Out of hypervisor (OoH) : when nested virtualization becomes practical

Stella Bitchebe

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

AU DELA DE L'HYPERVISEUR (OoH): QUAND LA VIRTUALISATION IMBRIQUEE DEVIENT PRATIQUE

Stella BITCHEBE

Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis (I3S)

Présentée en vue de l'obtention du grade de docteur en INFORMATIQUE **d'**Université Côte d'Azur

Dirigée par : Alain TCHANA

Soutenue le : 03 Février 2023

Devant le jury, composé de :
Edouard Bugnion, Professeur, EPFL
Gaël Thomas, Professeur, Telecom SudParis
Guillaume Urvoy-Keller, Professor, Université Côte d'Azur
Renaud Lachaize, MCF, Université de Grenoble
Oana Balmau, MCF, Université McGill
Natacha Crooks, MCF, Université de Berkeley
Laurent Réveillère, Professeur, Université de Bordeaux
Alain Tchana, Professeur, Université de Grenoble

Université Côte d'Azur

DOCTORAL SCHOOL STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

PHD THESIS

OUT OF HYPERVISOR (OoH): WHEN NESTED VIRTUALIZATION BECOMES PRACTICAL

Stella BITCHEBE

⋈

Thesis Advisor: Professor Alain TCHANA

Prepared at Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis (I3S)

Defended on February 3^{rd}, 2023

Jury :

Reviewers

Edouard Bugnion, Full Professor, Data Center Systems Laboratory, EPFL
Gaël Thomas, Full Professor, Computer Science Department, Telecom SudParis

Examiners

Guillaume Urvoy-Keller, Full Professor, I3S Laboratory, Université Côte d'Azur
Renaud Lachaize, Assistant Professor, LIG Laboratory, Université Grenoble Alpes
Oana Balmau, Assistant Professor, DISCS Laboratory, McGill University
Natacha Crooks, Assistant Professor, EECS Department, UC Berkeley
Laurent Réveillère, Full Professor, LaBRI Laboratory, Université de Bordeaux

Advisor

Alain Tchana, Full Professor, LIG Laboratory, Université Grenoble Alpes

An African proverb says: "For a child to grow up, it takes a whole village", and I would say that it takes a ***great*** *family...*

To my family

Acknowledgments

This thesis sanctions four years of hard work. Four years during which, thanks to my Ph.D. advisor, Professor Alain Tchana, I learned a lot and filled in many gaps. Yes, significant gaps, because four years ago, I knew nothing, sincerely nothing at all about the System or even about C :). I would therefore like to thank him for all these teachings. Albert Einstein said, "*it is the essential role of the teacher to awaken the joy of working and learning*" and Professor Alain, you played this role to perfection. Thank you for believing in my potential and for your patience when I didn't understand your explanations. Thank you for have been a great mentor and much more than a thesis director.

I would like to extend my thanks to the members of my jury for having accepted to be part of it and for the time spent reading this document. Thanks to Professor Edouard, who is, as Alain likes saying, one of the inventors of virtualization :). I particularly thank Professor Laurent, who has been there from the beginning of this experience and has contributed to making possible the internship that introduced me to the research world. A special thank also to Professor Gaël, whose letters of recommendation were (along with those of Natacha), I am sure, a great contribution to my awards applications.

This adventure, which started with deep pain and a many difficulties at Nice, which almost ended before it even started, would never have been possible without my motivation: my family. You are the reason why I have always risen and that, despite all the hardships I have endured, I continued on and on. I have not yet given birth, but I have four children, four daughters (Mervy, Ari, Fébi, and baby Vicky), who are the reason for my sleepless nights and dedication. And I owe this dedication to my parents, Mr. and Mrs. Bitchebe, who never spared their efforts for our education. You are now Doctors (Ph.D.) by procuration, I hope you will be proud of yourself :).

Away from my family, my thesis mates often helped fill the void. So I want to thank you guys (and girls, haha) for the laughs, the *barbeuks*, and even the useless problems. I am thinking in particular of Djob Mvondo (my very first friend in France, who never stopped giving us advice), Kevin Nguetchouang, Lucien Ngaleu, Calvin A. Haidar, Peterson Yuhala, Lavoisier Wapet, Kevin Jiokeng, and Josiane Kouam.

I would be remiss if I did not give a special thanks to Chiraz Benamor, our administrative manager at ENS Lyon, who understood our problems and difficulties and our status as foreigners in managing administrative procedures.

I would also like to thank all the sponsors who funded my thesis work. In particular, the NEC lab Europe, the L'Oreal-UNESCO foundation for Women In Science, Microsoft Research Cambridge, and Google EMEA.

And last but not least, me! Yes, I would like to thank me. Thanks to me for my abnegation and for believing in myself. Thanks for staying strong and holding on

despite all I have been through during these past four years, which have taught me many things, especially, as an African proverb says, "*let him who has not crossed not mock him who has drowned*". Thanks for staying fixed on my goals and a lot of courage to me to stay fixed and keep moving forward, because as Albert Einstein said, "*life is like a bicycle, you have to move forward to keep your balance*".

Résumé

De nos jours, le cloud est l'environnement d'exécution par excellence des applications modernes, grâce à son coût très attractif et la simplification des tâches d'administration qu'elle offre. Les utilisateurs du cloud déploient parfois des hyperviseurs dans leurs machines virtuelles (on parle de virtualisation imbriquée) à des fins de tests de déploiement ou pour exploiter les mécanismes des hyperviseurs. Cette pratique exacerbe la dégradation induite par la virtualisation, conduisant à des performances catastrophiques et rendant la virtualisation imbriquée généralement peu pratique et rarement utilisée.

Afin de pallier la dégradation induite par la virtualisation, les fabricants de processeurs ont commencé à commercialiser, en 2005/2006, des processeurs dotés de technologies permettant de prendre en charge la virtualisation au niveau du matériel et de ce fait, offrir de meilleures performances. Plusieurs fonctionnalités de virtualisation matérielle ont été proposées telles que Intel PML, SPP, CAT ou EPT. Ces dernières ne peuvent actuellement être utilisées que par l'hyperviseur, bien qu'elles pourraient également être bénéfiques aux processus s'exécutant dans les machines virtuelles, et donc aux applications des utilisateurs dans le cloud.

Cette thèse introduit *Out of Hypervisor* (OoH), un nouvel axe de recherche motivé par des besoins semblables à ceux de la virtualisation imbriquée. Au lieu d'essayer de virtualiser entièrement une machine à l'intérieur d'une machine virtuelle pour supporter un hyperviseur, OoH propose d'exposer individuellement les fonctionnalités de virtualisation matérielle au système d'exploitation invité afin que ses processus puissent également en bénéficier, faisant ainsi de OoH une excellente alternative à la virtualisation imbriquée.

Nous montrons la pertinence de OoH, dans cette thèse, avec la protection de la mémoire et la traque des pages mémoire. La protection de la mémoire en écriture est l'un des mécanismes clés des techniques de protection contre les débordements de tampon mémoire. Un débordement de tampon mémoire est un bug informatique très répandu dont la prévalence a augmenté au fil des ans pour devenir en 2022, le problème de sécurité le plus important et le plus critique. Quant à la traque des pages sales, elle est au coeur de diverses tâches essentielles telles que l'estimation de la quantité de mémoire effectivement utilisée par un processus (qui facilite la gestion de ressources par le fournsseur de cloud), le checkpoint des processus (qui facilite la récupération de données après une défaillance) et le ramasse-miettes (qui favorise une gestion optimale de la mémoire). Pour rendre ces tâches primordiales plus efficaces pour les utilisateurs du cloud, nous appliquons OoH aux technologies Intel PML (Page Modification Logging) et Intel SPP (Sub-Page write Permissions).

PML est une fonctionnalité publiée en 2015 qui permet la traque efficace des pages mémoire modifiées, pour améliorer la migration en live des machines virtuelles. SPP a été introduit en 2018, et permet de protéger la mémoire à la granularité d'une sous-page de 128 octets au lieu de 4 Ko.

Mots Clés: Virtualisation, Virtualisation Imbriquée, Out of Hypervisor, Intel PML, Intel SPP.

Abstract

Nowadays, virtualized clouds are the de facto execution environment of modern applications, thanks to their very attractive costs and administration tasks simplification. Cloud users sometimes adopt nested virtualization by deploying hypervisors in their virtual machines (for test deployment purposes or to exploit hypervisor mechanisms), which duplicates the intrinsic costs of virtualization, leading to catastrophic performances and making nested virtualization generally unpractical and rarely utilized.

In order to alleviate drawbacks induced by virtualization, hardware vendors started releasing, in 2005/2006, processors with technologies to support virtualization and provide better performance. Several hardware-virtualization features such as Intel PML, SPP, CAT, and EPT have been proposed, which currently can only be used by the hypervisor. Nonetheless, these features could also be beneficial to processes running inside virtual machines and then to cloud user applications.

This thesis introduces Out of Hypervisor (OoH), a novel research axis driven by similar needs as nested virtualization. Instead of emulating full virtual hardware inside a virtual machine to support a hypervisor, OoH proposes to individually expose hypervisor-oriented hardware virtualization features to the guest OS so that its processes could also benefit from those features, making OoH an excellent alternative to nested virtualization.

We prove the relevance of OoH in this thesis with memory write-protection and dirty page tracking in guest userspace. Memory write-protection is one of the key mechanisms to buffer overflow mitigation, a widespread memory safety violation whose prevalence has increased over the years to reach the top vulnerability reported in 2022. And dirty page tracking is at the heart of various essential tasks such as working set size estimation, process checkpointing, and concurrent garbage collection. Working set size estimation is a critical need for cloud providers as it enables efficient overcommitment, which allows efficient resource management. Checkpointing facilitates recovery after failure, and garbage collection promotes efficient memory management and saving.

To make these primordial tasks more efficient for cloud users, we apply OoH to Intel SPP (Sub-Page write Permissions) and Intel PML (Page Modification Logging). SPP was introduced in 2018 and provides fine-grained write-protect accesses at a 128B sub-page granularity instead of a traditional 4KB page. And PML is a feature released in 2015 that allows efficient dirty page tracking for improving virtual machines' live migration.

Keywords: Virtualization, Nested Virtualization, Out of Hypervisor, Intel PML, Intel SPP.

Contents

Chapter 1

Introduction

Contents

1.1 Context and Problem Statement

Virtualization has become the foundation of data centers because it allows resource mutualization (thus, optimal resource utilization) between several clients while ensuring isolation. Thanks to this mutualization, virtualization can offer attractive costs to cloud users. Moreover, virtualization also provides adequate support for administration, including resource management. In the 1970s, Popek and Goldberg [138] evoked the idea of *recursive virtualization, where a virtual machine runs under itself a copy of a hypervisor*. They were hence the first to formalize the concept of nested virtualization, definable as the stacking of multiple hypervisors [114, 67, 162, 116, 150, 94, 65], see Figure 1.1.

Nested virtualization induces substantial overhead than non-nested virtualized systems due to the significant number of VM (virtual machine) traps (at least $\times 2$ [150]). Figure 1.2 gives an insight into such traps. If the L2 VM, in the Figure, performs a privileged instruction that would have normally trapped to L1 in a non-virtualized system, the latter instead traps to L0. When L0 handles the exit and notices that it is not the one concerned, it returns to L1. L1 then properly treats the exit and should now set the VM's data structures accordingly. However, since it does not have privileged access to the hardware, it must, in turn, trap to L0. Herein will follow a less or more long communication between L0 and L1 hypervisors in the sort of *questions-answers*, where L1 instructs L0 to set the processor registers and VM's data structures accordingly to the cause of the exit. And only at the end of this *dialogue* the L0 hypervisor finally resumes the VM.

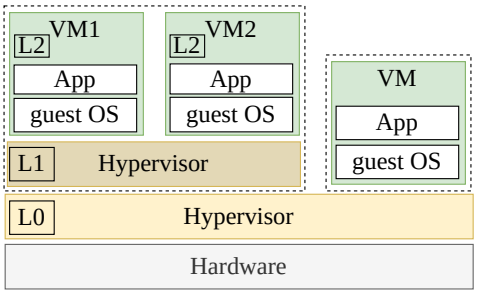


Figure 1.1: Nested virtualization general architecture.

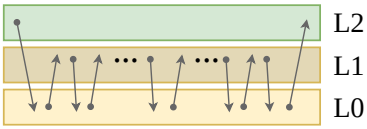


Figure 1.2: Traps between nested levels.

For example, Vilanova et al. [150] measured 73% degradation in executing the `cpuid` instruction in a nested virtualized system. Despite the efforts of both the industry and academia to reduce this overhead [114, 67, 150, 116], nested virtualization is recommended, by cloud providers such as Microsoft Azure [18], only for testing, development, and demo. Even if there is some niche utilization of nested virtualization (to realize rootkits [83]), its adoption in production is not currently envisioned by cloud users.

Nested virtualization is usually adopted for two major reasons globally: to enforce isolation within guest environments and to take advantage of hypervisor properties and functionalities. In the first case, when cloud customers need more isolation and deployability within VMs, they prefer deploying containers (e.g., Docker [6]), which provide better performance and a rich ecosystem (e.g., Kubernetes [16]). In the second case, among the hypervisor tasks often valued are the working set size estimation (for overcommitment) and the memory page tracking. Memory page tracking is at the heart of several tasks, such as checkpoint/restore [163] (for recovery after failure) and live migration [82] (for maintenance and dynamic packing). When these tasks are performed by the nested hypervisor, due to traps (see Figure 1.2) between nested levels and the root hypervisor, even the ideal VM-based nested virtualization solution leads to a higher overhead than container-based nested virtualization because containers originally outperform VMs. Instead of emulating full virtual hardware inside a VM to support a hypervisor, we propose in this thesis, to leverage hardware support for virtualization to provide guest operating systems with the ability to perform the same tasks that were intended by the nested hypervisors, with much less overhead than native nested virtualization mechanisms.

Following the trend of virtualization and its adoption by the industry, processors manufacturers started releasing, in 2005/2006, hardware with technology extensions to enhance virtualization and reduce its overhead. These efforts have been the way to further extensions for nested virtualization, such as VMCS shadowing (2010 [67]) for x86 architectures and Nested Virtual-

ization Extensions (NEVE, 2017 [116]) for ARM. These extensions redirect some guest hypervisor instructions in order to reduce traps and exit multiplication (see Figure 1.2) while preserving isolation (for example, by limiting the number and type of direct privileged instructions allowed).

A myriad of functionalities have emerged since the advent of hardware assisted virtualization, with the core objective of improving more and more the efficiency of hypervisors and then that of virtualization. In this thesis, we establish the first-ever categorization of these functionalities (especially Intel ones) into two main groups. (G_1) Features that facilitate resource multiplexing. G_1 includes EPT (Extended Page Table) [96], SRIOV (Single Root I/O Virtualization) [77], APICv (Intel's Advanced Programmable Interrupt Controller virtualization) [129], etc. (G_2) Features that facilitate and improve VM management tasks realized by the hypervisor, such as Intel Page Modification Logging (PML) [34], CAT (Cache Allocation Technology) [42], SPP (Sub Page write Permissions) [32], etc.

Like a traditional operating system (OS) creates processes and manages them, the hypervisor also creates several virtual machines, typically one for each target OS, and manages them likewise. Because of this similarity, features from G_2's group have the particularity to be able to provide the same facilities to a guest OS as to a hypervisor and, therefore, remove the need for an intermediate hypervisor level to leverage hypervisor properties. For this reason, this class of features is the one targeted in this work.

The main concern here is that all hardware functionalities are hypervisor-oriented. That is, they can be activated, accessed, and managed only by the hypervisor (the L0 hypervisor in Figure 1.1), which is the only component with sufficient rights to manipulate the hardware without any restrictions securely.

1.2 Contributions

***Out of Hypervisor* (OoH).** This thesis introduces OoH, a new virtualization research axis advocating the exposure of individually hypervisor-oriented hardware virtualization features to the guest OS so that its processes can also benefit from those features. OoH aims to have processor vendors rethink the logic of virtualization features and incorporate their categorization and exposure to VMs from their conception and design. For existing features, the OoH logic is to try exposing them using both software and hardware approaches.

Regarding software methods, OoH designers should leverage hypercalls and event channel mechanisms to communicate with the hypervisor that remains the root component; provide the guest with libraries to facilitate the usage of the exposed feature; implement kernel modules to avoid guest OS modification and preserve the privilege of the kernel on multiplexing the exposed feature. Concerning the hardware approach, OoH designers can take

advantage of VMCS shadowing, originally included for nested virtualization, to reduce the hypervisor intervention. Finally, and only when significantly improving performances, hardware changes can be considered to even more remove the hypervisor in the guest execution path. With the latter option, security considerations must be taken into account to keep ensuring isolation between VMs and vis-à-vis the hypervisor.

OoH can be implemented for both unprivileged and privileged VMs (respectively, *dom0* and *domU* in Xen). This thesis illustrates the soundness of OoH in both environments by leveraging Intel PML and Intel SPP for working set size estimation of VMs, checkpoint/restore, garbage collection, and memory vulnerability prevention applied to buffer overflow mitigation in guest userspace.

OoH in dom0. Intel page modification logging (PML) is a hardware feature introduced in 2015 for tracking modified memory pages of virtual machines. Although initially designed to improve VMs checkpointing and live migration, we present in this thesis how we can use and extend this virtualization technology to efficiently estimate the working set size (WSS) of a VM. To this end, we primarily conduct the first study of PML with the Xen hypervisor to investigate its performance impact on VMs and the accuracy of a WSS estimation system that relies on the current version of PML. Our three main findings are that PML reduces both VM live migration and checkpointing, and slightly reduces the negative impact of live migration on application performance. Furthermore, we also observed that a WSS estimation system based on the current version of PML provides inaccurate results, and write-intensive applications are negatively impacted when using PML to estimate the WSS of a VM that runs these applications. Based on these findings, we introduce page reference logging (PRL), an extended version of PML that is more suitable for WSS estimation. We propose a WSS estimation system that leverages PRL and shows how it can be used in a data center that practices memory overcommitment. We implement PRL and the underlying WSS estimation system under Gem5, a popular open-source computer architecture simulator. Evaluation results validate the accuracy of the WSS estimation system and show that PRL does not incur more performance degradation on users' VMs.

PRL has been published at VEE'2021 [74] and its artifacts are available on GitHub [73].

OoH in domU. In domU, we focus on dirty page tracking (OoH for PML) and memory vulnerability prevention (OoH for SPP).

OoH for PML. Dirty page tracking is at the heart of many essential tasks, including process checkpointing (e.g., CRIU [5]) and concurrent garbage collection (e.g., Boehm GC [9]). We use OoH to expose PML for accelerating

these tasks in the guest. We present in this work two OoH-based solutions, namely Shadow PML (SPML) and Extended PML (EPML), that we integrated into CRIU and Boehm GC. SPML follows the OoH software approach and brings no modification to the hardware. Conversely, EPML makes minimal changes to virtualization extensions in order to overcome SPML overheads and meet the OoH's expectations in terms of performance.

OoH for PML has been published at SC'2022 [75], and its source code is available on GitHub [72].

OoH for SPP. Memory safety violations in C/C++ often lead to a buffer overflow, reported as the top vulnerability in 2022. Secure memory allocators are generally used to protect systems against attacks that may exploit buffer overflows. Existing allocators mainly rely on two types of countermeasures to prevent or detect overflows: canaries and guard pages, each with its own pros and cons in terms of detection latency and memory footprint. For virtualized cloud applications, this thesis adopts OoH to introduce GuaNary, a novel safety guard against overflows allowing synchronous detection at a low memory footprint cost. To this end, GuaNary leverages Intel SPP, which allows to write-protect guest memory at the granularity of 128B (namely, sub-page) instead of 4KB. We implement a software stack, LeanGuard, which promotes the utilization of SPP, from inside virtual machines, by new secure allocators that use GuaNary.

OoH for SPP has been submitted and is under review at SIGMETRICS'2023. Its source code is also available on GitHub [71].

1.3 Scientific Publications

International Conferences.

- **S. Bitchebe** and A. Tchana, "Out of Hypervisor (OoH): Efficient Dirty Page Tracking in Userspace Using Hardware Virtualization Features", *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC22.

- **S. Bitchebe**, D. Mvondo, L. Réveillère, N. de Palma, and A. Tchana, "Extending intel pml for hardware-assisted working set size estimation of vms," *in Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE 2021.

International Workshops.

- **Stella Bitchebe**, Alain Tchana and, Laurent Réveillère, "Study of Intel PML Effectiveness", *the 13th EuroSys Doctoral Workshop, 2019.*

Other Publications.

These are publications resulting from parallel work that is not part of this dissertation.

- Tu Dinh Ngoc, Bao Bui, **Stella Bitchebe**, Alain Tchana, Valerio Schiavoni, Pascal Felber, and Daniel Hagimont, "Everything You Should Know About Intel SGX Performance on Virtualized Systems", *In Proceedings of the 2019 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science, SIGMETRICS'19.*
- K. Nguetchouang, **S. Bitchebe**, T. Dubuc, C. Mar, C. Hubert, P. Olivier, and A.Tchana, *Virtual Disk Snapshot Management at Scale*, CoRR abs/2205.06842, 2022.

1.4 Scientific Awards

The work presented in this thesis has also received many honorific distinctions in the research domain, among which the most outstanding are:

- 2022 Google Scholarship for Women in Computer Science
- 2021 Microsoft Research Ph.D. Fellowship
- 2021 L'Oréal-UNESCO For Women in Science Program
- 2021 NEC Lab Ph.D. Research Fellowship

1.5 Outline

The rest of the document is structured as follows. Chapter 2 introduces some background notions and presents the virtualization features exploited in this thesis. Chapter 3 details the principles of OoH and characterizes it against the state-of-the-art. Chapter 4 presents PRL, the OoH application of PML in dom0 for working set size estimation. Chapter 5 presents SPML and EPML, the OoH application of PML in domU for checkpoint/restore and garbage collection. Chapter 6 presents GuaNary, the OoH application of SPP in domU for buffer overflow mitigation. Chapter 7 discusses the application of OoH to other types of hypervisors. Finally, Chapter 8 concludes the thesis and exhibits future work orientations.

Chapter 2

Background

This chapter presents the necessary background on hardware-assisted virtualization and the hardware virtualization features discussed in this thesis, notably Intel PML and SPP.

Contents

2.1 Hardware-Assisted Virtualization (HAV)

As virtualization has been widely adopted by the industry, processor vendors have started releasing chips with technologies and extensions to provide architectural support for virtualization: these are virtualization technologies or VT. VT intends to address the main sources of overhead and latencies introduced by virtualization. As an example, in 2005 and 2006, Intel and AMD produced processors with Intel VT-x and AMD-V extensions, respectively, to support CPU virtualization and accelerate context switching between the hypervisor and VMs. They later released processors with Extended and Nested page tables technologies (Intel EPT and AMD NPT, respectively). These are hardware page tables added in the Memory Management Unit (MMU) to speed up page table mappings from guest VMs to physical memory. They also proposed Intel VT-d and AMD-Vi extensions to enable I/O direct access from VMs.

As part of this work, we are mainly interested in VT from Intel, who will further propose other virtualization features based on these key extensions,

such as Page Modification Logging (PML) and EPT-based Sub-Page write Protection (SPP).

2.1.1 CPU Virtualization: Intel VT-x

VT-x is a set of Virtual Machine Extensions (VMX) that enable the virtualization of the processor hardware. This need to virtualize the CPU originates from the necessity to control and monitor guest instructions when it *owns* the CPU. Before the advent of hardware support for virtualization, guest OSes were modified [1] not to allow a virtual context to execute privileged instructions directly on the processor. This modification implied demoting guest OS' privileged level of execution (*cpl* or *ring*) from 0 to 1. VT was hence driven by the need not to change the semantics of legacy OSes.

Following this goal, VT-x introduces two new processor execution modes, *vmx root* and *vmx non-root*, and replicates for each one the original states of the processor, as depicted in Figure 2.1. When the VM executes, the processor enters in vmx non-root mode, allowing the guest OS and its processes to run at the privileged level for which they were originally designed, exactly as on bare metal. Because it is aware of being virtualized, the CPU knows which instructions the guest is permitted to perform. If the latter attempts to execute non-authorized privileged instructions, the CPU exits from the VM and traps into the hypervisor: this vmx transition is called a *VM exit*. Before switching to the hypervisor, the CPU enters vmx root mode. Only the hypervisor runs in vmx root mode. VT-x comes with several VMX instructions that act on the Virtual Machine Control Structure (VMCS).

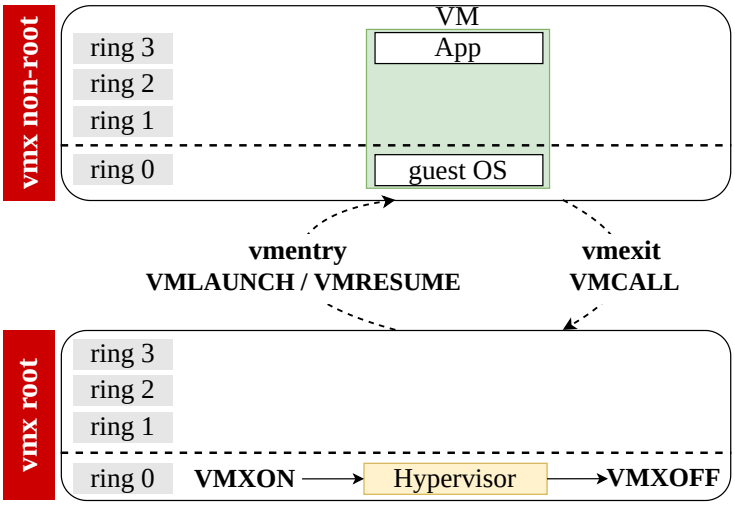


Figure 2.1: VMX execution modes and transitions.

[1] *Paravirtualization* [155].

2.1.2 Virtual Machine Control Structure (VMCS)

VMCS is an in-memory data structure that allows the hypervisor to manage VMX transitions and processor behavior in vmx non-root mode. The hypervisor may maintain several *active* VMCSs, typically one per virtual CPU (vCPU) for each running virtual machine. When the hypervisor schedules a VM on a physical CPU, it loads the address of the corresponding VMCS (using `VMPTRLD`) to the *VMCS pointer* register. This VMCS then becomes the *current* one: only one active VMCS can be current on the processor at a given time. All instructions targetting the VMCS apply to the current one. For example, `VMREAD` and `VMWRITE` instructions will read and write to the current VMCS.
The VMCS data is organized in areas among which some relevant are:

- **Guest-state and Host-state areas.** Context switching often implies saving the context or state of the processor to be able to resume execution from the same point. This is what happens when the processor switches between vmx root and non-root modes. On VM exits, it saves its state in the **guest-state** area of the VMCS before leaving vmx non-root mode and loads it from the **host-state** area before entering vmx root mode. On VM entry, it reloads its state from the guest-state area. Related fields are, for instance, control registers (e.g., `cr3`), interrupt descriptor table (`idt`) register, etc.

- **VM-execution control.** Fields in this area control the execution of the processor when in vmx non-root mode and allow determining operations that may generate VM exits. This area comprises, for example, flags that govern the handling of interrupts (i.e., which interrupts and asynchronous events may cause VM exits), I/O operations, etc.

- **VM-exit and VM-entry controls.** These areas dictate the behavior of VM exits and VM entries, respectively. Related fields essentially hold information on MSR registers to be stored or loaded on VM exit and VM entry.

Each VMCS contains, in the guest-state area, a field called the VMCS link pointer that points to another VMCS. The latter is called a ***shadow*** VMCS, and the former is an ***ordinary*** one. A VMCS is exclusively either ordinary or shadow. The notion of VMCS shadowing was introduced in 2013 by Intel to support nested virtualization (i.e., deploying a hypervisor on top of another one). This said, a shadow VMCS is intended to be manipulated by software not running in vmx root mode, which implies restrictions on VMX instructions. Indeed, a shadow VMCS cannot, for example, be used for VM entry. Any attempt to execute a `VMPTRLD` instruction on a shadow VMCS will cause a VM exit. However, Intel has extended the VMX ISA to authorize

`VMREAD` and `VMWRITE` instructions in vmx non-root mode to allow the guest hypervisor to read and write to shadow VMCSs.

2.1.3 MMU Virtualization: Intel EPT (Extended Page Table)

In traditional OS, memory is already virtualized to ensure process isolation. Each process manipulates so-said virtual addresses (VA) that are translated upon memory accesses into real physical addresses (PA). The OS maintains VA - PA mappings in data structures called page tables (PT). In guest OSes, the same principle applies except that VM's physical addresses (gPA -guest physical address-) are still virtual from the hypervisor point of view and therefore need to be translated once before accessing real RAM memory (hPA -host physical address-).

Before VT, the hypervisor was maintaining translations between guest virtual memory (gVA -guest virtual address-) and physical memory in a PT called shadow page table (SPT). This is because, in this context, the processor does not know it is virtualized. So, on page fault (#PF), it reads the `CR3` register as on bare-metal and walks a unique PT. Here, the `CR3` register points to the SPT. As a consequence, any attempt by the guest OS to update its PT is captured by the hypervisor to also update the SPT (i.e., gVA - hPA mappings). This, obliviously, was a heavy source of latency.

The Extended Page Table overcomes this by adding a second translation level that maintains gPA-hPA translations and a new `CR3` register called *nested `CR3` (nCR3)*. With EPT, the processor is aware of virtualization. Therefore, if a #PF occurs when the CPU is in vmx non-root mode, it knows it has two PTs to walk: we talk about *2-Dimensional (2D)* page walk. The processor first reads the `CR3` register that points to the guest PT, walks the guest PT to obtain the gPA, and then reads the `nCR3` register that points to EPT. It finally walks the EPT to obtain the RAM entry targetted by the #PF. If the processor does not find the mapping for a given gPA, the hypervisor simply updates the EPT independently from the guest. This way, the hypervisor no longer needs to trap guest PT updates.

Intel has further added access and dirty flags (A/D bits) for EPT, which are set when a guest page is accessed and modified, respectively. These bits will open the way to additional EPT-based features such as Page Modification Logging and Sup-Page write Permissions.

2.2 Intel Page Modification Logging (PML)

PML was introduced in 2015 to extend the capability of a hypervisor, allowing it to efficiently track or monitor the guest's physical memory by logging all pages (guest physical addresses, GPAs) modified by a VM during its execu-

tion. The hypervisor can take advantage of PML for its administrative tasks, such as VM live migration or VM checkpointing, which are essential in cloud environments as they respectively facilitate maintenance and enable recovery after failure. Instead of using the traditional write-protection technique of invalidating a page so that the next accesses to it trigger a page fault, the hypervisor can now simply collect addresses logged by PML. PML relies on EPT's A/D bits (§2.1.3) and requires specific changes to the VMCS(§2.1.2).

2.2.1 Changes to VT-x

To support PML, a new 64-bit field is introduced in the VM-execution control area of the VMCS, named **PML address**. PML address points to a 4KB aligned physical memory page called **PML buffer**. This buffer is organized into 512 64-bit entries that store the logged GPAs.

A new 16-bit field is added to the guest-state area of the VMCS called **PML index**. PML index is the logical index of the next entry in the PML buffer. Because the buffer includes 512 entries, the PML index has values ranging from 0 to 511, starting at 511. When PML is enabled, each `write` instruction that sets a dirty flag in the EPT during a page walk triggers the logging of the corresponding GPA (concerned by the dirty flag entry). The PML index is then decremented by 1. When it goes below 0, the PML buffer is considered full.

A new VM exit reason is added to the VM-exit information fields with the name **page-modification log full**. Whenever the PML buffer is full, the processor triggers a VM exit, and the hypervisor comes into play. The logging process restarts after the PML index is reset to 511. The actions taken by the hypervisor in response to this VM exit depend on its objective and needs.

2.2.2 Functioning

Figure 2.2 illustrates the general workflow of PML when used to improve a virtualization operation (e.g., live migration). The figure shows, on the one side, the user's VM (in green) targeted by the virtualization operation, and on the other side, the dom0 that runs the system implementing this virtualization operation. The execution of that system generally begins with the activation of PML for the target user's VM ✪. Then, the CPU of that VM can start logging GPA ❶. When the *PML logging buffer* is full, the CPU triggers a VMExit trapped into the hypervisor ❷. The handler of that VMExit performs a certain task (e.g., copying the content of the *PML logging buffer* to a larger buffer that is shared with the *dom0* ❸). Then, the *PML index* is reset to 511, and the VM resumes (VMEnter). The system implementing the virtualization operation (in the dom0) operates periodically on the results generated by the log full handler ❹. This is done as part of the virtualization

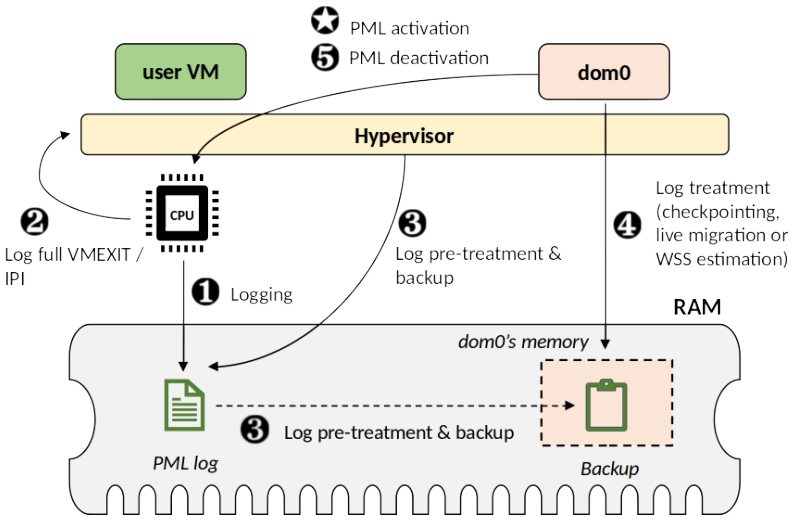


Figure 2.2: General functioning of PML.

operation, e.g., remigrate dirty pages in case of live migration. When the virtualization operation is complete, PML is disabled ❺.

2.3 Intel Sub-Page write Permissions (SPP)

SPP was introduced in 2018, and it builds on top of EPT. It refines guest memory accesses, allowing the hypervisor to write-protect guest pages at a sub-page (128B) granularity instead of the traditional 4KB.

2.3.1 Changes to VT-x

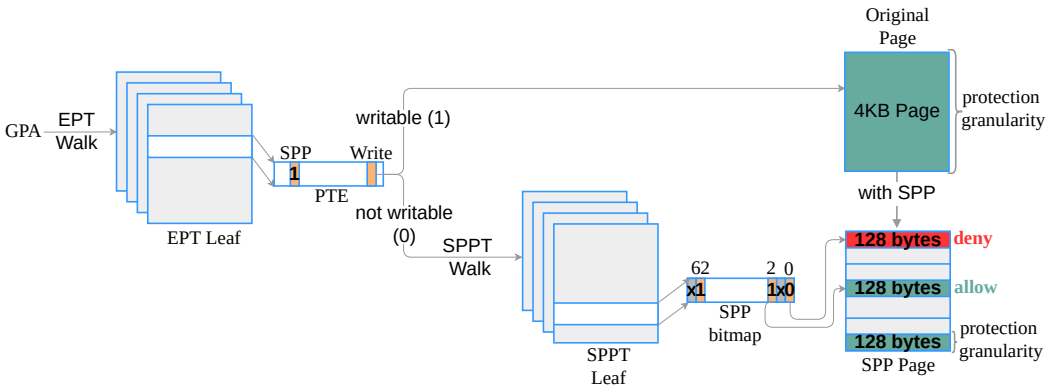


Figure 2.3: General functioning of SPP.

Figure 2.3 highlights the main changes introduced by SPP. SPP defines a new flag in the EPT, called the **SPP bit**, which controls the use of the

functionality. If the bit `write` in the EPT's last level is unset (i.e., the guest page is write-protected) and the SPP bit is set, this indicates to the processor that SPP is enabled on the page concerned.

The bitmap indicating which sub-pages are write-protected on a page is called an **SPP vector** or **SPP bitmap**. SPP bitmaps are maintained using a new data structure called the **SPP table** (SPPT). SPPT is a 4-level radix tree in which leaf tables contain the SPP bitmaps (64 bits) that are configured by the hypervisor. Only even bits of the bitmap are interpreted by the processor, leading to 32 sub-pages per 4KB-page.

In the VM-exit information area of the VMCS, new VM exit reasons are added and called **SPP-related events**, such as SPPT misconfigurations.

2.3.2 Functioning

If the hypervisor wants to protect sub-pages for a given guest physical address `g`, it must first take the following actions: build the desired protection bitmap; find the EPT entry that maps `g` to the RAM; set its WRITE flag and the SPP flag to 0 and 1 respectively and, modify the SPPT entry corresponding to `g` according to its needs. For example, the following bitmap 1X0X... tells that the first sub-page is write-accessible (1) while write-access to the second sub-page is denied (0). X means that the bit is ignored. In addition, the first upper bit of the bitmap represents the first lower sub-page on the page.

SPPT extends the traditional page table walk (PTW) process occurring on TLB miss. At the end of the EPT walk (i.e., on the EPT leaf), if the processor sees the WRITE flag and the SPP flag set to 0 and 1, respectively, it also walks the SPPT. If the target sub-page is write-protected, the processor raises a VM exit; otherwise, the instruction is authorized (see Figure 2.3).

Chapter 3

OoH: Out of Hypervisor

This chapter presents the first categorization of hardware virtualization features into exportable ones or not. The chapter further explains the OoH principle and distinguishes it from state-of-the-art efforts for nested virtualization.

Contents

3.1 Hardware Features Categorization

Virtualization features can be categorized into two groups: multiplexing and management.

(G_1) Features destinated to the multiplexing of resources are intended to accelerate and ease the virtualization. G_1 includes EPT, SRIOV (Single-Root I/O Virtualization), APICV (Advanced-Programmable Interrupt Controller Virtualization), etc. These features are not relevant for exposure to the guest OS in the spirit of OoH.

(G_2) Management features are intended to facilitate and improve VM management tasks realized by the hypervisor, like migration of VMs, checkpointing, etc. G_2 comprises features such as Intel PML, CAT (Cache Allocation Technology), SPP (Sub Page write Permission), PT (Processor Tracing), and so forth. Considering a VM as a process, it makes sense to expose G_2's features to the guest, thus allowing it to improve its management tasks that are, in some manner, similar to that of the hypervisor vis-à-vis virtual machines.

This thesis focuses on illustrating OoH with Intel PML to improve dirty page tracking in virtualized clouds and Intel SPP to improve secure heap memory allocators that run from inside VMs.

3.2 OoH Principles

OoH advocates the exposure of some hardware virtualization features to make them usable within the guest OS. The goal of this new research axis is to make CPU providers rethink the logic of virtualization features invention. Researchers must incorporate categorization (Section 3.1) and exposure, to guest applications, of new functionalities at the conception and design stages. For existing features, a key point of the OoH principle is to export them with minimal changes in the hardware, the hypervisor, and the guest kernel. In addition, an OoH-based solution must propose a simple utilization interface to application developers.

OoH argues for software and hardware approaches. Obviously, the latter should be envisioned only when the former is not efficient. When followed, the hardware approach should try to re-use as maximum as possible existing functionalities before thinking of any changes. To expose a given hardware virtualization feature, the OoH principle is as follows. To facilitate the exploitation of the exposed feature, OoH designers should provide userspace applications with a library. The latter should rely on a guest kernel module that preserves the privilege of the kernel on multiplexing the exposed feature. OoH designers use hypercalls and event channels as communication mechanisms between the hypervisor and the guest. Hypercalls allow the guest (OoH kernel module) to instruct the hypervisor (feature initialization for instance) while event channels allow the hypervisor to send a specific signal to the guest. VMCS shadowing, invented by Intel for improving nested virtualization, can be leveraged to implement OoH. It allows to reduce the hypervisor involvement, thus improving performance. Finally, and only when unavoidably necessary, some hardware changes can be made such as ISA extension or VMCS data structure modifications.

3.3 OoH Positionning against Dune

The evocation of virtualization features exploitation in user space often makes one think first of Dune. Therefore, one may legitimately ask what the difference is between OoH and Dune [64].

Dune leverages hypervisor-oriented hardware virtualization features (such as Extended Page Table) to make privileged instructions, usually only accessible in kernel mode (ring 0), available to processes. The Dune system architecture, presented in Figure 3.1, clearly differentiates it from OoH in the following aspects. First, Dune cannot be launched in a VM because it needs to be in vmx root ring 0 to be able to exploit hypervisor-oriented hardware virtualization features. Thus, Dune does not relate to nested virtualization. Second, contrary to OoH, Dune cannot make those virtualization features

available to the guest userspace since a Dune process executes in vmx non-root ring 0 and not ring 3. In contrast, Dune processes may instead take benefit from OoH.

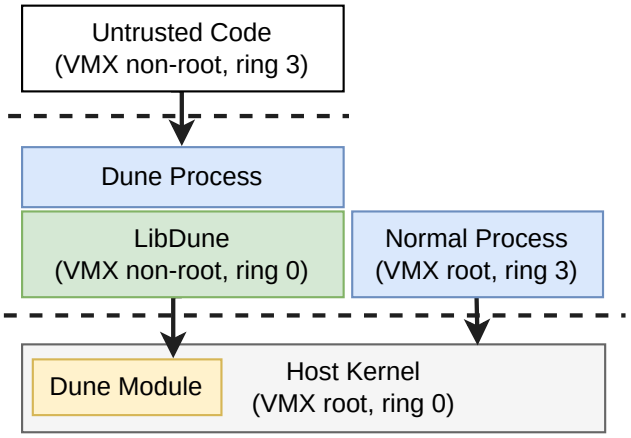


Figure 3.1: Dune system architecture.

3.4 OoH vs. State-of-the-art

Many works have tried to reduce the overhead of virtualization in general, and nested virtualization in particular. Some works completely moved the hypervisor to the hardware, while others proposed solutions close to the OoH spirit. The OoH's idea of giving virtual machines a direct access to the hardware is not new and dates back to 1970s [110, 66].

As Figure 3.2 suggests for nested virtualization, none of the solutions removes the nested hypervisor from their architecture (and therefore still need a double emulation that will incur a non-negligible overhead). In addition, these solutions are mainly narrowed to I/O passthrough and do not allow guest applications to benefit from virtualization features. Conversely, OoH materializes as a kernel module, which is lighter.

And regarding hardened hypervisors, whose architecture is drawn in Figure 3.3, they either make resource sharing unpractical or make the hypervisor rigid, since any update will require to change the hardware. OoH, on the other hand, does not impact resource overcommitment and updates are simpler as just recompiling and reloading a kernel module.

3.4.1 Hardened Hypervisors

The two popular systems that have tried hardnening the hypervisor are NoHype [104] and AWS Nitro [140].

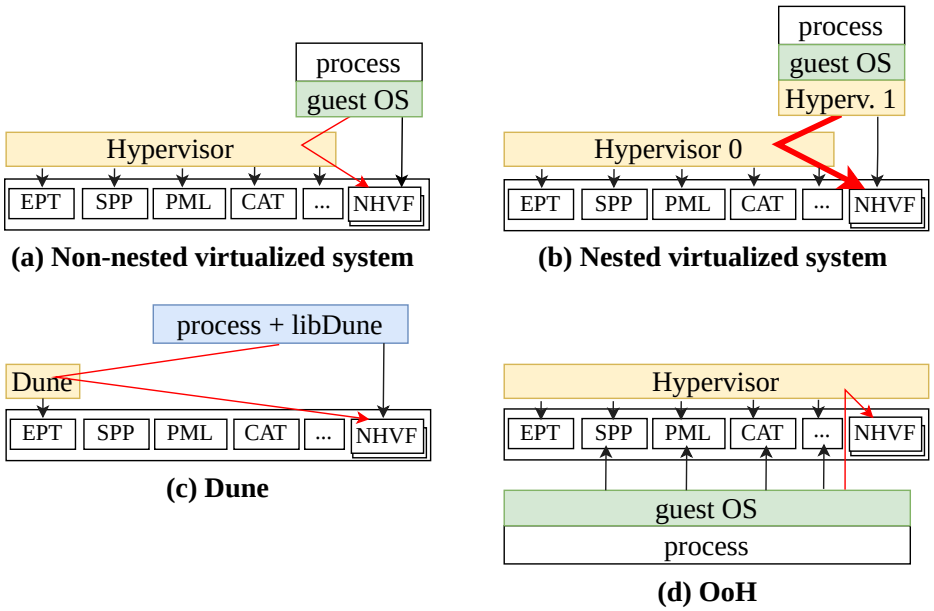


Figure 3.2: The position of OoH (d) in the virtualization landscape: (a) non-nested virtualization, (b) nested virtualization (including DVH [114]) and (c) Dune. Red arrows materialize VM traps. Its width indicates the intensity of VM traps. NHVF stands for Non Hardware Virtualization Feature.

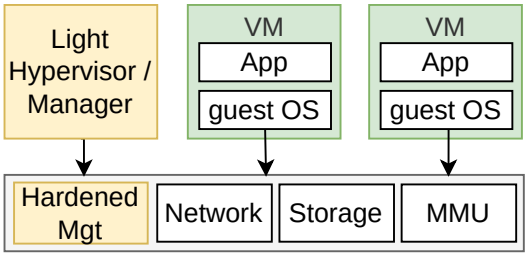


Figure 3.3: Hardened hypervisor general architecture.

NoHype [104]. NoHype was introduced in 2010 by Eric Keller et al. NoHype removes the hypervisor layer from the virtualization architecture and makes the VMs run directly on the hardware, with the goal of providing near bare-metal performances. It, however, provides a lighter software management component responsible, for example, for starting and stopping VMs.

Once a VM is booted in the NoHype architecture, it is never interrupted (i.e., no more VM exits) and directly accesses the devices. Since the VM's scheduling is ensured by the hypervisor, by removing the latter, NoHype eliminates the possibility of scheduling VMs among cores. So, it rather dedicates one CPU core per VM, which implies that cores can no longer be shared among guests.

NoHype partitions the RAM and assigns a precise and dedicated portion

of the memory to each VM. Therefore, the memory allocated to a VM cannot be modified until the VM stops, which makes overcommitment impossible, while it is one of the advantages of virtualization. In addition, the hardware is modified to translate itself, guest physical to host memory (which was the role of the hypervisor).

Finally, NoHype gives VMs direct access to the network and assigns to each VM its own physical device with limited and controlled access to the I/O bus. However, if there are not as many devices as VMs, this is in practice not possible.

Because NoHype is based on a statical allocation of resources, it precludes overcommitment and restrcits the number of instances that can be allocated. Furthermore, it makes impossible any security or management update of the hypervisor because this will require to completely change the hardware, which is almost impossible in production. All these reasons intrude on NoHype adoption.

AWS Nitro [140]. Amazon launched AWS Nitro in 2017 with the aim of providing *bare metal-like performance*. The Nitro system has three main components.

Nitro cards: Nitro replaces the main emulation functions of the hypervisor by corresponding cards that provide controllers for networking and storage. Nitro security chip: since Nitro gives instances direct access to the hardware, a specific security chip is incorporated into motherboard to provide security features such as secure boot. Nitro hypervisor: alike NoHype, even if moving most hypervisor functionalities to the hardware, the Nitro system, however, keeps a tiny part of the software that is responsible for memory and CPU allocation and VM management.

Nitro is specific to Amazon and is not open source, and as for NoHype, it makes the hypervisor rigid and may require complet hardware change in case of updates.

3.4.2 Nested Virtualization

Recursive Virtual Machine Architectures [110, 66]. H. c. Lauer and D. Wyeth stated that the need for a central supervisor (i.e., the hypervisor) is mainly to protect the guest from accessing unauthorized registers or memory spaces. So, like G. Belpaire and N. Hsu, they proposed an architecture for nested virtualization in which a stack of registers is reserved for each level of VM. This way, the processor is able to handle interrupts (and direct them directly to the controlling level) and access memory for each nested level without the need to go through the central hypervisor.

In these architectures, any virtual machine is allowed to create a virtual memory within its own virtual memory, called *virtual-virtual* memory and

represented as a *segment table*. The latter defines how the memory segments are mapped between the creator and the created processes. The address of the segment tables is stored in the stack of registers, and to determine the physical address corresponding to virtual memory at a given level, the processor merely composes the mappings of each of the segment tables.

The stack of registers proposed by [110] and [66] is comparable to the actual VMCS shadowing and interrupt redirection is similar to the Intel virtual-IPI that OoH leverages in its current implementations.

Fluke [91]. B. Ford et al. proposed the Fluke architecture that replaces the notion of nested VM with that of nested processes and introduces a new ISA (Instruction Set Architecture) that substitutes the traditional hypervisor to a microkernel by modifying the software stack at all levels. The Fluke microkernel then decomposes the OS features into modules to provide the nested processes with only the functionalities needed and thus lightening the cost of recursive VMs. Furthermore, the Fluke microkernel does not emulate the hardware. Instead, Fluke modifies the *syscall* API to allow all processes (nested or not) to execute directly on the hardware.

Like Fluke, OoH provides guest VMs with functionalities instead of booting a complete OS. However, OoH keeps the basic and traditional scheme of virtualization whereas, Fluke cannot run traditional unmodified operating systems and hypervisors.

Dichotomy [158] D. williams et al. splitted the hypervisor into two main components: hyperplexor, that multiplexes the hardware, and featurevisor, that implements the hypervisor functionalities. Based on this distinction, they proposed Dichotomy, a new architecture in which the root/L0-hypervisor is hyperplexor, and featurevisor acts as nested hypervisor. Dichotomy makes the nested hypervisor (featurevisor) to temporarily transfer the control and management of nested VMs to hyperplexor. The guest memory is then mapped and remapped when switching between featurevisor and hyperplexor, which will still incur non-negligible overhead.

Direct Virtual Hardware (DVH) [114]. DVH, by Lim et al., proposed that the host hypervisor provides virtual devices directly to nested VMs without the intervention of intermediate hypervisors. The intermediate hypervisors only intervene at virtual device initialization time to make it visible and directly accessible to the nested VM. The authors illustrated DVH with four devices: virtual IO, virtual timer, virtual IPI, and virtual idle. Although DVH is promising, its application to all devices that compose full hardware is unpractical. With OoH, we are advocating for exposing only hardware virtualization features that could help applications, making OoH more tractable.

SMT-based VirTualization (SV_T) [150]. L. Vilanova et al. have leveraged resources provided by simultaneous multithreaded (SMT) processors to propose SV_T. The SV_T mechanism consists of executing each virtualization level (i.e., main hypervisor, nested hypervisors, and nested VMs) on different hardware threads of a hardware core. This allows for eliminating the heavy context switches of VM traps. SV_T does so by making some changes to the VMCS and VMX operations to capture all VM trap and VM resume events and replacing them with *cross-context register accesses*. SV_T, therefore, simply makes a virtualization context-level accessing the registers of context-level targeted by the switch.

However, by pinning each layer to a single core, SV_T limits resource sharing and overcommitment and changes the VM traps costs by that of register context switches. And contrary to OoH, the SV_T architecture does not remove the need for the nested hypervisor.

Nested Virtualization Extensions (NEVE) and VMCS Shadowing [116]. The VMCS shadowing concept was introduced by the Turtles Project [67] in 2010 and first released on Intel processors in 2013. VMCS shadowing was the first architectural effort for nested virtualization, with the goal of reducing the inference of the root/L_0 hypervisor. With VMCS shadowing, a nested hypervisor has its own dedicated VMCS that it can manage using some dedicated VMX instructions without trapping to L_0. However, a shadow VMCS makes accessible to the guest hypervisor only a few fields, thus still limiting its *autonomy*. NEVE is ARM support for nested virtualization introduced by J.T. Lim et al. in 2017. NEVE is similar to VMCS shadowing. They both share the same basic idea of redirection to reduce exit multiplication and traps from guest hypervisors.

Part I

OoH in Privileged VMs (Xen-dom0)

Chapter 4

Page Reference Logging (PRL): Efficient Hardware-Assisted Working Set Size Estimation of VMs

This chapter presents PRL, an application of OoH for privileged domains (dom0 in Xen) in virtualized clouds. PRL extends PML to provide the dom0 with an efficient working set size (WSS) estimation system that does not affect guest applications.

Contents

4.1 Introduction

Memory page tracking is at the heart of several essential tasks in cloud environments, such as checkpointing [163] for recovery after failure, live migration [82] for maintenance and dynamic packing, and working set size (WSS)[1] estimation [86] for memory overcommitment [44], and fast restore [163].

[1]The working set refers to the set of memory pages a process/OS/VM is using at a given time[86].

WSS estimation is an essential task for data center operators because it allows, among other things, memory overcommitment [131, 133] (by periodically adapting the memory size of the VM according to its real needs), fast restore [163] and efficient processor cache partitioning [40]. Regarding memory overcommitment, for example, its implementation and adoption are necessary for the following reasons. First, VM owners use to over-estimate resources [85, 102] for their tasks. Jyothi et al. [102] analyzed the resource reservation for a 50k-node production data center and found that 75% of jobs were over-provisioned (even at their peak), with 20% over ten times over-provisioned. E. Cortez et al. [84] made similar observations in recent traces of the Microsoft Azure cloud. Second, some cloud-native workloads are fundamentally based on the dynamic management of overcommitted VMs. In particular, this is the case for serverless or function as a service (FaaS) systems, whose design and pricing model are intrinsically linked to an aggressive packing of hundreds or thousands of micro-VMs on the same physical machine [76, 152]. Third, memory is a finite resource whose evolution does not follow that of other resources (especially the CPU), so researchers talk about the problem of the memory wall [117, 143].

The most widely used approach for WSS estimation through memory page tracking relies on present bit invalidation. Such an approach may lead to severe performance degradation caused by the generated page faults. It is particularly true when a significant amount of pages have to be tracked, as in WSS estimation [131, 79, 151, 100, 120, 106]. We assess this issue using a synthetic application that parses an array. The present bit is invalidated every second for all memory pages of the virtual machine (VM). We measure up to 96.22% of performance degradation for a VM with 1GB memory size.

To overcome the limits of this approach, we can take advantage of hardware virtualization features available for memory page tracking, specifically PML. As presented in Section 2.2, PML allows the MMU to log in a `4KB` page (called the PML logging buffer) in RAM, all guest physical addresses (GPAs) that led to the setting of the dirty bit in the EPT during page walks. In this chapter, we extend PML to make it suitable for WSS estimation. We make the following contributions.

First. We conduct a comprehensive study to assess the effectiveness of Intel Page Modification Logging (PML) and its impact on the performance of user applications. Our main findings are summarized as follows. (1) PML reduces by up to 10.18% the time of both VM live migration and checkpointing. (2) PML slightly reduces the negative impact of live migration and checkpointing on application performance by up to 0.95%, which is not negligible for tail latencies [103]. (3) PML can be used for WSS estimation but needs to be improved to provide an accurate estimation, summarily, because read accesses are not tracked, and hot pages cannot be identified. In fact, the current

design of PML focuses on write workloads. It only logs the guest physical address (GPA) of a page once, even if the page is accessed several times. Therefore, cold pages are likely to be counted in the working set, overestimating the latter, which leads to memory waste. In addition, PML would incur an unacceptable overhead for the VM whose WSS is estimated. Indeed, when the PML logging buffer is full (after 512 GPAs are logged), the CPUs of the VM whose WSS is computed trigger a VMExit. The handler of that VMExit consumes CPU time which is taken from the VM's CPU quota. We measure up to 34.9% of performance degradation. This would not be acceptable for cloud users because they are not the beneficiaries of the WSS estimation, which is executed for the needs of the data center operator.

Second. We introduce Page Reference Logging (PRL), an extended version of PML to track both read and write working sets without impacting VM performances. PRL can be used in two exclusive modes: PRL_{PML} and PRL_{PAML}. PRL_{PML} is similar to the current PML functioning, making PRL effective for live migration and checkpointing. In contrast, PRL_{PAML} focuses on working set size (WSS) estimation. In PRL_{PAML} mode, read accesses are taken into account, and several recordings of the same page are also possible, allowing hot page tracking. In addition, PRL_{PAML} avoids performance overhead on user VMs for the following reason. VMExits related to PRL_{PAML} are redirected to the *dom0*[2] CPUs. *Dom0* is responsible for running VM administration services. Therefore, it makes sense to use it to host WSS estimation computations since the data center operator is the main beneficiary of this task. Technically, when the PRL_{PAML} logging buffer is full, the actual CPU (which is running the user's VM whose WSS is computed) sends an interprocessor interrupt (IPI) to one of the *dom0*'s CPU, thus raising a VMExit on it. By redirecting VMExits related to PRL_{PAML} to the *dom0*, the user VM can continue its execution while handling these VMExits (unlike in the current PML design), thus avoiding the negative impact on user VMs. The handler of this IPI identifies hot pages and makes them available to a WSS estimation system. We describe an implementation of PRL in Gem5 [70], a popular computer architecture simulator.

Third. We present a prototype implementation of a WSS estimation system that uses PRL in the context of the Xen hypervisor. Using both real (HPL Linpack [39], BigDataBench [38]) and synthetic applications, we evaluate and compare our solution with an implementation of a software-based solution, precisely VMware's WSS estimation solution described in [151][3], following the same evaluation methodology as the work of Nitu et al. in

[2] *dom0* is the privileged VM (noted pVM) in Xen [61]. In fact most virtualization systems rely on a pVM, *host OS* in KVM [15], *parent partition* in Hyper-V [13] and *Service Console* in VMware [36, 35].

[3] To the best of our knowledge, [151] is the only publication made by VMware concerning their WSS estimation.

SIGEMETRICS 2018 [131]. The evaluation results confirm that: (1) our solution is accurate, (2) our solution has no impact on user VMs, (3) our solution is not intrusive (no modification of the guest OS is required), unlike most state-of-the-art solutions [131, 79, 151, 100, 120, 106].

4.2 PML Study

To better understand PML, we conduct a study to investigate both its effectiveness and its performance impact on user applications. We target live migration and WSS estimation operations. We do not elaborate on checkpointing because live checkpointing, which could benefit from PML, is not implemented in current hypervisors. We carried out the experiments on a laptop with the following characteristics: Single socket Intel(R) core (TM) i7-3768, 16GB memory, 500GB SSD, 4-way 64 TLB entries. We used Xen 4.7 as the hypervisor and Linux 4.15.0 for the guest kernel. For the applications that run inside the virtual machine, we used HPL Linpack [39], BigDataBench [38] (read, write and sort applications, 10GB data set size), and a synthetic application for which the code structure is shown in Listing 4.1. The synthetic application consists in parsing an array several times during a period. Each array entry points to a `4KB` data structure (the size of a memory page). The type of operation (read or write) performed on an array entry is decided according to a write intensity parameter (`wi`) which represents the proportion of write operations. Unless otherwise indicated, the array uses `400MB` of memory, and the VM has one vCPU and 1GB of memory for the synthetic application and four vCPUs and 12GB of memory for the macro benchmarks.

```
1  /* Main variables description
2      * wi: proportion of write operations
3      * tab: array to be processed
4      * PERIOD: duration of each iteration
5      * SIZE: number of tab memory pages
6      * nbOps: # of read and write operations computed
7      * throughput: mean number of operations per nanoseconds
8      * ns: duration in milliseconds
9      */
10 #define PERIOD ...
11 #define SIZE ...//SIZE*size_page=size_tab
12 struct page{
13     unsigned long entry[512];
14 };
15 void synthetic_workload(int wi){ //write intensity
16     /*
17     * declare & initialise variables
18     * nbOps, throughput, ns, ...
19     */
20     void *tab;
```

```
21      struct page *temp;
22      struct timespec start, end;
23      clock_gettime(..., &start);
24      do{
25      posix_memalign(&tab, ...);
26      for(i = 0; i < SIZE; i++){
27          temp = (tab + size_of_page*i);
28          op = rand() % 100;
29          if(op < wi)
30          temp->entry[...] = ...;//write operation
31          else
32          read = temp->entry[...];//read operation
33          nbOps++;
34      }
35      clock_gettime(..., &end);
36      //convert duration (from start to end)
37      //in nanoseconds
38      ns = ...;
39      throughput = (nbOps-nbOps_prev)/(ns-ns_prev);
40      ns_prev = ns;
41      nbOps_prev = nbOps;
42      }while(ns < PERIOD);
43      free(tab);
44  }
```

Listing 4.1: Synthetic application skeleton. Its performance metric is the number of operations per nanosecond.

4.2.1 PML-based VM migration

In Xen, the heart of live migration is mainly implemented through the function `int save()`[4]. The use of PML is thus limited to this memory-saving phase. We compare the use of PML with the classical memory page tracking approach, which consists of write-protecting memory pages so that the following write operations lead to page faults. We use the synthetic application as a baseline for this evaluation because its behavior is predictable compared to the macro-benchmark.

We consider two metrics: *performance*, the performance of the user application during migration, and *duration*, the duration of the `save()` method execution. *performance* checks whether PML reduces or increases the negative impact of these operations on the application, while *duration* indicates whether PML accelerates migration or not. Two successive live migration operations are performed while running the application during a certain period, with different proportions of write operations.

[4] In file `tools/libxc/xc_sr_save.c`

Figure 4.1 and Table 4.1 present the results for *performance*. We observe that live migration, no matter which technique, negatively impacts the performance of the application, as illustrated by the two descending peaks in all curves. However, PML slightly minimizes this impact by 0.06% to 0.95%. The magnitude of the reduction depends on the write intensity of the workload. Indeed, although using PML for a read-intensive workload slightly reduces the application's performance, its advantages are more important for write-intensive workloads.

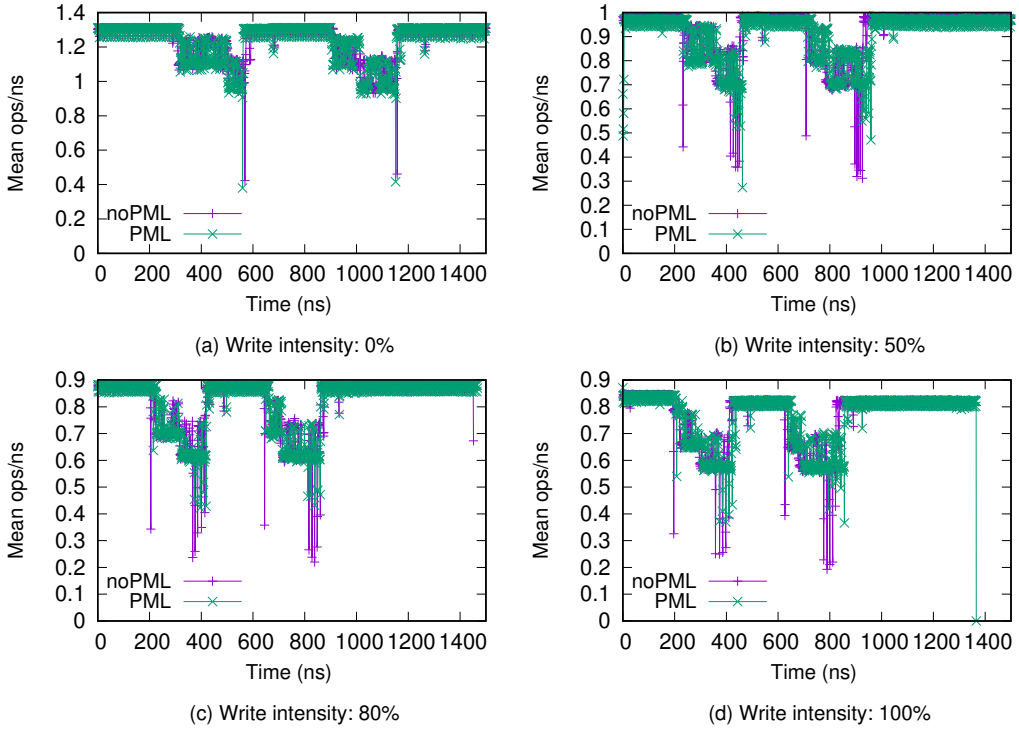


Figure 4.1: Operations per nanosecond while two live migrations are performed. We run this experiment with different write intensity values (for the synthetic application): 0%, 50%, 80%, and 100%.

Write intensity (%)	0	50	80	100
Improvement (%)	6×10^{-2}	0.14	0.39	0.95

Table 4.1: PML benefits during live migration. We compare the performance degradation of the running application during live migration, with and withouth PML.

Figure 4.2 presents the results for *duration*. We observe that PML reduces the execution time of method `save()` during live migration by 0.98% to 10.18%. In particular, read-intensive applications migrate much faster when PML is used. This is due to two main reasons. First, when using PML, if a

page has not been logged (the GPA of the page is not present in the *PML logging buffer*), it has not been modified and can immediately be migrated. The hypervisor no longer needs to invalidate it upstream. Second, as the workload performs fewer write operations, there are fewer logged GPAs, thus less *logging buffer* walk operations by the hypervisor and then a faster migration. Note that it is very important to speed up live migration because it allows you to quickly free a machine for maintenance, quarantine a corrupted VM, etc.

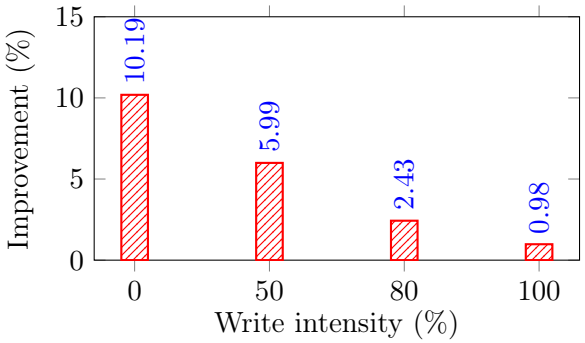


Figure 4.2: Execution time improvement of the method `save()` using PML, compared to the version without PML.

4.2.2 PML-based WSS estimation

A WSS estimation system based on PML would work as follows. Once PML is enabled for the VM, the system collects GPAs until no new GPAs are visible in the logging buffer. The estimated WSS would then be the total number of different GPAs collected. We implemented this system in the Xen hypervisor. However, such a system fails to provide an accurate WSS estimation for the following reasons. First, PML only tracks write WSS (hence the name *page* ***modification*** *logging*). Second, PML does not track hot pages (even the dirty ones). A page is said to be *hot* if it is referenced several times over a short period. According to the current PML design, an accessed page is logged only once. Using this design for WSS estimation, it is not possible to distinguish between *hot* and *cold* pages, which leads to an overestimation of the actual memory requirements of the virtual machine. To evaluate this limitation, we modified the synthetic application by adding a *for* loop at the beginning, which modifies all the entries in the array (xMB). The remaining application code works on a small portion of the array (noted y, with $y < x$). Although the correct value of the WSS estimation is y, the system reports x, thus wasting memory. Third, PML degrades the performance of the VM whose WSS is estimated. In fact, the handling of PML logging buffer full events should not be done by the CPU of the VM whose WSS is estimated.

Indeed, depriving the user's VM of its CPU quota is unfair because the WSS estimation is only beneficial for the data center operator. One could legitimately say that this limitation is also true for live migration. However, it has been proven [132] that a slight reduction in CPU time used by the migrated VM accelerates live migration (as we previously observed in figure 4.2). We measured the overhead of the current PML design to estimate the WSS of applications from BigDataBench [38] (read, write and sort applications) and HPL Linpack [39][5]. For BigDataBench applications, the input dataset is 10GB. We run each application with and without PML and calculate the overhead, as shown in Figure 4.3. Read-intensive applications are not affected by the use of PML because it only tracks page modifications. However, other workloads, such as HPL Linpack, are significantly impacted by the use of PML, with a performance degradation of up to 34.9%.

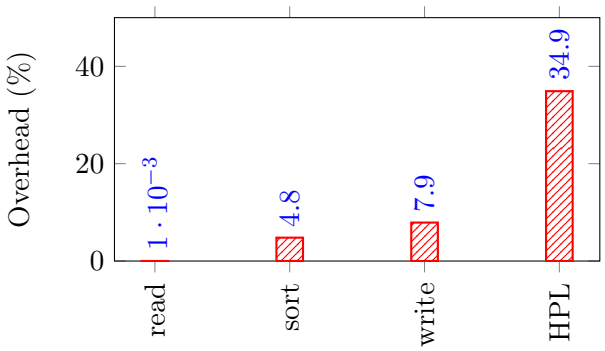


Figure 4.3: Impact of PML when used for WSS estimation. We plot the overhead for applications in terms of FLOps (FLoating point Operations per second).

We summarize the main conclusions of this study on PML as follows. (1) PML reduces VM live migration time. (2) PML slightly reduces the application performance overhead during live migration. (1)we cannot use PML with its current design for efficient WSS estimation because a WSS estimation system based on it will fail to accurately estimate the whole working set of a VM.

4.3 Page Reference Logging

We introduce Page Reference Logging, an extended version of PML, to facilitate efficient WSS estimation.

[5]HPL is a High-Performance benchmark implementation whose code solves a uniformly random system of linear equations

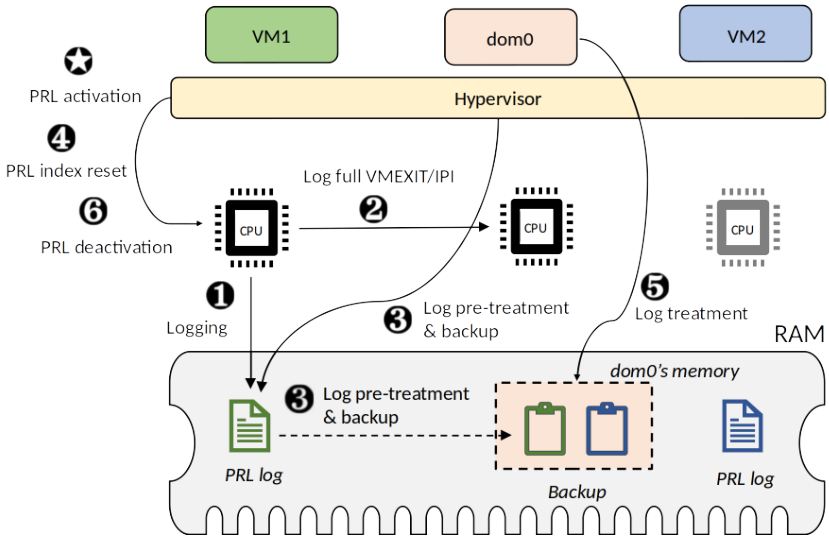


Figure 4.4: Overview of PRL functioning.

4.3.1 PRL functioning

Figure 4.4 illustrates the general functioning of PRL. Let us consider a user's VM ($VM1$ in green), which is the target of the WSS estimation operation, and the *dom0*, which runs the WSS estimation system. The execution of that system generally begins with the activation of PRL for the target user's VM ✪ so that the CPU of that VM can start logging GPAs ❶. When the *PRL log buffer* is full, the VM's CPU sends an IPI to a dedicated *dom0*'s CPU (which will be responsible for computing the working set of the VM) ❷. Then the hypervisor copies the content of the *PRL log buffer* to a larger buffer that is shared with the *dom0* ❸ (while the VM continues its execution with no interruption), and the *PRL index* is reset to 511. After that, the logging process restarts, ❹ and in the meantime, the WSS estimation system operates on the results generated by the log full handler ❺.

4.3.2 PRL architectural design

A processor supporting PRL can be used in two exclusive modes: PRL_{PML} and PRL_{PAML}. The first mode mimics the current PML functioning (§2.2.2), making PRL effective for live migration and checkpointing, as it is for PML. In contrast, PRL_{PAML} focuses on WSS estimation.

As for PML, specific changes are needed in VMX to support PRL. To activate PRL_{PAML}, the system software must set a new bit of the *Secondary Processor-Based VM-Execution Controls* (see §2.1.2) called *PRL enable* (similarly to setting bit 17 of the *Secondary Processor-Based VM-Execution Controls* for PML activation). A new 16-bit host-state field called *log full handler CPU* indicates the index of the CPU to which an interrupt is sent when

the PRL log buffer is full. A new 8-bit host-state field called *log full vector* indicates the interrupt vector that will be executed by the target CPU upon reception of a *log full* interrupt. This destination CPU must belong to the dom0, which serves as the execution room of the WSS estimation system (Fig. 4.4, ❷ - ❺). In this way, PRL_{PAML} avoids scheduling out the VM whose WSS is estimated. Remember that the dom0 belongs to the data center operator, so using it for WSS estimation makes sense. For each GPA which is the input of the PRL process (which begins at EPT walk on TLB miss), the following algorithm takes place:

1. *PRL index* is initialized to 511.
2. If the *PRL index* value is 0, this means that the PRL log buffer has been detected as full. In such a case, the *PRL index* is decremented, and an interrupt is sent to the processor of the dom0, which is responsible for handling log full events. The PRL process ends without interrupting the VM whose WSS is estimated. The processor restarts the logging when the *PRL index* is reset by the system software.
3. If the *PRL index* is negative, it means the log full event handler is running. The GPA is, therefore, not logged, and the PRL process ends. Some may argue that the PRL process will miss some GPAs when processing the event handler. However, it does not affect the WSS estimation because if a missed GPA belongs to the working set, it is likely to be seen in the near future (after reactivation of the PRL mechanism) as it is hot. Otherwise, the GPA is cold, and its loss does not change the WSS estimation. The results of the evaluation described in Section 4.4 support our claim.
4. Otherwise, the *PRL index* is positive (and less than 511) and is decremented, and the GPA is logged. This is done regardless of the value of the dirty flag (in contrast to the current PML, where the GPA is logged only if the dirty bit is set). In this way, PRL_{PAML} can log both accessed and modified pages. Besides, PRL_{PAML} can log the same page access several times.

4.3.3 PRL log full event handling

When the PRL log buffer is full, the processor sends (through its LAPIC[6]) an IPI to the CPU of the dom0 that was designated at VMCS configuration time. The LAPIC is configured with the identifier of the target CPU. We have introduced a new interrupt vector that points to the log full event handler. This mechanism is similar to *Lightweight inter-core notifications* introduced

[6]Local APIC (Advanced Programmable Interrupt Controller)

by Jeffrey C. Mogul et al. [125]. Since the log full event handler must run in the VMX root mode because it processes VMCS data structures, the target processor must trigger a VMExit upon receipt of the IPI. This behavior is enforced by setting bit 0 of the *Pin-Based VM-Execution Controls* of all dom0's vCPUs. Using this configuration, any external interrupt sent to any CPU of a dom0 triggers a VMExit.

When called, the handler first masks the interrupt related to the log full event. Then, it copies the contents of all the PRL log buffers, which are full (for all VMs that have sent an interrupt), to larger buffers. Note that a large buffer (also called a cumulative buffer) is allocated to each VM by the system software and that this buffer is unique per VM, even for a multi-vCPU VM. During the copy phase, the handler accumulates the number of occurrences of each GPA in the PRL log buffer. In this way, the WSS estimation system can identify hot pages. Once the copy of PRL log buffers ends, the handler resets the *PRL index* of all VMCSs detected as full to its initial value (511). Note that modifying a *PRL index* only affects its VMCS memory region, not the internal registers of the corresponding processor. Indeed, the synchronization of the VMCS memory region and the processor registers is not automatic. To enforce this, we introduce a new instruction that updates the internal VMCS state of a specific processor using its corresponding VMCS memory region. The execution of the handler ends with the unmasking of the log full interrupt. This algorithm handles several log full events using a single generated interrupt. It is inspired by the New API (NAPI) implemented in modern Linux kernels to handle network packet reception [146].

4.3.4 PRL-based WSS estimation system

We implemented a WSS estimation system that leverages PRL. Our system launches as many WSS estimation processes as the number of tenant VMs. Each process calculates WSS using the equation:

$$WSS = hotPages \times pageSize + \varepsilon \tag{4.1}$$

where $hotPages$ is the number of computed hot pages, $pageSize$ is the size of a memory page and ε is the size of the guest kernel footprint (the minimal amount of memory needed by the kernel).

Hot pages. The number of hot pages is based on the GPAs that PRL logs. Its computation takes three parameters as input:

- τ: the minimal number of times a page's GPA must be logged for this page to be considered hot;
- ω: the stability duration used to determine if the VM has already covered its working set;

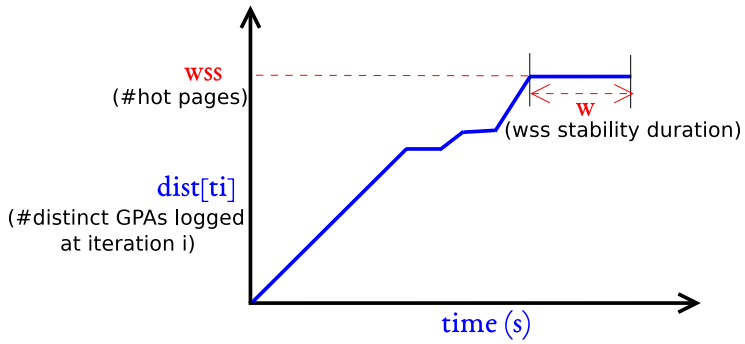


Figure 4.5: Estimation of the number of hot pages.

μ: the observation interval.

The values of these parameters are defined by the external entity which launches the WSS estimation system. Different approaches exist to determine these values [165].

Let *Cbuff* be the cumulative PRL buffer of the VM whose WSS is calculated. The estimation of the number of hot pages works iteratively, as follows. For each iteration i, the number of distinct GPAs present in *Cbuff* that have been logged more than τ times is computed and stored in an array $dist[t_i]$ (where t_i is the iteration time). The loop ends when $dist[t_i] - dist[t_i - \omega] = 0$, which means that the VM has touched/referenced all memory pages belonging to its current working set. Otherwise, the process goes to sleep for μ seconds before continuing the iteration. Figure 4.5 illustrates how this algorithm works. We can see the evolution of $dist[t_i]$ over time which corresponds to an increasing monotonic function.

Guest kernel footprint. The value of the guest kernel footprint ε depends on the guest kernel binary. It is estimated once by the data center operator for each kernel binary. The following algorithm can be used:

1. Starts a 2GB VM from the kernel binary;
2. Initialize ε and $currentMem$ (an auxiliary variable) to 2GB;
3. Set $currentMem$ to $95\% \times \varepsilon$;
4. Change the VM memory size to $currentMem$;
5. If the VM crashes then:
   - stop the algorithm and return ε;
   - else set ε to $currentMem$ and go to step 3.

These steps can be automated for a machine virtualized with the Xen hypervisor.

4.4 PRL Evaluation

Our evaluation of PRL mainly focuses on WSS estimation. We do not evaluate PRL for live migration as the PRL_{PAML} mode is strictly similar to the current PML design (see Section 4.2.1 for in-depth PML live migration evaluations).

Our assessment of the WSS estimation system based on PRL covers accuracy and overhead. Accuracy is the capability to estimate the WSS of a VM accurately. Overhead is the impact on the VM whose WSS is estimated and the number of resources consumed by the WSS estimation system inside the dom0. We used empirical values for the parameters of the WSS estimation algorithm (let us recall that these inputs are decided and defined by the external entity which launches the WSS estimation system and that different approaches exist to determine them [165]). We set τ to 50, meaning that a page's GPA needs to be logged at least 50 times for this page to be considered hot[7]. We set μ to $30s$, i.e., the algorithm iterates every $30s$. We used this value following the VMWare technique. We set ω to $120s$, corresponding to the observation period.

4.4.1 Experimental environment

We use the same experimental environment as the one presented in Section 4.2 to assess PRL. In addition, we use the hardware simulator Gem5 [70] to emulate a machine that implements PRL. We chose Gem5 because it is a very popular hardware simulator (specifically adapted for research), which was used by 90+ research papers at the time of writing [37]. Although Gem5 allows the execution of a complete Linux distribution, we have extended it to simulate a virtualized system. This improvement consists of adding the Extended Page Table (EPT) support, the extension of the hardware page table walker to perform a 2D page walk through the EPT, and the implementation of PRL/PML logging mechanisms.

```
1 /*
2  * Main variables description
3  * tab: array to be processed
4  * PERIOD: application duration
5  * SIZE: number of tab memory pages
6  * time: total time in seconds
7  */
8 #define PERIOD ...
9 #define SIZE ... //SIZE * 4096 = size_of_tab
10
11 void synthetic_app()
```

[7] We use a small value to evaluate a worst-case scenario because a small τ value increases the buffer's log filling rate, thus, more log full events.

```
12 {
13   do{
14     for(int j=0;j<SIZE;j++)
15       /* Operate on tab (read/write) */
16       operation(tab[j]);
17       /* Compute the time */
18       time = ...;
19   }while(time < PERIOD);
20   free(tab);
21 }
```

Listing 4.2: Second synthetic application skeleton.

The emulation methodology is as follows. We run the application under Gem5 and we collect the logged GPAs including timestamps. The collected traces are then replayed inside the dom0 of a real virtualized environment that runs the WSS estimation system. A dom0's CPU (noted CPU_0) is dedicated to the latter. The other dom0's CPUs run processes that replay the traces, thus mimicking the functioning of a real machine equipped with PRL. Each process replaying the traces sends an IPI to CPU_0 each time it plays N traces, N being the size of the log buffer. This emulation is similar to the actual operation of a PRL-compatible machine, as shown in Figure 4.4. To be fair, we also assessed the accuracy of PML and VMware solution using the same methodology. As a reminder, the VMware WSS estimation solution consists in periodically selecting (every 30 seconds) a sample of 100 memory pages whose present bits are invalidated. The proportion of pages, among these selected 100 pages, which will cause a page fault represents the WSS of the VM.

The skeleton of the synthetic application we use for our experiments is shown in Listing 4.2. The parameter $SIZE$ controls the WSS: SIZE $\times$ 4KB. $4KB$ is the size of `tab[j]`, which is also the size of a memory page (in our architecture). The parameter $PERIOD$ represents the duration of the application, and `operation(tab[j])` is the operation performed on the array entry. We consider three types of workload: every read is followed by a write ($RWRW$), a set of reads followed by a set of writes ($RRWW$), and a set of writes followed by a set of reads ($WWRR$). We choose an application with well-known read and write phases instead of random operations, as in Section 4.2, because we want to clearly show how PRL performs facing each operation type.

4.4.2 Accuracy of the simulator

We validate the accuracy of our EPT and PML simulation by running each workload type of the synthetic application atop our simulator, then on a real machine supporting EPT and PML. We compare the content of the PML

buffer in both cases. Since, in the real environment, the application runs inside a VM (which includes a guest OS), several GPAs that do not belong to the synthetic application's address space can be logged into the PML buffer. These GPAs are part of other components inside the VM which perform write operations. Once eliminating these GPAs, we observe that the content of the PML buffer in both environments is the same, validating the accuracy of our simulator. From now, we can study the efficiency of PRL in the task of WSS estimation.

4.4.3 PRL efficiency

We evaluate the accuracy of the WSS estimation of an application running in a VM. We consider the synthetic application described in Section 4.4.1 and the HPL Linpack described in Section 4.2. The WSS of the synthetic application is fixed to 400 MB by setting $SIZE$ accordingly (i.e., `102400 pages`). We do not use BigDataBench applications because their execution never completes under Gem5 in our experimental environment. We compare our results against PML and VMWare sampling-based WSS estimation techniques. We confront the VMWare WSS estimation technique mainly due to its popularity, albeit its skewed results due to hard sampling period configuration. Additionally, other WSS estimation techniques benefit from extensive evaluation in different research works, such as[131].

The results of our experiments are shown in Figure 4.6. We observe that the WSS estimation based on PRL has a margin of error of less than 1MB for all workloads, i.e., a precision greater than 99.75%[8]. In contrast, the VMware solution provides inaccurate results, which is in line with previous research observations [133]. The accuracy of the WSS estimation based on PML mainly depends on the number of write operations. In the RWRW and WWRR workloads, all the array entries are updated, allowing PML to correctly estimate the WSS. In contrast, the RRWW workload highlights the incapacity of PML to identify hot pages when only read operations are performed.

Regarding the HPL Linpack application, we observe that both PRL and PML provide similar WSS estimations, while the solution based on VMware still diverges a lot. Because the real WSS is not known in advance, we estimate its value as follows. We dynamically set the memory size of the VM to the WSS estimated using the PRL-based system. Since we do not observe any VM crash and no performance degradation, the PRL-based WSS value is either overestimated or accurate. We decrease the memory size of the VM by 100MB and observe a crash of the VM, which confirms that the previously estimated WSS was accurate.

As illustrated in Table 4.2, we observe that the number of GPAs missed

[8]Notice that due to the workload nature (array scan), the working set is mostly constant.

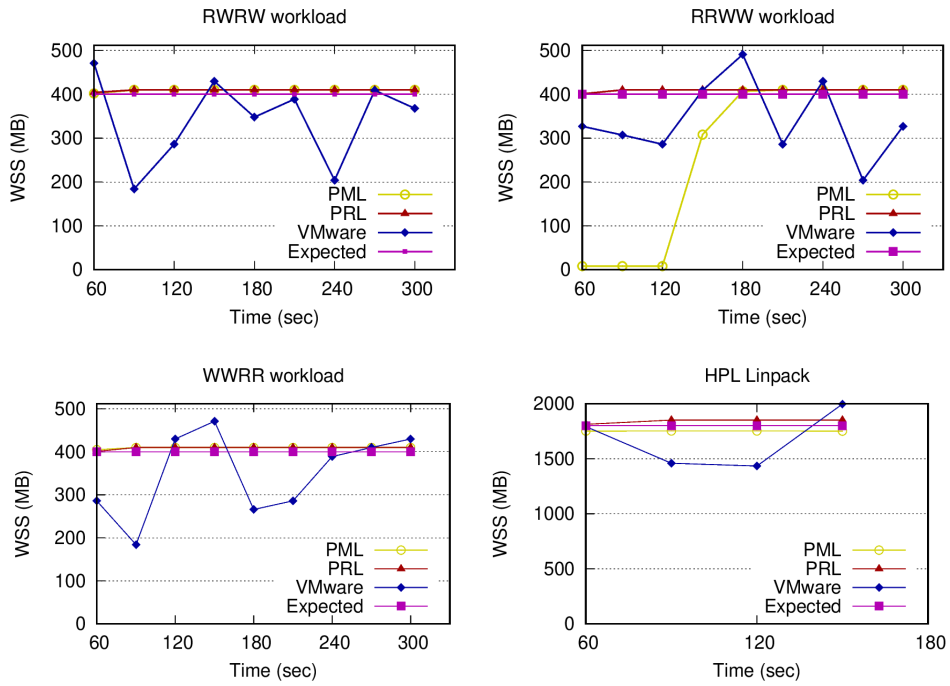


Figure 4.6: Efficiency of PRL compared with PML and VMware. `Expected` is the baseline. It corresponds to the WSS of the application ($400MB$).

when handling full log events in PRL is negligible. Indeed, it is very likely that a GPA that is part of the working set is seen very often. Missing one GPA, thus, does not compromise the accuracy of the WSS estimation.

Buffer size (MB)	RRWW			HPL		
	# full log events	# missed GPAs	mean of missed GPAs per full log event	# full log events	# missed GPAs	mean of missed GPAs per full log event
512	17094	0	0	20701	741	0.04
1024	8543	213	0.02	10510	116	0.01

Table 4.2: Number of missed GPAs during the treatment of full log events. Note that results for WWRR and RWRW are almost the same as for RRWW, thus we decide not to present them.

To confirm our findings, we extend our evaluations by including the applications of PARSEC [46], a benchmark suite composed of multithreaded programs. We used PARSEC because it is widely used by the research community, and the WSS of some applications in the suite are known from the technical report of PARSEC in 2008 [81]. Our results are presented in Figure 4.7. Apart from `freqmine`, all the values estimated by PRL match the initial technical report[9]. We observe that the PML-based solution is accurate

[9]Regarding freqmine, the latter's dataset evolved since the [81] report, which explains the discrepancy between the [81] results and ours.

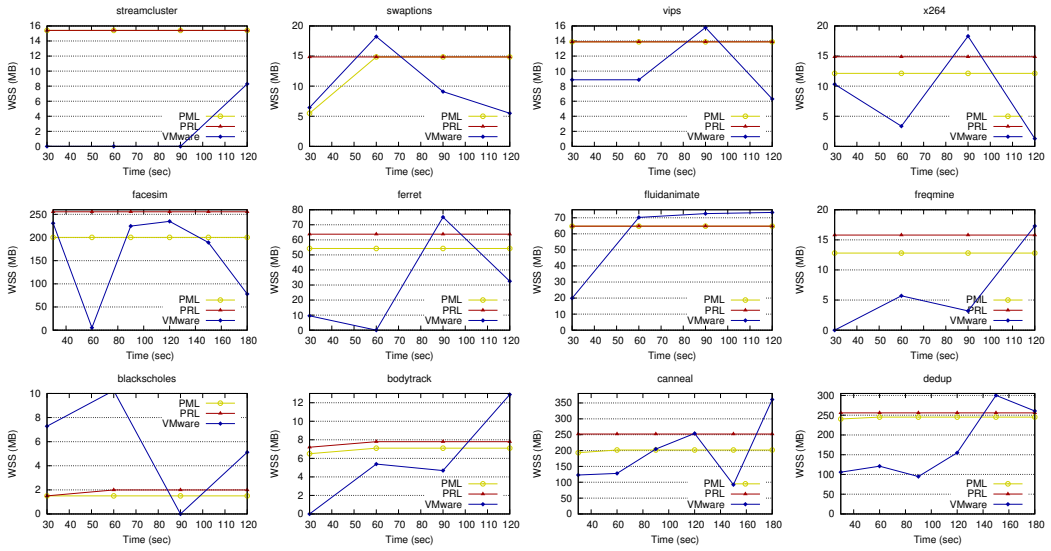


Figure 4.7: WSS estimation of PARSEC applications using PRL.

for six applications (streamcluster, vips, fluidaminate, blacksholes, bodytrack, dedup) while the VMware-based solution fails to provide correct values. By making the implementation of PRL under Gem5 publicly available, researchers can use it for estimating the WSS of other benchmarks.

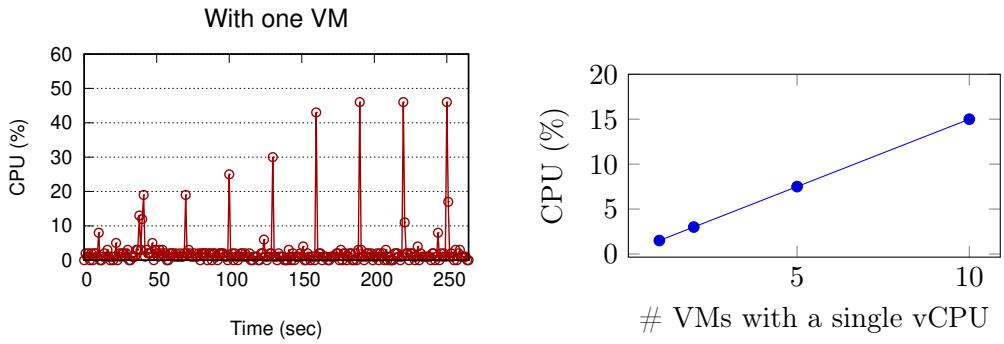
4.4.4 PRL overhead

We evaluate both the impact induced by PRL on the VM whose WSS is estimated and the number of resources consumed by the WSS estimation system inside the dom0.

The overhead on the VM whose WSS is estimated is the total number of CPU cycles used by the PRL internal circuitry. Because a PRL-capable machine does not exist yet and Gem5 does not measure internal CPU circuitry costs, we assume that this number of CPU cycles should be roughly equivalent to that of PML since the two modes are very similar. Therefore, to assess PRL overhead, we use a PML-capable machine on which the VM runs an application that performs only read operations. This scenario avoids having `VMExits` that can not occur with a PRL-based machine for which `VMExits` are executed on a different processor. Remember that both PML and PRL logging take place during page table walk on each TLB miss. Thus, we repeated this experiment by varying the number of tenant VMs to increase the pressure on the TLB. We hardly measure any difference in performance (around 0.001%), which means that PRL will likely not experience any performance degradation on the VM whose WSS is computed.

We use the emulated environment to evaluate the CPU consumption of the WSS estimation system inside the dom0, which is pinned to a specific

core. We use a single VM and a write-only workload (WWRR in which RR is null) as the worse case. Figure 4.8a presents the percentage of CPU consumed by the WSS estimation process in the dom0 for a VM with a single vCPU. For readability, only a representative portion of the results is presented. The CPU consumed during the treatment of the full log event increases (with the size of the cumulative buffer) until the WSS is discovered. However, we can observe that the CPU is most of the time, idle, waiting for a PRL log full event. On average, the total CPU time consumed by the WSS estimation process for a VM with a single vCPU is less than 1.5%. We repeated the same experiment by varying the number of VMs. As illustrated in Figure 4.8b, the average percentage of CPU consumed by the WSS estimation system increases linearly.



(a) CPU consumption over time for a single VM.

(b) Average CPU consumption for several VMs.

Figure 4.8: CPU consumption for the WSS estimation process on VMs with a single vCPU

4.5 Related work

Hardware assisted virtualization (HAV) Contrary to A. Baumann's conclusion in his HotOS'17 paper [63], we believe that virtualization is the System's sub-domain which mainly influences hardware architecture research. In fact, HAV contributions have evolved at the rhythm of the limitations of software solutions. Notably, the Extended/Nested Pages Table [68] were introduced to address the tremendous number of context switches caused by shadow paging [151]. In summary, several hardware features (e.g., Intel VT [41], VMFUNC [48], Intel CAT [42], APICv [129]) have been integrated inside CPU, memory subsystems, I/O devices and many other motherboard components by hardware manufacturers these recent years for achieving basic virtualization functionalities. An extreme application of the HAV approach has been proposed by E. Keller with NoHype [105], which is a hardware-

only hypervisor. In 2017, Amazon anounced its new hypervisor called Nitro [113], which can be seen as a concretization of the NoHype vision. In academia, a lot of efforts have been made on the topic of memory virtualization [62, 160, 55, 154, 59, 93, 92, 115] to minimize the overhead of the 2D page walk imposed by EPT.

Page tracking Page tracking is, in essence, the core of working set estimation techniques. The most popular approach for page tracking consists in denying access to memory pages, which need to be monitored so that the next accesses trap inside the system software (hypervisor or OS). This approach is used by the majority of checkpointing, live migration, and WSS estimation solutions. Very few research works have investigated hardware features for page access tracking. Pin Zhou et al. [166] proposed a Miss Ratio Curve (MRC) monitoring hardware feature, which can be used as an alternative to page access tracking in the task of WSS estimation [165]. [166] proposed a solution consisting in snooping the address bus and requires collaboration with the OS page fault handler. As with PML/PRL, [166] showed that tracking page accesses at the hardware level is possible. However, [166] is dedicated to native systems and needs to collaborate with the OS, unlike PML/PRL.

Live migration [123, 147] presented surveys that the reader could refer to. We would like to highlight among them [54, 130] who studied live migration of the VM storage along with its memory. Very few research work has investigated VM storage live migration because it increases VM downtime. Migrating the VM storage is necessary when dealing with data-intensive applications because they generally use local storage instead of classical network storage. The utilization of PRL/PML is also beneficial for this use case because disk accesses go through the buffer cache, which resides in RAM. [164] addressed another important aspect of live migration, which is the prediction of the right migration instant. In fact, live migration could fail due to a lack of resources or sudden VM behavior changes. The authors use the VM WSS to track such behavior changes. Thus PRL is likely to improve [164]'s contribution.

WSS estimation *Committed_AS*, a Linux kernel statistic, is generally used (e.g., by Xen) to estimate the VM WSS. This statistic corresponds to the total number of anonymous memory pages allocated by all processes but not necessarily backed by physical pages. Therefore, *Committed_AS* overestimates the WSS. Another limitation of this approach is the fact that it requires a collaboration between the hypervisor and the guest OS. Zballoond [80] relies on the following observation: when a VM's memory size is larger than or equal to its WSS, the number of swap-in and re-fault (occurs when a previously evicted page is later accessed) events is close to zero. The basic idea behind

Zballoond consists in gradually decreasing the VM's memory size until these counters start to increase. The VM's WSS is the smallest memory size which leads the VM to zero swap-in and re-fault events. Like *Committed_AS*, *Zballoond* requests collaboration with the guest kernel. Furthermore, *Zballoond* is very active in the sense that it performs memory pressure on the VM, which could degrade the VM performance. Geiger [101] monitors the evictions and subsequent reloads from the guest OS buffer cache from/to the swap device. It relies on a ghost buffer [137] which represents an imaginary memory buffer that extends the VM's physical memory (noted m_{cur}). The size of this buffer (noted m_{ghost}) represents the amount of extra memory which would prevent the VM from swapping out. Knowing the ghost buffer size, the VM's WSS can be computed using the following formula: $WSS = m_{cur} + m_{ghost}$ if $m_{ghost} > 0$. Unlike the two previous solutions, *Geiger* is transparent from the VM's point of view. However, Geiger has an important drawback which derives from its non-intrusiveness. It is able to estimate the WSS only when the size of the ghost buffer is greater than zero (the VM is in a swapping state). Geiger is inefficient if the VM's WSS is smaller than the current memory allocation. Hypervisor Exclusive Cache [121] is fairly similar to Geiger. Badis [131] combined VMware and *Geiger* in order to take advantage of their non-intrusivity on the VM's codebase. Badis suffers from VMware and Geiger's drawbacks presented above. [165] computes the WSS of an application based on its miss-ratio curve (MRC). The latter shows the fraction of the cache misses that would turn into cache hits if the VM's allocated memory increases. [165] presents a set of methods to determine the values of the input parameters of our WSS estimation system (τ, ω, and μ). [112] presents an application-assisted WSS estimation solution in virtualized systems. In contrast to our solution, which considers the VM as a black box, [112] relied on the application inside the VM to estimate the WSS, which is very intrusive.

4.6 Summary

In this chapter, we analyzed Page Modification Logging (PML), a memory page tracking technology introduced by Intel and VMware as a key virtualization functionality. We showed that the current design of PML makes it effective for VM live migration but not for WSS estimation. Based on this observation, we proposed Page Reference Logging (PRL), an extension to PML which makes it also effective for WSS estimation. We implemented PRL in Gem5, a popular hardware simulator. We further described a WSS estimation system that takes advantage of PRL. We evaluated our solution using real and synthetic applications and compared it with existing solutions. Our results demonstrate that our solution is both accurate and does not impact user VMs.

Part II

OoH in Unprivileged VMs (Xen-domU)

Chapter 5

OoH for PML: Efficient Dirty Page Tracking In Virtualized Clouds

This chapter demonstrates the application of OoH with PML for cloud user applications. Because dirty page tracking is at the heart of many essential tasks, including process checkpointing (e.g., CRIU) and concurrent garbage collection (e.g., Boehm GC), OoH exposes PML to accelerate these tasks in the guest. We present two OoH solutions, namely Shadow PML (SPML) and Extended PML (EPML), that we integrated into CRIU and Boehm GC.

Contents

5.1 Introduction

In this context of cloud computing, dirty page tracking is an important need for both the guest kernel and its userspace processes. The former tracks dirty pages to know if a file-backed memory page should be copied to disk when swapped out. In userspace, dirty page tracking is used by several applications such as garbage collectors (GC) [9], container or process checkpoint/restore systems [149, 90], use-after-free vulnerability mitigation systems [56], etc. For illustration, dirty page tracking is used by concurrent GCs like Boehm [9] to reduce application pause time during the construction of reachable objects. Existing dirty page tracking solutions rely on *userfaultfd* (hereafter *ufd*) or */proc/<PID>/pagemap* (hereafter */proc*), two interfaces offered by Linux. As assessed in Section 5.2, these two interfaces are extremely expensive since they are based on page write protection, which induces a lot of page faults and world transitioning (kernel space to userspace and vice versa). We measured an overhead of up to 15× with *ufd* and 4× with */proc*.

Intel PML tracks dirty pages *at the scale of a whole VM by the hypervisor* to accelerate VM live migration [74]. In this chapter, we study how PML can be diverted to be used by an unprivileged guest kernel and its applications to accelerate their execution. We identify three main challenges to achieving that. (C_1) PML can be exploited uniquely by the hypervisor. In other words, only software (the hypervisor) running in *VMX root ring 0* CPU mode can use PML. Yet, the guest kernel and its applications run in CPU *VMX non-root ring* $\geq$ *0*. (C_2) PML operates in coarse-grained that is, it concerns the entire VM. We want to use PML at the granularity of a process within the VM while allowing the hypervisor to continue using it at the scale of the VM. (C_3) PML only logs guest physical addresses (GPAs), while userspace processes need guest virtual addresses (GVAs).

In this chapter, we describe two solutions of OoH for PML and present two systems that can use them. These systems are Boehm (a GC) and CRIU (a process checkpoint/restore system). The first OoH solution, called Shadow PML (noted SPML), requires no hardware modification. In contrast, the second OoH, solution called Extended PML (noted EPML), slightly extends the hardware implementation of PML to avoid the limitations of SPML. We investigated SPML to point out the need for a hardware extension of PML. SPML relies on hypercalls (VM $\rightarrow$ hypervisor) and virtual interrupts (hypervisor $\rightarrow$ VM) to respectively enable/disable PML and to periodically copy GPAs (logged by the CPU to a PML buffer in the hypervisor) to a ring buffer shared between the hypervisor and the guest OS. It is then up to the guest OS to perform GPA to GVA reverse mapping. EPML, on its side, hijacks two hardware virtualization features (VMCS shadowing [1] and posted-interrupt [30]) and slightly extends PML to avoid the intervention of the hypervisor on the

critical path. EPML is able to log the address of a dirty page into two buffers simultaneously. The first buffer is exclusively managed by the guest OS, while the second buffer is managed by the hypervisor. Finally, EPML logs GVA to the guest level buffer and logs GPA to the hypervisor level one.

To facilitate the utilization of OoH, we provide a generic library that we implemented following the UIO driver principle [27]. The library has two parts: a kernel module and a userspace template code. The former does not need to be managed by the application developers, who simply need to integrate the template code into their applications. We prototyped and evaluated SPML on a DELL Intel Core i7-8565U processor machine that supports PML using the Xen hypervisor (a popular hypervisor used by Amazon). Regarding EPML, we rely on BOCHS, the only emulator (to the best of our knowledge) that implements PML. In both designs, the guest OS is Linux. We use both micro- and macro-benchmarks for evaluations. For the latter, we use all five in-memory database engines from tkrzw [25] and six applications from Phoenix [139] (a MapReduce application set). To evaluate our prototypes, we set up a rigorous methodology, especially for EPML, which extends the hardware. To this end, we build a mathematical formula that accurately approximates the overhead or improvement of each solution. We systematically compare SPML and EPML with `/proc` and `ufd`. We are interested in the impact of each solution on the tracked application (e.g., Phoenix) and the tracking system (e.g., CRIU). The solutions are classified as follows, in decreasing order of the overhead they induce: SPML, `ufd`, `/proc`, and EPML. `/proc`, which is the default solution implemented in both CRIU and Boehm, incurs an overhead of up to 102% with CRIU on the Phoenix `pca` application and up to 232% with Boehm on the Phoenix `string-match` application. The overhead of SPML is up to 114% with CRIU and 273% with Boehm on the same applications. EPML leads to the lowest overhead, which is about 7% with CRIU on `pca` and 24% with Boehm on `string-match`. Hence, EPML can improve existing systems by up to 62%. Concerning the tracking system and compared to `/proc`, SPML induces up to 5× slowdown on CRIU and 3× slowdown on Boehm GC. EPML brings up to 4× speedup compared to `/proc` and 13× speedup compared to SPML for CRIU. Finally, for Boehm, EPML brings up to 2× speedup compared to `/proc` and up to 6× speedup compared to SPML.

In summary, this chapter makes the following contributions:

- (Empirical contribution) We finely quantify the impact of page fault-based dirty page tracking.
- (Conceptual contribution) We present OoH and two solutions, namely SPML and EPML.

- (Technical contribution) We prototype SPML and EPML in real and emulated environments (BOCHS) using popular system software (Xen hypervisor and Linux guest OS).
- (Technical contribution) We integrate PML into two popular tracker systems, namely CRIU and Boehm GC.
- (Empirical contribution) We rigorously evaluate our designs using micro- and macro-benchmarks.

5.2 Motivations

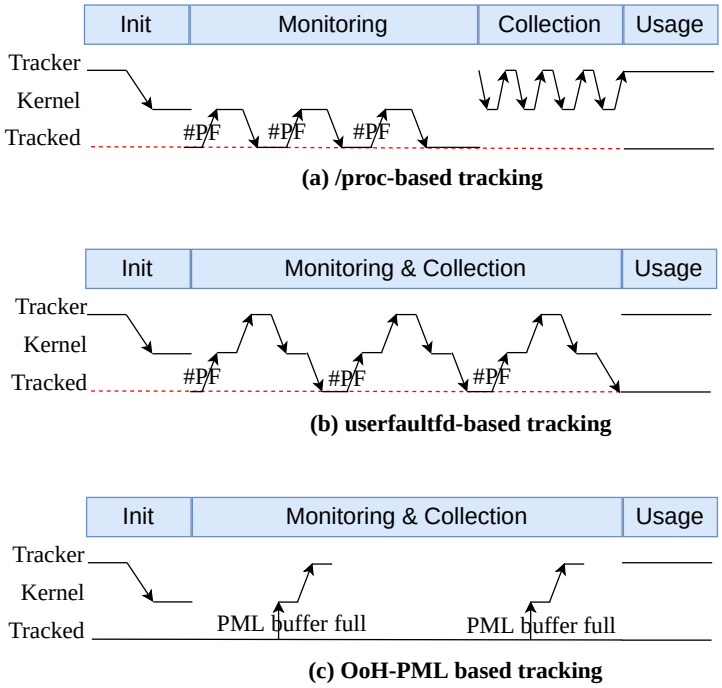


Figure 5.1: Impact of `/proc`, `ufd`, and OoH-PML methods on Tracked and Tracker. The two former methods lead to several suspensions (red dashed lines) of Tracked, due to #PF -Page Faults- and context switches. `ufd` induces the longest suspension time (#PF are handled in userspace). However, dirty page address collection takes much more time with `/proc` (due to the parsing of `/proc/PID/pagemap`), thus impacting Tracker. OoH has the benefits of both worlds and does not require the suspension of Tracked.

Dirty page tracking, thus PML, is not only essential for hypervisors. A thread running inside a VM may also need to monitor dirty pages for garbage collection or checkpointing. We call Tracker the monitoring thread and Tracked the thread whose memory is monitored. The traditional approach used by

Tracker is the invalidation of dirty and present bits from Tracked's page table entries (PTE). Linux offers two interfaces that Tracker can leverage. These are `ufd` and `/proc`. Fig. 5.1 summarizes their functioning compared to an OoH-PML-based solution. `ufd` and `/proc` are introduced in this section, while OoH-PML is presented in Section 5.3. The activity of Tracker can generally be organized into four main phases: the initialization of the tracking method, the monitoring, the collection of dirty page addresses, and the exploitation of the latter (e.g., for checkpointing).

We consider in this section that the fourth phase is empty as its duration is agnostic to the tracking method in comparison with the three other phases. We launch Tracker and Tracked at the same time, but the latter is suspended during the initialization phase. The ideal execution time of Tracked is when it runs without being tracked. The ideal execution time of Tracker is the ideal execution time of Tracked. As one can deduce from Fig. 5.1, the choice of the tracking method can impact both Tracker and Tracked. We can see that OoH is the only method that theoretically leads both systems to their ideal execution time.

5.2.1 The cost of `ufd`

Fig. 5.1.b summarizes the functioning of a `ufd`-based dirty page monitoring solution. To use `ufd`, Tracker first registers the memory region it wants to monitor. After the registration, it will be notified by the kernel each time a page fault concerning the registered region occurs. `ufd` supports two monitoring modes: `miss` and `write_protect`. For `miss`, a notification is sent to Tracker when Tracked accesses a monitored page for the first time. Concerning `write_protect`, a notification is sent when Tracked attempts to modify a monitored page. In both modes, Tracked is suspended until the fault is resolved. In the case of `write_protect`, the Tracker should write-unprotect the faulted page in order to unpause Tracked. One can see that, with `ufd`, the collection of dirty page addresses can be done during the monitoring phase.

We assess the overhead of `ufd` using as Tracked, a synthetic program (presented in Section 5.5) that just parses and writes to an array of buffers. The size of each buffer is 4KB, allocated at page boundaries. We are interested in monitoring the entire array. Table 5.1 second and fifth rows show the overhead of `ufd` while we vary the array size. We can see that the overhead linearly increases with the array size. We measured an overhead of up to 15× and 14× for 1GB on Tracked and Tracker respectively. We break down the page fault handling time into two components: the time spent inside the kernel (about 33.6ms for 1GB) and the time spent in Tracker (about 3,3383ms for 1GB). The total suspension time of Tracked represents, on average, about 93% of its execution time.

On Tracked	1MB	10MB	50MB	100MB	250MB	500MB	1GB
`ufd`	195	272	583	1,050	1,266	1,462	1,463
`/proc`	104	55	114	208	302	307	335

On Tracker	1MB	10MB	50MB	100MB	250MB	500MB	1GB
`ufd`	93	169	477	940	1,269	1,153	1,349
`/proc`	47	43	58	148	151	143	147

Table 5.1: Overhead (in %) of `ufd`- and `/proc`-based dirty page tracking methods.

5.2.2 The cost of `/proc`

Fig. 5.1.a summarizes the functioning of a `/proc`-based dirty page monitoring solution. Tracker first instructs the kernel to clear soft-dirty bits of Tracked's PTEs. This is done by writing 4 to `/proc/PID/clear_refs` file, where PID is the process identifier of Tracked. This operation is dominated by the time taken by the kernel to parse Tracked's PTEs and to flush the TLB (about $2.234ms$ when the monitored memory is 1GB). All of this lengthens the initialization step compared to ufd. After this, once Tracked tries to modify a monitored page, a fault occurs. The handler of that fault sets in Tracked's PTE the soft-dirty bit of the faulted page (this operation costs about $33.5\mu s$). At the end of the monitoring period, Tracker reads the soft-dirty bits (bit 55) from `/proc/PID/pagemap` to determine all dirty page addresses (this costs about $594.187ms$ when the monitored memory is 1GB). The total suspension time represents about 73% of the total execution time of Tracked. As shown in Table 5.1 top, the impact of `/proc` on Tracked varies with the memory size. We measured an overhead of more than 4× for 1GB of memory. This overhead is lower than the one induced by `/ufd`, as shown above. Concerning Tracker, see Table 5.1 bottom, the overhead is up to 2× (147%). When compared to `/ufd`, although `/proc` increases the address collection phase, its cost is compensated by the smaller suspension it induces during the monitoring phase.

5.2.3 Alternative

A way to use PML for a process is to dedicate a VM to the latter, thus exploiting PML as is only by the hypervisor. This approach works for some use cases, such as checkpointing. In fact, to checkpoint the process, the user would checkpoint the corresponding VM.

However, this approach would not be effective for use cases where PML is required at runtime. It is the case for garbage collectors that we study in this chapter. The GC runs inside the guest, so it needs to access PML from there. It is not possible, at the hypervisor level, to perform garbage collection for processes running inside a guest (which is a black box for the hypervisor).

Another important drawback of this approach is that it is contrary to the current trends of colocating tasks within VMs to save resources (thus money) and reduce Inter-Process Communication costs (to which HPC applications are very sensitive). This practice is common in FaaS platforms, where functions of the same client are co-located within the same VM [153]. Besides, several HPC FaaS platforms are initiatives developing [8, 122].

5.3 Design

The goal of OoH is to make some hardware virtualization features usable from inside the guest OS. We do this with as minimal changes as possible in the hardware, the hypervisor, and the guest kernel. In addition, we want to propose a simple utilization interface for application developers. §5.5.2 presents the methodology that OoH designers can follow when applying this principle to a hardware virtualization feature. Then §5.3.1-5.3.4 illustrate that methodology to the exposure of Intel PML.

5.3.1 Overview

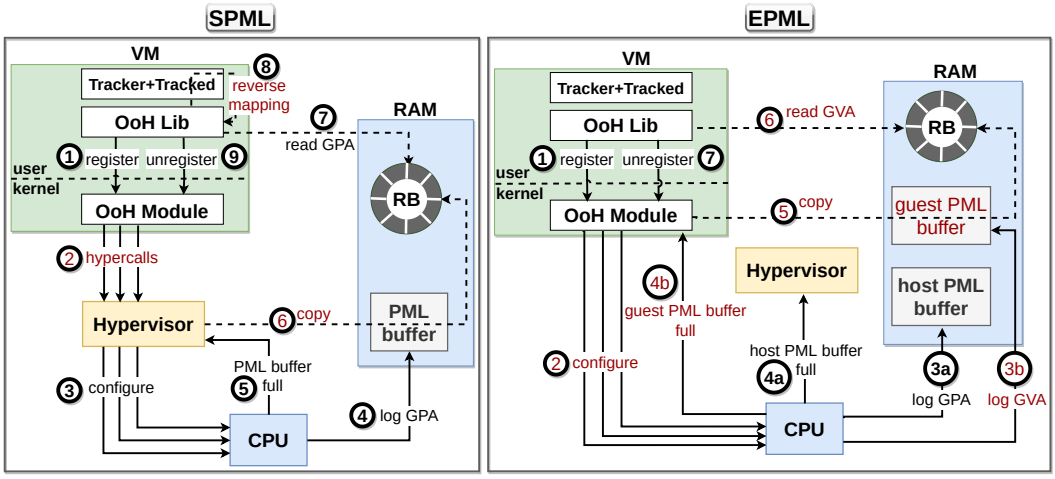


Figure 5.2: OoH: SPML and EPML architectures. Sofware/Hardware changes are highlighted in red.

We present two solutions of OoH for PML, namely Shadow PML (noted SPML) and Extended PML (noted EPML). SPML requires no hardware modification, while EPML slightly extends the hardware for better performance. Fig. 5.2 presents the architecture of the two solutions (detailed in §5.3.2 and §5.3.3). In the guest, we provide OoH as a userspace I/O (UIO) driver composed of a kernel module (OoH Module) and a userspace library (OoH Lib). At load time, the former does a set of initialization operations, including ring

buffer allocation that is shared with userspace (and the hypervisor in SMPL only). Tracker uses OoH Lib to register the PID of Tracked with OoH Module. From there on, the processor can log dirty pages' addresses to a 512KB PML buffer, which is copied to the ring buffer once full. Relying on OoH Lib, Tracker can periodically fetch the collected addresses to achieve its goal (e.g., checkpointing).
EPML differs from SPML in two ways: (1) With EPML, the processor also logs GVAs, thus avoiding costly reverse mapping in OoH Lib; (2) With EPML, the guest kernel can directly deal with the processor, thus avoiding costly hypercalls.

5.3.2 Shadow PML (SPML)

The basic idea behind SPML is to make the hypervisor emulate PML for guests' processes. Indeed, in this solution, the hypervisor is the only component able to perform PML's instructions to the processor. Fig. 5.2 left summarizes SPML functioning with three main features.
(1) To ensure that the processor only logs GPAs for tracked processes, we introduce two new hypercalls: `disable_logging` and `enable_logging`. The former is called by OoH Module every time a tracked process is scheduled-out. This hypercall copies the content of the PML buffer to the ring buffer and instructs the processor to stop logging. `enable_logging` is the inverse operation invoked by OoH Module when a tracked process is scheduled-in.
(2) In SPML, the processor logs GPAs while Tracker needs GVAs. To fill this gap, OoH Lib *reverse maps* GPA to GVA by parsing the page table of Tracked using the `/proc` interface.
(3) Recall that the hypervisor can also use PML for its own purposes (e.g., VM live migration). To coordinate the two levels, we introduce two flags (`enabled_by_guest` and `enable_by_hyp`) that indicate which level has enabled PML. When the PML buffer is full, the hypervisor does not fill the ring buffer unless `enabled_by_guest` is set. And similarly, if `enabled_by_hyp` is not set, the hypervisor bypasses the operations corresponding to its use of PML, thus avoiding unnecessary additional CPU time consumption. If the hypervisor wants to deactivate PML, it first checks that `enabled_by_guest` is not set and vice versa.

The main limitation of SPML lies in its performance overhead caused by the high number of hypercalls and reverse mapping operations that it generates, justifying EPML.

5.3.3 Extended PML (EPML)

Fig. 5.2 right summarizes EPML. The basic idea behind it is to provide a second level of PML directly controlled by the guest OS (OoH Module). Every

tracked process is associated with a guest-level PML buffer by OoH Module, exactly as the hypervisor manages a PML buffer per vCPU.

To minimize hardware changes, EPML leverages the existing VMCS shadowing (see §2.1.2) feature, which allows a guest to perform `vmread` and `vmwrite` instructions without vmexit to the hypervisor. At load time, OoH Module calls the hypervisor to enable and configure VMCS shadowing. This is the only hypercall performed in EPML. Therefore, when a tracked process is scheduled-in or out, OoH Module accordingly enables or disables address logging using `vmwrite` instruction. Contrary to SPML, EPML does not interfere with the hypervisor's needs.

From a hardware point of view, EPML makes the following small changes. We introduce in the VMCS a new field (called `Guest PML Address`) that represents the address of the guest-level PML buffer. Because the guest (OoH Module here) only sees GPA, the value that it sets to the Guest PML Address should be translated to a host physical address (HPA) so that the processor can log to the right location in RAM. To tackle this challenge, we extend the VMX ISA so that if a `vmwrite` instruction to the `Guest PML Address` field is performed when the processor is in guest mode, it first translates the address to a HPA (using EPT) before writing it to the shadow VMCS.

Another improvement brought by EPML is the capability to log GVAs, thus avoiding reverse mapping by OoH Lib. We modify the page walk circuit to make the processor log the GVA to the guest-level PML buffer and the GPA to the hypervisor-level PML buffer.

The last hardware extension is to handle guest-level PML buffer full events. We modify the hardware so that when the guest-level PML buffer is full, the processor raises a virtual *self-IPI (Inter-Processor Interrupt)* which is handled in the guest by OoH Module. Notice that this is a very small modification that leverages an existing feature called Posted-Interrupts. With the latter, the processor is able to directly deliver interrupts to the guest OS when running in the guest mode without vmexits.

5.3.4 Implementation

We implemented OoH in Xen 4.10.0 hypervisor, Linux 4.15.0 guest OS, and BOCHS 2.6.11 Intel x86 emulator (only for EPML, which extends the hardware). We use CRIU and Boehm GC as Trackers.

Xen. It is a popular hypervisor used by Amazon EC2. The main changes we made concern the introduction of the two hypercalls, `disable_logging` and `enable_logging`, and the copy of the content of the PML buffer to RB.

Linux Core. We essentially modified the interrupt table to handle the virtual *self-IPI* raised by the processor when the guest-level PML buffer is full in

EPML. Notice that OoH Module is a kernel module, thus it is not part of the Linux core.

BOCHS. It is a very popular Intel machine emulator, which yet implements PML. We modified it to implement EPML, mainly by extending: the `vmwrite` instruction, the page table walk circuit to log GVAs, and the PML logging process to raise *self-IPI* on buffer full.

CRIU. It is a popular checkpoint/restore tool integrated into many well-known projects, such as OpenVZ [20], Podman [23], or Docker [6]. CRIU relies on `/proc`. To integrate OoH with it, we mainly patched two steps: (1) Initialization: OoH avoids pausing (`echo 4 > /proc/PID/clear_refs`) Tracked at the initialization phase because the activation of PML is immediate and does not interfere with the execution of Tracked, as illustrated in Fig. 5.1.c. (2) Address collection: OoH avoids parsing the `/proc/<PID>/pagemap` file to retrieve dirty pages' addresses. The ring buffer is read instead.

Boehm GC. Boehm GC [9] is a popular C and C++ garbage collector that is included in many well-known projects, such as Mozilla [17], GNU Java compiler [12], or Inkscape [14]. Boehm GC provides incremental and generational collection based on dirty page tracking. In its current implementation, Boehm relies on `/proc`. To integrate OoH, we mainly patched the *mark phase*, corresponding to where the GC checks for modified pages. As with CRIU, OoH avoids reading from `/proc/PID/pagemap` and resetting dirty bits via `/proc/PID/clear_refs`.

5.4 Security and Isolation

OoH empowers the guest processes by sharing some hypervisor-oriented hardware virtualization features (PML in this chapter) with guest OSes. One may legitimately see this as a source of potential threats. In this section, we show that neither SPML nor EPML increases the vulnerability of the hypervisor (against guest OSes), the guest OS (against others), and processes (against others within the same guest VM). Our trust model is the same as that of `/proc` and `ufd` that is, the hypervisor does not trust guest OSes, which in turn do not trust their processes. We elaborate below on the security of the hypervisor, VMs, and processes.

Concerning the hypervisor, (1) only SPML requires its modification, which represents only 194 LOC. The latter is negligible compared to the hypervisor code size (about 900K+ LOC for Xen). Accordingly, we are pretty confident that our added code is at least as safe as existing hypercalls. (2) In both SPML and EPML, the guest OS does not see hardware physical addresses

(HPA). Remember that SPML logs GPA and EPML logs GVA, both being virtual addresses. These address types are traditionally seen and managed by the guest OS, including when `/proc` and `ufd` are used. In OoH, the hypervisor remains the sole layer that sees and manages HPA. (3) The ring buffer that the hypervisor shares with each guest OS is allocated within the guest's address space and not that of the hypervisor. Thus, a guest can not leverage it to corrupt the hypervisor.

Concerning guest OSes, neither SPML nor EPML reduces their isolation level. In SPML, a dedicated ring buffer is used per guest. Therefore, a guest can only see logged addresses that belong to its address space.

In the previous version of our prototype, all tracked processes of the same guest could see the same SPML/EPML logging buffer. That implementation could potentially lead to side-channel attacks as a tracked process could learn the memory access pattern of tenant-tracked processes. (Thanks to reviewer feedback, we have (easily) updated that implementation to dedicate a per-process ring buffer and restrict its access to tracker processes only. This was an implementation detail.)

5.5 Evaluations

This section presents the evaluation results. We want to answer the following questions: (1) what is the potential overhead or improvement of SPML and EPML compared to existing solutions (`/proc` and `ufd`)? (2) what is the scalability of SPML and EPML? (3) to what extent SPML and EPML are able to efficiently capture all dirty pages?

5.5.1 Experimental environment

Machines and Systems. We carried out the experiments on a machine with 8 Intel Core i7-8565U and 16 GB of RAM. Especially to evaluate `ufd`, we use Linux 5.11 because version 4.15.0 does not support the `write_protect` mode. To emulate EPML, we used BOCHS. In the emulated environment, the VM has 1 vCPU and 1GB of memory (due to BOCHS constraints). In all experiments on the real machine, the VM has 1 dedicated CPU (to meet the emulated setup) with 5GB of memory.

Benchmarks. We used both micro- and macro-benchmarks. The former is composed of two applications: an array parser (shown in Listing 6.1) and GCBench [10], a popular micro-benchmark to evaluate GCs. For macro-benchmarks, we used two benchmark suites: tkrzw [25], a suite of key-value data processing engines, and Phoenix [139], a shared-memory implementation of Google's MapReduce data processing model. Concerning tkrzw, we

focused on the five in-memory engines and we injected `set` requests. For each of the macro-benchmarks, we consider three configuration sizes, namely *Small*, *Medium*, and *Large*. Table 5.2 presents the per-application memory size for each configuration. All results presented in this section are mean of 5 runs.

```
...
#define PAGE_SIZE sysconf(_SC_PAGE_SIZE)
#define num_pg xx //memory size=xx*PAGE_SIZE

void main(void)
{
  unsigned long *region=malloc(num_pg*PAGE_SIZE);
  /*
    * Pin all the pages in-memory to be sure that
    * they are not swapped out
    */
  mlockall(MCL_CURRENT|MCL_FUTURE|MCL_ONFAULT);
  for( ; ; )
    for(unsigned long i=0;i<num_pg;i++)
      region[(i*PAGE_SIZE)/sizeof(unsigned long)]=i;
}
```

Listing 5.1: Micro-benchmark code.

5.5.2 Methodology

Since we do not have a real machine equipped with EPML, we build a formula to estimate its impact compared to other techniques. We first present a generic formula that captures the functioning of all techniques. Then we specialize the formula for each technique. Finally, we demonstrate the accuracy of each formula using metric values collected during real experiments. For ease of understanding, we consider in the rest of this section that Tracked is a single-threaded application. Let us recall that Tracker executes in the same thread as Tracked, so each time Tracker runs, it disrupts the execution flow of Tracked.

Let x be a tracking technique. x is either `/proc`, ufd, SPML, EPML, or *oracle*. The latter represents a hypothetical technique able to provide all dirty pages with no additional cost. The Tracker's code (noted C_{tker}) can be organized into two parts: the tracking technique (noted C_x) and the tracking routine (noted C_p). The latter is the part of Tracker that implements the tracking goal, e.g., writing to disk during checkpointing. In the Tracker's execution flow, C_x and C_p alternate. We note C_{tked} the original code of Tracked (i.e., without being monitored by any Tracker). We are interested in the overhead of the tracking technique x on the execution time of Tracker and Tracked.

Configuration Small							
Application	GCbench	hist	kmeans	matrix-m	pca	string-m	word-c
Config.	500K 16 18	0.1GB	-d 500 -c 500 -p 500 -p 100	500 500	-r 1K -c 1K -s 200	50MB	50MB
Memory	15.07MB	102.27MB	4.26MB	5.56MB	8.12MB	56.40MB	100.65MB
Configuration Medium							
Application	GCbench	hist	kmeans	matrix-m	pca	string-m	word-c
Config.	650K 18 20	0.5GB	-d 1K -c 1K -p 1K -s 100	1K 1K	-r 5K -c 5K -s 200	100MB	100MB
Memory	67.76MB	441.28MB	16.41MB	16.21MB	97.85MB	106.14MB	143.99MB
Configuration Large							
Application	GCbench	hist	kmeans	matrix-m	pca	string-m	word-c
Config.	750K 20 22	1.5GB	-d 5K -c 5K -p 5K -s 100	2K 2K	-r 10K -c 10K -s 200	200MB	200MB
Memory	223.41MB	1.49GB	195.64MB	47.33MB	195.50MB	212.09MB	205.88MB

(a) Phoenix & GCBench

Configuration Small					
Application	baby	cache	stdhash	stdtree	tiny
Config.	–iter 3M –th 3	–iter 3M –cap_rec 3M –th 5	–iter 3M –buck 100K –rec zlib –th 2	–iter 3M –th 2	–iter 5M –buck 30M –th 3
Memory	253.64MB	218.21MB	358.64MB	415.12MB	681.35MB
Configuration Medium					
Application	baby	cache	stdhash	stdtree	tiny
Config.	–iter 5M –th 3	–iter 5M –cap_rec 5M –th 5	–iter 5M –buck 100K –comp zlib –th 2	–iter 5M –th 2	–iter 5M –buck 30M –th 5
Memory	421.48MB	361.91MB	595.80MB	694.07MB	977.66MB
Configuration Large					
Application	baby	cache	stdhash	stdtree	tiny
Config.	–iter 10M –th 3	–iter 10M –cap_rec 10M –th 5	–iter 10M –buck 100K –comp zlib –th 2	–iter 10M –th 2	–iter 5M –buck 30M –th 7
Memory	848.56MB	721.46MB	1.18GB	1.38GB	1.27GB

(b) tkrzw

Table 5.2: Configuration setup and memory consumption for each tkrzw 5.2b, Phoenix and GCbench 5.2a applications. For `GCBench`, the parameters are the array size, the lived tree depth and, the stretch tree. `histogram`, `string-match`, and `word-count` use datafile as input parameter.

Overhead of x on Tracker. The execution time of Tracker when it implements x can be computed using Formula 5.1:

$$E(C_{tker}) = E(C_x) + E(C_p) + I(C_x, C_p) \tag{5.1}$$

where $E(C)$ is the execution time of code C (with $E(C_{oracle}) = 0$), and $I(C1, C2)$ is the impact of C_1 on C_2. This impact mainly consists of cache pollution. We experimentally observed that $I(C_x, C_p)$ is negligible. Therefore, the overhead of x on Tracker is reduced to $E(C_x)$. Formula 5.1 can be

developed for each technique as follows:

$$\begin{aligned}
E(C_{/proc}) &= E(C_{echo\ 4 > /proc/PID/clear_refs}) \\
&\quad + E(C_{page\ table\ walk\ in\ userspace}) \\
E(C_{UFD}) &= E(C_{ioctl\ write_protect}) \\
&\quad + E(C_{ioctl\ register}) \\
&\quad + E(C_{ioctl\ write_unprotect}) \\
E(C_{SPML}) &= E(C_{ring\ buffer\ copy}) \\
&\quad + E(C_{reverse\ mapping}) \\
&\quad + E(C_{enable/disable\ PML}) \\
E(C_{EPML}) &= E(C_{ring\ buffer\ copy}) \\
&\quad + E(C_{enable/disable\ PML})
\end{aligned} \tag{5.2}$$

Table 5.4a and Table 5.4b present the measured costs for all events involved in Formula 5.2.

Overhead of x on Tracked. The execution time of Tracked when it is monitored by a tracker using the technique x can be expressed by Formula 5.3:

$$E(C_{tked\_tker}) = E(C_{tked}) + E(C_{tker}) + I(C_x, C_{tked}) \tag{5.3}$$

where $I(C_x, C_{tked})$ consists of page faults, vmexits, etc., which are not negligible. Thus, the overhead of x on Tracked is $E(C_{tker}) + I(C_x, C_{tked})$. Formula 5.4 develops $I(C_x, C_{tked})$ for each technique:

$$\begin{aligned}
I(C_{/proc}, C_{tked}) &= E(C_{PFH\ in\ kernelspace}) \\
&\quad + E(C_{context\ switch}) \\
I(C_{UFD}, C_{tked}) &= E(C_{PFH\ in\ userspace}) \\
&\quad + E(C_{context\ switch}) \\
I(C_{SPML}, C_{tked}) &= E(C_{vmexits}) \\
&\quad + N \times E(C_{vmread/vmwrite}) \\
I(C_{EPML}, C_{tked}) &= N \times E(C_{vmread/vmwrite})
\end{aligned} \tag{5.4}$$

where N is the number of context switches during PML execution, and PFH stands for *Page Fault Handling*.

Validation of the formulas. To validate our formulas, we executed Tracker and Tracked for each technique, and we collected the following metrics: the execution time and the number of occurrences of each event related to the tracking technique. Using these values, we compute $E(C_{tker})$ and $E(C_{tked\_tker})$. Then, we compare the obtained results with real measurements, except for

EPML. Notice that by validating `/proc`, ufd, and SPML formulas, we are also validating, by construction, the formula for EPML. For illustration, Table 5.3a and Table 5.3b present results respectively for SPML and `/proc` when Tracker is CRIU and Tracked is `baby` (from the tkrzw benchmark suite). We can see that Formula 5.2 and Formula 5.4 estimate $E(C_{tker})$ and $E(C_{tked\_tker})$ with an average accuracy of 96.34% and 99%, respectively. We can then consider that the formula that estimates the impact of EPML is relevant.

About the estimation of $I(C_{EPML}, C_{tked})$, we rely on SPML to obtain N, the number of context switches required to compute $I(C_{EPML}, C_{tked})$. Indeed, by construction, this number is the same in both SPML and EPML. We validated this by repeating the previous experiments with EPML and SPML techniques in the emulated environment provided by BOCHS. We collected N with a percentage difference of 2%.

Metric	Time (ms)
$E(C_{tker})$ **measured**	**5503.79**
$E(C_{tked\_tker})$ **measured**	**135255.35**
$E(C_p)$	251.35
$E(C_{copy\_rb})$	0.49
$E(C_{disable\ pml})$	2.06
$E(C_{rev.\ mapping})$	5419
$E(C_{tker})$ **estimated**	**5672.9**
$E(C_{vmexits})$	18000
N	39
$E(C_{vmread,vmwrite})$	1.73×10^{-3}
$E(C_{tked\_tker})$ **estimated**	**136919.85**

(a) SPML

Metric	Time (ms)
$E(C_{tker})$ **measured**	**1097.99**
$E(C_{tked\_tker})$ **measured**	**115283.98**
$E(C_p)$	251.35
$E(C_{clear\_refs})$	1.409
$E(C_{PTwalk})$	0.89
$E(C_{tker})$ **estimated**	**1116.09**
$E(C_{PFHkernel})$	0.27
$E(C_{tked\_tker})$ **estimated**	**114418.58**

(b) `/proc`

Table 5.3: Metrics collected to estimate $E(C_{tker})$ and $E(C_{tked\_tker})$ for techniques SPML and `/proc` using Formulas 5.1, 5.2, 5.3, and 5.4.

5.5.3 Basic costs

We present in this section the cost of all internal metrics that allow us to understand the higher-level performance metrics. The values of these metrics also tell us about the scalability of each tracking technique. The first column of Table 5.4a lists the metrics, organized into nine categories. For each metric, we indicate whether or not its value depends on Tracked memory size (second

column of Table 5.4a). For metrics that are agnostic to the size of Tracked memory, their basic costs are presented in the third column of Table 5.4a. For the other metrics, we report their basic costs in Table 5.4b while varying the Tracked memory size. We also indicate in column four of Table 5.4a which tracking methods the metric is associated with.
Table 5.5 summarizes our analysis of the values reported in Table 5.4a and Table 5.4b:

- `/proc`: it involves 4 metrics, among which 3 depend on Tracked memory size. The most costly metric is page table walk in userspace (M_{16}) which can take up to $594ms$. Page fault handling in kernel space (M_5) is the second most costly metric. It can take up to $33ms$, which is quite significant since it may be involved frequently during the monitoring phase. This is why it dramatically impacts `/proc` scalability (from both Tracked and Tracker perspectives).

- `ufd`: it involves 3 metrics, among which 2 depend on Tracked memory size. The most costly metric is page fault handling in userspace (M_6), which costs up to $3,483ms$ when Tracked memory size is 1GB. This metric is further involved during the monitoring phase, thus impacting the scalability of `ufd` (from both Tracked and Tracker perspectives).

- SPML: it involves 10 metrics, among which 4 depend on Tracked memory size. The most costly metric is reverse mapping (M_{17}) which takes up to $15s$ for 1GB Tracked memory size. This metric will only impact the scalability of Tracker because it is not involved in the monitoring phase. Figure 5.3 presents the proportion of time taken by each step of the collection phase in SPML. We can observe that reverse mapping is the bottleneck of SPML and represents, on average, more than 68% of the total collection time.

- EPML: it involves 8 metrics, among which only one depends on Tracked memory size. The most costly metric is PML initialization (M_{10}) which also includes VMCS shadowing initialization. It costs about $5,878ms$. Because this metric does not depend on Tracked memory, it does not impact the scalability of Tracker. The metrics that impact the scalability of Tracked are `vmread` (M_7) and `vmwrite` (M_8), whose costs are very low (less than $1\mu s$). This makes EPML scalable.

Metric	Depend on mem.	Cost (μs)	Technique
$M1$. context switch (from user to kernel space)	No	0.315	All
ioctl syscalls			
$M2$. write_protect	Yes	-	ufd
$M3$. init. PML	No	5,651	SPML & EPML
$M4$. deactivate PML	No	2,816	SPML & EPML
page fault handling			
$M5$. in kernel space	Yes	-	/proc, ufd
$M6$. in userspace	Yes	-	ufd
vmx operations			
$M7$. vmread	No	0.936	EPML
$M8$. vmwrite	No	0.801	EPML
hypercalls			
$M9$. init. PML	No	5,495	SPML
M_{10}. + init. VMCS shadowing	No	5,878	EPML
M_{11}. PML deact.	No	2,060	SPML
M_{12}. + VMCS shadowing deact.	No	2,755	EPML
M_{13}. enable PML logging	No	0.3	SPML
M_{14}. disable PML logging	Yes	-	SPML
M_{15}. `clear_refs`	Yes	-	/proc
M_{16}. page table walk (in userspace)	Yes	-	/proc & SPML
M_{17}. reverse mapping	Yes	-	SPML
M_{18}. ring buffer copy	Yes	-	EPML & SPML

(a) Metrics that are agnostic to Tracked memory size. `clear_refs` is the shortcut for `echo 4 > /proc/PID/clear_refs`.

	1MB	10MB	50MB	100MB	250MB	500MB	1GB
M_{15}	0.032	0.0912	0.174	0.288	0.613	1.153	2.234
M_{16}	1.912	14.479	41.832	82.289	161.973	307.109	594.187
M_5	0.003	0.3	1.68	3.34	8.39	16.79	33.58
M_6	2.5	27.3	152.3	347.1	882.8	1,585	3,483
M_{14}	0.042	0.047	0.138	0.156	0.189	0.203	0.208
M_{18}	0.003	0.01	0.03	0.048	0.109	0.383	0.671
M_{17}	6.183	24.653	85.117	255.437	1,211	4,123	15,738

(b) Metrics that depend on Tracked memory size. Times are given in milliseconds (ms).

Table 5.4: Cost of internal metrics.

	/proc	ufd	SPML	EPML
associated metrics	M_1, M_5, M_{15}, M_{16}	M_1, M_2, M_5, M_6	M_1, M_3, M_4, M_9, M_{11}, M_{13}, M_{14}, M_{16}, M_{17}, M_{18}	M_1, M_3, M_4, M_7, M_8, M_{10}, M_{12}, M_{18}
metrics depending on Tracked mem. size	3 (M_5, M_{15}, M_{16})	3 (M_2, M_5, M_6)	4 (M_{14}, M_{16}, M_{17}, M_{18})	1 (M_{18})
metrics involved in the monitoring phase	1 (M_5)	2 (M_5, M_6)	2 (M_{13}, M_{14})	2 (M_7, M_8)
the two most costly metrics	M_5, M_{16}	M_5, M_6	M_{16}, M_{17}	M_{10}, M_{12}
metrics which impact scalability from Tracker point of view	**3** (M_5, M_{15}, M_{16})	**3** (M_2, M_5, M_6)	**4** (M_{14}, M_{16}, M_{17}, M_{18})	**1** (M_{18})
metrics which impact scalability from Tracked point of view	**3** (M_5, M_{15}, M_{16})	**2** (M_5, M_6)	**2** (M_{13}, M_{14})	**2** (M_7, M_8)

Table 5.5: Influence of `/proc`, `ufd`, SPML and EPML on internal metrics. $\{M_i\}$ are defined in Table 5.4a.

5.5.4 Micro-benchmark Results

This section evaluates the impact of each dirty page tracking technique on a micro-benchmark application. We vary the memory size of Tracked, and we measure its execution time with and without the tracking technique. Figure 5.4 plots the slowdown incurred by each technique. We can observe that, except EPML, the overhead on Tracked increases with its memory size. For almost all memory sizes, SPML incurs the greatest slowdown, up to 66×. This high overhead is due to reverse mapping that can take up to 15s for 1GB of memory (see Figure 5.3). Figure 5.4 reveals that `ufd` also highly increases the execution time of Tracked (by up to 15×). When the memory size is less than 250MB, `ufd` appears to be the worst technique. This is explained by the fact that page fault handling in userspace, which is the bottleneck of `ufd` technique (see Table 5.4b), is more costly than reverse mapping for a working set of less than 250MB. The overhead of EPML is negligible regardless of the memory size of Tracked, confirming its scalability. With a maximum overhead of about 0.6%, EPML appears to be the best technique.

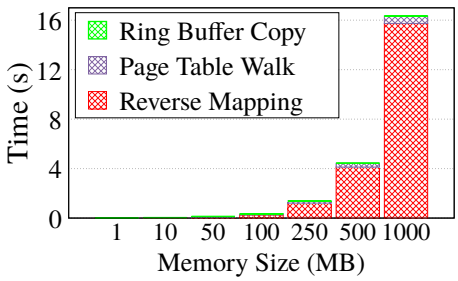


Figure 5.3: Time of metrics *Reverse mapping*, *PT walk*, and *Ring buffer copy* when using SPML technique. Reverse mapping appears to be the bottleneck of SPML.

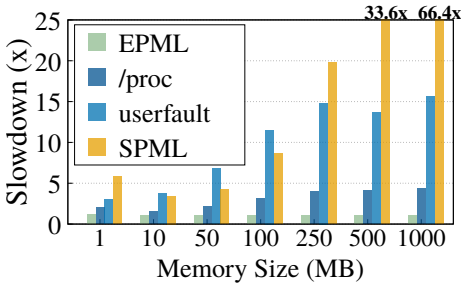


Figure 5.4: Overhead of each tracking technique on the micro-benchmark.

5.5.5 Boehm Results

In this section, we evaluate the impact of `/proc`, SPML, and EPML on Boehm GC using `GCBench` and Phoenix macro-benchmarks. We do not evaluate Boehm using the tkrzw benchmark because the former only works properly for C applications [2]. We implemented SPML and EPML in Boehm. We evaluate both the impact on Tracker and Tracked.

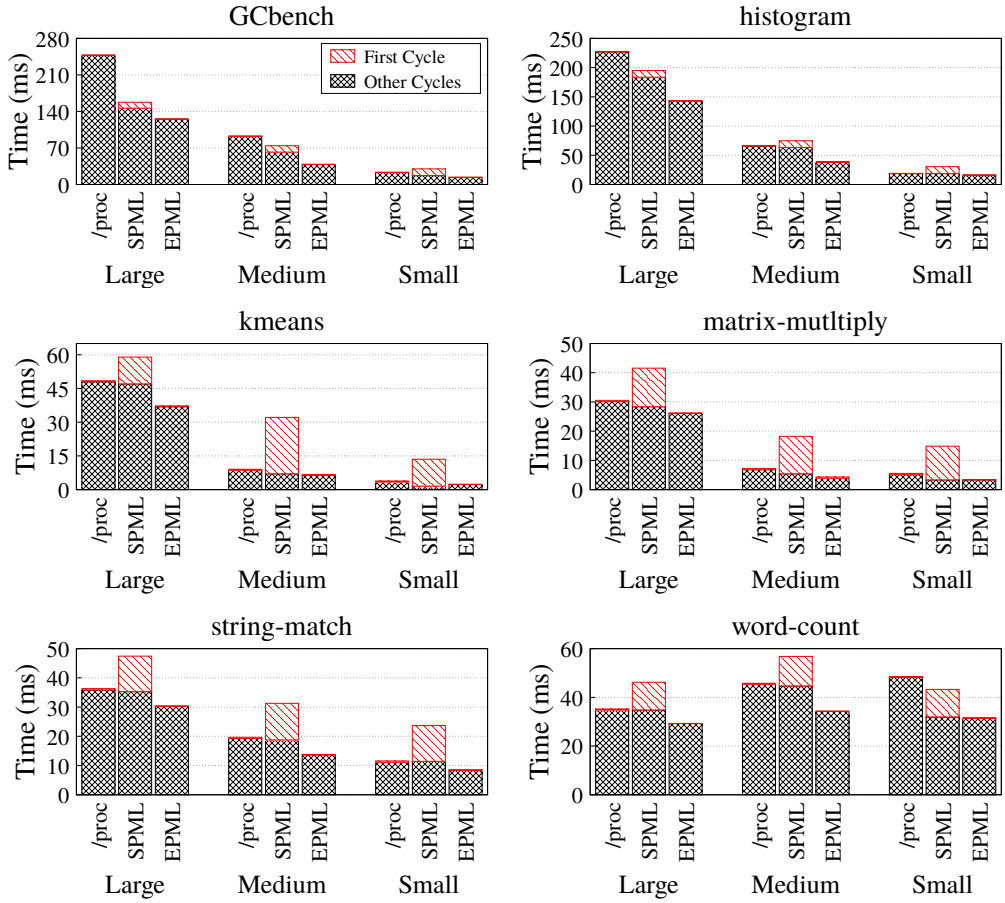


Figure 5.5: Execution time of Boehm GC when implemented with `/proc`, SPML, and EPML. The figure highlights the first garbage collection cycle during which Boehm performs the reverse mapping with SPML, the reason why its execution time is higher for SPML compared to the two other techniques.

Impact on Tracker (Boehm GC). Figure 5.5 presents the impact of each technique on Boehm GC. Due to page limitations, we do not present the results for all applications. Boehm GC can perform from 2 (e.g., for `histogram` config. *Small*) to 23 (e.g., for `GCBench` config. *Large*) cycles of garbage collection depending on the allocation intensity of the workload. Figure 5.5 plots the garbage collection time during the execution of each application. We

emphasize the first cycle as it highlights the overhead of SPML on Boehm, which performs the reverse mapping during this cycle [1]. We can observe in Figure 5.5 that if we ignore the first cycle, SPML outperforms `/proc`. This explains why EPML is the best solution, as it avoids reverse mapping (see Figure 5.3). Nonetheless, SPML outperforms `/proc` by up to 36% for applications `histogram` (configuration *Large*), `word-count` (configuration *Medium*), and `GCBench` (configurations *Large* and *Medium*). EPML outperforms both `/proc` and SPML, respectively, by up to 58% and 47% (with `GCBench` config. *Medium*).

Impact on Tracked. The execution time of applications (Tracked) that use Boehm GC will be impacted at least by the duration of the GC (see Figure 5.5). Figure 5.6 assesses the level of this impact according to the technique that Boehm uses. We can see that compared to `/proc`, SPML increases the overhead of Boehm GC on most applications. But for `GCBench` configuration *Medium*, SPML reduces the overhead of Boehm compared to `/proc` by about 1.7%. About `matrix-multiply` that runs in 51ms, SPML increases by about 63% the overhead of Boehm compared to `/proc` . This increase is only 2% for `GCBench` configuration *Medium*, which runs in 6s, and 0.5% for `histogram` configuration *Large*, which runs in 7.3s. EPML significantly reduces the overhead of Boehm compared to `/proc` and SPML for all applications by about 62% with the `string-match` application.

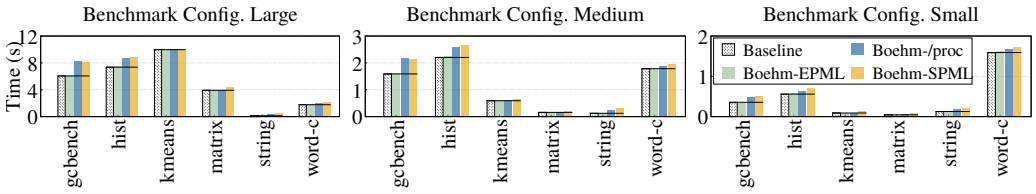


Figure 5.6: Impact of Boehm GC on Tracked execution time when using `/proc`, SPML and EPML techniques. The baseline is the ideal execution time of the application, i.e. when not tracked.

5.5.6 CRIU Results

In this section, we evaluate the impact of `/proc`, SPML, and EPML on CRIU using macro-benchmarks. For Phoenix applications, we use the *Large* configuration, and we increase the number of map-reduce tasks to make the applications have long execution times for checkpointing.

[1] During the following cycles, Boehm just reuses the addresses collected during the first cycle.

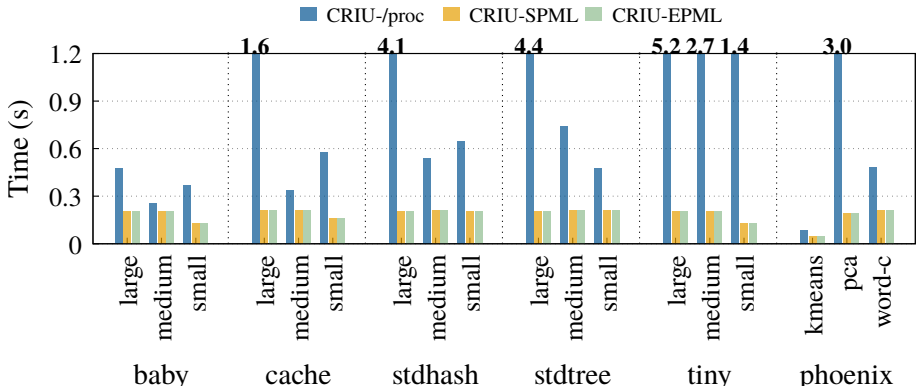


Figure 5.7: Memory write time during CRIU checkpointing.

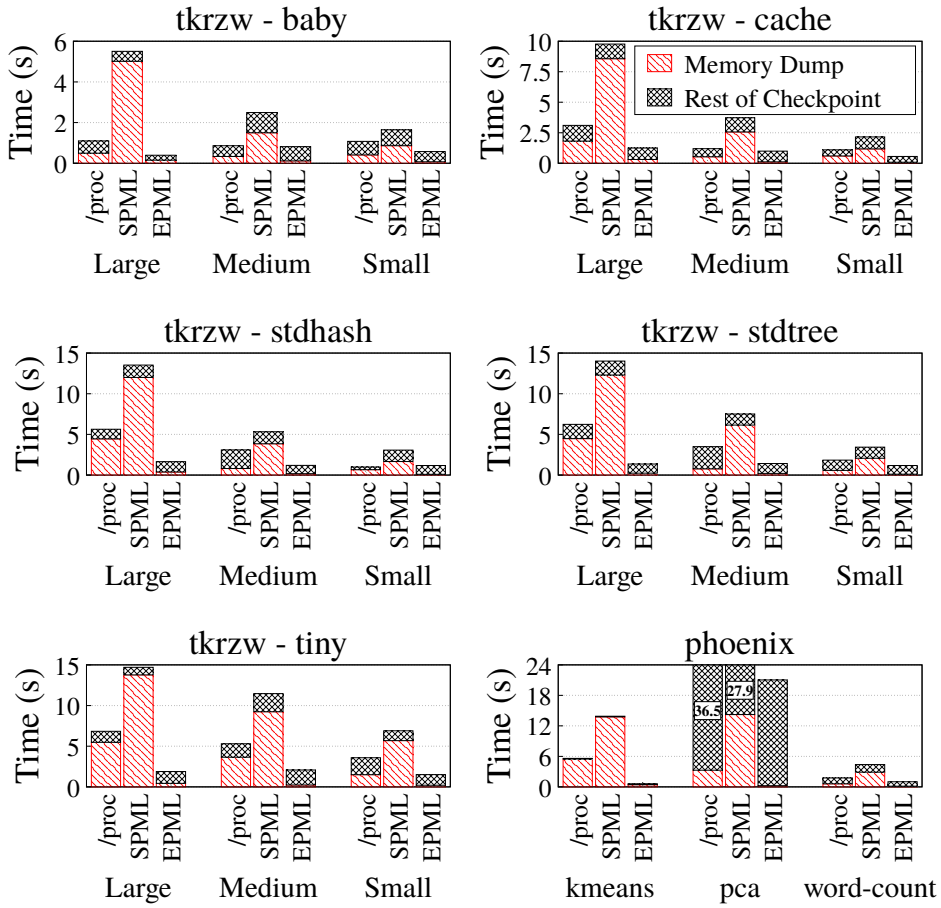


Figure 5.8: Execution time of CRIU when implemented with `/proc`, SPML, and EPML. The figure MD phase during which CRIU performs the reverse mapping with SPML.

Impact on Tracker (CRIU). We evaluated each stage of the CRIU checkpointing algorithm, which includes, among others, the memory write (MW)

phase and the memory dump (MD) phase. CRIU collects the dirty pages to be dumped during the MD phase and writes them to the disk during the MW phase. Depending on the tracking technique implemented by CRIU, these two phases can be done sequentially or simultaneously. When CRIU implements the `/proc` technique, it walks the process page table to get dirty pages and writes them to the disk as it finds them. While with SPML and EPML, it first collects all dirty pages from the ring buffer and then writes them to the disk. This is the reason why SPML and EPML significantly improve the MW phase compared to `/proc`, as shown in Figure 5.7. We measured an improvement of up to 26× with the `tiny` application, configuration *Large*. In addition, MW time is almost constant with SPML and EPML, while with `/proc` it can take up to 5.7s.

Figure 5.8 presents the complete checkpointing time, highlighting the MD phase. When using SPML, CRIU performs reverse mapping during the MD phase, which drastically increases the checkpointing time. This leads to a non-negligible overhead, as we can see in Figure 5.8. Indeed, complete checkpointing is up to 5× slower with SPML compared to `/proc` for both tkrzw-`baby` (configuration *Large*) and Phoenix-`kmeans` (configuration *Large*). Since reverse mapping represents, on average, more than 66% of MD time with SPML, avoiding it may lead to better performance compared to `/proc`. This is why EPML allows CRIU to execute faster compared to `/proc` and SPML. Indeed, EPML brings up to 4× speedup to CRIU checkpointing compared to `/proc` and up to 13× speedup compared to SPML, respectively, with `tiny` and `baby` configuration *Large* (in Figure 5.8).

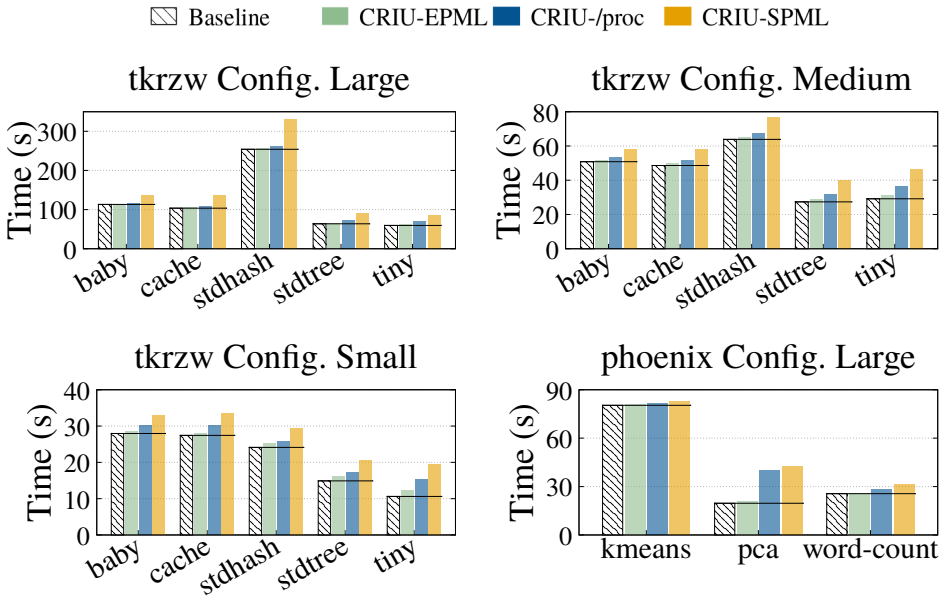


Figure 5.9: Impact of CRIU on Tracked execution time when using `/proc`, SPML, and EPML techniques. The baseline is the ideal execution time of the application, i.e., when not tracked.

Impact on Tracked. Applications are paused during checkpointing. Therefore, CRIU increases the execution time of the checkpointed application. Figure 5.9 presents the overhead of CRIU. The default implementation of CRIU (that uses `/proc`) can significantly impact the checkpointed application, by up to ∼102% for the `pca` application. Concerning SPML, it incurs a higher overhead on the application execution time compared to `/proc`. Figure 5.9 shows that this overhead can vary from ∼1% (with `kmeans`) to ∼114% (with `pca`). EPML leads to the best results. Its overhead does not exceed 14%, with an average of only 3%.

5.5.7 Scalability

In Sections 5.5.5 and 5.5.6, we evaluated the scalability of SPML and EPML with different working set sizes. In the present section, we vary the number of tenant VMs from one up to 5. The evaluation scenario is the same as in the previous sections. We use Boehm and the Phoenix-`histogram` application (Config. *Large*). Results are presented in Fig. 5.10 and 5.11. We can observe that the performance impact of both SPML and EPML on Tracker (Fig. 5.10) and Tracked (Fig. 5.11) is the same as what we obtained with one VM (Fig. 5.5 and Fig. 5.6 config. Large). In addition, this performance remains quite constant when the number of VMs increases.

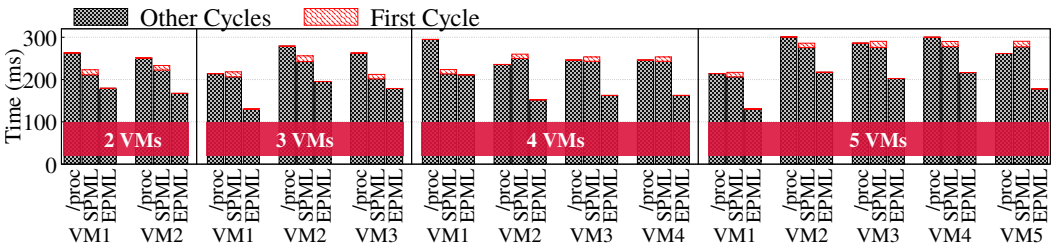


Figure 5.10: Impact of each tracking technique on Tracker when varying the number of VMs.

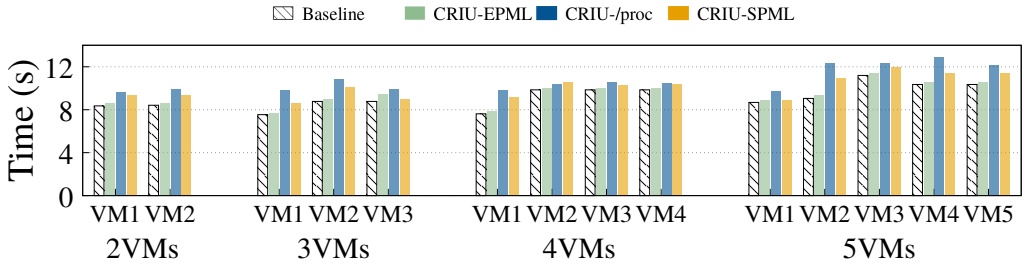


Figure 5.11: Impact of each tracking technique on Tracked when varying the number of VMs.

5.6 Related work

Nested virtualization. The main source of performance degradation in virtualized environments is VM traps. The latter lead to VM execution suspension and also to cache pollution [109] due to context switches. The number of VM traps increases at least by a factor of two in nested virtualized environments [150]. The reduction of VM traps is a hot topic in both non-nested [95, 78, 109] and nested virtualized systems [114, 150, 116]. Device passthrough is a simple approach for improving I/O performance in nested and non-nested virtualized environments by providing direct access to the VM. However, it dedicates the entire device to a single VM, resulting in sub-optimal resource utilization. In addition, device passthrough does not permit VM live migration, which is an important operation for cloud providers as it is used for maintenance. VMCS shadowing [1], that EPML leverages, has been introduced by Intel to reduce traps when a nested hypervisor accesses some VMCS fields. SV_T, by Vilanova et al. [150], exploited simultaneous multithreading (SMT) processors to minimize VM traps. SV_T runs every nested virtualization level on a separate SMT thread, and it replaces VM trap and VM resume to avoid context switches between nested hypervisors and the host hypervisor (the one that directly runs atop the hardware). In SV_T, only one SMT thread can run at a given time leading to core waste.

DVH, by Lim et al. [114], proposed that the host hypervisor provides virtual devices directly to nested VMs without the intervention of intermediate hypervisors. The intermediate hypervisors only intervene at virtual device initialization time to make it visible and directly accessible to the nested VM. The authors illustrated DVH with four devices: virtual IO, virtual timer, virtual IPI, and virtual idle. Although DVH is promising, its application to all devices that compose full hardware is unpractical. With OoH, we are advocating for exposing only hardware virtualization features that could help applications, which is tractable.

Dirty page tracking. This activity is necessary for both hypervisors and processes. The hypervisor relies on it to perform pre-copy-based live migration and also checkpointing. Dirty page monitoring is at the heart of concurrent garbage collectors and other userspace processes such as CRIU for container and processes checkpointing or Redis for dumping the database. So far, the main approach used for monitoring dirty pages is two steps: invalidation of PTE dirty bit and present bit, and page faults interception. To minimize the overhead of this approach, some alternatives have been proposed. For the hypervisor, Intel introduced PML, the hardware feature that we study in the present chapter. Bitchebe et al. [74] showed that PML could decrease both VM live migration and checkpointing duration. The authors also extended PML

to log read pages in order to efficiently estimate VM working set size. In non-virtualized environments, Lu et al. [119] built a memory allocator which maps several objects to the same physical page, thus reducing the number of tracked pages. Nevertheless, this solution does not avoid frequent interruptions of the tracked process due to page faults.

5.7 Summary

In this chapter, we illustrated the application of OoH with PML for guest applications. We prototyped OoH for PML following two solutions, namely Shadow (SPML) and Extended PML (EPML). The former requires no hardware changes but incurs significant performance overhead. It is not the case of EPML that extends the original PML to avoid SPML limitations. We evaluated and compared SPML and EPML with two popular dirty page tracking techniques, namely `/proc` and `ufd`. We implemented OoH for PML using the Xen hypervisor, Linux OS, and the Bochs emulator. We considered CRIU and Boehm GC as the use cases, where the target applications are Phoenix (a shared-memory data processing model) and tkrzw (a key value data processing model). The evaluation results showed that the different techniques could be classified as follows, from the most to the least costly: SPML, `ufd`, `/proc`, and EPML. Indeed, EPML brings up to 13× speedup to systems using the other techniques, while reducing their overhead on applications by up to 16×.

Chapter 6

OoH for SPP: Efficient Buffer Overflow Mitigation In Virtualized Clouds

This chapter illustrates OoH with SPP by introducing GuaNary*, a novel security feature to efficiently mitigate write buffer overflows in cloud user applications. Based on this novel feature, we propose a seure memory allocator that offers better security guarantees and memory consumption than the state-of-the-art.*

Contents

6.1 Introduction

For decades, the widespread usage of memory-unsafe languages like C and C++ raised the threat of security-related memory corruption errors [148].

These are still ongoing issues today: Google developers recently revealed that 70% of Google Chrome's bugs are related to memory management [50]. Microsoft also made a similar observation [47]. Memory errors such as buffer overflows may represent vulnerabilities that attackers can exploit to execute malicious code, tamper with, and/or leak critical data. This chapter focuses on buffer overflow, ranked the top vulnerability in 2022 by SANS Institute [51] (see § 6.2.1).

Over the last few years, the occurrence of heap-related vulnerabilities has drastically increased [144, 145, 118]. In response, secure heap memory allocators [144, 118, 145, 134, 124, 89, 161, 57] have been proposed to protect against such vulnerabilities. Allocators targetting buffer overflow mitigation need to answer an important question: *How to detect and prevent an overflow?* Two common techniques have been studied to answer this question: canaries and guard pages [124] (we call them *guardians* hereafter). Canaries are small 1-byte magic values, located after a buffer and checked to detect overflow. They have a modest memory overhead but can only detect overflows asynchronously, i.e., when the value is checked. Guard pages are unmapped pages in the virtual address space, located after a buffer. Overflowing the buffer will trigger a fault if the page is hit. Guard pages offer better security guarantees vs. canaries, as they prevent overflows through synchronous detection but at the cost of significant memory consumption. We measured up to 80× memory overhead for the PARSEC-freqmine application (see § 6.2.3) with guard pages.

Only a few allocators (or even improvements of existing allocators) have been developed with the primary goal of reducing the memory overhead while preserving other important properties such as security and performance. Some allocators, such as OpenBSD [124], Cling [57], and DieHarder [134], attempted to reduce the memory footprint of linked list-based metadata by using bitmaps. However, they lead to significant performance degradation when the allocator performs randomization, which is a popular security guarantee technique. Hardware solutions have also been introduced to address buffer overflow. We can underline CHERI [159], which doubles pointer size to include the bounds to the pointed buffer. This way, the hardware can check bounds violations. Such hardware solutions overcome buffer overflow. However, they include several limitations that we detailed in § 6.6, mainly related to performance degradation and unpredictability, and the need to rewrite applications (which limits their adoption). Intel MPX [135] suffers from the same limitations that have led to their obsolescence.

In this chapter, we introduce GuaNary, a novel type of guardian for virtualized cloud-based applications. Nowadays, virtualized clouds are the de facto execution environment of modern applications, thanks to their very attractive costs and administration tasks simplification. GuaNary provides the same

security guarantee as guard pages against write overflows while drastically reducing memory overhead and with negligible performance overhead. GuaNary leverages a recent Intel hardware virtualization feature called Sub-Page Write Permission [32] (hereafter SPP). SPP reduces write-protection granularity to 128B (called a sub-page) instead of 4KB. Intel SPP was initially introduced to help hypervisors accelerate virtual machine's (VM) live migration/checkpointing. In this chapter, we repurpose SPP for security and make it exploitable by unprivileged VMs without breaking isolation between them.

The design of GuaNary is not as straightforward as it might seem and raises two conceptual and two technical challenges that we present in § 6.3.1: (C_1, Conceptual) Costly hypercalls: for isolation purposes, SPP is only configured by the hypervisor whereas for our work it is requested by the guest operating system (OS). (C_2, Technical) Page heterogeneity: the guest kernel should now be able to manage non-SPP and SPP pages. (C_3, Technical) SPP bitmap heterogeneity: SPP pages need to offer different protection layouts and densities to cope with allocation requests of various sizes. (C_4, Conceptual) Coexistence with the hypervisor's needs: a system using SPP should make sure not to interfere with the original needs of the hypervisor. We address these challenges in LeanGuard, a system that leverages GuaNary to provide the user with a memory-efficient secure heap allocation. We build LeanGuard by modestly extending the Xen [61] hypervisor, the Linux kernel, and the SlimGuard [118] secure allocator. We thoroughly evaluated LeanGuard using both micro- and macro-benchmarks (PARSEC applications [69]). The results show that GuaNary (via LeanGuard), with the same memory consumption, can protect 25× more buffers compared to SlimGuard. Inversely, to protect the same amount of buffer as SlimGuard, GuaNary requires about 8.3× less memory.
In summary, in this chapter, we make the following contributions:

- We introduce GuaNary, a novel guardian type that repurposes Intel SPP for buffer overflow detection on the heap.
- We design LeanGuard, a system that exemplifies GuaNary in popular system software stacks.
- We evaluate LeanGuard using micro- and macro-benchmarks, demonstrating its benefits.

6.2 Background and Motivations

This section provides the necessary background and the motivations of GuaNary.

6.2.1 Buffer Overflow Importance

According to SANS Institute's report [51], buffer overflow is the top vulnerability in 2022. A buffer overflow occurs when a program reads/writes over/under the boundaries of a buffer. This programming error is common in unsafe languages such as C/C++. It makes the application vulnerable to several issues, such as crashes, information leakage, denial-of-service, malicious code injection, privilege escalation, etc. This vulnerability concerns all application types, including popular ones. For instance, in January 2021, a severe heap-related buffer overflow vulnerability (CVE-2021-3156[1]) was discovered in `sudo` (versions before 1.9.5p2), a very popular Linux tool for managing root privileges. That bug, introduced in 2011, could be exploited to elevate root privileges even for users who are not listed in the `sudoers` file.

An overflow can happen in the stack as well as the heap. Liu et al. [118] reported that the number of heap-related vulnerabilities has tripled from 2010 to 2018 to more than 600. Our own analysis found that more than 63% of 2021's overflow vulnerabilities were heap-related. GUANARY focuses on write overflow in the heap because it is the most dangerous. Contrary to a read overflow, an attacker can exploit a write overflow vulnerability to inject arbitrary code into a system (by writing to the memory adjacent to the vulnerable buffer).

6.2.2 Secure Allocators

Fundamentally, a heap allocator implements memory allocation functions such as `malloc` and `free` and is linked with the application at compilation or run time. At application boot time, the heap allocator initializes its own data structures and one or several virtual memory segments using `brk/sbrk` or `mmap` syscalls. When the process calls `malloc`, the allocator selects a free area from a heap segment, adapts its metadata accordingly, and returns the address of the chosen area to the process. The first access, by the process, to the virtual page containing that area, will generate a page fault that will trap inside the kernel. The page fault handler will allocate and associate a physical page with that virtual faulting page. When the process frees the area by calling `free`, the allocator updates its metadata accordingly. If the allocator ends up with too many free segments, it can consider releasing some of them to the kernel.

A secure allocator (hereafter allocator) interposes in this flow by trying to answer three main questions: How to organize objects (freed, available, and metadata)? How to increase security? How to detect overflows when they occur?

[1] https://nvd.nist.gov/vuln/detail/CVE-2021-3156

Objects Organization. Although seeking greater security guarantees, allocators strive to be efficient in terms of memory overhead and allocation latency. BIBOP-style (Big Bag of Pages) has proven to meet these requirements thanks to its massive adoption in recent secure allocators such as OpenBSD [19], DieHarder [134], FreeGuard [144], Guarder [145], and Slimguard [118]. It includes the following object management features.

Allocated and freed objects: BIBOP-style allocators group objects according to their sizes. Objects of the same size *class* are allocated within the same virtual memory segment (corresponding to a VMA in the kernel), treated as "bags". This organization not only accelerates heap object allocations but also reduces internal memory fragmentation. Freed objects are maintained separately, using either lists or bitmaps.

Meta-data: BIBOP allocators store metadata (e.g., location and availability information, size, etc.) of allocated objects in a separate area. Isolating such information from the objects enhances security as it prevents metadata-based attacks such as FastBin [7], Unlink [26], and Chunk Overlap [3].

Security Increase. To further strengthen security, allocators employ mechanisms to confuse intruders who might like to infer allocations to carry out attacks.

Randomization: It provides unpredictability of allocated objects' locations to protect against use-after-free and related vulnerabilities. By offering the memory allocator a non-deterministic workflow, randomization can prevent malicious codes from foreseeing a heap address to settle and launch a subsequent attack.

Over-provisioning: It consists of allocating memory slots so that some heap objects will never be used. This increases randomization.

Overflow Detection. Existing allocators mainly use two types of *guardians* that they place at the boundary of buffers to detect an overflow, as drawn in Fig. 6.1.

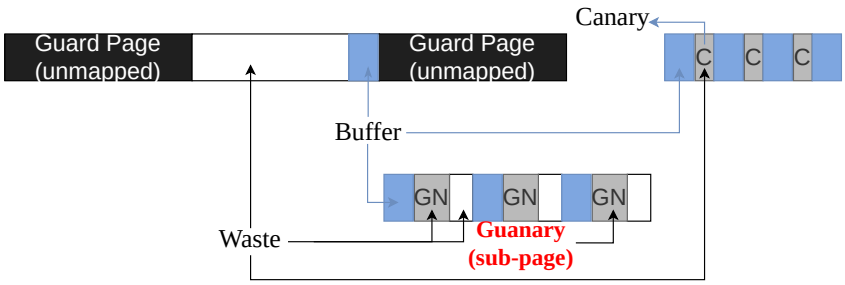


Figure 6.1: Canary, Guard pages, and GuaNary illustration. For the two latter, buffers are aligned with the lower boundary of the (sub)page.

Canaries: A canary is a known value (e.g., one-byte size) placed at the boundary of an allocated buffer. Its value is generally randomly generated by the allocator. A simple check on buffer free allows the allocator to detect if an overflow happened. As one can see, canary-based detection can only detect an overflow *asynchronously*, leaving the attacker time to perform an exploit. However, its advantage is its low memory footprint overhead.

Guard pages: A guard page is an unmapped or read- and write-protected memory page placed at the boundary of the buffer. This way, an overflow will *synchronously* trigger a page fault (#PF). However, guard pages lead to significant internal memory fragmentation as the residual memory of the page which holds the allocated buffer is wasted. In fact, all buffers should be placed at the boundary of a guard page in order to be protected.

Hardware Capabilities: R. Watson et al. [159] released CHERI (Capability Hardware Enhanced RISC Instructions), an extension of conventional hardware Instruction-Set Architectures (ISAs) with capabilities. In CHERI, each pointer includes the pointed buffer's memory bounds. Its dereference is controlled by the hardware, which can synchronously detect an overflow. Recently, the authors of CHERI have prototyped complete software stacks by adapting Clang/LLVM and FreeBSD. CHERI allows synchronous detection with low memory consumption (depending on the number of manipulated buffers). However, to be compatible with CHERI, applications need to be re-written, with respect to their idiomatic use of C (see Section 4 of [156]). For example, the CHERI team made a lot of effort to port the popular SPEC benchmark suite with moderate success: *"Even though some of the benchmarks could be compiled, it required extensive build system modifications to use the CHERI clang compiler. Unfortunately, the CHERI clang compiler cannot compile the C++ benchmarks."* in [98]. A second limitation of CHERI is its significant performance overhead for applications that heavily manipulate pointers [98]. We present other limitations in § 6.6.

In the present chapter, we focus on guard page and canary, the two guardian types that work with legacy applications (an important property for quick adoption) and incur acceptable performance overhead.

6.2.3 Synchronous Detection vs Memory Overhead: The Dilemma

An overflow vulnerability is exploitable only if a guardian cannot synchronously detect it. *Guard pages* are, so far, the guardian type providing synchronous detection. Let us consider buf a vulnerable buffer. We define the *security distance of buf* as the number of bytes that separate it from a guard page. A zero security distance allows to immediately catch overflow attempts. The protection of all application buffers with a zero security distance is not practical for most existing allocators, as it would result in considerable memory overhead. Let us imagine that we want to protect all buffers allocated by an application,

i.e., using a guard page for every buffer returned by `malloc`. Fig. 6.2 shows for PARSEC applications [69] what would be the memory overhead induced by such a policy when SlimGuard [118] is used as the allocator. The overhead is normalized to the application's actual amount of memory effectively consumed by the buffers. For example, to protect all `blackscholes`' buffers with a security distance of zero, SlimGuard would consume about 75.8× memory.

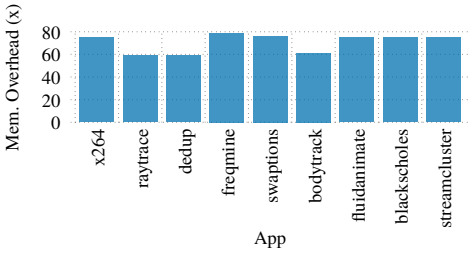


Figure 6.2: Memory over-consumption that SlimGuard would incur for PARSEC applications if all the allocated buffers of PARSEC applications are placed at the boundary of a guard page (worse case).

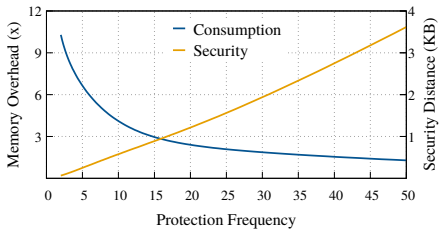


Figure 6.3: Memory overhead and average security distances for PARSEC-`blackscholes` when varying the protection frequency from 2 to 50.

To cope with this problem, allocator designers use to combine guard pages with canaries and allow users to configure the security distance they desire (implicitly the memory waste they accept) according to their performance or memory budget. A protection frequency N places a guard page for every N allocated buffers in each class, and canaries are used between buffers that are not at the boundary of a guard page. Fig. 6.3 presents, for different protection frequency values, the memory overhead, and the security distance for the PARSEC-`blackscholes` application. As one might have imagined, the lower the frequency, the more memory waste and better the protection.

This situation is a real conundrum for users who thereby have to sacrifice security for better memory utilization or vice versa. The objective of GuaNary is to *reduce* this pressure. Given a memory budget, GuaNary provides more security than existing solutions. To this end, GuaNary leverages Intel Sub-Page write Permission (SPP), a hardware virtualization feature. Consequently, the scope of GuaNary is virtualized clouds, which is the de facto execution environment nowadays according to its attractive costs.

6.3 GuaNary

We introduce GuaNary, a novel type of safety guard that is midway between **Gua**rd page and ca**Nary** (see Fig. 6.1), thus including the advantages of both

solutions: synchronous buffer overflow detection and low memory consumption. GUANARY is comparable to guard pages as it uses a sub-page as a guardian to detect overflows synchronously. It is similar to a canary in that it resides in the RAM and the memory space it occupies can hold allocated buffers, thus GUANARY consumes physical memory. However, its value does not matter, unlike canary.

Because an SPP sub-page is 128 bytes, one can guess GUANARY consumes more physical memory than a canary. Nonetheless, as we can observe in Fig. 6.4, this is not the case (note that we are talking here about the consumption of the guardian itself and not the memory overhead induced by the latter). To plot Fig. 6.4, we run the PARSEC-`blackscholes` application and track the number of distinct SPP pages used by the kernel allocator (see §6.3.4 below) for different protection frequencies F. Using the SPP bitmap of these pages, we compute the memory size of the protected sub-pages. The protection frequency in Fig. 6.4 concerns only GUANARY because when the allocator uses canaries, it necessarily places them for each buffer (i.e., the protection frequency with canaries is always 1). Unlike canaries for which the memory usage remains constant, the memory usage of GUANARIES decreases inversely with F. This is because, as §6.3.5 explains, the kernel allocator can re-use the same SPP page for different allocations.

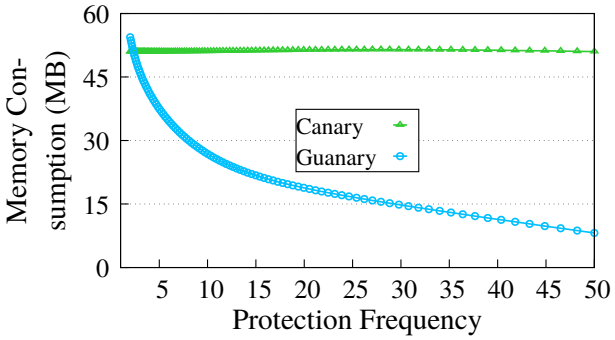


Figure 6.4: Physical memory consumption of GUANARY compared to canary for PARSEC-`blackscholes`.

Our second contribution in this chapter is LEANGUARD, a complete software stack that uses GUANARY to mitigate buffer overflows in virtualized clouds. LEANGUARD includes the hypervisor (Xen), the guest kernel (Linux), and the memory allocator (SLIMGUARD). The following section exposes the challenges related to the design of LEANGUARD.

6.3.1 Challenges

We identify four main challenges, two of which are conceptual and the other two technical.

(C_1, Conceptual) Costly hypercalls. Conceptually, Intel SPP can only be controlled by the hypervisor, whereas the heap allocator runs inside the guest VM in the userspace. A naive implementation invoking the hypervisor (using hypercalls) untimely each time an SPP page is requested (on page fault, hereafter #PF) will dramatically degrade application performance as hypercalls are very costly. The cost of an empty hypercall is about 587 nanoseconds, while the cost of an empty syscall is about 392 nanoseconds on average. The challenge is to use GuaNary with almost the same performance level as existing secure heap allocators. SlimGuard vanilla is our baseline.

To address this challenge, LeanGuard implements two optimizations. First, it groups pages into pools to issue a single SPP configuration hypercall per pool. We call an *SPP page* a page configured to be tracked by the processor using Intel SPP. Second, we manage to provide the guest kernel ahead with a pool of pre-configured SPP pages.

(C_2, Technical) Page heterogeneity. A kernel that uses GuaNary must deal with two kinds of memory pages: SPP pages and non-SPP pages (hereafter, normal pages). Technically, this heterogeneity complexifies two kernel subsystems: the memory allocator (e.g., buddy allocator in Linux) and the memory reclaimer (e.g., `kswapd` in Linux). Regarding the former, an SPP page can only be allocated to a process that uses our secure allocator, and the allocator should provide other processes with normal pages. In addition, allocations to `secure` processes should not delay or overlap with allocations to other processes. Concerning the memory reclaim subsystem, reclaiming an SPP page is much more complex than reclaiming a normal page. So, while integrating GuaNary, we must build an efficient algorithm to optimize performance.

We address this challenge by organizing the kernel's memory allocation subsystem into two cooperative components: one managing normal pages and the other dedicated to SPP pages.

(C_3, Technical) SPP bitmap heterogeneity. If the user configures the protection frequency to be $F = 1$, for example, he expects all the buffers of its application to be placed at the boundary of a guardian. Let us consider buffers with sizes $s < 128B$ (i.e., they can fit in an SPP sub-page). For such buffers, the SPP bitmap used for allocation looks like this: 1010...1010... (we only show even bits that the processor interprets), and the corresponding SPP pages in their VMA all have the same bitmap as in Fig. 6.5.

If, on the other hand, $128B < s < 256B$ (i.e., the allocator needs at least two SPP sub-pages for each buffer), the bitmap of the first page in the corresponding VMA looks instead like this: 110110...110110.... Contrary to the previous case, if two #PFs are triggered on two consecutive pages in

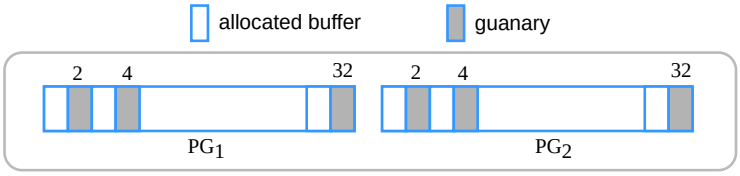


Figure 6.5: Bitmaps of SPP pages for a protection frequency $F = 1$ and classes whose size is $< 128B$. 1 sub-page out of 2 is protected: all pages (PG_i) have the same bitmap in the VMA.

this VMA, the allocator cannot use the same page's configuration to handle them. Otherwise, the VMA will look like this: 110110...**011110**110...011. If we observe the bits in red, we remark that this bitmap no longer suits the protection frequency F defined by the user since there are two consecutive buffers that are not protected (remind that a buffer size requires two sub-pages). A VMA that respects the frequency asked by the user in this scenario would rather look like in Fig. 6.6: here then arises the problem of SPP bitmap heterogeneity inside a VMA.

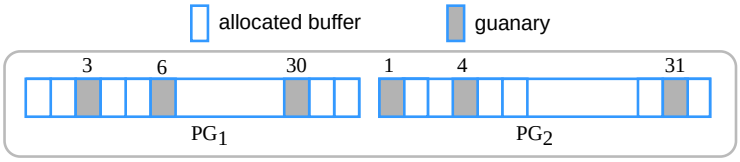


Figure 6.6: Bitmaps of SPP pages for a protection frequency $F = 1$ and classes whose size is $> 128B$ and < 256. 1 sub-page out of 3 is protected: pages (PG_i) have different bitmaps along the VMA.

Why is this considered a technical challenge? For the simple reason that it complexifies both #PF handling and memory reclaim for SPP pages. Indeed, in such cases, the allocator cannot merely apply the same bitmap to all the pages of the VMA, nor can it allocate/configure pages for each bitmap while serving each #PF. Because this will obviously drastically affects performance. Moreover, unlike unused *normal* pages that the kernel's buddy allocator can return to serve a #PF regardless of how they were previously allocated, an SPP page can only serve faulted pages following the same pattern. We address this challenge by implementing an $O(1)$ algorithm (presented in §6.3.5) that allows the kernel allocator to quickly determine for any faulted virtual address what is the corresponding SPP bitmap.

(C_4, Conceptual) Coexistence with the hypervisor's needs. As stated in the introduction, SPP can be used by the hypervisor for accelerating VM management tasks such as live migration and checkpointing. By allowing the guest OS to manipulate SPP, GuaNary may interfere with the hypervisor's

needs. Furthermore, the hypervisor conceptually works at the granularity of the whole VM, while LeanGuard only monitors some pages of the guest. Finally, the hypervisor is the first level called on SPP violation. According to these constraints, the challenge arises upon SPP violation: how does the hypervisor's handler efficiently determine which level is concerned by the fault?

To answer this question, we propose to reuse some reserved bits in the architecture and make a plausible assumption about fault handling.

6.3.2 Overview

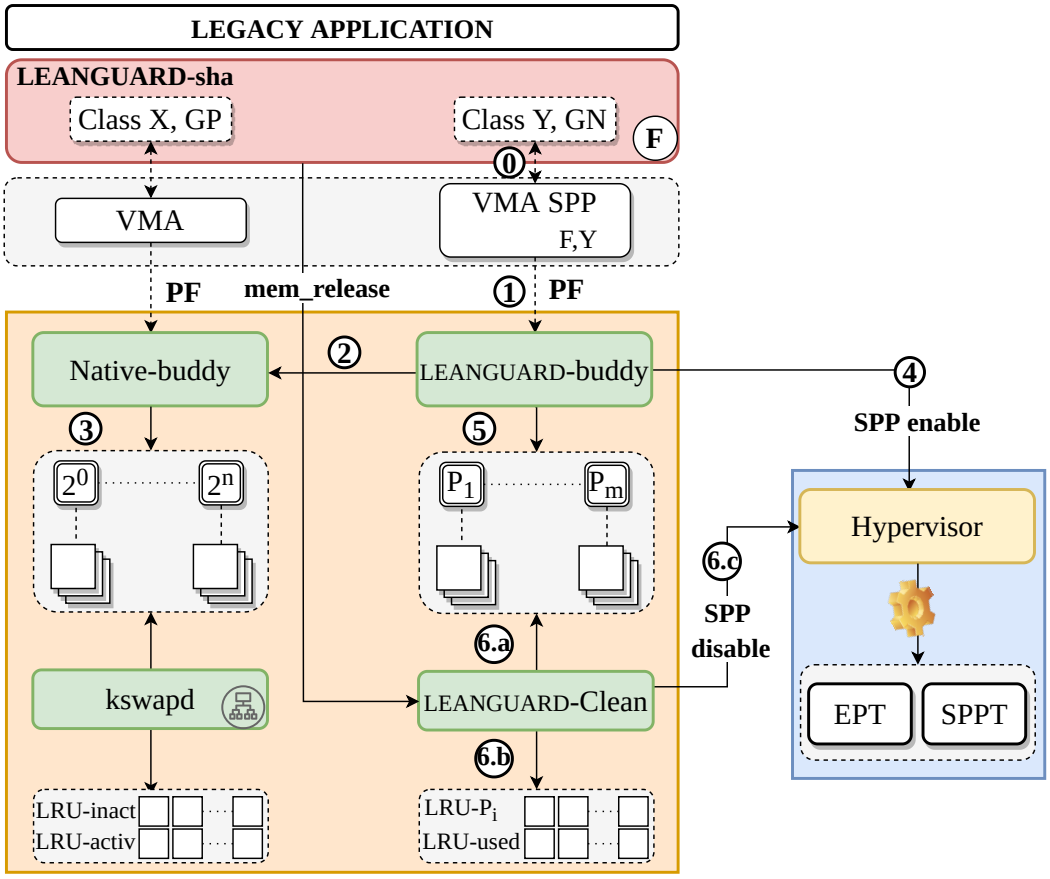


Figure 6.7: Architecture of LeanGuard.

Fig. 6.7 depicts the architecture of LeanGuard. The user links its legacy application (e.g., `LD_PRELOAD`) with our secure heap allocator (noted LeanGuard-sha) that uses GuaNaries and guard pages. At class initialization time ⓪, LeanGuard-sha tags the allocated VMAs as SPP (§ 6.3.3 details LeanGuard-sha). We extend the guest kernel with an SPP-aware buddy allocator (LeanGuard-buddy). On page fault ①, LeanGuard-buddy is in charge of handling ⑤ #PFs which involves SPP-tagged VMAs. To avoid SPP

configuration (due to hypercalls ④) in the critical path, LeanGuard-buddy maintains a pool of pre-configured SPP pages acquired ②-③ from the native kernel allocator (noted Native-buddy). § 6.3.4 details LeanGuard-buddy.

The pages held by LeanGuard-buddy are not inserted into the Native-buddy's active/inactive LRU lists, thus preventing SPP pages from being reclaimed by the traditional kernel's reclaimer sub-component (`kswapd` in Linux). Instead, we build a custom SPP page reclaim ⑥ component called LeanGuard-cleaner (presented in § 6.3.5).

We optimize the SPP configuration algorithm in the hypervisor by reducing the number of EPT and SPP page table walks (see 6.3.6 for details).

6.3.3 Secure Heap Allocator (LeanGuard-sha)

LeanGuard-sha aims to propose a compromise between security and memory consumption. Its allocation policy includes two parameters: `G`, the type of guardian, and `F`, the protection frequency. G is defined by LeanGuard-sha that optimally sets G = *guard page* for classes whose size is a multiple of 4KB (a page size) and G = GuaNary for the others. F is defined by the user and configured at LeanGuard-sha compilation time. Contrary to G, an optimal value of F depends on the application and is empirically obtained by a calibration campaign (see Section 6.5.2). The latter consists of varying `F`, e.g., from 1 to 50, measuring the induced memory consumption and the security distance, and plotting for the application the memory overhead and the security distance as a function of F. The intersection between the memory and the security curves in the latter plot gives the optimal frequency for the given application. For example, in Fig. 6.3, the optimal frequency for PARSEC-blackscholes will be 16. We perform this calibration campaign and provide optimal F for all PARSEC applications in Section 6.5.2.

We introduce two new flags for the `mmap` and `madvice` syscalls that the allocator uses to tag a VMA as SPP. When `malloc` involves a class for the very first time, LeanGuard-sha performs `mmap(SPP,...)` to allocate a large virtual memory for that class, leading to the creation of a VMA (by the kernel). Note that this is the standard functioning of a heap allocator. We merely modify the `mmap` call. The `SPP` flag indicates to the kernel not to merge the newly allocated VMA with other ones that use normal pages. After this, LeanGuard-sha configures the VMA using `madvice` by indicating its frequency F to the kernel. LeanGuard-buddy will further use this information to compute the SPP bitmap that addresses the #PF (see below).

6.3.4 SPP Page Allocator (LeanGuard-buddy)

LeanGuard-buddy maintains a distinct pool P_m of free and pre-configured SPP pages for each SPP bitmap m. As explained earlier in challenge C_3, m is

the bitmap indicating which sub-page is write-protected. Our prototype has a total of 202 pools $(P_{m_i})_{i=\{1..202\}}$. Let us recall that depending on F, pages of a VMA may have different patterns (Fig. 6.6).

We note N_{max}, the maximum size of each pool. We empirically observed that 512 is a good value for N_{max} (see § 6.5.3). Each pool is populated during the very first #PF, and every time its size goes under a pre-defined threshold (see below). The pooling mechanism allows LeanGuard-buddy to directly serve configured SPP pages on the following #PFs, thus, avoiding performing multiple hypercalls that would drastically slow down this critical path.

Upon a #PF in an SPP VMA, the challenge for LeanGuard-buddy is to decide which pool to draw the page from, i.e., which SPP protection pattern m_i should be used to serve that fault. The correct pattern for the faulting virtual page depends on the protection frequency F and the class' size (both defined by LeanGuard-sha at VMA initialization using `madvice` -see §6.3.3-), and the position/offset of the page within its VMA. LeanGuard-buddy determines m_i using a three-steps algorithm. Firstly, LeanGuard-buddy calculates the total number of distinct patterns associated with the VMA using the following formula 6.1:

$$\#patterns\_vma = \frac{least\_common\_multiple(32, F)}{32} \tag{6.1}$$

where 32 is the maximum number of sub-pages in an SPP page. Secondly, LeanGuard-buddy determines the index/position of the faulting virtual page within the VMA. If va_fault is the faulting address and va_start the start address of the VMA, this index is determined as follows:

$$index = \frac{(va\_fault - va\_start)}{4KB} \tag{6.2}$$

with 4KB being the page size. Finally, LeanGuard-buddy can determine the appropriate m_i for va_fault using this formula 6.3:

$$m_i = index \ \% \ \#patterns\_vma \tag{6.3}$$

LeanGuard-buddy can then handle the #PF by returning a free SPP page from pool P_{m_i}.

At the same time, if LeanGuard-buddy notices that the target pool size is below a pre-defined threshold N_{min} (the default value is 100 in our prototype), LeanGuard-buddy fills it back to N_{max} by requesting new pages from Native-buddy.

LeanGuard-buddy maintains a list of LRU pools which it updates at the end of each SPP page allocation request. This list is used by LeanGuard-cleaner when it needs to reclaim SPP pages at the request of `kswapd` (e.g., in case of memory pressure).

6.3.5 SPP Page Release And Reclaim (LeanGuard-cleaner)

LeanGuard-cleaner has two responsibilities: handling SPP page release on process termination and returning SPP pages to Native-buddy on memory pressure.

SPP page release. When a process ends, LeanGuard-cleaner reinserts its SPP pages into the corresponding pools. When the pool size goes up to a configurable threshold N_{limit} (1024 is the default value in our prototype), LeanGuard-cleaner returns the surplus ($N_{limit} - N_{max}$) to Native-buddy. At the same time, for all other pools, it also returns ($N_{cur} - N_{max}$) pages to Native-buddy, where N_{cur} is the number of pages currently held by the pool. Notice that LeanGuard-cleaner disables SPP on released pages before returning them to Native-buddy. This is done by batching hypercalls for optimal performance.

SPP page reclaim. LeanGuard-cleaner does not replace the native kernel's memory reclaim sub-component (e.g., `kswapd` in Linux). If necessary, the latter may instead request more pages from LeanGuard-cleaner on memory pressure. In this case, LeanGuard-cleaner would first return free pages from the SPP pools regarding the pools' LRU list it maintains. If the number of requested pages is still not satisfied, LeanGuard-cleaner reclaims SPP pages in use from LeanGuard-sha (the application level), relying on a second LRU list maintained by LeanGuard-buddy for in-use SPP pages. We consider in our prototype that all SPP pages in use are active. Thus, we leave the implementation of effective active/inactive LRU SPP page lists for future work.

LeanGuard-cleaner must swap out SPP pages in use before returning them to Native-buddy, which does not require special precautions. In contrast, unlike the classical swap-in implementation that needs a single memory copy operation, the swap-in of an SPP page requires copying data from the swap space to the SPP page at the rate of sub-pages, since some portions of the physical page may be write-protected (according to its SPP bitmap). Therefore, the swap-in of an SPP page is much more costly than a normal page, and the kernel's memory reclaimer should prioritize swapping out normal pages when possible.

6.3.6 Hypervisor Extension

LeanGuard extends the hypervisor for two purposes: SPP page configuration and SPP table violation handling and coexistence with the hypervisor.

SPP page configuration optimization. LeanGuard extends the hypervisor for two purposes: SPP page configuration, SPP table violation handling, and coexistence with the hypervisor.

SPP page configuration optimization We extend the hypervisor to optimally configure SPP for a group of contiguous pages. This optimization concerns SPP activation and deactivation. First, we introduce a new hypercall type for each operation. These hypercalls take as input a range of memory pages and their protection bitmaps (for the activation operation). This avoids performing a per-page hypercall. Second, we reduce the number of EPT and SPP table (SPPT) walks performed by the hypervisor during the SPP bitmap configuration. Originally, the hypervisor realizes this operation at the rate of one page. Thus, it performs $(2 \times N)$ EPT+SPPT walks to configure N pages. Given that LeanGuard-buddy requests a contiguous range of pages from Native-buddy, we instead implement a single EPT+SPPT walk per group of pages that share the same EPT leaf, considerably reducing the configuration time (see §6.5.3).

SPP table violation handling and coexistence with the hypervisor We propose a two-phase solution. When the hypervisor wants to use SPP for a VM, it sets a flag in the VM control structure (VMCS). When the VM, via LeanGuard, requests an SPP page configuration, we set the 2nd odd bit of its SPP bitmap (odd bits are currently unused by the hardware, see Section 2.3) to indicate that the VM is interested in monitoring this page. When an SPP violation occurs, there is a VMExit, and the handler provided by the hypervisor takes over. We assume the processor places the SPP bitmap at the origin of the fault in a register or the VMCS. The handler checks the SPP activation flag in the VMCS. If the latter is set, the hypervisor uses the faulted address for its own needs. Before resuming the VM, the handler also checks the 2nd odd bit in the SPP bitmap. If set (meaning that the process running inside the VM is trying to overflow), the hypervisor injects the fault into the guest, and a handler in the guest OS kills the faulted process. Other more elaborate actions can be considered.

6.4 Implementation

We implemented our prototype in Xen 4.10 (the hypervisor, 660LOC addition), Linux 5.11.14 (the guest OS, 750LOC addition), and SlimGuard (the baseline secure heap allocator, 100LOC addition). We use the latter to provide LeanGuard-sha for four main reasons: (1) It is the up-to-date secure heap allocator, published in 2019. (2) Its authors experimentally compared it to other state-of-the-art secure allocators, such as Guarder and FreeGuard,

and have proven more efficient (in terms of memory consumption and security guarantees) than those allocators. (3) It implements both guard pages and canaries. (4) Last, but not least, its code is available and functional.

To provide LeanGuard-sha, we extend SlimGuard in two ways. First, we changed the interpretation of the protection frequency that guides guard page placement within classes (VMAs). In its original version, SlimGuard's protection frequency answers this question: *After how many allocatable pages should we place a guard page?* This can also be interpreted as the proportion P of guard pages in the VMA. We updated this policy to answer the following question: *After how many buffer allocation requests F, within the class, should we place a (sub)guard page?* The latter question is more comprehensive to the user because it tells the proportion of allocated buffers that will precisely reside at the boundary of a guardian (i.e., security distance equals 0B). In fact, with the first question, the user is supposed to be VMA-friendly, which is not always true.

Fig. 6.8 displays the CDF of the security distances for the two policies while varying P and F. Fig. 6.8 right confirms that our policy efficiently expresses the proportion of protected buffers ($1/F$). For illustration, with F=2 at least 50% of allocated buffers have a security distance of 0B. On the other hand, from Fig. 6.8 left, we can see that for P=50% no buffer has a security distance of 0B, demonstrating the weak expressiveness of this parameter from the user perspective. In the remaining, we only apply our custom policy.

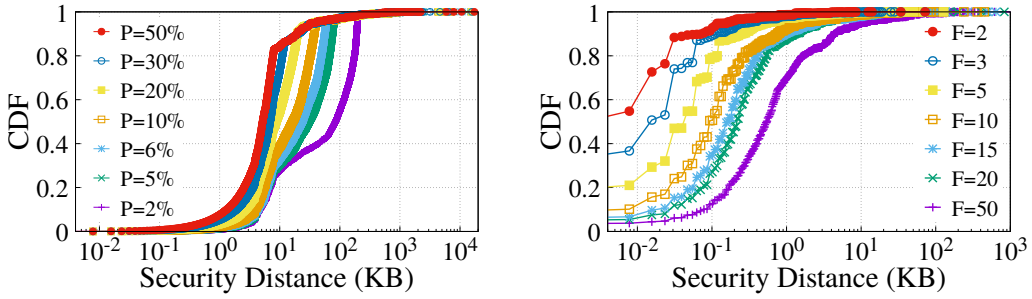


Figure 6.8: CDF of the security distances when the placement policy of guard pages is related to the number of allocatable pages (left) or `malloc()` calls (right).

6.5 Evaluations

Our experiments hinge on three main axes: Memory consumption and Security (6.5.2) and Performance (including scalability) (6.5.3). We only compare LeanGuard with SlimGuard because it has already proven more efficient than recent state-of-the-art secure allocators. All reported results in this section are a mean of 5 runs.

6.5.1 Experimental Environment

Setup For our experiments, we use virtual machines (VM) running Ubuntu 18.04 LTS under an Intel 8-Core i7-8565U CPU at 1.80GHz with 15GB RAM. Otherwise indicated, each VM is configured with 8GB of memory and four vCPUs pinned to four dedicated cores to avoid interference.

Benchmarks We use both micro- and macro-benchmarks. The former consists of two applications: one with a 1GB working set size that randomly allocates buffers from all SlimGuard classes and the other that deliberately generates overflows. In addition to this micro-bench, we write a simple program that addresses the entire VM memory at boot time to force EPT construction so that this cost does not affect performance evaluation results. Macro-benchmarks are composed of PARSEC [69] applications. Because of build or compilation errors with some applications (due to either the GCC or the kernel version), we use 9 out of 13 applications from the PARSEC suite: `blackscholes`, `bodytrack`, `dedup`, `fluidanimate`, `freqmine`, `raytrace`, `streamcluster`, `swaptions`, and `x264`. These applications are the same ones used by SlimGuard's authors.

6.5.2 Memory Consumption

Methodology Intel initially announced, in 2019, the release of SPP with Ice-lake Xeon processors. Due to the COVID'19 pandemic, Intel delayed the release to the second semester of 2020 [49, 33]. Yet, till today and to the best of our knowledge, there is still no SPP-capable processor. Therefore, we use two emulation environments to test and evaluate our prototype. The first one is BOCHS, which version 2.6.11 emulates Intel SPP. We implement the second one inside the hypervisor by leveraging the Intel CPU's single-step feature present on existing processors. This second emulation environment aims to make the hypervisor mimic the functioning of the processor when SPP is supported.

On our non-SPP-capable machine, if the processor tries executing a write-instruction upon a #PF for an SPP page, because the bit `write` is unset (see Section 2.3), it will trigger a VMEXIT even if the bit `SPP` is set since it does not know how to interpret the latter. It is then up to the hypervisor to handle the exit and execute our algorithm, which is as follows. The hypervisor walks the EPT to check if it is an SPP fault (i.e., bit `SPP` in the EPT leaf is set) and calls the `SPP violation` handler if applicable. This handler further walks the SPPT to check wether the write-access is granted (using the bitmap in the SPPT leaf) for the sub-page targetted by the #PF. If the sub-page is not write-accessible, the hypervisor injects a general protection exception into the guest: this is interpreted as a buffer overflow by the guest kernel

that simply terminates the faulted process. Else, if the sub-page is write-accessible, the hypervisor performs four actions: (1) sets the `write` flag of the EPT entry to authorize the faulted instruction, (2) sets the Trap Flag (TF) in the `EFLAGS` register to put the CPU in single-step mode, (3) flushes the TLB, (4) and resumes the VM. When the VM resumes, the processor will successfully execute the faulted memory store instruction (since the hypervisor has reset the `write` flag). Because the processor is in single-stepping mode, it will immediately trap into the guest kernel at the end of the instruction. The handler of that fault (in the guest kernel) will perform a hypercall instructing the hypervisor to write-protect back the page (so that next accesses to this SPP page can be trapped) and disable single-stepping.

As one might guess, these emulation environments will negatively affect performance. They are therefore used only for LeanGuard assessment and memory consumption evaluation. §6.5.3 presents the methodology for performance overhead evaluation.

To validate the effectiveness of hypervisor emulation, we write a micro-benchmark that intentionally performs a buffer overflow and run it under LeanGuard with a protection frequency F=1 so that all allocated buffers are placed at the boundary of a sub-page. The simplified code of this microbench, shown in Listing 6.1, only allocates buffers with sizes that are not multiple of 4KB to impose the use of GuaNary exclusively. After some iterations, the application tries overflowing the last allocated buffer by one byte, and for each run, this resulted in a `segfault` that immediately terminated the application, validating the effectiveness of our emulation.

```
1  ...
2  #define ITER  //number of allocations before overflowing
3
4  unsigned int slimguard_classes[n];
5  void init_slim_cls(void)
6  {
7    //fill in the slimguard_classes array with all the Slimguard classes that are not 4KB-multiple
8    ...
9  }
10
11 unsigned int _random_func(void) {...}
12
13 void main(void)
14 {
15   unsigned long *tmp, size, length;
16   unsigned short random;
17   int flag = 0;
18
19   init_slim_cls();
20   /*while we have not allocated ITER buffers*/
21   do
22   {
23     /*pick a random class*/
24     random = _random_func() % n;
25     /*retrieve the size of the class*/
26     size = slim_cls[random];
27     tmp = malloc(size);
```

```
28     if(tmp)
29     *tmp = size; //force physical allocation
30   }while( ++flag <= ITER );
31   /*try overflowing*/
32   length = size / sizeof(unsigned long);
33   *(tmp + length) = size; //try 1-byte overflow
34 }
```

Listing 6.1: Simplified code of the microbenchmark used to validate SPP emulation in the hypervisor.

Memory Evaluation Results. The main goal of LeanGuard is minimizing memory overhead while allowing synchronous buffer overflow detection, and validating this goal is the purpose of this section. To this end, we compare LeanGuard with different configurations of SlimGuard: SlimGuard with only canaries (SlimG+Canary), SlimGuard with only guard pages (SlimG+GP), and SlimGuard with only GuaNary (SlimG+GuaNary). SlimG+Canary is theoretically the lower bound regarding memory overhead, but it does not allow synchronous detection. So, from the perspective of memory overhead, the closer to SlimG+Canary, the better.

We run the PARSEC applications under each configuration while varying the protection frequency F. As we can see in Fig. 6.9 presenting the results, (1) SlimG+Canary is indeed the lower bound in terms of memory overhead, and (2) LeanGuard leads almost to the same memory consumption as SlimG+Canary for these specific applications. The highest gap we measured between these two configurations is only 23%, observed on application `raytrace` when F=2. (3) Our new safety guard GuaNary effectively leads to memory consumption reduction, as SlimG+GuaNary results suggest. The slight improvement that LeanGuard brings to SlimG+GuaNary (26% `swaptions`, 25% on `raytrace`, and 18% `blackscholes`, when F=2) is explained by the fact that, unlike SlimG+GuaNary that exclusively uses GuaNary, LeanGuard can use 4KB guard pages when necessary. This validates the necessity to combine the two guardian types. (4) We can also observe that to protect a given proportion of buffers (i.e., synchronously detecting an overflow on these buffers), SlimG+GP incurs a significant memory overhead compared to LeanGuard. For example, to protect 50% of the allocated buffers (F=2), LeanGuard, on average, uses 60% less memory than SlimG+GP. Using the same amount of memory as SlimG+GP, LeanGuard allows protecting 25× more buffers than SlimG+GP.

For the remaining experiments, we only compare LeanGuard with SlimG+GP as it is the default configuration of SlimGuard for synchronous overflow detection.

We also notice from Fig. 6.9 that the optimal (memory consumption and security trade-off) protection frequency for LeanGuard is $F = 2$. However, to fairly compare the two systems, we must apply the same values of F to

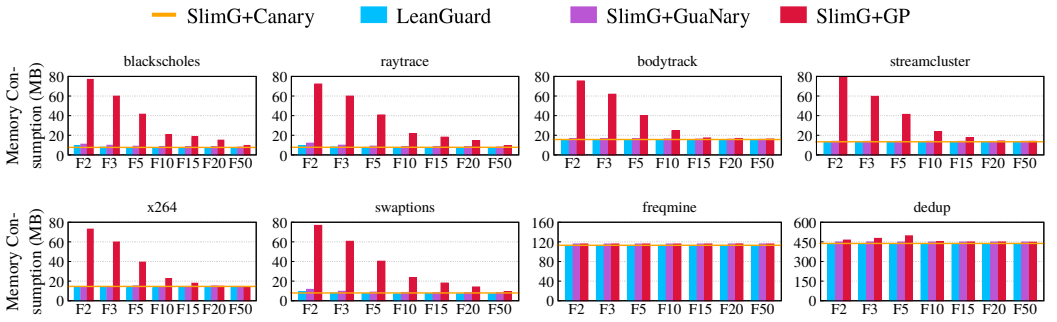


Figure 6.9: Memory consumption of each allocator configuration for PARSEC applications while varying the protection frequency.

each. This is why we draw, in Fig. 6.10, the trade-off curves of SLIMG+GP for PARSEC applications. We can see that the optimal F is 2 for `dedup` and `freqmine` (since their memory overhead remains constant while varying F), and 15 on average for the others.

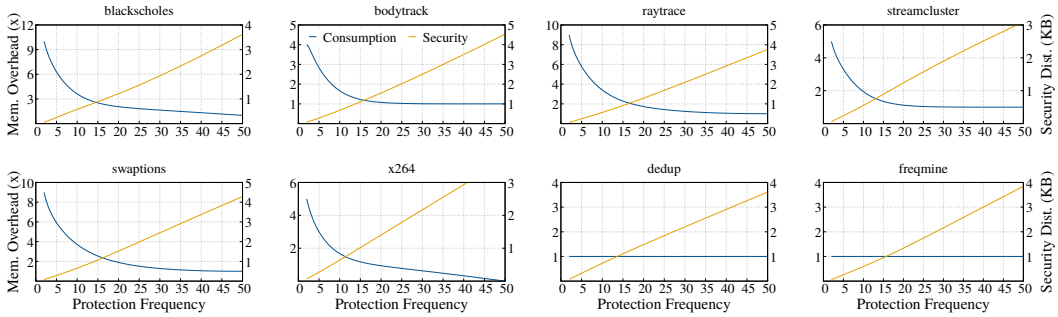


Figure 6.10: Trade-off between memory consumption and security for PARSEC applications under `Slim+GP`.

6.5.3 Performance Overhead

This section evaluates the overhead of LEANGUARD. We first describe the evaluation methodology we used, as there is not an SPP-capable machine on the market.

Methodology. We carefully build a formula (equation 6.4) to accurately estimate the execution time of an application *App* when it runs under LEANGUARD. The idea behind our methodology is as follows. We can run *App* under LEANGUARD atop a machine that is not equipped with SPP (noted $Machine_{noSPP}$). We need, for this, to disable the hypervisor's portion of code that sets the SPP control bits. This execution configuration captures almost all LEANGUARD overhead, including cache pressure. The only overhead this configuration will miss is the one generated by the SPP table (SPPT) walks

on TLB misses. Remember that for any write-protected page for which the SPP flag is set, the processor will walk the SPPT in addition to the traditional 2-dimensional guest page table and EPT walking. We note T_{noSPP} the execution time of *App* on $Machine_{noSPP}$. The overhead of SPP can be obtained by multiplying the number of SPP-based TLB misses (noted TLB_{spp}) by the time to walk the SPPT. The latter can, in turn, be obtained by multiplying the memory latency (noted $T_{mem\_latency}$) by 4 (because the SPPT is a 4-level radix tree). In summary, we can estimate the execution time of *App* when it runs under LeanGuard atop a hypothetical SPP-capable machine using this formula:

$$T_{est} = T_{noSPP} + TLB_{spp} \times 4 \times T_{mem\_latency} \tag{6.4}$$

According to [88], we empirically estimate $T_{mem\_latency}$, which is 111 ns for our testbed.

To estimate TLB_{spp}, we use the following method. We run *App*, and we collect two metrics: the proportion of SPP pages in the *App*'s working set (noted k and collected by instrumenting LeanGuard-buddy and Native-buddy); and the total number of Data TLB misses (noted TLB_{tot}, using the Linux `perf` tool). Finally, we compute TLB_{spp} using the following formula 6.5:

$$TLB_{spp} = k \times TLB_{tot} \tag{6.5}$$

Low level metrics. Low-level metrics provide a better understanding of higher-level metrics and include the functions and events that LeanGuard and SlimGuard integrate: `malloc()`, `mmap()`, `madvise()`, `mprotect()`, and hypercalls to configure SPP. We are interested in their execution time. Among these metrics, only `malloc()` will be impacted by the protection frequency. Except for `mprotect()` (with SlimGuard), the others are involved only once during class initialization, corresponding to the first `malloc()` call. We use the micro-benchmark for these experiments. Table 6.1 and Fig. 6.11 summarize the results.

	`mmap`	`madvise`	`mprotect`	enable SPP	disable SPP
SlimGuard	1291	-	2349	-	-
LeanGuard	1332	727	-	7872	4194

Table 6.1: Cost, in nanoseconds, of some basic metrics.

From Table 6.1, we can notice that the introduction of our new SPP flag adds only $41ns$ to the execution time of the `mmap` syscall. The syscall `madvise` to configure a VMA as SPP takes $727ns$. Even if SPP activation and deactivation hypercalls are more costly ($7.8ns$ and $4.1ns$, respectively) than

`mprotect` ($2.3ns$), unlike the latter, they are used only once by LeanGuard. Indeed, both allocators will progressively `mprotect` pages in the VMA as more buffers are allocated during the application's execution. And because SlimGuard uses only guard pages, it will likely perform more calls to `mprotect` than LeanGuard. This will compensate the cost of other metrics involved in LeanGuard and explains that LeanGuard and SlimGuard take about the same amount of time to perform `malloc()` (as we can see in Fig. 6.11): about $1346\mu s - 1448\mu s$ for the first call and $0.57\mu s$ for the following ones.

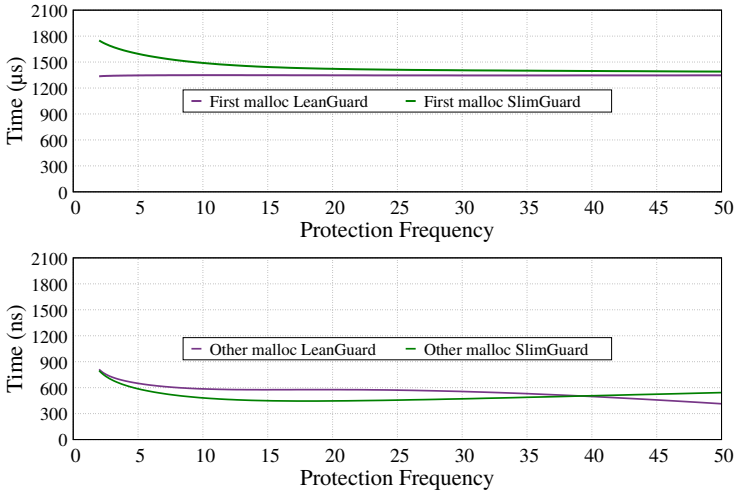


Figure 6.11: Average execution time of `malloc()`. On top is the time of the *first* call to `malloc`. On the bottom, the following calls after the class was initialized.

SPP Table Walk Optimization. This section evaluates to what extent our optimizations (§6.3.6) reduce the time to execute the SPP activation hypercalls for a group of pages. Recall that these hypercalls are performed only during some #PFs when the lower page threshold is reached for a pool (§6.3.4). While they may certainly slow some #PFs, they will allow for avoiding frequent slowdowns during this critical path, as we can see in Table 6.2. From the latter, we can see that worse cases are less than 3% of total #PFs for pools of 512 pages. When the pool has only one page (column two in Table 6.2), i.e., all #PFs should trigger a configuration hypercall, we might expect the number of worse cases to be the same as the total number of #PFs. However, the slight difference observed is explained by the fact that some pages can be reinserted in their corresponding pools by LeanGuard-cleaner (see §6.3.5).

While varying the size of pools from 1 to 512 pages, we measure, for the microbench, the time taken by LeanGuard-buddy to service the #PF in both the best and worse cases. Boxplots in Fig. 6.12 assess that when pools are filled, LeanGuard-buddy similarly handles PF regardless of the

Pool size order (n)	0	1	3	5	7	9
Worse case #PFs	7835	8312	1266	701	427	358
Total #PFs	7995	8472	9093	12979	12749	12626

Table 6.2: Number of #PFs triggering a configuration hypercall compared to the total number of #PFs. The size of pool is 2^n.

optimizations, with a time between $0.7\mu s$ (25th percentile) and $2.2\mu s$ (95th percentile), with a mean of $0.9\mu s$.

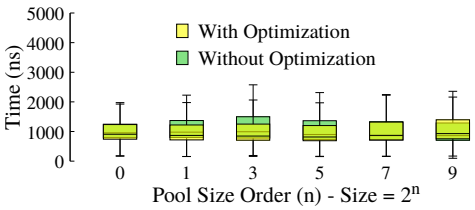


Figure 6.12: Distribution of best-case-#PF handling time. This corresponds to normal #PF handling when the buddy allocator does not have any hypercall to perform before servicing the #PF.

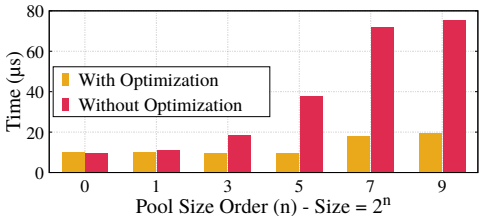


Figure 6.13: Avg time of worse-case-#PF handling with and without our SPP page table walk optimization.

Fig. 6.13 presents the average execution time of the #PF in the worse case, i.e. when configuration hypercalls are performed. As we can see, our optimizations efficiently improve the #PF time, over non-optimized hypercalls, by up to 3.8× for a pool size of 512 (2^9). Notice that the latter is the maximum array size that the hypervisor allows in a hypercall parameter (in our testbed). Furthermore, these optimizations ensure that the page fault time no longer grows exponentially as it does without optimization. From 1 to 512 pages, we have only 1.9× slowdown with optimizations versus 7.8× without.

Based on this observation, we realize all the following experiments using pools of 512 SPP pages.

Performance Overhead and Scalability. To evaluate the performance overhead of GuaNary, we run PARSEC applications under $Machine_{noSPP}$ (see methodology in §6.5.3) with SlimGuard and LeanGuard. Execution times are physically measured for SlimGuard and estimated for LeanGuard using Formulas 6.4 and 6.5. Fig. 6.14 plots the results. We observe that the drop between SlimGuard and LeanGuard is essentially the SPPT walk (dark blue portion on LeanGuard bars). The execution time is almost the

same for most applications, and when it is not the case, LeanGuard incurs only $\sim 7.7\%$ overhead on average. This overhead can be acceptable to a user whose priority is memory security. Remember that the main problem here is the dilemma between security and memory overhead. Therefore, given the significant trade-off that GuaNary provides (§6.5.2), this overhead counterparty can be tolerable.

We also remark that the time of hypercalls is almost not visible, and this is because hypercall cost is negligible compared to the execution time of applications. Indeed, hypercalls cost less than $3ms$ for all applications, which is one to five orders of magnitude lower than execution times.

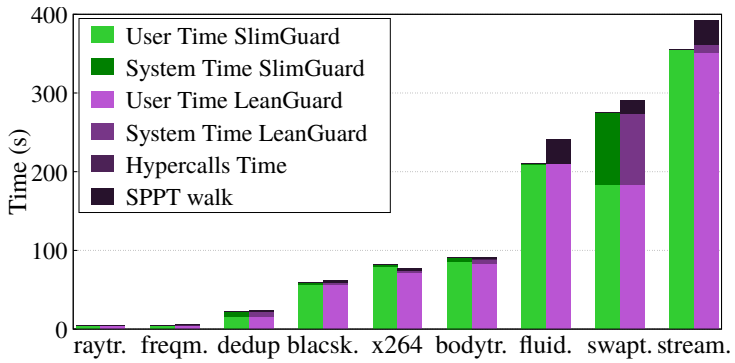


Figure 6.14: Performance overhead of SlimGuard and LeanGuard on PARSEC application.

We also evaluate how LeanGuard-buddy scales when many applications run simultaneously. Fig. 6.15 plots the mean #PF handling time when we vary the number of concurrent applications from 2 to 9 (the number of PARSEC applications used in our experiments). We observe that this time increases with the number of concurrent applications, which is explained by the fact LeanGuard-buddy uses a lock mechanism on pools. However, this is not an exponential increase which we claim can be accepted for two main reasons. First, #PF handling is not the main source of overhead, as we saw in Fig. 6.14. Second, LeanGuard-buddy can further be optimized by implementing a distributed version, each instance managing a distinct set of physical pages. We leave this optimization for future work.

6.6 Related work

This section completes the related work that we started in § 6.2.2.

Secure Allocators. Several research works have studied secure heap allocators to fight against memory vulnerabilities. Some of them focus on a single

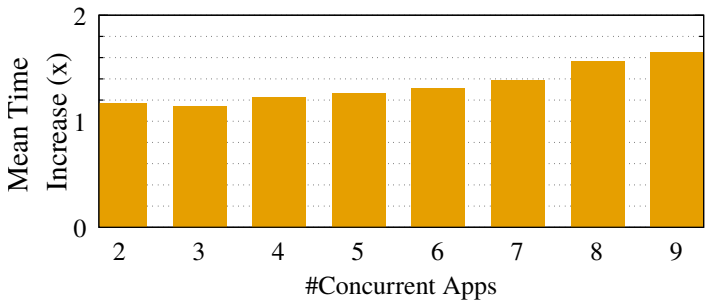


Figure 6.15: Overhead on page fault handling when the number of concurrent applications varies.

vulnerability like ffmalloc [157] and MarkUs [56], which try to prevent use-after-free. The remaining secure allocators are more generic. This is the case for DieHarder [134], FreeGuard [144], Guarder [145], hardened_malloc [11], isoalloc [43], mimalloc-secure [45], Scudo [24], and SlimGuard [118]. Yun et al. [161] did an exhaustive study of the design space of existing secure allocators. The authors compared qualitatively and quantitatively almost all existing secure allocators from the perspective of the security features they implement. The authors recognize that by being specialized, ffmalloc [157] and MarkUs [56] meet their security goals, which is not the case for other generic allocators. This reinforces our choice of focusing on a specific class of vulnerabilities based on buffer write overflow.

Yun et al. [161] introduced HardsHeap [161], a flexible tool to automatically evaluate secure allocators. In the present chapter, we reduce memory overhead in secure allocators without significantly impacting performance. Although performance and memory overhead are two important constraints that secure allocators should meet to favor their adoption, they have not been studied by Yun et al. [161]. We could have extended HardsHeap to use it in our experiments. We leave this for future work.

Very few works tried to reduce the memory overhead of page guards while preserving synchronous overflow detection. There have been works to reduce the memory overhead of metadata and free list management. Some allocators, such as DieHarder [134], rely on a bitmap, but it incurs a performance overhead, especially when the allocator does randomization, which is the case for several secure allocators.

Software Protection against Spatial Safety Violations. Many past works have attempted to enhance safety guarantees in C/C++ by proposing software solutions to protect pointers and/or buffers against spatial safety violations. Some early works [60, 97, 127] suffered from high-performance overheads and a low degree of compatibility, as they generally required code changes. The issue

of backward compatibility was the focus of other early works [99, 141]. Performance impact and compatibility are still the main concerns of recent works. SoftBound [126] relies on code transformations to encode disjoint metadata, including bounds information for every pointer. Memory access operations corresponding to pointer dereferencing can then be checked against this metadata. Other solutions, such as Baggy Bounds [58], Low-Fat Pointers [108], or Delta Pointers [107], rely on pointer tagging, i.e., they leverage unused bits in the pointer itself to encode bounds or pointer validity metadata. Finally, systems like Valgrind [128] or Address Sanitizer [142] use compiler and runtime memory allocator instrumentation to protect memory objects with so-called red zones (guard pages and canaries). They have a heavy impact on performance and memory consumption; hence, they are used only for debug/test and not in production.

Hardware Protection against Spatial Safety Violations CHERI rejuvenated the concept of *capabilities* invented in the 60s [87]. Under a CHERI-capable machine, buffer overflows are synchronously detected by the hardware, as with GuaNary. The main source of CHERI's memory waste is that it doubles pointer size to keep permissions and buffer boundaries directly into the pointer. Therefore, the memory overhead of CHERI (and, in general, capability-based solutions) is related to the number of pointers manipulated by the application. With GuaNary, the memory overhead is related to the number of allocated buffers, leading to a more predictable memory overhead.

CHERI has two important limitations compared to GuaNary. First, CHERI requires application rewrite (see §4 of [156]), which limits its utilization for legacy applications. Second, CHERI significantly degrades performance, especially for heavy pointer manipulation applications (e.g., those from Olden benchmark suite [53]) due to cache miss increase, TLB miss increase, DRAM traffic increase, execution path increase (e.g., the compiler needs to wrap several functions such as `memcpy`), etc. For illustration, Joannou et al. [98] measured a cache miss overhead of up to 250% with respect to MIPS. The need to track pointers propagation to maintain accurate point-to relationships between pointers and buffers would also introduce severe performance degradation. In fact, pointer propagation is very costly (e.g., 80% in DANGNULL [111]). GuaNary does not include all these limitations: it is transparent for applications and does not need to track pointer propagation. GuaNary is a guardian placed at the buffer border. Therefore, it catches any use of a pointer that tries to overflow the buffer. Its efficiency is not linked with pointers.

MPX [135] is the hardware solution from Intel, similar in that it provides support for checking pointer references. It is now deprecated [135, 52] for several reasons [136], including those listed above. In addition, it does not

support multithreading. GuaNary does not include these limitations.

6.7 Summary

This paper focused on write buffer overflow vulnerabilities in virtualized environments. We were interested in the problem of memory overhead generated by existing safety guards, namely, canaries and guard pages. We experimentally showed the limitation of these guardians using SlimGuard, an up-to-date state-of-the-art secure allocator. To address this problem, we presented two contributions: GuaNary, a new guardian type, and LeanGuard, a software stack that integrates GuaNary in secure allocators (including SlimGuard) relying on the safety guards mechanism. GuaNary repurposes Intel SPP, a hardware virtualization feature initially introduced to accelerate VM management tasks such as checkpointing. We thoroughly evaluated LeanGuard using micro- and macro-benchmarks. The results showed that LeanGuard effectively reduces memory overhead by up to 8.3× compared to guard pages. The results also showed that LeanGuard incurs negligible overhead and maintains the security level enforced by the secure allocator.

Chapter 7

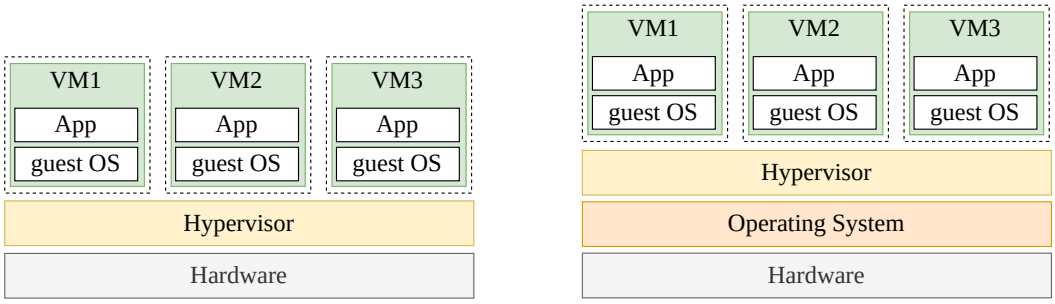
Discussion

This section presents how we can apply the OoH implementations exposed in this dissertation to other types of hypervisors.

Contents

7.1 Hypervisor Typology

Depending on the privileged level at which the hypervisor runs, it can be of different types: Type 1 or Type 2.



(a) Type-1 hypervisor (b) Type-2 hypervisor

Figure 7.1: General architecture of Type-1 and Type-2 hypervisors.

Figure 7.1 illustrates the architecture's layers stack for each type of hypervisor. A *Type-1* hypervisor (Figure 7.1a) runs on bare-metal, that is, immediately atop the physical hardware. Therefore, it can directly manage the physical resources (CPU, memory, I/O devices) for virtualization. Some salient type-1 hypervisors are Xen [61], Microsoft Hyper-V [13], and VMWare ESXi [28]. Contrary to *type-1*, *type-2* hypervisors do not run on bare-metal and are instead embedded with the host OS (e.g., as a kernel module). This is the origin of the two main drawbacks of this hypervisor category. First, to obtain physical resources for virtualization tasks, a type-2 hypervisor must pass through the extra layer of the host OS, thus increasing the latency compared to

type-1 hypervisors. Second, any failure that affects the base OS can also affect the hypervisor and, therefore, the guest OS and the virtual machine, making type-2 hypervisors less secure than their counterpart. Some prominent type-2 hypervisors are KVM [15], Virtualbox [22], or VMWare Workstation [29].

7.2 Type-1 Hypervisors

As part of this work, we have principally used the Xen hypervisor, which is the only type-1 hypervisor available as an open source [31]. It is therefore used as the basis of many other commercial server virtualization products, such as Citrix Hypervisor [4] or Oracle VM for X86 [21]. However, all the OoH implementations can similarly be applied to other type-1 hypervisors as they share a common base architecture. Indeed, as we can see in Figure 7.2 and Figure 7.3, VMWare and Hyper-v both have a privileged component comparable to the Xen's dom0: *primary/parent partition* for Hyper-v and *service console* for VMWare. These privileged components provide (along with the hypervisor) the same services as dom0, which are VM management tasks (creation, destruction, etc.), I/O drivers management, and tools management (storage, network, etc.). Therefore, OoH will simply need modifications at the same level as with the Xen architecture, that is, in the hypervisor and the guest OS.

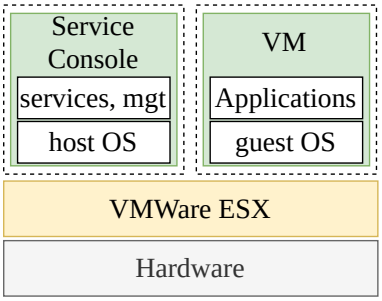


Figure 7.2: VMWare ESX architecture.

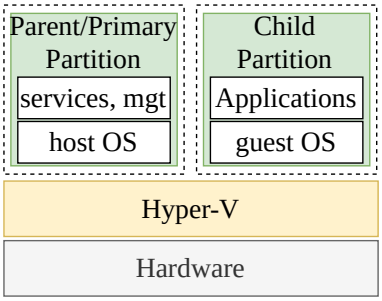


Figure 7.3: Hyper-v architecture.

7.3 Type-2 Hypervisors

Type-2 hypervisors can be part of the host kernel, like with QEMU-KVM in Figure 7.4. This does not matter, as the OoH modifications only concern the hypervisor itself, not the host kernel.

Let us consider, for example, implementing OoH for PML (Chapter 5) with KVM. OoH for PML required adding two hypercalls to Xen for PML activation and deactivation by the guest. Since the integration of PML to

KVM did not need modifying the Linux kernel because only the hypervisor manages the EPT and the PML mechanism, adding these new hypercalls will also only modify KVM and not the kernel. As for Xen, the handler code of the hypercalls will be added to KVM.

OoH for PML also required establishing a shared memory between the hypervisor and the guests, in the form of ring buffers, to copy addresses (GPA with SPML and GVA with EPML) from the hypervisor or the guest kernel to the guest userspace. With KVM, this can be done using `virtio`, which provides guest-to-hypervisor communication. Because the `virtio` API in the host is managed by QEMU, in addition to modifying KVM, the OoH implementation may also necessitate modifying QEMU. In any case, QEMU-KVM could be modified following the same logic as for Xen without involving the host kernel.

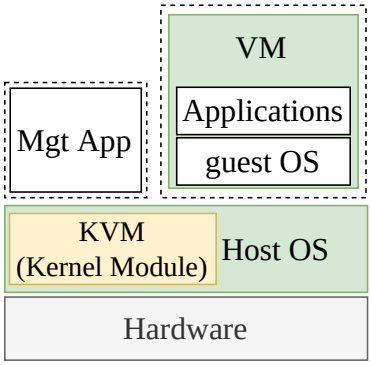


Figure 7.4: KVM hypervisor architecture.

Chapter 8

Conclusion et Future Work

Contents

8.1 Conclusion

This thesis has introduced Out of Hypervisor (OoH), an alternative to nested virtualization. OoH is a novel research axis that argues for the exposure of current hypervisor-oriented hardware virtualization features to the guest OS so that its processes could also benefit from those features. We have shown that the OoH principle can be applied to both privileged and unprivileged domains in virtualized clouds.

This dissertation has illustrated the practicality of OoH with Intel PML (allowing guest memory tracking by the processor) and Intel SPP (allowing guest memory protection at a granularity of 128 bytes) features, applied respectively to dirty page tracking and memory protection in guest userspace. For privileged domains, we proposed PRL (Page Reference Logging), an extension of PML for efficient working set size estimation of VMs. We also showed how PRL could be used by presenting a working set estimation system that leverages it.

For unprivileged domains, contributions are twice. First, we proposed OoH for PML, which provides an efficient solution for dirty page tracking in guest userspace. We presented two implementations of OoH for PML. A software-only solution called Shadow PML (SPML), that requires no hardware changes but incurs significant overheads. And Extended PML (EPML), that addresses the drawbacks of SPML by making some modest changes to the hardware. We proved the efficiency of EPML by introducing it into two well-regarded systems that are CRIU, the Linux Checkpoint/Restore In Userspace tool and the Boehm Garbage Collector.

Second, we proposed OoH for SPP, which allows for improving buffer overflow mitigation in guest userspace. We introduced GuaNary, a novel de-

fense against overflows that provides synchronous detection at a low memory footprint cost compared to the state-of-the-art. We further presented LeanGuard, a complete software toolstack that demonstrates the usability and integrability of GuaNary with existing secure memory allocators. For this purpose, we took as a use case SlimGuard, a recent (2019) secure allocator that has already proven to be more efficient than recent state-of-the-art secure allocators.

8.2 Furture Work

8.2.1 Improvement of Current Contributions

Because any work is always perfectible, it would be worth bringing some amelioration to the current implementation of some contributions presented in this thesis.

The current integration of LeanGuard is done by modifying the guest kernel, which can limit its adoption because users are often reluctant to make changes to legacy applications and, moreover, when it concerns the operating system. Furthermore, in-core kernel modifications are generally not suitable for fast updates as they require recompilation and reinstallation. For this reason, it would be more convenient to release the kernel part of LeanGuard as a kernel module, thus following OoH principle and methodology. We first proceeded this way because we wanted to show and prove the efficiency of GuaNary, and porting the code to a kernel module is simply an engineering matter and will not require additional research work.

8.2.2 Future Directions

OoH is a research axis and, therefore, opens the way to further directions.

OoH in Bare-metal. The first orientation that could be derived from this thesis is applying the OoH principle to non-virtualized environments. Indeed, as stated in Section 3.3, works like Dune can take advantage of OoH to make host userspace applications take advantage of hardware virtualization functionalities without the need for a hypervisor. One of our ongoing works is then to revisit and extend Dune to make it OoH-compatible if possible or propose another system from scratch.

OoH for Other Processors and Architectures. Akin to any principle, OoH is generic and can be applied to processors other than Intel. Although this thesis has focused on the latter, similar virtualization extensions can be found for AMD, for example, and can be exploited following the same methodology.

Beyond x86-64, OoH can also find applicability to AArch64 architectures. Indeed, NEVE, the ARM's virtualization extension, which is similar to Intel VMCS shadowing, can just as well be exploited to leverage ARM virtualization extensions when implemented.

Bibliography

[1] 4th generation intel core vpro processors with intel vmcs shadowing. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-vmcs-shadowing-paper.pdf. Accessed: 2021-11-21. (Cited on pages 46 and 68.)

[2] Boehm garbage collector: C++ interface details. https://hboehm.info/gc/gcinterface.html. Accessed: 2022-03-01. (Cited on page 63.)

[3] Chunk overlap attack. https://ctf-wiki.mahaloz.re/pwn/linux/glibc-heap/chunk_extend_overlapping/. Accessed: 2022-05-29. (Cited on page 75.)

[4] Citrix hypervisor. https://www.citrix.com/en-gb/products/citrix-hypervisor/. Last Checked: 04-11-2022. (Cited on page 100.)

[5] Criu. https://www.criu.org/Main_Page. Accessed: 2021-11-21. (Cited on page 4.)

[6] Docker. https://www.docker.com/. Accessed: 2021-11-21. (Cited on pages 2 and 54.)

[7] Fast bin attack. https://guyinatuxedo.github.io/28-fastbin_attack/explanation_fastbinAttack/index.html. Accessed: 2022-05-29. (Cited on page 75.)

[8] Federated function as a service. https://funcx.org/. Accessed: 2022-05-11. (Cited on page 51.)

[9] A garbage collector for c and c++. https://www.hboehm.info/gc/. Accessed: 2021-11-21. (Cited on pages 4, 46 and 54.)

[10] Gc benchmark. https://www.hboehm.info/gc/gc_bench/. Accessed: 2021-11-21. (Cited on page 55.)

[11] Github : Security vulnerabilities. https://github.com/GrapheneOS/hardened_malloc. (Cited on page 95.)

[12] Gnu java compiler. https://gcc.gnu.org/wiki/GCJ. Accessed: 2022-02-01. (Cited on page 54.)

[13] Hyper-v technology overview. https://learn.microsoft.com/en-us/windows-server/virtualization/hyper-v/hyper-v-technology-overview. Last Checked: 04-11-2022. (Cited on pages 25 and 99.)

[14] Inkscape. https://inkscape.org/fr/. Accessed: 2022-02-01. (Cited on page 54.)

[15] Kernel virtual machine. https://www.linux-kvm.org/page/Main_Page. Last Checked: 04-11-2022. (Cited on pages 25 and 100.)

[16] Kubernetes. https://kubernetes.io/. Accessed: 2021-11-21. (Cited on page 2.)

[17] Mozilla project. http://www.mozilla.org/. Accessed: 2022-02-01. (Cited on page 54.)

[18] Nested virtualization in azure. https://azure.microsoft.com/en-us/blog/nested-virtualization-in-azure/. Accessed: 2021-11-21. (Cited on page 2.)

[19] The openbsd project. https://www.openbsd.org/. Accessed: 2022-04-24. (Cited on page 75.)

[20] Openvz. https://openvz.org/. Accessed: 2021-11-21. (Cited on page 54.)

[21] Oracle vm server for x86 virtualization and management. https://www.oracle.com/uk/a/ocom/docs/ovm-server-for-x86-459312.pdf. Last Checked: 04-11-2022. (Cited on page 100.)

[22] Oracle vm virtualbox. https://www.oracle.com/uk/virtualization/virtualbox/. Last Checked: 04-11-2022. (Cited on page 100.)

[23] podman. https://podman.io/. Accessed: 2021-11-21. (Cited on page 54.)

[24] Scudo hardened allocator. https://microsoft.github.io/mimalloc/. (Cited on page 95.)

[25] Tkrzw: a set of implementations of dbm. https://dbmx.net/tkrzw/. Accessed: 2021-11-21. (Cited on pages 47 and 55.)

[26] Unlink exploit. https://heap-exploitation.dhavalkapil.com/attacks/unlink_exploit. Accessed: 2022-05-29. (Cited on page 75.)

[27] The userspace i/o howto. https://www.kernel.org/doc/html/v4.12/driver-api/uio-howto.html. Accessed: 2021-11-21. (Cited on page 47.)

[28] Vmware esxi. https://www.vmware.com/uk/products/esxi-and-esx.html. Last Checked: 04-11-2022. (Cited on page 99.)

[29] Vmware workstation pro. https://www.vmware.com/uk/products/workstation-pro.html. Last Checked: 04-11-2022. (Cited on page 100.)

[30] Vt-d posted interrupts. https://bit.ly/3tlgA1o. Accessed: 2021-11-21. (Cited on page 46.)

[31] Xen project software overview. https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview. Last Checked: 06-12-2022. (Cited on page 100.)

[32] Intel ept-based sub-page write protection support. https://lwn.net/Articles/736322/, Oct. 2017. (Cited on pages 3 and 73.)

[33] Intel's new virtualization features on xeon platforms. https://bit.ly/3g9SFP8, 2018. (Cited on page 87.)

[34] Page-modification logging for virtual-machine monitor. https://www.intel.fr/content/www/fr/fr/processors/page-modification-logging-vmm-white-paper.html, Feb. 2018. (Cited on page 3.)

[35] Vmkernel architecture. https://communities-gbot.vmware.com/thread/93870, Dec. 2018. (Cited on page 25.)

[36] Vmware vmkernel. https://microage.com/wp-content/uploads/2016/02/ESXi_architecture.pdf., Dec. 2018. (Cited on page 25.)

[37] July 2019. (Cited on page 35.)

[38] A Scalable Big Data and AI Benchmark Suite, ICT, Chinese Academy of Sciences. http://prof.ict.ac.cn/BigDataBench/, Mar. 2019. (Cited on pages 25, 26 and 30.)

[39] HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. http://www.netlib.org/benchmark/hpl/, Mar. 2019. (Cited on pages 25, 26 and 30.)

[40] improving real-time performance by utilizing cache allocation technology. White paper. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf, July 2019. (Cited on page 24.)

[41] Intel® virtualization technology (intel® vt). https://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html, Apr. 2019. (Cited on page 40.)

[42] Introduction to cache allocation technology in the intel® xeon® processor e5 v4 family. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology, Apr. 2019. (Cited on pages 3 and 40.)

[43] Isolation alloc. https://github.com/GrapheneOS/hardened_malloc, 2019. (Cited on page 95.)

[44] Memory Overcommit. https://www.techopedia.com/definition/14761/memory-overcommi, July 2019. (Cited on page 23.)

[45] mi-malloc general purpose allocator. https://microsoft.github.io/mimalloc/, 2019. (Cited on page 95.)

[46] Parsec. http://parsec.cs.princeton.edu/, Apr. 2019. (Cited on page 38.)

[47] A proactive approach to more secure code. https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/, 2019. (Cited on page 72.)

[48] Vmfunc - invoke vm function. https://www.felixcloutier.com/x86/vmfunc, Apr. 2019. (Cited on page 40.)

[49] Enable sub-page write protection support. https://lwn.net/Articles/810033/, Jan. 2020. (Cited on page 87.)

[50] Google security blog: An update on memory safety in chrome. https://security.googleblog.com/2021/09/an-update-on-memory-safety-in-chrome.html, 2021. (Cited on page 72.)

[51] 2021 cwe top 25 most dangerous software weaknesses. https://www.sans.org/top25-software-errors, 2022. (Cited on pages 72 and 74.)

[52] Intel mpx support is dead with linux 5.6. https://www.phoronix.com/news/Intel-MPX-Is-Dead, 2022. (Cited on page 96.)

[53] Olden benchmarks. https://github.com/compor/olden, 2022. (Cited on page 96.)

[54] Abhinit Modi, Raghavendra Achar, P. S. T. Live migration of virtual machines with their local persistent storage in a data intensive cloud. *Int. J. High Perform. Comput. Netw. 10*, 1-2 (Jan. 2017), 134–147. (Cited on page 41.)

[55] Ahn, J., Jin, S., and Huh, J. Revisiting hardware-assisted page walks for virtualized systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (Washington, DC, USA, 2012), ISCA '12, IEEE Computer Society, pp. 476–487. (Cited on page 41.)

[56] Ainsworth, S., and Jones, T. M. Markus: Drop-in use-after-free prevention for low-level languages. In *2020 IEEE Symposium on Security and Privacy (SP)* (2020), pp. 578–591. (Cited on pages 46 and 95.)

[57] Akritidis, and Periklis. Cling: A memory allocator to mitigate dangling pointers. pp. 177–192. (Cited on page 72.)

[58] Akritidis, P., Costa, M., Castro, M., and Hand, S. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium* (2009), vol. 10. (Cited on page 96.)

[59] Alam, H., Zhang, T., Erez, M., and Etsion, Y. Do-it-yourself virtual memory translation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2017), ISCA '17, ACM, pp. 457–468. (Cited on page 41.)

[60] Austin, T. M., Breach, S. E., and Sohi, G. S. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation* (1994), pp. 290–301. (Cited on page 95.)

[61] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (2003), SOSP '03, ACM, pp. 164–177. (Cited on pages 25, 73 and 99.)

[62] Barr, T. W., Cox, A. L., and Rixner, S. Translation caching: Skip, don't walk (the page table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2010), ISCA '10, ACM, pp. 48–59. (Cited on page 41.)

[63] Baumann, A. Hardware is the new software. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (New York, NY, USA, 2017), HotOS '17, ACM, pp. 132–137. (Cited on page 40.)

[64] Belay, A., Bittau, A., Mashtizadeh, A., Terei, D., Mazières, D., and Kozyrakis, C. Dune: Safe user-level access to privileged CPU features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (Hollywood, CA, Oct. 2012), USENIX Association, pp. 335–348. (Cited on page 15.)

[65] Belpaire, G., and Hsu, N.-T. Hardware architecture for recursive virtual machines. In *Proceedings of the 1975 Annual Conference* (New York, NY, USA, 1975), ACM '75, Association for Computing Machinery, p. 14–18. (Cited on page 1.)

[66] Belpaire, G., and Hsu, N.-T. Hardware architecture for recursive virtual machines. In *Proceedings of the 1975 Annual Conference* (New York, NY, USA, 1975), ACM '75, Association for Computing Machinery, p. 14–18. (Cited on pages 16, 18 and 19.)

[67] Ben-Yehuda, M., Day, M. D., Dubitzky, Z., Factor, M., Har'El, N., Gordon, A., Liguori, A., Wasserman, O., and Yassour, B.-A. The turtles project: Design and implementation of nested virtualization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (USA, 2010), OSDI'10, USENIX Association, p. 423–436. (Cited on pages 1, 2 and 20.)

[68] Bhargava, R., Serebrin, B., Spadini, F., and Manne, S. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2008), ASPLOS XIII, ACM, pp. 26–35. (Cited on page 40.)

[69] Bienia, C., Kumar, S., Singh, J. P., and Li, K. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2008), PACT '08, Association for Computing Machinery, p. 72–81. (Cited on pages 73, 77 and 87.)

[70] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. The gem5 simulator. *SIGARCH Comput. Archit. News 39*, 2 (Aug. 2011), 1–7. (Cited on pages 25 and 35.)

[71] Bitchebe, S. GitHub repository of LeanGuard (OoH for SPP). https://github.com/bstellaceleste/OoH/tree/SPML/OoH-SPP, 2022. (Cited on page 5.)

[72] Bitchebe, S. GitHub repository of SPML and EPML (OoH for PML). https://github.com/bstellaceleste/OoH/tree/SPML/OoH-PML, 2022. (Cited on page 5.)

[73] Bitchebe, S. PRL's GitHub repository. https://github.com/bstellaceleste/OoH/tree/SPML/PRL, 2022. (Cited on page 4.)

[74] Bitchebe, S., Mvondo, D., Réveillère, L., de Palma, N., and Tchana, A. Extending intel pml for hardware-assisted working set size estimation of vms. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2021), VEE 2021, Association for Computing Machinery, p. 111–124. (Cited on pages 4, 46 and 68.)

[75] Bitchebe, S., and Tchana, A. Out of hypervisor (ooh): Efficient dirty page tracking in userspace using hardware virtualization features. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2022), SC '22, IEEE Press. (Cited on page 5.)

[76] Brooker, M., and Mesrobian, H. A Serverless Journey: Under the Hood of AWS Lambda. https://www.youtube.com/watch?v=QdzV04T_kec, Nov. 2018. (Cited on page 24.)

[77] Chen, G. P., and Bozman, J. S. *Optimizing I/O Virtualization: Preparing the Datacenter for Next-Generation Applications*. 2009. (Cited on page 3.)

[78] Cheng, K., Doddamani, S., Chiueh, T.-C., Li, Y., and Gopalan, K. Directvisor: Virtualization for bare-metal cloud. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2020), VEE '20, Association for Computing Machinery, p. 45–58. (Cited on page 68.)

[79] Chiang, J.-H., Li, H.-L., and cker Chiueh, T. Working set-based physical memory ballooning. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)* (San Jose, CA, 2013), USENIX, pp. 95–99. (Cited on pages 24 and 26.)

[80] Chiang, J.-H., Li, H.-L., and cker Chiueh, T. Working set-based physical memory ballooning. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)* (San Jose, CA, 2013), USENIX, pp. 95–99. (Cited on page 41.)

[81] Christian Bienia, Sanjeev Kumar, J. P. S., and Li, K. The parsec benchmark suite: Characterization and architectural implications. https://parsec.cs.princeton.edu/doc/parsec-report.pdf, Jan. 2008. (Cited on page 38.)

[82] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. Live migration of virtual machines. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2* (Berkeley,

CA, USA, 2005), NSDI'05, USENIX Association, pp. 273–286. (Cited on pages 2 and 23.)

[83] Connelly, J., Roberts, T., Gao, X., Xiao, J., Wang, H., and Stavrou, A. Cloudskulk: A nested virtual machine based rootkit and its detection. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2021), pp. 350–362. (Cited on page 2.)

[84] Cortez, E., Bonde, A., Muzio, A., Russinovich, M., Fontoura, M., and Bianchini, R. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 153–167. (Cited on page 24.)

[85] Delimitrou, C., and Kozyrakis, C. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2014), ASPLOS '14, ACM, pp. 127–144. (Cited on page 24.)

[86] Denning, P. J. The working set model for program behavior. *Commun. ACM 11*, 5 (May 1968), 323–333. (Cited on page 23.)

[87] Dennis, J. B., and Van Horn, E. C. Programming semantics for multiprogrammed computations. *Commun. ACM 9*, 3 (mar 1966), 143–155. (Cited on page 96.)

[88] Drepper, U. *What Every Programmer Should Know About Memory*. drepper@redhat.com, 2007. (Cited on page 91.)

[89] Duck, G. J., and Yap, R. H. C. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction* (New York, NY, USA, 2016), CC 2016, Association for Computing Machinery, p. 132–142. (Cited on page 72.)

[90] Fingler, H., Akshintala, A., and Rossbach, C. J. Usetl: Unikernels for serverless extract transform and load why should you settle for less? In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems* (New York, NY, USA, 2019), APSys '19, Association for Computing Machinery, p. 23–30. (Cited on page 46.)

[91] Ford, B., Hibler, M., Lepreau, J., Tullmann, P., Back, G., and Clawson, S. Microkernels meet recursive virtual machines. *SIGOPS Oper. Syst. Rev. 30*, SI (oct 1996), 137–151. (Cited on page 19.)

[92] Gandhi, J., Basu, A., Hill, M. D., and Swift, M. M. Efficient memory virtualization: Reducing dimensionality of nested page walks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2014), MICRO-47, IEEE Computer Society, pp. 178–189. (Cited on page 41.)

[93] Gandhi, J., Hill, M. D., and Swift, M. M. Agile paging: Exceeding the best of nested and shadow paging. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Piscataway, NJ, USA, 2016), ISCA '16, IEEE Press, pp. 707–718. (Cited on page 41.)

[94] Goldberg, R. P. Survey of virtual machine research. *Computer 7*, 6 (1974), 34–45. (Cited on page 1.)

[95] Gordon, A., Amit, N., Har'El, N., Ben-Yehuda, M., Landau, A., Schuster, A., and Tsafrir, D. Eli: Bare-metal performance for i/o virtualization. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS XVII, Association for Computing Machinery, p. 411–422. (Cited on page 68.)

[96] Intel. vol. 3C. 2022. (Cited on page 3.)

[97] Jim, T., Morrisett, J. G., Grossman, D., Hicks, M. W., Cheney, J., and Wang, Y. Cyclone: a safe dialect of c. In *USENIX Annual Technical Conference, General Track* (2002), pp. 275–288. (Cited on page 95.)

[98] Joannou, A. J. P. High-performance memory safety: optimizing the cheri capability machine. Tech. Rep. UCAM-CL-TR-936, Computer Laboratory, June 2020. (Cited on pages 76 and 96.)

[99] Jones, R. W., and Kelly, P. H. Backwards-compatible bounds checking for arrays and pointers in c programs. In *AADEBUG* (1997), vol. 97, Citeseer, pp. 13–26. (Cited on page 96.)

[100] Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Geiger: Monitoring the buffer cache in a virtual machine environment. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2006), ASPLOS XII, ACM, pp. 14–24. (Cited on pages 24 and 26.)

[101] Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Geiger: Monitoring the buffer cache in a virtual machine environment. *SIGARCH Comput. Archit. News 34*, 5 (Oct. 2006), 14–24. (Cited on page 42.)

[102] Jyothi, S. A., Curino, C., Menache, I., Narayanamurthy, S. M., Tumanov, A., Yaniv, J., Mavlyutov, R., Goiri, I. n., Krishnan, S., Kulkarni, J., and Rao, S. Morpheus: Towards automated slos for enterprise clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2016), OSDI'16, USENIX Association, pp. 117–134. (Cited on page 24.)

[103] Kasture, H., and Sánchez, D. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization, IISWC 2016, Providence, RI, USA, September 25-27, 2016* (2016), pp. 3–12. (Cited on page 24.)

[104] Keller, E., Szefer, J., Rexford, J., and Lee, R. B. Nohype: Virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2010), ISCA '10, Association for Computing Machinery, p. 350–361. (Cited on pages 16 and 17.)

[105] Keller, E., Szefer, J., Rexford, J., and Lee, R. B. Nohype: Virtualized cloud infrastructure without the virtualization. *SIGARCH Comput. Archit. News 38*, 3 (June 2010), 350–361. (Cited on page 40.)

[106] Kim, J., Fedorov, V., Gratz, P. V., and Reddy, A. L. N. Dynamic memory pressure aware ballooning. In *Proceedings of the 2015 International Symposium on Memory Systems* (New York, NY, USA, 2015), MEMSYS '15, ACM, pp. 103–112. (Cited on pages 24 and 26.)

[107] Kroes, T., Koning, K., van der Kouwe, E., Bos, H., and Giuffrida, C. Delta pointers: Buffer overflow checks without the checks. In *Proceedings of the Thirteenth EuroSys Conference* (2018), pp. 1–14. (Cited on page 96.)

[108] Kwon, A., Dhawan, U., Smith, J. M., Knight Jr, T. F., and DeHon, A. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), pp. 721–732. (Cited on page 96.)

[109] Landau, A., Ben-Yehuda, M., and Gordon, A. Splitx: Split guest/hypervisor execution on multi-core. In *3rd Workshop on I/O Virtualization (WIOV 11)* (Portland, OR, June 2011), USENIX Association. (Cited on page 68.)

[110] Lauer, H. C., and Wyeth, D. A recursive virtual machine architecture. In *Proceedings of the Workshop on Virtual Computer Systems* (New York, NY, USA, 1973), Association for Computing Machinery, p. 113–116. (Cited on pages 16, 18 and 19.)

[111] Lee, B., Song, C., Jang, Y., Wang, T., Kim, T., Lu, L., and Lee, W. Preventing use-after-free with dangling pointers nullification. In *NDSS* (2015). (Cited on page 96.)

[112] Lee, M., Krishnakumar, A. S., Krishnan, P., Singh, N., and Yajnik, S. Hypervisor-assisted application checkpointing in virtualized environments. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks* (Washington, DC, USA, 2011), DSN '11, IEEE Computer Society, pp. 371–382. (Cited on page 42.)

[113] Liguori, A. Powering Next-Gen EC2 Instances – Deep Dive into the Nitro System. https://www.youtube.com/watch?v=e8DVmwj3OEs, Nov. 2018. (Cited on page 41.)

[114] Lim, Tack, J., Nieh, and Jason. Optimizing nested virtualization performance using direct virtual hardware. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2020), ASPLOS '20, Association for Computing Machinery, p. 557–574. (Cited on pages 1, 2, 17, 19 and 68.)

[115] Lim, J. T., Dall, C., Li, S., Nieh, J., and Zyngier, M. NEVE: nested virtualization extensions for ARM. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17), Shanghai, China, October 28-31, 2017* (2017), pp. 201–217. (Cited on page 41.)

[116] Lim, J. T., Dall, C., Li, S.-W., Nieh, J., and Zyngier, M. Neve: Nested virtualization extensions for arm. SOSP '17, Association for Computing Machinery. (Cited on pages 1, 2, 3, 20 and 68.)

[117] Lim, K. T., Chang, J., Mudge, T. N., Ranganathan, P., Reinhardt, S. K., and Wenisch, T. F. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proceedings of the 2009 ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'09)* (2009), ACM, pp. 267–278. (Cited on page 24.)

[118] Liu, Beichen, Olivier, Pierre, Ravindran, and Binoy. Slimguard: A secure and memory-efficient heap allocator. In *Proceedings of the 20th International Middleware Conference* (New York, NY, USA,

2019), Middleware '19, Association for Computing Machinery, p. 1–13. (Cited on pages 72, 73, 74, 75, 77 and 95.)

[119] Lu, K., Zhang, W., Wang, X., Luján, M., and Nisbet, A. Flexible page-level memory access monitoring based on virtualization hardware. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2017), VEE '17, ACM, pp. 201–213. (Cited on page 69.)

[120] Lu, P., and Shen, K. Virtual machine memory access tracing with hypervisor exclusive cache. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference* (Berkeley, CA, USA, 2007), ATC'07, USENIX Association, pp. 3:1–3:15. (Cited on pages 24 and 26.)

[121] Lu, P., and Shen, K. Virtual machine memory access tracing with hypervisor exclusive cache. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference* (Berkeley, CA, USA, 2007), ATC'07, USENIX Association, pp. 3:1–3:15. (Cited on page 42.)

[122] Malla, S., and Christensen, K. J. Hpc in the cloud: Performance comparison of function as a service (faas) vs infrastructure as a service (iaas). *Internet Technology Letters 3* (2020). (Cited on page 51.)

[123] Medina, V., and García, J. M. A survey of migration mechanisms of virtual machines. *ACM Comput. Surv. 46*, 3 (Jan. 2014), 30:1–30:33. (Cited on page 41.)

[124] Moerbeek, O. A new malloc (3) for openbsd. In *Proceedings of the 2009 European BSD Conference* (2009), vol. 9. (Cited on page 72.)

[125] Mogul, J. C., Baumann, A., Roscoe, T., and Soares, L. Mind the gap: Reconnecting architecture and os research. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems (HotOS)* (May 2011). (Cited on page 33.)

[126] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2009), pp. 245–258. (Cited on page 96.)

[127] Necula, G. C., McPeak, S., and Weimer, W. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2002), pp. 128–139. (Cited on page 95.)

[128] Nethercote, N., and Seward, J. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *ACM Sigplan notices 42*, 6 (2007), 89–100. (Cited on page 96.)

[129] Nguyen, K. T. I. Apic virtualization performance testing and iozone. https://software.intel.com/en-us/blogs/2013/12/17/apic-virtualization-performance-testing-and-iozone, Dec. 2018. (Cited on pages 3 and 40.)

[130] Nicolae, B., and Cappello, F. A hybrid local storage transfer scheme for live migration of i/o intensive workloads. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA, 2012), HPDC '12, ACM, pp. 85–96. (Cited on page 41.)

[131] Nitu, V., Kocharyan, A., Yaya, H., Tchana, A., Hagimont, D., and Astsatryan, H. Working set size estimation techniques in virtualized environments: One size does not fit all. *Proc. ACM Meas. Anal. Comput. Syst. 2*, 1 (Apr. 2018), 19:1–19:22. (Cited on pages 24, 26, 37 and 42.)

[132] Nitu, V., Olivier, P., Tchana, A., Chiba, D., Barbalace, A., Hagimont, D., and Ravindran, B. 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* (New York, NY, USA, 2017), VEE '17, ACM, pp. 1–14. (Cited on page 30.)

[133] Nitu, V., Teabe, B., Tchana, A., Isci, C., and Hagimont, D. Welcome to zombieland: Practical and energy-efficient memory disaggregation in a datacenter. In *Proceedings of the Thirteenth EuroSys Conference* (New York, NY, USA, 2018), EuroSys '18, ACM, pp. 16:1–16:12. (Cited on pages 24 and 37.)

[134] Novark, G., and Berger, E. D. Dieharder: Securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2010), CCS '10, Association for Computing Machinery, p. 573–584. (Cited on pages 72, 75 and 95.)

[135] Oleksenko, O., Kuvaiskii, D., Bhatotia, P., Felber, P., and Fetzer, C. Intel mpx explained: A cross-layer analysis of the intel mpx system stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems 2*, 2 (2018), 1–30. (Cited on pages 72 and 96.)

[136] Oleksenko, O., Kuvaiskii, D., Bhatotia, P., Felber, P., and Fetzer, C. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. (Cited on page 96.)

[137] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J. Informed prefetching and caching. *SIGOPS Oper. Syst. Rev. 29*, 5 (Dec. 1995), 79–95. (Cited on page 42.)

[138] Popek, G. J., and Goldberg, R. P. Formal requirements for virtualizable third generation architectures. *Commun. ACM 17*, 7 (jul 1974), 412–421. (Cited on page 1.)

[139] Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., and Kozyrakis, C. Evaluating mapreduce for multi-core and multiprocessor systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture* (2007), pp. 13–24. (Cited on pages 47 and 55.)

[140] Ravi Murty, A. W. S. P. E. Powering next-gen amazon ec2: Deep dive into the nitro system. https://d1.awsstatic.com/events/reinvent/2019/REPEAT_2_Powering_next-gen_Amazon_EC2_Deep_dive_into_the_Nitro_system_CMP303-R2.pdf, 2019. (Cited on pages 16 and 18.)

[141] Ruwase, O., and Lam, M. S. A practical dynamic buffer overflow detector. In *NDSS* (2004), vol. 4, pp. 159–169. (Cited on page 96.)

[142] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)* (2012), pp. 309–318. (Cited on page 96.)

[143] Shan, Y., Huang, Y., Chen, Y., and Zhang, Y. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)* (Carlsbad, CA, USA, 2018), USENIX Association, pp. 69–87. (Cited on page 24.)

[144] Silvestro, Sam, Liu, Hongyu, Crosser, Corey, Lin, Zhiqiang, Liu, and Tongping. Freeguard: A faster secure heap allocator. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2017), CCS '17, Association for Computing Machinery, p. 2389–2403. (Cited on pages 72, 75 and 95.)

[145] Silvestro, S., Liu, H., Liu, T., Lin, Z., and Liu, T. Guarder: A tunable secure allocator. In *27th USENIX Security Symposium (USENIX Security 18)* (Baltimore, MD, Aug. 2018), USENIX Association, pp. 117–133. (Cited on pages 72, 75 and 95.)

[146] Subramanian, B. Napi - the new api for linux network drivers. https://bharathisubramanian.wordpress.com/2010/04/13/napi-the-new-api-for-linux-network-drivers/, Dec. 2018. (Cited on page 33.)

[147] Svärd, P., Hudzia, B., Walsh, S., Tordsson, J., and Elmroth, E. Principles and performance characteristics of algorithms for live vm migration. *SIGOPS Oper. Syst. Rev. 49*, 1 (Jan. 2015), 142–155. (Cited on page 41.)

[148] Szekeres, L., Payer, M., Wei, T., and Song, D. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy* (2013), pp. 48–62. (Cited on page 71.)

[149] Ustiugov, D., Petrov, P., Kogias, M., Bugnion, E., and Grot, B. *Benchmarking, Analysis, and Optimization of Serverless Function Snapshots*. Association for Computing Machinery, New York, NY, USA, 2021, p. 559–572. (Cited on page 46.)

[150] Vilanova, L., Amit, N., and Etsion, Y. Using smt to accelerate nested virtualization. In *Proceedings of the 46th International Symposium on Computer Architecture* (New York, NY, USA, 2019), ISCA '19, Association for Computing Machinery, p. 750–761. (Cited on pages 1, 2, 20 and 68.)

[151] Waldspurger, C. A. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev. 36*, SI (Dec. 2002), 181–194. (Cited on pages 24, 25, 26 and 40.)

[152] Wang, L., Li, M., Zhang, Y., Ristenpart, T., and Swift, M. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, 2018), USENIX Association, pp. 133–146. (Cited on page 24.)

[153] Wang, L., Li, M., Zhang, Y., Ristenpart, T., and Swift, M. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 133–146. (Cited on page 51.)

[154] Wang, X., Zang, J., Wang, Z., Luo, Y., and Li, X. Selective hardware/software memory virtualization. In *Proceedings of the 7th ACM*

SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (New York, NY, USA, 2011), VEE '11, ACM, pp. 217–226. (Cited on page 41.)

[155] Waoo, A., Patidar, D., and Patheja, P. An efficient approach for cloud computing based on hierarchical secure paravirtualization system resource model. *International Journal of Applied Engineering Research 7* (01 2012), 1527–1534. (Cited on page 8.)

[156] Watson, R. N. M., Richardson, A., Davis, B., Baldwin, J., Chisnall, D., Clarke, J., Filardo, N., Moore, S. W., Napierala, E., Sewell, P., , and Neumann, P. G. Cheri c/c++ programming guide. Tech. Rep. UCAM-CL-TR-947, Computer Laboratory, June 2020. (Cited on pages 76 and 96.)

[157] Wickman, B., Hu, H., Yun, I., Jang, D., Lim, J., Kashyap, S., and Kim, T. Preventing Use-After-Free attacks with fast forward allocation. In *30th USENIX Security Symposium (USENIX Security 21)* (Aug. 2021), USENIX Association, pp. 2453–2470. (Cited on page 95.)

[158] Williams, D., Hu, Y., Deshpande, U., Sinha, P. K., Bila, N., Gopalan, K., and Jamjoom, H. Enabling efficient hypervisor-as-a-service clouds with ephemeral virtualization. *SIGPLAN Not. 51*, 7 (mar 2016), 79–92. (Cited on page 19.)

[159] Woodruff, Jonathan, Watson, N.M., R., Chisnall, David, Moore, W., S., Anderson, Jonathan, Davis, Brooks, Laurie, Ben, Neumann, G., P., Norton, Robert, Roe, and Michael. The cheri capability model: Revisiting risc in an age of risk. *SIGARCH Comput. Archit. News 42*, 3 (jun 2014), 457–468. (Cited on pages 72 and 76.)

[160] Yaniv, I., and Tsafrir, D. Hash, don't cache (the page table). In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science* (New York, NY, USA, 2016), SIGMETRICS '16, ACM, pp. 337–350. (Cited on page 41.)

[161] Yun, I., Song, W., Min, S., and Kim, T. Hardsheap: A universal and extensible framework for evaluating secure allocators. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2021), CCS '21, Association for Computing Machinery, p. 379–392. (Cited on pages 72 and 95.)

[162] Zhang, F., Chen, J., Chen, H., and Zang, B. Cloudvisor: Retrofitting protection of virtual machines in multi-tenant cloud with

nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, Association for Computing Machinery, p. 203–216. (Cited on page 1.)

[163] Zhang, I., Garthwaite, A., Baskakov, Y., and Barr, K. C. Fast restore of checkpointed memory using working set estimation. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2011), VEE '11, ACM, pp. 87–98. (Cited on pages 2, 23 and 24.)

[164] Zhang, J., Dong, E., Li, J., and Guan, H. Migvisor: Accurate prediction of vm live migration behavior using a working-set pattern model. *SIGPLAN Not. 52*, 7 (Apr. 2017), 30–43. (Cited on page 41.)

[165] Zhao, W., Jin, X., Wang, Z., Wang, X., Luo, Y., and Li, X. Low cost working set size tracking. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2011), USENIXATC'11, USENIX Association, pp. 17–17. (Cited on pages 34, 35, 41 and 42.)

[166] Zhou, P., Pandey, V., Sundaresan, J., Raghuraman, A., Zhou, Y., and Kumar, S. Dynamic tracking of page miss ratio curve for memory management. *SIGOPS Oper. Syst. Rev. 38*, 5 (Oct. 2004), 177–188. (Cited on page 41.)