Fast and Scalable VMM Live Upgrade in Large Cloud Infrastructure

Xiantao Zhang
Alibaba Group
xiantao.zxt@alibaba-inc.com

Xiao Zheng
Alibaba Group
zhengxiao.zx@alibaba-inc.com

Zhi Wang
Florida State University
zwang@cs.fsu.edu

Qi Li
Tsinghua University
qi.li@sz.tsinghua.edu.cn

Junkang Fu
Alibaba Group
junkang.fjk@alibaba-inc.com

Yang Zhang
Alibaba Group
zy107165@alibaba-inc.com

Yibin Shen
Alibaba Group
zituan@taobao.com

Abstract

High availability is the most important and challenging problem for cloud providers. However, virtual machine monitor (VMM), a crucial component of the cloud infrastructure, has to be frequently updated and restarted to add security patches and new features, undermining high availability. There are two existing live update methods to improve the cloud availability: kernel live patching and Virtual Machine (VM) live migration. However, they both have serious drawbacks that impair their usefulness in the large cloud infrastructure: kernel live patching cannot handle complex changes (e.g., changes to persistent data structures); and VM live migration may incur unacceptably long delays when migrating millions of VMs in the whole cloud, for example, to deploy urgent security patches.

In this paper, we propose a new method, VMM live upgrade, that can promptly upgrade the whole VMM (KVM & QEMU) without interrupting customer VMs. Timely upgrade of the VMM is essential to the cloud because it is both the main attack surface of malicious VMs and the component to integrate new features. We have built a VMM live upgrade system called Orthus. Orthus features three key techniques: dual KVM, VM grafting, and device handover. Together, they enable the cloud provider to load an upgraded KVM instance while the original one is running and "cut-and-paste" the VM to this new instance. In addition, Orthus can seamlessly hand over passthrough devices to the new KVM instance without losing any ongoing (DMA) operations. Our evaluation shows that Orthus can reduce the total migration time and downtime by more than 99% and 90%, respectively. We have deployed Orthus in one of the largest cloud infrastructures for a long time. It has become the most effective and indispensable tool in our daily maintenance of hundreds of thousands of servers and millions of VMs.

CCS Concepts • **Security and privacy → Virtualization and security**; • **Computer systems organization** → *Availability*.

Keywords virtualization; live upgrade; cloud infrastructure

ACM Reference Format:
Xiantao Zhang, Xiao Zheng, Zhi Wang, Qi Li, Junkang Fu, Yang Zhang, and Yibin Shen. 2019. Fast and Scalable VMM Live Upgrade in Large Cloud Infrastructure. In *ASPLOS '19: ACM International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/1122445.1122456

1 Introduction

High availability is the most important but challenging problem for cloud providers. It requires the service to be accessible anywhere and anytime without perceivable downtime. However, there are two important issues that can undermine high availability: new features and security. Specifically, a cloud usually provides a range of services, such as the storage, database, and program language services, in addition to running customer VMs. Features like the cloud storage need to be integrated into the core cloud component, the VMM. New features will only take effect after the VMM is restarted, thus interrupting the service. Meanwhile, security demands the system software to be frequently updated to patch newly discovered vulnerabilities. All the major Linux distributions push security updates weekly, if not daily. Cloud providers have to keep their systems patched otherwise risk being compromised. Even though there are efforts to proactively

ASPLOS '19, April 13–17, 2019, Providence, RI

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

Methods	Application	Advantages	Limitations
Kernel live patching	Simple security fixes	No downtime	Cannot handle complex fixes Maintenance headache
VMM live upgrade	New cloud features Complex fixes in the VMM	Short downtime Scalable, no migration traffic Passthrough device support	VMM-only update (justified)
VM live migration	Whole system upgrade	Update everything on the server	Not scalable, needs spare servers Migration traffic consumes network bandwidth Performance downgrade and long downtime Cannot support passthrough devices

Table 1. Comparison of three live update methods

discover and fix vulnerabilities in the VMM, vulnerabilities are inevitable given the complexity of the VMM. For example, KVM, the VMM used by many major public cloud providers, is a type-2 VMM running in the host Linux kernel, which has more than 16 millions lines of source code. Therefore, it is critical to keep the VMM updated to minimize the window of vulnerability. In summary, both new features and security demand a method to quickly update the cloud system with minimal downtime.

Limitations of the existing methods: currently, there are two live update methods available to cloud providers: kernel live patching and VM live migration. They are valuable in improving the cloud availability but have serious drawbacks: kernel live patching applies patches to the running kernel without rebooting the system [7, 27, 33]. However, live patching cannot handle complex updates that, for example, change the persistent data structures or add new features. It is most suitable for simple security fixes, such as checking buffer sizes. We instead aim at creating an update mechanism that can not only support simple security checks but also add new features and apply complex security patches (e.g., the Retpoline fix for Spectre that requires recompiling the code [39]).

VM live migration is another valuable tool used by cloud providers to facilitate software update [10, 12]. It iteratively copies the VM states (e.g., memory) from its current server to an idle backup server while the VM is still running. At the final iteration, it completely stops the VM, copies the last changed states, and restarts the VM on the backup server. This allows the cloud provider to update/replace everything on the source server, including the host kernel, the VMM, failed hardware devices, etc. The VM can be migrated back to the source server after the update if necessary. Nevertheless, VM live migration has three limitations that severely limit its usefulness in large cloud datacenters:

- First, it requires spare servers to temporarily hold the migrating VMs. These servers must be equally powerful in order not to degrade the VM performance. If a large number of VMs have to be migrated simultaneously, it brings significant pressure to the inventory control for spared servers. In large clouds like ours, it is common for hundreds of thousands of or even millions of VMs to be migrated, for example, to fix urge security vulnerabilities. In addition, large-scale VM migration can cause severe network congestion.
- Second, VM live migration will incur temporary performance downgrade and service downtime [40]. The more memory a VM has, the longer the migration takes. High-end cloud customers may use hundreds of GBs of memory per VM to run big data, massive streaming, and other applications. These applications are particularly sensitive to performance downgrade and downtime. Such customers often closely monitor the real-time system performance. VM live migration is simply not an option for those VMs.
- Last, VM live migration cannot support passthrough devices. Device passthrough grants a VM direct access to the hardware devices. Passthrough devices are widely deployed in the cloud. For example, network adapter passthrough is often used to speed up network traffic, and many heterogeneous computing services rely on GPU or FPGA passthrough. Existing VM live migration systems do not support device passthrough.

VMM live upgrade, a new method: in this paper, we propose a third method complimentary to, but more useful than, kernel live update and VM live migration – VMM live upgrade, a method to *replace* the live VMM. Unlike kernel live patch, VMM live upgrade replaces the whole running VMM. Consequently, it can apply more complex fixes to the VMM, for example, to support a new cloud storage system. Because VMM live upgrade is conducted on the same physical server, it does not require spare servers to temporarily hold VMs or introduce significant network traffic; its performance and downtime are significantly better than VM live migration; and it can seamlessly support passthrough devices. Table 1 compares these three methods.

VMM live upgrade replaces the VMM only. This is consistent with the threat model in the cloud, where the host kernel is well isolated from the untrusted VMs through network segregation, device passthrough, and driver domains; and the main attack surface lies in the interaction between the VMM and VMs. This is reflected in Table 2, which lists all the known vulnerabilities related to KVM/QEMU [3, 34]. These vulnerabilities are exposed to the VMs and pose the most imminent threat to the cloud infrastructure.

	x86				Other			
Core	Emu	VMX	PIC	Other	Arch	Kernel	QEMU	Total
17	13	14	7	22	6	2	14	95

Table 2. Distribution of reported vulnerabilities in KVM

Out of these 95 vulnerabilities, only two of them (2.1%) lie in the main kernel; fourteen lie in QEMU, the user-space part of the VMM; and the rest 79 lie in KVM, the kernel part of the VMM (column 1 to 6). The distribution of those vulnerabilities highlights the troublesome components in the VMM: KVM has a layered structure with a common core and the architecture-specific layers. The core has 17 reported vulnerabilities (column 1) and the x86 layer alone has 56 vulnerabilities (column 2 to 5). These vulnerabilities include information leaks, race conditions, buffer overflows, etc. In addition, most of the fourteen vulnerabilities in QEMU appear in the device emulation. Many of these vulnerabilities are not fixable by kernel live patching: we manually examined all the eight KVM-related vulnerabilities reported in 2018 and found that only three of them can be supported by kernel live patching. The rest five all incur changes to persistent data structures and thus cannot be supported by kernel live patching. Nevertheless, VMM live upgrade can easily fix all the vulnerabilities in both KVM and QEMU, thus eliminating most of the threats to the cloud infrastructure.

Our focus on the VMM is also justified by the fact that new cloud features are integrated mostly through the VMM; the host kernel is rarely involved. For example, new GPU/FPGA devices are often passed through to VMs directly. The host kernel does not need to drive these devices.

Our system: we have designed, built, and deployed a VMM live upgrade system called Orthus.[1] Orthus has three key techniques: *dual KVM*, *VM grafting*, and *device handover*. Dual KVM runs two instances of the KVM kernel module side-by-side. One is currently running the VMs and the other is the upgraded version. Instead of migrating the VM between these two instances, Orthus directly "grafts" the VM from the original VMM to the upgraded VMM, avoiding the lengthy memory copies and long downtime in the VM live migration. Lastly, device handover seamlessly transfers the ownership of passthrough devices to the new VMM without losing any ongoing (DMA) operations. With these three techniques, Orthus can quickly upgrade the VMM to apply critical security patches and integrate new features.

Even though Orthus is designed for KVM, we believe a similar approach can be applied to type-1 hypervisors like Xen because Xen uses the same hardware features as KVM (i.e., hardware virtualization support) and relies on a privileged domain for device drivers. Though, the implementation could be more complex.

Orthus has been deployed in the Alibaba Cloud, one of the largest public cloud infrastructures, serving thousands of clusters (each cluster consists of hundreds of servers with the same hardware configurations. Each physical server can serve up to 100 VMs). It has become the most indispensable system in our daily operation. We rely on it to deploy security patches and new features. The other two live update methods are used much less frequently. Particularly, we use VM live migration only when there is a hard requirement to replace the node, for example, due to hardware failure. Orthus is very scalable as it can be applied to from a single VM to hundreds of VMs in a physical server simultaneously. In summary, this paper makes the following contribution:

- We propose a new method called VMM live upgrade that can live-update the whole VMM (KVM/QEMU) without disrupting the running VMs. VMM live upgrade is complementary to, but more useful than, the existing live update methods.
- We have built Orthus, a VMM live upgrade system, that can effectively "cut-and-paste" a VM from its running VMM to the upgraded VMM. In addition, Orthus can seamlessly transit passthrough devices to the new VMM without involving any manual efforts or losing ongoing operations.
- We have deployed Orthus in one of the largest public cloud infrastructures and demonstrated its effectiveness in promptly deploying urgent security patches (e.g., Spectre) and new features.

The rest of the paper is organized as follows. We first further motivate the problem by measuring the negative impact of VM live migration and discussing its limitations in Section 2. We then present the design and evaluation of Orthus in Section 3 and 4, respectively. Section 5 compares Orthus to the related work, and Section 6 discusses the potential improvements to Orthus. We conclude the paper in Section 7.

2 Background and Motivation

In this section, we describe how kernel live patching and VM live migration are used in cloud datacenters, in particular, their limitations that raise the need for VMM live upgrade.

High availability is the most important goal of public clouds. Many decisions are made to improve availability. In particular, server hardware replacement and major kernel upgrades are rather rare; security patches and feature upgrades to the VMM are far more frequent in order to improve security and enable new features. Most of these updates cannot be applied by kernel live patching because they often involve complex changes in multiple files and changes to persistent data structures. Frequent applications of kernel live patching can also make the kernel/VMM difficult to maintain. On the other hand, VM live migration is too costly for regular VMM updates, especially when a large number of VMMs need to be updated, for example, to apply an urgent security patch.

[1] Orthus is a two-headed dog guarding the cattle in the Greek mythology.

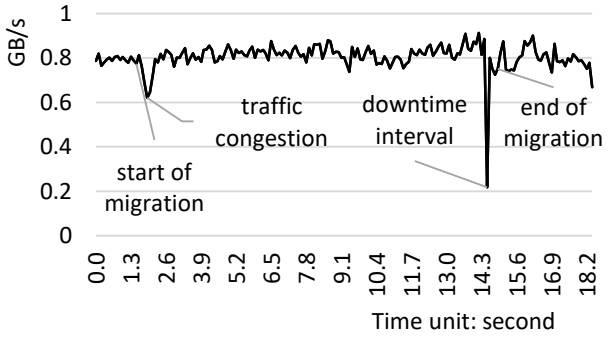

Figure 1. Impact on the VM network traffic during live migration of a 16GB VM over 10GB dedicated network

2.1 Large-scale VM Migration is Impractical

Concurrently migrating lots of VMs is impractical due to the limited availability of spare servers and network bandwidth. A cluster in the datacenter consists of hundreds of servers with the same hardware specification. The cloud operator often reserves only a small number of spare servers in each cluster for urgent/peak business needs. This significantly limits the number of VMs that can be simultaneously migrated. To upgrade the whole cluster, we need to use these spare servers back and forth. Even worse, VM migration can only happen when the datacenter is relatively quiet (e.g., the midnight of the majority of cloud users) to minimize the impact on the customer network traffic. Consequently, it is not uncommon to take an entire month to completely update a cluster. This poses a serious threat to the security because the lag between the release of a security patch to the first appearance of exploits targeting the patched vulnerabilities is only days.

The design of Orthus allows us to concurrently upgrade as many VMMs as possible because Orthus does not need spare machines to temporarily hold VMs. It performs the upgrade within the same machine with a small memory footprint.

2.2 Impact of VM Live Migration

VM live migration has three stages: pre-copy, stop-and-copy, and post-copy. In the pre-copy stage, the VM's memory is iteratively copied to a backup server while the VM is still running. Specifically, KVM keeps track of the VM's dirty memory (i.e., the memory changed since the last iteration of copy) and copies that dirty memory in the next iteration. In the stop-and-copy stage, the VM is paused and the leftover dirty memory and other VM states are copied over to the backup server. In the post-copy stage, the VM is resumed on the backup server. Dirty memory tracking and copying downgrade the performance of the VM, while stop-and-copy incurs the downtime.

The total migration time is decided by the memory size of the VM and the available network bandwidth for migration. Our measurement shows that it could take as long as 12 seconds in total to migrate a VM with 16GB of memory; and the service downtime can last up to 600ms. Fig. 1 shows the impact on the customer network traffic when migrating the VM. The customer traffic drops significantly immediately after the migration starts due to the burst transmission of the initial VM memory. The impact is minimal during the subsequent iterations of memory copies with a big drop during the stop-and-copy phase.[2] Note that we dedicate 10Gbps of network bandwidth for migration. The overhead thus comes from KVM's dirty memory tracking and the competition on the memory bandwidth. Fig. 1 only shows the result for the migration of a single VM. A cluster-wide VM migration could lead to much worse results due to network congestion. Importantly, VM live migration affects high-end customers most because their VMs have larger memory sizes and use more CPU power. Some high-end customers do not trust the provider's statistics; instead, they run their own bandwidth and CPU monitoring tools to detect any service downtime. Hundreds of milliseconds in downtime are sufficient to cause severe problems for customers like stock exchanges and banks.

Unlike VM migration, Orthus uses VM grafting to directly move the VM to the new VMM instance, avoiding the lengthy memory copies and downtime. Our collected data in the real cloud environment shows that Orthus can reduce 99% of the total migration time and 90% of the service downtime with the same VM.

2.3 Migration of Passthrough Devices

Device passthrough gives VMs direct access to hardware devices. It is critical for high-performance VMs and reduces the attack surface of the host kernel because it does not need to drive those devices. Device passthrough is used widely in public clouds. Even though some research has studied the migration of specific devices such as GPUs [31, 38] and SRIOV-based network adapters, there is no generic solution to migrate passthrough devices. Device migration needs to save a snapshot of its internal states, including the ongoing DMA transactions, and restore it in the target server. Currently, no hardware devices natively support live migration.

A possible solution is to detach from the current device and attach to a same one in the target server. However, all the ongoing transactions will be lost. This may cause, say, network packets to be lost. In worse cases, it could corrupt the customer data, for example, if GPUs are being used to accelerate computation and all the pending GPU tasks are lost. To avoid that, the cloud provider has to either notify the customer to stop the GPU tasks or wait for the VM to terminate. Neither is an ideal solution. This problem is addressed in Orthus by the device handover, which can seamlessly migrate all sorts of passthrough devices, including GPUs,

[2]We sample the data in 0.2-second intervals; therefore, there is still some traffic during the stop-and-copy phase.

SRIOV-based devices, and FPGAs, without losing ongoing transactions.

3 System Design

The design of Orthus can be summarized by three key techniques: dual KVM, VM grafting, and device handover. Specifically, dual KVM creates two instances of the KVM kernel module – one is called the source KVM that is currently running the VM, and the other is the target KVM, an upgraded version of KVM. VM grafting directly moves the VM from the source KVM to the target KVM. Lastly, device handover seamlessly transfers the ownership of passthrough devices to the target VM. In the rest of this section, we describe each technique in detail.

3.1 Dual KVM

KVM is the most popular hypervisor used in public clouds, such as Google Compute Engine and Amazon EC2. It is a type-II hypervisor, which runs inside a host OS kernel and relies on it for the resource management.[3] KVM consists of a few kernel modules. On the Intel platform, it consists of kvm.ko, the architecture-independent layer of KVM, and kvm-intel.ko, the module that manages Intel's hardware virtualization extension [14] (kvm-amd.ko on the AMD platform). To load two instances of KVM, we need to resolve name conflict of the symbols (i.e., functions and global variables) and provide an arbitration mechanism to allow them to share the access to the virtualization hardware.

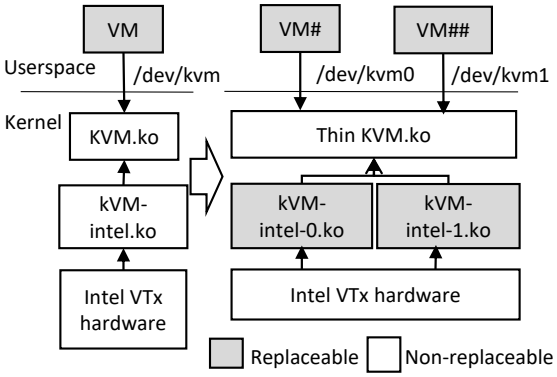

Figure 2. Dual KVM system in Orthus

Fig. 2 shows Orthus' dual KVM structure compared to the original KVM design. In Orthus, we move most of kvm.ko's functions into kvm-intel.ko and keep kvm.ko as a very thin layer for interfacing with the host kernel. This essentially moves all the vulnerable parts of KVM into kvm-intel.ko. To load two copies of kvm-intel.ko, we associate all the original global variables in KVM with kvm-intel.ko and make all the global functions local. When loaded, kvm-intel.ko registers itself to kvm.ko by providing an array of pointers into its related internal functions. It then creates the device node at /dev/kvm*n* (*n* is 0, 1, ...) for QEMU to interact with the KVM kernel module. Multiple kvm-intel.ko instances can be loaded and work concurrently without conflicts. They can access the virtualization hardware (e.g., Intel VTx) in a time-shared fashion. This is feasible because Intel VTx is mostly stateless; all the VM-related states are saved in the VM's VMCS data structure[13]. A typical scenario is for kvm-intel-0.ko to run all the VMs and for kvm-intel-1.ko to contain the upgraded version with new features and/or security patches. We can then "graft" the VMs from kvm-intel-0.ko to kvm-intel-1.ko one by one until all the VMs are moved to the new version. kvm-intel-0.ko can then be unloaded to remove the vulnerable/unused code from the kernel.

[3] A type-I hypervisor like Xen runs directly on the bare metal.

3.2 VM Grafting

With KVM, each VM is represented by a QEMU process. All the resources a VM owns are allocated and managed by this process, including the guest memory, virtual CPUs (vCPUs), storage, network adapters, etc. In particular, the VM's memory is allocated from the process' heap. To move a VM from one KVM instance to another, we need to move both the VM's memory and all its internal states. Technically, we could reuse VM live migration for this purpose. However, that requires the server having twice the memory to copy the VM's memory, thus not feasible. Instead, we just "cut-and-paste" the VM's memory and internal states from one KVM instance to another since both KVM instances run on the same machine. We call this process VM grafting. As previously mentioned, QEMU has a sizable number of vulnerabilities and new features are mostly integrated through QEMU. Consequently, we must allow QEMU to be upgraded as well. We implement VM grafting in QEMU because the states maintained by QEMU for each VM are large and complex. Some cloud providers replace QEMU with proprietary implementations. A similar design can be applied.

Fig. 3 shows the VM grafting process in Orthus. In step 1, we mark the VM's memory as reserved in order to move it to the new VM later. In the second step, we first fork the original QEMU process; the parent process then pauses and saves the VM's internal states using QEMU's savevm_state function; the child process subsequently calls the execv function to load the upgraded QEMU program (step 3). If the initialization succeeds, the new QEMU process (i.e., the child) restores the VM states with QEMU's loadvm_state function (step 4). We use this multi-process design because VM initialization and restoration is a lengthy process and could potentially fail. By forking the process, the parent can unpause the VM if the child fails to restore the VM. In the final step (step 5), we map the VM's memory in the new QEMU process and resume the VM. If any failure occurs in step 3, 4, and 5, we discard the child process and resume the saved VM in the parent process.

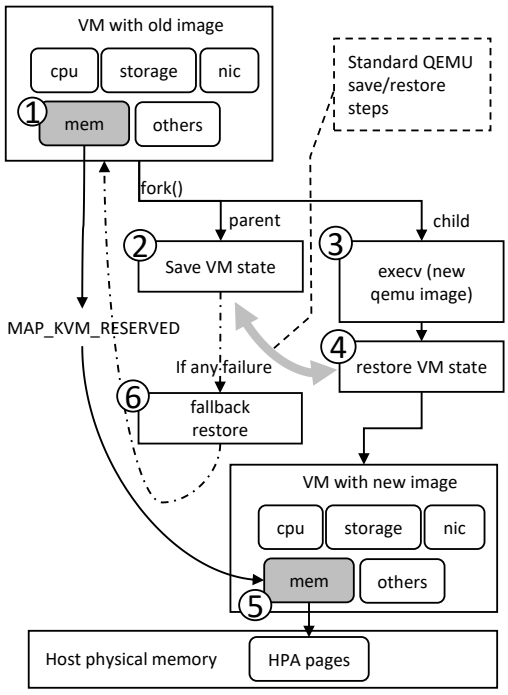

Figure 3. VM grafting in Orthus

In order for the new QEMU process to map the VM's memory, it needs to share the memory with the old one. This can potentially be implemented using the existing POSIX shared memory (e.g, mmap with MAP_ANONYMOUS | MAP_SHARED). Unfortunately, POSIX shared memory will not survive the execve system call, which reloads the calling process with a new program. Note that the child process uses execv to load the upgraded QEMU program. To address that, we add a new flag to the mmap system call (MAP_KVM_RESERVED). Memory allocated with this flag is shared between the parent and child processes and survives the execve system call. As usual, such memory will be released back to the kernel when the VM exits. We would like to mention that this change to the kernel only adds a few lines of source code and the flag is protected from being used by regular processes. As such, this new flag should not become a security concern.

With VM grafting, we can significantly reduce the migration time and the downtime compared to VM live migration. Moreover, we can upgrade as many VMMs as possible simultaneously because of the absence of network traffic.

3.3 Passthrough Device Handover

Device passthrough gives the VM direct access to hardware devices. Heterogeneous cloud services rely on GPU or FPGA passthrough to accelerate customer applications, such as deep learning and AI. Modern hardware devices are often programmed through DMA (direct memory access). For example, the device driver in the VM fills a buffer with GPU commands and submits the buffer to the GPU via its DMA engine. The GPU executes the commands and writes the results back to the memory, also via DMA. However, direct device access allows a malicious VM to compromise the VMM or other VMs by misusing the DMA to read/write their data, i.e., the so-called DMA attack. To address that, modern CPUs provide IOMMU to remap the DMA memory access (e.g., Intel VT-d [2, 5, 20], and AMD IOMMU [6]). Normally, a VM has three types of memory addresses: GVA (guest virtual address), GPA (guest physical address), and HPA (host/actual physical address). GVA is translated into GPA by the guest page table in the same way as a non-virtualized system; GPA is then translated into HPA by the extended page table (EPT) to access the physical memory. However, this two-level address translation is only applied to the software access, not the DMA access. DMA access instead is translated by IOMMU from GPA to HPA.

To migrate a VM with a passthrough device, we need to migrate the device's internal states, rebuild the IOMMU remapping table on the target server, and save/restore the ongoing DMA operations in order to avoid losing data. So far, there is no universal solution to migrate passthrough devices, especially for complex devices like GPUs and FPGAs. In Orthus, we address these challenges by the device handover.

In Linux, device passthrough is enabled by VFIO (virtual function I/O), an IOMMU/device agnostic framework to allow direct device access by the user space. VFIO exposes a number of device nodes under /dev/vfio/* for QEMU to use. Orthus introduces a new user-space component, the VFIO connector, to wrap all the VFIO-related file descriptors and interfaces, including /dev/vfio, /dev/vfio/grp*, VFIO eventfd, and KVM irqfd. VFIO connector wraps QEMU's access to these file descriptors. It is the only path to access the VFIO kernel. During the live upgrade, the ownership of the VFIO connection is handed over to the new QEMU process, as shown in Fig. 4. Specifically, the access to KVM is switched from path 1 (kvm-intel-0) to path 3 (kvm-intel-1), and the VFIO connector is owned by the new VM (path 4). By handing over the VFIO connector, Orthus avoids closing and reopening the stored file descriptors. From the kernel's point of view, nothing is changed during the live upgrade.

With the design of Orthus, there is no need to migrate the IOMMU mapping or the device's internal states. Specifically, the VM's memory is shared by the new and old QEMU processes. Consequently, the mapping from GPA to HPA is not changed during the live upgrade; the IOMMU translation table thus remains valid. Because of this, any ongoing DMA operations can continue execution without interruption, even when the VM is stopped. If a DMA operation is completed while the VM is paused for upgrade, the device will raise an interrupt, which is routed to KVM by the kernel. KVM temporarily caches the interrupt in its irqfd. The irqfd

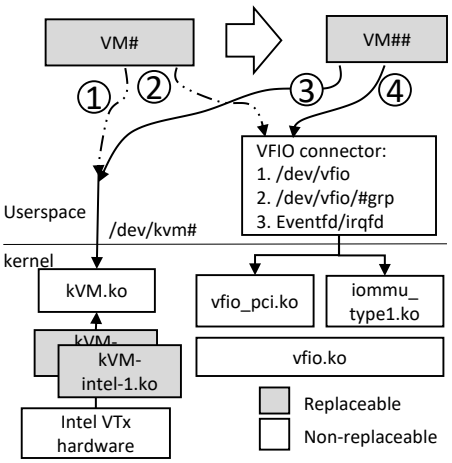

Figure 4. Device handover in Orthus

is usually connected to an eventfd. QEMU can receive the interrupt by reading from this eventfd. As mentioned earlier, the eventfd is stored in the VFIO connector and handed over to the new QEMU process during the live upgrade. When the VM is resumed, it reads from the eventfd and receives the pending interrupts. During our implementation, we spent significant efforts to prevent interrupts from being lost. However in large-scale tests, we still found data loses due to the missing interrupts. To address that, we inject a virtual irq into the VM right after handing over the device.

Device Types	Supported Devices
AMD GPUs	S7150, S7150 SRIOV
NVIDIA GPUs	NVIDIA Tesla M40, P100, P4, V100 up to 8 cards
FPGA	Xilinx, Stratix
Network adapters	Intel 82599ES SFP+

Table 3. Passthrough devices supported by Orthus

Device handover is a generic solution to migrate passthrough devices to the restored VM. It supports migrating all the passthrough devices currently used in our cloud. The list of these devices is given in Table 3.

4 Evaluation

Orthus has been deployed in the production environment of Alibaba cloud datacenters. It has significantly improved our ability to promptly upgrade VMMs with security patches or new features. In this section, we evaluate the performance of Orthus in the real cloud environment. Specifically, we will demonstrate that Orthus can significantly reduce the total migration time and downtime over VM live migration, and it can be used for simultaneous large-scale VMM upgrade.

4.1 Experiment Setup

Two types of the host configurations were used in the experiments, as shown in Table 4. Computing service is typically used by web servers, image processing, media servers, and database servers; GPU service is popular among machine learning, deep learning, AI, and graphic rendering applications. The VM configurations are shown in Table 5. All the VMs were configured to use 2MB huge pages for the memory. We conducted the experiments on both individual servers and a number of clusters with 45, 000 VMs. The former measured the performance impact on an individual VM; while the latter measured the overall improvement to the large-scale VMM upgrade.

Host Type	Hardware Configurations
Computing Service	Intel(R) Xeon(R) CPU E5-2682 v4 CPU: 32 cores/64 threads@2.50GHz in 2 sockets 256GB RAM, 1TB SSD
GPU Service	CS host + NVIDIA Tesla V100 V100 up to 8 cards

Table 4. Evaluation host nodes configuration

Guest VM	Configurations
CS Guest	16vCPUs, 4~32GB RAM, 40GB Cloud Disk (SSD)
GS Guest	16vCPUs, 128GB RAM, 40GB Cloud Disk (SSD) NVIDIA Tesla V100 GPU

Table 5. Evaluation guest VM's configuration

We aimed at measuring the performance of Orthus on the typical use cases of our cloud, such as web servers, database servers, media processing, and GPU-accelerated deep learning workloads. To this end, we used the following standard benchmarks to simulate these services and measured the total migration time and downtime caused by Orthus:

- **ApacheBench and Netperf**: ApacheBench [9] and Netperf [22] were used to simulate traffic to web servers.
- **MySQL and sysbench**: sysbench was used to measure the performance of the MySQL database during VMM live upgrade [26].
- **SPEC CPU2006**: SPEC CPU2006 consists of a number of CPU and memory intensive workloads [19] . We chose SPEC benchmark suites because they are similar to the real customer usage of our cloud.
- **Tensorflow**: Tensorflow is a popular machine learning framework from Google [4]. The MNIST dataset is a large database of hand-written digits commonly used to train image processing systems [28]. We used this dataset to train a Tensorflow model.

4.2 Service Downtime

Minimizing service downtime is the major concern of cloud providers. We measured the service downtime of Orthus by upgrading the VMM while the benchmarks were running. We recorded the service downtime as the time between saving

the VM state (Fig. 3, step 2) and resuming the VM execution (step 5). With this definition, the service downtime is virtually the same as the total migration time for Orthus. We used different benchmarks to simulate the common use cases of our cloud services, including computation, database, web, and machine learning. The goal was to understand how Orthus might impact different types of customer tasks in case of the downtime.

Service downtime for computing services: we used SPEC CPU2006, MySQL/sysbench, and web server to simulate the computation, database, and web services. While running these benchmarks, we upgraded the VMM with Orthus every 5 seconds until 100 samples of downtime were recorded. The guest VM had 16 virtual CPUs, 32GB memory, and 40GB SSD-based Cloud disk. We run just this VM on the test server to avoid the interference from co-hosted VMs.

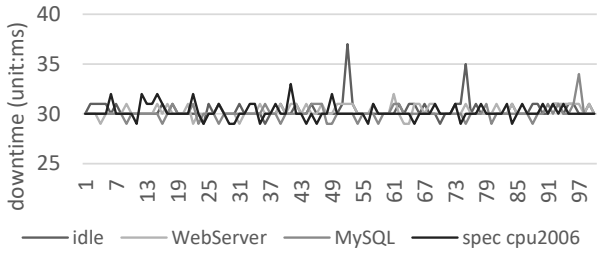

Figure 5. Service downtime for four cases (unit *ms*), sampled every 5sec, total 100 sample

Fig. 5 shows service downtime of these four cases (including the idle VM for comparison). Overall, the downtime was between 29*ms* to 37*ms* with an average of 30*ms*. We also measured the downtime and total migration time for MySQL under VM live migration with 800 samples. They were 268*ms* and 27.3*s* on average, respectively. Compared to VM live migration, Orthus reduced about 90% of the downtime and 99% of the total migration time. Notice that all the sampled data of Orthus lied within a narrow range (29*ms* to 37*ms*). As such, different workloads of the VM had limited impact on the downtime of Orthus. Our experiments also showed that the memory size of the VM did not affect the downtime much (this is expected since we do not copy the VM's memory.) Unlike Orthus, a busier or larger VM has longer total migration time and downtime in VM live migration.

Fig. 6 shows the breakdown of a typical sample of the downtime (30*ms* in total). It took 12*ms* to wait for the vm_start signal from libvirt, a popular toolkit to manage virtualization platforms.[4] It took another 4*ms* to restore the VGA device (cirrus_vga), 4*ms* for the 16 vCPUs, and 3*ms* for virtio-net and virtio-balloon, each.

Service downtime for GPU services: unlike VM live migration, Orthus can seamlessly handle VMs with passthrough

[4]This part was caused by the quirks in the design of libvirt. We are investigating the way to reduce it.

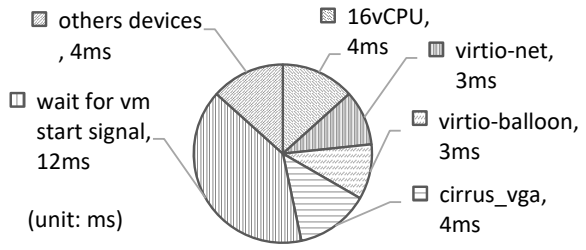

Figure 6. Breakdown of service downtime in Orthus

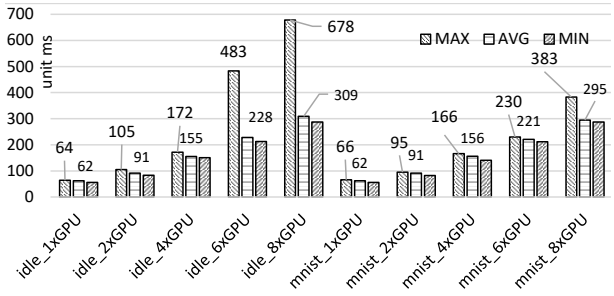

Figure 7. Downtime of Orthus for TensorFlow training workloads with 1,2,4,6,8 GPUs (unit: *ms*)

devices. In our evaluation, we measured the downtime caused by Orthus for VMs with GPUs. Specifically, we used the MNIST dataset to train a Tensorflow model. The dataset contains 70, 000 images of handwritten digits (0 to 9), divided into the training set (60, 000 images) and the testing set (10, 000 images). During the training, we upgraded the VMM every 5 seconds and recorded the downtime. We took 100 samples of downtime. The results are summarized in Fig. 7. The VM was assigned with 1, 2, 4, 6, or 8 GPUs. Fig. 7 also shows the downtime of the idle VM as a comparison.

Similar to computing services, the downtime for GPU services also fell within a small range of time, i.e., the performance of Orthus was consistent with only minor fluctuation. It seems that the downtime increased by about 30*ms* for each additional assigned GPU. This is because each GPU introduces additional PCI configure space and VFIO-pci status that need to be saved and restored by Orthus. Interestingly, the maximum downtime of idle VM was about 300*ms* longer than the active VM. This is because that idle GPUs may fall into power saving states. To restore the VM on the new VMM (Fig. 3, step 4), the target QEMU needs to wake up the GPU in order to read its PCI config space. This takes much longer than reading the PCI config space of an awake device.

Comparison to local VM live migration: technically, it is possible to run live migration locally, i.e., use the same server as both the source and target servers during the migration. This eliminates the network traffic for migration. However, it cannot upgrade the KVM kernel module or migrate passthrough devices (unless with our dual KVM and

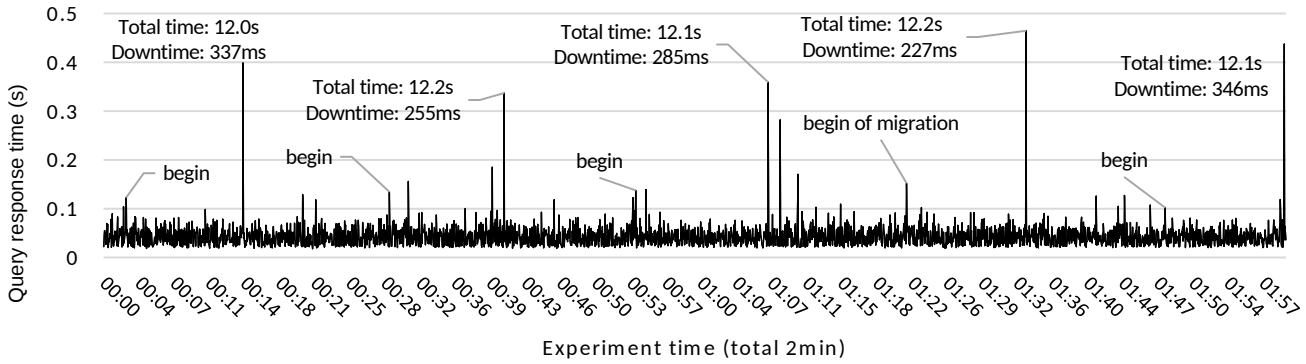

Figure 8. Mysql random query performance during 5 times live migration (lower is better)

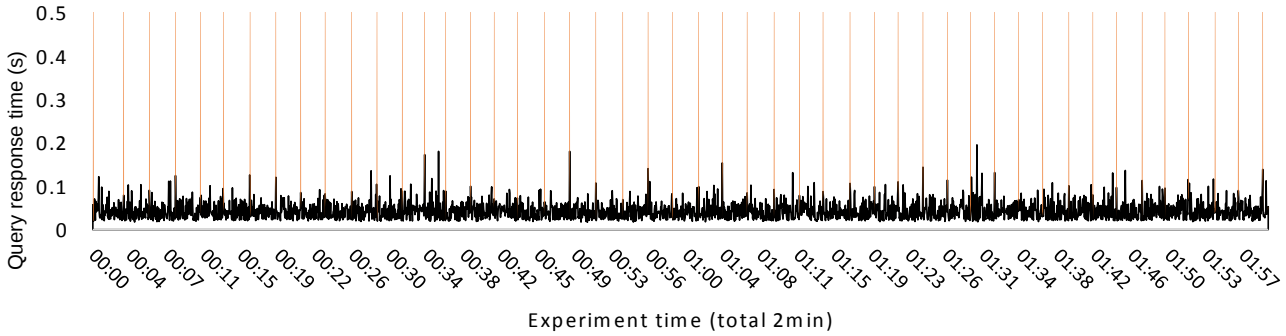

Figure 9. Mysql random query performance during 50 times Orthus live upgrade (lower is better)

device handover techniques). Compared to local VM live migration, Orthus' VM grafting can significantly reduce the downtime. The average downtime for Orthus is about 31*ms*, while that for local migration is about 268*ms* for 16GB VM with the MySQL workload. The distribution of downtime is also much wider for the local migration (in a range of about 400*ms* vs. 20*ms*). Therefore, Orthus is a better solution than local VM live migration.

4.3 Performance

We measured the impact of Orthus and VM live migration on the performance of services running in the VM, including the MySQL and Tensorflow workloads as the typical examples of computing and GPU services.

Performance impact on computing services: The test VM had 16 virtual CPUs and 16GB memory. The host server had two SPF+ network adapters bonded together for a total of 20Gbps bandwidth. We reserved 10 Gbps bandwidth for live migration in order to separate it from the customer traffic. The VM run the MySQL database server. We used sysbench to measure the response time for random queries under Orthus and VM live migration.

Fig. 8 shows the response time of MySQL queries under live migration. We executed 5 live migration sessions within 2*min* while the benchmark was running. The average total migration time was 12*s* and the downtime was between 255*ms* and 346*ms*. The figure shows notable performance drops at the beginning of each session and more significant drops during the downtime. As mentioned before, the overhead comes from mainly the initial burst memory transfer (competition in the memory bandwidth), the dirty memory tracking, and the downtime. These performance drops, albeit short, could lead to complaints and lose customers.

Figure 9 shows a totally different picture under Orthus. We collected the data within the same 2*min* period but run Orthus 50 times (10 times of live migration). Each vertical line in the figure corresponds to one execution. Each session incurred about 30*ms* in downtime. As shown in the figure, the impact of Orthus on the MySQL service was minimal.

Performance impact on GPU services: We used the same VM but with two Nvidia Telsa V100 GPUs. We run the same Tensorflow training task. The entire training task took around 60*s* without Orthus. We then run Orthus 15 times during the training. We did not notice any statistically significant difference in the time of the training task in the repeated tests. According to Fig. 7, Orthus should have caused 1.35*s* of downtime in total ($15 \times 90ms$). The reason that we did not find noticeable difference is that, even though the

vCPUs were stopped during the downtime, the GPUs were still processing the pending workloads. We would like to point out that VM live migration simply does not work well in this case because it cannot migrate GPU tasks.

4.4 Real-world Differences Made by Orthus

We have deployed Orthus in most servers in our cloud datacenters. It has become the most used tool in our daily maintenance. We use Orthus to deploy security patches and new features across the datacenters. It has significantly simplified and shorten the system upgrade process. We only use VM live migration sporadically, mostly to replace failed hardware devices (e.g., disks). With Orthus, we can deploy a new version of VMM across the whole cloud in days (with the staged rollout), while it would previously take months with live migration. In particular, we previously had to *manually* upgrade the VMs with passthrough GPUs or FPGAs; these tasks can now be fully automated by Orthus. To demonstrate the gain in the efficiency, we describe our experience in deploying KVM's halt polling feature.

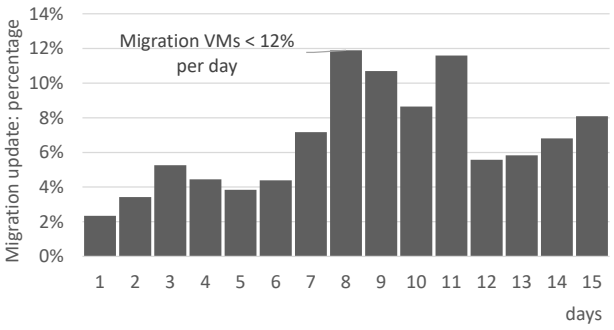

Figure 10. Upgrading 45, 000 VMs with live migration

Halt polling is a feature in KVM to reduce the guest latency by polling wake conditions in the host for a short period before halting the guest (upon its own request). Kernel live patching cannot be used to apply this new feature because some fundamental data structures, such as kvm_vcpu, are changed. In our first setup, we used VM live migration to upgrade 45, 000 VMs in a couple of clusters. We applied our usual procedure in upgrading these VMMs. In total, it took 15 days to complete the upgrade. Fig. 10 shows the percentage of VMs upgraded each day. Everyday, no more than 12% of VMs could be upgraded. The reasons that slowed down the process were:

- Most time was wasted to wait for the spare servers. These clusters, like others, were close to their capability with just enough spare servers to handle urgent/-peak traffic.
- To avoid impacting the customer traffic, live migration was only conducted in the midnight.
- For the same reason, we limited the network bandwidth for migration to 10Gbps. Therefore, we could not simultaneously migrate too many VMs.
- The upgrade was deployed with the staged rollout, a common practice in the upgrade process. Initially, only less than 2% VMs were migrated each day while we closely monitored the customer feedback. The rollout would have been stopped if we received any complaints. Note that the maximum we could do was to migrate about 12% VMs each day due to the previous three constraints. A more aggressive rollout schedule may cause problems because of the network congestion and the competition for spare servers.

VMs	Downtime	Total Migration Time
0.8%	>0.48s	>96.3s
0.2%	>2.3s	>174.9s
Max	9.82s	427.0s

Table 6. Distribution of downtime and total migration time for the worst 1% VM

We analyzed the whole migration logs. Even in the best case, we could not avoid long downtime for some VMs. Statistics showed that almost 1% of VMs suffered a downtime of more than 0.48*s* and the total migration time of more than 1.5*min*. Even worse, the VM that suffered most often had higher configurations with more vCPUs and memory. Table 6 shows the distribution of downtime and total migration time for these 1% VMs. Notice that the worst downtime was 9.82*s* and the worst migration time was more than 7*min*.

Next, we used Orthus to deploy this feature to all the other VMs in *the whole cloud with millions of VMs*. In total, it only took 45*min* to upgrade all the VMs across the cloud, staged in three days. All we needed to do was to send the upgraded KVM/QEMU to all the servers. Orthus then upgraded them to the new version, no matter whether VMs were busy or not. In each case, the downtime was about 30*ms*. Our previous experience shows that it would have taken *a few months* to accomplish the same cloud-wide update by live migration.

Overall, Orthus has tremendously simplified the daily maintenance work in one of the largest public clouds.

Using Orthus to fix Spectre v1 attack: Meltdown and Spectre are side-channel attacks that exploit speculative execution in modern CPUs. Meltdown is a rouge data load that can break the kernel/user-space isolation [29]. Meltdown does not work against KVM because the memory of KVM or other VMs is not mapped in the address space of the malicious VM. However, Spectre can break the isolation between different applications (potentially VMs), posing a severe security threat to the cloud [24]. The official recommendation to fix Spectre by Intel requires loading a new microcode and upgrading the system software [15]. The microcode adds three new capabilities to compatible CPUs: indirect-branch restricted speculation (IBRS), single-thread

indirect-branch predictors (STIBP), and indirect-branch predictor barrier (IBPB). These capabilities can be used by the updated system software to prevent the misuse of branch prediction by Spectre. The KVM community made the necessary changes to both KVM and QEMU.[5] Fortunately, Intel has provided a method to update the microcode without rebooting [16].

To fix Spectre in our cloud, we first upgraded each server's microcode at run-time to add these new capabilities; we then patched our KVM/QEMU with the upstream fixes; and finally deployed the upgraded KVM/QEMU by Orthus. With Orthus, we were able to fix Spectre across our whole cloud within just a few days, even with the staged rollout. This demonstrated the tremendous advantage of Orthus over VM live migration in the large-scale cloud infrastructure – it would have taken a few months for VM live migration to upgrade the whole datacenters. As mentioned before, it only takes days for new exploits to appear after a vulnerability is publicized.

5 Related Work

Kernel live patching, VMM live upgrade, and VM live migration are three powerful, complementary methods to update the cloud infrastructures. In this section, we compare Orthus to the representative systems in these other two methods.

5.1 Kernel Live Patching

Kernel live patching is a lightweight solution to apply temporary patches to the running kernel. It can apply patches at the function level (e.g., Kpatch [27] and KGraft [33]) or the instruction level (e.g., Ksplice [7]). Recently, KARMA automatically adapts an official kernel patch to many fragmented Android kernels [11]. Kernel live patching is most suitable for applying simple security patches. It does not support patches that may change persistent data structures (i.e., data structures that have allocated instances on the kernel heap or stacks). Moreover, kernel live patching could become a maintenance headache. It is especially problematic to the public cloud, which has hundreds of thousands of servers. The cloud provider needs to keep track of all the live patches applied to individual servers and test whether these kernel patches are compatible with each other or not. Eventually, each server needs to be rebooted to install a clean upgrade. Orthus instead focuses on the live upgrade of the whole VMM. It can support complex changes to KVM/QEMU, including adding new features. Such changes simply cannot be applied by kernel live patching. Nevertheless, kernel live patching is still useful for the cloud provider.

5.2 VM Live Migration

VM live migration temporarily moves the VMs to a backup server, upgrades the system, and then moves the VMs back [12]. VM live migration allows the cloud provider to upgrade almost everything in the original server, from hardware devices to the operating system. However, the cluster-wide live migration is inherently limited by the availability of backup servers and the network bandwidth. Therefore, it cannot meet the time requirement when deploying urgent (security) updates. Moreover, live migration incurs relatively long downtime and total migration time and cannot handle passthrough devices. Compared to VM live migration, Orthus can reduce the downtime and total migration time by 90% and 99%; it does not need backup servers or consume network bandwidth; it can upgrade a large number of VMs in parallel. Since its deployment in our cloud, Orthus has mostly replaced VM live migration, which is only used to replace failed hardware devices and major kernel upgrade (new versions).

Efforts to reduce the impact of live migration: A study of the performance in the cloud shows that application service levels drop during the migration, especially for Internet applications[40]. The overhead comes from iterative memory copy between the source and target VMs. Jin et al. propose a compression-based migration approach to reduce its network traffic [21]. Specifically, memory pages are selectively compressed in batches on the source VM and decompressed on the target VM. This system can reduce service downtime by 27%. Liu et al. propose to use full system tracing and replay to provide fast, transparent VM migration[30]. Availability of network bandwidth is another constraint to the live migration, especially for the large-scale migration in the datacenters. Researchers have proposed to limit the migration network traffic within the inner racks, instead of the core network links where network congestion is more costly. Deshpande et al. propose a distributed system for inter-rack migration [17]. They were able to reduce the total network traffic *on the core network* by 44%.

Compared to these systems, Orthus can reduce the migration time and downtime by 90% and 99%, respectively, and completely eliminate the network traffic caused by iterative memory copying. As such, Orthus can support massive parallel VMM upgrade.

Live migration of passthrough device: How to migrate pass-through devices is still a big challenge to public cloud providers. There are disconnected solutions to migrate specific devices, essentially by introducing another layer of abstraction. For example, an approach has been proposed to live-migrate Intel GPUs [31]. Specifically, they assign sliced vGPUs [37] that share a physical GPU to multiple VMs. The sliced vGPUs can be migrated between different Intel GPUs. There is also a similar system designed for GPU-based machine-learning and AI applications that exposes a virtual CUDA interface to the VM [35]. Both systems are tied to the GPUs they support and require a new implementation for each used GPU. Kadav and Swift propose to use shadow drivers to capture and reset the state of a device driver in order

[5] QEMU was changed to expose these capabilities to the VMs.

to migrate passthrough devices [23]. Intel provides a solution to migrate network devices. Specifically, it bonds a virtual NIC and physical NIC together and hot-unplug/hot-plug the virtual NIC during migration [8, 18, 32, 42]. As shown in Table 3, our cloud supports several types of passthrough devices, including FPGAs.

Orthus' device handover is a generic solution for device migration during the VMM live upgrade. It can seamlessly handover the passthrough devices to the target VM without losing the ongoing operations. It can support all the passthrough devices in our cloud without device-specific code.

5.3 Dual VMM

Dual VMM has been used to isolate resource and improve security. For example, it is used to consolidate the HPC (high-performance computing) and commodity applications in the cloud [25]. MultiHype uses multiple VMMs in the single host to improve security [36], so that a compromised VMM is confined to its own domain without affecting the others. Dual KVM is one of three components of Orthus. Together, these three components allow us to upgrade both KVM and QEMU to a new version without disrupting the running VMs.

6 Discussion

In this section, we discuss potential improvements to Orthus and the future work. First, by design, Orthus is a method to upgrade the live VMM and thus cannot upgrade the host kernel. This design is based on the cloud threat model in which the host kernel is mostly isolated from the guest through network segregation, device passthrough, and driver domains. The main attack surface is the VMM that directly interacts with untrusted VMs. Our survey of all the KVM-related vulnerabilities in the CVE database shows that only 2 out of 95 vulnerabilities in the kernel can be exploited by VMs. The other 93 vulnerabilities lie in the VMM. Therefore, keeping the VMM up-to-date is critical in protecting the cloud infrastructure from malicious VMs. The other two vulnerabilities can be patched by kernel live patching or VM live migration. Moreover, new functions usually need to be integrated in the VMM. Kernel live patching, VMM live upgrade, and VM live migration form a complete toolset for cloud providers. Our own experience shows that Orthus is the most useful of the three because of its effectiveness, scalability, and minimal impact to the customer traffic.

Orthus uses QEMU's save and restore VM functions to "cut-and-paste" VMs from the old VMM to the updated one. As such, it assumes that the saved VM image can be restored in the update VMM. This assumption holds as long as the VM format in QEMU remains backward compatible. QEMU, as a mature open-source project, rarely breaks the backward compatibility. We have not met such cases either during the deployment of Orthus in our cloud. If this happens, Orthus will fail to restore the VM image on the new KVM. However, by design, Orthus will resume the execution of the VM on the original VMM instance. Consequently, there will be no interruption to the VM. This problem can be addressed later by adding the compatibility code to QEMU. Note that, VM live migration uses the same mechanism to save and restore VMs. As such, they have exactly the same problem (this is also the reason why QEMU is unlikely to break backward compatibility.) Note that we cannot resort to the live migration to update the VMM in this case because the live migration will also fail.

VMM live upgrade is an important defense again malicious VMs. There are orthogonal efforts to mitigate this problem from different angles. For example, fuzz testing has been used to discover unknown vulnerabilities in the KVM/QEMU stack. It essentially feeds the VMM with random/unexpected inputs, trying to trigger abnormal behaviors. Fuzz testing has been proved to be effective in finding vulnerabilities in KVM [1]. There are also efforts to reduce the privileged code in KVM by moving non-performance critical code to the user space [1], and to sandbox and isolate KVM so that a compromised KVM cannot affect other VMs [41]. All these efforts improve the security of the cloud. On the other hand, vulnerabilities are inevitable given the complexity of KVM/QEMU and its host OS (Linux). A method to promptly upgrade the VMM like Orthus is essential to the cloud security.

7 Conclusion

We have presented the details of Orthus, a system that can upgrade the live VMM in the cloud. Orthus features three key techniques: dual KVM, VM grafting, and device handover. Together, they allow us to upgrade the running KVM/QEMU to an updated version with new features or (complex) security fixes. In addition, Orthus can seamlessly handover passthrough devices to the new instance without losing any ongoing (DMA) operations. Our evaluation shows that Orthus can reduce the total migration time and service downtime by more than 99% and 90%, respectively. It has become the most effective and indispensable tool in our daily maintenance and operations of hundreds of thousands of servers and millions of VMs.

8 Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments that helped improve the presentation of this paper. Zhi Wang was partially supported by National Science Foundation (NSF) under Grant 1453020; Qi Li was partially supported by National Natural Science Foundation of China (NSFC) under Grant 61572278 and U1736209. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF or NSFC. Xiantao Zhang and Qi Li are the corresponding authors of this paper.

References

[1] 7 ways we harden our kvm hypervisor at google cloud: security in plaintext. https://bit.ly/2jSjru3, 2017.

[2] Intel® virtualization technology for directed i/o architecture specification. https://software.intel.com/sites/default/files/managed/c5/15/vt-directed-io-spec.pdf, 2018.

[3] Search results for cve in kvm. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=kvm, 2018.

[4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[5] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. Intel virtualization technology for directed i/o. *Intel technology journal*, 10(3), 2006.

[6] AMD. Amd i/o virtualization technology (iommu) specification. 2016.

[7] Jeff Arnold and M Frans Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 187–198. ACM, 2009.

[8] Adam M Belay. Migrating virtual machines configured with pass-through devices, March 27 2012. US Patent 8,146,082.

[9] Apache Bench. ab-apache http server benchmarking tool.

[10] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 169–179. ACM, 2007.

[11] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. Adaptive android kernel live patching. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[12] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[13] Intel Corporation. Intel® 64 and ia-32 architectures software developer manuals. https://software.intel.com/en-us/articles/intel-sdm, 2016.

[14] Intel Corporation. Intel® virtualization technology (intel® vt). https://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html, 2016.

[15] Intel Corporation. Intel analysis of speculative execution side channels. https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf, 2018.

[16] Intel Corporation. Intel processor microcode package for linux description. https://downloadcenter.intel.com/download/27945/Linux-Processor-Microcode-Data-File?product=873, 2018.

[17] Umesh Deshpande, Unmesh Kulkarni, and Kartik Gopalan. Inter-rack live migration of multiple virtual machines. In *Proceedings of the 6th international workshop on Virtualization Technologies in Distributed Computing Date*, pages 19–26. ACM, 2012.

[18] Yaozu Dong. Efficent migration of virtual functions to enable high availability and resource rebalance, September 10 2013. US Patent 8,533,713.

[19] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[20] Radhakrishna Hiremane. Intel virtualization technology for directed i/o (intel vt-d). *Technology@ Intel Magazine*, 4(10), 2007.

[21] Hai Jin, Li Deng, Song Wu, Xuanhua Shi, and Xiaodong Pan. Live virtual machine migration with adaptive, memory compression. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE, 2009.

[22] Rick Jones et al. Netperf: a network performance benchmark. *Information Networks Division, Hewlett-Packard Company*, 1996.

[23] Asim Kadav and Michael M. Swift. Live Migration of Direct-access Devices. In *Proceedings of the First Conference on I/O Virtualization*, WIOV'08, Berkeley, CA, USA, 2008. USENIX Association.

[24] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

[25] Brian Kocoloski, Jiannan Ouyang, and John Lange. A case for dual stack virtualization: consolidating hpc and commodity applications in the cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 23. ACM, 2012.

[26] Alexey Kopytov. Sysbench: a system performance benchmark. *http://sysbench. sourceforge. net/*, 2004.

[27] Kpatch. kpatch: Dynamic kernel patching. 2018.

[28] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, 2, 2010.

[29] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[30] Haikun Liu, Hai Jin, Xiaofei Liao, Liting Hu, and Chen Yu. Live migration of virtual machine based on full system trace and replay. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 101–110. ACM, 2009.

[31] Jiacheng Ma, Xiao Zheng, Yaozu Dong, Wentai Li, Zhengwei Qi, Bingsheng He, and Haibing Guan. gmig: Efficient gpu live migration optimized by software dirty page for full virtualization. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 31–44. ACM, 2018.

[32] Zhenhao Pan, Yaozu Dong, Yu Chen, Lei Zhang, and Zhijiao Zhang. Compsc: live migration with pass-through devices. *ACM SIGPLAN Notices*, 47(7):109–120, 2012.

[33] V Pavlík. kgraft–live patching of the linux kernel. Technical report, Technical report, SUSE, Maxfeldstrasse 5 90409 Nuremberg Germany, 2014.

[34] Diego Perez-Botero, Jakub Szefer, and Ruby B Lee. Characterizing hypervisor vulnerabilities in cloud computing servers. In *Proceedings of the 2013 international workshop on Security in cloud computing*, pages 3–10. ACM, 2013.

[35] Lin Shi, Hao Chen, Jianhua Sun, and Kenli Li. vcuda: Gpu-accelerated high-performance computing in virtual machines. *IEEE Transactions on Computers*, 61(6):804–816, 2012.

[36] Weidong Shi, JongHyuk Lee, Taeweon Suh, Dong Hyuk Woo, and Xinwen Zhang. Architectural support of multiple hypervisors over single platform for enhancing cloud computing security. In *Proceedings of the 9th conference on Computing Frontiers*, pages 75–84. ACM, 2012.

[37] Jike Song, Zhiyuan Lv, and Kevin Tian. Kvmgt: A full gpu virtualization solution. In *KVM Forum*, volume 2014, 2014.

[38] Kun Tian, Yaozu Dong, and David Cowperthwaite. A full gpu virtualization solution with mediated pass-through. In *USENIX Annual Technical Conference*, pages 121–132, 2014.

[39] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886, 2018.

[40] William Voorsluys, James Broberg, Srikumar Venugopal, and Rajkumar Buyya. Cost of virtual machine live migration in clouds: A performance evaluation. In *IEEE International Conference on Cloud Computing*, pages 254–265. Springer, 2009.

[41] Zhi Wang, Chiachih Wu, Michael Grace, and Xuxian Jiang. Isolating Commodity Hosted Hypervisors with HyperLock. In *Proceedings of the 7th ACM european conference on Computer Systems*, April 2012.

[42] Edwin Zhai, Gregory D Cummings, and Yaozu Dong. Live migration with pass-through device for linux vm. In *OLS'08: The 2008 Ottawa Linux Symposium*, pages 261–268, 2008.