HyperTP: A unified approach for live hypervisor replacement in datacenters

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

HyperTP: A unified approach for live hypervisor replacement in datacenters*

Tu Dinh Ngoc[a,*], Boris Teabe[a], Alain Tchana[b], Gilles Muller[c], Daniel Hagimont[a]

[a] *IRIT, University of Toulouse, CNRS, Toulouse INP, UT3, Toulouse, France*
[b] *ENS Lyon, Lyon, France*
[c] *Inria, France*

Abstract

Maintenance of virtualized datacenters is often needed for the purposes of introducing new features, fixing bugs or mitigating security problems. However, current maintenance methods are either highly disruptive to the operation of VMs, utilize large amounts of computing resources or require high development efforts.

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*). *HyperTP* hinges on a *VM state hierarchy* for organizing different types of hypervisor memory states in terms of their relation to VM execution, and an *Unified Intermediate State Representation* that abstracts VM-relevant memory states between multiple different hypervisors. We describe our implementations of both approaches, including technical details of our UISR design and the transplant process. Our evaluation results show that *HyperTP* delivers satisfactory performance: (1) *MigrationTP* changes a VM's underlying hypervisor while taking the same time and impacting virtual machines (VMs) with the same performance degradation as normal live migration; and (2) *InPlaceTP* imposes minimal VM downtime, even under increasing number of VMs and memory sizes. Finally, we discuss how the combination of *InPlaceTP* and *MigrationTP* can be used to address the challenges of upgrading a hypervisor cluster, and to mitigate known unpatched hypervisor vulnerabilities.

1. Introduction

The increasing usage of virtualization in modern datacenters is accompanied with a simultaneous increase in the need for regular preventative maintenance and

https://doi.org/10.1016/j.jpdc.2023.104733
*Corresponding author.
Email address: dinhngoc.tu@irit.fr (Tu Dinh Ngoc)

updating of hypervisors and related software. Hypervisor updates are often done for one of two reasons: either for introducing new features, or to mitigate a certain vulnerability. These updates involve one of several methods: (1) a full reboot of the host and all running guests; (2) live migration of running VMs from a host running old versions of the hypervisor to another host running updated software; and (3) live patching of the running hypervisor. Each method has its own set of limitations: full reboots are highly disruptive to the infrastructure's operations; live migration consumes large amounts of time and network bandwidth, and prevents the usage of certain virtualization features; live patching is limited to small fixes, mostly of security issues, and requires extra development effort to create a customized livepatch for each fix. Evidently, there remains a need for a timely and efficient hypervisor maintenance system capable of delivering both feature and security updates without causing service disruption or restricting desirable features.

To address this need, we introduce the concept of *hypervisor transplant*. Our goal is to quickly replace one running hypervisor $H_{current}$ with another H_{target} *without rebooting running VMs* for the purpose of speeding up preventative maintenance of virtualization infrastructure. Note that H_{target} can be anywhere from an updated version of $H_{current}$ to a completely different hypervisor, allowing more flexibility in choosing an appropriate hypervisor for the required workload.

We materialized our concept of hypervisor transplant in a platform called *HyperTP* (initially presented in [1]), which combines in a unified way two complementary approaches: *live VM migration-based transplant* (noted *MigrationTP*) which causes almost no VM downtime; *in-place micro-reboot-based transplant* (noted *InPlaceTP*) which replaces a running hypervisor with little downtime and no extra resources. The combination of these two approaches answers the constraints put on both applications running in VMs and on datacenter infrastructures.

Indeed, *InPlaceTP* and *MigrationTP* present a tradeoff between maintenance deadline, downtime tolerance and upgrade resource availability. For instance, *InPlaceTP*'s micro-reboot-based transplant requires several seconds of downtime. However, such downtime figures are not without precedent. Namely, Microsoft Azure presents downtimes of up to 30 seconds for maintenance operations [2]. Orthus [3] reports figures of up to 9.8 seconds for VMM upgrades. Similarly, HyFiX [4] requires 8.1 to 12.3 seconds of downtime for the same task. In exchange for an extended downtime, *InPlaceTP*, by its in-place nature, does not require large amounts of extra resources and significantly shortens the maintenance timeframe. In comparison, *MigrationTP* causes minimal downtime to running VMs but with the additional cost of spare machines and network bandwidth, like other live-migration-based maintenance operations [3]. In the current state of *HyperTP*, it is up to the datacenter operator to decide which transplant approach is the most appropriate for their maintenance operation, since equivalent policies are already provided for dealing with periodic platform updates.

While live migration and micro-reboot are known approaches, the main novelty in designing *HyperTP* is to ease the support of *multiple different hypervisors*. Naturally, this raises the question of managing the heterogeneity of their VM

state representations. To resolve this, we build both approaches of *HyperTP* around two common principles, a *VM state hierarchy* which identifies and defines the various types of memory states in relation to their functionalities relative to a VM's operation, and an *Unified Intermediate State Representation* (UISR) to facilitate the creation of *HyperTP*-compatible hypervisors.

We demonstrated our platform by re-engineering Xen and KVM, the two most popular open source hypervisors, into *HyperTP*-compliant hypervisors. They represent the two types of hypervisors: type-I (Xen) and type-II (KVM), thus demonstrating the scope and flexibility of our solution. We evaluated our prototype at a machine scale to validate its ability to transplant both idle VMs as well as active VMs running various types of benchmarks. We also presented the downtime incurred by *HyperTP* while running various workloads such as SPEC CPU 2017, MySQL and Redis.

We investigated the usage of *HyperTP* at a cluster scale in a datacenter. We highlighted two direct usages of the platform: for *hypervisor updating*, where *HyperTP* shortens the time and reduces the resources needed to apply a new hypervisor version; and for *hypervisor security*, where *HyperTP* helps reduce the time window where a virtualized infrastructure is exposed to known vulnerabilities.

To summarize, in this paper, we present the following contributions:

- We present *HyperTP*, a two-pronged solution including *MigrationTP* and *InPlaceTP* to help simplify hypervisor updates and maintain hypervisor security.
- We implement *HyperTP* in multiple directions: Xen→KVM, KVM→Xen, and Xen→Xen, thus demonstrating *HyperTP*'s scope and flexibility.
- *InPlaceTP* Xen→KVM causes minimal downtime to running VMs (1.91 seconds for a VM with 1 vCPU and 1 GB of RAM), with negligible memory and I/O overhead and without requiring VM reboots. With KVM→Xen and Xen→Xen, the downtime is about 7.8 seconds for the same VM configuration. *MigrationTP* offers similar performance to traditional homogeneous VM live migration.
- We show the benefits of *InPlaceTP* over migration-based solutions for upgrading an existing virtualization cluster. Namely, we demonstrate that upgrading 10 servers each running 10 VMs using *InPlaceTP* for 80% of the VMs takes 3 minutes and 54 seconds while using *MigrationTP* alone would take up to 19 minutes.
- We conduct a study of vulnerabilities in Xen and KVM over the last 7 years. We observe that most vulnerabilities are specific to a single hypervisor and caused by faulty implementations, and show how *HyperTP* can be used to reduce vulnerability windows of virtualized infrastructures.

The rest of the article is organized as follows. Section 2 present the general overview of *HyperTP*. Section 3 presents the implementation of *HyperTP*.

Section 4 presents the evaluation results, followed by Section 5 which provides observations over the limitations of our approach. Section 6 discusses the related works. Finally, Section 7 concludes our paper.

2. Hypervisor transplant

In this section, we first present the two main principles of the design of *HyperTP*, *VM state hierarchy* and *Unified Intermediate State Representation*. We then show how these principles are applied to *InPlaceTP* and *MigrationTP*, and demonstrate their application in our aforementioned use cases.

2.1. Design principles

To reiterate, the main goal of *HyperTP* is to rehost a VM running on one hypervisor to another hypervisor without causing a VM reboot. Let us note $H_{current}$ and H_{target} as the current and target hypervisors of the transplant process respectively. A hypervisor transplant is conducted by performing the following five generic work items:

1. Suspend running VMs;
2. Translate VM states into the UISR neutral format;
3. Transfer VM states to the new target hypervisor;
4. Restore VM states from UISR to H_{target} format;
5. Resume VMs and finish the transplant operation.

Note that the aforementioned workflow is not meant to be taken in strict sequence; we optimize the contents and ordering of this workflow depending on the scenario being executed (*InPlaceTP* or *MigrationTP*). These optimizations are described in the next sections of our paper.

VM state hierarchy. Generally, we consider that the VMs' states include all the data structures in the hypervisor for the management of virtual resources (CPUs, memory, devices). We observe that a VM's in-memory representation consists of multiple types of data, where each data type needs to be translated in a different way. For example, a guest memory page needs to be treated differently from a scheduling object associated to a vCPU. Nevertheless, different hypervisors running on the same platform typically aim to provide a common-ground virtual hardware that accomodates most guest OSes; not to mention, these hypervisors necessarily share some common behaviors by virtue of running on the same architecture. This implies *there exists a commonality between how different hypervisors manage their internal states.*

In *HyperTP*, we propose a hierarchy of memory resources in a VM, which serves to inform us of which kinds of data need to be kept as-is, transformed or discarded. Our resource hierarchy is divided into four main categories:

- *Guest State*, like its name, represents the memory states that are specific and visible to the VM, like memory pages. During transplantation, guest states require the least transformation, i.e. they can stay mostly untouched throughout the whole process.

- *VM_i State* corresponds to data structures that are specific to the execution of one VM, but are not necessarily visible to the VM in their raw form. An example of *VM_i State* are 2D page tables (2DPT) or vCPU register states. In fact, while the structure and content of the 2DPT is usually specific to the hardware virtualization technology being used (e.g. Intel's EPT, AMD's NPT), each hypervisor has its own policies for managing the 2DPT, and therefore the contents of each VM's 2DPT are not directly translatable between hypervisors. Similarly, while vCPU register states are closely linked to the vCPU's execution, each hypervisor saves a vCPU's states in its own different data structure, and therefore these states must be translated if a VM were to be transplanted between different hypervisors.

- *VM Management States* are in-memory states that serve to manage the VM, but does not necessarily contain the VM's state itself. For example, a hypervisor's scheduler queue might refer to a VM's vCPUs, but does not contain any vCPU states. In general, these states can be easily reconstructed if necessary from the previously-mentioned types of state.

- *HV State* finally represents the set of hypervisor states that are not specific to any VM, such as the memory assigned to hardware drivers. *HyperTP* does not save or transform these states; they are considered to be disposable (in the case of *InPlaceTP*, where they are reinitialized using micro-reboot) and/or reconstructable (in the case of *MigrationTP*, where the migration does not take them into account).

Unified Intermediate State Representation. From the hierarchy of VM states presented above, we observe that many types of VM states are at least partially specific to each hypervisor that is managing the VM. Yet, it is unreasonable to demand all hypervisors to use the same data structures for its functionalities in order to support hypervisor transplant, since such standardization not only limits the range of functionalities each hypervisor can potentially support, but also might introduce accidental common vulnerabilities.

To realize the concept of hypervisor transplant, we transform each VM's states into a *Unified Intermediate State Representation* (UISR). UISR represents the hypervisor-dependent state of each VM with a hypervisor-independent intermediate representation that facilitates the transfer of VM states across different hypervisors. In this sense, UISR shares the same objectives as the network-level neutral data representation XDR [5]. 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.

The goal of UISR is to sufficiently represent a VM for its reconstruction in a *HyperTP*-capable, compatible hypervisor. Following the VM state hierarchy presented above, we posit that *knowledge of* Guest State *and VM_i* State *is sufficient for the reconstruction of a VM.* These states are therefore the targets of our UISR. In general, the following VM states are collected by *HyperTP* and distilled into UISR: VM memory pages; CPU registers and control registers;

interrupt controller and timer states; and virtual hardware states (including *hidden states* required to reconstitute the virtual device). However, we acknowledge that the above list is not an exhaustive list of all VM states; the restorability of VMs under *HyperTP* depends on the compatibility of their configuration and the UISR format[1].

To transform a VM's states between its hypervisor-specific representation and UISR, each hypervisor needs to implement a pair of translation functions for each class of VM state. These functions can be as simple as the identity function for hypervisors that directly use UISR as their internal VM states, or an explicit translation from a hypervisor's internal state to the corresponding UISR. Nevertheless, knowing that hypervisors often share certain commonalities (as argued in the above section), we expect most hypervisors to be able to support UISR without needing extensive modifications.

2.2. In-place hypervisor transplant

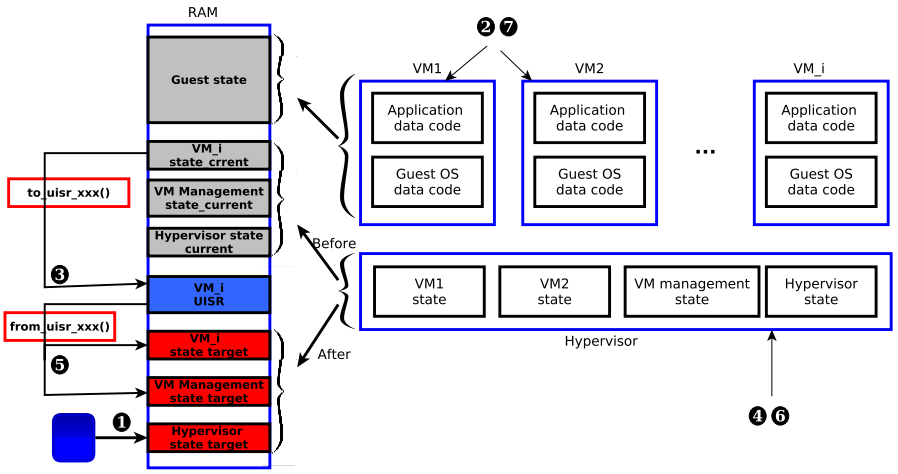

Figure 1: Basic workflow of *InPlaceTP*.

Figure 1 summarizes the working principles of *InPlaceTP*. In short, *InPlaceTP* performs a hypervisor transplant through the means of replacing the running hypervisor. To accomplish this, step ❶ first load the target hypervisor into memory. After pausing any running guest VMs to be transplanted ❷, we invoke the corresponding UISR translation functions to convert the corresponding $Guest/VM_i$ *States* to the UISR format ❸. We perform a micro-reboot to hand over control of the hardware to H_{target} ❹, while passing to it any relevant UISR

[1]Note that this limitation is present in same-hypervisor live VM migration as well; current hypervisors prevent VMs using certain hardware features from being migrated.

Guest/VM_i States. H_{target} then converts the received *Guest/VM_i States* into its own native format ❺, and uses these states to reconstruct the VMs ❻. Finally, the VMs to be transplanted are resumed ❼ and the transplantation process is completed.

For the purpose of *HyperTP*, *Guest States* refer specifically to memory pages owned by the VM and used as its memory. These pages naturally do not require any specific transformation or rewriting to be converted into UISR. As long as these pages remain intact, they can be easily reincorporated into the new VM. That is why during the entire process, *InPlaceTP* ensures that these *Guest States* are protected from accidental deletion and corruption. The steps ❸ and ❺ in fact simply involve recording their location in the host's physical memory, and giving them back to the VM afterwards. This represents a large time saving for *InPlaceTP* as costly memory copies and disk writes are minimized.

2.3. Migration-based hypervisor transplant

Under *MigrationTP*, the target of transplantation is no longer the current server, but rather a remote server, in the same way as normal VM live migration. As such, *MigrationTP* follows the same procedure as VM live migration, which largely matches up with *HyperTP*'s own steps. The differences come during the sending of VM state to the destination server; during this step, *MigrationTP* makes use of *state proxies* to translate the VM's *VM_i States* into UISR. Note that these proxies are based on the same state transformations as used by *InPlaceTP*. On the destination server, another proxy then translates the UISR back into H_{target}'s VM state format. While *Guest States* need to be copied over network to the destination server unlike *InPlaceTP*, this copying step does not need to involve the state proxy.

2.4. Using hypervisor transplant in a datacenter

In summary, *HyperTP* is a combination of two related approaches *InPlaceTP* and *MigrationTP*, based on a common UISR for the representation of VM states. In this section, we elaborate on the application of *HyperTP* to two exemplary use cases: firstly, for installing updates on virtualized infrastructure; and secondly, for the mitigation of hypervisor vulnerabilities.

Hypervisor update. As stated in Section 1, hypervisor updates are used not only to fix system bugs but also to deploy new software features. A public cloud provider might wish to quickly upgrade their hypervisor fleet to add new services and to increase feature velocity of their cloud solution; a private cloud customer might instead want to install a new hypervisor version to maintain software support, or to fix performance and reliability issues, etc. However, existing hypervisor update methods are either disruptive or limited in scope, causing risk-averse operators to hesitate applying updates. In comparison, *HyperTP* offers datacenter operators rapid deployment of new software with *InPlaceTP* in complement with other upgrade choices (*MigrationTP*, normal migration or live patching). Additionally, *InPlaceTP* is not only capable of upgrading the current

hypervisor core, but can even completely replace it with a different hypervisor altogether. This is especially useful e.g. in cases where the operator wishes to switch their hypervisor vendor, where the two different hypervisors can be managed using the same tooling (like OpenStack). In conclusion, *HyperTP* helps facilitate the deployment of desirable updates while minimizing their impact on the infrastructure.

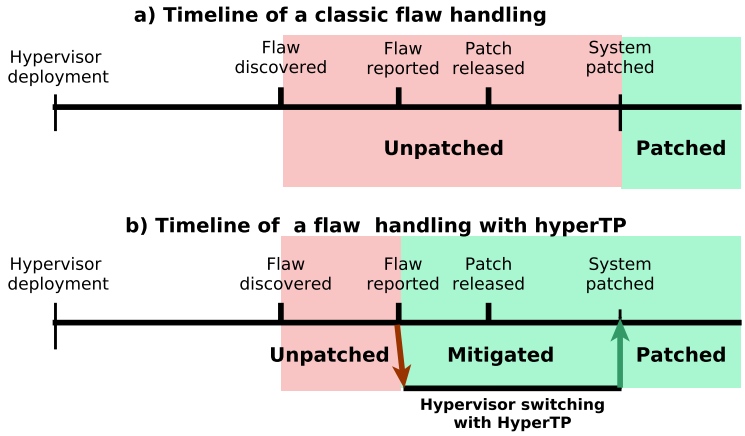

Figure 2: (a) Traditional vulnerability mitigation in data centers and (b) our hypervisor transplantation-based solution.

Hypervisor security. Hypervisors are continuously subject to multiple security vulnerabilities. Similar to previous works [6], we define the *hypervisor vulnerability window* regarding a given security flaw as the time between the identification of said flaw (whether by a good or bad actor) and the integration of a patch in the running hypervisor (see the red zone in Figure 2a). 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. The time to release of a patch is highly dependent on the corresponding vulnerability's severity, and can vary from one week with vulnerabilities such as the MD5 collision attack [7], to 7 months with vulnerabilities such as Spectre and Meltdown[2] [8, 9]. Meanwhile, the time to apply a patch mainly depends on the datacenter operators' patching policies. Together, this timeframe leaves plenty of time to launch an attack against a vulnerable installation.

To alleviate this issue, *HyperTP* can be used to preemptively and temporarily replace the actual datacenter hypervisor (e.g. Xen) with a different hypervisor (e.g. KVM) which is immune to the given vulnerability (see Figure 2b). This approach mitigates the impact of a vulnerability given one of the following condi-

[2]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.

tions is met: (1) there is already a known-safe hypervisor when the vulnerability is discovered, and (2) 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.

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

Year	Xen	KVM	Common
2013	3	3	0
2014	4	1	0
2015	11	1	1
2016	6	3	0
2017	17	1	0
2018	7	2	0
2019	7	2	0
Total	55	13	1

To investigate the viability of hypervisor transplant in this context, we collected a list of critical vulnerabilities over the last 7 years for Xen and KVM (see Table 1). A vulnerability is considered as critical when its Common Vulnerability Scoring System 2.0 score is higher than 7 [10]. Over that period, we found only one common critical vulnerability (CVE-2015-3456) originating from QEMU, a common component used by both Xen and KVM. This low number supports our starting assumption that a safe alternate hypervisor exists. Overall, the number of critical vulnerabilities per year remains low, which means that even if hypervisor transplant cannot be done too frequently, it would still bring an improvement in security.

3. Prototype

We implemented *HyperTP* on top of two commonly-used open-source hypervisors, Xen and Linux KVM. We used Xen 4.12.1 with fully-virtualized HVM domains; Xen PV was not used due to its tight coupling with the Xen API. On the KVM side, we used Linux 5.3.1 along with the standalone *kvmtool*. We implemented *InPlaceTP* in three transplantation directions: Xen→KVM and KVM→Xen as examples of our heterogeneous hypervisor transplant; and finally Xen→Xen as an example for enabling live upgrade of Xen hypervisor instances. KVM→KVM *InPlaceTP* can be implemented with the same principles; however, we did not implement this scenario as it is already covered by existing works [3, 4]. *MigrationTP* was additionally implemented for the Xen→KVM direction. We configured our VMs to use remote storage to concentrate I/O activity onto virtual networking.

Our *HyperTP* prototype represents a total of approximately 8.5 KLOC, of which 2.2 KLOC belong to the hypervisors, 5.2 KLOC in userspace management tools (*libxl*, *kvmtool* and PRAM/Kexec), and 1.1 KLOC for HyperTP orchestration purposes. We based our prototype mostly in userspace; in fact, only

10% of our implementation involves Xen/Linux kernel code. Such a prototype takes advantage of existing tools and libraries for controlling VMs (*libxenctrl*, *kvmtool*), therefore being highly compatible with different versions of Xen and Linux/KVM. Our implementation of *HyperTP* also runs the bulk of its code with minimal privilege and only during the transplant process, thus minimizing *HyperTP*'s security footprint.

In the following sections, we describe the common implementations of VM state management with UISR, as well as our implementations of *InPlaceTP* and *MigrationTP* in detail.

3.1. UISR and device management

When considering only *HyperTP* from Xen to Xen itself, our translation and restoration functions can simply be the identity function. However, when transplanting VMs between Xen and KVM, the need arises for a common UISR format that can be used to represent VM states. Xen provides a relatively stable and well-defined VM state format that covers the majority of VM-critical hardware. As a result, we use a slightly modified and extended version of Xen's VM format as our UISR.

Platform device management. We use the term *platform device* to refer to non-replaceable devices inside the VM that are critical to its execution (CPU registers, interrupt controllers, timers), in contrast to *bulk I/O devices* such as network and storage devices that the guest OS can function without. Naturally, platform devices attract special attention during the hypervisor transplant process, since they are needed for the guest OS to safely resume its operation.

We first established a mapping to be implemented in our state translation functions between each of a VM's platform components on Xen and KVM and our UISR equivalent. Subsequently, we extracted the native VM states of Xen and KVM using the appropriate system calls (using *libxenctrl* on Xen and IOCTLs on KVM). Our translation functions then handled the reading and writing of various VM formats, and at the same time provided fixes to certain platform components that are not mutually compatible between the two hypervisors. Table 2 shows the correspondence between UISR states and native VM states of Xen and KVM in our implementation.

Table 2: Correspondence between hypervisor platform device states and UISR on x86.

Xen HVM	**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

Management of bulk I/O devices. Generally speaking, virtual device functionalities are presented to VMs in one of two main types: device passthrough and device emulation. Our handling of I/O devices depends on the type of device, the details of which are presented below.

Device passthrough grants a VM direct access to hardware installed in its physical server. The VM therefore communicates with hardware using the same driver as a native system would. To prevent the VM from abusing passthrough hardware to attack the host, a I/O memory management unit (IOMMU) is often used to limit that hardware's access to physical memory. Device passthrough gives VMs an I/O performance that is as close as possible to native performance, but also binds the VM to its underlying hardware, and therefore requires special attention in certain cases (e.g. during live migration). In the case of *HyperTP*, we treat passthrough-assigned devices the same way as a live migration event: the VM is informed of the transplantation event and performs the necessary steps to stop its device driver in a consistent fashion. Once the transplantation event completes, the device is reconnected to the guest and resumes its operation.

Emulated devices (including paravirtualized ones) are implemented in software through the use of trap-and-emulate techniques. However, *HyperTP* might lead to a change in the software that is performing the emulation. In our solution, emulated devices can be handled using one of two ways: either by copying and translating the emulation state for use in the target hypervisor (as we've done with our emulated platform devices), or by using the same stop-reconnect technique described above. The stop-reconnect technique has the advantage of being able to replace the paravirtualization API being used by the VM; in other words, a VM making use of Xen's *netfront* driver can later switch to a *virtio* network driver on KVM after a successful hypervisor transplant.

We added a kernel module in the guest that listens for transplant events from the host. The kernel module is responsible for suspending running processes inside the guest and preparing them for the I/O device transition. Notably, this technique does not break existing network connections and therefore does not interfere with applications running on the VM outside of the expected downtime, as the guest's in-kernel connection states are not altered by our device replacement.

VM memory management. The treatment of VM memory differs depending on the approach being used. Regardless, VM memory, as part of the *Guest States* category of VM states, is transferred over to the new hypervisor without needing modifications. We describe the memory handling of each approach in the below sections.

3.2. Implementing InPlaceTP

Following the general workflow of *HyperTP* described in Section 2.1, we observe that *InPlaceTP* requires the customization of steps 2 (translation), 4 (restoration) and especially step 3 (micro-reboot). Besides the VM state transformation described above, we detail the technical aspects of *InPlaceTP*'s VM memory management and micro-reboot implementation.

VM memory management. VM memory is often made of hundreds to thousands of fragments spanning multiple gigabytes in size. It therefore deserves special attention in *InPlaceTP*, as memory copies, or worse, disk writes must be minimized during the transplantation process to avoid causing excessive downtime. As pointed out in Section 2.2, our solution is to keep a VM's memory in place during the entire transplantation process, while protecting it from accidental corruption while the new hypervisor is reinitialized. To accomplish this, we store a representation of our transplanted VMs in an in-memory filesystem structure called a *PRAM structure*, adapted from the PRAM patchset [11]. Figure 3 shows the detailed construction of our PRAM structure, which consists of metadata pages that record the physical location of a VM's memory pages, allowing each VM's memory to be reconstructed after the new hypervisor boots up. In short, the PRAM structure consists of a list of *file pointers*, each of which points to a *file information page* which identifies an individual VM's unique name and memory size. Each file information page then points to a linked list of *page entries*; each page entry maps a range of VM memory pages to its location in physical memory. This memory can later be mapped in the restoration step 4 with an appropriate API (e.g. *mmap*).

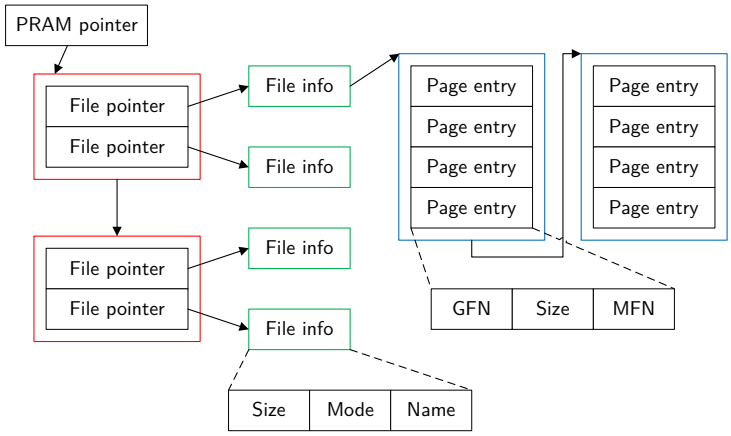

Figure 3: PRAM structure, used for identifying VM memory pages.

Micro-reboot. Step 3 in the hypervisor transplant process requires quick and efficient switching between two hypervisors on the same system without corrupting the VM states already stored in host memory. For this purpose, we make use of Kexec, which allows quickly booting a new OS kernel without having to reinitialize all devices.

While booting a new host kernel with Kexec does not require passing through BIOS (and therefore resetting memory contents), the new host kernel can still overwrite any VM states living in host memory when the system is being reinitialized. To protect the in-memory VM state information from being

corrupted during transplantation, we made three main modifications to the Kexec process:

- We reserved a dedicated memory location in which to load the new kernel to avoid Kexec itself from overwriting the relevant PRAM pages.
- We passed the PRAM pointer to the new host kernel during Kexec via its kernel command line. Knowing the PRAM structure's location, we modified the target kernel (Xen and Linux) to protect the PRAM pages and VM memory contents from being accidentally overwritten. This protection is applied during each kernel's early boot process. After the new kernel boots up, each VM's memory is presented again in a virtual filesystem where it can be picked up and reinjected back to its corresponding VM.
- Finally, we implemented various fixes in Kexec and the hypervisor kernels to ensure that the new hypervisors operate correctly after *InPlaceTP*. This ranges from correctly initializing boot structures provided by the Kexec userspace tools to ensuring that hardware devices keep operating on the new host kernel.

Optimizations. The main limitation of *InPlaceTP* is that of its required downtime. Particularly, as *InPlaceTP* suspends the VM when its host kernel is being replaced, any transplantation step happening in the meantime will impact the overall downtime of the system. However, our general hypervisor transplant workflow is not meant to be fixed, and as a result leaves room for certain optimizations. For the interest of reducing downtime during the transplantation process, we implemented the following optimizations in our *InPlaceTP* prototype:

- We *optimized the execution ordering* between different transplantation steps in the same spirit as the pre-copy step of VM live migration. This optimization is implemented in two aspects. Firstly, we build PRAM substructures ahead of time before the VM ever gets suspended. Especially in the case of Xen, where VMs' memory pages are preallocated ahead of time, this step presents a significant time save as very little work remains once the VM is finally suspended. Secondly, as a small optimization, we load the new hypervisor kernel well before the final Kexec boot. In short, the step of suspending all running VMs is deferred until absolutely necessary, thus helping to reduce *InPlaceTP*'s downtime.
- We *parallelized* our VM state construction process in order to significantly speed up the translation of VM_i *States* and creation of PRAM structures needed for transplanting VMs.
- We implemented *large page support* natively inside *InPlaceTP*. The large page CPU feature, commonly used by hypervisors, allows combining multiple appropriately-aligned pages into one large page to lower the overhead of 2D page translations. For example, on the x86-64 architecture, 512 consecutive 4 KB pages can be combined into a 2 MB large page. With

our optimization, we encode each large page's *order* (i.e. level of page combination) in our PRAM structure; each large page therefore takes up only one page entry (see Figure 3). Not only does this speed up our PRAM construction by lowering the number of VM pages that must be traversed, it also reduces the overhead of PRAM structures w.r.t. VM memory size.

- We adapted Linux/KVM to *prioritize the VM restoration process*. We found that on our test platforms, the default service priorities necessitated a long wait before VM restoration could begin (as reported by the *systemd-analyze* tool). In response, we adjusted the service boot priority on our host operating system to resume all VMs as soon as the basic services required by the VMs are ready. On a busy host running multiple services, this optimization again serves to minimize VM downtime, in lieu of waiting for non-critical, unrelated services.

3.3. Implementing MigrationTP *Xen-to-KVM transplantation*

As stated in Section 2.3, the workflow of *MigrationTP* closely matches that of *InPlaceTP*, as well as that of a normal live VM migration. Based on the existing *InPlaceTP* device management primitives, we implemented the necessary VM state transformations on the destination *kvmtool*. The incoming migration state stream from Xen is translated accordingly and applied to the destination VM. We describe below each individual step of *MigrationTP* Xen→KVM in detail:

- *Pre-copy:* In this step, VM memory pages are copied from the source host to the destination host over several iterations while the VM continues its execution. Page modification tracking is used to keep track of which pages have changed since the last copy iteration. Similar to *InPlaceTP*, the received memory pages do not need additional transformation and *MigrationTP* applies them as-is.

- *VM suspension:* Once a sufficient number of VM memory pages has been transferred to the destination, the source VM is suspended to prepare for the final migration step (cf. workflow Step 1).

- *State transfer and reconstitution:* This step covers Steps 2 to 4 of the general *HyperTP* workflow. In this step, the individual device state translations described above are applied to the incoming migration stream containing Xen VM states to form our UISR; *kvmtool* then receives the resulting UISR and applies it to the destination VM using the appropriate KVM API calls.

- *Starting the destination VM:* We arrive at the final Step 5 of our *HyperTP* workflow. Once all necessary VM states have been transferred and successfully applied, following positive confirmation from both the source and destination hypervisors, the source VM is destroyed and *MigrationTP* puts the destination VM back into operation.

4. Evaluations

This section presents the performance evaluations of the two approaches of *HyperTP*, *InPlaceTP* and *MigrationTP*. In particular, we present a time breakdown of each approach under various configurations, as well as their impacts on several different kinds of application workloads. We also evaluate the impact of *HyperTP* on the two use cases of hypervisor update and hypervisor security, and its various memory overheads. Our evaluations aim to answer the following questions:

- What are the time and memory costs incurred by each step of the transplantation for both approaches, *InPlaceTP* and *MigrationTP*?
- How scalable is each approach with varying VM sizes and number of VMs?
- What is the performance impact of *HyperTP* on user applications?
- How does *HyperTP* perform at the cluster scale for hypervisor update?
- Finally, how does *HyperTP* help improve security in a datacenter?

4.1. Experimental setup

Hardware. For our evaluations, we used two kinds of machines: two machines, each equipped with an Intel i5-8400H CPU and 16 GB of RAM (called *M1*) and one equipped with 2x Intel E5-2650L v4 and 64 GB of RAM (called *M2*). All machines are linked with a 1 Gbps Ethernet connection. We evaluated *InPlaceTP* on both M1 and M2; to ensure that *MigrationTP* experiments were conducted between similar machines, we performed them on M1 machines only. We reserved 2 CPUs for the administration OS (dom0 in Xen and host Linux in KVM), and configured hypervisors to use 2 MB huge pages for guest memory.

We made use of the Grid'5000 testbed in our cluster-scale evaluations. Each cluster member was equipped with 2x Intel Xeon E5-2630 v3 and 96 GB of RAM. All machines in the cluster were connected together with 10 Gbps Ethernet.

Applications. We evaluated *HyperTP* using three main application types: SPEC CPU 2017, MySQL and Redis. Table 3 shows a list of our tested workloads in detail.

Table 3: Description of *HyperTP* evaluation workloads.

Benchmark (metric)	**Description**
SPECrate 2017 (execution time)	23 CPU- and memory-intensive workloads
Sysbench MySQL 5.7 (latency)	Stressing a relational database with a SQL load injector
redis-benchmark (QPS)	Stressing an in-memory KV store with its included load injector

4.2. Time breakdown

In this experiment, we aim to analyze the duration of each phase of *InPlaceTP* and *MigrationTP* for each transplantation direction. We used idle VMs for this evaluation since VM activity does not impact the transplantation time.

For *InPlaceTP*, we break down the transplant process into four steps: (1) PRAM structure construction, where the VM's memory layout is analyzed and stored into a PRAM structure (noted as *PRAM* in Figure 4); (2) UISR translation, where the VM is suspended and then its execution state is taken and translated into UISR (noted as *Translation*); (3) micro-reboot, where the target hypervisor and supporting software are started using Kexec (noted as *Reboot*); and (4) UISR restoration, where the previously-taken UISR is used to restore and consequently resume the VM (noted as *Restoration*). Since our PRAM structure is constructed before pausing VMs, the downtime therefore equals *Translation* + *Reboot* + *Restoration*. Note that $time = 0$ on Figure 4 corresponds to the moment when VMs are paused, therefore the *PRAM* step is always located below the x-axis. Since network services are not needed 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 (a.k.a. downtime) and the total migration time. This is in comparison to normal live VM migration, which follows a very similar procedure (without our *MigrationTP* proxy in particular).

4.2.1. Basic evaluations

This scenario allows us to gather basic information about the performance of each step of our *HyperTP* workflow. In this scenario, our machines ran a single VM configured with 1 GB of memory and 1 vCPU. This VM size is representative of cloud workloads such as Microsoft Azure [12]. Our smallest machine (M1) could host up to 12 VMs of this size each. We repeated each experiment 5 times, while presenting average values when standard deviation is very low, and box plots otherwise.

InPlaceTP: *Xen→KVM (Figure 4).* We start with a focus on Xen→KVM to detail the time costs of each step of hypervisor transplant. The total transplantation time is 2.03 and 4.74 seconds on M1 and M2 respectively, of which 0.13/0.20 seconds is spent on *PRAM*; 0.05/0.19s on *Translation*; 1.71/3.60s on *Reboot*; and 0.15/0.75s on *Restoration*. *Reboot* is evidently the dominant step of the process, representing 83% and 76% of the total transplantation time on M1 and M2 respectively. The resulting total downtime is 1.91s on M1 and 4.54s on M2. When networking is taken into account, the process takes 6.7s on M1 (of which 6.6s is spent waiting for the network card) and 5.8s on M2 (with 5.6s spent on networking). Despite the long network downtime, we observe that these interruptions do not affect the operation of network connections.

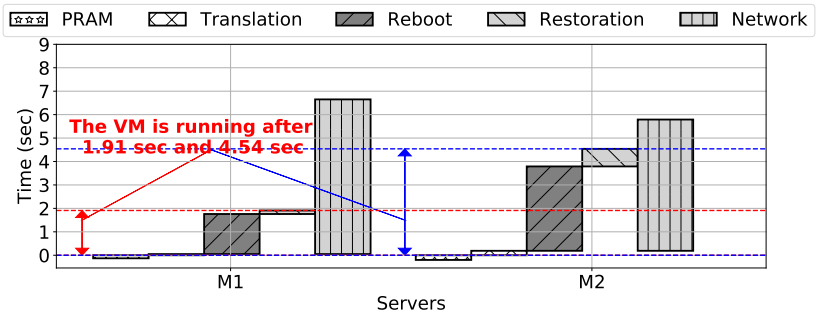

Figure 4: Time breakdown of each step of *InPlaceTP* Xen→KVM with a single VM. See Section 4.2 for a description of individual steps and our measuring process.

InPlaceTP: *KVM→Xen and Xen→Xen (Table 4).* We compare different *InPlaceTP* directions in this experiment. We observe that the reboot times for KVM→Xen and Xen→Xen are higher than that of Xen→KVM:

- KVM→Xen vs Xen→KVM: 6.67s vs 1.71s on M1; 17.92s vs 3.60s on M2;
- Xen→Xen vs Xen→KVM: 6.61s vs 1.71s on M1; 17.84s vs 3.60s on M2.

These differences are mainly caused by Xen's boot process. In fact, being a type-I hypervisor, Xen requires launching two kernels: the Xen hypervisor and the Linux dom0 kernel. Regardless, we note that the downtime caused by these directions of *InPlaceTP* is still far from the 30s maintenance window proposed by Microsoft [2] even with several VMs, as we will demonstrate in the next section.

Table 4: *InPlaceTP* for KVM→Xen and Xen→Xen. Downtime in seconds.

	PRAM	**Translat.**	**Reboot**	**Restore**	**Downtime**
M1					
Xen→KVM	0.13	0.05	1.71	0.15	1.91
KVM→Xen	0.22	<0.01	6.67	0.72	7.39
Xen→Xen	0.73	0.06	6.61	0.60	7.27
M2					
Xen→KVM	0.20	0.19	3.60	0.75	4.54
KVM→Xen	0.22	<0.01	17.92	1.23	19.15
Xen→Xen	2.87	0.22	17.84	1.19	19.25

MigrationTP: *Xen→KVM (Table 5).* We demonstrated live migration between two Xen hosts to establish a baseline for analyzing the performance of *MigrationTP*. Firstly, we observe that the total migration time is almost the same, about 9.5 seconds (dominated by memory page copies). Secondly, the downtime of *MigrationTP* is 27× lower than that of live migration between two Xen hosts. The reason is that on the destination host, *MigrationTP* uses *kvmtool* which is

more lightweight compared to Xen's *libxenctrl* and therefore needs less time to resume the VM.

Table 5: *MigrationTP* Xen→KVM compared to Xen VM live migration.

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

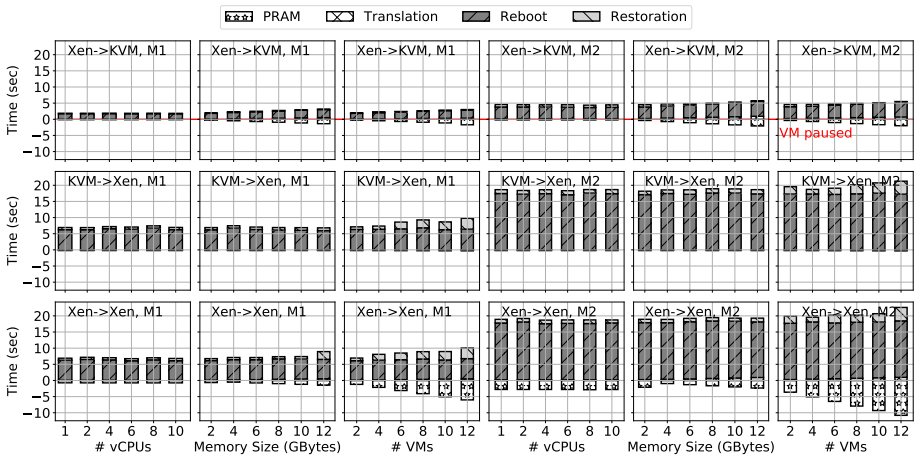

Figure 5: Scalability breakdown of *InPlaceTP* with regard to various transplantation directions and VM configurations.

4.2.2. Horizontal and vertical VM scalability

We evaluated all *HyperTP* directions (Xen→KVM, KVM→Xen, as well as Xen→Xen) while varying the VM size (number of vCPUs and memory size) and number of VMs running on each machine. Figure 5 presents our results on both M1 and M2: each row of figures corresponds to a transplantation direction (e.g the first row is Xen→KVM on M1 and M2), while each column contains results when varying an experimental parameter (e.g. the first column contain results with varying number of vCPUs on M1).

InPlaceTP *scalability.* From the first and fourth columns of Figure 5, we first notice that the number of vCPUs has no impact on the transplantation time, regardless of the transplantation direction. However, the second and fifth columns demonstrate a slight growth in downtime when varying the VM memory size on both M1 and M2 for all transplantation directions. This is mostly due to the restoration step taking more time with increasing memory size for the VM. Similarly, the total transplantation time increased slightly with the number of VMs, especially in the case of KVM→Xen and Xen→Xen which necessitates the use of Xen's slightly slower VM stack (columns 3 and 6).

Similarly to our previous evaluations, we observe that the reboot times of KVM→Xen and Xen→Xen (second and third rows) are higher than that of Xen→KVM (first row) due to Xen's longer boot sequence. Finally, we can observe that the PRAM time increases in respect to the number of VMs or memory size when the transplantation starts from Xen (first and third rows). This is due to the need to use Xen hypercalls to get access to the memory mapping of VMs for building PRAM structures. Nevertheless, this does not impact running applications because most of the PRAM structure is built with the VMs still running.

In summary, thanks to the fact that we build PRAM before pausing VMs, the VM downtime remains minimal, within 1.91 seconds and ≈10 seconds for M1; and 4.54 and ≈22 seconds for M2. Notably, our results are comparable to that of Orthus [3] (from 0.48 seconds up to 9.8 seconds), which only upgrades the KVM module and QEMU without rebooting the physical machine. Moreover, we again note that this downtime is smaller than the 30s proposed by Microsoft [2] during maintenance windows.

MigrationTP *scalability: Xen→KVM.* Figure 6 presents our results of *MigrationTP* downtime compared to that of normal VM migration. Generally, *MigrationTP* downtime is lower than that of Xen→Xen migration because of *kvmtool*'s more efficient stop-and-copy step. Additionally, while this downtime increases slightly with increasing numbers of vCPUs, it is impacted only 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 Xen's serialized migration process, which migrates multiple VMs in parallel on the sending side, but not on the receiving side. In particular, the first migrated VM's downtime will be lower than that of the second's, and so on. In comparison, *MigrationTP* offers a constant downtime on each VM by allowing multiple VMs to be migrated at the same time.

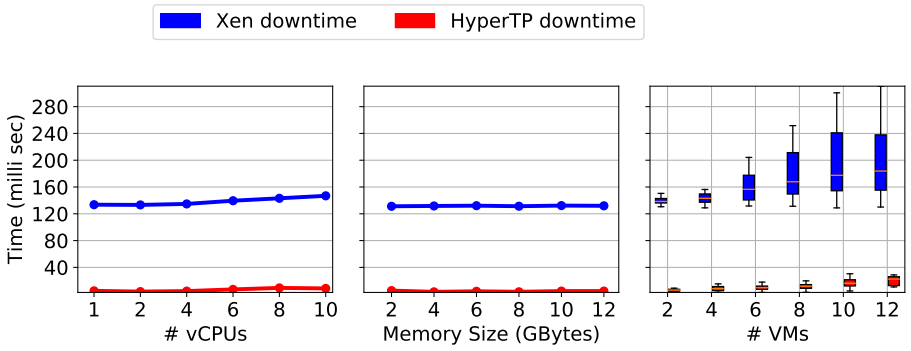

Figure 6: *MigrationTP* Xen→KVM downtime compared to Xen migration.

Figure 7 presents the total migration time of each solution. *MigrationTP* and

Xen have almost the same results when migrating a single VM while varying its memory size (see the first two subfigures). Namely, while the number of vCPUs has no impact on the migration time, migration time scales almost linearly to VM memory size due to the need for transferring VM memory over the network. When varying the number of VMs, we observe that while *MigrationTP* has a higher median VM migration time, the variance in migration time is far less than that of Xen→Xen migration. This is again caused by Xen's serialized migration which blocks multiple VMs from being migrated at the same time.

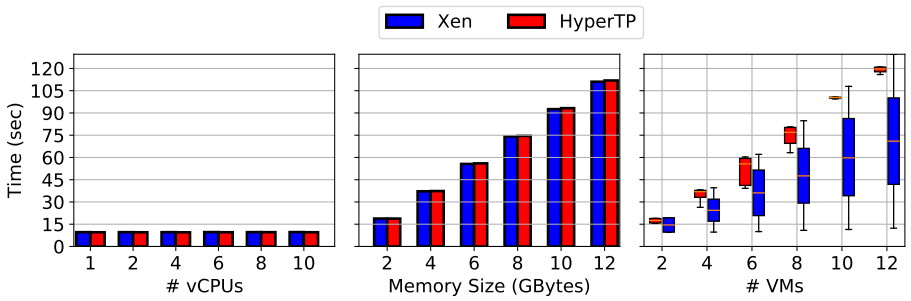

Figure 7: *MigrationTP* Xen→KVM migration time compared to Xen→Xen.

4.3. Impact on applications

We evaluated *HyperTP* using macro-benchmarks with common workloads in the following fashion: each benchmark is launched inside a Xen VM with 2 vCPUs and 8 GB of RAM; we then trigger the transplantation operation during each benchmarks' execution. We compare the results to that of a machine running purely on Xen, thus observing the impact of *InPlaceTP* on application performance.

Redis. We used the *redis-benchmark* tool to stress a Redis server running on a VM. The underlying host is then upgraded with *HyperTP* during the benchmark run.

Figure 9 presents the results for *InPlaceTP* on M1 and M2. The downtime of Redis is 8 seconds for Xen→KVM, 12s for KVM→Xen, and finally 13s for Xen→Xen on M1. On M2, the results are 11s for Xen→KVM, 22s for KVM→ Xen and 23s for Xen→Xen. Note that this downtime includes the time needed to reestablish the physical network link on the host, which is done in parallel with other phases of *InPlaceTP*. While Redis continues to perform well after transplantation, we also observe a performance difference of approximately 16% between Xen and KVM for this particular workload.

Figure 8 (right) shows the Redis performance under *MigrationTP*, which like Xen→Xen migration, shows a "classical" live migration performance pattern with a performance drop during the memory copy phase (from 50s to 124s, or 78s in total), followed by a negligible downtime when the VM is paused, and finally a return to normal performance.

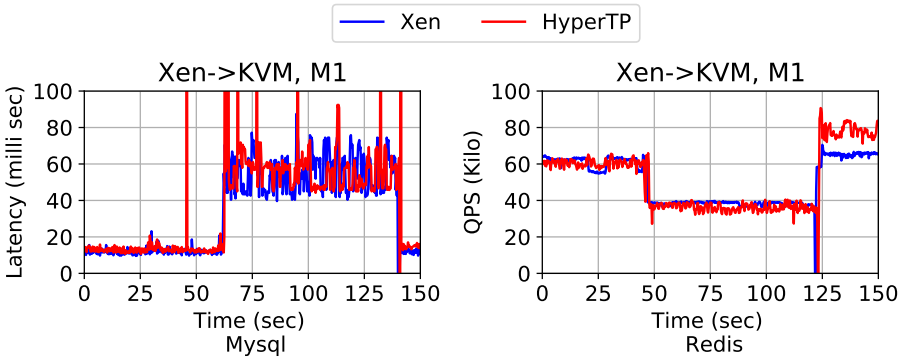

Figure 8: Impact of *MigrationTP* on MySQL (left) and Redis (right).

MySQL. We used Sysbench to generate load on a MySQL VM while applying *HyperTP*. With *InPlaceTP*, we observe a similar behavior as with Redis, where it causes a downtime of approximately 10 seconds on M1 and 21 seconds on M2 (see Figure 10). Both *MigrationTP* (Figure 8 left) and Xen→Xen migration caused a period of 252% increase in latency lasting 76 seconds during the migration process.

SPEC CPU2017. We ran all 23 SPECrate workloads included in the SPEC CPU2017 benchmark suite. We estimated the performance degradation caused by *HyperTP* as the maximum of the degradation w.r.t. Xen and KVM, i.e.

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

Table 6 presents each benchmark's execution time in seconds for Xen and KVM, as well as the performance degradation in percentage for each transplantation direction on M1 and M2. The maximum degradations for *InPlaceTP* are 5.12% on M1 and 5.00% on M2 for KVM→Xen, 4.45% on M1 and 5.29% on M2 for Xen→KVM, and 5.18% on M1 and 6.86% on M2 for Xen→Xen. *MigrationTP*'s maximum degradation on M1 is 6.27%. Note that these differences not only come from 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 the Xen and KVM columns of Table 6). Moreover, since *HyperTP*'s duration of performance degradation is quite constant, its impact on applications with longer execution times (e.g. scientific simulations) will be negligible.

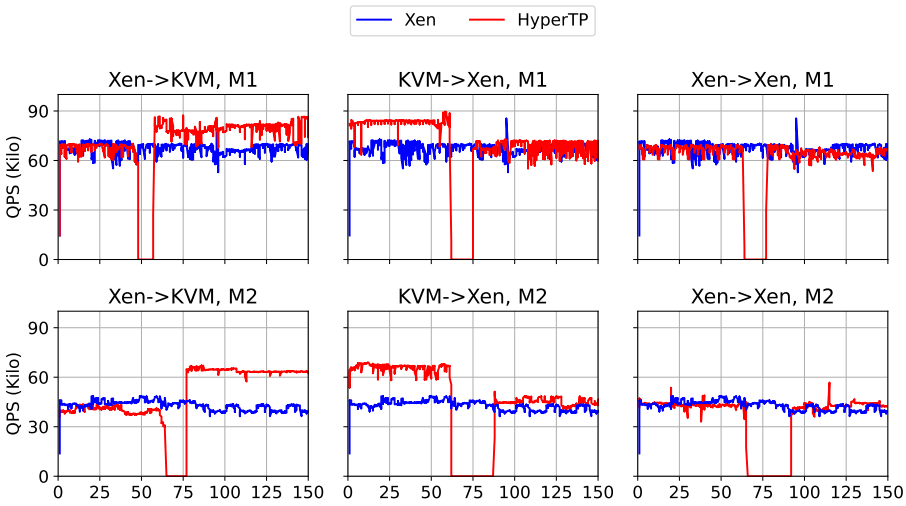

Figure 9: Impact of *InPlaceTP* on Redis throughput on M1 and M2.

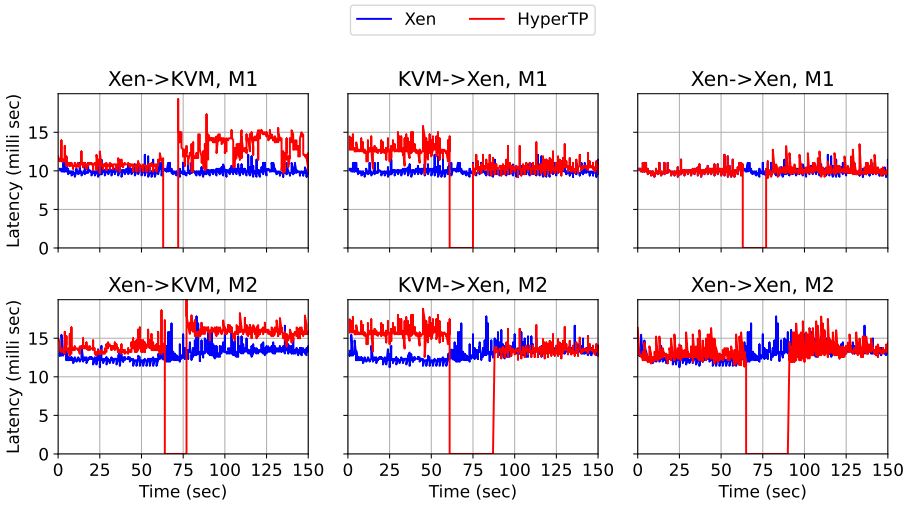

Figure 10: Impact of *InPlaceTP* on MySQL latency on M1 and M2.

Table 6: Impact of *InPlaceTP* and *MigrationTP* on SPECrate 2017 benchmarks.

					InPlaceTP						MigrationTP
	Xen		KVM		KVM to Xen		Xen to Xen		Xen to KVM		Xen to KVM
Benchmarks	Time M1	Time M2	Time M1	Time M2	Deg M1	Deg M2	Deg M1	Deg M2	Deg M1	Deg M2	Deg M1
perlbench	449.23	641.84	448.33	643.42	2.77	4.32	2.13	5.20	3.30	2.68	2.63
gcc	311.77	467.48	310.48	448.99	5.00	5.68	3.05	5.29	5.18	5.89	4.12
bwaves	939.70	1447.81	924.90	1420.64	2.92	3.74	2.33	3.70	1.65	1.00	1.83
mcf	450.80	712.02	449.33	664.51	4.04	3.88	2.82	4.01	3.45	1.49	3.12
cactuBSSN	312.87	524.55	310.01	507.50	5.12	4.14	4.45	4.03	3.58	2.39	3.82
namd	298.13	472.29	298.06	473.15	4.58	5.47	2.96	4.43	3.10	3.05	1.62
parest	636.41	1090.42	638.20	1088.68	2.28	2.29	1.47	2.46	2.24	1.39	1.15
povray	536.13	827.42	538.70	823.01	3.09	3.43	1.75	3.16	2.07	2.36	1.54
lbm	296.49	650.88	296.08	608.98	4.98	5.08	2.86	1.71	3.98	6.61	6.27
omnetpp	543.35	612.14	539.03	599.10	3.27	6.24	2.29	3.48	4.21	6.86	3.14
wrf	635.07	985.51	626.98	973.32	2.93	3.91	1.97	3.38	2.03	0.04	1.24
xalancbmk	473.08	625.59	472.07	617.82	2.40	5.28	1.47	5.00	6.68	5.69	0.79
x264	549.48	813.34	548.73	809.34	2.67	3.11	2.18	3.24	3.98	1.67	0.59
blender	423.02	582.61	422.51	580.16	3.41	4.59	2.16	4.14	3.05	2.64	1.68
cam4	524.44	811.05	522.12	807.74	3.35	3.48	2.14	2.33	2.81	2.38	2.24
deepsjeng	442.18	650.24	442.82	644.92	3.36	2.97	2.23	3.48	4.34	1.99	4.33
imagick	693.98	1008.05	693.31	1007.75	2.16	2.59	1.50	2.10	1.58	1.33	0.70
leela	725.80	1022.77	726.39	1018.50	2.10	2.82	1.26	2.56	1.55	1.73	0.65
nab	544.06	767.31	543.11	764.83	2.61	3.45	1.79	3.19	1.76	1.81	0.48
exchange2	566.03	856.35	565.98	857.11	2.44	2.99	1.63	2.70	1.73	1.67	0.96
fotonik3d	390.19	763.08	386.25	679.30	4.14	6.94	2.50	2.03	4.65	0.13	3.86
roms	422.13	662.71	417.98	656.53	4.41	1.67	3.78	4.16	4.89	0.31	5.76
xz	522.24	591.80	515.12	675.08	3.04	2.70	2.78	3.59	5.00	3.63	1.89

4.4. Hypervisor update

In this section, we evaluated the time taken to upgrade a cluster using *MigrationTP*. We used the *BtrPlace* VM scheduler framework [13] to define the structure of a simple server cluster including 10 physical hosts. On each hypervisor host, we defined 10 VMs each equipped with 1 vCPU and 4 GB of RAM, for a grand total of 100 VMs across all hosts. In this group of VMs, we configured 30% to run 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 simulated an upgrade event by dividing the cluster into smaller groups, sequentially putting each group offline using *BtrPlace*'s constraints (placing VMs from the offline group into other groups), followed by recording the resulting migration plans. For information, *BtrPlace* generated a migration plan with a total of 154 VM migration operations. We then prepared a real software cluster running the above-defined VMs, executed the migration plans proposed by *BtrPlace*, and recorded the total migration times of the cluster.

We repeated the same experiments while varying the percentage of VMs that are *InPlaceTP* compatible. Figure 11 shows the number of migrations and reduction in total migration times (compared to normal migration-based upgrades) with varying proportions of *InPlaceTP*-compatible VMs. We observe that increasing the proportion of *InPlaceTP*-compatible VMs reduces the number of migrations necessary to upgrade the cluster, as well as the total migration times. For example, with 20% *InPlaceTP*-compatible VMs, our migration plan required 109 migrations, corresponding to a 17% shorter migration duration. With 60% compatible VMs, the cluster needed 73% fewer migrations and 68% less migration time, and with 80% compatible VMs, the cluster required only 25 migrations, or reducing total migration time by almost 80%. Coupled with the fact that *InPlaceTP* takes only seconds to complete, these results show how *HyperTP* can substantially speed up the upgrading of a hypervisor cluster.

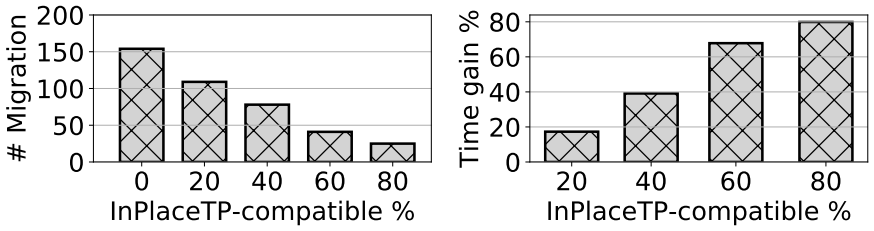

Figure 11: Impact of *InPlaceTP* on cluster updating: a) w.r.t. the number of migrations; b) w.r.t. total update time.

4.5. Hypervisor security

As we claimed in Section 2.4, *HyperTP* can be used to reduce the vulnerability window of a hypervisor. To qualify our claim, we study the case of a vulnerability

on Xen. For example, consider CVE-2018-18883 [14], a critical denial-of-service vulnerability with a CVSS v2 score of 7.2 affecting Xen 4.9.x-4.11.x on Intel x86 platforms. In short, the vulnerability allows Xen HVM guests to configure CPU virtualization features even if they are disabled in its configuration, leading the host Xen to access uninitialized memory and causing Xen to crash. The root cause of CVE-2018-18883 is mismanagement of Intel's virtualization features; hosts running KVM are not vulnerable to the same issue. In this case, *HyperTP* can help quickly handle such a vulnerability by switching from Xen to KVM.

Another use case of *HyperTP* is to apply certain *software diversity*-based security measures on-the-fly during operation of an hypervisor. For example, certain operating systems support link-time randomized binary layouts [15, 16]. However, these defensive measures can only be applied once at boot-time, meaning they become less effective over time in long-running systems. *HyperTP* can be used to periodically reapply these measures by switching from $H_{current}$ to a H_{target} with a different random binary layout, making attacks requiring running the same binary for a long time (e.g. those that need to probe the address space) more difficult to execute.

4.6. Memory overhead

The memory overhead of *HyperTP* includes the extra memory required for storing PRAM structures and UISR states. Figure 12 presents our overhead measurements for various transplantation scenarios as explored in Section 4.2.1. We can see that the memory footprint of PRAM structures increases with the VM memory size, 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 needed for each VM (see Figure 3); 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) leading to a worst-case overhead of 2 megabytes of metadata 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).

The memory footprint of UISR states 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. Note that this extra memory is immediately given back to the hypervisor as soon as the transplantation process finishes.

5. Discussion

UISR and VM compatibility. Section 2.1 specified that UISR serves as a representation of VM state that is sufficient for its reconstruction. However, restoration of a VM from a given UISR instance is affected by several factors: (1) hardware compatibility between the source and target hypervisor platforms, e.g. matching processor features; (2) each hypervisor has numerous different implementations

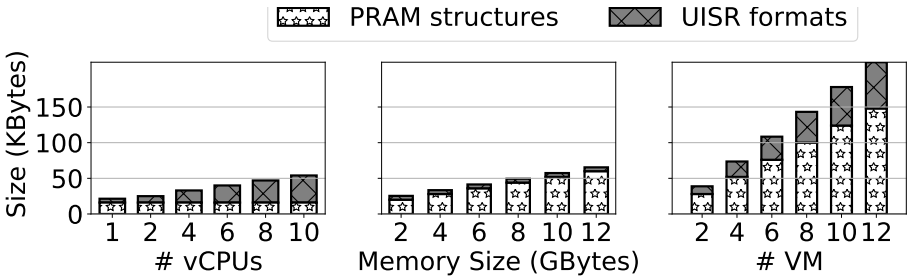

Figure 12: Memory overheads of *InPlaceTP* and *MigrationTP*.

of devices and resources, some of which cannot be easily reconstructed (e.g. passthrough devices); and (3) breaking changes to the hypervisor's paravirtualization API contract. As a result, the set of features made available to the VM must be chosen such that either (1) the feature could be reproduced in our UISR and in the target hypervisor; or (2) the target VM is tolerant to loss of said feature's states. In our implementation, devices that fall into the second category (e.g. PCI network devices) are implemented using the stop-reconnect technique, as the guest Linux operating system in combination with our kernel driver allows removing and reinstalling these devices without affecting network connections.

Downtime-resource tradeoff. As stated above, datacenter operators must choose whether *InPlaceTP* or *MigrationTP* is more appropriate for the maintenance of their virtualization platform. While our evaluations show that *InPlaceTP* shortens the maintenance duration, it also comes with an intrinsically longer downtime (dominated by the booting of the new hypervisor, as seen in Section 4.2.1). The individual downtime of each VM (several seconds) might not be acceptable compared to the milliseconds of downtime offered by migration-based techniques. Moreover, this downtime tends to slightly increase along with the number of VMs; this increase, while predictable, must be taken into account during the maintenance planning. One way to perform this is by mixing and matching the two techniques: for instance, VMs can be separated into service tiers; as an example, one can define two tiers denoted "default" and "mission-critical". Most VMs that can tolerate a short downtime can be placed in the "default" tier on machines serviced using *InPlaceTP*; VMs denoted "mission-critical" can instead be hosted on machines using *MigrationTP*, where upgrading will not cause a significant disruption.

Hypervisor security. It is worth noting that in the context of hypervisor security, *HyperTP* first of all serves as a *mitigation*, meaning it is applied *after* a vulnerability becomes known, as illustrated in Figure 2. Moreover, before the mitigation could be applied, the vulnerability must be analyzed to determine which *HyperTP*-capable hypervisor is an appropriate transplant target (e.g. not vulnerable to the vulnerability in question). Secondly, *HyperTP* cannot currently

serve as a replacement for remediation of a compromised system. Note that techniques exist to bring a running system to a known, verifiable state (e.g. Dynamic Root for Trusted Measurement (DRTM) [17, 18]) and can potentially be used to reinforce such a remediation. To elaborate, a DRTM launch event can be used to implement micro-reboot by taking over a running hypervisor, putting the system in a known trusted state, then booting the target hypervisor. As the target hypervisor started from a trusted state, it is considered "fresh" and free from any leftover compromised state. It can then pick up the UISR left by the running hypervisor and resume any running VMs. However, a limitation of *InPlaceTP* is that it currently does not implement these techniques. In comparison, *MigrationTP* and other live migration-based techniques [3, 19] can be used to transfer VMs from a potentially-compromised host to another hypervisor. While none of these techniques (whether micro-reboot-based or migration-based) can guarantee the integrity of the aforementioned VMs post-compromise, this limitation can again be mitigated by the use of encrypted virtual machines (e.g. AMD SEV [20]).

6. Related work

Our investigation of the state of the art will focus on the applications of *HyperTP* on hypervisor update and security.

6.1. Hypervisor update

Live patching. The least disruptive method for updating the hypervisor is kernel live patching [21, 22]. Live patching is a lightweight solution for applying 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 such patches are not sufficient, VM live migration or in-place hypervisor update with server reboot should be used instead.

Live migration. VM live migration [23, 24] allows the cloud provider to upgrade almost everything on the origin server, from hardware devices to the hypervisor, once it no longer hosts any running VMs. Several works [25, 26] have investigated downtime reduction during live migration. Tsakalozos et al. [27] proposes the use of a special-purpose *MigrateFS* file system and a network of brokers for synchronizing virtual disk states to ease the migration of VMs without making use of shared remote storage. These approaches can also be combined with *MigrationTP* to further improve the performance of migrating VMs that are not compatible with *InPlaceTP*.

To our knowledge, Liu et al. [28] 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. [28] 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.

In-place hypervisor update. Zhang et al. [3] introduces *Orthus*, which targets the upgrading of both the user-space emulator 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. However, *Orthus* is specific to KVM, and does not target heterogeneous hypervisors like *HyperTP*. Secondly, *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 [6, 19, 29] uses nested virtualization to enable quick and transparent in-place updates. LivCloud [30] solves the related problem of migration compatibility between different cloud providers by making use of a common L1 hypervisor. While these works are comparable with *MigrationTP* in that they also propose a low-downtime solution for updating the L1 (i.e. "inside") hypervisor, they did not propose a mechanism for updating the L0 (i.e. "outside") hypervisor. *HyperTP*, in comparison, does not include this limitation as the hypervisor kernel is entirely restarted. Nested virtualization in this fashion also incurs an additional overhead, especially on the commonly-used x86 architecture where virtualization instructions executing inside the L1 hypervisor must be trapped and emulated by the L0 hypervisor.

6.2. Hypervisor security

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

Preventive approaches, e.g. hardening the hypervisor: Many research works advocate a micro-kernel architecture for the hypervisor in order to (1) reduce the trusted computing base (TCB), thus reducing the attack surface [31, 32]; (2) formally verify this TCB to prove the absence of known classes of vulnerabilities [33]; and (3) isolate buggy or untrusted device drivers of the hypervisor [31, 34, 35]. This approach often imposes a strict implementation of a micro-kernel architecture, which requires considerable efforts in the hypervisor's design and implementation. In addition, most of the contributions in this approach require hardware changes that are not yet available [36]. Moreover, no implementation is 100% sure; such an approach has to be combined with regular security updates as studied in the next section.

Preventive approaches, e.g. software diversity: The concept of software diversity involves using multiple software versions to mitigate vulnerabilities. Schaefer et al. [37] describe general approaches for utilizing and managing diverse software systems; In the context of virtualization, Winarno et al. [38] studies the use of multiple hypervisors to ensure system resilience. Tan et al. [39] proposes a similar scheme based on one hypervisor running on multiple cloud platforms.

However, to our best knowledge, our work is the first that allows a VM to be transplanted between multiple hypervisors on the same machine with minimal disruption.

Reparative approaches, e.g. consistent state restoration: These mainly rely on fast reboot and restoration, and can be implemented at the OS or hypervisor level [40, 41, 42, 43, 44, 45, 46]. Notably, Otherworld [41] 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 [40, 42] 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. Cerveira et al. [46] 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 approaches, e.g. mitigation during vulnerability windows: The aforementioned approaches have a limitation: that **they cannot protect against a vulnerability if the corresponding security patch is not yet available**. As we highlighted in Section 2.4, the creation of such a security patch can take anywhere from several days to multiple months. In contrast, *HyperTP* provides an unique "escape hatch" that protects virtualization infrastructure during a vulnerability window with little downtime (as long as the vulnerability does not impact the target hypervisor). The combined approach of *HyperTP* also gives operators a flexible tradeoff between downtime and resource usage.

7. Conclusion

We introduced *HyperTP*, a platform for replacing a running hypervisor with a different hypervisor while incurring minimal downtime in a process called *hypervisor transplant*. We discussed our ideas of *VM state hierarchy* and *Unified Intermediate State Representation*, and presented details of the two approaches that make up *HyperTP*, in-place hypervisor transplant (*InPlaceTP*) and migration-based transplant (*MigrationTP*). We evaluated our prototype of *HyperTP* with well-known benchmarks, and showed that *HyperTP* causes minimal interference to running workloads. Namely, *InPlaceTP* needs less than 2 seconds to transplant a VM running on Xen to KVM, while requiring negligible memory and I/O overhead. *MigrationTP* transplants a VM to another hypervisor with essentially the same cost as normal live migration. We showed that a hypervisor cluster with 80% of VMs supporting *InPlaceTP* can reduce its upgrade time by a proportional 80%, and demonstrated how *HyperTP* can simplify the process of securing hypervisor infrastructure.

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.

References

[1] T. Dinh Ngoc, B. Teabe, A. Tchana, G. Muller, D. Hagimont, Mitigating vulnerability windows with hypervisor transplant, in: Proceedings of the Sixteenth European Conference on Computer Systems, 2021, pp. 162–177.

[2] Microsoft Corp., Maintenance for virtual machines in Azure, `https://docs.microsoft.com/en-us/azure/virtual-machines/maintenance-and-updates` (2022).

[3] X. Zhang, X. Zheng, Z. Wang, Q. Li, J. Fu, Y. Zhang, Y. Shen, 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, 2019, pp. 93–105.

[4] A. Segalini, D. Lopez-Pacheco, G. Urvoy-Keller, F. Hermenier, Q. Jaquemart, Hy-FiX: Fast in-place upgrades of KVM hypervisors, IEEE Transactions on Cloud Computing (01) (2021) 1–1.

[5] Sun Microsystems, Inc., XDR: External data representation standard, `https://tools.ietf.org/html/rfc1014` (1987).

[6] H. Bagdi, R. Kugve, K. Gopalan, Hyperfresh: Live refresh of hypervisors using nested virtualization, in: Proceedings of the 8th Asia-Pacific Workshop on Systems, 2017, pp. 1–8.

[7] X. Wang, H. Yu, How to break MD5 and other hash functions, in: Annual international conference on the theory and applications of cryptographic techniques, Springer, 2005, pp. 19–35.

[8] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, et al., Meltdown: Reading kernel memory from user space, Communications of the ACM 63 (6) (2020) 46–56.

[9] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, et al., Spectre attacks: Exploiting speculative execution, Communications of the ACM 63 (7) (2020) 93–101.

[10] NIST, National Vulnerability Database - vulnerability metrics, `https://nvd.nist.gov/vuln-metrics/cvss`.

[11] V. Davydov, pram: persistent over-kexec memory file system, `https://lists.openvz.org/pipermail/criu/2013-July/009877.html` (2013).

[12] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, R. Bianchini, 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, 2017, p. 153–167. doi:10.1145/3132747.3132772.
URL https://doi.org/10.1145/3132747.3132772

[13] F. Hermenier, J. Lawall, G. Muller, Btrplace: A flexible consolidation manager for highly available applications, IEEE Transactions on dependable and Secure Computing 10 (5) (2013) 273–286.

[14] CVE-2018-18883, https://www.cvedetails.com/cve/CVE-2018-18883/ (2018).

[15] OpenBSD kernel address randomized link, https://lwn.net/Articles/727697/ (2017).

[16] Function granular KASLR, https://lore.kernel.org/lkml/20211223002209.1092165-1-alexandr.lobakin@intel.com/ (2021).

[17] TCG D-RTM architecture, https://trustedcomputinggroup.org/resource/d-rtm-architecture-specification/ (2013).

[18] Intel Trusted Execution Technology (Intel TXT) overview, https://www.intel.com/content/www/us/en/support/articles/000025873/processors.html (2022).

[19] K. Kourai, H. Ooba, Zero-copy migration for lightweight software rejuvenation of virtualized systems, in: Proceedings of the 6th Asia-Pacific Workshop on Systems, 2015, pp. 1–8.

[20] D. Kaplan, J. Powell, T. Woller, AMD memory encryption, https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v9-Public.pdf (2016).

[21] J. Kosina, P. Mládek, V. Pavlík, J. Slaby, kGraft: Live patching of the Linux kernel, https://kernel-recipes.org/en/2014/kgraft-live-patching-of-the-linux-kernel/ (2014).

[22] J. Arnold, M. F. Kaashoek, 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, 2009, p. 187–198. doi:10.1145/1519065.1519085.
URL https://doi.org/10.1145/1519065.1519085

[23] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, A. Warfield, 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, 2005, p. 273–286.

[24] F. Machida, D. S. Kim, K. S. Trivedi, Modeling and analysis of software rejuvenation in a server virtualized system with live VM migration, Performance Evaluation 70 (3) (2013) 212–230.

[25] H. Jin, L. Deng, S. Wu, X. Shi, X. Pan, Live virtual machine migration with adaptive, memory compression, in: 2009 IEEE International Conference on Cluster Computing and Workshops, IEEE, 2009, pp. 1–10.

[26] U. Deshpande, U. Kulkarni, K. Gopalan, 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, 2012, p. 19–26. doi:10.1145/2287056.2287062.
URL https://doi.org/10.1145/2287056.2287062

[27] K. Tsakalozos, V. Verroios, M. Roussopoulos, A. Delis, Live VM migration under time-constraints in share-nothing IaaS-clouds, IEEE Transactions on Parallel and Distributed Systems 28 (8) (2017) 2285–2298.

[28] P. Liu, Z. Yang, X. Song, Y. Zhou, H. Chen, B. Zang, Heterogeneous live migration of virtual machines, in: International Workshop on Virtualization Technology (IWVT'08), 2008.

[29] H. Yamada, K. Kono, Traveling forward in time to newer operating systems using shadowreboot, in: Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, 2013, pp. 121–130.

[30] I. E. A. Mansour, K. Cooper, H. Bouchachia, Effective live cloud migration, in: 2016 IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud), IEEE, 2016, pp. 334–339.

[31] D. G. Murray, G. Milos, S. Hand, 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, 2008. doi:10.1145/1346256.1346278.
URL https://doi.org/10.1145/1346256.1346278

[32] J. Szefer, E. Keller, R. B. Lee, J. Rexford, 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, 2011, p. 401–412. doi:10.1145/2046707.2046754.
URL https://doi.org/10.1145/2046707.2046754

[33] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al., seL4: Formal verification of an OS kernel, in: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, 2009, pp. 207–220.

[34] L. Shi, Y. Wu, Y. Xia, N. Dautenhahn, H. Chen, B. Zang, J. Li, 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, 2017.
URL https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/deconstructing-xen/

[35] F. Zhang, J. Chen, H. Chen, B. Zang, CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization, in: T. Wobber, P. Druschel (Eds.), Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011, ACM, 2011, pp. 203–216. doi:10.1145/2043556.2043576.
URL https://doi.org/10.1145/2043556.2043576

[36] W. Shi, J. Lee, T. Suh, D. H. Woo, X. Zhang, 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, 2012. doi:10.1145/2212908.2212920.
URL https://doi.org/10.1145/2212908.2212920

[37] I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, K. Villela, Software diversity: state of the art and perspectives (2012).

[38] I. Winarno, T. Okamoto, Y. Hata, Y. Ishida, Increasing the diversity of resilient server using multiple virtualization engines, Procedia Computer Science 96 (2016) 1701–1709.

[39] Y. Tan, D. Luo, J. Wang, CC-VIT: Virtualization intrusion tolerance based on cloud computing, in: 2010 2nd International Conference on Information Engineering and Computer Science, IEEE, 2010, pp. 1–6.

[40] K. Kourai, S. Chiba, Fast software rejuvenation of virtual machine monitors, IEEE Transactions on Dependable and Secure Computing 8 (6) (2011) 839–851.

[41] A. Depoutovitch, M. Stumm, Otherworld: giving applications a chance to survive OS kernel crashes, in: Proceedings of the 5th European conference on Computer systems, 2010, pp. 181–194.

[42] M. Russinovich, N. Govindaraju, M. Raghuraman, D. Hepkin, J. Schwartz, A. Kishan, Virtual machine preserving host updates for zero day patching in public cloud, in: Proceedings of the Sixteenth European Conference on Computer Systems, 2021, pp. 114–129.

[43] M. Le, Y. Tamir, Applying microreboot to system software, in: 2012 IEEE Sixth International Conference on Software Security and Reliability, 2012, pp. 11–20.

[44] X. Xu, H. H. Huang, DualVisor: Redundant hypervisor execution for achieving hardware error resilience in datacenters, in: 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2015, pp. 485–494.

[45] D. Zhou, Y. Tamir, Fast hypervisor recovery without reboot, in: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2018, pp. 115–126.

[46] F. Cerveira, R. Barbosa, H. Madeira, Fast Local VM Migration Against Hypervisor Corruption, in: 2019 15th European Dependable Computing Conference (EDCC), 2019, pp. 97–102.