Mitigating vulnerability windows with hypervisor transplant

Tu Dinh Ngoc, Boris Teabe, Alain Tchana, Gilles Muller, Daniel Hagimont

Mitigating vulnerability windows with hypervisor transplant

Tu Dinh Ngoc
University of Toulouse, France

Boris Teabe
University of Toulouse, France

Alain Tchana
ENS Lyon, France
Inria, France

Gilles Muller
Inria, France

Daniel Hagimont
University of Toulouse, France

ABSTRACT

The vulnerability window of a hypervisor regarding a given security flaw is the time between the identification of the flaw and the integration of a correction/patch in the running hypervisor. Most vulnerability windows, regardless of severity, are long enough (several days) that attackers have time to perform exploits. Nevertheless, the number of critical vulnerabilities per year is low enough to allow an exceptional solution. This paper introduces *hypervisor transplant*, a solution for addressing vulnerability window of critical flaws. It involves temporarily replacing the current datacenter hypervisor (e.g., Xen) which is subject to a critical security flaw, by a different hypervisor (e.g., KVM) which is not subject to the same vulnerability.

We build *HyperTP*, a generic framework which combines in a unified way two approaches: in-place server micro-reboot-based hypervisor transplant (noted *InPlaceTP*) and live VM migration-based hypervisor transplant (noted *MigrationTP*). We describe the implementation of *HyperTP* and its extension for transplanting Xen with KVM and vice versa. We also show that *HyperTP* is easy to integrate with the OpenStack cloud computing platform. Our evaluation results show that *HyperTP* delivers satisfactory performance: (1) *MigrationTP* takes the same time and impacts virtual machines (VMs) with the same performance degradation as normal live migration. (2) the downtime imposed by *InPlaceTP* on VMs is in the same order of magnitude (1.7 seconds for a VM with 1 vCPU and 1 GB of RAM) as in-place upgrade of homogeneous hypervisors based on server micro-reboot.

CCS CONCEPTS

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

1 INTRODUCTION

As with any software, popular hypervisors are continuously subject to multiple security vulnerabilities [19, 21, 41, 51]. A *Hypervisor vulnerability window* regarding a given security flaw is defined as the time between the identification of said flaw (whether by a good or bad guy) and the integration of a correction/patch in the running hypervisor (see the red zone in Fig. 1.a). In fact, the vulnerability window is the sum of two durations: (1) the time required to propose a patch once the vulnerability is discovered and (2) the time to apply this patch in the system (see Fig 1.a). The time to release a patch is highly dependent on the vulnerability's severity and can vary from one week with vulnerabilities such as the MD5 collision attack [54], to 7 months with vulnerabilities such as Spectre and Meltdown [25, 32][1]. The time to apply the patch mainly depends on the datacenter operators' patching policy. Together, this timeframe leaves plenty of time to launch an attack against a vulnerable installation.

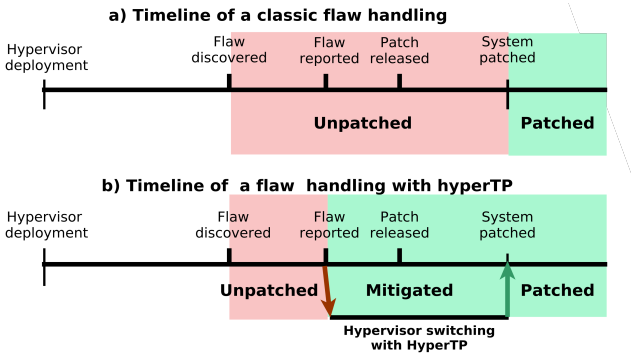

Figure 1: (a) Traditional vulnerability mitigation in data centers and (b) our hypervisor transplantation-based solution.

In this paper, we want to address the following question: *If a datacenter uses a hypervisor affected by a critical vulnerability/flaw*[2]*, how to protect the datacenter during the vulnerability window?* To this end, we introduce *hypervisor transplant*, a novel solution that involves a temporary replacement of the normal datacenter hypervisor (e.g., Xen) with a different hypervisor (e.g., KVM) which is immune to the aforementioned vulnerability (see Fig. 1.b). Once the operators are informed of said vulnerabilities, they can apply hypervisor transplant to effectively mitigate this flaw across their systems.

In our vision, each hypervisor vulnerability should be investigated for its impact on various other hypervisors. Hypervisor transplant is beneficial in two cases: either when (i) there exists another hypervisor which is not known to be vulnerable to any flaw at the time of discovery, in which case the vulnerability can be disclosed immediately to the datacenter operators; or (ii) a patch solving the vulnerability can be developed in a shorter amount of time for an alternate hypervisor than the one used in the datacenter, in which case the vulnerability can be disclosed as soon as the alternate hypervisor has been patched. In fact, if there are multiple choices of target hypervisors for transplant, the datacenter operator

[1]Note that Spectre and Meltdown are CPU-specific vulnerabilities with CVEs declared on Intel products. Hypervisors and operating systems were not directly concerned by the CVE declaration.
[2]A flaw is considered as critical when its Common Vulnerability Scoring System (CVSS) v2.0 Rating is higher than 7 [53].

will have more choices in terms of replacement even when some candidate hypervisors in the hypervisor pool prove to be vulnerable. As long as an alternative exists that is not vulnerable to any flaw, *HyperTP* can be used to ensure the security of the system.

To investigate the viability of hypervisor transplant, we collected a list of medium[3] and critical-rated vulnerabilities over the last 7 years for Xen and KVM (see Section 2). Over that period, we found only three common vulnerabilities shared by Xen and KVM: one rated critical and two rated medium. This low number supports our starting assumption that a safe alternate hypervisor exists. Overall, if *HyperTP* is reserved for mitigating critical vulnerabilities, the number of transplants required per year remains low; this means even if hypervisor transplant cannot be done too frequently, it would still bring an improvement in security.

Implementing hypervisor transplant raises several challenges in terms of dealing with hypervisor heterogenity and minimizing VM downtime. We address these challenges by proposing *HyperTP*, a framework which combines in a unified way two approaches: (i) live VM migration based transplant (noted *MigrationTP*) which offers almost no downtime but requires spare network and machine resources, and (ii) in-place micro-reboot based transplant (noted *InPlaceTP*) which does not require additional resources at the cost of a few seconds of downtime.

MigrationTP alone does not perform well at the size of a complete datacenter, since the need for spare machines and network bandwidth limits the number of VMs that can be simultaneously reconfigured. Additionally, live VM migration is only possible between homogeneous hardware, whereas datacenters use a wide range of heterogeneous hardware [15], which further reduces the number of destination servers. As an example, Alibaba reported that the migration of 45,000 VMs from a couple of clusters took 15 days of maintenance [59].

In *HyperTP*, *InPlaceTP* is preferable to *MigrationTP* when VMs can tolerate a few seconds of downtime. Microsoft Azure [36] announces downtimes of up to 30 seconds for maintenance operations; we therefore use this value as an acceptable upper bound for downtime.In our current prototype, it is up to the datacenter operator to decide which transplant approach is the most appropriate for reducing vulnerability windows, since equivalent policies are already provided for dealing with periodic platform updates [36]. Our evaluations on a small cluster in Section 5.4 show that the total reconfiguration time can be reduced by up to 80% when 80% of the VMs use *InPlaceTP*. We expect similar results at a larger scale since *InPlaceTP* avoid using network and machine resources.

While live migration and micro-reboot are known approaches, the main novelty in designing *HyperTP* is to ease the support of heterogeneous hypervisors without compromising performance. We build *HyperTP* around two principles, *Unified Intermediate State Representation (UISR)*, and *memory separation*. UISR defines a hypervisor-independent state representation whose main benefit is to simplify the re-engineering of a hypervisor into a *HyperTP*-compliant one. Memory separation minimizes downtime during *InPlaceTP* and aims to identify parts of the VM state which are hypervisor-independent and do not need to be converted when launching the target hypervisor.

[3]A vulnerability is considered medium if its CVSS v2 score is ≥ 4 and < 7.

To validate the transplant solution, we built a prototype of *HyperTP* and re-engineered Xen and KVM, the two most popular open source hypervisors, into *HyperTP*-compliant hypervisors. Xen and KVM also represent the two types of hypervisors: type-I hypervisors (Xen) and type-II hypervisors (for KVM), thus demonstrating the scope of the transplant solution. We have evaluated our prototype both at a machine scale and at a small cluster scale. At the machine scale, we are concerned with the downtime incurred by *HyperTP* when running typical cloud workloads such as SPEC CPU 2017 [8], MySQL [38], Redis [43], and Darknet [44]. At a cluster scale, we aim at evaluating the time saved by using *InPlaceTP* transplants instead of *MigrationTP* transplants when reconfiguring 10 servers, each running 10 VMs which host the above mentioned benchmarks. For this experiment, we have extended the *BtrPlace* VM scheduler [20] to orchestrate the various transplants. Our results are as follows:

- We conduct a study of vulnerabilities in Xen and KVM over the last 7 years (see Section 2). We observe that most vulnerabilities are specific to a single hypervisor and are caused by wrong implementation choices.
- We present a *HyperTP* prototype which includes both *MigrationTP* and *InPlaceTP*, therefore demonstrating the feasibility of heterogeneous transplant from Xen and KVM and vice versa.
- We describe the integration of *HyperTP* in OpenStack (see Section 4.5.2), a popular cloud orchestrator. Following our conversations with three commercial cloud operators (e.g. 3DS Outscale) and one small scale datacenter operator (Chameleon), we conclude that *HyperTP* is not likely to burden sysadmins as they never directly interact with hypervisors using vendor libraries, but rather through generic libraries (that we have adapted) such as `libvirt` (see Section 4.5.1).
- *InPlaceTP* from Xen to KVM causes minimal downtime to running VMs (as low as 1.7 seconds for a VM with 1 vCPU and 1 GB of RAM), with negligible memory and I/O overhead and without requiring VM reboots. From KVM to Xen, the downtime is about 7.8 seconds for the same VM configuration.
- *MigrationTP* offers similar performance to traditional homogeneous VM live migration.
- On a small cluster, upgrading 10 servers each running 10 VMs, using *InPlaceTP* for 80% of the VMs takes 3 minutes and 54 seconds while using *MigrationTP* would take up to 19 minutes.

2 VULNERABILITIES IN XEN AND KVM

We studied critical and medium-rated vulnerabilities in Xen and KVM over the last 7 years, using data extracted from the NIST NVD website [53]. A vulnerability is rated **critical** when its CVSS v2 score is ≥ 7, and **medium** when its CVSS v2 score is ≥ 4 and < 7. Our results are listed in Table 1.

2.1 Global analysis

Critical vulnerabilities. In Xen, 38.4% of critical vulnerabilities are related to Xen's PV mechanisms such as event channels and hypercalls, 28.2% are from resource management mechanisms (e.g

Year	Xen		KVM		Common	
	Crit.	Medium	Crit.	Medium	Crit.	Medium
2013	3	38	3	21	0	0
2014	4	27	1	12	0	0
2015	11	20	1	4	1	2
2016	6	12	3	3	0	0
2017	17	38	1	7	0	0
2018	7	21	2	5	0	0
2019	7	15	2	4	0	0
Total	55	136	13	56	1	2

Table 1: Number of critical and medium vulnerabilities per year in Xen and KVM

CPU scheduler), 15.3% are due to hardware mishandling (e.g mismanagement of VT-x states), 7.5% are from Xen toolstack (libxl) and 10.2% are linked to QEMU. Concerning KVM's critical vulnerabilities, 27% are related to ioctls, 36% on hardware mishandling, 36% are linked to QEMU, and 9% on resource management mechanisms.

We counted only one common critical vulnerability between Xen and KVM over the studied period. This vulnerability stems from QEMU, a common component on both systems. The description of this common critical vulnerability on the National Vulnerability Database website indicates that it affects QEMU's virtual floppy disk controller implementation, namely a lack of bounds checking leading to a buffer overflow vulnerability.

We additionally note the two hardware-level vulnerabilities Spectre and Meltdown, which affected both Xen and KVM in the same fashion. Both were originally reported to hardware vendors on June 1st, 2017 [1]; however, the patching and public disclosure took until January 3rd, 2018; a period of 7 months. Such a period originated from the need to coordinate patches across multiple vendors and products, a lengthy, complex and error-prone process that lead to the vulnerability being leaked ahead of time [2].

Medium vulnerabilities. In this section, we focus our analysis on common vulnerabilities. Out of 136 medium Xen vulnerabilities that we studied, only 2 are shared with KVM: CVE-2015-8104 and CVE-2015-5307, both of which are DoS vulnerabilities caused by incomplete handling of two hardware exceptions (*Alignment Check and Debug Exception*).

2.2 Timeline analysis

We attempted to build the timeline of each vulnerability. This includes the time of discovery, time of report, time of patch release, and time before patch application in the datacenter (see Fig. 1). The time to patch application is specific to each datacenter's patch policy, therefore we exclude this datapoint from our studies.

Timeline of Xen vulnerabilities. The easiest method of reconstructing vulnerability timelines is through analyzing bug trackers. Unfortunately, Xen has no central vulnerability tracker, as acknowledged by several members of Xen's security team that we contacted. Instead, Xen offers an advisory website [3] that only lists the description of each vulnerability and its associated patch. To reconstruct the timeline of each vulnerability, we adopted the following methodology: for each vulnerability, we first searched from various Xen and security forums the email of the vulnerability discoverers. We then sent to each researcher an email asking for the timeline, for a total of 30 emails concerning 191 vulnerabilities, noting that several vulnerabilities were reported by the same author. From the 30 authors that we contacted, despite numerous reminders, we only received answers from 7 authors covering 15 vulnerabilities (7 critical, 8 medium). **Unfortunately, none of the authors were able to remember the exact dates, making building the vulnerability timeline impossible.** The authors were only able to estimate the time between the flaw discovered and the release of the patch. The author of CVE-2016-6258 [4] stated that the associated patch was publicly released 7 days after it was discovered. The 6 other authors, who are very active contributors to the Xen project (with more than 2 reported vulnerabilities for each author), stated that the time between the report and the patch release was 1 to 2 months on average.

Timeline of KVM flaws. KVM is a hypervisor integrated into Linux, so it is difficult to track hypervisor-specific changes in the huge bug report flow of Linux. Therefore, to have statistics on vulnerability timelines, we used Red Hat's bug tracker [5] which provides for some KVM vulnerabilities the exact reported date and the patch release date. We were able to identify 24 vulnerabilities presented in Table 1. From the extracted data, we observed that the average duration of the vulnerability window is 71 days, with 60% of the 24 vulnerabilities having a vulnerability window higher than 60 days. The maximum vulnerability window was 180 days (for CVE-2017-12188) and the lowest 8 days (for CVE-2013-0311).

3 HYPERVISOR TRANSPLANT

We first present the two main principles, *Memory separation* and *Unified Intermediate State Representation* (UISR) used in designing *HyperTP*. We then describe in detail the implementations of *InplaceTP* and *MigrationTP*.

3.1 Design principles

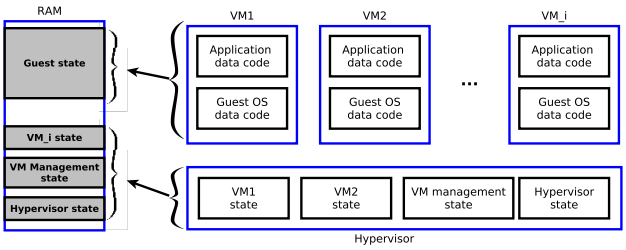

Figure 2: Memory separation organizes the content of the RAM in four categories: *Guest State*, VM_i *State, VM Management State* and *HV State.*

Although the document describes the utilization of *HyperTP* with two hypervisors, we underline that the datacenter operators can have several hypervisors in their repertoire, thus increasing the chance to find a safe replacement hypervisor when several hypervisors are vulnerable at the same time, whether to the same or different flaws. Let us note $H_{current}$ and H_{target} as the current and substitute hypervisor of the datacenter, respectively. Hypervisor transplant works as follows: (1) Suspend running VMs; (2) Translate

VM states into the UISR neutral format; (3) Transfer VM states to the replacement server in case of *MigrationTP*, or micro-reboot into H_{target} in case of *InPlaceTP*; (4) Translate VM states back from UISR to the H_{target} format; (5) Resume VMs; and (6) cleanup. We consider that the VMs' states include all the data structures in the hypervisor for the management of virtual resources (CPUs, memory, devices).

Memory separation.
The purpose of memory separation is to identify inside the RAM state which kinds of data are specific to VMs or the hypervisor, and which memory contents need (or not) to be translated and restored into the H_{target} hypervisor. This allows to reduce translations and therefore the necessary downtime. In addition, *HyperTP* manages to keep hypervisor-independent data in-place in case of *InPlaceTP* in order to accelerate the transplant process.

Fig. 2 summarizes the RAM organization in a virtualized system, which can be classified into four categories:

- *Guest State* corresponds to all the memory managed by the guest (its address space), including the guest operating system and its hosted applications. This memory is hypervisor-independent and can be kept untouched. It stays as is in the case of *InPlaceTP* or is transfered to the destination server in the case of *MigrationTP*.
- VM_i *State* corresponds to all the data structures which are specific to the execution of one VM_i by the hypervisor. Nested page tables (NPT) or vCPU contexts are examples of such data structures. This memory is mostly hypervisor-dependent, and therefore has to be translated and restored during transplant. For instance, the structure and the content of the NPT is enforced by processor vendors, yet is hypervisor-dependent as each hypervisor has its own NPT management policy.
- *VM Management State* corresponds to the data structures which are used for VM management and include references to VM_i *State*. An example is the queues from a vCPU scheduler. This state is hypervisor-dependent, but *HyperTP* does not have to translate it as it can be reconstructed from the VM_i *State* of all the VMs.
- *HV State* corresponds to the memory managed by the hypervisor which is not linked with any VM. Such state is reinitialized by the micro-reboot in the case of *InPlaceTP*, or already exists on the destination server in the case of *MigrationTP*.

Overall, the main state handled by *HyperTP* is VM_i *State*.

Unified Intermediate State Representation.
UISR allows translating the hypervisor-dependent state of each VM (VM_i *State*) into a hypervisor-independent intermediate state representation. *UISR* shares the same objectives as XDR [49], a network neutral data representation. Relying on a neutral format simplifies the re-engineering of a hypervisors into a *HyperTP*-compliant one, since the hypervisor developer only has to understand the UISR format instead of the representation formats of all existing hypervisors.

UISR describes the structures of a VM which are necessary to restore it in any hypervisor. It includes the VM state information for all virtualized resources (CPU, memory, I/O devices ...). *UISR* construction in *HyperTP* is done using `struct uisr* to_uisr_xxx` functions (e.g., `struct uisr* to_uisr_vCPU`). The restoration from *UISR* representation into H_{target}'s format is achieved using `void *from_uisr_xxx` functions (e.g., `void *from_uisr_vCPU`), which return the address where the corresponding state representation is stored. The implementation of save and restoration functions must be done by an expert of each hypervisor since they rely on each hypervisor's internal APIs.

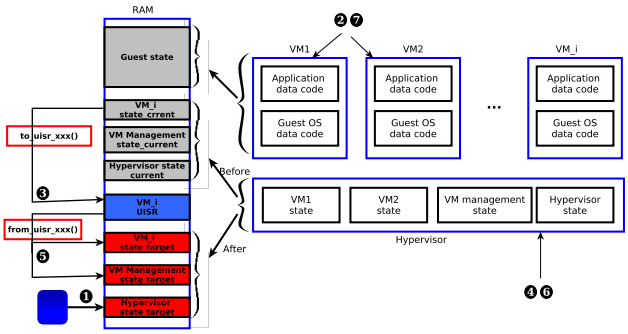

Figure 3: InPlaceTP workflow.

3.2 In-place hypervisor transplant

Fig. 3 summarizes the general workflow of *InPlaceTP*. The binaries of H_{target} are loaded ahead of time into the physical RAM (Fig. 3, ❶). Then any running guest VMs are paused (Fig. 3, ❷). The VM_i *States* of all VMs are translated into the UISR format using UISR functions (see Fig. 3, ❸). A micro-reboot of the machine is then performed using H_{target} as the new kernel entry point (Fig. 3, ❹). After reboot, the target hypervisor restores the VM_i *States* from the UISR into the target format, rebuilds the *VM Management State* (Fig. 3, ❺), and then links the new VM_i *States* for all VMs with the new hypervisor (Fig. 3, ❻). Finally, the target hypervisor resumes all guests (Fig. 3, ❼) and the portions of the RAM which were used to store ephemeral data are freed. Note that *Guest States* (e.g., guest physical address spaces) are kept untouched and in-place. This accelerates the transplant process and minimizes the amount of additional memory resource needed, since *Guest States* represent the largest part of hardware resources consumed by VMs.

3.3 Migration-based hypervisor transplant

MigrationTP mostly follows the same workflow as a normal VM live migration [12], a pre-copy loop (during which the VM continues to run) followed by a final suspend-and-copy phase. Once the suspend-and-copy completes, the source hypervisor signals the destination to resume the VM and complete the live migration. The novel aspect in *MigrationTP* is the introduction of proxies to translate the VMs' VM_i *States* into UISR. The UISRs is built by the proxy on the source machine. On the destination machine, the proxy is responsible for restoring the state into the H_{target} format. Note that *Guest*

States, which are hypervisor-independent, are not translated by the proxies.

4 PROTOTYPE

Although our *HyperTP* prototype allows transplanting a hypervisor host from Xen onto KVM and vice versa, we focus on the implementation of Xen to KVM transplant to stay within the page limit. We organize our description of the prototype as follows: Section 4.1 describes our choice of experimental environment. Section 4.2 presents the implementation of *InPlaceTP*. Finally, section 4.3 describes *MigrationTP* and its differences compared to *InPlaceTP*.

4.1 Experimental environment

We use Xen 4.12.1 in HVM (Hardware-assisted virtualization) mode because of its popularity (e.g. Amazon's recommendation of favoring HVM [6]). [4] On the KVM side, we use Linux 5.3.1 combined with `kvmtool`. Table 3 describes the hardware characteristics of our experimental machines. Following common VM storage design in datacenters, we use network-based remote storage for the VM's root disk. As a result, storage operations are only network dependent.

The *HyperTP* prototype is mostly implemented in user-space. Such an implementation has several advantages: First, it takes advantage of existing libraries (e.g., `libxenctrl` [52], a low-level library for interacting with Xen), thus minimizing the development effort. Second, it allows easy reuse of our prototype across different Xen and KVM versions. Last but not least, it facilitates the usage of *HyperTP* by sysadmins, as we do not require upgrading the current hypervisor.

4.2 In-place Xen to KVM transplantation

The transplantation process follows the steps of the *HyperTP* workflow presented in Section 3.1. The implementation of steps *(1)* (suspension) and *(5)* (resume) is not detailed because they are natively provided by all hypervisors. We detail the implementations of steps *(2)* (translation) and *(4)* (restoration) for each type of virtualized resource: platform (Section 4.2.1), memory (Section 4.2.2) and IO devices (Section 4.2.3). Finally, Section 4.2.4 describes the micro-reboot process used by step *(3)*.

The translation and restoration of a virtualized resource within the VM_i *State* corresponds to the translation of its state (in Xen format) to UISR, followed by the saving of the latter in RAM, the translation from the UISR in RAM to a KVM-understandable format, and finally the integration of this state into KVM. We use a slight modification of Xen's virtual resource state representation as UISR. This choice is motivated by the fact that Xen is open source and mature, developed over a period of over 15 years.

4.2.1 Platform management. *Platform* refers to the CPU and other critical virtual devices necessary for the operation of the guest. Table 2 shows the mapping between hypervisor state types and the UISR platform state (which is part of VM_i *State*).

Platform translation and restoration. Xen already contains the necessary functions for saving and loading VM platform states (`xc_domain_hvm_get/setcontext`). These functions can be directly integrated into *InPlaceTP* as part of the VM save/load process.

Xen HVM state	UISR	KVM
CPU regs	CPU	(S)REGS, MSRS, FPU
LAPIC	LAPIC	MSRS
LAPIC regs	LAPIC_REGS	LAPIC_REGS
MTRR	MTRR	MSRS
XSAVE	XSAVE	XCRS, XSAVE
IOAPIC	IOAPIC	IRQCHIP
PIT	PIT	PIT2

Table 2: Xen-KVM VM state mapping

On the KVM side, we extended `kvmtool` to understand and use UISR states. Upon restoring a VM, the `kvmtool` process is therefore responsible for translating each platform device's state to KVM's internal formats, then calling the corresponding KVM IOCTL to integrate this state into the VM to be restored. During the translation and restoration processes, several platform states required compatibility fixes; for example, Xen uses a 48-pin virtual IOAPIC implementation, compared to KVM's 24 pins. Right now, for experimental purposes, our implementation simply disconnects the higher 24 IOAPIC pins during transplantation (without affecting the applications we experimented with). However, future implementations can ensure that the IOAPIC on both hypervisors are compatible with each other, or expose a way to inform the VM of any change in IOAPIC topology. While the VM states of Xen and KVM are largely similar due to both being based on hardware virtualization, we implemented fixes for specific virtual devices to ensure that these devices will continue functioning on the new hypervisor, for both directions of transplantation (Xen↔KVM).

4.2.2 VM memory management. Our goal is to transform the VM's memory into a universal format that can be understood by multiple hypervisors. This format includes the VM's actual memory content (which we consider *Guest State*), but not hypervisor-specific information stored in VM_i *State* (e.g. nested page tables), which is handled differently by different hypervisors.

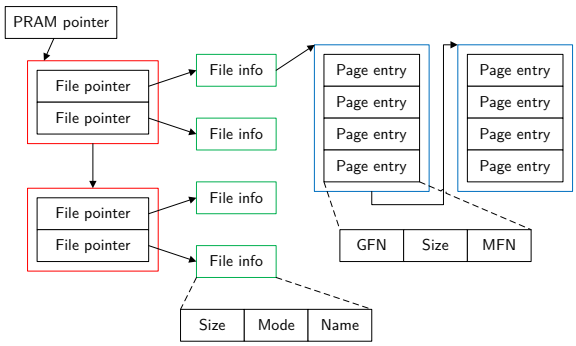

Figure 4: PRAM structure, used for identifying VM's memory pages.

VM memory translation/restoration. In this step, we create a

[4] Xen PV is unsuitable for transplantation between multiple hypervisors due to its tight coupling to the Xen API.

memory map of each VM to be integrated in its UISR. Since a VM's memory does not form a contiguous region in physical RAM, but is rather scattered in different random locations,we use a file system table structure adapted from the *PRAM* patchset [14] to represent this information. Fig. 4 shows the detailed description of our PRAM structure, which consists of metadata pages aligned on a page boundary (to easily manage PRAM's memory). The structure begins with the PRAM pointer, which points to a linked list of root directory pages (colored red in Fig. 4). Each root directory page refers to multiple file information pages (colored green), each of which describe a single VM's memory using a chain of nodes (colored blue) containing page entries. Each page entry itself contains the guest frame number of the chunk, its corresponding machine frame number, and its size (in power-of-2 number of pages) so as to support hypervisor-side large pages. In short, the PRAM structure records the memory contents of a VM as a file. This file can later be used at restoration time to ensure that *Guest State* is kept untouched, and to restore a consistent VM_i *State* (e.g. NPT). For InPlaceTP with Xen→KVM, at restoration time, we simply map the VM's memory into the VMM using `mmap`, and pass the resulting memory address to KVM to be used as guest memory. For KVM→Xen, we implemented a PRAM filesystem API into Xen to allow integration of each VM's memory into the new hypervisor.

4.2.3 IO device management. We considered two types of virtual devices, pass-through and emulated. With pass-through devices, each VM has direct access to the hardware device at near native performance, but it forbids the use of VM migration. With emulated devices, the VMM (QEMU or `kvmtool`) provides a hardware device using the trap-and-emulate technique to simulate a real hardware device or to provide a paravirtualization API. One noticeable difference between these two types of devices is that with pass-through devices, the (hardware) device is still the same after transplantation; whereas with emulated devices, the emulation software may change with the hypervisor replacement. In both cases, we rely on notifying the guest to prepare before the transplantation, similarly to what is done on Azure with the Scheduled Events API [36].

In the case of pass-through devices, we request the guest driver to pause the device, therefore putting both the device and its driver into a consistent state. Afterwards, as the state of the driver is stored inside the guest memory (*Guest State*), it is preserved during transplantation; the restoration consists of simply notifying the guest to resume the device.

In the case of emulated devices, after pausing the device, we copy and translate its emulation state for use in the target hypervisor. With certain devices (e.g. networking devices), we choose a different approach: notifying the guest to unplug the device before transplantation, then rescanning and reinstalling the device during restoration. We observed that this service interruption has negligible impact on VMs as it does not break existing TCP connections.

4.2.4 Micro-reboot. Both Linux and Xen implement *Kexec*, a method for booting a new kernel on top of a running system. Kexec is analogous to a bootloader which can be invoked at any time, allowing quick booting of new OS kernels without reinitializing most hardware state. We inform the target hypervisor of any existing VM memory maps by passing the PRAM pointer through the target's boot command line. The target hypervisor reserves any memory pages with PRAM information, and constructs a virtual filesystem containing each VM's memory for later use. Additionally, we implemented logic into both Xen and KVM to ensure that the VM memory regions managed by PRAM are not accidentally erased or modified during the transplant process.

4.2.5 Optimizations. In order to optimize the transplantation process, we implemented the following four techniques:

- *Preparation work without pausing the guest.* Some of the translation works can be performed before pausing the VMs, akin to the pre-copy step during live migration.
- *Parallelization.* We parallelized VM_i *State* translations and restorations, where each VM is translated by a different thread. This offers a significant reduction on the downtime and transplantation time, especially during PRAM handling.
- *Huge page support.* We adapted the PRAM patchset to support huge pages (2MB). This reduces the memory footprint of the PRAM structures, and increases performance when reading/writing PRAM structures.
- *Early restoration.* In order to speed up the restoration process, we adapted Linux/KVM so that VM creations and restorations can be initiated earlier, as soon as all services used by KVM VMs have started.

4.3 VM migration-based Xen-to-KVM transplantation

The implementation of *MigrationTP* is almost the same as *InPlaceTP* presented above, with two significant differences: *(1)* UISRs are not saved in the source machine's RAM, but are rather sent over the network to the destination machine with the target hypervisor; and *(2)* only parts of the UISR information have to be transferred, as described below.

VM state transfer from Xen to KVM follows the same workflow as a traditional live migration between two Xen hypervisor hosts. To minimize VM downtime during migration, Xen implements the pre-copy approach [12], which consists of two main phases: *Pre-copy*, where memory pages of the VM in the source host are copied/sent to the target in a loop; and *Stop-and-copy*, where the target hypervisor finalizes the memory copy process and resumes the VM.

To implement *MigrationTP*, we adapted the transfer of VM_i *State* so that they are translated and restored following the client-server design presented in Section 3.3. The translation is otherwise identical to *InPlaceTP*. Note that PRAM is not necessary with *MigrationTP*, as guest pages are copied to the target VM and memory maps are implicitly rebuilt.

4.4 Impact of HyperTP on hypervisors' trusted computing base

The Trusted Computing Base (TCB) of a system is defined as the minimum set of components that must be trusted in said system. In the context of virtualization, Zhang et al. [58] defines the TCB as both the hypervisor and the management VM, the total of which is in the scale of millions of LOCs.

HyperTP contributes a comparatively minimal amount of code, totaling 15 KLOCs in size. Of these 15 KLOCs, 2.2 KLOCs belong to

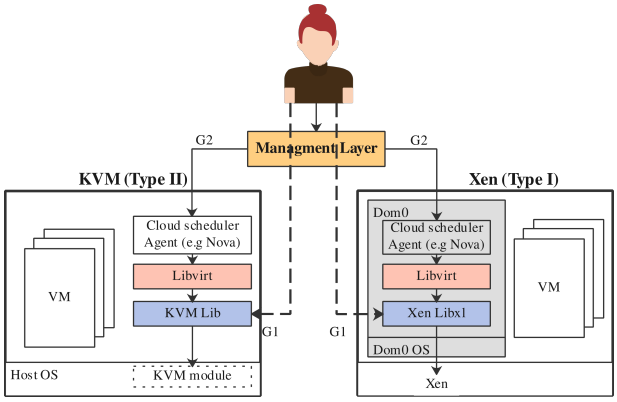

Figure 5: Overview of a KVM/Xen server's components. A dotted line indicates when the sysadmin directly uses a hypervisor vendor's library.

the hypervisors, 5.2 KLOCs are added in userspace management tools (libxl for Xen, kvmtool for KVM, as well as PRAM/Kexec code), 1.1 KLOCs are for HyperTP orchestration purposes, and finally 6.1 KLOCs are for testing, utilities and evaluation. Thus, 8.5 KLOCs of which contribute to *HyperTP*'s TCB, of which nearly 90% is situated in userspace. In addition, *HyperTP* code is activated **only** during the transplant process. It is furthermore isolated between different VMs, and does not process VM inputs. In conclusion, *HyperTP* does not considerably increase the TCB of current hypervisors. Moreover, *HyperTP* mainly relies on existing functionalities in hypervisors. Therefore, we believe that *HyperTP* presents a minimal increase in attack surface.

4.5 Integration in datacenters

HyperTP requires the administration of several hypervisors in the same datacenter. We show in this section that a slight adaptation of the cloud orchestrator is sufficient to easily integrate *HyperTP* into a datacenter. Note that the utilization of several hypervisors in the same datacenter through an abstraction layer is well known, as seen in existing products (OpenStack, Nutanix [42]) or research (Xen-Blanket [55]).

4.5.1 Potential administrative burden of HyperTP. To answer this question, we contacted and analyzed the administration practices of several cloud orchestrator products and cloud service providers (Apache CloudStack, OpenNebula, Proxmox VE and OpenStack). These products were chosen because of their support for multiple hypervisors.

We classify each cloud orchestrator's interactions with the hypervisor into two categories: (G1) those that directly use specific hypervisor libraries (e.g., `xl` for Xen) and (G2) those that call the hypervisor via a generic VM management library (e.g., `libvirt`). We found that all cloud orchestrators interact with the hypervisor using (G2), namely `libvirt`.

Concerning cloud providers practices, we contacted three large scale commercial clouds (3DS Outscale[5], cloud A and cloud B[6]) and Chameleon[7], a cloud-scale datacenter dedicated to researchers. Out of these providers, 3DS Outscale, Cloud A and parts of Chameleon are virtualized with KVM. Outscale uses a homemade cloud orchestrator[8], Cloud A uses a commercial cloud orchestrator, while Chameleon relies on OpenStack for its operations. Cloud B is virtualized with VMware ESXi and is orcherstrated by vCenter. We asked each operator to provide us the ratio of administration tasks their sysadmins perform using (G1) and (G2). We found that no sysadmin in these clouds use (G1). For example, 3DS Outscale stated that: "*In a large scale cloud, the daily work of sysadmins must limit actions on hypervisors as much as possible, because they are very time-consuming, with a high risk of error and difficult to generalize.*" 3DS Outscale expressed a reservation for small scale clusters, in which direct interactions with the hypervisor could make sense. However, our conversation with Chameleon's operators revealed that even in a small scale cloud, sysadmins do not use (G1). In particular, the chief sysadmin of Chameleon stated that "*I use OpenStack, I don't directly operate the hosts' hypervisors. The Nova service is configured to use libvirt, and the rest of the OpenStack system manages the rest.*"

Following our inquiries, we can reasonably say that the integration of *HyperTP* into the datacenter only requires the adaptation of the cloud orcherstrator, without requiring sysadmins to administer each hypervisor type separately.

4.5.2 Integration onto OpenStack. We want to automate the changing of hypervisors into a "one-click" procedure that can be quickly deployed by cloud administrators. We do this using existing mechanisms built into OpenStack Nova. Namely, Nova's API already include the necessary functionalities for managing the host and its guest VMs, including rebooting the host, reconfiguring network, managing shared storage, etc. In this instance, we can add an additional "host live upgrade" operation that performs the operations belonging to the *HyperTP* workflow.

In particular, the addition of *HyperTP* to OpenStack spans the following components. *(1)* Extend Nova's ComputeDriver interface to add *HyperTP*-related operations: guest state saving (akin to the existing suspend operation), loading and executing the new hypervisor kernel, and guest state restoring (akin to the existing resume operation). *(2)* Implement *HyperTP* into each compute driver (libvirt, Xen interface, KVM interface in Fig. 5), so that they support the previously-mentioned *HyperTP* operations. *(3)* Extend Nova's compute API interface to enable automatic upgrading of a host using *HyperTP* (e.g. similar to the existing Evacuate API): all VMs not supporting HyperTP are migrated away from the affected host using the existing ComputeDriver's live_migration operation, Nova manager saves all VM on affected host using the new guest state saving operation, Nova manager triggers host upgrade and updates its internal database to reflect the change in hypervisor, and Nova manager waits for successful host upgrade and restores all VM onto the newly-upgraded host; *(4)* Extend Nova scheduler with additional filters for *HyperTP*-aware consolidation of VMs, by keeping transplantable VMs together so that they can be upgraded with a single operation. *(5)* Update relevant OpenStack interfaces to support calling *HyperTP* API.

[5]https://en.outscale.com/
[6]We didn't obtain the authorization to name these two cloud operators.
[7]https://www.chameleoncloud.org/
[8]https://fr.outscale.com/pourquoi-outscale/tina-os-orchestrateur-cloud/

Server name	Characteristics
M1	Intel(R) i5-8400H 4 cores/8 threads 2.5GHz, 16GB RAM 256GB SSD, 1 Gbps Ethernet
M2	2xIntel(R) Xeon(R) CPU E5-2650L v4 14 cores/28 threads 1.7GHz, 64GB RAM 4x111GB SSD, 1 Gbps Ethernet

Benchmark (metric)	Description
SPECrate 2017 Int/FP (execution time)	23 CPU- and memory-intensive workloads
MySQL 5.7 and Sysbench (QPS and latency)	Stressing a relational database with a SQL load injector
Redis and redis-benchmark (QPS)	Stressing an in-memory key-value store with its included load injector
Darknet (iteration time)	Training a neural network using the MNIST dataset

Table 3: Description of the experimental environment.

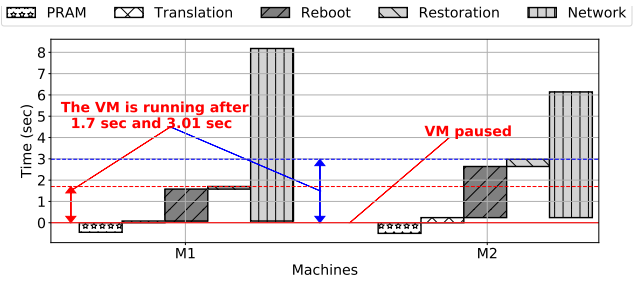

Figure 6: *InPlaceTP* time breakdown on a machine hosting a single VM for Xen→KVM.

	Xen to Xen	MigrationTP (Xen to KVM)
Downtime	133.59 ms	4.96 ms
Migration time	9.564 sec	9.63 sec

Table 4: *MigrationTP* for Xen→KVM and Xen VM live migration - Downtime and migration time.

5 EVALUATIONS

This section presents the performance evaluations of *HyperTP*, including both approaches (*InPlaceTP* and *MigrationTP*). Our evaluations aim to answer the following questions:

- What are the costs incurred by each transplantation step in each approach?
- What is the total transplantation time? This for InplaceTP and MigrationTP for both Xen→KVM and KVM→Xen.
- What is the downtime imposed on VMs by each approach?
- What is the performance gain from each optimization?
- What is the performance impact of each approach on user VMs?
- How scalable is each approach with varying sizes and number of VMs?
- What is the overhead of *HyperTP*?
- How does *HyperTP* perform at the cluster scale?

5.1 Experimental setup

Hardware. To realize the micro-evaluations, we use two kinds of machines: two M1 machines and one M2 machine, whose characteristics are presented in Table 3. We carried out *InPlaceTP* experiments on both machine types, while *MigrationTP* is only performed on M1 machines because live migration requires homogeneous hardware. The two M1 machines are linked with a 1 Gbps Ethernet connection. The software versions used in these machines are presented in Section 4. We reserved 2 CPUs for the administration OS (dom0 in Xen and host Linux in KVM). We configured guest OSes to use 2 MB huge pages for memory allocation.

Concerning cluster scale evaluations, we use 10 machines from a public research infrastructure. Each machine has 2x Intel Xeon E5-2630 v3 and 96 GB of RAM. They are interconnected using a 10 Gbps network.

Applications. We aim to evaluate *HyperTP* using several application types, including CPU-, memory- and IO-intensive applications, thus covering typical datacenter workloads. The benchmark list as described in Table 3 includes SPEC CPU2017 [8], MySQL [26], Redis [43] and Darknet [44].

5.2 Time breakdown

Our experiments in Fig. 6 aim to analyze the duration of each phase of *HyperTP*. We focused on Xen→KVM transplantation in this section. We used idle VMs for this evaluation since VM activity does not impact the transplantation time.

For *InPlaceTP*, the time breakdown consists of the PRAM structure construction (noted *PRAM* in Fig. 6), UISR translation (noted *Translation*), server micro-reboot (noted *Reboot*), and UISR restoration (noted *Restoration*). *Reboot* includes the time for booting the new hypervisor plus PRAM structure parsing time during the early boot phase (to preserve memory from guests). In fact, we were not able to measure the time of these steps separately because monitoring tools are not available during the early boot phase. Since PRAM construction is performed before pausing VMs, the downtime for VMs is $Translation + Reboot + Restoration$. Notice that the horizontal axis (time = 0) corresponds to the pause of VMs, therefore *PRAM* is always below this axis. Since network service is not mandatory for all application types, we present its initialization time separately from the overall transplant time (noted $Network$). Therefore, this time will not be counted in the downtime of network-independent applications, such as the SPEC CPU2017 benchmark, but counted for network-dependent applications.

For *MigrationTP*, we show the duration that the VM is paused (also called downtime) and the total migration time. Note that VM live migration has been subject of many works, and that *MigrationTP* and migration between homogeneous hypervisors follow almost the same procedure.

5.2.1 Basic evaluations. In this experiment, the machine runs a single VM configured with 1 GB of memory and 1 vCPU. This VM size is representative of cloud workloads such as Microsoft Azure [13]. With this VM size, our smallest machine (M1) can host up to 12 VMs. This basic scenario allows us to understand the performance of each phase of *HyperTP*. We repeat each experiment 5 times. We present average figures when standard deviation is very low, and box plots otherwise. Except for the scalability experiments (Section 5.2.2), we

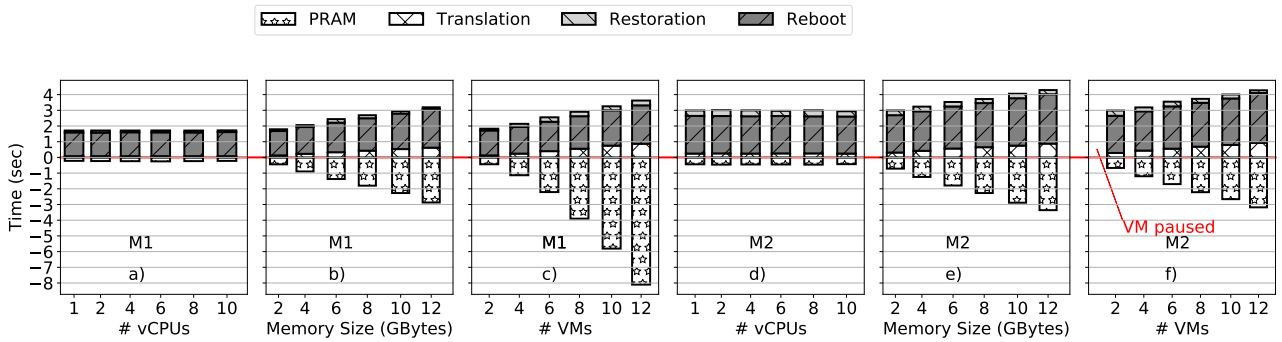

Figure 7: Scalability of InPlaceTP for Xen→KVM transplantation.

focused on Xen→KVM transplantation. Note that only InPlaceTP will lead to different results in the case of KVM→Xen. In MigrationTP Xen→KVM and KVM→Xen are interchangeable.

InPlaceTP: Xen→KVM (Fig. 6). The total transplantation time is 2.15 and 3.56 seconds on M1 and M2 respectively, of which 0.45/0.5 seconds on *PRAM*; 0.08/0.24 seconds are spent on *Translation*; 1.52/2.40 seconds on *Reboot*; and 0.12/0.34 seconds on *Restoration*. *Reboot* is evidently the dominant step of the process, representing respectively 71% and 69% of the total transplantation time on M1 and M2. The downtime is 1.7 seconds on M1 and 3.01 seconds on M2. When networking is taken into account, the process takes 8.1 seconds on M1 (of which 6.6 seconds is spent waiting for the network card) and 5.9 seconds on M2 (of which network card initialization takes 2.3 seconds). We observe below that these interruptions do not affect the operation of network-intensive applications.

MigrationTP: Xen→KVM (Table. 4). We demonstrated the results of live migration between two Xen hosts to establish a baseline for analyzing the performance of *MigrationTP*. First, we observe that the total migration time is almost the same, about 9.5 seconds (dominated by memory page copies). Second, the downtime in *MigrationTP* is 27× lower than in live migration between two Xen hosts. The reason is that on the destination host, *MigrationTP* uses *kvmtool* which is lightweight compared to Xen.

5.2.2 Scalability. We evaluated each solution while varying the VM size (number of vCPUs and memory size), the number of VMs running on each machine and the transplantation direction, i.e Xen→KVM and KVM→Xen. We only analyzed our results on M1 for readability reasons, but results for M2 are also shown in the figures.

InPlaceTP: Xen→KVM (Fig. 7). Firstly, the number of vCPUs has no impact on the transplantation time (Fig. 7a). Secondly, we observe that the evolution of *PRAM* time on M1 and M2 when varying the number of VMs (Fig. 7c and 7f) shows the benefits of parallelizing the construction of PRAM structures. In fact, the time taken by the *PRAM* step increases much more quickly on M1 than on M2. This is because M1 has fewer cores than M2, and cannot benefit as much from parallelization. Thirdly, *Translation* (the first step above the horizontal axis) slightly increases in respect to the VM size and the number of VMs. This is because this step involves finalizing the PRAM structure before it can be used by the target hypervisor. *Reboot* slightly increases (from 1.55 seconds up to 2.46 seconds for M1) when varying either the VM memory size or the number of VMs (Fig. 7b and 7c). This is due to sequential PRAM structure parsing at early boot. Finally, *Restoration* is quite constant regardless the situation. In summary, thanks to the fact that we build PRAM ahead before pausing VMs, the downtime remains minimal, within 1.7 seconds and 3.6 seconds for M1; and within 2.94 and 4.28 seconds for M2. This result is very promising if we compare them with those obtained by Alibaba [59] (from 0.48 seconds up to 9.8 seconds) for live upgrade of the KVM module, without rebooting the physical machine.

MigrationTP: Xen→KVM (Fig. 8 and 9). Fig. 8 presents the results for the downtime caused by migration. Generally, the downtime for *MigrationTP* is lower than that of Xen→Xen migration because of `kvmtool`'s more efficient stop-and-copy step. Additionally, this downtime increases slightly with increasing numbers of vCPUs, but is impacted minimally by the VM's memory size. We use box plots in the last subfigure because of the high variation in downtime induced by Xen when migrating several VMs at the same time. This variation is explained by the sequentiality of Xen's migration process (which migrates multiple VMs in parallel on the sending side, but not on the receiving side [39]). In particular, the downtime for the first migrated VM will be lower than the downtime of the second VM, etc. *MigrationTP* in comparison offers a constant downtime on each VM.

Fig. 9 presents the total migration time. *MigrationTP* and Xen have almost the same results when migrating a single VM while varying its memory size (the first two curves in Fig. 9). The number of vCPUs does not impact the total migration time while the memory size does because of memory copying. Things are different for multiple VMs as shown on the last curve in Fig. 9. We observe the same behavior as described above, where while *MigrationTP* has a higher median VM migration time, the variance in migration time is far less than that of Xen-Xen migration. Nevertheless, the total migration time on *MigrationTP* is shorter than that of Xen→Xen migration.

InPlaceTP: KVM→Xen (Fig 10). We compare these results with Xen→KVM as presented above (Fig. 7). The main observation is

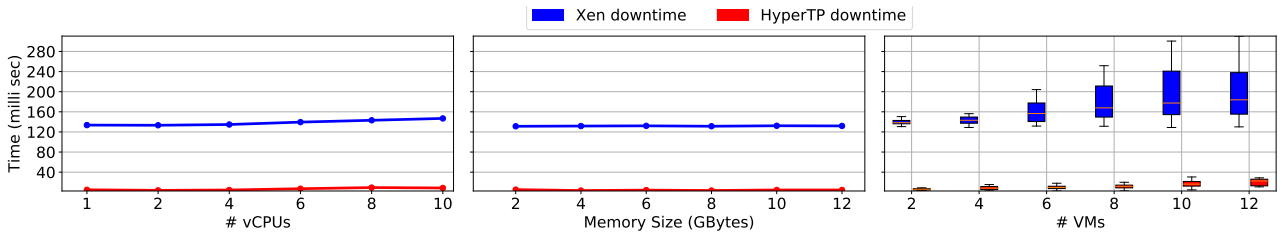

Figure 8: Downtime in MigrationTP for Xen→KVM compared with Xen, which services as a baseline.

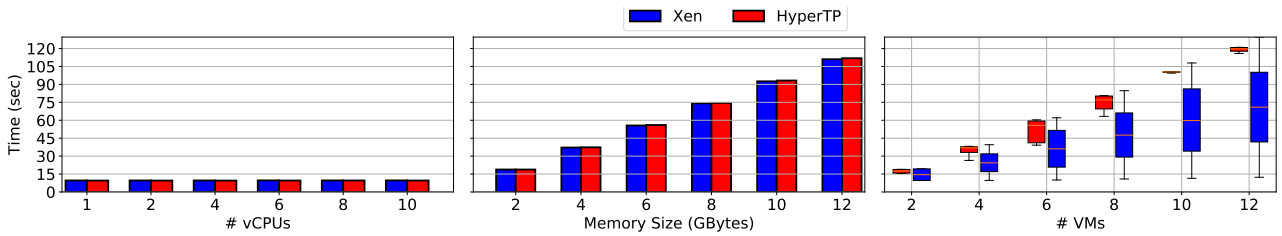

Figure 9: Total migration time in MigrationTP for Xen→KVM and Xen.

that the transplantation time for KVM→Xen is higher compared to Xen→KVM (7.6 sec vs 1.52 sec on M1; and 17.8 sec vs 3.56 on M2). This difference is mainly caused by the boot process of Xen. In fact, being a type-I hypervisor, booting a virtualized machine means launching two kernels: Xen hypervisor and dom0 Linux kernel. However, we note that this time is still far from the 30 sec imposed by Microsoft [36] during maintenance operations.

5.3 Impact on applications

We evaluated *HyperTP* using macro-benchmarks while focusing on Xen→KVM transplantation. The evaluation is done on M1 as follows: each benchmark is launched inside a Xen VM with 2 vCPUs and 8 GB of RAM; at the middle of the execution, we trigger the Xen→KVM HyperTP operation. The metrics provided by the benchmark are used for performance analysis. Since *HyperTP* involves Xen then KVM during the execution of the benchmark, we decided to also show (when applicable) the performance of the application when it performs entirely without transplantation both on Xen and on KVM. This allows to understand the potential performance change after the transplantation process.

Redis. We used the *redis-benchmark* tool included with Redis [43] to stress the Redis server. Fig. 11 shows the evaluation results. For *InPlaceTP* (Fig. 11 left), we can see that the downtime starts at 50 seconds and ends at 59 seconds, representing about 9 seconds. Note that downtime here includes the time needed to reestablish the physical network link on the host, which is done in parallel with the other *InPlaceTP* phases. We can see that Redis continues to perform well after transplantation. However, there is a performance improvement of about 37% which is explained by the efficiency of KVM over Xen for this particular workload. Concerning MigrationTP, as well as Xen→Xen migration, (Fig. 11 right): these results show a "classical" live migration performance pattern, namely a performance drop during the memory copy phase (from the 46th second to the 124th second, or 78 seconds in total), followed by a negligible downtime when the VM is paused, and finally a return to normal performance.

MySQL. We used Sysbench for workload generation. Fig. 12 presents the results. We observe a similar behavior as with Redis. *InPlaceTP* causes service interruption during approximately 9 seconds. *MigrationTP*, as well as Xen, causes a 252% increase in latency and 68% decrease in throughput during the migration, which lasts about 76 seconds.

SPEC CPU2017. We run all the 23 applications of SPECrate from SPEC CPU2017 benchmarks suite. We estimated the performance degradation caused by *HyperTP* as the maximum of the degradation with respect to Xen and KVM, i.e.

$$Degradation = max(\frac{HyperTP - Xen}{Xen}, \frac{HyperTP - KVM}{KVM})$$

Table 5 presents the obtained results. The maximum degradations are therefore respectively 4.19% and 4.81% for *InPlaceTP* and *MigrationTP*. This impact comes from not only the transplantation process itself, but also from the native performance difference between Xen and KVM. Indeed, we can see that these benchmark applications do not have the same performance in both hypervisors, see Xen and KVM columns in Table 5. Note that since the degradation caused by *HyperTP* is quite constant, its percentage on applications with longer execution time (in the range of hours, e.g., scientific simulations) will be invisible.

Darknet. We ran Darknet [44] to train a neural network on MNIST data-set which requires 100 iterations. We recorded the duration of each iteration, as reported in Table 6. The average iteration duration when no migration hypervisor update operation is performed is about 2.044 seconds; this time rises to 4.97 seconds during one single

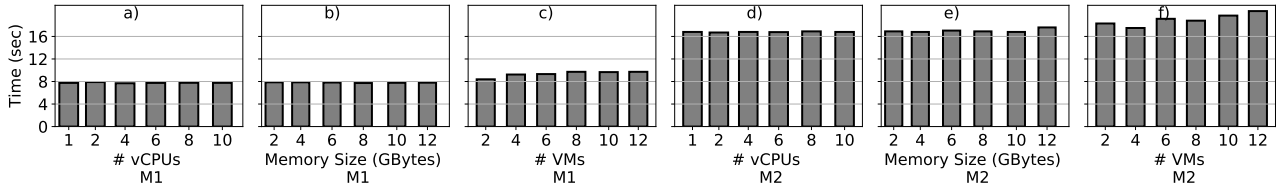

Figure 10: Scalability of InPlaceTP for KVM→Xen.

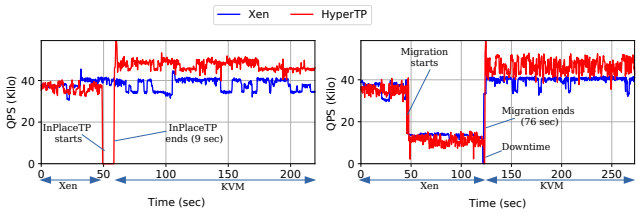

Figure 11: Redis evaluation with InPlaceTP (left) and MigrationTP (right).

Benchmarks	KVM	Xen	InPlaceTP		MigrationTP	
	Time (sec)	Time (sec)	Time (sec)	Deg (%)	Time (sec)	Deg (%)
perlbench	474.31	477.39	475.42	0.23	474.43	0.02
gcc	345.92	346.24	355.71	2.83	353.54	2.20
bwaves	943.96	941.36	950.29	0.95	944.23	0.30
mcf	466.78	465.83	472.40	1.41	468.57	0.59
cactuBSSN	323.78	325.74	334.54	3.32	334.31	3.25
namd	308.77	310.58	312.53	1.22	311.59	0.91
parest	663.50	666.87	668.50	0.75	669.85	0.96
povray	558.38	550.73	562.22	2.09	567.80	3.10
lbm	308.55	306.27	312.03	1.88	315.97	3.17
omnetpp	557.65	560.94	562.05	0.78	566.54	1.59
wrf	650.81	686.62	655.27	0.68	655.90	0.78
xalancbmk	496.66	488.86	497.06	1.68	491.24	0.49
x264	630.68	634.67	632.37	0.26	631.60	0.15
blender	457.93	456.97	461.57	1.01	457.51	0.12
cam4	539.63	569.20	545.21	1.04	549.78	1.88
deepsjeng	456.65	457.75	475.80	4.19	476.27	4.30
imagick	707.99	712.16	712.25	0.60	721.41	1.90
leela	738.87	741.29	741.51	0.36	741.12	0.30
nab	554.47	570.73	557.18	0.49	557.53	0.55
exchange2	580.84	578.83	582.25	0.59	581.99	0.55
fotonik3d	405.29	398.53	415.84	4.34	417.69	4.81
roms	432.87	442.74	443.10	2.36	449.22	3.78
xz	530.10	527.98	546.68	3.54	537.98	1.89

Table 5: Impact of *InPlaceTP* and *MigrationTP* on SPECrate 2017 benchmarks.

iteration for *InPlaceTP*, because the VM is paused during transplant. In comparison, *MigrationTP*'s downtime does not have as much of an impact on Darknet, with the longest iteration lasting only 2.24 seconds. This is better than the case of Xen→Xen migration, with around 2.67 seconds for the longest iteration. Note that with the long execution time of the training phase of AI workloads (in the range of hours), the impacts of *InPlaceTP* and *MigrationTP* are negligible.

Default	Xen migration	InPlaceTP	MigrationTP
2.044 sec	2.672 sec	4.970 sec	2.244 sec

Table 6: Average duration of Darknet training iterations with *InPlaceTP* and *MigrationTP*. Default means neither migration nor transplantation has been performed.

5.4 Cluster migration

We evaluated the time taken for upgrading a cluster using the migration method. We used the BtrPlace VM scheduler framework [20] to define the structure of a simple server cluster including 10 physical hosts. On each hypervisor host, we ran 10 VMs each with 1 vCPU and 4 GB of RAM. In this group of VMs, we configured 30% to run as a video streaming server (each with a matching client running outside of the cluster); 30% running a CPU- and memory-intensive benchmark; and the remaining 40% being idle. We then simulated an upgrade event by dividing the cluster into smaller groups, sequentially putting each group offline using BtrPlace's constraints (the VMs from the offline group are placed in other groups), and then recording the resulting migration plans. For the aforementioned cluster, BtrPlace generated a migration plan with a total of 154 VM migration operations.

We repeated the same experiments while varying the percentage of VMs that are *InPlaceTP* compatible. Figure 13 shows the number of migrations and reduction in total migration times for varying proportions of *InPlaceTP*-compatible VMs. We observe that increasing the proportion of in-place *InPlaceTP*-compatible VMs reduces the number of migrations necessary to upgrade the cluster, and therefore the total migration times. With 20% *InPlaceTP*-compatible VMs, the migration plan required 109 migrations, corresponding with 17% shorter total migration duration. With 60% *InPlaceTP*-compatible VMs, it took 73% fewer migrations and 68% less migration time, and with 80% *InPlaceTP*-compatible VMs, it required only 25 migrations, resulting in a reduction in total migration time of almost 80%. Coupled with the fact that hypervisor host upgrades using *InPlaceTP* takes only seconds to complete, these results show how *HyperTP* can substantially speed up a hypervisor cluster's upgrade process.

5.5 Memory overhead

The memory overhead of *HyperTP* involves the extra memory required for storing PRAM structures and virtual resources in UISR formats (only the latter in the case of MigrationTP). Fig. 14 presents our overhead measurements for various transplantation scenarios as explored in Section 5.2.1. We can see that the memory footprint of PRAM structures increases with the total memory size of VMs, from 16 KB (for a single 1 GB VM) up to 60 KB (for a 12 GB VM). In the case of multiple simultaneously-running VMs, the overhead increases slightly due to additional file info and metadata pages necessary for separate VMs (see Figure 4); however, these overheads remain minimal at only 148 KB for 12 VMs with 1 GB of RAM each. More generally, PRAM structures consist of 8-byte records for every VM's memory page (which can be 4K or 2M in size) in the worst case, leading to an overhead of 2 megabytes of metadata

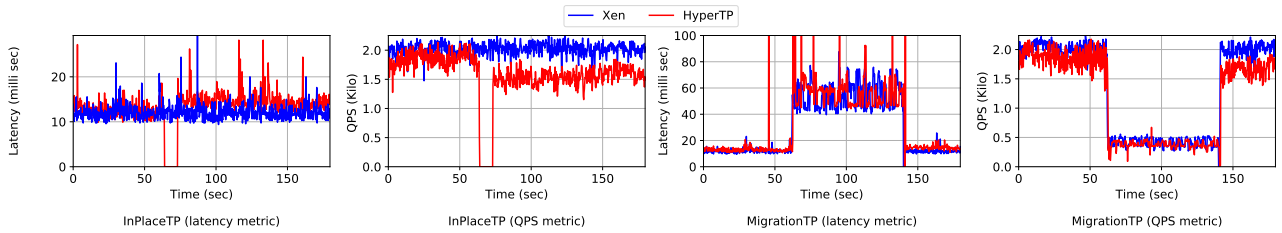

Figure 12: Impact of *InPlaceTP* and *MigrationTP* on MySQL. The first two figures respectively present request latency and Queries-Per-Second (QPS) metrics for *InPlaceTP*. The last two figures show the results for *MigrationTP*.

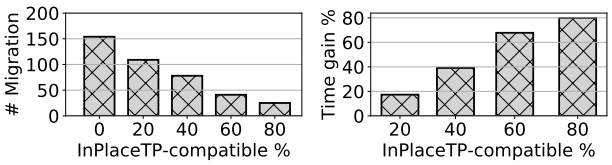

Figure 13: Evaluating *HyperTP* improvement on a real cluster update process: a) impact on the number of migrations and b) impact on the total update time.

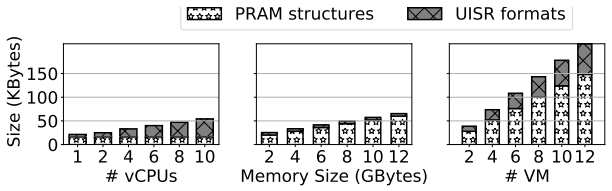

Figure 14: UISR memory size of InPlaceTP and MigrationTP.

per GB of guest memory (in the case of all-4K guest pages), or 4 KB per GB of guest memory (in the case of all-2M guest pages).

Concerning the memory footprint of virtual resources in UISR formats, they increases with the total number of vCPUs, from 5 KB with 1 vCPU up to 38 KB with 10 vCPUs. In summary, the total memory overhead of HyperTP varies from 21 KB up to 98 KB per VM, which is negligible. Notice this extra memory is given back to the hypervisor after the transplantation or migration process.

6 RELATED WORK

We classify hypervisor protection strategies in four categories: preventive (stopping attacks by design), corrective (applying updates), reparative (restoring consistency), and defensive (protecting during the vulnerability window).

Preventive approaches, *hardening the hypervisor.* Many research works advocate a micro-kernel architecture for the hypervisor in order to: (1) reduce the Trusted Code Base (TCB), thus reducing the attack surface [31, 37, 45, 46, 48, 50, 58], (2) formally verify this TCB to prove the absence of known classes of vulnerabilities [24] and (3) isolate buggy or untrusted device drivers of the hypervisor [37, 46, 58]. Although very interesting, this approach imposes a strict implementation of a micro-kernel architecture which requires considerable efforts in the design of the hypervisor. In addition, most of the contributions in this approach require hardware changes that are not yet available [47]. Moreover, no implementation is 100% sure and such an approach has anyway to be combined with regular security updates as studied in the next section.

Corrective approaches, *applying updates.* The common way to protect a hypervisor is to carry out regular security updates. The easiest way to update the hypervisor is kernel live patching [7, 10, 40]. The latter is a lightweight solution to apply simple temporary patches to a running kernel. Unfortunately, it does not support patches that may change persistent data structures (i.e., data structures which have allocated instances in the kernel heap or stacks). When simple patches are not sufficient, VM live migration or in-place hypervisor update (with server reboot) should be used.

Live migration. VM live migration [12, 34] allows the cloud provider to upgrade almost everything on the origin server (because it no longer runs VMs), from hardware devices to the hypervisor. Several works [17, 22] have investigated downtime reduction during live migration. To our knowledge, Liu et al. [33] is the only work which studied VM migrations between heterogeneous hypervisors as *HyperTP*. It was not possible for us to quantitatively compare our MigrationTP solution with Liu et al. [33] because no public prototype exists. From the design perspective, our UISR principle facilitates the integration of new hypervisors, making *HyperTP* generic. Finally, *HyperTP* combines live migration with in-place hypervisor transplantation to address the scalability limitation of the former [59].

In-place hypervisor update. Xiantao Zhang et al. [59] introduces *Orthus* which targets KVM update, including both the emulator user-space software (qemu) and the kvm kernel module, with minimal downtime. Orthus modifies the kvm module to incorporate state-transition capabilities between two consecutive versions, coupled with a lightweight mechanism to checkpoint/restore VMs. *Orthus* is dedicated to KVM, thus does not target heterogeneous hypervisors as *HyperTP*. Second, *Orthus* does not target the update of the entire kernel, which explained their very low downtime (0.48-9 seconds). Other research works such as [18, 28, 57] used nested virtualization to enable quick and transparent in-place updates. Nothing is said about the update of that low level hypervisor. In fact, they just transfer the problem to the latter. *HyperTP* does not include this limitation.

Reparative approaches, *consistent state restoration.* These mainly rely on fast reboot and restoration, and can be implemented at the OS or hypervisor level [9, 11, 16, 23, 27, 29, 30, 35, 56, 60]. Works such as [16, 23, 27, 29] proposed fast loading of a new OS or hypervisor to answer to a system failure. Otherworld [16] restores applications running on a kernel in the event of a crash by booting a previously-loaded second kernel image, and restoring the application from main memory. The authors of [27, 29] went in the same direction with hypervisors by saving the states of VMs in memory and restoring them to a new loaded hypervisor on the same server. Frederico Cerveira et al. [9] proposed to respond to hypervisor corruption by migrating VMs over the same physical host instantly and with no overhead, by avoiding memory copy and taking advantage of Intel EPT's inner workings.

Defensive approach, *during vulnerability windows.* The above approaches have the same limitation which is the fact that **they cannot ensure the hypervisor safety against a flaw as long as the security patch is not available**. As we highlight in Section 1, several days may go before the security patch is available. *HyperTP* allows protecting a datacenter hypervisor during the vulnerability window (as long as the flaw is not impacting one other hypervisor) with very little downtime. As far as we know, *HyperTP* is the first system to address this issue. To facilitate the utilization of *HyperTP*, we build it using existing approaches including VM live migration and server micro-reboot. We apply the latter on heterogeneous hypervisors.

7 CONCLUSION

We introduced *hypervisor transplant*, an idea for minimizing the vulnerability window of critical flaws by temporarily replacing the current hypervisor by a different one which is not sensitive to the vulnerability. We instantiated this idea with *HyperTP*, a framework which combines two hypervisor transplantation solutions, namely *InPlaceTP* and *MigrationTP*. We described a working prototype for transplanting Xen with KVM. We thoroughly evaluated *HyperTP* using well know benchmarks. The results demonstrate the viability of our solution, which introduces a negligible overhead.

ACKNOWLEDGEMENTS

This work is supported by the French National Research Agency (ANR-20-CE25-0005) and Région Occitanie under the Prématuration-2020 program. Some experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr/). Thanks to Mar Callau-Zori and Christophe Hubert from 3DS Outscale, Cody Hammock from Chameleon Cloud, Simon Delamare from Grid'5000 and the Xen security team for your insights. We would also like to thank our shepherd Yubin Xia and the anonymous reviewers for their helpful feedbacks.

REFERENCES

[1] 2017. Information leak using speculative execution. https://bugs.chromium.org/p/project-zero/issues/detail?id=1272.
[2] 2018. Keeping Spectre Secret. https://www.theverge.com/2018/1/11/16878670/meltdown-spectre-disclosure-embargo-google-microsoft-linux.
[3] 2021. Advisories, publicly released or pre-released. https://xenbits.xen.org/xsa/.
[4] 2021. CVE-2016-6258 description. https://nvd.nist.gov/vuln/detail/CVE-2016-6258.
[5] 2021. Red Hat Bugzilla. https://bugzilla.redhat.com/.
[6] Amazon Web Services, Inc. [n.d.]. Migrating to latest generation instance types. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/virtualization_types.html.
[7] Jeff Arnold and M. Frans Kaashoek. 2009. Ksplice: Automatic Rebootless Kernel Updates. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*. Association for Computing Machinery, New York, NY, USA, 187–198. https://doi.org/10.1145/1519065.1519085
[8] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE '18)*. Association for Computing Machinery, New York, NY, USA, 41–42. https://doi.org/10.1145/3185768.3185771
[9] F. Cerveira, R. Barbosa, and H. Madeira. 2019. Fast Local VM Migration Against Hypervisor Corruption. In *2019 15th European Dependable Computing Conference (EDCC)*. 97–102.
[10] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. 2017. Adaptive Android Kernel Live Patching. In *Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17)*. USENIX Association, USA, 1253–1270.
[11] Cheng Tan, Yubin Xia, Haibo Chen, and Binyu Zang. 2012. TinyChecker: Transparent protection of VMs against hypervisor failures with nested virtualization. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*. 1–6.
[12] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live Migration of Virtual Machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation - Volume 2 (NSDI'05)*. USENIX Association, USA, 273–286.
[13] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 153–167. https://doi.org/10.1145/3132747.3132772
[14] Vladimir Davydov. 2013. pram: persistent over-kexec memory file system. https://lists.openvz.org/pipermail/criu/2013-July/009877.html.
[15] Christina Delimitrou. [n.d.]. The Increasing Heterogeneity of Cloud Hardware and What It Means for Systems. https://www.sigops.org/2020/the-increasing-heterogeneity-of-cloud-hardware-and-what-it-means-for-systems/.
[16] Alex Depoutovitch and Michael Stumm. 2010. Otherworld: Giving Applications a Chance to Survive OS Kernel Crashes. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*. Association for Computing Machinery, New York, NY, USA, 181–194. https://doi.org/10.1145/1755913.1755933
[17] Umesh Deshpande, Unmesh Kulkarni, and Kartik Gopalan. 2012. Inter-Rack Live Migration of Multiple Virtual Machines. In *Proceedings of the 6th International Workshop on Virtualization Technologies in Distributed Computing Date (VTDC '12)*. Association for Computing Machinery, New York, NY, USA, 19–26. https://doi.org/10.1145/2287056.2287062
[18] Spoorti Doddamani, Piush Sinha, Hui Lu, Tsu-Hsiang K. Cheng, Hardik H. Bagdi, and Kartik Gopalan. 2019. Fast and Live Hypervisor Replacement. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2019)*. Association for Computing Machinery, New York, NY, USA, 45–58. https://doi.org/10.1145/3313808.3313821
[19] A. Gkortzis, S. Rizou, and D. Spinellis. 2016. An Empirical Analysis of Vulnerabilities in Virtualization Technologies. In *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 533–538.
[20] Fabien Hermenier, Julia Lawall, and Gilles Muller. 2013. Btrplace: A flexible consolidation manager for highly available applications. *IEEE Transactions on dependable and Secure Computing* 10, 5 (2013), 273–286.
[21] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. 2014. Fine Grain Cross-VM Attacks on Xen and VMware. In *2014 IEEE Fourth International Conference on Big Data and Cloud Computing*. 737–744.
[22] H. Jin, L. Deng, S. Wu, X. Shi, and X. Pan. 2009. Live virtual machine migration with adaptive, memory compression. In *2009 IEEE International Conference on Cluster Computing and Workshops*.
[23] Sanidhya Kashyap, Changwoo Min, Byoungyoung Lee, Taesoo Kim, and Pavel Emelyanov. 2016. Instant OS Updates via Userspace Checkpoint-and-Restart. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '16)*. USENIX Association, USA, 605–619.
[24] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. SeL4: Formal verification of an OS kernel. *SOSP'09 - Proceedings of the 22nd ACM SIGOPS*

Symposium on Operating Systems Principles, 207–220. https://doi.org/10.1145/1629575.1629596

[25] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[26] Alexey Kopytov. [n.d.]. Sysbench: a system performance benchmark. https://github.com/akopytov/sysbench.

[27] K. Kourai and S. Chiba. 2011. Fast Software Rejuvenation of Virtual Machine Monitors. *IEEE Transactions on Dependable and Secure Computing* 8, 6 (2011), 839–851.

[28] Kenichi Kourai and Hiroki Ooba. 2015. Zero-Copy Migration for Lightweight Software Rejuvenation of Virtualized Systems. In *Proceedings of the 6th Asia-Pacific Workshop on Systems (APSys '15)*. Association for Computing Machinery, New York, NY, USA, Article Article 7, 8 pages. https://doi.org/10.1145/2797022.2797026

[29] Michael Le and Yuval Tamir. 2011. ReHype: Enabling VM Survival across Hypervisor Failures. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '11)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/1952682.1952692

[30] M. Le and Y. Tamir. 2012. Applying Microreboot to System Software. In *2012 IEEE Sixth International Conference on Software Security and Reliability*. 11–20.

[31] Min Li, Wanyu Zang, Kun Bai, Meng Yu, and Peng Liu. 2013. MyCloud: Supporting User-Configured Privacy Protection in Cloud Computing. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC '13)*. Association for Computing Machinery, New York, NY, USA, 59–68. https://doi.org/10.1145/2523649.2523680

[32] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[33] Pengcheng Liu, Ziye Yang, Xiang Song, Yixun Zhou, Haibo Chen, and Binyu Zang. 2008. Heterogeneous live migration of virtual machines. In *In International Workshop on Virtualization Technology (IWVT'08*.

[34] Fumio Machida, Dong Seong Kim, and Kishor S. Trivedi. 2013. Modeling and analysis of software rejuvenation in a server virtualized system with live VM migration. *Performance Evaluation* 70, 3 (2013), 212 – 230. https://doi.org/10.1016/j.peva.2012.09.003 Special Issue on Software Aging and Rejuvenation.

[35] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 218–233. https://doi.org/10.1145/3132747.3132763

[36] Microsoft Corp. [n.d.]. Maintenance for virtual machines in Azure. https://docs.microsoft.com/en-us/azure/virtual-machines/maintenance-and-updates.

[37] Derek Gordon Murray, Grzegorz Milos, and Steven Hand. 2008. Improving Xen Security through Disaggregation. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1346256.1346278

[38] MySQL [n.d.]. MySQL. https://www.mysql.com.

[39] Vlad Nitu, Pierre Olivier, Alain Tchana, Daniel Chiba, Antonio Barbalace, Daniel Hagimont, and Binoy Ravindran. 2017. Swift Birth and Quick Death: Enabling Fast Parallel Guest Boot and Destruction in the Xen Hypervisor. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2017, Xi'an, China, April 8-9, 2017*. ACM, 1–14. https://doi.org/10.1145/3050748.3050758

[40] V Pavlík. 2014. kgraft–live patching of the linux kernel. *Technical report, Technical report, SUSE, Maxfeldstrasse*.

[41] Diego Perez-Botero, Jakub Szefer, and Ruby B. Lee. 2013. Characterizing Hypervisor Vulnerabilities in Cloud Computing Servers. In *Proceedings of the 2013 International Workshop on Security in Cloud Computing (Cloud Computing '13)*. Association for Computing Machinery, New York, NY, USA, 3–10. https://doi.org/10.1145/2484402.2484406

[42] Steven Poitras. 2021. The Nutanix Bible. https://nutanixbible.com/.

[43] redis [n.d.]. Redis. https://redis.io/.

[44] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/.

[45] Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. 2009. Secure In-VM Monitoring Using Hardware Virtualization. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*. Association for Computing Machinery, New York, NY, USA, 477–487. https://doi.org/10.1145/1653662.1653720

[46] Le Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. 2017. Deconstructing Xen. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/deconstructing-xen/

[47] Weidong Shi, JongHyuk Lee, Taeweon Suh, Dong Hyuk Woo, and Xinwen Zhang. 2012. Architectural Support of Multiple Hypervisors over Single Platform for Enhancing Cloud Computing Security. In *Proceedings of the 9th Conference on Computing Frontiers*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2212908.2212920

[48] Udo Steinberg and Bernhard Kauer. 2010. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*. Association for Computing Machinery, New York, NY, USA, 209–222. https://doi.org/10.1145/1755913.1755935

[49] Sun Microsystems, Inc. [n.d.]. XDR: External Data Representation Standard. https://tools.ietf.org/html/rfc1014.

[50] Jakub Szefer, Eric Keller, Ruby B. Lee, and Jennifer Rexford. 2011. Eliminating the Hypervisor Attack Surface for a More Secure Cloud. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*. Association for Computing Machinery, New York, NY, USA, 401–412. https://doi.org/10.1145/2046707.2046754

[51] A. Thongthua and S. Ngamsuriyaroj. 2016. Assessment of Hypervisor Vulnerabilities. In *2016 International Conference on Cloud Computing Research and Innovations (ICCCRI)*. 71–77.

[52] David Vrabel, Andrew Cooper, Wen Congyang, and Yang Hongyang. [n.d.]. libxenctrl (libxc) Domain Image Format. https://xenbits.xen.org/docs/unstable/specs/libxc-migration-stream.html.

[53] Vulnerability Metrics [n.d.]. Vulnerability Metrics. https://nvd.nist.gov/vuln-metrics/cvss.

[54] Xiaoyun Wang and Hongbo Yu. 2005. How to Break MD5 and Other Hash Functions. In *Advances in Cryptology – EUROCRYPT 2005*, Ronald Cramer (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 19–35.

[55] Dan Williams, Hani Jamjoom, and Hakim Weatherspoon. 2012. The Xen-Blanket: virtualize once, run everywhere. In *Proceedings of the 7th ACM european conference on Computer Systems*. 113–126.

[56] X. Xu and H. H. Huang. 2015. DualVisor: Redundant Hypervisor Execution for Achieving Hardware Error Resilience in Datacenters. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 485–494.

[57] Hiroshi Yamada and Kenji Kono. 2013. Traveling forward in time to newer operating systems using ShadowReboot. In *VEE '13*.

[58] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. 2011. CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, Ted Wobber and Peter Druschel (Eds.). ACM, 203–216. https://doi.org/10.1145/2043556.2043576

[59] 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 *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 93–105.

[60] D. Zhou and Y. Tamir. 2018. Fast Hypervisor Recovery Without Reboot. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 115–126.