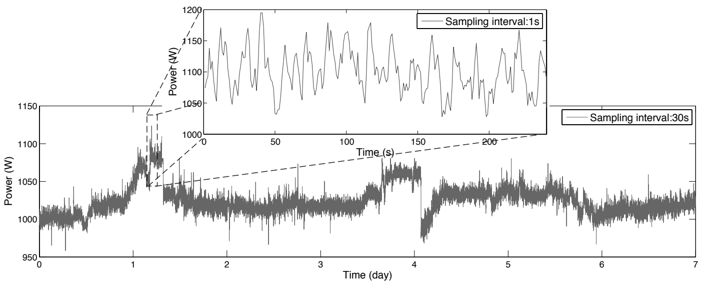


Fig. 4: The power consumption for 8 servers in one week.

Ganesan et al. [16], [17] leveraged genetic algorithms to automatically generate power viruses that consume more power than normal *stress* benchmarks. However, launching a power attack from scratch or being agnostic about the surrounding environment wastes unnecessary attacking resources.

In a real-world data center, the average utilization is around 20% to 30%, as reported by Barroso et al. [10]. With such low utilization, the chance of tripping the circuit breaker by indiscriminately launching power attacks is extremely low. However, although the average utilization is low, data centers still encounter power outage threats under peak demands [41]. This indicates that the power consumption of physical servers fluctuates enormously with the changing workloads. To confirm this assumption, we conduct an experiment to monitor the whole-system power consumption (via the RAPL leakage channel in case study II of Section 3) of eight physical servers in a container cloud for one week. We present the result in Figure 4. We first average the power data with a 30-second interval and observe drastic power changes on both Day 2 and Day 5. Furthermore, we pick a high power consumption region in Day 2 and average the data at the interval of one second (which is a typical time window for generating a power spike). The peak power consumption could reach 1,199 Watts (W). In total, there was a 34.72% (899W $\sim$ 1,199W) power difference in this one-week range. We anticipate that the power consumption difference would be even larger if we could monitor it for a longer time period, such as on a holiday like Black Friday, when online shopping websites hosted on a cloud may incur a huge power surge.

For a synergistic power attack in a container cloud, instead of indiscriminately starting a power-intensive workload, the adversaries can monitor the whole-system power consumption through the RAPL channel and learn the crests and troughs of the power consumption pattern at real time. Therefore, they can leverage the background power consumption (generated by benign workloads from other tenants on the same host) and superimpose their power attacks when the servers are at their peak running time. This is similar to the phenomenon of *insider trading* in the financial market—the one with more insider information can always trade at the right time. The adversaries can boost their power spikes, by adding on already-high power consumption, to a higher level with the "insider" power consumption information leaked through the RAPL channel.

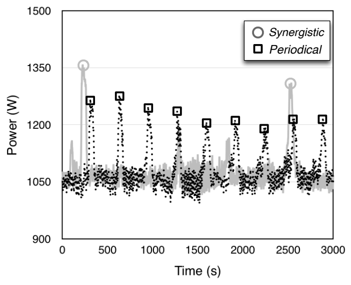


Fig. 5: Power Consumption of 8 Servers under Attack.

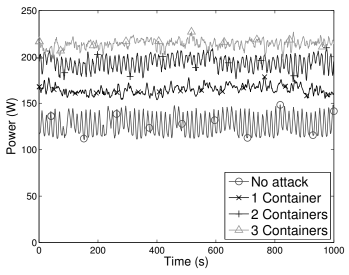


Fig. 6: Power Consumption of a Server under Attack.

5.2 Reduction of Attack Costs

From the attackers' perspective, they always intend to maximize attack outcomes with the lowest costs. Running power-intensive workloads continuously could definitely catch all the crests of benign power consumption. However, it may not be practical for real-world attacks for several reasons. First, it is not stealthy. To launch a power attack, the attacker needs to run power-intensive workloads. Such behavior has obvious patterns and could be easily detected by cloud providers. Second, utilization-based billing models are now becoming more popular. More cloud services provide finer-grained prices based on CPU/memory utilization and the volume of network traffic. For instance, *Elastic Container* provides containers with CPU metering-based billing for customers [3]. *IBM Cloud* provides billing metrics for computing resources in the cloud [4]. Amazon EC2 [1] offers *Burstable Performance Instances* that could occasionally burst but do not fully run most of the time. The VMware OnDemand Pricing Calculator [5] even gives an estimate for different utilization levels. For example, it charges \$2.87 per month for an instance with 16 VCPUs with an average of 1% utilization, and \$167.25 for the same server with full utilization. Under these cloud billing models, *continuous* power attacks may finally lead to an expensive bill.

For synergistic power attacks, monitoring power consumption through RAPL has almost zero CPU utilization. To achieve the same effects (the height of power spikes), synergistic power attacks can significantly reduce the attack costs compared to continuous and periodic attacks. In Figure 5, we compare the attack effects of a synergistic power attack with a periodic attack (launching power attacks every 300 seconds). Synergistic power attacks can achieve a 1,359W power spike with only two trials in 3,000 seconds, whereas periodic attacks were launched nine times and could only reach 1,280W at most.

5.3 Attack Orchestration

Different from traditional power attacks, another unique characteristic of synergistic power attack is its attack orchestration. Assume an attacker is already controlling a number of container instances. If these containers scatter in different locations within a data center, their power additions on multiple physical servers put no pressure on power facilities. Existing power-capping mechanisms can tolerate multiple small power surges from different locations with no difficulty. The only way to launch a practical power attack is to aggregate all "ammunition" into adjacent locations and attack a single power supply simultaneously. Here we discuss in depth on the orchestration of attacking container instances.

As we mentioned in Section 3, by exploiting multiple leakage channels[7], attackers can aggregate multiple container instances into one physical server. Specifically in our experiment on $\mathbf{CC_1}$, we choose to use *timer_list* to verify the co-residence of multiple containers. The detailed verification method is explained in Section 3.3. We repeatedly create container instances and terminate instances that are not on the same physical server. By doing this, we succeed in deploying three containers on the same server with trivial effort. We run four copies of *Prime* [7] benchmark within each container to fully utilize the four allocated cores. The results are illustrated in Figure 6. As we can see, each container can contribute approximately 40W power. With three containers, an attacker can easily raise the power consumption to almost 230W, which is about 100W more than the average power consumption for a single server.

We also find */proc/uptime* to be another interesting leakage channel. The *uptime* includes two data entries, the booting time of the physical server and the idle time of all cores. In our experiment, we find that some servers have similar booting times but different idle times. Typically servers in data centers do not reboot once being installed and turned on. A different idle time indicates that they are not the same physical server, while a similar booting time demonstrates that they have a high probability of being installed and turned on at the same time period. This is strong evidence that they might also be in close proximity and share the same circuit breaker. Attackers can exploit this channel to aggregate their attack container instances into adjacent physical servers. This greatly increases their chances of tripping circuit breakers to cause power outages.

6 Defense Approach

6.1 A Two-Stage Defense Mechanism

Intuitively, the solution should eliminate all the leakages so that no leaked information could be retrieved through those channels. We divide the defense mechanism into two stages to close the loopholes: (1) masking the channels and (2) enhancing the container's resource isolation model.

In the first stage, the system administrators can explicitly deny the read access to the channels within the container, e.g., through security policies in *AppArmor* or mounting the pseudo file "unreadable". This does not require any change to the kernel code (merging into the upstream Linux kernel might take some time) and can immediately eliminate information leakages. This solution depends on whether legitimate applications running inside the container use these channels. If such information is orthogonal to the containerized applications, masking it will not have a negative impact on the container tenants. We have reported our results to *Docker* and all the cloud vendors listed in Table 1, and we have received active responses. We are

7. Typically, if a channel is a strong co-residence indicator, leveraging this one channel only should be enough.

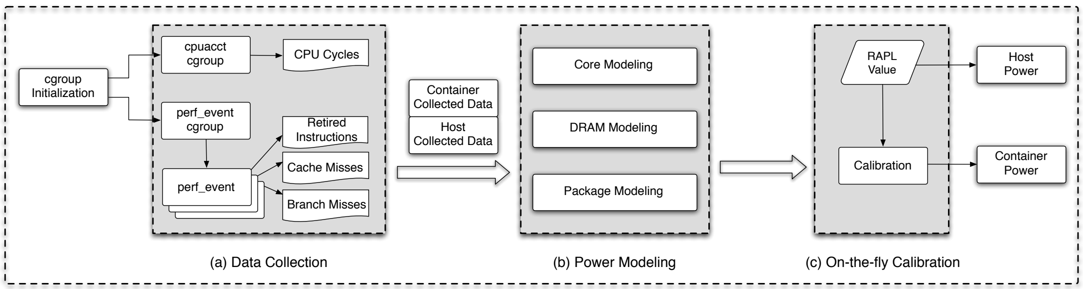


Fig. 7: The workflow of power-based namespace.

working together with container cloud vendors to fix this information leakage problem and minimize the impact upon applications hosted in containers. This masking approach is a quick fix to all leakages in memory-based pseudo file systems, but it may add restrictions for the functionality of containerized applications, which contradicts the container's concept of providing a generalized computation platform.

In the second stage, the defense approach involves fixing missing namespace context checks and virtualizing more system resources (i.e., the implementation of new namespaces) to enhance the container's isolation model. We first reported information disclosure bugs related to existing namespaces to Linux kernel maintainers, and they quickly released a new patch for one of the problems ([CVE-2017-5967]). For the other channels with no namespace isolation protection, we need to change the kernel code to enforce a finer-grained partition of system resources. Such an approach could involve non-trivial efforts since each channel needs to be fixed separately. Virtualizing a specific kernel component might affect multiple kernel subsystems. In addition, some system resources are not easy to be precisely partitioned to each container. However, we consider this to be a fundamental solution to the problem. In particular, to defend against synergistic power attacks, we design and implement a proof-of-concept power-based namespace in the Linux kernel to present the partitioned power usage to each container.

6.2 Power-based Namespace

We propose a power-based namespace to present per-container power usage through the unchanged RAPL interface to each container. Without leaking the system-wide power consumption information, attackers cannot infer the power state of the host, thus eliminating their chance of superimposing power-intensive workloads on benign power peaks. Moreover, with per-container power usage statistics at hand, we can dynamically throttle the computing power (or increase the usage fee) of containers that exceed their predefined power thresholds. It is possible for container cloud administrators to design a finer-grained billing model based on this power-based namespace.

There are three goals for our design. (1) Accuracy: as there is no hardware support for per-container power partitioning, our software-based power modeling needs to reflect the accurate power usage for each container. (2) Transparency: applications inside a container should be unaware of the power variations outside this namespace, and the interface of power subsystem should remain unchanged. (3) Efficiency: power partitioning should not incur non-trivial performance overhead in or out of containers.

We illustrate the workflow of our system in Figure 7. Our power-based namespace consists of three major components: *data collection*, *power modeling*, and *on-the-fly calibration*. We maintain the same Intel RAPL interface within containers, but change the implementation of handling read operations on energy usages. Once a read operation of energy usage is detected, the modified RAPL driver retrieves the per-container performance data (*data collection*), uses the retrieved data to model the energy usage (*power modeling*), and finally calibrates the modeled energy usage (*on-the-fly calibration*). We discuss each component in detail below.

6.2.1 Data collection

In order to model per-container power consumption, we need to obtain the fine-grained performance data for each container. Each container is associated with a *cpuacct* cgroup. A *cpuacct* cgroup accounts for the CPU cycles on a processor core for a container. The CPU cycles are accumulated. We only use CPU cycles to compute the rate of the cache miss rate and branch miss rate later. The Linux kernel also has a *perf_event* subsystem, which supports accounting for different types of performance events. The granularity of performance accounting could be a single process or a group of processes (considered as a *perf_event* cgroup). By now, we only retrieve the data for retired instructions, cache misses, and branch misses (which are needed in the following *power modeling* component) for each *perf_event* cgroup. Our current implementation is extensible to collect more performance event types corresponding to the changes of *power modeling* in the future.

We monitor the performance events from the initialization of a power-based namespace and create multiple *perf_events*, each associated with a specific performance event type and a specific CPU core. Then we connect the *perf_cgroup* of this container with these *perf_events* to start accounting. In addition, we need to set the owner of all created *perf_events* as *TASK_TOMBSTONE*, indicating that such performance accounting is decoupled from any user process.

6.2.2 Power modeling

To implement a power-based namespace, we need to attribute the power consumption to each container. Instead of providing transient power consumption, RAPL offers accumulated energy usages for *package*, *core*, and *DRAM*, respectively. The power consumption can be calculated by measuring the energy consumption over a time unit window.

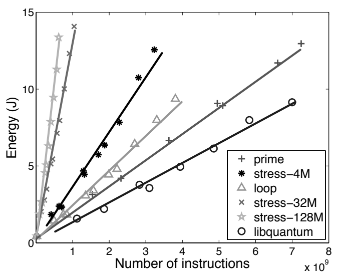


Fig. 8: Power modeling: the relation between core energy and the number of retired instructions.

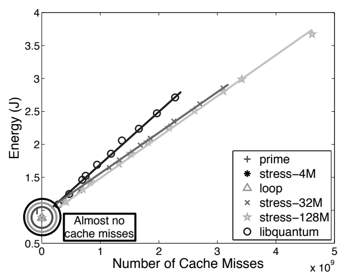


Fig. 9: Power modeling: the relation between DRAM energy and the number of cache misses.

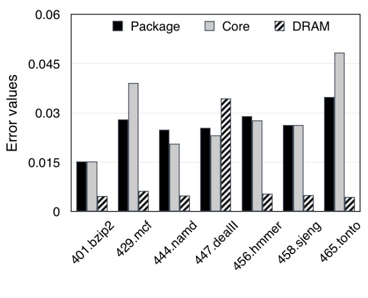


Fig. 10: The accuracy of our energy modeling approach to estimate the active power for the container from aggregate event usage and RAPL.

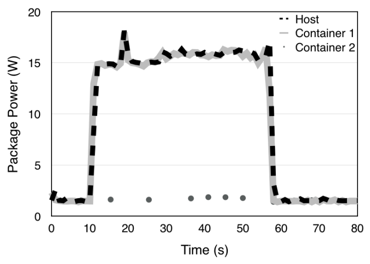


Fig. 11: Transparency: a malicious container (Container 2) is unaware of the power condition for the host.

Our power-based namespace also provides accumulative per-container energy data, in the same format as in the original RAPL interface.

We first attribute the power consumption for the *core*. Traditional power modeling leverages CPU utilization [33] to attribute the power consumption for cores. However, Xu et al. [47] demonstrated that the power consumption could vary significantly with the same CPU utilization. The underlying pipeline and data dependence could lead to CPU stalls and idling of function units. The actual numbers of retired instructions [26], [37] under the same CPU utilization are different. Figure 8 reveals the relation between retired instructions and energy. We test on four different benchmarks: the idle loop written in *C*, *prime*, *462.libquantum* in SPECCPU2006, and *stress* with different memory configurations. We run the benchmarks on a host and use *Perf* [6] to collect performance statistics data. We can see that for each benchmark, the energy consumption is almost strictly linear to the number of retired instructions. However, the gradients of fitted lines change correspondingly with application types. To make our model more accurate, we further include the cache miss rate [26] and branch miss rate to build a multi-degree polynomial model to fit the slope.

For the *DRAM*, we use the number of cache misses to profile the energy. Figure 9 presents the energy consumption for the same benchmarks with the same configurations in the *core* experiment. It clearly indicates that the number of cache misses is approximately linear to the *DRAM* energy. Based on this, we use the linear regression of cache misses to model the *DRAM* energy.

For the power consumption of *package*, we sum the values of *core*, *DRAM*, and an extra constant. The specific models are illustrated in Formula (2), where M represents the modeled energy; CM, BM, C indicate the number of cache misses, branch misses, and CPU cycles, respectively; and F is the function derived through multiple linear regressions to fit the slope. I is the number of retired instructions. $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$, and $\boldsymbol{\lambda}$ are the constants derived from the experiment data in Figures 8 and 9.

$$\begin{aligned} \mathrm{M_{core}} &= \mathrm{F}(\frac{\mathrm{CM}}{\mathrm{C}}, \frac{\mathrm{BM}}{\mathrm{C}}) \cdot \mathrm{I} + \boldsymbol{\alpha}, \\ \mathrm{M_{dram}} &= \boldsymbol{\beta} \cdot \mathrm{CM} + \boldsymbol{\gamma}, \\ \mathrm{M_{package}} &= \mathrm{M_{core}} + \mathrm{M_{dram}} + \boldsymbol{\lambda}. \end{aligned} \tag{2}$$

Here we discuss the influence of floating point instructions for power modeling. While an individual floating point instruction might consume more energy than an integer operation, workloads with high ratios of floating point instructions might actually result in lower power consumption overall, since the functional units might be forced to be idle in different stages of the pipeline. It is necessary to take the micro-architecture into consideration to build a more refined model. We plan to pursue this direction in our future work. Furthermore, the choices of parameters α, β, γ are also affected by the architecture. Such a problem could be mitigated in the following calibration step.

6.2.3 *On-the-fly calibration*

Our system also models the energy data for the host and cross-validates it with the actual energy data acquired through RAPL. To minimize the error of modeling data, we use the following Formula (3) to calibrate the modeled energy data for each container. The $\mathrm{E_{container}}$ represents the energy value returned to each container. This on-the-fly calibration is conducted for each read operation to the RAPL interface and can effectively reduce the number of errors in the previous step.

$$\mathrm{E_{container}} = \frac{\mathrm{M_{container}}}{\mathrm{M_{host}}} \cdot \mathrm{E_{RAPL}}. \tag{3}$$

7 Defense Evaluation

In this section, we evaluate our power-based namespace on a local machine in three aspects: accuracy, security, and performance. Our testbed is equipped with Intel i7-6700 3.40GHz CPU with 8 cores, 16GB of RAM, and running Ubuntu Linux 16.04 with kernel version 4.7.0.

7.1 Accuracy

We use the SPECCPU2006 benchmark to measure the accuracy of the power modeling. We compare the modeled power usage with the ground truth obtained through RAPL. The power consumption is equal to the energy consumption per second. Due to the restriction of the security policy of the *Docker* container, we select a subset of SPECCPU2006 benchmarks that are feasible to run inside the container and have no overlap with the benchmarks used for power modeling. The error ξ is defined as follows:

$$\xi = \frac{|(\mathrm{E_{RAPL}} - \Delta_{\mathrm{diff}}) - \mathrm{M_{container}}|}{\mathrm{E_{RAPL}} - \Delta_{\mathrm{diff}}}, \tag{4}$$

TABLE 5: PERFORMANCE RESULTS OF UNIX BENCHMARKS.

	1 Parallel Copy			8 Parallel Copies		
Benchmarks	**Original**	**Modified**	**Overhead**	**Original**	**Modified**	**Overhead**
Dhrystone 2 using register variables	3,788.9	3,759.2	0.78%	19,132.9	19,149.2	0.08%
Double-Precision Whetstone	926.8	918.0	0.94%	6,630.7	6,620.6	0.15%
Execl Throughput	290.9	271.9	6.53%	7,975.2	7,298.1	8.49%
File Copy 1024 bufsize 2000 maxblocks	3,495.1	3,469.3	0.73%	3,104.9	2,659.7	14.33%
File Copy 256 bufsize 500 maxblocks	2,208.5	2,175.1	0.04%	1,982.9	1,622.2	18.19%
File Copy 4096 bufsize 8000 maxblocks	5,695.1	5,829.9	-2.34%	6,641.3	5,822.7	12.32%
Pipe Throughput	1,899.4	1,878.4	1.1%	9,507.2	9,491.1	0.16%
Pipe-based Context Switching	653.0	251.2	61.53%	5,266.7	5,180.7	1.63%
Process Creation	1416.5	1289.7	8.95%	6618.5	6063.8	8.38%
Shell Scripts (1 concurrent)	3,660.4	3,548.0	3.07%	16,909.7	16,404.2	2.98%
Shell Scripts (8 concurrent)	11,621.0	11,249.1	3.2%	15,721.1	15,589.2	0.83%
System Call Overhead	1,226.6	1,212.2	1.17%	5,689.4	5,648.1	0.72%
System Benchmarks Index Score	2,000.8	1,807.4	9.66%	7,239.8	6,813.5	7.03%

where $\mathrm{E}_{\mathrm{RAPL}}$ is the power usage read from RAPL on the host, and $\mathrm{M}_{\mathrm{container}}$ is the modeled power usage for the same workload read within the container. Note that both the host and container consume power at an idle state with trivial differences. We use a constant Δ_{diff} as the modifier reflecting the difference in power consumption at an idle state for a host and a container. The results, illustrated in Figure 10, show that our power modeling is accurate as the error values of all the tested benchmarks are lower than 0.05.

7.2 Security

We also evaluate our system from the security perspective. With the power-based namespace enabled, the container should only retrieve the power consumed within the container and be unaware of the host's power status. We launch two containers in our testbed for comparison. We run the SPECCPU2006 benchmark in one container and leave the other one idle. We record the power usage per second of both containers and the host. We show the results of *401.bzip2* in Figure 11. All other benchmarks exhibit similar patterns.

When both containers have no workload, their power consumption is at the same level as that of the host, e.g., from 0s to 10s. Once container 1 starts a workload at 10s, we can find that the power consumption of container 1 and the host surges simultaneously. From 10s to 60s, container 1 and the host have a similar power usage pattern, whereas container 2 is still at a low power consumption level. Container 2 is unaware of the power fluctuation on the whole system because of the isolation enforced by the power-based namespace. This indicates that our system is effective for isolating and partitioning power consumption for multiple containers. Without power-related information, attackers will not be able to launch synergistic power attacks.

7.3 Performance

We use UnixBench to compare the performance overhead before and after enabling our system. Table 5 lists all results.

As the results show, CPU benchmarks such as *Dhrystone* (testing integer and string operations) and *Whetstone* (testing float point arithmetic performance) incur negligible overhead. Other benchmarks like *shell scripts*, *pipe throughput*, and *system call* also trigger little overhead.

The *pipe-based context switching* does incur a 61.53% overhead in the case of one parallel copy, but it decreases to 1.63% for 8 parallel copies. We anticipate that inter-cgroup context switching involves enabling/disabling the performance event monitor, whereas intra-cgroup context switching does not involve any such overhead. This could explain why 8 parallel copies can maintain a similar performance level with the power-based namespace disabled. In addition, context switching only contributes to a very small portion of the whole-system performance overhead, so there is trivial impact for the normal use.

In our system, for each newly created container, the kernel will create multiple perf_events on each core to collect the performance related data. This measurement process is only conducted for container processes. Thus, the measurement overhead will be increased linearly with the number of containers. However, the increasing number of containers will have little impact upon the system. With this mechanism, for creating all the processes, the kernel will check whether this process is a container process or not. This checking process is the main cause of the overhead of process creation in Unix benchmarks.

As demonstrated in the last row of Table 5, the overall performance overheads for the UnixBench are 9.66% for one parallel copy and 7.03% for 8 parallel copies, respectively. Our system's performance depends heavily on the implementation of *perf_event* cgroup and could improve with the advancement of a performance monitoring subsystem.

8 Discussion

8.1 Synergistic Power Attacks without the RAPL Channel

We also notice that servers in some container clouds are not equipped with RAPL or other similar embedded power meters. Those servers might still face power attacks. Without power-capping tools like RAPL, those servers might be vulnerable to host-level power attacks on a single machine. In addition, if power data is not directly available, advanced attackers will try to approximate the power status based on the resource utilization information, such as the CPU and memory utilization, which is still available in the identified information leakages. It would be better to make system-wide performance statistics unavailable to container tenants.

8.2 Complete Container Implementation

The root cause for information leakage and synergistic power attack is the incomplete implementation of the isolation mechanisms in the Linux kernel. It would be better to introduce more security features, such as implementing more namespaces and control groups. However, some system resources are still difficult to be partitioned, e.g., interrupts, scheduling information, and temperature. People also argue that the complete container implementation is no different from a virtual machine, and loses all the container's advantages. It is a trade-off for containers to deal with. The question of how to balance security, performance, and usability in container clouds needs further investigation.

9 Related Work

9.1 Performance and Security Research on Containers

Since containers have recently become popular, researchers are curious about the performance comparison between containers and hardware virtualization. Felter et al. compared the performance of *Docker* and *KVM* by using a set of benchmarks covering CPU, memory, storage, and networking resources [15]. Their results show that *Docker* can achieve equal or better performance than *KVM* in all cases. Spoiala et al. [38] used the *Kurento Media Server* to compare the performance of *WebRTC* servers on both *Docker* and *KVM*. They also demonstrated that *Docker* outperforms *KVM* and could support real-time applications. Morabito et al. [34] compared the performance between traditional hypervisor and OS-level virtualization with respect to computing, storage, memory, and networks. They conducted experiments on *Docker*, *LXC*, and *KVM* 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 can achieve a higher performance than hardware virtualization. Besides performance, the security of a container cloud is always an important research area. Gupta [22] gave a brief overview of *Docker* security. Bui [12] also performed an analysis on *Docker* containers, including the isolation problem and corresponding kernel security mechanisms. They claimed that *Docker* containers are fairly secure with default configurations. Grattafiori et al. [19] summarized a variety of potential vulnerabilities of containers. They also mentioned several channels in the memory-based pseudo file systems. Luo et al. [30] demonstrated that mis-configured capabilities could be exploited to build covert channels in Docker. Previous research efforts on the performance and security of containers encourage us to investigate more on how containers can achieve the same security guarantees as hardware virtualization, but with trivial performance trade-offs. We are among the first to systematically identify the information leakage problem in containers and investigate potential container-based power attack threats built upon these leakage channels.

9.2 Cloud Security and Side/Covert Channel Attacks

Cloud security has received much attention from both academia [18] and industry. Co-residence detection in the cloud settings is the most closely related research topic to our work. Co-residence detection was first proposed by Ristenpart et al. [35]. They demonstrated that an attacker can place a malicious VM co-resident with a target VM on the same sever and then launch side-channel and covert-channel attacks. Two previous works [40], [46] show that it is still practical to achieve co-residence in existing mainstream cloud services. To verify co-residence on the same physical server, attackers typically leverage side channels or covert channels, e.g., one widely adopted approach is to use cache-based covert channels [24], [39], [45]. Multiple instances locating on the same package share the last-level caches. By using some dedicated operations, such as *cflush* [48], attackers can detect co-residence by measuring the time of cache accessing. Liu et al. [29] demonstrated that l3 cache side-channel attacks are practical for cross-core and cross-VM attacks. Zhang et al. conducted real side-channel attacks on the cloud [50], [51] and proposed several defense mechanisms to mitigate those attacks [44], [49], [52]. In particular, they demonstrated that cross-tenant side-channel attacks can be successfully conducted in PaaS with co-resident servers [51]. Besides the cache-based channel, memory bus [42] and memory deduplication [43] have also proved to be effective for covert-channel construction. Different from existing research efforts on side/covert channels, we discover a system-wide information leakage in the container cloud settings and design a new methodology to quantitatively assess the capacity of leakage channels for co-residence detection. In addition, compared to the research on minimizing the kernel attack surface for VMs [20], we proposed a two-stage defense mechanism to minimize the space for information leakages and power attacks on container clouds.

System status information, such as core temperature and system power consumption, have also been used to build side/covert channels. Thiele et al. [11], [31] proposed a thermal covert channel based on the temperature of each core and tested the capacity in a local testbed. Power consumption could also be abused to break AES [25]. In our work, we do not use the power consumption data as a covert channel to transfer information. Instead, we demonstrate that adversaries may leverage the host power consumption leakage to launch more advanced power attacks.

9.3 Power Modeling

When hardware-based power meter is absent, power modeling is the approach to approximating power consumption. Russell et al. [36] and Chakrabarti et al. [13] proposed instruction-level power modeling. Their works indicate that the number of branches affects power consumption. There are several works of approximating power consumption for VMs. Both works [23], [26] demonstrate that VM-level power consumption can be estimated by CPU utilization and last-level cache miss. Mobius et al. [33] broke the power consumption of VM into CPU, cache, memory, and disk. BITWATTS [14] modeled the power consumption at a finer-grained process level. Shen et al. [37] proposed a power container to account for energy consumption of requests in multi-core systems. Our defense against the synergistic power attack is mainly inspired by the power modeling approach for VMs. We propose a new power partitioning technique to approximate the per-container power consumption and reuse the RAPL interface, thus addressing the RAPL data leakage problem in the container settings.

10 Conclusion

Container cloud services have become popular for providing lightweight OS-level virtual hosting environments. The emergence of container technology profoundly changes the eco-system of developing and deploying distributed applications in the cloud. However, due to the incomplete implementation of system resource partitioning mechanisms in the Linux kernel, there still exist some security concerns for multiple container tenants sharing the same kernel. In this paper, we first present a systematic approach to discovering information leakage channels that may expose host information to containers. By exploiting such leaked host information, malicious container tenants can launch a new type of power attack that may potentially jeopardize the dependability of power systems in the data centers. Additionally, we discuss the root causes of these information leakages and propose a two-stage defense mechanism. Our evaluation demonstrates that the proposed solution is effective and incurs trivial performance overhead.

References

[1] Burstable Performance Instances. https://aws.amazon.com/ec2/instance-types/#burst.

[2] Containers Not Virtual Machines Are the Future Cloud. http://www.linuxjournal.com/content/containers.

[3] ElasticHosts: Linux container virtualisation allows us to beat AWS on price. http://www.computerworlduk.com/news/it-leadership/elastichosts-linux-container-virtualisation-allows-us-beat-aws-on-price-3510973/.

[4] IBM Cloud metering and billing. https://www.ibm.com/developerworks/c-loud/library/cl-cloudmetering/.

[5] OnDemand Pricing Calculator. http://vcloud.vmware.com/service-offering/pricing-calculator/on-demand.

[6] Perf Wiki. https://perf.wiki.kernel.org/index.php/Main_Page.

[7] Prime95 Version 28.9. http://www.mersenne.org/download/ .

[8] Travel nightmare for fliers after power outage grounds Delta. http://money.cnn.com/2016/08/08/news/companies/delta-system-outage-flights/.

[9] A. F. Atya, Z. Qian, S. Krishnamurthy, T. La Porta, P. McDaniel, and L. Marvel. Malicious co-residency on the cloud: Attacks and defense. In *IEEE INFOCOM*, 2017.

[10] L. A. Barroso and U. Hölzle. The Case for Energy-Proportional Computing. *Computer*, 2007.

[11] D. B. Bartolini, P. Miedl, and L. Thiele. On the Capacity of Thermal Covert Channels in Multicores. In *ACM EuroSys*, 2016.

[12] T. Bui. Analysis of Docker Security. *arXiv preprint arXiv:1501.02967*, 2015.

[13] C. Chakrabarti and D. Gaitonde. Instruction Level Power Model of Microcontrollers. In *IEEE ISCAS*, 1999.

[14] M. Colmant, M. Kurpicz, P. Felber, L. Huertas, R. Rouvoy, and A. Sobe. Process-level Power Estimation in VM-based Systems. In *ACM EuroSys*, 2015.

[15] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An Updated Performance Comparison of Virtual Machines and Linux Containers. In *IEEE ISPASS*, 2015.

[16] K. Ganesan, J. Jo, W. L. Bircher, D. Kaseridis, Z. Yu, and L. K. John. System-level Max Power (SYMPO) - A Systematic Approach for Escalating System-level Power Consumption using Synthetic Benchmarks. In *ACM PACT*, 2010.

[17] K. Ganesan and L. K. John. MAximum Multicore POwer (MAMPO) - An Automatic Multithreaded Synthetic Power Virus Generation Framework for Multicore Systems. In *ACM SC*, 2011.

[18] X. Gao, Z. Xu, H. Wang, L. Li, and X. Wang. Reduced cooling redundancy: A new security vulnerability in a hot data center.

[19] A. Grattafiori. NCC Group Whitepaper: Understanding and Hardening Linux Containers, 2016.

[20] Z. Gu, B. Saltaformaggio, X. Zhang, and D. Xu. FACE-CHANGE: Application-Driven Dynamic Kernel View Switching in a Virtual Machine. In *IEEE/IFIP DSN*, 2014.

[21] P. Guide. Intel® 64 and IA-32 Architectures Software Developers Manual, 2011.

[22] U. Gupta. Comparison between security majors in virtual machine and linux containers. *arXiv preprint arXiv:1507.07816*, 2015.

[23] Z. Jiang, C. Lu, Y. Cai, Z. Jiang, and C. Ma. VPower: Metering Power Consumption of VM. In *IEEE ICSESS*, 2013.

[24] M. Kayaalp, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel. A High-Resolution Side-Channel Attack on Last-Level Cache. In *IEEE DAC*, 2016.

[25] P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *Annual International Cryptology Conference*, 1999.

[26] B. Krishnan, H. Amur, A. Gavrilovska, and K. Schwan. VM Power Metering: Feasibility and Challenges. *ACM SIGMETRICS Performance Evaluation Review*, 2011.

[27] C. Lefurgy, X. Wang, and M. Ware. Power Capping: A Prelude to Power Shifting. *Cluster Computing*, 2008.

[28] C. Li, Z. Wang, X. Hou, H. Chen, X. Liang, and M. Guo. Power Attack Defense: Securing Battery-Backed Data Centers. In *IEEE ISCA*, 2016.

[29] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE S&P*, 2015.

[30] Y. Luo, W. Luo, X. Sun, Q. Shen, A. Ruan, and Z. Wu. Whispers between the Containers: High-Capacity Covert Channel Attacks in Docker. In *IEEE Trustcom/BigDataSE/ISPA*, 2016.

[31] R. J. Masti, D. Rai, A. Ranganathan, C. Müller, L. Thiele, and S. Capkun. Thermal Covert Channels on Multi-core Platforms. In *USENIX Security*, 2015.

[32] C. Maurice, C. Neumann, O. Heen, and A. Francillon. C5: cross-cores cache covert channel. In *Springer DIMVA*, 2015.

[33] C. Mobius, W. Dargie, and A. Schill. Power Consumption Estimation Models for Processors, Virtual Machines, and Servers. *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[34] R. Morabito, J. Kjällman, and M. Komu. Hypervisors vs. Lightweight Virtualization: A Performance Comparison. In *IEEE IC2E*, 2015.

[35] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *ACM CCS*, 2009.

[36] J. T. Russell and M. F. Jacome. Software Power Estimation and Optimization for High Performance, 32-bit Embedded Processors. In *IEEE ICCD*, 1998.

[37] K. Shen, A. Shriraman, S. Dwarkadas, X. Zhang, and Z. Chen. Power Containers: An OS Facility for Fine-Grained Power and Energy Management on Multicore Servers. *ACM ASPLOS*, 2013.

[38] C. C. Spoiala, A. Calinciuc, C. O. Turcu, and C. Filote. Performance comparison of a WebRTC server on Docker versus Virtual Machine. In *IEEE DAS*, 2016.

[39] E. Tromer, D. A. Osvik, and A. Shamir. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology*, 2010.

[40] V. Varadarajan, Y. Zhang, T. Ristenpart, and M. Swift. A Placement Vulnerability Study in Multi-Tenant Public Clouds. In *USENIX Security*, 2015.

[41] Q. Wu, Q. Deng, L. Ganesh, C.-H. Hsu, Y. Jin, S. Kumar, B. Li, J. Meza, and Y. J. Song. Dynamo: Facebooks Data Center-Wide Power Management System. *IEEE ISCA*, 2016.

[42] Z. Wu, Z. Xu, and H. Wang. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *USENIX Security*, 2012.

[43] J. Xiao, Z. Xu, H. Huang, and H. Wang. Security Implications of Memory Deduplication in a Virtualized Environment. In *IEEE/IFIP DSN*, 2013.

[44] Q. Xiao, M. K. Reiter, and Y. Zhang. Mitigating Storage Side Channels Using Statistical Privacy Mechanisms. In *ACM CCS*, 2015.

[45] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting. An Exploration of L2 Cache Covert Channels in Virtualized Environments. In *ACM CCSW*, 2011.

[46] Z. Xu, H. Wang, and Z. Wu. A Measurement Study on Co-residence Threat inside the Cloud. In *USENIX Security*, 2015.

[47] Z. Xu, H. Wang, Z. Xu, and X. Wang. Power Attack: An Increasing Threat to Data Centers. In *NDSS*, 2014.

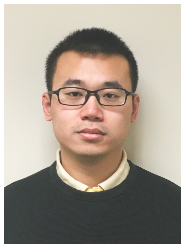
[48] Y. Yarom and K. Falkner. FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*, 2014.

[49] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *IEEE S&P*, 2011.

[50] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *ACM CCS*, 2012.

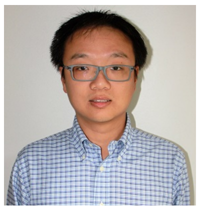
[51] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *ACM CCS*, 2014.

[52] Y. Zhang and M. K. Reiter. Düppel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud. In *ACM CCS*, 2013.



Xing Gao received the Ph.D. degree in computer science from the College of William and Mary, Williamsburg, VA, USA, in 2018. He is an Assistant Professor in the Department of Computer Science at the University of Memphis, Memphis, TN, USA. His research interests include security, cloud computing, and mobile computing.

Benjamin Steenkamer will receive his Bachelor of Computer Engineering degree from the University of Delaware in the Spring of 2018. His research interests are security, hardware design, and high performance computing. He is also pursuing a Master's of Electrical and Computer Engineering degree at the University of Delaware.



Zhongshu Gu is a Research Staff Member in the Security Research Department of the IBM T.J. Watson Research Center. He received his Ph.D. from Purdue University in 2015 and B.S. from Fudan University in 2007, both in Computer Science. His research interests are in the areas of systems security, AI security, security analytics, and cyber forensics.

Mehmet Kayaalp received his B.S. and M.S. degrees in Computer Engineering from TOBB University of Economics and Technology in 2008 and 2010 respectively, and his Ph.D. degree in Computer Science from Binghamton University, State University of New York, in 2015. He joined the systems security group at IBM Research as a postdoctoral researcher, in 2015. He is now an assistant professor at the Department of Electrical and Computer Engineering at University of New Hampshire. His research interests include systems security, hardware design, side channels, code reuse attacks.

Dimitrios Pendarakis is a Principal Research Staff Member and Senior Manager of the Secure Cloud and Systems Group at the IBM T.J. Watson Research Center. His current research interests are in the areas of cloud computing security, hardware-enabled security, IoT and security analytics. In his position, Dimitrios directs several research projects that are developing new technologies targeted for commercialization in IBM products and services. Prior to this position, Dimitrios was a Research Staff Member in the Distributed Systems Group in Watson, where he led projects in the areas of on-demand computing, security for cyber-physical systems, and distributed system resilience. Dimitrios received the Diploma degree from the National Technical University of Athens, Greece and the M.S. and Ph.D. degrees in Electrical Engineering from Columbia University, NY, NY.

Haining Wang received the Ph.D. degree in computer science and engineering from the University of Michigan, Ann Arbor, MI, USA, in 2003. He is a Professor of electrical and computer engineering with the University of Delaware, Newark, DE, USA. His research interests lie in the areas of security, networking system, and cloud computing.